On Tic-Tac-Toe

Ramprasad S Joshi

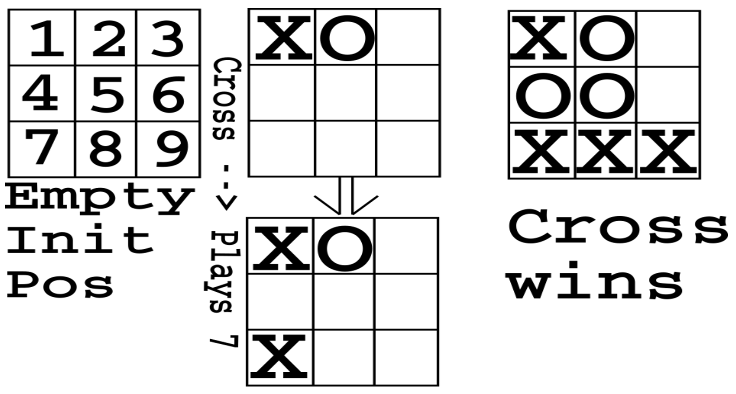
10/22/17

Abstract

This is about the game of Tic-Tac-Toe being analysed and played using FOL. It contains some tools and an assignment to do using those tools (or otherwise, but to do somehow).

1 Introduction

Take the game of Tic-Tac-Toe. Though it was discussed in the class for several weeks, to avoid accusations of pressurising students to attend classes, we describe it here. Two players, Cross and Nought, take turns to place crosses (`X`) and noughts (`O`) in empty cells from a 3X3 array. The one making a row or a column or a longest diagonal of her mark first wins the game; if it doesn't happen after all 9 cells are filled, it is a draw. In the following diagram, the initial empty position, then an intermediate move, and a win for the Cross player, are shown.



1.1 The Role of Logic in Tic-Tac-Toe

Since there are two players, each with their exclusive marks, this is an almost perfect example of the Law of the Excluded Middle which is not indeed one. I mean, since the squares can be empty also, there *is* a middle possibility. But still, we can just make a KB out of each board situation, in which only crosses and noughts are mentioned; the rest are ignored except for "what if" queries.

An Example KB from the Figures The two middle positions, showing one move, can be described by two `Prolog` KBs thus:

```
mark(1,x).
mark(2,o).
```

and

```
mark(1,x).
mark(7,x).
mark(2,o).
```

along with the rules:

```
win(P) :- mark(1,P), mark(2,P), mark(3,P).
win(P) :- mark(4,P), mark(5,P), mark(6,P).
win(P) :- mark(3,P), mark(5,P), mark(7,P).
```

etc.

2 The Assignment

2.1 The Problem

Submit any one of the following:

1. A proposal to change a game rule in TIC-TAC-TOE that will make it possible for the first player to win, and a proof – FOL and/or a `Prolog/Lisp/C/C++` program – that this change indeed guarantees this win. The change proposal must not change the alternate move rule; both the players will have to still place their marks in single cells in each move at alternate turns. But you can put restrictions on responses based on some relation with the situation on the board. Or you can introduce some give-and-take such that a player can shift the other's mark elsewhere and replace it by her own, with some restrictions. Use your imagination.

2. An exhaustive list of best strategies (i.e., sequences of alternate moves of both the players guaranteeing draws), with a (FOL/program) proof of soundness and completeness.

2.2 The Mode of Submission

Choose your group (or choose to work individually) without any formality. Send me an email from the BITS mail id of one of the members of your group, with the subject line "`LiCS 2017 Submission of`" followed by (in the subject line) all the full ids in the group separated by a comma and a space each. We will automate some processes, therefore, if you err in this and the automated processes do not detect your submission, that's your problem. An example subject line is:
`LiCS 2017 Submission of 2117B6A79999G, 2007A7QS0000G`
In the mail, nothing except a single file attachment must be there. If it is one document, it should be in pdf format, and self-contained. If you send programs and proofs in pdf documents, then there must be some README, some makefile or shell-script to make it work with clear instructions, such that it can be checked without requiring your presence. Imagine all possible scenarios in which this can fail, in order to design to avoid them.

2.3 The Deadline

11 through 21 November. The earlier the better, since we will give you our opinion and two chances to improve. After 21 November, if you have less time to improve in the direction we suggest, that's your problem. Do not ask concessions.

2.4 Evaluation

We will immediately (in 1 business day) send you back our approval or suggestion for improvement. This can happen twice. The third submission will be final and we will not explain our assessment after that; the explanation will be there only with the first two chances.

2.5 The Principles

Remember, you actually need to give us proof of your sincere effort. If you copy (instead of merging a group and coming clean on it) and we find out, we will summarily punish you without giving a chance to establish whose submission is the original. We also do not bind ourselves to a fixed mode or quantum of punishment. Please consult Mr. Mouli, the Placement Manager, first before even thinking of copying. And start thinking early on. He will not give advice when he is busy.

2.6 Whine-free Zone

Yelp, we will help. But be forthright. Clear. Consistent. We will also teach you how to be precise.

3 Tools

We provide two programs. One `Prolog`, that does not do much strategy except avoiding immediate loss, and the other my own `Lisp` program. How to make use of them is your lookout. They are provided on an "as is where is" basis.

A The UoW `Prolog` Program

```
% A Tic-Tac-Toe program in Prolog.   S. Tanimoto, May 11, 2003.
% To play a game with the computer, type
% playo.
% To watch the computer play a game with itself, type
% selfgame.

% Predicates that define the winning conditions:

win(Board, Player) :- rowwin(Board, Player).
win(Board, Player) :- colwin(Board, Player).
win(Board, Player) :- diagwin(Board, Player).

rowwin(Board, Player) :- Board = [Player,Player,Player,_,_,_,_,_,_].
rowwin(Board, Player) :- Board = [_,_,_,Player,Player,Player,_,_,_].
rowwin(Board, Player) :- Board = [_,_,_,_,_,_,Player,Player,Player].

colwin(Board, Player) :- Board = [Player,_,_,Player,_,_,Player,_,_].
colwin(Board, Player) :- Board = [_,Player,_,_,Player,_,_,Player,_].
colwin(Board, Player) :- Board = [_,_,Player,_,_,Player,_,_,Player].

diagwin(Board, Player) :- Board = [Player,_,_,_,Player,_,_,_,Player].
diagwin(Board, Player) :- Board = [_,_,Player,_,Player,_,Player,_,_].

% Helping predicate for alternating play in a "self" game:

other(x,o).
other(o,x).

game(Board, Player) :- win(Board, Player), !, write([player, Player, wins]).
game(Board, Player) :-
  other(Player,Otherplayer),
  move(Board,Player,Newboard),
  !,
  display(Newboard),
  game(Newboard,Otherplayer).

move([b,B,C,D,E,F,G,H,I], Player, [Player,B,C,D,E,F,G,H,I]).
move([A,b,C,D,E,F,G,H,I], Player, [A,Player,C,D,E,F,G,H,I]).
move([A,B,b,D,E,F,G,H,I], Player, [A,B,Player,D,E,F,G,H,I]).
move([A,B,C,b,E,F,G,H,I], Player, [A,B,C,Player,E,F,G,H,I]).
move([A,B,C,D,b,F,G,H,I], Player, [A,B,C,D,Player,F,G,H,I]).
move([A,B,C,D,E,b,G,H,I], Player, [A,B,C,D,E,Player,G,H,I]).
move([A,B,C,D,E,F,b,H,I], Player, [A,B,C,D,E,F,Player,H,I]).
move([A,B,C,D,E,F,G,b,I], Player, [A,B,C,D,E,F,G,Player,I]).
move([A,B,C,D,E,F,G,H,b], Player, [A,B,C,D,E,F,G,H,Player]).

display([A,B,C,D,E,F,G,H,I]) :- write([A,B,C]),nl,write([D,E,F]),nl,
 write([G,H,I]),nl,nl.

selfgame :- game([b,b,b,b,b,b,b,b,b],x).

% Predicates to support playing a game with the user:

x_can_win_in_one(Board) :- move(Board, x, Newboard), win(Newboard, x).

% The predicate orespond generates the computer's (playing o) reponse
% from the current Board.

orespond(Board,Newboard) :-
  move(Board, o, Newboard),
  win(Newboard, o),
  !.
orespond(Board,Newboard) :-
  move(Board, o, Newboard),
  not(x_can_win_in_one(Newboard)).
orespond(Board,Newboard) :-
  move(Board, o, Newboard).
orespond(Board,Newboard) :-
  not(member(b,Board)),
  !,
  write('Cats game!'), nl,
  Newboard = Board.

% The following translates from an integer description
% of x's move to a board transformation.

xmove([b,B,C,D,E,F,G,H,I], 1, [x,B,C,D,E,F,G,H,I]).
xmove([A,b,C,D,E,F,G,H,I], 2, [A,x,C,D,E,F,G,H,I]).
xmove([A,B,b,D,E,F,G,H,I], 3, [A,B,x,D,E,F,G,H,I]).
xmove([A,B,C,b,E,F,G,H,I], 4, [A,B,C,x,E,F,G,H,I]).
xmove([A,B,C,D,b,F,G,H,I], 5, [A,B,C,D,x,F,G,H,I]).
xmove([A,B,C,D,E,b,G,H,I], 6, [A,B,C,D,E,x,G,H,I]).
xmove([A,B,C,D,E,F,b,H,I], 7, [A,B,C,D,E,F,x,H,I]).
xmove([A,B,C,D,E,F,G,b,I], 8, [A,B,C,D,E,F,G,x,I]).
xmove([A,B,C,D,E,F,G,H,b], 9, [A,B,C,D,E,F,G,H,x]).
xmove(Board, _N, Board) :- write('Illegal move.'), nl.

% The 0-place predicate playo starts a game with the user.

playo :- explain, playfrom([b,b,b,b,b,b,b,b,b]).

explain :-
  write('You play X by entering integer positions followed by a period.'),
  nl,
  display([1,2,3,4,5,6,7,8,9]).

playfrom(Board) :- win(Board, x), write('You win!').
playfrom(Board) :- win(Board, o), write('I win!').
playfrom(Board) :- read(N),
  xmove(Board, N, Newboard),
  display(Newboard),
  orespond(Newboard, Newnewboard),
  display(Newnewboard),
  playfrom(Newnewboard).
```

B A Lisp Program

```
(defun lll () (load "tictactoeFinal.lisp"))
(defun basicBoard () '(1 2 3 4 5 6 7 8 9))
(defun init-pos () (mapcar #'(lambda (c) (cons c '_)) (basicBoard)))

(defun win (Board Player)
  (or
    (rowwin Board Player)
    (colwin Board Player)
    (diagwin Board Player)
  )
)

(defun rows () '((1 2 3)(4 5 6)(7 8 9)))
(defun cols () '((1 4 7)(2 5 8)(3 6 9)))
(defun diags () '((1 5 9)(3 5 7)))

(defun getmark (cell Board) (rest (assoc cell Board)))
(defun setmark (cell Board Player) (setf (rest (assoc cell Board)) Player))

(defun rowwin (Board Player)
  (some
    #'(lambda (r) (every #'(lambda (c) (eq (getmark c Board) Player)) r))
    (rows)
  )
)

(defun colwin (Board Player)
  (some
    #'(lambda (r) (every #'(lambda (c) (eq (getmark c Board) Player)) r))
    (cols)
  )
)

(defun diagwin (Board Player)
  (some
    #'(lambda (r) (every #'(lambda (c) (eq (getmark c Board) Player)) r))
    (diags)
  )
)

(defun draw (Board)
  (and
    (eq
      (reduce #'+ (mapcar #'(lambda (mark)
        (if (not (win Board mark)) 1 0)) '(x o _))
      )
      3
    )
    (not
      (some
        #'(lambda (rcd)
            (some
              #'(lambda (mark)
                  (every
                    #'(lambda (cell)
                        (let ((cmark (getmark cell Board)))
                          (or
                            (eq cmark mark)
                            (eq cmark '_)
                          )
                        )
                      )
                    rcd
                  )
                )
              '(x o)
            )
          )
        (append (rows) (cols) (diags))
      )
    )
  )
)

(defun other (xo) (if (eq xo 'x) 'o 'x))

(defun empty (cell Board) (eq (getmark cell Board) '_))

(defun display (Board &optional strim)
  (mapcar
    #'(lambda (r) (print (mapcar #'(lambda (c) (getmark c Board)) r) strim))
    (rows)
  )
  (print "Board printed" strim)
  Board
)

(defun makemove (Board Player cell)
  (mapcar
    #'(lambda (c)
        (if (eq c cell)
          (cons cell Player)
          (cons c (getmark c Board))
        )
      )
    (basicBoard)
  )
)

(defun getmove (Board Player)
  (display Board)
  (print (list Player '(:Give your move filling a "_" square)))
  (setq cell (read))
  (if (numberp cell)
    (if (some #'(lambda (c) (eq c cell)) (basicBoard))
      (if (empty cell Board)
        cell
        (and (print '(The cell is not empty)) (getmove Board Player))
      )
      (and (print '(The cell is not in the Board)) (getmove Board Player))
    )
    (and (print '(That is not a number)) (getmove Board Player))
  )
)

(defun whose-move (board)
  (let
    (
      (x
        (reduce #'+
          (mapcar
            #'(lambda (c) (if (eq (getmark c board) 'x) 1 0))
            (basicboard)
          )
        )
      )
      (o
        (reduce #'+
          (mapcar
            #'(lambda (c) (if (eq (getmark c board) 'o) 1 0))
            (basicboard)
          )
        )
      )
      (e
        (reduce #'+
          (mapcar
            #'(lambda (c) (if (eq (getmark c board) '_) 1 0))
            (basicboard)
          )
        )
      )
    )
    (cond
      ((eq (+ x o) (length (basicboard))) '_none_)
      ((eq x o) 'x)
      ((eq x (+ o 1)) 'o)
      ('t (error (list 'Board 'Error ': 'x x 'o o 'empty e)))
    )
  )
)

(defun win-in-1 (Board Player)
  (remove 0
    (mapcar
      #'(lambda (c)
          (if
            (and
              (empty c Board)
              (win (makemove Board Player c) Player)
            )
            c
            0
          )
        )
      (basicBoard)
    )
  )
)

(defun win-in-n (Board Player n)
  (if
    (and
      (numberp n)
      (not
        (or
          (< n 1)
          (> n (count-if #'(lambda (c) (empty c Board)) (basicBoard)))
        )
      )
    )
    (cond
      ((eq n 0)
        (win Board Player)
      )
      ((eq n 1)
        (win-in-1 Board Player)
      )
      ((oddp n)
        (remove 0
          (mapcar
            #'(lambda (c)
                (if
                  (and
                    (empty c Board)
                    (let ((newBoard (makemove Board Player c)))
                      (and
                        (all-empty newBoard)
                        (every
                          #'(lambda (d)
                              (and
                                (not
                                  (win
                                    (makemove
                                      newBoard
                                      (other Player)
                                      d
                                    )
                                    (other Player)
                                  )
                                )
                                (some
                                  #'(lambda (m)
                                      (win-in-n
                                        (makemove newBoard (other Player) d)
                                        Player
                                        m
                                      )
                                    )
                                  (remove 0 (mapcar
                                    #'(lambda (mm)
                                        (if (and (oddp mm) (< mm n)) mm 0)
                                      )
                                    (basicBoard)
                                  ))
                                )
                              )
                            )
                          (all-empty newBoard)
                        )
                      )
                    )
                  )
                  c
                  0
                )
              )
            (basicBoard)
          )
        )
      )
      ('t ;; (evenp n)
        (every
          #'(lambda (c)
              (win-in-n (makemove Board (other Player) c) Player (- n 1))
            )
          (all-empty Board)
        )
      )
    )
  )
)

(defun optimal-play-with-explanation (Board Player &optional strim)
  (if Board
  (let ((empties (all-empty Board)))
    (cond
      (
        (setq l (win-in-n Board Player 1))
        (print (list Player 'got 'win 'in 1 '- 'playing l) strim)
      )
      (
        (setq l (win-in-n Board (other Player) 1))
        (print (list Player 'stopping 'loss 'in 1 '- 'playing l) strim)
      )
      (
        (setq l (win-in-n Board Player 3))
        (print (list Player 'got 'win 'in 3 '- 'playing l) strim)
      )
      (
        (setq l (remove 0 (mapcar
          #'(lambda (c)
              (if
                (and
                  (not (win-in-n (makemove Board Player c) (other Player) 3))
                  (win-in-n (makemove Board Player c) Player 4)
                )
                c
                0
              )
            )
          empties
        )))
        (print
          (list Player 'got 'win 'in 5 'without 'loss 'in 3 '- 'playing l)
          strim
        )
      )
      (
        (setq l (remove 0 (mapcar
          #'(lambda (c)
              (if (win-in-n (makemove Board Player c) (other Player) 5) 0 c)
            )
          empties
        )))
        (if (> (length empties) (length l))
          (print (list Player 'stopping 'loss 'in 5 '- 'playing l) strim)
          (print
            (list 'nothing 'to 'strategize 'now 'getting 'empties l)
            strim
          )
        )
      )
      ('t
        (and
          (print
            (list 'nothing 'to 'strategize 'now 'getting 'empties l)
            strim
          )
          (setq l empties)
        )
      )
    )
    l
  )
  'optimal-play)
)

(defun optimal-play (Board Player)
  (if Board
  (let ((empties (all-empty Board)))
    (setq l (win-in-n Board Player 1))
    (if (not l)
      (setq l (win-in-n Board (other Player) 1))
    )
    (if (not l)
      (setq l (win-in-n Board Player 3))
    )
    (if (not l)
      (setq l (remove 0 (mapcar
        #'(lambda (c)
            (if
              (and
                (not (win-in-n (makemove Board Player c) (other Player) 3))
                (win-in-n (makemove Board Player c) Player 4)
              )
              c
              0
            )
          )
        empties
      )))
    )
    (if (not l)
      (setq l (remove 0 (mapcar
        #'(lambda (c)
            (if (win-in-n (makemove Board Player c) (other Player) 5) 0 c)
          )
        empties
      )))
    )
    (if (not l) (setq l empties))
    l
  )
  'optimal-play)
)

(defun best-play (Board Player)
  (if Board
  (append
    (remove 'nil
      (mapcan #'(lambda (n) (win-in-n Board Player n)) '(0 1 3 5))
    )
    (remove 0 (mapcar
      #'(lambda (c)
          (if
            (some
              #'(lambda (n)
                  (win-in-n (makemove Board Player c) (other Player) n)
                )
              '(0 1 3 5)
            )
            0
            c
          )
        )
      (all-empty Board)
    ))
    (all-empty Board)
  )
  'best-play)
)
```

```
(defun all-empty (Board)
  (remove 0 (mapcar #'(lambda (c) (if (empty c Board) c 0)) (basicBoard)))
)

(defun play (Board Player)
  (if Board
  (append
    (remove 0
      (mapcar
        #'(lambda (c)
            (if
              (and
                (empty c Board)
                (win (makemove Board Player c) Player)
              )
              c
              0
            )
          )
        (basicBoard)
      )
    )
    (remove 0
      (mapcar
        #'(lambda (c)
            (if
              (and
                (empty c Board)
                (not (or
                  (win-in-1 (makemove Board Player c) (other Player))
                  (win-in-n (makemove Board Player c) (other Player) 3)
                  (win-in-n (makemove Board Player c) (other Player) 5)
                ))
              )
              c
              0
            )
          )
        (basicBoard)
      )
    )
    (remove 0
      (mapcar
        #'(lambda (c)
            (if
              (and
                (empty c Board)
                (let ((newBoard (makemove Board Player c)))
                  (every
                    #'(lambda (d)
                        (or
                          (not (empty d newBoard))
                          (win-in-1
                            (makemove newBoard (other Player) d) Player
                          )
                        )
                      )
                    (basicBoard)
                  )
                )
              )
              c
              0
            )
          )
        (basicBoard)
      )
    )
    (remove 0
      (mapcar #'(lambda (c) (if (empty c Board) c 0)) (basicBoard))
    )
  )
  'play)
)

(defun game (Board Player Turn)
  (if (not (some #'(lambda (c) (empty c Board)) (basicBoard)))
    (and (display Board) "Draw! Don't say now you already saw!")
    (if (eq Player Turn)
      (let
        (
         (newBoard (makemove Board Player (first (best-play Board Player))))
        )
        (if (not (win newBoard Player))
          (game newBoard Player (other Turn))
          (progn
            (display newBoard)
            "I win! Now bear and grin!"
          )
        )
      )
      (let
        (
          (newBoard
            (makemove Board (other Player)
              (getmove Board (other Player))
            )
          )
        )
        (if (not (win newBoard (other Player)))
          (game newBoard Player Player)
          (progn
            (display newBoard)
            "You win, I lose! I must stop the booze!"
          )
        )
      )
    )
  )
)

(defun alt-game (Board Player Turn Strategy1 Strategy2)
  (if (not (some #'(lambda (c) (empty c Board)) (basicBoard)))
    (and (display Board) "Draw! Don't say now you already saw!")
    (if (eq Player Turn)
      (let
        (
         (newBoard
            (makemove
              Board
              Player
              (first (funcall Strategy1 Board Player))
            )
         )
        )
        (if (not (win newBoard Player))
          (progn
            (print (format 'nil "After ~S's move" Player))
            (display newBoard)
            (alt-game newBoard Player (other Turn) Strategy1 Strategy2)
          )
          (progn
            (display newBoard)
            (princ (format 'nil "~{~S ~}"
              (list
                Player "wins," (other Player) "loses!"
                (other Player)
                "must use" (funcall Strategy1 'nil 'nil)
                ", not use" (funcall Strategy2 'nil 'nil)
              )
            ))
            'Done
          )
        )
      )
      (let
        (
          (newBoard
            (makemove Board (other Player)
              (first (funcall Strategy2 Board (other Player)))
            )
          )
        )
        (if (not (win newBoard (other Player)))
          (progn
            (print (format 'nil "After ~S's move" (other Player)))
            (display newBoard)
            (alt-game newBoard Player Player Strategy1 Strategy2)
          )
          (progn
            (display newBoard)
            (princ (format 'nil "~{~S ~}"
              (list
                (other Player) "wins," Player "loses!"
                Player
                "must use" (funcall Strategy2 'nil 'nil)
                ", not use" (funcall Strategy1 'nil 'nil)
              )
            ))
            'Done
          )
        )
      )
    )
  )
)

(defun give-all-strategies (ofilename)
  (let
    (
      (fsi (open "alltictactoepos.txt" :direction :input))
      (fso (open ofilename :direction :output))
      (k 0)
    )
    (loop
      (setq board (read fsi 'nil))
      (if board
        (progn
          (setq k (+ k 1))
          (print (print (list 'board k) fso))
          (display board fso)
          (display board)
          (setq who (whose-move board))
          (print (print (list who 'to 'move) fso))
          (if (win board 'x) (print (print (list 'x 'has 'won) fso)))
          (if (win board 'o) (print (print (list 'o 'has 'won) fso)))
          (optimal-play-with-explanation board who)
          (optimal-play-with-explanation board who fso)
          (print
            (print
              '(-------------------------------------------------------------)
              fso
            )
          )
          (print (print '(Next) fso))
        )
      )
      (when (not board) (and (close fso) (return k)))
    )
  )
)

(defun enumerate-strategies (boards &optional ofstream)
  (setq k 0)
  (loop
    (when (not boards) (return k))
    (setq board (first boards))
    (setq boards (remove (first boards) boards))
    (setq who (whose-move board))
    (setq empties (all-empty board))
    (loop
      (when (not empties) (return boards))
      (setq c (first empties))
      (setq empties (remove (first empties) empties))
      (display board ofstream)
      (print (list who 'plays c) ofstream)
      (display (makemove board who c))
      (print (list
        'board k '-
        who 'played c
        '- (other who) 'would 'play 'as 'follows
        )
        ofstream
      )
      (optimal-play-with-explanation
        (makemove board who c)
        (other who)
        ofstream
      )
      (cond
        ((win (makemove board who c) who)
          (print (list who 'has 'won) ofstream)
        )
        ((draw (makemove board who c))
          (print (list 'it 'is 'a 'draw) ofstream)
        )
        ('t (setq boards (append boards (list (makemove board who c)))))
      )
      (setq k (+ k 1))
    )
  )
)

(defun strategy-listing (board &key who ofn strim mvl)
  (if (not who) (setq who (whose-move board)))
  (if ofn (setq strim (open ofn :direction :output)))
  (map
    'nil
    #'(lambda (c)
        (progn
          (print (list 'board *kount* '- (format 'nil "~{~S~}" mvl)) strim)
          (display board strim)
          (optimal-play-with-explanation board who)
          (print (list who 'plays c) strim)
          (display (setq nb (makemove board who c)) strim)
          (setq *kount* (+ *kount* 1))
          (cond
            ((win nb who)
              (print
                (list who 'has 'won '- *kount* (format 'nil "~{~S~}" mvl))
                strim
              )
            )
            ((draw nb) (print 'drawn strim))
            ('t
              (progn
                (optimal-play-with-explanation nb (other who) strim)
                (strategy-listing nb
                  :who (other who)
                  :strim strim
```

```
         :mvl (append mvl (list c))
        )
       )
      )
     )
    )
   )
  (all-empty board)
 )
)
```