```
#|

 Copyright (C) 1994 by Computational Logic, Inc.  All Rights Reserved.

 This script is hereby placed in the public domain, and therefore unlimited
 editing and redistribution is permitted.

 NO WARRANTY

 Computational Logic, Inc. PROVIDES ABSOLUTELY NO WARRANTY.  THE EVENT SCRIPT
 IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED,
 INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND
 FITNESS FOR A PARTICULAR PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND
 PERFORMANCE OF THE SCRIPT IS WITH YOU.  SHOULD THE SCRIPT PROVE DEFECTIVE, YOU
 ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

 IN NO EVENT WILL Computational Logic, Inc. BE LIABLE TO YOU FOR ANY DAMAGES,
 ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL
 DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THIS SCRIPT (INCLUDING BUT
 NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES
 SUSTAINED BY THIRD PARTIES), EVEN IF YOU HAVE ADVISED US OF THE POSSIBILITY OF
 SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

|#
```

EVENT: Start with the initial **nqthm** theory.

DEFINITION:
boardp (x)
= **if** $x \simeq$ **nil then** $x =$ **nil**
else $((\mathrm{car}\,(x) =$ '`x`$) \vee (\mathrm{car}\,(x) =$ '`o`$) \vee (\mathrm{car}\,(x) =$ **f**$))$
$\wedge$ boardp $(\mathrm{cdr}\,(x))$ **endif**

DEFINITION:
endp $(board)$
= **if** $board \simeq$ **nil then t**
elseif $\mathrm{car}\,(board)$ **then** $\mathrm{endp}\,(\mathrm{cdr}\,(board))$
else f endif

DEFINITION:
legal-movep $(board1, player, board2)$
= **if** $board1 \simeq$ **nil then f**
elseif $\mathrm{car}\,(board1) = \mathrm{car}\,(board2)$

then legal-movep (cdr (*board1*), *player*, cdr (*board2*))
elseif (car (*board1*) = **f**) ∧ (car (*board2*) = *player*)
then cdr (*board1*) = cdr (*board2*)
else f endif

DEFINITION:
nth (*i*, *board*)
= **if** *i* ≃ `0` **then** car (*board*)
else nth (*i* − 1, cdr (*board*)) **endif**

DEFINITION:
winp1 (*x*, *line*, *board*)
= **if** *line* ≃ **nil then t**
elseif *x* = nth (car (*line*), *board*) **then** winp1 (*x*, cdr (*line*), *board*)
else f endif

DEFINITION:
winp (*x*, *board*)
= (winp1 (*x*, `'(0 1 2)`, *board*)
∨ winp1 (*x*, `'(3 4 5)`, *board*)
∨ winp1 (*x*, `'(6 7 8)`, *board*)
∨ winp1 (*x*, `'(0 3 6)`, *board*)
∨ winp1 (*x*, `'(1 4 7)`, *board*)
∨ winp1 (*x*, `'(2 5 8)`, *board*)
∨ winp1 (*x*, `'(0 4 8)`, *board*)
∨ winp1 (*x*, `'(2 4 6)`, *board*))

EVENT: Disable winp.

DEFINITION:
other-player (*player*)
= **if** *player* = `'x` **then** `'o`
else `'x` **endif**

DEFINITION:
end (*outcome*, *player*, *lst*)
= **let** *outcome* **be if** *outcome* = `'win`
then if *player* = `'o` **then** `'o-win`
else `'x-win` **endif**
else `'draw` **endif**
in
if *lst* = list (*outcome*) **then** *outcome*
else f endif endlet

DEFINITION:
tic-tac-toep1 (*board*, *player*, *lst*)
= **if** *lst* ≃ **nil** **then** end ('win, other-player (*player*), *lst*)
elseif ¬ boardp (car (*lst*)) **then** **f**
elseif ¬ legal-movep (*board*, *player*, car (*lst*))
then end ('win, other-player (*player*), cdr (*lst*))
elseif winp (*player*, car (*lst*)) **then** end ('win, *player*, cdr (*lst*))
elseif endp (car (*lst*)) **then** end ('draw, *player*, cdr (*lst*))
else tic-tac-toep1 (car (*lst*), other-player (*player*), cdr (*lst*)) **endif**

DEFINITION:
tic-tac-toep (*lst*)
= **if** listp (*lst*) ∧ (car (*lst*) = list (**f**, **f**, **f**, **f**, **f**, **f**, **f**, **f**, **f**))
then tic-tac-toep1 (car (*lst*), 'x, cdr (*lst*))
else **f** **endif**

```
; We now develop a function, tic-tac-toe, that plays a game of
; tic-tac-toe against a given list of opponent moves.  We prove
; two things about it:  that it plays tic-tac-toe and that it
; never loses.  Something else we could prove about it is that
; it plays "fair", i.e., it respects the user's moves and it
; doesn't cheat by looking ahead.  For example, we could arrange
; for the program to win by ignoring the user's request to put his
; 'x in a winning spot!  Or we could look ahead to his second move
; and put our first 'o on that spot, guaranteeing that the poor
; user always makes an illegal second move!  We'll return to these
; issues after exhibiting the definition.
```

DEFINITION:
move (*x*, *i*, *board*)
= **if** *board* ≃ **nil** **then** *board*
elseif *i* ≃ 0 **then** cons (*x*, cdr (*board*))
else cons (car (*board*), move (*x*, *i* − 1, cdr (*board*))) **endif**

DEFINITION:
legalp (*i*, *board*) = ((*i* < '9) ∧ (nth (*i*, *board*) = **f**))

DEFINITION:
open-squares (*board*)
= **if** *board* ≃ **nil** **then** 0
elseif car (*board*) **then** open-squares (cdr (*board*))
else 1 + open-squares (cdr (*board*)) **endif**

THEOREM: legal-moves-reduce-open-squares
(*x* ∧ legalp (*i*, *board*))
→ (open-squares (move (*x*, *i*, *board*)) < open-squares (*board*))

THEOREM: move-never-increases-open-squares
$x \rightarrow (\text{open-squares}\,(\mathit{board}) \not< \text{open-squares}\,(\text{move}\,(x,\ i,\ \mathit{board})))$

```
; A carefully crafted aspect of pick-move, defined below, is that
; it *always* returns a number less than 9.  This required a
; subtle coding of pick-move1, where we return 0 when we fail to
; find an open square.  The reason we want pick-move always to
; return a number less than 9 has nothing to do with its
; correctness vis-a-vis tic-tac-toep.  Rather, it is concerned
; with its easy implementation.  If pick-move sometimes returns
; 9 (say), then the implementation may violate certain bounds
; (on say the size of numerically represented boards when a
; moven is carried out on 9).  The only way we can prevent it
; is to carry along the invariant that the board is not at
; and endp.  But we don't want endp in the implementation -- it
; suffices just to count the moves.  Since we just count moves, the
; legality of pick-move's answer depends on the invariant that
; the number of moves is related in a certain way to the number of
; open squares and hence to endp.  Rather than maintain this invariant,
; I am just defining pick-move1 to return 0 when it is tempted to
; return 9.
```

DEFINITION:
pick-move1 (*board*, *i*)
= **if** *board* $\simeq$ **nil then** 0
elseif car (*board*) = **f then** *i*
else pick-move1 (cdr (*board*), 1 + *i*) **endif**

DEFINITION:
pick-move (*board*)
= **let** *book-move* **be** assoc (*board*,

```
                    '(((*1*false *1*false *1*false
                        *1*false o x o x x)
                       . 2)
                      ((*1*false *1*false *1*false x o
                        *1*false o x x)
                       . 2)
                      ((*1*false x x *1*false o o o x
                        x)
                       . 3)
                      ((x *1*false x *1*false o o o x
                        x)
                       . 3)
                      ((*1*false *1*false x *1*false o
```

```
  *1*false o x x)
 . 5)
((*1*false x *1*false *1*false o
  *1*false o x x)
 . 2)
((x *1*false *1*false *1*false o
  *1*false o x x)
 . 2)
((*1*false *1*false *1*false
  *1*false o *1*false *1*false x
  x)
 . 6)
((*1*false *1*false *1*false
  *1*false o x x o x)
 . 1)
((*1*false *1*false *1*false x o
  *1*false x o x)
 . 1)
((*1*false *1*false x *1*false o
  *1*false x o x)
 . 1)
((o x x *1*false o *1*false x o
  x)
 . 5)
((*1*false *1*false *1*false
  *1*false o *1*false x *1*false
  x)
 . 7)
((*1*false *1*false o *1*false o
  x *1*false x x)
 . 6)
((*1*false *1*false o x o x x o
  x)
 . 1)
((*1*false *1*false o *1*false o
  x x *1*false x)
 . 7)
((*1*false *1*false o x o x
  *1*false *1*false x)
 . 6)
((*1*false x o *1*false o x
  *1*false *1*false x)
 . 6)
((x *1*false o *1*false o x
```

```
  *1*false *1*false x)
 . 6)
((*1*false *1*false *1*false
  *1*false o x *1*false *1*false
  x)
 . 2)
((*1*false *1*false *1*false x o
  *1*false o x x)
 . 2)
((*1*false *1*false *1*false x o
  x o *1*false x)
 . 2)
((*1*false *1*false x x o
  *1*false o *1*false x)
 . 5)
((*1*false x *1*false x o
  *1*false o *1*false x)
 . 2)
((x *1*false *1*false x o
  *1*false o *1*false x)
 . 2)
((*1*false *1*false *1*false x o
  *1*false *1*false *1*false x)
 . 6)
((*1*false *1*false x *1*false o
  o *1*false x x)
 . 3)
((*1*false *1*false x *1*false o
  o x *1*false x)
 . 3)
((o *1*false x x o o *1*false x
  x)
 . 6)
((o *1*false x x o o x *1*false
  x)
 . 7)
((*1*false x x *1*false o o
  *1*false *1*false x)
 . 3)
((x *1*false x *1*false o o
  *1*false *1*false x)
 . 3)
((*1*false *1*false x *1*false o
  *1*false *1*false *1*false x)
```

```
 . 5)
((*1*false x o *1*false o
  *1*false *1*false x x)
 . 6)
((*1*false x o *1*false o
  *1*false x *1*false x)
 . 7)
((*1*false x o *1*false o x
  *1*false *1*false x)
 . 6)
((*1*false x o x o *1*false
  *1*false *1*false x)
 . 6)
((x x o *1*false o *1*false
  *1*false *1*false x)
 . 6)
((*1*false x *1*false *1*false o
  *1*false *1*false *1*false x)
 . 2)
((x o x *1*false o *1*false o x
  x)
 . 5)
((x o *1*false *1*false o
  *1*false *1*false x x)
 . 6)
((x o *1*false *1*false o
  *1*false x *1*false x)
 . 7)
((x o *1*false *1*false o x
  *1*false *1*false x)
 . 7)
((x o *1*false x o *1*false
  *1*false *1*false x)
 . 7)
((x o x *1*false o *1*false
  *1*false *1*false x)
 . 7)
((*1*false *1*false *1*false
  *1*false *1*false *1*false
  *1*false *1*false x)
 . 4)
((*1*false *1*false *1*false
  *1*false o x o x x)
 . 2)
```

```
((*1*false *1*false *1*false x o
  *1*false o x x)
 . 2)
((*1*false x x *1*false o o o x
  x)
 . 3)
((x *1*false x *1*false o o o x
  x)
 . 3)
((*1*false *1*false x *1*false o
  *1*false o x x)
 . 5)
((*1*false x *1*false *1*false o
  *1*false o x x)
 . 2)
((x *1*false *1*false *1*false o
  *1*false o x x)
 . 2)
((*1*false *1*false *1*false
  *1*false o *1*false *1*false x
  x)
 . 6)
((x *1*false x o o *1*false x x
  o)
 . 5)
((x x *1*false o o *1*false x x
  o)
 . 5)
((x *1*false *1*false *1*false o
  *1*false x x o)
 . 3)
((*1*false *1*false *1*false
  *1*false o *1*false x x
  *1*false)
 . 8)
((x o *1*false *1*false o x x x
  o)
 . 3)
((x o *1*false x o x *1*false x
  o)
 . 6)
((*1*false *1*false *1*false
  *1*false o x *1*false x
  *1*false)
```

```
 . 8)
((*1*false *1*false *1*false x o
  *1*false o x x)
 . 2)
((*1*false *1*false *1*false x o
  x o x *1*false)
 . 2)
((o *1*false x x o *1*false o x
  x)
 . 5)
((o *1*false x x o x o x *1*false)
 . 8)
((o x x x o *1*false o x *1*false)
 . 8)
((*1*false x *1*false x o
  *1*false o x *1*false)
 . 2)
((x *1*false *1*false x o
  *1*false o x *1*false)
 . 2)
((*1*false *1*false *1*false x o
  *1*false *1*false x *1*false)
 . 6)
((x o x x o *1*false *1*false x
  o)
 . 6)
((*1*false *1*false x *1*false o
  *1*false *1*false x *1*false)
 . 8)
((*1*false x o *1*false o
  *1*false *1*false x x)
 . 6)
((*1*false x o x o *1*false x x
  o)
 . 5)
((x x o *1*false o *1*false x x
  o)
 . 5)
((*1*false x o *1*false o
  *1*false x x *1*false)
 . 8)
((*1*false x o *1*false o x
  *1*false x *1*false)
 . 6)
```

```
((*1*false x o x o *1*false
  *1*false x *1*false)
 . 6)
((x x o *1*false o *1*false
  *1*false x *1*false)
 . 6)
((*1*false x *1*false *1*false o
  *1*false *1*false x *1*false)
 . 2)
((x *1*false *1*false *1*false o
  *1*false o x x)
 . 2)
((x *1*false *1*false *1*false o
  x o x *1*false)
 . 2)
((x *1*false *1*false x o
  *1*false o x *1*false)
 . 2)
((x o x *1*false o *1*false o x
  x)
 . 5)
((x o x *1*false o x o x *1*false)
 . 8)
((x *1*false *1*false *1*false o
  *1*false *1*false x *1*false)
 . 6)
((*1*false *1*false *1*false
  *1*false *1*false *1*false
  *1*false x *1*false)
 . 4)
((*1*false *1*false *1*false
  *1*false o x x o x)
 . 1)
((*1*false *1*false *1*false x o
  *1*false x o x)
 . 1)
((*1*false *1*false x *1*false o
  *1*false x o x)
 . 1)
((o x x *1*false o *1*false x o
  x)
 . 5)
((*1*false *1*false *1*false
  *1*false o *1*false x *1*false
```

```
  x)
 . 7)
((x *1*false x o o *1*false x x
  o)
 . 5)
((x x *1*false o o *1*false x x
  o)
 . 5)
((x *1*false *1*false *1*false o
  *1*false x x o)
 . 3)
((*1*false *1*false *1*false
  *1*false o *1*false x x
  *1*false)
 . 8)
((x *1*false *1*false *1*false o
  x x *1*false o)
 . 3)
((*1*false *1*false *1*false
  *1*false o x x *1*false
  *1*false)
 . 8)
((o *1*false *1*false x o
  *1*false x *1*false x)
 . 7)
((o *1*false *1*false x o
  *1*false x x *1*false)
 . 8)
((o *1*false *1*false x o x x
  *1*false *1*false)
 . 8)
((o *1*false x x o *1*false x
  *1*false *1*false)
 . 8)
((o x *1*false x o *1*false x
  *1*false *1*false)
 . 8)
((*1*false o x *1*false o
  *1*false x *1*false x)
 . 7)
((*1*false o x *1*false o
  *1*false x x *1*false)
 . 8)
((*1*false o x *1*false o x x
```

```
  *1*false *1*false)
 . 7)
((*1*false o x x o *1*false x
  *1*false *1*false)
 . 7)
((x o x *1*false o *1*false x
  *1*false *1*false)
 . 7)
((*1*false *1*false x *1*false o
  *1*false x *1*false *1*false)
 . 1)
((o x x *1*false o *1*false x o
  x)
 . 5)
((o x *1*false *1*false o
  *1*false x *1*false x)
 . 7)
((o x *1*false *1*false o
  *1*false x x *1*false)
 . 8)
((o x *1*false *1*false o x x
  *1*false *1*false)
 . 8)
((o x *1*false x o *1*false x
  *1*false *1*false)
 . 8)
((o x x *1*false o *1*false x
  *1*false *1*false)
 . 8)
((x *1*false *1*false o o
  *1*false x *1*false x)
 . 5)
((x *1*false *1*false o o
  *1*false x x *1*false)
 . 5)
((x o *1*false o o x x *1*false
  x)
 . 7)
((x o *1*false o o x x x *1*false)
 . 8)
((x *1*false x o o *1*false x
  *1*false *1*false)
 . 5)
((x x *1*false o o *1*false x
```

```
  *1*false *1*false)
 . 5)
((x *1*false *1*false *1*false o
  *1*false x *1*false *1*false)
 . 3)
((*1*false *1*false *1*false
  *1*false *1*false *1*false x
  *1*false *1*false)
 . 4)
((*1*false *1*false o *1*false o
  x *1*false x x)
 . 6)
((*1*false *1*false o x o x x o
  x)
 . 1)
((*1*false *1*false o *1*false o
  x x *1*false x)
 . 7)
((*1*false *1*false o x o x
  *1*false *1*false x)
 . 6)
((*1*false x o *1*false o x
  *1*false *1*false x)
 . 6)
((x *1*false o *1*false o x
  *1*false *1*false x)
 . 6)
((*1*false *1*false *1*false
  *1*false o x *1*false *1*false
  x)
 . 2)
((x o *1*false *1*false o x x x
  o)
 . 3)
((x o *1*false x o x *1*false x
  o)
 . 6)
((*1*false *1*false *1*false
  *1*false o x *1*false x
  *1*false)
 . 8)
((x *1*false *1*false *1*false o
  x x *1*false o)
 . 3)
```

```
((*1*false *1*false *1*false
  *1*false o x x *1*false
  *1*false)
 . 8)
((o *1*false *1*false x o x
  *1*false *1*false x)
 . 2)
((o *1*false *1*false x o x
  *1*false x *1*false)
 . 8)
((o *1*false *1*false x o x x
  *1*false *1*false)
 . 8)
((o *1*false x x o x *1*false
  *1*false *1*false)
 . 8)
((o x *1*false x o x *1*false
  *1*false *1*false)
 . 8)
((x o x *1*false o x x *1*false
  o)
 . 7)
((x o x x o x *1*false *1*false
  o)
 . 7)
((*1*false *1*false x *1*false o
  x *1*false *1*false *1*false)
 . 8)
((*1*false x o *1*false o x
  *1*false *1*false x)
 . 6)
((*1*false x o *1*false o x
  *1*false x *1*false)
 . 6)
((o x o *1*false o x x *1*false
  x)
 . 7)
((o x o *1*false o x x x *1*false)
 . 8)
((o x o x o x x *1*false *1*false)
 . 8)
((*1*false x o x o x *1*false
  *1*false *1*false)
 . 6)
```

```
((x x o *1*false o x *1*false
  *1*false *1*false)
 . 6)
((*1*false x *1*false *1*false o
  x *1*false *1*false *1*false)
 . 2)
((x *1*false o *1*false o x
  *1*false *1*false x)
 . 6)
((x *1*false o *1*false o x
  *1*false x *1*false)
 . 6)
((x *1*false o o o x x *1*false
  x)
 . 7)
((x *1*false o o o x x x *1*false)
 . 8)
((x *1*false o *1*false o x x
  *1*false *1*false)
 . 3)
((x *1*false o x o x *1*false
  *1*false *1*false)
 . 6)
((x x o *1*false o x *1*false
  *1*false *1*false)
 . 6)
((x *1*false *1*false *1*false o
  x *1*false *1*false *1*false)
 . 2)
((*1*false *1*false *1*false
  *1*false *1*false x *1*false
  *1*false *1*false)
 . 4)
((o x o *1*false x *1*false
  *1*false *1*false x)
 . 7)
((o *1*false *1*false *1*false x
  *1*false *1*false *1*false x)
 . 2)
((o o x *1*false x *1*false
  *1*false x *1*false)
 . 6)
((o x o *1*false x *1*false x
  *1*false *1*false)
```

```
 . 7)
((o *1*false *1*false *1*false x
  *1*false x *1*false *1*false)
 . 2)
((o *1*false *1*false o x x
  *1*false *1*false x)
 . 6)
((o *1*false *1*false o x x
  *1*false x *1*false)
 . 6)
((o *1*false *1*false o x x x
  *1*false *1*false)
 . 2)
((o *1*false x o x x *1*false
  *1*false *1*false)
 . 6)
((o x *1*false o x x *1*false
  *1*false *1*false)
 . 6)
((o *1*false *1*false *1*false x
  x *1*false *1*false *1*false)
 . 3)
((o x o x x o x *1*false *1*false)
 . 8)
((o *1*false *1*false x x o x
  *1*false *1*false)
 . 2)
((o *1*false x x x o *1*false
  *1*false *1*false)
 . 6)
((o x *1*false x x o *1*false
  *1*false *1*false)
 . 7)
((o *1*false *1*false x x
  *1*false *1*false *1*false
  *1*false)
 . 5)
((o *1*false x *1*false x
  *1*false o *1*false x)
 . 3)
((o *1*false x *1*false x
  *1*false o x *1*false)
 . 3)
((o *1*false x *1*false x x o
```

```
  *1*false *1*false)
 . 3)
((o *1*false x x x *1*false o
  *1*false *1*false)
 . 5)
((o *1*false x *1*false x
  *1*false *1*false *1*false
  *1*false)
 . 6)
((o x *1*false o x x *1*false o
  x)
 . 6)
((o x *1*false *1*false x x
  *1*false o *1*false)
 . 3)
((o x *1*false x x *1*false
  *1*false o *1*false)
 . 5)
((o x x *1*false x x o o *1*false)
 . 8)
((o x x x x *1*false o o *1*false)
 . 8)
((o x x *1*false x *1*false
  *1*false o *1*false)
 . 6)
((o x *1*false *1*false x
  *1*false *1*false *1*false
  *1*false)
 . 7)
((*1*false *1*false *1*false x o
  *1*false o x x)
 . 2)
((*1*false *1*false *1*false x o
  x o *1*false x)
 . 2)
((*1*false *1*false x x o
  *1*false o *1*false x)
 . 5)
((*1*false x *1*false x o
  *1*false o *1*false x)
 . 2)
((x *1*false *1*false x o
  *1*false o *1*false x)
 . 2)
```

```
((*1*false *1*false *1*false x o
  *1*false *1*false *1*false x)
 . 6)
((*1*false *1*false *1*false x o
  *1*false o x x)
 . 2)
((*1*false *1*false *1*false x o
  x o x *1*false)
 . 2)
((o *1*false x x o *1*false o x
  x)
 . 5)
((o *1*false x x o x o x *1*false)
 . 8)
((o x x x o *1*false o x *1*false)
 . 8)
((*1*false x *1*false x o
  *1*false o x *1*false)
 . 2)
((x *1*false *1*false x o
  *1*false o x *1*false)
 . 2)
((*1*false *1*false *1*false x o
  *1*false *1*false x *1*false)
 . 6)
((o *1*false *1*false x o
  *1*false x *1*false x)
 . 7)
((o *1*false *1*false x o
  *1*false x x *1*false)
 . 8)
((o *1*false *1*false x o x x
  *1*false *1*false)
 . 8)
((o *1*false x x o *1*false x
  *1*false *1*false)
 . 8)
((o x *1*false x o *1*false x
  *1*false *1*false)
 . 8)
((o *1*false *1*false x o x
  *1*false *1*false x)
 . 2)
((o *1*false *1*false x o x
```

```
  *1*false x *1*false)
 . 8)
((o *1*false *1*false x o x x
  *1*false *1*false)
 . 8)
((o *1*false x x o x *1*false
  *1*false *1*false)
 . 8)
((o x *1*false x o x *1*false
  *1*false *1*false)
 . 8)
((o *1*false x x o o *1*false x
  x)
 . 6)
((o *1*false x x o o x *1*false
  x)
 . 7)
((o *1*false x x o *1*false
  *1*false *1*false x)
 . 5)
((o *1*false x x o *1*false
  *1*false x *1*false)
 . 8)
((o *1*false x x o *1*false x
  *1*false *1*false)
 . 8)
((o *1*false x x o x *1*false
  *1*false *1*false)
 . 8)
((o x x x o *1*false *1*false
  *1*false *1*false)
 . 8)
((o x o x o *1*false *1*false x
  x)
 . 6)
((o x o x o *1*false x *1*false
  x)
 . 7)
((o x *1*false x o *1*false
  *1*false x *1*false)
 . 8)
((o x *1*false x o *1*false x
  *1*false *1*false)
 . 8)
```

```
((o x *1*false x o x *1*false
  *1*false *1*false)
 . 8)
((o x x x o *1*false *1*false
  *1*false *1*false)
 . 8)
((x *1*false *1*false x o
  *1*false o *1*false x)
 . 2)
((x *1*false *1*false x o
  *1*false o x *1*false)
 . 2)
((x *1*false *1*false x o x o
  *1*false *1*false)
 . 2)
((x o x x o *1*false o *1*false
  x)
 . 7)
((x *1*false *1*false x o
  *1*false *1*false *1*false
  *1*false)
 . 6)
((*1*false *1*false *1*false x
  *1*false *1*false *1*false
  *1*false *1*false)
 . 4)
((*1*false *1*false x *1*false o
  o *1*false x x)
 . 3)
((*1*false *1*false x *1*false o
  o x *1*false x)
 . 3)
((o *1*false x x o o *1*false x
  x)
 . 6)
((o *1*false x x o o x *1*false
  x)
 . 7)
((*1*false x x *1*false o o
  *1*false *1*false x)
 . 3)
((x *1*false x *1*false o o
  *1*false *1*false x)
 . 3)
```

```
((*1*false *1*false x *1*false o
  *1*false *1*false *1*false x)
 . 5)
((x o x x o *1*false *1*false x
  o)
 . 6)
((*1*false *1*false x *1*false o
  *1*false *1*false x *1*false)
 . 8)
((*1*false o x *1*false o
  *1*false x *1*false x)
 . 7)
((*1*false o x *1*false o
  *1*false x x *1*false)
 . 8)
((*1*false o x *1*false o x x
  *1*false *1*false)
 . 7)
((*1*false o x x o *1*false x
  *1*false *1*false)
 . 7)
((x o x *1*false o *1*false x
  *1*false *1*false)
 . 7)
((*1*false *1*false x *1*false o
  *1*false x *1*false *1*false)
 . 1)
((x o x *1*false o x x *1*false
  o)
 . 7)
((x o x x o x *1*false *1*false
  o)
 . 7)
((*1*false *1*false x *1*false o
  x *1*false *1*false *1*false)
 . 8)
((o *1*false x x o o *1*false x
  x)
 . 6)
((o *1*false x x o o x *1*false
  x)
 . 7)
((o *1*false x x o *1*false
  *1*false *1*false x)
```

```
 . 5)
((o *1*false x x o *1*false
  *1*false x *1*false)
 . 8)
((o *1*false x x o *1*false x
  *1*false *1*false)
 . 8)
((o *1*false x x o x *1*false
  *1*false *1*false)
 . 8)
((o x x x o *1*false *1*false
  *1*false *1*false)
 . 8)
((o x x *1*false o *1*false
  *1*false *1*false x)
 . 5)
((o x x *1*false o *1*false
  *1*false x *1*false)
 . 8)
((o x x *1*false o *1*false x
  *1*false *1*false)
 . 8)
((o x x *1*false o x *1*false
  *1*false *1*false)
 . 8)
((o x x x o *1*false *1*false
  *1*false *1*false)
 . 8)
((x o x *1*false o *1*false
  *1*false *1*false x)
 . 7)
((x o x o o x *1*false x *1*false)
 . 8)
((x o x *1*false o *1*false x
  *1*false *1*false)
 . 7)
((x o x *1*false o x *1*false
  *1*false *1*false)
 . 7)
((x o x x o *1*false *1*false
  *1*false *1*false)
 . 7)
((*1*false *1*false x *1*false
  *1*false *1*false *1*false
```

```
  *1*false *1*false)
 . 4)
((*1*false x o *1*false o
  *1*false *1*false x x)
 . 6)
((*1*false x o *1*false o
  *1*false x *1*false x)
 . 7)
((*1*false x o *1*false o x
  *1*false *1*false x)
 . 6)
((*1*false x o x o *1*false
  *1*false *1*false x)
 . 6)
((x x o *1*false o *1*false
  *1*false *1*false x)
 . 6)
((*1*false x *1*false *1*false o
  *1*false *1*false *1*false x)
 . 2)
((*1*false x o *1*false o
  *1*false *1*false x x)
 . 6)
((*1*false x o x o *1*false x x
  o)
 . 5)
((x x o *1*false o *1*false x x
  o)
 . 5)
((*1*false x o *1*false o
  *1*false x x *1*false)
 . 8)
((*1*false x o *1*false o x
  *1*false x *1*false)
 . 6)
((*1*false x o x o *1*false
  *1*false x *1*false)
 . 6)
((x x o *1*false o *1*false
  *1*false x *1*false)
 . 6)
((*1*false x *1*false *1*false o
  *1*false *1*false x *1*false)
 . 2)
```

```
((o x x *1*false o *1*false x o
  x)
 . 5)
((o x *1*false *1*false o
  *1*false x *1*false x)
 . 7)
((o x *1*false *1*false o
  *1*false x x *1*false)
 . 8)
((o x *1*false *1*false o x x
  *1*false *1*false)
 . 8)
((o x *1*false x o *1*false x
  *1*false *1*false)
 . 8)
((o x x *1*false o *1*false x
  *1*false *1*false)
 . 8)
((*1*false x o *1*false o x
  *1*false *1*false x)
 . 6)
((*1*false x o *1*false o x
  *1*false x *1*false)
 . 6)
((o x o *1*false o x x *1*false
  x)
 . 7)
((o x o *1*false o x x x *1*false)
 . 8)
((o x o x o x x *1*false *1*false)
 . 8)
((*1*false x o x o x *1*false
  *1*false *1*false)
 . 6)
((x x o *1*false o x *1*false
  *1*false *1*false)
 . 6)
((*1*false x *1*false *1*false o
  x *1*false *1*false *1*false)
 . 2)
((o x o x o *1*false *1*false x
  x)
 . 6)
((o x o x o *1*false x *1*false
```

```
  x)
 . 7)
((o x *1*false x o *1*false
  *1*false x *1*false)
 . 8)
((o x *1*false x o *1*false x
  *1*false *1*false)
 . 8)
((o x *1*false x o x *1*false
  *1*false *1*false)
 . 8)
((o x x x o *1*false *1*false
  *1*false *1*false)
 . 8)
((o x x *1*false o *1*false
  *1*false *1*false x)
 . 5)
((o x x *1*false o *1*false
  *1*false x *1*false)
 . 8)
((o x x *1*false o *1*false x
  *1*false *1*false)
 . 8)
((o x x *1*false o x *1*false
  *1*false *1*false)
 . 8)
((o x x x o *1*false *1*false
  *1*false *1*false)
 . 8)
((x x o *1*false o *1*false
  *1*false *1*false x)
 . 6)
((x x o *1*false o *1*false
  *1*false x *1*false)
 . 6)
((x x o *1*false o x *1*false
  *1*false *1*false)
 . 6)
((x x o x o *1*false *1*false
  *1*false *1*false)
 . 6)
((*1*false x *1*false *1*false
  *1*false *1*false *1*false
  *1*false *1*false)
```

```
 . 4)
((x o x *1*false o *1*false o x
  x)
 . 5)
((x o *1*false *1*false o
  *1*false *1*false x x)
 . 6)
((x o *1*false *1*false o
  *1*false x *1*false x)
 . 7)
((x o *1*false *1*false o x
  *1*false *1*false x)
 . 7)
((x o *1*false x o *1*false
  *1*false *1*false x)
 . 7)
((x o x *1*false o *1*false
  *1*false *1*false x)
 . 7)
((x *1*false *1*false *1*false o
  *1*false o x x)
 . 2)
((x *1*false *1*false *1*false o
  x o x *1*false)
 . 2)
((x *1*false *1*false x o
  *1*false o x *1*false)
 . 2)
((x o x *1*false o *1*false o x
  x)
 . 5)
((x o x *1*false o x o x *1*false)
 . 8)
((x *1*false *1*false *1*false o
  *1*false *1*false x *1*false)
 . 6)
((x *1*false *1*false o o
  *1*false x *1*false x)
 . 5)
((x *1*false *1*false o o
  *1*false x x *1*false)
 . 5)
((x o *1*false o o x x *1*false
  x)
```

```
 . 7)
((x o *1*false o o x x x *1*false)
 . 8)
((x *1*false x o o *1*false x
  *1*false *1*false)
 . 5)
((x x *1*false o o *1*false x
  *1*false *1*false)
 . 5)
((x *1*false *1*false *1*false o
  *1*false x *1*false *1*false)
 . 3)
((x *1*false o *1*false o x
  *1*false *1*false x)
 . 6)
((x *1*false o *1*false o x
  *1*false x *1*false)
 . 6)
((x *1*false o o o x x *1*false
  x)
 . 7)
((x *1*false o o o x x x *1*false)
 . 8)
((x *1*false o *1*false o x x
  *1*false *1*false)
 . 3)
((x *1*false o x o x *1*false
  *1*false *1*false)
 . 6)
((x x o *1*false o x *1*false
  *1*false *1*false)
 . 6)
((x *1*false *1*false *1*false o
  x *1*false *1*false *1*false)
 . 2)
((x *1*false *1*false x o
  *1*false o *1*false x)
 . 2)
((x *1*false *1*false x o
  *1*false o x *1*false)
 . 2)
((x *1*false *1*false x o x o
  *1*false *1*false)
 . 2)
```

```
((x o x x o *1*false o *1*false
  x)
 . 7)
((x *1*false *1*false x o
  *1*false *1*false *1*false
  *1*false)
 . 6)
((x o x *1*false o *1*false
  *1*false *1*false x)
 . 7)
((x o x o o x *1*false x *1*false)
 . 8)
((x o x *1*false o *1*false x
  *1*false *1*false)
 . 7)
((x o x *1*false o x *1*false
  *1*false *1*false)
 . 7)
((x o x x o *1*false *1*false
  *1*false *1*false)
 . 7)
((x x o *1*false o *1*false
  *1*false *1*false x)
 . 6)
((x x o *1*false o *1*false
  *1*false x *1*false)
 . 6)
((x x o *1*false o x *1*false
  *1*false *1*false)
 . 6)
((x x o x o *1*false *1*false
  *1*false *1*false)
 . 6)
((x *1*false *1*false *1*false
  *1*false *1*false *1*false
  *1*false *1*false)
 . 4)))
```

in
if *book-move* **then** cdr (*book-move*)
else pick-move1 (*board*, `0`) **endif endlet**

EVENT: Disable pick-move.

```
; Note that in the following automation of the game, we do not even
```

```
; check that the user might have won nor do we check that our moves
; are legal.  We know these checks are redundant.  It is worth noting
; that our formal spec of the game will force us to prove this
; redundancy.  Suppose the user did win but we then replied with another
; move.  Then the game would be ill-formed.  Similarly, if our move
; is illegal, either because our 'o replaces an 'x or because our 'o
; goes into a nonexistent square, the resulting board will not pass
; legal-movep.
```

DEFINITION:
tic-tac-toe1 (*xmoves*, *board*)
= **let** *board1* **be** move ('x, car (*xmoves*), *board*)
in
if legalp (car (*xmoves*), *board*)
then if endp (*board1*) **then** list (*board1*, 'draw)
else let *board2* **be** move ('o,
pick-move (*board1*),
board1)
in
if winp ('o, *board2*)
then list (*board1*, *board2*, 'o-win)
else cons (*board1*,
cons (*board2*,
tic-tac-toe1 (cdr (*xmoves*),
board2))) **endif endlet endif**
else list (*board1*, 'o-win) **endif endlet**

DEFINITION:
tic-tac-toe (*xmoves*)
= **let** *board0* **be** list (**f**, **f**, **f**, **f**, **f**, **f**, **f**, **f**, **f**)
in
cons (*board0*, tic-tac-toe1 (*xmoves*, *board0*)) **endlet**

```
; We now set about trying to prove that tic-tac-toe plays the game.
; We start with some fundamentals that I know will be needed even
; though I haven't seen their explicit use.
```

DEFINITION:
length (*x*)
= **if** *x* ≃ **nil** **then** 0
else 1 + length (cdr (*x*)) **endif**

THEOREM: length-move
length (move (*x*, *i*, *board*)) = length (*board*)

DEFINITION:
legal-bookp (*alist*)
= **if** *alist* ≃ **nil then t**
else legalp (cdar (*alist*), caar (*alist*))
∧ legal-bookp (cdr (*alist*)) **endif**

THEOREM: legal-bookp-implies-legalp-cdr-assoc
(legal-bookp (*alist*) ∧ assoc (*board*, *alist*))
→ legalp (cdr (assoc (*board*, *alist*)), *board*)

DEFINITION:
pick-move2 (*board*)
= **if** *board* ≃ **nil then** 0
elseif car (*board*) = **f then** 0
else 1 + pick-move2 (cdr (*board*)) **endif**

```
; The following theorem probably holds a record for successfully
; completed subgoals:

;     That finishes the proof of *1.1.1.1.1.1.1.1.1.1, which, in turn, finishes
;the proof of *1.1.1.1.1.1.1.1.1, which finishes the proof of *1.1.1.1.1.1.1.1,
;which finishes the proof of *1.1.1.1.1.1.1, which, consequently, finishes the
;proof of *1.1.1.1.1.1, which finishes the proof of *1.1.1.1.1, which finishes
;the proof of *1.1.1.1, which finishes the proof of *1.1.1, which, in turn,
;finishes the proof of *1.1, which finishes the proof of *1.  Q.E.D.

;[ 0.0 6.5 3.4 ]   (On a Sparc 2)

; This would be even more impressive if boards had length 1000!
```

THEOREM: not-endp-implies-legalp-pick-move2
((¬ endp (*board*)) ∧ (length (*board*) < 10))
→ legalp (pick-move2 (*board*), *board*)

THEOREM: pick-move1-is-pick-move2
((¬ endp (*board*)) ∧ (*i* ∈ **N**))
→ (pick-move1 (*board*, *i*) = (*i* + pick-move2 (*board*)))

```
; If you don't do the following we get stack overflow while trying to confirm
; that the alist built into pick-moves is a legal-bookp.
```

EVENT: For efficiency, compile those definitions not yet compiled.

THEOREM: legalp-pick-move
((¬ endp (*board*)) ∧ (length (*board*) < 10))
→ legalp (pick-move (*board*), *board*)

THEOREM: boardp-move
(boardp (*board*) ∧ ((*player* = 'x) ∨ (*player* = 'o)))
→ boardp (move (*player*, *i*, *board*))

```
; We now establish that pick-move indeed has our "carefully crafted
; aspect" of less than 9, when (endp board)!
```

THEOREM: legalp-implies-lessp-9
(legalp (cdr (assoc (*board*, *alist*)), *board*) ∧ (length (*board*) = 9))
→ (cdr (assoc (*board*, *alist*)) < 9)

THEOREM: endp-implies-pick-move1-0
endp (*board*) → (pick-move1 (*board*, *i*) = 0)

THEOREM: lessp-pick-move
(length (*board*) = 9) → (pick-move (*board*) < 9)

```
; The strategy of our proof that tic-tac-toe never loses is to arrange
; for tic-tac-toe to expand so as to play all possible games!  The first
; step is the lemma below, which forces the function open provided the board
; is given as an explicit list of 9 elements.  The lemma below is just
; the equation of (tic-tac-toe xmoves board) with its body, for board
; replaced by an explicit list.
```

THEOREM: tic-tac-toe1-opener
tic-tac-toe1 (*xmoves*, list (*s0*, *s1*, *s2*, *s3*, *s4*, *s5*, *s6*, *s7*, *s8*))
= **if** legalp (car (*xmoves*), list (*s0*, *s1*, *s2*, *s3*, *s4*, *s5*, *s6*, *s7*, *s8*))
then if endp (move ('x, car (*xmoves*), list (*s0*, *s1*, *s2*, *s3*, *s4*, *s5*, *s6*, *s7*, *s8*)))
then list (move ('x, car (*xmoves*), list (*s0*, *s1*, *s2*, *s3*, *s4*, *s5*, *s6*, *s7*, *s8*)),
'draw)
elseif winp ('o,
move ('o,
pick-move (move ('x,
car (*xmoves*),
list (*s0*, *s1*, *s2*, *s3*, *s4*, *s5*, *s6*, *s7*, *s8*))),
move ('x,
car (*xmoves*),
list (*s0*, *s1*, *s2*, *s3*, *s4*, *s5*, *s6*, *s7*, *s8*))))
then list (move ('x, car (*xmoves*), list (*s0*, *s1*, *s2*, *s3*, *s4*, *s5*, *s6*, *s7*, *s8*)),

```
                move ('o,
                      pick-move (move ('x,
                                       car (xmoves),
                                       list (s0, s1, s2, s3, s4, s5, s6, s7, s8))),
                      move ('x,
                            car (xmoves),
                            list (s0, s1, s2, s3, s4, s5, s6, s7, s8))),
                'o-win)
else cons (move ('x,
                 car (xmoves),
                 list (s0, s1, s2, s3, s4, s5, s6, s7, s8)),
           cons (move ('o,
                       pick-move (move ('x,
                                        car (xmoves),
                                        list (s0,
                                              s1,
                                              s2,
                                              s3,
                                              s4,
                                              s5,
                                              s6,
                                              s7,
                                              s8))),
                       move ('x,
                             car (xmoves),
                             list (s0, s1, s2, s3, s4, s5, s6, s7, s8))),
                 tic-tac-toe1 (cdr (xmoves),
                               move ('o,
                                     pick-move (move ('x,
                                                      car (xmoves),
                                                      list (s0,
                                                            s1,
                                                            s2,
                                                            s3,
                                                            s4,
                                                            s5,
                                                            s6,
                                                            s7,
                                                            s8))),
                                     move ('x,
                                           car (xmoves),
                                           list (s0,
                                                 s1,
                                                 s2,
```

s3,
s4,
s5,
s6,
s7,
s8)))))) **endif**
else list (move ('x, car (*xmoves*), list (*s0*, *s1*, *s2*, *s3*, *s4*, *s5*, *s6*, *s7*, *s8*)),
'o-win) **endif**

EVENT: Disable tic-tac-toe1.

```
; With that lemma in place, the function will expand repeatedly, as soon as
; board2 is reduced to normal form.

; The next problem is to force the first legalp expression above to cause us
; to case split on all the legal moves on the given board.  Our first attempt
; at this proof just let legalp expand and we considered all 9 possible moves.
; But this caused a tremendous explosion.

; The sequence of lemmas below culminate in the following wondrous
; effect.  (legalp x (list s0 s1 ... s8)) expands to (or ...(equal x
; i)...)  for just those i such that si is f.  To achieve this we
; transform legalp in two steps.  First we show that it is equal to
; legalp1, below, which does not use NTH and instead decrements x and
; cdrs board.  Then we transform legalp1 into legalp2, which keeps x
; unchanged but cdrs board as it increments an accumulator i.
; Finally, we prove the opener lemmas that let the cdring of board
; drive the production of a bunch of disjuncts against the running i.
; This development was fairly subtle for me.
```

DEFINITION:
legalp1 (*x*, *board*)
= **if** *x* ≃ 0 **then** ¬ car (*board*)
elseif *board* ≃ **nil then f**
else legalp1 (*x* − 1, cdr (*board*)) **endif**

THEOREM: legalp1-fact1
(*x* ≮ length (*board*)) → (¬ legalp1 (*x*, *board*))

THEOREM: legalp1-fact2
nth (*x*, *board*) → (¬ legalp1 (*x*, *board*))

THEOREM: legalp1-fact3
((*x* < length (*board*)) ∧ (¬ nth (*x*, *board*))) → legalp1 (*x*, *board*)

```
; So here is the first half...
```

THEOREM: legalp-is-legalp1
$(\text{length}\,(\mathit{board}) = \texttt{'9}) \rightarrow (\text{legalp}\,(x,\ \mathit{board}) = \text{legalp1}\,(x,\ \mathit{board}))$

```
; Now we introduce legalp2.
```

DEFINITION:
legalp2 $(x,\ \mathit{board},\ i)$
$=$ **if** $\mathit{board} \simeq$ **nil then f**
elseif $(\neg\ \text{car}\,(\mathit{board})) \wedge (\text{fix}\,(x) = i)$ **then t**
else legalp2 $(x,\ \text{cdr}\,(\mathit{board}),\ 1 + i)$ **endif**

THEOREM: legalp2-fact1
$(x < i) \rightarrow (\neg\ \text{legalp2}\,(x,\ \mathit{board},\ i))$

THEOREM: sub1-difference
$((x - y) - 1) = ((x - 1) - y)$

THEOREM: legalp1-is-legalp2-gen
$((i \in \mathbf{N}) \wedge (x \not< i)) \rightarrow (\text{legalp1}\,(x - i,\ \mathit{board}) = \text{legalp2}\,(x,\ \mathit{board},\ i))$

EVENT: Disable sub1-difference.

```
; Here is the second half...
```

THEOREM: legalp1-is-legalp2
$\text{legalp1}\,(x,\ \mathit{board}) = \text{legalp2}\,(x,\ \mathit{board},\ \texttt{'0})$

```
; Now we'll show how to use semi-explicit boards to drive the production
; of disjuncts.
```

THEOREM: legalp2-opener-f
$(i \in \mathbf{N})$
$\rightarrow$ (legalp2 $(x,\ \text{cons}\,(\mathbf{f},\ \mathit{board}),\ i)$
$=$ (**if** $i = \texttt{0}$ **then** $x \simeq \texttt{0}$
else $x = i$ **endif**
$\vee$ legalp2 $(x,\ \mathit{board},\ 1 + i)$))

THEOREM: legalp2-opener-x
$\text{legalp2}\,(x,\ \text{cons}\,(\texttt{'x},\ \mathit{board}),\ i) = \text{legalp2}\,(x,\ \mathit{board},\ 1 + i)$

THEOREM: legalp2-opener-o
legalp2 (*x*, cons (`'o`, *board*), *i*) = legalp2 (*x*, *board*, 1 + *i*)

THEOREM: legalp2-opener-nil
legalp2 (*x*, `'nil`, *i*) = **f**

EVENT: Disable legalp2.

EVENT: Disable legalp.

```
; And now we put it all together and cut out the middlemen:
```

THEOREM: legalp-simplifier
legalp (*x*, list (*s0*, *s1*, *s2*, *s3*, *s4*, *s5*, *s6*, *s7*, *s8*))
= legalp2 (*x*, list (*s0*, *s1*, *s2*, *s3*, *s4*, *s5*, *s6*, *s7*, *s8*), 0)

EVENT: Disable legalp-is-legalp1.

EVENT: Disable legalp1-is-legalp2.

EVENT: Disable member.

EVENT: Disable move.

THEOREM: noop-move-gen
(*i* ≮ length (*board*)) → (move (*x*, *i*, *board*) = *board*)

THEOREM: noop-move
((length (*board*) = 9)
∧ (*i* ∈ **N**)
∧ (*i* ≠ 0)
∧ (*i* ≠ 1)
∧ (*i* ≠ 2)
∧ (*i* ≠ 3)
∧ (*i* ≠ 4)
∧ (*i* ≠ 5)
∧ (*i* ≠ 6)
∧ (*i* ≠ 7)
∧ (*i* ≠ 8))
→ (move (*x*, *i*, *board*) = *board*)

THEOREM: move-zerop
$((\text{length}\,(\textit{board}) = 9) \wedge (i \notin \mathbf{N}))$
$\rightarrow \quad (\text{move}\,(x,\ i,\ \textit{board}) = \text{cons}\,(x,\ \text{cdr}\,(\textit{board})))$

THEOREM: not-legal-movep-x-x
$\neg$ legal-movep (*board*, `'x`, *board*)

THEOREM: not-legal-movep-if-occupied
$\text{nth}\,(i,\ \textit{board}) \rightarrow (\neg\ \text{legal-movep}\,(\textit{board},\ \texttt{'x},\ \text{move}\,(\texttt{'x},\ i,\ \textit{board})))$

THEOREM: not-legal-movep-if-too-big
$(i \not< \text{length}\,(\textit{board})) \rightarrow (\neg\ \text{legal-movep}\,(\textit{board},\ \texttt{'x},\ \text{move}\,(\texttt{'x},\ i,\ \textit{board})))$

```
; This is another hideously silly descent from 9 to 0 in subgoals.  But, hey,
; it works...
```

THEOREM: legal-movep-move
$(\text{boardp}\,(\textit{board}) \wedge (\text{length}\,(\textit{board}) = 9))$
$\rightarrow \quad (\text{legal-movep}\,(\textit{board},\ \texttt{'x},\ \text{move}\,(\texttt{'x},\ i,\ \textit{board})) = \text{legalp}\,(i,\ \textit{board}))$

EVENT: Disable legal-movep.

```
; The following theorem produces 4.8 Mb of .proofs output.  The proof
; takes roughly 7.5 minutes and generates 2930 cases.
```

THEOREM: tic-tac-toe-correct1
$\text{tic-tac-toep}\,(\text{tic-tac-toe}\,(\textit{xmoves})) \in$ `'(draw o-win)`

EVENT: Disable tic-tac-toe1-opener.

EVENT: Disable legalp-simplifier.

```
; Comments on What is Not Proved

; This is not a very satisfying specification for an implementation of tic-tac-toe.
; I can think of two ways to cheat while satisfying this spec.

; (1) Return a fixed game or, more generally, ignore the user's moves
;     when they get in your way of winning.  One could still prove the
;     above theorem.

; (2) Look ahead into the user's moves and always make your first move
```

```
;     to the square to which he'll make his second.  This guarantees
;     that he will always play illegal games and lose by default.

; We could avoid (1) by proving another theorem that says "the user's
; actual moves (extracted from the sequence of boards returned by
; tic-tac-toe) is an initial subsequence of the specified moves."
; I think this would not be hard to prove.

; We could avoid (2) by proving a theorem that says "If two lists of user
; moves, xmoves1 and xmoves2, agree on the first i moves, then
; (tic-tac-toe xmoves1) and (tic-tac-toe xmoves2) agree on the first
; 2i boards."  I believe this theorem rules out the possibility that
; the machine is looking ahead.

; We address these two concerns below, in slightly reformulated ways.

; We now turn to the next problem, proving that the function honors
; the requested moves by 'x.
```

DEFINITION:
x-moves-honored1 (*board*, *xmoves*, *lst*)
= **if** *lst* ≃ **nil** **then** **t**
elseif cdr (*lst*) ≃ **nil** **then** **t**
else (car (*lst*) = move ('x, car (*xmoves*), *board*))
∧ x-moves-honored1 (cadr (*lst*), cdr (*xmoves*), cddr (*lst*)) **endif**

DEFINITION:
x-moves-honored (*xmoves*, *lst*) = x-moves-honored1 (car (*lst*), *xmoves*, cdr (*lst*))

THEOREM: tic-tac-toe-correct2-lemma
boardp (*board*) → x-moves-honored1 (*board*, *xmoves*, tic-tac-toe1 (*xmoves*, *board*))

THEOREM: tic-tac-toe-correct2
x-moves-honored (*xmoves*, tic-tac-toe (*xmoves*))

```
; The final conjunction of the specification is that the game playing
; program does not look-ahead into xmoves!
```

DEFINITION:
init-game1 (*n*, *lst*)
= **if** *n* ≃ 0 **then** **nil**
elseif *lst* ≃ **nil** **then** **nil**
elseif cdr (*lst*) ≃ **nil** **then** list (car (*lst*))
else cons (car (*lst*), cons (cadr (*lst*), init-game1 (*n* − 1, cddr (*lst*)))) **endif**

DEFINITION:
init-game (n, *lst*) = cons (car (*lst*), init-game1 (n, cdr (*lst*)))

THEOREM: tic-tac-toe-correct3-lemma
(init-game1 (length (*opening*), tic-tac-toe1 (append (*opening*, *xmoves1*), *board*))
= init-game1 (length (*opening*),
tic-tac-toe1 (append (*opening*, *xmoves2*), *board*)))
= **t**

THEOREM: tic-tac-toe-correct3
(init-game (length (*opening*), tic-tac-toe (append (*opening*, *xmoves1*)))
= init-game (length (*opening*), tic-tac-toe (append (*opening*, *xmoves2*))))
= **t**

THEOREM: tic-tac-toe-correct
(tic-tac-toep (tic-tac-toe (*xmoves*)) ∈ '(draw o-win))
∧ x-moves-honored (*xmoves*, tic-tac-toe (*xmoves*))
∧ (init-game (length (*opening*), tic-tac-toe (append (*opening*, *xmoves1*)))
= init-game (length (*opening*),
tic-tac-toe (append (*opening*, *xmoves2*))))

```
; The following theorem establishes that the outcome reported by a
; legal game is in fact the outcome reported by the definition of the
; game.  Note that this is a fact about legal games, not about our
; particular strategy.
```

THEOREM: tic-tac-toep1-outcome
tic-tac-toep1 (*board*, *player*, *lst*)
→ (tic-tac-toep1 (*board*, *player*, *lst*) = nth (length (*lst*) − 1, *lst*))

THEOREM: tic-tac-toep-outcome
tic-tac-toep (*lst*) → (tic-tac-toep (*lst*) = nth (length (*lst*) − 1, *lst*))

```
#|
Historical Plaque:

My original formulation of the problem did not contain tic-tac-toep.
Instead I had a tic-tac-toe function which allegedly played a game
against the user.  The intent was that the buyer read this function
and was convinced that it played the game we call tic-tac-toe.  Then
I proved that it never lost, and the buyer would have been surprised
and pleased.  Here is the definition and the main theorem:

(defn tic-tac-toe (xmoves board)
```

```
  (if (legalp (car xmoves) board)
      (if (winp 'x
                (move 'x (car xmoves) board))
          'x-won
          (if (endp (move 'x (car xmoves) board))
              'draw
              (if (legalp (pick-move (move 'x (car xmoves) board))
                          (move 'x (car xmoves) board))
                  (if (winp 'o
                            (move 'o
                                  (pick-move (move 'x (car xmoves) board))
                                  (move 'x (car xmoves) board)))
                      'o-won
                      (tic-tac-toe (cdr xmoves)
                                   (move 'o
                                         (pick-move (move 'x (car xmoves) board))
                                         (move 'x (car xmoves) board))))
                  'o-illegal)))
      'x-illegal)
  ((lessp (open-squares board))))

(prove-lemma pick-move-never-loses nil
  (member (tic-tac-toe xmoves
                       (list f f f
                             f f f
                             f f f))
          '(x-illegal draw o-won)))

I found this an unsatisfying formalization when I noted to Bill Young
that, as the implementor of this program, he could omit the test
that the user won and the test that the program's move is legal.
That made me realize that tic-tac-toe, as defined above, is equivalent
to the version that makes no such tests.  But when that shorter
version is taken as a spec, it is not credible.  Simple syntactic inspection
reveals the user can't win (that token is never mentioned by the program)
and hence, without complete analysis of pick-move, the buyer can't
come to the opinion that tic-tac-toe is even the game implemented.

|#

; Before we start the implementation, we will introduce the standard
; small set of arithmetic facts.  We'll prove more as needed.
```

THEOREM: plus-add1
$(x + (1 + y)) = (1 + (x + y))$

THEOREM: plus-commutes1
$(x + y) = (y + x)$

THEOREM: plus-commutes2
$(x + y + z) = (y + x + z)$

THEOREM: plus-associates
$((x + y) + z) = (x + y + z)$

THEOREM: times-0
$(x * 0) = 0$

THEOREM: times-non-numberp
$(z \notin \mathbf{N}) \rightarrow ((x * z) = 0)$

THEOREM: times-add1
$(x * (1 + y)) = (x + (x * y))$

THEOREM: times-distributes1
$(x * (y + z)) = ((x * y) + (x * z))$

THEOREM: times-commutes1
$(x * y) = (y * x)$

THEOREM: times-commutes2
$(x * y * z) = (y * x * z)$

THEOREM: times-associates
$((x * y) * z) = (x * y * z)$

THEOREM: times-distributes2
$((x + y) * z) = ((x * z) + (y * z))$

THEOREM: difference-is-0
$(y \not< x) \rightarrow ((x - y) = 0)$

THEOREM: difference-plus-cancellation1
$((i + x) - i) = \mathrm{fix}\,(x)$

THEOREM: difference-plus-cancellation2
$((i + x) - (i + y)) = (x - y)$

THEOREM: difference-plus-cancellation3
$((i + j + x) - j) = (i + x)$

THEOREM: lessp-remainder
$((x \textbf{ mod } y) < y) = (y \not\simeq 0)$

THEOREM: remainder-quotient-elim
$(x \in \mathbf{N}) \rightarrow (((x \textbf{ mod } y) + (y * (x \div y))) = x)$

```
; Our strategy for implementing tic-tac-toe is to transform the
; function in several steps to one that has a very straightforward
; micro-Gypsy implementation.  Each of the steps will produce a
; function with the name tic-tac-toe-x, where x = a, b, c, etc.  Each
; time we define such a function we will prove that it is equivalent
; to the previous such function, starting with tic-tac-toe at the top.
; When the sequence of steps is done, we will combine the results into
; a single equation relating tic-tac-toe to the (final) implementation
; function.

; We first eliminate the need for the endp test by just counting the
; moves.
```

DEFINITION:
tic-tac-toe1-a (*xmoves*, *board*, *i*)
= **let** *board1* **be** move ('x, car (*xmoves*), *board*)
in
if legalp (car (*xmoves*), *board*)
then if *i* < 9
then let *board2* **be** move ('o, pick-move (*board1*), *board1*)
in
if winp ('o, *board2*)
then list (*board1*, *board2*, 'o-win)
else cons (*board1*,
cons (*board2*,
tic-tac-toe1-a (cdr (*xmoves*),
board2,
1 + (1 + *i*)))) **endif endlet**
else list (*board1*, 'draw) **endif**
else list (*board1*, 'o-win) **endif endlet**

DEFINITION:
tic-tac-toe-a (*xmoves*)
= **let** *board0* **be** list (**f**, **f**, **f**, **f**, **f**, **f**, **f**, **f**, **f**)
in
cons (*board0*, tic-tac-toe1-a (*xmoves*, *board0*, 1)) **endlet**

EVENT: Enable tic-tac-toe1.

EVENT: Enable legalp.

EVENT: Enable move.

THEOREM: endp-open-squares
endp (*board*) = (open-squares (*board*) ≃ 0)

THEOREM: open-squares-move-rewrite
(*x* ∧ legalp (*i*, *board*))
→ (open-squares (move (*x*, *i*, *board*)) = (open-squares (*board*) − 1))

THEOREM: add1-difference
(*x* ≮ *y*) → ((1 + (*x* − *y*)) = ((1 + *x*) − *y*))

THEOREM: lessp-open-squares-length
length (*board*) ≮ open-squares (*board*)

```
; The proof of the following lemma is tedious.
```

THEOREM: lemma-a1
(length (*board*) = 9)
→ (tic-tac-toe1 (*xmoves*, *board*)
= tic-tac-toe1-a (*xmoves*, *board*, 10 − open-squares (*board*)))

EVENT: Disable add1-difference.

THEOREM: lemma-a
tic-tac-toe (*xmoves*) = tic-tac-toe-a (*xmoves*)

```
; Next, we will convert the board from a list representation to a
; numeric interpretation.  Suppose that tic-tac-toe-b is defined
; analogously to tic-tac-toe-a but deals with numeric boards.
; Then we will want to prove:

; (equal (tic-tac-toe-a xmoves)
;        (number->board* (tic-tac-toe-b xmoves)))
```

DEFINITION:
piece->number (*x*)
= **if** *x* = **f then** 0
elseif *x* = 'x **then** 1
else 2 **endif**

DEFINITION:
number->piece (j)
= **if** $j = 0$ **then f**
elseif $j = 1$ **then** 'x
else 'o **endif**

DEFINITION:
board->number (*board*)
= **if** *board* $\simeq$ **nil then** 0
else piece->number (car (*board*))
+ (4 $*$ board->number (cdr (*board*))) **endif**

DEFINITION:
number->board (n, *len*)
= **if** *len* $\simeq$ 0 **then nil**
else cons (number->piece (n **mod** 4),
number->board ($n \div 4$, *len* $-$ 1)) **endif**

DEFINITION:
boardp* (*lst*, *len*)
= **if** *lst* $\simeq$ **nil then** *lst* = **nil**
else boardp (car (*lst*))
$\wedge$ (length (car (*lst*)) = *len*)
$\wedge$ boardp* (cdr (*lst*), *len*) **endif**

DEFINITION:
number->board* (*lst*, *len*)
= **if** *lst* $\simeq$ **nil then nil**
else cons (**if** car (*lst*) = -2 **then** 'o-win
elseif car (*lst*) = -1 **then** 'draw
else number->board (car (*lst*), *len*) **endif**,
number->board* (cdr (*lst*), *len*)) **endif**

```
; Some more arithmetic...
```

THEOREM: remainder-plus-times
$((x \in \mathbf{N}) \wedge (x < y)) \rightarrow (((x + (y * z)) \textbf{ mod } y) = x)$

THEOREM: remainder-plus-times-instance
$(y \not\simeq 0) \rightarrow (((y * z) \textbf{ mod } y) = 0)$

THEOREM: quotient-plus-times
$((z \in \mathbf{N}) \wedge (x < y)) \rightarrow (((x + (y * z)) \div y) = z)$

THEOREM: quotient-plus-times-instance
$((z \in \mathbf{N}) \wedge (y \not\simeq 0)) \rightarrow (((y * z) \div y) = z)$

```
; The crucial fact about this representation is:
```

THEOREM: board->number->board
boardp (*board*)
→ (number->board (board->number (*board*), length (*board*)) = *board*)

```
; Now we begin a litany of definitions of the numeric counterparts of
; of the subroutines of tic-tac-toe-a.
```

DEFINITION:
moven (*x*, *i*, *board*)
= **if** *i* ≃ 0 **then** *x* + (4 ∗ (*board* ÷ 4))
else (*board* **mod** 4) + (4 ∗ moven (*x*, *i* − 1, *board* ÷ 4)) **endif**

THEOREM: move-is-moven
(((*x* = 'x) ∨ (*x* = 'o)) ∧ boardp (*board*))
→ (move (*x*, *i*, *board*)
= number->board (moven (piece->number (*x*), *i*, board->number (*board*)),
length (*board*)))

DEFINITION:
nthn (*i*, *board*)
= **if** *i* ≃ 0 **then** *board* **mod** 4
else nthn (*i* − 1, *board* ÷ 4) **endif**

THEOREM: nth-is-nthn
((*i* < length (*board*)) ∧ boardp (*board*))
→ (nth (*i*, *board*) = number->piece (nthn (*i*, board->number (*board*))))

DEFINITION:
legalpn (*i*, *board*) = ((*i* < 9) ∧ (nthn (*i*, *board*) = 0))

THEOREM: legalp-is-legalpn
(boardp (*board*) ∧ (length (*board*) = 9))
→ (legalp (*i*, *board*) = legalpn (*i*, board->number (*board*)))

DEFINITION:
winp1n (*x*, *i*, *j*, *k*, *board*)
= ((*x* = nthn (*i*, *board*))
∧ (*x* = nthn (*j*, *board*))
∧ (*x* = nthn (*k*, *board*)))

THEOREM: lessp-nthn-3-lemma
((*i* < length (*board*)) ∧ boardp (*board*))
→ (piece->number (nth (*i*, *board*)) = nthn (*i*, board->number (*board*)))

THEOREM: lessp-nthn-3
((*i* < length (*board*)) ∧ boardp (*board*))
→ (nthn (*i*, board->number (*board*)) < 3)

THEOREM: winp1-is-winp1n
(boardp (*board*)
∧ (*i* < length (*board*))
∧ (*j* < length (*board*))
∧ (*k* < length (*board*))
∧ ((*x* = 'x) ∨ (*x* = 'o)))
→ (winp1 (*x*, list (*i*, *j*, *k*), *board*)
= winp1n (piece->number (*x*), *i*, *j*, *k*, board->number (*board*)))

DEFINITION:
winpn (*x*, *board*)
= (winp1n (*x*, 0, 1, 2, *board*)
∨ winp1n (*x*, 3, 4, 5, *board*)
∨ winp1n (*x*, 6, 7, 8, *board*)
∨ winp1n (*x*, 0, 3, 6, *board*)
∨ winp1n (*x*, 1, 4, 7, *board*)
∨ winp1n (*x*, 2, 5, 8, *board*)
∨ winp1n (*x*, 0, 4, 8, *board*)
∨ winp1n (*x*, 2, 4, 6, *board*))

THEOREM: winp-is-winpn
((length (*board*) = 9) ∧ boardp (*board*) ∧ ((*x* = 'x) ∨ (*x* = 'o)))
→ (winp (*x*, *board*) = winpn (piece->number (*x*), board->number (*board*)))

THEOREM: quotient-4-decreases
(*n* ≠ 0) → ((*n* ÷ 4) < *n*)

```
; I originally defined pick-move1n as follows:
; (defn pick-move1n (board i)
;   (cond ((zerop board) 0)
;         ((zerop (remainder board 4)) i)
;         (t (pick-move1n (quotient board 4) (add1 i)))))
; And to my great surprise, this function isn't anything
; like pick-move1!  Consider the fact that
; on (list 'x 'o f f f f f f f) pick-move1 returns 2 but
; (on the numeric image of the above board) pick-move1n
; returns 0.  This seems to me a likely bug to have occurred
; in a program attempting to implement this.
```

DEFINITION:

```
pick-move1n (board, i, len)
=   if len ≃ 0 then 0
    elseif (board mod 4) ≃ 0 then i
    else pick-move1n (board ÷ 4, 1 + i, len − 1) endif
```

DEFINITION:

```
board->number-alist (alist)
=   if alist ≃ nil then nil
    else cons (cons (board->number (caar (alist)), cdar (alist)),
                    board->number-alist (cdr (alist))) endif
```

DEFINITION:

```
boardp-alistp (alist)
=   if alist ≃ nil then alist = nil
    else boardp (caar (alist))
         ∧  (length (caar (alist)) = 9)
         ∧  boardp-alistp (cdr (alist)) endif
```

DEFINITION:

```
pick-moven (board)
=   let book-move be assoc (board,
                            '((91648 . 2)
                              (90688 . 2)
                              (92692 . 3)
                              (92689 . 3)
                              (90640 . 5)
                              (90628 . 2)
                              (90625 . 2)
                              (82432 . 6)
                              (103936 . 1)
                              (102976 . 1)
                              (102928 . 1)
                              (102934 . 5)
                              (70144 . 7)
                              (83488 . 6)
                              (104032 . 1)
                              (71200 . 7)
                              (67168 . 6)
                              (67108 . 6)
                              (67105 . 6)
                              (67072 . 2)
                              (90688 . 2)
                              (75328 . 2)
                              (74320 . 5)
                              (74308 . 2)
```

```
(74305 . 2)
(66112 . 6)
(84496 . 3)
(72208 . 3)
(84562 . 6)
(72274 . 7)
(68116 . 3)
(68113 . 3)
(66064 . 5)
(82468 . 6)
(70180 . 7)
(67108 . 6)
(66148 . 6)
(66085 . 6)
(66052 . 2)
(90649 . 5)
(82441 . 6)
(70153 . 7)
(67081 . 7)
(66121 . 7)
(66073 . 7)
(65536 . 4)
(91648 . 2)
(90688 . 2)
(92692 . 3)
(92689 . 3)
(90640 . 5)
(90628 . 2)
(90625 . 2)
(82432 . 6)
(152209 . 5)
(152197 . 5)
(152065 . 3)
(20992 . 8)
(153097 . 3)
(149065 . 6)
(17920 . 8)
(90688 . 2)
(26176 . 2)
(90706 . 5)
(26194 . 8)
(25174 . 8)
(25156 . 2)
(25153 . 2)
```

```
(16960 . 6)
(148057 . 6)
(16912 . 8)
(82468 . 6)
(152164 . 5)
(152101 . 5)
(21028 . 8)
(17956 . 6)
(16996 . 6)
(16933 . 6)
(16900 . 2)
(90625 . 2)
(26113 . 2)
(25153 . 2)
(90649 . 5)
(26137 . 8)
(16897 . 6)
(16384 . 4)
(103936 . 1)
(102976 . 1)
(102928 . 1)
(102934 . 5)
(70144 . 7)
(152209 . 5)
(152197 . 5)
(152065 . 3)
(20992 . 8)
(136705 . 3)
(5632 . 8)
(70210 . 7)
(21058 . 8)
(5698 . 8)
(4690 . 8)
(4678 . 8)
(70168 . 7)
(21016 . 8)
(5656 . 7)
(4696 . 7)
(4633 . 7)
(4624 . 1)
(102934 . 5)
(70150 . 7)
(20998 . 8)
(5638 . 8)
```

```
(4678 . 8)
(4630 . 8)
(70273 . 5)
(21121 . 5)
(71305 . 7)
(22153 . 8)
(4753 . 5)
(4741 . 5)
(4609 . 3)
(4096 . 4)
(83488 . 6)
(104032 . 1)
(71200 . 7)
(67168 . 6)
(67108 . 6)
(67105 . 6)
(67072 . 2)
(153097 . 3)
(149065 . 6)
(17920 . 8)
(136705 . 3)
(5632 . 8)
(67138 . 2)
(17986 . 8)
(5698 . 8)
(1618 . 8)
(1606 . 8)
(136729 . 7)
(132697 . 7)
(1552 . 8)
(67108 . 6)
(17956 . 6)
(71206 . 7)
(22054 . 8)
(5734 . 8)
(1636 . 6)
(1573 . 6)
(1540 . 2)
(67105 . 6)
(17953 . 6)
(71329 . 7)
(22177 . 8)
(5665 . 3)
(1633 . 6)
```

```
(1573 . 6)
(1537 . 2)
(1024 . 4)
(65830 . 7)
(65794 . 2)
(16666 . 6)
(4390 . 7)
(4354 . 2)
(66946 . 6)
(17794 . 6)
(5506 . 2)
(1426 . 6)
(1414 . 6)
(1282 . 3)
(6502 . 8)
(6466 . 2)
(2386 . 6)
(2374 . 7)
(322 . 5)
(74002 . 3)
(24850 . 3)
(9490 . 3)
(8530 . 5)
(274 . 6)
(99718 . 6)
(34054 . 3)
(33094 . 5)
(42262 . 8)
(41302 . 8)
(33046 . 6)
(262 . 7)
(90688 . 2)
(75328 . 2)
(74320 . 5)
(74308 . 2)
(74305 . 2)
(66112 . 6)
(90688 . 2)
(26176 . 2)
(90706 . 5)
(26194 . 8)
(25174 . 8)
(25156 . 2)
(25153 . 2)
```

```
(16960 . 6)
(70210 . 7)
(21058 . 8)
(5698 . 8)
(4690 . 8)
(4678 . 8)
(67138 . 2)
(17986 . 8)
(5698 . 8)
(1618 . 8)
(1606 . 8)
(84562 . 6)
(72274 . 7)
(66130 . 5)
(16978 . 8)
(4690 . 8)
(1618 . 8)
(598 . 8)
(82534 . 6)
(70246 . 7)
(16966 . 8)
(4678 . 8)
(1606 . 8)
(598 . 8)
(74305 . 2)
(25153 . 2)
(9793 . 2)
(74329 . 7)
(577 . 6)
(64 . 4)
(84496 . 3)
(72208 . 3)
(84562 . 6)
(72274 . 7)
(68116 . 3)
(68113 . 3)
(66064 . 5)
(148057 . 6)
(16912 . 8)
(70168 . 7)
(21016 . 8)
(5656 . 7)
(4696 . 7)
(4633 . 7)
```

```
(4624 . 1)
(136729 . 7)
(132697 . 7)
(1552 . 8)
(84562 . 6)
(72274 . 7)
(66130 . 5)
(16978 . 8)
(4690 . 8)
(1618 . 8)
(598 . 8)
(66070 . 5)
(16918 . 8)
(4630 . 8)
(1558 . 8)
(598 . 8)
(66073 . 7)
(18073 . 8)
(4633 . 7)
(1561 . 7)
(601 . 7)
(16 . 4)
(82468 . 6)
(70180 . 7)
(67108 . 6)
(66148 . 6)
(66085 . 6)
(66052 . 2)
(82468 . 6)
(152164 . 5)
(152101 . 5)
(21028 . 8)
(17956 . 6)
(16996 . 6)
(16933 . 6)
(16900 . 2)
(102934 . 5)
(70150 . 7)
(20998 . 8)
(5638 . 8)
(4678 . 8)
(4630 . 8)
(67108 . 6)
(17956 . 6)
```

```
(71206 . 7)
(22054 . 8)
(5734 . 8)
(1636 . 6)
(1573 . 6)
(1540 . 2)
(82534 . 6)
(70246 . 7)
(16966 . 8)
(4678 . 8)
(1606 . 8)
(598 . 8)
(66070 . 5)
(16918 . 8)
(4630 . 8)
(1558 . 8)
(598 . 8)
(66085 . 6)
(16933 . 6)
(1573 . 6)
(613 . 6)
(4 . 4)
(90649 . 5)
(82441 . 6)
(70153 . 7)
(67081 . 7)
(66121 . 7)
(66073 . 7)
(90625 . 2)
(26113 . 2)
(25153 . 2)
(90649 . 5)
(26137 . 8)
(16897 . 6)
(70273 . 5)
(21121 . 5)
(71305 . 7)
(22153 . 8)
(4753 . 5)
(4741 . 5)
(4609 . 3)
(67105 . 6)
(17953 . 6)
(71329 . 7)
```

```
(22177 . 8)
(5665 . 3)
(1633 . 6)
(1573 . 6)
(1537 . 2)
(74305 . 2)
(25153 . 2)
(9793 . 2)
(74329 . 7)
(577 . 6)
(66073 . 7)
(18073 . 8)
(4633 . 7)
(1561 . 7)
(601 . 7)
(66085 . 6)
(16933 . 6)
(1573 . 6)
(613 . 6)
(1 . 4)))
```

in
if *book-move* **then** cdr (*book-move*)
else pick-move1n (*board*, 0, 9) **endif endlet**

EVENT: For efficiency, compile those definitions not yet compiled.

THEOREM: board->number-one-to-one
((length (*board1*) = length (*board2*)) ∧ boardp (*board1*) ∧ boardp (*board2*))
→ ((board->number (*board1*) = board->number (*board2*))
= (*board1* = *board2*))

THEOREM: assoc-board->number
((length (*board*) = 9) ∧ boardp (*board*) ∧ boardp-alistp (*alist*))
→ (assoc (*board*, *alist*)
↔ assoc (board->number (*board*), board->number-alist (*alist*)))

THEOREM: cdr-assoc-board->number
((length (*board*) = 9) ∧ boardp (*board*) ∧ boardp-alistp (*alist*))
→ (cdr (assoc (*board*, *alist*))
= cdr (assoc (board->number (*board*), board->number-alist (*alist*))))

THEOREM: pick-move1-is-pick-move1n
boardp (*board*)
→ (pick-move1 (*board*, *i*)
= pick-move1n (board->number (*board*), *i*, length (*board*)))

THEOREM: pick-move-is-pick-moven
((length (*board*) = 9) ∧ boardp (*board*))
→ (pick-move (*board*) = pick-moven (board->number (*board*)))

EVENT: Disable pick-moven.

DEFINITION:
tic-tac-toe1-b (*xmoves*, *board*, *i*)
= **let** *board1* **be** moven (1, car (*xmoves*), *board*)
in
if legalpn (car (*xmoves*), *board*)
then if *i* < 9
then let *board2* **be** moven (2, pick-moven (*board1*), *board1*)
in
if winpn (2, *board2*)
then list (*board1*, *board2*, -2)
else cons (*board1*,
cons (*board2*,
tic-tac-toe1-b (cdr (*xmoves*),
board2,
1 + (1 + *i*)))) **endif endlet**
else list (*board1*, -1) **endif**
else list (*board1*, -2) **endif endlet**

DEFINITION:
tic-tac-toe-b (*xmoves*) = cons (0, tic-tac-toe1-b (*xmoves*, 0, 1))

THEOREM: legalpn-implies-lessp
(legalpn (*i*, board->number (*board*)) ∧ (length (*board*) = 9))
→ ((*i* < 9) = **t**)

THEOREM: length-number->board
length (number->board (*n*, *len*)) = fix (*len*)

THEOREM: boardp-number-board
boardp (number->board (*n*, *len*))

DEFINITION:
boardpn (*n*)
= **if** *n* ≃ 0 **then t**
elseif (*n* **mod** 4) < 3 **then** boardpn (*n* ÷ 4)
else f endif

THEOREM: boardpn-board->number
boardp (*board*) → boardpn (board->number (*board*))

THEOREM: boardpn-moven
((*board* ∈ **N**) ∧ boardpn (*board*) ∧ ((*x* = 1) ∨ (*x* = 2)))
→ boardpn (moven (*x*, *i*, *board*))

```
; The following function is not the length of the board because the
; high numbered squares might be empty!
```

DEFINITION:
lengthn (*board*)
= **if** *board* ≃ 0 **then** 0
else 1 + lengthn (*board* ÷ 4) **endif**

THEOREM: number->board->number
((*board* ∈ **N**) ∧ boardpn (*board*) ∧ (*len* ≮ lengthn (*board*)))
→ (board->number (number->board (*board*, *len*)) = *board*)

THEOREM: lengthn-board->number
length (*board*) ≮ lengthn (board->number (*board*))

THEOREM: lengthn-moven
((*x* = 1) ∨ (*x* = 2))
→ (lengthn (moven (*x*, *i*, *board*))
= **if** *i* < lengthn (*board*) **then** lengthn (*board*)
else 1 + *i* **endif**)

THEOREM: lengthn-moven-hack-1
((*i* < 9) ∧ boardp (*board*) ∧ (length (*board*) = 9))
→ (9 ≮ lengthn (moven (1, *i*, board->number (*board*))))

THEOREM: lessp-pick-moven
((*i* < 9) ∧ boardp (*board*) ∧ (length (*board*) = 9))
→ (pick-moven (moven (1, *i*, board->number (*board*))) < 9)

THEOREM: lengthn-moven-hack-2
((*i* < 9) ∧ boardp (*board*) ∧ (length (*board*) = 9))
→ (9 ≮ lengthn (moven (2,
pick-moven (moven (1, *i*, board->number (*board*))),
moven (1, *i*, board->number (*board*))))))

```
; The theorem prover does the wrong induction below.  It too chooses
; the one suggested by tic-tac-toe1-a, but it throws away the board
; instantiation because of the competition from board->number.
```

THEOREM: lemma-b1
(boardp (*board*) ∧ (length (*board*) = 9))
→ (tic-tac-toe1-a (*xmoves*, *board*, *i*)
= number->board* (tic-tac-toe1-b (*xmoves*, board->number (*board*), *i*),
9))

THEOREM: lemma-b
tic-tac-toe-a (*xmoves*) = number->board* (tic-tac-toe-b (*xmoves*), 9)

```
; The next transformation will remove the consing of the list of
; boards and replace it by an accumulator treated as an array.
; Because of the explicit test that i is less than 9 -- and the
; knowledge that i starts at 0 and so never gets larger than 9 itself
; -- we know that an array of length 10 will suffice to hold all the
; boards we might generate and thus an array of length 11 will hold all
; the boards plus the final outcome.  We call the array t3.  Thus, in t3(0)
; through t3(9) one will find boards generated.  But we may not fill
; it because the game may end with a win.  We store the outcome in the
; last array element filled.  Thus, to recreate the game, make a list of
; boards form the elements of the array, starting at t3(0) and going through
; the one containing -2 or -1.
```

DEFINITION:
array->lst (*i*, *array*)
= **if** *i* ≮ length (*array*) **then nil**
elseif nth (*i*, *array*) ∈ **N**
then cons (nth (*i*, *array*), array->lst (1 + *i*, *array*))
else list (nth (*i*, *array*)) **endif**

DEFINITION:
put (*val*, *n*, *lst*)
= **if** *n* ≃ 0
then if listp (*lst*) **then** cons (*val*, cdr (*lst*))
else list (*val*) **endif**
else cons (car (*lst*), put (*val*, *n* − 1, cdr (*lst*))) **endif**

THEOREM: nth-put
((*i* ∈ **N**) ∧ (*j* ∈ **N**) ∧ (*j* < length (*array*)))
→ (nth (*i*, put (*val*, *j*, *array*))
= **if** *i* = *j* **then** *val*
else nth (*i*, *array*) **endif**)

DEFINITION:
tic-tac-toe1-c (*xmoves*, *board*, *i*, *t3*)

```
=  let board1 be moven(1, car(xmoves), board)
   in
   if legalpn(car(xmoves), board)
   then if i < 9
        then let board2 be moven(2, pick-moven(board1), board1)
             in
             if winpn(2, board2)
             then put(-2,
                      1 + (1 + i),
                      put(board2, 1 + i, put(board1, i, t3)))
             else tic-tac-toe1-c(cdr(xmoves),
                                 board2,
                                 1 + (1 + i),
                                 put(board2,
                                     1 + i,
                                     put(board1, i, t3))) endif endlet
        else put(-1, 1 + i, put(board1, i, t3)) endif
   else put(-2, 1 + i, put(board1, i, t3)) endif endlet
```

DEFINITION:
tic-tac-toe-c(*xmoves*)
= tic-tac-toe1-c(*xmoves*, 0, 1, list(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0))

THEOREM: length-put
$(i < \text{length}(array)) \rightarrow (\text{length}(\text{put}(val, i, array)) = \text{length}(array))$

```
; The following is just the definition of array->lst and forces us to
; open it.  This would run forever if given the chance.  But there is
; another rule coming...
```

THEOREM: array->lst-opener
$((j < \text{length}(array)) \wedge (i < \text{length}(array)))$
$\rightarrow$ (array->lst(*i*, put(*val*, *j*, *array*))
= **if** nth(*i*, put(*val*, *j*, *array*)) $\in$ **N**
then cons(nth(*i*, put(*val*, *j*, *array*)),
array->lst(1 + *i*, put(*val*, *j*, *array*)))
else list(nth(*i*, put(*val*, *j*, *array*))) **endif**)

THEOREM: array->lst-opener-stopper
$((i \in \mathbf{N}) \wedge (j \in \mathbf{N}) \wedge (j < \text{length}(array)) \wedge (j < i))$
$\rightarrow$ (array->lst(*i*, put(*val*, *j*, *array*)) = array->lst(*i*, *array*))

```
; It surprised me that I had to state that i is always odd below.  I mean,
; it shouldn't have surprised me, but it did.
```

THEOREM: length-tic-tac-toe1-c
((*board* ∈ **N**)
∧ (*i* ∈ **N**)
∧ ((*i* **mod** 2) = 1)
∧ (*i* < 10)
∧ (length (*t3*) = 11))
→ (length (tic-tac-toe1-c (*xmoves*, *board*, *i*, *t3*)) = 11)

THEOREM: nth-tic-tac-toe1-c
((*board* ∈ **N**)
∧ (*i* ∈ **N**)
∧ ((*i* **mod** 2) = 1)
∧ (*i* < 10)
∧ (length (*t3*) = 11)
∧ (*n* ∈ **N**)
∧ (*n* < *i*))
→ (nth (*n*, tic-tac-toe1-c (*xmoves*, *board*, *i*, *t3*)) = nth (*n*, *t3*))

THEOREM: remainder-add1-add1-2
((1 + (1 + *i*)) **mod** 2) = (*i* **mod** 2)

THEOREM: odd-lessp-9-is-lessp-8
(((*i* **mod** 2) = 1) ∧ (*i* < 9)) → ((*i* < 8) = **t**)

THEOREM: lemma-c1
((*board* ∈ **N**)
∧ (*i* ∈ **N**)
∧ ((*i* **mod** 2) = 1)
∧ (*i* < 10)
∧ (length (*t3*) = 11))
→ (tic-tac-toe1-b (*xmoves*, *board*, *i*)
= array->lst (*i*, tic-tac-toe1-c (*xmoves*, *board*, *i*, *t3*)))

THEOREM: lemma-c
tic-tac-toe-b (*xmoves*) = array->lst (0, tic-tac-toe-c (*xmoves*))

```
; I should have thought of this earlier but it escaped me:  I need
; to treat xmoves as an array, not a list.  This means using another
; index, j, to get its elements.
```

DEFINITION:
tic-tac-toe1-d (*xmoves*, *j*, *board*, *i*, *t3*)
= **let** *board1* **be** moven (1, nth (*j*, *xmoves*), *board*)
in

if legalpn (nth (*j*, *xmoves*), *board*)
then if *i* < 0
then let *board2* **be** moven (2, pick-moven (*board1*), *board1*)
in
if winpn (2, *board2*)
then put (-2,
1 + (1 + *i*),
put (*board2*, 1 + *i*, put (*board1*, *i*, *t3*)))
else tic-tac-toe1-d (*xmoves*,
1 + *j*,
board2,
1 + (1 + *i*),
put (*board2*,
1 + *i*,
put (*board1*, *i*, *t3*))) **endif endlet**
else put (-1, 1 + *i*, put (*board1*, *i*, *t3*)) **endif**
else put (-2, 1 + *i*, put (*board1*, *i*, *t3*)) **endif endlet**

DEFINITION:
tic-tac-toe-d (*xmoves*)
= tic-tac-toe1-d (*xmoves*, 0, 0, 1, list (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0))

DEFINITION:
nth-cdr (*j*, *lst*)
= **if** *j* ≃ 0 **then** *lst*
else nth-cdr (*j* − 1, cdr (*lst*)) **endif**

THEOREM: car-nth-cdr
car (nth-cdr (*j*, *lst*)) = nth (*j*, *lst*)

THEOREM: cdr-nth-cdr
cdr (nth-cdr (*j*, *lst*)) = nth-cdr (*j*, cdr (*lst*))

THEOREM: lemma-d1
tic-tac-toe1-c (nth-cdr (*j*, *xmoves*), *board*, *i*, *t3*)
= tic-tac-toe1-d (*xmoves*, *j*, *board*, *i*, *t3*)

THEOREM: lemma-d
tic-tac-toe-c (*xmoves*) = tic-tac-toe-d (*xmoves*)

DEFINITION: up (*t3*) = number->board* (array->lst (0, *t3*), 9)

THEOREM: tic-tac-toe-d-correct
(tic-tac-toep (up (tic-tac-toe-d (*xmoves*))) ∈ '(draw o-win))
∧ x-moves-honored (*xmoves*, up (tic-tac-toe-d (*xmoves*)))
∧ (init-game (length (*opening*), up (tic-tac-toe-d (append (*opening*, *xmoves1*))))
= init-game (length (*opening*),
up (tic-tac-toe-d (append (*opening*, *xmoves2*)))))

Index