CITS3005 Knowledge Representation

Lecture 4: Logic Programming Theory

Tim French

The University of Western Australia

2023

Overview

This lecture will look at the theory of *clausal logic programming*, and then consider some logic programming techniques.
We will:

- Define clausal logic as a syntactic restriction of first order logic.
- Define Herbrand base and Herbrand models as an interpretation of clausal logic.
- Consider proof theoretic properties of clausal logic: the soundness and completeness or the resolution rule.
- Define (pure) logic programs as the restriction of clausal logic to definite clauses.
- Discuss the concept of SLD resolution and proof trees.
- Consider the complexity of proof trees and logic programs, and the effective use of cuts.
- Define accumulators as a means to manage the complexity of programs.
- Present an small game as an example of a complete program.

This material comes from chapters 2 and 3 of *Simply Logical*, by Peter Flach
`https://book.simply-logical.space/src/simply-logical.html`

First Order Logic to Clauses

We have seen the process for transforming FOL to quantifier free clauses. To convert a formula into conjunctive normal form we need to:

1. push all the negations ($\neg$) down to the literals, using de Morgan's laws, double negation and the duality of quantifiers.
2. remove all of the existential quantifiers, by adding in Skolemisation functions.
3. remove all of the universal quantifiers, leaving the variables unbound
4. move all of the conjunctions to the outside of the formula, using the distributivity laws.
5. remove all of the true/false constants.

This allows us to transform a first order formula:

$$\forall X(p(X) \rightarrow \exists Y \forall Z(q(X, Y) \wedge q(Y, Z)))$$

into a clausal representation:

```
q(X, f(X)) :- p(X).
q(f(X), Z) :- p(X).
```

Simple Case: Propositional Clausal Logic

Propositional clausal logic allows clauses of the form:

```
a1; a2 :- b1, b2, b3
```

where `ai`, `bi` are atoms, `;` is disjunction, `:-` is implication and `,` is conjunction.
A program is a set of clauses, for example the program P may be:

```
awake; sleeping :- person.
sleeping; young :- bored.
bored :- old.
:- young, old.
:- awake, sleeping.
awake.
```

Semantics

We have considered interpretations for First order logic, but we can give semantics for logic clauses directly.

Definition

Herbrand Interpretation The *Herbrand base* of a program P is the set of atoms occurring in P.
A *Herbrand Interpretation* for P is a mapping of the Herbrand base to $\{\text{true}, \text{false}\}$.

In the given program the base is
`{awake, sleeping, person, young, old, bored}`, and an interpretation may be $I =$ `{awake, person, young}` (i.e. the atoms that are true).

Definition

An interpretation I is *a model* of a program P if every clause in P is true in I.
A clause `C` is a *consequence* of a program P (written $P \models$ `C`) if `C` is true in every model of P.

Exercise: How many models does P have?

Resolution

To apply inference in clausal logic, we only require one rule, *resolution*:

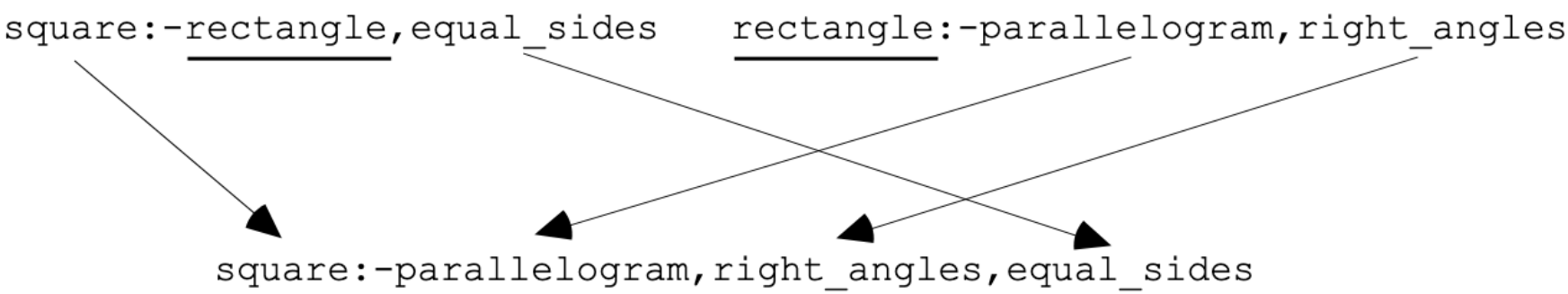
Definition

The *resolution rule* takes two clauses, $C1$ and $C2$, with a common atom a in the head of $C1$ and the tail of $C2$, and produces a new clause whose head is the union of the head of $C1$ and $C2$ with a, and whose tail is the union of the tails of $C1$ and $C2$ with a.

For example, given:

```
h1; h2 :- t1, t2, t3.          h3; t2 :- t4, t5, t6
```

resolution would produce `h1; h2; h3 :- t1, t3, t4, t5, t6.`



Soundness, Completeness and Refutation

Resolution is *sound*

If $C1$ and $C2$ are clauses and $C3$ results from applying resolution to $C1$ and $C2$, then for every program P where $P \models C1$ and $P \models C2$ we have $P \models C3$.

Resolution is *not complete*

Given a program P, resolution cannot derive every clause that is true in every model of P. For example, `h1 :- h1` is true in every model, but the clause cannot always be derived.

However, we can show that resolution is *refutation complete*. That is, if a program has *no models* we can apply resolution to derive the *empty clause* `:-`.

Propositional clausal logic is decidable

Since the Herbrand base of a Program is finite, and resolution is refutation complete, we can apply resolution systematically until either we find a refutation or we find no new clauses.

Full Clausal Logic

Full clausal logic replaces the atoms of propositional clausal logic with predicates that act on *terms* (atoms or functions of terms).

```
teaches(X,course(Z)); supervises(X,Z) :- student_of(X,Z)
likes(peter,S):-student_of(S,peter).
student_of(maria,peter).
```

The disjunctions, conjunctions and implications are the same, but now the truth values are about *things*.
Clauses now have variables (terms beginning with capital letters) which range over all terms.

Full Herbrand Models

Without functions, the Herbrand base is just every predicate applied to every atom, e.g.:

```
{ likes(peter,peter), likes(peter,maria),
likes(maria,peter), likes(maria,maria),
student_of(peter,peter), student_of(peter,maria),
student_of(maria,peter), student_of(maria,maria) }
```

With functions, the Herbrand base becomes every predicate applied *every function* applied to every *term*, and therefore infinite:

```
{ plus(0,0,0), plus(s(0),0,0), ...,
plus(0,s(0),0), plus(s(0),s(0),0), ...,
...,
plus(s(0),s(s(0)),s(s(s(0)))), ... }
```

However, the definitions of Herbrand interpretation and Herbrand model remain. For a Herbrand interpretation to satisfy a clause containing a variable, the interpretation must satisfy *every* clause resulting from a uniform substitution of terms for the variables in the clause.

Resolution and Most General Unifier

To apply resolution to clauses we are required to find a substitution that makes the clauses the same with respect to the Herbrand base (and then propositional resolution can be applied).
There are typically many ways to do this, so we define a *most general unifier* (mgu) which is a substitution that makes the least commitment.
For:
`plus(s(0),X,s(X))` and `plus(s(Y),s(0),s(s(Y)))` the mgu
`{Y -> 0, X -> s(0)}` gives the predicate `plus(s(0),s(0),s(s(0)))`.
Note, some naive attempts to find a unifier can lead to infinite terms so a looping check is required: e.g. `p(X, f(X))` and `p(X, X)`.
Resolution can the be applied:

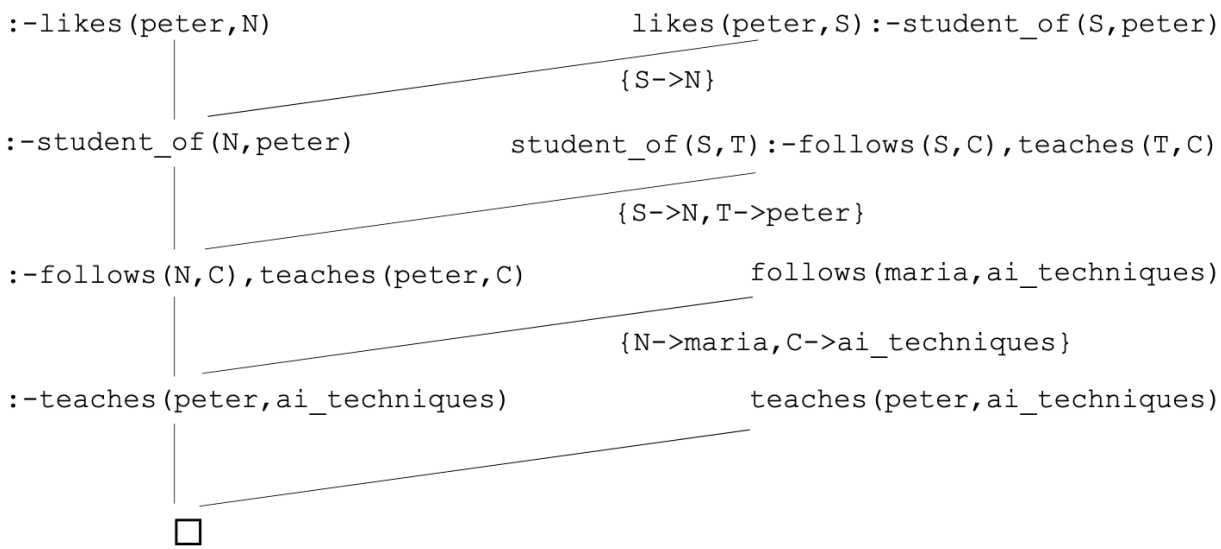
```
likes(peter,S):-student_of(S,peter).
student_of(X,T):-follows(X,C),teaches(T,C).
```

resolves to `likes(peter,S):-follows(S,C),teaches(peter,C).`
through the substitution `{X->S, T->peter}`.

Example

The following proof is a refutation of of `:-likes(peter, N).` given the program:

```
likes(peter,S):-student_of(S,peter).
student_of(S,T):-follows(S,C),teaches(T,C).
teaches(peter,ai_techniques).
follows(maria,ai_techniques).
```



Completeness

Given the common notion of a *Herbrand Interpretation*, the soundness and refutation completeness results are the the same as for propositional clausal logic.
However, since the Herbrand base is now infinite, consistency is now only *semi-decidable*

Resolution is *semi-decidable*

While resolution is refutation complete: any inconsistent program can be shown to have a refutation; it is only *semi-decidable* because there is no definite procedure to show that a program is not consistent.

Definite Clauses

Full clausal logic allows indefinite clauses:
`married(peter);bachelor(peter).`
Definite clauses add the restriction that every clause in a program must have *exactly one positive literal*, so `-:married(X).` or `male(X); female(X).` are not allowed.
The restriction to definite clauses:

- decreases the expressive power of the logic
- does not affect any of the soundness, completeness or decidability results.
- gives a clear systematic way to apply resolution.
- supports the open world assumption: every Herbrand base is a Herbrand model of every program.

Logic Programming as Definite Clause Logic

Pure logic programs are a direct implementation of Definite Clause Logic, and are implemented in *Prolog*

- A program is a set of definite clauses (with predicates, variables and functions).
- A query is a conjunction of literals.
- An execution of a program is an attempted proof of the refutation of the query.
- If the query is refuted, `No.` is reported.
- If a query is not refuted (a substitution is found in some Herbrand Model satisfying the program) the substitution is reported.

Prolog also adds some no logical features, such as *cuts*, *assert* and IO functions with side affects. These are *not* part of definite clause logic. Definite Clause Logic is declarative: it just describes *what is true*. Logic programming is a declarative programming paradigm, but you can consider an *execution* as a *search for a proof* of what is true.

SLD Resolution

Logic programming constrains definite clause logic to use *SLD resolution*:

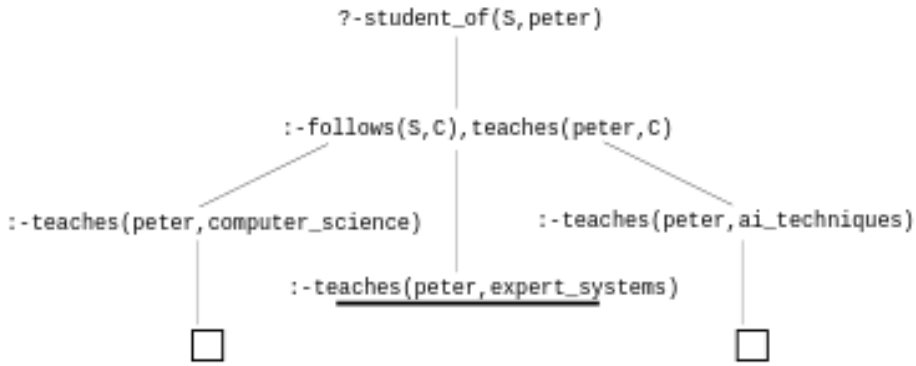
- **S**election rule: Prolog applies a selection rule to find the next clause and literal to resolve: this is typically:
  1. the left most clause in the query string (or resolvent) as the literal
  2. the first clause in the program whose head can unify with that literal
  3. and Prolog inserts the tail of that clause (after substitution) at the front of the resolvent.
- **L**inear resolution: the process always produces linear proofs, and we only need to keep track of a single resolvent.
- **D**efinite clauses: exactly one positive literal per clause.

If nothing matches the selection rule we backtrack to find the next clause from the top of the program that might apply.
These constraints make it possible to design very efficient computation engines for Prolog, and we can visualise program executions as *SLD-trees*.

SLD Trees

Given a program, we can view the execution of that program simply by recording the resolvent at each step of the program.
The trees record the branches that fail (the head of the resolvent may not be able to unify with anything) as well as the linear proof.

```
student_of(X,T):-follows(X,C),teaches(T,C).
follows(paul,computer_science).
follows(paul,expert_systems).
follows(maria,ai_techniques).
teaches(adrian,expert_systems).
teaches(peter,ai_techniques).
teaches(peter,computer_science).
```



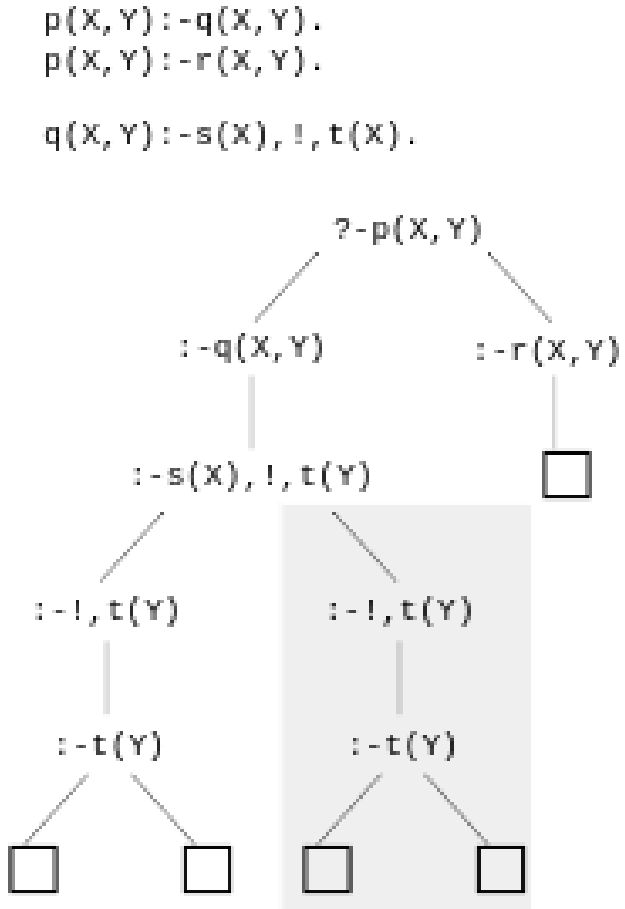
Cuts and SLD Trees

SLD Trees or executions can be infinite, so *cuts* can be using to control the execution flow.

Syntactically a cut is a literal ! that appears in a clause. Once a cut is reached, all *choice points* between the head of the clause and the cut are removed from the top of the stack.

Cuts can removed pointless search space *green cuts*, or can remove successful branches *red cuts*.

Red cuts change the meaning of a program but can be exploited.



Negation as Failure

Red cuts can be useful We can define a program to represent "negation" in Prolog, by using the *cut operation*

```
not(P) :- P, !, fail.
not(P).
```

Here we have a predicate `not` that takes any predicate, `P` as an argument, and if `P` can be satisfied, a cut is applied to prevent backtracking and the clause fails. Otherwise, `not(P)` will be true. This is *not* classical negation, and better described as "failure to prove". For example, what is the result of the query `?- supp(X,Y)`:

```
supp(X,Y) :- not(pass(X,Y)), sat(X,Y).
sat(tim,cits3005).
sat(jane,cits3005).
pass(jane, cits3005).
```

It also breaks the *Open World Assumption*.

SLD Tree Complexity

Cuts can also be used to improve the efficiency of programs:

```
if_then_else(S,T,U):-S,!,T.
if_then_else(S,T,U):-U.
```

vs

```
p:-q,r,s,!,t.
p:-q,r,u.
q.
r.
u.
```

or to free up memory in large search spaces:

```
play(Board,Player):-lost(Board,Player).
play(Board,Player):-
    find_move(Board,Player,Move),
    make_move(Board,Move,NewBoard),!
    next_player(Player,Next),
    play(NewBoard,Next).
```

Head and Tail Recursion

SLD trees can show where memory and computation may be inefficient:

```
naive_length([],0).
naive_length([_H|T],N):-naive_length(T,M),N is M+1.
```

versus

```
length_acc(L,N):-length_acc(L,0,N).

length_acc([],N,N).
length_acc([_H|T],N0,N):-N1 is N0+1,length_acc(T,N1,N).
```

Here an *accumulator* is used to compute the running length as we go, rather than maintaining a large expression.

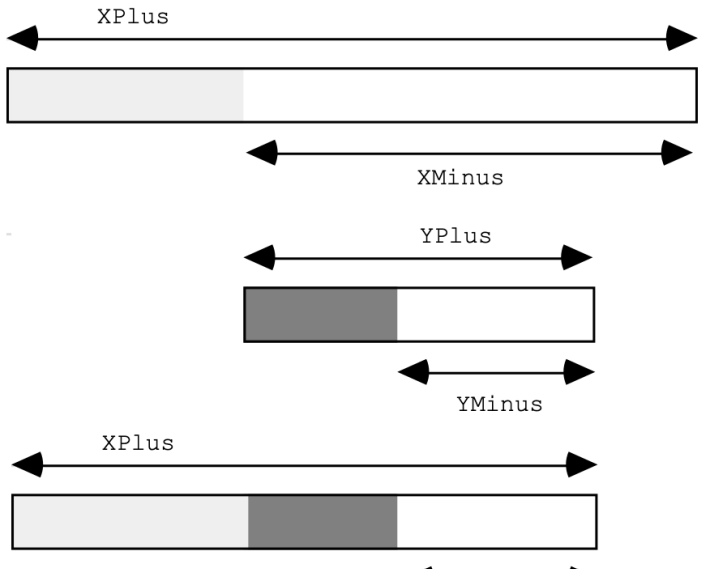
Accumulators

Another example of an accumulator can be seen in the reversal of a list:

```
naive_reverse([],[]).
naive_reverse([H|T],R):-naive_reverse(T,R1),append(R1,[H],R).
append([],Y,Y).
append([H|T],Y,[H|Z]):-append(T,Y,Z).
```

versus

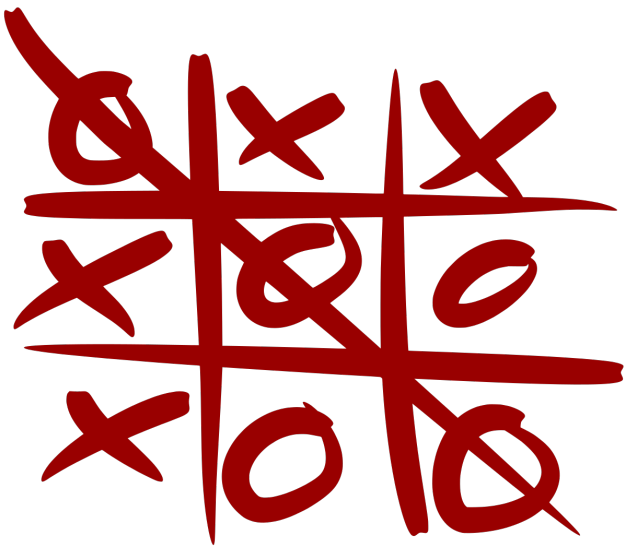
```
reverse(X,Y):- reverse(X,[],Y).
reverse([],Y,Y).
reverse([H|T],Y0,Y):- reverse(T,[H|Y0],Y).
```



Bringing It Together: Tic tac toe

Imagine we would like to design Prolog program for playing tic tac toe. We can break the task down:

- Get valid moves.
- Determine if a move is a winning move.
- Display the board.
- Run the game loop.



The following game was written by S. Tanimoto and is available from `https://swish.swi-prolog.org/p/Tic-Tac-Toe.swinb`

Tic Tac Toe: Get Valid Moves.

```
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]).

% The following translates integers to moves
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, _, Board) :- write('Illegal move.'), nl.
```

Tic tac toe: Check winning position

```
% 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,_,_].
```

Tic tac toe: Run the game loop

```
% 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).

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

```
| ?- selfgame.
[x,b,b]
[b,b,b]
[b,b,b]

[x,o,b]
[b,b,b]
[b,b,b]

[x,o,x]
[b,b,b]
[b,b,b]

[x,o,x]
[o,b,b]
[b,b,b]

[x,o,x]
[o,x,b]
[b,b,b]

[x,o,x]
[o,x,o]
[b,b,b]

[x,o,x]
[o,x,o]
[x,b,b]

[x,o,x]
[o,x,o]
[x,o,b]

[player,x,wins]

yes
```

Tic tac toe: Adding "AI".

```
% 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
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.
```

Tic tac toe: Run the game

```
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.

explain :-
  write('You play X by entering integer position.'),
  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).

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