

PROLOG

- Widely used in Europe and Japan
- Basis of the 5-th Generation project
- FOL inference system
 - Knowledge base represented using Horn clauses.
 - * Program: set of (Horn) clauses.
 - * Input data: queries (i.e., FOL sentences to be proved).
 - * Output: failure or success + variable bindings.
 - Uses input modus ponens for proofs
 - * i.e., depth-first, left-to-right backward chaining.
 - * returns on request all the possible solutions.
 - Uses database semantics instead of the general FOL semantics

SYNTAX

- Atoms: constants (including function names and predicate names). Everything **not** starting with a capital letter or underscore, and everything surrounded by simple quotes is an atom.

- Variables: everything starting with a capital letter or underscore.

- Convenient notation for lists:

$.(a, b) \rightsquigarrow [A|B]$ $.(Joe, .(Jack, .(Jill, []))) \rightsquigarrow [joe, jack, jill]$.

- Clauses: **facts**

```
parent(ann,bob).           parent(cecil,dave).
parent(ann,cecil).         parent(cecil,eric).
```

and **rules** (with all variables universally quantified).

```
ancestor(A,B) :- parent(A,B).
ancestor(A,C) :- ancestor(A,B), ancestor(B,C).
```

- Queries: FOL sentences (with all variables existentially quantified).

DATABASE SEMANTICS

- Suppose Richard has two brothers, John and Geoffrey:

$$\text{Brother}(\text{John}, \text{Richard}) \wedge \text{Brother}(\text{Geoffrey}, \text{Richard})$$

- This assertion is always true in a model where Richard has one brother, so we need to add $\text{John} \neq \text{Geoffrey}$
- The sentence does not rule out that Richard has more than two brothers
- The correct assertion therefore is:

$$\begin{aligned} &\text{Brother}(\text{John}, \text{Richard}) \wedge \text{Brother}(\text{Geoffrey}, \text{Richard}) \wedge \\ &\text{John} \neq \text{Geoffrey} \wedge \\ &\forall x \text{ Brother}(x, \text{Richard}) \Rightarrow (x = \text{John} \vee x = \text{Geoffrey}) \end{aligned}$$

- Things get complex pretty fast, translating knowledge into knowledge bases becomes exceedingly difficult
- We can use a semantics that provides for more straightforward expressions:
database semantics

DATABASE SEMANTICS (CONT'D)

Unique-names assumption: every constant symbol refers to a distinct object

- Breaks down skolemization but makes life easier otherwise

Domain closure: no model contains more domain elements than those named in constant symbols

- Makes model checking feasible (but still very complex)

Closed world assumption: whatever is not known to be true is assumed false

EXAMPLE

FOL:

$\neg \text{Member}(a, [])$
 $\text{Member}(a, \text{.(}a, b))$
 $\text{Member}(a, c) \Rightarrow$
 $\text{Member}(a, \text{.(}b, c))$

Program (file “memb.pl”):

```
memb(A, .(A, B)).  
memb(A, .(B, C)) :- memb(A, C).
```

...or...

```
memb(A, [A|B]).  
memb(A, [B|C]) :- memb(A, C).
```

Queries:

```
?- consult(memb).      %% [memb].  
Warning: ...some singleton variables...  
% memb compiled 0.00 sec, 560 bytes
```

Yes

```
?- memb(X, [1,2,3]).
```

X = 1

Yes

```
?- memb(X, [1,2,3]).
```

X = 1 ;

X = 2 ;

X = 3 ;

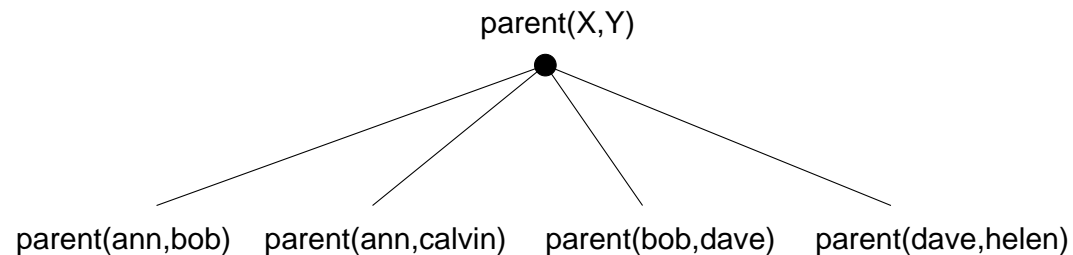
No

```
?-
```

PROOF TREES

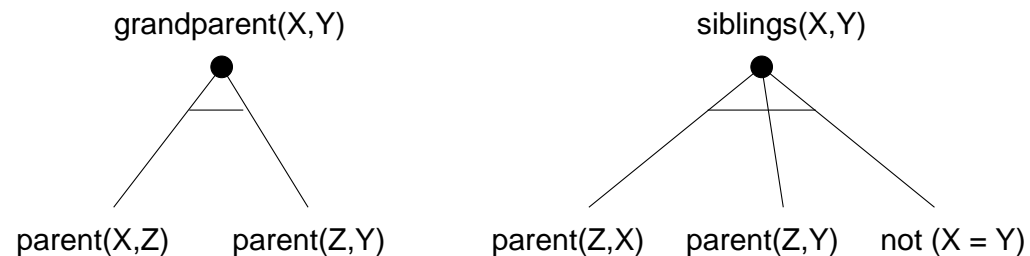
- A family tree:

```
parent(ann,bob).    parent(ann,calvin).  
parent(bob,dave).  parent(dave,helen).
```



- Other family relations:

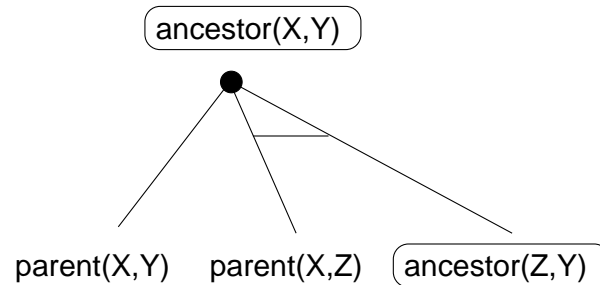
```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).  
siblings(X,Y) :- parent(Z,X), parent(Z,Y), not(X = Y).
```



PROOF TREES (CONT'D)

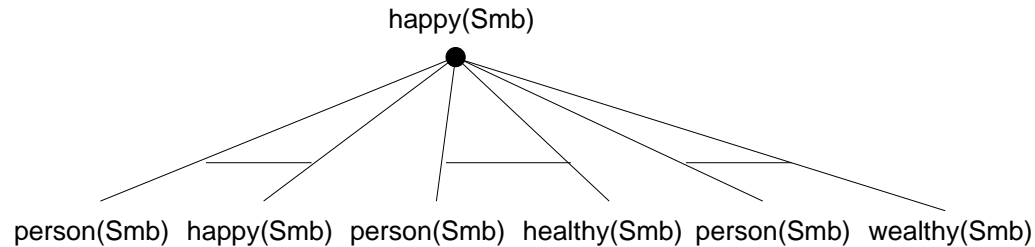
- Yet another family relation:

```
ancestor(X,Y) :- parent(X,Y).  
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```



- A person is happy if she is healthy, wealthy, **or** wise:

```
happy(Smb) :- person(Smb), happy(Smb).  
happy(Smb) :- person(Smb), healthy(Smb).  
happy(Smb) :- person(Smb), wise(Smb).
```



BELLS AND WHISTLES

- Some functions are also defined as infix operators.
 - e.g., $+(1, 2)$ can be written $1+2$.
- There are some functions and predicates with predefined meaning (e.g., usual relational comparison for numbers).
- Built-in predicate `is` for arithmetic
 - ?- `X = 1+2.`

BELLS AND WHISTLES

- Some functions are also defined as infix operators.
 - e.g., $+(1, 2)$ can be written $1+2$.
- There are some functions and predicates with predefined meaning (e.g., usual relational comparison for numbers).
- Built-in predicate `is` for arithmetic

```
?- X = 1+2.
```

```
X = 1+2 ;
```

```
No
```

```
?- X is 1+2.
```

```
X = 3 ;
```

```
No
```

KNIGHTS AND THEIR TOUR

```
% Knight jumping on an n x n board.
% The board size is given by the predicate size/1:
size(5).

% The position of the knight is represented by the function pos(X,Y).
% There are 8 possible moves in the middle of the board:
move(pos(I,J), pos(K,L)) :- K = I+1, L = J-2.
move(pos(I,J), pos(K,L)) :- K = I+1, L = J+2.
move(pos(I,J), pos(K,L)) :- K = I+2, L = J+1.
move(pos(I,J), pos(K,L)) :- K = I+2, L = J-1.
move(pos(I,J), pos(K,L)) :- K = I-1, L = J+2.
move(pos(I,J), pos(K,L)) :- K = I-1, L = J-2.
move(pos(I,J), pos(K,L)) :- K = I-2, L = J+1.
move(pos(I,J), pos(K,L)) :- K = I-2, L = J-1.
```

KNIGHTS AND THEIR TOUR

```
% Knight jumping on an n x n board.
% The board size is given by the predicate size/1:
size(5).

% The position of the knight is represented by the function pos(X,Y).
% There are 8 possible moves in the middle of the board:
move(pos(I,J), pos(K,L)) :- K is I+1, L is J-2.
move(pos(I,J), pos(K,L)) :- K is I+1, L is J+2.
move(pos(I,J), pos(K,L)) :- K is I+2, L is J+1.
move(pos(I,J), pos(K,L)) :- K is I+2, L is J-1.
move(pos(I,J), pos(K,L)) :- K is I-1, L is J+2.
move(pos(I,J), pos(K,L)) :- K is I-1, L is J-2.
move(pos(I,J), pos(K,L)) :- K is I-2, L is J+1.
move(pos(I,J), pos(K,L)) :- K is I-2, L is J-1.

% However, if the knight is somewhere close to board's margins, some
% moves might fall out of the board:
inside(pos(A,B)) :- size(Max), A > 0, A =< Max, B > 0, B =< Max.

search1(A,A,[]).
search1(A,B,[X|Moves]) :- move(A,X), inside(X), search1(X,B,Moves).
```

KNIGHTS AND THEIR TOUR (CONT'D)

```
?- [knights].  
% knights compiled 0.00 sec, 336 bytes
```

Yes

```
?- search1(pos(0,0),pos(1,2),M).
```

```
M = [pos(1, 2)]
```

KNIGHTS AND THEIR TOUR (CONT'D)

```
?- [knights].  
% knights compiled 0.00 sec, 336 bytes
```

Yes

```
?- search1(pos(0,0),pos(1,2),M).
```

```
M = [pos(1, 2)] ;
```

```
ERROR: Out of local stack
```

```
Exception: (16,218) move(pos(5, 2), _G116577) ? abort
```

```
% Execution Aborted
```

```
?-
```

KNIGHTS AND THEIR TOUR (CONT'D)

```
search(A,B,R) :- search_aux(A,B,[A],R).
```

```
search_aux(Z,Z,L,R) :- reverse(L,R).
```

```
search_aux(X,Y,L,R) :- move(X,Z), inside(Z),  
                        not(member(Z,L)),      % ISO: \+ member(Z,L)  
                        search_aux(Z,Y,[Z|L],R).
```

...and then:

```
?- search(pos(0,0),pos(1,2),M).
```

```
M = [pos(0, 0), pos(1, 2)] ;
```

```
M = [pos(0, 0), pos(2, 1), pos(3, 3), pos(4, 1), pos(2, 2) |...] ;
```

```
M = [pos(0, 0), pos(2, 1), pos(3, 3), pos(4, 1), pos(2, 2), |...] ;
```

...and so on (about 127 solutions!).

CSP

Map colouring.

```
% problem instance
border(a,b).      border(a,c).      border(d,e).      border(b,e).
border(a,d).      border(b,c).      border(e,c).      border(d,c).

adj(X,Y) :- border(X,Y).
adj(X,Y) :- border(Y,X).

colour(X) :- member(X,[red,green,blue]).

colour_map([],Colouring,Colouring).
colour_map([Country|Countries], Colouring, R) :-
    colour(X), \+ conflict(Country,X,Colouring),
    colour_map(Countries,[colour(X,Country)|Colouring],R).

% violates constraint?
conflict(Country,X,Colouring) :-
    adj(Country,Country1), % more efficient if we call adj/2 first
    member(colour(X,Country1),Colouring).
```

NEGATION AS FAILURE

- Negation in Prolog: `not/1` or `\+/1`.
- Recall that Prolog assumes the **closed world paradigm**. The negation is therefore different from logical negation:

```
?- member(X,[1,2,3]).
```

```
X = 1 ;
```

```
X = 2 ;
```

```
X = 3 ;
```

```
No
```

```
?- not(member(X,[1,2,3])).
```

```
No
```

```
?- not(not(member(X,[1,2,3]))).
```

```
X = _G332 ;
```

```
No
```

- `not/1` **fails upon resatisfaction** (a goal can fail in only one way).
- `not/1` does **not** bind variables.

NEGATION IN CASE SELECTIONS

```
positive(X) :- X > 0.
negative(X) :- X < 0.

sign(X,+) :- positive(X).
sign(X,-) :- negative(X).
sign(X,0).
```

?- sign(1,X).

NEGATION IN CASE SELECTIONS

```
positive(X) :- X > 0.  
negative(X) :- X < 0.
```

```
sign(X,+) :- positive(X).  
sign(X,-) :- negative(X).  
sign(X,0).
```

```
?- sign(1,X).
```

```
X = + ;
```

```
X = 0 ;
```

```
No
```

NEGATION IN CASE SELECTIONS

```
positive(X) :- X > 0.  
negative(X) :- X < 0.
```

```
sign(X,+) :- positive(X).  
sign(X,-) :- negative(X).  
sign(X,0).
```

```
?- sign(1,X).
```

```
X = + ;
```

```
X = 0 ;
```

```
No
```

```
sign1(X,+) :- positive(X).  
sign1(X,-) :- negative(X).  
sign1(X,0) :- not(positive(X)),  
              not(negative(X)).
```

```
?- sign1(1,X).
```

```
X = + ;
```

```
No
```

MODIFYING THE SEARCH SPACE

The `!/0` predicate (pronounced “cut”) does not allow backtracking over it. All attempts to redo goals to the left of the cut fail.

Commit:

```
negative(X) :- X < 0.  
  
sign(X,-) :- negative(X).  
sign(X,+).  
  
even(X) :- R is X rem 2, R = 0.  
  
positive_even1(X) :- sign(X,+), even(X).  
positive_even(X) :- sign(X,+), !, even(X).  
  
?- positive_even1(-4).  
  
Yes  
?- positive_even(-4).
```

No

Succeed once:

```
member(X,[X,_]).  
member(X,[_,Y]) :- member(X,Y).  
  
membchk(X,[X,_]) :- !.  
membchk(X,[_,Y]) :- membchk(X,Y).
```

MODIFYING THE SEARCH SPACE (CONT'D)

- Succeed once (cont'd):

```
fact1(1,1).  
fact1(N,R) :- N1 is N-1,  
              fact1(N1,R1),  
              R is N*R1.
```

```
fact2(1,1).  
fact2(N,R) :- N>1, N1 is N-1,  
              fact2(N1,R1), R is N*R1.
```

```
fact3(1,1) :- !.  
fact3(N,R) :- N1 is N-1,  
              fact3(N1,R1), R is N*R1.
```

- Fail goal now

- An apparently useless predicate: `fail/0` always fails.

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

```
not(P).
```

MODIFYING THE SEARCH SPACE (CONT'D)

- Succeed once (cont'd):

```
fact1(1,1).
fact1(N,R) :- N1 is N-1,
              fact1(N1,R1),
              R is N*R1.
```

```
fact2(1,1).
fact2(N,R) :- N>1, N1 is N-1,
              fact2(N1,R1), R is N*R1.
```

```
fact3(1,1) :- !.
fact3(N,R) :- N1 is N-1,
              fact3(N1,R1), R is N*R1.
```

- Fail goal now

- An apparently useless predicate: `fail/0` always fails.

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

- Another useful predicate: `call/1`.

* `call(P)` behaves as if `P` were passed as a goal to the interpreter.

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

AN ADVENTURE GAME

- Consider the following knowledge base:

```
location(egg,duck_pen).
location(ducks,duck_pen).
location(fox,woods).
location(you,house).

connect(yard,house).
connect(yard,woods).

is_closed(gate).
connect(duck_pen,yard) :- is_open(gate).
```

- We want to move around, be able to open and close the gate, pick the egg, and so on.
- In order to do this we need to modify the knowledge base dynamically.

MODIFYING THE KNOWLEDGE BASE

- Adding a fact to the knowledge base:
 - `assert` adds the fact given as argument **somewhere**
 - `asserta` adds the fact given as argument **at the beginning** of the knowledge base
 - `assertz` adds the fact given as argument **at the end**
 - all variants succeed only once
- Removing a fact from the knowledge base:
 - `retract` removes one instance that unifies with the argument
 - removes one more instance at each redo attempt
 - fails when no removal is possible

CONVINCING THE KNOWLEDGE BASE TO ALLOW CHANGES

- All the industrial grade PROLOG implementations compile the knowledge base as to increase the speed of retrieving facts from it.
 - SWI PROLOG is one such an example
- In these variants, you need to specify which facts are changeable at run time.
 - These predicates will be stored separately, in an un-optimized fashion
 - The `dynamic` declaration must precede the predicate definition

```
:- dynamic(you_have/1),  
   dynamic(location/2),  
   dynamic(is_closed/1),  
   dynamic(is_open/1).
```

ADVENTURE GAME (CONT'D)

- Moving around:

```
goto(X) :-
    location(you,L),
    (connect(L,X); connect(X,L)),
    retract(location(you,L)),
    assert(location(you,X)),
    write(' You are in the '),
    write(X), nl.
goto(X) :- write(' You cannot get there '), nl.
```

- Picking up the egg:

```
pick(egg) :-
    location(you,duck_pen),
    not you_have(egg),
    assert(you_have(egg)),
    write(' You picked the egg '), nl.
pick(X) :- write(' There is nothing to pick '), nl.
```

ADVENTURE GAME (CONT'D)

- Opening the gate:

```
open(gate) :-
    location(you,yard),
    is_closed(gate),
    retract(is_closed(gate)),
    assert(is_open(gate)),
    write(' Opened. '), nl.
open(X) :- write(' You cannot open that '), nl.
```

- How the other creatures react:

```
ducks :-
    is_opened(gate),
    retract(location(ducks,duck_pen)),
    assert(location(ducks,yard)).
ducks.

fox :-
    location(ducks,yard),
    location(you,house),
    write(' The fox has taken a duck '), nl.
fox.
```

ADVENTURE GAME (CONT'D)

- The main loop:

```
go :- done.
go :-
    write('>>'),
    read(X),
    call(X),
    go.

done :-
    location(you,house),
    you_have(egg),
    ducks, fox,
    write(' Thanks for getting the egg. '), nl.
```

SAMPLE INTERACTION

```
?- go.  
>>goto(yard).  
  You are in the yard  
>>goto(duck_pen).  
  You cannot get there from here  
>>pick(egg).  
  There is nothing to pick  
>>open(gate).  
  Opened.  
>>goto(duck_pen).  
  You are in the duck_pen  
>>pick(egg).  
  You picked the egg  
>>goto(house).  
  You cannot get there from here  
>>goto(yard).  
  You are in the yard  
>>goto(house).  
  You are in the house  
  The fox has taken a duck  
  Thanks for getting the egg.  
yes
```