PROLOG OVERVIEW

Example of logic programming – deductive reasoning

– All Birds fly, a robin is a bird, therefore …….

Symbolic non-numeric programming suited to problems involving objects and relations between objects

Goal in AI programming is to program close to the way you think, and to be self-documenting - so that the program helps clarify your thinking and vice-versa

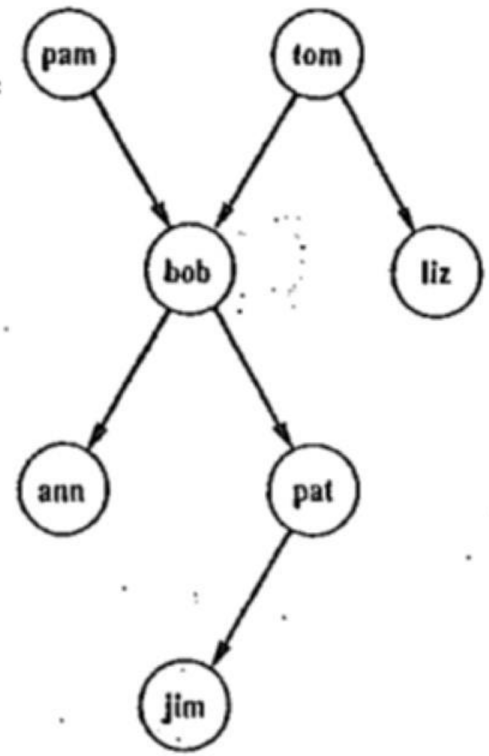
Uses predicates (true/false) to define relationship

e.g. parent(tom,bob). - the meaning has to be defined by the programmer – could mean tom is parent of bob or vice-versa

QUICK GUIDE TO PROLOG

Example Program

```
parent ( pam, bob ).
parent ( tom, bob ).
parent ( tom, liz ).
parent ( bob, ann  ).
parent ( bob, pat ).
parent ( pat, jim ).
```



All six facts (also called clauses) are instances of the parent relation

parent (tom, bob). **'tom is the parent of bob' where *parent* is the name of the relation between two objects, *tom* and *bob* which are its arguments. The full-stop at the end of the clause is compulsory to signify end -of-clause.**

Two Modes in Prolog, Assertion and Query(?-)

```
parent ( pam, bob ).
parent ( tom, bob ).
parent ( tom, liz ).
parent ( bob, ann ).
parent ( bob, pat ).
parent ( pat, jim ).
```

?- parent (bob, pat).   Is Bob Pat's parent?
    YES

?- parent (liz, pat).
    NO   because not in Prolog's database

?- parent (X, liz).   X is single but unspecified individual

- Who is liz's parent?

    X = tom

Assertion mode adds to database (but normally put your program in a file e.g. use consult predicate)

In query mode, if you mis-spell, Prolog returns NO

```
?- parent ( bob, X ).
      X = ann
      X = pat
      NO
```

Any argument beginning with capital is variable, otherwise constant

if Prolog forced to find an alternate solution

if Prolog forced to find an alternate solution

```
?- parent ( X, Y ).
      X = pam
      Y = bob

      X = tom
      Y = bob
```

if Prolog forced to find an alternate solution

```
?- parent ( Y, jim ), parent ( X, Y ).
      X = bob
      Y = pat
```

Who is jim's grandparent?

note: scope of clause up to full-stop so Y is shared

```
?- parent ( X, ann ), parent ( X, pat ).
      X = bob
```

comma indicates logical and

X is shared, a common parent

```
parent ( pam, bob ).
parent ( tom, bob ).
parent ( tom, liz ).
parent ( bob, ann ).
parent ( bob, pat ).
parent ( pat, jim ).
```

Prolog Rules

```
parent ( pam, bob ).
parent ( tom, bob ).
parent ( tom, liz ).
parent ( bob, ann ).
parent ( bob, pat ).
parent ( pat, jim ).
```

Consider adding offspring relation - offspring(liz, tom).

- Logically, this can be related to the parent relation using a rule:
  - For All X and Y (universal quantification in predicate logic)
  - Y is an offspring of X if X is a parent of Y
- In Prolog: offspring(Y, X) :- parent(X, Y).
  - This is also a Prolog clause. A fact is a unit clause. The conclusion part (head of the clause) is true on the condition that the conditon part (body of the clause) is true.

?-offspring(liz, tom).
YES

Using the offspring rule, the goal becomes subgoal:
parent(tom, liz). with X = tom, Y = liz

Recursion

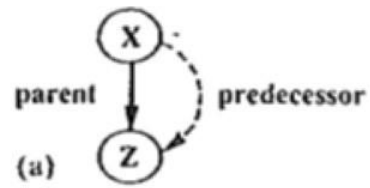
Example: Predecessor Relation requires two clauses

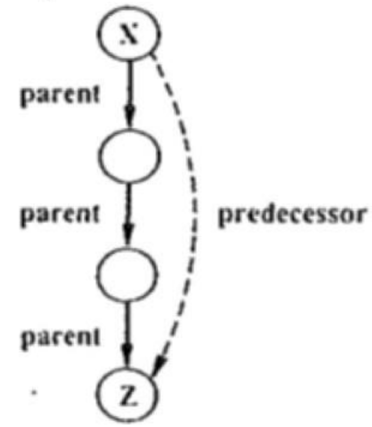
For All X and Z
X is a predecessor of Z
if X is a parent of Z

```
predecessor( X, Z) :-
        parent( X, Z).
```

For All X and Z
X is a predecessor of Z if
there exists a Y such that
- a) **X is a parent of Y and**
- b) **Y is a predecessor of Z**

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





- A set of clauses (two in this case), with the same predicate as head, is referred to as a procedure. Since *predecessor* calls itself it is a recursive procedure.

Answering Queries

Prolog answers questions by trying to satisfy goals i.e. tries to find an appropriate instantiation of variables. Goals are satisfied left to right within a clause and top to bottom within the program.

The predecessor procedure operates as follows:

```
predecessor( X, Z) :-
                parent( X, Z).

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

```
parent ( pam, bob ).
parent ( tom, bob ).
parent ( tom, liz ).
parent ( bob, ann ).
parent ( bob, pat ).
parent ( pat, jim ).
```

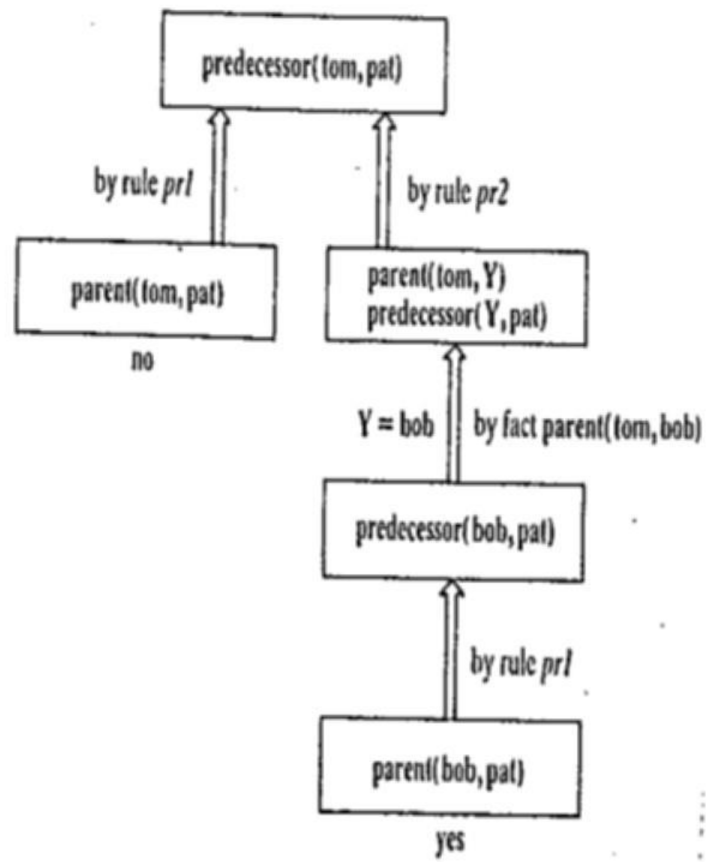
- ?- predecesor(tom, pat). uses the first rule to infer parent(tom, pat) which becomes the subgoal to be satisfied.
- Since the subgoal fails because no such fact exists in the program, Prolog 'backtracks' and tries an alternative, in this case the second rule. This results in two subgoals : parent(tom, Y), predecessor(Y, pat).
- The first subgoal results in *Y = bob* from the parent fact in the program, which leads to an attempt to satisfy predecessor(bob, pat).
- The first rule is matched and succeeds because the resultant subgoal *parent(bob, pat)* appears in the program. If however this first rule failed the recursion would be repeated until all possible instantiations had been tried.

Family Relation Program and trace

```
parent( pam, bob).                  % Pam is a parent of Bob
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).

female( pam).                       % Pam is female
male( tom).                         % Tom is male
male( bob).
female( liz).
female( ann).
female( pat).
male( jim).

offspring( Y, X) :-                 % Y is an offspring of X if
   parent( X, Y).                   % X is a parent of Y

mother( X, Y) :-                    % X is the mother of Y if
   parent( X, Y),                   % X is a parent of Y and
   female( X).                      % X is female

grandparent( X, Z) :-               % X is a grandparent of Z if
   parent( X, Y),                   % X is a parent of Y and
   parent( Y, Z).                   % Y is a parent of Z

sister( X, Y) :-                    % X is a sister of Y if
   parent( Z, X),
   parent( Z, Y),                   % X and Y have the same parent and
   female( X),                      % X is female and
   different( X, Y).                % X and Y are different

predecessor( X, Z) :-               % Rule pr1: X is a predecessor of Z
   parent( X, Z).

predecessor( X, Z) :-               % Rule pr2: X is a predecessor of Z
   parent( X, Y),
   predecessor( Y, Z).
```

Built-in operator

trace of
predecessor(tom,pat).



```
parent ( pam, bob ).
parent ( tom, bob ).
parent ( tom, liz ).
parent ( bob, ann ).
parent ( bob, pat ).
parent ( pat, jim ).

female(ann).
```

sister relation:

```
sister( X, Y) :-
parent( Z, X),
parent( Z, Y),
female( X),           assumes female and male unary relations
different( X, Y).     defined as true if X and Y not equal
```

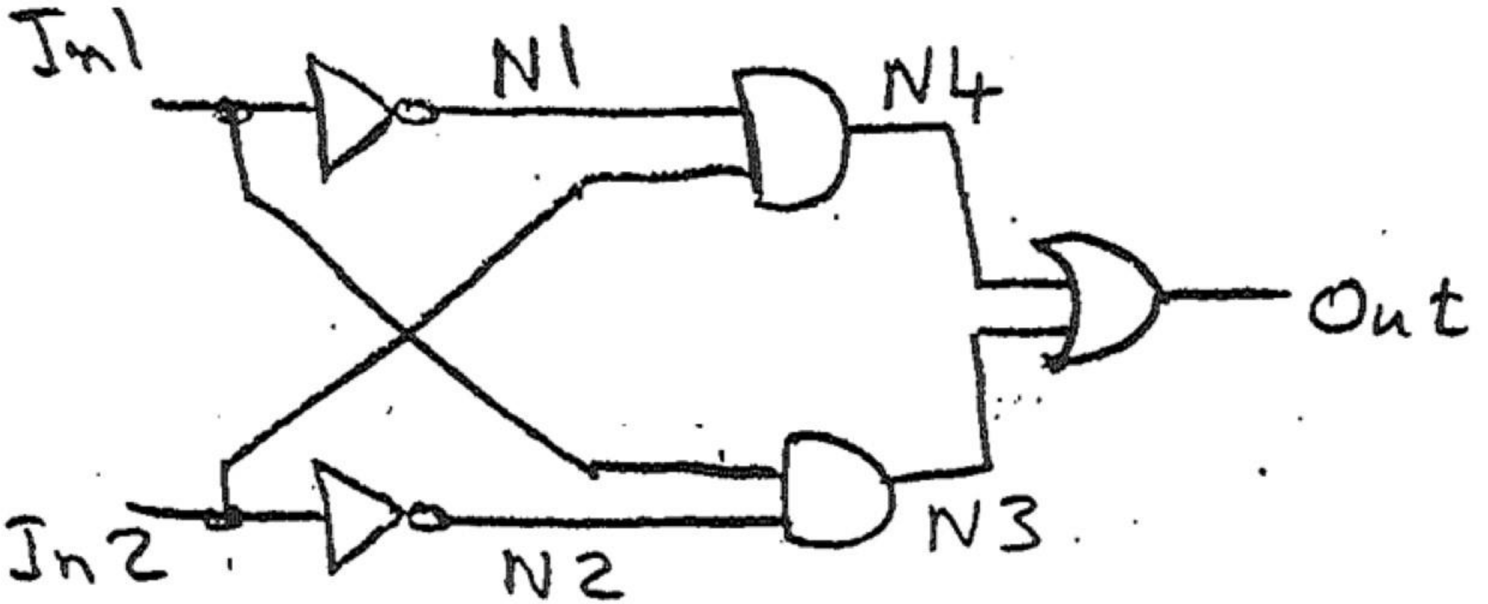
BUILT-IN OPERATOR $X \neq Y$

```
?- sister( X, pat).
X = ann               X = pat is prevented by last goal (different) in sister.
```

Declarative vs Procedural Meaning

- The declarative meaning is given by the relations defined in the program, for example the two predecessor rules.
- The procedural meaning is concerned with how Prolog determines the response to a query. In principle, but not in practice, the programmer need only be concerned with the declarative meaning.
- 'The Art of Prolog programming is concerned with getting this balance right for a particular problem i.e. making the program as declarative as possible within efficiency limits.
- A major difficulty is that some Prolog constructs are specifically designed to help with procedural aspects and if used improperly, the resulting program is more opaque than its conventional procedural programming language counterparts.

Example circuit design



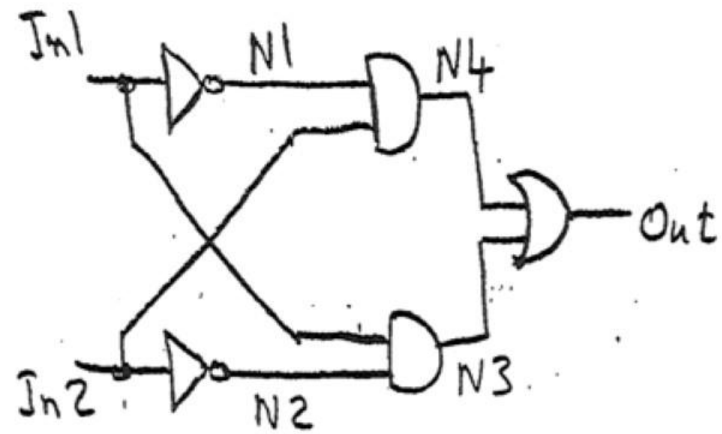
1) and(0,0,0).
2) and(0,1,0).
3) and(1,0,0).
4) and(1,1,1).

5) or(0,0,0).
6) or(0,1,1).
7) or(1,0,1).
8) or(1,1,1).

9) inv(1,0).
10) inv(0,1).

Answering Queries using Backtracking

```
xor(In1,In2,Out) :-   inv(In1,N1),
                      inv(In2,N2),
                      and(In1,N2,N3),
                      and(In2,N1,N4),
                      or(N3,N4,Out).
```




```
? xor(1,1,Out).
      Out=0
```

Forward simulation

```
? xor(A,B,1).
      A=1,B=0.
      A=0,B=1.
```

Backward simulation

```
? xor(A,B,Out).
```

generates truth table on backtracking.

```
xor(In1,In2,Out) =   inv(In1,N1),
                     inv(In2,N2),
                     and(In1,N2,N3),
                     and(In2,N1,N4),
                     or(N3,N4,Out).
```

```
? xor(In1,In2,1)
```

1) and (0,0,0).
2) and (0,1,0).
3) and (1,0,0).
4) and (1,1,1).
5) or (0,0,0).
6) or (0,1,1).
7) or (1,0,1).
8) or (1,1,1).
9) inv (1,0).
10) inv (0,1).

c9 inv(1,0), ... In1=1, N1=0

c9 " , inv(1,0), .. In2=1, N2=0

c3 " " , and(1,0,0), ... N3=0

c3 " " " , and(1,0,0) N4=0

Backtrack on failing or(0,0,1)

* REDO c9: c10 inv(0,1) In2=0, N2=1

c4 inv(1,0), inv(0,1), and(1,1,1) .... N3=1

c1 " " " , and(0,0,0)... N4=0

c7 YES

In1 = 1, In2 = 0