**Prolog**

* **Prolog or PROgramming in LOGics is a logical and declarative programming language.**
* Programs that involve **symbolic** or **non-numeric computation**.
* **Logic Programming is one of the Computer Programming Paradigm, in which the program statements express the facts and rules about different problems within a system of formal logic.**

**Prolog language basically has three different elements**

**Facts**

**Syntax**

**The syntax for facts is as follows − relation(object1,object2...).**

**We can define fact as an explicit relationship between objects, and properties these objects might have. So facts are unconditionally true in nature. Suppose we have some facts as given below.**

* **Tom is a cat**
* **Kunal loves to eat Idly**
* **Hair is black**
* **Nawaz loves to play games**
* **Pratyusha is lazy.**

**Following are some guidelines to write facts −**

* **Names of properties/relationships begin with lower case letters.**
* **The relationship name appears as the first term.**
* **Objects appear as comma-separated arguments within parentheses.**
* **A period "." must end a fact.**
* **Objects also begin with lower case letters. They also can begin with digits (like 1234), and can be strings of characters enclosed in quotes e.g. color(penink, ‘red’).**
* **phoneno(agnibha, 1122334455). is also called a predicate or clause.**

**Rules**

**Syntax**

rule\_name(object1, object2, ...) :-fact/rule(object1, object2, ...)

We can define rule as an implicit relationship between objects. So facts are conditionally true. So when one associated condition is true, then the predicate is also true. Suppose we have some rules as given below −

* Lili is happy if she dances.
* Tom is hungry if he is searching for food.
* Jack and Bili are friends if both of them love to play cricket.
* will go to play if school is closed, and he is free.

**So these are some rules that are conditionally true, so when the right hand side is true, then the left hand side is also true.**

**Note : Here the symbol ( :- ) will be pronounced as “If”, or “is implied by”. This is also known as neck symbol, the LHS of this symbol is called the Head, and right hand side is called Body.**

**Here we can use comma (,) which is known as conjunction, and we can also use semicolon, that is known as disjunction.**

**Questions −** **Queries are some questions on the relationships between objects and object properties. So question can be anything, as given below −**

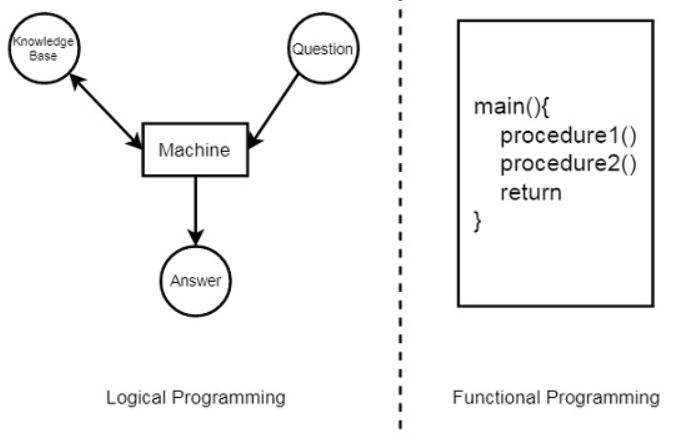
* **Is tom a cat?**
* **Does Kunal love to eat pasta?**
* **Is Lili happy?**
* **Will Ryan go to play?**

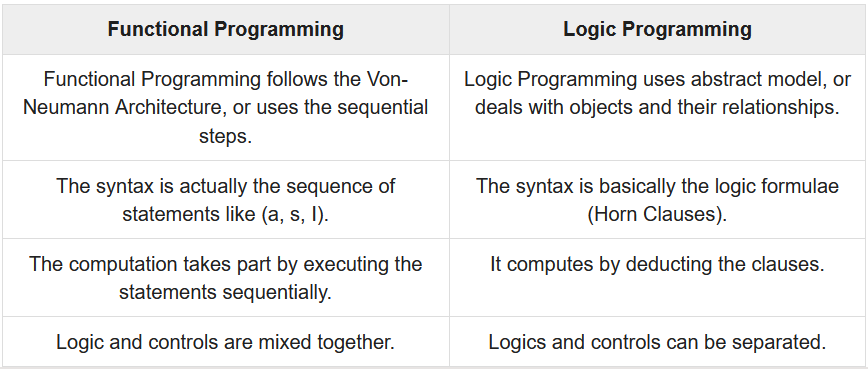
**So according to these queries, Logic programming language can find the answer and return them.**

**Knowledge Base in Logic Programming**

**Knowledge Base**. So we can say that the **knowledge base** is a **collection of facts and rules**.

Now, we will see how to write some knowledge bases. Suppose we have our very first knowledge base called KB1. Here in the KB1, we have some facts. The facts are used to state things, that are unconditionally true of the domain of interest.





**Knowledge Base 1**

**Suppose we have some knowledge, that**

* **Priya, Tiyasha, and Jaya are three girls,**
* **among them, Priya can cook.**

**Let’s try to write these facts in a more generic way as shown below −**

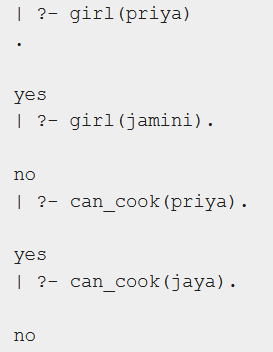
**girl(priya).**

**girl(tiyasha).**

**girl(jaya).**

**can\_cook(priya).**

**Quries**



**Knowledge Base 2**

**So there are some facts and rules given above. The first two are facts, but the rest are rules. As we know that Ananya sings a song, this implies she also listens to music.**

**sing\_a\_song(ananya).**

**listens\_to\_music(rohit).**

**listens\_to\_music(ananya) :- sing\_a\_song(ananya).**

**happy(ananya) :- sing\_a\_song(ananya).**

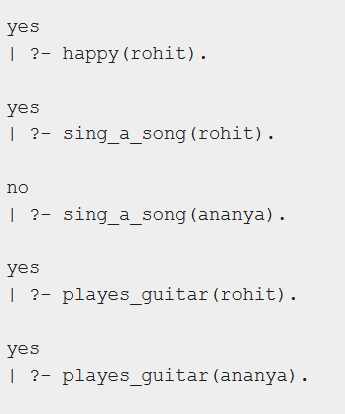
**happy(rohit) :- listens\_to\_music(rohit).**

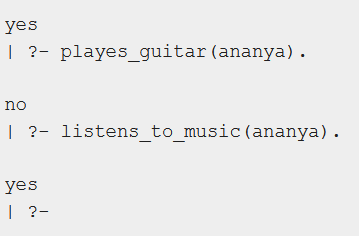
**playes\_guitar(rohit) :- listens\_to\_music(rohit).**

**So if we ask “Does Ananya listen to music?”, the answer will be true.**

**Similarly, “is Rohit happy?”, this will also be true because he listens to music.**

**But if our question is “does Ananya play guitar?”, then according to the knowledge base, it will say “No”.**





**Knowledge Base 3**

**The facts and rules of Knowledge Base 3 are as follows −**

**can\_cook(priya).**

**can\_cook(jaya).**

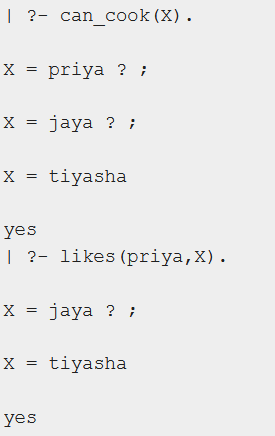
**can\_cook(tiyasha).**

**likes(priya,jaya) :- can\_cook(jaya).**

**likes(priya,tiyasha) :- can\_cook(tiyasha).**

**Suppose we want to see the members who can cook, we can use one variable in our query. The variables should start with uppercase letters. In the result, it will show one by one. If we press enter, then it will come out, otherwise if we press semicolon (;), then it will show the next result.**

**Let us see one practical demonstration output to understand how it works.**



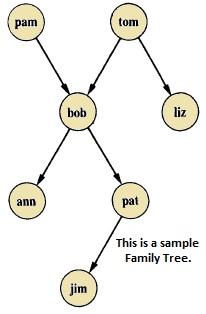
**Relations in Prolog**

**In Prolog programs, it specifies relationship between objects and properties of the objects.**

**Family Relationship in Prolog**

Here we will see the family relationship. This is an example of complex relationship that can be formed using Prolog. We want to make a family tree, and that will be mapped into facts and rules, then we can run some queries on them.

Suppose the family tree is as follows −



Here from this tree, we can understand that there are few relationships. Here bob is a child of pam and tom, and bob also has two children — ann and pat. Bob has one brother liz, whose parent is also tom. So we want to make predicates as follows −

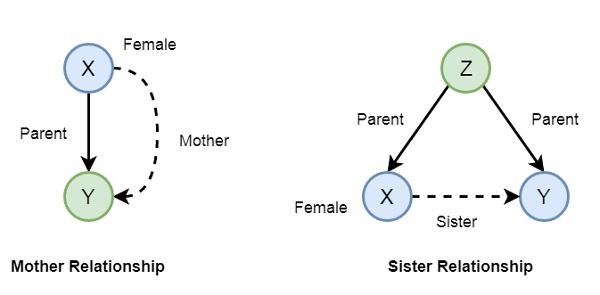
### Predicates

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

Some facts can be written in two different ways, like sex of family members can be written in either of the forms −

* female(pam).
* male(tom).
* male(bob).
* female(liz).
* female(pat).
* female(ann).
* male(jim).

Now if we want to make mother and sister relationship, then we can write as given below −



In Prolog syntax, we can write −

* mother(X,Y) :- parent(X,Y), female(X).
* sister(X,Y) :- parent(Z,X), parent(Z,Y), female(X), X \== Y.

Now let us see the practical demonstration −

**Knowledge Base (family.pl)**

female(pam).

female(liz).

female(pat).

female(ann).

male(jim).

male(bob).

male(tom).

male(peter).

parent(pam,bob).

parent(tom,bob).

parent(tom,liz).

parent(bob,ann).

parent(bob,pat).

parent(pat,jim).

parent(bob,peter).

parent(peter,jim).

mother(X,Y):- parent(X,Y),female(X).

father(X,Y):- parent(X,Y),male(X).

haschild(X):- parent(X,\_).

sister(X,Y):- parent(Z,X),parent(Z,Y),female(X),X\==Y.

brother(X,Y):-parent(Z,X),parent(Z,Y),male(X),X\==Y.

**Output**

| ?- [family].

compiling D:/TP Prolog/Sample\_Codes/family.pl for byte code...

D:/TP Prolog/Sample\_Codes/family.pl compiled, 23 lines read - 3088 bytes written, 9 ms

yes

| ?- parent(X,jim).

X = pat ? ;

X = peter

yes

| ?-

mother(X,Y).

X = pam

Y = bob ? ;

X = pat

Y = jim ? ;

no

| ?- haschild(X).

X = pam ? ;

X = tom ? ;

X = tom ? ;

X = bob ? ;

X = bob ? ;

X = pat ? ;

X = bob ? ;

X = peter

yes

| ?- sister(X,Y).

X = liz

Y = bob ? ;

X = ann

Y = pat ? ;

X = ann

Y = peter ? ;

X = pat

Y = ann ? ;

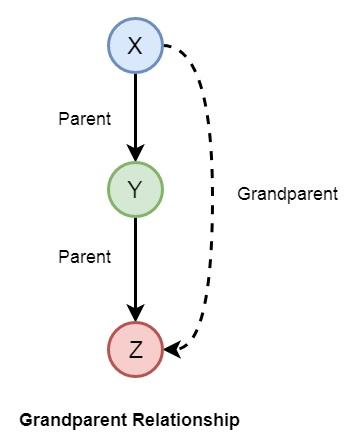
X = pat

Y = peter ? ;

(16 ms) no

| ?-

Now let us see some more relationships that we can make from the previous relationships of a family. So if we want to make a grandparent relationship, that can be formed as follows −



We can also create some other relationships like wife, uncle, etc. We can write the relationships as given below −

* grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
* grandmother(X,Z) :- mother(X,Y), parent(Y,Z).
* grandfather(X,Z) :- father(X,Y), parent(Y,Z).
* wife(X,Y) :- parent(X,Z),parent(Y,Z), female(X),male(Y).
* uncle(X,Z) :- brother(X,Y), parent(Y,Z).

So let us write a prolog program to see this in action. Here we will also see the trace to trace-out the execution.

**Knowledge Base (family\_ext.pl)**

female(pam).

female(liz).

female(pat).

female(ann).

male(jim).

male(bob).

male(tom).

male(peter).

parent(pam,bob).

parent(tom,bob).

parent(tom,liz).

parent(bob,ann).

parent(bob,pat).

parent(pat,jim).

parent(bob,peter).

parent(peter,jim).

mother(X,Y):- parent(X,Y),female(X).

father(X,Y):-parent(X,Y),male(X).

sister(X,Y):-parent(Z,X),parent(Z,Y),female(X),X\==Y.

brother(X,Y):-parent(Z,X),parent(Z,Y),male(X),X\==Y.

grandparent(X,Y):-parent(X,Z),parent(Z,Y).

grandmother(X,Z):-mother(X,Y),parent(Y,Z).

grandfather(X,Z):-father(X,Y),parent(Y,Z).

wife(X,Y):-parent(X,Z),parent(Y,Z),female(X),male(Y).

uncle(X,Z):-brother(X,Y),parent(Y,Z).

**Output**

| ?- [family\_ext].

compiling D:/TP Prolog/Sample\_Codes/family\_ext.pl for byte code...

D:/TP Prolog/Sample\_Codes/family\_ext.pl compiled, 27 lines read - 4646 bytes written, 10 ms

| ?- uncle(X,Y).

X = peter

Y = jim ? ;

no

| ?- grandparent(X,Y).

X = pam

Y = ann ? ;

X = pam

Y = pat ? ;

X = pam

Y = peter ? ;

X = tom

Y = ann ? ;

X = tom

Y = pat ? ;

X = tom

Y = peter ? ;

X = bob

Y = jim ? ;

X = bob

Y = jim ? ;

no

| ?- wife(X,Y).

X = pam

Y = tom ? ;

X = pat

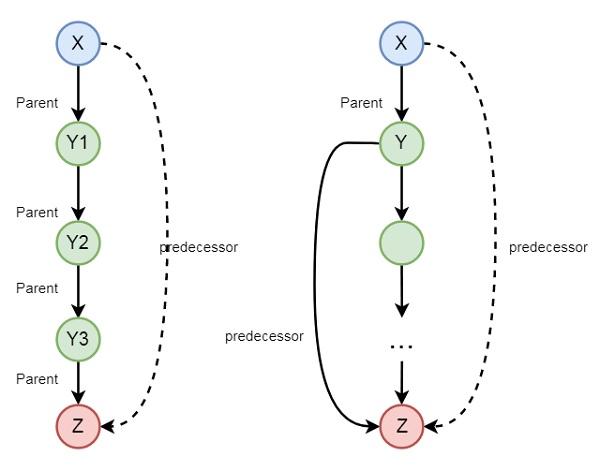
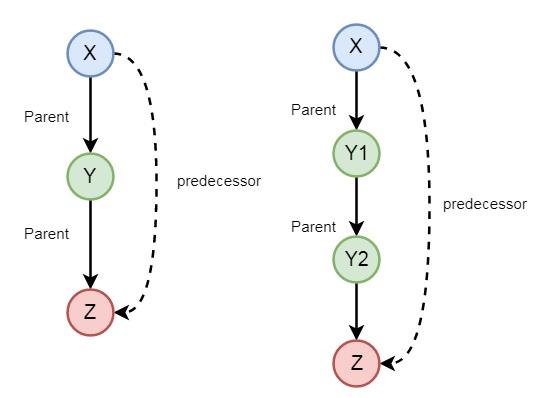
Y = peter ? ;

(15 ms) no

| ?-

**Recursion in Family Relationship**

In the previous section, we have seen that we can define some family relationships. These relationships are static in nature. We can also create some recursive relationships which can be expressed from the following illustration −



So we can understand that predecessor relationship is recursive. We can express this relationship using the following syntax −

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

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

Now let us see the practical demonstration.

**Knowledge Base (family\_rec.pl)**

female(pam).

female(liz).

female(pat).

female(ann).

male(jim).

male(bob).

male(tom).

male(peter).

parent(pam,bob).

parent(tom,bob).

parent(tom,liz).

parent(bob,ann).

parent(bob,pat).

parent(pat,jim).

parent(bob,peter).

parent(peter,jim).

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

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

**Output**

| ?- [family\_rec].

compiling D:/TP Prolog/Sample\_Codes/family\_rec.pl for byte code...

D:/TP Prolog/Sample\_Codes/family\_rec.pl compiled, 21 lines read - 1851 bytes written, 14 ms

yes

| ?- predecessor(peter,X).

X = jim ? ;

no

| ?- trace.

The debugger will first creep -- showing everything (trace)

yes

{trace}

| ?- predecessor(bob,X).

1 1 Call: predecessor(bob,\_23) ?

2 2 Call: parent(bob,\_23) ?

2 2 Exit: parent(bob,ann) ?

1 1 Exit: predecessor(bob,ann) ?

X = ann ? ;

1 1 Redo: predecessor(bob,ann) ?

2 2 Redo: parent(bob,ann) ?

2 2 Exit: parent(bob,pat) ?

1 1 Exit: predecessor(bob,pat) ?

X = pat ? ;

1 1 Redo: predecessor(bob,pat) ?

2 2 Redo: parent(bob,pat) ?

2 2 Exit: parent(bob,peter) ?

1 1 Exit: predecessor(bob,peter) ?

X = peter ? ;

1 1 Redo: predecessor(bob,peter) ?

2 2 Call: parent(bob,\_92) ?

2 2 Exit: parent(bob,ann) ?

3 2 Call: predecessor(ann,\_23) ?

4 3 Call: parent(ann,\_23) ?

4 3 Fail: parent(ann,\_23) ?

4 3 Call: parent(ann,\_141) ?

4 3 Fail: parent(ann,\_129) ?

3 2 Fail: predecessor(ann,\_23) ?

2 2 Redo: parent(bob,ann) ?

2 2 Exit: parent(bob,pat) ?

3 2 Call: predecessor(pat,\_23) ?

4 3 Call: parent(pat,\_23) ?

4 3 Exit: parent(pat,jim) ?

3 2 Exit: predecessor(pat,jim) ?

1 1 Exit: predecessor(bob,jim) ?

X = jim ? ;

1 1 Redo: predecessor(bob,jim) ?

3 2 Redo: predecessor(pat,jim) ?

4 3 Call: parent(pat,\_141) ?

4 3 Exit: parent(pat,jim) ?

5 3 Call: predecessor(jim,\_23) ?

6 4 Call: parent(jim,\_23) ?

6 4 Fail: parent(jim,\_23) ?

6 4 Call: parent(jim,\_190) ?

6 4 Fail: parent(jim,\_178) ?

5 3 Fail: predecessor(jim,\_23) ?

3 2 Fail: predecessor(pat,\_23) ?

2 2 Redo: parent(bob,pat) ?

2 2 Exit: parent(bob,peter) ?

3 2 Call: predecessor(peter,\_23) ?

4 3 Call: parent(peter,\_23) ?

4 3 Exit: parent(peter,jim) ?

3 2 Exit: predecessor(peter,jim) ?

1 1 Exit: predecessor(bob,jim) ?

X = jim ?

(78 ms) yes

{trace}

| ?-