Prolog - Introduction

Prolog as the name itself suggests, is the short form of LOGical PROgramming. It is a logical and declarative programming language. Before diving deep into the concepts of Prolog, let us first understand what exactly logical programming is.

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. Here, the rules are written in the form of logical clauses, where head and body are present. For example, H is head and B1, B2, B3 are the elements of the body. Now if we state that “H is true, when B1, B2, B3 all are true”, this is a rule. On the other hand, facts are like the rules, but without any body. So, an example of fact is “H is true”.

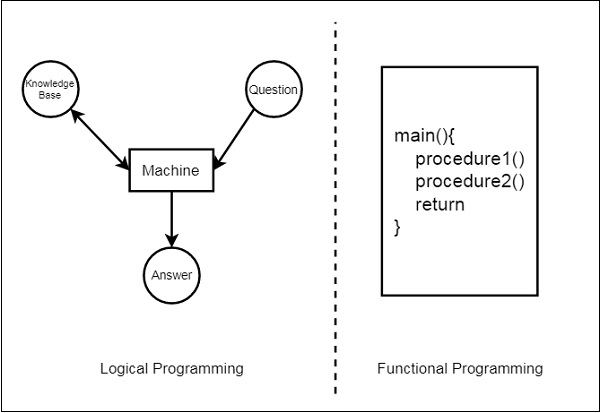
Some logic programming languages like Datalog or ASP (Answer Set Programming) are known as purely declarative languages. These languages allow statements about what the program should accomplish. There is no such step-by-step instruction on how to perform the task. However, other languages like Prolog, have declarative and also imperative properties. This may also include procedural statements like “To solve the problem H, perform B1, B2 and B3”.

Some logic programming languages are given below −

* ALF (algebraic logic functional programming language).
* ASP (Answer Set Programming)
* CycL
* Datalog
* FuzzyCLIPS
* Janus
* Parlog
* Prolog
* Prolog++
* ROOP

Logic and Functional Programming

We will discuss about the differences between Logic programming and the traditional functional programming languages. We can illustrate these two using the below diagram −



From this illustration, we can see that in Functional Programming, we have to define the procedures, and the rule how the procedures work. These procedures work step by step to solve one specific problem based on the algorithm. On the other hand, for the Logic Programming, we will provide knowledge base. Using this knowledge base, the machine can find answers to the given questions, which is totally different from functional programming.

In functional programming, we have to mention how one problem can be solved, but in logic programming we have to specify for which problem we actually want the solution. Then the logic programming automatically finds a suitable solution that will help us solve that specific problem.

Now let us see some more differences below −

|  |  |
| --- | --- |
| **Functional Programming** | **Logic Programming** |
| Functional Programming follows the Von-Neumann Architecture, or uses the sequential steps. | Logic Programming uses abstract model, or deals with objects and their relationships. |
| The syntax is actually the sequence of statements like (a, s, I). | The syntax is basically the logic formulae (Horn Clauses). |
| The computation takes part by executing the statements sequentially. | It computes by deducting the clauses. |
| Logic and controls are mixed together. | Logics and controls can be separated. |

What is Prolog?

Prolog or **PRO**gramming in **LOG**ics is a logical and declarative programming language. It is one major example of the fourth generation language that supports the declarative programming paradigm. This is particularly suitable for programs that involve **symbolic** or **non-numeric computation**. This is the main reason to use Prolog as the programming language in **Artificial Intelligence**, where **symbol manipulation** and **inference manipulation** are the fundamental tasks.

In Prolog, we need not mention the way how one problem can be solved, we just need to mention what the problem is, so that Prolog automatically solves it. However, in Prolog we are supposed to give clues as the **solution method**.

Prolog language basically has three different elements −

**Facts** − The fact is predicate that is true, for example, if we say, “Tom is the son of Jack”, then this is a fact.

**Rules** − Rules are extinctions of facts that contain conditional clauses. To satisfy a rule these conditions should be met. For example, if we define a rule as −

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

This implies that for X to be the grandfather of Y, Z should be a parent of Y and X should be father of Z.

**Questions** − And to run a prolog program, we need some questions, and those questions can be answered by the given facts and rules.

History of Prolog

The heritage of prolog includes the research on theorem provers and some other automated deduction system that were developed in 1960s and 1970s. The Inference mechanism of the Prolog is based on Robinson’s Resolution Principle, that was proposed in 1965, and Answer extracting mechanism by Green (1968). These ideas came together forcefully with the advent of linear resolution procedures.

The explicit goal-directed linear resolution procedures, gave impetus to the development of a general purpose logic programming system. The **first** Prolog was the **Marseille Prolog** based on the work by **Colmerauer** in the year 1970. The manual of this Marseille Prolog interpreter (Roussel, 1975) was the first detailed description of the Prolog language.

Prolog is also considered as a fourth generation programming language supporting the declarative programming paradigm. The well-known Japanese Fifth-Generation Computer Project, that was announced in 1981, adopted Prolog as a development language, and thereby grabbed considerable attention on the language and its capabilities.

Some Applications of Prolog

Prolog is used in various domains. It plays a vital role in automation system. Following are some other important fields where Prolog is used −

* Intelligent Database Retrieval
* Natural Language Understanding
* Specification Language
* Machine Learning
* Robot Planning
* Automation System
* Problem Solving

Prolog - Environment Setup

In this chapter, we will discuss how to install Prolog in our system.

Prolog Version

In this tutorial, we are using GNU Prolog, Version: 1.4.5

Official Website

This is the official GNU Prolog website where we can see all the necessary details about GNU Prolog, and also get the download link.

<http://www.gprolog.org/>

Direct Download Link

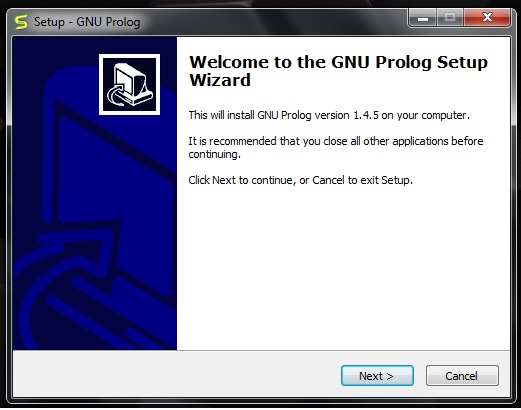
Given below are the direct download links of GNU Prolog for Windows. For other operating systems like Mac or Linux, you can get the download links by visiting the official website (Link is given above) −

<http://www.gprolog.org/setup-gprolog-1.4.5-mingw-x86.exe>(32 Bit System)

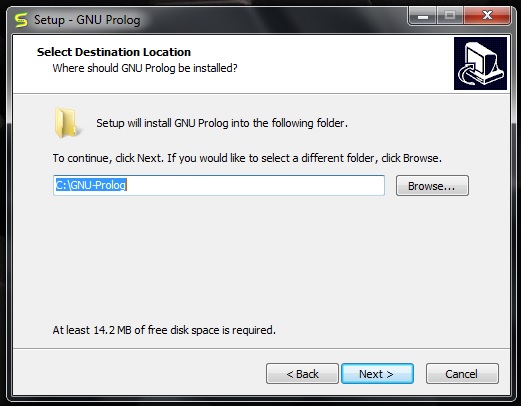
<http://www.gprolog.org/setup-gprolog-1.4.5-mingw-x64.exe>(64 Bit System)

Installation Guide

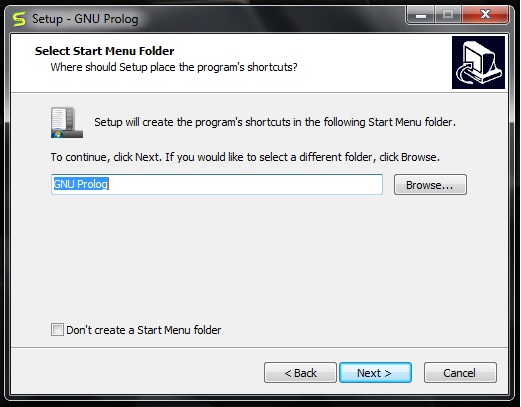
* Download the exe file and run it.
* You will see the window as shown below, then click on **next** −



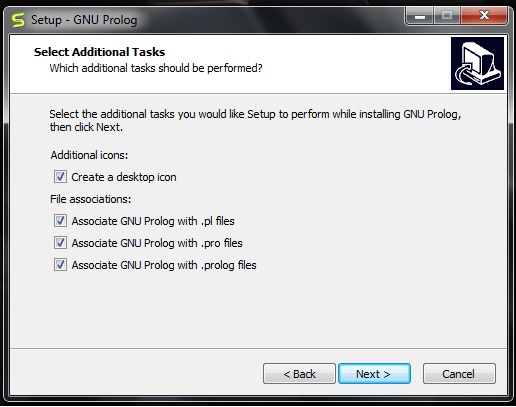
Select proper **directory** where you want to install the software, otherwise let it be installed on the default directory. Then click on **next**.



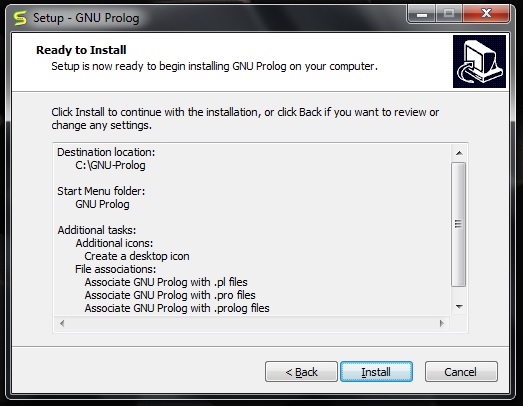
You will get the below screen, simply go to **next**.



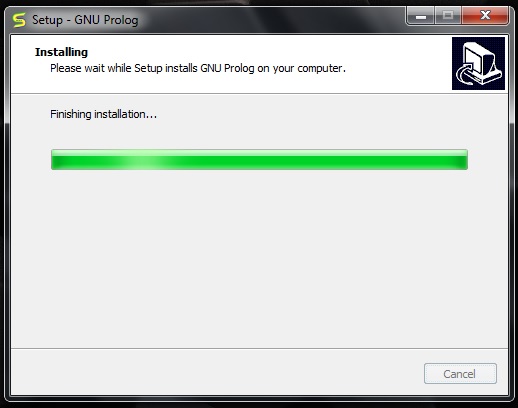
You can verify the below screen, and **check/uncheck**appropriate boxes, otherwise you can leave it as default. Then click on **next**.



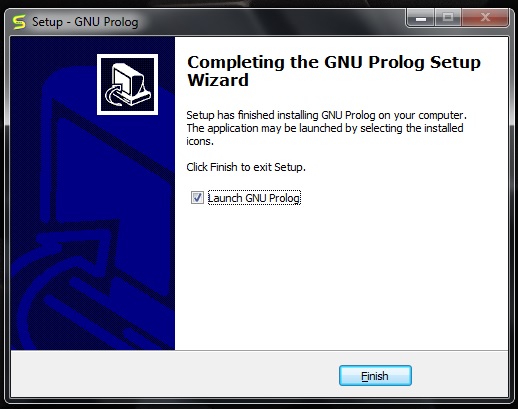
In the next step, you will see the below screen, then click on **Install**.



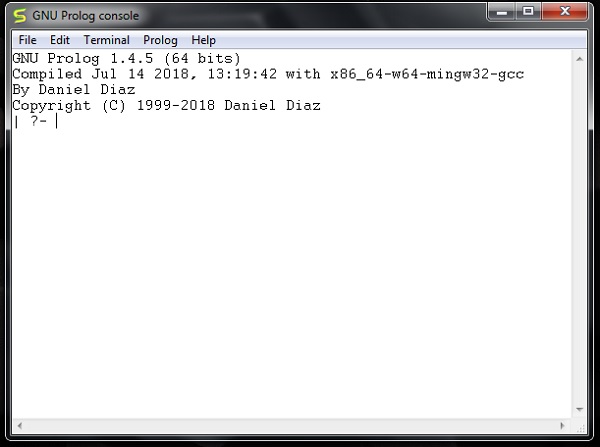
Then wait for the installation process to finish.



Finally click on **Finish** to start GNU Prolog.



The GNU prolog is installed successfully as shown below −



Prolog - Hello World

In the previous section, we have seen how to install GNU Prolog. Now, we will see how to write a simple **Hello World** program in our Prolog environment.

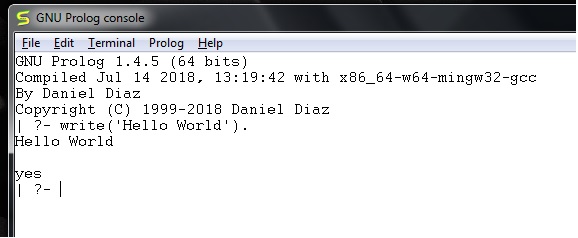
Hello World Program

After running the GNU prolog, we can write hello world program directly from the console. To do so, we have to write the command as follows −

write('Hello World').

**Note** − After each line, you have to use one period (.) symbol to show that the line has ended.

The corresponding output will be as shown below −

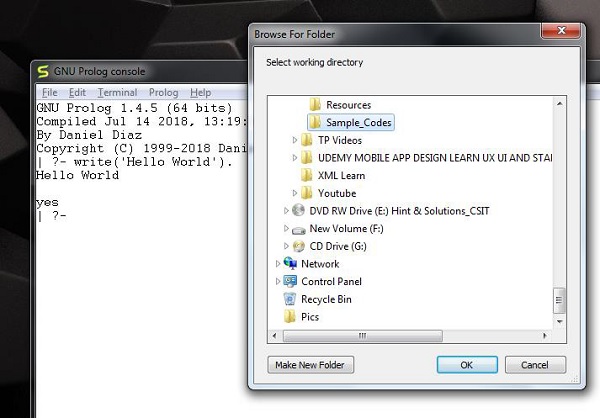


Now let us see how to run the Prolog script file (extension is \*.pl) into the Prolog console.

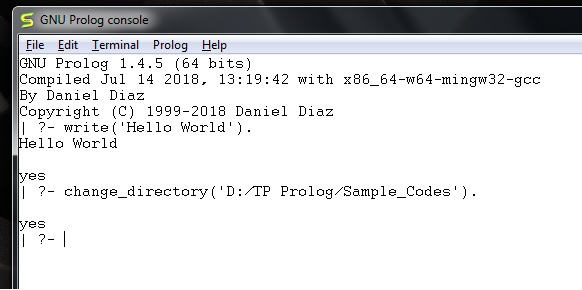
Before running \*.pl file, we must store the file into the directory where the GNU prolog console is pointing, otherwise just change the directory by the following steps −

**Step 1** − From the prolog console, go to File > Change Dir, then click on that menu.

**Step 2** − Select the proper folder and press **OK**.



Now we can see in the prolog console, it shows that we have successfully changed the directory.



**Step 3** − Now create one file (extension is \*.pl) and write the code as follows −

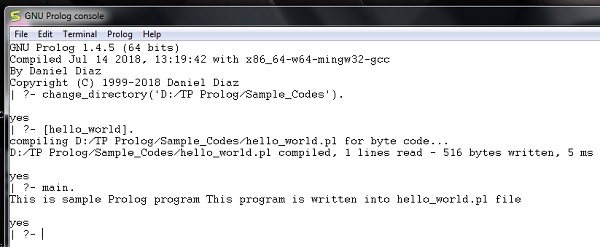
main :- write('This is sample Prolog program'),

write(' This program is written into hello\_world.pl file').

Now let’s run the code. To run it, we have to write the file name as follows −

[hello\_world]

The output is as follows −



Prolog - Basics

In this chapter, we will gain some basic knowledge about Prolog. So we will move on to the first step of our Prolog Programming.

The different topics that will be covered in this chapter are −

**Knowledge Base** − This is one of the fundamental parts of Logic Programming. We will see in detail about the Knowledge Base, and how it helps in logic programming.

**Facts, Rules and Queries** − These are the building blocks of logic programming. We will get some detailed knowledge about facts and rules, and also see some kind of queries that will be used in logic programming.

Here, we will discuss about the essential building blocks of logic programming. These building blocks are **Facts, Rules and the Queries**.

Facts

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 Pasta
* Hair is black
* Nawaz loves to play games
* Pratyusha is lazy.

So these are some facts, that are unconditionally true. These are actually statements, that we have to consider as true.

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.

Syntax

The syntax for facts is as follows −

relation(object1,object2...).

Example

Following is an example of the above concept −

cat(tom).

loves\_to\_eat(kunal,pasta).

of\_color(hair,black).

loves\_to\_play\_games(nawaz).

lazy(pratyusha).

Rules

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.

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.

Syntax

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

object2, ...)

Suppose a clause is like :

P :- Q;R.

This can also be written as

P :- Q.

P :- R.

If one clause is like :

P :- Q,R;S,T,U.

Is understood as

P :- (Q,R);(S,T,U).

Or can also be written as:

P :- Q,R.

P :- S,T,U.

Example

happy(lili) :- dances(lili).

hungry(tom) :- search\_for\_food(tom).

friends(jack, bili) :- lovesCricket(jack), lovesCricket(bili).

goToPlay(ryan) :- isClosed(school), free(ryan).

Queries

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

In this section, we will see what knowledge base in logic programming is.

Well, as we know there are three main components in logic programming − **Facts, Rules** and **Queries**. Among these three if we collect the facts and rules as a whole then that forms a **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).

**Note** − Here we have written the name in lowercase letters, because in Prolog, a string starting with uppercase letter indicates a **variable**.

Now we can use this knowledge base by posing some queries. “Is priya a girl?”, it will reply “yes”, “is jamini a girl?” then it will answer “No”, because it does not know who jamini is. Our next question is “Can Priya cook?”, it will say “yes”, but if we ask the same question for Jaya, it will say “No”.

Output

GNU Prolog 1.4.5 (64 bits)

Compiled Jul 14 2018, 13:19:42 with x86\_64-w64-mingw32-gcc

By Daniel Diaz

Copyright (C) 1999-2018 Daniel Diaz

| ?- change\_directory('D:/TP Prolog/Sample\_Codes').

yes

| ?- [kb1]

.

compiling D:/TP Prolog/Sample\_Codes/kb1.pl for byte code...

D:/TP Prolog/Sample\_Codes/kb1.pl compiled, 3 lines read - 489 bytes written, 10 ms

yes

| ?- girl(priya)

.

yes

| ?- girl(jamini).

no

| ?- can\_cook(priya).

yes

| ?- can\_cook(jaya).

no

| ?-

Let us see another knowledge base, where we have some rules. Rules contain some information that are conditionally true about the domain of interest. Suppose our knowledge base is as follows −

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 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. 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”. So these are some examples of queries based on this Knowledge base.

Output

| ?- [kb2].

compiling D:/TP Prolog/Sample\_Codes/kb2.pl for byte code...

D:/TP Prolog/Sample\_Codes/kb2.pl compiled, 6 lines read - 1066 bytes written, 15 ms

yes

| ?- happy(rohit).

yes

| ?- sing\_a\_song(rohit).

no

| ?- sing\_a\_song(ananya).

yes

| ?- playes\_guitar(rohit).

yes

| ?- playes\_guitar(ananya).

no

| ?- listens\_to\_music(ananya).

yes

| ?-

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.

Output

| ?- [kb3].

compiling D:/TP Prolog/Sample\_Codes/kb3.pl for byte code...

D:/TP Prolog/Sample\_Codes/kb3.pl compiled, 5 lines read - 737 bytes written, 22 ms

warning: D:/TP Prolog/Sample\_Codes/kb3.pl:1: redefining procedure can\_cook/1

D:/TP Prolog/Sample\_Codes/kb1.pl:4: previous definition

yes

| ?- can\_cook(X).

X = priya ? ;

X = jaya ? ;

X = tiyasha

yes

| ?- likes(priya,X).

X = jaya ? ;

X = tiyasha

yes

| ?-

Prolog - Relations

Relationship is one of the main features that we have to properly mention in Prolog. These relationships can be expressed as facts and rules. After that we will see about the family relationships, how we can express family based relationships in Prolog, and also see the recursive relationships of the family.

We will create the knowledge base by creating facts and rules, and play query on them.

Relations in Prolog

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

Suppose, there’s a statement, “Amit has a bike”, then we are actually declaring the ownership relationship between two objects — one is Amit and the other is bike.

If we ask a question, “Does Amit own a bike?”, we are actually trying to find out about one relationship.

There are various kinds of relationships, of which some can be rules as well. A rule can find out about a relationship even if the relationship is not defined explicitly as a fact.

We can define a brother relationship as follows −

Two person are brothers, if,

* They both are male.
* They have the same parent.

Now consider we have the below phrases −

* parent(sudip, piyus).
* parent(sudip, raj).
* male(piyus).
* male(raj).
* brother(X,Y) :- parent(Z,X), parent(Z,Y),male(X), male(Y)

These clauses can give us the answer that piyus and raj are brothers, but we will get three pairs of output here. They are: (piyus, piyus), (piyus, raj), (raj, raj). For these pairs, given conditions are true, but for the pairs (piyus, piyus), (raj, raj), they are not actually brothers, they are the same persons. So we have to create the clauses properly to form a relationship.

The revised relationship can be as follows −

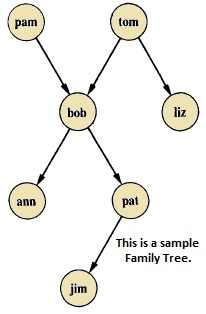
A and B are brothers if −

* A and B, both are male
* They have same father
* They have same mother
* A and B are not same

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).

From our example, it has helped to illustrate some important points −

* We have defined parent relation by stating the n-tuples of objects based on the given info in the family tree.
* The user can easily query the Prolog system about relations defined in the program.
* A Prolog program consists of clauses terminated by a full stop.
* The arguments of relations can (among other things) be: concrete objects, or constants (such as pat and jim), or general objects such as X and Y. Objects of the first kind in our program are called atoms. Objects of the second kind are called variables.
* Questions to the system consist of one or more goals.

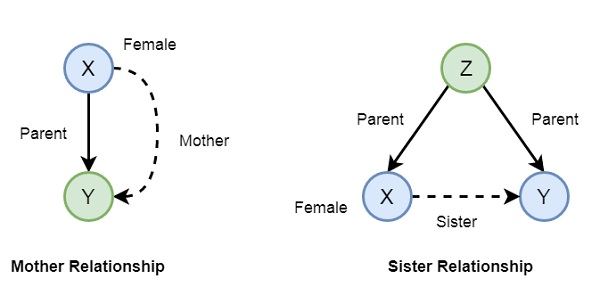
Some facts can be written in two different ways, like sex of family members can be written in either of the forms −

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

Or in the below form −

* sex( pam, feminine).
* sex( tom, masculine).
* sex( bob, masculine).
* … and so on.

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 ? ;

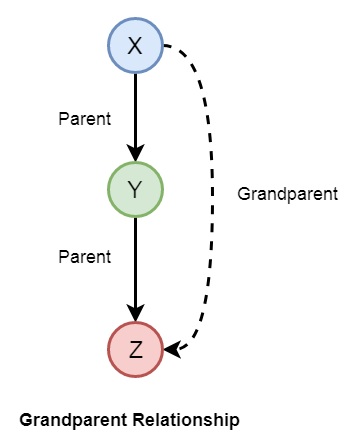
X = pat

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

| ?-

Tracing the output

In Prolog we can trace the execution. To trace the output, you have to enter into the trace mode by typing “trace.”. Then from the output we can see that we are just tracing “pam is mother of whom?”. See the tracing output by taking X = pam, and Y as variable, there Y will be bob as answer. To come out from the tracing mode press “notrace.”

Program

| ?- [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

(16 ms) yes

| ?- mother(X,Y).

X = pam

Y = bob ? ;

X = pat

Y = jim ? ;

no

| ?- trace.

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

yes

{trace}

| ?- mother(pam,Y).

1 1 Call: mother(pam,\_23) ?

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

2 2 Exit: parent(pam,bob) ?

3 2 Call: female(pam) ?

3 2 Exit: female(pam) ?

1 1 Exit: mother(pam,bob) ?

Y = bob

(16 ms) yes

{trace}

| ?- notrace.

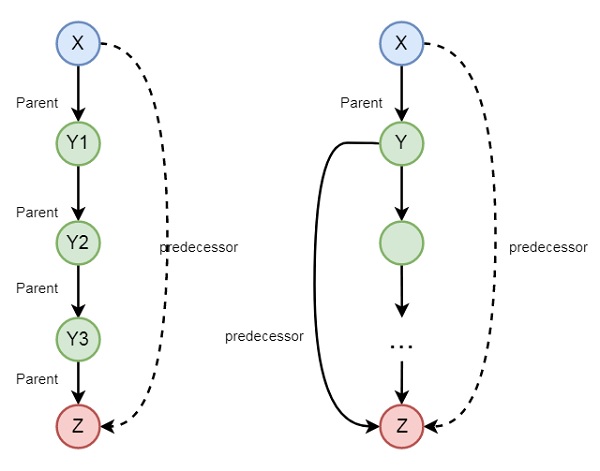
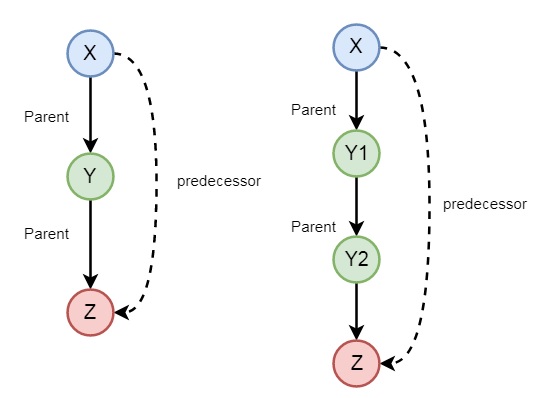
The debugger is switched off

yes

| ?-

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 ?

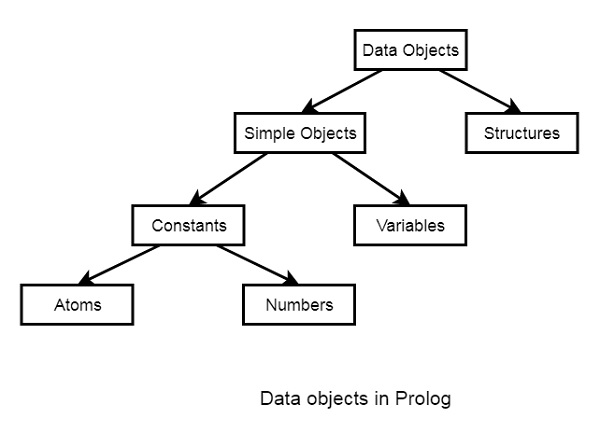
(78 ms) yes

{trace}

| ?-

Prolog - Data Objects

In this chapter, we will learn data objects in Prolog. They can be divided into few different categories as shown below −



Below are some examples of different kinds of data objects −

* Atoms − tom, pat, x100, x\_45
* Numbers − 100, 1235, 2000.45
* Variables − X, Y, Xval, \_X
* Structures − day(9, jun, 2017), point(10, 25)

Atoms and Variables

In this section, we will discuss the atoms, numbers and the variables of Prolog.

Atoms

Atoms are one variation of constants. They can be any names or objects. There are few rules that should be followed when we are trying to use Atoms as given below −

**Strings of letters, digits and the underscore character, ‘\_', starting with a lower-case letter.** For example −

* azahar
* b59
* b\_59
* b\_59AB
* b\_x25
* antara\_sarkar

Strings of special characters

We have to keep in mind that when using atoms of this form, some care is necessary as some strings of special characters already have a predefined meaning; for example ':-'.

* <--->
* =======>
* ...
* .:.
* ::=

Strings of characters enclosed in single quotes.

This is useful if we want to have an atom that starts with a capital letter. By enclosing it in quotes, we make it distinguishable from variables −

* ‘Rubai'
* ‘Arindam\_Chatterjee'
* ‘Sumit Mitra'

Numbers

Another variation of constants is the Numbers. So integer numbers can be represented as 100, 4, -81, 1202. In Prolog, the normal range of integers is from -16383 to 16383.

Prolog also supports real numbers, but normally the use-case of floating point number is very less in Prolog programs, because Prolog is for symbolic, non-numeric computation. The treatment of real numbers depends on the implementation of Prolog. Example of real numbers are 3.14159, -0.00062, 450.18, etc.

The variables come under the **Simple Objects** section. Variables can be used in many such cases in our Prolog program, that we have seen earlier. So there are some rules of defining variables in Prolog.

We can define Prolog variables, such that variables are strings of letters, digits and underscore characters. They **start with** an **upper-case** letter or an **underscore character**. Some examples of Variables are −

* X
* Sum
* Memer\_name
* Student\_list
* Shoppinglist
* \_a50
* \_15

Anonymous Variables in Prolog

Anonymous variables have no names. The anonymous variables in prolog is written by a single underscore character ‘\_’. And one important thing is that each individual anonymous variable is treated as **different**. They are not same.

Now the question is, where should we use these anonymous variables?

Suppose in our knowledge base we have some facts — “jim hates tom”, “pat hates bob”. So if tom wants to find out who hates him, then he can use variables. However, if he wants to check whether there is someone who hates him, we can use anonymous variables. So when we want to use the variable, but do not want to reveal the value of the variable, then we can use anonymous variables.

So let us see its practical implementation −

Knowledge Base (var\_anonymous.pl)

hates(jim,tom).

hates(pat,bob).

hates(dog,fox).

hates(peter,tom).

Output

| ?- [var\_anonymous].

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

D:/TP Prolog/Sample\_Codes/var\_anonymous.pl compiled, 3 lines read - 536 bytes written, 16 ms

yes

| ?- hates(X,tom).

X = jim ? ;

X = peter

yes

| ?- hates(\_,tom).

true ? ;

(16 ms) yes

| ?- hates(\_,pat).

no

| ?- hates(\_,fox).

true ? ;

no

| ?-

Prolog - Operators

In the following sections, we will see what are the different types of operators in Prolog. Types of the comparison operators and Arithmetic operators.

We will also see how these are different from any other high level language operators, how they are syntactically different, and how they are different in their work. Also we will see some practical demonstration to understand the usage of different operators.

Comparison Operators

Comparison operators are used to compare two equations or states. Following are different comparison operators −

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| X > Y | X is greater than Y |
| X < Y | X is less than Y |
| X >= Y | X is greater than or equal to Y |
| X =< Y | X is less than or equal to Y |
| X =:= Y | the X and Y values are equal |
| X =\= Y | the X and Y values are not equal |

You can see that the ‘=<’ operator, ‘=:=’ operator and ‘=\=’ operators are syntactically different from other languages. Let us see some practical demonstration to this.

Example

| ?- 1+2=:=2+1.

yes

| ?- 1+2=2+1.

no

| ?- 1+A=B+2.

A = 2

B = 1

yes

| ?- 5<10.

yes

| ?- 5>10.

no

| ?- 10=\=100.

yes

Here we can see 1+2=:=2+1 is returning true, but 1+2=2+1 is returning false. This is because, in the first case it is checking whether the value of 1 + 2 is same as 2 + 1 or not, and the other one is checking whether two patterns ‘1+2’ and ‘2+1’ are same or not. As they are not same, it returns no (false). In the case of 1+A=B+2, A and B are two variables, and they are automatically assigned to some values that will match the pattern.

Arithmetic Operators in Prolog

Arithmetic operators are used to perform arithmetic operations. There are few different types of arithmetic operators as follows −

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division |
| \*\* | Power |
| // | Integer Division |
| mod | Modulus |

Let us see one practical code to understand the usage of these operators.

Program

calc :- X is 100 + 200,write('100 + 200 is '),write(X),nl,

Y is 400 - 150,write('400 - 150 is '),write(Y),nl,

Z is 10 \* 300,write('10 \* 300 is '),write(Z),nl,

A is 100 / 30,write('100 / 30 is '),write(A),nl,

B is 100 // 30,write('100 // 30 is '),write(B),nl,

C is 100 \*\* 2,write('100 \*\* 2 is '),write(C),nl,

D is 100 mod 30,write('100 mod 30 is '),write(D),nl.

**Note** − The nl is used to create new line.

Output

| ?- change\_directory('D:/TP Prolog/Sample\_Codes').

yes

| ?- [op\_arith].

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

D:/TP Prolog/Sample\_Codes/op\_arith.pl compiled, 6 lines read - 2390 bytes written, 11 ms

yes

| ?- calc.

100 + 200 is 300

400 - 150 is 250

10 \* 300 is 3000

100 / 30 is 3.3333333333333335

100 // 30 is 3

100 \*\* 2 is 10000.0

100 mod 30 is 10

yes

| ?-

Prolog - Loop & Decision Making

In this chapter, we will discuss loops and decision making in Prolog.

Loops

Loop statements are used to execute the code block multiple times. In general, for, while, do-while are loop constructs in programming languages (like Java, C, C++).

Code block is executed multiple times using recursive predicate logic. There are no direct loops in some other languages, but we can simulate loops with few different techniques.

Program

count\_to\_10(10) :- write(10),nl.

count\_to\_10(X) :-

write(X), nl,

Y is X + 1,

count\_to\_10(Y).

Output

| ?- [loop].

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

D:/TP Prolog/Sample\_Codes/loop.pl compiled, 4 lines read - 751 bytes written, 16 ms

(16 ms) yes

| ?- count\_to\_10(3).

3

4

5

6

7

8

9

10

true ?

yes

| ?-

Now create a loop that takes lowest and highest values. So, we can use the between() to simulate loops.

Program

Let us see an example program −

count\_down(L, H) :-

between(L, H, Y),

Z is H - Y,

write(Z), nl.

count\_up(L, H) :-

between(L, H, Y),

Z is L + Y,

write(Z), nl.

Output

| ?- [loop].

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

D:/TP Prolog/Sample\_Codes/loop.pl compiled, 14 lines read - 1700 bytes written, 16 ms

yes

| ?- count\_down(12,17).

5

true ? ;

4

true ? ;

3

true ? ;

2

true ? ;

1

true ? ;

0

yes

| ?- count\_up(5,12).

10

true ? ;

11

true ? ;

12

true ? ;

13

true ? ;

14

true ? ;

15

true ? ;

16

true ? ;

17

yes

| ?-

Decision Making

The decision statements are If-Then-Else statements. So when we try to match some condition, and perform some task, then we use the decision making statements. The basic usage is as follows −

If <condition> is true, Then <do this>, Else

In some different programming languages, there are If-Else statements, but in Prolog we have to define our statements in some other manner. Following is an example of decision making in Prolog.

Program

% If-Then-Else statement

gt(X,Y) :- X >= Y,write('X is greater or equal').

gt(X,Y) :- X < Y,write('X is smaller').

% If-Elif-Else statement

gte(X,Y) :- X > Y,write('X is greater').

gte(X,Y) :- X =:= Y,write('X and Y are same').

gte(X,Y) :- X < Y,write('X is smaller').

Output

| ?- [test].

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

D:/TP Prolog/Sample\_Codes/test.pl compiled, 3 lines read - 529 bytes written, 15 ms

yes

| ?- gt(10,100).

X is smaller

yes

| ?- gt(150,100).

X is greater or equal

true ?

yes

| ?- gte(10,20).

X is smaller

(15 ms) yes

| ?- gte(100,20).

X is greater

true ?

yes

| ?- gte(100,100).

X and Y are same

true ?

yes

| ?-

Prolog - Conjunctions & Disjunctions

In this chapter, we shall discuss Conjunction and Disjunction properties. These properties are used in other programming languages using AND and OR logics. Prolog also uses the same logic in its syntax.

Conjunction

Conjunction (AND logic) can be implemented using the comma (,) operator. So two predicates separated by comma are joined with AND statement. Suppose we have a predicate, **parent(jhon, bob)**, which means “Jhon is parent of Bob”, and another predicate, **male(jhon)**, which means “Jhon is male”. So we can make another predicate that **father(jhon,bob)**, which means “Jhon is father of Bob”. We can define predicate **father**, when he is parent **AND** he is male.

Disjunction

Disjunction (OR logic) can be implemented using the semi-colon (;) operator. So two predicates separated by semi-colon are joined with OR statement. Suppose we have a predicate, **father(jhon, bob)**. This tells that “Jhon is father of Bob”, and another predicate, **mother(lili,bob)**, this tells that “lili is mother of bob”. If we create another predicate as **child()**, this will be true when **father(jhon, bob)** is true **OR mother(lili,bob)** is true.

Program

parent(jhon,bob).

parent(lili,bob).

male(jhon).

female(lili).

% Conjunction Logic

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

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

% Disjunction Logic

child\_of(X,Y) :- father(X,Y);mother(X,Y).

Output

| ?- [conj\_disj].

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

D:/TP Prolog/Sample\_Codes/conj\_disj.pl compiled, 11 lines read - 1513 bytes written, 24 ms

yes

| ?- father(jhon,bob).

yes

| ?- child\_of(jhon,bob).

true ?

yes

| ?- child\_of(lili,bob).

yes

| ?-

Prolog - Lists

In this chapter, we will discuss one of the important concepts in Prolog, The Lists. It is a data structure that can be used in different cases for non-numeric programming. Lists are used to store the atoms as a collection.

In the subsequent sections, we will discuss the following topics −

* Representation of lists in Prolog
* Basic operations on prolog such as Insert, delete, update, append.
* Repositioning operators such as permutation, combination, etc.
* Set operations like set union, set intersection, etc.

Representation of Lists

The list is a simple data structure that is widely used in non-numeric programming. List consists of any number of items, for example, red, green, blue, white, dark. It will be represented as, [red, green, blue, white, dark]. The list of elements will be enclosed with **square brackets**.

A list can be either **empty** or **non-empty**. In the first case, the list is simply written as a Prolog atom, []. In the second case, the list consists of two things as given below −

* The first item, called the **head** of the list;
* The remaining part of the list, called the **tail**.

Suppose we have a list like: [red, green, blue, white, dark]. Here the head is red and tail is [green, blue, white, dark]. So the tail is another list.

Now, let us consider we have a list, L = [a, b, c]. If we write Tail = [b, c] then we can also write the list L as L = [ a | Tail]. Here the vertical bar (|) separates the head and tail parts.

So the following list representations are also valid −

* [a, b, c] = [x | [b, c] ]
* [a, b, c] = [a, b | [c] ]
* [a, b, c] = [a, b, c | [ ] ]

For these properties we can define the list as −

A data structure that is either empty or consists of two parts − a head and a tail. The tail itself has to be a list.

Basic Operations on Lists

Following table contains various operations on prolog lists −

|  |  |
| --- | --- |
| **Operations** | **Definition** |
| Membership Checking | During this operation, we can verify whether a given element is member of specified list or not? |
| Length Calculation | With this operation, we can find the length of a list. |
| Concatenation | Concatenation is an operation which is used to join/add two lists. |
| Delete Items | This operation removes the specified element from a list. |
| Append Items | Append operation adds one list into another (as an item). |
| Insert Items | This operation inserts a given item into a list. |

Membership Operation

During this operation, we can check whether a member X is present in list L or not? So how to check this? Well, we have to define one predicate to do so. Suppose the predicate name is **list\_member(X,L)**. The goal of this predicate is to check whether X is present in L or not.

To design this predicate, we can follow these observations. X is a member of L if either −

* X is head of L, or
* X is a member of the tail of L

Program

list\_member(X, [X|\_]).

list\_member(X,[\_|TAIL]) :- list\_member(X,TAIL).

Output

| ?- [list\_basics].

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

D:/TP Prolog/Sample\_Codes/list\_basics.pl compiled, 1 lines read - 467 bytes written, 13 ms

yes

| ?- list\_member(b,[a,b,c]).

true ?

yes

| ?- list\_member(b,[a,[b,c]]).

no

| ?- list\_member([b,c],[a,[b,c]]).

true ?

yes

| ?- list\_member(d,[a,b,c]).

no

| ?- list\_member(d,[a,b,c]).

Length Calculation

This is used to find the length of list L. We will define one predicate to do this task. Suppose the predicate name is **list\_length(L,N)**. This takes L and N as input argument. This will count the elements in a list L and instantiate N to their number. As was the case with our previous relations involving lists, it is useful to consider two cases −

* If list is empty, then length is 0.
* If the list is not empty, then L = [Head|Tail], then its length is 1 + length of Tail.

Program

list\_length([],0).

list\_length([\_|TAIL],N) :- list\_length(TAIL,N1), N is N1 + 1.

Output

| ?- [list\_basics].

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

D:/TP Prolog/Sample\_Codes/list\_basics.pl compiled, 4 lines read - 985 bytes written, 23 ms

yes

| ?- list\_length([a,b,c,d,e,f,g,h,i,j],Len).

Len = 10

yes

| ?- list\_length([],Len).

Len = 0

yes

| ?- list\_length([[a,b],[c,d],[e,f]],Len).

Len = 3

yes

| ?-

Concatenation

Concatenation of two lists means adding the list items of the second list after the first one. So if two lists are [a,b,c] and [1,2], then the final list will be [a,b,c,1,2]. So to do this task we will create one predicate called list\_concat(), that will take first list L1, second list L2, and the L3 as resultant list. There are two observations here.

* If the first list is empty, and second list is L, then the resultant list will be L.
* If the first list is not empty, then write this as [Head|Tail], concatenate Tail with L2 recursively, and store into new list in the form, [Head|New List].

Program

list\_concat([],L,L).

list\_concat([X1|L1],L2,[X1|L3]) :- list\_concat(L1,L2,L3).

Output

| ?- [list\_basics].

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

D:/TP Prolog/Sample\_Codes/list\_basics.pl compiled, 7 lines read - 1367 bytes written, 19 ms

yes

| ?- list\_concat([1,2],[a,b,c],NewList).

NewList = [1,2,a,b,c]

yes

| ?- list\_concat([],[a,b,c],NewList).

NewList = [a,b,c]

yes

| ?- list\_concat([[1,2,3],[p,q,r]],[a,b,c],NewList).

NewList = [[1,2,3],[p,q,r],a,b,c]

yes

| ?-

Delete from List

Suppose we have a list L and an element X, we have to delete X from L. So there are three cases −

* If X is the only element, then after deleting it, it will return empty list.
* If X is head of L, the resultant list will be the Tail part.
* If X is present in the Tail part, then delete from there recursively.

Program

list\_delete(X, [X], []).

list\_delete(X,[X|L1], L1).

list\_delete(X, [Y|L2], [Y|L1]) :- list\_delete(X,L2,L1).

Output

| ?- [list\_basics ].

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

D:/TP Prolog/Sample\_Codes/list\_basics.pl compiled, 11 lines read - 1923 bytes written, 25 ms

yes

| ?- list\_delete(a,[a,e,i,o,u],NewList).

NewList = [e,i,o,u] ?

yes

| ?- list\_delete(a,[a],NewList).

NewList = [] ?

yes

| ?- list\_delete(X,[a,e,i,o,u],[a,e,o,u]).

X = i ? ;

no

| ?-

Append into List

Appending two lists means adding two lists together, or adding one list as an item. Now if the item is present in the list, then the append function will not work. So we will create one predicate namely, list\_append(L1, L2, L3). The following are some observations −

* Let A is an element, L1 is a list, the output will be L1 also, when L1 has A already.
* Otherwise new list will be L2 = [A|L1].

Program

list\_member(X,[X|\_]).

list\_member(X,[\_|TAIL]) :- list\_member(X,TAIL).

list\_append(A,T,T) :- list\_member(A,T),!.

list\_append(A,T,[A|T]).

In this case, we have used (!) symbol, that is known as cut. So when the first line is executed successfully, then we cut it, so it will not execute the next operation.

Output

| ?- [list\_basics].

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

D:/TP Prolog/Sample\_Codes/list\_basics.pl compiled, 14 lines read - 2334 bytes written, 25 ms

(16 ms) yes

| ?- list\_append(a,[e,i,o,u],NewList).

NewList = [a,e,i,o,u]

yes

| ?- list\_append(e,[e,i,o,u],NewList).

NewList = [e,i,o,u]

yes

| ?- list\_append([a,b],[e,i,o,u],NewList).

NewList = [[a,b],e,i,o,u]

yes

| ?-

Insert into List

This method is used to insert an item X into list L, and the resultant list will be R. So the predicate will be in this form list\_insert(X, L, R). So this can insert X into L in all possible positions. If we see closer, then there are some observations.

* If we perform list\_insert(X,L,R), we can use list\_delete(X,R,L), so delete X from R and make new list L.

Program

list\_delete(X, [X], []).

list\_delete(X,[X|L1], L1).

list\_delete(X, [Y|L2], [Y|L1]) :- list\_delete(X,L2,L1).

list\_insert(X,L,R) :- list\_delete(X,R,L).

Output

| ?- [list\_basics].

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

D:/TP Prolog/Sample\_Codes/list\_basics.pl compiled, 16 lines read - 2558 bytes written, 22 ms

(16 ms) yes

| ?- list\_insert(a,[e,i,o,u],NewList).

NewList = [a,e,i,o,u] ? a

NewList = [e,a,i,o,u]

NewList = [e,i,a,o,u]

NewList = [e,i,o,a,u]

NewList = [e,i,o,u,a]

NewList = [e,i,o,u,a]

(15 ms) no