## UNIT -5

PROLOG:

## What is Prolog?

* Prolog stands for **Programming in logic.** It is used in artificial intelligence programming.
* Prolog is a **declarative** programming language.  
  **For example:** While implementing the solution for a given problem, instead of specifying the ways to achieve a certain goal in a specific situation, user needs to specify about the situation (rules and facts) and the goal (query). After these stages, Prolog interpreter derives the solution.
* Prolog is useful in AI, NLP, databases but useless in other areas such as graphics or numerical algorithms.

#### **Prolog facts**

* A fact is something that seems to be true.  
  **For example:** It's raining.
* In Prolog, facts are used to form the statements. Facts consist of a specific item or relation between two or more items.

#### **How to convert English to prolog facts using facts and rules?**

* It is very simple to convert English sentence into Prolog facts. Some examples are explained in the following table.

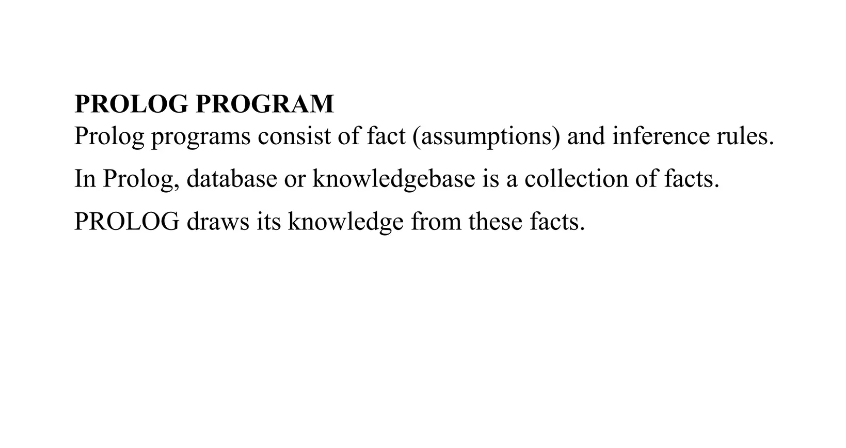
| **English Statements** | **Prolog Facts** |
| --- | --- |
| Dog is barking | barking(dog) |
| Jaya likes food if it is delicious. | likes( Jaya, Food):-delicious(Food) |

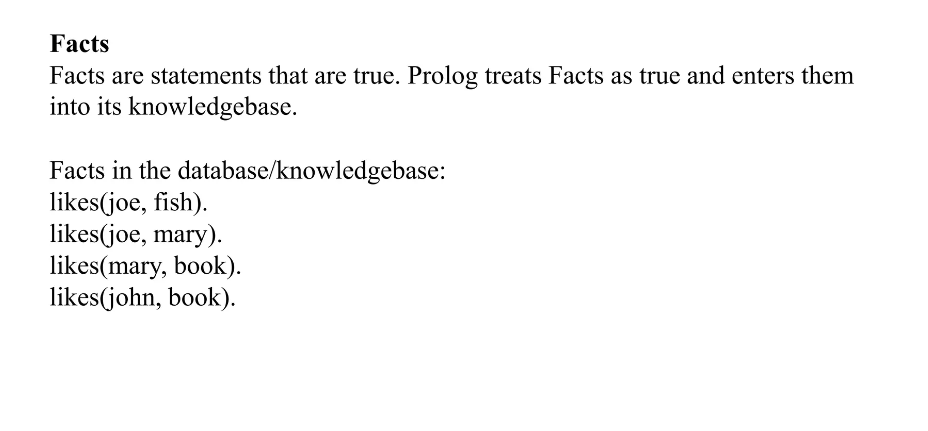
In the above table, the statement **'Dog is barking'** is a **fact,** while the statement **'Jaya likes food if it is delicious'** is called **rule.** In this statement, variable like 'Food' has a first letter in capital, because its value came from previous fact. The symbol **':-'** is used to denote that “Jaya likes delicious food”.

Syntax Prolog programs are constructed from terms: constants, variables, or structures. Constants can be either atoms or numbers:

• Atoms are strings of characters starting with a lowercase letter or enclosed in apostrophes.

• Numbers are strings of digits with or without a decimal point and a minus sign. Variables are strings of characters beginning with an uppercase letter or an underscore.





# 

**Examples of Simple Facts**

Here are some simple facts about an imaginary world. /\* and \*/ are comment delimiters

john\_is\_cold. /\* john is cold \*/

raining. /\* it is raining \*/

john\_Forgot\_His\_Raincoat. /\* john forgot his raincoat \*/

fred\_lost\_his\_car\_keys. /\* fred lost is car keys \*/

peter\_footballer. /\* peter plays football \*/

These describe a particular set of circumstances for some character john. We can interrogate this database of facts, by again posing a query. For example: {note the responses of the Prolog interpreter are shown in italics}

?- john\_Forgot\_His\_Raincoat.

yes

?- raining.

yes

?- foggy.

no

**Facts with Arguments**

More complicated facts consist of a relation and the items that this refers to. These items are called arguments. Facts can have arbitrary number of arguments from zero upwards. A general model is shown below:

**relation(<argument1>,<argument2>,....,<argumentN> ).**

Relation names must begin with a lowercase letter

**likes(john,mary).**

The above fact says that a relationship likes links john and mary. This fact may be read as either john likes mary or mary likes john. This reversibility can be very useful to the programmer, however it can also lead to some pitfalls. Thus you should always be clear and consistent how you intend to interpret the relation and when.

Finally, remember names of the relations are defined by you. With the exception of a few relations that the system has built into it, the system only knows about relations that you tell it about.

Facts with Arguments Examples 1

An example database. It details who eats what in some world model.

eats(fred,oranges). /\* "Fred eats oranges" \*/

eats(fred,t\_bone\_steaks). /\* "Fred eats T-bone steaks" \*/

eats(tony,apples). /\* "Tony eats apples" \*/

eats(john,apples). /\* "John eats apples" \*/

eats(john,grapefruit). /\* "John eats grapefruit" \*/

**If we now ask some queries we would get the following interaction:**

?- eats(fred,oranges). /\* does this match anything in the database? \*/

yes /\* yes, matches the first clause in the database \*/

?- eats(john,apples). /\* do we have a fact that says john eats apples? \*/

yes /\* yes we do, clause 4 of our eats database \*/

?- eats(mike,apples). /\* how about this query, does mike eat apples \*/

no /\* not according to the above database. \*/

?- eats(fred,apples). /\* does fred eat apples \*/

no /\* again no, we don't know whether fred eats apples \*/

**Facts with Arguments Examples 2**

Let us consider another database. This time we will use the predicate age to indicate the ages of various individuals.

age(john,32). /\* John is 32 years old \*/

age(agnes,41). /\* Agnes is 41 \*/

age(george,72). /\* George is 72 \*/

age(ian,2). /\* Ian is 2 \*/

age(thomas,25). /\* Thomas is 25 \*/

If we now ask some queries we would get the following interaction:

?- age(ian,2). /\* is Ian 2 years old? \*/

yes /\* yes, matches against the fourth clause of age \*/

?- agnes(41). /\* for some relation agnes are they 41 \*/

no /\* No. In the database above we only know about the relation \*/

/\* age, not about the relation agnes, so the query fails \*/

?- age(ian,two) /\* is Ian two years old? \*/

no /\* No. two and 2 are not the same and therefore don't match \*/

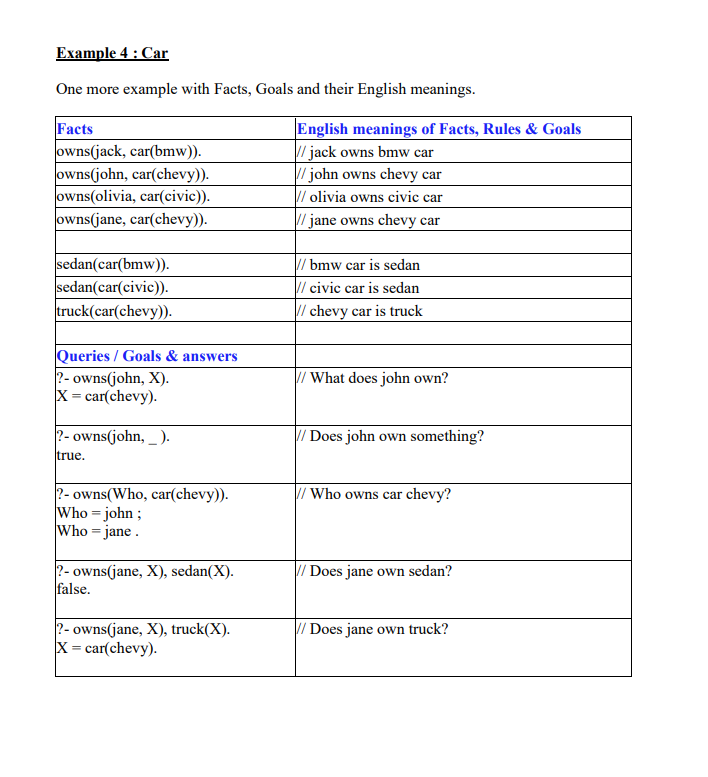
This is the end of the facts with arguments topic.

**Example 1 : Below food table shows the facts, rules, goals and their english meanings.**

| **Facts** | **English meanings** |
| --- | --- |
| food(burger). | // burger is a food |
| food(sandwich). | // sandwich is a food |
| food(pizza). | // pizza is a food |
| lunch(sandwich). | // sandwich is a lunch |
| dinner(pizza). | // pizza is a dinner |
|  |  |
| **Rules** |  |
| meal(X) :- food(X). | // Every food is a meal OR Anything is a meal if it is a food |
|  |  |
| **Queries / Goals** |  |
| ?- food(pizza). | // Is pizza a food? |
| ?- meal(X), lunch(X). | // Which food is meal and lunch? |
| ?- dinner(sandwich). | // Is sandwich a dinner? |

**Example 2 :** Below student-professor relation table shows the facts, rules, goals and their english meanings.

| **Facts** | **English meanings** |
| --- | --- |
| studies(charlie, csc135). | // charlie studies csc135 |
| studies(olivia, csc135). | // olivia studies csc135 |
| studies(jack, csc131). | // jack studies csc131 |
| studies(arthur, csc134). | // arthur studies csc134 |
|  |  |
| teaches(kirke, csc135). | // kirke teaches csc135 |
| teaches(collins, csc131). | // collins teaches csc131 |
| teaches(collins, csc171). | // collins teaches csc171 |
| teaches(juniper, csc134). | // juniper teaches csc134 |
|  |  |
| **Rules** |  |
| professor(X, Y) :- teaches(X, C), studies(Y, C). | // X is a professor of Y if X teaches C and Y studies C. |
| **Queries / Goals** |  |
| ?- studies(charlie, What). | // charlie studies what? OR What does charlie study? |
| ?- professor(kirke, Students). | // Who are the students of professor kirke. |



# **TRY ONLINE:https://swish.swi-prolog.org/**

# **Write a program in prolog to implement simple facts and Queries**

% Facts

1. Ram likes mango.
2. Seema is a girl
3. Bill likes Cindy
4. Rose is red.
5. John owns gold.

% Clauses

likes(ram ,mango).

girl(seema).

red(rose).

likes(bill ,cindy).

owns(john ,gold).

% Queries

?-likes(ram,What).

What= mango

?-likes(Who,cindy).

Who= cindy

?-red(What).

What= rose

?-owns(Who,What).

Who= john

What= gold

## 1.2    A Simple Thought (Basic Inference)

People wish to live in piece. Men, women and children are people. I am a man (a woman). Therefore I wish to live in peace.

Use Prolog to prove this statement!

### [Solution 1.2](https://www.nosco.ch/ai/en/solution_01.php#1.2)

## [1.2    A Simple Thought (Basic Inference)](https://www.nosco.ch/ai/en/exercise_01.php#1.2)

wish\_to\_live\_in\_peace( X) :- people( X).

people( X) :- man( X).

people( X) :- woman( X).

people( X) :- child( X).

man( 'I')./\* ?-wish\_to\_live\_in\_peace( 'I') ==> true \*/

### Select code

### 

### 

### 

### 

### 

PROGRAM 1 : Program to add two numbers.

predicates

add

Clauses

add:-write("input first number"), readint(X), write("input second number"), readint(Y), Z=X+Y,write("output=",Z).

**PROGRAM 2 :Program to categorise animal characteristics.**

### predicates

small(symbol) large(symbol) color(symbol,symbol)

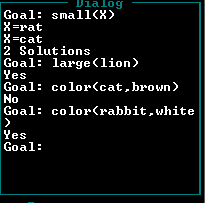
### clauses

small(rat). small(cat).

large(lion).

color(dog,black). color(rabbit,white). color(X,dark):-

color(X,black);color(X,brown).



**PROGRAM 3 : Program to read address of a person using compound variable .**

### domains

person=address(name,street,city,state,zip) name,street,city,state,zip=String

### predicates

readaddress(person) go

### clauses

go:- readaddress(Address),nl,write(Address),nl,nl,write("Accept(y/n)?"),readchar(Reply),Reply='y',!. go:-

nl,write("please re-enter"),nl,go. readaddress(address(N,street,city,state,zip)):- write("Name:"),readln(N), write("Street:"),readln(street), write("City:"),readln(city), write("State:"),readln(state), write("Zip:"),readln(zip).

## Output:-

**PROGRAM 6 : Program to reverse the list .**

### domains

x=integer list=integer\*

### predicates

append(x,list,list) rev(list,list)

### clauses

append(X,[],[X]). append(X,[H|T],[H|T1]):-append(X,T,T1). rev([],[]).

rev([H|T,rev):-rev(T,L),append(H,L,rev).

**Output:-**

**PROGRAM 7 : Program to append an integer into the list .**

### domains

x=integer list=integer\*

### predicates

append(x,list,list)

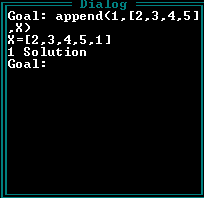
### clauses

append(X,[],[X]).

append(X,[H|T],[H|T1]):-

append(X,T,T1).

**Output:-**





**PROGRAM 8: Program to replace an integer from the list .**

### domains

list=integer\*

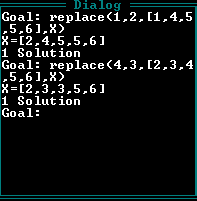
### predicates

replace(integer,integer,list,list)

### clauses

replace(X,Y,[X|T],[Y|T]). replace(X,Y,[H|T],[H|T1]):-replace(X,Y,T,T1).

**Output:-**





**PROGRAM 9 : Program to delete an integer from the list .**

### domains

list=integer\*

### predicates

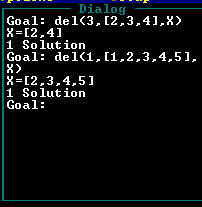
del(integer,list,list)

### clauses

del(X,[X|T],T). del(X,[H|T],[H|T1]):-

del(X,T,T1).

**Output:-**





**PROGRAM 10 : Program to show concept of list.**

### domains

name=symbol\*

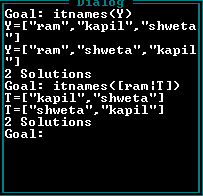
### predicates

itnames(name)

### clauses

itnames([ram,kapil,shweta]). itnames([ram,shweta,kapil]).

**Output:-**





**PROGRAM 11 : Program to demonstrate family relationship**

### predicates

parent(symbol,symbol) child(symbol,symbol) mother(symbol,symbol)

brother(symbol,symbol) sister(symbol,symbol) grandparent(symbol,symbol) male(symbol) female(symbol)

### clauses

parent(a,b).

sister(a,c).

male(a). female(b).

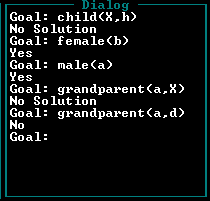
child(X,Y):-parent(Y,X).

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

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

brother(X,Y):-male(X),parent(V,X),parent(V,Y).

**Output:-**





**PROGRAM 12 : Program to show how integer variable is used in prolog program**

### predicates

go

### clauses

go:-X=10,

write(X), nl,X=20,

write(X),nl.

## Output:-C:\Users\hp\Desktop\13.png

# **Write a program in prolog to solve Monkey banana problem**

## **Solution :**

/\* Description:

Imagine a room containing a monkey, chair and some bananas. That have been hanged from the center of ceiling. If the monkey is clever enough he can reach the bananas by placing the chair directly below the bananas and climb on the chair .

The problem is to prove the monkey can reach the bananas.

The monkey can perform the following actions:

1) Walk on the floor

2) Climb the box

3) Push the box around(if it is beside the box).

4) Grasp the banana if it is standing on the box directly under the banana.

\*/

% Production rules:

can\_reach 🡪 clever,close.

get\_on: 🡪 can\_climb.

under 🡪 in room,in\_room, in\_room,can\_climb.

Close 🡪 get\_on,under | tall

% Clauses:

in\_room(bananas).in\_room(chair).in\_room(monkey).clever(monkey).can\_climb(monkey, chair).tall(chair).can\_move(monkey, chair, bananas).can\_reach(X, Y):-clever(X),close(X, Y).get\_on(X,Y):-

can\_climb(X,Y).under(Y,Z):-

in\_room(X),in\_room(Y),

in\_room(Z),can\_climb(X,Y,Z).close(X,Z):-

get\_on(X,Y), under(Y,Z);

tall(Y).

## **Output:**

% Queries:

?- can\_reach(A, B).

A = monkey.

B = banana.

?- can\_reach(monkey, banana).

Yes.

# Introduction to LISP

Lisp is a programming language that has an overall style that is organized around expressions and functions. Every Lisp procedure is a function, and when called, it returns a data object as its value. It is also commonly referred to as “functions” even though they may have side effects.

Lisp is the second-oldest high-level programming language in the world which is invented by John McCarthy in the year 1958 at the Massachusetts Institute of Technology.

### Features of  LISP Programming Language:

1. It is a machine-independent language
2. It uses iterative design methodology and is easily extensible
3. It allows us to create and update the programs and applications dynamically.
4. It provides high-level debugging.
5. It supports object-oriented programming.
6. It supports all kinds of data types like objects, structures, lists, vectors, adjustable arrays, set, trees,hash-tables, and symbols.
7. It is an expression-based language
8. It can support different decision-making statements like if, when,case, and cond
9. It will also support different iterating statements like do, loop,loopfor, dotimes and dolist.
10. It will support input and output functions
11. By using lisp we can also create our own functions

# Functions in LISP

A function is a set of statements that takes some input, performs some tasks, and produces the result. Through functions, we can split up a huge task into many smaller functions. They also help in avoiding the repetition of code as we can call the same function for different inputs.

### Defining Functions in LISP:

Functions in LISP are defined using the **DEFUN** macro. The basic syntax looks like this :

(defun function-name (parameters)

"Documentation string"

body-of-function

)

### Example:

1. Let’s create a function named ***hello-world*** that doesn’t take any parameters and returns a hello world string.

* Lisp

| (defun hello**-**world ()    (format **t** "Hello, World!")  )  (hello**-**world) |
| --- |

 Function to add two integers

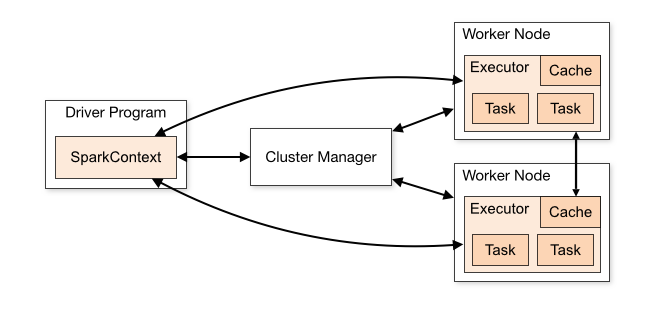
| (defun add**-**two**-**number (n1 n2)    "Adds two numbers"    (**+** n1 n2)  )  (write(add**-**two**-**number 10 20)) |
| --- |

Apache Spark is an open-source distributed computing system designed for big data processing and analytics. Spark is known for its speed and efficiency. If you want more introduction about spark I have already covered that part in [this](https://medium.com/@amitjoshi7/an-introduction-to-apache-spark-big-data-processing-made-easy-d1288153607d) blog. In this article, we will explore the following topics:

* spark Architecture And Applications
* working of spark Architecture
* Abstractions of Apache Spark
* Cluster Manager Types
* Execution Modes

Let's start...

The Apache Spark framework uses a **master-slave architecture** that consists of a driver, which runs as a master node, and many executors that run across as worker nodes in the cluster. Apache Spark can be used for batch processing and real-time processing as well.



Reference:[https://www.analyticsvidhya.com](https://www.analyticsvidhya.com/)

Before understanding the Spark architecture let's understand the Applications of Spark Architecture which exists in the above diagram:

## **The Spark driver**

The driver is the program or process responsible for coordinating the execution of the Spark application. It runs the main function and creates the SparkContext, which connects to the cluster manager.

## **The Spark executors**

Executors are worker processes responsible for executing tasks in Spark applications. They are launched on worker nodes and communicate with the driver program and cluster manager. Executors run tasks concurrently and store data in memory or disk for caching and intermediate storage.

## **The cluster manager**

The cluster manager is responsible for allocating resources and managing the cluster on which the Spark application runs. Spark supports various cluster managers like Apache Mesos, Hadoop YARN, and standalone cluster manager.

## **sparkContext**

SparkContext is the entry point for any Spark functionality. It represents the connection to a Spark cluster and can be used to create RDDs (Resilient Distributed Datasets), accumulators, and broadcast variables. SparkContext also coordinates the execution of tasks.

## **Task**

A task is the smallest unit of work in Spark, representing a unit of computation that can be performed on a single partition of data. The driver program divides the Spark job into tasks and assigns them to the executor nodes for execution.