- Most programming languages like Scheme/Java/C++ are organized around computing the values of mathematical functions.

  All these langauges reinforce the idea that programming is about procedures.

- These programming languages are strongly biased toward unidirectional computations (i.e. computations with well-defined inputs and outputs).

- Most programming languages like Scheme/Java/C++ are organized around computing the values of mathematical functions.

  All these langauges reinforce the idea that programming is about procedures.

- These programming languages are strongly biased toward unidirectional computations (i.e. computations with well-defined inputs and outputs).

- Prolog is one such language that relaxes this bias.

- The Logic programming approach, when it works, can be a very powerful way to write programs.
- We shall look at Logic Programming with the help of Prolog which is a declarative language.

.

- The Logic programming approach, when it works, can be a very powerful way to write programs.

- We shall look at Logic Programming with the help of Prolog which is a declarative language.

  .

- We focus on what we are interested in stating.
  We express what is true about solutions we want to find.

- The Logic programming approach, when it works, can be a very powerful way to write programs.

- We shall look at Logic Programming with the help of Prolog which is a declarative language.

  .

- We focus on what we are interested in stating.
  We express what is true about solutions we want to find.

- We are less concerned about ~~how~~ the Prolog implementation finds these solutions.

There is a single language element, called a **Rule**.
A Rule is of the form:

Head :- Body

There is a single language element, called a **Rule**.
A Rule is of the form:

$$\textbf{Head :- Body}$$

The Rule is read as an implication.
If *Body* holds true, then *Head* holds true.

$$\textbf{Head} \Longleftarrow \textbf{Body}$$

$$\therefore \neg\, \textbf{Head} \Longrightarrow \neg\, \textbf{Body}$$

but

$$\therefore \neg\, \textbf{Head} \Longleftarrow \neg\, \textbf{Body}$$

is not a valid conclusion.

There is a single language element, called a **Rule**.
A Rule is of the form:

$$\textbf{Head :- Body}$$

The Rule is read as an implication.
If *Body* holds true, then *Head* holds true.

$$\textbf{Head} \Longleftarrow \textbf{Body}$$

$$\therefore \neg\, \textbf{Head} \Longrightarrow \neg\, \textbf{Body}$$

but

$$\therefore \neg\, \textbf{Head} \Longleftarrow \neg\, \textbf{Body}$$

is not a valid conclusion.

If *Head* always holds true, then *Body* can be omitted.

All programs which we shall look at consist only of such clauses.

A fact is written as:

Head.

This is equivalent to the rule:

Head :- true.

Logically, this means that the rule always holds, because the built-in predicate true/0 is always true.

The normal way of working is for the user to **load a program** written in the Prolog language and then **enter queries** at the prompt, to make use of the information that has been loaded into the database.

Prolog programs can be created in a text editor and saved as a text file with .pl extension.

Loading a program simply causes the clauses to be placed in a storage area called the Prolog database.

Entering a sequence of one or more goals in response to the system prompt causes Prolog to search for and use the clauses necessary to evaluate the goal(s).

A pure Prolog program consists of a set of **Horn clauses**.

A Horn clause is a clause (a disjunction of literals) with at most one positive, i.e. unnegated, literal.

$$(\neg p \vee \neg q \vee \ldots \vee \neg t \vee u)$$

$$(\neg(p \wedge q \wedge \ldots \wedge t) \vee u)$$

alternatively,

$$p \wedge q \wedge \ldots \wedge t \rightarrow u$$

Prolog program's execution can be regarded as a special case of resolution which is an algorithm that is rooted in formal logic.

Prolog programs are constructed from terms: constants, variables, or structures.

1. 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.

Prolog programs are constructed from terms: constants, variables, or structures.

1. 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.
2. Variables are strings of characters beginning with an uppercase letter or an underscore.

Prolog programs are constructed from terms: constants, variables, or structures.

1. 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.
2. Variables are strings of characters beginning with an uppercase letter or an underscore.
3. Structures consist of a functor or function symbol, which looks like an atom, followed by a list of terms inside parentheses, separated by commas.
   Structures can be interpreted as predicates (relations):
   likes(john,mary).
   male(john).
   sitsBetween(X,mary,helen).

A Prolog program is a sequence of statements - of the form $P_0 :- P_1, P_2, \ldots, P_n.$ where each of $P_1, P_2, \ldots, P_n$ is an atom or a structure.

- A clause can be read **declaratively** as
  • P_0 is true if P_1 and $P_2 \ldots P_n$ are true
  or **procedurally** as
  • To satisfy goal P_0, satisfy goal P_1 and then $P_2 \ldots$ and then P_n.

- A clause $P_0.$ without a body is a *unit* clause or fact and means P_0 is true/ goal P_0 is satisfied.

- A clause without a head,
  $:-P_1, P_2, \ldots, P_n.$
  is a *goal* clause or *query* and means
  - Are P_1 and P_2 and P_n true? or
  - Satisfy goal P_1 and then P_2 ... and then P_n.

- A clause without a head,
  $:-P_1, P_2, \ldots, P_n.$
  is a *goal* clause or *query* and means
  - Are P_1 and P_2 and P_n true? or
  - Satisfy goal P_1 and then P_2 ... and then P_n.

A Horn clause with exactly one literal in the head is called a strict-Horn clause or definite clause.
Logic programs mostly consist of definite clauses.

A period terminates every clause/program.

- A clause without a head,
  $:-P_1, P_2, \ldots, P_n.$
  is a *goal* clause or *query* and means
  - Are P_1 and P_2 and P_n true? or
  - Satisfy goal P_1 and then P_2 ... and then P_n.

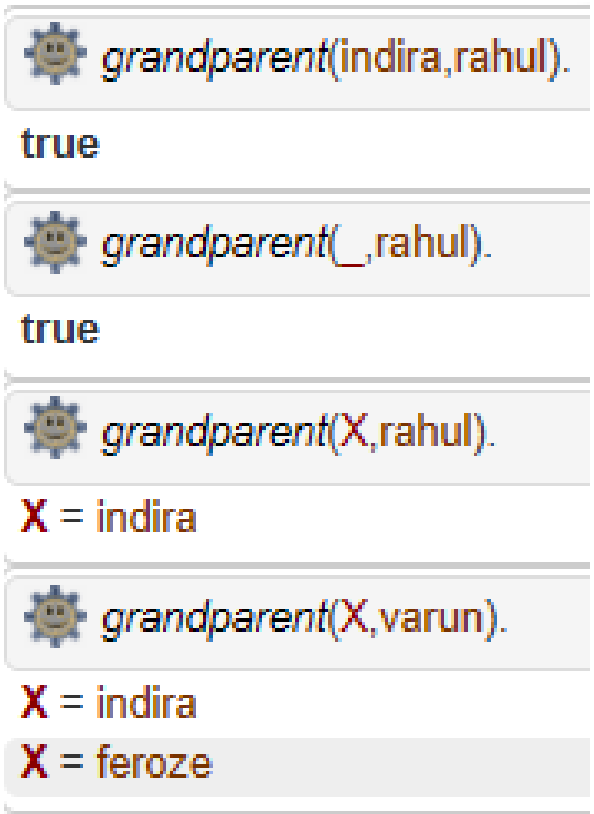
A Horn clause with exactly one literal in the head is called a strict-Horn clause or definite clause.
Logic programs mostly consist of definite clauses.

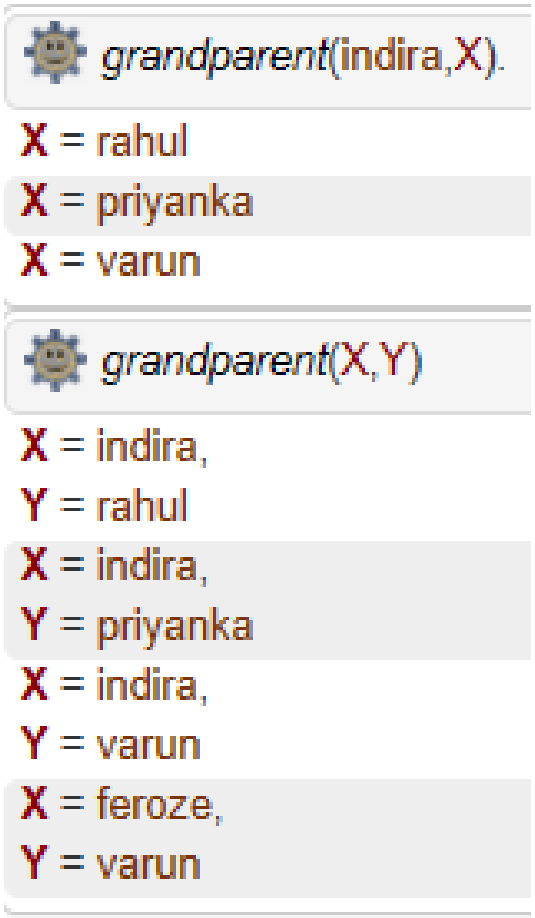
A period terminates every clause/program.

Logic programming emphasizes on relations and how
a single "what is" fact can be used to solve a number of different problems that would have different "how to" components.

```
% Facts
% Facts consist of atoms which start
% with lowercase letters.

parent(indira,rajiv).
parent(indira,sanjay).
parent(sanjay,varun).
parent(rajiv,rahul).
parent(rajiv,priyanka).
parent(feroze,sanjay).

% Rules
% variables are atoms inside rules which start
% with uppercase letters or underscore.

grandparent(X,Y):-
            parent(X,Zee),parent(Zee,Y).
```

The first six clauses are given as facts which are known to be true. The grandparent rule helps us infer relationships based on the facts provided.





Based on these facts we can formulate several queries and in logic programming, we can answer several such queries.

There are two big differences

- We are not going to be computing functions. We are not going to talk about things to take input and give output. We are talking about relations (akin to an equation). That means in principles, these relations do not have directionality. So the knowledge you specify can be used to answer different questions.
- These relations do not necessarily have one answer. It may return a whole bunch of answers.

The aim of logic programming is to

1. Use logic to express what is true.
2. Use logic to check whether something is true.
3. Use logic to find out what is true.

All known computations can be described in terms of such clauses, making Prolog a Turing-Complete programming language.

One way to implement a Turing machine in Prolog is to describe the relation between different states of the machine with *clauses* of the form

"If the current state is S_0 and the symbol under the tape head is T, and ... then the next state is S".

One page Prolog emulators of a Turing machine exist.

When introduced to a new language one must first asks these questions

1. What are the primitives ?

When introduced to a new language one must first asks these questions

1. What are the primitives ?
   Query

When introduced to a new language one must first asks these questions

1. What are the primitives ?
   Query
2. What are the means of combination ?

When introduced to a new language one must first asks these questions

1. What are the primitives ?
   Query
2. What are the means of combination ?
   And denoted by **,**
   Or denoted by **;**
   Not denoted by **not**

When introduced to a new language one must first asks these questions

1. What are the primitives ?
   Query
2. What are the means of combination ?
   And denoted by **,**
   Or denoted by **;**
   Not denoted by **not**
3. What are the means of abstraction ?

When introduced to a new language one must first asks these questions

1. What are the primitives ?
   Query
2. What are the means of combination ?
   And denoted by **,**
   Or denoted by **;**
   Not denoted by **not**
3. What are the means of abstraction ?
   Rules

When introduced to a new language one must first asks these questions

1. What are the primitives ?
   Query
2. What are the means of combination ?
   And denoted by **,**
   Or denoted by **;**
   Not denoted by **not**
3. What are the means of abstraction ?
   Rules