ARTIFICIAL INTELLIGENCE

Unit 6 Prolog

Parul® University

CHAPTER-6

Prolog

What is Prolog ?

- Prolog or **PROgramming in LOGics** is a logical and **declarative** programming language. . In artificial intelligence applications,prolog is used.
- In the logic programming paradigm, prolog language is most widely available.
- Prolog is a declarative language, which means that a program consists of data based on the facts and rules (Logical relationship).
- Which also means we can specify what problem we want to solve rather than how to solve it.
- A logical relationship describes the relationships which hold for the given application.

Applications of Prolog

- Specification Language
- Robot Planning
- Natural language understanding
- Machine Learning
- Problem Solving
- Intelligent Database retrieval
- Expert System
- Automated Reasoning

Applications of Prolog

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.

Elements of Prolog

- 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/Queries –** And to run a prolog program, we need some questions, and those questions can be answered by the given facts and rules.

Prolog Syntax - Symbols

- Using the following truth-functional symbols, the Prolog expressions are comprised. These symbols have the same interpretation as in the predicate calculus.

English	Predicate Calculus	Prolog
If	-->	:
Not	~	Not
Or	V	;
and	^	,

Variable

- Variable is a string. The string can be a combination of lower case or upper case letters.
- The string can also contain underscore characters that begin with an underscore or an upper-case letter.
- Rules for forming names and predicate calculus are the same.
- **For example:**

➢ female
➢ Male
➢ X
➢ y
➢ mother_of
➢ _father
➢ Pro34

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 –

1. Tom is a cat
2. Kunal loves to eat Pasta
3. Hair is black
4. Nawaz loves to play games
5. Pratyusha is lazy.

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

Facts

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.

Facts

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 –

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

Rules

- 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.

Rules

- 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.

Rules

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 –

1. Is tom a cat?

2. Does Kunal love to eat pasta?

3. Is Lili happy?

4. Will Ryan go to play?

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

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 eating	eating(dog)
Jay likes Pasta if it is delicious.	likes(Jay, Pasta):-delicious(Pasta)

- In the above table, the statement 'Dog is eating' is a fact, while the statement 'Jay likes pasta 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 “Jay likes delicious Pasta”.

Arithmetic Operations in Prolog

- As per the requirement of the user, arithmetic operations can be divided into some special purpose integer predicates and a series of general predicates for integer, floating point and rational arithmetic.
- The general arithmetic predicates are handled by the expressions.
- An expression is either a function or a simple number.
- Prolog arithmetic is slightly different than other programming languages.
- **For example:**

```
?- X is  2 + 1.
X = 3 ?
yes
```

- In the above example, 'is' is used as a special predefined operator.

Arithmetic Operations in Prolog

Sr. No	Operator	Explanation
1	X+Y	The sum of 'X' and 'Y' ?- X is 2 + 1. X=3.
2	X-Y	the difference of 'X' and 'Y'
3	X*Y	The product of 'X' and 'Y'
4	X/Y	The quotient of 'X' and 'Y'
5	X^Y	'X' to the power of 'Y'
6	-X	Negation of 'X'
7	abs(X)	Absolute value of 'X'
8	sqrt(X)	The square root of X ?- X is sqrt(3).
9	sin(X)	The sine of X

If-Then-Else

- In prolog , if A then B else C is written as (A->B;C).
- To prolog this means : Try A. If you can prove it , go to prove B and ignore C.
- If A fails however go to prove C ignoring B.
- The max predicate using the if-then-else construct looks like as follows:

```
Max(X,Y,Z):
(X=<Y
-> Z=Y
; Z=X
).
```

Parul® University

Backtracking in Prolog

How does it work?

- It starts by trying to solve each goal in a query, left to right. Recall that goals are connected by “,”,which is the and operator.
- For each goal, it tries to match a fact or the head of a corresponding rule.
- If a fact or a head matches, it goes on to match any remaining goals.
- But what shall we do if we reach a point where a goal cannot be matched?
- **Prolog uses backtracking.**
- When we reach a point where a goal cannot be matched, we backtrack to the most
- recent spot where a choice of matching a particular fact or rule was made.
- We try to match a different fact or rule. If this fails, we go back to the next
- previous point where a choice was made, and try a different match there.
- We try alternatives until we are able to solve all the goals in the query or until all
- possible choices have been tried and found to fail.

Backtracking in Prolog

Example:

Consider the following facts,

bird(type (sparrow) name (steve)))
bird(type (penguin) name (sweety)))
bird(type (penguin)name (jones)))

consider the following query:

?- bird(type (penguin)name(X)

So, prolog will try to match the first query, but this query will not match because sparrow doesn't match with penguin. Then, it will try to find next query to match the fact and succeed with X = sweety. Later, if the query or subgoals are failed, it will go to the saved option and look for more solutions. For example: X = jones

Backtracking in Prolog

- We can also have backtracking in rules. For example consider the following program.
- hold_party(X):- •
- birthday(X), happy(X).
- birthday(tom).
- birthday(fred). •
- birthday(helen).
- happy(mary).
-
- happy(jane).
- happy(helen).

If we now pose the query • • ?- hold_party(Who).

In order to solve the above, Prolog first attempts to find a clause of birthday, it being the first subgoal of birthday. This binds X to tom. We then attempt the goal happy(tom). This will fail, since it doesn't match the above database.

As a result, Prolog backtracks. This means that Prolog goes back to its last choice point and sees if there is an alternative solution. In this case, this means going back and attempting to find another clause of birthday. This time we can use clause two, binding X to fred. This then causes us to try the goal happy(fred). Again this will fail to match our database.

As a result, we backtrack again. This time we find clause three of birthday, and bind X to helen, and attempt the goal happy(helen).

This goal matches against clause 3 of our happy database. As a result, hold_party will succeed with X=helen.

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.

- **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**.

Lists

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 | [ ]]

Basic Operations on 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.

Misc. Operations on Lists :

Misc Operations	Definition
Even and Odd Length Finding	Verifies whether the list has odd number or even number of elements.
Divide	Divides a list into two lists, and these lists are of approximately same length.
Max	Retrieves the element with maximum value from the given list.
Sum	Returns the sum of elements of the given list.
Merge Sort	Arranges the elements of a given list in order (using Merge Sort algorithm).

Cuts

- A cut prunes or “cuts out” and unexplored part of a Prolog search tree.
- Cuts can therefore be used to make a computation more efficient by eliminating futile searching and backtracking.
- Cuts can also be used to implement a form of negation

Cuts

- A cut, written as !, appears as a condition within a rule. When rule
- `B :- C`$_1$`,…, C`$_{j-1}$`, !,C`$_{j+1}$`,…,C`$_k$
- is applied, the cut tells control to backtrack past Cj-1,…,C1,B, without considering any remaining rules for them.

Cuts

- A cut prunes or “cuts out” and unexplored part of a Prolog search tree.
- Automatic backtracking is one of the most characteristic features of prolog.
- There is an in built prolog predicate ! , called cut , which offers more direct way of exercising control over the way prolog looks for solutions.
- Cuts can therefore be used to make a computation more efficient by eliminating futile searching and backtracking.
- Cuts can also be used to implement a form of negation.
- Cut is simply a special atom that we can use when writing clauses.
- Ex. : p(X) :- b(X),c(X),!,d(X),e(X).

Cut in Rules

- In practice, the cut is used in rules rather than in multi-goal queries, and some particular idioms apply in such cases.
- For example, consider the following code for max1(X, Y, Max), which is supposed to bind Max to the larger of X and Y, which are assumed to be numbers (see note [1] below).

```
max1(X, Y, X) :- X > Y, !.
max1(_X, Y, Y). % See note [2]
```

- This is a way of saying: "if the first rule succeeds, use it and don't try the second rule. (Otherwise, use the second rule.)

Cut in Rules

- We could instead have written:
  max2(X, Y, X) :- X > Y. max2 (X,
  Y, Y): - X = <Y.

- in which case both rules will often be tried (unless backtracking is prevented by a cut in some other part of the code).
- This is slightly less efficient if X is in fact greater than Y (unnecessary backtracking occurs) but easier for people to understand, though regular Prolog programmers rapidly get to recognize this type of idiom.
- The extra computation in the case of max2 is trivial, but in cases where the second rule involves a long computation, there might be a strong argument for using the cut on efficiency grounds.

Pattern Matching

- The way in which Prolog matches two terms is called unification.
- The idea is similar to that of unification in logic: we have two terms and we want to see if they can be made to represent the same structure.
- For example, we might have in our database the single Prolog clause:

- parent(alan, clive).
- and give the query:
- |?- parent(X,Y).
- We would expect X to be instantiated to alan and Y to be instantiated to clive when the query succeeds. We would say that the term parent(X,Y) unifies with the term parent(alan, clive) with X bound to alan and Y bound to clive. The unification algorithm in Prolog is roughly this.

Pattern Matching

- df:un Given two terms and which are to be unified:

- If T1 and T2 are constants (i.e. atoms or numbers) then if they are the same succeed. Otherwise fail.
- If T1 is a variable then instantiate T1 to T2 .
- Otherwise, If T2 is a variable then instantiate T2 to T1.
- Otherwise, if T1 and T2 are complex terms with the same arity (number of arguments), find the principal function F1 of T1 and principal function F2 of T2. If these are the same, then take the ordered set of arguments of arguments (A1,…,AN) of T1 and the ordered set of arguments (B1,…,BN) of T2. For each pair of arguments AM and BM from the same position in the term, AM must unify with BM.
- Otherwise fail.

Pattern Matching

- For example: applying this procedure to unify foo(a,X) with foo(Y,b) we get:

- foo(a,X) and foo(Y,b) are complex terms with the same arity (2).
- The principal functor of both terms is foo.
- The arguments (in order) of foo(a,X) are a and X.
- The arguments (in order) of foo(Y,b) are Y and b.
- So a and Y must unify , and X and b must unify.
- Y is a variable so we instantiate Y to a.
- X is a variable so we instantiate X to b.
- The resulting term, after unification is foo(a,b).

Pattern Matching

- The built in Prolog operator '=' can be used to unify two terms. Below are some examples of its use. Annotations are between ** symbols.

```
| ?- a = a.      ** Two identical atoms unify **
yes
| ?- a = b.      ** Atoms don't unify if they aren't identical **
no
| ?- X = a.      ** Unification instantiates a variable to an atom **
   X=a
yes
```

Pattern Matching

```
•  | ?- X = Y.         ** Unification binds two differently named variables **
•      X=_125451      ** to a single, unique variable name **
•      Y=_125451
•  yes
•  | ?- foo(a,b) = foo(a,b).      ** Two identical complex terms unify **
•  yes
•  | ?- foo(a,b) = foo(X,Y).      ** Two complex terms unify if they are **
•      X=a                ** of the same arity, have the same principal**
•      Y=b                ** functor and their arguments unify **
•  yes
•  | ?- foo(a,Y) = foo(X,b).      ** Instantiation of variables may occur **
•      Y=b                ** in either of the terms to be unified **
•      X=a
•  yes
```

DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in