Logic programming

UNIT-6

Logic programming is an approach to express programs in a form of symbolic logic and use a logic inferencing process to produce results.

Programs in logic programming languages are collections of facts and rules.

Programming that uses a form of symbolic logic as a programming language is often called logic programming and languages based on symbolic logic are called logic programming language or declarative languages.

Basics of Formal Logic

A proposition can be thought of a logical statement that may or may not be true. It consists of objects and the relationship among objects.

Symbolic logic can be used for the three basic needs of formal logic:

1) To express propositions.
2) To express the relationships between propositions.
3) To describe how new propositions can be inferred from other propositions that are assumed to be true.

The particular form of symbolic logic that is used for logic programming is called first-order predicate calculus.

propositions

- The objects in logic programming, propositions are represented by simple terms, which are either constants or variables.
- A constant is a symbol that represents an object.
- A variable is a symbol that can represent different objects at different times.
- The simplest propositions, which are called atomic propositions, consist of compound terms.
- A compound term is composed of two parts: a functor, which is the function symbol that names the relation and an ordered list of parameters, which represent an element of the relation.

man (Jake) - 1-tuple compound term

like (bob, icecream) - 2-tuple compound term.

man (fred)

- All the simple terms in these propositions i.e man, jake, like, bob, icecream, fred are all constants.

- Propositions can be stated in two modes

1) one in which the proposition is defined to be true.
2) one in which that truth of the proposition is to be determined.

- Compound propositions have two or more atomic propositions, which are connected by logical connectors or operators.
- The various logical connectors as follows:

Name	Symbol	Example	Meaning
negation	$\neg$	$\neg a$	not a
conjunction	$\cap$	$a \cap b$	a and b
disjunction	$\cup$	$a \cup b$	a or b
equivalence	$\equiv$	$a \equiv b$	a is equivalent to b
implication	$\supset$	$a \supset b$	a implies b
	$\subset$	$a \subset b$	b implies a

→ $\neg$ has high precendce over $\cap, \cup, \equiv$, which have high precedence over $\supset, \subset$.

Examples

$a \cap b \supset c$

$a \cap \neg b \supset d \equiv ((a \cap (\neg b)) \supset d)$

→ Variables can appear in propositions but only when introduced by special symbols called quantifiers.

- predicate calculus includes two quantifiers:

Name	Example	Meaning
universal	$\forall x.P$	For all x, P is true.
Existential	$\exists x.P$	There exists a value of x, such that P is true.

Example

1) $\forall x.(woman(x)) \supset human(x)$

- For any value of x, if x is woman, then x is a human.

2) $\exists x.\ (\text{mother}(\text{mary}, x) \cap \text{male}(x))$

- This means that there exists a value of x, such that mary is the mother of x and x is a male; in other words mary has a son.

Clausal Form

- one of the problems with predicate calculus is that, there are too many different ways of stating propositions that have the same meaning; i.e there is a great deal of redundancy.

- Clausal form, which is relatively simple form of propositions, which reduces this redundancy.

- A proposition in a clausal form has the following general syntax:

$$B_1 \cup B_2 \cup \ldots B_n \subset A_1 \cap A_2 \cap \ldots \cap A_m.$$

in which A's and B's are terms.

- The meaning of this clausal form proposition is that if all of the A's are true, then at least one B is true.

- In clausal form

1) Existential quantifiers are not required.

2) No operations other than conjunction and disjunction are required.

3) conjunction and disjunction need to appear only in the order shown in the general clausal form ie disjunctions on the left side and conjunctions on the right side.

- The right side of a clausal form proposition is called the antecedent. The left side is called the consequent.

Ex:- 1) likes(bob, trout) ⊂ likes(bob, fish) ∩ fish(trout)

→ if bob like fish and trout is a fish, then bob likes trout.

2) father(louis, al) ∪ father(louis, violet) ⊂

father(al, bob) ∩ mother(violet, bob) ∩ grandfather(louis, bob)

- if al is a bob's father and violet is bob's mother and louis is bob's grandfather then louis is either father of al or violet.

Imperative programming languages Vs Declarative programming languages

Imperative PL	Declarative PL
1) Imperative PL are of two types (i) procedural oriented (ii) Object-oriented	1) Declarative PL are of two types (i) Functional PL (ii) Logical PL
2) Ex:- FORTRAN, Ada, C, C++, java, python, C#	2) Eg:- LISP, Scheme, ML, Haskell, Ruby, prolog, SQL etc.
3) In imperative PL, the solution is based on algorithm or commands	3) In case of declarative, the solution is based on queries.
4) The main objective of imperative PL is how to solve the given problem based on algorithm.	4) The main objective of declarative PL is what to do with the given problem.

5) These languages severialy suffer from side effects due to variable declarations and assignment operators.	5) These languages does not suffer from sideeffects due to lack of variable declarations.
6) These languages supports concepts like iterations & Recursion	6) Majority of these PL's support only recursion.

Resolution

- Resolution is an inference rule that allows new propositions to be computed from given propositions.
- Resolution was devised to be applied to propositions in clausal form.
- The concept of resolution is the following:
  - Suppose there are two propositions with the forms.

$$P_2 \rightarrow P_1$$

$$Q_2 \rightarrow Q_1$$

  - Suppose that P_1 is identical to Q_2, So we could rename P_1 and Q_2 as T. Then we could rewrite the two propositions as.

$$P_2 \rightarrow T$$

$$T \rightarrow Q_1$$

  - It is logically obvious that P_2 implies Q_1

$$P_2 \rightarrow Q_1$$

- The process of inferring this proposition from the original two propositions is resolution.

Overview of Logic programming

- Languages used for logic programming are called declarative languages, because programs written in these consist of declarations rather than assignments and control flow statements.
- These declarations are actually statements or propositions in symbolic logic.
- The semantic of logic programming language is simpler compared to imperative languages.
    - For example, the meaning of a proposition is determined from the statement itself but in imperative languages, the semantics of simple assignment statment needs understanding of local declaration, scoping rules etc.
- programming in both imperative and fonctional languages is procedural (steps), which means that the programmer knows what is to accomplished by a program and instructs the computer on exactly how the computation is to be done.

programming in logic programming is non procedural, programs don't state exactly how a result is computed but rather describe the form of the result. Here we assume the computer system can somehow determine how the result is to be computed.

- Logic programming languages need relevant information about the problem and a method of inference for computing desired results.

Predicate calculus supplies the basic information to the computer and resolution provides the inference technique.

Programming with Prolog

- Prolog is one type of logic programming that uses predicate logic and solves the given problem based on queries.
- prolog programs consists of collection of statements.
- All prolog statements, as well as prolog data are constructed from terms.
- A prolog term is a constant, a variable or a structure.
- A constant is either an atom or an integer.
  - Atom is either a string of letters, digits and underscores.
- A variable is any string of letters, digits and underscores
- The last kind of term is called a structure. Structures represent the atomic propositions of predicate calculus, and their general form is

  functor(parameter list).

Facts, Rules

- prolog contains sequence of facts and rules.
- Fact is simply a proposition that is assumed to be true.

```
father(James, Robert)
father(mike, williams)
father(williams, James)
father(Robert, Henry)
```

→ These are called headless horn clauses (or) unit clauses.

- Rules are the clausal form statements, of the form

  consequence :- antecedent_expression.

- Rules are also called headed horn clauses due to the presence of left hand side consequents.

- Example parent(z,y) :- father(z,y)

grandfather(x,y) :- father(x,z), parent(z,y)

- connectors are placed using the following symbols

operators	Symbol
if	:-
AND	,
OR	;
NOT	not

Goal / queries

- A goal statement is one, which requests an answer.
- The syntax form of fact statement and goal statement must be identical otherwise solving the problem is highly impossible.
- Rules and facts in prolog are always terminated using period symbol (.)
- Goal can be a single or combination of subgoals terminated by period symbol.

Ex: likes(A, B).
likes(B, A).
likes(C, D).

dating(X,Y) :- likes(x,Y), likes(Y,x).

friendship(x,Y) :- likes(x,Y) ; likes(Y,x).

Run - ['location/test.pl']
?- likes(A,B).
true

?- dating(A, B)

love

2. Friendship (B C)

false.

⇒ Variables names are defined as same as predicate names, but it starts with upper case letters only.

- Variables are just place holders, they donot have any memory in the compoter.
- Variables allow us to ask more complex queries.

Ex: /* weather (City, Season, Temp).*/

weather (phoenix, Sommer, hot).

weather (la, summer, warm).

weather (phoenix, winter, warm).

?- weather (City, Summer, hot). /* @All the cities hot in Sommer*/

city - phoenix

?- weather (City, _, warm). /* All the cities which are warm*/.

city - la

city - phoenix

Unification/matching

- It is a process of matching or finding the most general unifier (MGU).
- A constant matches with another constant.
- A variable can match with other constant or any other variable.

-This process determines specific instances, which is called instantiation (assigning a value to variable), that can be made available during series of resolutions.

Ex:- ?- weather (City, Summer, hot)

City - phonix.

- weather (City, Summer, hot) unified with weather (phoenix, Summer, hot).

Ex 2:- weather (phoenix, hot, Summer).

weather (la, warm, Summer).

warmer_than (C1, C2) :-

weather (C1, hot, Summer),

weather (C2, warm, summer),

write (C1), write (' is warmer than'), write (C2), nl.

?- warmer_than (phoenix, la).

phoenix is warmer than la

true.

Structs

Ex:- course (ItA, mon, Tue, 11, 12, 11, 12, Ravi, Kishore, LSFQ).

course (ItA,

day (mon, Tue),

time (11, 12).

prof (Ravi, Kishore).

LSFQ).

Programming paradigms

Unit - 1 PPL

A programming paradigm is a style or way of programming.

When a language implements only one paradigms, we call its pure and when a language implements more than one paradigms, we call its multi-paradigm languages.

programming paradigms provides a way to classify programming languages based on their features.

There are several kinds of major programming paradigms:

1) Imperative - C, C++, Java
2) Functional - C, Haskell
3) object-oriented - Java, Small talk
4) Logical - prolog

Imperative programming

Imperative programming is a paradigm that explicitly tells the computer what to do and how to do it. They allow side effects.

The essential aspects of imperative programming are sequenced instructions and mutable data.

- The most prominent to this paradigm is the sequential flow of instructions and branching.

```
var a = 5;
var b = calculate(a);
if (b > 10) {
    printf ("Too big");
    return;
}
```

- code lines are executed one at a time from top to bottom.

- The sequencing of statements is dependent on mutable variables (variables that can change).
  - variables represent a storage location, not just a value.
- Imperative language has a direct concept of memory.
- In this we write statements that changes the current state of program. These are referred to as side effects.
- popular programming languages are imperative because of the following reasons:
  1) The imperative paradigm closely resembles the actual machine itself, so the programmer is much closer to the machine.
  2) Due to its closeness to the machine, it is highly efficient.

Advantages
- Efficient
- Close to machine.
- familiar

Disadvantages
- Side effects makes debugging harder.
- The semantics of a program can be complex to understand (due to side effects).
- Abstraction is more limited.
- order of statements is crucial, which doesn't always suit the problems.

Functional programming
- It constitutes subprograms, that take-in arguments and returns a solution.
- The solution is based solely entirely on the input, and the time at

which a function is called has no relevance.

Advantages

1) It provides high level of abstraction.
2) It reduces the dependency on assignment operators.
3) It allows programs to be evaluated in many different orders. This evaluation order independence makes function-oriented languages a good candidate to program massive parallel computers.

Disadvantages

1) Less efficient.
2) problems involving many variables and sequential activity are easier to be handled by imperative or object-oriented languages.

Object-oriented programming

- oop is a paradigm in which real-world objects are each viewed as seperate entities having their own state which is modified only by built in procedures called methods.
- objects are organized into classes.
- The object-oriented paradigm provides key benefits of reusable code and code extensibility.
- Encapsulation and information hiding are other benefits of oop.

Logic programming

- It takes declarative approach to problem-solving. [what to do, rather than how to do].
- A logical program is divided into three sections:
    1) A series of definitions/declarations that define the problem domain.
    2) Statements of relevant facts.
    3) Statement of goals in the form of a query.

Advantages

- The system solves the problem, so programming themselves are kept to minimum.

Side effect

- A function or expression is said to have sideeffect if it modifies some state outside its local environment.
- In the presence of side effects, a programs behaviour may depend on history i.e the previous steps. So the order of evaluation matters.

The inferencing process of prolog

The queries are called goals.

When a goal is a compound proposition, each of the facts is called a subgoal.

To prove that a goal is true, the inferencing process must find a chain of inference rules or facts in the database that connect the goal to one or more facts in the database.

The process of proving a subgoal is done through a proposition matching process called unification or matching.

Ex: Consider the following query

man(bob).

- This goal statement is very simple. It is easy to resolve and determine whether it is true or false.
- The pattern of this goal is compared with the facts and rules in the database.
- If the database includes the fact

  man(bob).

  then the proof is true.

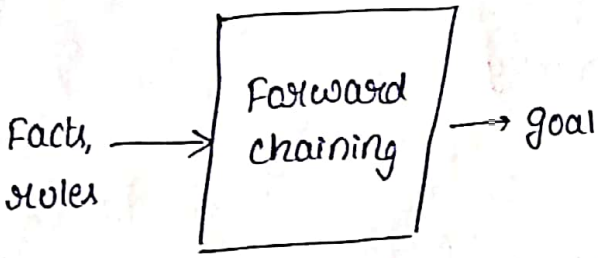
- But if the database contains the following fact and inference rule,

  father(bob).

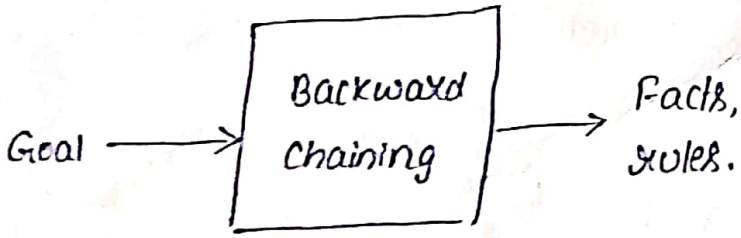
  man(x) :- father(x).

  - In this case prolog must match the goal against the propositions in the database.

- There are two opposite approaches to attempting to match a given goal or fact in the database.
- The system can be either use:
    1) Bottom-up resolution or Forward chaining.
    2) Top-down resolution or Backward chaining.
- In Forward chaining, the system begins with the facts and rules of the database and attempt to find a sequence of matches that lead to the goal.

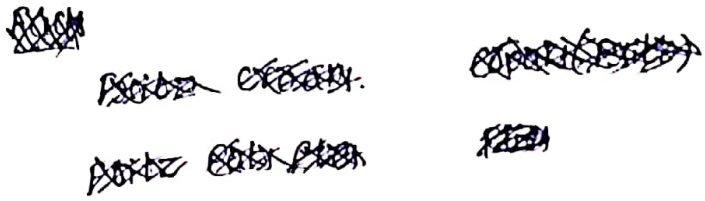


- In Backward chaining, the system begins with the goal and attempt to find a sequence of matching propositions that lead to the set of original facts in the database.



- The forward chaining is a better approach, when the number of possible correct answers is large, and backward chaining works well with small number of correct answers.

- prolog implementations use backward chaining for resolution. as backward chaining is more suitable for a larger class of problems.
- The following example depicts the forward and backward chaining processes:



Database

father (bob).

man (x) :- father (x).

Goal/query

man (bob).

Forward Chaining

man (x) :- father (x).

man (bob) :- father (bob) instantiation of x with bob.

man (bob) ✓

Backward chaining

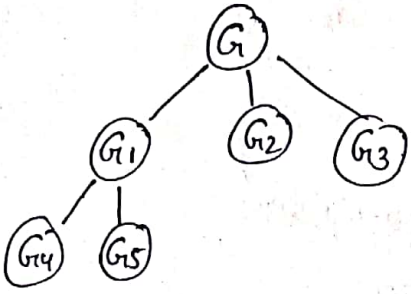
man (bob).

man (bob) :- father (bob)

man (x) :- father (x) ✓

- The next design question arises whenever the goal has more than one structure.
  - The question is whether the solution search is done depth first or breadth - first.

- A depth first search finds a match or satisfy the first subgoal, before working on the other subgoals.



- First subgoal G_1 is satisfied, then it goes for G_2 then G_3.

- In breadth-first search, all subgoals of a given goal are searched in parallel. The breadth-first approach requires a large amount of memory and processing.

- G_1, G_2, G_3 are searched in parallel at the same time.
- prolog uses depth-first search as it uses less computational resources.

Backtracking

- when a goal has multiple subgoals and when one of these subgoals is being processed and the system fails to show the truth of this subgoal, then the system abondens this subgoals and searches for the alternative solution to it.

- The backing up in the goal to the reconsideration of a previously proven subgoal is called backtracking.

- Backtracking requires a great deal of time and space because it may have to find all possible proofs to every subgoal.

Simple arithmetic

- prolog supports integer variables and integer arithmetic.
- originally the arithmetic operators were functors, so that the sum of 7 and the variable X was formed as:

```
+(7, X).
```

prolog now allows arithmetic by using "is" operator.
This operator takes an arithmetic expression as its right operand and a variable as its left operand.

```
A is B/17+C.
```

⇒ All the variables must be already instantiated, but left side variable cannot be previously instantiated.

- B and C must be instantiated, but A is not instantiated, then the above statement will cause A to be instantiated with the value of the expression "B/17+C".
- If the left side is instantiated before, the statement will fail.

- So, statements like

```
Sum is Sum + Number.
```

are not legal in prolog.

- Example

- Suppose we know the average speeds of several cars on a particular racetrack.

- Also the amount of time they spent on the track. Then distance rule is as follows:

```
speed(ford, 100).
speed(chevy, 100).
speed(dodge, 95).
speed(volvo, 80).
time(ford, 20).
time(chevy, 21).
time(dodge, 24).
time(volvo, 24).
distance(X, Y) :- speed(X, S),
                  time(X, T),
                  Y is S*T.
```

- Now queries can request the distance travelled by a particular car.

```
?- distance(chevy, chevy_distance).
chevy_distance = 2205
```

Tracing model

- prolog has a built-in structure named trace that displays the instantiations of values to variables at each step during the attempt to satisfy a given goal.
- Trace is used to understand and debug prolog programs
- Tracing model is a model of execution of prolog programs.

Tracing model has four events:

(1) call, which occurs at each step during the attempt to satisfy a given goal.

exit, which occurs when a goal has been satisfied.

3) redo, which occurs when a backtrack occurs.

4) fail, which occurs when a goal fails.

Example

trace.

distance (chevy, chevy_distance).

(1) 1 Call : distance(chevy, _0) ?

(2) 2 call : speed (chevy, _5) ?

(2) 2 Exit : speed (chevy, 105) ?

(3) 2 call : time (chevy, _6) ?

(3) 2 Exit : time (chevy, 21).

(4) 2 call : _0 is 105 * 21 ?

(4) 2 Exit : 2205 is 105 * 21

(1) 1 Exit : distance (chevy, 2205)

Subgoals

chevy_distance = 2205.

- Symbols in the trace that begin with the underscore character (_) are internal variables used to store instantiated values.

List Structures

- one of the data structure supported by prolog is list.
- Lists are sequences of any number of elements, where the elements can be atoms, atomic propositions of any other terms, including other lists.

- The list elements are separated by commas and the entire list is delimited by square brackets.

[apple, prune, grape, kumquat].

- The notation [] is used to denote the empty list.
- A list can be created with a simple structure

new_list ([apple, prune, grape, kumquat]).

- The elements of list can be divided into two parts : Head and Tail, by using '|' symbol

[apple, prune, grape, kumquat].

Examples

① ?- [Head | Tail] = [Apple, prune, grape, kumquat].

Head = Apple

Tail = prune, grape, kumquat.

2) Return more than one elements as head

?- [X, Y. | W] = [[], ryan, John, eat(cheese)].

X = []

Y = ryan

W = John, eat(cheese).

3) Empty lists cannot have head | tail

?- [Head | Tail] = [].

False.

) Nested lists

?- [_, _, [_ | X] _] = [[], dead(x), [2, [b, c], [], z, [2, [b, c]]].

X = [b, c].

5) A built in function-like operation called member, which results in finding wether a item exists in the list or not.

-It has the following syntax:

member (X, [X, T]).

member (X, [H, T]) :-

member (X, T).

?- member (Ian, [Ian, Josh, Judy]).

true.

?- member (Judy, [Ian, Josh, Judy]).

true.

6) Find the list of squares whose square value is less than 400 from the given list.

?- member [X, [23,25,67,12,12,25,19,9,6], Y is X * X, Y<400).

X = 12
Y = 144
X = 19
Y = 361
X = 9
Y = 81
X = 6
Y = 36.

Prolog programming techniques are divided into two type

1) Recursions
2) Iterations.

Recursion

- Recursion is one of the most important execution control technique to implement any type of logic.

Example

- calculate factorial of a number using recursion.

```
fact(0,1)
fact(N,R):-
 (
    N>0 ->
    (
      N1 is N-1,
      fact(N1,R1),
       R is N * R1
    );
   write('N should be greater than 0').
 ).
?- fact(5,R).
```

R = 120

Iteration

- prolog doesn't Iteration directly. It can be achieved indirectly by
  (i) making recursive call to be the last subgoal in the rule.
  (ii) using the concept of accumulators, which store the intermediate values.

- prolog programming language introduced accumulator concept to hold the intermediate values. These are similar to local variables in C, C++ & Java.
- In iterative form of programs, make sure the recursive call as the last call.
- In the following example, accumulator I and T are used for storing counter value and intermediate computations.

```
fact(N,R) :-
        fact1(N,R,0,1).

fact1(N,R,I,T) :-
        I<N ->
            J is I+1,
            T1 is J*T,
            fact1(N,R,J,T1).
        fact(N,R,N,T).
```

⇒ N=3

fact(3,R)

(1)

fact1(3,R,0,1)

(0)

~~fact1(3,R,0,1)~~ 0<3, J is 1, T1 is 1*1, fact1(3,R,1,1)

↓ (1)

1<3, J is 2, T1 is 2*1, fact1(3,R,2,2)

↓ (2)

2<3, J is 3, T1 is 3*2, fact1(3,R,3,6)

↓ (3)

R = 6

Deficiencies of prolog

- Although prolog is a useful tool, it is neither a pure nor a perfect logic programming language.
- The various deficiencies of prolog are:

1) Resolution order control
2) Closed world assumption.
3) The negation problem.

Resolution order control

- prolog allows the user to control the ordering of pattern matching during resolution for increasing the efficiency.
- In a pure logic programming language, the order should never be deterministic. So, as a result prolog is not a pure logic programming language.
- prolog allows users to control the ordering of rules in the database. so that most important rules are placed first, in the database.
- This also allows control of backtracking.
- This ability of the programmer to tamper with control flow in a prolog program is a deficiency, because it is directly affects one of the important advantages of logic programming - that programs do not specify how solutions are to be found.

The closed-world Assumption

- The nature of prolog's resolution sometime creates misleading results.

- The truth's ~~of~~ prolog is concerned, are those that can be proved using its database.
- It has no knowledge about the world outside the database.
- So prolog is a true/fail system, rather than true/false system.

The Negation problem

- Another problem with prolog is its difficulty to deal with negation.
  - consider the following database

```
parent(bill, jake).
parent(bill, shelley).
sibiling(x,Y) :- parent(M, x), parent(M,Y).
```

  - Now suppose we type a query

```
sibiling(x, Y).
```

prolog will respond with

```
X = jake.
Y = jake.
```

- To avoid this result, the rules have to state the x and Y have same parent and x and Y are not same.
- But stating that they are not equal is not easy in prolog. This is called the negation problem.

Applications of Logic programming

- The important applications of logic programming include:
  1) Relational database management systems

RDBMS

- RDBMS store data in the form of tables and queries are run on these tables using structured query language (SQL).
- Tables of information can be described using prolog structures and relationships between tables can be described by using rules. The queries as goal statements in prolog.
- one more advantage is that a single language is used to implement RDBMS.
- But the disadvantage of implementing RDMS with logical programming is that it is slower.

Natural language processing

- certain kinds of natural-language processing can be done with logic programming.
- The language syntax, parsing and semantics can be implemented effectively with logic programming.

Expert Systems

- Expert systems are computer systems designed to imitate human expertise in some particular domain.
- prolog can be used to construct expert systems.

Multi-paradigm languages

- Most programming languages support more than one programming paradigm to allow programmers to use the most suitable programming style and associated language constructs for a given job.

Example

C++ → Functional,
Imperative,
Logic (library),
Object-oriented