UNIT - II

What are different Data objects in Prolog?

Prolog uses various data objects to represent and manipulate information. These objects form the fundamental building blocks of the language and include atoms, numbers, variables, structures, and lists .

1. Atoms => Atoms are constant symbolic values used to represent names or identifiers. They begin with a lowercase letter or are enclosed in single quotes.
   - Example: 'apple', 'New York', 'john'.

2. Numbers => Prolog supports integer and floating-point numbers for arithmetic operations.
   - Example: '42', '3.14'.

3. Variables => Variables are placeholders for values and start with an uppercase letter or an underscore . They get instantiated during execution.
   - Example: 'X', 'Person', '_Temp'.

4. Structures (Compound Terms) => These represent complex objects by grouping multiple components into a single entity.
   - Example: 'date(2024, march, 23)', 'point(X, Y)'.

5. Lists => Lists store collections of elements enclosed in square brackets '[]', with the head-tail notation
'[H | T]' for recursive processing.
   - Example: '[apple, banana, cherry]', '[X | Y]'.

These data objects allow Prolog to perform logical reasoning, pattern matching, and rule-based computation efficiently.

What is matching in Prolog computation?

In Prolog, matching is the process of checking if two terms can be made the same. When Prolog matches two terms, it assigns values to variables in a way that keeps them as free as possible. This is called the most general instantiation, meaning variables are only restricted as much as necessary. Keeping instantiations general allows Prolog to handle more conditions later in the program.

Sometimes, a less general instantiation happens when variables are given specific values unnecessarily. This makes the program less flexible because the values are fixed when they do not need to be. If two variables only need to be equal, Prolog does not assign them a particular value unless required. More restrictive instantiations can make solving problems harder, as they limit Prolog's ability to find multiple solutions.

As Prolog runs a program, variables become more specific step by step. At first, they may have no value, but as matching continues, they take on values to satisfy conditions. This gradual process helps

Prolog solve problems logically. Instead of assigning values immediately, Prolog waits until the program needs a specific value.

This method makes Prolog useful for problem-solving, databases, and artificial intelligence. Unlike other programming languages, Prolog allows flexible reasoning, which helps in complex situations where logic needs to be applied step by step.

Explain declarative meaning of Prolog programs.

The declarative meaning of a Prolog program explains what is logically true based on the given facts and rules. It defines the relationship between goals without considering the specific steps or order in which Prolog processes them. In simple terms, the declarative meaning tells us whether a particular goal is true and for what values of variables it is true.

A Prolog program consists of clauses, which are either facts (statements that are always true) or rules (statements that depend on other facts or rules). A rule in Prolog follows the form:
P :- Q, R.
This means that P is true if both Q and R are true. In declarative terms, this states a logical relationship: whenever Q and R hold, P must also hold. The semicolon (;) represents an OR condition, meaning that if at least one condition is true, the goal succeeds. For example, the clause:
P :- Q; R.
means P is true if either Q is true or R is true. This is logically the same as writing two separate rules:
P :- Q.
P :- R.

To understand the declarative meaning better, we use the concept of instances of a clause. An instance of a clause is a version of it where variables are replaced by specific values. For example, in the clause:
hasachild(X) :- parent(X, Y).
two variants could be:
hasachild(A) :- parent(A, B).
hasachild(X1) :- parent(X1, X2).
Instances of this rule occur when actual values replace the variables, such as:
hasachild(peter).
hasachild(barry).

In summary, the declarative meaning of a Prolog program helps determine what is true, independent of how Prolog finds the solution. It is based on logical relationships between facts and rules rather than execution steps. This approach makes Prolog suitable for logical reasoning and problem-solving, especially in fields like artificial intelligence and database querying.

Explain procedural meaning of prolog program.

The procedural meaning of a Prolog program describes how Prolog solves a problem step by step. It defines the execution process, explaining the order in which Prolog selects and evaluates rules to find a solution. Unlike the declarative meaning, which focuses only on logical truth, the procedural meaning shows the actual sequence of operations performed by Prolog to reach a conclusion.

In Prolog, a rule follows the form:
P :- Q, R.
The procedural meaning of this rule is: To prove P, first prove Q, and then prove R. Here, Q and R are called subgoals, and Prolog tries to solve them one by one. The comma (,) represents an AND condition, meaning that all subgoals must be satisfied for P to be true. The semicolon (;) represents an OR condition, meaning that if one subgoal fails, Prolog tries the next alternative. For example, in the clause:
P :- Q; R.
Prolog will first try to satisfy Q. If Q fails, then it will try R. This is different from the declarative meaning, which only states that at least one of Q or R must be true, without specifying the order of evaluation.

Prolog follows a depth-first search strategy and backtracking mechanism to find solutions. It starts from the main goal and searches through rules step by step. If a rule fails, Prolog backtracks to try another possible solution. This means the order of rules in a Prolog program affects execution. For example, if Prolog is given two rules for the same goal, it will try them in the order they appear in the program.

In summary, the procedural meaning of a Prolog program explains how Prolog finds answers by following a specific search process. It is based on goal ordering, execution steps, and backtracking rather than just logical correctness. Understanding the procedural meaning is important for writing efficient Prolog programs and controlling the way solutions are computed.

Example: monkey and banana:

The Monkey and Banana Problem is a classic artificial intelligence problem in the field of planning and problem-solving. It involves a monkey in a room where a bunch of bananas is hanging from the ceiling, out of its direct reach. The room contains objects such as a box that the monkey can use to reach the bananas. The challenge is to develop a sequence of actions that will allow the monkey to obtain the bananas. This problem is often used to illustrate how AI systems can plan and execute actions in a structured way to achieve a goal. Solutions to this problem involve techniques such as state-space search, means-ends analysis, and logic-based planning.

To solve the problem computationally, the environment is represented as a set of states, and the monkey can perform a series of actions such as walking, climbing, pushing objects, or grabbing the bananas. The AI system must determine the correct sequence of actions, such as moving towards the box, pushing it under the bananas, climbing on it, and finally reaching the bananas. Various AI techniques have been applied to model and solve the problem. The

Monkey and Banana Problem demonstrates how AI can break down complex tasks into smaller steps and use logical reasoning to achieve goals efficiently.

WRITE CODE HERE :

```
CODE.
```

Order of clauses and goals:

In Prolog, the order of clauses and goals significantly affects how a program executes, even if its declarative meaning remains correct. A major issue that can arise is **indefinite looping**, where Prolog continuously tries to satisfy a goal without making progress. For example, a clause like “p :- p.” leads to an infinite loop when Prolog attempts to prove “?- p.”

A similar issue can occur when solving problems, as demonstrated in the monkey and banana problem. If the movement rules are reordered incorrectly—such as placing "walk" before more essential actions like "grasp"—Prolog may enter a loop where the monkey keeps walking without ever reaching a solution. This example highlights how procedural semantics influence the efficiency and correctness of a Prolog program.

Another case where order matters is in recursive definitions, such as the `predecessor` relation. One problematic version is “pred4”.

```
pred4(X,Z) :-
        pred4(X,Y),
        parent(Y,Z).
pred4(X,Z) :-
        parent(X,Z).
```

This always tries the most complex step first, leading to infinite recursion in some cases. For example, querying `?- pred4(liz, jim).` results in an endless sequence of recursive calls. This shows how improper ordering can make a logically correct program unusable.

While different versions of the `predecessor` program may have the same logical meaning, some variations are more efficient than others. The best approach is to try **simpler conditions first**, reducing unnecessary recursive calls. Improper ordering can cause inefficiency or even prevent Prolog from finding an answer.

Although Prolog provides a powerful declarative approach, programmers must still consider procedural behaviour. The best practice is to focus on a correct declarative solution first, then refine the order of clauses and goals to ensure efficient execution. This balance between declarative correctness and procedural efficiency is key to writing effective Prolog programs.

Remarks on the relation between Prolog and logic:

Prolog is a programming language that is closely related to mathematical logic, particularly first-order predicate logic. The syntax of Prolog is based on clause form, which is a

representation of logical formulas where explicit quantifiers are removed, and only Horn clauses (clauses with at most one positive literal) are used. This logical foundation makes Prolog a powerful tool for representing knowledge and performing automated reasoning. While its syntax appears similar to formal logic notation, certain practical adaptations have been made to enhance its usability as a programming language. These adaptations include specific rules for structuring Prolog programs, making them more intuitive for problem-solving while maintaining a logical basis.

The procedural meaning of Prolog is derived from the resolution principle, a fundamental concept in logic-based automated theorem proving. This principle was introduced by J.A. Robinson in 1965 and serves as the foundation for Prolog's inference mechanism. Prolog employs a variant of resolution known as SLD-resolution (Selective Linear Definite clause resolution), which is optimized for computational efficiency. This resolution method allows Prolog to systematically derive conclusions from given facts and rules, making it particularly well-suited for logic programming and artificial intelligence applications. Researchers such as Lloyd (1984) and Nilsson (1981) have analyzed Prolog's logical properties and its effectiveness in solving problems based on formal logic.

A key operation in Prolog is matching, which corresponds to unification in logic. Unification is a process that finds substitutions to make different logical expressions identical. However, in practical Prolog implementations, matching is often optimized for efficiency, and the exact theoretical definition of unification may not always hold in execution. Despite these modifications, Prolog remains logically sound for most practical applications. The language is structured to ensure that logical consistency is maintained while enabling efficient computation, making it a practical tool for logic-based programming tasks.

In summary, Prolog maintains a strong connection to logic but also incorporates optimizations to function as an effective programming language. Its syntax is derived from first-order logic, while its execution model is based on logical inference techniques like resolution and unification. These features make Prolog an essential tool in artificial intelligence, knowledge representation, and problem-solving domains. Although understanding the deeper logical foundations of Prolog can be beneficial, it is not always necessary for using the language effectively as a programming tool.
