# Lesson 9: Introduction to Logic Programming and Prolog Basics

## Introduction to logic programming paradigm

Logic programming is a programming paradigm that is based on formal logic and is fundamentally different from traditional imperative or procedural programming paradigms. Instead of specifying explicit instructions to perform computations, logic programming focuses on describing relationships, rules, and constraints using logical statements.

The central concept in logic programming is the "predicate." A predicate is a statement that defines a relationship between objects or entities. It consists of a head and a body. The head represents the name of the predicate and its arguments, while the body contains logical conditions that must be satisfied for the predicate to be true.

One of the most well-known and widely used logic programming languages is Prolog (short for "Programming in Logic"). Prolog allows you to define predicates and rules, and it uses a process called "resolution" to infer new relationships from the given facts and rules.

**The basic components of logic programming paradigm include:**

**1. Facts:** Facts are simple statements that are assumed to be true. They define the initial state of the program's knowledge base. For example:

```
father(john, david).
mother(lisa, david).
```

**2. Rules:** Rules define logical relationships between predicates. They consist of a head and a body separated by the ":-" symbol. The head represents the conclusion of the rule, while the body contains conditions that must be satisfied for the rule to be applicable. For example:

```
parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).
```

**3. Queries:** In a logic programming language, you can ask questions or query the knowledge base to find solutions that satisfy certain conditions. The system will search for solutions based on the rules and facts provided. For example:

```
?- parent(john, david).
```

**4. Backtracking:** Logic programming languages use backtracking to explore multiple solutions. When a query is made, the system will attempt to find a solution. If it fails to find a solution, it will backtrack and explore other possible paths until all potential solutions have been exhausted.

Logic programming is particularly well-suited for solving problems that involve symbolic reasoning, pattern matching, and constraint satisfaction. It has been used in various fields, including artificial intelligence, natural language processing, expert systems, and knowledge representation.

Overall, logic programming offers a declarative and elegant way of describing problems and their solutions, focusing on what relationships exist rather than specifying how to compute them step-by-step.

# Understanding logical reasoning and rule-based systems

Logical reasoning and rule-based systems are closely related concepts, often used together in artificial intelligence and expert systems to make intelligent decisions based on logical rules and inferences. Let's explore each of these concepts separately:

## 1. Logical Reasoning:

Logical reasoning is a process of drawing conclusions from given premises or statements using formal logic. It involves applying rules of inference to derive new information from existing knowledge. The reasoning is based on the principles of deductive and inductive logic.

- **Deductive Logic:** In deductive reasoning, conclusions are derived from general principles (premises) to specific instances. If the premises are true, the conclusion must also be true. For example:

Premise 1: All humans are mortal.
Premise 2: John is a human.
Conclusion: Therefore, John is mortal.

- **Inductive Logic:** Inductive reasoning involves making generalizations from specific observations. Conclusions drawn from inductive reasoning are probable but not guaranteed to be true. For example:

Observation 1: Every crow observed so far is black.
Conclusion: Therefore, all crows are black (with some uncertainty).

Logical reasoning is fundamental to many AI applications, including knowledge representation, automated theorem proving, expert systems, and logical programming languages like Prolog.

## 2. Rule-Based Systems:

A rule-based system is an artificial intelligence system that uses a collection of rules to make decisions or draw conclusions about a given problem domain. These rules typically follow an "if-then" format, also known as production rules. Each rule consists of an antecedent (the "if" part) and a consequent (the "then" part).

When a rule-based system is presented with new data or a query, it matches the input against the antecedents of its rules and applies the rules whose antecedents match the input. This process is called pattern matching. The system then executes the corresponding consequents to produce the result.

**For example, consider a simple rule-based system for diagnosing diseases:**

Rule 1: IF symptoms include fever AND cough THEN diagnose with flu.
Rule 2: IF symptoms include headache AND sore throat THEN diagnose with cold.
Rule 3: IF symptoms include rash AND itching THEN diagnose with allergy.

If a patient presents symptoms of fever and cough, the rule-based system would apply Rule 1 and diagnose the patient with the flu.

Rule-based systems are widely used in expert systems, decision support systems, and problem-solving domains where the knowledge can be expressed in the form of rules.

They provide a transparent and understandable way to model complex reasoning processes, making them useful in domains where human expertise is required.

In conclusion, logical reasoning forms the foundation for rule-based systems, which leverage formal rules and inferences to make intelligent decisions based on given data and knowledge. These systems have proven to be valuable tools in various AI applications, where explicit rules and clear decision-making processes are crucial.

## Logic programming vs. other programming paradigms

Logic programming, imperative programming, and functional programming are distinct programming paradigms, each offering a different approach to problem-solving and computation.

Logic programming, represented by languages like Prolog, centers around defining relationships, rules, and constraints using formal logic. Instead of providing explicit step-by-step instructions, logic programming languages utilize a process called resolution to infer relationships from the provided facts and rules. The paradigm is declarative in nature, meaning programmers specify what they want to achieve rather than the exact sequence of steps to achieve it. This makes it particularly suitable for tasks involving symbolic reasoning, knowledge representation, and constraint satisfaction problems, making it a common choice in various AI domains.

Imperative programming, found in languages like C, Java, and Python (to some extent), revolves around describing a sequence of instructions that modify the program's state to reach a specific outcome. The program is composed of statements that explicitly manipulate variables and control flow. This paradigm is procedural, as it defines procedures (functions or methods) that manipulate data. Imperative programming is well-suited for tasks that require explicit control flow and direct manipulation of data, making it popular for a wide range of general-purpose programming scenarios.

Functional programming, exemplified by languages like Haskell, Lisp, and, in part, JavaScript (with functional programming libraries), emphasizes the evaluation of mathematical functions and immutable data structures. Rather than changing state, functional programming functions transform data through pure function calls, avoiding side effects. Like logic programming, functional programming is also declarative, focusing on what should be computed rather than how it should be computed. This paradigm is often chosen for tasks that involve data transformation, parallel processing,

and a preference for immutability, which can lead to more concise and easier-to-reason code.

Each programming paradigm has its strengths and weaknesses, and the choice of paradigm largely depends on the nature of the problem at hand and the preferences and expertise of the programmer. In some cases, combining paradigms can lead to more effective and expressive solutions, such as blending functional and logic programming to leverage the strengths of both in specific problem domains. Overall, the selection of a programming paradigm is an important decision that can significantly impact the design and efficiency of a software solution.

## Overview of Prolog language

Prolog (Programming in Logic) is a widely used logic programming language that was developed in the 1970s. It is based on formal logic and is known for its declarative nature, making it suitable for solving problems that involve symbolic reasoning, knowledge representation, and rule-based systems. Prolog is widely used in artificial intelligence research, natural language processing, expert systems, and various other applications.

**Here's an overview of some key features and characteristics of the Prolog language:**

**Declarative Nature:** Prolog is a declarative language, meaning programmers specify what they want to achieve rather than explicitly detailing how to achieve it. Prolog programs consist of a set of facts and rules that describe relationships and logic constraints.

**Facts and Rules:** The basic building blocks of Prolog programs are facts and rules. Facts represent simple statements about relationships between objects or entities, and rules define logical relationships between predicates.

**Predicate and Argument:** In Prolog, a predicate is a statement or a relationship between objects. It consists of a name (atom) and a list of arguments enclosed in parentheses. Arguments can be constants or variables.

**Unification:** Unification is a core operation in Prolog that involves matching predicates and resolving variables. It is used during rule resolution and query execution to find values for variables that satisfy the specified conditions.

**Queries:** Prolog programs can be interactively queried. Users can ask questions or make logical queries using predicates with variables or specific values. Prolog's search algorithm, based on backtracking, explores the knowledge base to find solutions to the queries.

**Backtracking:** Prolog uses backtracking to explore multiple paths and find all possible solutions to a given query. If a particular path fails to find a solution, Prolog backtracks and explores alternative paths to find additional solutions.

**Negation:** Prolog includes a negation operator ("not" or "negation as failure") that allows expressing negation in queries. It enables stating that a particular fact or condition does not hold.

**Cut (!):** The "cut" symbol is a special operator in Prolog. It is used to control backtracking and limit the search space, providing a way to prioritize certain solutions and avoid unnecessary exploration of alternative paths.

**Prolog Implementations:** There are several Prolog implementations available, both open-source and commercial. Some popular implementations include SWI-Prolog, GNU Prolog, and SICStus Prolog.

Prolog's strengths lie in its ability to handle complex symbolic reasoning and search tasks efficiently. Its simple syntax and powerful pattern matching capabilities make it an expressive language for certain types of problem domains. However, its execution model based on backtracking can lead to inefficiencies in certain scenarios.

Overall, Prolog remains a relevant and influential language in the field of artificial intelligence and symbolic reasoning, with applications in various knowledge-based systems and expert systems.

## Syntax and structure of Prolog programs

Prolog is a programming language with a straightforward and declarative syntax. Prolog programs consist of a collection of facts and rules that describe relationships between entities, along with queries to interact with the knowledge base. Atoms are used to represent constants and start with lowercase letters or are enclosed in single quotes if

they include special characters or spaces. Variables, on the other hand, begin with an uppercase letter or an underscore character.

Predicates are at the core of Prolog programs. They define relationships between entities and are formed by an atom (predicate name) followed by a list of arguments enclosed in parentheses. These arguments can be either atoms or variables. Facts are used to establish the initial knowledge base and are expressed as predicates with specific arguments. Each fact ends with a period (.) symbol.

Rules, another crucial element in Prolog, define logical relationships between predicates. They consist of a head and a body, separated by the ":-" symbol. The head represents the conclusion of the rule, while the body contains conditions that must be satisfied for the rule to be applicable. Like facts, rules also end with a period (.) symbol.

To interact with the knowledge base, users can make queries. Queries are questions or logical queries entered at the Prolog prompt, and they end with a question mark (?). Queries consist of predicates with variables or specific values to find solutions based on the rules and facts provided in the program.

Prolog supports comments, which start with the percent (%) symbol and continue to the end of the line. Comments are used to add explanatory notes to the code.

The structure of a Prolog program typically involves the declaration of facts and rules, followed by queries made to the knowledge base. Prolog systems use resolution and backtracking to find solutions to the queries based on the provided rules and facts. The order of rules and facts in a Prolog program can influence the order in which solutions are found since Prolog performs a top-down search with backtracking.

## Knowledge representation using facts and rules

Knowledge representation using facts and rules is a fundamental concept in logic programming, and it plays a crucial role in languages like Prolog. The main idea behind knowledge representation is to structure information in a way that allows the program to make intelligent decisions and draw logical inferences. This is achieved through two primary components: facts and rules.

Facts are simple statements that describe relationships between entities. They are the building blocks of the knowledge base and serve to define the initial state of the program's understanding. Facts are represented as predicates, which consist of an atom (the predicate name) followed by a list of arguments enclosed in parentheses.

These arguments can be constants, representing specific entities or values, or variables, representing placeholders for unknown values. Facts establish known relationships and form the foundation of the knowledge base.

On the other hand, rules define logical relationships and dependencies between predicates. A rule consists of a head and a body, separated by the ":-" symbol. The head represents the conclusion of the rule, indicating what can be inferred, while the body contains conditions that must be satisfied for the rule to be applicable. Rules are used to derive new information from existing facts and relationships. By applying rules to the facts, the program can make logical deductions and expand its understanding of the domain.

Together, facts and rules allow us to build a knowledge base that represents complex relationships and logical constraints in a structured manner. This knowledge base can then be queried to answer questions or make logical inferences about the information it contains. When a query is made, the Prolog system utilizes resolution and backtracking to explore the knowledge base, apply relevant rules, and find all the solutions that satisfy the query conditions.

## Unification and backtracking in Prolog

In Prolog, unification and backtracking are two fundamental concepts that lie at the heart of its resolution mechanism. Unification is the process by which Prolog matches and resolves predicates during query evaluation. It involves finding values for variables in predicates that make two predicates identical. When a query is posed in Prolog, the system attempts to unify the query with the facts and rules present in the knowledge base to find matching solutions.

During the unification process, Prolog follows specific rules: atoms unify only with the same atom, variables can unify with any atom or variable, and their values are instantiated accordingly, and two complex terms (predicates with arguments) unify if their functors (predicate names) are the same, and their arguments unify accordingly.

Unification is critical for finding solutions to queries as it allows the system to match queries with facts and rules, instantiate variables, and draw logical inferences. It is an essential step in the resolution process that enables Prolog to evaluate queries and explore potential solutions.

Backtracking, on the other hand, is a powerful search mechanism used by Prolog to explore multiple paths and find all possible solutions to a given query. When Prolog

attempts to find a solution to a query, it applies facts and rules, seeking to satisfy the query's conditions. If it successfully finds a solution, it reports it to the user. However, if it reaches a dead-end or a point where no more solutions can be found, Prolog backtracks to explore alternative paths.

Backtracking works by undoing choices made during the resolution process, such as unifications or rule applications. This enables Prolog to explore different branches of the search tree, searching for additional solutions. The use of backtracking makes Prolog a highly expressive language for solving problems with multiple solutions.

The combination of unification and backtracking gives Prolog its non-deterministic and exploratory capabilities. It allows Prolog to efficiently search through a vast knowledge base, considering various possibilities and finding all possible solutions to a query. This makes Prolog well-suited for tasks involving symbolic reasoning, constraint satisfaction, and rule-based systems, where exploring multiple paths and considering different interpretations of the data is essential for finding solutions to complex problems.

## Recursive programming in Prolog

Recursive programming is a powerful technique in Prolog that allows functions or predicates to call themselves within their own definitions. In Prolog, recursion is particularly well-suited for solving problems that involve repetitive or iterative operations. Recursive predicates are essential for traversing data structures, performing depth-first searches, and handling tasks with complex nested relationships.

The structure of a recursive predicate typically consists of two parts: a base case and a recursive case.

**1. Base Case:**
The base case is the stopping condition that defines when the recursion should terminate. It represents the simplest or smallest possible input for which the predicate's solution is already known without further recursive calls. The base case is crucial to avoid infinite recursion and ensure that the recursive predicate eventually terminates.

**Example of a base case in a predicate to calculate the factorial of a number:**

```
factorial(0, 1).
```

In this example, when the input is 0, the factorial is 1. This serves as the base case for the recursion.

***2. Recursive Case:***

The recursive case defines the relationship between the problem and its subproblems. It describes how the predicate calls itself with smaller or simpler inputs to approach the base case. By breaking down the problem into smaller instances of itself, the predicate can gradually approach the base case and combine the results to obtain the final solution.

**Example of a recursive case in the same factorial predicate:**

```
factorial(N, Result) :-
    N > 0,
    N1 is N - 1,
    factorial(N1, SubResult),
    Result is N * SubResult.
```

In this example, the predicate calculates the factorial of a number N by recursively computing the factorial of (N-1) and then multiplying it by N.

Using recursive programming, Prolog can elegantly handle problems that require iteration, such as tree or graph traversal, list manipulation, and certain mathematical computations. However, it's essential to ensure that the base case is correctly defined to avoid infinite recursion.

Overall, recursive programming is a valuable technique in Prolog that enhances the language's expressive power and enables the concise representation of complex algorithms and problem-solving strategies.