Declarative Programming

Declarative programming is a programming paradigm that focuses on describing what you want the computer to do, rather than specifying how to do it. In other words, you tell the computer what you want, and it figures out the best way to accomplish it.

Think of it like giving instructions to a friend. When you ask your friend to make a cup of coffee, you don't need to explain the step-by-step process of grinding the beans, boiling the water, and pouring it into a cup. Instead, you simply state your desire, and your friend knows how to make coffee and will handle the details.

Similarly, in declarative programming, you write code that describes the desired outcome or the problem you want to solve. You define the relationships between different parts of your program without specifying the exact sequence of steps. The computer then uses this information to automatically figure out the most efficient way to achieve the desired result.

This is different from imperative programming, where you explicitly specify the steps and the order in which they should be executed. In declarative programming, you focus more on the "what" rather than the "how," which can make the code easier to read and understand. It also allows for more flexibility and can make it easier to reason about and maintain your code.

Examples of Declarative programming languages

* SQL (Structured Query Language): SQL is a popular declarative language used for interacting with databases. You can write SQL queries to specify what data you want to retrieve or manipulate, without needing to specify how the database management system should perform those operations.
* HTML (Hypertext Markup Language): HTML is a markup language used for creating web pages. It is declarative in nature because you describe the structure and content of the page using tags, without explicitly specifying how the browser should render the page.
* CSS (Cascading Style Sheets): CSS is another language used in web development to describe the visual appearance of HTML elements. It is declarative as you define styles, such as colours, fonts, and layouts, without specifying the exact steps the browser should take to apply those styles.
* Functional Programming: Functional programming is a programming paradigm that emphasizes writing code in a declarative style. Languages like **Haskell and Lisp** promote functional programming principles, where you define functions that transform data rather than specifying explicit steps of execution.

Prolog

Prolog is a logical programming language that was developed in the 1970s. Its name stands for "Programming in Logic." Prolog is based on the concept of logic programming, where programs are defined in terms of logical relations and rules rather than traditional imperative or procedural instructions.

Prolog programming language that is associated with artificial intelligence and computational linguistics.

In Prolog, you define a knowledge base consisting of facts and rules. Facts represent basic statements about the world, while rules establish relationships between those facts. The language uses a declarative syntax, meaning that you specify what you want to achieve rather than how to achieve it.

There are three basic constructs in Prolog:

* Facts
* Rules
* queries.

What are facts?

Facts are statements that represent basic information about the world or the domain you are working with. They provide the foundation of the knowledge base in a Prolog program. Facts are used to assert or declare certain properties or relationships between objects or entities.

facts are typically represented as logical assertions or predicates. ***A predicate is a statement or a proposition that can be true or false***.

A fact consists of a predicate name followed by a comma-separated list of arguments, enclosed in parentheses. The arguments can be constants, variables, or other predicates. The predicate names the relationship or property, and the arguments are the entities involved in this relationship or the subjects of the property. The number of arguments (also known as the arity of the predicate) can vary depending on the nature of the relationship or property being described.

Here's an example of a Prolog fact:

father(john, jim).

In the example above, father is the predicate, and john and jim are the arguments. This fact asserts that "John is the father of Jim." The fact is true if it can be unified with a query that matches the predicate and arguments.

country(france). *The fact is that France is a country.*

Facts can also have multiple arguments to represent more complex relationships. For example:

parent(john, jim, ann).

This fact states that "John is the parent of both Jim and Ann."

Facts can be used to define properties, relationships, or any other information that you want to represent in the knowledge base of your Prolog program. They serve as the building blocks for reasoning and deriving new information through rules and queries.

When you execute a Prolog program, the facts in the knowledge base are consulted to determine the truth or falsehood of queries or to derive new information based on the rules defined in the program.

Note that all facts in Prolog use lower-case letters and end with a full stop.

Rules

Rules are logical statements that define relationships or logical implications based on the facts in the knowledge base. Rules allow you to derive new information or make logical inferences by combining facts and applying logical conditions.

A rule consists of a head and a body, separated by the :- symbol, which can be read as "if." The head of the rule is what is concluded (or can be proven) if the body (the conditions following the :-) is true. The body of the rule can contain one or more conditions that need to be satisfied for the rule to apply, and these conditions are separated by commas, which act as logical AND operators. If the body of the rule has multiple conditions, all of them must be true for the rule to be applied.

Here's an example of a Prolog rule:

grandparent(X, Z) :- parent(X, Y), parent(Y, Z).

SYNTAX of the rule

Head :- Body.

* **Head**: This is a predicate that represents the conclusion that can be drawn if the body of the rule is true. It can contain variables, which are placeholders that get instantiated with specific values when the rule is applied.
* **Body**: This consists of one or more predicates (which can be either facts or other rules) that serve as conditions for the rule. These predicates are separated by commas, indicating a logical AND relationship. The body can also include logical OR and NOT constructs through different means (e.g., using semicolons for OR, and the built-in predicate \+ for NOT).

likes(john, X) :- hobby(X), \+ member(X, [football, tennis]).

In this rule, likes is the predicate in the head, john is a constant, and X is a variable. The body of the rule specifies that John likes something (X) if X is a hobby and is not a member of the list [football, tennis]. Here, \+ is the built-in predicate for negation, and member is a built-in predicate to check if an element is a member of a list.

Example of a Rule

Consider the following example, which builds on the parent and grandparent relationships:

% Facts

parent(alice, bob).

parent(bob, charlie).

% Rule

grandparent(X, Y) :- parent(X, Z), parent(Z, Y).

This rule states that X is a grandparent of Y if X is a parent of Z and Z is a parent of Y. The variables X, Y, and Z allow the rule to be applied to any entities that satisfy these conditions within the knowledge base.

Rules are used in Prolog to **query the knowledge base for information that is not explicitly stated but can be inferred**. For example, given the facts and rule above, if you query Prolog with grandparent(alice, Who)., Prolog will use the rule for grandparent to infer that Alice is a grandparent of Charlie by finding a Z (in this case, Bob) that satisfies both conditions in the body of the rule.

Queries

In Prolog, queries are a way to ask questions or make requests to the Prolog interpreter about the facts and rules defined in a program. A query is a statement or a proposition that you want to find out if it is true or false based on the knowledge base.

To perform a query, you write a predicate followed by a list of arguments, enclosed in parentheses. The Prolog interpreter then tries to find a solution or answer to the query by searching through the facts and applying the rules defined in the program.

Example

father(john, jim).

parent(john, jim, ann).

likes(john, X) :- hobby(X), \+ member(X, [football, tennis]).

We can now ask queries based on this knowledge base. For instance, we can ask:

?- father(john, jim).

The ?- symbol indicates the start of a query. In this query, we are asking if "John is the father of Jim" is true. The Prolog interpreter will search the facts and check if there is a fact that matches the query. If a matching fact is found, the interpreter will respond with "true." If no matching fact is found, it will respond with "false."

You can also use variables in queries to ask more general questions. For example:

?- father(john, X).

In this query, we are asking the Prolog interpreter to find a value for the variable X such that "John is the father of X" is true. The interpreter will search the facts and try to find a value that satisfies the query. In this case, it will respond with the value of X that makes the query true (if any).

Queries allow you to interact with the Prolog program and obtain information or solutions based on the defined facts and rules. They are a way to explore and reason about the knowledge base and retrieve relevant information or make logical inferences.

Backtacking

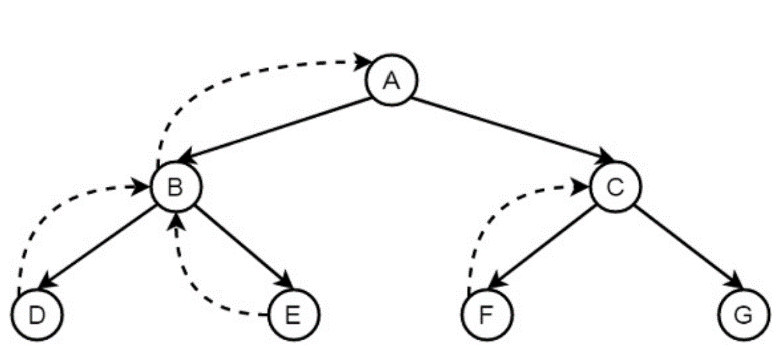
Backtracking in Prolog is a fundamental mechanism that allows the language to explore different possibilities in order to find all the answers or solutions to a query. It's how Prolog deals with situations where there is more than one way to satisfy a query, or when an attempted solution path doesn't lead to a solution and another path needs to be tried.

When you make a query in Prolog, the interpreter tries to find a solution by matching the query with the facts and rules in the knowledge base. If a match is found, the interpreter provides the corresponding answer. However, if no match is found or if you request additional solutions, Prolog employs backtracking to explore other possible paths.

How backtracking works:

1. **Trying Solutions:** When Prolog tries to solve a query, it starts by looking at the first fact or rule that matches the query. If the query can be satisfied by this fact or rule, Prolog considers it a potential solution.
2. **Encountering Failure:** If, while trying to satisfy a query (or part of a query), Prolog reaches a point where no facts or rules apply, or it cannot satisfy the conditions of a rule, it encounters a failure. This means the current path to solving the query cannot continue.
3. **Backtracking:** Upon encountering a failure, Prolog automatically "backtracks" to the most recent point where it had a choice of different facts or rules to apply. Prolog then tries the next available fact or rule that could satisfy the part of the query it is working on. This process involves undoing any variable bindings that were made along the unsuccessful path.
4. **Finding All Solutions:** Prolog continues this process of trying, failing, and backtracking until it has explored all possible ways to satisfy the query. This mechanism allows Prolog to find all solutions to a query, if multiple solutions exist.

Using a tree structure this is how backtracking works.



Suppose A to G are some rules and facts. We start from A and want to reach G. The proper path will be A-C-G, but at first, it will go from A to B, then B to D. When it finds that D is not the destination, it backtracks to B, then go to E, and backtracks again to B, as there is no other child of B, then it backtracks to A, thus it searches for G, and finally found G in the path A-C-G. (Dashed lines are indicating the backtracking.) So when it finds G, it stops.

Example fo a prlog program

% Facts

mammal(dog).

mammal(cat).

mammal(human).

has\_fur(dog).

has\_fur(cat).

has\_hands(human).

% Rules

animal(X) :- mammal(X).

has\_hair(X) :- mammal(X), has\_fur(X).

% Queries

?- mammal(cat).

?- has\_fur(dog).

?- animal(X).

?- has\_hair(X).

SWI-Prolog