Introduction to Prolog

Berlenko Tatiana
2025

What is it

- PROgramming in LOGic is a declarative programming language.
- In Prolog, programs are represented as a set of facts and rules that describe relationships between different entities and processes.
- We can ask the system questions and Prolog uses its facts and rules to answer yes or no (true or false).
- Prolog automatically looks for answers by linking facts and rules to solve the problem at hand.

Applications and usage

- Artificial Intelligence
- Natural Language Processing
- Databases and expert systems
- ProB Animator and Model Checker
- In part of Watson IBM Prolog was used primarily for NLP patterns

How to execute a program

https://swish.swi-prolog.org/

Facts

- ➢ Facts are the primary form of expressing knowledge.
- ➢ They are used to represent immutable statements about some objects or relationships between objects.
- ➢ Facts in Prolog are written as sentences with a **.** at the end.
- ➢ Examples:
    - human(socrates).
    - parent(john, mary).

Queries

- A query is a way to ask Prolog to find answers using the available facts and rules.
- Example:
  - ?- human(socrates).
- Prolog responds *true* if the statement matches its database. If Prolog cannot validate the statement, it responds *false*.
- We can use variables:
  - ?- human(X).

  Prolog will respond with a list of all entities X for which it can be proven to be a person, based on its database.

Variables

- ➢ Variable names begin with a capital letter or underscore (_).
- ➢ An unnamed variable (_) is used when its name is indifferent to the programmer.
- ➢ Can take any value that satisfies the conditions of a given statement or query.
- ➢ If a variable is assigned a value, that value cannot be changed in the same branch of the proof.
- ➢ Variables X that occur in two different rules are different X variables.

Queries

We can use variables:

```
?- human(X).
```

Prolog will respond with a list of all entities X for which it can be proven to be a person, based on its database.

Rules

- ➢ Used to describe relations between statements.
- ➢ Syntax:

  head :- tail.

  where head is a statement and tail (or body) is a sequence of statements.
- ➢ The left part is true if the whole right part is true.

Elements of a Prolog-based program

- <u>Atoms</u> are basic characters or strings that are used in Prolog to represent specific data or identifiers.
    - sister
    - ‘Hello, World!’
- <u>Numbers</u>:
    - 1.2
    - 10
- <u>Functors</u> are used to organize atoms into structures representing relations or properties of objects. A functor has a name (which is itself an atom) and an arity (the number of arguments).
    - likes(mother, ann), the functor *likes* has an arity of 2, and mother and ann are its arguments, which are also atoms.

Predicates

- Predicates use functors to formulate logical statements, which can be facts or rules. A predicate can be simple, consisting of a single functor and its arguments, or more complex, involving several functors and logical relations between them.
  - parent(X, Y) :- father(X, Y).
  - parent(X, Y) :- mother(X, Y).

Selective Linear Definite clause resolution

Resolution is a general method of automatic proof in predicate logic.

Prolog programs can be seen as a set of Horn clauses, each of which clause has the form

head :- body.

SLD (Selective Linear Definite clause resolution) is a specialisation of resolution for linear and definite Horn formulas used in Prolog.

SLD-resolution

```
p1 :- p2, p3, ..., pN. % rule

p_fact. (p_fact :- true) % fact

?- p3(X), p4(Y). % query or goal
```

There is a goal which is a list of subgoals.

The SLD algorithm sequentially takes one goal (the leftmost one in Prolog) and tries to unify it with the ‘head’ of a fact or rule in the knowledge base:

- ➢ Successful unification means that the subgoal matched the head of some rule or fact.
- ➢ In this case, the substitution is applied both to the current subgoal and to the remaining subgoals in the query.
- ➢ After unification, the current subgoal is replaced with the body of the rule (taking into account the substitution).
- ➢ As a result, we get a new list of subgoals.

Recursive proof:

- ➢ If the list of subgoals becomes empty, then all subgoals have been successfully proven, and a solution has been found.
- ➢ Otherwise, the process continues with the new leftmost subgoal.

Example

- *Our query:*

?- mother(X, oceanus).
Prolog maps mother(X, Y) to mother(X, oceanus), substituting Y = oceanus.

- *Prolog has subgoals:*

?- female(X), parent(X, oceanus).
Prolog checks the first subgoal female(X), finds the fact female(gaea) and makes the substitution X = gaea.

- *Prolog substitutes the value gaea to X in the second subgoal:*

parent(gaea, oceanus).

- Prolog finds parent(gaea, oceanus) in the database → the proof is successful.

Program

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

female(gaea).

parent(gaea, oceanus).
```

Backtracking

If the current path has reached an dead end (the next goal cannot be unified with any rule), Prolog backtracks to the previous choice point (where there was another unification alternative) and tries the next option.

Backtracking allows Prolog to explore different rules for the same target without manual intervention.

We can use **!** (cut operator) to deny backtracking:

```
        mother(X, Y) :- female(X), parent(X, Y), !.
```

Tasks

- ➢ Write a Prolog query to find all grandmothers and uncles.
- ➢ Write a Prolog query that determines if two people are siblings.
- ➢ Write a Prolog query to find the oldest known ancestor of a person.
- ➢ Write a Prolog query to determine if two people are cousins.
- ➢ Write a Prolog query to find the common ancestor of two people.
- ➢ Write a Prolog rule that, given two people, determines their exact family relationship (e.g., parent, grandparent, sibling, cousin, etc.).

Arithmetic and logical operations and comparisons

- The *is* predicate in Prolog is used to perform arithmetic calculations and assign the result to a variable.
  - Sum is X + Y.
- We can create predicates:
  - subtract(X, Y, Result) :- Result is X - Y.
  - multiply(X, Y, Result) :- Result is X * Y.
  - divide(X, Y, Result) :- Result is X / Y.
- equal(X, Y) :- X =:= Y.
- not_equal(X, Y) :- X =\= Y.
- >, <, >=, <=
- , is logical AND
- ; is logical OR
- \+ is logical NOT

Conditional expressions

- Format:

```
(Condition -> Action1 ; Action2).
```

- How it works:
  - If the Condition is true, Action1 is performed.
  - If the Condition is false, Action2 is executed.
- Example:

```
is_even(N) :- (0 is N mod 2 -> writeln('Even') ; writeln('Odd')).
```

Tail recursion

- It is a type of recursion in which the recursive call is the last operation in the predicate.
- Prolog interpreter can optimise such calls and avoid unnecessary memory usage (stack overflow).
- Example:

```
factorial(0, 1).
factorial(N, Fact) :- N > 0,
   N1 is N - 1,
   factorial(N1, F1),
   Fact is N * F1.
```

```
fact_helper(0, Acc, Acc).
fact_helper(N, Acc, Fact) :- N > 0,
   NewAcc is Acc * N,
   N1 is N - 1,
   fact_helper(N1, NewAcc, Fact).
factorial(N, Fact) :- fact_helper(N, 1, Fact).
```

Some of the techniques

1. functor(_, X, Result) :- Result = X.

   the same as

   functor(_, X, X).

2. We can add one more argument in our query:

   functor(X, Result) :- helper_for_functor(Var, X, Result).

3. Instead of conditional expression use several predicates:

   abs_value(N, N) :- N >= 0.

   abs_value(N, PosN) :- N < 0, PosN is -N.

Tasks

- ➢ Write a predicate max(X, Y, Result) that computes the maximum from 2 digits.
- ➢ Write a recursive predicate sum_digits(N, Sum) that computes the sum from 1 to N.
- ➢ Create a predicate count_digits(N, Count) to count the number of digits in a number.
- ➢ Create sum_even_squares(N, Sum) to compute the sum of squares of even numbers from 1 to N.
- ➢ Write is_palindrome(N) to check if a number reads the same forward and backward recursively.
- ➢ Write a program fibonacci(N, Res) to calculate Fibonacci numbers.