

# Meta-predicates in Prolog

---

## Equivalence of programs and data

### 1. Introduction

There is one thing that make Prolog a really powerful programming language: **predicates** and **terms** (also viewed as programs and data) share the same **syntactic structure**. As a consequence of this equivalence, programs and data can be easily interchanged. But what does it really mean?

Prolog provides several predefined predicates, known as **meta-predicates**, to deal with these structures and work with them.

### 2. Call Predicate

The **call** predicate is a fundamental meta-predicate that allows the execution of a term as if it were a goal. The key thing to observe is that with the **call** predicate, we can dynamically test a program inside the same program!

#### Functionality

1. If a term  $T$  is meant to be a predicate, we can prepare and create that term in a proper way and then **execute it**.
2. **call( $T$ )**: the term  $T$  is treated as a goal, and the Prolog interpreter is requested to evaluate it.
3. The term  $T$  must be a **non-numeric term** at the moment of the evaluation, since a number cannot be evaluated in a logical sense.

#### Why Call is a Meta-Predicate

The predicate **call** is considered to be a meta-predicate because:

1. Its evaluation interfere with the Prolog interpreter, as it stops the evaluation of the current **goal** in order to evaluate the subject of the **call** predicate.
2. It directly alters the program's execution flow.

#### Example

```
p(a).  
q(x) :- p(x).  
  
:- call(q(y)).  
yes Y = a.
```

Asking for **q(y)**, it is unified with the clause **q(x)**. Our goal becomes **p(y)**, and **p(y)** is true if  $Y$  is unified with constant **a**. The final answer is: **q(y)** is verified if  $Y = a$ .

This code snippet demonstrates the previous definition: we can pass to the `call` meta-predicate pieces of a program instead of just data.

The predicate `call` can also be used to call other predicates.

```
p(X):- call(X).
q(a).

:- p(q(Y)).
yes Y = a.
```

### Example `if_then_else` Construct

`call` is crucial for implementing control flow structures, such as an `if_then_else` construct. The goal is to define a `if_then_else(Cond, Goal1, Goal2)` clause such that: if `Cond` is true, execute `Goal1`; otherwise, execute `Goal2`.

```
if_then_else(Cond,Goal1,Goal2):-
    call(Cond), !,
    call(Goal1).
if_then_else(Cond,Goal1,Goal2):-
    call(Goal2).
```

The program `if_then_else` takes in input **three different programs**: `Cond`, `Goal1` and `Goal2`. (Note: in this case, the **CUT operator** is essential to prevent the interpreter from backtracking the second clause if `Cond` succeeds).

## 3. Fail predicate

The `fail` predicate is a simple, arity-zero predicate used primarily to **explicitly control backtracking**.

### Functionality

1. `fail` takes **no argument**.
2. Its evaluation **always fails**.
3. This failure forces the interpreter to **explore other alternatives** (in other words, it activates the backtracking).

### Applications

Forcing a proof to fail might seem counter-intuitive, but it serves three main purposes:

1. To obtain some form of **iteration** over data.
2. To implement the **Negation as Failure** mechanism.
3. To implement a form of **logical implication**.

### Iteration Example

Let us consider a **Knowledge Base** with facts  $p(X)$ . and suppose we want to apply a predicate  $q(X)$  on all  $X$  that satisfy a fact  $p(X)$ .

```
iterate :-
    call(p(X)),
    verify(q(X)),
    fail.

iterate.

verify(q(X)) :- call(q(X)), !.
```

- The first clause of `iterate` finds a solution for  $p(X)$ , executes  $q(X)$ , and then fails, triggering the search for the next solution for  $p(X)$ .
- This process continues until `call(p(X))` eventually fails, at which point Prolog moves to the second clause, `iterate.`, which succeeds, stopping the overall goal.

## Negation as Failure Example

Defining a predicate `not(P)` which is true if  $P$  is not a **logical consequence** of the program.

```
not(P) :-
    call(P),
    !,
    fail.

not(P).
```

- If `call(p)` succeeds, the cut `!` prevents backtracking, and `fail` ensures `not(P)` fails.
- If `call(p)` fails, the first clause is skipped, and the second clause `not(P).` succeeds.

(Note: in this example, there are two items that don't have any **declarative meaning**: the predicate `call` and the CUT operator `!`).

## Combining fail and CUT Example

The sequence `!, fail` is often used to force a **global failure** of a predicate, stopping not only backtracking within the predicate but also preventing all the other possible alternatives for it.

Define the **fly property**, that is true for all the birds except penguins and ostriches.

```
fly(X) :-
    penguin(X),
    !,
    fail.

fly(X) :-
```

```

    ostrich(X),
    !,
    fail.

fly(X) :-
    bird(X).

```

If  $X$  is a penguin, the first clause succeeds up to the `fail..` The `!` prevents Prolog from trying the next `fly(X)` clauses, forcing the global failure for that specific  $X$ . Although this program is a good starting point for handling exceptions, it grows linearly according to the number of exceptions handled.

## 4. Setof and Bagof Predicates

These meta-predicates address **second-order queries**, which ask for the collection of elements that satisfy a goal, rather than just a single solution.

### Existentially Quantification

In Prolog, the usual query `:- p(X).` returns a **possible substitution** for variables of `p` that satisfies the query, implying that  $X$  is existentially quantified (is there an  $X$  such that `p(x)` is true?).

Sometimes, it can be helpful to set up a query that asks: *which is the set  $S$  of element  $X$  such that `p(X)` is true?* This type of query is named **second-order query**.

#### `setof(X, P, S)`

- **Functionality.**  $S$  is the **set** of instances  $X$  that satisfy the goal  $P$ .
- **Property.** It generally returns a set **without repetitions** and the elements are typically sorted.
- **Failure.** If no  $X$  satisfied  $P$ , the predicate **fails**.

#### `bagof(X, P, L)`

- **Functionality.**  $L$  is the **set** of instances  $X$  that satisfy the goal  $P$ .
- **Property.** It returns a list that may contain **repetitions**.
- **Failure.** If no  $X$  satisfies  $P$ , the predicate **fails**.

### Examples

Given the Knowledge Base:

```

p(1).
p(2).
p(0).
p(1).
q(2).
r(7).

```

The results of the following second-order queries are:

```
:- setof(X, p(X), S).
   yes S = [0, 1, 2]
   X = X

:- bagof(X, p(X), L).
   yes L = [1, 2, 0, 1]
   X = X
```

As we can see, the set  $S$  from `setof(X, P, S)` does not include repetitions, while `bagof(X, P, L)` returns a list that does.

Furthermore, these meta-predicates allow for the **conjunction** of goals within their own scope.

```
:- setof(X, (p(X), q(X)), S).
   yes S = [2]
   X = X

:- bagof(X, (p(X), q(X)), L).
   yes L = [2]
   X = X

:- setof(X, (p(X), r(X)), S).
   no

:- bagof(X, (p(X), r(X)), L).
   no
```

The last two queries tell us that the Knowledge Base does not have any  $X$  that satisfies the conjunction of goals `p(x), r(x)`.

```
:- setof(p(X), p(X), S).
   yes S = [p(0), p(1), p(2)]
   X = X

:- bagof(p(X), p(X), S).
   yes S = [p(1), p(2), p(0), p(1)]
   X = X
```

These meta-predicates can also retrieve the **terms** that make our goal true. For instance, the query `:- setof(p(x), p(x), S)` returns the set of terms `p(X)` that makes the goal `p(X)` verified.

## Example

Given the Knowledge Base:

```
father(mario, aldo).
father(mario, paola).
father(giovanni, mario).
father(giuseppe, maria).
father(giovanni, giuseppe).
```

We want to derive which individuals are fathers.

```
:- setof(X, Y^father(X, Y), S).
   yes [giovanni, mario, giuseppe]
   X = X
   Y = Y
```

The goal part uses a new **syntactic rule**, the **existential quantifier**  $Y^{\wedge}$ . This allows us to retrieve the set of  $X$  values such that there **exists** a  $Y$  that satisfies the goal `father(X, Y)`. If the existential quantifier is not used, the final result will be displayed as multiple solutions, one for each unique  $(X, Y)$  pair that makes the goal `father(X, Y)` true.

```
:- setof((X, Y), father(X, Y), S).
   yes S = [(giovanni, mario), (giovanni, giuseppe),
            (mario, paola), (mario, aldo),
            (giuseppe, maria)]
   X = X
   Y = Y
```

The final code snippet describes all the **tuples** retrieved by the Knowledge Base that make the goal `father(X, Y)` true.

## 5. Findall predicate

The `findall` meta-predicate returns the list  $S$  of instances  $X$  for which predicate  $P$  is true. If there is no  $X$  satisfying  $P$ , the meta-predicate returns an empty list.

Its behavior is essentially equal to the **existential quantifier**, it searches for the list  $S$  of instances  $X$  such that there exists  $Y$  that satisfies the predicate  $P$ .

### Example

Given the Knowledge Base:

```
father(mario, aldo).
father(mario, paola).
father(giovanni, mario).
father(giuseppe, maria).
father(giovanni, giuseppe).
```

We want to define which individuals are father.

```
:- findall(X, father(X, Y), S)
   yes S = [mario, giovanni, giuseppe]
   X = X
   Y = Y
```

This code snippet is the same as:

```
:- setof(X, Y^father(X, Y), S)
   yes S = [mario, giovanni, giuseppe]
   X = X
   Y = Y
```

The meta-predicates `setof`, `bagof` and `findall` works also when the property to be verified is not a simple fact, but it is defined by rules.

## Example

Given the Knowledge base.

```
p(X, Y) :- q(X), r(X).

q(0).
q(1).
r(0).
r(2).
```

The `findall` predicate returns the set of instances  $X$  that satisfy the rule `p(X, Y) :- q(X), r(X)`.

```
:- findall(X, p(X, Y), S)
   yes S = [0]
   X = X
   Y = Y
```

## 6. Implication through setof

Let's suppose we have a Knowledge Base containing facts about `father(X, Y)` and `employee(Y)`. We want to verify if it is true that for every  $Y$  for which `father(X, Y)` holds, then  $Y$  is an *employee* (so, basically we are asking if exists  $Y$  such that  $X$  is the father of  $Y$  and it is an *employee*).

In logical terms, this query can be seen as a simple implication.

$$father(X, Y) \rightarrow employee(Y)$$

```
imply(Y) :- findall(Y, father(X, Y), S), verify(S).  
  
verify([]).  
verify([H|T]) :- employee(H), verify(T).
```

First of all, `findall(Y, father(X, Y), S)` returns a list  $S$  containing all the sons already present in the Knowledge Base (remember: the left-most part of the clause is always evaluated first by the Prolog interpreter). The same list  $S$  is used to verify if all the instances  $Y$  are *employee*. If only one of them is not an *employee*, the whole clause `imply(Y)` fails.

## 7. Iteration through setof

Given a Prolog program, we can execute a procedure `q` on each element for which `p` is true.

```
iterate :- setof(X, p(X), L), filter(L).  
  
filter([]).  
filter([H|T]) :- call(q(H)), filter(T).
```