Operators and function in Prolog

07-10-2022

Prolog Operator

- The prolog operator is a function to work arithmetic, logic, comparison, and other operations.
- The prolog operator categorizes several operators for different operations.
- It’s types and its subcategory are given below.

The arithmetic operator types show bellow:

- Addition (+) operator
- Subtraction (-) operator
- Multiplication (*) operator
- Division (%) operator
- Power (**) operator
- Integer division (//) operator
- Modulus (mod) operator
- Square root (sqrt) operator
- maximum (max) operator

```
|?- ADD is 45 + 12.

|?- SUB is 45 - 12.

|?- MUL is 45 * 12.

|?- POW is 12 ^ 2.
```

```
| ?- ADD is 45 + 12.
ADD = 57
yes
```

Comparison operator:

- Greater than (>)operator
- Less than (<) operator
- Greater than equal to (>=) operator
- Less than equal to (=<) operator
- Equal (=:=) operator
- Not equal (=\=) operator

```
?- 13 * 2 > 12 + 13 .
```

```
| ?- 13 * 2 > 12 + 13 .

yes
```

```
?- 13 * 2 < 12 + 13 .
```

```
| ?- 13 * 2 < 12 + 13 .

no
```

Equal (=:=) operator

This operator uses for comparing two variables to “equal to” value operation.

The Equal (=:=) operator example shows below

```
?- 12 + 13 =:= 13 + 12.
```

```
| ?- 12 + 13 =:= 13 + 12.

yes
```

```
?- 12 + 13 =:= 13 + 14.
```

```
| ?- 12 + 13 =:= 13 + 14.

no
```

Create a calc.pl

```
calc :- X is 100 + 200,write('100 + 200 is '),write(X),nl,
        Y is 400 - 150,write('400 - 150 is '),write(Y),nl,
        Z is 10 * 300,write('10 * 300 is '),write(Z),nl,
        A is 100 / 30,write('100 / 30 is '),write(A),nl,
        B is 100 // 30,write('100 // 30 is '),write(B),nl,
        C is 100 ** 2,write('100 ** 2 is '),write(C),nl,
        D is 100 mod 30,write('100 mod 30 is '),write(D),nl.
```

```
yes
| ?- calc.
100 + 200 is 300
400 - 150 is 250
10 * 300 is 3000
100 / 30 is 3.3333333333333335
100 // 30 is 3
100 ** 2 is 10000.0
100 mod 30 is 10

yes
| ?-
```

Loops

Loop statements are used to execute the code block multiple times. In general, for, while, do-while are loop constructs in programming languages (like Java, C, C++).

```
count_to_10(10) :- write(10),nl.

count_to_10(X) :-
  write(X),nl,
  Y is X + 1,
  count_to_10(Y).
```

```
(16 ms) yes
| ?- count_to_10(3).
3
4
5
6
7
8
9
10

true ?
yes
| ?-
```

Create a loop that takes lowest and highest values. So, we can use the between() to simulate loops.

```
count_down(L, H) :-
  between(L, H, Y),
  Z is H - Y,
  write(Z), nl.

count_up(L, H) :-
  between(L, H, Y),
  Z is L + Y,
   write(Z), nl.
```

```
yes
| ?- count_down(12,17).
5

true ? ;
4

true ? ;
3

true ? ;
2

true ? ;
1

true ? ;
0

yes
| ?- count_up(5,12).
10

true ? ;
11

true ? ;
12
```

Decision Making

The decision statements are If-Then-Else statements. So when we try to match some condition, and perform some task, then we use the decision making statements.

If <condition> is true, Then <do this>, Else

```
| ?- gt(10,100).
X is smaller

yes
| ?- gt(150,100).
X is greater or equal
```

```
% If-Then-Else statement

gt(X,Y) :- X >= Y,write('X is greater or equal').
gt(X,Y) :- X < Y,write('X is smaller').

% If-Elif-Else statement

gte(X,Y) :- X > Y,write('X is greater').
gte(X,Y) :- X =:= Y,write('X and Y are same').
gte(X,Y) :- X < Y,write('X is smaller').
```

Inputs and output

The write() Predicate- To write the output we can use the write() predicate. This predicate takes the parameter as input, and writes the content into the console by default. write() can also write in files.

The read() Predicate:- The read() predicate is used to read from console. User can write something in the console, that can be taken as input and process it. The read() is generally used to read from console, but this can also be used to read from files. Now let us see one example to see how read() works.

```
cube :-
   write('Write a number: '),
   read(Number),
   process(Number).
process(stop) :- !.
process(Number) :-
   C is Number * Number * Number,
   write('Cube of '),write(Number),write(': '),write(C),nl, cube.
```

The tab() Predicate

The tab() is one additional predicate that can be used to put some blank-spaces while we write something. So it takes a number as an argument, and prints those many number of blank spaces.

```
| ?- write('hello'),tab(15),write('world').
hello          world

yes
| ?- write('We'),tab(5),write('will'),tab(5),write('use'),tab(5),write('tab
We    will  use   tabs

yes
| ?-
```

Built-In Predicates

Predicate	Description
var(X)	succeeds if X is currently an un-instantiated variable.
novar(X)	succeeds if X is not a variable, or already instantiated
atom(X)	is true if X currently stands for an atom
number(X)	is true if X currently stands for a number
integer(X)	is true if X currently stands for an integer
float(X)	is true if X currently stands for a real number.
atomic(X)	is true if X currently stands for a number or an atom.
compound(X)	is true if X currently stands for a structure.
ground(X)	succeeds if X does not contain any un-instantiated variables.

The var(X) Predicate

When X is not initialized, then, it will show true, otherwise false. So let us see an example.

```
| ?- var(X).

yes
| ?- X = 5, var(X).

no
| ?- var([X]).

no
| ?-
```

The novar(X) Predicate

When X is not initialized, the, it will show false, otherwise true. So let us see an example.

```
| ?- nonvar(X).

no
| ?- X = 5,nonvar(X).

X = 5

yes
| ?- nonvar([X]).

yes
| ?-
```

The atom(X) Predicate

This will return true, when a non-variable term with 0 argument and a not numeric term is passed as X, otherwise false.

```
| ?- atom(paul).

yes
| ?- X = paul,atom(X).

X = paul

yes
| ?- atom([]).

yes
| ?- atom([a,b]).

no
| ?-
```

The number(X) Predicate

This will return true, X stands for any number, otherwise false.

```
| ?- number(X).

no
| ?- X=5,number(X).

X = 5

yes
| ?- number(5.46).

yes
| ?-
```

The integer(X) Predicate

This will return true, when X is a positive or negative integer value, otherwise false.

```
| ?- integer(5).

yes
| ?- integer(5.46).

no
| ?-
```

The float(X) Predicate

This will return true, X is a floating point number, otherwise false.

```
| ?- float(5).

no
| ?- float(5.46).

yes
| ?-
```

The compound(X) Predicate

If atomic(X) fails, then the terms are either one non-instantiated variable (that can be tested with var(X)) or a compound term. Compound will be true when we pass some compound structure.

```
| ?- compound([]).

no
| ?- compound([a]).

yes
| ?- compound(b(a)).

yes
| ?-
```

The ground(X) Predicate

This will return true, if X does not contain any un-instantiated variables. This also checks inside the compound terms, otherwise returns false.

```
| ?- ground(X).

no
| ?- ground(a(b,X)).

no
| ?- ground(a).

yes
| ?- ground([a,b,c]).

yes
| ?-
```

The functor(T,F,N) Predicate

This returns true if F is the principal functor of T, and N is the arity of F.

Note – Arity means the number of attributes.

```
| ?- functor(t(f(X),a,T),Func,N).

Func = t
N = 3

(15 ms) yes
| ?-
```

Mathematical Predicates

Predicates	Description
random(L,H,X).	Get random value between L and H
between(L,H,X).	Get all values between L and H
succ(X,Y).	Add 1 and assign it to X
abs(X).	Get absolute value of X
max(X,Y).	Get largest value between X and Y
min(X,Y).	Get smallest value between X and Y
round(X).	Round a value near to X
truncate(X).	Convert float to integer, delete the fractional part
loor(X).	Round down
ceiling(X).	Round up
sqrt(X).	Square root

```
yes
| ?- random(0,10,X).

X = 1

yes
| ?- between(0,10,X).

X = 0 ? a

X = 1

X = 2

X = 3

X = 4

X = 5
```

Function

- We know that predicates represent always True or False:
- Functions are predicates that represent a value.
- The sin() predicate, for instance, is a function. sin(0) represents the value 0 and sin(1) represents the value 0.841471.
- Functions can be used anywhere a number or constant can be used, in queries, predicates and rules. For instance, if the fact **p(0).** is in your program, the query **?- p(sin(0)).** will unify with it.

One difference between functions and predicates is that the meaning (or definition) of a predicate is usually defined by you, in your program. When you use functions like **sin()**, they've already been defined in your prolog implementation. In other words, prolog will not find the definition in your program, but in it's library of *built-in* predicates. It is possible to create your own functions, but that's something you will usually not need.