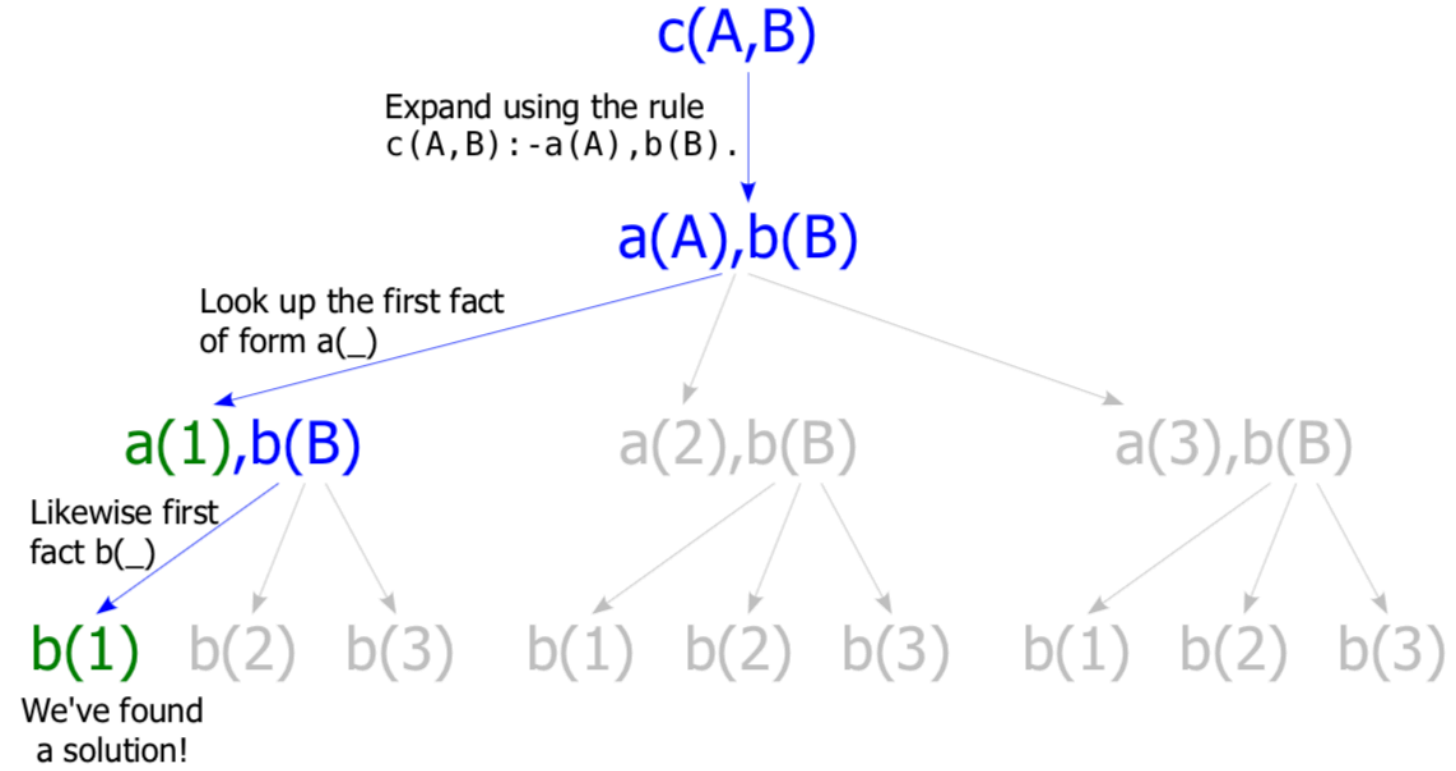
Controlling Execution

Prolog – Finding Answers

Prolog uses depth first search to find answers

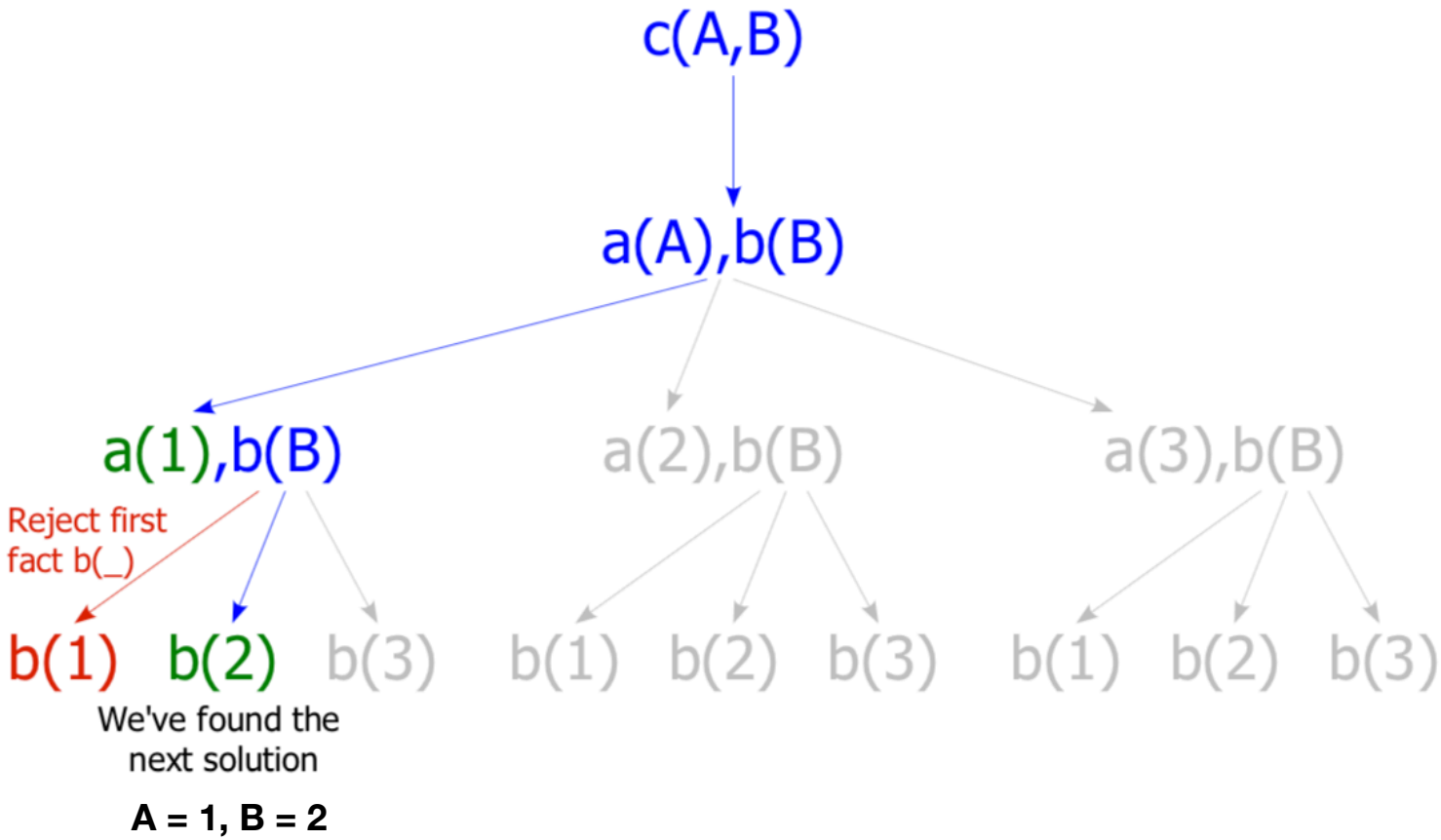
```
a(1).
a(2).
a(3).
b(1).
b(2).
b(3).

c(A, B) :- a(A), b(B).
```

Depth-first solution of query c(A,B)

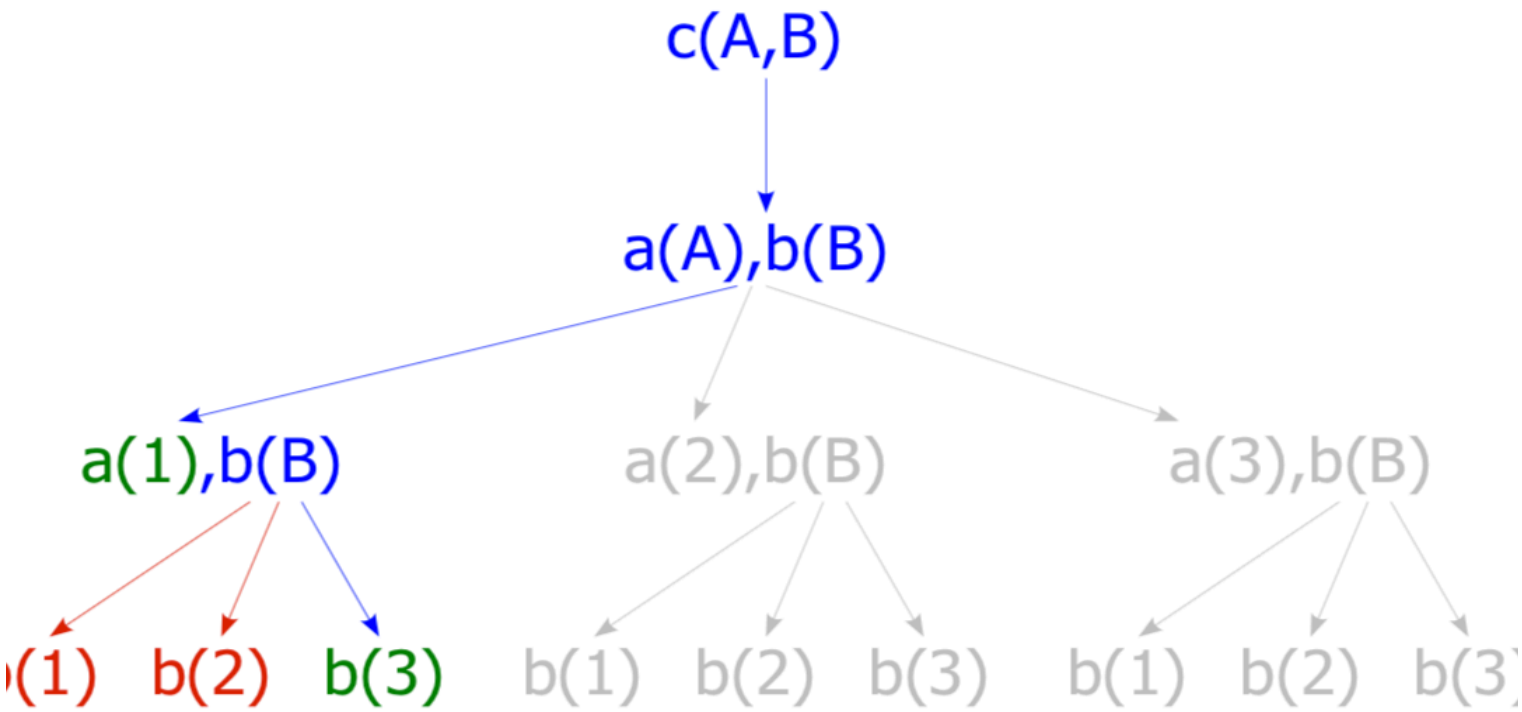


A = 1, B = 1

Backtrack to find another solution

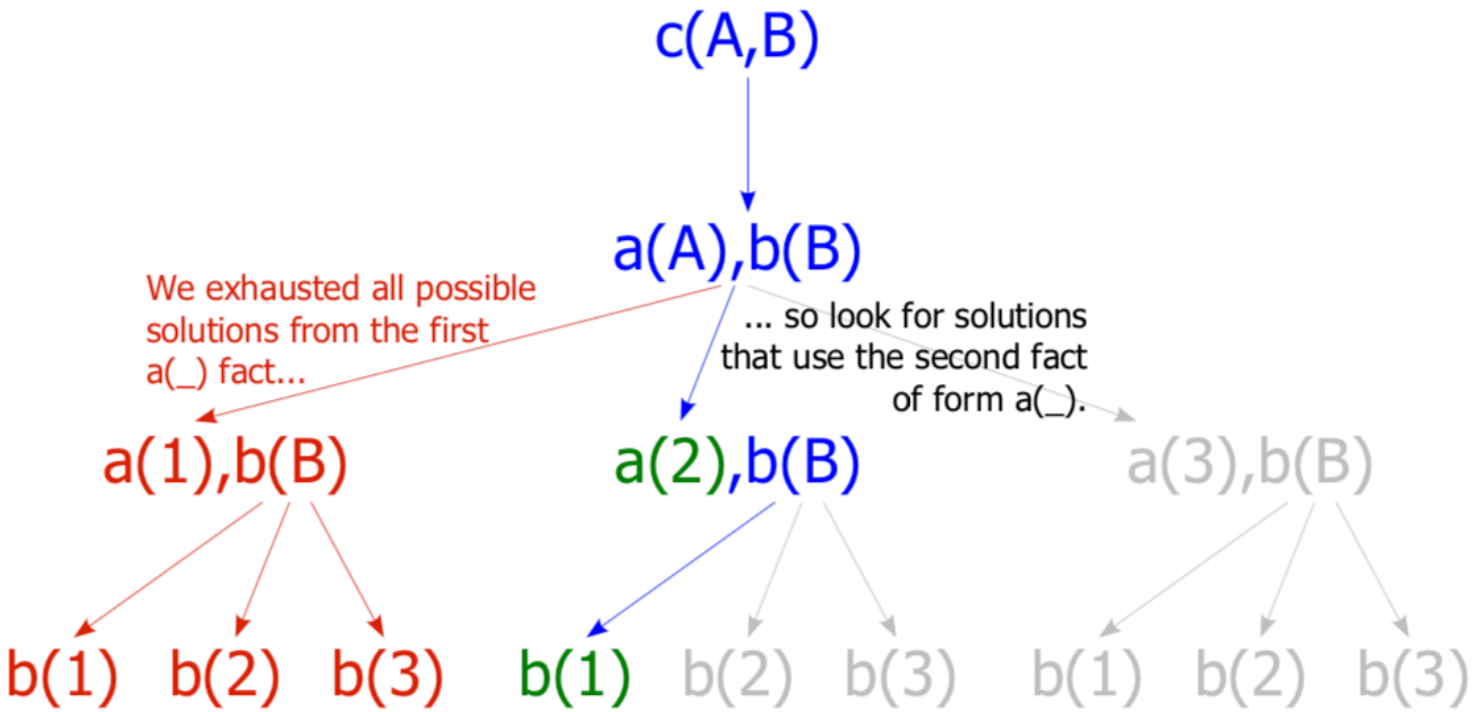


Backtrack to find another solution



A = 1, B = 3

Backtrack to find another solution



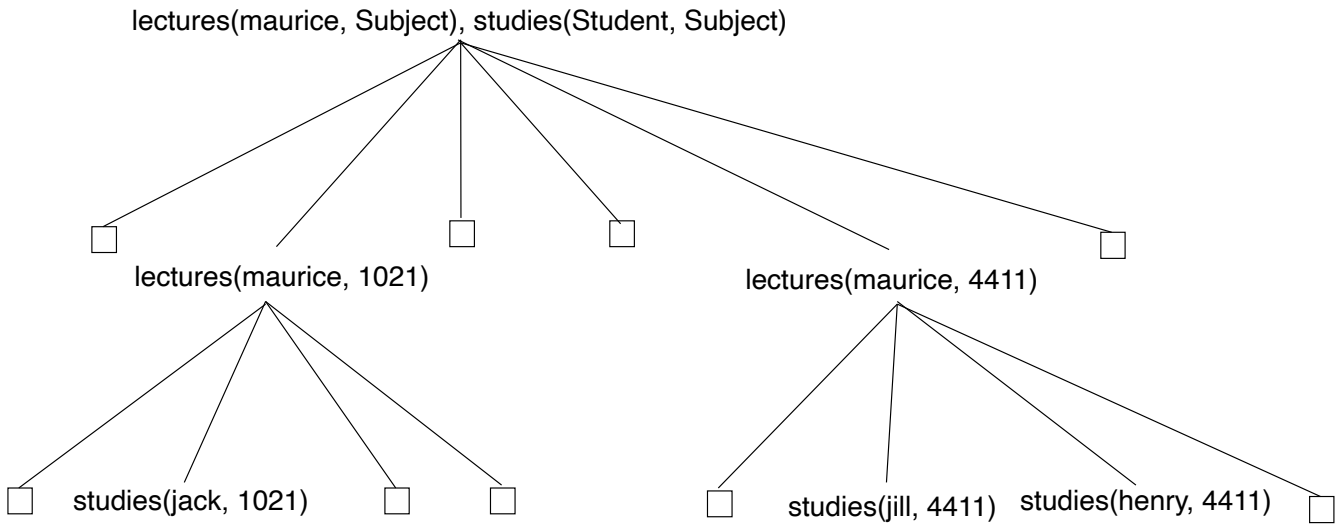
A = 2, B =1

The Cut (!)

- Sometimes we need a way of preventing Prolog from finding all solutions
- The *cut* operator is a built-in predicate that prevents backtracking
- It violates the declarative reading of a Prolog programming
- Use it *VERY sparingly!!!*

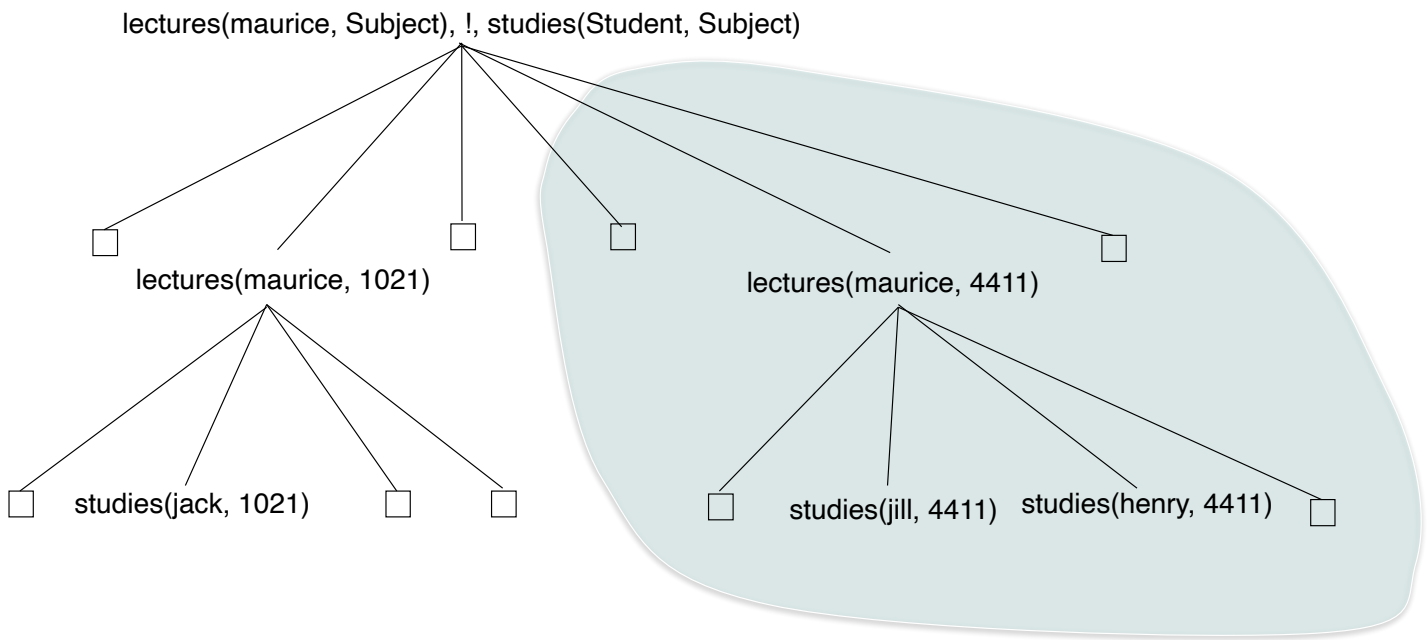
Backtracking

```
lectures(maurice, Subject), studies(Student, Subject)?
Subject = 1021
Student = jack ;

Subject = 4411
Student = Jill ;

Subject = 4411
Student = Henry
```

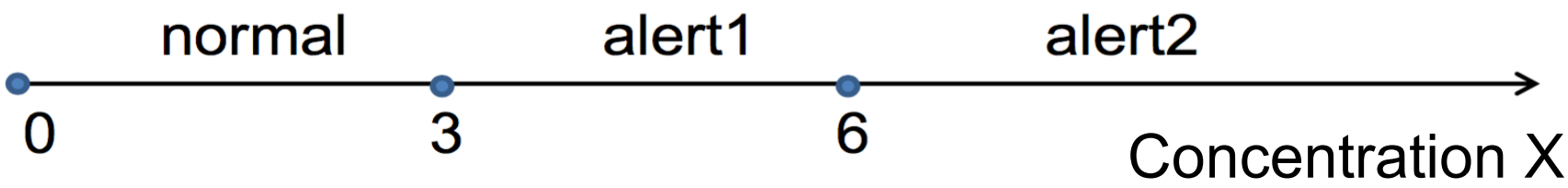


Cut prunes the search

Prevents backtracking to goals left of the cut.



Example



Rules for determining the degree of pollution

Rule 1: if X < 3 then Y = normal

Rule 2: if 3 ≤ X and X < 6 then Y = alert1

Rule 3: if 6 ≤ X then Y = alert2

In Prolog: **`f(Concentration, Pollution_Alert)`**

```
f(X, normal) :-  X < 3.                 % Rule1
f(X, alert1) :-  3 =< X, X < 6.         % Rule2
f(X, alert2) :-  6 =< X.                % Rule3
```

Alternative Version

```
f(X, normal) :-  X < 3, !.          % Rule1
f(X, alert1) :-  X < 6, !.          % Rule2
f(X, alert2).                       % Rule3
```

Which version is easier to read?

Operators

Operator Notation

- Operators are just function (or compound terms)

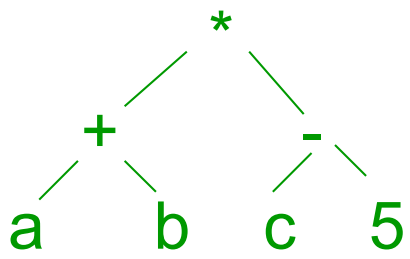
```
2*a + b*c = +(*(2,a), *(b,c))
```

- `+`, `*` are infix operators in Prolog
  - They are only interpreted as arithmetic expressions when the appear on the right-hand side of the ***is*** operator.

Operator Expressions are also Trees

- For example: (a + b) * (c - 5)
- Written as an expression with the functors:

*(+(a, b), -(c, 5))



Operators in Prolog

- You can define your own operators.

  :- op(Precedence, Type, Name).

- Precedence is a number between 0 and 1200.

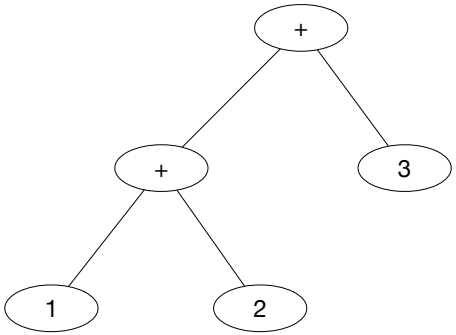
For example,

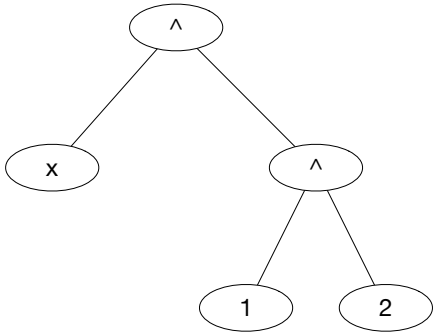
- the precedence of “ = “ is 700,
- the precedence of “ + ” is 500,
- the precedence of “ * “ is 400.

Operators in Prolog

:- op(Precedence, Type, Name).

- Type is an atom specifying the associativity of the operator.
- Infix operators:
  - yfx - left associate (e.g. 1 + 2 + 3 = ((1 + 2) + 3)
  - xfy - right associative (e.g. x ^ 2 ^ 2 = (x ^ (2 ^ 2))
  - xfx - non-associative (e.g. wa = green; a = b= c is not valid)
- Prefix operators
  - fy, fx (associative, non-associative)
- Postfix opertators
  - yf, xf (associative, non-associative)





Predefined Operators

- Operators with the same properties can be specified in one statement by giving a list of their names instead of a single name as third argument of op.
- Operator definitions don't specify the meaning of an operator, only how it can be used syntactically.
- Operator definition doesn't say how a query involving operator is evaluated to true.

```
:- op(1200, xfx, [:-, –>]).
:- op(1200,  fx, [:-, ?-]).
:- op(1100, xfy, [ ; ]).
:- op(1000, xfy, [‘,’]).
:- op( 700, xfx, [=, is, =.., ==. \==, \==, =:=, =\=, <, >, =<, >=]).
:- op( 500, yfx, [+, -]).
:- op( 500,  fx, [+, -]).
:- op( 300, xfx, [ mod ]).
:- op( 200, xfy, [ ^ ]).
```

User Defined Operators

Relations can be defined as operators, e.g.

```
has(peter, information).
supports(floor, table).
```

can be written with operators:

```
:- op(600, xfx, has).
:- op(600, xfx, supports).

peter has information.
floor supports table.
```

Example - IF statement

```
% Appropriate operator declaration
:- op(500, fx, if).
:- op(400, xfx, then).
:- op(300, xfx, else).


% Interpreter

if Condition then S1 else S2 :-
        Condition, !, S1.            % Don't allow backtracking if Condition is true
if Condition then S1 else S2 :-
        S2.
```

Built-in Predicates

- Testing the type of terms
- Construction and decomposition of terms: =. . , functor, arg, name
- Comparison
- *bagof*, *setof* and *findall*
- Input, output

Testing the type of terms

var(X)	true if X is unbound or instantiated to an unbound variable
nonvar(X)	X is not a variable or instantiated to an unbound variable
atom(X)	true if X is an atom
integer(X)	true if X is an integer
float(X)	true if X is a real number
number(X)	true if X is a number
atomic(X)	true if X is a number or an atom
compound(X)	true if X is a compound term (a structure)

Example: Arithmetic Operations

```
...,
number( X),          % Value of  X number?

number( Y),          % Value of  Y number?

 Z is X + Y,         % Then addition it is possible

...
```

Construction and decomposition of terms:

=.. , *functor*, *arg*, *name*

```
Term =.. [Functor, Arg1, Arg2, Arg3, ...]   % “univ”

Term =.. L
```

true if **`L`** is a list that conations the principal functor of **`Term`**, followed by its arguments.

Example:

```
?- f(a, b) =.. L.
L = [f, a, b]

?- T =.. [rectangle, 3,5].
T = rectangle(3, 5)
```

Construction and decomposition of terms:

=.. , *functor*, *arg*, *name*

```
?- functor(a(), N, A).
N = a, A = 0.

?- functor(T, a, 0).
T = a.

?- arg(2, f(a, b), X).
X = b.

?- arg(N, f(a, b), V).
N = 1, V = a ;
N = 2, V = b.
```

Comparison

X = Y	true if X and Y match
X == Y	if X and Y are identical
X \== Y	if X and Y are not identical
X @< Y	X is lexicographically smaller then Y, term X precedes term Y by alphabetical or numerical ordering (e.g. paul @< peter)

findall, bagof, setof

% Find all values of **Object** that satisfy **Condition** and collect in **List**
findall(Object, Condition, List)

% Same as final except only stores unique values and ***fails*** if no solution found
bagof(Object, Condition, List

% Find all values of **Object** that satisfy **Condition** and collect in ***sorted*** **List**
setof(Object, Condition, List)

Example: robot world

```
?- findall(B, on(B,_), L).                    % L is a List of all blocks
L = [a,b,c,d,e]
```

Procedure findall, bagof in setof

Examples:

```
child(joze, ana).     child(miha, ana).
child(lili, ana).     child(lili, andrej).

?- findall(X, child(X, ana), S).
S = [joze, miha, lili]

?- setof(X, child(X, ana), S).
S = [joze, lili, miha]

?- findall(X, child(X, Y), S).
S = [joze, miha, lili, lili]

?- bagof(X, child(X, Y), S).
S = [joze, miha, lili]
Y = ana;
```

Input / Output

```
consult(File)    % Load File into Prolog's database

see(File)        % File becomes the current input stream

see(user)        % user input (i.e. from terminal)

seen             % close the current input stream

seeing(X)        % binds X to the current input file

tell(File)       % File becomes the current output  stream

tell(user)       % user output (i.e. output to terminal)

told             % close the current output stream

telling(X)       % binds X to the current output file
```

Input / Output

```
write(Term)    % write Term to current output stream

writeln(Term) % write Term and append newline

nl             % write newline to current output stream

read(Term)     % read Term from current input stream
```

SWI Prolog Manual

There is a lot more to learn in the user manual:

https://www.swi-prolog.org/pldoc/doc_for?object=manual