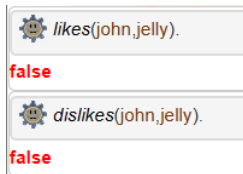


→ Database (left)

```
likes(john,chocolate).  
likes(john,icecream).  
%likes(X,Y):-not(dislikes(X,Y)).  
  
dislikes(john,butter).  
dislikes(john,jam).  
  
%dislikes(X,Y):-not(Likes(X,Y)).
```



→ Queries and respective output (right)

The two responses may seem contradictory, but make perfect sense. By, outputting **false** as a response, the prolog interpreter is trying to say that it is unable to deduce neither of the statements.

This is akin to assuming that whatever information has not been provided is false. This is called the **closed world assumption**.

In some cases this assumption is quite useful. For example, if we are trying to solve a simple graph problem, where we have specified the weights of edges present in the graph. We may choose not to specify edge weights of edges which are not present and in such a situation, prolog interpreter would be correct in assuming that the edge weights which have not been provided, are not part of the input graph.

However in several real life situations, we are not sure if know all the facts pertaining to the problem we are trying to solve.

However in several real life situations, we are not sure if know all the facts pertaining to the problem we are trying to solve.

In those situations this can be an hurdle.

In such situations, one needs to careful with the usage of negation(not).

For example, if we are trying to prove a certain theorem, using certain lemmas, we cannot assume that whatever statements have not been given as lemma's are false.

If one is not satisfied with this interpretation of false, then one may try to fix the interpretation.

In this situation, the facts do not seem to capture the idea that *likes* and *dislikes* capture two opposing notions.

First attempt at fixing the interpretation of false

```
likes(john,chocolate).  
likes(john,icecream).  
likes(X,Y):-not(dislikes(X,Y)).  
  
dislikes(john,butter).  
dislikes(john,jam).  
  
%dislikes(X,Y):-not(likes(X,Y)).
```

	<code>likes(john,butter).</code>
false	
	<code>dislikes(john,jelly).</code>
false	
	<code>likes(john,jelly).</code>
true	

In this first attempt to fix, we may add a rule to capture the idea that if X dislikes Y is true, then X likes Y is false.

Then we get a possibly unintended inference, that john likes jelly.

Second attempt at fixing the interpretation of false

```
likes(john,chocolate).  
likes(john,icecream).  
likes(X,Y):-not(dislikes(X,Y)).  
  
dislikes(john,butter).  
dislikes(john,jam).  
dislikes(X,Y):-not(likes(X,Y)).
```

 likes(john,butter).

false

 dislikes(john,icecream).

false

 likes(john,jelly).

Stack limit (0.2Gb) exceeded

Stack sizes: local: 0.2Gb, global: 57.2Mb, trail: 2Kb

Stack depth: 2,499,298, last-call: 50%, Choice points: 1,249,647

Possible non-terminating recursion:

[2,499,297] system:not(<compound (:)/2>)

[2,499,295] system:not(<compound (:)/2>)

In this second attempt, we may add another rule to capture the idea that if X likes Y is true, then X dislikes Y is false.

The two rules now race around and this time we get trapped in an infinite loop.

There maybe alternative ways to fix this issue, but what these examples are trying to show is that one may not be able to fix all the issues which arise from the difference between the boolean false and the prolog false using finitely many rules.

Contemporary logic programming languages (like prolog) have substantial deficiencies, in that their general “how to” methods can lead them into spurious infinite loops or other undesirable behavior.

```
supervisor(simon,raymond).  
supervisor(skylar,simon).  
supervisor(billy,musk).  
supervisor(raymond,walter).
```

```
job(simon,programmer).  
job(billy,programmer).
```

```
filter1(X):-  
    not(job(X,programmer)),  
    supervisor(X,_).
```

```
filter2(X):-  
    supervisor(X,_),  
    not(job(X,programmer)).
```

 filter1(X).

false

 filter2(X).

X = skylar

X = raymond

filter1 and filter2 are very similar rules.

In boolean logic, $A \wedge B$ and $B \wedge A$ are equivalent. So one might expect filter1 and filter2 to be equivalent.


```
supervisor(simon,raymond).
supervisor(skylar,simon).
supervisor(billy,musk).
supervisor(raymond,walter).
```

```
job(simon,programmer).
job(billy,programmer).
```

```
filter1(X):-
    not(job(X,programmer)),
    supervisor(X,_).
```

```
filter2(X):-
    supervisor(X,_),
    not(job(X,programmer)).
```

 filter1(X).

false

 filter2(X).

X = skylar

X = raymond

filter1 and **filter2** are very similar rules.

In boolean logic, $A \wedge B$ and $B \wedge A$ are equivalent. So one might expect **filter1** and **filter2** to be equivalent.

However, procedural interpretation of the logic provided by the prolog interpreter causes the difference in the output of **filter1** and **filter2**.

- ▶ In case of **filter2**, we first find all possible facts which match this pattern `supervisor(X, ...)` and from these facts (four in this case) we exclude facts, where the job of X is a programmer. Only two facts remain at this point and their corresponding X values are returned.
- ▶ In case of **filter1**, the **not** operator is not designed to add new facts to frame, but it is designed to reduce/remove facts from a frame (or set of facts) which do not meet some criteria.

Since the **not** operator is called first, it starts with an empty frame and passes on an empty frame to the next clause, which checks if X happens to be supervised by somebody. Since the frame is empty, the output is false, implying that no such X was found which met these criterias.

```
parent(indira,rajiv).
parent(indira,sanjay).
parent(sanjay,varun).
parent(rajiv,rahul).
parent(rajiv,priyanka).
parent(feroze,sanjay).
```

```
ancestor(X,Y):-parent(X,Y).
```

```
ancestor(X,Y):-
    parent(X,Z),
    ancestor(Z,Y).
```

```
ancestor2(X,Y):-parent(X,Y).
```

```
ancestor2(X,Y):-
    ancestor2(Z,Y),
    parent(X,Z).
```

```
?- ancestor(indira,rahul).
true .
```

```
?- ancestor2(indira,rahul).
```

```
ERROR: Stack limit (1.0Gb) exceeded
ERROR: Stack sizes: local: 0.9Gb, global: 84.2Mb, trail: 0Kb
ERROR: Stack depth: 11,028,052, last-call: 0%, Choice points: 3
ERROR: Probable infinite recursion (cycle):
ERROR:      [11.028.052] user:ancestor2(_22064460, rahul)
ERROR:      [11.028.051] user:ancestor2( 22064480, rahul)
```

One can notice this issue on swi-prolog. ancestor and ancestor2 are very similar looking rules.





One of the rules gets stuck in infinite loop, while the other correctly answers that indira is an ancestor of rahul.

Reason:-

In the first rule, we first find all relations of the form `parent(X,Z)` and use that as a base to filter out for the condition `ancestor(Z,Y)`.

Whereas in the second rule, `ancestor2(X,Y)` is replaced by another query of the same nature `ancestor2(Z,Y)` which traps it in the infinite loop.

The prolog we are studying is not capturing the boolean logic in a way we perceive it. For example, one could feed *seemingly* contradicting information to the database and that could create some unexpected implications.

<pre>likes(malcolm,mango).</pre> <pre>dislikes(malcolm,mango).</pre> <pre>dislikes(X,Y):-not(likes(X,Y)).</pre>	<div>  likes(malcolm,mango). </div> <div>true</div> <div>  dislikes(malcolm,mango). </div> <div>true</div> <div> <div>Next</div> <div>10</div> <div>100</div> <div>1,000</div> <div>Stop</div> </div>	<div>  likes(malcolm,mango). </div> <div>true</div> <div>  dislikes(malcolm,mango). </div> <div>true</div> <div>false</div>

For the second query, it has two different answers. The fault lies with the database which allows such possibilities. Based on the fact 'dislikes(malcolm,mango)', the first response is **true**, whereas the rule for dislikes gives the second response **false** for the query 'dislikes(malcolm,mango)'.

```
likes(malcolm,mango).  
likes(milton,mango).  
likes(john,orange).  
  
dislikes(malcolm,orange).  
dislikes(john,mango).  
dislikes(X,Y):-not(likes(X,Y)).  
  
% If at least one among X,Y likes Z, they buy Z  
  
bought-at-shop(X,Y,Z):-  
    likes(X,Z);           % usage of OR  
    likes(Y,Z).
```

	bought-at-shop(john,malcolm,mango)
true	
	bought-at-shop(john,malcolm,banana)
false	

The semi-colon denotes that if at least one of the clauses evaluate to true, then the rule is true.

Since neither john nor malcolm can be deduced to like banana, the second query returns false.

One could also capture the logical OR without using a semi-colon, by writing two separate rules.

% alternative for OR

```
bought-at-shop(X,Y,Z):-  
    likes(X,Z).  
bought-at-shop(X,Y,Z):-  
    likes(Y,Z).
```

Understanding the logical OR operator is easy, however implementing the prolog OR operator is tedious, as one requires to look at all possible rules to assess the truth value of a query.

One could also capture the logical OR without using a semi-colon, by writing two separate rules.

% alternative for OR

```
bought-at-shop(X,Y,Z):-  
    likes(X,Z).  
bought-at-shop(X,Y,Z):-  
    likes(Y,Z).
```

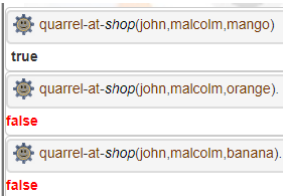
Understanding the logical OR operator is easy, however implementing the prolog OR operator is tedious, as one requires to look at all possible rules to assess the truth value of a query.

This implementation would be akin to spawning several non-deterministic turing machines and running them in parallel for every rule that might get satisfied and returning those rules/facts that get satisfied.


```
likes(malcolm,mango).  
likes(malcolm,orange).  
dislikes(john,mango).  
dislikes(john,banana).
```

```
% If only one among X,Y likes Z,  
% and the other dislikes, they quarrel
```

```
quarrel-at-shop(X,Y,Z):-  
    ( dislikes(Y,Z),  
      likes(X,Z)  
    );  
    ( dislikes(X,Z),  
      likes(Y,Z)  
    ).
```



In this case, the comma tries to mimic the behaviour of the logical AND operator.

- ▶ In the first query, malcolm likes mango and john dislikes mango, thus the query returns true.
- ▶ In the second query, one cannot deduce that john dislikes orange and in the third query, one cannot deduce that malcolm likes banana.

Thus the second and third query both return false to denote prolog's inability to prove these claims using the given data.