The Closed World Assumption

In Prolog, `Yes` means a statement is *provably true*. Consequently, `No` means a statement is *not provably true*. This only means that such a statement is *false*, if we assume that all relevant information is present in the respective Prolog program.

For the semantics of Prolog programs we usually do make this assumption. It is called the *Closed World Assumption*: we assume that nothing outside the world described by a particular Prolog program exists (is true).

The \+-Operator

If we are not interested whether a certain goal succeeds, but rather whether it fails, we can use the \+-operator (negation). `\+ Goal` succeeds, if `Goal` fails (and vice versa). Example:

```
?- \+ member(17, [1, 2, 3, 4, 5]).
Yes
```

This is known as *negation as failure:* Prolog's negation is defined as the failure to provide a proof.

Negation as Failure: Example

Consider the following program:

```
married(peter, lucy).
married(paul, mary).
married(bob, juliet).
married(harry, geraldine).

single(Person) :-
  \+ married(Person, _),
  \+ married(_, Person).
```

Example (cont.)

After compilation Prolog reacts as follows:

```
?- single(mary).
No

?- single(claudia).
Yes
```

In the closed world described by our Prolog program Claudia has to be single, because she is not known to be married.

Where to use \+

Note that the \+-operator can only be used to negate *goals*. These are either (sub)goals in the *body of a rule* or (sub)goals of a *query*. We cannot negate facts or the heads of rules, because this would actually constitute a redefinition of the \+-operator (in other words an explicit definition of Prolog's negation, which wouldn't be compatible with the closed world assumption).

Disjunction

We already know *conjunction* (comma) and *negation* (`\+`). We also know *disjunction*, because several rules with the same head correspond to a disjunction.

Disjunction can also be implemented directly within one rule by using `;` (semicolon). Example:

```
parent(X, Y) :- father(X, Y); mother(X, Y).
```

This is equivalent to the following program:

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

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