UNIFICATION & INSTANTIATION

Deduction in prolog is based on the Unification and Instantiation. Let's understand these terminologies by examples rather than by definitions. Remember one thing, matching terms are unified and variables get instantiated. In other words, "Unification leads to Instantiation".

Example : Let's say you have prolog program with two clauses
- `(1) studies(charlie, csc135). (2) studies(olivia, csc135).`

Then the query `?- studies(charlie, csc135).` unifies with 'fact' studies(charlie, csc135) because terms match with each other but when you have query `?- studies(charlie, X).` then first unification takes place - terms are matched and then variable X gets instantiated to csc135.

Example 1: Let's see for below prolog program - how unification and instantiation take place after querying.

Facts :

```
    likes(john, jane).
    likes(jane, john).
```

Query :

```
    ?- likes(john, X).
```

Answer : `X = jane.`

Here upon asking the query first prolog start to search matching terms in 'Facts' in top down manner for 'likes' predicate with two arguments and it can match `likes(john, ...)` i.e. Unification. Then it looks for the value of X asked in query and it returns answer X = jane i.e. Instantiation - X is instantiated to 'jane'.

Example 2 : At the prolog query prompt, when you write below query,

```
?- owns(X, car(bmw)) = owns(Y, car(C)).
```

You will get Answer : `X = Y, C = bmw.`

Here `owns(X, car(bmw))` and `owns(Y, car(C))` unifies -- because (i) predicate names '`owns`' are same on both side (ii) number of arguments for that predicate, i.e. 2, are equal both side. (iii) 2nd argument with '`car`' predicate inside the brackets are same both side and even in that predicate again number of arguments are same. So, here terms unify in which X=Y. So, Y is substituted with X -- i.e. written as {X | Y} and C is instantiated to bmw, -- written as {bmw | C} and this is called **Unification with Instantiation**.

But when you write *?- owns(X, car(bmw)) = likes(Y, car(C)).* then prolog will return 'false' since it can not match the 'owns' and 'likes' predicates.

Example:

The built in Prolog operator `'='` can be used to unify two terms. Below are some examples of its use. Annotations are between `**` symbols.

```
 | ?- a = a. ** Two identical atoms unify **
yes
 | ?- a = b. ** Atoms don't unify if they aren't identical ** no
 | ?- X = a. ** Unification instantiates a variable to an atom **  X=a
yes
 | ?- X = Y. ** Unification binds two differently named variables **
X=_125451 ** to a single, unique variable name **  Y=_125451
yes
 | ?- foo(a,b) = foo(a,b). ** Two identical complex terms unify ** yes
 | ?- foo(a,b) = foo(X,Y). ** Two complex terms unify if they are **  X=a
** of the same arity, have the same  principal**
 Y=b ** functor and their arguments unify ** yes
 | ?-  foo(a,Y) = foo(X,b). ** Instantiation of variables may occur **  Y=b
** in either of the terms to be unified **  X=a
yes
 | ?- foo(a,b) = foo(X,X). ** In this case there is no unification ** no **
because foo(X,X) must have the same **  ** 1st and 2nd arguments **
 | ?- 2*3+4 = X+Y. ** The term 2*3+4 has principal functor + **  X=2*3
** and therefore unifies with X+Y with X  instantiated**
 Y=4 ** to 2*3 and Y instantiated to 4 ** yes
 | ?- [a,b,c] = [X,Y,Z]. ** Lists unify just like other terms **
X=a
 Y=b
 Z=c
yes
 | ?- [a,b,c] = [X|Y]. ** Unification using the '|' symbol can be used **  X=a
** to find the head element, X, and tail list, Y, **  Y=[b,c] ** of a list **
yes
 | ?- [a,b,c] = [X,Y|Z]. ** Unification on lists doesn't have to be **  X=a
** restricted to finding the first head element **  Y=b ** In this case we
find the 1st and 2nd elements **  Z=[c] ** (X and Y) and then the tail list
(Z) ** yes
 | ?- [a,b,c] = [X,Y,Z|T]. ** This is a similar example but here **  X=a **
the first 3 elements are unified with **  Y=b ** variables X, Y and Z,
leaving the **  Z=c ** tail, T, as an empty list [] **  T=[]
yes
 | ?- [a,b,c] = [a|[b|[c|[]]]]. ** Prolog is quite happy to unify these **
yes ** because they are just notational **  ** variants of the same Prolog
term **
```

Resolution : In simple words resolution is inference mechanism. Let's say we have clauses m :- b. and t :- p, m, z. So from that we can infer t :- p, b, z. - that is called resolution. Means, when you resolve two clauses you get one new clause. Another easy example, we have two sentences (1) *All women like shopping.* (2) *Olivia is a woman*. Now we ask query 'Who likes shopping'. So, by resolving above sentences we can have one new sentence *Olivia likes shopping*.

Refutation : Resolution as Refutation - means proof by contradiction using resolution. Like for every proof by contradiction, we start with assuming and proving that opposite of the given will be true and then we show that this will lead to the contradiction.

Resolution is one kind of proof technique that works this way - (i) select two clauses that contain conflicting terms (ii) combine those two clauses and (iii) cancel out the conflicting terms.

For example we have following **statements**,

(1) If it is a pleasant day you will do strawberry picking

(2) If you are doing strawberry picking you are happy.

Above statements can be written **in propositional logic** like this -

```
(1) strawberry_picking ← pleasant
(2) happy ← strawberry_picking
```

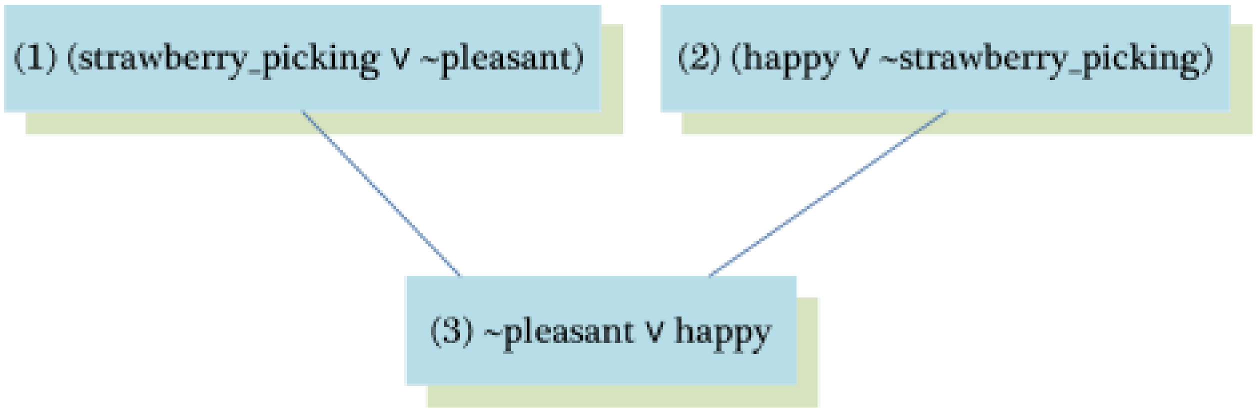
And again these statements can be written **in CNF** like this -

```
(1) (strawberry_picking V~pleasant) ∧
(2) (happy V~strawberry_picking)
```

By resolving these two clauses and cancelling out the conflicting terms 'strawberry_picking' and '~strawberry_picking', we can have one **new clause**,

```
(3) ~pleasant Vhappy
```

How ? See the **figure below**.


(1) (strawberry_picking ∨ ~pleasant)
(2) (happy ∨ ~strawberry_picking)
(3) ~pleasant ∨ happy

But sometimes from the collection of the statements we have, we want to know the answer of this question - "Is it possible to prove some other statements from what we actually know?" In order to prove this we need to make some inferences and those other statements can be shown true using **Refutation** proof method i.e. proof by contradiction using Resolution. So for the asked goal we will negate the goal and will add it to the given statements to prove the contradiction.

Let's see an **example** to understand how Resolution and Refutation work. In below example, **Part(I)** represents the English meanings for the clauses. **Part(II) : Other statements we want to prove by Refutation(Inference)**

(Goal 1) You are not doing strawberry picking.

(Goal 2) You will enjoy.

(Goal 3) Try it yourself : You will get wet.

Part(I) : English Sentences

(1) If it is sunny and warm day you will enjoy.

(2) If it is warm and pleasant day you will do strawberry picking

(3) If it is raining then no strawberry picking.

(4) If it is raining you will get wet.

(5) It is warm day

(6) It is raining

(7) It is sunny

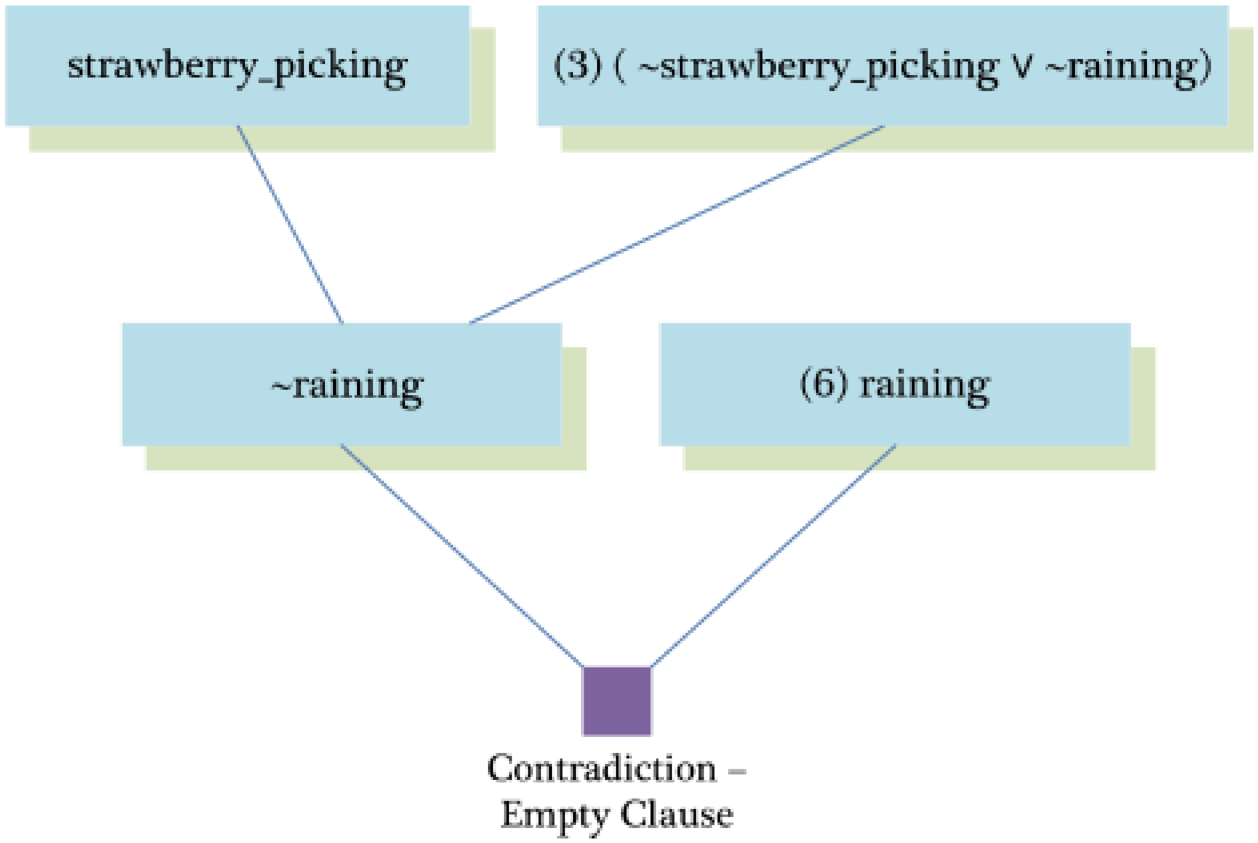
Part(II) : Other statements we want to prove by Refutation

(Goal 1) You are not doing strawberry picking.

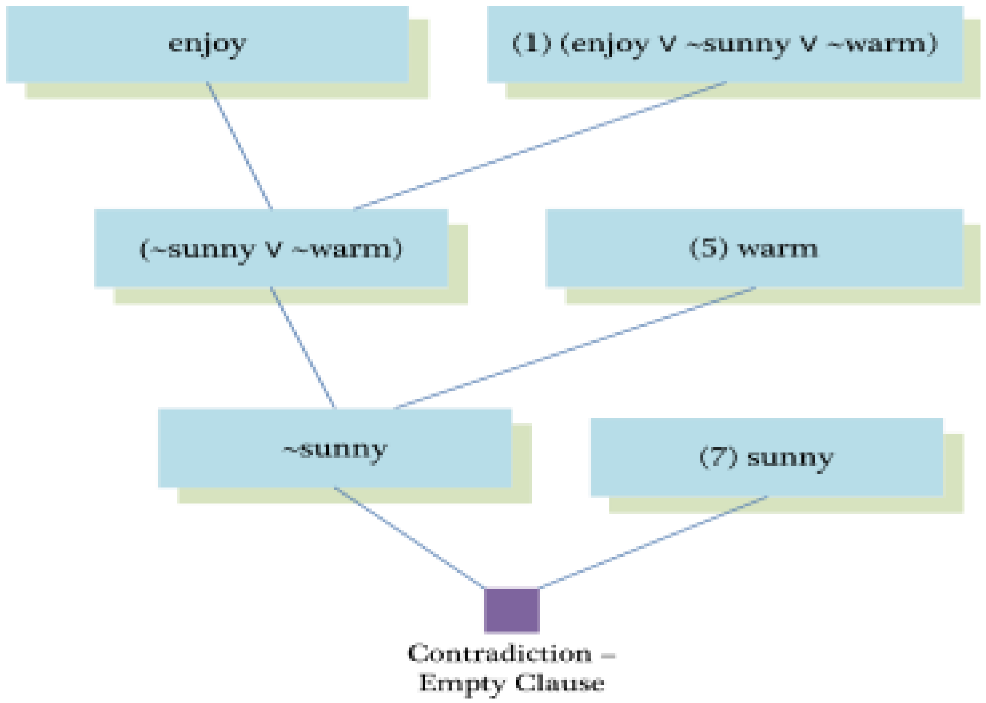
(Goal 2) You will enjoy.

(Goal 3) Try it yourself : You will get wet.

Goal 1 : You are not doing strawberry picking.
Prove : ~strawberry_picking
Assume : strawberry_picking (negate the goal and add it to given clauses).



Goal 2 : You will enjoy.
Prove : enjoy
Assume : ~enjoy (negate the goal and add it to given clauses)


enjoy
(1) (enjoy ∨ ~sunny ∨ ~warm)
(~sunny ∨ ~warm)
(5) warm
~sunny
(7) sunny
Contradiction –
Empty Clause