KNOWLEDGE REPRESENTATION First Order Predicate Logic – Prolog Programming – Unification – Forward Chaining-Backward Chaining – Resolution – Knowledge Representation – Ontological Engineering-Categories and Objects – Events – Mental Events and Mental Objects – Reasoning Systems for Categories – Reasoning with Default Information

Propositional logic in Artificial intelligence:

Propositional logic (PL) is the simplest form of logic where all the statements are made by propositions. A proposition is a declarative statement which is either true or false. It is a technique of knowledge representation in logical and mathematical form.

Example:

1. It is Sunday.
2. The Sun rises from West (False proposition)
3. 3+3= 7(False proposition)
4. 5 is a prime number.

Tautology: A proposition formula which is always true is called tautology, and it is also called a valid sentence.
Contradiction: A proposition formula which is always false is called **Contradiction**.
Statements which are questions, commands, or opinions are not propositions such as "**Where is Rohini**", "**How are you**", "**What is your name**", are not propositions.

Propositional Logic Connectives:

Connective symbols	Word	Technical term	Example
∧	AND	Conjunction	A ∧ B
∨	OR	Disjunction	A ∨ B
→	Implies	Implication	A → B
⇔	If and only if	Biconditional	A⇔ B
¬ or ~	Not	Negation	¬ A or ¬ B

Truth Table:

Combine all the possible combination with logical connectives, and the representation of these combinations in a tabular format is called **Truth table**. Following are the truth table for all logical connectives:

For Negation:

P	¬ P
True	**False**
False	**True**

For Conjunction:

P	Q	P∧ Q
True	True	**True**
True	False	**False**
False	True	**False**
False	False	**False**

For disjunction:

P	Q	P ∨ Q.
True	True	**True**
False	True	**True**
True	False	**True**
False	False	**False**

For Implication:

P	Q	P→ Q
True	True	**True**
True	False	**False**
False	True	**True**
False	False	**True**

For Biconditional:

P	Q	P⇔ Q
True	True	**True**
True	False	**False**
False	True	**False**
False	False	**True**

Precedence of connectives:

Just like arithmetic operators, there is a precedence order for propositional connectors or logical operators. This order should be followed while evaluating a propositional problem. Following is the list of the precedence order for operators:

Precedence	Operators
First Precedence	Parenthesis
Second Precedence	Negation
Third Precedence	Conjunction(AND)
Fourth Precedence	Disjunction(OR)
Fifth Precedence	Implication
Six Precedence	Biconditional

Properties of Operators:

- **Commutativity:**
  - P∧ Q= Q ∧ P, or
  - P ∨ Q = Q ∨ P.
- **Associativity:**
  - (P ∧ Q) ∧ R= P ∧ (Q ∧ R),
  - (P ∨ Q) ∨ R= P ∨ (Q ∨ R)
- **Identity element:**
  - P ∧ True = P,
  - P ∨ True= True.
- **Distributive:**
  - P∧ (Q ∨ R) = (P ∧ Q) ∨ (P ∧ R).
  - P ∨ (Q ∧ R) = (P ∨ Q) ∧ (P ∨ R).
- **DE Morgan's Law:**
  - ¬ (P ∧ Q) = (¬P) ∨ (¬Q)
  - ¬ (P ∨ Q) = (¬ P) ∧ (¬Q).
- **Double-negation elimination:**
  - ¬ (¬P) = P.

Limitations of Propositional logic:

- We cannot represent relations like ALL, some, or none with propositional logic. Example:

  a. **All the girls are intelligent.**
  b. **Some apples are sweet.**

Propositional logic has limited expressive power.

In propositional logic, we cannot describe statements in terms of **their properties or logical relationships.**

PL logic is not sufficient, so we required some more powerful logic, such as first-order logic.

First-Order logic:

- First-order logic is another way of knowledge representation in artificial intelligence. It is an **extension to propositional logic**.
- FOL is sufficiently expressive to represent the natural language statements in a concise way.
- First-order logic is also known as **Predicate logic or First-order predicate logic**. First-order logic is a powerful language that develops information about the objects in a more easy way and can also **express the relationship** between those objects.
- As a natural language, first-order logic also has two main parts:
    a. **Syntax**
    b. **Semantics**
- The syntax of FOL determines which collection of symbols is a logical expression in first-order logic. The basic syntactic elements of first-order logic are symbols.

Basic Elements of First-order logic:

Following are the basic elements of FOL syntax:

Constant	1, 2, A, John, Mumbai, cat,....
Variables	x, y, z, a, b,....
Predicates	Brother, Father, >,....
Function	sqrt, LeftLegOf, ...
Connectives	∧, ∨, ¬, ⇒, ⇔
Equality	==
Quantifier	∀, ∃

Atomic sentences:

- Atomic sentences are the **most basic** sentences of first-order logic. These sentences are formed from a predicate symbol followed by a parenthesis with a sequence of terms.
- We can represent atomic sentences as **Predicate (term1, term2, ..., term n)**.

Example: Ravi and Ajay are brothers: => Brothers(Ravi, Ajay).

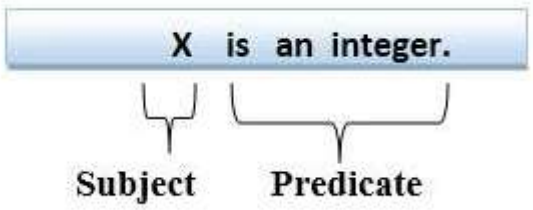
Complex Sentences:

- Complex sentences are made by combining atomic sentences using connectives.

First-order logic statements can be divided into two parts:

- **Subject:** Subject is the main part of the statement.
- **Predicate:** A predicate can be defined as a relation, which binds two atoms together in a statement.

Consider the statement: "x is an integer.", it consists of two parts, the first part x is the subject of the statement and second part "is an integer," is known as a predicate.



Quantifiers in First-order logic:

- A quantifier is a language element which generates **quantification**, and quantification specifies the **quantity** of specimen in the universe of discourse.
- These are the symbols that permit to determine or identify the range and scope of the variable in the logical expression. There are two types of quantifier:

    a. **Universal Quantifier, (for all, everyone, everything)**

    b. **Existential quantifier, (for some, at least one).**

Universal Quantifier:

Universal quantifier is a symbol of logical representation, which specifies that the statement within its range is true for everything or every instance of a particular thing.

The Universal quantifier is represented by a symbol ∀, which resembles an inverted A.

If x is a variable, then ∀x is read as:

- **For all x**
- **For each x**
- **For every x.**

Example:

All man drink coffee.

∀x man(x) → drink (x, coffee).

It will be read as: There are all x where x is a man who drink coffee.

Existential Quantifier:

Existential quantifiers are the type of quantifiers, which express that the statement within its scope is true for at least one instance of something.

It is denoted by the logical operator ∃, which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier.

> Note: In Existential quantifier we always use AND or Conjunction symbol (∧).

If x is a variable, then existential quantifier will be ∃x or ∃(x). And it will be read as:

- **There exists a 'x.'**
- **For some 'x.'**
- **For at least one 'x.'**

Example:

Some boys are intelligent.

∃x: boys(x) ∧ intelligent(x)

It will be read as: There are some x where x is a boy who is intelligent.

Points to remember:

- The main connective for universal quantifier **∀** is implication →.
- The main connective for existential quantifier ∃ is and **∧**.

Properties of Quantifiers:

- In universal quantifier, ∀x∀y is similar to ∀y∀x.
- In Existential quantifier, ∃x∃y is similar to ∃y∃x.
- ∃x∀y is not similar to ∀y∃x.

Prolog Programming:

- Prolog stands for **programming in logic**. In the logic programming paradigm, prolog language is most widely available.

- Prolog is a **declarative language**, which means that a program consists of data based on the facts and rules (Logical relationship) rather than computing how to find a solution.
- A logical relationship describes the relationships which hold for the given application.
- **To obtain the solution**, the user asks a question rather than running a program. When a user asks a question, then to determine the answer, the run time system searches through the database of facts and rules.
- In prolog, logic is expressed as **relations (called as Facts and Rules**). Core heart of prolog lies at the **logic** being applied. Formulation or Computation is carried out by running a query over these relations.

Applications of Prolog:

- Specification Language
- Robot Planning
- Natural language understanding
- Machine Learning
- Problem Solving
- Intelligent Database retrieval
- Expert System
- Automated Reasoning

Syntax and Basic Fields:

- In prolog, we declare **some facts**. These facts constitute the **Knowledge Base** of the system. We can query against the Knowledge Base. We get output as affirmative if our query is already in the knowledge Base or it is implied by Knowledge Base, otherwise we get output as negative.
- So, Knowledge Base can be considered similar to database, against which we can query. Prolog facts are expressed in **definite pattern**.
- Facts contain **entities and their relation**. Entities are written within the **parenthesis** separated by comma (,).Their relation is expressed at the start and outside the parenthesis. Every fact/rule ends with a **dot (.).** So, a typical prolog fact goes as follows:

Format : relation(entity1, entity2, ...k'th entity).

Example:

friends(raju, mahesh).

singer(sonu).

odd_number(5).

Explanation :

These facts can be interpreted as :

raju and mahesh are friends.

sonu is a singer.

5 is an odd number.

Key Features :
1. Unification : The basic idea is, can the given terms be made to represent the same structure.
2. Backtracking : When a task fails, prolog traces backwards and tries to satisfy previous task.
3. Recursion : Recursion is the basis for any search in program.

Advantages :
1. Easy to build database. Doesn't need a lot of programming effort.
2. Pattern matching is easy. Search is recursion based.
3. It has built in list handling. Makes it easier to play with any algorithm involving lists.

Disadvantages :
1. LISP (another logic programming language) dominates over prolog with respect to I/O features.
2. Sometimes input and output is not easy.

Applications :
Prolog is highly used in **artificial intelligence**(AI). Prolog is also used for **pattern matching** over natural language parse trees.

Unification:

- Unification is a process of making two different logical atomic expressions identical by finding a **substitution**.
- Unification depends on the **substitution process**.
- It takes two literals as input and makes them identical using substitution.
- Let Ψ_1 and Ψ_2 be two atomic sentences and σ be a unifier such that, $\boldsymbol{\Psi_1\sigma = \Psi_2\sigma}$, then it can be expressed as **UNIFY(Ψ_1, Ψ_2)**.
- **Example: Find the MGU for Unify{King(x), King(John)}**

Let Ψ_1 = King(x), Ψ_2 = King(John),

Substitution θ = {John/x} is a unifier for these atoms and applying this substitution, and both expressions will be identical.

- The UNIFY algorithm is used for unification, which takes two atomic sentences and **returns a unifier** for those sentences (If any exist).

- Unification is a **key component** of all first-order inference algorithms.
- It returns fail if the expressions do not match with each other.
- The substitution variables are called Most General Unifier or MGU.

E.g. Let's say there are two different expressions, **P(x, y), and P(a, f(z))**.

In this example, we need to make both above statements identical to each other. For this, we will perform the substitution.

P(x, y)..... (i)

P(a, f(z))..... (ii)

- Substitute x with a, and y with f(z) in the first expression, and it will be represented as **a/x** and f(z)/y.
- With both the substitutions, the first expression will be identical to the second expression and the substitution set will be: **[a/x, f(z)/y]**.

Conditions for Unification:

Following are some basic conditions for unification:

- Predicate symbol must be same, atoms or expression with different predicate symbol can never be unified.
- Number of Arguments in both expressions must be identical.
- Unification will fail if there are two similar variables present in the same expression.

Unification Algorithm:

Algorithm: Unify(Ψ_1, Ψ_2)

Step. 1: If Ψ_1 or Ψ_2 is a variable or constant, then:
- a) If Ψ_1 or Ψ_2 are identical, then return NIL.
- b) Else if $\Psi_1$is a variable,
  - a. then if Ψ_1 occurs in Ψ_2, then return FAILURE
  - b. Else return { (Ψ_2/ Ψ_1)}.
- c) Else if Ψ_2 is a variable,
  - a. If Ψ_2 occurs in Ψ_1 then return FAILURE,
  - b. Else return {(Ψ_1/ Ψ_2)}.
- d) Else return FAILURE.

Step.2: If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE.

Step. 3: IF Ψ_1 and Ψ_2 have a different number of arguments, then return FAILURE.

Step. 4: Set Substitution set(SUBST) to NIL.

Step. 5: For i=1 to the number of elements in Ψ_1.

a) Call Unify function with the ith element of Ψ_1 and ith element of Ψ_2, and put the result into S.

b) If S = failure then returns Failure

c) If S ≠ NIL then do,

a. Apply S to the remainder of both L1 and L2.

b. SUBST= APPEND(S, SUBST).

Step.6: Return SUBST.

Implementation of the Algorithm

Step.1: Initialize the substitution set to be empty.

Step.2: Recursively unify atomic sentences:

a. Check for Identical expression match.

b. If one expression is a variable v_i, and the other is a term t_i which does not contain variable v_i, then:

a. Substitute t_i / v_i in the existing substitutions

b. Add t_i /v_i to the substitution setlist.

c. If both the expressions are functions, then function name must be similar, and the number of arguments must be the same in both the expression.

For each pair of the following atomic sentences find the most general unifier (If exist).

EXAMPLE:

1. Find the MGU of {p(f(a), g(Y)) and p(X, X)}

Sol: S_0 => Here, Ψ_1 = p(f(a), g(Y)), and Ψ_2 = p(X, X)
SUBST θ= {f(a) / X}
S1 => Ψ_1 = p(f(a), g(Y)), and Ψ_2 = p(f(a), f(a))
SUBST θ= {f(a) / g(y)}, **Unification failed.**

Unification is not possible for these expressions.

2. Find the MGU of {p(b, X, f(g(Z))) and p(Z, f(Y), f(Y))}

Here, Ψ_1 = p(b, X, f(g(Z))) , and Ψ_2 = p(Z, f(Y), f(Y))
S_0 => { p(b, X, f(g(Z))); p(Z, f(Y), f(Y))}
SUBST θ={b/Z}

S_1 => { p(b, X, f(g(b))); p(b, f(Y), f(Y))}
SUBST θ={f(Y) /X}

S_2 => { p(b, f(Y), f(g(b))); p(b, f(Y), f(Y))}
SUBST θ= {g(b) /Y}

S_2 => { p(b, f(g(b)), f(g(b)); p(b, f(g(b)), f(g(b))} **Unified Successfully.**

And Unifier = { b/Z, f(Y) /X , g(b) /Y}.

3. Find the MGU of {p (X, X), and p (Z, f(Z))}

Here, Ψ_1 = {p (X, X), and Ψ_2 = p (Z, f(Z))
S_0 => {p (X, X), p (Z, f(Z))}
SUBST θ= {X/Z}
S1 => {p (Z, Z), p (Z, f(Z))}
SUBST θ= {f(Z) / Z}, **Unification Failed.**

4. Find the MGU of UNIFY(prime (11), prime(y))

Here, Ψ_1 = {prime(11) , and Ψ_2 = prime(y)}
S_0 => {prime(11) , prime(y)}
SUBST θ= {11/y}
S_1 => {prime(11) , prime(11)} , **Successfully unified.**
Unifier: {11/y}.

5. Find the MGU of Q(a, g(x, a), f(y)), Q(a, g(f(b), a), x)}

Here, Ψ_1 = Q(a, g(x, a), f(y)), and Ψ_2 = Q(a, g(f(b), a), x)
S_0 => {Q(a, g(x, a), f(y)); Q(a, g(f(b), a), x)}
SUBST θ= {f(b)/x}
S_1 => {Q(a, g(f(b), a), f(y)); Q(a, g(f(b), a), f(b))}
SUBST θ= {b/y}
S_1 => {Q(a, g(f(b), a), f(b)); Q(a, g(f(b), a), f(b))}, **Successfully Unified.**
Unifier: [a/a, f(b)/x, b/y].

6. UNIFY(knows(Richard, x), knows(Richard, John))

Here, Ψ_1 = knows(Richard, x), and Ψ_2 = knows(Richard, John)
S_0 => { knows(Richard, x); knows(Richard, John)}
SUBST θ= {John/x}
S_1 => { knows(Richard, John); knows(Richard, John)}, **Successfully Unified.**
Unifier: {John/x}.

Forward Chaining-Backward Chaining:

Inference engine:

The inference engine is the component of the intelligent system in artificial intelligence, which applies logical rules to the knowledge base to infer new information from known facts. The first inference engine was part of the expert system. Inference engine commonly proceeds in two modes, which are:

a) **Forward chaining**
b) **Backward chaining**

A. Forward Chaining

Forward chaining is also known as a forward deduction or forward reasoning method when using an inference engine. Forward chaining is a form of reasoning which start with atomic sentences in the knowledge base and applies inference rules (Modus Ponens) in the forward direction to extract more data until a goal is reached.

The Forward-chaining algorithm starts from known facts, triggers all rules whose premises are satisfied, and add their conclusion to the known facts. This process repeats until the problem is solved.

Properties of Forward-Chaining:

- It is a down-up approach, as it moves from bottom to top.
- It is a process of making a conclusion based on known facts or data, by starting from the initial state and reaches the goal state.

- Forward-chaining approach is also called as data-driven as we reach to the goal using available data.
- Forward -chaining approach is commonly used in the expert system, such as CLIPS, business, and production rule systems.

Consider the following famous example which we will use in both approaches:

Example:

"As per the law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robert, who is an American citizen."

Prove that **"Robert is criminal."**

To solve the above problem, first, we will convert all the above facts into first-order definite clauses, and then we will use a forward-chaining algorithm to reach the goal.

Facts Conversion into FOL:

- It is a crime for an American to sell weapons to hostile nations. (Let's say p, q, and r are variables)
  American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p) ...(1)
- Country A has some missiles. **∃p Owns(A, p) ∧ Missile(p)**. It can be written in two definite clauses by using Existential Instantiation, introducing new Constant T1.

  Owns(A, T1) .....(2)
  Missile(T1) .....(3)
- All of the missiles were sold to country A by Robert.
  ∀p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A) .....(4)

- Missiles are weapons.
  Missile(p) → Weapons (p) .....(5)

- Enemy of America is known as hostile.
  Enemy(p, America) →Hostile(p) .....(6)

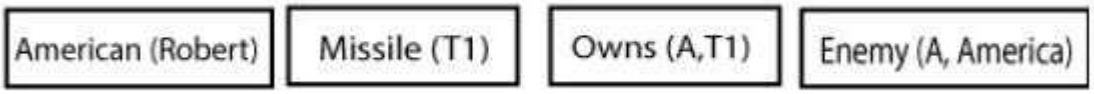
- Country A is an enemy of America.
  Enemy (A, America) .....(7)

- Robert is American
  American(Robert). .....(8)

Forward chaining proof:

Step-1:

In the first step we will start with the known facts and will choose the sentences which do not have implications, such as: **American(Robert), Enemy(A, America), Owns(A, T1), and Missile(T1)**. All these facts will be represented as below.



Step-2:

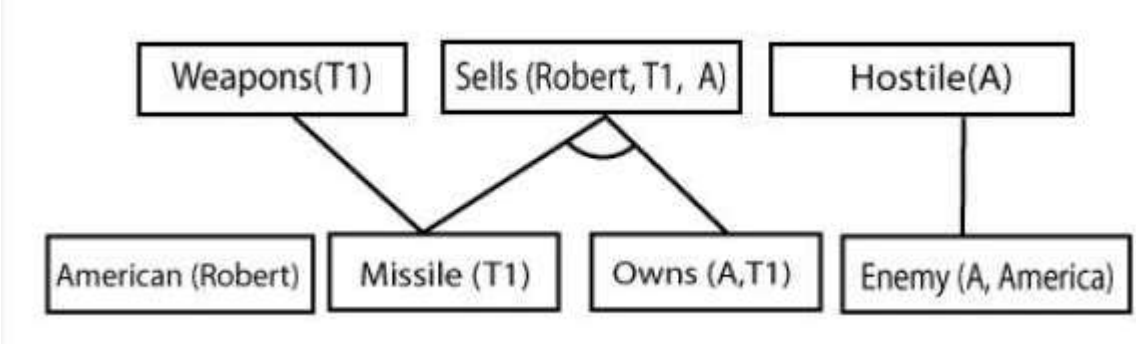
At the second step, we will see those facts which infer from available facts and with satisfied premises.

Rule-(1) does not satisfy premises, so it will not be added in the first iteration.

Rule-(2) and (3) are already added.

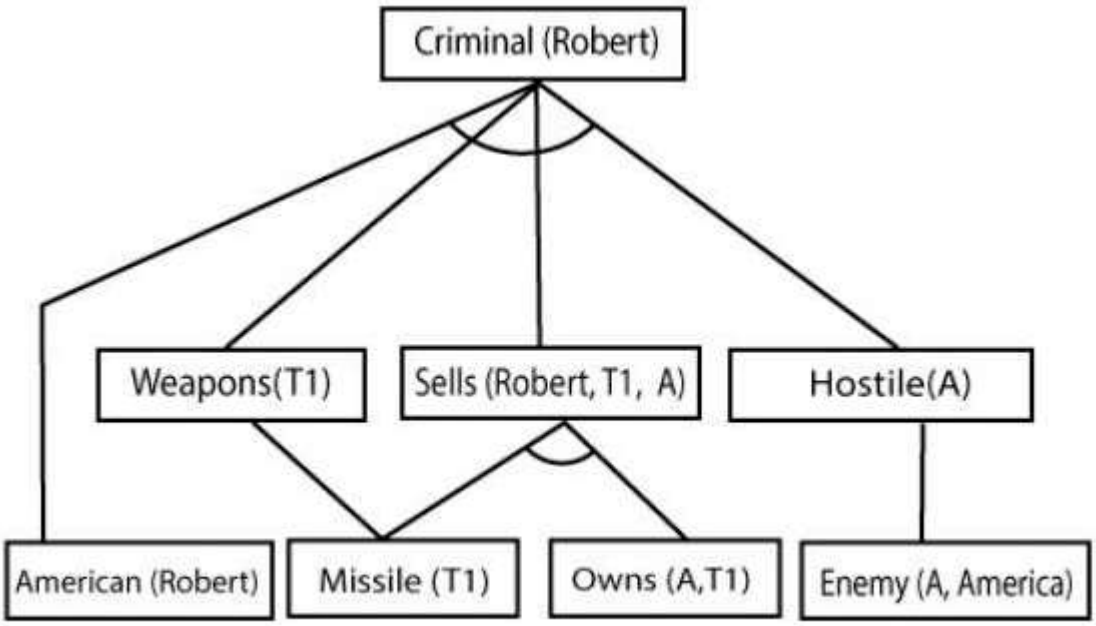
Rule-(4) satisfy with the substitution {p/T1}, **so Sells (Robert, T1, A)** is added, which infers from the conjunction of Rule (2) and (3).

Rule-(6) is satisfied with the substitution(p/A), so Hostile(A) is added and which infers from Rule-(7).



Step-3:

At step-3, as we can check Rule-(1) is satisfied with the substitution **{p/Robert, q/T1, r/A}, so we can add Criminal(Robert)** which infers all the available facts. And hence we reached our goal statement.



B. Backward Chaining:

Backward-chaining is also known as a backward deduction or backward reasoning method when using an inference engine. A backward chaining algorithm is a form of reasoning, which starts with the goal and works backward, chaining through rules to find known facts that support the goal.

Properties of backward chaining:

- It is known as a top-down approach.
- Backward-chaining is based on modus ponens inference rule.
- In backward chaining, the goal is broken into sub-goal or sub-goals to prove the facts true.
- It is called a goal-driven approach, as a list of goals decides which rules are selected and used.
- Backward -chaining algorithm is used in game theory, automated theorem proving tools, inference engines, proof assistants, and various AI applications.
- The backward-chaining method mostly used a **depth-first search** strategy for proof.

Example:

In backward-chaining, we will use the same above example, and will rewrite all the rules.

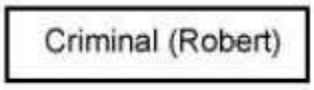
- **American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p) ...(1)**
  Owns(A, T1) .....(2)
- **Missile(T1)**
- **?p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A) .....(4)**
- **Missile(p) → Weapons (p) .....(5)**
- **Enemy(p, America) →Hostile(p) .....(6)**
- **Enemy (A, America) .....(7)**
- **American(Robert). .....(8)**

Backward-Chaining proof:

In Backward chaining, we will start with our goal predicate, which is **Criminal(Robert)**, and then infer further rules.

Step-1:

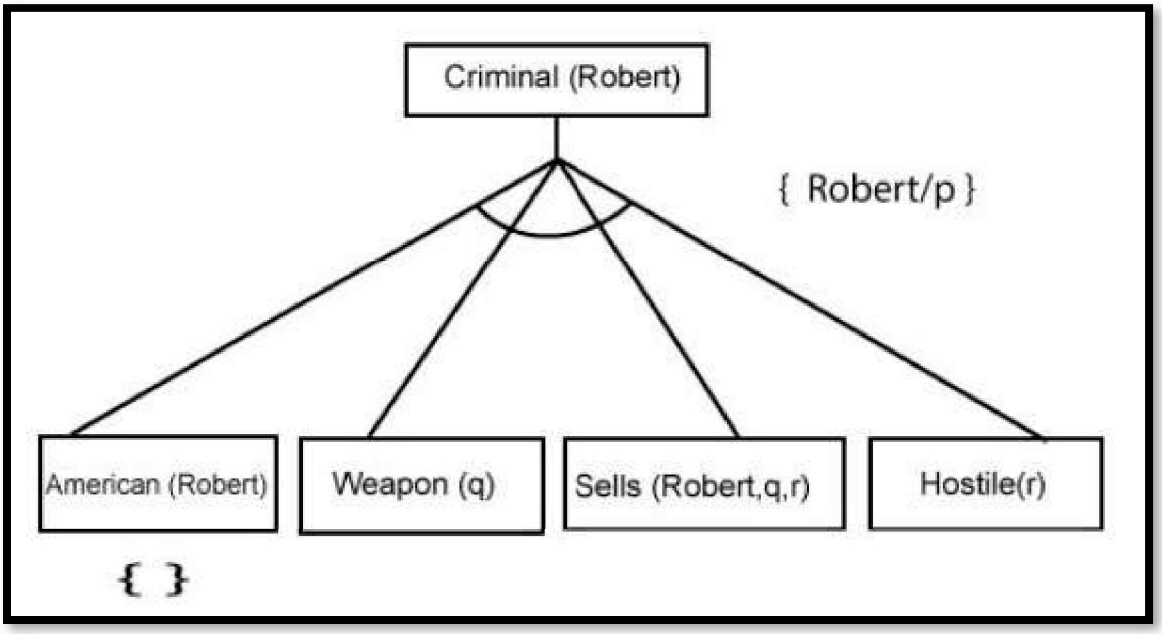
At the first step, we will take the goal fact. And from the goal fact, we will infer other facts, and at last, we will prove those facts true. So our goal fact is "Robert is Criminal," so following is the predicate of it.



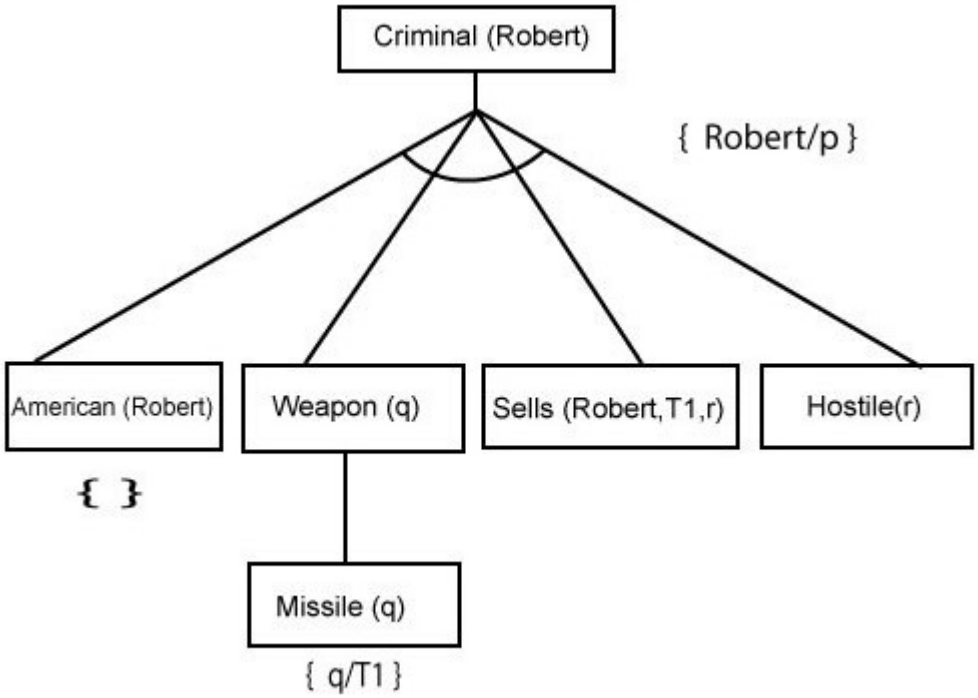
Step-2:

At the second step, we will infer other facts form goal fact which satisfies the rules. So as we can see in Rule-1, the goal predicate Criminal (Robert) is present with substitution {Robert/P}. So we will add all the conjunctive facts below the first level and will replace p with Robert.

Here we can see American (Robert) is a fact, so it is proved here.

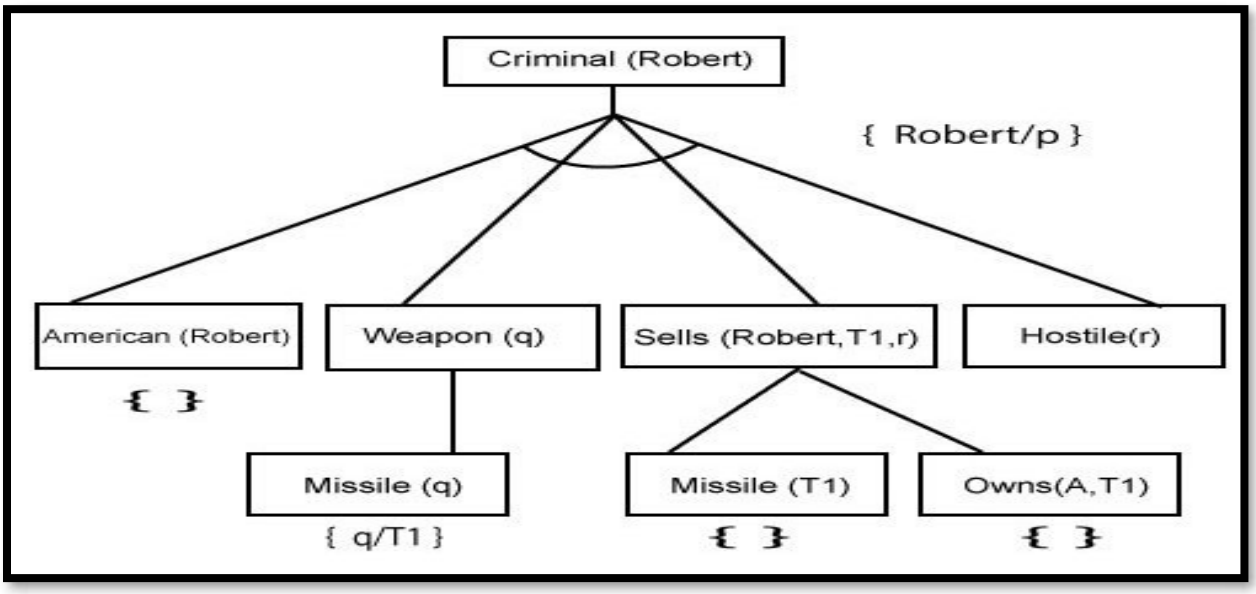


Step-3:t At step-3, we will extract further fact Missile(q) which infer from Weapon(q), as it satisfies Rule-(5). Weapon (q) is also true with the substitution of a constant T1 at q.



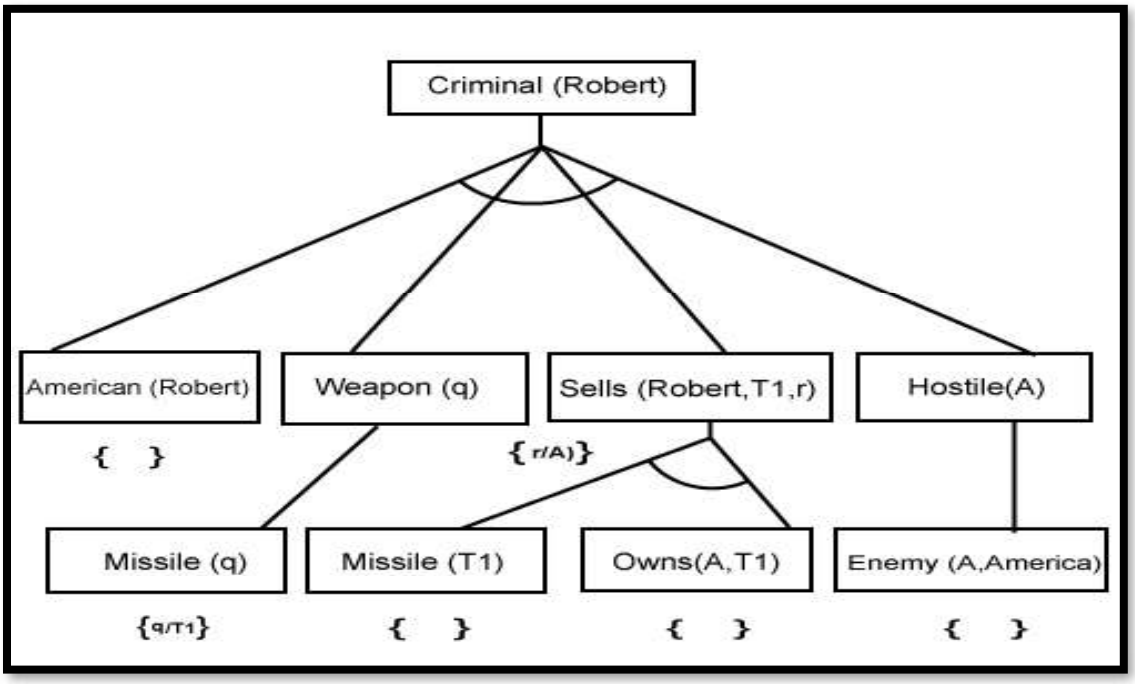
Step-4:

At step-4, we can infer facts Missile(T1) and Owns(A, T1) form Sells(Robert, T1, r) which satisfies the **Rule- 4**, with the substitution of A in place of r. So these two statements are proved here.



Step-5:

At step-5, we can infer the fact **Enemy(A, America)** from **Hostile(A)** which satisfies Rule- 6. And hence all the statements are proved true using backward chaining.



S. No.	Forward Chaining	Backward Chaining
1.	Forward chaining starts from known facts and applies inference rule to extract more data unit it reaches to the goal.	Backward chaining starts from the goal and works backward through inference rules to find the required facts that support the goal.
2.	It is a bottom-up approach	It is a top-down approach
3.	Forward chaining is known as data-driven inference technique as we reach to the goal using the available data.	Backward chaining is known as goal-driven technique as we start from the goal and divide into sub-goal to extract the facts.
4.	Forward chaining reasoning applies a breadth-first search strategy.	Backward chaining reasoning applies a depth-first search strategy.
5.	Forward chaining tests for all the available rules	Backward chaining only tests for few required rules.
6.	Forward chaining is suitable for the planning, monitoring, control, and interpretation application.	Backward chaining is suitable for diagnostic, prescription, and debugging application.
7.	Forward chaining can generate an infinite number of possible conclusions.	Backward chaining generates a finite number of possible conclusions.
8.	It operates in the forward direction.	It operates in the backward direction.
9.	Forward chaining is aimed for any conclusion.	Backward chaining is only aimed for the required data.

Resolution

Resolution is a theorem proving technique that proceeds by building refutation proofs, i.e., proofs by contradictions. It was invented by a Mathematician John Alan Robinson in the year 1965.

Resolution is used, if there are various statements are given, and we need to prove a conclusion of those statements. Unification is a key concept in proofs by resolutions. Resolution is a single inference rule which can efficiently operate on the **conjunctive normal form or clausal form**.

Steps for Resolution:

1. Conversion of facts into first-order logic.
2. Convert FOL statements into CNF
3. Negate the statement which needs to prove (proof by contradiction)
4. Draw resolution graph (unification).

To better understand all the above steps, we will take an example in which we will apply resolution.

Example:

a. **John likes all kind of food.**
b. **Apple and vegetable are food**
c. **Anything anyone eats and not killed is food.**
d. **Anil eats peanuts and still alive**
e. **Harry eats everything that Anil eats.**
   Prove by resolution that:
f. **John likes peanuts.**

Step-1: Conversion of Facts into FOL

In the first step we will convert all the given statements into its first order logic.

a. $\forall x$: food(x) → likes(John, x)
b. food(Apple) ∧ food(vegetables)
c. $\forall x \forall y$: eats(x, y) ∧ ¬ killed(x) → food(y)
d. eats (Anil, Peanuts) ∧ alive(Anil).
e. $\forall x$: eats(Anil, x) → eats(Harry, x)
f. $\forall x$: ¬ killed(x) → alive(x) } **added predicates.**
g. $\forall x$: alive(x) →¬ killed(x) }
h. likes(John, Peanuts)

Step-2: Conversion of FOL into CNF

In First order logic resolution, it is required to convert the FOL into CNF as CNF form makes easier for resolution proofs.

- **Eliminate all implication (→) and rewrite**

  a. $\forall x$ ¬ food(x) V likes(John, x)
  b. food(Apple) ∧ food(vegetables)
  c. $\forall x \forall y$ ¬ [eats(x, y) ∧ ¬ killed(x)] V food(y)
  d. eats (Anil, Peanuts) ∧ alive(Anil)
  e. $\forall x$ ¬ eats(Anil, x) V eats(Harry, x)
  f. $\forall x$¬ [¬ killed(x)] V alive(x)
  g. $\forall x$ ¬ alive(x) V ¬ killed(x)
  h. likes(John, Peanuts).

Move negation (¬)inwards and rewrite

a. ∀x ¬ food(x) V likes(John, x)
b. food(Apple) Λ food(vegetables)
c. ∀x ∀y ¬ eats(x, y) V killed(x) V food(y)
d. eats (Anil, Peanuts) Λ alive(Anil)
e. ∀x ¬ eats(Anil, x) V eats(Harry, x)
f. ∀x ¬killed(x)] V alive(x)
g. ∀x ¬ alive(x) V ¬ killed(x)
h. likes(John, Peanuts).

Rename variables or standardize variables

a. ∀x ¬ food(x) V likes(John, x)
b. food(Apple) Λ food(vegetables)
c. ∀y ∀z ¬ eats(y, z) V killed(y) V food(z)
d. eats (Anil, Peanuts) Λ alive(Anil)
e. ∀w¬ eats(Anil, w) V eats(Harry, w)
f. ∀g ¬killed(g)] V alive(g)
g. ∀k ¬ alive(k) V ¬ killed(k)
h. likes(John, Peanuts).

Eliminate existential instantiation quantifier by elimination.

In this step, we will eliminate existential quantifier ∃, and this process is known as **Skolemization**. But in this example problem since there is no existential quantifier so all the statements will remain same in this step.

Drop Universal quantifiers.

In this step we will drop all universal quantifier since all the statements are not implicitly quantified so we don't need it.

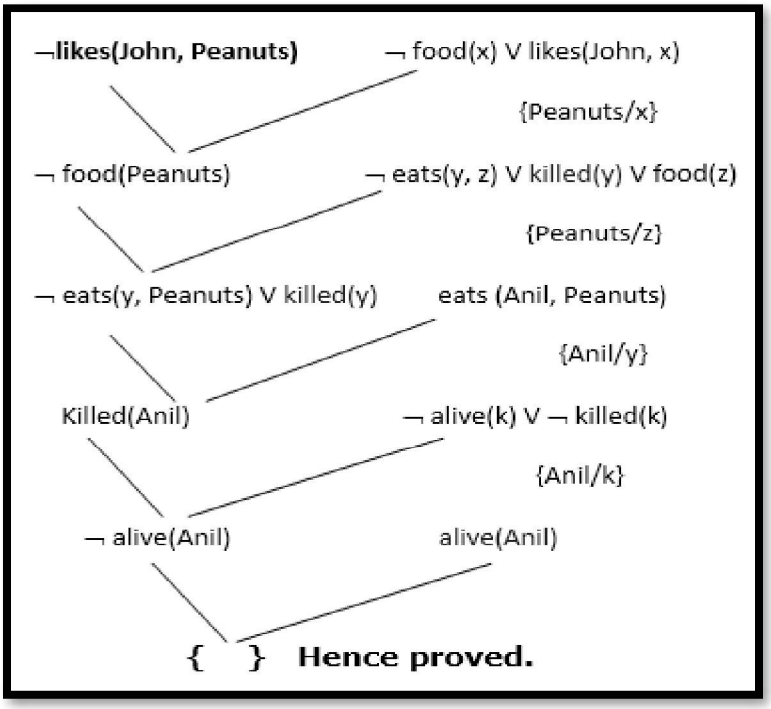
a. ¬ food(x) V likes(John, x)
b. food(Apple)
c. food(vegetables)
d. ¬ eats(y, z) V killed(y) V food(z)
e. eats (Anil, Peanuts)
f. alive(Anil)
g. ¬ eats(Anil, w) V eats(Harry, w)
h. killed(g) V alive(g)
i. ¬ alive(k) V ¬ killed(k)
j. likes(John, Peanuts).

Step-3: Negate the statement to be proved

In this statement, we will apply negation to the conclusion statements, which will be written as ¬likes(John, Peanuts)

Step-4: Draw Resolution graph:

Now in this step, we will solve the problem by resolution tree using substitution. For the above problem, it will be given as follows:



Hence the negation of the conclusion has been proved as a complete contradiction with the given set of statements.

Explanation of Resolution graph:

- In the first step of resolution graph, **¬likes(John, Peanuts)** , and **likes(John, x)** get resolved(canceled) by substitution of **{Peanuts/x}**, and we are left with **¬ food(Peanuts)**
- In the second step of the resolution graph, **¬ food(Peanuts)** , and **food(z)** get resolved (canceled) by substitution of **{ Peanuts/z}**, and we are left with **¬ eats(y, Peanuts) V killed(y)** .
- In the third step of the resolution graph, **¬ eats(y, Peanuts)** and **eats (Anil, Peanuts)** get resolved by substitution **{Anil/y}**, and we are left with **Killed(Anil)** .
- In the fourth step of the resolution graph, **Killed(Anil)** and **¬ killed(k)** get resolve by substitution **{Anil/k}**, and we are left with **¬ alive(Anil)** .
- In the last step of the resolution graph **¬ alive(Anil)** and **alive(Anil)** get resolved.