# UNIT III

**Logic and Knowledge Representation:**

**First-Order Logic:** Representation, Syntax and Semantics of First-Order Logic, Using First-Order Logic, Knowledge Engineering in First-Order Logic.

**Inference in First-Order Logic:** Propositional vs. First-Order Inference, Unification and Lifting, Forward Chaining, Backward Chaining, Resolution.

# First-Order Logic (FOL)

First-Order Logic (also known as Predicate Logic, First-Order Predicate Calculus, or FOPL) is a powerful and highly expressive formal language used in AI for **knowledge representation** and **reasoning**. It extends propositional logic by introducing the ability to quantify over individuals and to express relationships and properties of objects.

## 1. Representation and Key Components

**FOL allows us to represent knowledge about objects, their properties, and their relationships. Its core components are:**

- **Constants:** Represent specific objects or individuals. They are usually denoted by names starting with an uppercase letter or specific symbols.
  - **Examples:** `John`, `Mary`, `Table3`, `Hyderabad`, `A` (if "A" refers to a specific object).
- **Predicates (Relations):** Represent properties of objects or relationships between objects. They take one or more arguments, which are terms. Predicates evaluate to True or False.
  - Examples:
    - `IsBird(Tweety)` (Tweety has the property of being a bird)
    - `Likes(John, Mary)` (John likes Mary – a binary relation)
    - `IsBetween(Hyderabad, Chennai, Delhi)` (a ternary relation)
    - `Color(Car1, Red)` (Car1 has the property of being Red)
- **Functions:** Represent mappings from one or more objects to another object. Unlike predicates, functions return a value (an object), not a truth value.
  - Examples:
    - `FatherOf(John)` (returns John's father)
    - `Plus(2, 3)` (returns 5)
    - `Legs(Elephant)` (returns the number of legs an elephant has)
    - `Max(x, y)`
- **Variables:** Represent unspecified objects or individuals. They are usually denoted by names starting with a lowercase letter.
  - Examples: `x`, `y`, `person`, `city`.
- **Connectives:** The same logical connectives as in propositional logic:
  - $\neg$ (NOT)
  - $\wedge$ (AND)
  - $\vee$ (OR)
  - $\Rightarrow$ (IMPLIES)
  - $\Leftrightarrow$ (EQUIVALENCE)

- **Quantifiers:** The crucial addition in FOL that allows for expressing general statements about collections of objects.
  - **Universal Quantifier (For all / Every): ∀x**
    - Means **"for all objects x in the domain..."**
    - **Example: ∀x.IsBird(x)⇒Flies(x) (For all x, if x is a bird, then x flies).**
  - **Existential Quantifier (There exists / Some): ∃y**
    - Means **"there exists at least one object y in the domain such that..."**
    - **Example: ∃y.Likes(y,IceCream) (There exists some y such that y likes IceCream).**

## 2. Syntax of First-Order Logic

The syntax defines the rules for constructing well-formed formulas (WFFs) or sentences in FOL.

- **Terms:** Refer to objects.
  - Constants are terms. (e.g., `John`)
  - Variables are terms. (e.g., `x`)
  - A function symbol applied to a list of terms is a term. (e.g., `FatherOf(John)`, `Legs(x)`)
- **Atomic Sentences:** Formed by a predicate symbol applied to a list of terms. An atomic sentence is a proposition that can be true or false.
  - Examples: `IsBird(Tweety)`, `Likes(John, Mary)`, `IsBetween(Hyderabad, Chennai, Delhi)`.
- **Complex Sentences:** Formed by combining atomic sentences or other complex sentences using logical connectives.
  - If S is a sentence, then ¬S is a sentence.
  - If S1 and S2 are sentences, then (S1∧S2), (S1∨S2), (S1⇒S2), and (S1⇔S2) are sentences.
- **Quantified Sentences:** Formed by applying quantifiers to variables and sentences.
  - If S is a sentence and x is a variable, then ∀x.S and ∃x.S are sentences.
  - The variable x in S is said to be **bound** by the quantifier. Variables not bound by any quantifier are **free variables**. A sentence with no free variables is called a **closed formula** or **ground sentence**.

**Example of a FOL Sentence:** ∀x.(Human(x)⇒Mortal(x)) (All humans are mortal)
∃y.(Dog(y)∧Barks(y)) (There exists a dog that barks)

## 3. Semantics of First-Order Logic

The semantics of FOL defines the meaning of sentences by specifying how their truth values are determined. This is done with respect to a **model**.

A **model** in FOL consists of:

- **A Domain (or Universe) D:** A non-empty set of objects (individuals) in the "world" that the sentences refer to.
- **An Interpretation Function I:** This function maps:
  - Each **constant symbol** to an object in the domain D.
  - Each **predicate symbol** (of arity k) to a k-ary relation over D (a set of k-tuples of objects from D).
  - Each **function symbol** (of arity k) to a k-ary function from Dk to D.

**Truth Evaluation in a Model:**

- **Atomic Sentences:** An atomic sentence P(t1,…,tk) is true in a model if the tuple of objects referred to by (t1,…,tk) is in the relation that the predicate P is interpreted as.
- **Complex Sentences:** The truth values of complex sentences are determined by the truth tables of the logical connectives, exactly as in propositional logic, given the truth values of their constituent sentences.
- **Quantified Sentences:**
  - ∀x.S: Is true in a model if S is true for *every possible assignment* of objects from D to the variable x.
  - ∃x.S: Is true in a model if S is true for *at least one assignment* of an object from D to the variable x.

**Entailment:** A sentence α is **logically entailed** by a knowledge base KB (written KB⊨α) if α is true in *every model* in which KB is true.

## 4. Using First-Order Logic (Examples)

FOL's expressive power allows it to represent a vast range of knowledge:

- **Facts about the World:**
  - `King(John)`
  - `Person(Richard)`
  - `Brother(Richard, John)`
- **General Laws/Rules:**
  - `∀x. (King(x) ∧ Greedy(x)) ⇒ Evil(x)` (All kings who are greedy are evil)
  - `∀x, y. (Brother(x, y) ∧ Sibling(y, x))` (Brotherhood is a symmetric relation if siblinghood is)
  - `∀x. Person(x) ⇒ Mortal(x)` (All persons are mortal)
  - `∀c. (Car(c) ⇒ HasWheels(c))` (All cars have wheels)
- **Problem Domains:**
  - **Wumpus World:** ∀x,y.Breeze(x,y)⇔(Pit(x−1,y)∨Pit(x+1,y)∨Pit(x,y−1)∨Pit(x,y+1))
    - This single FOL rule captures the knowledge about breezes and pits for *all* squares, unlike propositional logic which would need a separate rule for each square.
  - **Family Relations:** `Father(x, y)` means x is father of y.
    - `∀x, y, z. (Father(x, y) ∧ Father(y, z)) ⇒ Grandfather(x, z)`

## 5. Knowledge Engineering in First-Order Logic

**Knowledge Engineering** is the process of building a knowledge base for an AI system. When using FOL, it involves:

1. **Identify the Task:** What problem needs to be solved? What knowledge is relevant?
2. **Assemble the Relevant Knowledge:** Gather all known facts, rules, and relationships from domain experts, texts, or observations.
3. **Choose a Vocabulary (Ontology):**
    - Decide on the **constants** (specific objects).
    - Decide on the **predicates** (properties and relations) and their arity (number of arguments).
    - Decide on the **functions** (mappings) and their arity.
    - This step is crucial and forms the **ontology** of the domain. A well-designed ontology makes representation clearer and reasoning more efficient.
4. **Encode General Knowledge as FOL Axioms:** Translate the collected general rules and relationships into FOL sentences using predicates, functions, variables, and quantifiers. These are often called **axioms**.
    - Example: "A person's parent is older than them" $\rightarrow \forall p,c,r.(Parent(p,c) \Rightarrow Older(p,c))$
5. **Encode Specific Facts:** Represent specific instances or observations as ground atomic sentences or quantified statements with specific constants.
    - Example: "John is a person" $\rightarrow Person(John)$
6. **Determine the Reasoning Method:** Choose appropriate inference algorithms (e.g., resolution, forward chaining, backward chaining adapted for FOL using **unification**) to query the KB and derive conclusions.
7. **Test and Debug:** Test the KB with queries. If it produces incorrect or unexpected answers, debug the axioms, facts, or the vocabulary itself. This is often an iterative process.

## Challenges in Knowledge Engineering with FOL:

- **Expressiveness vs. Tractability:** FOL is very expressive, but full FOL inference is semi-decidable (meaning if a sentence is entailed, the proof procedure will eventually find it, but if it's not entailed, it might run forever). For practical AI, often a subset of FOL (e.g., Horn clauses in FOL) or specialized reasoners are used.
- **Ambiguity:** Natural language is ambiguous; translating it precisely into unambiguous FOL is challenging.
- **Completeness:** Ensuring that all necessary knowledge is captured in the KB.
- **Consistency:** Ensuring that the KB does not contain contradictory statements.
- **Frame Problem:** How to represent what *doesn't* change when an action occurs.
- **Qualification Problem:** How to list all conditions that could prevent an action from achieving its intended effect.
- **Ramification Problem:** How to represent all the indirect effects of an action.

## Propositional vs. First-Order Inference

**PL deals with truth values of atomic sentences**,

**FOL deals with objects, properties, relations, and quantifiers.**

- **PL Inference:** Operates on fixed, unchangeable atomic units. Its methods (truth tables, resolution, chaining) are relatively straightforward applications of logical rules on these fixed propositions. Entailment is **decidable**.

- **FOL Inference:** Requires mechanisms to handle variables, terms, and the process of matching general rules to specific facts. This leads to the need for **unification** and **instantiation**. Entailment in full FOL is **semi-decidable**.

---

## Unification and Lifting

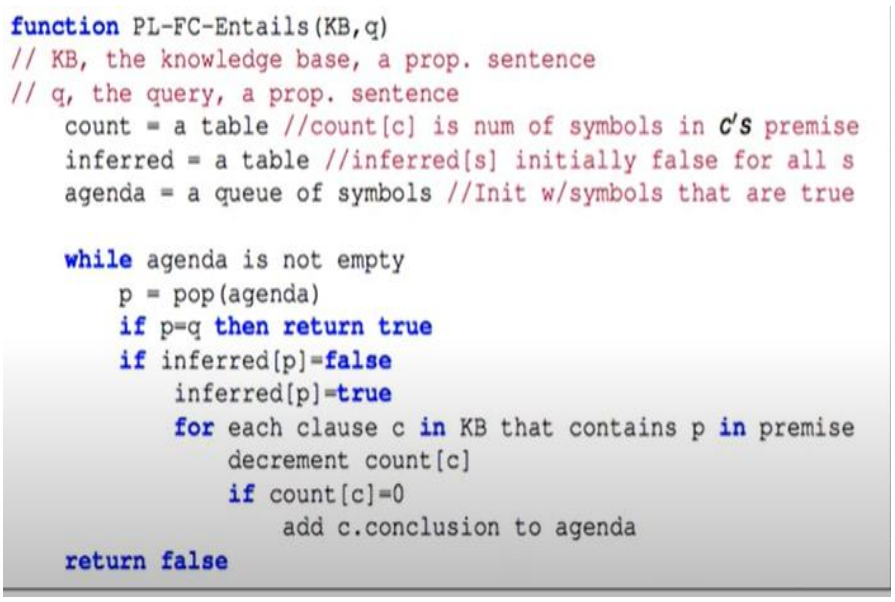
These are the bedrock of FOL inference:

- **Unification:** The process of finding a substitution (mapping variables to terms) that makes two logical expressions identical. The goal is to find the **Most General Unifier (MGU)**, which imposes the fewest restrictions.
  - **Example: Unifying `P(x, A)` and `P(B, y)` yields `{x/B, y/A}`.**
- **Lifting:** The idea of taking propositional inference rules (like Modus Ponens) and generalizing them to operate on FOL sentences containing variables, by incorporating unification. This allows rules to apply to *classes* of objects rather than just specific ground instances.
  - The **"Generalized Modus Ponens" (GMP)** is a prime example of a lifted inference rule.

## Forward Chaining in First-Order Logic

Forward chaining is a **data-driven** or **bottom-up** inference strategy. It starts with a set of known facts (atomic sentences) and repeatedly applies rules (implications) to infer new facts, until the desired goal is derived or no new facts can be generated.

### Data → Conclusion



```
function PL-FC-Entails(KB,q)
// KB, the knowledge base, a prop. sentence
// q, the query, a prop. sentence
    count = a table //count[c] is num of symbols in c's premise
    inferred = a table //inferred[s] initially false for all s
    agenda = a queue of symbols //Init w/symbols that are true

    while agenda is not empty
        p = pop(agenda)
        if p=q then return true
        if inferred[p]=false
            inferred[p]=true
            for each clause c in KB that contains p in premise
                decrement count[c]
                if count[c]=0
                    add c.conclusion to agenda
    return false
```

**How it works with FOL (using Generalized Modus Pones):**

1. **Initialization:** Start with a knowledge base (KB) containing facts (ground atomic sentences) and rules (definite clauses in FOL). Maintain a list of `known_facts`.
2. **Loop:**
   - Repeatedly look for rules whose premises can be satisfied by existing `known_facts` using **unification**.
   - For a rule like `P1(x,y) ∧ P2(y) ⇒ Q(x)`:
     - Find a **substitution θ** such that `P1(x,y)θ` is in `known_facts` AND `P2(y)θ` is in `known_facts`.
     - If such a θ exists, then the conclusion `Q(x)θ` can be inferred.
     - Add `Q(x)θ` to `known_facts` if it's not already there.
     - If the goal is `Q(x)θ`, then the proof is found.
3. **Termination:** The process continues until no new facts can be added, or the goal is found. If the goal is not found, it is not entailed by the given KB via forward chaining.

## Example:

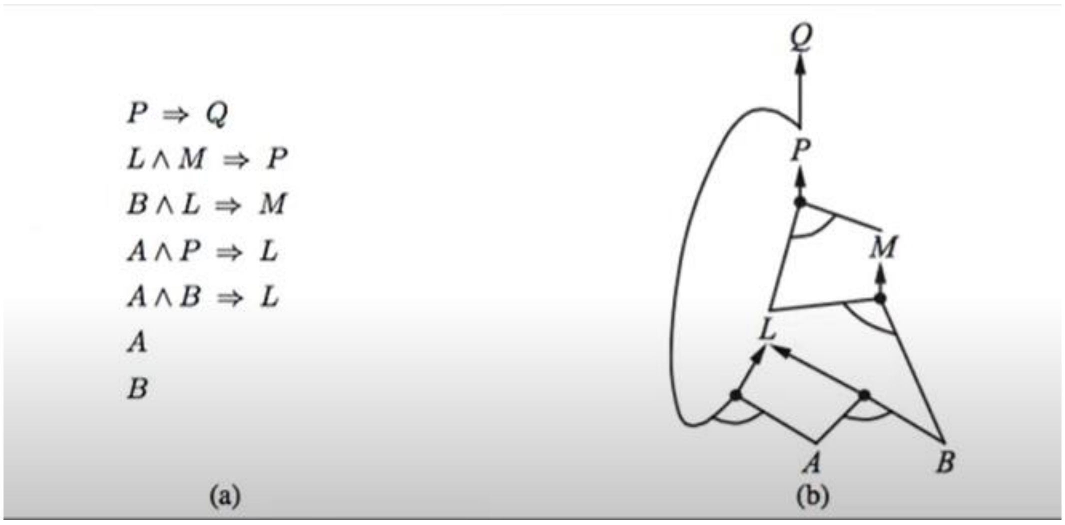
- **KB:**
  1. `King(John)`
  2. `Greedy(John)`
  3. `∀x. (King(x) ∧ Greedy(x)) ⇒ Evil(x)` (Rule)
- **Goal:** `Evil(John)`

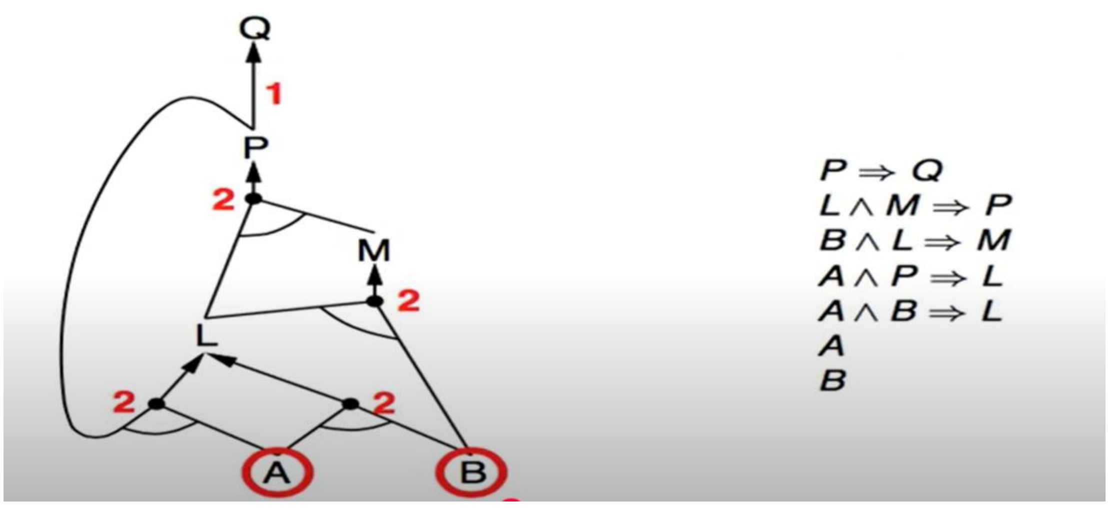
**Forward Chaining Trace:**

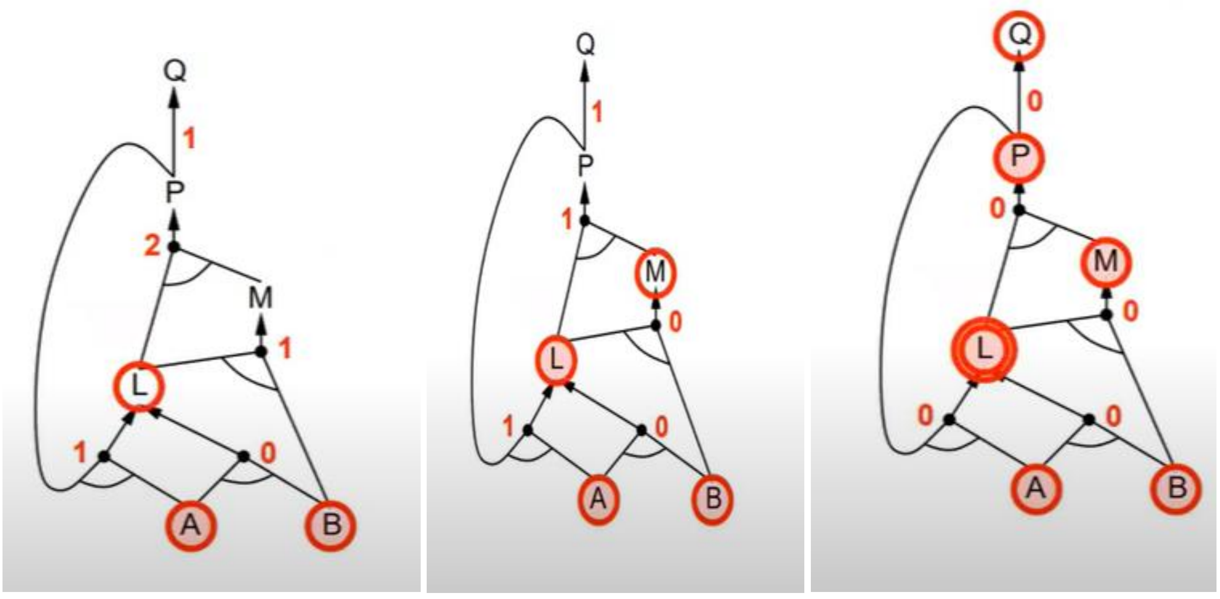
1. `known_facts = {King(John), Greedy(John)}`
2. Consider Rule 3: `(King(x) ∧ Greedy(x)) ⇒ Evil(x)`
   - **Unify** `King(x)` with `King(John)`. This gives `θ = {x/John}`.
   - Apply `θ` to the second premise: `Greedy(x)θ` becomes `Greedy(John)`.
   - Check if `Greedy(John)` is in `known_facts`. Yes, it is.
   - Since all premises are satisfied with `θ = {x/John}`, infer `Evil(x)θ`, which is `Evil(John)`.
3. Add `Evil(John)` to `known_facts`.
4. `Evil(John)` matches the goal. **Proof found!**

**Use Cases:** When new data constantly arrives and you want to know all its logical consequences (e.g., in a monitoring system, a production rule system, or a data analysis pipeline).

## Example :


$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
$A$
$B$
(a)
Q
P
M
L
A
B
(b)


Q
1
P
2
M
2
L
2
2
A
B
$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
$A$
$B$



A and B => L

Marked **A** and **B**

A and P =>L

Marked **L**

B and L => M

Marked **M**

L and M =>P

Marked **P**

P=>Q

Marked **Q**

**So the forward chaining : A->B->B->L->M->P->Q**

## Backward Chaining in First-Order Logic

Backward chaining is a **goal-driven** or **top-down** inference strategy. It starts with the query (the goal to be proven) and works backward, trying to find rules whose conclusions match the current goal or subgoals, until all subgoals are matched with known facts.

**Data <---   Conclusion**

**How it works with FOL (using GMP recursively):**

1. **Function `PROVE(goal, KB)`:**
   - If `goal` is already a fact in `KB`, return `True`.
   - For each rule `(P1 ∧ ... ∧ Pn) ⇒ Q` in `KB`:
     - Attempt to **unify** the head `Q` of the rule with the `goal`. Let the substitution be `θ`.
     - If unification succeeds, then the new subgoals are the premises `P1θ, ..., P_nθ`.
     - Recursively call `PROVE` for each of these subgoals: `PROVE(P1θ, KB) AND PROVE(P2θ, KB) AND ... AND PROVE(Pnθ, KB)`.
     - If all recursive calls return `True`, then the `goal` is proven, return `True`.
   - If no rule or fact can prove the `goal`, return `False`.

**Example:** (Same KB and Goal as above)

- **KB:**
  1. `King(John)`
  2. `Greedy(John)`
  3. `∀x. (King(x) ∧ Greedy(x)) ⇒ Evil(x)`
- **Goal:** `Evil(John)`

**Backward Chaining Trace:**

1. `PROVE(Evil(John))`
2. Look for rules whose head unifies with `Evil(John)`.
   - Rule 3: `(King(x) ∧ Greedy(x)) ⇒ Evil(x)`
   - Unify `Evil(x)` with `Evil(John)`. This gives `θ = {x/John}`.
3. New subgoals: `King(x)θ` and `Greedy(x)θ`, which are `King(John)` and `Greedy(John)`.
4. **Recursively `PROVE(King(John))`:**
   - Is `King(John)` a fact in KB? Yes. Return `True`.
5. **Recursively `PROVE(Greedy(John))`:**
   - Is `Greedy(John)` a fact in KB? Yes. Return `True`.
6. Since both `King(John)` and `Greedy(John)` are proven, the original goal `Evil(John)` is proven. **Proof found!**

**Use Cases:** When you have a specific question to answer (e.g., in a diagnostic system, an expert system querying a user for symptoms, or a logic programming language like Prolog).

## Resolution in First-Order Logic

**Concept:** Resolution is a powerful, sound, and **refutation-complete** inference rule for FOL. Refutation-complete means if a sentence is logically entailed by a KB, resolution can derive a contradiction from the negation of the sentence combined with the KB. This implies that if `KB |= α`, then `KB ∧ ¬α` is unsatisfiable, and resolution will find this unsatisfiability.

**Key Steps for Resolution in FOL:**

1. **Convert to Conjunctive Normal Form (CNF) (or Clausal Form):** All sentences in the KB and the negated query (`¬α`) must be transformed into a conjunction of clauses, where each clause is a disjunction of literals. This is a crucial preprocessing step and involves several sub-steps:
   - **Eliminate Implications (⇒) and Biconditionals (⇔):** Use equivalences like `A ⇒ B ≡ ¬A ∨ B`.
   - **Move ¬ Inwards:** Apply De Morgan's laws and `¬∀x. P(x) ≡ ∃x. ¬P(x)`, `¬∃x. P(x) ≡ ∀x. ¬P(x)`.
   - **Standardize Variables:** Rename variables so that each quantifier binds a unique variable (e.g., `∀x. P(x) ∨ ∃x. Q(x)` becomes `∀x. P(x) ∨ ∃y. Q(y)`).
   - **Skolemization:** Eliminate existential quantifiers.
     - If `∃x. P(x)`, replace `x` with a Skolem constant (e.g., `P(A)` where `A` is a new, unique constant).
     - If `∀y. ∃x. P(x, y)`, replace `x` with a Skolem function of `y` (e.g., `∀y. P(F(y), y)` where `F` is a new, unique function symbol).
   - **Drop Universal Quantifiers:** Once all existential quantifiers are removed, all remaining variables are implicitly universally quantified.
   - **Distribute ∧ over ∨:** Convert to CNF (e.g., `A ∨ (B ∧ C)` becomes `(A ∨ B) ∧ (A ∨ C)`).
   - **Flatten nested conjunctions/disjunctions:** `(A ∧ B) ∧ C` becomes `A ∧ B ∧ C`.
   - **Create separate clauses:** Each conjunct becomes a separate clause in the set.

2. **Add Negated Query:** Take the query α, negate it (¬α), and convert ¬α to CNF clauses. Add these clauses to the KB.
3. **Resolution Rule:** Repeatedly apply the resolution rule until either:
   - **Empty Clause (`[]` or ⊥) is derived:** This indicates a contradiction, meaning `KB ∧ ¬α` is unsatisfiable, so `KB |= α`.
   - **No more resolutions are possible:** This means `KB ∧ ¬α` is satisfiable, and `KB <binary data, 1 bytes><binary data, 1 bytes><binary data, 1 bytes> α`.

   **The Resolution Step (Lifting Propositional Resolution):** Given two clauses C1 and C2 :

   - Select a literal L1 from C1 and a literal L2 from C2 such that L1 and ¬L2 (or ¬L1 and L2) are unifiable.
   - Find the **MGU θ** for L1 and ¬L2.
   - The new clause (resolvent) is (C1−{L1})θ∪(C2−{L2})θ.

**Example:** (Proving `Evil(John)` from previous KB using Resolution)

- **KB:**
  1. `King(John)`
  2. `Greedy(John)`
  3. `∀x. (King(x) ∧ Greedy(x)) ⇒ Evil(x)`
- **Negated Query:** `¬Evil(John)`
- **Conversion to CNF:**

  1. `King(John)` (Clause C1)
  2. `Greedy(John)` (Clause C2)
  3. `¬King(x) ∨ ¬Greedy(x) ∨ Evil(x)` (from Rule 3 after eliminating ⇒ and dropping `∀x`; Clause C3)
  4. `¬Evil(John)` (from negated query; Clause C4)
- **Resolution Steps:**
  - **Step 1:** Resolve C3 (`¬King(x) ∨ ¬Greedy(x) ∨ Evil(x)`) and C4 (`¬Evil(John)`).
    - Literals: `Evil(x)` from C3 and `¬Evil(John)` from C4.
    - MGU: `θ = {x/John}`
    - Resolvent: `(¬King(x) ∨ ¬Greedy(x))θ` = `¬King(John) ∨ ¬Greedy(John)` (Clause C5)
  - **Step 2:** Resolve C5 (`¬King(John) ∨ ¬Greedy(John)`) and C1 (`King(John)`).
    - Literals: `¬King(John)` from C5 and `King(John)` from C1.
    - MGU: `{}` (already ground)
    - Resolvent: `¬Greedy(John)` (Clause C6)
  - **Step 3:** Resolve C6 (`¬Greedy(John)`) and C2 (`Greedy(John)`).
    - Literals: `¬Greedy(John)` from C6 and `Greedy(John)` from C2.
    - MGU: `{}`
    - Resolvent: `[]` (Empty Clause!)
- **Conclusion:** Since the empty clause is derived, the original query `Evil(John)` is entailed by the KB.

**Advantages of Resolution:**

- **Completeness:** It's a complete inference procedure for FOL (specifically, refutation complete). If a logical consequence exists, resolution will find it.
- **Automated Theorem Proving:** It's the basis for many automated theorem provers and logic programming languages (like Prolog, which uses a restricted form called SLD Resolution).

**Disadvantages of Resolution:**

- **Conversion to CNF:** The conversion process, especially Skolemization, can be complex and sometimes increases the size of the formula.
- **Search Space:** The search space for finding pairs of clauses to resolve can be enormous, leading to computational inefficiency in complex KBs. Heuristics are crucial for practical implementations.

| **Feature** | **Forward Chaining** | **Backward Chaining** |
|---|---|---|
| **Driven By** | Data/Facts | Goal/Hypothesis |
| **Approach** | Bottom-Up / Data-driven | Top-Down / Goal-driven |
| **Search Strategy** | Often Breadth-First | Often Depth-First |
| **Generates** | All possible conclusions | Only conclusions relevant to the specific goal |
| **Goal Known?** | Not necessarily (can find any conclusion) | Must have a clear goal upfront |
| **Flexibility** | More flexible; can find multiple conclusions | Less flexible; typically finds one answer/path |
| **Interactive Query** | Less direct for specific queries | Good for interactive query answering |
| **Complexity** | Simpler to implement (basic loop and matching) | Can be more complex due to recursion and backtracking |
| **Common Use** | Production systems, monitoring, planning | Expert systems, diagnostics, theorem proving, query answering |