AY: 2025-26

Class:	**TE**	**Semester:**	**V**
Course Code:	**CSL 502**	**Course Name:**	**Artificial Intelligence**

Name of Student:	**Pranita Kumbhar**
Roll No. :	**70**
Experiment No.:	**08**
Title of the Experiment:	**Implementation of Forward, Backward, and Resolution Inference Techniques in First- Order Predicate Logic (FOPL) to Prove a Goal Sentence**
Date of Performance:	**26/09/25**
Date of Submission:	**03/10/25**

Evaluation

Performance Indicator	**Max. Marks**	**Marks Obtained**
Performance	5	
Understanding	5	
Journal work and timely submission	10	
Total	20	

Performance Indicator	**Exceed Expectations (EE)**	**Meet Expectations (ME)**	**Meet Expect Below Expectations (BE)**
Performance	4-5	2-3	1
Understanding	4-5	2-3	1
Journal work and timely submission	8-10	5-8	1-4

Checked by
Name of Faculty : Ms. Rujuta Vartak
Signature :
Date:

Aim: Implementation of Forward, Backward, and Resolution Inference Techniques in First-Order Predicate Logic (FOPL) to Prove a Goal Sentence.

Objective:. To prove a given goal/query using Forward, Backward, and Resolution inference techniques in First-Order Predicate Logic (FOPL) through Prolog implementation.

Requirement: Turbo Prolog 2.0 or above / Windows Prolog.

Theory:

First-Order Predicate Logic (FOPL) is a symbolic logic system that extends propositional logic by including quantifiers and predicates with variables. It allows expressing facts and rules in a form that is both precise and general. FOPL enables reasoning about objects, their properties, and their relationships.

Key components of FOPL include:

Constants: Represent specific entities (e.g., john, apple).

Variables: Represent arbitrary entities (e.g., X, Y).

Predicates: Describe properties or relationships (e.g., father(john, bob)).

Quantifiers:

- Universal quantifier (∀): States that something is true for all instances.
- Existential quantifier (∃): States that something is true for at least one instance.

Functions: Return values based on arguments (not often used in Prolog).

1. Forward Chaining (Data-Driven Inference)

Forward chaining starts with a set of known facts and repeatedly applies inference rules to derive new facts until the goal is found or no more facts can be deduced.

Example:

fact(father(john, bob)).
fact(father(bob, steve)).
rule(grandfather(X, Z), [father(X, Y), father(Y, Z)]).

In this example, the system uses the rule to infer that grandfather(john, steve) is true.

2. Backward Chaining (Goal-Driven Inference)

Backward chaining starts with a goal and attempts to prove it by recursively finding rules that support it. It is used extensively in logic programming languages like Prolog.

Example:

father(john, bob).
father(bob, steve).
grandfather(X, Z) :- father(X, Y), father(Y, Z).

Query:

?- grandfather(john, steve).

Prolog checks:

- Is father(john, Y) true? Yes, Y = bob.
- Is father(bob, steve) true? Yes.
- So the goal is proven.

3. Resolution (Refutation-Based Proof)

Resolution is a powerful inference technique used in automated theorem proving. It involves refuting the negation of the goal by deriving a contradiction.

Steps involved:

1. Convert all knowledge and the negation of the goal into Conjunctive Normal Form (CNF).
2. Use the resolution rule to resolve pairs of clauses.
3. If the empty clause is derived, the original goal is proven.

Example:

KB: $P(a)$ and $\neg P(x) \vee Q(x)$
Negated Goal: $\neg Q(a)$
Resolution: From $P(a)$ and $\neg P(x) \vee Q(x) \Rightarrow Q(a)$
Now resolve $Q(a)$ and $\neg Q(a) \Rightarrow$ contradiction.
Thus, $Q(a)$ is proven to be true.
Resolution is sound and complete but computationally expensive. It forms the basis of many AI and logic system

PROGRAM-

```
# Inference Techniques in FOPL
# Example: Man(Socrates) -> Mortal(Socrates)

# --- Knowledge Base ---
facts = ["Man(Socrates)"]       # Known fact
rules = ["IF Man(X) THEN Mortal(X)"]  # Implication rule
goal = "Mortal(Socrates)"       # Goal to prove

# --- Forward Chaining ---
def forward_chaining(facts, rules):
    derived = set(facts)
    changed = True
    while changed:
        changed = False
        for rule in rules:
            if "IF Man(X) THEN Mortal(X)" in rule:
                for fact in list(derived):
                    if "Man(" in fact:
                        person = fact[4:-1]  # extract name
                        new_fact = f"Mortal({person})"
                        if new_fact not in derived:
```

```
                derived.add(new_fact)
                changed = True
    return derived

# --- Backward Chaining ---
def backward_chaining(goal, facts, rules):
    if goal in facts:
        return True
    if goal.startswith("Mortal("):
        person = goal[7:-1]
        subgoal = f"Man({person})"
        if subgoal in facts:
            return True
    return False

# --- Resolution (very simple simulation) ---
def resolution(facts, rules, goal):
    if goal.startswith("Mortal("):
        person = goal[7:-1]
        if f"Man({person})" in facts:
            return True
    return False

# --- Main Program ---
print("Facts:", facts)
print("Rules:", rules)
print("Goal:", goal)

print("\n--- Forward Chaining ---")
derived = forward_chaining(facts, rules)
print("Derived Facts:", derived)
print("Goal Proven?", goal in derived)

print("\n--- Backward Chaining ---")
print("Goal Proven?", backward_chaining(goal, facts, rules))

print("\n--- Resolution ---")
print("Goal Proven?", resolution(facts, rules, goal))
```

OUTPUT-

```
52    print("Derived Facts:", derived)

OUTPUT    TERMINAL    DEBUG CONSOLE    PORTS

PS C:\Users\HP> python -u "C:\Users\HP\AppData\Local\Temp\tempCodeRunnerFile.python"
Facts: ['Man(Socrates)']
Rules: ['IF Man(X) THEN Mortal(X)']
Goal: Mortal(Socrates)

--- Forward Chaining ---
Derived Facts: {'Mortal(Socrates)', 'Man(Socrates)'}
Goal Proven? True

--- Backward Chaining ---
Goal Proven? True

--- Resolution ---
Goal Proven? True
PS C:\Users\HP>
```

Conclusion:

Prolog's unification explanation feature provides insights into how terms are matched step by step. It begins by checking if two constants (atoms or numbers) are identical. If they are, unification succeeds, and if not, it fails. For variables, Prolog explains how they are instantiated or bound to terms, allowing for flexible matching. When unifying complex terms, Prolog ensures the functors and their arguments match; if not, unification fails. In case of variable unification across multiple locations, Prolog also highlights successful or conflicting bindings, offering clarity on why unification either succeeds or fails