Vidyavardhini's College of Engineering and Technology

Department of Artificial Intelligence & Data Science

AY: 2025-26

Class:	**TE**	**Semester:**	**V**
Course Code:	**CSL502**	**Course Name:**	**Artificial Intelligence Lab**

Name of Student:	**Shravani Sandeep Raut**
Roll No. :	**51**
Experiment No.:	**8**
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:	**16/09/25**
Date of Submission:	**23/09/25**

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)**	**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 : Mrs. 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 ¬P(x) ∨ Q(x)

Negated Goal: ¬Q(a)

Resolution: From P(a) and ¬P(x) ∨ Q(x) ⇒ Q(a)

Now resolve Q(a) and ¬Q(a) ⇒ 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-

```
# Simple implementation of Forward, Backward, and Resolution inference in FOPL

# Here we use basic examples for understanding.

# --- Forward Chaining ---
def forward_chaining(KB, goal):
    inferred = set()
    new_inferred = True

    while new_inferred:
        new_inferred = False
        for rule in KB:
            if '=>' in rule:
                premise, conclusion = rule.split('=>')
                premise = [x.strip() for x in premise.split('&')]
                conclusion = conclusion.strip()
                if all(p in inferred for p in premise) and conclusion not in inferred:
```

```
                inferred.add(conclusion)
                new_inferred = True
            else:
                inferred.add(rule.strip())
    return goal in inferred

# --- Backward Chaining ---
def backward_chaining(KB, goal):
    if goal in KB:
        return True

    for rule in KB:
        if '=>' in rule:
            premise, conclusion = rule.split('=>')
            conclusion = conclusion.strip()
            if conclusion == goal:
                premise = [x.strip() for x in premise.split('&')]
                if all(backward_chaining(KB, p) for p in premise):
                    return True
    return False

# --- Resolution ---
def resolution(KB, goal):
    clauses = [set(x.replace(' ', '').split('v')) for x in KB]
    goal_clause = set(['~' + goal])
    clauses.append(goal_clause)

    new = set()
    while True:
        n = len(clauses)
        pairs = [(clauses[i], clauses[j]) for i in range(n) for j in range(i + 1, n)]

        for (ci, cj) in pairs:
            resolvents = resolve(ci, cj)
            if set() in resolvents:
                return True
            new = new.union(resolvents)

        if new.issubset(set(map(frozenset, clauses))):
            return False
        for c in new:
            if c not in clauses:
                clauses.append(c)
```

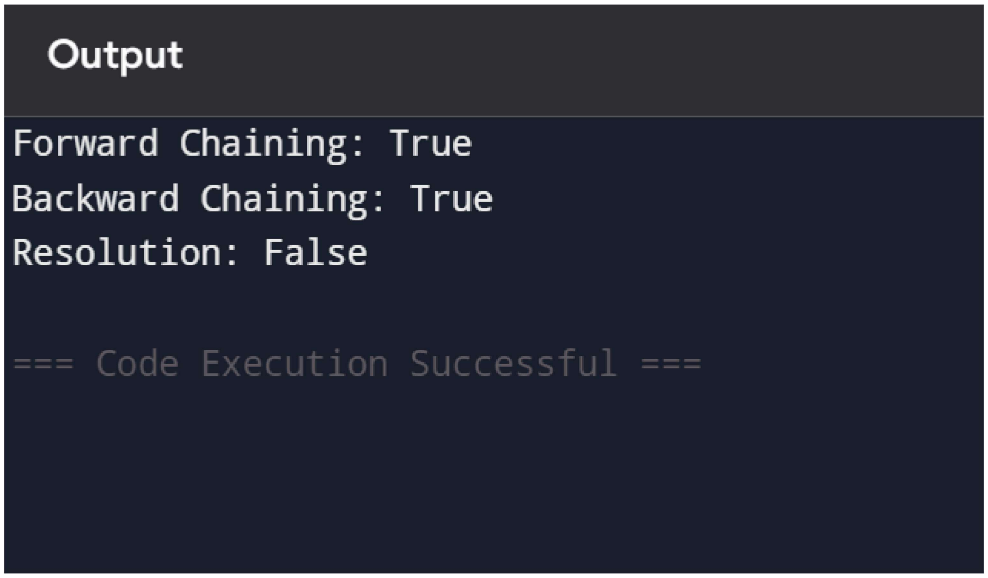
```
def resolve(ci, cj):
    resolvents = set()
    for di in ci:
        for dj in cj:
            if di == '~' + dj or dj == '~' + di:
                resolvent = (ci - {di}).union(cj - {dj})
                resolvents.add(frozenset(resolvent))
    return resolvents

# --- Example Knowledge Base ---
KB = [
    'A',
    'A => B',
    'B => C'
]

goal = 'C'

print('Forward Chaining:', forward_chaining(KB, goal))
print('Backward Chaining:', backward_chaining(KB, goal))
print('Resolution:', resolution(KB, goal))
```

OUTPUT-



```
Forward Chaining: True
Backward Chaining: True
Resolution: False


=== Code Execution Successful ===
```

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