Experiment no.:5

Aim:

To implement the python code for PROLOG.

Code:

```
from typing import List, Dict, Tuple, Union

Term = Union[str, Tuple]  # Either a constant/variable or a compound term

def is_variable(x):
    return isinstance(x, str) and x[0].isupper()

def unify(x, y, subst):
    if subst is None:
        return None
    elif x == y:
        return subst
    elif is_variable(x):
        return unify_var(x, y, subst)
    elif is_variable(y):
        return unify_var(y, x, subst)
    elif isinstance(x, tuple) and isinstance(y, tuple) and x[0] == y[0] and len(x) == len(y):
        for a, b in zip(x[1:], y[1:]):
            subst = unify(a, b, subst)
        return subst
    else:
        return None
```

```
def unify_var(var, x, subst):
    if var in subst:
        return unify(subst[var], x, subst)
    elif is_variable(x) and x in subst:
        return unify(var, subst[x], subst)
    elif occurs_check(var, x, subst):
        return None
    else:
        subst[var] = x
        return subst

def occurs_check(var, x, subst):
    if var == x:
        return True
    elif isinstance(x, tuple):
        return any(occurs_check(var, arg, subst) for arg in x[1:])
    elif is_variable(x) and x in subst:
        return occurs_check(var, subst[x], subst)
    return False

class PrologEngine:
    def __init__(self):
        self.facts = []
        self.rules = []

    def add_fact(self, fact: Term):
        self.facts.append(fact)
```

```
    def add_rule(self, head: Term, body: List[Term]):
        self.rules.append((head, body))

    def ask(self, goal: Term) -> bool:
        return self.solve(goal, {})

    def solve(self, goal: Term, subst: Dict[str, Term]) -> bool:
        for fact in self.facts:
            s = unify(goal, fact, subst.copy())
            if s is not None:
                print(f"✓ Matched fact: {fact} with substitution: {s}")
                return True

        for head, body in self.rules:
            s = unify(goal, head, subst.copy())
            if s is not None:
                print(f"Trying rule: {head} :- {body} with substitution: {s}")
                if all(self.solve(substitute(b, s), s) for b in body):
                    return True

        return False

def substitute(term: Term, subst: Dict[str, Term]) -> Term:
    if is_variable(term):
        return subst.get(term, term)
    elif isinstance(term, tuple):
        return (term[0],) + tuple(substitute(arg, subst) for arg in term[1:])
    return term
```

```
# Example Usage
if __name__ == "__main__":
    pl = PrologEngine()

    # Facts
    pl.add_fact(("parent", "john", "mary"))
    pl.add_fact(("parent", "mary", "susan"))

    # Rule: grandparent(X, Y) :- parent(X, Z), parent(Z, Y)
    pl.add_rule(("grandparent", "X", "Y"), [
        ("parent", "X", "Z"),
        ("parent", "Z", "Y")
    ])

    # Query
    query = ("grandparent", "john", "susan")
    print(f"Query: {query}")
    result = pl.ask(query)
    print("Result:", result)
```

Output:

Query: ('grandparent', 'john', 'susan')

✓ Matched fact: ('parent', 'john', 'mary') with substitution: {'X': 'john', 'Z': 'mary'}

✓ Matched fact: ('parent', 'mary', 'susan') with substitution: {'X': 'john', 'Z': 'mary', 'Y': 'susan'}

Result: True

Result:

Thus the code has been successfully compiled.

Experiment no.:6

Aim:

To Implement of Unification and Resolution Algorithm.

Code:

```
from typing import Union, Dict, List, Optional

# Terms can be constants, variables, or compound expressions (functors)
Term = Union[str, tuple]  # e.g., 'X', 'john', ('father', 'X')

def is_variable(term: Term) -> bool:
  return isinstance(term, str) and term[0].isupper()

def unify(x: Term, y: Term, subst: Dict[str, Term] = {}) -> Optional[Dict[str, Term]]:
  if subst is None:
    return None
  elif x == y:
    return subst
  elif is_variable(x):
    return unify_var(x, y, subst)
  elif is_variable(y):
    return unify_var(y, x, subst)
  elif isinstance(x, tuple) and isinstance(y, tuple) and len(x) == len(y):
    for a, b in zip(x, y):
      subst = unify(a, b, subst)
      if subst is None:
        return None
    return subst
```

```
    else:
        return None

def unify_var(var: str, x: Term, subst: Dict[str, Term]) -> Optional[Dict[str, Term]]:
    if var in subst:
        return unify(subst[var], x, subst)
    elif x in subst:
        return unify(var, subst[x], subst)
    elif occurs_check(var, x, subst):
        return None
    else:
        subst[var] = x
        return subst

def occurs_check(var: str, x: Term, subst: Dict[str, Term]) -> bool:
    if var == x:
        return True
    elif isinstance(x, tuple):
        return any(occurs_check(var, arg, subst) for arg in x)
    elif is_variable(x) and x in subst:
        return occurs_check(var, subst[x], subst)
    return False

# Resolution using simple Horn clauses
def resolve(clause1: List[Term], clause2: List[Term]) -> Optional[List[Term]]:
    for lit1 in clause1:
        for lit2 in clause2:
            subst = unify(lit1, complement(lit2), {})
            if subst is not None:
```

```
            new_clause = list(set(
                substitute_list(clause1, subst) + substitute_list(clause2, subst)
            ))
            new_clause = [l for l in new_clause if l != substitute(lit1, subst) and l != substitute(lit2, subst)]
            return new_clause
    return None

def complement(literal: Term) -> Term:
    if isinstance(literal, tuple) and literal[0] == 'not':
        return literal[1]
    else:
        return ('not', literal)

def substitute(term: Term, subst: Dict[str, Term]) -> Term:
    if is_variable(term):
        return subst.get(term, term)
    elif isinstance(term, tuple):
        return tuple(substitute(arg, subst) for arg in term)
    else:
        return term

def substitute_list(terms: List[Term], subst: Dict[str, Term]) -> List[Term]:
    return [substitute(term, subst) for term in terms]

# Example usage
if __name__ == "__main__":
    # Unification example
    t1 = ('father', 'X')
```

```
t2 = ('father', 'john')
result = unify(t1, t2, {})
print("Unification Result:", result)

# Resolution example
clause1 = [('not', ('man', 'X')), ('mortal', 'X')]
clause2 = [('man', 'socrates')]
resolvent = resolve(clause1, clause2)
print("Resolvent:", resolvent)
```

Sample output:

```
Unification Result: {'X': 'john'}
Resolvent: [('mortal', 'socrates')]
```

Result:

Thus the code has successfully compiled.

Experiment no.:7

:Aim

To Implement Backward Chaining using python programming.

Code:

```
from typing import List, Dict

# Knowledge base
class KnowledgeBase:
    def __init__(self):
        self.facts = set()
        self.rules = []

    def add_fact(self, fact: str):
        self.facts.add(fact)

    def add_rule(self, premises: List[str], conclusion: str):
        self.rules.append((premises, conclusion))

    def backward_chain(self, goal: str, seen=None) -> bool:
        if seen is None:
            seen = set()

        print(f"Trying to prove: {goal}")
        if goal in self.facts:
            print(f"✓ Found fact: {goal}")
            return True
```

```
        if goal in seen:
            print(f"✗ Already tried: {goal}, avoiding infinite loop")
            return False

        seen.add(goal)

        for premises, conclusion in self.rules:
            if conclusion == goal:
                print(f"Considering rule: {' ∧ '.join(premises)} => {conclusion}")
                if all(self.backward_chain(p, seen) for p in premises):
                    print(f"✓ Rule proves: {goal}")
                    self.facts.add(goal)  # Optional: memoize
                    return True

        print(f"✗ Cannot prove: {goal}")
        return False

# Example usage
if __name__ == "__main__":
    kb = KnowledgeBase()

    # Facts
    kb.add_fact("human(socrates)")
    kb.add_fact("human(plato)")

    # Rules
    kb.add_rule(["human(X)"], "mortal(X)")  # This needs variable support (advanced)
```

```
# Instead, simulate without variables:
kb.add_rule(["human(socrates)"], "mortal(socrates)")
kb.add_rule(["human(plato)"], "mortal(plato)")

# Queries
print("\nIs Socrates mortal?")
print("Result:", kb.backward_chain("mortal(socrates)"))

print("\nIs Aristotle mortal?")
print("Result:", kb.backward_chain("mortal(aristotle)"))
```

Output:

```
Is Socrates mortal?
Trying to prove: mortal(socrates)
Considering rule: human(socrates) => mortal(socrates)
Trying to prove: human(socrates)
✓ Found fact: human(socrates)
✓ Rule proves: mortal(socrates)
Result: True

Is Aristotle mortal?
Trying to prove: mortal(aristotle)
✗ Cannot prove: mortal(aristotle)
Result: False
```

Result:

Thus the code has been successfully compiled.

Experiment no.:8

Aim:

To Implement Forward Chaining using python programming.

Code:

```
from typing import List, Tuple, Set

class KnowledgeBase:
  def __init__(self):
    self.facts: Set[str] = set()
    self.rules: List[Tuple[List[str], str]] = []

  def add_fact(self, fact: str):
    self.facts.add(fact)

  def add_rule(self, premises: List[str], conclusion: str):
    self.rules.append((premises, conclusion))

  def forward_chain(self, goal: str) -> bool:
    added = True
    while added:
      added = False
      for premises, conclusion in self.rules:
        if conclusion not in self.facts and all(p in self.facts for p in premises):
          self.facts.add(conclusion)
          print(f"Inferred: {conclusion}")
```

```
                added = True
                if conclusion == goal:
                    return True
        return goal in self.facts

# Example usage
if __name__ == "__main__":
    kb = KnowledgeBase()
    kb.add_fact("human(socrates)")
    kb.add_rule(["human(socrates)"], "mortal(socrates)")

    print("\nIs Socrates mortal?")
    print("Result:", kb.forward_chain("mortal(socrates)"))
```

Output:

```
Inferred: mortal(socrates)
Is Socrates mortal?
Result: True
```

Result:

Thus the code has been successfully compiled.