Week – 9

Create a knowledge base consisting of first order logic statements and prove the given query using Resolution.

Algorithm:

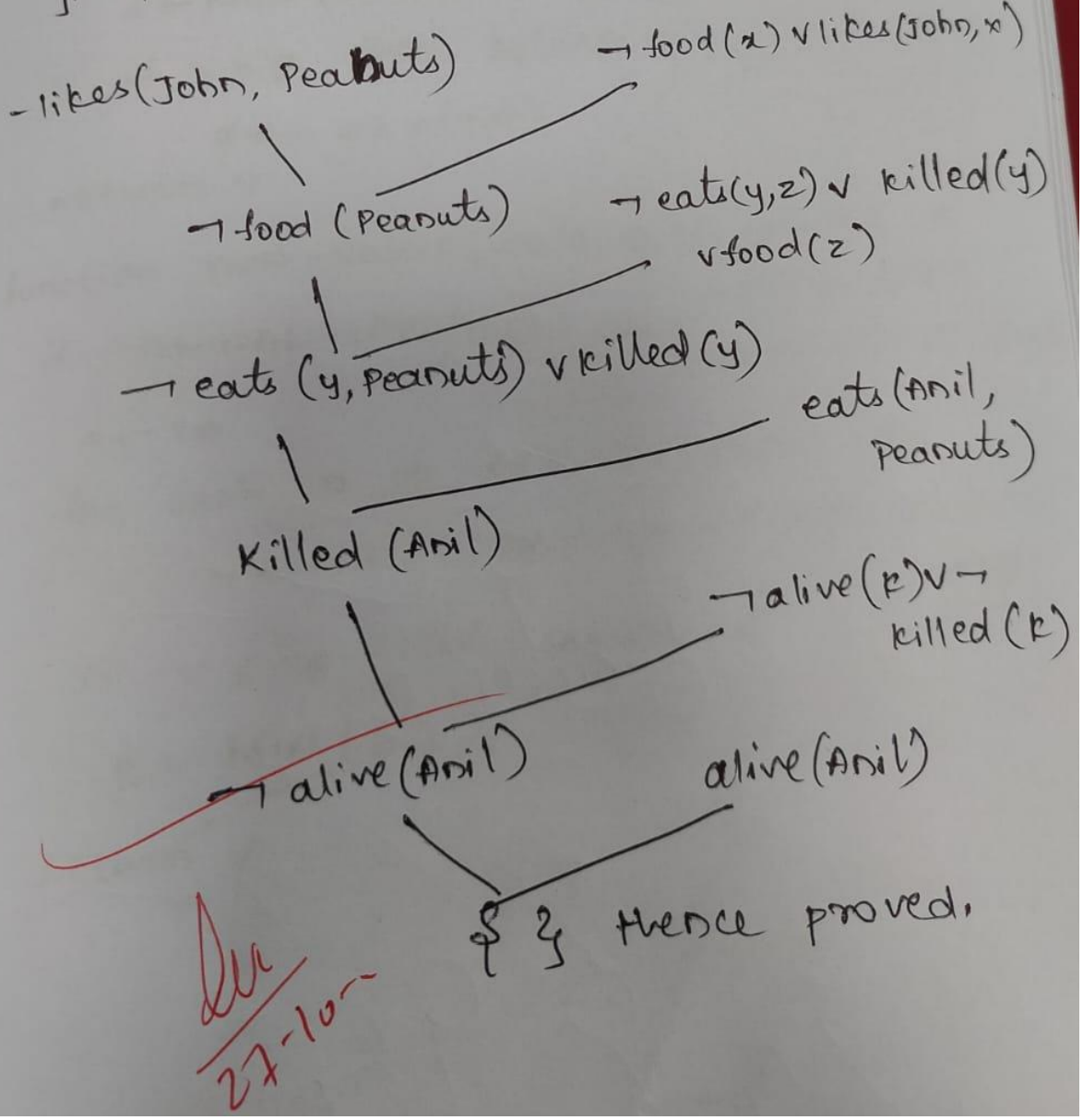
Lab-9

Algorithm: logic statement → CNF
1. Eliminate biconditionals and implications.
2. Move ¬ inwards:
3. Standardize variables apart by renaming them: each quantifier should use a different variable.
4. Skolemize: each existential variable is replace by a skolem constant or skolem function of the closing universally quantified variables.
5. Drop universal functions.
6. Distribute ∧ over ∨.

1. Convert all sentences to CNF
2. Negate conclusion S & convert result to CNF.
3. Add negated conclusion S to the premise clauses.
4. Repeat until contradiction or no progress is made:
   a. Select 2 clauses.
   b. Resolve them together, performing all required unifications.
   c. If resolvent is the empty clause, a contradiction has been found.
   d. If not, add resolvent to the premises.

If we succed in step 4, we have proved the conclusion.

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



27-10--

Output:

```
FOL resolution prover (basic example)

Knowledge base clauses:
   Food(apple)
   Likes(John,x) OR ~Food(x)
   ~Alive(x) OR ~Killed(x)
   ~Eats(x,y) OR Food(y) OR Killed(y)
   Alive(x) OR ~Killed(x)
   Alive(Anil)
   Food(vegetable)
   Eats(Anil,peanuts)
   Eats(Harry,x) OR ~Eats(Anil,x)

Query: Likes(John,peanuts)
Negated query clause will be added to KB and resolution attempted.

Result: True | Derived empty clause (success)
```

Code:

```
from collections import deque

import itertools

import copy

import pprint

print('Shreya Raj 1BM23CS317')

# ----- Data structures -----

class Var:

   def __init__(self, name):

      self.name = name

   def __repr__(self):

      return f"Var({self.name})"

   def __eq__(self, other):

      return isinstance(other, Var) and self.name == other.name

   def __hash__(self):

      return hash(('Var', self.name)) class Const:

   def __init__(self, name):

      self.name = name
```

```
    def __repr__(self):
        return f"Const({self.name})"
    def __eq__(self, other):
        return isinstance(other, Const) and self.name == other.name
    def __hash__(self):
        return hash(('Const', self.name))

class Func:
    def __init__(self, name, args):
        self.name = name
        self.args = args
    def __repr__(self):
        return f"Func({self.name}, {self.args})"
    def __eq__(self, other):
        return isinstance(other, Func) and self.name == other.name and self.args == other.args
    def __hash__(self):
        return hash(('Func', self.name, tuple(self.args)))

class Literal:
    # predicate_name: str, args: list of Terms, negated: bool
    def __init__(self, predicate, args, negated=False):
        self.predicate = predicate
        self.args = tuple(args)
        self.negated = negated
    def negate(self):
        return Literal(self.predicate, list(self.args), not self.negated)
    def __repr__(self):
        sign = "~" if self.negated else ""
```



```
        args = ",".join(map(term_to_str, self.args))
        return f"{sign}{self.predicate}({args})"
```

```
    def __eq__(self, other):
        return (self.predicate, self.args, self.negated) == (other.predicate, other.args, other.negated)
    def __hash__(self):
        return hash((self.predicate, self.args, self.negated))

# Clause is frozenset of Literal
def clause_to_str(cl):
    return " OR ".join(map(str, cl)) if cl else "EMPTY"

def term_to_str(t):
    if isinstance(t, Var):
        return t.name
    if isinstance(t, Const):
        return t.name
    if isinstance(t, Func):
        return f"{t.name}({','.join(term_to_str(a) for a in t.args)})"
    return str(t)

# ----- Substitution utilities -----
def apply_subst_term(term, subst):
    if isinstance(term, Var):
        if term in subst:
            return apply_subst_term(subst[term], subst)
        else:
            return term
    elif isinstance(term, Const):
        return term
    elif isinstance(term, Func):
        return Func(term.name, [apply_subst_term(a, subst) for a in term.args])
    else:
        return term
```

```
def apply_subst_literal(lit, subst):
    return Literal(lit.predicate, [apply_subst_term(a, subst) for a in lit.args], lit.negated)

def apply_subst_clause(clause, subst):
    return frozenset(apply_subst_literal(l, subst) for l in clause)

# ----- Unification (Robust, with occurs-check) -----
def occurs_check(var, term, subst):
    term = apply_subst_term(term, subst)
    if term == var:
        return True
    if isinstance(term, Func):
        return any(occurs_check(var, arg, subst) for arg in term.args)
    return False
def unify_terms(x, y, subst):
    # returns updated subst or None on failure
    x = apply_subst_term(x, subst)
    y = apply_subst_term(y, subst)

    if isinstance(x, Var):
        if x == y:
            return subst
        if occurs_check(x, y, subst):
            return None
        new = subst.copy()
        new[x] = y
        return new
    if isinstance(y, Var):
        return unify_terms(y, x, subst)
    if isinstance(x, Const) and isinstance(y, Const):
```

```
        return subst if x.name == y.name else None
    if isinstance(x, Func) and isinstance(y, Func) and x.name == y.name and len(x.args) == len(y.args):
        for a, b in zip(x.args, y.args):
            subst = unify_terms(a, b, subst)
            if subst is None:
                return None
        return subst
    return None

def unify_literals(l1, l2):
    # l1 and l2 must have same predicate and opposite polarity for resolution
    if l1.predicate != l2.predicate or l1.negated == l2.negated or len(l1.args) != len(l2.args):
        return None
    subst = {}
    for a, b in zip(l1.args, l2.args):
        subst = unify_terms(a, b, subst)
        if subst is None:
            return None
    return subst

# ----- Standardize apart variables (to avoid name clashes) -----
_var_count = 0
def standardize_apart(clause):
    global _var_count
    varmap = {}
    new_literals = []
    for lit in clause:
        new_args = []
        for t in lit.args:
            new_args.append(_rename_term_vars(t, varmap))
        new_literals.append(Literal(lit.predicate, new_args, lit.negated))
```

```
    return frozenset(new_literals)


def _rename_term_vars(term, varmap):
    global _var_count
    if isinstance(term, Var):
        if term.name not in varmap:
            _var_count += 1
            varmap[term.name] = Var(f"{term.name}_{_var_count}")
        return varmap[term.name]
    if isinstance(term, Const):
        return term
    if isinstance(term, Func):
        return Func(term.name, [_rename_term_vars(a, varmap) for a in term.args])
    return term


# ----- Resolution operation between two clauses -----
def resolve(ci, cj):
    # returns set of resolvent clauses (frozenset of literals)
    resolvents = set()
    ci = standardize_apart(ci)
    cj = standardize_apart(cj)
    for li in ci:
        for lj in cj:
            if li.predicate == lj.predicate and li.negated != lj.negated and len(li.args) == len(lj.args):
                subst = unify_literals(li, lj)
                if subst is not None:
                    # build resolvent: (Ci - {li}) U (Cj - {lj}) with subst applied
                    new_clause = set(apply_subst_literal(l, subst) for l in (ci - {li}) | (cj - {lj}))
                    # remove tautologies: a clause containing P and ~P after subst
                    preds = {}
```

```
            taut = False
            for l in new_clause:
                key = (l.predicate, tuple(map(term_to_str, l.args)))
                if key in preds and preds[key] != l.negated:
                    taut = True
                    break
                preds[key] = l.negated
            if not taut:
                resolvents.add(frozenset(new_clause))
    return resolvents

# ----- Main resolution loop -----
def fol_resolution(kb_clauses, query_clause, max_iterations=20000):
    """
    kb_clauses: set/list of clauses (each clause is frozenset of Literal)
    query_clause: single Literal (to be proved), will be negated and added to KB
    Returns True if contradiction (empty clause) is derived.
    """
59
    # Negate the query and add its literals as separate clauses (each literal is a clause)
    negated_query = [query_clause.negate()]
    clauses = set(kb_clauses)
    for l in negated_query:
        clauses.add(frozenset([l]))

    new = set()
    processed_pairs = set()
    queue = list(clauses)

    iterations = 0
    while True:
```

```
        pairs = []
        clause_list = list(clauses)
        n = len(clause_list)
        # iterate over all unordered pairs
        for i in range(n):
            for j in range(i+1, n):
                pairs.append((clause_list[i], clause_list[j]))
something_added = False
        for (ci, cj) in pairs:
            pair_key = (ci, cj)
            if pair_key in processed_pairs:
                continue
            processed_pairs.add(pair_key)
            resolvents = resolve(ci, cj)
            iterations += 1
            if iterations > max_iterations:
                return False, "max_iterations_exceeded"
            for r in resolvents:
                if len(r) == 0:
                    return True, "Derived empty clause (success)"
                if r not in clauses and r not in new:
                    new.add(r)
                    something_added = True
        if not something_added:
            return False, "No new clauses — failure (KB does not entail query)"
        clauses.update(new)
        new = set()

# ----- Helper to create easy constants/vars -----
def C(name): return Const(name)
def V(name): return Var(name)
```

```
def F(name, *args): return Func(name, list(args))
def L(pred, args, neg=False): return Literal(pred, args, neg)

# Build clauses (using variables V('x'), constants C('Anil'), etc.)
60
x = V('x')
y = V('y')

kb = set()

# 1. ¬Food(x) ∨ Likes(John,x)
kb.add(frozenset([L('Food', [x], neg=True), L('Likes', [C('John'), x], neg=False)]))

# 2a. Food(Apple)
kb.add(frozenset([L('Food', [C('apple')], neg=False)]))
# 2b. Food(vegetable)
kb.add(frozenset([L('Food', [C('vegetable')], neg=False)]))

# 3. ¬Eats(x,y) ∨ Killed(y) ∨ Food(y)
kb.add(frozenset([L('Eats', [x,y], neg=True), L('Killed', [y], neg=False), L('Food', [y], neg=False)]))

# 4a. Eats(Anil,peanuts)
kb.add(frozenset([L('Eats', [C('Anil'), C('peanuts')], neg=False)]))
# 4b. Alive(Anil)
kb.add(frozenset([L('Alive', [C('Anil')], neg=False)]))

# 5. ¬Eats(Anil,x) ∨ Eats(Harry,x)
kb.add(frozenset([L('Eats', [C('Anil'), x], neg=True), L('Eats', [C('Harry'), x], neg=False)]))

# 6. ¬Alive(x) ∨ ¬Killed(x)
kb.add(frozenset([L('Alive', [x], neg=True), L('Killed', [x], neg=True)]))
```

```
# 7. Killed(x) ∨ Alive(x)
kb.add(frozenset([L('Killed', [x], neg=True), L('Alive', [x], neg=False)]))

# Query
query = L('Likes', [C('John'), C('peanuts')], neg=False)

def show_kb(kb):
    print("Knowledge base clauses:")
    for c in kb:
        print("  ", clause_to_str(c))
    print()

if __name__ == "__main__":
    print("FOL resolution prover (basic example)\n")
    show_kb(kb)
    print("Query:", query)
    print("Negated query clause will be added to KB and resolution attempted.\n")
    success, info = fol_resolution(kb, query, max_iterations=20000)
    print("Result:", success, "|", info)
```