Lab Experiment No. 6

Implementation of Unification and Resolution

NAME: PALAMALA D SAI NIHARIKA

REG NO: RA1911026010040

SEC: K1

UNIFICATION

AIM: To perform Unification of given two atomic expressions.

PROBLEM STATEMENT:

Unification is a process of making two different logical atomic expressions identical by finding a substitution. Unification depends on the substitution process. It takes two literals as input and makes them identical using substitution. Let Ψ_1 and Ψ_2 be two atomic sentences and σ be a unifier such that, $\boldsymbol{\Psi_1\sigma = \Psi_2\sigma}$, then it can be expressed as **UNIFY(Ψ_1, Ψ_2)**

ALGORITHM:

Algorithm: Unify(Ψ_1, Ψ_2)

```
Step. 1: If $\Psi_1$ or $\Psi_2$ is a variable or constant, then:
    a) If $\Psi_1$ or $\Psi_2$ are identical, then return NIL.
    b) Else if $\Psi_1$is a variable,
        a. then if $\Psi_1$ occurs in $\Psi_2$, then return FAILURE
        b. Else return { ($\Psi_2$/ $\Psi_1$)}.
    c) Else if $\Psi_2$ is a variable,
        a. If $\Psi_2$ occurs in $\Psi_1$ then return FAILURE,
        b. Else return {( $\Psi_1$/ $\Psi_2$)}.
    d) Else return FAILURE.
Step.2: If the initial Predicate symbol in $\Psi_1$ and $\Psi_2$ are not same, then return FAILURE.
Step. 3: IF $\Psi_1$ and $\Psi_2$ have a different number of arguments, then return FAILURE.
Step. 4: Set Substitution set(SUBST) to NIL.
Step. 5: For i=1 to the number of elements in $\Psi_1$.
```

```
        a) Call Unify function with the ith element of $\Psi_1$ and ith element of $\Psi_2$,
  and put the result into S.
        b) If S = failure then returns Failure
        c) If S ≠ NIL then do,
                a. Apply S to the remainder of both L1 and L2.
                b. SUBST= APPEND(S, SUBST).
  Step.6: Return SUBST.
```

CODE:

Unification:

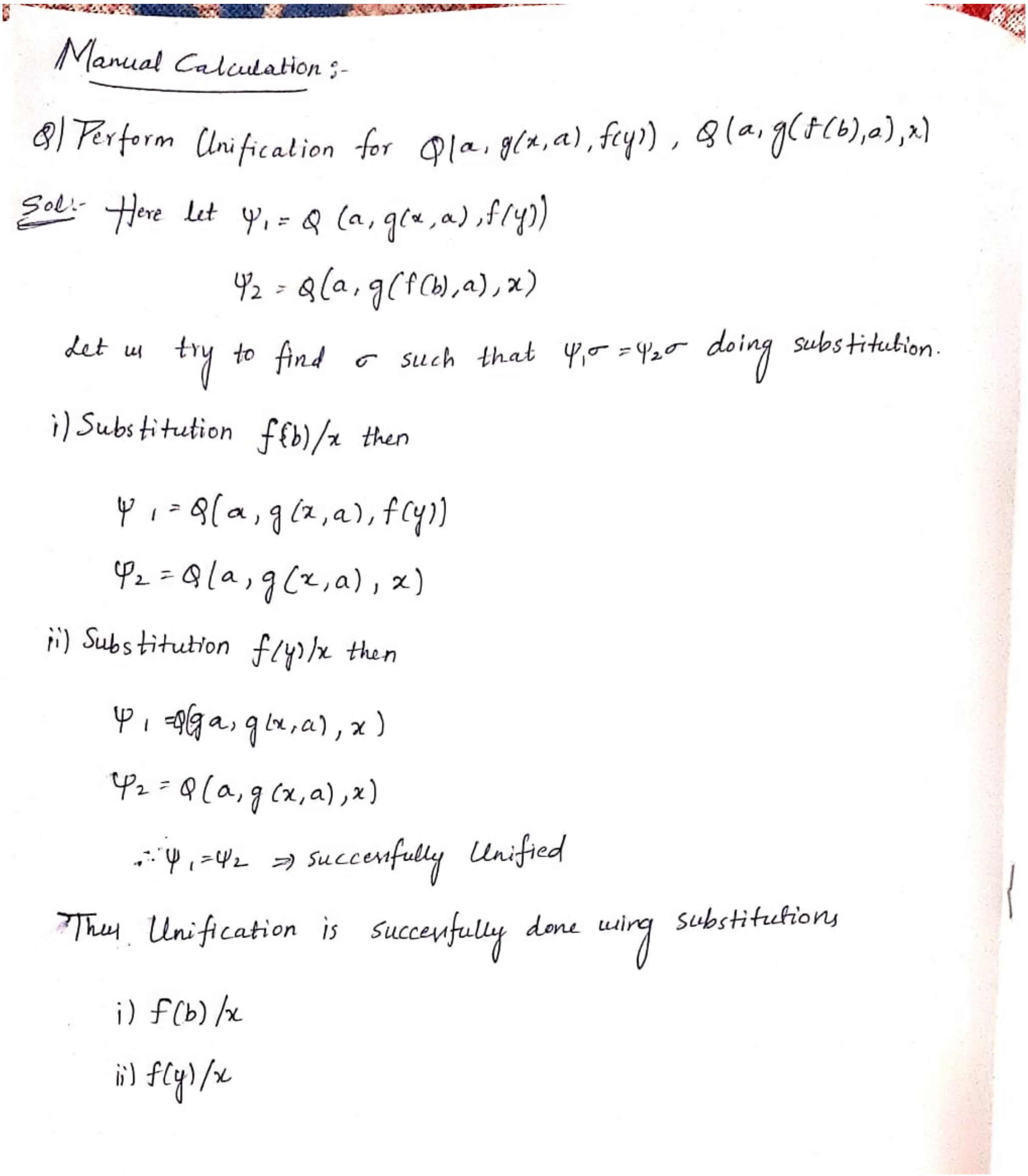
```
def get_index_comma(string):

    index_list = list()

    par_count = 0

    for i in range(len(string)):

        if string[i] == ',' and par_count == 0:

            index_list.append(i)

        elif string[i] == '(':

            par_count += 1

        elif string[i] == ')':

            par_count -= 1

    return index_list

def is_variable(expr):

    for i in expr:

        if i == '(' or i == ')':

            return False
    return True
def process_expression(expr):
    expr = expr.replace(' ', '')
    index = None
    for i in range(len(expr)):
        if expr[i] == '(':
            index = i
            break
    predicate_symbol = expr[:index]
    expr = expr.replace(predicate_symbol, '')
    expr = expr[1:len(expr) - 1]
    arg_list = list()
    indices = get_index_comma(expr)
    if len(indices) == 0:
        arg_list.append(expr)
    else:
        arg_list.append(expr[:indices[0]])
        for i, j in zip(indices, indices[1:]):
            arg_list.append(expr[i + 1:j])
        arg_list.append(expr[indices[len(indices) - 1] + 1:])
    return predicate_symbol, arg_list
```

```
def get_arg_list(expr):
    _, arg_list = process_expression(expr)
    flag = True
    while flag:
        flag = False
        for i in arg_list:
            if not is_variable(i):
                flag = True
                _, tmp = process_expression(i)
                for j in tmp:
                    if j not in arg_list:
                        arg_list.append(j)
                arg_list.remove(i)
    return arg_list
def check_occurs(var, expr):
    arg_list = get_arg_list(expr)
    if var in arg_list:
        return True
    return False
def unify(expr1, expr2):
    if is_variable(expr1) and is_variable(expr2):
        if expr1 == expr2:
            return 'Null'
        else:
            return False
    elif is_variable(expr1) and not is_variable(expr2):
        if check_occurs(expr1, expr2):
            return False
        else:
            tmp = str(expr2) + '/' + str(expr1)
            return tmp
    elif not is_variable(expr1) and is_variable(expr2):
        if check_occurs(expr2, expr1):
            return False
        else:
            tmp = str(expr1) + '/' + str(expr2)
            return tmp
    else:
        predicate_symbol_1, arg_list_1 = process_expression(expr1)
        predicate_symbol_2, arg_list_2 = process_expression(expr2)
        # Step 2
        if predicate_symbol_1 != predicate_symbol_2:
```

```
    return False
# Step 3
elif len(arg_list_1) != len(arg_list_2):
    return False
else:
    # Step 4: Create substitution list
    sub_list = list()
    # Step 5:
    for i in range(len(arg_list_1)):
        tmp = unify(arg_list_1[i], arg_list_2[i])
        if not tmp:
            return False
        elif tmp == 'Null':
            pass
        else:
            if type(tmp) == list:
                for j in tmp:
                    sub_list.append(j)
            else:
                sub_list.append(tmp)
    # Step 6
```

```
            return sub_list
if __name__ == '__main__':
    f1 = 'Q(a, g(x, a), f(y))'
    f2 = 'Q(a, g(f(b), a), x)'
    # f1 = input('f1 : ')
    # f2 = input('f2 : ')
    result = unify(f1, f2)
    if not result:
        print('The process of Unification failed!')
    else:
        print('The process of Unification successful!')
        print(result)
```

MANUAL CALCULATION:



Manual Calculation :-

Q) Perform Unification for $Q(a, g(x,a), f(y))$, $Q(a, g(f(b),a), x)$

Sol:- Here let $\psi_1 = Q(a, g(x,a), f(y))$

$\psi_2 = Q(a, g(f(b),a), x)$

Let us try to find σ such that $\psi_1\sigma = \psi_2\sigma$ doing substitution.

i) Substitution $f(b)/x$ then

$\psi_1 = Q(a, g(x,a), f(y))$

$\psi_2 = Q(a, g(x,a), x)$

ii) Substitution $f(y)/x$ then

$\psi_1 = Q(a, g(x,a), x)$

$\psi_2 = Q(a, g(x,a), x)$

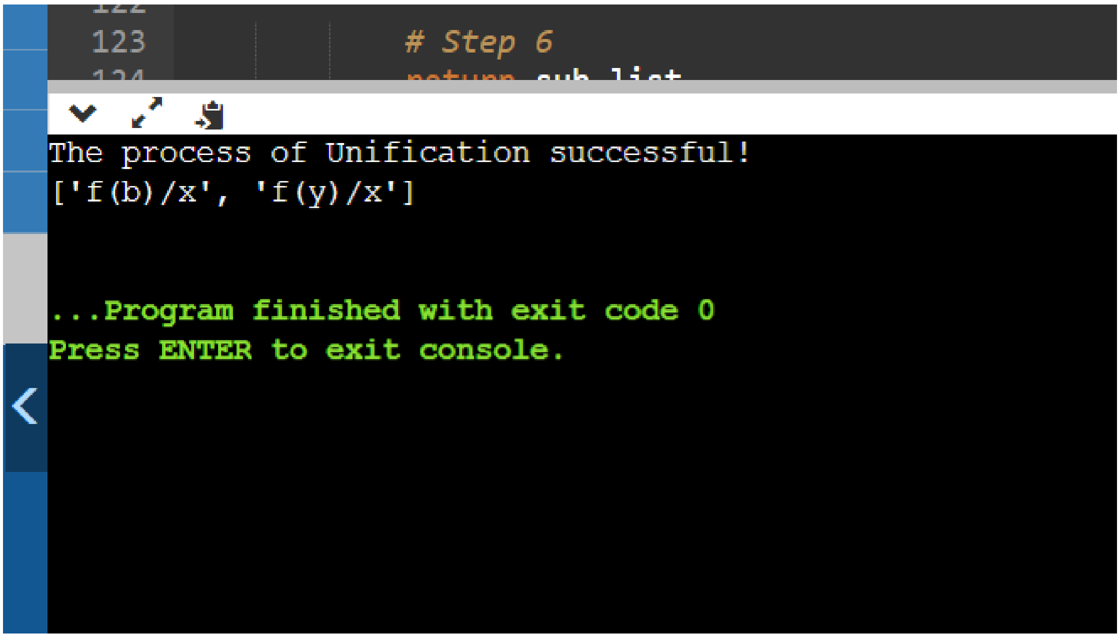
$\therefore \psi_1 = \psi_2 \Rightarrow$ succesfully Unified

Thus Unification is succesfully done using substitutions

i) $f(b)/x$

ii) $f(y)/x$

OUTPUT:



```
# Step 6
```

```
The process of Unification successful!
['f(b)/x', 'f(y)/x']


...Program finished with exit code 0
Press ENTER to exit console.
```

OBSERVATION: Thus the result of both the code and the manual calculation are same.

Conditions for Unification:

Following are some basic conditions for unification:

- Predicate symbol must be same, atoms or expression with different predicate symbol can never be unified.
- Number of Arguments in both expressions must be identical.
- Unification will fail if there are two similar variables present in the same expression.

RESULT: Hence , Unification is implemented successfully in python.

RESOLUTION

AIM: To perform Resolution for real world problems.

PROBLEM STATEMENT: The key idea for resolution method is to use the knowledge base and negated goal to obtain null clause(which indicates proof of contradiction).

ALGORITHM:

1)Conversion of facts into first-order logic.

2)Convert FOL statements into CNF

1. Eliminate all implication (→) and rewrite
2. Move negation (¬)inwards and rewrite
3. Rename variables or standardize variables
4. Eliminate existential instantiation quantifier by elimination.
5. Drop Universal quantifiers.

3)Negate the statement which needs to prove (proof by contradiction)

4)Draw resolution graph (unification).

CODE:

Resolution:

```
import copy

import time

class Parameter:

   variable_count = 1

   def __init__(self, name=None):

      if name:

         self.type = "Constant"

         self.name = name

      else:

         self.type = "Variable"

         self.name = "v" + str(Parameter.variable_count)

         Parameter.variable_count += 1
```

```
    def isConstant(self):
        return self.type == "Constant"
    def unify(self, type_, name):
        self.type = type_
        self.name = name
    def __eq__(self, other):
        return self.name == other.name
    def __str__(self):
        return self.name
class Predicate:
    def __init__(self, name, params):
        self.name = name
        self.params = params
    def __eq__(self, other):
        return self.name == other.name and all(a == b for a, b in zip(self.params, other.params))
    def __str__(self):
        return self.name + "(" + ",".join(str(x) for x in self.params) + ")"
    def getNegatedPredicate(self):
        return Predicate(negatePredicate(self.name), self.params)
class Sentence:
    sentence_count = 0
```

```
def __init__(self, string):
    self.sentence_index = Sentence.sentence_count
    Sentence.sentence_count += 1
    self.predicates = []
    self.variable_map = {}
    local = {}
    for predicate in string.split("|"):
        name = predicate[:predicate.find("(")]
        params = []
        for param in predicate[predicate.find("(") + 1: predicate.find(")")].split(","):
            if param[0].islower():
                if param not in local:  # Variable
                    local[param] = Parameter()
                    self.variable_map[local[param].name] = local[param]
                new_param = local[param]
            else:
                new_param = Parameter(param)
                self.variable_map[param] = new_param
            params.append(new_param)
        self.predicates.append(Predicate(name, params))
def getPredicates(self):
        return [predicate.name for predicate in self.predicates]


    def findPredicates(self, name):
        return [predicate for predicate in self.predicates if predicate.name == name]
    def removePredicate(self, predicate):
        self.predicates.remove(predicate)
        for key, val in self.variable_map.items():
            if not val:
                self.variable_map.pop(key)
    def containsVariable(self):
        return any(not param.isConstant() for param in self.variable_map.values())
    def __eq__(self, other):
        if len(self.predicates) == 1 and self.predicates[0] == other:
            return True
        return False
    def __str__(self):
        return "".join([str(predicate) for predicate in self.predicates])
class KB:
    def __init__(self, inputSentences):
        self.inputSentences = [x.replace(" ", "") for x in inputSentences]
        self.sentences = []
        self.sentence_map = {}

    def prepareKB(self):

        self.convertSentencesToCNF()

        for sentence_string in self.inputSentences:

            sentence = Sentence(sentence_string)

            for predicate in sentence.getPredicates():

                self.sentence_map[predicate] = self.sentence_map.get(

                    predicate, []) + [sentence]

    def convertSentencesToCNF(self):

        for sentenceIdx in range(len(self.inputSentences)):

            # Do negation of the Premise and add them as literal

            if "=>" in self.inputSentences[sentenceIdx]:

                self.inputSentences[sentenceIdx] = negateAntecedent(

                    self.inputSentences[sentenceIdx])

    def askQueries(self, queryList):

        results = []

        for query in queryList:

            negatedQuery = Sentence(negatePredicate(query.replace(" ", "")))

            negatedPredicate = negatedQuery.predicates[0]

            prev_sentence_map = copy.deepcopy(self.sentence_map)

            self.sentence_map[negatedPredicate.name] = self.sentence_map.get(
                negatedPredicate.name, []) + [negatedQuery]
            self.timeLimit = time.time() + 40
            try:
                result = self.resolve([negatedPredicate], [
                                      False]*(len(self.inputSentences) + 1))
            except:
                result = False
            self.sentence_map = prev_sentence_map
            if result:
                results.append("TRUE")
            else:
                results.append("FALSE")
        return results
    def resolve(self, queryStack, visited, depth=0):
        if time.time() > self.timeLimit:
            raise Exception
        if queryStack:
            query = queryStack.pop(-1)
            negatedQuery = query.getNegatedPredicate()
            queryPredicateName = negatedQuery.name
            if queryPredicateName not in self.sentence_map:
```

```
    return False
else:
    queryPredicate = negatedQuery
    for kb_sentence in self.sentence_map[queryPredicateName]:
        if not visited[kb_sentence.sentence_index]:
            for kbPredicate in kb_sentence.findPredicates(queryPredicateName):

                canUnify, substitution = performUnification(
                    copy.deepcopy(queryPredicate), copy.deepcopy(kbPredicate))
                if canUnify:
                    newSentence = copy.deepcopy(kb_sentence)
                    newSentence.removePredicate(kbPredicate)
                    newQueryStack = copy.deepcopy(queryStack)
                    if substitution:
                        for old, new in substitution.items():
                            if old in newSentence.variable_map:
                                parameter = newSentence.variable_map[old]
                                newSentence.variable_map.pop(old)
                                parameter.unify(
                                    "Variable" if new[0].islower() else "Constant", new)
                                newSentence.variable_map[new] = parameter
```

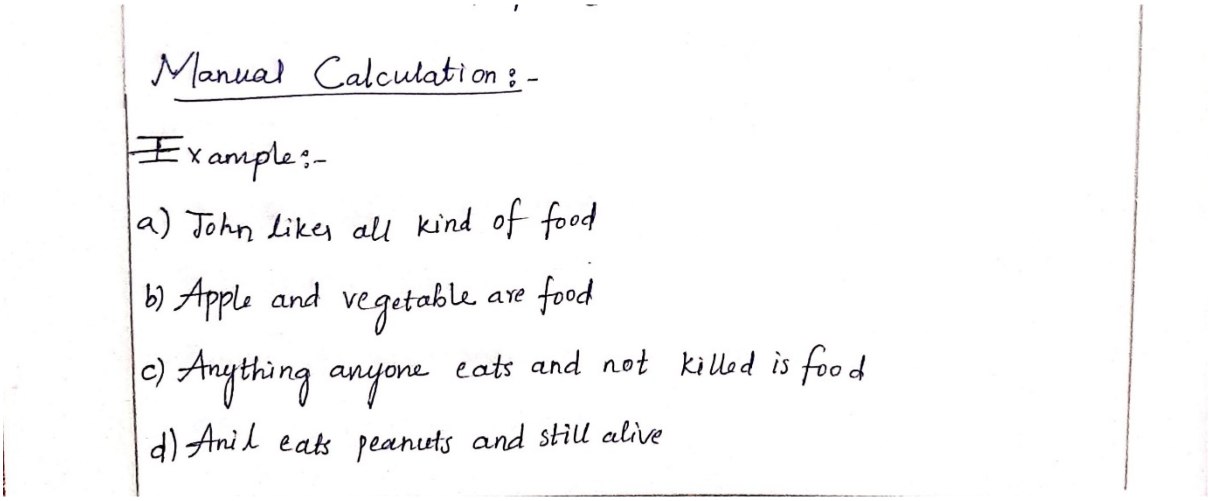
```
                                    for predicate in newQueryStack:

                                        for index, param in enumerate(predicate.params):

                                            if param.name in substitution:

                                                new = substitution[param.name]

                                                predicate.params[index].unify(

                                                    "Variable" if new[0].islower() else "Constant", new)

                                for predicate in newSentence.predicates:

                                    newQueryStack.append(predicate)

                                new_visited = copy.deepcopy(visited)

                                if kb_sentence.containsVariable() and len(kb_sentence.predicates) > 1:

                                    new_visited[kb_sentence.sentence_index] = True



                                if self.resolve(newQueryStack, new_visited, depth + 1):

                                    return True

                return False

        return True

def performUnification(queryPredicate, kbPredicate):

    substitution = {}

    if queryPredicate == kbPredicate:

        return True, {}

    else:
```

```
for query, kb in zip(queryPredicate.params, kbPredicate.params):
    if query == kb:
        continue
    if kb.isConstant():
        if not query.isConstant():
            if query.name not in substitution:
                substitution[query.name] = kb.name
            elif substitution[query.name] != kb.name:
                return False, {}
            query.unify("Constant", kb.name)
        else:
            return False, {}
    else:
        if not query.isConstant():
            if kb.name not in substitution:
                substitution[kb.name] = query.name
            elif substitution[kb.name] != query.name:
                return False, {}
            kb.unify("Variable", query.name)
        else:
            if kb.name not in substitution:
```

```
                        substitution[kb.name] = query.name
                    elif substitution[kb.name] != query.name:
                        return False, {}
    return True, substitution
def negatePredicate(predicate):
    return predicate[1:] if predicate[0] == "~" else "~" + predicate
def negateAntecedent(sentence):
    antecedent = sentence[:sentence.find("=>")]
    premise = []
    for predicate in antecedent.split("&"):
        premise.append(negatePredicate(predicate))
    premise.append(sentence[sentence.find("=>") + 2:])
    return "|".join(premise)
def getInput(filename):
    with open(filename, "r") as file:
        noOfQueries = int(file.readline().strip())
        inputQueries = [file.readline().strip() for _ in range(noOfQueries)]
        noOfSentences = int(file.readline().strip())
        inputSentences = [file.readline().strip()
                          for _ in range(noOfSentences)]
        return inputQueries, inputSentences
```

```
def printOutput(filename, results):
    print(results)
    with open(filename, "w") as file:
        for line in results:
            file.write(line)
            file.write("\n")
    file.close()
if __name__ == '__main__':
    inputQueries_, inputSentences_ = getInput(r"C:\Users\HP \Desktop\input.txt")
    knowledgeBase = KB(inputSentences_)
    knowledgeBase.prepareKB()
    results_ = knowledgeBase.askQueries(inputQueries_)
    printOutput("output.txt", results_)
```

MANUAL CALCULATION:



Manual Calculation:-

Example:-

a) John likes all kind of food

b) Apple and vegetable are food

c) Anything anyone eats and not killed is food

d) Anil eats peanuts and still alive

e) Harry eats everything that Anil eats

Prove by resolution that:

f) John likes peanuts.

Step1 : Conversion of Facts into FOL

a) $\forall x : food(x) \rightarrow likes(John, x)$

b) $food(Apple) \wedge food(vegetables)$

c) $\forall x \forall y : eats(x, y) \wedge \neg killed(x) \rightarrow food(y)$

d) $eats(Anil, Peanuts) \wedge alive(Anil)$

e) $\forall x : eats(Anil, x) \rightarrow eats(Harry, x)$

f) $\forall x : killed(x) \rightarrow alive(x)$

g) $\forall x : alive(x) \longrightarrow \neg killed(x)$

} added Predicates

h) $likes(John, Peanuts)$

Step 2:- Convresion FOL into CNF

a) $\forall x\ \neg food(x) \vee likes(John, x)$

b) $food(Apple) \wedge food(vegetables)$

c) $\forall x \forall y\ \neg [eats(x, y) \wedge \neg killed(x)] \vee food(y)$

d) $eats(Anil, Peanuts) \wedge alive(Anil)$

e) $\forall x\ \neg eats(Anil, x) \vee eats(Harry, x)$

f) $\forall x \neg alive(x)\ \vee \neg killed(x)$

h) $likes(John, Peanuts)$

Move negation ($\neg$) inwards and rewrite

a) $\forall x\ \neg food(x) \vee likes(John, x)$

b) $food(Apple) \wedge food(vegetables)$

c) $\forall x \forall y \neg eats(x, y) \vee killed(x) \vee food(y)$

d) eats (Anil, Peanuts) ∧ alive (Anil)

e) ∀w ¬eats (Anil, w) ∨ eats (Harry, w)

f) ∀g ¬killed (g)] ∨ alive (g)

g) ∀k ¬alive (k) ∨ ¬killed (k)

h) likes (John, Peanuts)

Drop Universal quantifiers

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)

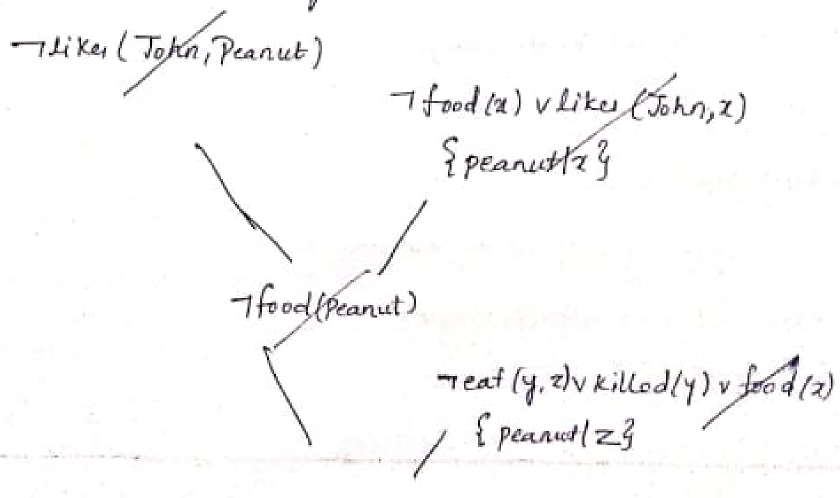
h) killed (g) ∨ alive (g)

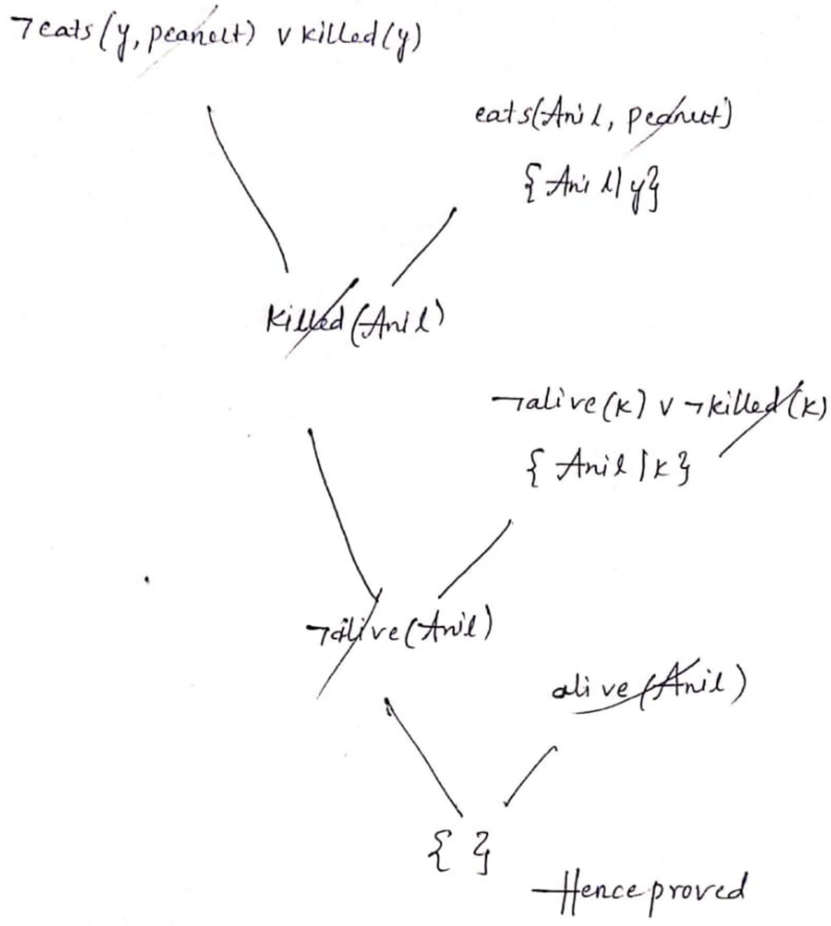
i) ¬alive (k) ∨ ¬killed (k)

j) likes (John, Peanuts)

<u>step 3</u>:- ¬likes (John, Peanuts)

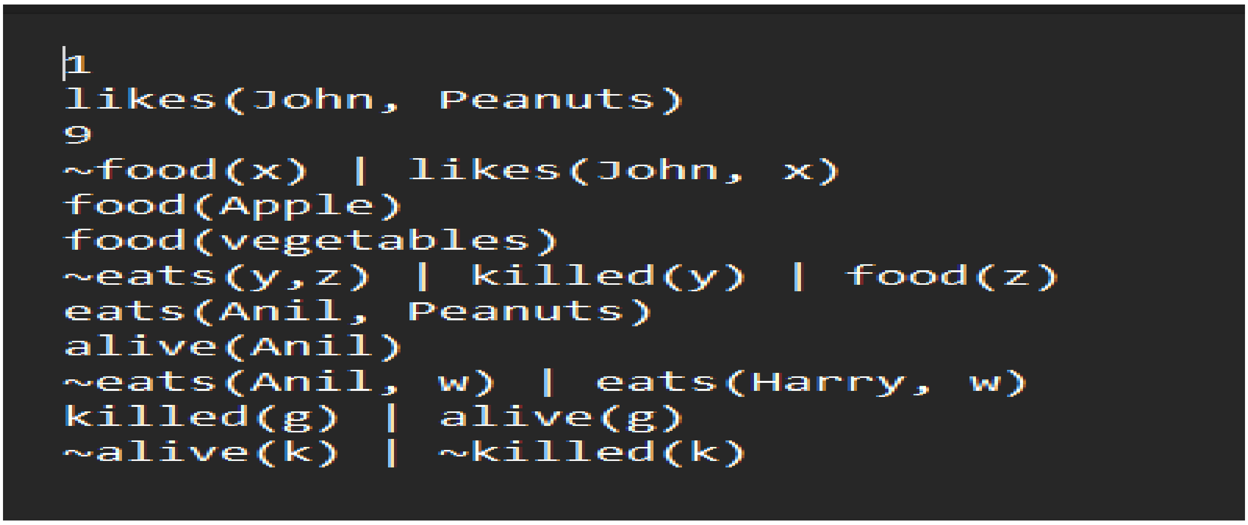
<u>step 4</u>:- Draw resolution graph


¬likes (John, Peanut)
¬food (x) ∨ likes (John, x)
{peanut/x}
¬food (Peanut)
¬eat (y, z) ∨ killed (y) ∨ food (z)
{peanut/z}

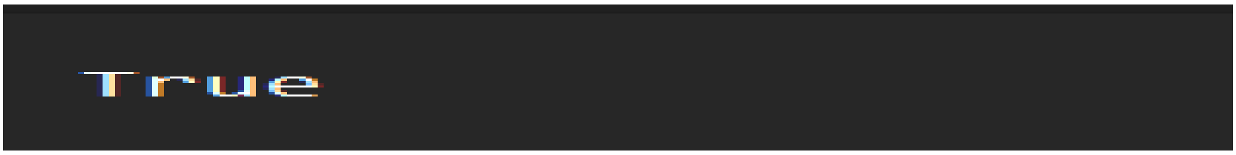


Hence the negation of the conclusion has been proved as a complete contradiction with given set of statements.

OUTPUT:



```
1
likes(John, Peanuts)
9
~food(x) | likes(John, x)
food(Apple)
food(vegetables)
~eats(y,z) | killed(y) | food(z)
eats(Anil, Peanuts)
alive(Anil)
~eats(Anil, w) | eats(Harry, w)
killed(g) | alive(g)
~alive(k) | ~killed(k)
```



```
True
```

OBSERVATION:

Thus the result of both the code and the manual calculation are same.

- In the first step of resolution graph, ¬likes(John, Peanuts) , and likes(John, x) get resolved(canceled) by substitution of {Peanuts/x}, and we are left with ¬ food(Peanuts)
- In the second step of the resolution graph, ¬ food(Peanuts) , and food(z) get resolved (canceled) by substitution of { Peanuts/z}, and we are left with ¬ eats(y, Peanuts) V killed(y) .
- In the third step of the resolution graph, ¬ eats(y, Peanuts) and eats (Anil, Peanuts) get resolved by substitution {Anil/y}, and we are left with Killed(Anil) .
- In the fourth step of the resolution graph, Killed(Anil) and ¬ killed(k) get resolve by substitution {Anil/k}, and we are left with ¬ alive(Anil) .
- In the last step of the resolution graph ¬ alive(Anil) and alive(Anil) get resolved.

RESULT: Hence , Resolution is implemented successfully in python.