EXP : 7 (Unification & Resolution)

Aim : Implementation of unification & to resolution in real world application.

(A) Unification (Pattern matching)

Problem Formulation :

To find a mapping between two expression that may both contain variables.

Print the variables to their values in the given expression until no bound variables remain.

Initial state :

exp 1 = 'f(x, h(x), Y, g(Y))'

exp 2 = 'f(g(z), w, z, x)'

Final state :

x : g(z)

w : h(x)

Y : z

exp 1 = f(g(z), h(g(z), z, g(z))

exp 2 = f(g(z), h(g(z)), z, g(z))

Problem Solving :

. unify f(x, h(x), Y, g(Y)) and f(g(z), w, z, x)

. It would loop through each argument.

. unify (x, g(z)) is involved.

here, x → variable , ∴ put x = g(z)

. unify (h(x), w) is involved

w → variable, ∴ w = h(x)

The substitutions are mapped to a python dictionary & it expands as $\{X = g(z), W = h(x)\}$

- unify (Y, z) is involved

  Y, z → variables, ∴ add to dictionary

  $\{X = g(z), W = h(x), Y = z\}$ # z = Y or Y = z are same

- unify (g(Y), x) is involved

  x → variable but already present in dict.

  ∴ (g(Y), g(z))

  ↓

  both have g

  ∴ unify Y & z

  ↳ already present.

- all variables are bounded, unification is complete.

Final result : $\{X = g(z), W = h(x), Y = z\}$

Algorithm :

1. Start
2. Declare a python dict mapping variables to terms
3. When either side is a variable, it calls unify - variable
4. Else, if both sides are func applications it ensures they apply the same func.
5. If v is bound in the substitution, we try to unify its defn
6. Stop

(B) Implementation of Resolution (Predicate logic)

Problem Formation :

By building reputation proofs, ie, proof by contradictions prove a conclution of most given statements based on the conjuctive normal form or clausal form

Initial state

a. John likes all kind of food
b. Apples & vegetable are food
c. Anything anyone eats & not killed is food
d. Anil eats peanuts & still alive
e. Harry eats everything that Anil eats

Final state

'TRUE' (proved)

Prove by resolution :

f. John likes peanuts

Problem solving

. 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. $\forall x : \neg killed(x) \rightarrow alive(x)$
g. $\forall x : alive(x) \rightarrow \neg killed(x)$
h. $Likes(John, Peanuts)$

- Elimination of implication, moving negation inwards & renaming variables

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

b. $food(Apple) \wedge food(vegetable)$

c. $\forall y \forall z \neg [eats(y,z) \vee killed(y)] \vee food(z)$

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

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

f. $\forall g \neg killed(g) \vee alive(g)$

g. $\forall k \neg alive(k) \vee killed(k)$

h. $likes(John, Peanuts)$

- Drop existential qualifiers.

a. $food(x) \vee likes(John, x)$

b. $food(Apple)$

c. $food(vegetables)$

d. $\neg eats(y,z) \vee killed(y) \vee food(z)$

e. $eats(Anil, Peanuts)$

f. $alive(Anil)$

g. $\neg eats(Anil, w) \vee eats(Harry, w)$

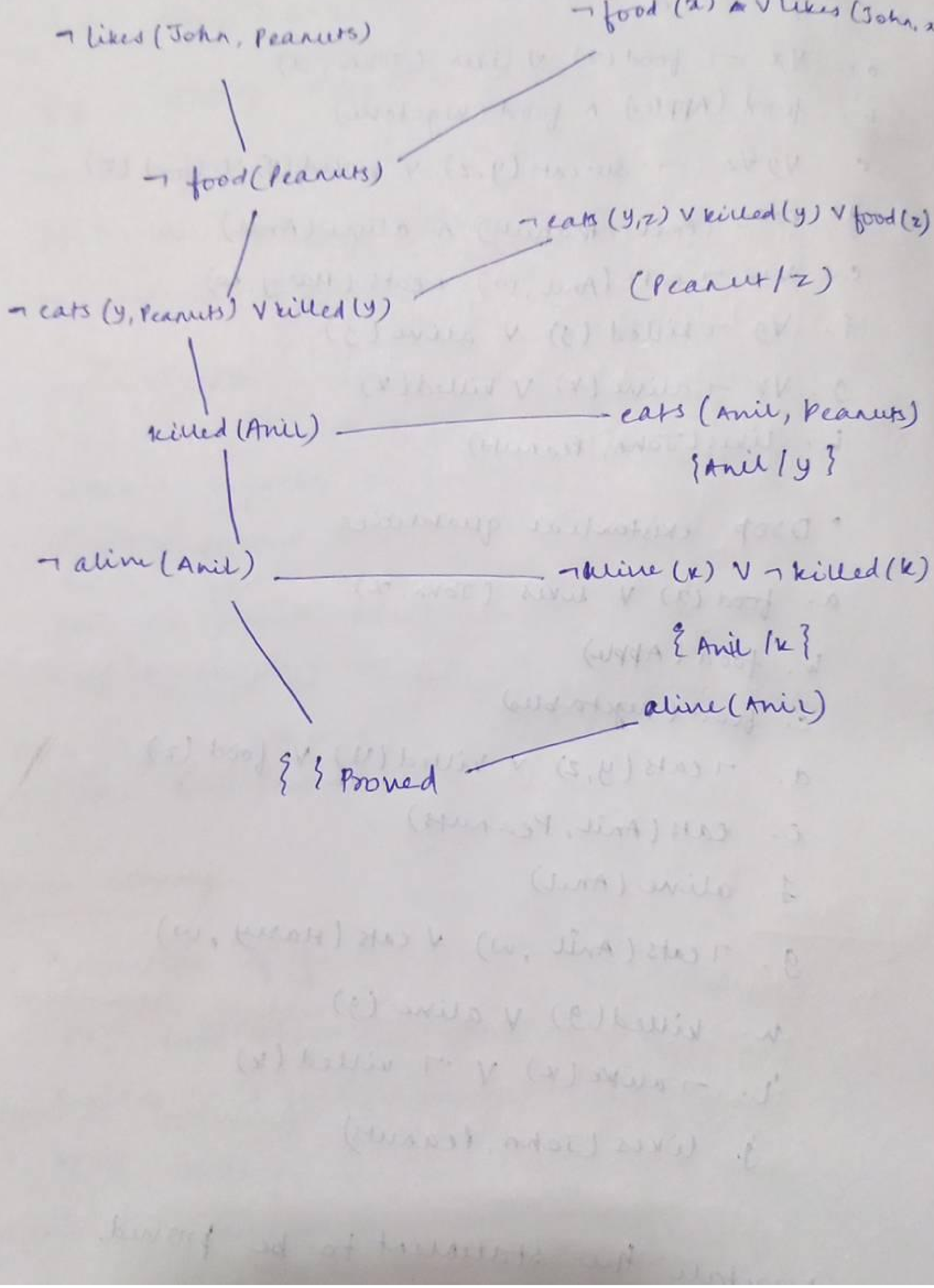
h. $killed(g) \vee alive(g)$

i. $\neg alive(k) \vee \neg killed(k)$

j. $likes(John, Peanuts)$

- Negate the statement to be proved.

k. $\neg likes(John, Peanuts)$



Unification

Code:

```
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
```

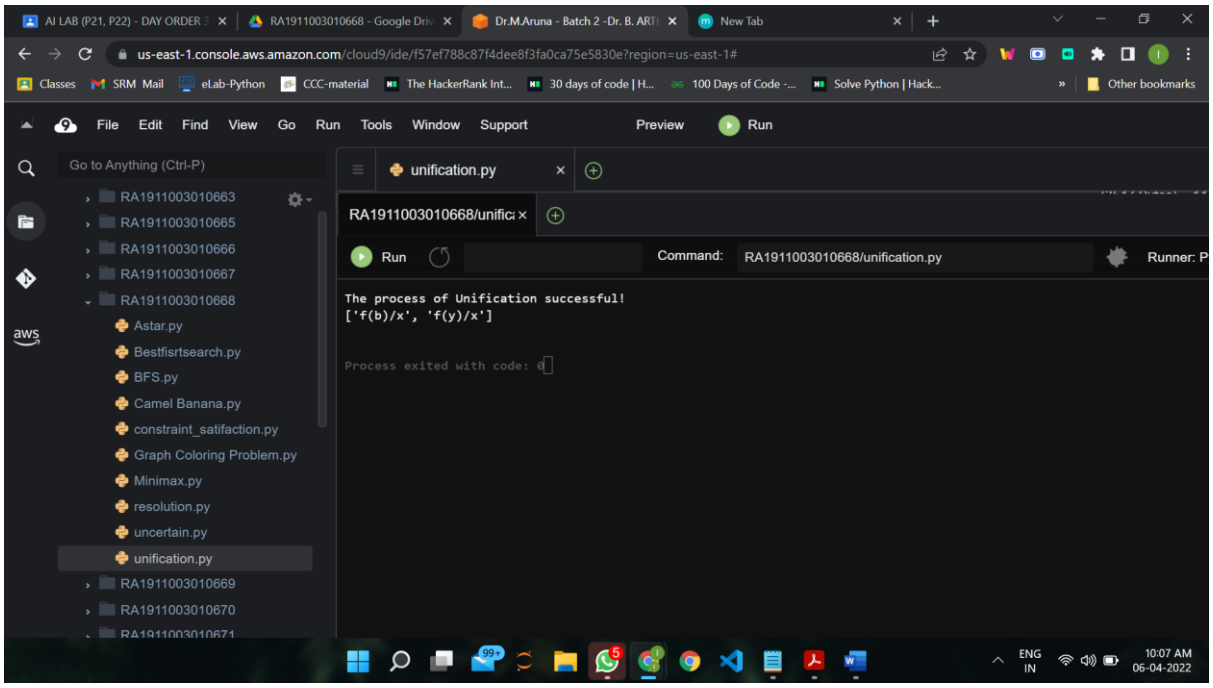
```
    for i in range(len(expr)):
        if expr[i] == '(':
            index = i
            break
    predicate_symbol = expr[:index]
    expr = expr.replace(predicate_symbol, '')
    expr = expr[1:len(expr) - 1]
    arg_list = list()
    indices = get_index_comma(expr)

    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 = 'Q(hello, f(why)'
    #f2 = 'Q(a, f(g(no))'


    result = unify(f1, f2)
    if not result:
        print('The process of Unification failed!')
    else:
        print('The process of Unification successful!')
        print(result)
```

Output:



Resolution

Algorithm:

Step-1: Start
Step-2: if L1 or L2 is an atom part of same thing do
(a) if L1 or L2 are identical then return NIL
(b) else if L1 is a variable then do
(i) if L1 occurs in L2 then return F else return (L2/L1)
else if L2 is a variable then do
(i) if L2 occurs in L1 then return F else return (L1/L2)
else return F.
Step-3: If length (L!) is not equal to length (L2) then return F.
Step-4: Set SUBST to NIL
(at the end of this procedure , SUBST will contain all the substitutions used to unify L1 and L2).
Step-5: For I = 1 to number of elements in L1 do
i) call UNIFY with the i th element of L1 and I'th element of L2, putting the result in S
ii) if S = F then return F
iii) if S is not equal to NIL then do
(A) apply S to the remainder of both L1 and L2
(B) SUBST := APPEND (S, SUBST) return SUBST.

Step-6: Stop.

Code:

```
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('RA1911003010675/Input.txt')
    knowledgeBase = KB(inputSentences_)
    knowledgeBase.prepareKB()
    results_ = knowledgeBase.askQueries(inputQueries_)
    printOutput("RA1911003010675/output.txt", results_)
```

Output: