LAB-2

Question:

1.Create a knowledgebase using prepositional logic and show that the given query entails the knowledge base or not.

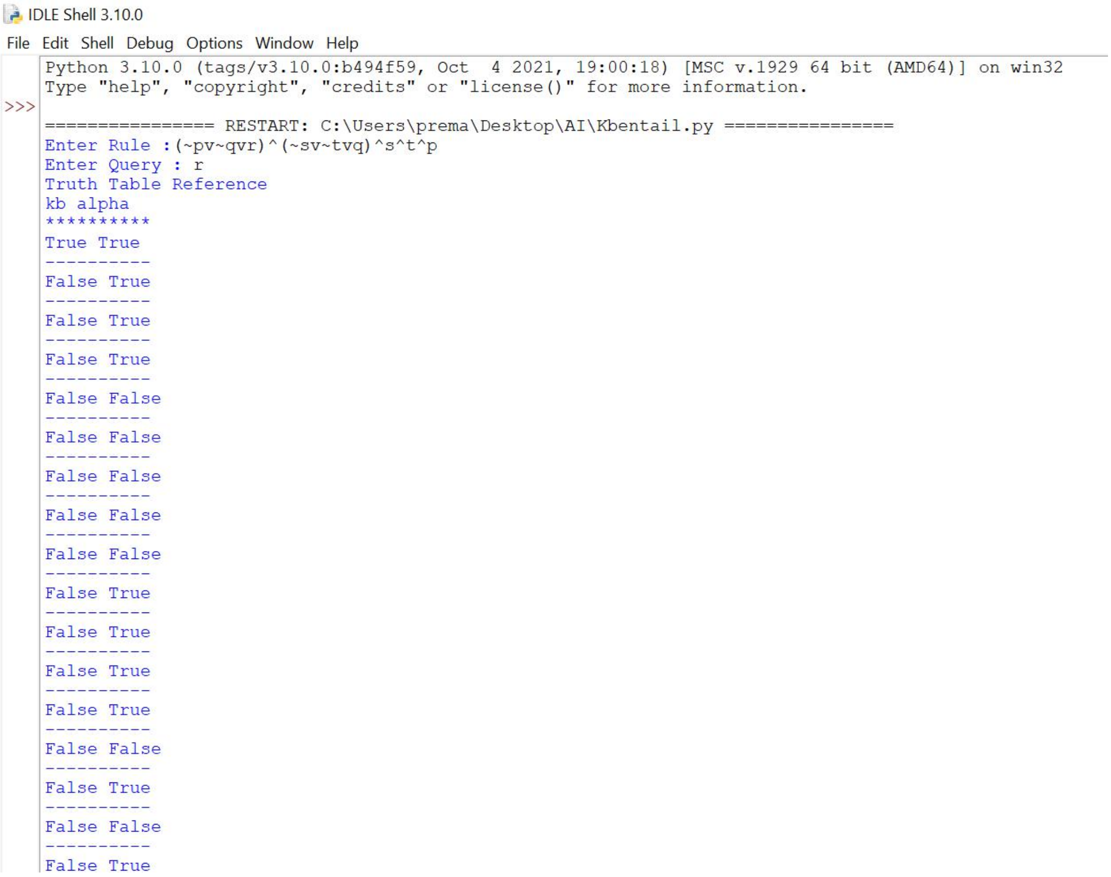
Program:

```
combinations = [(True, True, True,True,True),
(True, True, True,True,False),
(True, True, True,False,True),
(True, True, True,False,False),
(True, True, False,True,True),
(True, True,False,True,False),
(True, True, False,False,True),
(True, True, False,False,False),
(True, False, False,False,False),
(True, False, True,True,True),
(True, False, True,True,False),
(True, False, True,False,True),
(True, False, True,False,False),
(True, False, False,True,True),
(True, False, True,False,False),
(True, False, False,True,False),
(False, True, True,True,True),
(False, True, True,True,False),
(False, True, True,False,True),
(False, True, True,False,False),
(False, True, False,True,True),
(False, True, False,False,True),
(False, True, False,False,False),
(False, False, True,True,True),
(False, False, True,True,False),
(False, False, True,False,True),
(False, False, True,False,False),
(False, False, False,True,True),
(False, False, False,True,False),
(False, False, False,False,True),
(True, False, False,False,True),
(False, False, False,False,False),


]
variable = {'p':0, 'q':1, 'r':2,'s':3,'t':4}
priority = {'v':1, '^':2, '~':3}

# set of rules
kb = '' # should be a cnf
q =  '' # should be a cnf
def isOperand(c):
  return c.isalpha() and c != 'v'

def isLeftParenthesis(c):
  return c == "("

def isRightParenthesis(c):
  return c == ")"
```

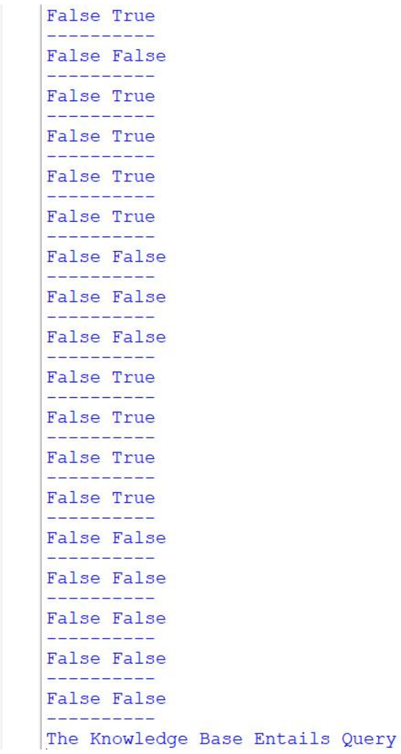
```
def isEmpty(stack):
 return len(stack) == 0

def peek(stack):
 return stack[-1]

def hasLessOrEqualPriority(c1, c2):
 try: return priority[c1] <= priority[c2]
 except KeyError: return False
def toPostfix(infix):
 stack = []
 postfix = ''
 for c in infix:
  if isOperand(c):
   postfix += c
  else:
   if isLeftParenthesis(c):
    stack.append(c)
   elif isRightParenthesis(c):
    operator = stack.pop()
    while not isLeftParenthesis(operator):
     postfix += operator
     operator = stack.pop()
   else:
    while (not isEmpty(stack)) and hasLessOrEqualPriority(c, peek(stack)):
     postfix += stack.pop()
    stack.append(c)
 while (not isEmpty(stack)):
  postfix += stack.pop()
 return postfix
def _eval(i, val1, val2):
  if i == '^': return val2 and val1
  return val2 or val1

def evaluatePostfix(exp, comb):
 stack = []
 for i in exp:
  if isOperand(i):
   stack.append(comb[variable[i]])
  elif i == '~':
   val1 = stack.pop()
   stack.append(not val1)
  else:
   val1 = stack.pop()
   val2 = stack.pop()
   stack.append(_eval(i, val2, val1))
 return stack.pop()
def input_rules():
 global kb, q
 kb = input("Enter Rule :")
 q = input("Enter Query : ")

def entailment():
 global kb, q
 print('' * 10 + "Truth Table Reference" + "" * 10)
 print("kb", "alpha")
 print("*" * 10)
```

```
  for comb in combinations:
    s = evaluatePostfix(toPostfix(kb), comb)
    f = evaluatePostfix(toPostfix(q), comb)
    print(s, f)
    print("-" * 10)
    if s and not f:
      return False
  return True
input_rules()
ans = entailment()
if ans: print("The Knowledge Base Entails Query")
else: print("The Knowledge Base Doesn't Entail Query")
```

Output:



```
    Python 3.10.0 (tags/v3.10.0:b494f59, Oct  4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
    Type "help", "copyright", "credits" or "license()" for more information.
>>>
    ================ RESTART: C:\Users\prema\Desktop\AI\Kbentail.py ================
    Enter Rule :(~pv~qvr)^(~sv~tvq)^s^t^p
    Enter Query : r
    Truth Table Reference
    kb alpha
    **********
    True True
    ----------
    False True
    ----------
    False True
    ----------
    False True
    ----------
    False False
    ----------
    False False
    ----------
    False False
    ----------
    False False
    ----------
    False False
    ----------
    False True
    ----------
    False True
    ----------
    False True
    ----------
    False True
    ----------
    False False
    ----------
    False True
    ----------
    False False
    ----------
    False True
```



```
False True
----------
False False
----------
False True
----------
False True
----------
False True
----------
False True
----------
False False
----------
False False
----------
False False
----------
False True
----------
False True
----------
False True
----------
False True
----------
False False
----------
False False
----------
False False
----------
False False
----------
False False
----------
The Knowledge Base Entails Query
```

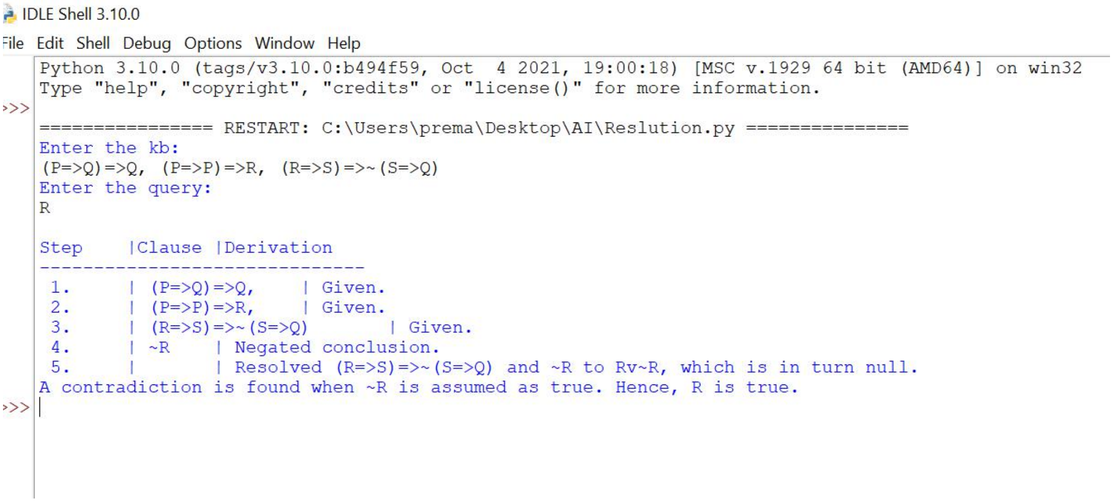
2.Create a knowledgebase using prepositional logic and prove the
given query using resolution

Program:

```
# Create a knowledgebase using prepositional logic and prove the given query using resolution.

import re


def negate(term):
    return f'~{term}' if term[0] != '~' else term[1]


def reverse(clause):
    if len(clause) > 2:
        t = split_terms(clause)
        return f'{t[1]}v{t[0]}'
    return ''


def split_terms(rule):
    exp = '(~*[PQRS])'
    terms = re.findall(exp, rule)
    return terms


def contradiction(query, clause):
    contradictions = [f'{query}v{negate(query)}', f'{negate(query)}v{query}']
    return clause in contradictions or reverse(clause) in contradictions


def resolve(kb, query):
    temp = kb.copy()
    temp += [negate(query)]
    steps = dict()
    for rule in temp:
        steps[rule] = 'Given.'
    steps[negate(query)] = 'Negated conclusion.'
    i = 0
    while i < len(temp):
        n = len(temp)
        j = (i + 1) % n
        clauses = []
        while j != i:
            terms1 = split_terms(temp[i])
            terms2 = split_terms(temp[j])
            for c in terms1:
                if negate(c) in terms2:
                    t1 = [t for t in terms1 if t != c]
                    t2 = [t for t in terms2 if t != negate(c)]
                    gen = t1 + t2
                    if len(gen) == 2:
                        if gen[0] != negate(gen[1]):
                            clauses += [f'{gen[0]}v{gen[1]}']
                        else:
                            if contradiction(query, f'{gen[0]}v{gen[1]}'):
                                temp.append(f'{gen[0]}v{gen[1]}')
                                steps[''] = f"Resolved {temp[i]} and {temp[j]} to {temp[-1]}, which is in turn null. \
                                \nA contradiction is found when {negate(query)} is assumed as true. Hence, {query}
is true."
                                return steps
                    elif len(gen) == 1:
                        clauses += [f'{gen[0]}']
                    else:
                        if contradiction(query, f'{terms1[0]}v{terms2[0]}'):
                            temp.append(f'{terms1[0]}v{terms2[0]}')
                            steps[''] = f"Resolved {temp[i]} and {temp[j]} to {temp[-1]}, which is in turn null. \
                            \nA contradiction is found when {negate(query)} is assumed as true. Hence, {query} is
true."
                            return steps
            for clause in clauses:
                if clause not in temp and clause != reverse(clause) and reverse(clause) not in temp:
                    temp.append(clause)
                    steps[clause] = f'Resolved from {temp[i]} and {temp[j]}.'
            j = (j + 1) % n
        i += 1
    return steps


def resolution(kb, query):
    kb = kb.split(' ')
    steps = resolve(kb, query)
    print('\nStep\t|Clause\t|Derivation\t')
    print('-' * 30)
    i = 1
    for step in steps:
        print(f' {i}.\t| {step}\t| {steps[step]}\t')
        i += 1
```

```
def main():
    print("Enter the kb:")
    kb = input()
    print("Enter the query:")
    query = input()
    resolution(kb, query)


# test 1
# (P^Q)<=>R : (Rv~P)v(Rv~Q)^(~RvP)^(~RvQ)
# Rv~P Rv~Q ~RvP ~RvQ
# R
main()

# test 2
# (P=>Q)=>Q, (P=>P)=>R, (R=>S)=>~(S=>Q)
# PvQ PvR ~PvR RvS Rv~Q ~Sv~Q
# R
```

Output:



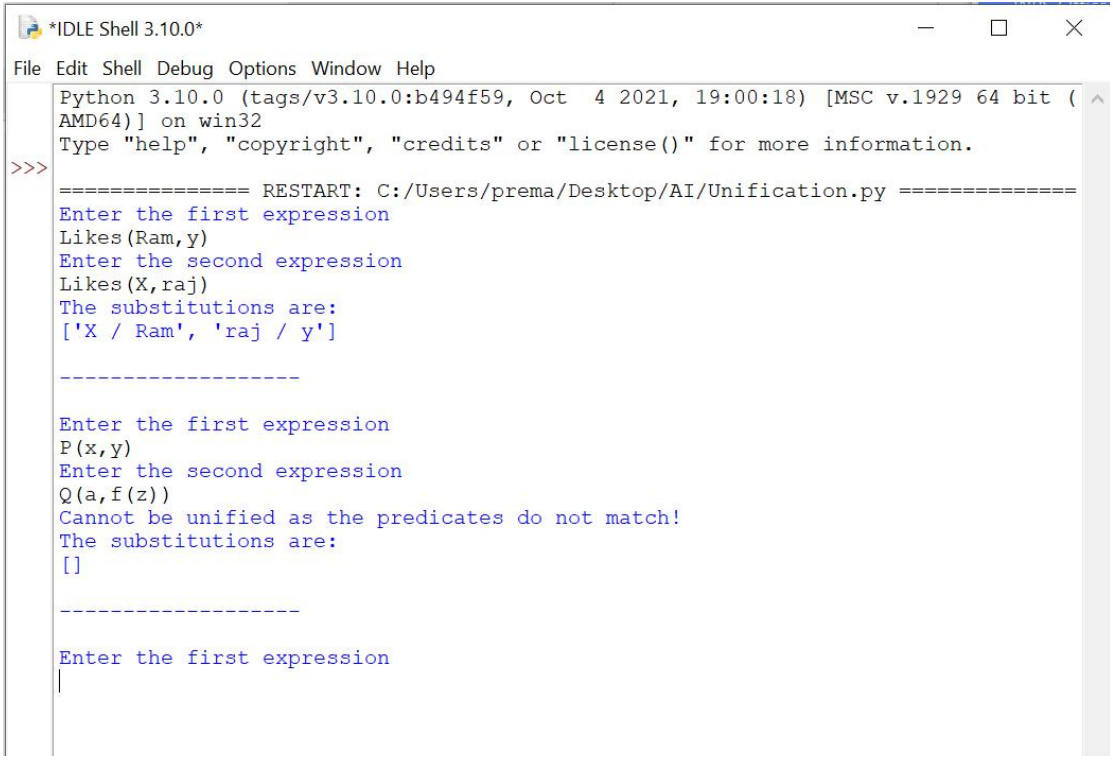
```
Python 3.10.0 (tags/v3.10.0:b494f59, Oct  4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
================ RESTART: C:\Users\prema\Desktop\AI\Reslution.py ===============
Enter the kb:
(P=>Q)=>Q, (P=>P)=>R, (R=>S)=>~(S=>Q)
Enter the query:
R

Step    |Clause |Derivation
------------------------------
 1.     | (P=>Q)=>Q,    | Given.
 2.     | (P=>P)=>R,    | Given.
 3.     | (R=>S)=>~(S=>Q)       | Given.
 4.     | ~R    | Negated conclusion.
 5.     |       | Resolved (R=>S)=>~(S=>Q) and ~R to Rv~R, which is in turn null.
A contradiction is found when ~R is assumed as true. Hence, R is true.
>>> 
```

3.Implement unification in first order logic

Program:

```
import re
def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression.split(")")[:-1]
    expression = ")".join(expression)
    attributes = expression.split(',')
    return attributes

def getInitialPredicate(expression):
    return expression.split("(")[0]
def isConstant(char):
    return char.isupper() and len(char) == 1

def isVariable(char):
    return char.islower() and len(char) == 1
def replaceAttributes(exp, old, new):
    attributes = getAttributes(exp)
    predicate = getInitialPredicate(exp)
    for index, val in enumerate(attributes):
        if val == old:
            attributes[index] = new
    return predicate + "(" + ",".join(attributes) + ")"

def apply(exp, substitutions):
    for substitution in substitutions:
        new, old = substitution
        exp = replaceAttributes(exp, old, new)
    return exp
def checkOccurs(var, exp):
    if exp.find(var) == -1:
        return False
    return True


def getFirstPart(expression):
    attributes = getAttributes(expression)
    return attributes[0]


def getRemainingPart(expression):
    predicate = getInitialPredicate(expression)
    attributes = getAttributes(expression)
    newExpression = predicate + "(" + ",".join(attributes[1:]) + ")"
    return newExpression
def unify(exp1, exp2):
    if exp1 == exp2:
        return []

    if isConstant(exp1) and isConstant(exp2):
        if exp1 != exp2:
            print(f"{exp1} and {exp2} are constants. Cannot be unified")
            return []

    if isConstant(exp1):
        return [(exp1, exp2)]

    if isConstant(exp2):
        return [(exp2, exp1)]

    if isVariable(exp1):
        return [(exp2, exp1)] if not checkOccurs(exp1, exp2) else []

    if isVariable(exp2):
        return [(exp1, exp2)] if not checkOccurs(exp2, exp1) else []

    if getInitialPredicate(exp1) != getInitialPredicate(exp2):
        print("Cannot be unified as the predicates do not match!")
        return []

    attributeCount1 = len(getAttributes(exp1))
    attributeCount2 = len(getAttributes(exp2))
    if attributeCount1 != attributeCount2:
        print(f"Length of attributes {attributeCount1} and {attributeCount2} do not match. Cannot be
unified")
        return []

    head1 = getFirstPart(exp1)
    head2 = getFirstPart(exp2)
    initialSubstitution = unify(head1, head2)
    if not initialSubstitution:
        return []
    if attributeCount1 == 1:
        return initialSubstitution

    tail1 = getRemainingPart(exp1)
    tail2 = getRemainingPart(exp2)

    if initialSubstitution != []:
        tail1 = apply(tail1, initialSubstitution)
        tail2 = apply(tail2, initialSubstitution)

    remainingSubstitution = unify(tail1, tail2)
    if not remainingSubstitution:
        return []

    return initialSubstitution + remainingSubstitution
def main():
    print("Enter the first expression")
    e1 = input()
    print("Enter the second expression")
    e2 = input()
    substitutions = unify(e1, e2)
    print("The substitutions are:")
    print([' / '.join(substitution) for substitution in substitutions])
main()
print(" ")
print("------------------- ")
print(" ")
main()
print(" ")
print("------------------- ")
print(" ")
main()
print(" ")
print("------------------- ")
print(" ")
main()
print("-------------------------- ")
print("---------------------------")
```

Output:



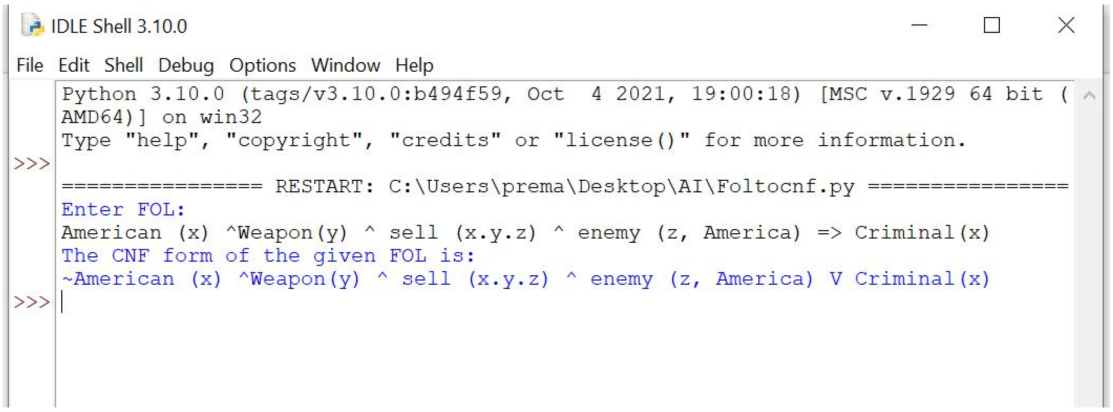
```
Python 3.10.0 (tags/v3.10.0:b494f59, Oct  4 2021, 19:00:18) [MSC v.1929 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=============== RESTART: C:/Users/prema/Desktop/AI/Unification.py ==============
Enter the first expression
Likes(Ram,y)
Enter the second expression
Likes(X,raj)
The substitutions are:
['X / Ram', 'raj / y']

--------------------

Enter the first expression
P(x,y)
Enter the second expression
Q(a,f(z))
Cannot be unified as the predicates do not match!
The substitutions are:
[]

--------------------

Enter the first expression
```

4.Convert given first order logic statement into Conjunctive Normal Form (CNF).

PROGRAM:

```
# Convert given first order logic statement into Conjunctive Normal Form (CNF).


import re


def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return [m for m in str(matches) if m.isalpha()]


def getPredicates(string):
    expr = '[a-z~]+\([A-Za-z,]+\)'
    return re.findall(expr, string)


def DeMorgan(sentence):
    string = ''.join(list(sentence).copy())
    string = string.replace('~~', '')
    flag = '[' in string
    string = string.replace('~[', '')
    string = string.strip(']')
    for predicate in getPredicates(string):
        string = string.replace(predicate, f'~{predicate}')
    s = list(string)
    for i, c in enumerate(string):
        if c == 'V':
            s[i] = '^'
        elif c == '^':
            s[i] = 'V'
    string = ''.join(s)
    string = string.replace('~~', '')
    return f'[{string}]' if flag else string


def Skolemization(sentence):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)]
    statement = ''.join(list(sentence).copy())
    matches = re.findall('[∀ ∃].', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, '')
        statements = re.findall('\[\[[^]]+\]]', statement)
        for s in statements:
            statement = statement.replace(s, s[1:-1])
        for predicate in getPredicates(statement):
            attributes = getAttributes(predicate)
            if ''.join(attributes).islower():
                statement = statement.replace(
                    match[1], SKOLEM_CONSTANTS.pop(0))
            else:
                aL = [a for a in attributes if a.islower()]
                aU = [a for a in attributes if not a.islower()][0]
                statement = statement.replace(
                    aU, f'{SKOLEM_CONSTANTS.pop(0)}({aL[0] if len(aL) else match[1]})')
    return statement


def fol_to_cnf(fol):

    statement = fol.replace("<=>", "_")
    while '_' in statement:
        i = statement.index('_')
        new_statement = '[' + statement[:i] + '=>' + statement[i +
                                                              1:] + ']^[' + statement[i+1:] + '=>' + statement[:i] + ']'
        statement = new_statement
    statement = statement.replace("=>", "-")
    expr = '\[([^]]+)\]'
    statements = re.findall(expr, statement)
    for i, s in enumerate(statements):
        if '[' in s and ']' not in s:
            statements[i] += ']'
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))
    while '-' in statement:
        i = statement.index('-')
        br = statement.index('[') if '[' in statement else 0
        new_statement = '~' + statement[br:i] + 'V' + statement[i+1:]
        statement = statement[:br] + new_statement if br > 0 else new_statement
    while '~∀' in statement:
        i = statement.index('~∀')
        statement = list(statement)
        statement[i], statement[i+1], statement[i+2] = '∃', statement[i+2], '~'
        statement = ''.join(statement)
    while '~∃' in statement:
        i = statement.index('~∃')
        s = list(statement)
        s[i], s[i+1], s[i+2] = '∀', s[i+2], '~'
        statement = ''.join(s)
    statement = statement.replace('~[∀', '[~∀')
    statement = statement.replace('~[∃', '[~∃')
    expr = '(~[∀V∃].)'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))
    expr = '~\[[^]]+\]'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, DeMorgan(s))
    return statement


def main():
    print("Enter FOL:")
    fol = input()
    print("The CNF form of the given FOL is: ")
    print(Skolemization(fol_to_cnf(fol)))


# Test 1
main()

# ∀x food(x) => likes(John, x)


# ∀x[∃z[loves(x, z)]]


# [american(x) ^ weapon(y) ^ sells(x, y, z) ^ hostile(z)] => criminal(x)
```

OUTPUT:



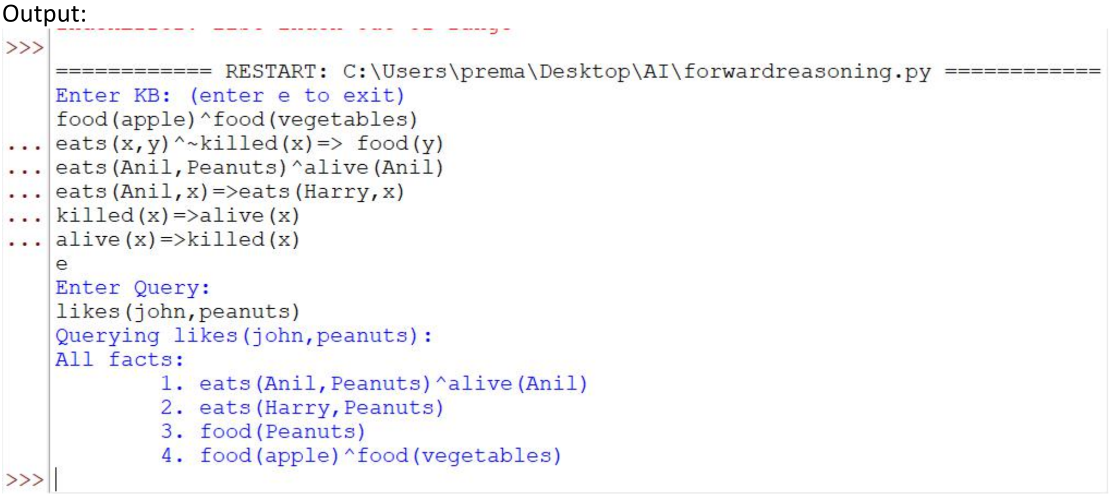
```
Python 3.10.0 (tags/v3.10.0:b494f59, Oct  4 2021, 19:00:18) [MSC v.1929 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
================ RESTART: C:\Users\prema\Desktop\AI\Foltocnf.py ================
Enter FOL:
American (x) ^Weapon(y) ^ sell (x.y.z) ^ enemy (z, America) => Criminal(x)
The CNF form of the given FOL is:
~American (x) ^Weapon(y) ^ sell (x.y.z) ^ enemy (z, America) V Criminal(x)
>>>
```

5.Create a knowledgebase consisting of first order logic statements
and prove the given query using forward reasoning

Program:

```
# Create a knowledgebase consisting of first order logic statements and prove the given
# query using forward reasoning.

import re


def isVariable(x):
    return len(x) == 1 and x.islower() and x.isalpha()


def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return matches


def getPredicates(string):
    expr = '([a-z~]+)\([^&|]+\)'
    return re.findall(expr, string)


class Fact:
    def __init__(self, expression):
        self.expression = expression
        predicate, params = self.splitExpression(expression)
        self.predicate = predicate
        self.params = params
        self.result = any(self.getConstants())

    def splitExpression(self, expression):
        predicate = getPredicates(expression)[0]
        params = getAttributes(expression)[0].strip('()').split(',')
        return [predicate, params]

    def getResult(self):
        return self.result
    def getConstants(self):
        return [None if isVariable(c) else c for c in self.params]

    def getVariables(self):
        return [v if isVariable(v) else None for v in self.params]

    def substitute(self, constants):
        c = constants.copy()
        f = f"{self.predicate}({','.join([constants.pop(0) if isVariable(p) else p for p in self.params])})"
        return Fact(f)


class Implication:
    def __init__(self, expression):
        self.expression = expression
        l = expression.split('=>')
        self.lhs = [Fact(f) for f in l[0].split('&')]
        self.rhs = Fact(l[1])

    def evaluate(self, facts):
        constants = {}
        new_lhs = []
        for fact in facts:
            for val in self.lhs:
                if val.predicate == fact.predicate:
                    for i, v in enumerate(val.getVariables()):
                        if v:
                            constants[v] = fact.getConstants()[i]
                    new_lhs.append(fact)
        predicate, attributes = getPredicates(self.rhs.expression)[
            0], str(getAttributes(self.rhs.expression)[0])
        for key in constants:
            if constants[key]:
                attributes = attributes.replace(key, constants[key])
        expr = f'{predicate}{attributes}'
        return Fact(expr) if len(new_lhs) and all([f.getResult() for f in new_lhs]) else None


class KB:
    def __init__(self):
        self.facts = set()
        self.implications = set()

    def tell(self, e):
        if '=>' in e:
            self.implications.add(Implication(e))
        else:
            self.facts.add(Fact(e))
        for i in self.implications:
            res = i.evaluate(self.facts)
            if res:
                self.facts.add(res)

    def query(self, e):
        facts = set([f.expression for f in self.facts])
        i = 1
        print(f'Querying {e}:')
        for f in facts:
            if Fact(f).predicate == Fact(e).predicate:
                print(f'\t{i}. {f}')
                i += 1

    def display(self):
        print("All facts: ")
        for i, f in enumerate(set([f.expression for f in self.facts])):
            print(f'\t{i+1}. {f}')


def main():
    kb = KB()
    print("Enter KB: (enter e to exit)")
    while True:
        t = input()
        if(t == 'e'):
            break
        kb.tell(t)
    print("Enter Query:")
    q = input()
    kb.query(q)
    kb.display()


main()

# missile(x) = >weapon(x)
# missile(M1)
# enemy(x, America) = >hostile(x)
# american(West)
#enemy(Nono, America)
# owns(Nono, M1)
# missile(x) & owns(Nono, x) = >sells(West, x, Nono)
# american(x) & weapon(y) & sells(x, y, z) & hostile(z) = >criminal(x)
# e

# criminal(x)
```

Output:



```
>>>
    ============ RESTART: C:\Users\prema\Desktop\AI\forwardreasoning.py ============
    Enter KB:  (enter e to exit)
    food(apple)^food(vegetables)
... eats(x,y)^~killed(x)=> food(y)
... eats(Anil,Peanuts)^alive(Anil)
... eats(Anil,x)=>eats(Harry,x)
... killed(x)=>alive(x)
... alive(x)=>killed(x)
    e
    Enter Query:
    likes(john,peanuts)
    Querying likes(john,peanuts):
    All facts:
            1. eats(Anil,Peanuts)^alive(Anil)
            2. eats(Harry,Peanuts)
            3. food(Peanuts)
            4. food(apple)^food(vegetables)
>>>
```