**Name: Sakshi P Khandoba**

**USN: 1BM19CS139**

**AI LAB REPORT**

**Program 1:**

**Implement Tic –Tac –Toe Game.**

**Code:**

board = [' ' for x in range(10)]

def insertLetter(letter, pos):

board[pos] = letter

def spaceIsFree(pos):

return board[pos] == ' '

def printBoard(board):

print(' | |')

print(' ' + board[1] + ' | ' + board[2] + ' | ' + board[3])

print(' | |')

print('-----------')

print(' | |')

print(' ' + board[4] + ' | ' + board[5] + ' | ' + board[6])

print(' | |')

print('-----------')

print(' | |')

print(' ' + board[7] + ' | ' + board[8] + ' | ' + board[9])

print(' | |')

def isWinner(bo, le):

return (bo[7] == le and bo[8] == le and bo[9] == le) or (bo[4] == le and bo[5] == le and bo[6] == le) or (

bo[1] == le and bo[2] == le and bo[3] == le) or (bo[1] == le and bo[4] == le and bo[7] == le) or (

bo[2] == le and bo[5] == le and bo[8] == le) or (

bo[3] == le and bo[6] == le and bo[9] == le) or (

bo[1] == le and bo[5] == le and bo[9] == le) or (bo[3] == le and bo[5] == le and bo[7] == le)

def playerMove():

run = True

while run:

move = input('Please select a position to place an \'X\' (1-9): ')

try:

move = int(move)

if move > 0 and move < 10:

if spaceIsFree(move):

run = False

insertLetter('X', move)

else:

print('Sorry, this space is occupied!')

else:

print('Please type a number within the range!')

except:

print('Please type a number!')

def compMove():

possibleMoves = [x for x, letter in enumerate(board) if letter == ' ' and x != 0]

move = 0

for let in ['O', 'X']:

for i in possibleMoves:

boardCopy = board[:]

boardCopy[i] = let

if isWinner(boardCopy, let):

move = i

return move

cornersOpen = []

for i in possibleMoves:

if i in [1, 3, 7, 9]:

cornersOpen.append(i)

if len(cornersOpen) > 0:

move = selectRandom(cornersOpen)

return move

if 5 in possibleMoves:

move = 5

return move

edgesOpen = []

for i in possibleMoves:

if i in [2, 4, 6, 8]:

edgesOpen.append(i)

if len(edgesOpen) > 0:

move = selectRandom(edgesOpen)

return move

def selectRandom(li):

import random

ln = len(li)

r = random.randrange(0, ln)

return li[r]

def isBoardFull(board):

if board.count(' ') > 1:

return False

else:

return True

def main():

print('Welcome to Tic Tac Toe!')

printBoard(board)

while not (isBoardFull(board)):

if not (isWinner(board, 'O')):

playerMove()

printBoard(board)

else:

print('Sorry, O\'s won this time!')

break

if not (isWinner(board, 'X')):

move = compMove()

if move == 0:

print('Tie Game!')

else:

insertLetter('O', move)

print('Computer placed an \'O\' in position', move, ':')

printBoard(board)

else:

print('X\'s won this time! Good Job!')

break

if isBoardFull(board):

print('Tie Game!')

while True:

answer = input('Do you want to play again? (Y/N)')

if answer.lower() == 'y' or answer.lower == 'yes':

board = [' ' for x in range(10)]

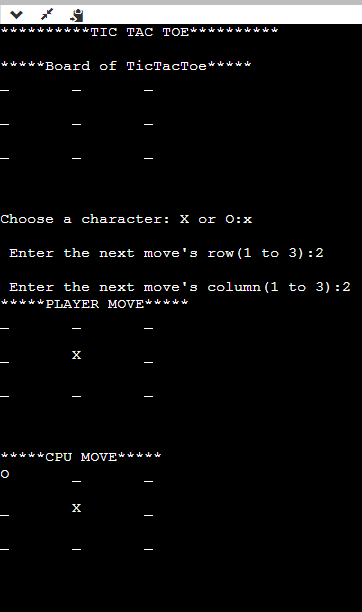
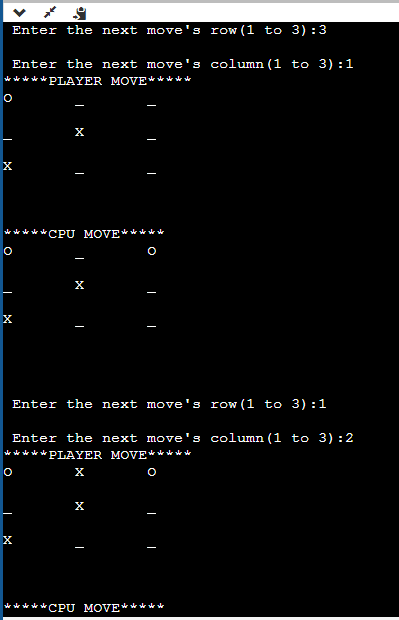
print('-----------------------------------')

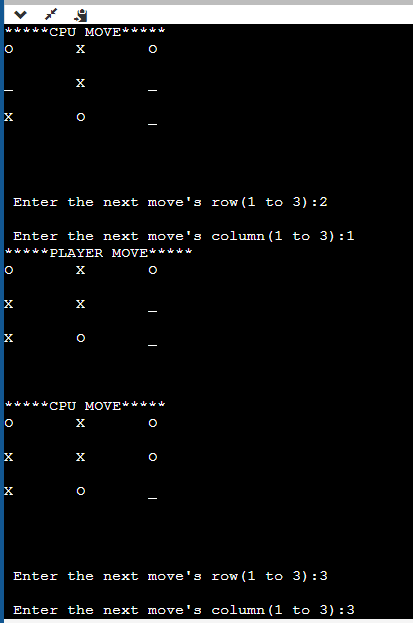
main()

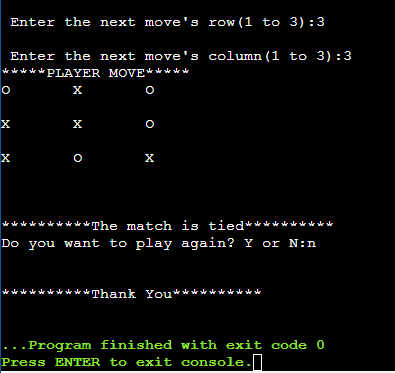
else:

break

**Output:**





**Program 2:**

**Solve 8 puzzle problem.**

**Code:**

def main():

goal=[1,2,3,4,5,6,7,8,-1]

start=[1,2,3,4,-1,6,7,5,8]

vis=[]

dfs(start,goal,vis)

print("GOAL NOT REACHABLE")

def dfs(cur,goal,vis):

if(len(vis)==10):exit()

if(cur==goal):

display(cur)

print("\nGOAL REACHED!!")

exit()

vis.append(cur)

display(cur)

next\_states=gen\_state(cur)

for state in next\_states:

if(not state in vis):

dfs(state,goal,vis)

def display(cur):

for i in range (9):

if(i%3==0):

print("")

print(cur[i],end=" ")

def gen\_state(cur):

ind=find\_space(cur)

moves=[]

if ind < 6:

moves.append('d')

if(ind % 3!=2):

moves.append('r')

if ind > 2:

moves.append('u')

if ind % 3 !=0:

moves.append('l')

next\_states=[]

for move in moves:

temp=create\_state(cur,move,ind)

next\_states.append(temp)

return next\_states

def create\_state(cur,move,ind):

c=cur[:]

if(move=='u'):

c[ind],c[ind-3]=c[ind-3],c[ind]

if(move=='d'):

c[ind],c[ind+3]=c[ind+3],c[ind]

if(move=='r'):

c[ind],c[ind+1]=c[ind+1],c[ind]

if(move=='l'):

c[ind],c[ind-1]=c[ind-1],c[ind]

return c

def find\_space(cur):

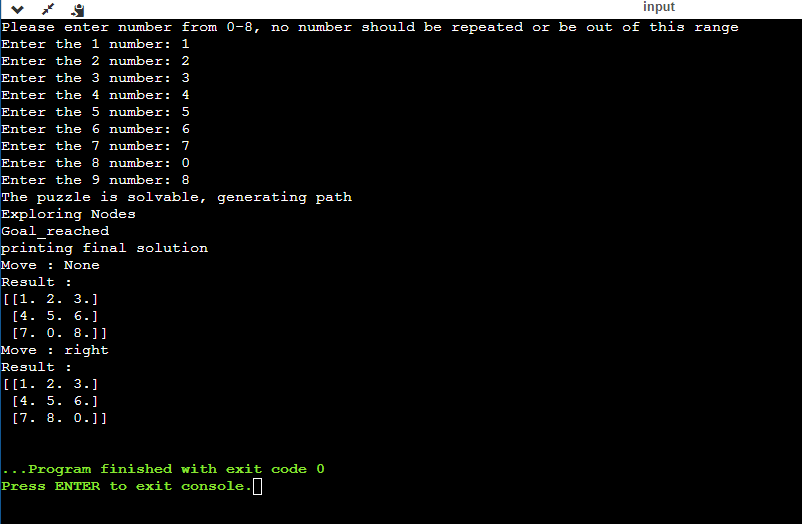
for i in range(9):

if(cur[i]==-1):return i

return -1

main()

**Output:**



**Program 3:**

**Implement Iterative Deepening Search Algorithm.**

**Code:**

from collections import defaultdict

class Graph:

def \_\_init\_\_(self, vertices):

self.V = vertices

self.graph = defaultdict(list)

def addEdge(self, u, v):

self.graph[u].append(v)

def DLS(self, src, target, maxDepth):

if src == target: return True

if maxDepth <= 0: return False

for i in self.graph[src]:

if (self.DLS(i, target, maxDepth - 1)):

return True

return False

def IDDFS(self, src, target, maxDepth):

for i in range(maxDepth):

if (self.DLS(src, target, i)):

return True

return False

g = Graph(7)

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 3)

g.addEdge(1, 4)

g.addEdge(2, 5)

g.addEdge(2, 6)

target = 6;

maxDepth = 3;

src = 0

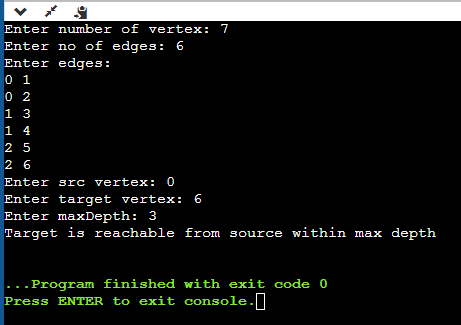
if g.IDDFS(src, target, maxDepth) == True:

print("Target is reachable from source " + "within max depth")

else:

print("Target is NOT reachable from source " + "within max depth")

**Output:**

****

**Program 4:**

**Implement A\* Search Algorithm.**

**Code:**

def aStarAlgo(start\_node, stop\_node):

open\_set = set(start\_node)

closed\_set = set()

g = {}

parents = {}

g[start\_node] = 0

parents[start\_node] = start\_node

while len(open\_set) > 0:

n = None

for v in open\_set:

if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):

n = v

if n == stop\_node or Graph\_nodes[n] == None:

pass

else:

for (m, weight) in get\_neighbors(n):

if m not in open\_set and m not in closed\_set:

open\_set.add(m)

parents[m] = n

g[m] = g[n] + weight

else:

if g[m] > g[n] + weight:

g[m] = g[n] + weight

parents[m] = n

if m in closed\_set:

closed\_set.remove(m)

open\_set.add(m)

if n == None:

print('Path does not exist!')

return None

if n == stop\_node:

path = []

while parents[n] != n:

path.append(n)

n = parents[n]

path.append(start\_node)

path.reverse()

print('Path found: {}'.format(path))

return path

open\_set.remove(n)

closed\_set.add(n)

print('Path does not exist!')

return None

def get\_neighbors(v):

if v in Graph\_nodes:

return Graph\_nodes[v]

else:

return None

def heuristic(n):

H\_dist = {

'A': 11,

'B': 6,

'C': 99,

'D': 1,

'E': 7,

'G': 0,

}

return H\_dist[n]

Graph\_nodes = {

'A': [('B', 2), ('E', 3)],

'B': [('C', 1), ('G', 9)],

'C': None,

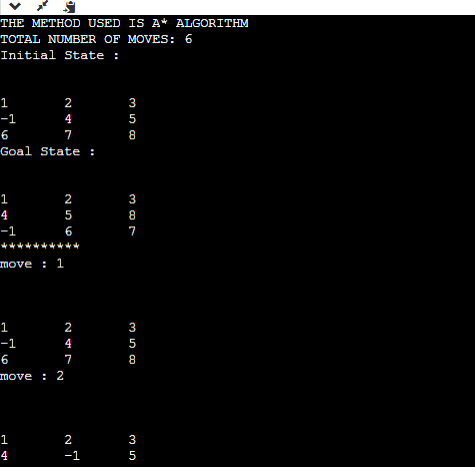
'E': [('D', 6)],

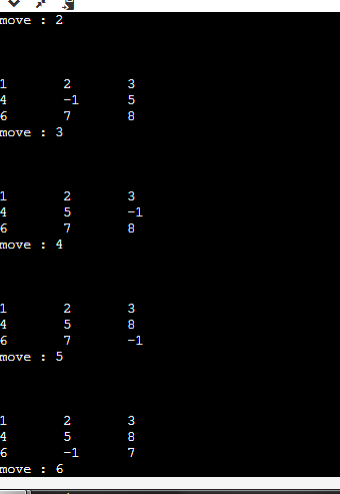
'D': [('G', 1)],

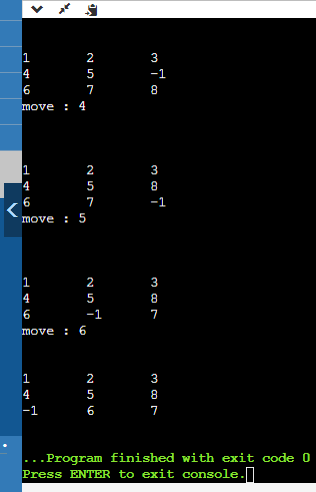
}

aStarAlgo('A', 'G')

**Output:**







**Program 5:**

**Implement Vacuum Cleaner Agent.**

**Code:**

def agent():

dic = dict()

nol = int(input("Enter the number of locations\n"))

print("Enter the status of the locations: 0 for clean and 1 for dirty")

for i in range(nol):

s = int(input())

dic[i + 1] = s

l = 1

while (1):

if (dic[l] == 1):

print("Action: Suck the dirt")

dic[l] = 0

else:

print("Action: No operation")

if (l != nol):

print("Action: Move to the next location")

l = l + 1

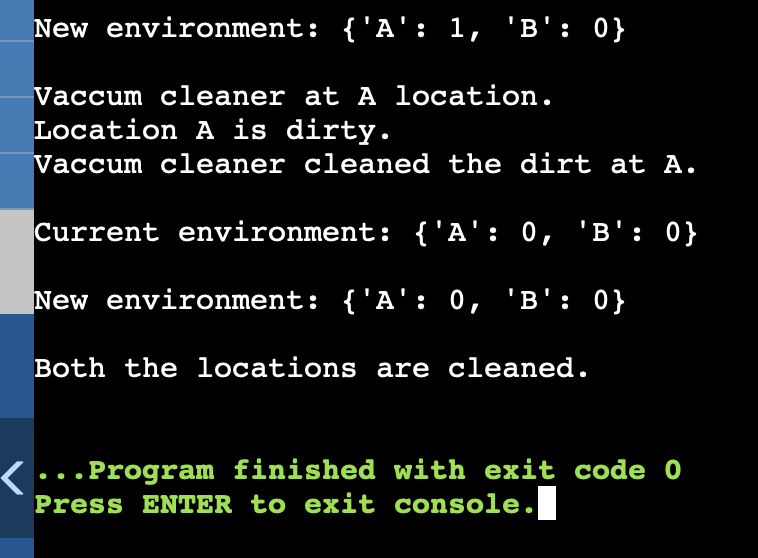
if (l > nol):

print("Goal reached. All the locations are clean")

break

agent()

**Output:**



**Program 6:**

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

**Code:**

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

variable={'p':0,'q':1, 'r':2}

kb=''

q=''

priority={'~':3,'v':1,'^':2}

def input\_rules():

global kb, q

kb = (input("Enter rule: "))

q = input("Enter the 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

def isOperand(c):

return c.isalpha() and c!='v'

def isLeftParanthesis(c):

return c == '('

def isRightParanthesis(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 isLeftParanthesis(c):

stack.append(c)

elif isRightParanthesis(c):

operator = stack.pop()

while not isLeftParanthesis(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 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 \_eval(i, val1, val2):

if i == '^':

return val2 and val1

return val2 or val1

#Test 1

input\_rules()

ans = entailment()

if ans:

print("Knowledge Base entails query")

else:

print("Knowledge Base does not entail query")

#Test 2

input\_rules()

ans = entailment()

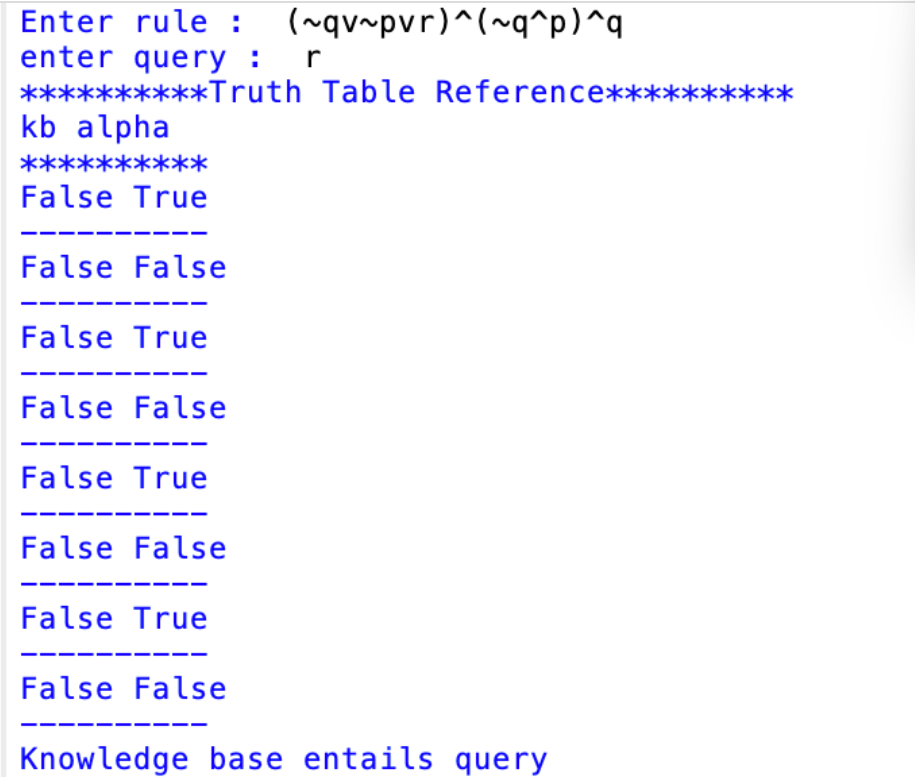
if ans:

print("Knowledge Base entails query")

else:

print("Knowledge Base does not entail query")

**Output:**



**Program 7:**

**Create a knowledge base using prepositional logic and prove the given query using resolution.**

**Code:**

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()

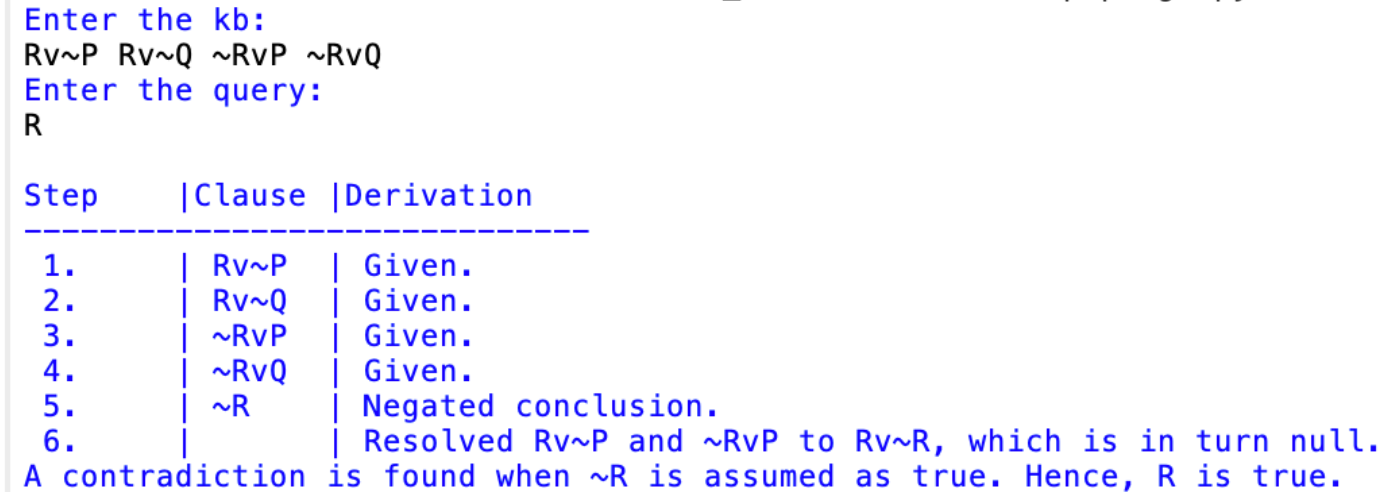
print("Enter the query:")

query = input()

resolution(kb,query)

main()

**Output:**



**Program 8:**

**Implement unification in first order logic.**

**Code:**

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")

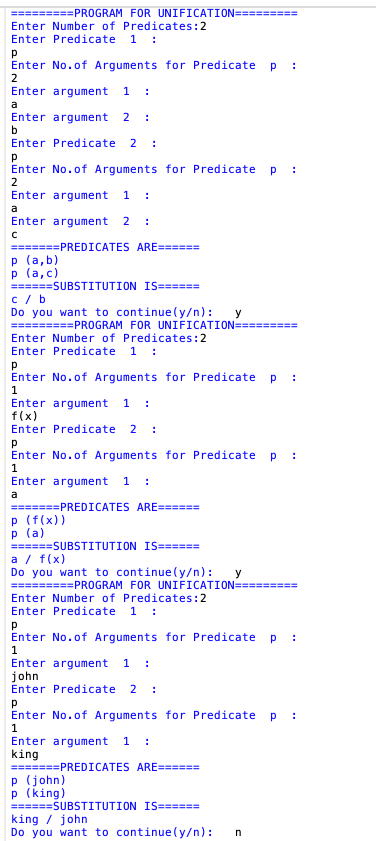
e2 = input()

substitutions = unify(e1, e2)

print("The substitutions are:")

print([' / '.join(substitution) for substitution in substitutions])

**Output:**



**Program 9:**

**Convert given first order logic statement into conjunctive normal form (CNF).**

**Code:**

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():

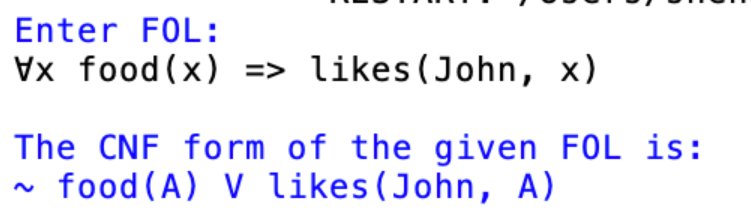
print("Enter FOL:")

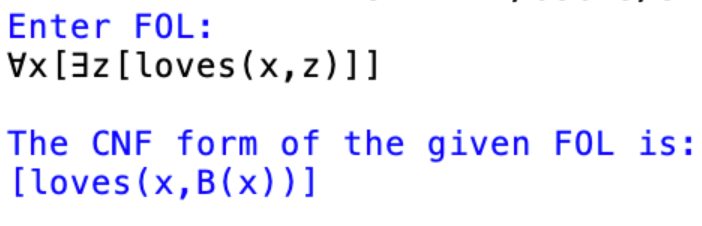
fol = input()

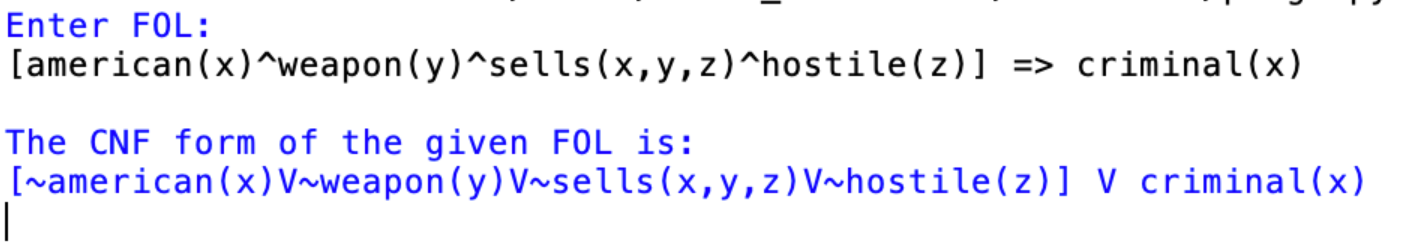
print("The CNF form of the given FOL is: ")

print(Skolemization(fol\_to\_cnf(fol)))

**Output:**







**Program 10:**

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

**Code:**

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()

**Output:**

