Name: Sakshi P Khandoba

USN: 1BM19CS139

AI LAB REPORT

Program 1:

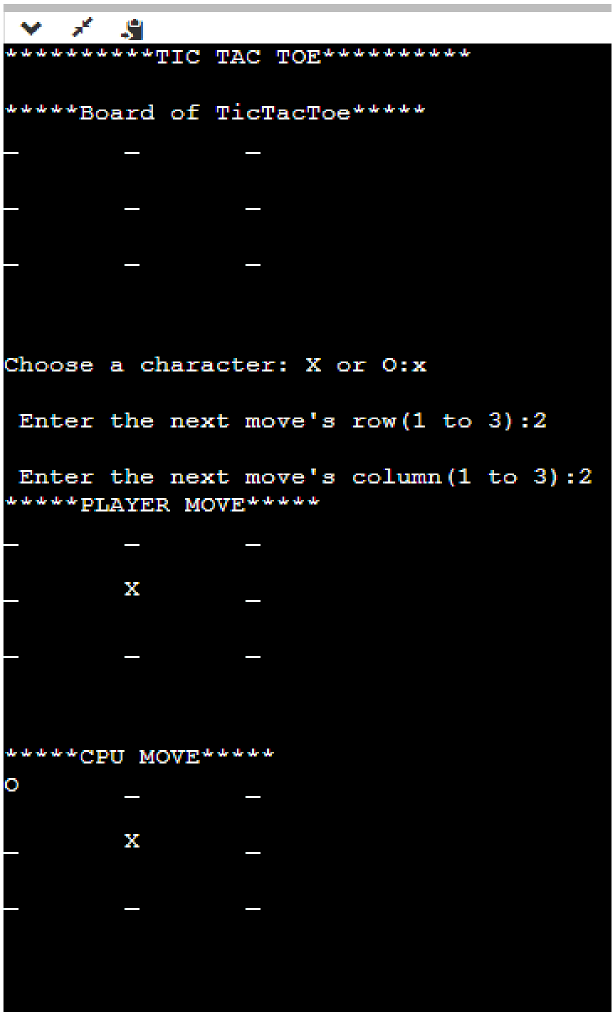
Implement Tic –Tac –Toe Game.

Code:

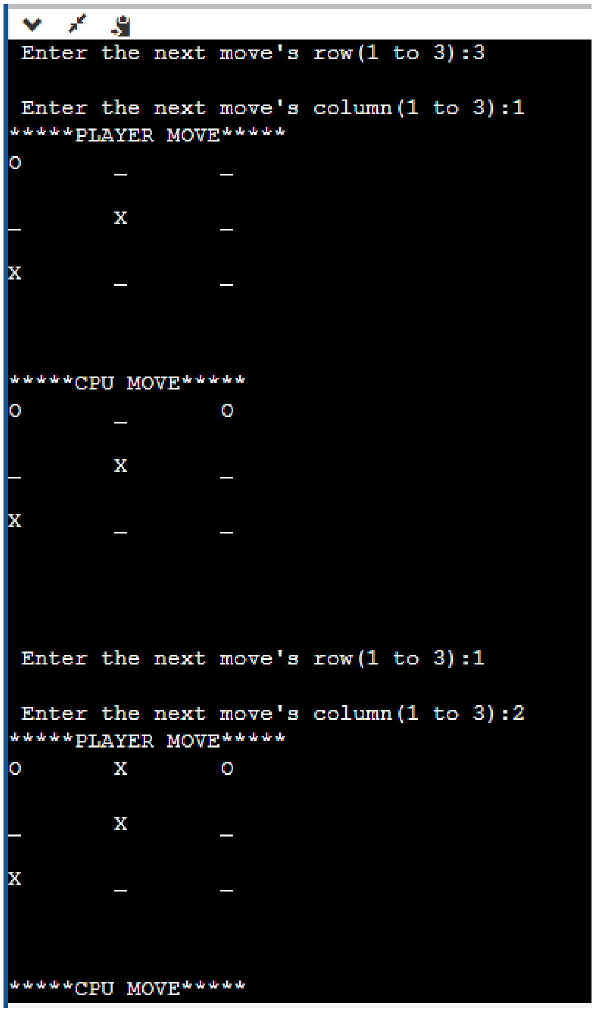
```
board = [' ' for x in range(10)]
def insertLetter(letter, pos):
   board[pos] = letter
def spaceIsFree(pos):
   return board[pos] == ' '
def printBoard(board):
   print('   |   |')
   print(' ' + board[1] + ' | ' + board[2] + ' | ' + board[3])
   print('   |   |')
   print('-----------')
   print('   |   |')
   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
```

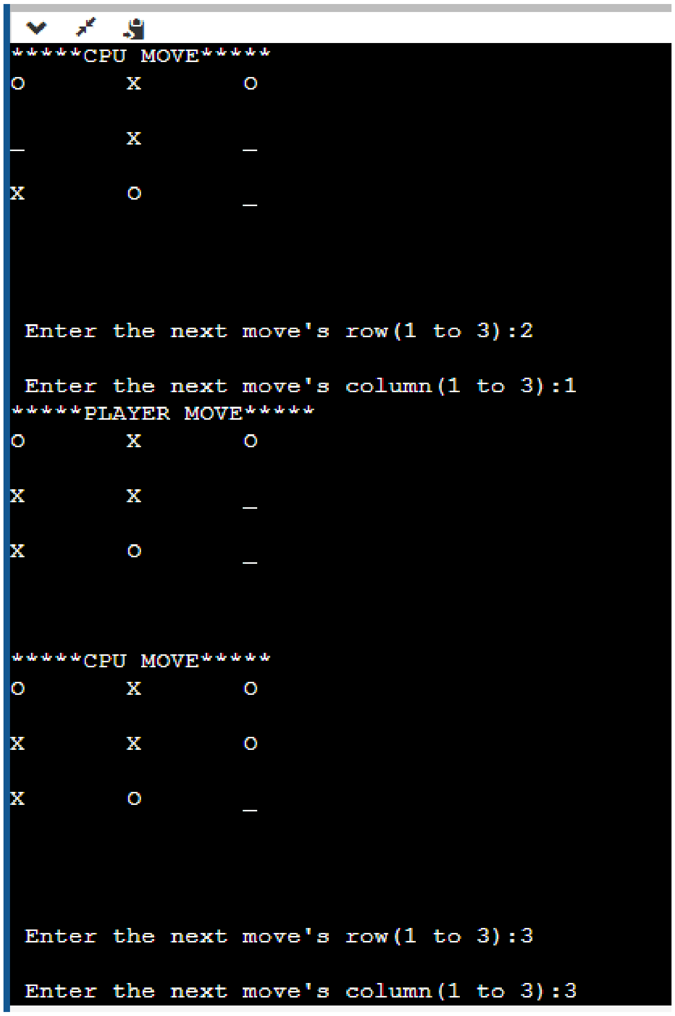
Output:



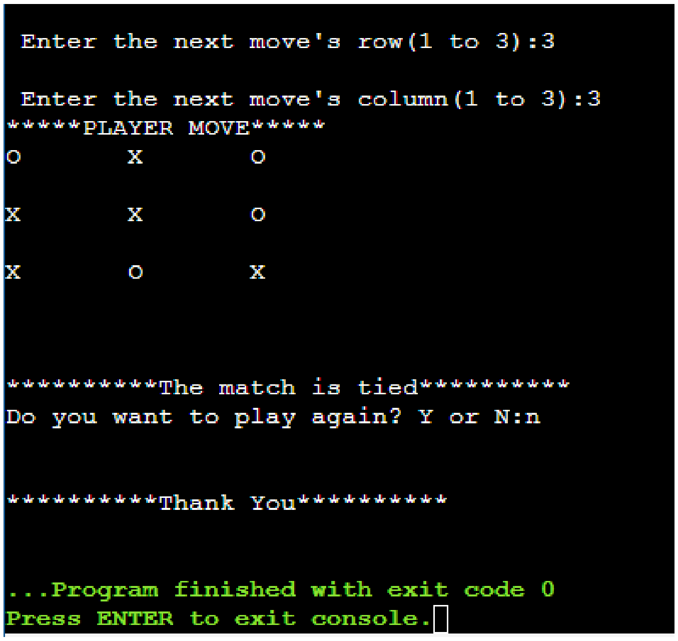
```
**********TIC TAC TOE**********

*****Board of TicTacToe*****
_       _       _

_       _       _

_       _       _



Choose a character: X or O:x

 Enter the next move's row(1 to 3):2

 Enter the next move's column(1 to 3):2
*****PLAYER MOVE*****
_       _       _

_       X       _

_       _       _



*****CPU MOVE*****
O       _       _

_       X       _

_       _       _
```



```
 Enter the next move's row(1 to 3):3

 Enter the next move's column(1 to 3):1
*****PLAYER MOVE*****
O       _       _

_       X       _

X       _       _



*****CPU MOVE*****
O       _       O

_       X       _

X       _       _




 Enter the next move's row(1 to 3):1

 Enter the next move's column(1 to 3):2
*****PLAYER MOVE*****
O       X       O

_       X       _

X       _       _



*****CPU MOVE*****
```



```
*****CPU MOVE*****
O       X       O

_       X       _

X       O       _




 Enter the next move's row(1 to 3):2

 Enter the next move's column(1 to 3):1
*****PLAYER MOVE*****
O       X       O

X       X       _

X       O       _



*****CPU MOVE*****
O       X       O

X       X       O

X       O       _




 Enter the next move's row(1 to 3):3

 Enter the next move's column(1 to 3):3
```



```
 Enter the next move's row(1 to 3):3

 Enter the next move's column(1 to 3):3
*****PLAYER MOVE*****
O       X       O

X       X       O

X       O       X



**********The match is tied**********
Do you want to play again? Y or N:n


**********Thank You**********


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

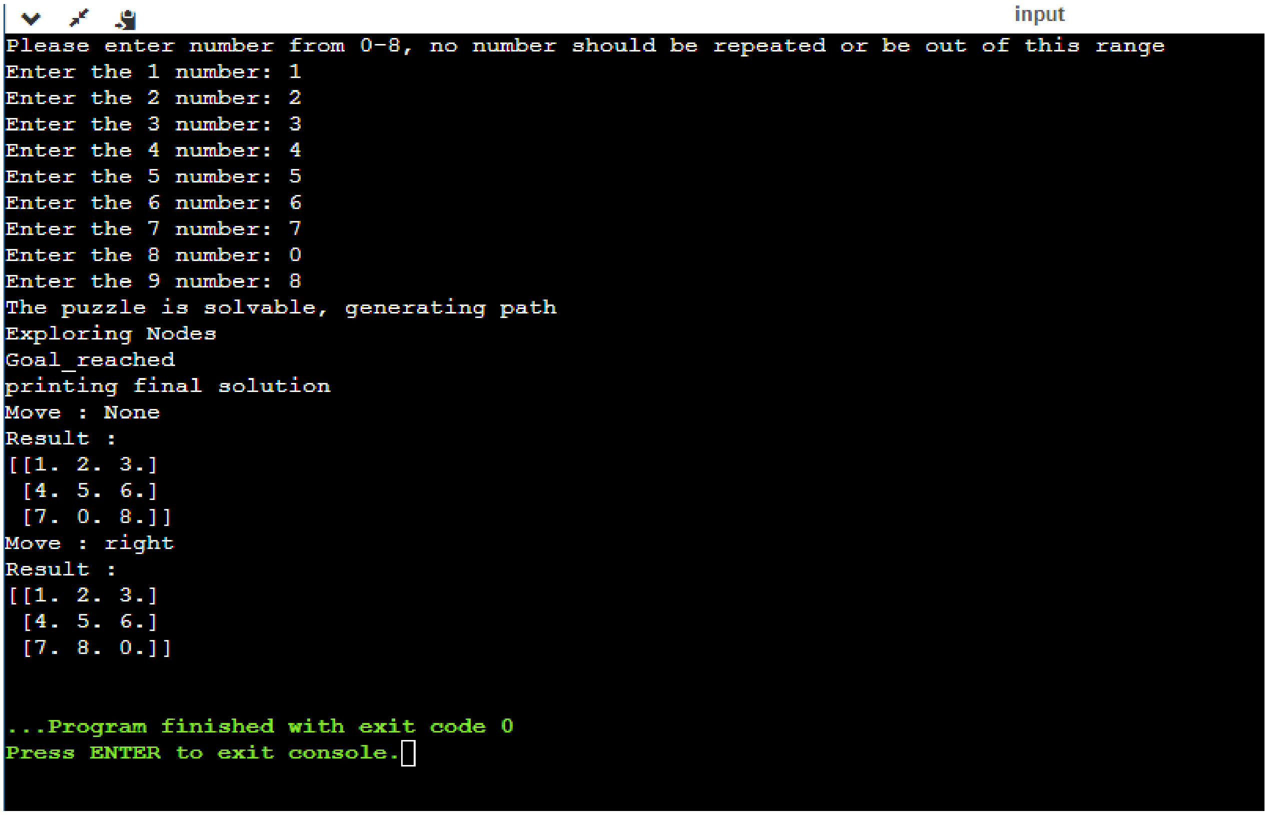
Program 2:

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:



```
Please enter number from 0-8, no number should be repeated or be out of this range
Enter the 1 number: 1
Enter the 2 number: 2
Enter the 3 number: 3
Enter the 4 number: 4
Enter the 5 number: 5
Enter the 6 number: 6
Enter the 7 number: 7
Enter the 8 number: 0
Enter the 9 number: 8
The puzzle is solvable, generating path
Exploring Nodes
Goal_reached
printing final solution
Move : None
Result :
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 0. 8.]]
Move : right
Result :
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 0.]]


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

Program 3:

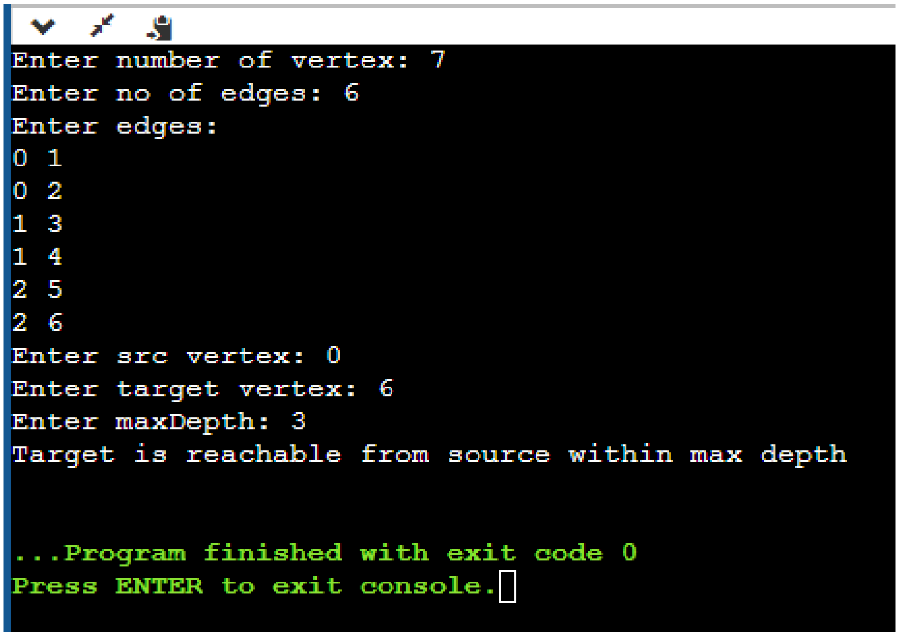
Implement Iterative Deepening Search Algorithm.

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:



```
Enter number of vertex: 7
Enter no of edges: 6
Enter edges:
0 1
0 2
1 3
1 4
2 5
2 6
Enter src vertex: 0
Enter target vertex: 6
Enter maxDepth: 3
Target is reachable from source within max depth


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

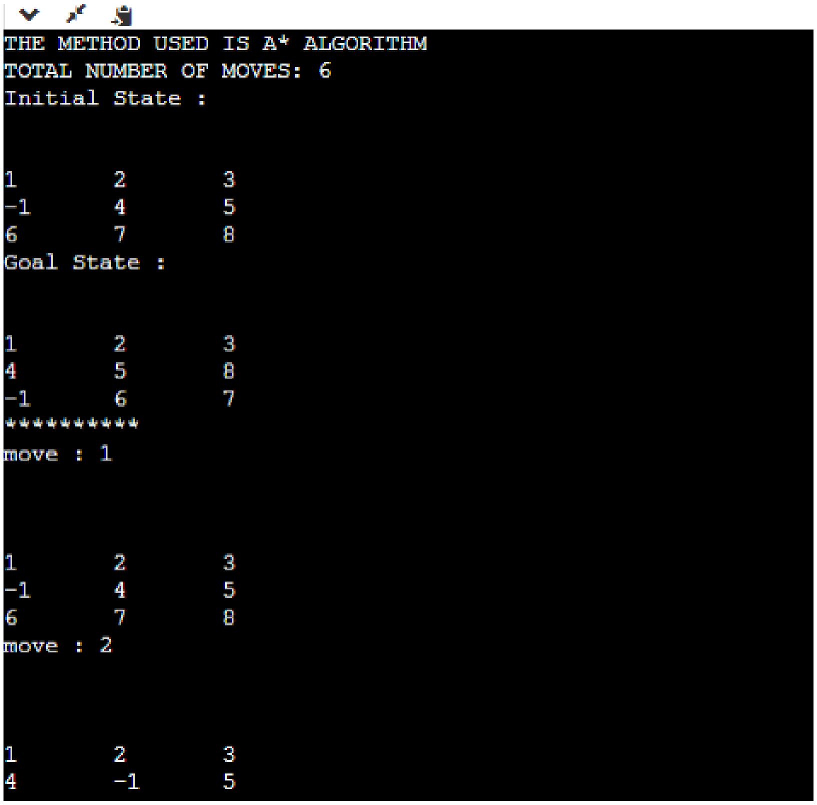
Program 4:

Implement A* Search Algorithm.

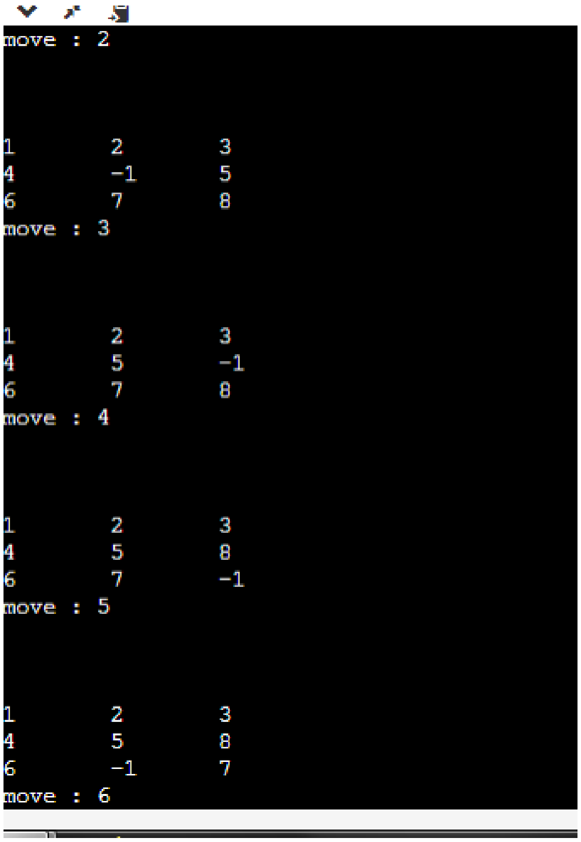
Code:

```
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')
```

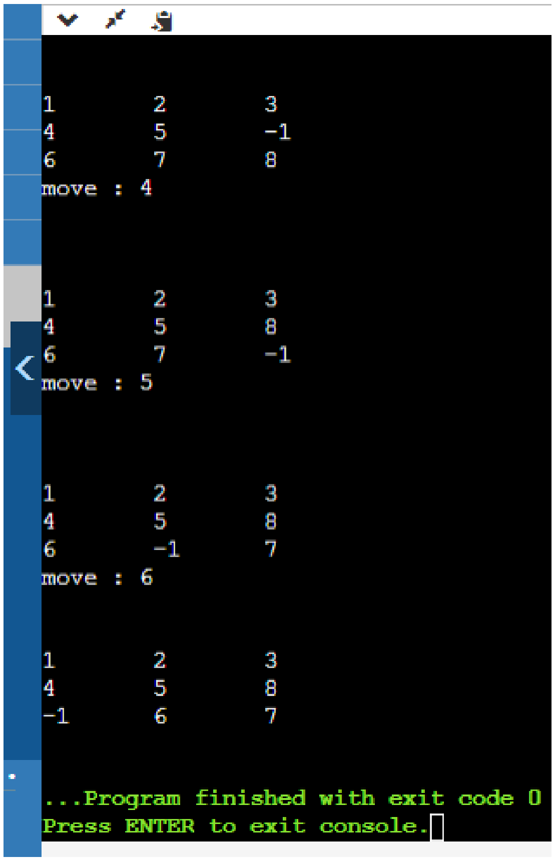
Output:



```
THE METHOD USED IS A* ALGORITHM
TOTAL NUMBER OF MOVES: 6
Initial State :


1       2       3
-1      4       5
6       7       8
Goal State :


1       2       3
4       5       8
-1      6       7
**********
move : 1



1       2       3
-1      4       5
6       7       8
move : 2



1       2       3
4       -1      5
```



```
move : 2



1       2       3
4       -1      5
6       7       8
move : 3



1       2       3
4       5       -1
6       7       8
move : 4



1       2       3
4       5       8
6       7       -1
move : 5



1       2       3
4       5       8
6       -1      7
move : 6
```



```
1       2       3
4       5       -1
6       7       8
move : 4


1       2       3
4       5       8
6       7       -1
move : 5


1       2       3
4       5       8
6       -1      7
move : 6


1       2       3
4       5       8
-1      6       7


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

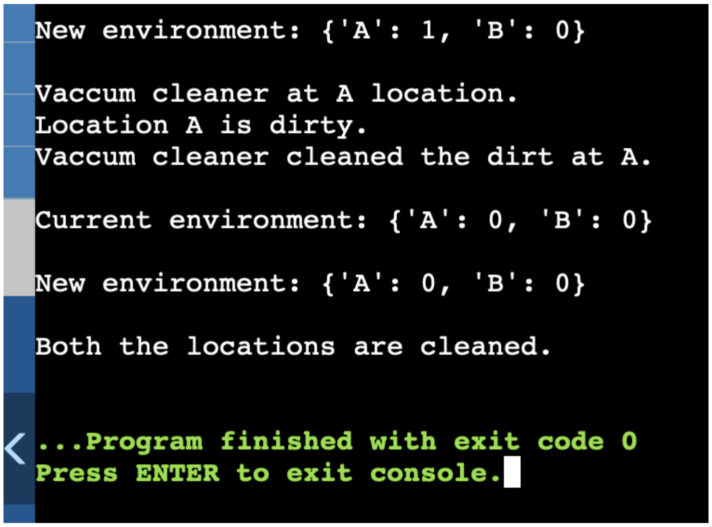
Program 5:

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:



```
New environment: {'A': 1, 'B': 0}

Vaccum cleaner at A location.
Location A is dirty.
Vaccum cleaner cleaned the dirt at A.

Current environment: {'A': 0, 'B': 0}

New environment: {'A': 0, 'B': 0}

Both the locations are cleaned.


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

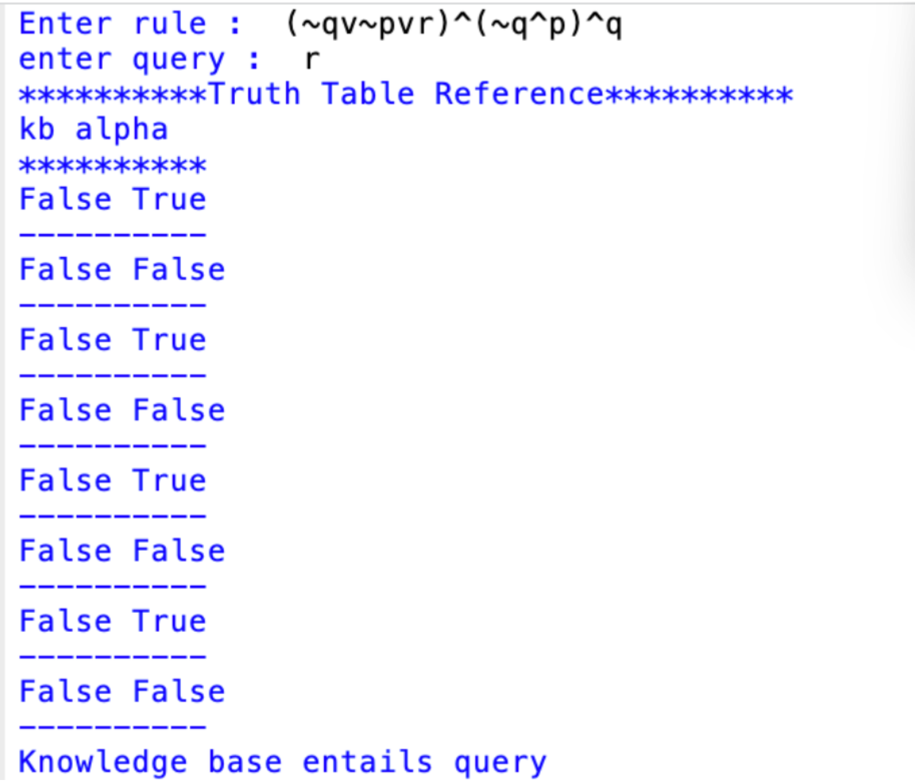
Program 6:

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:



```
Enter rule :  (~qv~pvr)^(~q^p)^q
enter query :  r
**********Truth Table Reference**********
kb alpha
**********
False True
----------
False False
----------
False True
----------
False False
----------
False True
----------
False False
----------
False True
----------
False False
----------
Knowledge base entails query
```

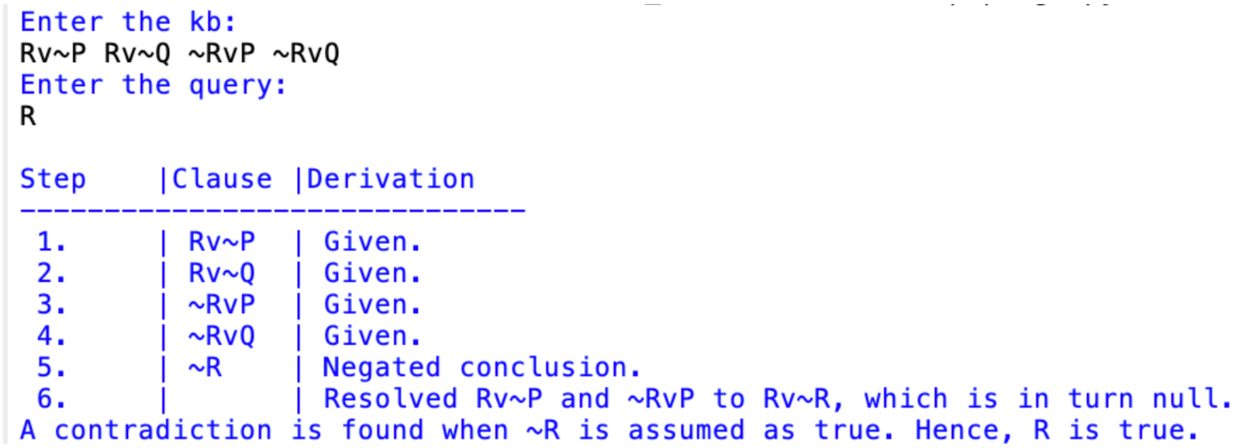
Program 7:

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:



```
Enter the kb:
Rv~P Rv~Q ~RvP ~RvQ
Enter the query:
R

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

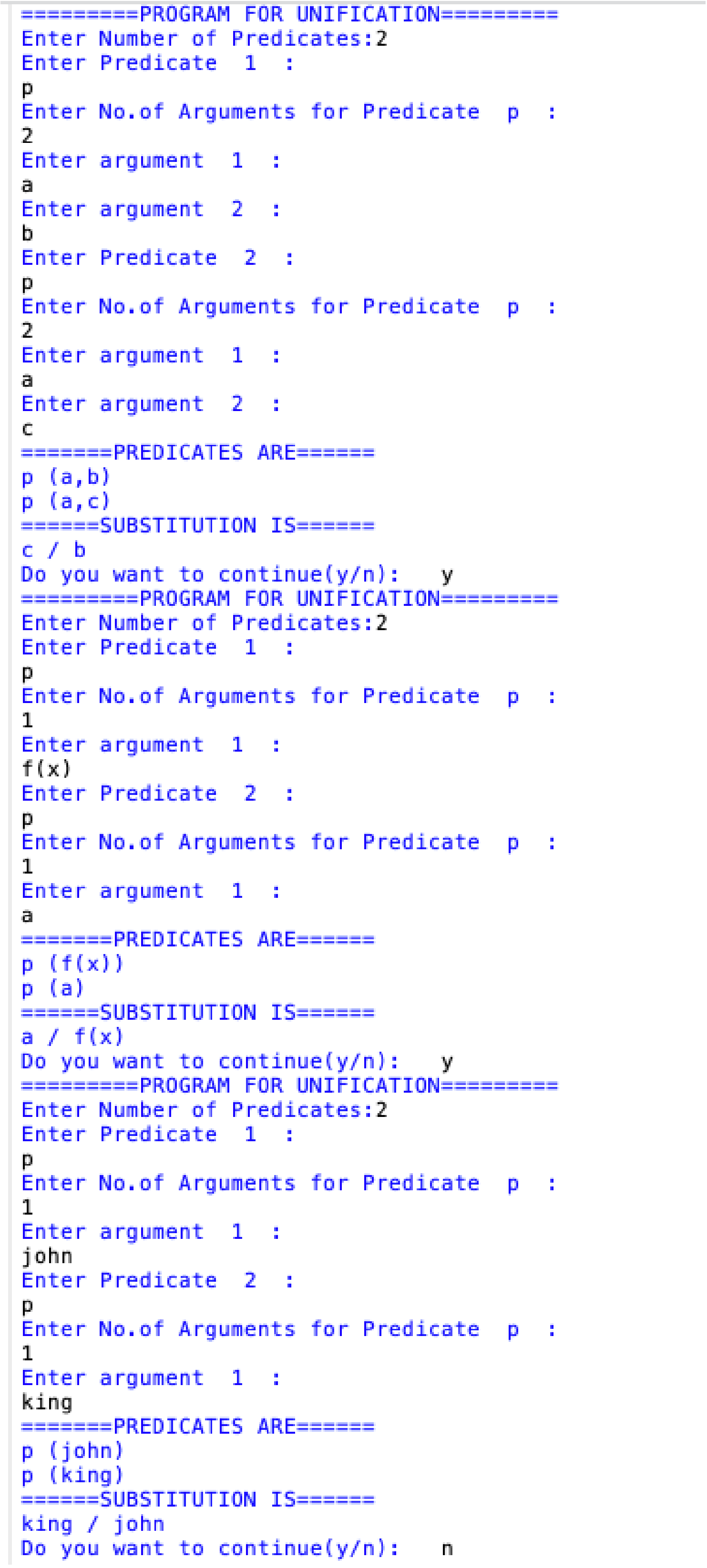
Program 8:

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 FOR UNIFICATION=========
Enter Number of Predicates:2
Enter Predicate  1  :
p
Enter No.of Arguments for Predicate  p  :
2
Enter argument  1  :
a
Enter argument  2  :
b
Enter Predicate  2  :
p
Enter No.of Arguments for Predicate  p  :
2
Enter argument  1  :
a
Enter argument  2  :
c
=======PREDICATES ARE======
p (a,b)
p (a,c)
======SUBSTITUTION IS======
c / b
Do you want to continue(y/n):   y
=========PROGRAM FOR UNIFICATION=========
Enter Number of Predicates:2
Enter Predicate  1  :
p
Enter No.of Arguments for Predicate  p  :
1
Enter argument  1  :
f(x)
Enter Predicate  2  :
p
Enter No.of Arguments for Predicate  p  :
1
Enter argument  1  :
a
=======PREDICATES ARE======
p (f(x))
p (a)
======SUBSTITUTION IS======
a / f(x)
Do you want to continue(y/n):   y
=========PROGRAM FOR UNIFICATION=========
Enter Number of Predicates:2
Enter Predicate  1  :
p
Enter No.of Arguments for Predicate  p  :
1
Enter argument  1  :
john
Enter Predicate  2  :
p
Enter No.of Arguments for Predicate  p  :
1
Enter argument  1  :
king
=======PREDICATES ARE======
p (john)
p (king)
======SUBSTITUTION IS======
king / john
Do you want to continue(y/n):   n
```

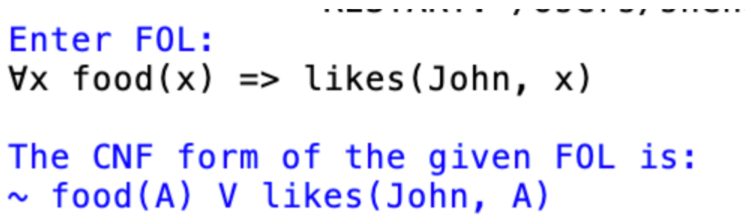
Program 9:

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

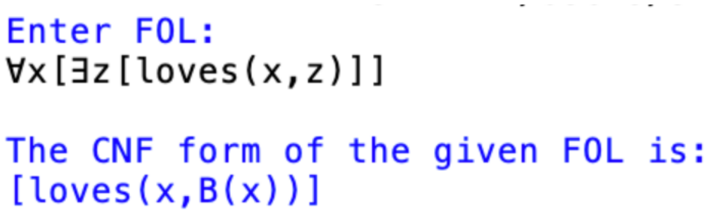
Code:

```
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)))
```

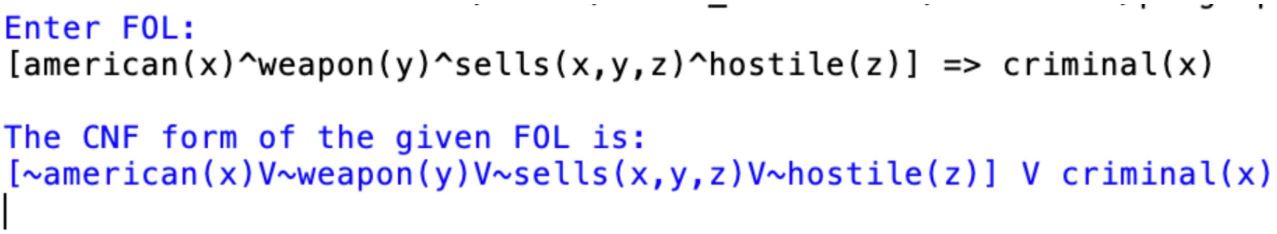
Output:



```
Enter FOL:
∀x food(x) => likes(John, x)

The CNF form of the given FOL is:
~ food(A) V likes(John, A)
```



```
Enter FOL:
∀x[∃z[loves(x,z)]]

The CNF form of the given FOL is:
[loves(x,B(x))]
```



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

The CNF form of the given FOL is:
[~american(x)V~weapon(y)V~sells(x,y,z)V~hostile(z)] V criminal(x)
|
```

Program 10:

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

Code:

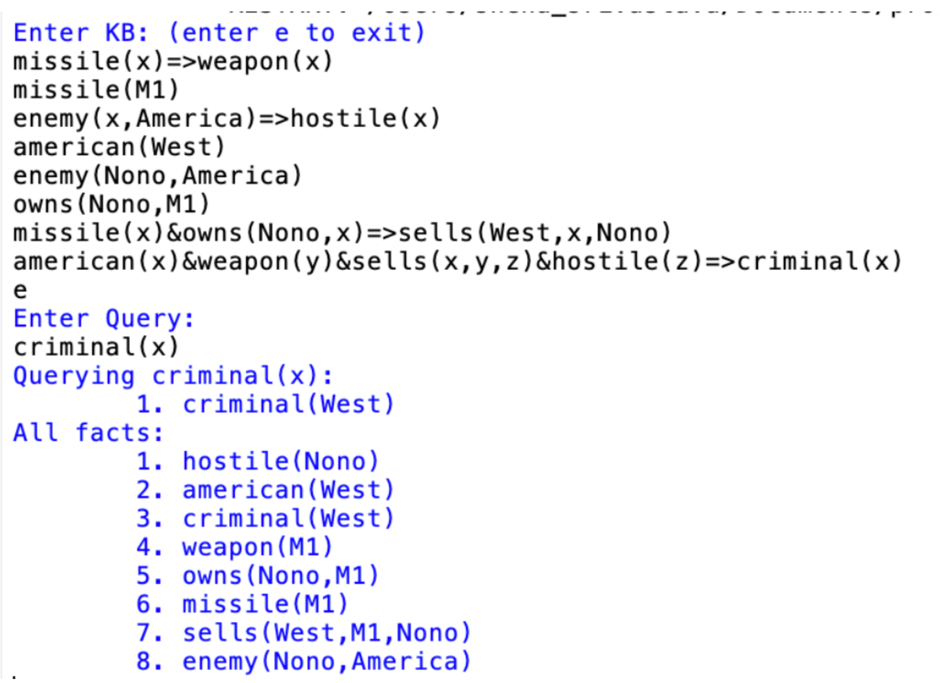
```
import re
def isVariable(x):
   return len(x) == 1 and x.islower() and x.isalpha()
def getAttributes(string):
   expr = '\([^)]+\)'
   matches = re.findall(expr, string)
```

```
    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:



```
Enter KB: (enter e to exit)
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
Enter Query:
criminal(x)
Querying criminal(x):
        1. criminal(West)
All facts:
        1. hostile(Nono)
        2. american(West)
        3. criminal(West)
        4. weapon(M1)
        5. owns(Nono,M1)
        6. missile(M1)
        7. sells(West,M1,Nono)
        8. enemy(Nono,America)
```