AI-LAB RECORD (1BM19CS159)

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[1]==le and bo[2]==le and bo[3]==le) or                    #left to right
   (bo[4]==le and bo[5]==le and bo[6]==le) or
```

```
    (bo[7]==le and bo[8]==le and bo[9]==le) or
    (bo[1]==le and bo[4]==le and bo[7]==le) or                  #top to bottom
    (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                  #diagonally
    (bo[3]==le and bo[5]==le and bo[7]==le))

def playerMove():
    run=True
    while run:
        move=input("Where would you like to place 'X'(1-9): ")
        try:
            move=int(move)
            if move>0 and move<10:
                if spaceIsFree(move):
                    run=False
                    insertLetter('X',move)
                else:
                    print("Sorry, space is occupied")
            else:
                print("Please insert 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']:                          #checking if the comp can win in the next
move or if the opponent can win and hence block it
```

```
    for i in possibleMoves:
        boardCopy=board[:]                              #creating a copy for the board
        boardCopy[i]=let
        if isWinner(boardCopy,let):
            move=i
            return move

cornersOpen=[]
for i in possibleMoves:                                 #checking if corners are free
    if i in [1,3,7,9]:
        cornersOpen.append(i)
if len(cornersOpen)>0:                                  #choosing one of the corners
    move=selectRandom(cornersOpen)
    return move

if 5 in possibleMoves:                                  #checking and choosing the middle
    move=5
    return move

edgesOpen=[]
for i in possibleMoves:                                 #checking if corners are free
    if i in [2,4,6,8]:
        edgesOpen.append(i)
if len(edgesOpen)>0:                                    #choosing one of the corners
    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('Welcom to Tic Tac Toe')
    printBoard(board)
    while not(isBoardFull(board)):
        if not(isWinner(board,'O')):
            playerMove()
            printBoard(board)
        else:
            print('Sorry computer won the game')
            break

        if not(isWinner(board,'X')):
            move = compMove()
            if move==0:
                print("Game is tied!")
            else:
                insertLetter('O',move)
```

```
            print("Computer placed in position ",move)
            printBoard(board)
        else:
            print('Congrats you won')
            break



    if isBoardFull(board):
        print('Game is tied')

main()
```

<u>Output: -</u>

```
Welcom to Tic Tac Toe
   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
Where would you like to place 'X'(1-9):
5
   |   |
```

```
   |   |
   |   |
-----------
   |   |
   | X |
   |   |
-----------
   |   |
   |   |
   |   |
Computer placed in position  3
   |   |
   |   | O
   |   |
-----------
   |   |
   | X |
   |   |
-----------
   |   |
   |   |
   |   |
Where would you like to place 'X'(1-9):
1
   |   |
 X |   | O
   |   |
-----------
   |   |
```

```
   | X |
   |   |
-----------
   |   |
   |   |
   |   |
Computer placed in position  9
   |   |
 X |   | O
   |   |
-----------
   |   |
   | X |
   |   |
-----------
   |   |
   |   | O
   |   |
Where would you like to place 'X'(1-9):
6
   |   |
 X |   | O
   |   |
-----------
   |   |
   | X | X
   |   |
-----------
   |   |
```

```
   |   | O
   |   |
Computer placed in position  4
   |   |
 X |   | O
   |   |
-----------
   |   |
 O | X | X
   |   |
-----------
   |   |
   |   | O
   |   |
Where would you like to place 'X'(1-9):
2
   |   |
 X | X | O
   |   |
-----------
   |   |
 O | X | X
   |   |
-----------
   |   |
   |   | O
   |   |
Computer placed in position  8
   |   |
```

```
 X | X | O
   |   |
-----------
   |   |
 O | X | X
   |   |
-----------
   |   |
   | O | O
   |   |
Where would you like to place 'X'(1-9):
1
Sorry, space is occupied
Where would you like to place 'X'(1-9):
7
   |   |
 X | X | O
   |   |
-----------
   |   |
 O | X | X
   |   |
-----------
   |   |
 X | O | O
   |   |
Game is tied!
Game is tied
```

** Process exited - Return Code: 0 **

Press Enter to exit terminal

Program 2: - Solve 8 puzzle problem.

Code: -

```
# from copy import copy;
from copy import copy


def bfs(src,target):
    if src==target:
        print("Target reached in 0 steps\n")
        return
    queue = []
    exp=[]
    val=0
    queue.append(src)
    while len(queue)>0:
        next=queue.pop(0)
        exp.append(next)
        print(next)
        if next==target:
            print("Target reached in "+str(val)+" steps")
            return
        possible=[]
        possible=gameRule(exp,next)
        val+=1
        for state in possible:
            if state not in queue and state not in exp:
```

```
            queue.append(state)
    print("State not possible. States searched = "+str(val))
def gameRule(expr,src):
    b=src.index(-1)
    possible_moves=[]
    if b not in [0,1,2]:
        possible_moves.append('U')
    if b not in [6,7,8]:
        possible_moves.append('D')
    if b not in [0,3,6]:
        possible_moves.append('L')
    if b not in [2,5,8]:
        possible_moves.append('R')

    states=[]
    for i in possible_moves:
        state=genState(src,i,b)
        states.append(state)
    return [state for state in states if state not in expr]


def genState(src,move,b):
    temp = src.copy()
    if move=='D':
        temp[b+3],temp[b]=temp[b],temp[b+3]
    if move=='U':
        temp[b-3],temp[b]=temp[b],temp[b-3]
    if move=='L':
        temp[b-1],temp[b]=temp[b],temp[b-1]
    if move=='R':
```

```
        temp[b+1],temp[b]=temp[b],temp[b+1]
    return temp
src=[1,-1,3,4,2,5,7,8,6]
target=[1,2,3,4,5,6,7,8,-1]
print("src: "+str(src))
print("target: "+str(target)+"\n")
bfs(src,target)
```

Output: -

```
src: [1, -1, 3, 4, 2, 5, 7, 8, 6]
target: [1, 2, 3, 4, 5, 6, 7, 8, -1]

[1, -1, 3, 4, 2, 5, 7, 8, 6]
[1, 2, 3, 4, -1, 5, 7, 8, 6]
[-1, 1, 3, 4, 2, 5, 7, 8, 6]
[1, 3, -1, 4, 2, 5, 7, 8, 6]
[1, 2, 3, 4, 8, 5, 7, -1, 6]
[1, 2, 3, -1, 4, 5, 7, 8, 6]
[1, 2, 3, 4, 5, -1, 7, 8, 6]
[4, 1, 3, -1, 2, 5, 7, 8, 6]
[1, 3, 5, 4, 2, -1, 7, 8, 6]
[1, 2, 3, 4, 8, 5, -1, 7, 6]
[1, 2, 3, 4, 8, 5, 7, 6, -1]
[-1, 2, 3, 1, 4, 5, 7, 8, 6]
[1, 2, 3, 7, 4, 5, -1, 8, 6]
[1, 2, -1, 4, 5, 3, 7, 8, 6]
[1, 2, 3, 4, 5, 6, 7, 8, -1]
Target reached in 14 steps
```

** Process exited - Return Code: 0 **

Press Enter to exit terminal

<u>Program 3: - Implement an Iterative deepening search algorithm.</u>

<u>Code: -</u>

```
# 8 puzzle iterative dfs method
src=[1,2,3,-1,4,5,6,7,8]
target=[1,2,3,4,5,-1,6,7,8]

def idfs(src,target,depth):
 for limit in range(0,depth+1):
  vis=[]
  if dfs(src,target,limit,vis):
   return True
 return False

def gen(state,m,b):
 temp=state[:]
 if m=='l':
  temp[b],temp[b-1]=temp[b-1],temp[b]
 if m=='r':
  temp[b],temp[b+1]=temp[b+1],temp[b]
 if m=='u':
  temp[b],temp[b-3]=temp[b-3],temp[b]
 if m=='d':
  temp[b],temp[b+3]=temp[b+3],temp[b]

 return temp
```

```
def possible_move(state,vis):
 b=state.index(-1)
 dir=[]
 move=[]
 if b<=5:
  dir.append('d')
 if b>=3:
  dir.append('u')
 if b%3>0:
  dir.append('l')
 if b%3<2:
  dir.append('r')

 for i in dir:
  temp=gen(state,i,b)
  if not temp in vis:
   move.append(temp)

 print(move)
 return move

def dfs(src,target,limit,vis):
 if src==target :
  return True
 if limit<0 :
  return False
 vis.append(src)
 move=possible_move(src,vis)
```

```
  for m in move :
    if dfs(m,target,limit-1,vis):
      return True
  return False



print(idfs(src,target,1))
```

Output: -

```
[[1, 2, 3, 6, 4, 5, -1, 7, 8], [-1, 2, 3, 1, 4, 5, 6, 7, 8], [1, 2, 3, 4, -1, 5, 6, 7, 8]]
[[1, 2, 3, 6, 4, 5, -1, 7, 8], [-1, 2, 3, 1, 4, 5, 6, 7, 8], [1, 2, 3, 4, -1, 5, 6, 7, 8]]
[[1, 2, 3, 6, 4, 5, 7, -1, 8]]
[[2, -1, 3, 1, 4, 5, 6, 7, 8]]
[[1, 2, 3, 4, 7, 5, 6, -1, 8], [1, -1, 3, 4, 2, 5, 6, 7, 8], [1, 2, 3, 4, 5, -1, 6, 7, 8]]
True



** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

Program 4: - Implement A* search algorithm.

Code: -

```
src=[1,2,3,-1,4,5,6,7,8]
target=[1,2,3,4,5,8,-1,6,7]
print("THE METHOD USED IS A* ALGORITHM")

def h(state):
    res=0
    for i in range(1,9):
```

```
        if state.index(i)!=target.index(i): res+=1
    return res


def gen(state,m,b):
    temp=state[:]
    if m=='l': temp[b],temp[b-1]=temp[b-1],temp[b]
    if m=='r':temp[b],temp[b+1]=temp[b+1],temp[b]
    if m=='u':temp[b],temp[b-3]=temp[b-3],temp[b]
    if m=='d':temp[b],temp[b+3]=temp[b+3],temp[b]
    return temp


def possible_moves(state,visited_states):
    b=state.index(-1)
    d=[]
    pos_moves=[]
    if b<=5: d.append('d')
    if b>=3 : d.append('u')
    if b%3 > 0: d.append('l')
    if b%3 < 2: d.append('r')
    for i in d:
        temp=gen(state,i,b)
        if not temp in visited_states: pos_moves.append(temp)
    return pos_moves



def search(src,target,visited_states,g):
    if src==target: return visited_states
    visited_states.append(src),
    adj=possible_moves(src,visited_states)
```

```
    scores=[]
    selected_moves=[]
    for move in adj: scores.append(h(move)+g)
    min_score=min(scores)
    for i in range(len(adj)):
        if scores[i]==min_score: selected_moves.append(adj[i])
    for move in selected_moves:
        if search(move,target,visited_states,g+1):return visited_states
    return 0

def solve(src,target):
    visited_states=[]
    res=search(src,target,visited_states,0)



    if type(res)!=type(int()):
        i=0
        for state in res:
            print('move :',i+1,end="\n")
            print()
            display(state)


            i+=1
        print('move :',i+1)
        display(target)
print("TOTAL NUMBER OF MOVES:",6)

def display(state):
    print()
```

```python
    for i in range(9):
        if i%3==0:print()
        if state[i]==-1: print(state[i],end="\t")
        else: print(state[i],end="\t")
    print(end="\n")

print('Initial State :')
display(src)
print('Goal State :')
display(target)
print('*'*10)
solve(src,target)
```

<u>Output: -</u>

```
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
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 8

-1 6 7

** Process exited - Return Code: 0 **

Press Enter to exit terminal

Program 5: - Implement vacuum cleaner agent.

Code: -

```
#INSTRUCTIONS

#Enter LOCATION A/B in captial letters

#Enter Status O/1 accordingly where 0 means CLEAN and 1 means DIRTY


def vacuum_world():

    # initializing goal_state

    # 0 indicates Clean and 1 indicates Dirty

  goal_state = {'A': '0', 'B': '0'}

  cost = 0
```

```
    location_input = input("Enter Location of Vacuum \t") #user_input of location vacuum is
placed
    status_input = input("Enter status of"+" " + location_input + "\t") #user_input if location
is dirty or clean
    status_input_complement = input("Enter status of other room \t")
    initial_state = {'A' : status_input , 'B' : status_input_complement}
    print("Initial Location Condition" + str(initial_state))

    if location_input == 'A':
        # Location A is Dirty.
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")
            # suck the dirt  and mark it as clean
            goal_state['A'] = '0'
            cost += 1                      #cost for suck
            print("Cost for CLEANING A " + str(cost))
            print("Location A has been Cleaned.")

            if status_input_complement == '1':
                # if B is Dirty
                print("Location B is Dirty.")
                print("Moving right to the Location B. ")
                cost += 1                      #cost for moving right
                print("COST for moving RIGHT" + str(cost))
                # suck the dirt and mark it as clean
                goal_state['B'] = '0'
                cost += 1                      #cost for suck
                print("COST for SUCK " + str(cost))
                print("Location B has been Cleaned. ")
```

```
        else:
            print("No action" + str(cost))
            # suck and mark clean
            print("Location B is already clean.")


    if status_input == '0':
        print("Location A is already clean ")
        if status_input_complement == '1':# if B is Dirty
            print("Location B is Dirty.")
            print("Moving RIGHT to the Location B. ")
            cost += 1                      #cost for moving right
            print("COST for moving RIGHT " + str(cost))
            # suck the dirt and mark it as clean
            goal_state['B'] = '0'
            cost += 1                      #cost for suck
            print("Cost for SUCK" + str(cost))
            print("Location B has been Cleaned. ")
        else:
            print("No action " + str(cost))
            print(cost)
            # suck and mark clean
            print("Location B is already clean.")


else:
    print("Vacuum is placed in location B")
    # Location B is Dirty.
    if status_input == '1':
        print("Location B is Dirty.")
        # suck the dirt  and mark it as clean
```

```
    goal_state['B'] = '0'
    cost += 1  # cost for suck
    print("COST for CLEANING " + str(cost))
    print("Location B has been Cleaned.")

    if status_input_complement == '1':
        # if A is Dirty
        print("Location A is Dirty.")
        print("Moving LEFT to the Location A. ")
        cost += 1  # cost for moving right
        print("COST for moving LEFT" + str(cost))
        # suck the dirt and mark it as clean
        goal_state['A'] = '0'
        cost += 1  # cost for suck
        print("COST for SUCK " + str(cost))
        print("Location A has been Cleaned.")

else:
    print(cost)
    # suck and mark clean
    print("Location B is already clean.")

    if status_input_complement == '1':  # if A is Dirty
        print("Location A is Dirty.")
        print("Moving LEFT to the Location A. ")
        cost += 1  # cost for moving right
        print("COST for moving LEFT " + str(cost))
        # suck the dirt and mark it as clean
        goal_state['A'] = '0'
```

```
            cost += 1  # cost for suck
            print("Cost for SUCK " + str(cost))
            print("Location A has been Cleaned. ")
        else:
            print("No action " + str(cost))
            # suck and mark clean
            print("Location A is already clean.")

    # done cleaning
    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))


vacuum_world()
```

<u>Output: -</u>

```
Enter Location of Vacuum
B
Enter status of B
1
Enter status of other room
1
Initial Location Condition{'A': '1', 'B': '1'}
Vacuum is placed in location B
Location B is Dirty.
COST for CLEANING 1
Location B has been Cleaned.
Location A is Dirty.
```

Moving LEFT to the Location A.

COST for moving LEFT2

COST for SUCK 3

Location A has been Cleaned.

GOAL STATE:

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

Performance Measurement: 3

** Process exited - Return Code: 0 **

Press Enter to exit terminal

Program 6: - Create a knowledge base using prepositional logic and show whether the given query entails the knowledge base or not.

Code: -

```
combination =
[(True,True,True),(True,True,False),(True,False,True),(True,False,False),(False,True,True),(Fa
lse,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 query :  "))
```

```
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,val1,val2))

    return stack.pop()


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 entailment():
    global kb,q
    print('*'*10 + "Truth Table Reference" + '*'*10)
    print('kb','alpha')
    print('*'*10)
    for comb in combination:
        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

input_rules()
ans = entailment()
if ans:
    print("Knowledge base entails query")
else:
    print("Knowledge base does not entail query")

    #test
    #(~qv~pvr)^(~q^p)^q
```

(pvq)^(~rvp)

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

** Process exited - Return Code: 0 **

Press Enter to exit terminal

Program 7: - Create a knowledgebase 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)
```

<u>Output: -</u>

```
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.

** Process exited - Return Code: 0 **

Press Enter to exit terminal

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

if __name__ == "__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: -

```
Enter the first expression
knows(f(x),y)
Enter the second expression
knows(J,John)
The substitutions are:
['J / f(x)', 'John / y']



** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

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)))

main()
```

Output: -

```
Enter FOL:
∀x food(x) => likes(Johns, x)
The CNF form of the given FOL is:
~ food(A) V likes(Johns, A)



** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

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()
main()
```

Output: -

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

** Process exited - Return Code: 0 **

Press Enter to exit terminal
