AI LAB RECORD

LAB 1: Implement Tic –Tac –Toe Game.

```
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:

```
Welcome to Tic Tac Toe!
   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
Please select a position to place an 'X' (1-9): 1
   |   |
 X |   |
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
Computer placed an 'O' in position 3 :
   |   |
 X |   | O
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
```

```
   |   |
Please select a position to place an 'X' (1-9): 2
   |   |
 X | X | O
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
Computer placed an 'O' in position 7 :
   |   |
 X | X | O
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
 O |   |
   |   |
Please select a position to place an 'X' (1-9): 4
   |   |
 X | X | O
   |   |
-----------
   |   |
 X |   |
   |   |
-----------
   |   |
 O |   |
   |   |
```

```
   |   |
Computer placed an 'O' in position 5 :
   |   |
 X | X | O
   |   |
-----------
   |   |
 X | O |
   |   |
-----------
   |   |
 O |   |
   |   |
Sorry, O's won this time!
```

LAB 2: Solve 8 puzzle problem

```
class Node:
    def __init__(self, data, level, fval):
        """ Initialize the node with the data, level of the node and the
calculated fvalue """
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        """ Generate child nodes from the given node by moving the blank
space
            either in the four directions {up,down,left,right} """
        x, y = self.find(self.data, '_')
        """ val_list contains position values for moving the blank space in
either of
            the 4 directions [up,down,left,right] respectively. """
        val_list = [[x, y - 1], [x, y + 1], [x - 1, y], [x + 1, y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data, x, y, i[0], i[1])
            if child is not None:
                child_node = Node(child, self.level + 1, 0)
                children.append(child_node)
        return children

    def shuffle(self, puz, x1, y1, x2, y2):
        """ Move the blank space in the given direction and if the position
value are out
            of limits the return None """
        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 <
len(self.data):
            temp_puz = []
            temp_puz = self.copy(puz)
            temp = temp_puz[x2][y2]
            temp_puz[x2][y2] = temp_puz[x1][y1]
            temp_puz[x1][y1] = temp
            return temp_puz
        else:
            return None

    def copy(self, root):
        """ Copy function to create a similar matrix of the given node"""
        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
            temp.append(t)
        return temp

    def find(self, puz, x):
        """ Specifically used to find the position of the blank space """
        for i in range(0, len(self.data)):
            for j in range(0, len(self.data)):
                if puz[i][j] == x:
                    return i, j
```

```
class Puzzle:
    def __init__(self, size):
        """ Initialize the puzzle size by the specified size,open and
closed lists to empty """
        self.n = size
        self.open = []
        self.closed = []

    def accept(self):
        """ Accepts the puzzle from the user """
        puz = []
        for i in range(0, self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz

    def f(self, start, goal):
        """ Heuristic Function to calculate hueristic value f(x) = h(x) +
g(x) """
        return self.h(start.data, goal) + start.level

    def h(self, start, goal):
        """ Calculates the different between the given puzzles """
        temp = 0
        for i in range(0, self.n):
            for j in range(0, self.n):
                if start[i][j] != goal[i][j] and start[i][j] != '_':
                    temp += 1
        return temp

    def process(self):
        """ Accept Start and Goal Puzzle state"""
        print("Enter the start state matrix \n")
        start = self.accept()
        print("Enter the goal state matrix \n")
        goal = self.accept()

        start = Node(start, 0, 0)
        start.fval = self.f(start, goal)
        """ Put the start node in the open list"""
        self.open.append(start)
        print("\n\n")
        while True:
            cur = self.open[0]
            print("")
            print("  | ")
            print("  | ")
            print(" \\\'/ \n")
            for i in cur.data:
                for j in i:
                    print(j, end=" ")
                print("")
            """ If the difference between current and goal node is 0 we
have reached the goal node"""
            if (self.h(cur.data, goal) == 0):
                break
            for i in cur.generate_child():
                i.fval = self.f(i, goal)
                self.open.append(i)
            self.closed.append(cur)
            del self.open[0]
```

```
            """ sort the open list based on f value """
            self.open.sort(key=lambda x: x.fval, reverse=False)


puz = Puzzle(3)
puz.process()
```

OUTPUT:

```
_ 2 3
1 8 4
7 6 5

  |
  |
 \'/

1 2 3
_ 8 4
7 6 5

  |
  |
 \'/

1 2 3
8 _ 4
7 6 5
```

```
2 _ 3
1 8 4
7 6 5
Enter the goal state matrix

1 2 3
8 _ 4
7 6 5




  |
  |
 \'/

2 _ 3
1 8 4
7 6 5

  |
  |
 \'/
```

LAB 3: Implement Iterative deepening search algorithm.

```
def dfs(src,target,limit,visited_states):
    if src == target:
        return True
    if limit <= 0:
        return False
    visited_states.append(src)
    moves = possible_moves(src,visited_states)
    for move in moves:
        if dfs(move, target, limit-1, visited_states):
            return True
    return False

def possible_moves(state,visited_states):
    b = state.index(-1)
    d = []
    if b not in [0,1,2]:
        d += 'u'
    if b not in [6,7,8]:
        d += 'd'
    if b not in [2,5,8]:
        d += 'r'
    if b not in [0,3,6]:
        d += 'l'
    pos_moves = []
    for move in d:
        pos_moves.append(gen(state,move,b))
    return [move for move in pos_moves if move not in visited_states]
def gen(state, move, blank):
    temp = state.copy()
    if move == 'u':
        temp[blank-3], temp[blank] = temp[blank], temp[blank-3]
    if move == 'd':
        temp[blank+3], temp[blank] = temp[blank], temp[blank+3]
    if move == 'r':
        temp[blank+1], temp[blank] = temp[blank], temp[blank+1]
    if move == 'l':
        temp[blank-1], temp[blank] = temp[blank], temp[blank-1]
    return temp
def iddfs(src,target,depth):
    for i in range(depth):
        visited_states = []
        if dfs(src,target,i+1,visited_states):
            return True
    return False
#Test 1
src = [1,2,3,-1,4,5,6,7,8]
target = [1,2,3,4,5,-1,6,7,8]

depth = 1
iddfs(src, target, depth)
#Test 2
src = [3,5,2,8,7,6,4,1,-1]
target = [-1,3,7,8,1,5,4,6,2]

depth = 1
iddfs(src, target, depth)
# Test 2
src = [1,2,3,-1,4,5,6,7,8]
```

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

depth = 1
iddfs(src, target, depth)

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

for i in range(1, 100):
    val = iddfs(src,target,i)
    print(i, val)
    if val == True:
        break
```

OUTPUT:

```
1 False
2 False
3 False
4 False
5 False
6 False
7 False
8 False
9 False
10 False
11 False
12 False
13 False
14 False
15 False
16 False
17 False
18 False
19 False
20 False
21 False
22 False
23 False
24 False
25 True
```

LAB 4: Implement A* search algorithm.

```
# This class represents a node
class Node:
    # Initialize the class
    def __init__(self, position: (), parent: ()):
        self.position = position
        self.parent = parent
        self.g = 0  # Distance to start node
        self.h = 0  # Distance to goal node
        self.f = 0  # Total cost

    # Compare nodes
    def __eq__(self, other):
        return self.position == other.position

    # Sort nodes
    def __lt__(self, other):
        return self.f < other.f

    # Print node
    def __repr__(self):
        return ('({0},{1})'.format(self.position, self.f))


# Draw a grid
def draw_grid(map, width, height, spacing=2, **kwargs):
    for y in range(height):
        for x in range(width):
            print('%%-%ds' % spacing % draw_tile(map, (x, y), kwargs),
end='')
        print()


# Draw a tile
def draw_tile(map, position, kwargs):
    # Get the map value
    value = map.get(position)
    # Check if we should print the path
    if 'path' in kwargs and position in kwargs['path']: value = '+'
    # Check if we should print start point
    if 'start' in kwargs and position == kwargs['start']: value = '@'
    # Check if we should print the goal point
    if 'goal' in kwargs and position == kwargs['goal']: value = '$'
    # Return a tile value
    return value


# A* search
def astar_search(map, start, end):
    # Create lists for open nodes and closed nodes
    open = []
    closed = []
    # Create a start node and an goal node
    start_node = Node(start, None)
    goal_node = Node(end, None)
    # Add the start node
    open.append(start_node)

    # Loop until the open list is empty
```

```
    while len(open) > 0:
        # Sort the open list to get the node with the lowest cost first
        open.sort()
        # Get the node with the lowest cost
        current_node = open.pop(0)
        # Add the current node to the closed list
        closed.append(current_node)

        # Check if we have reached the goal, return the path
        if current_node == goal_node:
            path = []
            while current_node != start_node:
                path.append(current_node.position)
                current_node = current_node.parent
            # path.append(start)
            # Return reversed path
            return path[::-1]
        # Unzip the current node position
        (x, y) = current_node.position
        # Get neighbors
        neighbors = [(x - 1, y), (x + 1, y), (x, y - 1), (x, y + 1)]
        # Loop neighbors
        for next in neighbors:
            # Get value from map
            map_value = map.get(next)
            # Check if the node is a wall
            if (map_value == '#'):
                continue
            # Create a neighbor node
            neighbor = Node(next, current_node)
            # Check if the neighbor is in the closed list
            if (neighbor in closed):
                continue
            # Generate heuristics (Manhattan distance)
            neighbor.g = abs(neighbor.position[0] - start_node.position[0])
+ abs(
                neighbor.position[1] - start_node.position[1])
            neighbor.h = abs(neighbor.position[0] - goal_node.position[0])
+ abs(
                neighbor.position[1] - goal_node.position[1])
            neighbor.f = neighbor.g + neighbor.h
            # Check if neighbor is in open list and if it has a lower f
value
            if (add_to_open(open, neighbor) == True):
                # Everything is green, add neighbor to open list
                open.append(neighbor)
    # Return None, no path is found
    return None


# Check if a neighbor should be added to open list
def add_to_open(open, neighbor):
    for node in open:
        if (neighbor == node and neighbor.f >= node.f):
            return False
    return True


# The main entry point for this module
def main():
    # Get a map (grid)
```

```
    map = {}
    chars = ['c']
    start = None
    end = None
    width = 0
    height = 0
    # Open a file
    fp = open('maze-grid.txt', 'r')

    # Loop until there is no more lines
    while len(chars) > 0:
        # Get chars in a line
        chars = [str(i) for i in fp.readline().strip()]
        # Calculate the width
        width = len(chars) if width == 0 else width
        # Add chars to map
        for x in range(len(chars)):
            map[(x, height)] = chars[x]
            if (chars[x] == '@'):
                start = (x, height)
            elif (chars[x] == '$'):
                end = (x, height)

        # Increase the height of the map
        if (len(chars) > 0):
            height += 1
    # Close the file pointer
    fp.close()
    # Find the closest path from start(@) to end($)
    path = astar_search(map, start, end)
    print()
    print(path)
    print()
    draw_grid(map, width, height, spacing=1, path=path, start=start,
goal=end)
    print()
    print('Steps to goal: {0}'.format(len(path)))
    print()


# Tell python to run main method
if __name__ == "__main__": main()
```

OUTPUT:

```
#+#.#############.#########.#.#.###.###.#.#.###.#.#####.#.#######.#.#.#.#####.#.#
#+#.#+++++++++++#.#.#.#.....#.#.....#...#.#.....#...#.#.#.#.#...#.#.#.#.#.....#.#
#+###+#########+#.#.#.#######.#######.###.#####.###.#.#.#.#.###.#.#.#.#.#####.#.#
#+++++#+++#+++++#...#.........#.....#...#.....#...#...#.#.....#.#...#.#.#.....#.#
#.#####+#+#+#######.###########.#######.#.#######.###.#.###.###.#####.#.#.#####.#
#.....#+#+#+++#...#.#++++++++#.........#.#...#.......#.#.#...#...#.....#.#.#...#.#
#######+#+###+#.###.#+#####+#.#####.###.#.#.#.#######.#.#####.###.#####.#.###.#.#
#+++++++#+#+++#.....#+#...#+#...#.#.....#.#.#.#.#.....#...#...#...#.....#...#.#.#
#+#######+#+#.#####.#+###.#+###.#.#######.#.#.#.#.#######.#.###.#.###.#####.#.#.#
#+#.#+++++#+#.#+++#.#+++#.#+++#...#.#...#.#...#.#.....#.#...#...#...#.......#...#
#+#.#+#####+#.#+#+#####+#.###+###.#.#.#.#.#####.#####.#.#####.#####.#########.###
#+#..+#..+++#.#+#+#+++#+++#.#+#...#...#.#.#...#.....#...#.#...#...#.....#...#.#.#
#+###+###+#.###+#+#+#+###+#.#+#.#######.#.#.#.#####.###.#.#.###.#.#####.###.#.#.#
#+++#+++#+#.#+++#+#+#+++#+#.#+#.#.......#...#.........#.#...#...#.#...#...#.#...#
#.#+###+#+#.#+###+#+###+#+#.#+#.###.###.###########.###.#.###.###.###.###.#.###.#
#.#+++#+#+#.#+++#+++#+++#+#.#+#.....#...#...#.....#.#...#.....#.....#.#...#...#.#
#.###+#+#+#####+#####+#.#+#.#+#######.###.#.#####.#.#.#############.#.#.###.#.#.#
#...#+#+++#+++#+++++#+#.#+#.#+#+++#...#.#.#.......#.#.#...#...#...#...#.#.#.#...#
###.#+#####+#+#####+#+###+#.#+#+#+#.###.#.#########.#.#.#.#.#.#.#.#####.#.#.#####
#...#+++++++#+++++++#+++++..#+++#+++++++@...........#...#...#...#.......#.......#
#################################################################################

Steps to goal: 339
```

```
[(39, 39), (38, 39), (37, 39), (36, 39), (35, 39), (34, 39), (33, 39), (33, 38), (33, 37), (32, 37), (31, 37), (31, 38), (31, 39), (30,
 39), (29, 39), (29, 38), (29, 37), (29, 36), (29, 35), (29, 34), (29, 33), (29, 32), (29, 31), (29, 30), (29, 29), (28, 29), (27, 29),
 (27, 28), (27, 27), (27, 26), (27, 25), (26, 25), (25, 25), (24, 25), (23, 25), (22, 25), (21, 25), (21, 26), (21, 27), (21, 28), (21,
 29), (22, 29), (23, 29), (23, 30), (23, 31), (24, 31), (25, 31), (25, 32), (25, 33), (25, 34), (25, 35), (25, 36), (25, 37), (25, 38),
 (25, 39), (24, 39), (23, 39), (22, 39), (21, 39), (21, 38), (21, 37), (21, 36), (21, 35), (22, 35), (23, 35), (23, 34), (23, 33), (22,
 33), (21, 33), (21, 32), (21, 31), (20, 31), (19, 31), (19, 32), (19, 33), (19, 34), (19, 35), (18, 35), (17, 35), (17, 34), (17, 33),
 (17, 32), (17, 31), (17, 30), (17, 29), (16, 29), (15, 29), (15, 30), (15, 31), (15, 32), (15, 33), (14, 33), (13, 33), (13, 34), (13,
 35), (14, 35), (15, 35), (15, 36), (15, 37), (16, 37), (17, 37), (18, 37), (19, 37), (19, 38), (19, 39), (18, 39), (17, 39), (16, 39),
 (15, 39), (14, 39), (13, 39), (13, 38), (13, 37), (12, 37), (11, 37), (11, 38), (11, 39), (10, 39), (9, 39), (8, 39), (7, 39), (6, 39)
, (5, 39), (5, 38), (5, 37), (5, 36), (5, 35), (4, 35), (3, 35), (3, 34), (3, 33), (2, 33), (1, 33), (1, 32), (1, 31), (1, 30), (1, 29)
, (1, 28), (1, 27), (2, 27), (3, 27), (4, 27), (5, 27), (6, 27), (7, 27), (7, 26), (7, 25), (7, 24), (7, 23), (8, 23), (9, 23), (9, 24)
, (9, 25), (9, 26), (9, 27), (9, 28), (9, 29), (8, 29), (7, 29), (6, 29), (5, 29), (5, 30), (5, 31), (5, 32), (5, 33), (6, 33), (7, 33)
, (7, 34), (7, 35), (7, 36), (7, 37), (8, 37), (9, 37), (9, 36), (9, 35), (9, 34), (9, 33), (9, 32), (9, 31), (10, 31), (11, 31), (11,
30), (11, 29), (11, 28), (11, 27), (12, 27), (13, 27), (13, 26), (13, 25), (12, 25), (11, 25), (11, 24), (11, 23), (12, 23), (13, 23),
(14, 23), (15, 23), (15, 22), (15, 21), (14, 21), (13, 21), (12, 21), (11, 21), (10, 21), (9, 21), (8, 21), (7, 21), (6, 21), (5, 21),
(5, 22), (5, 23), (4, 23), (3, 23), (2, 23), (1, 23), (1, 22), (1, 21), (1, 20), (1, 19), (1, 18), (1, 17), (2, 17), (3, 17), (3, 16),
(3, 15), (2, 15), (1, 15), (1, 14), (1, 13), (2, 13), (3, 13), (4, 13), (5, 13), (5, 12), (5, 11), (6, 11), (7, 11), (7, 12), (7, 13),
(8, 13), (9, 13), (9, 12), (9, 11), (9, 10), (9, 9), (8, 9), (7, 9), (6, 9), (5, 9), (4, 9), (3, 9), (3, 10), (3, 11), (2, 11), (1, 11)
, (1, 10), (1, 9), (1, 8), (1, 7), (1, 6), (1, 5), (2, 5), (3, 5), (4, 5), (5, 5), (6, 5), (7, 5), (8, 5), (9, 5), (10, 5), (11, 5), (1
1, 6), (11, 7), (11, 8), (11, 9), (11, 10), (11, 11), (11, 12), (11, 13), (11, 14), (11, 15), (10, 15), (9, 15), (8, 15), (7, 15), (7,
16), (7, 17), (6, 17), (5, 17), (5, 18), (5, 19), (6, 19), (7, 19), (8, 19), (9, 19), (10, 19), (11, 19), (12, 19), (13, 19), (14, 19),
 (15, 19), (15, 18), (15, 17), (15, 16), (15, 15), (14, 15), (13, 15), (13, 14), (13, 13), (14, 13), (15, 13), (16, 13), (17, 13), (18,
 13), (19, 13), (20, 13), (21, 13), (21, 12), (21, 11), (20, 11), (19, 11), (18, 11), (17, 11), (17, 10), (17, 9), (16, 9), (15, 9), (1
```

LAB 5: Implement vacuum cleaner agent.

```
#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()
```

OUTPUT:

```
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
```

LAB 6: Create a knowledgebase using prepositional logic and show that the given query entails the knowledge base or not.

```
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("The Knowledge Base entails query")
else:
    print("The Knowledge Base does not entail query")
#Test 2
input_rules()
ans = entailment()
if ans:
    print("The Knowledge Base entails query")
else:
    print("The Knowledge Base does not entail query")
```

OUTPUT :

```
Enter rule: (~qv~pvr)^(~q^p)^q
Enter the Query: r
**********Truth Table Reference**********
kb alpha
**********
False True
----------
False False
----------
False True
----------
False False
----------
False True
----------
False False
----------
False True
----------
False False
----------
The Knowledge Base entails query
Enter rule: (pvq)^(~rvp)
Enter the Query: r
**********Truth Table Reference**********
kb alpha
**********
True True
----------
True False
----------
The Knowledge Base does not entail query
> 
```

LAB 7: Create a knowledgebase using prepositional logic and prove the given query using resolution

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

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

```
                            \nA contradiction is found when {negate(query)}
is assumed as true. Hence, {query} is true."
                            return steps
            for clause in clauses:
                if clause not in temp and clause != reverse(clause) and
reverse(clause) not in temp:
                    temp.append(clause)
                    steps[clause] = f'Resolved from {temp[i]} and
{temp[j]}.'
            j = (j + 1) % n
        i += 1
    return steps
def resolution(kb, query):
    kb = kb.split(' ')
    steps = resolve(kb, query)
    print('\nStep\t|Clause\t|Derivation\t')
    print('-' * 30)
    i = 1
    for step in steps:
        print(f' {i}.\t| {step}\t| {steps[step]}\t')
        i += 1
def main():
    print("Enter the kb:")
    kb = input()
    print("Enter the query:")
    query = input()
    resolution(kb,query)
#test 1
#(P^Q)<=>R : (Rv~P)v(Rv~Q)^(~RvP)^(~RvQ)
main()
#test 2
#(P=>Q)=>Q, (P=>P)=>R, (R=>S)=>~(S=>Q)
main()
```

OUTPUT:

```
Enter the kb:
PVQ PVR ~PVR RVS RV~Q ~SV~Q
Enter the query:
R

Step    |Clause |Derivation
------------------------------
 1. | PVQ    | Given.
 2. | PVR    | Given.
 3. | ~PVR   | Given.
 4. | RVS    | Given.
 5. | RV~Q   | Given.
 6. | ~SV~Q | Given.
 7. | ~R     | Negated conclusion.
 8. | QvR    | Resolved from PVQ and ~PVR.
 9. | PvR    | Resolved from PVQ and RV~Q.
 10.    | Pv~S  | Resolved from PVQ and ~SV~Q.
 11.    | P | Resolved from PVR and ~R.
 12.    | ~P    | Resolved from ~PVR and ~R.
 13.    | Rv~S  | Resolved from ~PVR and Pv~S.
 14.    | R | Resolved from ~PVR and P.
 15.    | Rv~Q  | Resolved from RVS and ~SV~Q.
 16.    | S | Resolved from RVS and ~R.
 17.    | ~Q    | Resolved from RV~Q and ~R.
 18.    | Q | Resolved from ~R and QvR.
 19.    | ~S    | Resolved from ~R and Rv~S.
 20.    |   | Resolved ~R and R to ~RvR, which is in turn null.

A contradiction is found when ~R is assumed as true. Hence, R is true.
>
```

```
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.
```

LAB 8: Implement unification in first order logic

```
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):
```

```python
        return [(exp2, exp1)]

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

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

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

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

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

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

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

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

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

```
print("--------------------------")
```

OUTPUT:

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

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

Enter the first expression
Student(x)
Enter the second expression
Teacher(Rose)
Cannot be unified as the predicates do not match!
The substitutions are:
[]

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

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

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

Enter the first expression
like(A,y)
Enter the second expression
like(K,g(x))
A and K are constants. Cannot be unified
The substitutions are:
[]
--------------------------
---------------------------
>
```

LAB 9: Convert given first order logic statement into Conjunctive Normal Form (CNF).import re

```
print("Enter FOL")
def remove_brackets(source, id):
    reg = '\(([^\(]*?)\)'
    m = re.search(reg, source)
    if m is None:
        return None, None
    new_source = re.sub(reg, str(id), source, count=1)
    return new_source, m.group(1)


class logic_base:
    def __init__(self, input):
        self.my_stack = []
        self.source = input
        final = input
        while 1:
            input, tmp = remove_brackets(input, len(self.my_stack))
            if input is None:
                break
            final = input
            self.my_stack.append(tmp)
        self.my_stack.append(final)

    def get_result(self):
        root = self.my_stack[-1]
        m = re.match('\s*([0-9]+)\s*$', root)
        if m is not None:
            root = self.my_stack[int(m.group(1))]
        reg = '(\d+)'
        while 1:
            m = re.search(reg, root)
            if m is None:
                break
            new = '(' + self.my_stack[int(m.group(1))] + ')'
            root = re.sub(reg, new, root, count=1)
        return root

    def merge_items(self, logic):
        reg0 = '(\d+)'
        reg1 = 'neg\s+(\d+)'
        flag = False
        for i in range(len(self.my_stack)):
            target = self.my_stack[i]
            if logic not in target:
                continue
            m = re.search(reg1, target)
            if m is not None:
                continue
            m = re.search(reg0, target)
            if m is None:
                continue
            for j in re.findall(reg0, target):
                child = self.my_stack[int(j)]
                if logic not in child:
                    continue
```

```
                new_reg = "(^|\s)" + j + "(\s|$)"
                self.my_stack[i] = re.sub(new_reg, ' ' + child + ' ',
self.my_stack[i], count=1)
                self.my_stack[i] = self.my_stack[i].strip()
                flag = True
        if flag:
            self.merge_items(logic)


class ordering(logic_base):
    def run(self):
        flag = False
        for i in range(len(self.my_stack)):
            new_source = self.add_brackets(self.my_stack[i])
            if self.my_stack[i] != new_source:
                self.my_stack[i] = new_source
                flag = True
        return flag

    def add_brackets(self, source):
        reg = "\s+(and|or|imp|iff)\s+"
        if len(re.findall(reg, source)) < 2:
            return source
        reg_and = "(neg\s+)?\S+\s+and\s+(neg\s+)?\S+"
        m = re.search(reg_and, source)
        if m is not None:
            return re.sub(reg_and, "(" + m.group(0) + ")", source, count=1)
        reg_or = "(neg\s+)?\S+\s+or\s+(neg\s+)?\S+"
        m = re.search(reg_or, source)
        if m is not None:
            return re.sub(reg_or, "(" + m.group(0) + ")", source, count=1)
        reg_imp = "(neg\s+)?\S+\s+imp\s+(neg\s+)?\S+"
        m = re.search(reg_imp, source)
        if m is not None:
            return re.sub(reg_imp, "(" + m.group(0) + ")", source, count=1)
        reg_iff = "(neg\s+)?\S+\s+iff\s+(neg\s+)?\S+"
        m = re.search(reg_iff, source)
        if m is not None:
            return re.sub(reg_iff, "(" + m.group(0) + ")", source, count=1)


class replace_iff(logic_base):
    def run(self):
        final = len(self.my_stack) - 1
        flag = self.replace_all_iff()
        self.my_stack.append(self.my_stack[final])
        return flag

    def replace_all_iff(self):
        flag = False
        for i in range(len(self.my_stack)):
            ans = self.replace_iff_inner(self.my_stack[i],
len(self.my_stack))
            if ans is None:
                continue
            self.my_stack[i] = ans[0]
            self.my_stack.append(ans[1])
            self.my_stack.append(ans[2])
            flag = True
        return flag
```

```
    def replace_iff_inner(self, source, id):
        reg = '^(.*?)\s+iff\s+(.*?)$'
        m = re.search(reg, source)
        if m is None:
            return None
        a, b = m.group(1), m.group(2)
        return (str(id) + ' and ' + str(id + 1), a + ' imp ' + b, b + ' imp
' + a)


class replace_imp(logic_base):
    def run(self):
        flag = False
        for i in range(len(self.my_stack)):
            ans = self.replace_imp_inner(self.my_stack[i])
            if ans is None:
                continue
            self.my_stack[i] = ans
            flag = True
        return flag

    def replace_imp_inner(self, source):
        reg = '^(.*?)\s+imp\s+(.*?)$'
        m = re.search(reg, source)
        if m is None:
            return None
        a, b = m.group(1), m.group(2)
        if 'neg ' in a:
            return a.replace('neg ', '') + ' or ' + b
        return 'neg ' + a + ' or ' + b


class de_morgan(logic_base):
    def run(self):
        reg = 'neg\s+(\d+)'
        flag = False
        final = len(self.my_stack) - 1
        for i in range(len(self.my_stack)):
            target = self.my_stack[i]
            m = re.search(reg, target)
            if m is None:
                continue
            flag = True
            child = self.my_stack[int(m.group(1))]
            self.my_stack[i] = re.sub(reg, str(len(self.my_stack)), target,
count=1)
            self.my_stack.append(self.doing_de_morgan(child))
            break
        self.my_stack.append(self.my_stack[final])
        return flag

    def doing_de_morgan(self, source):
        items = re.split('\s+', source)
        new_items = []
        for item in items:
            if item == 'or':
                new_items.append('and')
            elif item == 'and':
                new_items.append('or')
            elif item == 'neg':
                new_items.append('neg')
```

```
            elif len(item.strip()) > 0:
                new_items.append('neg')
                new_items.append(item)
        for i in range(len(new_items) - 1):
            if new_items[i] == 'neg':
                if new_items[i + 1] == 'neg':
                    new_items[i] = ''
                    new_items[i + 1] = ''
        return ' '.join([i for i in new_items if len(i) > 0])


class distributive(logic_base):
    def run(self):
        flag = False
        reg = '(\d+)'
        final = len(self.my_stack) - 1
        for i in range(len(self.my_stack)):
            target = self.my_stack[i]
            if 'or' not in self.my_stack[i]:
                continue
            m = re.search(reg, target)
            if m is None:
                continue
            for j in re.findall(reg, target):
                child = self.my_stack[int(j)]
                if 'and' not in child:
                    continue
                new_reg = "(^|\s)" + j + "(\s|$)"
                items = re.split('\s+and\s+', child)
                tmp_list = [str(j) for j in range(len(self.my_stack),
len(self.my_stack) + len(items))]
                for item in items:
                    self.my_stack.append(re.sub(new_reg, ' ' + item + ' ',
target).strip())
                self.my_stack[i] = ' and '.join(tmp_list)
                flag = True
            if flag:
                break
        self.my_stack.append(self.my_stack[final])
        return flag


class simplification(logic_base):
    def run(self):
        old = self.get_result()
        for i in range(len(self.my_stack)):
            self.my_stack[i] = self.reducing_or(self.my_stack[i])
        # self.my_stack[i] = self.reducing_and(self.my_stack[i])
        final = self.my_stack[-1]
        self.my_stack[-1] = self.reducing_and(final)
        return len(old) != len(self.get_result())

    def reducing_and(self, target):
        if 'and' not in target:
            return target
        items = set(re.split('\s+and\s+', target))
        for item in list(items):
            if ('neg ' + item) in items:
                return ''
            if re.match('\d+$', item) is None:
                continue
```

```
            value = self.my_stack[int(item)]
            if self.my_stack.count(value) > 1:
                value = ''
                self.my_stack[int(item)] = ''
            if value == '':
                items.remove(item)
        return ' and '.join(list(items))

    def reducing_or(self, target):
        if 'or' not in target:
            return target
        items = set(re.split('\s+or\s+', target))
        for item in list(items):
            if ('neg ' + item) in items:
                return ''
        return ' or '.join(list(items))


def merging(source):
    old = source.get_result()
    source.merge_items('or')
    source.merge_items('and')
    return old != source.get_result()


def run(input):
    all_strings = []
    # all_strings.append(input)
    zero = ordering(input)
    while zero.run():
        zero = ordering(zero.get_result())
    merging(zero)

    one = replace_iff(zero.get_result())
    one.run()
    all_strings.append(one.get_result())
    merging(one)

    two = replace_imp(one.get_result())
    two.run()
    all_strings.append(two.get_result())
    merging(two)

    three, four = None, None
    old = two.get_result()
    three = de_morgan(old)
    while three.run():
        pass
    all_strings.append(three.get_result())
    merging(three)
    three_helf = simplification(three.get_result())
    three_helf.run()

    four = distributive(three_helf.get_result())
    while four.run():
        pass
    merging(four)
    five = simplification(four.get_result())
    five.run()
    all_strings.append(five.get_result())
    return all_strings
```

```
inputs = input().split('\n')
for input in inputs:
    for item in run(input):
        print(item)
    # output.write('\n')
```

OUTPUT:

```
Enter FOL
cold and precipitation imp snow
(cold and precipitation) imp snow
neg (cold and precipitation) or snow
(neg cold or neg precipitation) or snow
snow or neg cold or neg precipitation
>
```

```
Enter FOL
(animal(z) and kills (x,z)) imp (neg Loves(y,z))
(animal(z) and kills (x,z)) imp (neg Loves(y,z))
neg (animal(z) and kills (x,z)) or (neg Loves(y,z))
(neg animal(z) or neg kills (neg x,z)) or (neg Loves(y,z))
neg animal(z) or neg kills (neg x,z) or (neg Loves(y,z))
>
```

LAB 10: Create a knowledgebase consisting of first order logic statements and prove the given query using forward reasoning.

```
import re


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


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


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


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


    def splitExpression(self, expression):
```

```
        predicate = getPredicates(expression)[0]
        params = getAttributes(expression)[0].strip('()').split(',')
        return [predicate, params]

    def getResult(self):
        return self.result

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

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

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

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

    def evaluate(self, facts):
```

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


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


    def tell(self, e):
        if '=>' in e:
            self.implications.add(Implication(e))
        else:
```

```
            self.facts.add(Fact(e))
        for i in self.implications:
            res = i.evaluate(self.facts)
            if res:
                self.facts.add(res)

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

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

def main():
    kb = KB()
    print("Enter KB: (enter e to exit)")
    while True:
        t = input()
        if(t == 'e'):
```

```
            break
        kb.tell(t)
    print("Enter Query:")
    q = input()
    kb.query(q)
    kb.display()


main()
```

OUTPUT:

```
Enter KB: (enter e to exit)
missile(x)=>weapon(x)
misssile(M1)
enemy(x,America)=>hostile(x)
america(West)
enemy(None,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. america(West)
        2. owns(Nono,M1)
        3. sells(West,M1,Nono)
        4. misssile(M1)
        5. enemy(None,America)
        6. criminal(West)
        7. hostile(None)
```