VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence

Submitted by

VAIBHAV KIRAN P (1BM21CS233)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Nov-2023 to Feb-2024

B. M. S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "**Artificial Intelligence**" carried out by **VAIBHAV KIRAN P (1BM21CS233),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the academic semester June-2023 to Sep-2023. The Lab report has been approved as it satisfies the academic requirements in respect of a **Artificial Intelligence (22CS5PCAIN)** work prescribed for the said degree.

Sandhya A Kulkarni

Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak

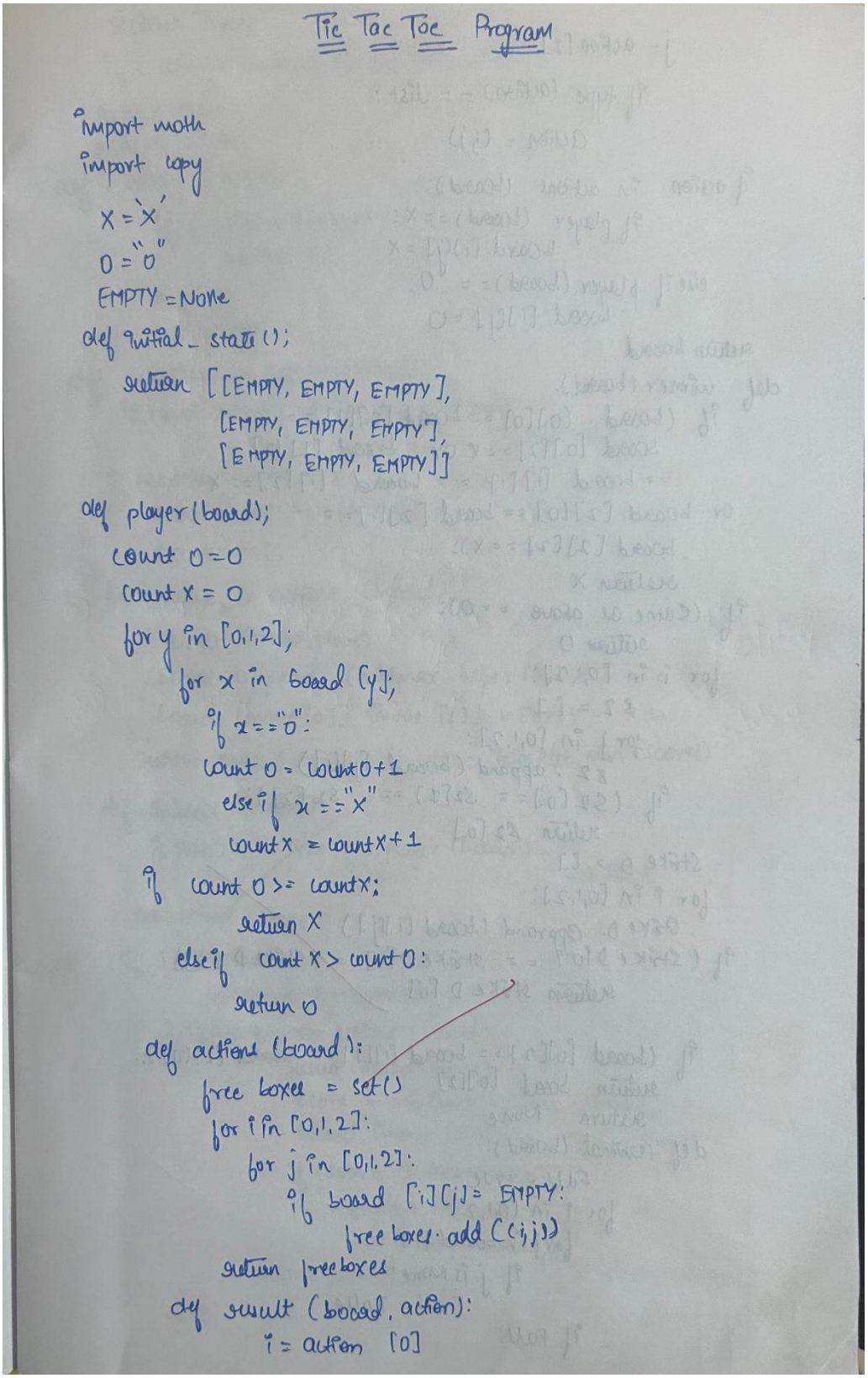
Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

Course Outcome

CO1	Apply knowledge of agent architecture, searching and reasoning techniques for different applications.
CO2	Analyse Searching and Inferencing Techniques.
CO3	Design a reasoning system for a given requirement.
CO4	Conduct practical experiments for demonstrating agents, searching and inferencing.

1. **Implement Tic –Tac –Toe Game.**



Tic Tac Toe Program

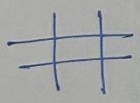
```
import math
import copy
  X = "X"
  O = "O"
  EMPTY = None
def initial_state();
   return [[EMPTY, EMPTY, EMPTY],
           [EMPTY, EMPTY, EMPTY],
           [EMPTY, EMPTY, EMPTY]]

def player(board);
  count O = 0
  count X = 0
  for y in [0,1,2];
     for x in board[y];
       if x == "O":
       count O = count O + 1
        else if x == "X"
        count X = count X + 1
   if count O >= count X;
        return X
   elseif count X > count O:
        return O
  def actions (board):
     free boxes = set()
     for i in [0,1,2]:
        for j in [0,1,2]:
          if board [i][j] = EMPTY:
             free boxes.add ((i,j))
     return free boxes
  def result (board, action):
        i = action [0]
```

```
        j = action[1]
        if type(action) == list:
            action = (i, j)
    if action in actions(board):
        if player(board) == X:
            board[i][j] = X
        else if player(board) == O:
            board[i][j] = O
    return board
def winner(board):
    if (board[0][0] == board[0][1] ==
        board[0][2] == X or board[1][0]
        == board[1][1] == board[1][2] == X
    or board[2][0] == board[2][1] ==
        board[2][2] == X):
        return X
    if (same as above == O):
        return O
    for i in [0,1,2]:
        s2 = []
        for j in [0,1,2]:
            s2.append(board[j][i])
        if (s2[0] == s2[1] == s2[2]):
            return s2[0]
    strikeD = []
    for i in [0,1,2]:
        strikeD.append(board[i][i])
    if (strikeD[0] == strikeD[1] == strikeD[2]):
        return strikeD[0]

    if (board[0][2] == board[1][1] == board[2][0]):
        return board[0][2]
    return None
def terminal(board):
        Full = True
        for i in [0,1,2]:
            for j in board[i]:
                if j is None:
                    Full = False
        if Full:
```

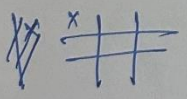
```
    return True
    if (winner(board) in not None):
    return True
    return False
def utility(board):
    if (winner(board) == 0:
        return -1
    else:
        return 0
def minimax - helper(board):
    isMaxTurn = True if player(board) == X
                                    else False
    if terminal(board):
        return utility(board)
        scores = []
    for move in actions(board):
        result(board, move)
        scores.append(minimax helper(board))
        board[move[0]][move[1]] = EMPTY
    return max(scores) if isMax Tern else min(scores)
def minimax(board):
    isMaxTurn = True if player(board) == X
                                    else False
    bestMove = None
    if isMaxTurn:
        bestScore = -math.inf
        for move in actions(board):
            result(board.move)
            Score = minimax.helper(board)
            board[move[0]][move[1]] = EMPTY
            if (score > bestScore):
                bestScore = score
                bestMove = move
        return bestMove
    else:
```

```
while not terminal (game_board):
    if player (game_board) == X:
        user_input = input ("\n Enter your move (row. column):")
        row. col = map (int, user_input . split (:))
        result (game_board, (row.col))
    else:
        print ("\n AI is making a move...")
        move = minimax (copy. deepcopy (game_board);
        result (game_board, move)
    print ("\n Current Board :")
    print_board (game_board)
if winner (game board) is not None:
    print ("\n The winner is: { winner (game board) }!);
    else:
        print ("\n It's a tie!")
```

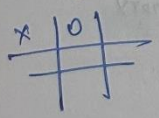
Output :-



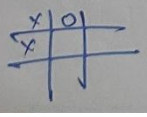
Enter your move (1-9): 1



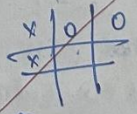
Enter your move (1-9): 2



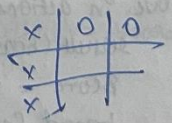
Enter your move (1-9): 4



Enter your move (1-9): 3

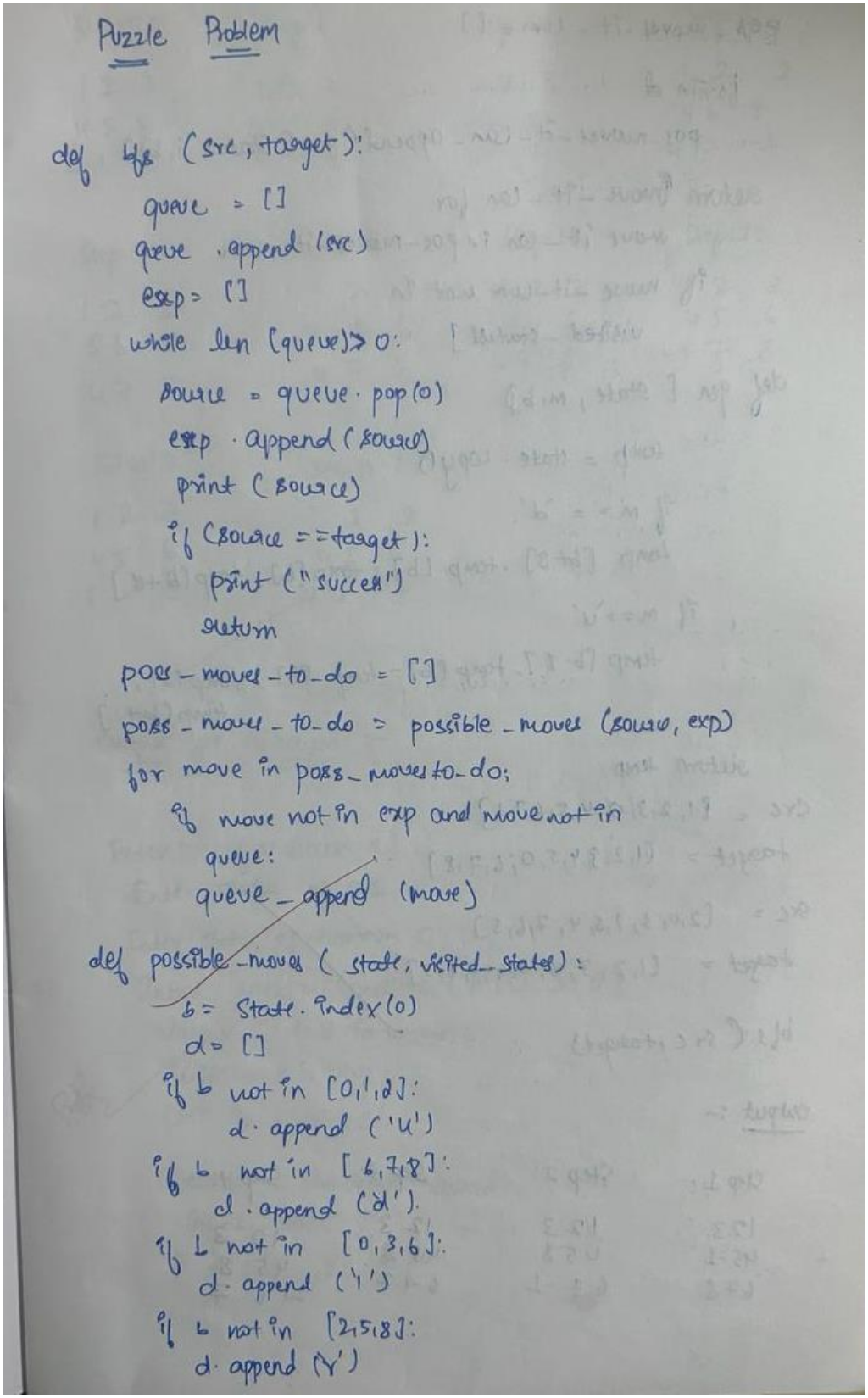


Enter your move (1-9): 7



player X wins!

2. Solve 8 puzzle problems.

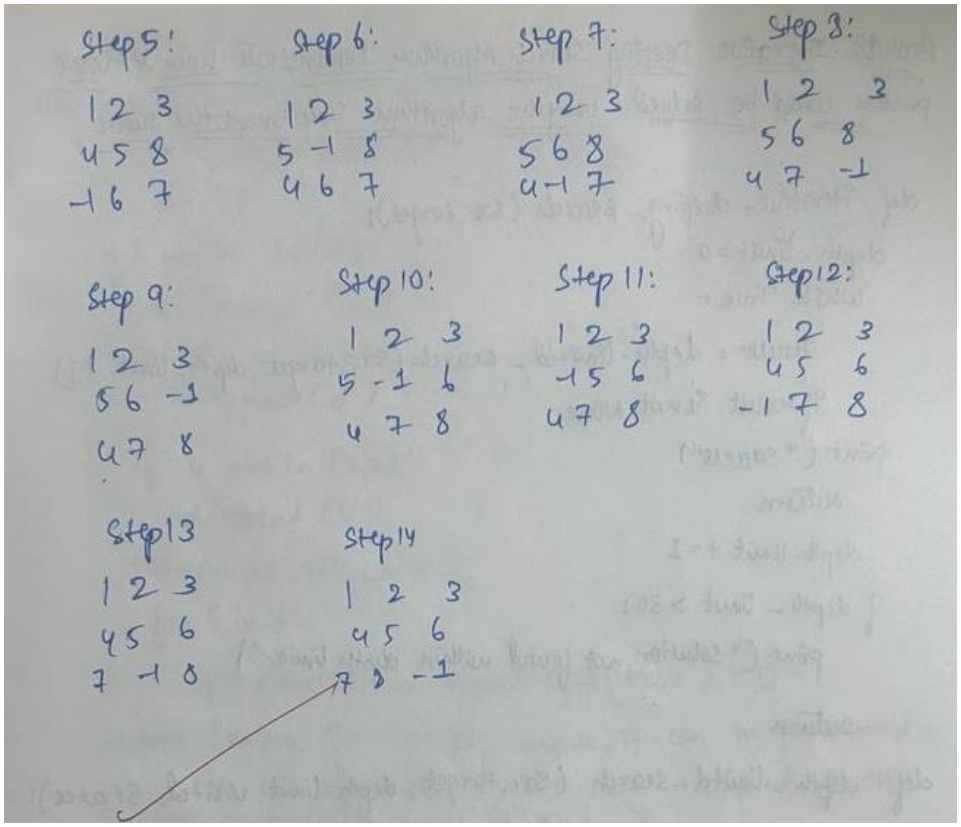


```
Puzzle Problem

def bfs (src, target):
    queue = []
    queue.append (src)
    exp = []
    while len (queue) > 0:
        source = queue.pop(0)
        exp.append (source)
        print (source)
        if (source == target):
            print ("success")
            return
    poss_moves_to_do = []
    poss_moves_to_do = possible_moves (source, exp)
    for move in poss_moves_to_do:
        if move not in exp and move not in
            queue:
            queue_append (move)

def possible_moves ( state, visited_states):
    b = state.index(0)
    d = []
    if b not in [0,1,2]:
        d.append ('u')
    if b not in [6,7,8]:
        d.append ('d')
    if b not in [0,3,6]:
        d.append ('l')
    if b not in [2,5,8]:
        d.append ('r')
```

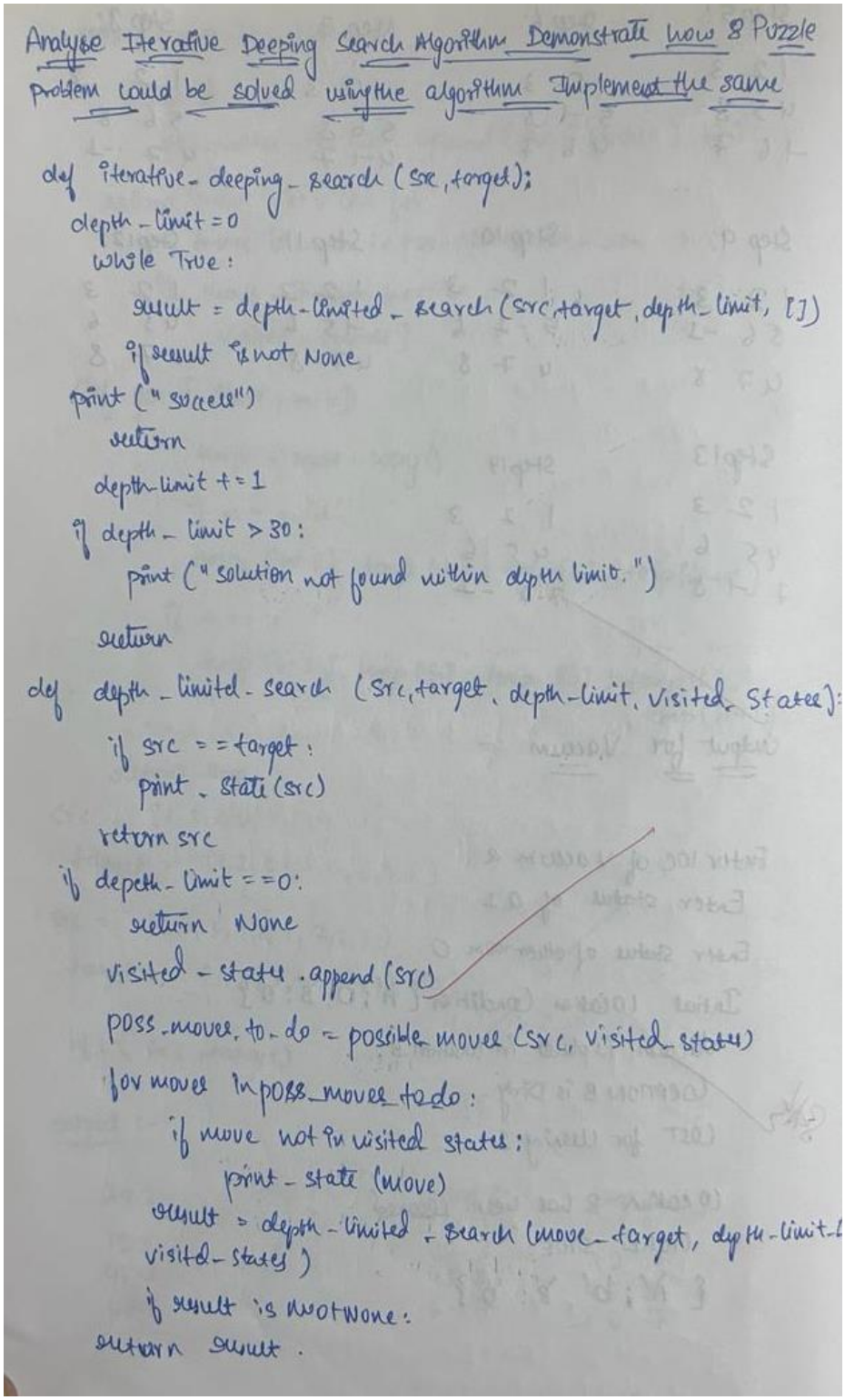
```
pos_moves_it_cur = []
    for i in d
        pos_moves_it_can.append(gen(state,i,b))
    return (move_it_can for
        move it_can in pos_moves_it_cur
        if move_it_cur not in
            visited_state)]

def gen [state, m, b]
        temp = state.copy()
        if m == 'd':
        temp [b+3].temp [b] = temp [b], temp [b+3]
        if m == 'u'
        temp [b-1], temp [b] = temp [b], temp [b],
                                temp (b+1)
    return temp
src = [1,2,3,0,4,5,6,7,8]
target = [1,2,3,4,5,0,6,7,8]
src = [2,0,3,1,8,4,7,6,5]
target = [1,2,3,8,0,4,7,6,5]
bfs(src,target)
```

output :-

Step 1:	Step 2:	Step 3:	Step 4:
123	123	123	123
45-1	458	458	458
678	67-1	6-17	-167


Step 5:
1 2 3
4 5 8
-1 6 7
Step 6:
1 2 3
5 -1 8
4 6 7
Step 7:
1 2 3
5 6 8
4 -1 7
Step 8:
1 2 3
5 6 8
4 7 -1
Step 9:
1 2 3
5 6 -1
4 7 8
Step 10:
1 2 3
5 -1 6
4 7 8
Step 11:
1 2 3
-1 5 6
4 7 8
Step 12:
1 2 3
4 5 6
-1 7 8
Step 13
1 2 3
4 5 6
7 -1 8
Step 14
1 2 3
4 5 6
7 8 -1

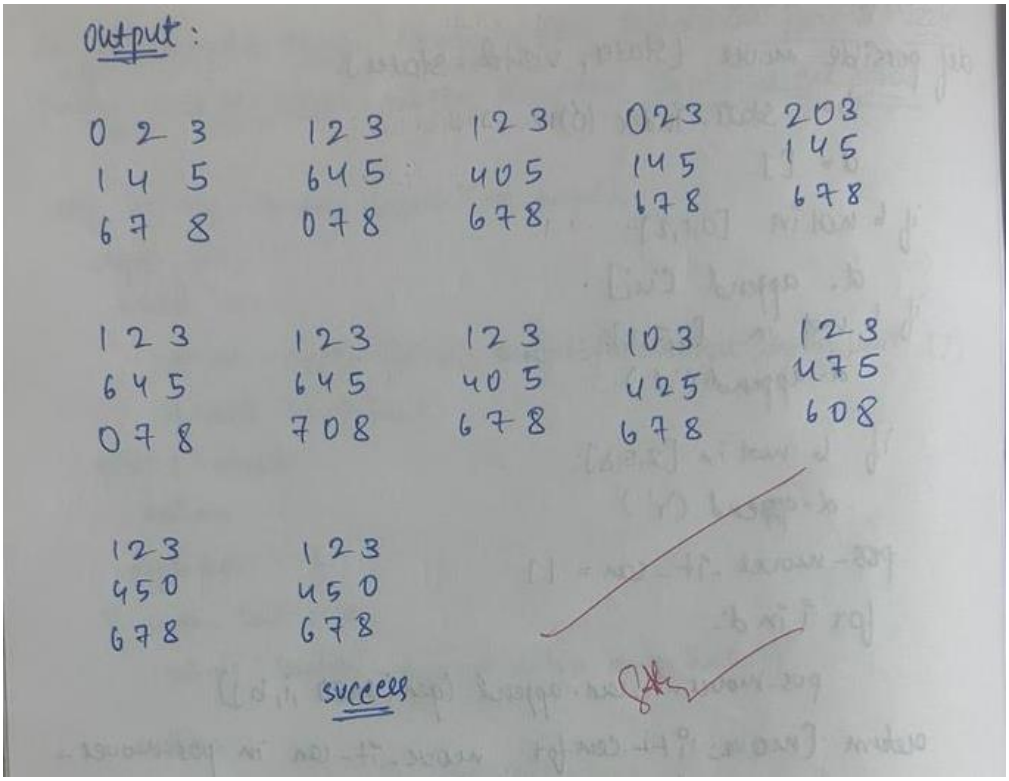
3. Implement Iterative deepening search algorithm.



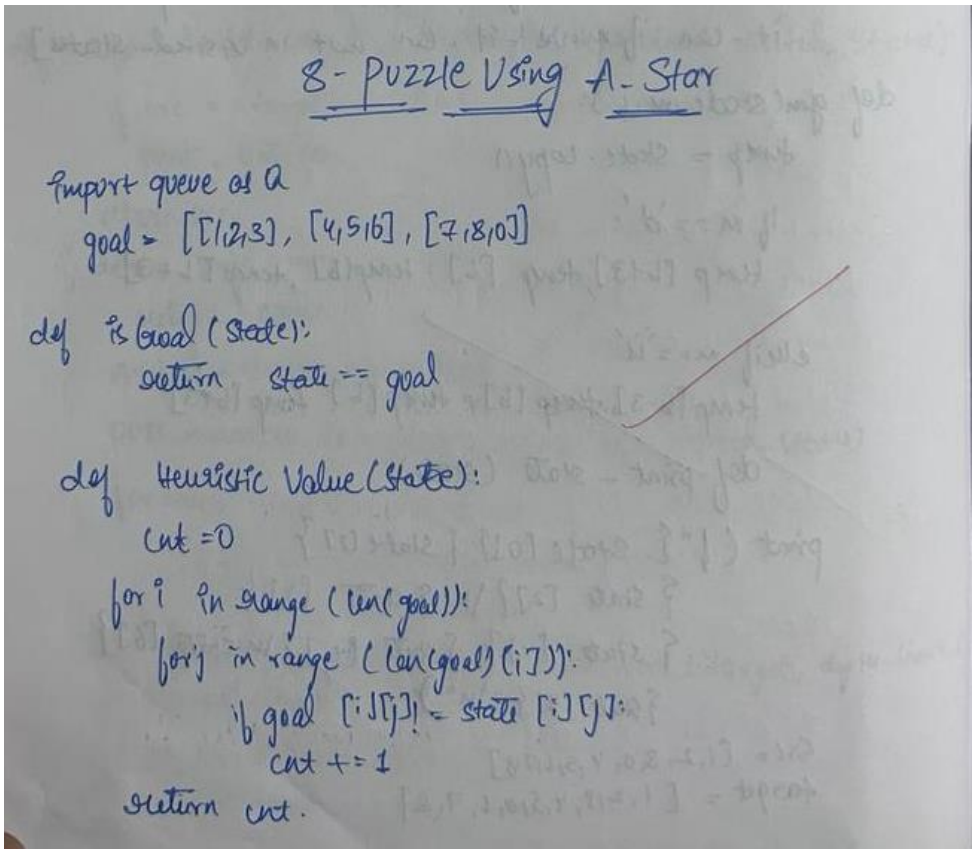
Analyse Iterative Deeping Search Algorithm Demonstrate how 8 Puzzle problem could be solved using the algorithm Implement the same

```
def iterative_deeping_search(src,target):
    depth_limit=0
    while True:
        result = depth_limited_search(src,target,depth_limit, [])
        if result is not None
    print("success")
        return
    depth_limit += 1
    if depth_limit > 30:
        print("solution not found within depth limit.")
        return
def depth_limited_search(src,target, depth_limit, visited_States):
    if src == target:
        print_state(src)
    return src
    if depth_limit == 0:
        return None
    visited_state.append(src)
    poss_moves_to_do = possible_moves(src, visited_state)
    for moves in poss_moves_to_do:
        if move not in visited states:
            print_state(move)
        result = depth_limited_search(move_target, depth_limit-1
        visited_states)
        if result is not None:
    return result.
```

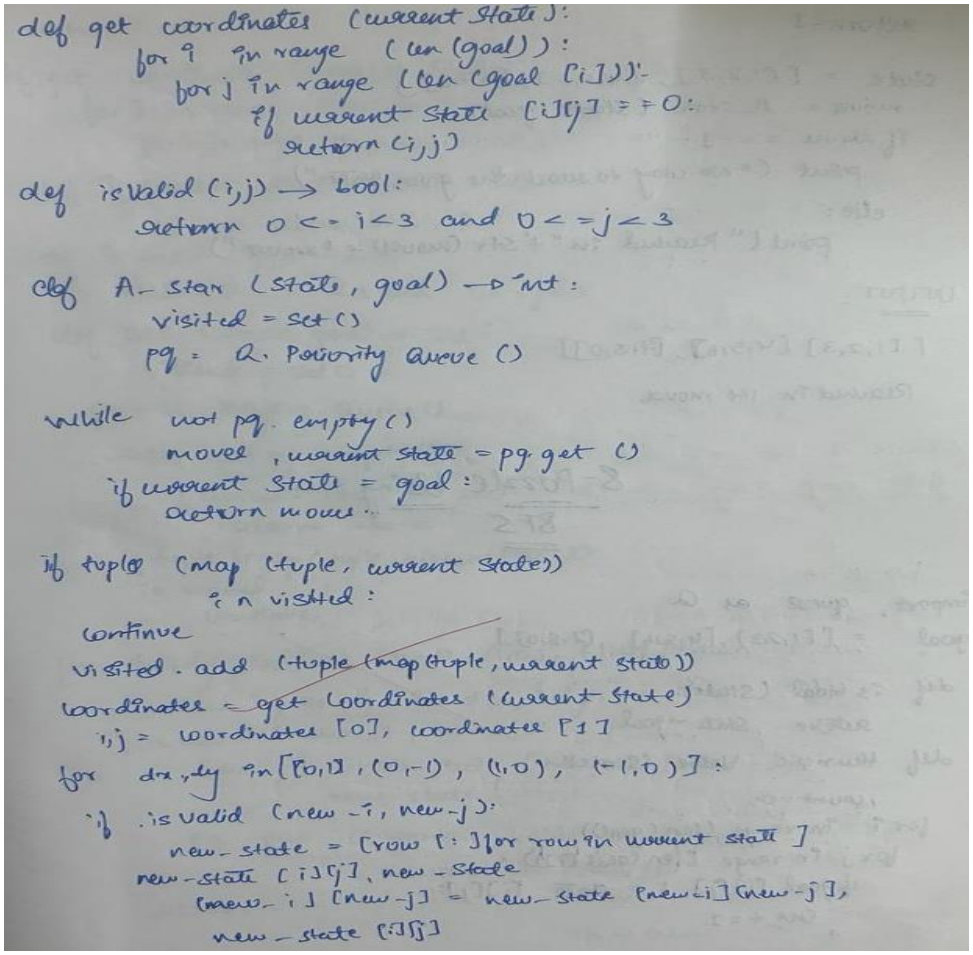
```
    return None
def possible_moves (state, visited_states):
    b = state.index(0)
    d = []
if b not in [0,1,2]:
    d.append('u')
if b not in [6,7,8]:
    d.append('d')
if b not in [2,5,8]:
    d.append('r')
pos_moves_it_can = []
for i in d:
    pos_moves_it_can.append(gen(state,i,b))
return [move_it_can for move_it_can in pos_moves_
    it_can if move_it_can not in visited_states]
def gen(state,m,b):
    temp = state.copy()
    if m=='d':
    temp[b+3], temp[b] = temp[b], temp[b+3]
    elif m=='u'
    temp[b-3], temp[b] = temp[b] temp[b+3]
    def print_state (state):
print (f"{ state[0]} { state[1]}
        { state[2]} \n { state[3]}
        { state[4]} { state[5]} \n state[6]}
        { state[7]}, \n")
src = [1,2,3,0,4,5,6,7,8]
target = [1,2,3,4,5,0,6,7,8]
```


output:
0 2 3
1 4 5
6 7 8
1 2 3
6 4 5
0 7 8
1 2 3
6 4 5
7 0 8
1 2 3
4 0 5
6 7 8
0 2 3
1 4 5
6 7 8
2 0 3
1 4 5
6 7 8
1 2 3
6 4 5
0 7 8
1 2 3
4 0 5
6 7 8
1 0 3
4 2 5
6 7 8
1 2 3
4 7 5
6 0 8
1 2 3
4 5 0
6 7 8
1 2 3
4 5 0
6 7 8
success

4. Implement A* search algorithm.



```
8- Puzzle Using A- Star

import queue as Q
goal = [[1,2,3], [4,5,6], [7,8,0]]

def isGoal(state):
    return state == goal

def Heuristic Value(state):
    cnt = 0
    for i in range(len(goal)):
        for j in range(len(goal)[i]):
            if goal[i][j] != state[i][j]:
                cnt += 1
    return cnt.
```



```
def get coordinates (current state):
    for i in range (len (goal)):
        for j in range (len (goal [i])):
            if current state [i][j] == 0:
                return (i,j)

def isValid (i,j) -> bool:
    return 0 <= i < 3 and 0 <= j < 3

def A- star (state, goal) -> int:
    visited = set()
    pq = Q. Poriority Queue ()

while not pq. empty()
    moves, current state = pq. get ()
    if current state = goal:
        return moves.

if tuple (map (tuple, current state))
        in visited:
    continue
visited. add (tuple (map (tuple, current state)))
coordinates = get coordinates (current state)
i,j = coordinates [0], coordinates [1]
for dx, dy in [(0,1), (0,-1), (1,0), (-1,0)]:
    if .is valid (new-i, new-j):
        new- state = [row [:] for row in current state]
    new-state [i][j], new-state
        [new-i] [new-j] = new- state [new-i] [new-j],
        new- state [i][j]
```

```
return -1

state = [(1,2,3], [4,0,5], (6,7,8])
  moves = A_star (state, goal)
  if moves == -1:
    print ("No way to reach the given state")
  else:
    print ("Reached in" + str (moves) + "moves")
```

OUTPUT:

```
[ [1,2,3] [4,5,6], [7,8,0]]

Reached in 14 moves.
```

5. Implement vacuum cleaner agent.

Vacuum Cleaner Agent

```
def vacuum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter location of Vaccum")
    status_input = input("Enter status of" + location_input)
    status_input_complement = input("Enter Status of other
                                     room")

    print("Initial location = input("enter "condition" + str
          (goal_state))

    if location_input == 'A':
        print("Vaccum is placed in location A")
        if status_input == '1':
            print("Location A is Dirty.")
            goal = state['A'] = '0'
            cost += 1
        print("cost for CLEANING A" + str(cost))
        print("Location A has been cleaned")

    if status_input_complement == '1':
        print("Location B is Dirty")
        print("Moving right to the Loca B")
        cost += 1
        print("cost for moving RIGHT" + str(cost))
        goal_state['B'] = '0'
        cost += 1
        print("COST for SUCK" + str(cost))
        print("LOC B has been cleaned.")

    else:
        print("No action" + str(cost))
        print("LOC B is already clean.")
```

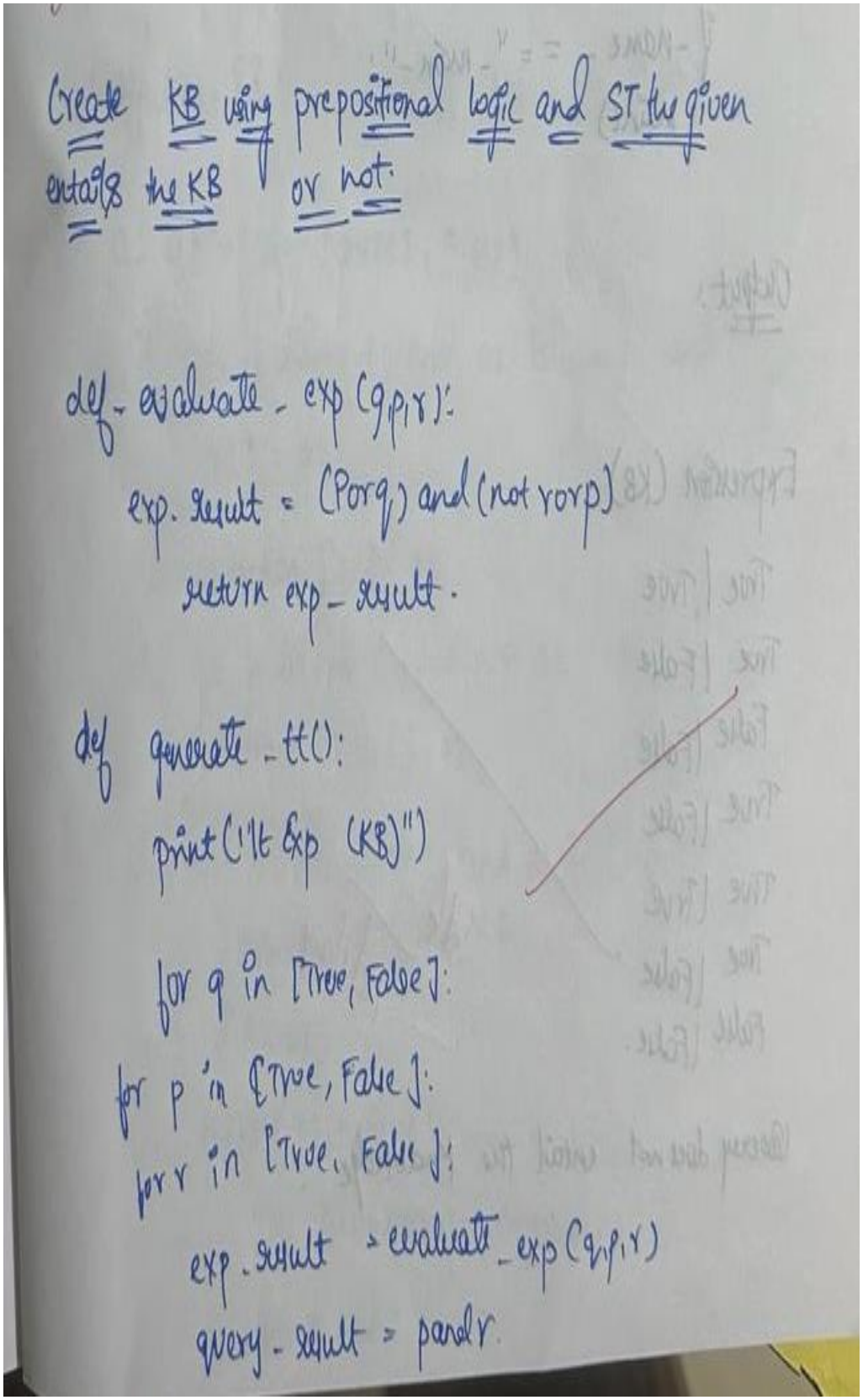
```
if Status_input == '0':
    print("LOC A is already clean.")
    if Status_input_complement == '1':
        Print("LOC B is Dirty.")
        print("Moving RIGHT to the LOC B".)
        cost += 1
    print("cost for SUCK" + str((cost))
    print("LOC B has been cleaned.")
else:
    print("No action" F str((cost))
    print(cost)
    print("LOC B is already clean").

print("COST for SUCK" + str(cost))
print("LOC A has been cleaned:")
else:
    print(cost)
    print("LOC B is already clean:")
    if Status_input - complement == '1':
        print("LOC A is Dirty.")
        print("Moving LEFT to the LOC A".)
        cost += 1
        print("COST for SUCK" + str((cost))
        print("LOC A has been cleaned.")
    else:
        print("NO action" + str((cost))
        print("LOC A is already clean:")
print("GOAL. STATE:")
print(goal-State)
print("Performing Measurement" +str(cost))
                                    Vaccum_world()
```

Output for Vaccum :-

Enter loc of Vaccum a
Enter status of a 1
Enter Status of other room 0
Intial location Condition {'A': '0', 'B': '0'}
Vacuum is placed in location B
LOCATION B is Dirty.
COST for Cleaning 1

Location B has been Cleaned
GOAL STATE:
{'A': '0', 'B': '0'}

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

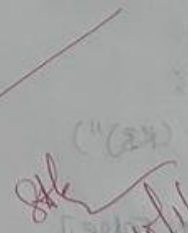


```
def query_entails_knowledge():
    for q in [True, False]:
    for p in [" ", False]:
        exp_result = evaluate_exp(q,p,r)
        query_result = p & r;
        return False.

return True
def main()
    generate_truth_table()
        print("In Query entails with knowledge"):

else
    if __name__ == "__main__":
    main()
```

Output:

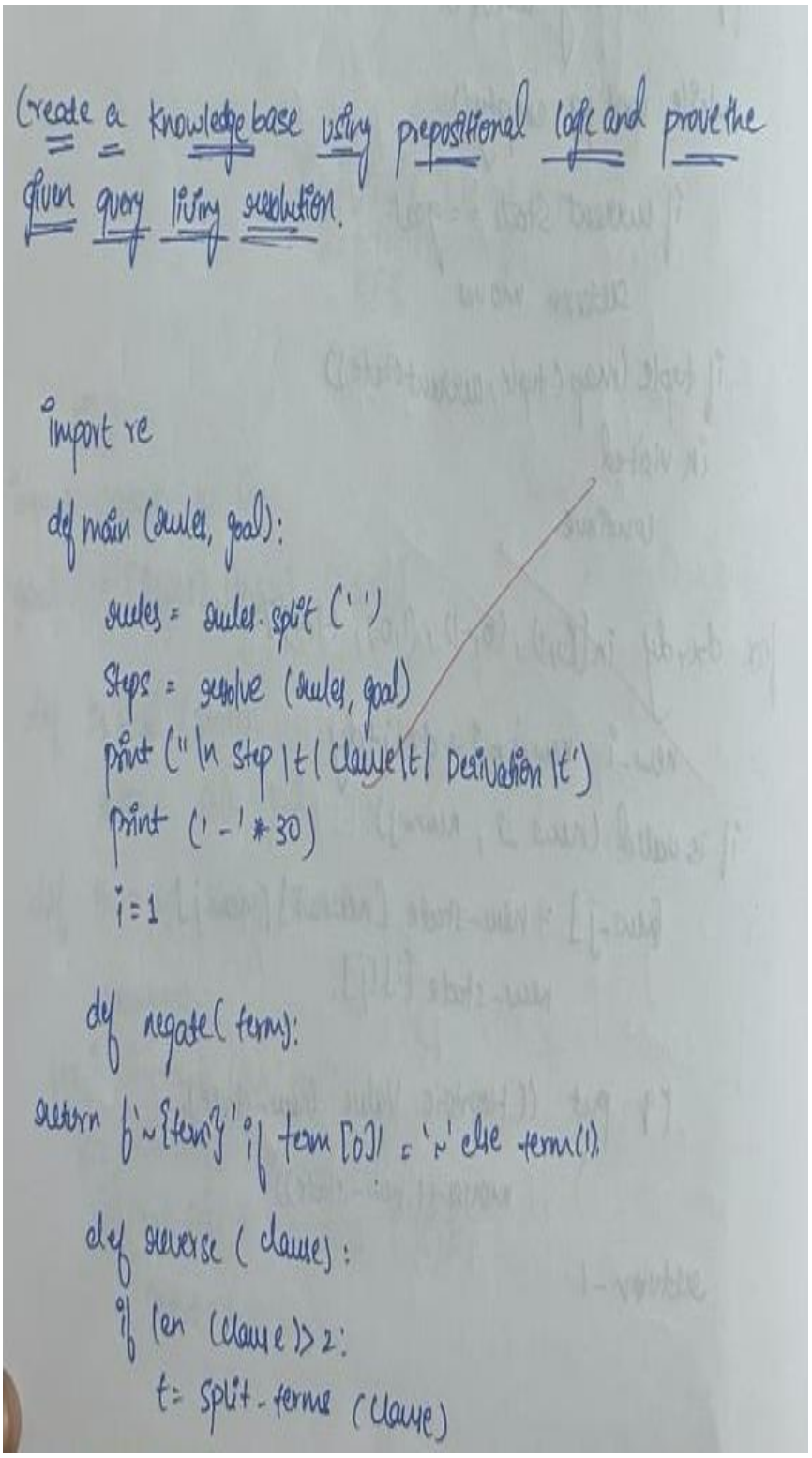
Expression (KB)

True | True
True | False
False | False
True | False
True | True
True | False
False | False.



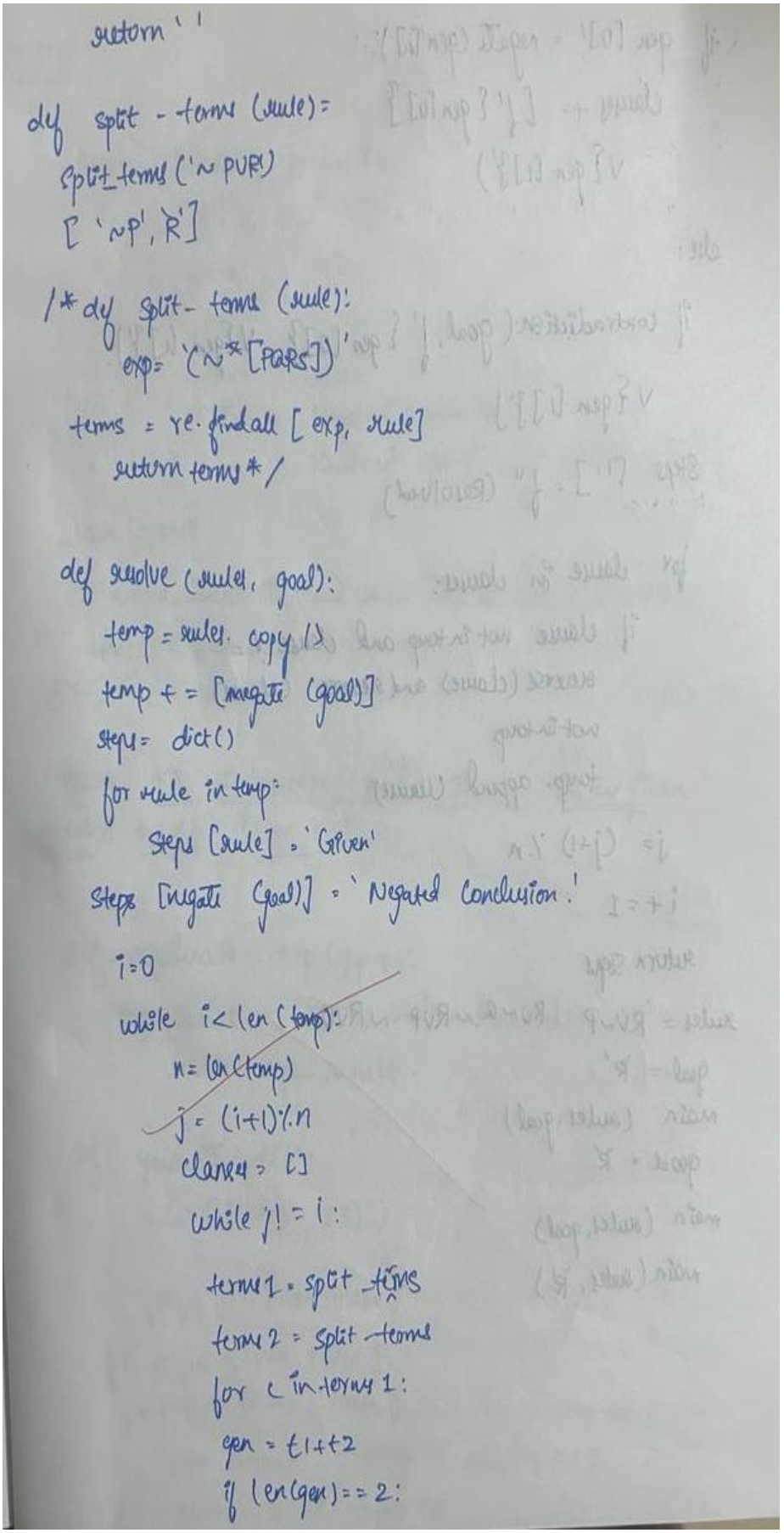
Query does not entail the knowledge.

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



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

```
import re
def main(rules, goal):
    rules = rules.split(' ')
    steps = resolve(rules, goal)
    print("\nStep\t|Clause\t|Derivation\t")
    print("-"*30)
    i=1

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

def reverse(clause):
    if len(clause)>2:
        t= split_terms(clause)
```



```
    return ''

def split_terms(rule):
    split_terms('~PUR')
    ['~p', 'R']

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

def resolve(rules, goal):
    temp = rules.copy()
    temp += [negate(goal)]
    steps = dict()
    for rule in temp:
        steps[rule] = 'Given'
    steps[negate(goal)] = 'Negated Conclusion.'
    i=0
    while i<len(temp):
        n=len(temp)
        j=(i+1)%n
        clauses=[]
        while j!=i:
            terms1=split_terms
            terms2=split_terms
            for c in terms1:
            gen=t1+t2
            if(len(gen)==2:
```

```
if gen[0]! = negate(gen[1]):
    clauses += [f'{gen[0]}
        V{gen[1]}')

else:
    if contradiction(goal, f'{gen[0]}, V{gen[1]}'):
        V{gen[1]}']
    steps['1'] = f" (Resolved)

    for clause in clauses:
        if clause not in temp and clause[::-1] =
            reverse(clause) and reverse(clause)
            not in temp
            temp.append(clause)
    j = (j+1) % n
    i += 1
    return steps
rules = 'RV~P  RV~Q  ~RVP  ~RVQ
goal = 'R'
main(rules, goal)
goal = 'R'
main(rules, goal)
main(rules, 'R')
```

Output:-

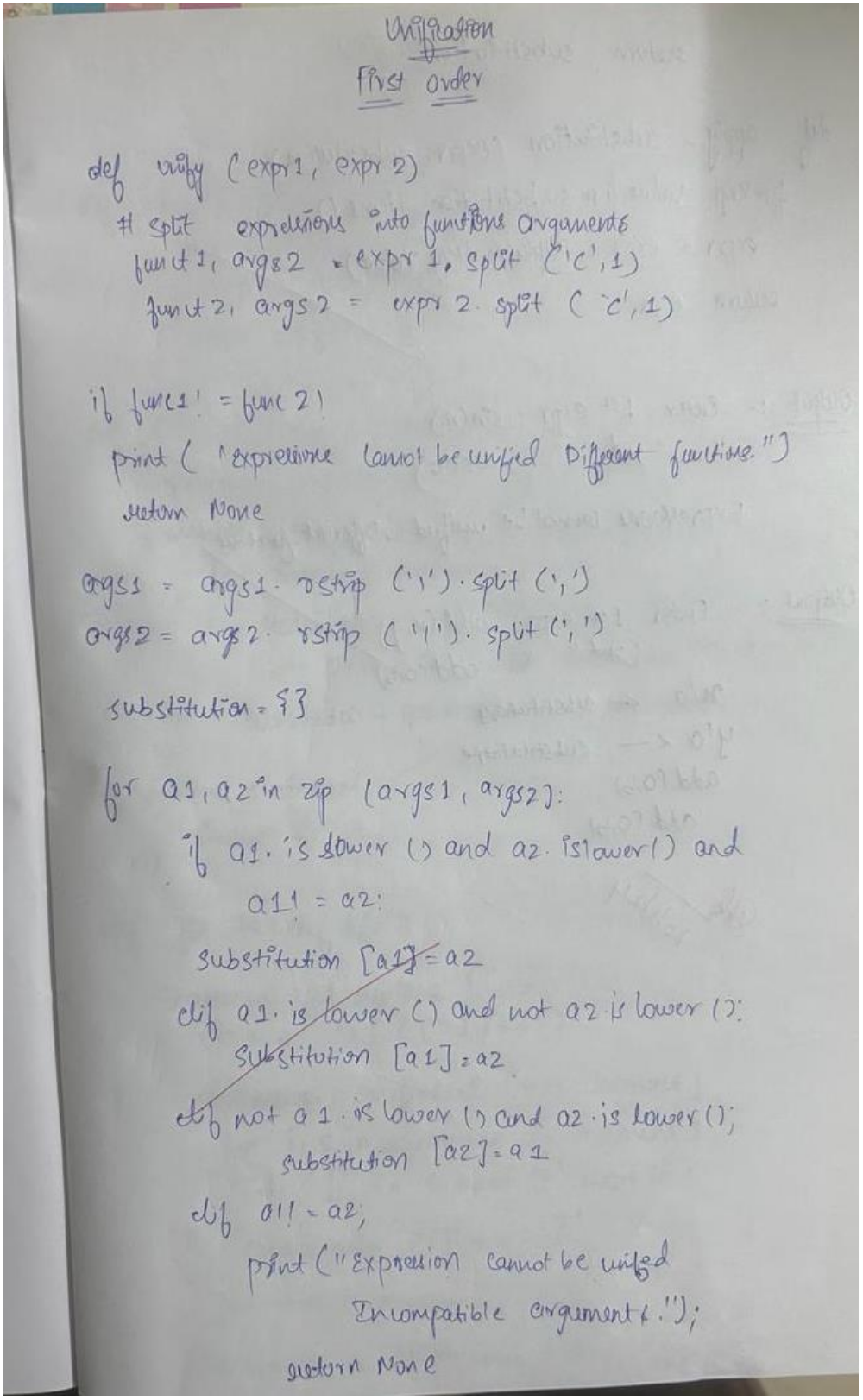
Step	Clause	Derivation
1	RVP	Given
2	RV~Q	Given
3	~RVP	Given
4	~RVQ	Given
5	~R	Negated conclusion
6		Resolved RVP and ~RVP to RVP,

which is NULL

A contradiction is found when ~R is assumed as true.

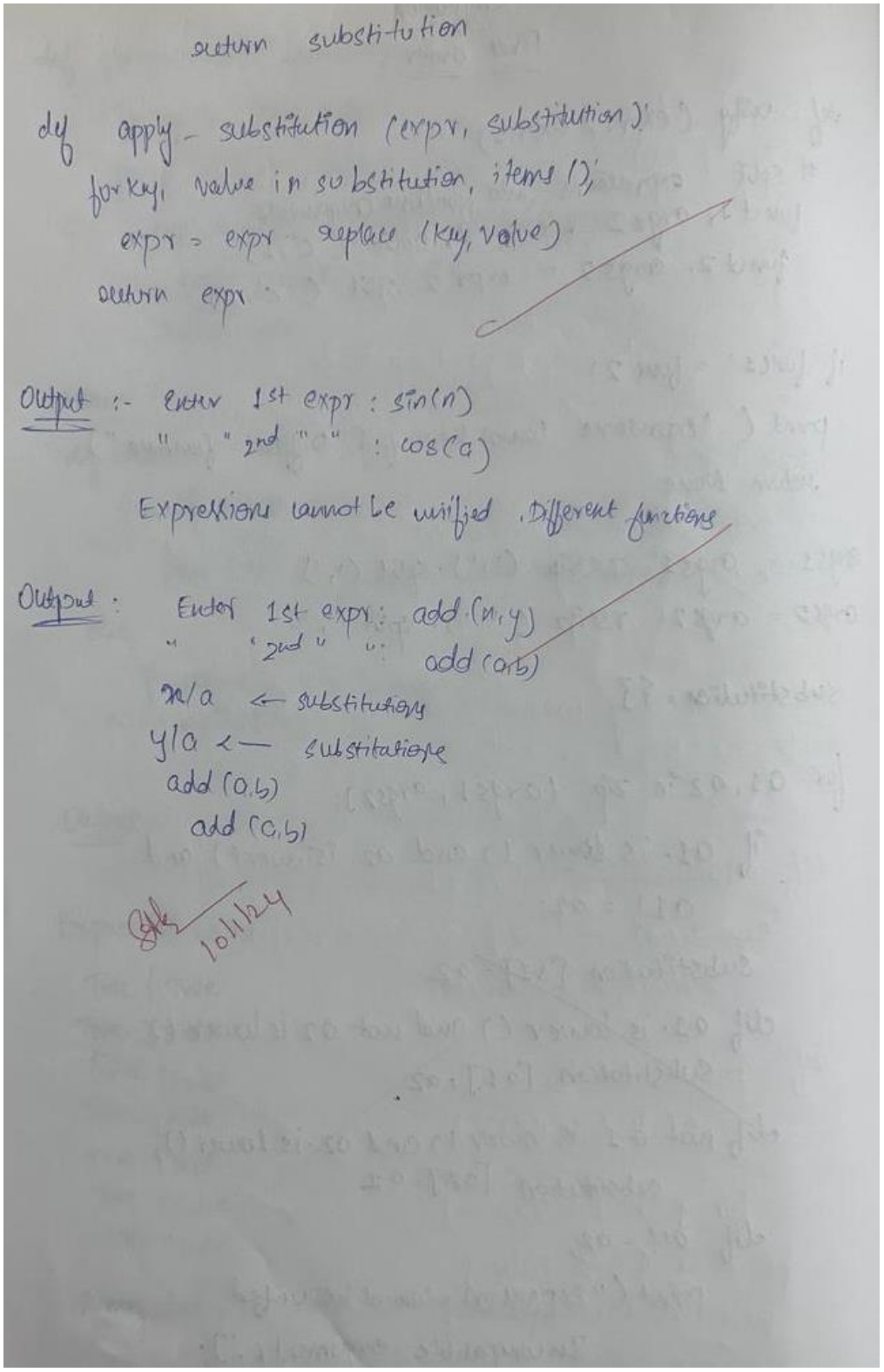
Hence, R is true.

8. Implement unification in first order logic



Unification
First order

```
def unify (expr1, expr2)
  # split expressions into functions arguments
    funct1, args2 = expr1. split ('(', 1)
    funct2, args2 = expr2. split ('(', 1)

if func1 != func2:
  print ("Expressions cannot be unified Different functions.")
  return None

args1 = args1. rstrip (')'). split (',')
args2 = args2. rstrip (')'). split (',')

substitution = {}

for a1, a2 in zip (args1, args2):
    if a1. islower () and a2. islower () and
       a1 != a2:
      substitution [a1] = a2
    elif a1. islower () and not a2 is lower ():
      substitution [a1] = a2
    elif not a1. is lower () and a2. is lower ():
      substitution [a2] = a1
    elif a1 != a2:
      print ("Expression cannot be unified
             Incompatible arguments.");
      return None
```



```
return substitution

def apply_substitution(expr, substitution):
    for key, value in substitution.items():
        expr = expr.replace(key, value)
    return expr
```

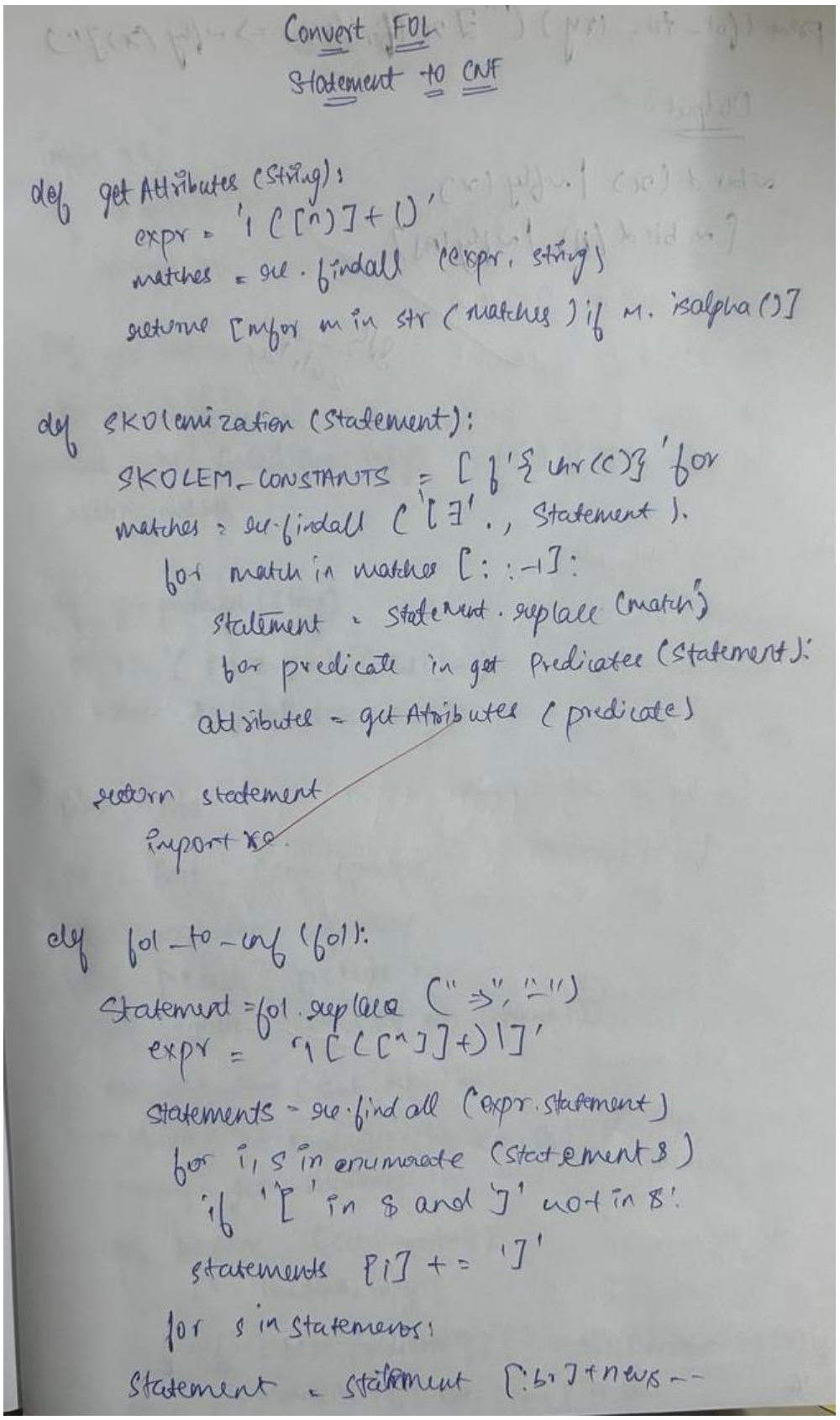
Output :- Enter 1st expr : sin(n)
" " 2nd " " : cos(a)

Expressions cannot be unified. Different functions

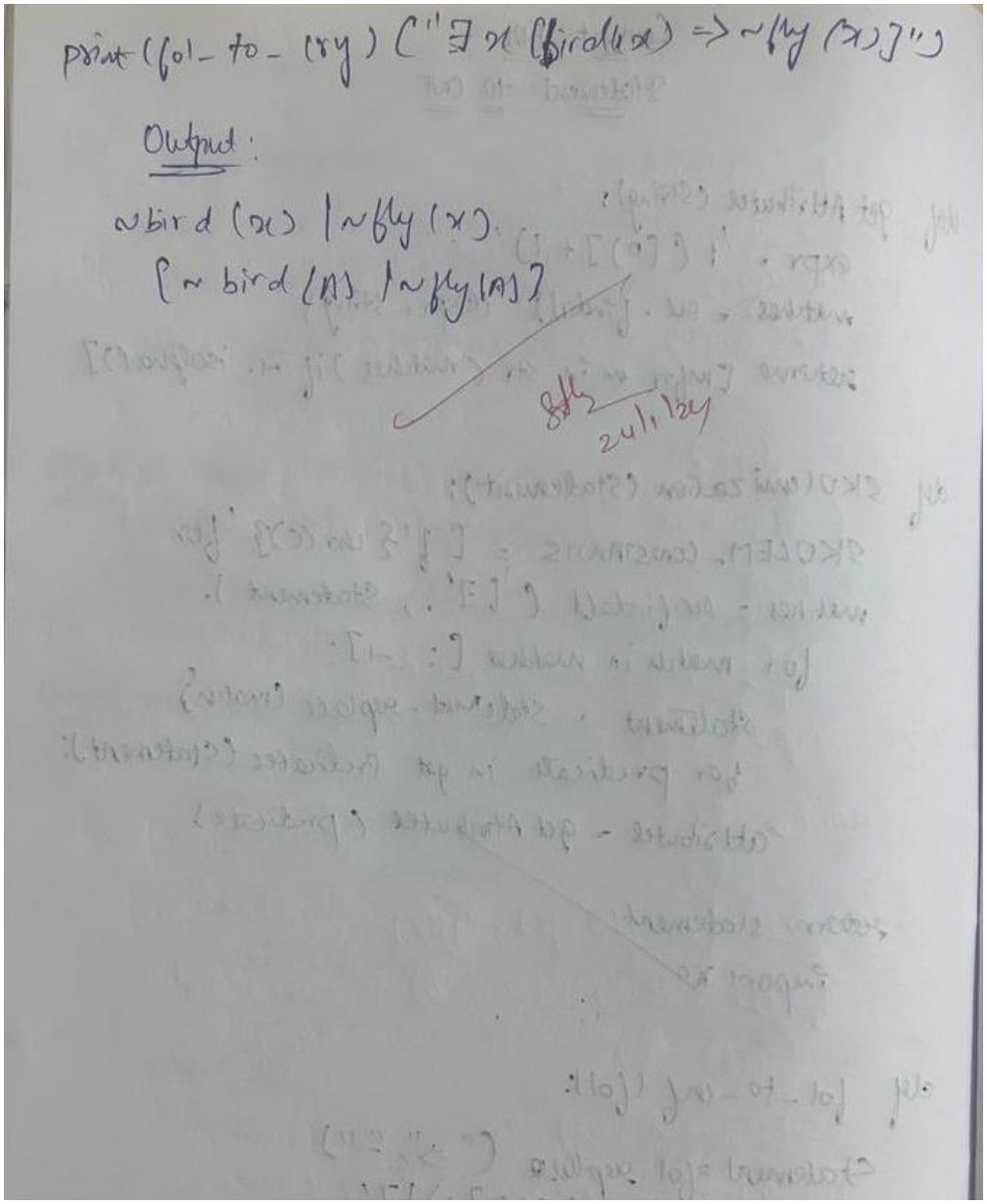
Output : Enter 1st expr : add(x, y)
" 2nd " " : add(a, b)
x/a ← substitutions
y/a ← substitutions
add(a, b)
add(a, b)

10/11/24

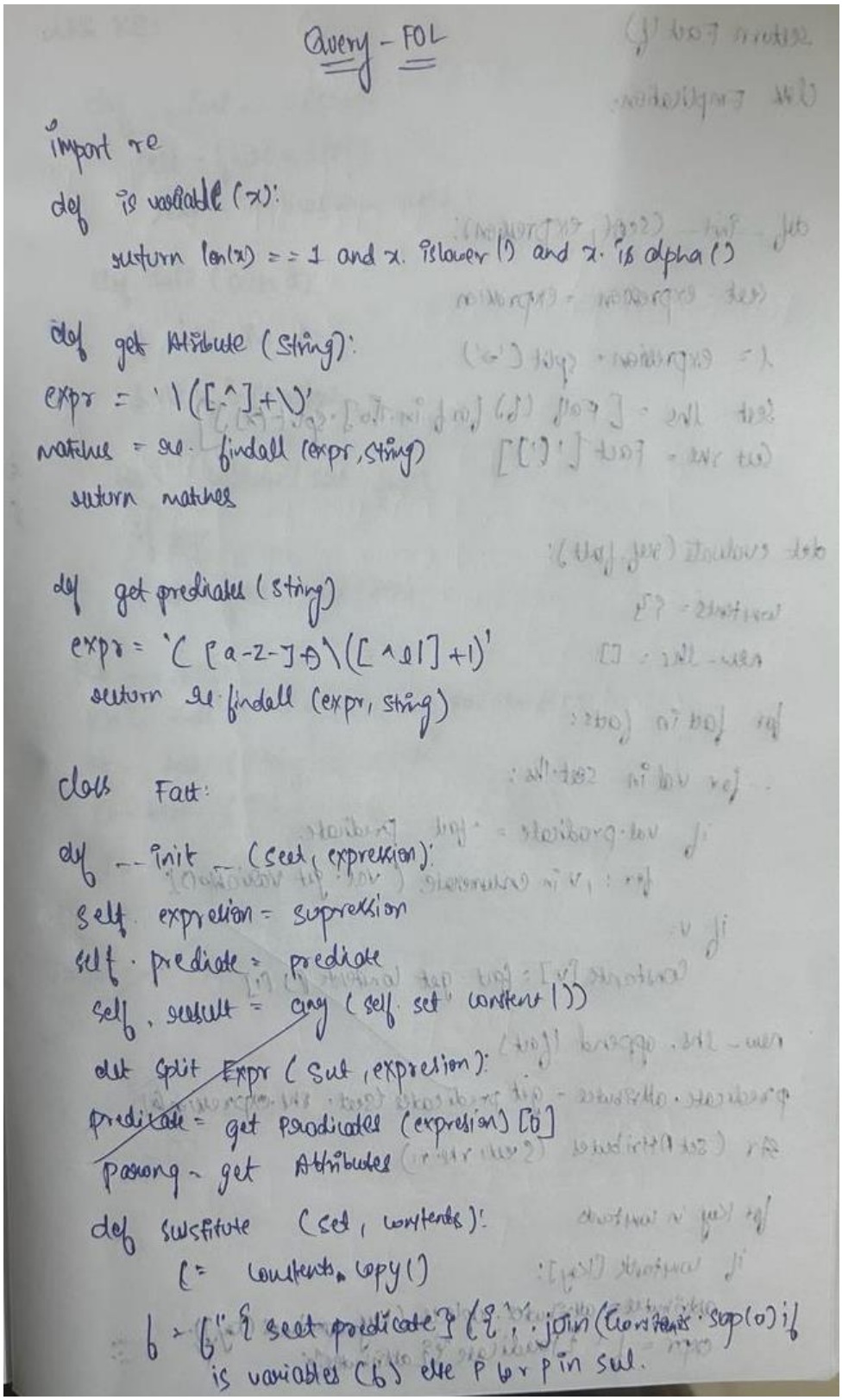
9. Convert a given first order logic statement into Conjunctive Normal Form (CNF).



```
Convert FOL
Statement to CNF

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

def SKOLemization(Statement):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for
    matches = re.findall('[∃'., Statement).
        for match in matches[::-1]:
            statement = statement.replace(match)
            for predicate in getPredicates(statement):
            attributes = getAttributes(predicate)

    return statement
    import re

def fol_to_cnf(fol):
    Statement = fol.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[:-1] + news--
```

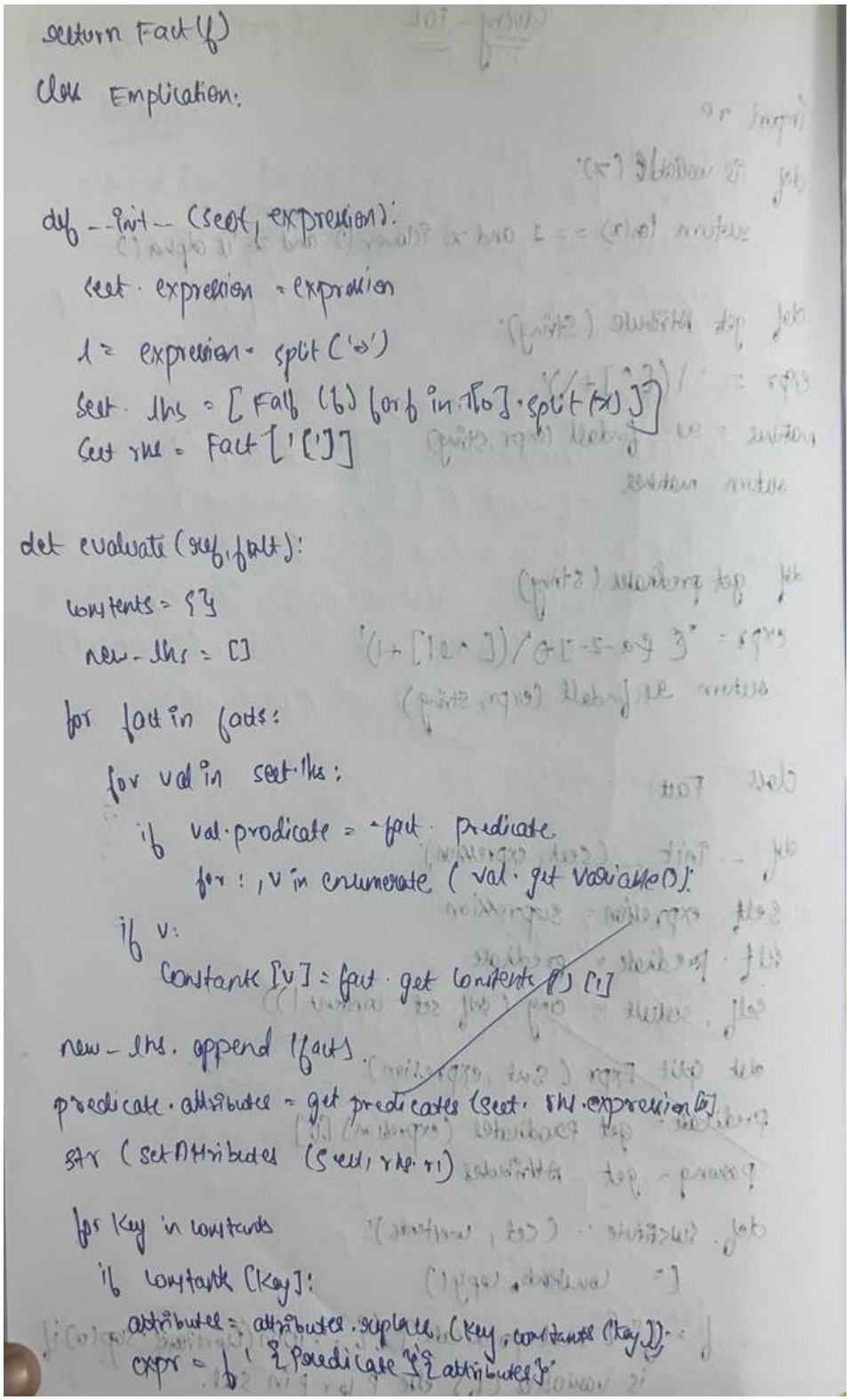

print(fol_to_cnf("∃x(bird(x) => ~fly(x))"))
Output:
~bird(x) | ~fly(x).
[~bird(A) | ~fly(A)]

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

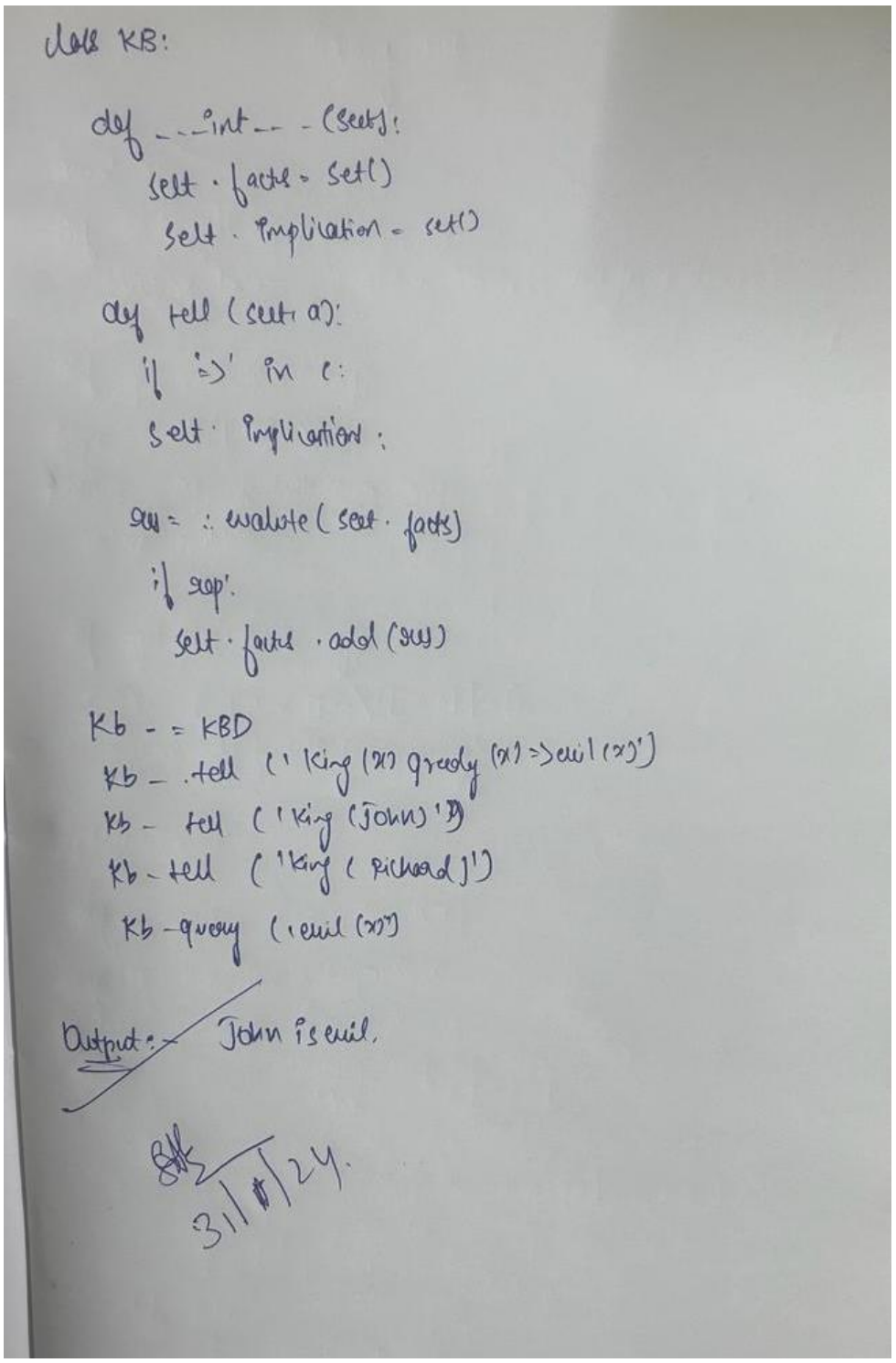


Query - FOL

```
import re
def is_variable(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
 self.predicate = predicate
 self.result = any(self.getContent())

def splitExpr(self, expression):
predicate = getPredicates(expression)[0]
 params = getAttributes
def substitute(self, constants):
    c = constants.copy()
 f = f"{self.predicate}({','.join(constants.pop(0) if
    is_variable(p) else p for p in self.
```



```
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.getContents()[i]
        new_lhs.append(fact)
    predicate, attributes = getPredicates(self.rhs.expression)
    for (self.getAttributes(self.rhs.expression))
    for key in constants:
        if constants[key]:
            attributes = attributes.replace(key, constants[key])
    expr = predicate + "(" + attributes + ")"
```



```
class KB:

    def __init__ (self):
        self.facts = set()
        self.implication = set()

    def tell (self, e):
        if '=>' in e:
            self.implication.add(e)
        else:
            self.facts.add(e)

    def query (self, e):
        fact = self.evaluate(e, self.facts)
        if fact:
            print(fact)

kb = KB()
kb.tell('king(x) greedy(x) => evil(x)')
kb.tell('king(John)')
kb.tell('greedy(John)')
kb.tell('king(Richard)')
kb.query('evil(x)')
```

Output:- John is evil.

31/8/24.