B.M.S. COLLEGE OF ENGINEERING BENGALURU
Autonomous Institute, Affiliated to VTU

Lab Record

Artificial Intelligence

Submitted in partial fulfillment for the 6th Semester Laboratory

Bachelor of Technology
in
Computer Science and Engineering

Submitted by:

Gauri Ramabhadran
1BM21CS066

Department of Computer Science and Engineering
B.M.S. College of Engineering
Bull Temple Road, Basavanagudi, Bangalore 560 019
Nov- Mar 2024

B.M.S. COLLEGE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that the Artificial Intelligence (20CS5PCAIP) laboratory has been carried out by Gauri Ramabhadran (1BM21CS066) during the 5th Semester November-March- 2024.

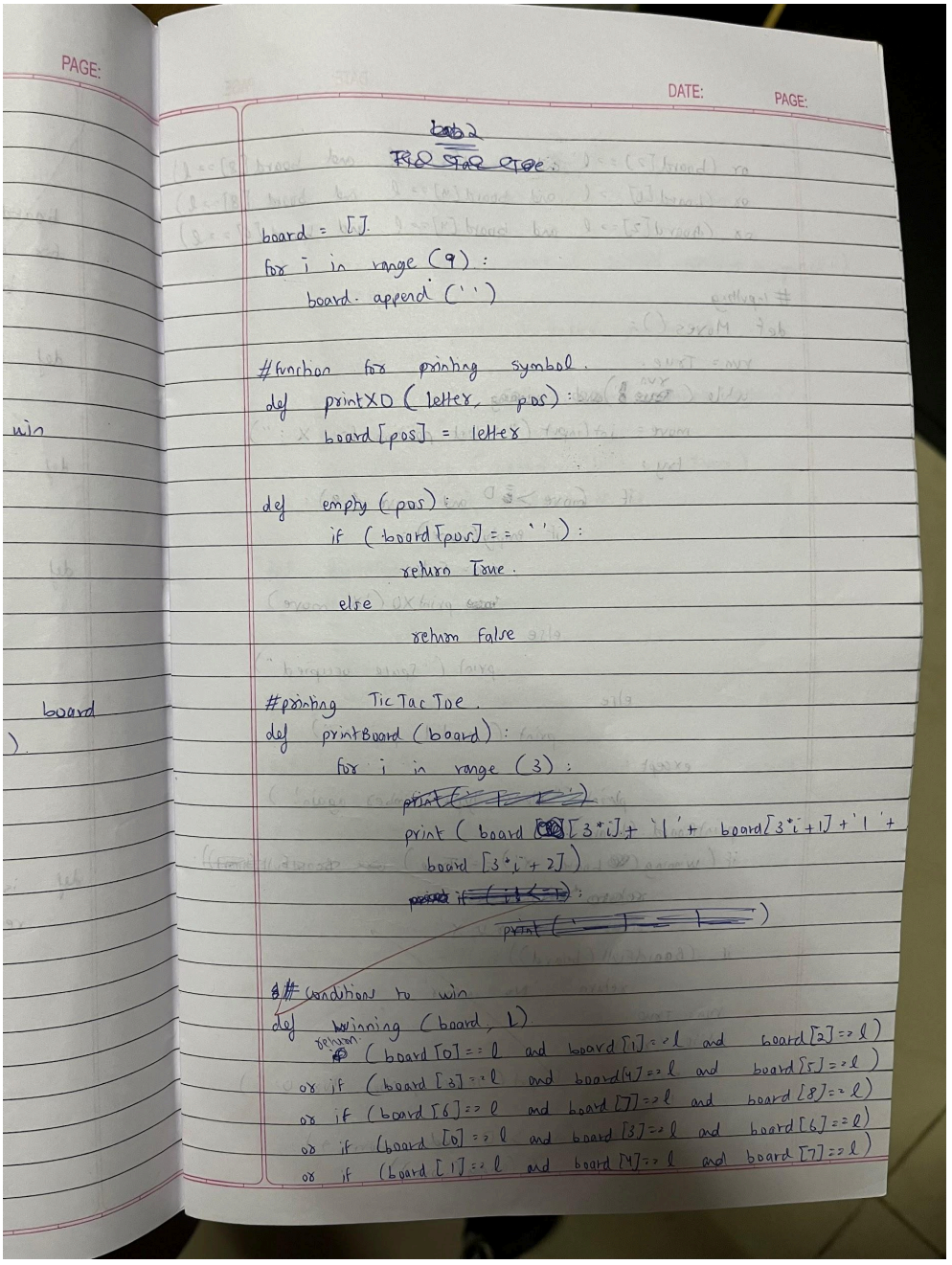
Signature of the Faculty Incharge:

Prof. Swathi Sridharan

Assistant Professor
Department of Computer Science and Engineering
B.M.S. College of Engineering, Bangalore

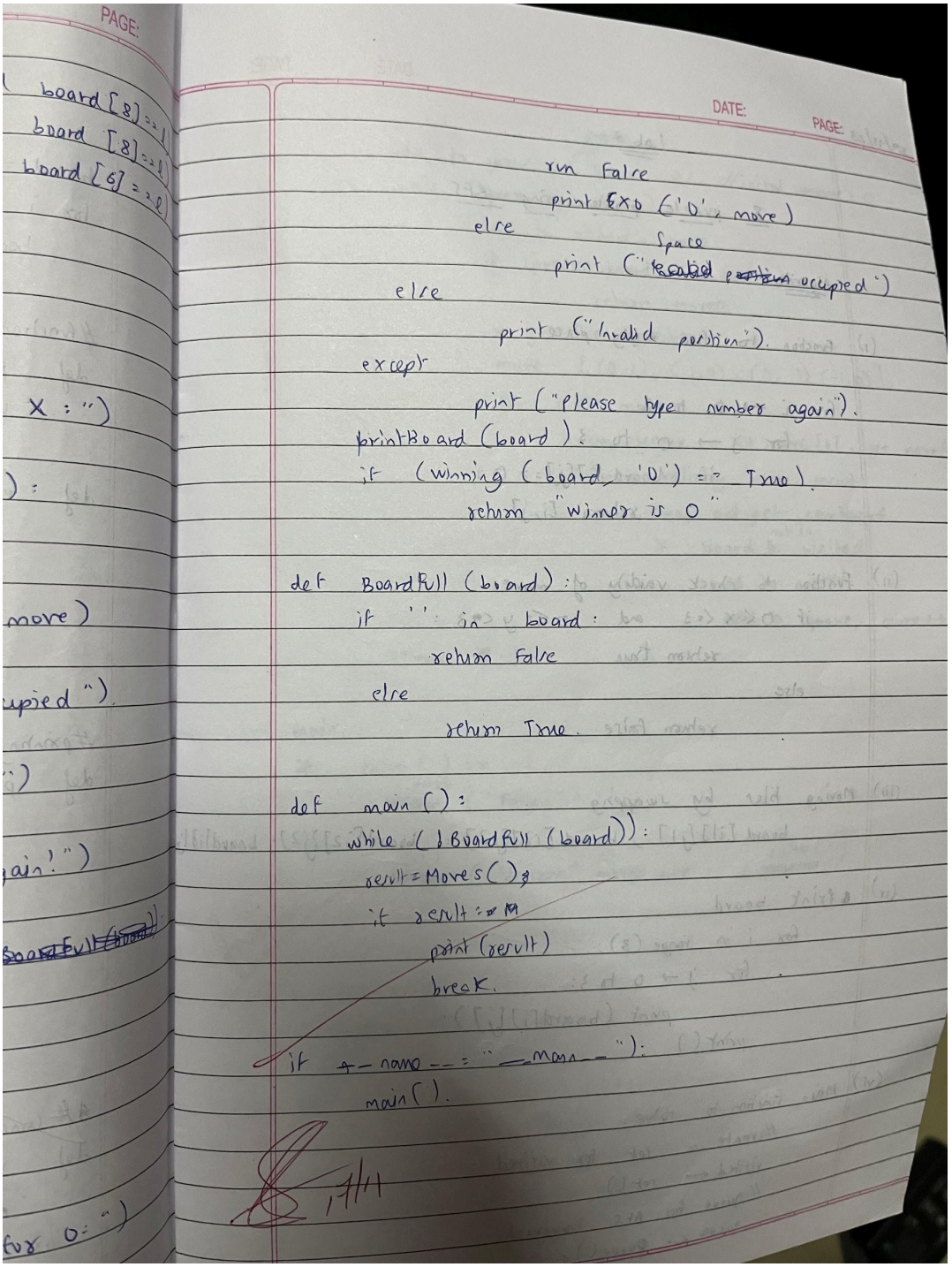
Table of Contents

OBSERVATION



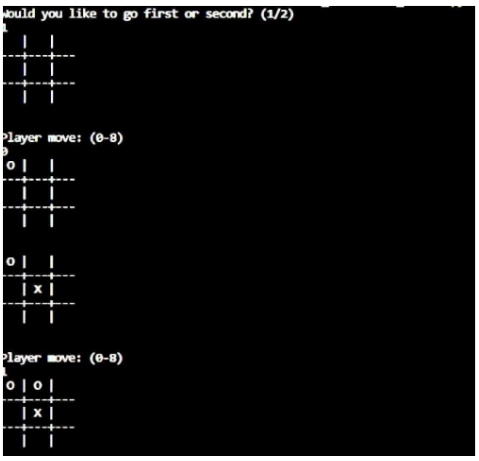
```
board = []
for i in range (9):
    board.append (' ')

#function for printing symbol.
def printXO (letter, pos):
    board[pos] = letter

def empty (pos):
    if (board[pos] == ' '):
        return True.
    else
        return False

#printing TicTacToe.
def printBoard (board):
    for i in range (3):
        print ( board[3*i] + '|' + board[3*i+1] + '|' +
        board [3*i+2])

##conditions to win
def winning (board, l)
    return
    if (board[0] == l and board[1] == l and board[2] == l)
    or if (board[3] == l and board[4] == l and board[5] == l)
    or if (board[6] == l and board[7] == l and board[8] == l)
    or if (board[0] == l and board[3] == l and board[6] == l)
    or if (board[1] == l and board[4] == l and board[7] == l)
```

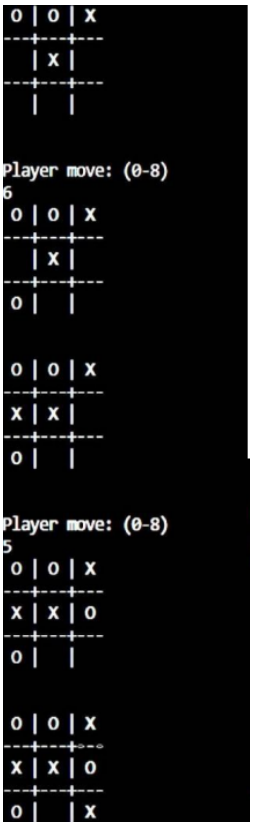
```
or (board[2] == l and board[5] == l and board[8]
or (board[0] == l and board[4] == l and board[8]
or (board[2] == l and board[4] == l and board[6]:

#inputting
def Moves():
    run = True.
    while (run).
        move = int(input("Select position for X:")
        try:
            if (move >= 0 and move < 9):
                if empty(move):
                    run = False.
                    printXO('X', move)
                else
                    print("Space occupied")
            else
                print("Invalid position")
        except:
            print("Please type number again!")
printBoard.(board)
if (winning(board, 'X') == True)
    return "Winner is X"
if (BoardFull(board)):
    return "No winner"
run: True.
while (run).
    move = int(input("Select position for O:")
    try:
        if (move >= 0 and move < 9):
            if (empty(move))
```

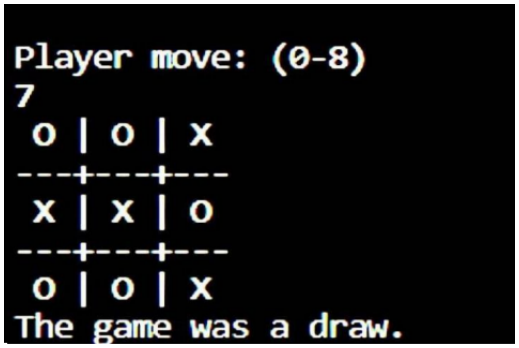


```
PAGE:
( board[8] == '
  board[8] == '
  board[6] == '

  X :")

  ):

  move)

  upied")

  :")

  ain!")

  BoardFull(board)):

  for O:")
```

```
DATE:                PAGE:

                run False
                print (f"{'O'}, move)
            else                Space
                print ("Position occupied")
        else
            print ("Invalid position").
    except
        print ("please type number again").
printBoard (board).
if (winning (board, 'O') == True).
    return "winner is O"

def BoardFull (board):
    if '' in board:
        return False
    else
        return True.

def main ():
while (!BoardFull (board)):
    result = Moves();
    if result != M
        print (result)
        break.

if __name__ == "__main__").
    main().
```


Would you like to go first or second? (1/2)
1
Player move: (0-8)
0
Player move: (0-8)
1


Player move: (0-8)
6
Player move: (0-8)
5


Player move: (0-8)
7
The game was a draw.

Code:

```
from queue import Queue
# check for empty positions
def get_blank_position(board):
    for i in range(3):
        for j in range(3):
            if (board[i][j] == 0):
                return i, j

# moves must be within board.
def is_valid(x, y):
    return 0 <= x < 3 and 0 <= y < 3

# swapping empty.
def swap(board, x1, y1, x2, y2):
    board[x1][y1], board[x2][y2] = board[x2][y2], board[x1][y1]

# solving.
def solve(initial_state, goal_state):
    visited = set()
    queue = Queue()
    # initial moves = 0
    queue.put((initial_state, 0))
    while not queue.empty():
        current_state, move_count = queue.get()
        visited.add(tuple(map(tuple, current_state)))
        if current_state == goal_state:
            return move_count
        x, y = get_blank_position(current_state)
        moves = [(0,1), (1,0), (0,-1), (-1,0)]
```

```
        for move in moves:
            newx, newy = x + moves[0], y + moves[1]
            if is_valid(newx, newy):
                new_state = [row.copy() for row
                in current_state]
                swap(new_state, x, y, newx, newy)

                if tuple(map(tuple, new_state))
                not in visited:
                    queue.put((new_state, move+1))
                    visited.add(tuple(map(tuple,
                    new_state)))

    return -1

initial_state = []
for i in range(3):
    x = []
    for j in range(3):
        print("enter for pos", i, j)
        n = int(input())
        x.append(n)
    initial_state.append(x)

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

move_count = solve(initial_state, goal_state)
```

```
if move_count != -1:
    print("Minimum number of moves:", move_count)
else
    print("No solution found")
```

```
1 | 2 | 3
4 | 5 | 6
0 | 7 | 8

1 | 2 | 3
0 | 5 | 6
4 | 7 | 8

1 | 2 | 3
4 | 5 | 6
7 | 0 | 8

0 | 2 | 3
1 | 5 | 6
4 | 7 | 8

1 | 2 | 3
5 | 0 | 6
4 | 7 | 8

1 | 2 | 3
4 | 0 | 6
7 | 5 | 8

1 | 2 | 3
4 | 5 | 6
7 | 8 | 0
```

8/12/23

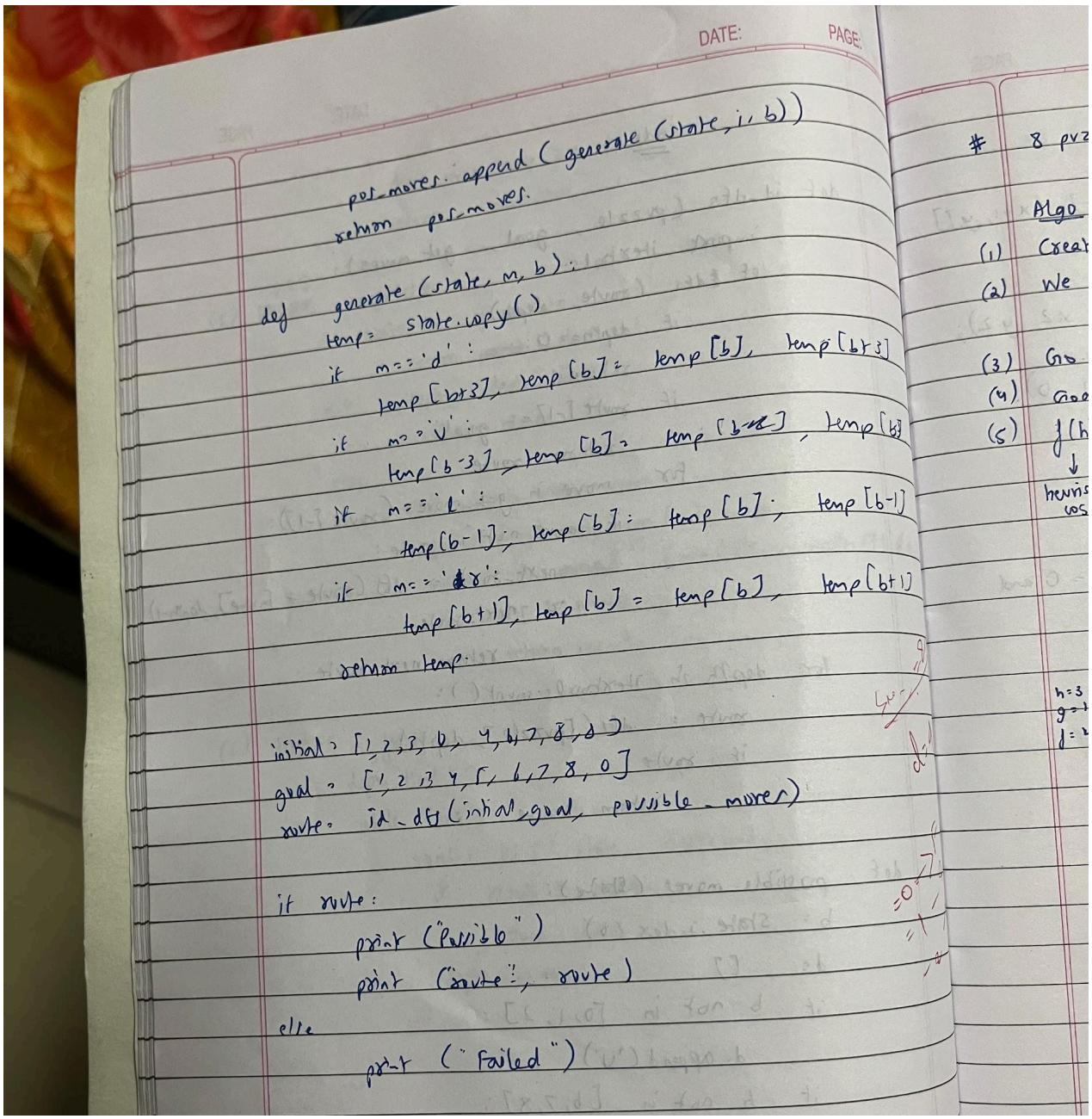
DATE: PAGE:

Code

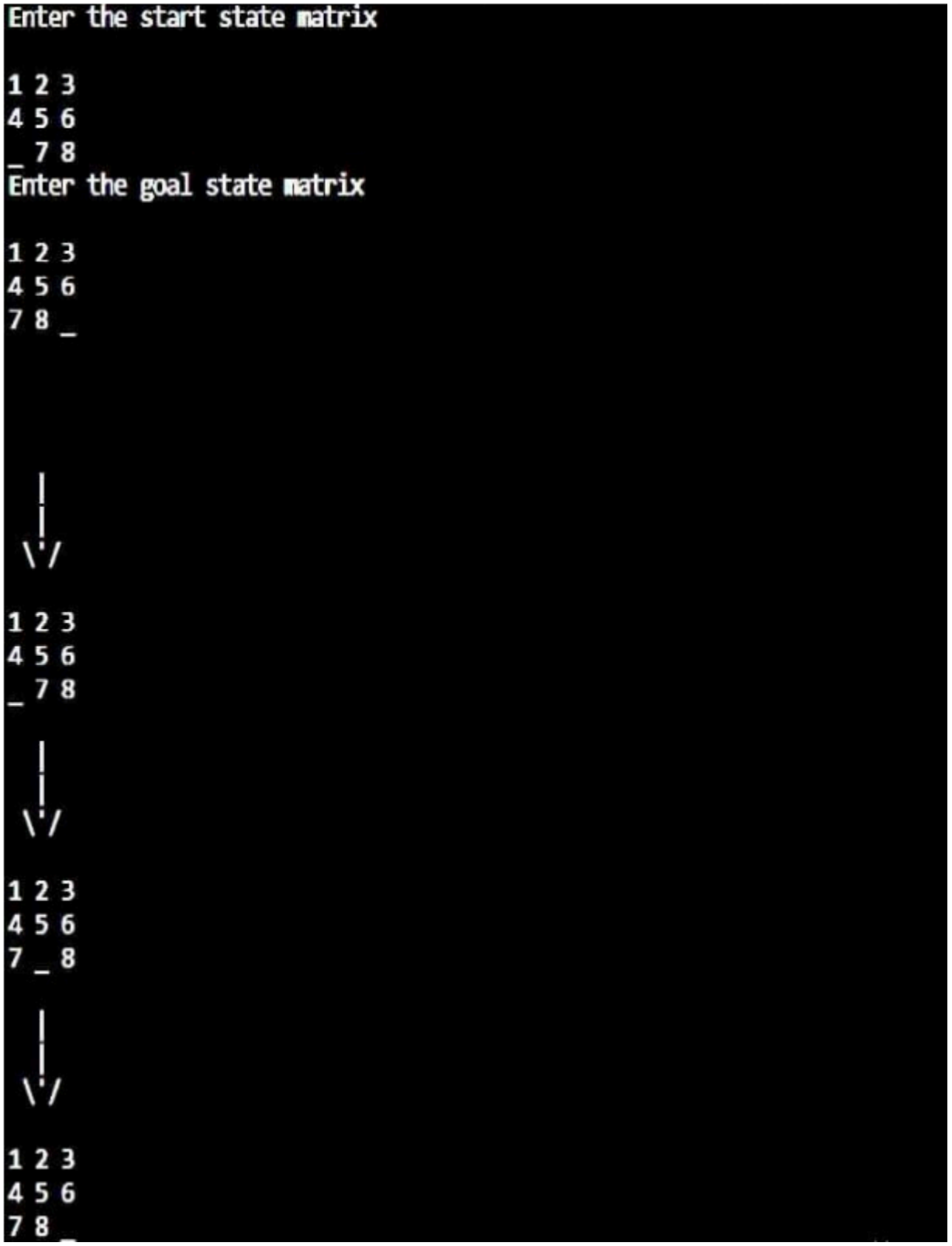
```
def id_dfs (puzzle, goal, get_moves):
    import itertools
    def dfs (route, depth):
        if depth == 0:
            return
        if route[-1] == goal:
            return route
        for move in get_moves (route[-1]):
            if move not in route:
                next_route = dfs (route + [move], depth-1)
                if next_route:
                    return next_route
    for depth in itertools.count ():
        route = dfs ([puzzle], depth)
        if route:
            return route
```

depth+1 depth_limit

```
def possible_moves (state):
    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 = []
    for i in d:
```

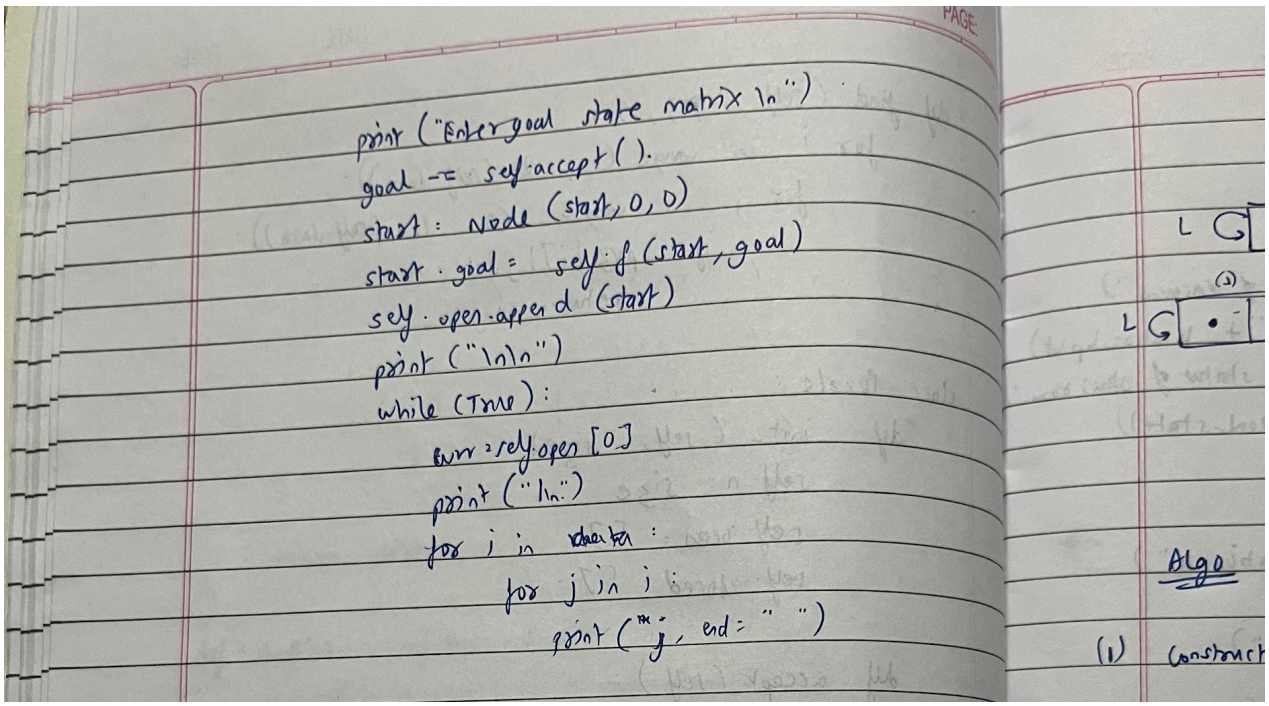


```
per_moves.append(generate(state, i, b))
return per_moves.

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

initial = [1,2,3,b,4,6,7,8,5]
goal = [1,2,3,4,5,6,7,8,0]
route. id_dfs(initial, goal, possible_moves)

if route:
    print("Possible")
    print("route:", route)
else
    print("Failed")
```


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

```
def generate_child (self):
    x, y = self.find (self.data, '-')
    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, x1, y1, x2, y2
        if child is not None:
            childNode = Node (child, self.level+1, 0)
            children.append (child_node).
    return children.

def shuffle. (self.key, x1, y1, x2, y2):
    if. x2 == 0 and x2 < len (self.data) and y2 == 0 and
    y2 < len (self.data):
        temp_puz = []
        temp_puz = self.copy (key).
        temp = temp_key [x2][y2]
        temp_puz [x2][y2] = temp_key [x1][y1].
        temp_puz [x1][y1] = temp
        return temp_puz
    else
        return None

def copy (self, root):
    temp = []
    for i in root:
        t = []
        for j in i:
            t.append (j).
        temp.append [t]
return temp.
```

```
def find (self , puz, x):
    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):
        self.n = size
        self.open = []
        self.closed = []

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

    def f(self, start, goal)
        return self.(start, data, goal) + start

def h (self, start, goal):
    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] !=
    return temp.        temp += 1

def process (self):
    print ("Enter the start state matrix \n");
    start = self.accept()
```

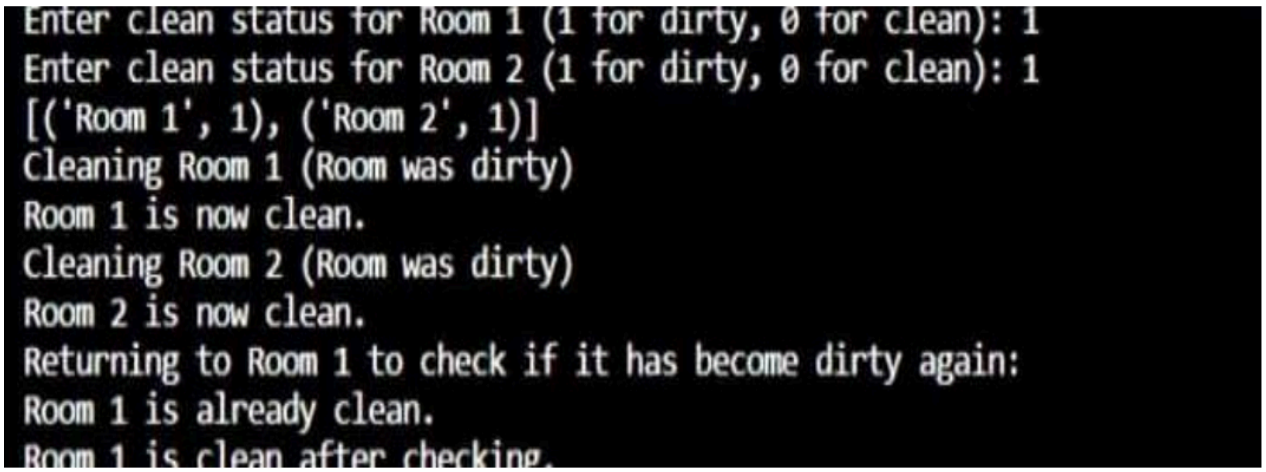


```
print ("Enter goal state matrix \n")
goal = self.accept().
start = Node (start, 0, 0)
start.goal = self.f (start, goal)
self.open.append (start)
print ("\n\n")
while (True):
    curr = self.open [0]
    print ("\n:")
    for i in :
        for j in i:
            print (j, end = " ")
```

```
Success!! It is possible to solve 8 Puzzle problem
Path: [[1, 2, 3, 0, 4, 6, 7, 5, 8], [1, 2, 3, 4, 0, 6, 7, 5, 8], [1, 2, 3, 4, 5, 6, 7, 0, 8], [1, 2, 3, 4, 5, 6, 7, 8, 0]]
```

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

    if location_input == 'A':
        print("Vacuum 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 location B.")
            cost += 1
            print("cost for moving right" + str(cost))
            goal_state['B'] = '0'
            cost += 1
            print("cost of Suck" + str(cost))
            print("location B has been cleaned")
    else
            print("no action" + str(cost))
            print("location B is already clean")

    if status_input == '0':
            print(location_A is clean")
            if (status_input_complement == '1")
```

```
        print ("location B is dirty")
        print ("Moving right to loc B").
        cost += 1
        print (cost for moving right" + str(cost))
        goal_state ['B'] = '0'
        cost += 1
        print ("cost for suck" + str(cost))
        print ("location B is cleaned");
    else
        print (cost)
        print ("Location B is cleaned")

else
    print ("Vacumis placed in location B")
    if (state_input == '1'):
        print ("location B is dirty").
        goal_state ['B'] = '0'
        cost += 1
        print ("cost for cleaning" + str (cost))
        print ("location B has been cleaned")
    else
        print (cost) :
print ("Goal state:")
print (goal_state)
print ("Performance measurement:" + str (cost))

vacuum_world ()
```


Enter clean status for Room 1 (1 for dirty, 0 for clean): 1
Enter clean status for Room 2 (1 for dirty, 0 for clean): 1
[('Room 1', 1), ('Room 2', 1)]
Cleaning Room 1 (Room was dirty)
Room 1 is now clean.
Cleaning Room 2 (Room was dirty)
Room 2 is now clean.
Returning to Room 1 to check if it has become dirty again:
Room 1 is already clean.
Room 1 is clean after checking.

DATE: PAGE:

Knowledge Base Entailment

Algorithm:

(i) creating Knowledge base function.

(i) define symbols: p = symbols ('p')
q = " ('q')
r = " ('r')

(2) define the KB. Kb = And (Implies (p, q),
Implies (q, r),
Not (r)
)

(ii) function to check entailment
entailment : satisfiable (And (Kb, Not (query))
return not entailment

Code

```
from sympy import symbols, And, Not, Implies, satisfiable.

def create_knowledge_base():
    p = symbols('p')
    q = symbols ('q')
    r = symbols ('r')
    KB = And (
        Implies (p,q),
        Implies (q,r),
        Not (r)
    )
```

```
    return knowledge_base.

def query_entails (knowledge_base, query):
    entailment = satisfiable (And (kb, Not (query)))
    return not entailment

if __name__ == '__main__':
    kb = create_knowledge_base()
    query = symbols ('p')
    result = query_entails (kb, query)
    print ("Knowledge Base : ", kb)
    print ("Query : ", query)
    print ("Query entails Knowledge Base : ", result)
```

OUTPUT

Knowledge Base : ~r & (Implies(p,q)) & (Implies(q,r))

Query : p

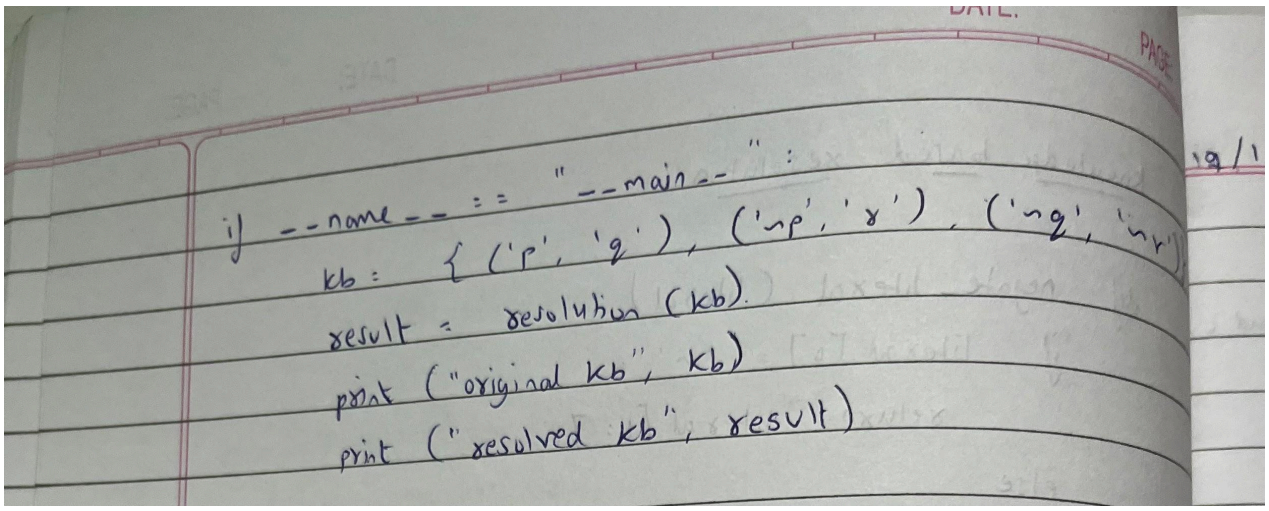
Query entails Knowledge Base : False

```
Knowledge Base: ~r & (Implies(p, q)) & (Implies(q, r))
Query: p
Query entails Knowledge Base: False
```

DATE: PAGE:

Knowledge based resolution

```
def negate_literal (literal)
    if literal[0] == '~' :
        return literal[1:]
    else
        '~' + literal.

def resolve (c1, c2) :
    resolved_clause = set(c1) | set(c2).
    for literal in c1 :
        if negate_literal.(literal) in c2 :
            resolved_clause = remove (literal)
            resolved_clause.remove (negate_literal(literal))
        return tuple (resolved_clause).

def resolution. (knowledge_base) :
    while True :
        new clauses= set ( )
        for i, c1 in enumerate (knowledge_base):
            for j, c2 in enumerate (knowledge_base):
                if i != j :
                    new_clause = resolve (c1, c2)
                if len (new_clause) >0 & new_clause not
                    : knowledge_base :
                    new_clause.add (new_clause).
    if not new_clause
        break
    knowledge_base != new_clause
    return knowledge_base.
```



Step	Clause	Derivation
1.	PvQ	Given.
2.	~PvR	Given.
3.	~QvR	Given.
4.	~R	Negated conclusion.
5.	QvR	Resolved from PvQ and ~PvR.
6.	PvR	Resolved from PvQ and ~QvR.
7.	~P	Resolved from ~PvR and ~R.
8.	~Q	Resolved from ~QvR and ~R.
9.	Q	Resolved from ~R and QvR.
10.	P	Resolved from ~R and PvR.
11.	R	Resolved from QvR and ~Q.
12.		Resolved R and ~R to Rv~R, which is in turn null.

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

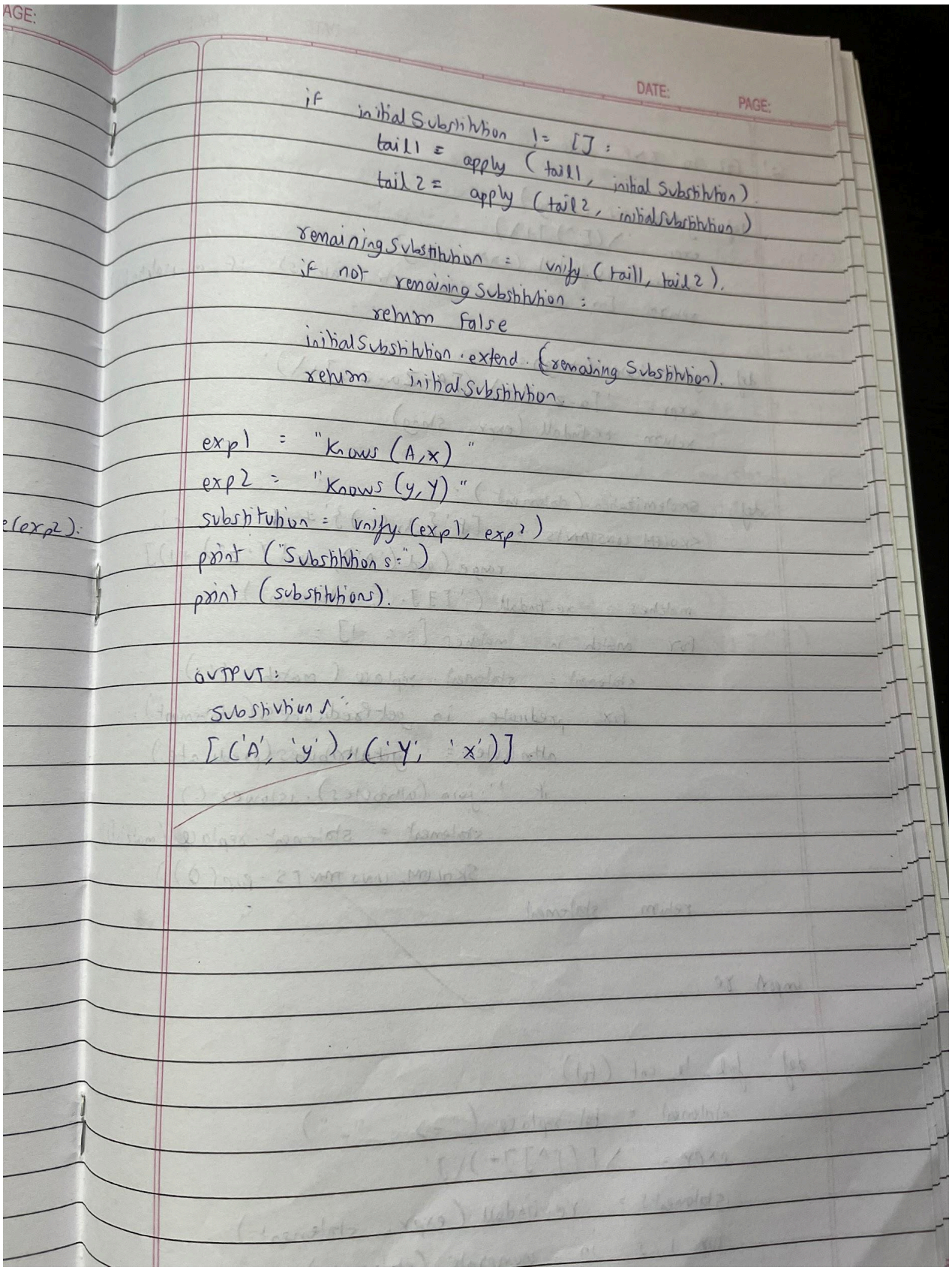
Unification Code

```
import re

def getAttributes (expression):
    expression = expression.split ("(")
    expression = "(".join (expression)
    expression = re.split ("?<!\(.),(?!.\)", expression)
    return expression

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)
    for index, val in enumerate (attributes):
        if val == old:
            attributes[index] = new
    predicate = getInitialPredicate (exp)
    return predicate + "(" + ",".join (attributes) + ")"

def apply (exp, substitutions):
    for substitution in substitutions
        new, old = substitution
        exp = replaceAttributes (exp, old, new)
```

DATE: PAGE:

```
        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
            return False

    if isConstant (exp1):
        return [(exp1, exp2)].

    if isConstant (exp2):
        return [(exp2, exp1)].
```

```
if isVariable(exp1):
    if checkOccurs(exp1, exp2):
        return False
    else
        return [(exp2, exp1)].

if isVariable(exp2):
    if checkOccurs(exp2, exp1):
        return False
    else
        return [(exp1, exp2)].

if getInitialPredicate(exp1) != getInitialPredicate(exp2)
    print("Predicates do not match. Cannot be
    unified")
    return False.

attributeCount1 = len(getAttributes(exp1))
attributeCount2 = len(getAttributes(exp2))
if attributeCount1 != attributeCount2:
    return False
head1 = getFirstPart(exp1)
head2 = getFirstPart(exp2).
initialSubstitution = unify(head1, head2).
if not initialSubstitution:
    return False
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 False
initialSubstitution.extend(remainingSubstitution)
return initialSubstitution

exp1 = "Knows(A,x)"
exp2 = "Knows(y,Y)"
substitution = unify(exp1, exp2)
print("Substitutions:")
print(substitutions)
```

output:

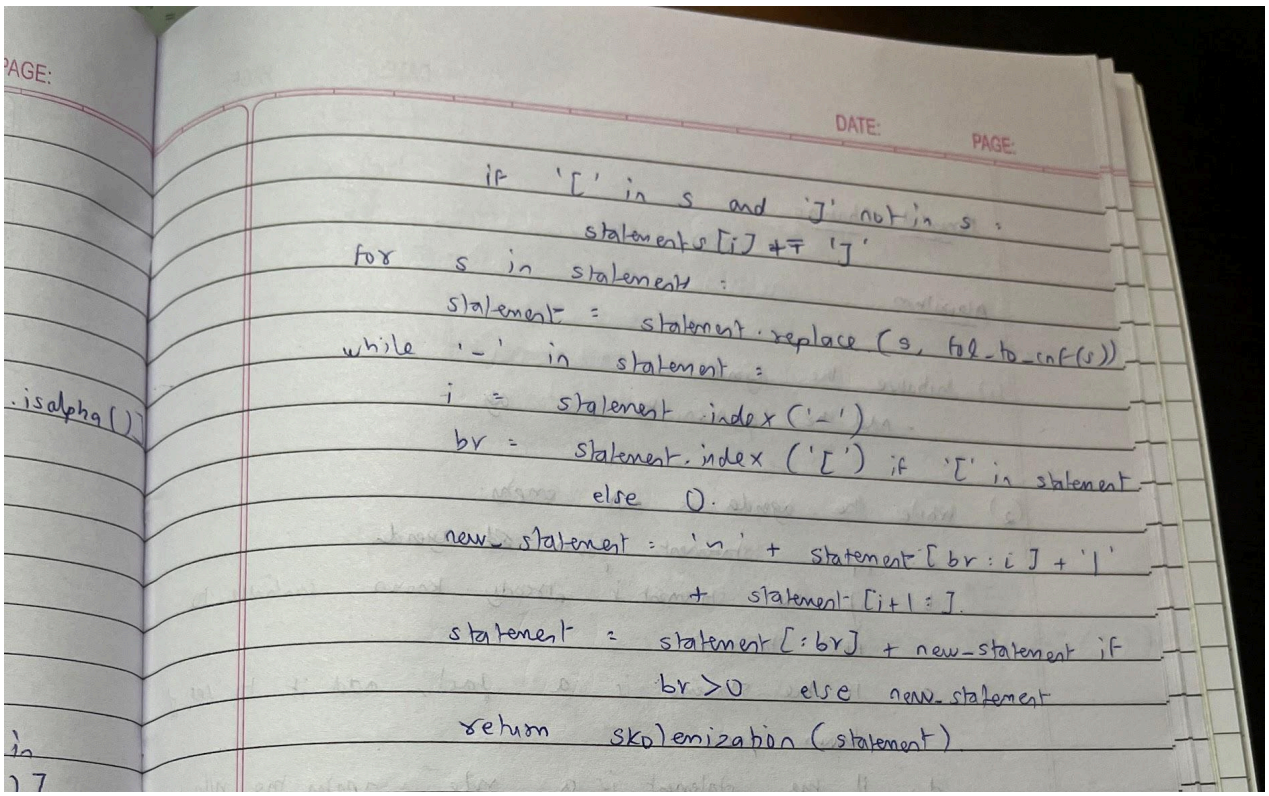
Substitution:

[('A', 'y'), ('Y', 'x')]

```
Substitutions:
[('A', 'y'), ('mother(y)', 'x')]
```

(2) FOL to CNF code

```
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 Skolemization(statement):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in
                        range(ord('A'), ord('Z')+1)]
    matches = re.findall('[∃]', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, '')
        for predicate in getPredicates(statement):
            attributes = getAttributes(predicate)
            if ''.join(attributes).islower():
                statement = statement.replace(match[1],
                SKOLEM_CONSTANTS.pop(0))
    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 :
    statement += ']'
for s in statement :
    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] + '|'
                    + statement[i+1:].
    statement = statement[:br] + new_statement if
                br > 0 else new_statement
    return skolemization(statement)
```

```
[~animal(y)|loves(x,y)]&[~loves(x,y)|animal(y)]
[animal(G(x))&~loves(x,G(x))]|[loves(F(x),x)]
[~american(x)|~weapon(y)|~sells(x,y,z)|~hostile(z)]|criminal(x)
```

DATE: PAGE:

Forward Chaining

Algorithm

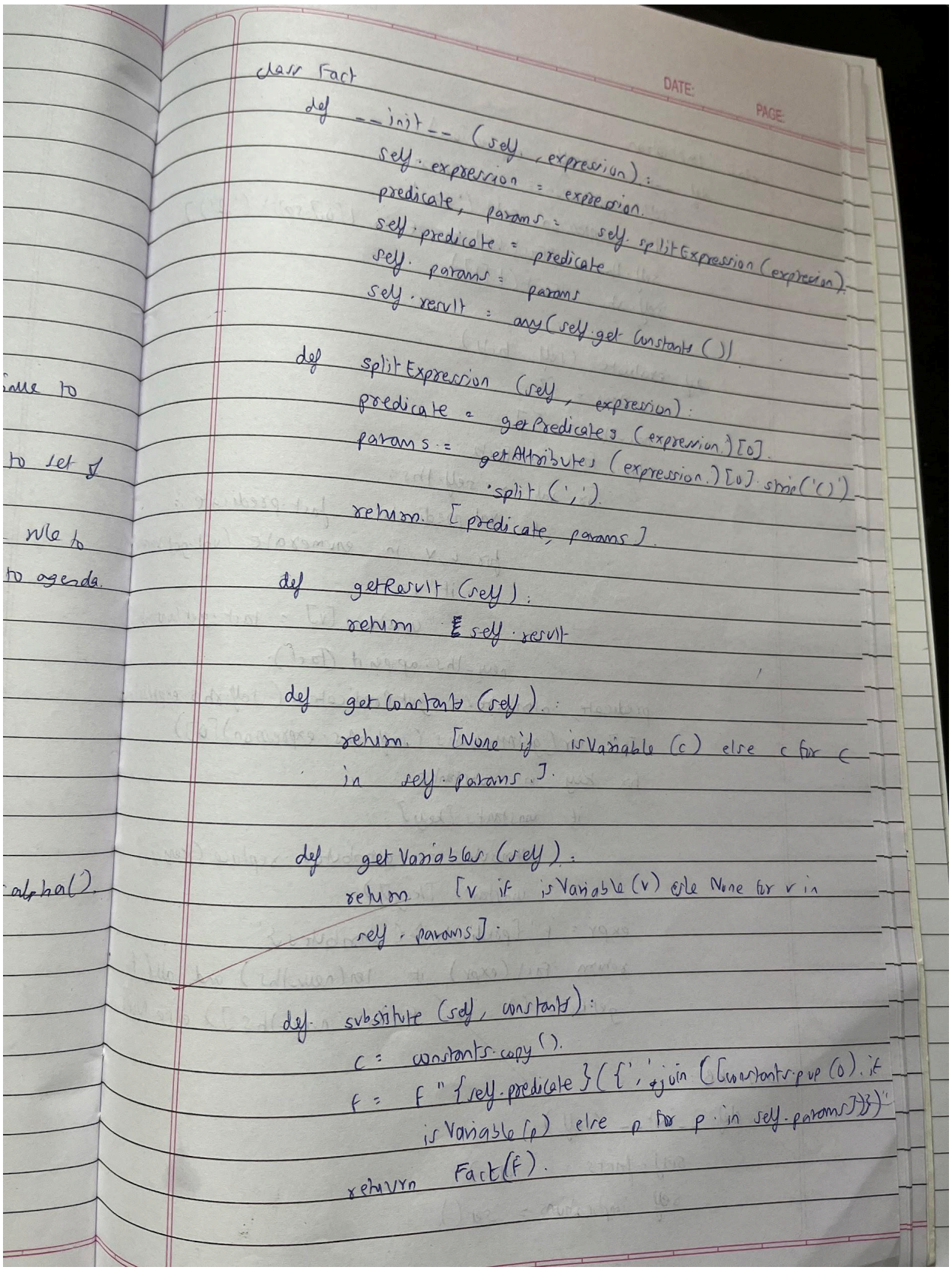
(1) Initialize the agenda
- Add the question to the ag

(2) While the agenda is not empty:
a. pop a statement from ~~the~~ agenda
b. if the statement is already known, continue to next statement
c. If the statement is a fact, add it to set of known facts
d. if the statement is a rule, apply the rule & generate new statement and add them to agenda

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 ~~re.findall(expr, string)~~ matches

def getPredicates(string).
    expr: '([a-zA-Z]+)\([^&|]+\)'
    return re.findall(expr, string)
```

class Fact
def
def
def
def



```
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 = 1

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


kb = KB()
kb.tell ('missile (x) => weapon (x)')
kb.tell ('missile (M1)')
kb.tell ('enemy (x, America) => hostile (x)')
kb.tell.('american(West)')
kb.tell ('enemy (Nono, America)')
kb.tell.('owns (Nono, M1)').
```

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