BMSCE
CSE

INDEX

NAME: Girish Kumar SK STD 5th sem SEC: ____ ROLL NO: 1BM21CS068

S.No.	Date	Title	Page No.	Teacher's Sign/ Remarks
		AI Lab		
1	17/11/23	WAP program to implement Tic-Tac-Toe		
2.	29/11/23	8-puzzle problem (bfs)		
3	8/11/23	8-puzzle problem (iterative deepening search algorithm		
4.	8/12/23	8-puzzle problem A* algorithm		
5.	22/12/23	Implement vacuum cleaner agent		
6.	29/12/23	Create knowledge base using propositional logic & check entailment for given query		
7.	29/12/23	Create knowledge base using propositional logic & prove given query using resolution		
8.	19/1/24	Unification in first order logic		
9.	19/1/24	Convert FOL to CNF		
10.	19/1/24	Create knowledge base consisting of FOL statements & prove given query using forward reasoning		

25

10/11/2023

Bafna Gold

WAP to demonstrate elif ladder to

1. Checks the age criteria

2. WAP to print multiplication table. (take number from user)

3. WAP pgm to print pattern
   - 1, 22, 333, 4444, ... n times n
   - 1, 12, 123, 1234, ...

4. Implement Bubble sort

5. Reverse the integer

Program.

1.

```
age = int(input("Enter your age: "))
if age <= 2:
    print("You are Baby")
elif age <= 12:
    print("You are a Child")
elif age < 18:
    print("You are a Teen")
elif age <= 40:
    print("You are a Adult Youth")
else:
    print("You are Old")
```

Output:

Enter your age: 34
You are a Adult Youth

2.

```
print ("**** Multiplication Table ****")
num = int (input ("Enter the number :"))
for i in range (1, 11):
    print (f"{num} x {i} = {num * i}")
```

Output:

**** Multiplication Table ****
Enter the number : 7
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70

3. Code

```
*
print ("*** Number Pattern ***")
num = input ("Enter the number (>=1):")
num_int = int (num)
res = ''
if num_int == 1:
    print ("1")
else:
    for i in range (1, num_int + 1):
        gk = str (i)
        k = i
```

```
    res = ''
    while (k>0):
        res = res + gk
        k = k-1
    print (res + ', ', end = '')
```

Output

```
*** Number Pattern ***
Enter the number (>=1) : 6
1, 22, 333, 4444, 55555, 666666,
```

```
*
print ("** Number Pattern **")
num = input ("Enter the number (>=1): ")
num_int = int(num)
res = ''
k = num_int
if (num_int == 1):
    print ("1")
else:
    while (k>0):
        res = ''
        for i in range (1, num_int - k + 2):
            gk = str(i)
            res = res + gk
        print (res + ', ', end = '')
        k = k-1
```

Output:

```
* * Number Pattern * *
Enter the number (>=1) : 4
1, 12, 123, 1234,
```

4. Bubble Sort

```
input_string = input("Enter list items with
    space between each item : ")
list_item = input_string.split()
n = len(list_item)
for i in range(n):
    list_item[i] = int(list_item[i])

print(list_item)

for k in range(n):
    for j in range(n-k-1):
        if list_item[j] > list_item[j+1]:
            temp = list_item[j]
            list_item[j] = list_item[j+1]
            list_item[j+1] = temp

print("List after sorting :")
print(list_item)
```

Output:

Enter list items with space between each item:
12 3 67 1 34 3 76 11
[12, 3, 67, 1, 34, 3, 76, 11]
List after sorting :
[1, 3, 3, 11, 12, 34, 67, 76]

5.

```
print ("** Reverse Integer **")
num = int(input ("Enter number :"))
res = 0
while num > 0:
    rem = num % 10
    res = res * 10 + rem
    num = num // 10
print (res)
```

Output:

** Reverse Integer **
Enter the number : 3456
6543

10/11

TIC-TAC-TOE

//Code:

```
import random

def comp_move():
    print("Computer's Turn")
    cin = random.randint(1,9)
    while not indexfree(cin):
        cin = random.randint(1,9)
    print(f"Computer chosen index : {cin}")
    r = (cin-1) % 3
    if cin < 4:
        a[0][r] = 'O'
    elif cin < 7:
        a[1][r] = 'O'
    else:
        a[2][r] = 'O'
    printbox()

def user_move():
    print("Your Turn")
    uin = int(input("Enter the index (1-9) : "))
    while not indexfree(uin):
        print("That position is occupied already.")
        uin = int(input("Enter the index (1-9): "))

    r = (uin-1) % 3
    if uin < 4:
        a[0][r] = 'X'
    elif uin < 7:
        a[1][r] = 'X'
    else:
        a[2][r] = 'X'
    Printbox()
```

```
def indexfree(ind):
    r = (ind-1)//3
    if ind < 4:
        if a[0][r] == ' ':
            return True
        else:
            return False
    elif ind < 7:
        if a[1][r] == ' ':
            return True
        else:
            return False
    else:
        if a[2][r] == ' ':
            return True
        else:
            return False

def win_comp():
    win = (( a[0][0] == 'O' and a[0][1] == 'O' and a[0][2] == 'O' )
           or
           # conditions.

    if win == True:
        print ("Computer Won")
        return 1
    else
        return 0

def win_user():
    ↓ same as above code with
        a[0][0] == 'X' ---
            ↓ condition
    if. —
        print ("You Won")
```

```
def printbox():
    print (f" {a[0][0]} | {a[0][1]} | {a[0][2]} ")
    print ("-------")
    print (f" {a[1][0]} | {a[1][1]} | {a[1][2]} ")
    print ("-------")
    print (f" {a[2][0]} | {a[2][1]} | {a[2][2]} ")

ch = 'y'
while ch == 'y':
    a = [[' ' for i in range(3)] for j in range(3)]
    print("*** Tic-Tac-Toe ***")
    print (" User symbol - 'X'
            Computer symbol - 'O' ")
    turn = random.randint(0,1)
    st = 0
    while st == 0:
        if turn == 0:
            st =
            comp_move()
            st = win_comp()
            if st == 1:
                print break
            user_move()
            st = win_user()
            if st == 1:
                break
            else:
                ↓ change above order

    ch = input ("Do you want to
                PLAY AGAIN ? (y/n) : ")
```

due on 17/11

Algorithm - (8 puzzle)

1. bfs (src, target)
   → Instantiate an empty queue & add source state 'src'
   queue = [] ; queue.append (src)
   while queue != empty:
   source ← queue.pop(0)
   → append source to 'exp' list
   exp.append (source)
   → If 'source' is equal to target state
   print 'success' & return
   → Generate possible moves from 'source' using possible moves function
   → for each possible move
   If move not in explored states or queue, add it to queue.

2. possible moves (state, visited_states)
   b = state.index (0) ← to find index of empty tile
   d = [ ] // initialize directions array
   if b not in [0,1,2]:
   d.append ('u')
   // In same way, determine possible directions to move based on position of empty tile
   // generate possible move in those direction
   // return a list of possible moves that have not been visited.

3. gen(state, m, b)
   - copy current state to temporary variable
   - perform move (swap) based on direction specified
   return new state after move

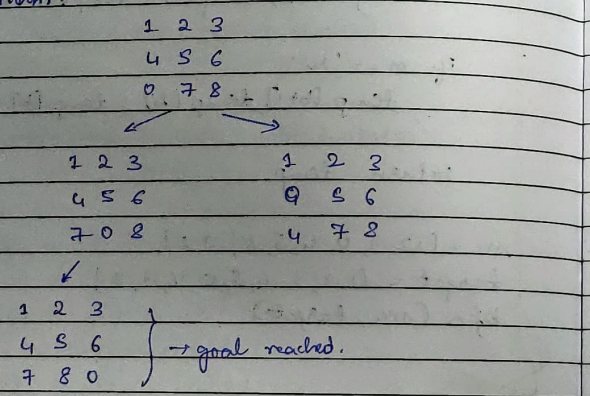
Code :-

```
import numpy as np
import pandas as pd
is
def bfs (src, target):
    queue = [ ]
    queue.append (src)
    exp = [ ]

    while len (queue) > 0:
        source = queue.pop(0)
        exp.append (source)
        print (source[0], '|', source[1], '|', source[2])
        print (source[3], '|', source[4], '|', source[5])
        print (source[6], '|', source[7], '|', source[8])

        if (source == target):
            print ("success")
            return

        poss_moves_to_do = [ ]
        poss_moves_to_do = possible_moves(source, exp)
```

```
for moves in pos_moves_to_do:
    if moves 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_can = []

    for i in d:
        pos_moves_it_can.append(gen(state, i, b))

    return [moves_it_can for move_it_can
    in pos_move_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]

    if m == 'u'
        temp[b-3], temp[b] = temp[b], temp[b-3]

    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

src = [1,2,3,0,4,5,6,7,8]
target = [1,2,3,4,5,0,6,7,8]
bfs(src, target)
```

08-12-2023

1. 8-puzzle using iterative deepening search algorithm.



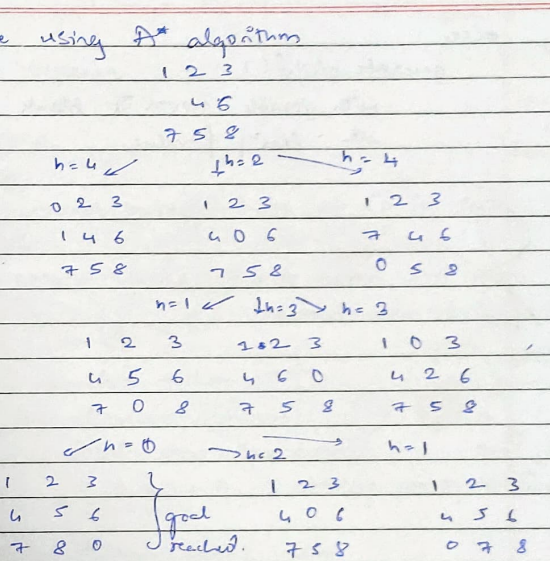
Algorithm:

1. Initialize initial state = [ ] & goal state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
2. Set depth = 1 & expand initial state. Perform depth-limited-search (depth)

```
if node.state = goal
    return node
else
    for neighbour in get_neighbour (node.state)
        child = puzzle_node (neighbour, node)
        result = depth_limited_search (depth-1)
        if result == True:
            return result.
```

3. After one iteration when depth = 1, increment depth by 1 & perform depth-limited-search again
4. Here get-neighbour will generate possible moves by swapping the 0th tile.
5. The path traversed is printed to reach goal state

2. 8-puzzle using A* algorithm



Algorithm

1. Create goal state = [1, 2, 3, 4, 5, 6, 7, 8, 0] & initial state = [ ]
2. Calculate f value i.e f = g + h

```
To calculate h
    for i in range (0, n)
        for j in range (0, n)
            if start[i][j] != goal[i][j] and
               start[i][j] != '_':
                temp += 1
    return temp
f = h + g
return f.
```

3. Append initial list to open
4. If h-value == 0

```
return goal state
print all traversed state.
```

else

generate_child()

with possible moves of blank tile, go forward with least f-value.

22-12-2023

Bafna Gold

Date: Page:

Vacuum Cleaner

* Two rooms.

Algorithm:

1. Two rooms represented as list (0 for clean, 1 for dirty)

initialize:

```
print("Room 0:") rooms = [0, 1]   # two rooms, first room clean,
print("Room 1:")
print("Rooms:")                    second room dirty
```

2. define function → clean_rooms

```
def clean_rooms(rooms):
    total_rooms = len(rooms)
    cur_pos = 0   # starting pos of vacuum cleaner
    print(f"Vacuum at room {cur_pos+1}")
    print(f"
    #clean initial room
    clean_room(rooms, cur_pos)

    #move to next room & clean it
    cur_pos = (cur_pos + 1) % total_rooms
    clean_room(rooms, cur_pos)
```

#come back to initial room

```
cur_pos = (cur_pos + 1) % total_rooms
clean_room(rooms, cur_pos)
```

3. define function → clean_room

```
def clean_rooms(rooms, position):
    if rooms[pos] == 1:
        print("Room {pos+1} is dirty. Cleaning...")
        rooms[position] = 0
        print("Room {pos+1} cleaned.")
    else:
        print("Room {pos+1} is already clean")
```

print(f"Vacuum at room {pos+1}")

print(f"Is room clean {0==rooms[pos]}")

R1	R2
R4	R3

* Four Rooms

Algorithm:

1. four rooms represented as a list (0 for clean, 1 for dirty)
   initialize - rooms = [0, 1, 0, 1] # four rooms
   for i in range(4):
   print(f"Room {i+1}") print(f" Is cleaned to == rooms[0]")
2. define clean_four_rooms function
   def clean_four_rooms(rooms):
   total_rooms = len(rooms)
   cur_pos = 0 # start pos of vacuum cleaner
   # clean initial room
   clean_room(rooms, cur_pos)
   # move to next room & clean it
   for i in range(total_rooms - 1):
   cur_pos = (cur_pos + 1) % total_rooms
   clean_room(rooms, cur_pos)
   # come back to first pos
   cur_pos = (cur_pos + 1) % total_rooms
   clean_room(rooms, cur_pos)
3. define clean_room(rooms, position):
   def clean_room(rooms, position):
   if rooms[position] == 1:
   print(f"Room {pos+1} is dirty. Cleaning...")
   rooms[position] = 0
   print(f"Room {position+1} cleaned.")
   else:
   print(f"Room {position+1} is already clean.")

Vacuum cleaner at Room {pos+1}")
In room cleaned:
{0 == rooms[pos]}")

Output:

→

Vacuum cleaner at Room 1
In room cleaned? True
Room 1 is already cleaned.
Vacuum cleaner at Room 2
In Room cleaned? False
Room 2 is dirty. Cleaning....
Room 2 is cleaned.
Vacuum cleaner at Room 1
In Room cleaned? True
Room 1 is already cleaned
Vacuum cleaner is turning OFF

Output:

Vacuum cleaner at Room 1
In room cleaned? True
Room 1 is already cleaned.
Vacuum cleaner at Room 2
In room cleaned? False
Room 2 is dirty. Cleaning....
Room 2 is cleaned.
Vacuum cleaner at Room 3
In Room cleaned? True
Room 3 is already cleaned
Room.
Vacuum cleaner at Room 4
In room cleaned? False
Room 4 is dirty. Cleaning....
Room 4 is cleaned.
Vacuum cleaner at Room 1
In room cleaned? True
Room 1 is already cleaned.
Vacuum cleaner is turning OFF

29/12/2023

Implement knowledge based resolution for
KB = p , ~p v q , p v ~q , p v ~q v r , ~q v r

class

```
class KnowledgeBase:
    def __init__(self):
        self.facts = set()

    def add_fact(self, fact):
        self.facts.add(fact)

    def check_entailment(self, statement):
        return all(rule(statement) for rule in
                   self.facts)

def rule_0(statement):
    return 'p' == statement.lower()

def rule_1(statement):
    return "~p" in statement.lower() or "q" in stat.lower()

def rule_2(statement):
    return "p" in statement.lower() or "~q" in stat.lower()

def rule_3(statement):
    return 'p' in statement.lower() or "~q" in
        statement.lower() or "r" in statement.lower()

def rule_4(statement)
    return "~q" in statement.lower() or "r" in
        statement.lower()
```

```
kb = KnowledgeBase()
kb.add_fact(rule_0)
kb.add_fact(rule_1)
kb.add_fact(rule_2)
kb.add_fact(rule_4)

us_st = input("Enter statement : ")
entailement_result = kb.check_entailement(us_st)

if entailement_result:
    print("The statement is entailed")
else:
    print("The statement is not entailed")
```

Output:

Enter statement : ~q

The statement is not entailed by knowledge base

Enter statement : ~p v q v ~r

Knowledge Based Entailment

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

def create_knowledge_base():
    p = symbols('p')
    q = symbols('q')
    r = symbols('r')

    knowledge_base = And(
        Implies(p, q),
        Implies(q, r),
        Not(p)
    )

    return knowledge_base

def query_entails(knowledge_base, query):
    entailment = satisfiable(And(knowledge_base,
                    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: ~p & (Implies(p,q)) &
                (Implies(q,r))
Query: p
Query entails Knowledge Base: True
```

19-01-2024

Bafna Gold
Date: Page:

9. Implement unification in first order logic

```
def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression[:-1]
    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(exp)
        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)
    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, exp2):
            return False
        else:
            return [(exp1, exp2)]
```

```
if getInitialPredicate(exp1) != getInitialPredicate(exp2):
    print("Predicates do not match. Can't 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(X)"
exp2 = "knows(Richard)"
substitutions = unify(exp1, exp2)
print("Substitutions:")
print(substitutions)

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

Output:

Substitutions:

[('x', 'Richard')]

Substitutions:

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

10. Convert FOL to CNF

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

def getPredicates(string):
    expr = '[a-zA-Z]+\([A-Za-z,]+\)'
    return re.findall(expr, string)

def DeMorgan(sentence):
    string = ''.join(list(sentence).copy())
    string = string.replace('~~', '')
    flag = '[' in string
    string = string.strip('[]')
    for predicate in getPredicates(string):
        string = string.replace(predicate, f'~{predicate}')
    s = list(string)
    for i, c in enumerate(string):
        if c == '|':
            s[i] = '&'
        elif c == '&':
            s[i] = '|'
    string = ''.join(s)
    string = string.replace('~~', '')
    return f'[{string}]' if flag else string

def Skolemization(sentence):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in
                        range(ord('A'), ord('Z')+1)]
    statement = ''.join(list(sentence).copy())
    matches = re.findall('[∀∃].', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, '')
        statements = re.findall('\[[^\]]+\]',
                                statement)
        for s in statements:
            statement = statement.replace(s, s[1:-1])
        for predicate in getPredicates(statement):
            attributes = getAttributes(predicate)
            if ''.join(attributes).islower():
                statement = statement.replace(
                    match[1], SKOLEM_CONSTANTS.pop(0))
            else:
                aL = [a for a in attributes if
                      a.islower()]
                aU = [a for a in attributes if
                      not a.islower()][0]
                statement = statement.replace(
                    match[1], f'{SKOLEM_CONSTANTS.pop(0)}({aL})')
    return statement

import re
def fol_to_cnf(fol):
    statement = fol.replace("<=>", "_")
    while '_' in statement:
        i = statement.index('_')
        new_st = '[' + statement[:i] + '=>'
                 + statement[i+1:] + '] & [' + statement[:i]
                 + '=> ' + statement[:i] + ']'
        statement = new_st
    statement = statement.replace("=>", "-")
    expr = '\[([^]]+)\]'
    statements = re.findall(expr, statement)
```

```
for i,s in enumerate(statements):
    if '[' in s and ']' not in s:
        statements[i] += ']'
for s in statements:
    statement = statement.replace(s, fol_to_cnf(s))
while '-' in statement:
    i = statement.index('-')
    br = statement.index('[') if '[' in statement else 0
    stmt = statement[br:i] + statement[i+1:] + statement[br:i]
    statement = statement[:br] + stmt
while '∀' in statement:
    i = statement.index('∀')
    statement = list(statement)
    statement[i], statement[i+1], statement[i+2] = statement[i+1], statement[i+2], statement[i]
    statement = ''.join(statement)
while '∃' in statement:
    i = statement.index('∃')
    s = list(statement)
    statement = statement.replace('∃'+s[i+1], '')
    statement = statement.replace(s[i+2], s[i+2]+'('+s[i+1]+')')
expr = '(~[∀|∃].,)'
statements = re.findall(expr, statement)
for s in statements:
    statement = statement.replace(s, fol_to_cnf(s))
for s in statements:
    statement = statement.replace(s, DeMorgan(s))
return statement
```

```
print(skolemization(fol_to_cnf("criminal(y)<=>loves(x,y)")))
print(skolemization(fol_to_cnf("∀x[∀y[animal(y)=>loves(x,y)]=>[∃z[loves(z,x)]]")))
print(fol_to_cnf("[american(x)&weapon(y)&sells(x,y,z)&hostile(z)]=>criminal(x)"))
```

Output:

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

10. Create KB consisting of FOL statements & prove 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 fact
                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(l[0])
        self.rhs = c
    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}')

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)")
kb.tell("missile(x) & owns(Nono,x) => sells
        (West,x,Nono)")
kb.tell("american(x) & weapon(y) & sells(x,y,z)
        & hostile(z) => criminal(x)")
kb.query("criminal(x)")
kb.display()
```

Output:

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

```
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 =>

Querying evil(x):
1. evil(John)