Date ___/___/___
Page _____

Experiment-1 : Implement Tic Tac Toe

Tic Tac Toe is a two-player, zero-sum game. We use the minimax algorithm to implement the game of tic tac toe.

Minimax Algorithm

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player.

There are two players called the maximizer and the minimizer. The maximizer tries to get the highest score, make positive score possible, while the minimizer does the opposite.
We program our AI ~~to program~~ to find the best possible move using DFS algorithm to search the game tree and find the path that leads to the maximum value (if the AI is the maximizer) and ~~hence~~ hence win the game.

~~→ Game Tree [Game Tree]:~~

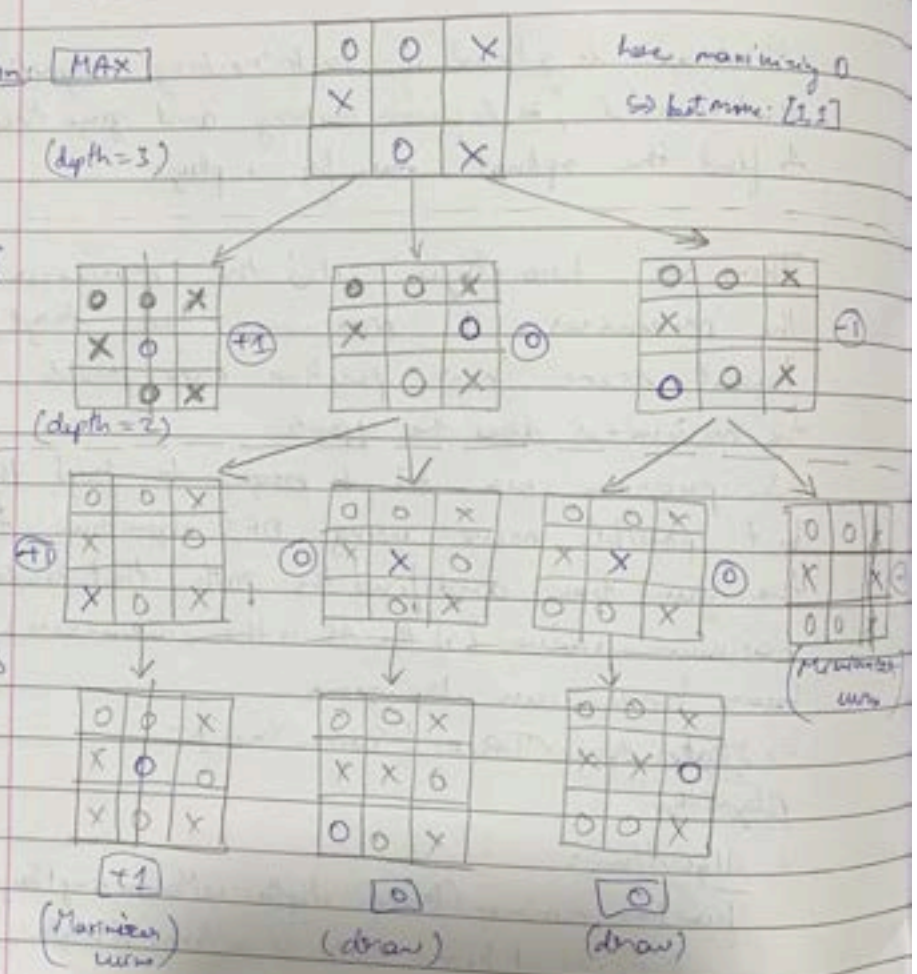
~~Algorithm~~ :

→ Algorithm :

```
function minimax (board, depth, isMaximizingPlayer):
    if current board state is a terminal state:
        return value of the board
    if isMaximizingPlayer:
        bestVal = -INFINITY
        for each move in board:
            value = minimax (board, depth+1, false)
            bestVal = max (bestVal, value)
    return bestVal
```

```
else:
    bestVal = +INFINITY
    for each move in board:
        value = minimax(board, depth+1, true)
        bestVal = min(bestVal, value)
    return bestVal.
```

→ Game Tree [State Space Tree]:



8/11/24

→ Code:

```
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):
    countO = 0
    countX = 0
    for y in [0,1,2]:
        for x in board [y]:
            if x == "O":
                countO = countO + 1
            elif x == "X":
                countX += 1
    if countO >= countX:
        return X
    elif countX > countO:
        return O

def actions(board):
    freeboxes = set()
    for i in [0,1,2]:
        for j in [0,1,2]:
            if board [i][j] == EMPTY:
                freeboxes.add ((i,j))
    return freeboxes
```

```
def result(board, action):
    i = action[0]
    j = option[1]
    if type(action) == list:
        action = (i,j)
    if action in actions(board):
        if player(board) == X:
            board[i][j] = X
        elif 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 (board[0][0] == board[0][1] == board[0][2] == O or
        board[1][0] == board[1][1] == board[1][2] == O or
        board[2][0] == board[2][1] == board[2][2] == 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) == 'X'):
        return ~~True~~ 1
    ~~return False~~
    elif (winner(board) == 'O' :
        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
```

```
EDDS:
    for move in actions(board):
        result(board, move)
        scores.append(minimax_helper(board))
        board[move(0)].(move(1)) = EMPTY
    sort
    return max(scores) if isMaxTerm else min(scores)
def minimax(board):
    isMaxTerm = True if player(board) == X else false
    bestScore = None
    if isMaxTerm:
        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~~ return bestMove
else else:
    bestScore = +math.inf
    for move in actions(board):
        result(board, move)
        score = minimax_helper(board)
        board[move[0]][move[1]] = EMPTY
        if (score is bestScore):
            bestScore = score
            bestMove = move
    return bestMove
```

```
def print_board(board):
    for Row in board:
        print(row)
game_board = initial_state()
print("Initial board:")
print_board(game_board)

while not terminal(game board):
    if player(gameboard) == X:
        user_input = input("Enter your move
                                (row, column): ")
    row, column = map(int user_input.split(','))
    result(game_board, (row, col))
else
    print("AI is making a move").
    move = minimax(copy.deepcopy(game board))
    result(game_board, move)

print("Current Board")
    move = minimax(copy.deepcopy(game board))
    result(gameboard, move)

if winner(game_board) is ~~max~~ not none:
    print("The winner is: {winner(gameboard)}")
else:
    print("It is a tie")
```

Program - 2 : ~~8-Puzzle Problem~~ ~~Vacuum Cleaner Agent~~

```
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
        pos_moves_todo = []
        pos-moves-todo = possible_moves (source, exp)

        for move in pos-moves-todo :

            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_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]

    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 = [1,2,3,0,4,5,6,0,7,8]
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:
def iddfs(src, target, depth):
    for i in range(depth):
        visited_states = []
        if dfs(src, target, i+1, visited_states):
            return True
    return False

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

result = bfs(src, target)
print("Solution:", result)
```

Program-3: Vaccum Cleaner Agent

```
def vacuum_world():

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

location_input = input("Enter location of Vocuum")

status_input = input("Enter status of " + location_input)

status_input = 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.")
    else:
        print("No action " + str(cost))
        print("Location B is already clean.")

if status_input == '0':
    print("Location A is already clean")
    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 for SUCK" + str(cost))
print("Location B has been cleaned.")
else
else:
    print("No action" + str(cost))
    print(cost)
    print("Location B is already clean")

else:
    print("Vaccum is placed in location B")

    if status_input == '1':
        print("Location B is dirty")

        goal_state['B'] = '0'
        Cost += 1
        print("COST of for CLEANING " + str(cost))
        print("Location B has been Cleaned")

    if status_input_complement == '1':
        print("Location A is dirty")
        print("Moving LEFT to the location A.")
        cost += 1
        print("COST for moving LEFT" + str(cost))
        goal_state['A'] = '0'
        cost += 1
        print("Cost for SUCK" + str(cost))
        print("Location A has been Cleaned")
```

```
else:
        print(cost)
        print("Location B is already clean")
    if status_input_complement == '1':
        print("Location A is Dirty.")
        print("Moving LEFT to the location A.")
        cost+=1
        print("COST for moving LEFT" + str(cost))
        goal_state['A'] = '0'
        cost+=1
        print("Cost for SUCK" + str(cost))
        print("Location A has been cleaned")
    else:
        print("No action" + str(cost))
        print("Location A is already clean.")
print("GOAL STATE:")
print(goal_state)
print("Performance Measurement:" + str(cost))

vacuum_world()
```

Program - 9 : 8-puzzle using iterative deepening search Algorithm

Test Case :

source state :

1	2	3
-1	4	5
6	7	8

target :

1	2	3
4	5	-1
6	7	8

depth = 2

Output : ~~False~~ True.

→ Code :

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

def possible_moves(state, visited):
    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]

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):
            return True
    return False

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

depth = 2
result = iddfs(src, target, depth)
print("solution", result)
```

Bar

Date / /
Page

5) A^* Algorithm

```
def print_b(src):
    state = src.copy()
    state = [state.index(-1)] = ' '
    print(f"""
        {state[0]} {state[1]} {state[2]}
        {state[3]} {state[4]} {state[5]}
        {state[6]} {state[7]} {state[8]}
        """)

def h(state, target):
    count = 0
    i=0
    for j in state:
        if state[i] != target[j]:
            count = count+1
    return count

def astar(state, target):
    states = [src]
    g=0
    visited_state = []
    while len(state):
        print(f"level{g}:")
        moves = []
        for state in states:
            visited_states.append(state)
        print(state)
        if state == target:
            print("success")
            return
```

```
moves += [move for move in possible_moves(
            state, visited_states) if move not in
            moves]
costs = [g + h(move, target) for move in moves]
states = [moves[i] for i in range(len(moves)) if
            costs[i] == min(costs)]
g += 1
print("fail")

def possible_moves(state, visited_states):
    b = state.index(-1)
    d = []
    if b in range(8):
        d.append('u')
    if b not in [0, 3, 6]:
        d.append('l')
    if b not in [2, 5, 8]:
        d.append('r')
    if b in range(9):
        d.append('d')
    pos_moves = []
    for m in d:
        pos_moves.append(gen(state, m, b))
    return [move for move in pos_moves if
            move not in visited_states]
def gen(state, m, b):
    temp = state.copy()
    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]
```

Date __/__/__
Page _____

one = [2, 8, 3, 1, 6, 4, 7, -1, 5]
target = [1, 2, 3, 8, -1, 4, 7, 6, 5]
state (one, target)

~~goal~~ :

1	2	3
6	-1	4
7	6	5

2	8	3
1	6	4
7	-1	5

2	8	3
1	6	4
-1	7	5

2	8	3
1	6	4
7	5	-1

2	8	3
1	-1	4
7	6	5

2	8	3
-1	1	4
7	6	5

2	-1	3
1	8	4
7	6	5

2	8	3
1	4	-1
7	6	5

[(not q) or (not p) or r] and [(not p or q) ...]
~q ~ ¬

6) Knowledge base using Propositional logic

-> ~~Knowledge~~ Expression: $(\neg q \vee \neg p \vee r) \wedge (\neg q \wedge p) \wedge q$

query: r.

p	q	r	KB	Query (r)
T	T	T	F	T
T	T	F	F	F
T	F	T	F	T
T	F	F	F	F
F	T	T	F	T
F	T	F	F	F
F	F	T	F	T
F	F	F	F	F

Query entails the knowledge.

10/10

Code:

```
def evaluate_expression(p, q, r):
    expression_result = (p or q) and (not q or p)
    return expression_result

def generate_truth_table():
    print("p|q|r|Expression (KB)|Query (p^r)")
    print("---|---|---|---------|---------")

    for p in [True, False]:
        for q in [True, False]:
            for r in [True, False]:
                expression_result = evaluate_expression(p,q,r)
                query_result = p and r
```

20/12/23

```
print(f" {p} | {q} | {r} | {expression_result} | {query_result}")

def query_entails_knowledge():
    for p in [True, False]:
        for q in [True, False]:
            for r in [True, False]:
                expression_result = evaluate_expression(p,q,r)
                query_result = p and r

                if expression_result and not query_result:
                    return False
    return True

def main():
    generate_truth_table()
    if query_entails_knowledge():
        print("\n Query entails the knowledge")
    else:
        print("\n Query does not entail the knowledge")

main()
```

Date / /
Page

7) KB using propositional logic and prove the given query using resolution

(i) KB: $R \vee \sim P$ $R \vee \sim Q$ $\sim R \vee P$ $\sim R \vee Q$
query : R

Step	Clause	Derivation
1	$R \vee \sim P$	Given.
2	$R \vee \sim Q$	Given
3	$\sim R \vee Q$	Given
4	$\sim R \vee P$	Given
5	$\sim R$	Negated conclusion
6	~~null~~ $R \vee \sim P$	~~Given~~ 2 and 3

Therefore null clause got.
Contradiction is found when $\sim R$ (negated query) taken. Thus R is true.

Program:

```
from z3 import Implies, Not, Bool, Solver, sat

p,q,r = Bool('p'), Bool('q'), Bool('r')

kb = Implies(p,q) & Implies(q,r) & Not(p)

query = r

def prove_query_with_resolution(knowledge_base, query):
    s = solver()
    s.add(Not(knowledge_base), query)
    result = s.check()
    return result == sat
```

```
proof_result = prove_query_with_resolution(kb, query)

if proof_result:
    print("The query is proved to be true")
else:
    print("The query is not proved to be true")
```

Date ___/___/___
Page ___

7) Unification of first Order logic

Enter the first Expression: Knows(f(x), y)
Enter the second Expression: Knows(J, John)
The substitutions are:
['g / f(x)', 'John / y']

Code:

```
def unify(expr1, expr2):

    func1, args1 = expr1.split('(', 1)
    func2, args2 = expr2.split('(', 1)

    if func1 != func2:
        print("Different functions, cannot unify")
        return None

    args1 = args1.strip(')').split(',')
    args2 = args2.strip(')').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.islower():
            substitution[a1] = a2
        elif not a1.islower() and a2.islower():
            substitution[a2] = a1
        elif a1 != a2:
            print("Expressions cannot be unified")
            return None
    return substitution
```

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

expr1 = input("Enter first expression: ")
expr2 = input("Enter Second expression: ")

substitution = unify(expr1, expr2)

if substitution:
    print("The substitutions are: ")
    out = []
    for key, value in substitution.items():
        out += [key + " / " + value]
    print(out)

expr1_result = apply_substitution(expr1, substitution)
expr2_result = apply_substitution(expr2, substitution)

print(f'Unified expression 1: {expr1_result}')
print(f'Unified expression 2: {expr2_result}')
```

Date __/__/__

Page _____

Convert given first Order logic statement into Conjunctive Normal Form (CNF)

Input : $\forall x \; food(x) \Rightarrow likes(John, x)$

Step-1 : Eliminate $\Rightarrow$ and rewrite

$$\forall x \; \neg food(x) \vee likes(John, x)$$

Step-2 : Move negation "$\neg$" inwards

Step-3 : Standardize variables apart by renaming them

Step-4 : Skolemize each existential variable is replaced by a skolem constant or Skolem function of the enclosing universally quantified variable

$$\forall x \; \neg food(A) \vee likes(John, A)$$

, where A is the new skolem ~~function~~ constant

Step-5 : Drop universal quantifiers

$$\neg food(A) \vee likes(John, A)$$

Step-6 : De Morgan's ~~la~~ application

(here we don't need to use De Morgan's law)

Date ___ / ___ / ___
Page ___

Example-2 : $\forall x [\exists z [loves(x, z)]]$

→ No "⇒" or negation in FOL statement, so continue to step 4

→ Drop universal quantifiers

~~O/p → [loves(x, B(x))] , new skolem function B(x)~~

→ ~~No~~

$\forall x [\exists z [loves(x, z)]]$

→ Step-4 : Skolemize: $z = B(x)$

new skolem function: B.

$\forall x [\exists z [loves(x, B(x))]]$

→ Step-5 : Drop universal quantifiers :

$[loves(x, B(x))]$

→

→ 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-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.replace('~[', '')
string = string.strip(']')
for predicate in getPredicates(string):
    string = string.replace(predicate, f'~{predicate}')
s = list(string)
for i, c in enumerate(string):
    if c == '|':
        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(matches[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(aL,
    f'{SKOLEM_CONSTANTS.pop(0)}' (f[aL[0] if len(aL)
    else match[1]]))
    return statement

def fol_to_cnf(fol):

    statement = fol.replace("<=>", "_")
    while '_' in statement:
        i= statement.index('_')
        new_statement = '[' +statement[:i]+ '=>'+
                statement[i+1:]+']&['+statement[i+1:]
            +'=>'+statement[:i]+']'
        statement = new_statement //(inside while block
        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
        newstatement = '~' + statement [br:i] + '∨' +
            statement [i+1:]

    while '~∀' in statement:
        i = statement.index('~∀')
        statement = list(statement)
```

```
statement [i], statement [i+1], statement [i+2] = '∃',
statement [i+2], '~'
statement = ''.join(statement)
while '~∃' in statement:
    i = statement.index ('~∃')
    s = list(statement)
    s[i], s[i+1], s[i+2] = '∀', s[i+2], '~'
    statent = ''.join(s)
statement = statement.replace ('~[∀', '[~∀')
statement = statement.replace ('~[∃', '[~∃')
expr = '(~[∀ | ∃].)'
statements = re.findall (expr, statement)
for s in statements:
    statements = statement.replace (s, fol_to_cnf(s))
for s in statements:
    statement = statement.replace (s, DeMorgan(s))
return statement.

inp_stm = input ("Enter the FOL statement:")
print(Skolemization (fol_to_cnf (inp_stm)))
```

Date __/__/__
Page ____

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

Output:

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

All facts:
1. evil (John)
2. greedy (John)
3. king (John)
4. king (Richard)

(2) Querying criminal (x):
1. criminal (West)

All facts:
1. hostile (nono)
2. criminal (West)
3. owns (Nono, M1)
4. Missile (M1)
5. American (west)
6. Weapon (M1)
7. Sells (West, M1, Nono)
8. Enemy (nono, America)

FOL: missile(x) ⇒ weapon(x)
missile (M1)
enemy (x, America) ⇒ hostile(x)
america (west)
enemy (nono, America)
owns (Nono, M1)

America (west)
enemy (Nano, America)
owns (Nano, M1)
missile (x) & owns (Nano, x) ⇒ sells (West, x, Nano)

hostile (z) ⇒ criminal (x)
Criminal (x)

FOL: King (x) & greedy (x) ⇒ evil (x)
King (John)
greedy (John)
King (Richard)
evil (x)

Code:

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

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

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

class Fact:
    def __init__(self, expression):
        self.expression = expression
        predicate, params = self.splitExpression(expression)
```

```
        self.predicate = predicate
        self.params = params ~~self.splitExpression(~~
        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 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 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, America
kb.tell('American(x) & weapon(y) & sells(x,y,z) &
                                hostile(z) => criminal(x)')
kb.query('criminal(x)')
kb.display()
```