

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.



ARTIFICIAL INTELLIGENCE LAB REPORT

Submitted by

Samarth C Shetty(1BM19CS141)

Under the Guidance of

Dr. Kavitha Sooda
Associate Professor, BMSCE

in partial fulfilment 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

Oct-2021 to Jan-2022

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 Advance Data Structures Lab for Cycle 2 (CIE 2) carried out by, **Samarth C Shetty** who are Bonafede students of **B. M. S. College of Engineering**. It is in partial fulfilment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visveswararajah Technological University, Belgaum during the year 2021-2022. The Lab report has been approved as it satisfies the academic requirements in respect of **ARTIFICIAL INTELLIGENCE (20CS5PCAIP)** work prescribed for the said degree.

Signature of the Guide

Dr. Kavitha Sooda

Associate Professor

BMSCE, Bengaluru

Signature of the HOD

Dr. Umadevi V

Associate Prof. & Head, Dept. of CSE

BMSCE, Bengaluru

Index

Serial No.	TITLE	PAGE NO.
1	Create a knowledgebase using prepositional logic and show that the given query entails the knowledge base or not.	03
2	Create a knowledgebase using prepositional logic and prove the given query using resolution	08
3	Implement unification in first order logic	17
4	Convert given first order logic statement into Conjunctive Normal Form (CNF)	21
5	Create a knowledgebase consisting of first order logic statements and prove the given query using forward reasoning.	25

1. Create a knowledgebase using propositional logic and show that the given query entails the knowledge base or not.

```
combinations=[(True,True, True),(True,True,False),(True,False,True),(True,False,
False),(False,True, True),(False,True, False),(False, False,True),(False,False, False)]
```

```
variable={'p':0,'q':1, 'r':2}
```

```
kb=""
```

```
q=""
```

```
priority={'~':3,'v':1,'^':2}
```

```
def input_rules():
```

```
    global kb, q
```

```
    kb = (input("Knowledge base : "))
```

```
    q = input("Query : ")
```

```
def entailment():
```

```
    global kb, q
```

```
    print("*10+"Truth Table Reference"+"*10)
```

```
    print('kb    α')
```

```
    print('-'*10)
```

```
    for comb in combinations:
```

```
        s = evaluatePostfix(toPostfix(kb), comb)
```

```
        f = evaluatePostfix(toPostfix(q), comb)
```

```
        print(s, f)
```

```
        if s is True and f is False:
```

```
            return False
```

```
    return True
```

```
def isOperand(c):
```

```
    return c.isalpha() and c!='v'
```

```
def isLeftParanthesis(c):
```

```
    return c == '('
```

```
def isRightParanthesis(c):
```

```
    return c == ')'
```

```
def isEmpty(stack):  
    return len(stack) == 0
```

```
def peek(stack):  
    return stack[-1]
```

```
def hasLessOrEqualPriority(c1, c2):  
    try:  
        return priority[c1]<=priority[c2]  
    except KeyError:  
        return False
```

```
def toPostfix(infix):  
    stack = []  
    postfix = "  
    for c in infix:  
        if isOperand(c):  
            postfix += c  
        else:  
            if isLeftParanthesis(c):  
                stack.append(c)  
            elif isRightParanthesis(c):  
                operator = stack.pop()  
                while not isLeftParanthesis(operator):  
                    postfix += operator  
                    operator = stack.pop()  
            else:  
                while (not isEmpty(stack)) and hasLessOrEqualPriority(c, peek(stack)):  
                    postfix += stack.pop()  
                stack.append(c)  
    while (not isEmpty(stack)):  
        postfix += stack.pop()  
  
    return postfix
```

```
def evaluatePostfix(exp, comb):  
    stack = []  
    for i in exp:  
        if isOperand(i):
```

```
    stack.append(comb[variable[i]])
elif i == '~':
    val1 = stack.pop()
    stack.append(not val1)
else:
    val1 = stack.pop()
    val2 = stack.pop()
    stack.append(_eval(i, val2, val1))
return stack.pop()
```

```
def _eval(i, val1, val2):
    if i == '^':
        return val2 and val1
    return val2 or val1
```

```
input_rules()
ans = entailment()
if ans:
    print("The Knowledge Base entails query")
    print(" KB  $\models \alpha$  ")
else:
    print("The Knowledge Base does not entail query")
print("\n")
```

OUTPUT

```
C:\SHIVANSHU\WORK\Scripts\python.exe C:/Users/Dell/PycharmProjects/pythonProject/knowledgebase.py
Enter the rule: (~Qv~PvR)^(~Q*P)*Q
Enter the query: R
Evaluating: ( not False or not False or False) and ( not False and False) and False
Knowledge Base: False Query: False
Evaluating: ( not False or not False or True) and ( not False and False) and False
Knowledge Base: False Query: True
Evaluating: ( not True or not False or False) and ( not True and False) and True
Knowledge Base: False Query: False
Evaluating: ( not True or not False or True) and ( not True and False) and True
Knowledge Base: False Query: True
Evaluating: ( not False or not True or False) and ( not False and True) and False
Knowledge Base: False Query: False
Evaluating: ( not False or not True or True) and ( not False and True) and False
Knowledge Base: False Query: True
Evaluating: ( not True or not True or False) and ( not True and True) and True
Knowledge Base: False Query: False
Evaluating: ( not True or not True or True) and ( not True and True) and True
Knowledge Base: False Query: True
Knowledge Base entails the query
Enter the rule: |
```

2. Create a knowledgebase using prepositional logic and prove the given query using resolution.

```
# Global variable kb (knowledge base)
kb = []

# Reset kb to an empty list
def Clear():
    global kb
    kb = []

# Insert sentence to the kb
def AddSentence(sentence):
    global kb
    # If the sentence is a clause, insert directly.
    if isClause(sentence):
        kb.append(sentence)
    # If not, convert to CNF, and then insert clauses one by one.
    else:
        sentenceCNF = convertCNF(sentence)
        if not sentenceCNF:
            print("Illegal input")
            return
        # Insert clauses one by one when there are multiple clauses
        if isAndList(sentenceCNF):
            for s in sentenceCNF[1:]:
                kb.append(s)
        else:
            kb.append(sentenceCNF)

# 'Query' the kb whether a sentence is True or not
def Query(sentence):
    global kb
    # Negate the sentence, and convert it to CNF accordingly.
    if isClause(sentence):
        neg = negation(sentence)
    else:
        sentenceCNF = convertCNF(sentence)
        if not sentenceCNF:
            print("Illegal input")
            return
        neg = convertCNF(negation(sentenceCNF))
```



```

# Insert individual clauses that we need to ask to ask_list.
ask_list = []
if isAndList(neg):
    for n in neg[1:]:
        nCNF = makeCNF(n)
        if type(nCNF).__name__ == 'list':
            ask_list.insert(0, nCNF)
        else:
            ask_list.insert(0, nCNF)
    else:
        ask_list = [neg]
# Create a new list combining the asked sentence and kb.
# Resolution will happen between the items in the list.
clauses = ask_list + kb[:]

# Recursively conduct resolution between items in the clauses list
# until it produces an empty list or there's no more progress.
while True:
    new_clauses = []
    for c1 in clauses:
        for c2 in clauses:
            if c1 is not c2:
                resolved = resolve(c1, c2)
                if resolved == False:
                    continue
                if resolved == []:
                    return True
                new_clauses.append(resolved)

    if len(new_clauses) == 0:
        return False
    new_in_clauses = True
    for n in new_clauses:
        if n not in clauses:
            new_in_clauses = False
            clauses.append(n)
    if new_in_clauses:
        return False
    return False
# Conduct resolution on two CNF clauses.
def resolve(arg_one, arg_two):
    resolved = False

```

```

s1 = make_sentence(arg_one)
s2 = make_sentence(arg_two)

resolve_s1 = None
resolve_s2 = None

# Two for loops that iterate through the two clauses.
for i in s1:
    if isNotList(i):
        a1 = i[1]
        a1_not = True
    else:
        a1 = i
        a1_not = False

    for j in s2:
        if isNotList(j):
            a2 = j[1]
            a2_not = True
        else:
            a2 = j
            a2_not = False

        # cancel out two literals such as 'a' & ['not', 'a']
        if a1 == a2:
            if a1_not != a2_not:
                # Return False if resolution already happend
                # but contradiction still exists.
                if resolved:
                    return False
                else:
                    resolved = True
                    resolve_s1 = i
                    resolve_s2 = j
                    break
            # Return False if not resolution happened
        if not resolved:
            return False
# Remove the literals that are canceled
s1.remove(resolve_s1)
s2.remove(resolve_s2)

```

```

# # Remove duplicates
result = clear_duplicate(s1 + s2)

# Format the result.
if len(result) == 1:
    return result[0]
elif len(result) > 1:
    result.insert(0, 'or')

return result

# Prepare sentences for resolution.
def make_sentence(arg):
    if isLiteral(arg) or isNotList(arg):
        return [arg]
    if isOrList(arg):
        return clear_duplicate(arg[1:])
    return

# Clear out duplicates in a sentence.
def clear_duplicate(arg):
    result = []
    for i in range(0, len(arg)):
        if arg[i] not in arg[i+1:]:
            result.append(arg[i])
    return result

# Check whether a sentence is a legal CNF clause.
def isClause(sentence):
    if isLiteral(sentence):
        return True
    if isNotList(sentence):
        if isLiteral(sentence[1]):
            return True
        else:
            return False
    if isOrList(sentence):
        for i in range(1, len(sentence)):
            if len(sentence[i]) > 2:
                return False
            elif not isClause(sentence[i]):
                return False

```

```
    return True
return False
```

Check if a sentence is a legal CNF.

```
def isCNF(sentence):
    if isClause(sentence):
        return True
    elif isAndList(sentence):
        for s in sentence[1:]:
            if not isClause(s):
                return False
        return True
    return False
```

Negate a sentence.

```
def negation(sentence):
    if isLiteral(sentence):
        return ['not', sentence]
    if isNotList(sentence):
        return sentence[1]
```

DeMorgan:

```
if isAndList(sentence):
    result = ['or']
    for i in sentence[1:]:
        if isNotList(sentence):
            result.append(i[1])
        else:
            result.append(['not', sentence])
    return result
if isOrList(sentence):
    result = ['and']
    for i in sentence[:]:
        if isNotList(sentence):
            result.append(i[1])
        else:
            result.append(['not', i])
    return result
return None
```

Convert a sentence into CNF.

```
def convertCNF(sentence):
```

```
while not isCNF(sentence):
    if sentence is None:
        return None
    sentence = makeCNF(sentence)
return sentence
```

```
def makeCNF(sentence):
    if isLiteral(sentence):
        return sentence
```

```
if (type(sentence).__name__ == 'list'):
    operand = sentence[0]
    if isNotList(sentence):
        if isLiteral(sentence[1]):
            return sentence
        cnf = makeCNF(sentence[1])
        if cnf[0] == 'not':
            return makeCNF(cnf[1])
        if cnf[0] == 'or':
            result = ['and']
            for i in range(1, len(cnf)):
                result.append(makeCNF(['not', cnf[i]]))
            return result
        if cnf[0] == 'and':
            result = ['or']
            for i in range(1, len(cnf)):
                result.append(makeCNF(['not', cnf[i]]))
            return result
    return "False: not"
```

Implication Elimination:

```
if operand == 'implies' and len(sentence) == 3:
    return makeCNF(['or', ['not', makeCNF(sentence[1])], makeCNF(sentence[2])])
```

Biconditional Elimination:

```
if operand == 'biconditional' and len(sentence) == 3:
    s1 = makeCNF(['implies', sentence[1], sentence[2]])
    s2 = makeCNF(['implies', sentence[2], sentence[1]])
    return makeCNF(['and', s1, s2])
```

```
if isAndList(sentence):
    result = ['and']
```

```

for i in range(1, len(sentence)):
    cnf = makeCNF(sentence[i])
    # Distributivity:
    if isAndList(cnf):
        for i in range(1, len(cnf)):
            result.append(makeCNF(cnf[i]))
        continue
    result.append(makeCNF(cnf))
return result

```

```

if isOrList(sentence):
    result1 = ['or']
    for i in range(1, len(sentence)):
        cnf = makeCNF(sentence[i])
        # Distributivity:
        if isOrList(cnf):
            for i in range(1, len(cnf)):
                result1.append(makeCNF(cnf[i]))
            continue
        result1.append(makeCNF(cnf))
    # Associativity:
    while True:
        result2 = ['and']
        and_clause = None
        for r in result1:
            if isAndList(r):
                and_clause = r
                break

        # Finish when there's no more 'and' lists
        # inside of 'or' lists
        if not and_clause:
            return result1

        result1.remove(and_clause)

    for i in range(1, len(and_clause)):
        temp = ['or', and_clause[i]]
        for o in result1[1:]:
            temp.append(makeCNF(o))
        result2.append(makeCNF(temp))
    result1 = makeCNF(result2)

```

```
    return None
return None
```

Below are 4 functions that check the type of a variable

def isLiteral(item):

```
    if type(item).__name__ == 'str':
        return True
    return False
```

def isNotList(item):

```
    if type(item).__name__ == 'list':
        if len(item) == 2:
            if item[0] == 'not':
                return True
    return False
```

def isAndList(item):

```
    if type(item).__name__ == 'list':
        if len(item) > 2:
            if item[0] == 'and':
                return True
    return False
```

def isOrList(item):

```
    if type(item).__name__ == 'list':
        if len(item) > 2:
            if item[0] == 'or':
                return True
    return False
```

AddSentence(['and', 'p', 'q'])

AddSentence(['or', 'r', 's'])

print(Query(['and', ['or', 'p', 'r'], ['or', 'q', 's']]))

OUTPUT

Test Case 1:

```
C:\SHIVANSHU\WORK\Scripts\python.exe C:/Users/Dell/PycharmProjects/pythonProject/main.py
Enter the kb:
~Qv~PvR ~Q^P Q
Enter the query:
R

Step    | Clause | Derivation
-----
1. | ~Qv~PvR | Given.
2. | ~Q^P | Given.
3. | Q | Given.
4. | ~R | Negated conclusion.

Process finished with exit code 0
|
```

NEGATED CONCLUSION \rightarrow NULL SET \rightarrow PROVED.

3.

Implement unification in first order logic.

```
import re

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(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)
    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
```

```
print("\n\nTest Case 1:\n")  
exp1 = "knows(A,x)"  
exp2 = "knows(y,Y)"  
substitutions = unify(exp1, exp2)  
print("Substitutions:")  
print(substitutions)
```

```
print("\n\nTest Case 2:\n")  
exp1 = "knows(A,x)"  
exp2 = "knows(y,mother(y))"  
substitutions = unify(exp1, exp2)  
print("Substitutions:")  
print(substitutions)
```

OUTPUT

Enter the first expression

$P(x, y)$

Enter the second expression

$P(x, m)$

The substitutions are:

$[z / x, m / y]$

Process finished with exit code 0

4. Convert given first order logic statement into Conjunctive Normal Form (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-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(match[1],SKOLEM_CONSTANTS.pop(0))
    else:
        aU = [a for a in attributes if not a.islower()][0]
        statement = statement.replace(aU,
f'{SKOLEM_CONSTANTS.pop(0)}({match[1]})')
    return statement

import re

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
    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
        new_statement = '~' + statement[br:i] + '|' + statement[i+1:]
        statement = statement[:br] + new_statement if br > 0 else new_statement
    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], '~'
        statement = "".join(s)
    statement = statement.replace('~[∀','[~∀')

```

```

statement = statement.replace('~[ $\exists$ ','[ $\sim\exists$ ')
expr = '(~[ $\forall$ | $\exists$ ].)'
statements = re.findall(expr, statement)
for s in statements:
    statement = statement.replace(s, fol_to_cnf(s))
expr = '~\[[ $\wedge$ ]]+\]'
statements = re.findall(expr, statement)
for s in statements:
    statement = statement.replace(s, DeMorgan(s))
return statement

```

```

print("\n Test Case: 1")

```

```

print(Skolemization(fol_to_cnf("animal(y)<=>loves(x,y)")))

```

```

print("\n Test Case: 2")

```

```

print(Skolemization(fol_to_cnf(" $\forall x[\forall y[\text{animal}(y) \Rightarrow \text{loves}(x,y)]] \Rightarrow [\exists z[\text{loves}(z,x)]]$ ")))

```

```

print("\n Test Case: 3")

```

```

print(Skolemization(fol_to_cnf("[american(x)&weapon(y)&sells(x,y,z)&hostile(z)]=>criminal(x)")))

```

```

print("\n \n ")

```

OUTPUT

```
C:\SHIVANSHUWORK\Scripts\python.exe C:/Users/Dell/PycharmProjects/pythonProject/foltcnf.py
```

```
Enter FOL:
```

```
animal(y) <=> loves(x,y)
```

```
The CNF form of the given FOL is:
```

```
[~animal(y) V loves(x,y)] ^ [~loves(x,y) V animal(y)]
```

```
Process finished with exit code 0
```

```
|
```


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

```
import re
```

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

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

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

```
class Fact:  
    def __init__(self, expression):  
        self.expression = expression  
        predicate, params = self.splitExpression(expression)  
        self.predicate = predicate  
        self.params = params  
        self.result = any(self.getConstants())  
  
    def splitExpression(self, expression):  
        predicate = getPredicates(expression)[0]  
        params = getAttributes(expression)[0].strip('(').split(',')  
        return [predicate, params]  
  
    def getResult(self):  
        return self.result  
  
    def getConstants(self):  
        return [None if isVariable(c) else c for c in self.params]  
  
    def getVariables(self):  
        return [v if isVariable(v) else None for v in self.params]  
  
    def substitute(self, constants):  
        c = constants.copy()
```

```

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

```

```

class Implication:

```

```

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

```

```

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

```

```

class KB:

```

```

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

```

```

    def tell(self, e):
        if '=>' in e:
            self.implications.add(Implication(e))
        else:
            self.facts.add(Fact(e))
        for i in self.implications:
            res = i.evaluate(self.facts)

```

```

        if res:
            self.facts.add(res)

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

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

print("\n\n Test Case 1:")

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()

print("\n\n Test Case 2:")

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

```
Enter KB: (enter e to exit)
missile(x)=>weapon(x)
missile(M1)
enemy(x,America)=>hostile(x)
american(West)
enemy(Nono,America)
owns(Nono,M1)
missile(x)&owns(Nono,x)=>sells(West,x,Nono)
american(x)&weapon(y)&sells(x,y,z)&hostile(z)=>criminal(x)
e
Enter Query:
criminal(x)
Querying criminal(x):
All facts:
1. kb.tell('owns(Nono,M1)')
2. kb.tell('enemy(Nono,America)')
3. kb.tell('missile(M1)')
4. kb.tell('american(West)')
5. sells(West,x,Nono)
6. kb.query('criminal(x)')
7. hostile(x)

Process finished with exit code 0
|
```