NAME: Ritvika Singh Rathore

USN: 1BM19CS131

SUBJECT: ARTIFICAL INTILLEGENGE

SECTION: C

Program 6

Create a knowledgebase using prepositional 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("Enter rule: "))

    q = input("Enter the Query: ")

def entailment():

    global kb, q

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

    print('kb','alpha')

    print('*'*10)

    for comb in combinations:

        s = evaluatePostfix(toPostfix(kb), comb)

        f = evaluatePostfix(toPostfix(q), comb)

        print(s, f)

        print('-'*10)

        if s and not f:

            return False

    return True

def isOperand(c):

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

```python
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
#Test 1
input_rules()
ans = entailment()
if ans:
    print("The Knowledge Base entails query")
else:
    print("The Knowledge Base does not entail query")
```

```
Enter rule:
(pvq)^(~rvp)
Enter the Query:
r
**********Truth Table Reference**********
kb alpha
**********
True True
----------
True False
----------
The Knowledge Base does not entail query
```

Program 7

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

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


def reverse(clause):
    if len(clause) > 2:
        t = split_terms(clause)
        return f'{t[1]}v{t[0]}'
    return ''
def split_terms(rule):
    exp = '(~*[PQRS])'
    terms = re.findall(exp, rule)
    return terms
def contradiction(query, clause):
    contradictions = [ f'{query}v{negate(query)}', f'{negate(query)}v{query}']
    return clause in contradictions or reverse(clause) in contradictions
def resolve(kb, query):
    temp = kb.copy()
    temp += [negate(query)]
    steps = dict()
    for rule in temp:
```

```
        steps[rule] = 'Given.'
    steps[negate(query)] = 'Negated conclusion.'
    i = 0
    while i < len(temp):
        n = len(temp)
        j = (i + 1) % n
        clauses = []
        while j != i:
            terms1 = split_terms(temp[i])
            terms2 = split_terms(temp[j])
            for c in terms1:
                if negate(c) in terms2:
                    t1 = [t for t in terms1 if t != c]
                    t2 = [t for t in terms2 if t != negate(c)]
                    gen = t1 + t2
                    if len(gen) == 2:
                        if gen[0] != negate(gen[1]):
                            clauses += [f'{gen[0]}v{gen[1]}']
                        else:
                            if contradiction(query,f'{gen[0]}v{gen[1]}'):
                                temp.append(f'{gen[0]}v{gen[1]}')
                                steps[''] = f"Resolved {temp[i]} and {temp[j]} to {temp[-1]}, which is in turn
null. \
                                \nA contradiction is found when {negate(query)} is assumed as true. Hence,
{query} is true."
                                return steps
                    elif len(gen) == 1:
                        clauses += [f'{gen[0]}']
                    else:
                        if contradiction(query,f'{terms1[0]}v{terms2[0]}'):
                            temp.append(f'{terms1[0]}v{terms2[0]}')
                            steps[''] = f"Resolved {temp[i]} and {temp[j]} to {temp[-1]}, which is in turn null. \
                            \nA contradiction is found when {negate(query)} is assumed as true. Hence, {query}
is true."
                            return steps
```

```
            for clause in clauses:
                if clause not in temp and clause != reverse(clause) and reverse(clause) not in temp:
                    temp.append(clause)
                    steps[clause] = f'Resolved from {temp[i]} and {temp[j]}.'
            j = (j + 1) % n
        i += 1
    return steps
def resolution(kb, query):
    kb = kb.split(' ')
    steps = resolve(kb, query)
    print('\nStep\t|Clause\t|Derivation\t')
    print('-' * 30)
    i = 1
    for step in steps:
        print(f' {i}.\t| {step}\t| {steps[step]}\t')
        i += 1
def main():
    print("Enter the kb:")
    kb = input()
    print("Enter the query:")
    query = input()
    resolution(kb,query)
#test 1
#(P^Q)<=>R : (Rv~P)v(Rv~Q)^(~RvP)^(~RvQ)
main()
#test 2
#(P=>Q)=>Q, (P=>P)=>R, (R=>S)=>~(S=>Q)
```

```
Enter the kb:
PVQ PVR ~PVR RVS RV~Q ~SV~Q
Enter the query:
R

Step    |Clause |Derivation
-----------------------------
 1.     | PVQ   | Given.
 2.     | PVR   | Given.
 3.     | ~PVR  | Given.
 4.     | RVS   | Given.
 5.     | RV~Q  | Given.
 6.     | ~SV~Q | Given.
 7.     | ~R    | Negated conclusion.
 8.     | QvR   | Resolved from PVQ and ~PVR.
 9.     | PvR   | Resolved from PVQ and RV~Q.
 10.    | Pv~S  | Resolved from PVQ and ~SV~Q.
 11.    | P     | Resolved from PVR and ~R.
 12.    | ~P    | Resolved from ~PVR and ~R.
 13.    | Rv~S  | Resolved from ~PVR and Pv~S.
 14.    | R     | Resolved from ~PVR and P.
 15.    | Rv~Q  | Resolved from RVS and ~SV~Q.
 16.    | S     | Resolved from RVS and ~R.
 17.    | ~Q    | Resolved from RV~Q and ~R.
 18.    | Q     | Resolved from ~R and QvR.
 19.    | ~S    | Resolved from ~R and Rv~S.
 20.    |       | Resolved ~R and R to ~RvR, which is in turn null.
A contradiction is found when ~R is assumed as true. Hence, R is true.
```

Program 8

Implement unification in first order logic

```
import re
def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression.split(")")[:-1]
    expression = ")".join(expression)
    attributes = expression.split(',')
    return attributes


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)
    predicate = getInitialPredicate(exp)
    for index, val in enumerate(attributes):
        if val == old:
            attributes[index] = new
    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:
            print(f"{exp1} and {exp2} are constants. Cannot be unified")
            return []

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

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

    if isVariable(exp1):
        return [(exp2, exp1)] if not checkOccurs(exp1, exp2) else []

    if isVariable(exp2):
        return [(exp1, exp2)] if not checkOccurs(exp2, exp1) else []

    if getInitialPredicate(exp1) != getInitialPredicate(exp2):
        print("Cannot be unified as the predicates do not match!")
        return []

    attributeCount1 = len(getAttributes(exp1))
    attributeCount2 = len(getAttributes(exp2))
    if attributeCount1 != attributeCount2:
        print(f"Length of attributes {attributeCount1} and {attributeCount2} do not match. Cannot be
unified")
        return []

    head1 = getFirstPart(exp1)
```

```
    head2 = getFirstPart(exp2)
    initialSubstitution = unify(head1, head2)
    if not initialSubstitution:
        return []
    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 []

    return initialSubstitution + remainingSubstitution
def main():
    print("Enter the first expression")
    e1 = input()
    print("Enter the second expression")
    e2 = input()
    substitutions = unify(e1, e2)
    print("The substitutions are:")
    print([' / '.join(substitution) for substitution in substitutions])
main()
print(" ")
print("------------------- ")
print(" ")
main()
print(" ")
print("------------------- ")
```

```
print(" ")
main()
print(" ")
print("------------------- ")
print(" ")
main()
print("------------------------- ")
print("--------------------------")
```

```
Enter the first expression
know(f(x),y)
Enter the second expression
kows(J,John)
Cannot be unified as the predicates do not match!
The substitutions are:
[]

-------------------

Enter the first expression
knows(f(x),y)
Enter the second expression
knows(J,John)
The substitutions are:
['J / f(x)', 'John / y']
```

Program 9

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

```
import re


print("Enter FOL")
def remove_brackets(source, id):
    reg = '\(([^\(]*?)\)'
    m = re.search(reg, source)
    if m is None:
        return None, None
    new_source = re.sub(reg, str(id), source, count=1)
    return new_source, m.group(1)
```

```
class logic_base:
    def __init__(self, input):
        self.my_stack = []
        self.source = input
        final = input
        while 1:
            input, tmp = remove_brackets(input, len(self.my_stack))
            if input is None:
                break
            final = input
            self.my_stack.append(tmp)
        self.my_stack.append(final)

    def get_result(self):
        root = self.my_stack[-1]
        m = re.match('\s*([0-9]+)\s*$', root)
        if m is not None:
            root = self.my_stack[int(m.group(1))]
        reg = '(\d+)'
        while 1:
            m = re.search(reg, root)
            if m is None:
                break
            new = '(' + self.my_stack[int(m.group(1))] + ')'
            root = re.sub(reg, new, root, count=1)
        return root

    def merge_items(self, logic):
        reg0 = '(\d+)'
        reg1 = 'neg\s+(\d+)'
        flag = False
        for i in range(len(self.my_stack)):
```

```
            target = self.my_stack[i]
            if logic not in target:
                continue
            m = re.search(reg1, target)
            if m is not None:
                continue
            m = re.search(reg0, target)
            if m is None:
                continue
            for j in re.findall(reg0, target):
                child = self.my_stack[int(j)]
                if logic not in child:
                    continue
                new_reg = "(^|\s)" + j + "(\s|$)"
                self.my_stack[i] = re.sub(new_reg, ' ' + child + ' ', self.my_stack[i], count=1)
                self.my_stack[i] = self.my_stack[i].strip()
                flag = True
        if flag:
            self.merge_items(logic)



class ordering(logic_base):
    def run(self):
        flag = False
        for i in range(len(self.my_stack)):
            new_source = self.add_brackets(self.my_stack[i])
            if self.my_stack[i] != new_source:
                self.my_stack[i] = new_source
                flag = True
        return flag

    def add_brackets(self, source):
        reg = "\s+(and|or|imp|iff)\s+"
        if len(re.findall(reg, source)) < 2:
```

```
        return source
      reg_and = "(neg\s+)?\S+\s+and\s+(neg\s+)?\S+"
      m = re.search(reg_and, source)
      if m is not None:
        return re.sub(reg_and, "(" + m.group(0) + ")", source, count=1)
      reg_or = "(neg\s+)?\S+\s+or\s+(neg\s+)?\S+"
      m = re.search(reg_or, source)
      if m is not None:
        return re.sub(reg_or, "(" + m.group(0) + ")", source, count=1)
      reg_imp = "(neg\s+)?\S+\s+imp\s+(neg\s+)?\S+"
      m = re.search(reg_imp, source)
      if m is not None:
        return re.sub(reg_imp, "(" + m.group(0) + ")", source, count=1)
      reg_iff = "(neg\s+)?\S+\s+iff\s+(neg\s+)?\S+"
      m = re.search(reg_iff, source)
      if m is not None:
        return re.sub(reg_iff, "(" + m.group(0) + ")", source, count=1)


class replace_iff(logic_base):
  def run(self):
    final = len(self.my_stack) - 1
    flag = self.replace_all_iff()
    self.my_stack.append(self.my_stack[final])
    return flag

  def replace_all_iff(self):
    flag = False
    for i in range(len(self.my_stack)):
      ans = self.replace_iff_inner(self.my_stack[i], len(self.my_stack))
      if ans is None:
        continue
      self.my_stack[i] = ans[0]
      self.my_stack.append(ans[1])
```

```
            self.my_stack.append(ans[2])
            flag = True
        return flag

    def replace_iff_inner(self, source, id):
        reg = '^(.*?)\s+iff\s+(.*?)$'
        m = re.search(reg, source)
        if m is None:
            return None
        a, b = m.group(1), m.group(2)
        return (str(id) + ' and ' + str(id + 1), a + ' imp ' + b, b + ' imp ' + a)


class replace_imp(logic_base):
    def run(self):
        flag = False
        for i in range(len(self.my_stack)):
            ans = self.replace_imp_inner(self.my_stack[i])
            if ans is None:
                continue
            self.my_stack[i] = ans
            flag = True
        return flag

    def replace_imp_inner(self, source):
        reg = '^(.*?)\s+imp\s+(.*?)$'
        m = re.search(reg, source)
        if m is None:
            return None
        a, b = m.group(1), m.group(2)
        if 'neg ' in a:
            return a.replace('neg ', '') + ' or ' + b
        return 'neg ' + a + ' or ' + b
```

```
class de_morgan(logic_base):
    def run(self):
        reg = 'neg\s+(\d+)'
        flag = False
        final = len(self.my_stack) - 1
        for i in range(len(self.my_stack)):
            target = self.my_stack[i]
            m = re.search(reg, target)
            if m is None:
                continue
            flag = True
            child = self.my_stack[int(m.group(1))]
            self.my_stack[i] = re.sub(reg, str(len(self.my_stack)), target, count=1)
            self.my_stack.append(self.doing_de_morgan(child))
            break
        self.my_stack.append(self.my_stack[final])
        return flag

    def doing_de_morgan(self, source):
        items = re.split('\s+', source)
        new_items = []
        for item in items:
            if item == 'or':
                new_items.append('and')
            elif item == 'and':
                new_items.append('or')
            elif item == 'neg':
                new_items.append('neg')
            elif len(item.strip()) > 0:
                new_items.append('neg')
                new_items.append(item)
        for i in range(len(new_items) - 1):
            if new_items[i] == 'neg':
```

```
                if new_items[i + 1] == 'neg':
                    new_items[i] = ''
                    new_items[i + 1] = ''
        return ' '.join([i for i in new_items if len(i) > 0])



class distributive(logic_base):
    def run(self):
        flag = False
        reg = '(\d+)'
        final = len(self.my_stack) - 1
        for i in range(len(self.my_stack)):
            target = self.my_stack[i]
            if 'or' not in self.my_stack[i]:
                continue
            m = re.search(reg, target)
            if m is None:
                continue
            for j in re.findall(reg, target):
                child = self.my_stack[int(j)]
                if 'and' not in child:
                    continue
                new_reg = "(^|\s)" + j + "(\s|$)"
                items = re.split('\s+and\s+', child)
                tmp_list = [str(j) for j in range(len(self.my_stack), len(self.my_stack) + len(items))]
                for item in items:
                    self.my_stack.append(re.sub(new_reg, ' ' + item + ' ', target).strip())
                self.my_stack[i] = ' and '.join(tmp_list)
                flag = True
            if flag:
                break
        self.my_stack.append(self.my_stack[final])
        return flag
```

```
class simplification(logic_base):
    def run(self):
        old = self.get_result()
        for i in range(len(self.my_stack)):
            self.my_stack[i] = self.reducing_or(self.my_stack[i])
        # self.my_stack[i] = self.reducing_and(self.my_stack[i])
        final = self.my_stack[-1]
        self.my_stack[-1] = self.reducing_and(final)
        return len(old) != len(self.get_result())

    def reducing_and(self, target):
        if 'and' not in target:
            return target
        items = set(re.split('\s+and\s+', target))
        for item in list(items):
            if ('neg ' + item) in items:
                return ''
            if re.match('\d+$', item) is None:
                continue
            value = self.my_stack[int(item)]
            if self.my_stack.count(value) > 1:
                value = ''
                self.my_stack[int(item)] = ''
            if value == '':
                items.remove(item)
        return ' and '.join(list(items))

    def reducing_or(self, target):
        if 'or' not in target:
            return target
        items = set(re.split('\s+or\s+', target))
        for item in list(items):
            if ('neg ' + item) in items:
```

```
            return ''
        return ' or '.join(list(items))


def merging(source):
    old = source.get_result()
    source.merge_items('or')
    source.merge_items('and')
    return old != source.get_result()


def run(input):
    all_strings = []
    # all_strings.append(input)
    zero = ordering(input)
    while zero.run():
        zero = ordering(zero.get_result())
    merging(zero)

    one = replace_iff(zero.get_result())
    one.run()
    all_strings.append(one.get_result())
    merging(one)

    two = replace_imp(one.get_result())
    two.run()
    all_strings.append(two.get_result())
    merging(two)

    three, four = None, None
    old = two.get_result()
    three = de_morgan(old)
    while three.run():
        pass
```

```
    all_strings.append(three.get_result())
    merging(three)
    three_helf = simplification(three.get_result())
    three_helf.run()

    four = distributive(three_helf.get_result())
    while four.run():
        pass
    merging(four)
    five = simplification(four.get_result())
    five.run()
    all_strings.append(five.get_result())
    return all_strings

inputs = input().split('\n')
for input in inputs:
    for item in run(input):
        print(item)
```

```
Enter FOL
(animal(z) and kills(x,z)) imp (neg Loves(y,z))
(animal(z) and kills(x,z)) imp (neg Loves(y,z))
neg (animal(z) and kills(x,z)) or (neg Loves(y,z))
(neg animal(z) or neg kills(x,z)) or (neg Loves(y,z))
neg animal(z) or (neg Loves(y,z)) or neg kills(x,z)
```

Program 10

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}')



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)')



kb_.display()
```

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