Sweb Assignment 2

Q1.a:

Ans 1.(a)

Given FOL :-

$F = ((A \wedge B) \Rightarrow C) \Rightarrow ((A \Rightarrow C) \vee (B \Rightarrow C))$

A	B	C	F	
0	0	0	1	A=0 then A⇒C True so F = True
0	0	1	1	
0	1	0	1	
0	1	1	1	
1	0	0	1	B=0 · B⇒C True F = True
1	0	1	1	
1	1	0	1	A=1 B=1 C=0 So ((A∧B)⇒C) = False so F = True
1	1	1	1	C = True. F = True.

DNF = $(\neg A \wedge \neg B \wedge \neg C) \vee (\neg A \wedge \neg B \wedge C) \vee (\neg A \wedge B \wedge \neg C) \vee (\neg A \wedge B \wedge C) \vee (A \wedge \neg B \wedge \neg C) \vee (A \wedge \neg B \wedge C) \vee (A \wedge B \wedge \neg C) \vee (A \wedge B \wedge C)$

all 8 terms will present

Q1b:

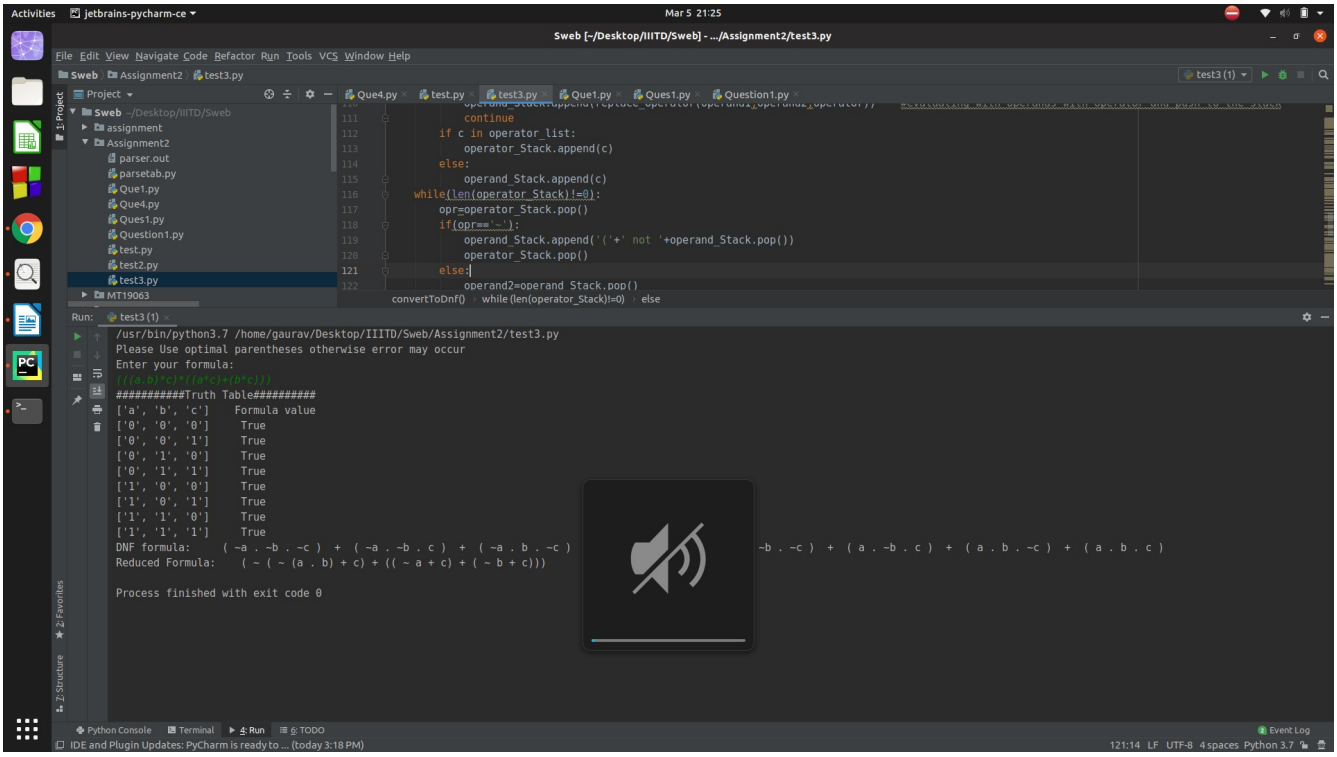
```
def replace_operator(operand1,operand2,operator): # changing the
operator
    if  operator =='.':
        return ' ( '+operand1 +' and '+operand2+' ) '
    elif  operator =='+':
        return ' ( '+operand1 +' or '+operand2+' ) '
    elif  operator =='*':
        return ' ( '+' not '+operand1 +' or '+operand2+' ) '
    elif  operator =='==':
        return ' ( ( '+' not '+operand1 +' or '+operand2+' ) '+' and
'+' ( '+' not '+operand2 +' or '+operand1+' ) ) '
def convert_dnf_natural(operand_Stack):  #to convert eval() known
operators
    dnf_formula_chars=[]
    for ch in (operand_Stack[0].split()):
        # print(ch)
        if ch=='not':
            dnf_formula_chars.append(' ~ ')
        elif ch=='or':
            dnf_formula_chars.append(' + ')
        elif ch=='and':
            dnf_formula_chars.append(' . ')
        else:
            dnf_formula_chars.append(ch)
    str=""
    for x in dnf_formula_chars:
        str=str+x
    return str
def convert_dnf_valued(operand_Stack,list_operands,bits):  #converting
to eval known form and replacing all operands to their bits values
    dnf_formula_chars=[]
    for ch in (operand_Stack[0].split()):
        # print(ch)
        if ch=='not':
            dnf_formula_chars.append(' not ')
        elif ch=='or':
            dnf_formula_chars.append(' or ')
        elif ch=='and':
            dnf_formula_chars.append(' and ')
        elif ch=='(':
            dnf_formula_chars.append(' ( ')
        elif ch==')':
            dnf_formula_chars.append(' ) ')
        else:
```

```
            ch_index=list_operands.index(ch)
            if(bits[ch_index]=='0'):
                dnf_formula_chars.append(" False ")
            else:
                dnf_formula_chars.append(" True ")

    str=""
    for x in dnf_formula_chars:
        str=str+x
    return str
def find_product(bits,list_operands):   #finding single product
    product=[]
    product.append( ' ( ')
    for i in range(len(bits)):
        if bits[i] == '0':
            product.append('~'+list_operands[i])
        else:
            product.append(list_operands[i])
        if(i<len(bits)-1):
            product.append(' . ')
    product.append(' ) ')
    return product
def convertToDnf(formula):
    formula_chars = [char for char in formula]
#changing formula characters to a list
    l=len(formula_chars)
    if(formula_chars[0]!='('):
        formula_chars.append(')')
        formula_chars.insert(0,'(')

    k=0
    for i in range(l-1):
        if(formula_chars[i]=='~' and (formula_chars[i+1]!='(')):
#handling negetion
            formula_chars[i+1]='~'+formula_chars[i+1]
            del formula_chars[i]
            k=k+1
    operator_list=['~','.','+','*','==']
    operator_Stack = []   #defining stacks for operator and operands
    operand_Stack = []
    flag = 0
    temp_list=[]
    flag=0
    for i in range(len(formula_chars)):
        if formula_chars[i]=='=':
            flag=flag+1
            if flag==2:
                temp_list.append(formula_chars[i]+formula_chars[i])
                flag=0
            continue
```

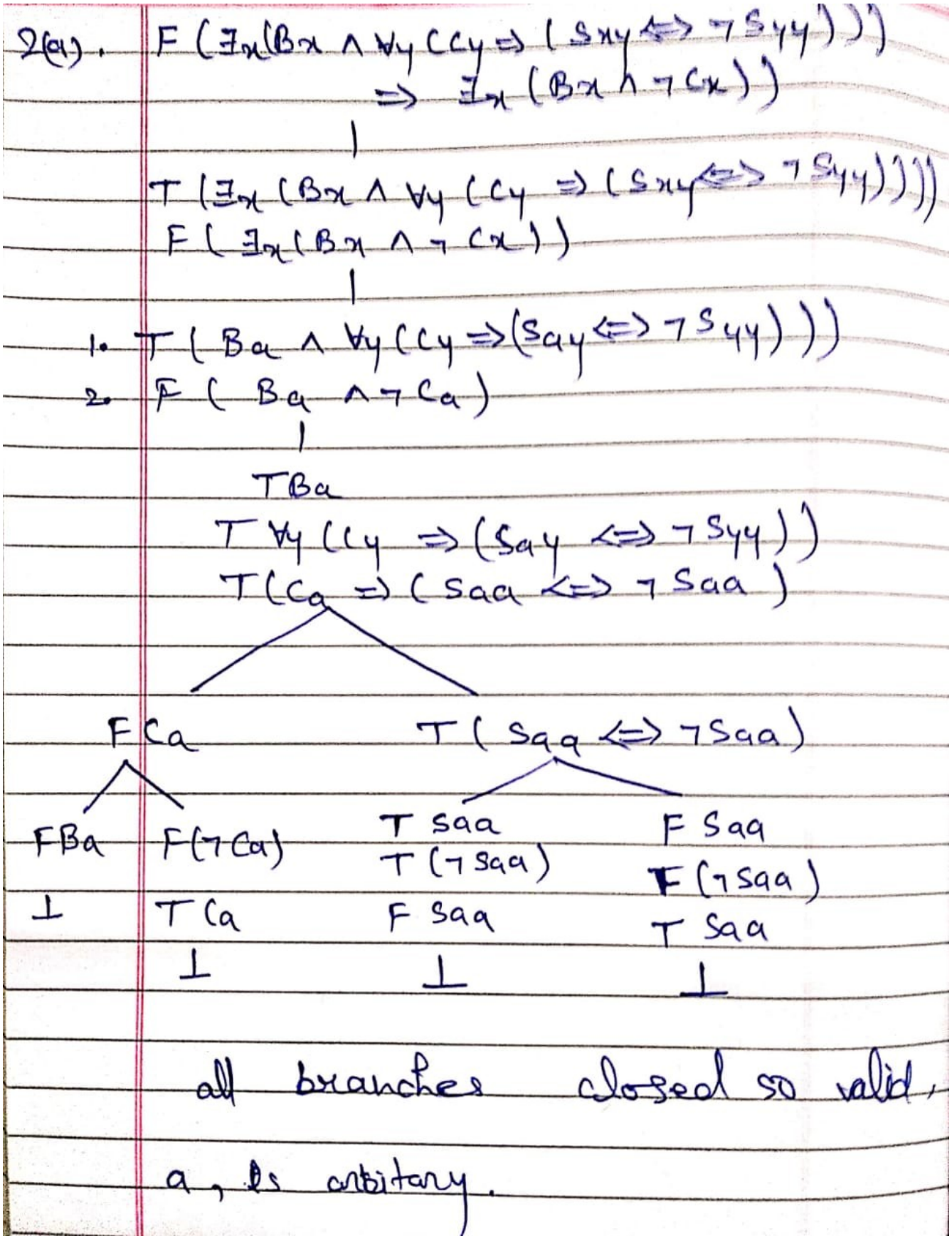
```
        temp_list.append(formula_chars[i])
    formula_chars=temp_list
    for i in range(len(formula_chars)):
        c = formula_chars[i]
        if (formula_chars[i] == '('):      #pushing '(' simply
            operator_Stack.append('(')
            continue
        elif(formula_chars[i]==')'):
#poping whwn ')' encounters
            if(operator_Stack[len(operator_Stack)-1]=='~'):
                operator_Stack.pop()
                operator_Stack.pop()
                val=operand_Stack.pop()
                operand_Stack.append('('+' not '+val+' ) ')
                continue
            operand2=operand_Stack.pop()
            operand1=operand_Stack.pop()
            operator=operator_Stack.pop()
            operator_Stack.pop()

operand_Stack.append(replace_operator(operand1,operand2,operator))
#evaluating with operands with operator and push to the stack
            continue
        if c in operator_list:
            operator_Stack.append(c)
        else:
            operand_Stack.append(c)
    while(len(operator_Stack)!=0):
        opr=operator_Stack.pop()
        if(opr=='~'):
            operand_Stack.append('('+' not '+operand_Stack.pop())
            operator_Stack.pop()
        else:
            operand2=operand_Stack.pop()
            operand1=operand_Stack.pop()
            operator_Stack.pop()

operand_Stack.append(replace_operator(operand1,operand2,opr))


##########################################################################
#####

    dnf_formula_chars=convert_dnf_natural(operand_Stack)    #converting
to reduced formula
    reduced_formula=dnf_formula_chars
    list_operands=[]
    for ch in formula_chars:
        if ch not in operator_list+['(',')']:
            list_operands.append(ch)
```

```
    list_operands=list(set(list_operands))
    list_operands.sort()                                    #all operand
list
    products=[]
    j=0
    for i in range (pow(2,len(list_operands))):       #running all bits
combinations to evaluate dnf using eval function
        bits="{0:b}".format(i)
        bits = [char for char in bits]
        l = len(list_operands) - len(bits)
        temp = ['0'] * l
        bits = temp + bits
        dnf_formula_chars1 =
convert_dnf_valued(operand_Stack,list_operands,bits)
        evaluation=str(eval(dnf_formula_chars1))
        if j==0:
            print("###########Truth Table##########")    #Generating
Truth table with eval function
            print(list_operands,"   Formula value ")
            j=j+1
        print(bits,"   ",evaluation)
        if  evaluation=='True':
            products =products+ find_product(bits,list_operands)
            products=products+[' + ']
    temp_dnf=""
    for t in products:
        temp_dnf=temp_dnf+t
    temp_dnf=temp_dnf[0:-2]
    temp_dnf_2 = [char for char in temp_dnf]
    for i in range (len(temp_dnf_2)):
        if(temp_dnf_2[i]=='~' and temp_dnf_2[i+1]=='~'):
            temp_dnf_2[i]=""
            temp_dnf_2[i+1]=""
    final_dnf=""
    for ch in temp_dnf_2:
        final_dnf=final_dnf+ch
    print("DNF formula:   ", final_dnf)
    print("Reduced Formula:   ", reduced_formula)

print("Please Use optimal parentheses otherwise error may occur")
print ("Enter your formula:")
formula=input()

convertToDnf(formula)
```

Output:


Activities
jetbrains-pycharm-ce
Mar 5 21:25
Sweb [~/Desktop/IIITD/Sweb] - .../Assignment2/test3.py
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Sweb Assignment2 test3.py
test3 (1)
Project
Sweb ~/Desktop/IIITD/Sweb
assignment
Assignment2
parser.out
parsetab.py
Que1.py
Que4.py
Ques1.py
Question1.py
test.py
test2.py
test3.py
MT19063
Que4.py
test.py
test3.py
Que1.py
Ques1.py
Question1.py
continue
if c in operator_list:
operator_Stack.append(c)
else:
operand_Stack.append(c)
while(len(operator_Stack)!=0):
opr=operator_Stack.pop()
if(opr=='~'):
operand_Stack.append('('+' not '+operand_Stack.pop())
operator_Stack.pop()
else:
operand2=operand_Stack.pop()
convertToDnf() while (len(operator_Stack)!=0) else
Run: test3 (1)
/usr/bin/python3.7 /home/gaurav/Desktop/IIITD/Sweb/Assignment2/test3.py
Please Use optimal parentheses otherwise error may occur
Enter your formula:
(((a.b)*c)*((a*c)+(b*c)))
##########Truth Table##########
['a', 'b', 'c'] Formula value
['0', '0', '0'] True
['0', '0', '1'] True
['0', '1', '0'] True
['0', '1', '1'] True
['1', '0', '0'] True
['1', '0', '1'] True
['1', '1', '0'] True
['1', '1', '1'] True
DNF formula: (~a . ~b . ~c) + (~a . ~b . c) + (~a . b . ~c)
~b . ~c) + (a . ~b . c) + (a . b . ~c) + (a . b . c)
Reduced Formula: (~ (~ (a . b) + c) + ((~ a + c) + (~ b + c)))
Process finished with exit code 0
Python Console Terminal 4: Run 6: TODO
Event Log
IDE and Plugin Updates: PyCharm is ready to ... (today 3:18 PM)
121:14 LF UTF-8 4 spaces Python 3.7

Q2:Ans:


2(a). F(∃x(Bx ∧ ∀y(Cy ⇒ (Sxy ⇔ ¬Syy))) ⇒ ∃x(Bx ∧ ¬Cx))
T(∃x(Bx ∧ ∀y(Cy ⇒ (Sxy ⇔ ¬Syy)))
F(∃x(Bx ∧ ¬Cx))
1. T(Ba ∧ ∀y(Cy ⇒ (Say ⇔ ¬Syy)))
2. F(Ba ∧ ¬Ca)
TBa
T∀y(Cy ⇒ (Say ⇔ ¬Syy))
T(Ca ⇒ (Saa ⇔ ¬Saa)
FCa
T(Saa ⇔ ¬Saa)
FBa
F(¬Ca)
⊥
TCa
⊥
T Saa
T(¬Saa)
F Saa
⊥
F Saa
F(¬Saa)
T Saa
⊥
all branches closed so valid,
a, is arbitary.

2(b)!

$F(\exists x(Ax \Rightarrow Bx)) \iff (\forall x Ax \Rightarrow \exists x Bx))$

2. $F(\exists x(Ax \Rightarrow Bx))$ 1. $T(\exists x(Ax \Rightarrow Bx))$

1. $T(\forall x(Ax) \Rightarrow \exists x Bx)$ 2. $F(\forall x Ax \Rightarrow \exists x Bx)$

$T(Aa \Rightarrow Ba)$

$F \forall x Ax$ $T \exists x Bx$ FAa TBa

FAa TBb $T \forall x Ax$ $T \forall x Ax$

$F(Aa \Rightarrow Ba)$ $F(Ab \Rightarrow Bb)$ $F \exists x Bx$ $F \exists x Bx$

TAa TAb TAa TAa

FBa FBb FBa FBa

all branches closed so valid

a, b are arbitary

2. c. $F((\exists x Ax \Rightarrow \exists x Bx) \Rightarrow (\forall x(Ax \Rightarrow Bx)))$

2. $T(\exists x Ax \Rightarrow \exists x Bx)$

1. $F(\forall x(Ax \Rightarrow Bx))$

$F(Aa \Rightarrow Ba)$

TAa

FBa

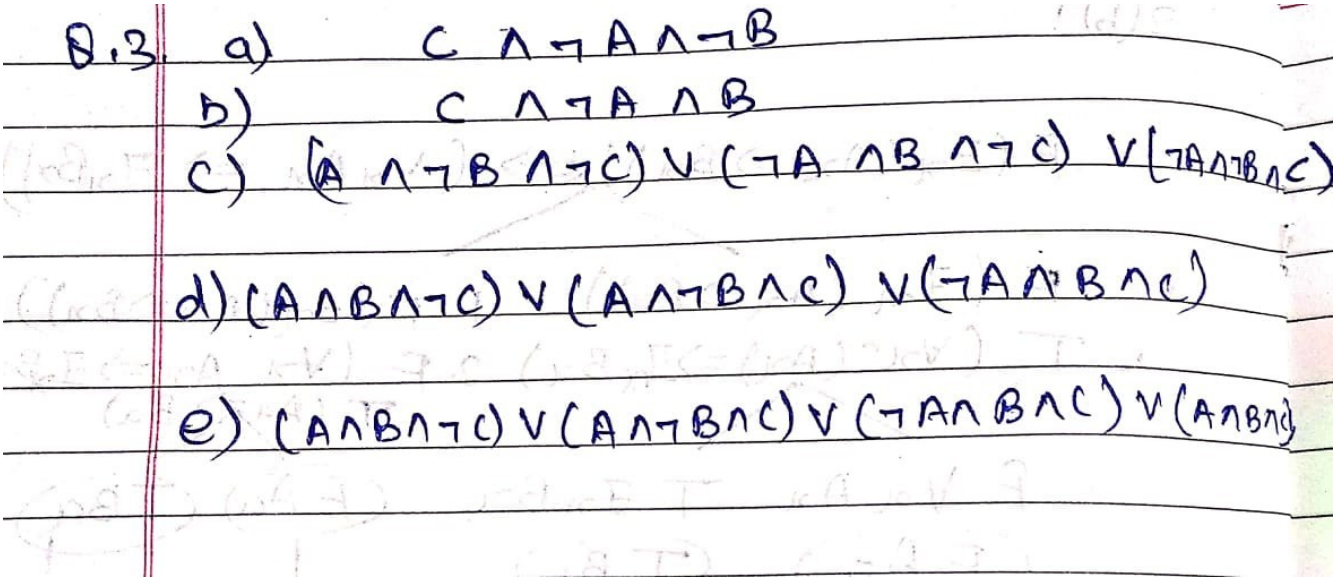
a, b are arbitary

$F(\exists x Ax)$ $T(\exists x Bx)$

FAa TBb

x not closed so not valid

Q3:Ans:



Q4.

```
def checkBranches(value,operator): #To check for a perticular operator
on a perticulat T/F how many branches should be there and what value of
left operand and right operand will take
    if(operator=='+'):
        if (value=='F'):
            return 1 ,'FF'
        else:
            return 2,'TT'
    elif(operator=='.'):
        if (value=='T'):
            return 1,'TT'
        else:
            return 2,'FF'
    elif (operator == '*'):
        if (value == 'T'):
            return 2,'FT'
        else:
            return 1,'TF'
    elif (operator == '='):
        if (value == 'T'):
            return 2,'00'
        else:
            return 2,'00'
def Tablue(expression_stack,l,t):     #Tablue call for expression
```

```
    # print("Fun ",t)
    end=True
    r=0
    for x in expression_stack:       # to check if all expression
changed to operand or not
        if len(x)==5 and x[2]=='~':
            if x[0]=='T':
                expression_stack.remove(expression_stack[r])
                expression_stack.insert(0,'F'+x[3:-1])
            else:
                expression_stack.remove(expression_stack[r])
                expression_stack.insert(0, 'T' + x[3:-1])

        if(len(expression_stack[r])!=2):
            end=False
        r=r+1
    print(expression_stack)
    if end:                                  # All the expression has
changed to expression
        expression_stack = list(set(expression_stack))
        i=0
        for x in expression_stack:
            # r=0
            for j in range(len(expression_stack)):
                if x[1]==(expression_stack[j])[1] and x[0]!
=(expression_stack[j])[0]:
                    return True        # return True if Contradiction
found
        return False                     # return false in branch open
    expression=expression_stack.pop()
    expression_temp = expression[2:-1]
    flag=0
    if expression_temp[0]=='~':                     # negation handling
        expression_temp=expression_temp[2:-1]
    count = 0
    i = 0
    for x in expression_temp:
        if (x == '('):
            count = count + 1
        elif (x == ')'):
            count = count - 1
        if (count == 0):
            break

        i = i + 1
    value = expression[0]
    operation = expression_temp[i + 1]
    no_of_branches, values = checkBranches(value, operation)  # to check
how many branches should be in the next step of tablue
    left_exp = expression_temp[0:i + 1]
```

```
    if (expression_temp[i + 1] == '='):
        right_exp = expression_temp[i + 3:]                #dividing the
left and right expression
    else:
        right_exp = expression_temp[i + 2:]
    expression_stack_temp = expression_stack.copy()
    if (left_exp[0] == '~'):
        if values[0] == 'F':
            left_exp_final = 'T' + left_exp[1:]
        else:
            left_exp_final = 'F' + left_exp[1:]
    else:
        left_exp_final = values[0] + left_exp
    if(right_exp[0]=='~'):
        if(values[1]=='F'):
            right_exp_final='T'+right_exp[1:]
        else:
            right_exp_final='F'+right_exp[1:]
    else:
        right_exp_final=values[1]+right_exp

    if no_of_branches == 1:          #When branch no is 1 push both
operands or expression into stack
        if (len(left_exp) == 1):
            m = 0
        else:
            m = len(expression_stack_temp)
        if (len(right_exp) == 1):
            n = 0
        else:
            n = len(expression_stack_temp)
        expression_stack_temp.insert(m, left_exp_final)
        expression_stack_temp.insert(n, right_exp_final)
        return Tablue(expression_stack_temp,l,t+1)

    elif no_of_branches == 2:    # When no of branches are 2 then push
each operand or expression accordingly
        expression_stack_temp1 = expression_stack.copy()
        expression_stack_temp2 = expression_stack.copy()
        if (len(left_exp) == 1):
            m = 0
        else:
            m = len(expression_stack_temp1)
            # m2=len(expression_stack_temp2)
        if (len(right_exp) == 1):

            n = 0
        else:
            n = len(expression_stack_temp1)
```

```
        if (operation == '='):   # specific handling for == operator
            if(left_exp[0]=='~'):
                expression_stack_temp1.insert(m, 'F' + left_exp)
                expression_stack_temp2.insert(m, 'T' + left_exp)
            else:
                expression_stack_temp1.insert(m, 'T' + left_exp)
                expression_stack_temp2.insert(m, 'F' + left_exp)
            if (right_exp[0]=='~'):
                expression_stack_temp1.insert(n, 'T' + right_exp)
                expression_stack_temp2.insert(n, 'F' + right_exp)

            else:
                expression_stack_temp1.insert(n, 'F' + right_exp)
                expression_stack_temp2.insert(n, 'T' + right_exp)

        else:
            expression_stack_temp1.insert(m, left_exp_final)
            expression_stack_temp2.insert(n, right_exp_final)
        return Tablue(expression_stack_temp1,l,t+1) and
Tablue(expression_stack_temp2,l,t+1)

def normalizeformula(formula):  #to make formula  formated
    count=0
    i=0
    for x in formula:
        if(x=='('):
            count=count+1
        elif(x==')'):
            count=count-1
        # print(count)
        if count==0:
            break
        i=i+1
    if i!=len(formula)-1:
        formula='('+formula+')'
    return formula




def find_validity(formula,consequence,l):  #main call for validiti
finding
    formula=normalizeformula(formula)
    consequence=normalizeformula(consequence)
    expression_stack=[]
    if len(consequence)==1:    #giving truth value to expression and
push it into stack
        expression_stack.append("F" + consequence)
        expression_stack.append("T" + formula)
    else:
```

```
            expression_stack.append("T" + formula)
            expression_stack.append("F" + consequence)
        t=0
        if(Tablue(expression_stack,l,t)):
            print("Yes")
        else:
            print("N0")
print("Please Use optimal parentheses otherwise error may occur") #for
input
print ("Enter your formula :")

formula=input()
print ("Enter your consequence :")
consequence=input()
operands=[]
for x in formula:
    if x not in ['+','.','*','(',')','~']:
        operands.append(x)
find_validity(formula,consequence,len(list(set(operands))))
```

Output: