ASSIGNMENT-2 : REPORT

Q1)

```
## Semantic Web - Winter 2020 : Assignment 2
## MT19123 - Nikhil Kolla
##For Convenience, the Boolean symbols are represented as
# '+' for Disjunction
# '.' for Conjunction
# '`' for Negation
# '*' for Implication
# '$' for Bi-implication
#storing the literals
# literals_list = ['p','q','r']
# no_of_literals = len(literals_list)
literals_list = []
no_of_literals = int(input("Enter number of literals you need:- "))
for i in range(0,no_of_literals):
        ele = input("Enter literal:- ")
        literals_list.append(ele)
# print("The literals of the expression are:- ")
# print(literals_list)

## Reference for generating initial Truth Table:
## https://www.youtube.com/watch?v=rf30vfA7NTA
#print(literals_list)

#calculating the number of rows to be in the truth table
no_of_rows = 2**len(literals_list)
#print("number of rows in truth table is: ",no_of_rows)

#generating the initial truth table

initial_tt = []

#iterating for number_of_rows and generating each row at a time
for i in range(no_of_rows):
        #appending zeros for ensuring each row would be of same length
        #print("The binary number for {0} is {1}".format(i,bin(i)[2:].zfill(len(literals_list))))
```

```
    bit_number = bin(i)[2:].zfill(len(literals_list))
    #generating a single row
    dummy_row = []
    for l in str(bit_number):
        if (l == '0'):
            dummy_row.append(False)
        else:
            dummy_row.append(True)
    #appending each generated row to initial list of truth table
    initial_tt.append(dummy_row)
# print("initial Truth Table: ")
# print(initial_tt)


##Functions to calculate Truth Values

##Negation
def negation(sym):
    return not sym

##Disjunction
def disjunction(sym1,sym2):
    if sym1 == True or sym2 == True:
        return True
    else:
        return False

##Conjunction
def conjunction(sym1,sym2):
    if sym1 == True and sym2 == True:
        return True
    else:
        return False

##Implication
def implication(sym1,sym2):
    if sym1 == True and sym2 == False:
        return False
```

```
	else:
		return True

##Bi-implication
def biImplication(sym1,sym2):
	if sym1 == sym2:
		return True
	else:
		return False

# print(biImplication(True,True))

##Class Definition for converting Infix expression to Postfix expression
##Reference:-
## geeksforgeeks.org/stack-set-2-infix-to-postfix/
class InfixToPostfix:

	##For initialising the class variables
	def __init__(self,capacityOfStack):
		self.topOfStack = -1
		self.capacityOfStack = capacityOfStack
		#stack for conversion
		self.stackArray = []
		#for storing the output_expr
		self.output_expr = []
		#for precedent setting
		self.op_precedence = {'+':1, '.':1, '*':2, '$':2, '`':3}

	##For checking whether the stack is empty or not
	def isEmpty(self):
		return True if self.topOfStack ==-1 else False

	##For getting the top value of the stack
	def peek(self):
		return self.stackArray[-1]

	##For removing the top most element of the stack
	def pop(self):
		#checking whether the stack is empty or not
```

```
        if not self.isEmpty():
            self.topOfStack -= 1
            return self.stackArray.pop()
        #if stack is empty return this character which says that the
        #stack is empty
        else:
            return "$"

    ##For inserting elements into the Stack
    def push(self,op):
        #incrementing the value of top indeed increments the capacity of stack
        self.topOfStack +=1
        #inserting new element
        self.stackArray.append(op)


    ##Checking whether the character passed is Operand or not
    def isOperand(self,ch):
        return ch.isalpha()


    ##Checking whether the operator precedence is strictly less than the
    ##operator present at the top of the stack
    def notGreater(self,i):
        try:
            a = self.op_precedence[i]
            b = self.op_precedence[self.peek()]
            return True if a <= b else False
        except KeyError:
            return False

    ##Function to convert an infix expression to
    ##Postfix expression
    def infixToPostfix(self,exp):

        #Traversing through the expression
        for i in exp:
            if self.isOperand(i):
                self.output_expr.append(i)
```

```
            #if the current symbol is open parenthesis we are pushing it
            #on to the stack
            elif i == '(':
                self.push(i)

            #when current symbol is closed parenthesis
            elif i == ')':
                #popping all the elements of the stack still we encounter a
                #open parenthesis
                while ((not self.isEmpty()) and self.peek() != '('):
                    a = self.pop()
                    self.output_expr.append(a)
                if (not self.isEmpty() and self.peek() != '('):
                    return -1
                else:
                    self.pop()

            else:
                while(not self.isEmpty() and self.notGreater(i)):
                    self.output_expr.append(self.pop())
                self.push(i)

        while not self.isEmpty():
            self.output_expr.append(self.pop())

        #printing the final converted expression
        print(self.output_expr)
        return self.output_expr


##Evaluation of Postfix expression
##Reference for evaluating postfix expression
## https://www.geeksforgeeks.org/stack-set-4-evaluation-postfix-expression/

class EvalPostfix:

    ##For initialising the class variables
    def __init__(self,capacityOfStack):
```

```
        self.topOfStack = -1
        self.capacityOfStack = capacityOfStack
        #stack for conversion
        self.stackArray = []

    ##For checking whether the stack is empty or not
    def isEmpty(self):
        return True if self.topOfStack ==-1 else False

    ##For getting the top value of the stack
    def peek(self):
        return self.stackArray[-1]

    ##For removing the top most element of the stack
    def pop(self):
        #checking whether the stack is empty or not
        if not self.isEmpty():
            self.topOfStack -= 1
            return self.stackArray.pop()
        #if stack is empty return this character which says that the
        #stack is empty
        else:
            return "$"

    ##For inserting elements into the Stack
    def push(self,op):
        #incrementing the value of top indeed increments the capacity of stack
        self.topOfStack +=1
        #inserting new element
        self.stackArray.append(op)


    def evalPostfix(self,exp,row):
        current_row = initial_tt[row]
        for i in exp:
            if i.isalpha():
                self.push(i)

            else:
```

```
                operator = i

                if operator == '`':
                    l1 = self.pop()
                    if l1!='0' and l1!='1':
                        index1 = current_row[literals_list.index(l1)]
                    else:
                        if l1 == '0':
                            index1 = False
                        elif l1 == '1':
                            index1 = True
                    result = negation(index1)
                    if result == False:
                        result = '0'
                    else:
                        result = '1'
                    self.push(result)

                else:
                    l1 = self.pop()
                    if l1!='0' and l1!='1':
                        index1 = current_row[literals_list.index(l1)]
                    else:
                        if l1 == '0':
                            index1 = False
                        elif l1 == '1':
                            index1 = True
                    l2 = self.pop()
                    if l2!='0' and l2!='1':
                        index2 = current_row[literals_list.index(l2)]
                    else:
                        if l2 == '0':
                            index2 = False
                        elif l2 == '1':
                            index2 = True
                    # index1 = literals_list.index(l1)
                    # index2 = literals_list.index(l2)
                    if operator == '+':
                        result = disjunction(index2,index1)
```

```
                    elif operator == '.':
                        result = conjunction(index2,index1)
                    elif operator == '*':
                        result = implication(index2,index1)
                    else:
                        result = biImplication(index2,index1)

                    if result == False:
                        result = '0'
                    else:
                        result = '1'

                    self.push(result)

        return self.pop()


## Code to Check
#expr = "(((`a)+b).c)"
#expr = "`a"
#expr = "((`x).y)+(x.(`y))"  ##XOR operation
#expr = "x*y"   ##Implication operation
#expr = "x$y"   ##BiImplication operation
#expr = "(`(((`p)*q)*r))"
#expr = "(p*(q.r))"
# expr = "((p.r)$((`q)+r))"
expr = input("Enter the boolean expression:- ")
# print("The given boolean expression is:- ")
# print(expr)
# print(type(expr))
obj = InfixToPostfix(len(expr))
postexpr = obj.infixToPostfix(expr)

# print("The obtained Postfix Expression is:-")
# print(postexpr)
# print(type(postexpr[1]))

#print("Obtaining the result of Boolean expression for every single row:- ")
obj1 = EvalPostfix(len(postexpr))
```

```
for k in range(0,len(initial_tt)):
        final_result = obj1.evalPostfix(postexpr,k)
        # if final_result == '0':
        #       final_result = False
        # else:
        #       final_result = True
        initial_tt[k].append(final_result)


# print(initial_tt)
## We get the result of boolean expression as 0's and 1's
for k in range(0,len(initial_tt)):
        c = initial_tt[k][no_of_literals]
        initial_tt[k].remove(c)
        # Converting 0's to "False"
        if c == '0':
                c = False
        # Converting 1's to "True"
        else:
                c = True
        initial_tt[k].append(c)

# print(initial_tt)
print()
print()
print("Truth Table after evaluating boolean expression:- ")
for ll in initial_tt:
        print(ll)


##For finding DNF of expression

def getDNF(tt,row_size):
        #initialising result as an empty string
        res = ""
        #Iterating through the truth table
        for i in range(0,len(tt)):
                #Checking for the rows in which result is "True"
                if tt[i][row_size] == True:
```

```
            #Storing the current row whose result is "True"
            current_row = tt[i]
            #For every "True" minterm starts with parentheses
            res += "("
            #Iterating through the values of the literals in truth table
            for j in range(0,len(current_row)-1):
                #Getting corresponding literal from literals list
                sym = literals_list[j]
                #If the value of the literal is "True"
                if current_row[j] == True:
                    #For the first literal in the row
                    if j == 0:
                        res = res + sym + "."
                    #For the last literal in the row
                    elif j == len(current_row)-2:
                        res = res + sym
                    #For literals other than first and last
                    else:
                        res = res + sym + "."
                #If the value of the literal is "False"
                else:
                    #For the first literal in the row
                    if j == 0:
                        res = res + "`" + sym + "."
                    #For the last literal in the row
                    elif j == len(current_row)-2:
                        res = res + "`" + sym
                    #For literals other than first and last
                    else:
                        res = res + "`" + sym + "."
            #Every minterm ends with closed parenthesis
            res += ")+"
    #As we are adding 'OR' symbol whenever the minterm finishes, we have to delete
    #the last added 'OR' symbol
    if res[len(res)-1] == "+":
        res = res[:-1]
    return res

## We have to define a function which takes Truth Table as input and produces
```

```
#  minterms as output

def ttToMinterms(tt):
    #list for storing the minterms
    minterms = []
    # Iterating through the truth table
    for i in range(len(tt)):
        # storing the current row
        current = tt[i]
        # Checking whether the result of row is True or not
        if current[-1] == True:
            # Adding particular minterm to list
            minterms.append(i)

    # We have to convert the minterms into binary values
    mt = []
    for i in minterms:
        term = bin(i)[2:].zfill(len(tt[0])-1)
        mt.append(str(term))
    # print(minterms)
    # print(mt)
    return mt

def mergeBoth(m1,m2):
    #Creating an empty string for storing after merging
    after_merge = ''
    size_of_minterm = len(m1)
    #Storing the number of dashes to identify the number of different values in minterms
    num_of_dashes = 0
    #iterating through each value in minterms
    for i in range(size_of_minterm):
        # if both the values are different, we make dash at that position and increment
        # the number of dashes
        if m1[i] != m2[i]:
            after_merge = after_merge+"-"
            num_of_dashes+=1
        # if both the values are same, we attach the same value to 'after_merge' also
        elif m1[i] == m2[i]:
            after_merge = after_merge+m1[i]
```

```
        # We will return the 'after_merge' only if the number of different values are less than or
        # equal to 1
        if num_of_dashes <= 1:
                return after_merge
        else:
                return None

def findingMinimisedMinterms(true_min_terms):
        ## Reference for this approach
        ## https://github.com/tpircher/quine-mccluskey/blob/master/quine_mccluskey/qm.py
        ## https://en.wikipedia.org/wiki/Quine%E2%80%93McCluskey_algorithm

        #Converting the set into list
        true_min_terms = list(true_min_terms)
        # Creating a empty list which stores the initial implicants
        initial_implicants = []
        # Creating an empty list which stores the further implicants
        further_implicants = []
        # Creating an empty list which stores the final implicants
        final_implicants = []
        # Getting the number of minterms
        size = len(true_min_terms)
        # Creating a list of size equal to number of minterms and initialising with zeros
        # to mark which minterm is used and which is not
        used = [0]*size
        # We have to store the count of minterms added to final list
        size_of_final = 0

        # Calling 'mergeBoth' method for every combination of minterms

        for i in range(0,size-1):
                for j in range(i+1,size):
                        dummy_minterm = mergeBoth(true_min_terms[i],true_min_terms[j])

                        if dummy_minterm != None:
                                # Adding minterm to implicants list
                                initial_implicants.append(dummy_minterm)
```

```
            # marking the corresponding minterms as used
            used[i] = 1
            used[j] = 1
        # If it is 'None', try for next minterm
        else:
            continue

print(initial_implicants)
# Now we have to add the implicants which are not used when merging to
# Obtain initial implicants
## Adding minterms to final list which are not used to form initial implicants list

for i in range(size):
    # if the minterm is not used, we have to add to the final implicant list,
    # because, once we can't find a particular match to merge the current minterm
    # even in the further iterations we cannot find the match
    if used[i] == 0:
        final_implicants.append(true_min_terms[i])
        # Incrementing the count of minterms in the final list of implicants
        size_of_final = size_of_final + 1


# Now we are done with obtaining initial implicants, We have to process this list
# again, as we may get more reduced implicants among these

# Checking for the duplicate implicants we obtained
# We create a list initialized with zeros
repeated = [0]*len(initial_implicants)

for i in range(0,len(initial_implicants)-1):
    for j in range(i+1,len(initial_implicants)):
        if i!=j and repeated[j] == 0:
            if initial_implicants[i] == initial_implicants[j]:
                # we are marking only one implicant as repeated because we
                #have to add
                # the another implicant to further implicants list
                repeated[j] = 1
```

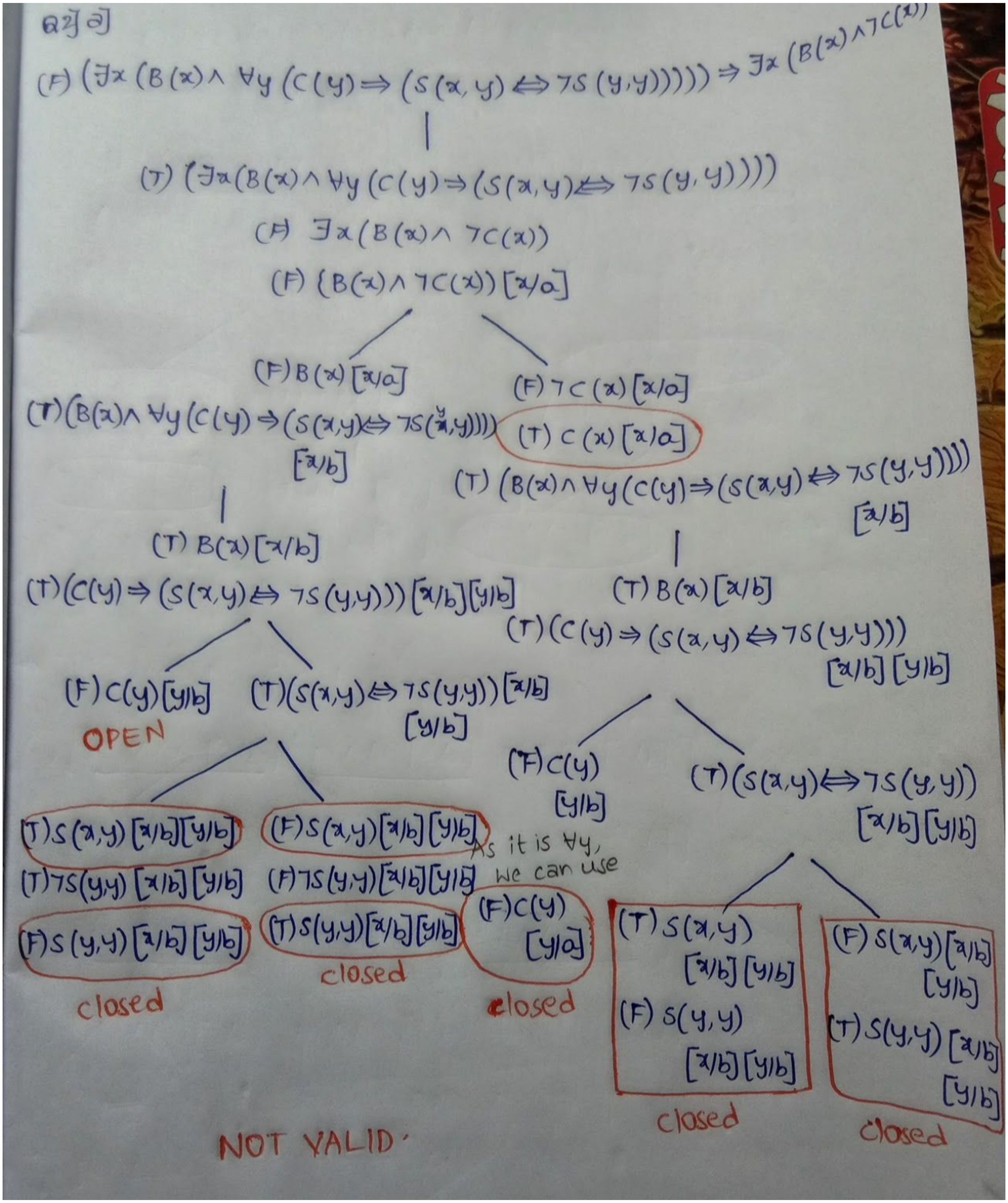
```
        # Now add the implicants which are not repeated to the further implicants

        # Iterating through the initial implicants
        for i in range(len(initial_implicants)):
                # If the current implicant is not repeated
                if repeated[i] == 0:
                        # Add them to further implicants list
                        further_implicants.append(initial_implicants[i])

        ## Now we have totally three cases:
        ## Case1:- When the size of final_implicants and size of minterms list
        #  We got for this function the same. In this case, we do not need to further
        #  process to repeat the merging and return final implicants list

        if size_of_final == size:
                return final_implicants

        ## Case2:- When we obtain only one minterm to this function
        #  we don't need to process it again and again, we return final
        #  implicants list

        elif size == 1:
                return final_implicants

        ## Case3:- If the above two cases is not true, it means that we are left
        # some of the minterms still for merging

        else:
                return final_implicants + findingMinimisedMinterms(further_implicants)

def convertMinToLiterals(list1,literals_list):
   # print('inside')
   # print(list1)
   #print(literals_list[0])
   expr = ""
   for l in list1:
      # print(l)
      # print(type(l))
      expr = expr+"("
```

```
        for c in range(0,len(l)):
            # print(l[c])
            # print(type(l[c]))
            sym = literals_list[c]
            #print(c)
            if l[c] == '1':
                expr = expr + sym + "."
            elif l[c] == '0':
                expr = expr + "`" + sym + "."
        expr = expr[:-1]
        expr = expr + ")+"

    expr = expr[:-1]
    return expr

print()
print()
##Getting DNF Expression
res = getDNF(initial_tt,no_of_literals)
print()

true_min_terms = ttToMinterms(initial_tt)

print("The initial ture minterms are:- ")
print(true_min_terms)
print()
print()
print("The resultant DNF for given Boolean expression is:- ")
print(res)
print()

final_minimised_minterms = findingMinimisedMinterms(true_min_terms)

print()
print("The minimised minterms are:- ")
print(final_minimised_minterms)
print()
reduced_dnf = convertMinToLiterals(final_minimised_minterms,literals_list)
```

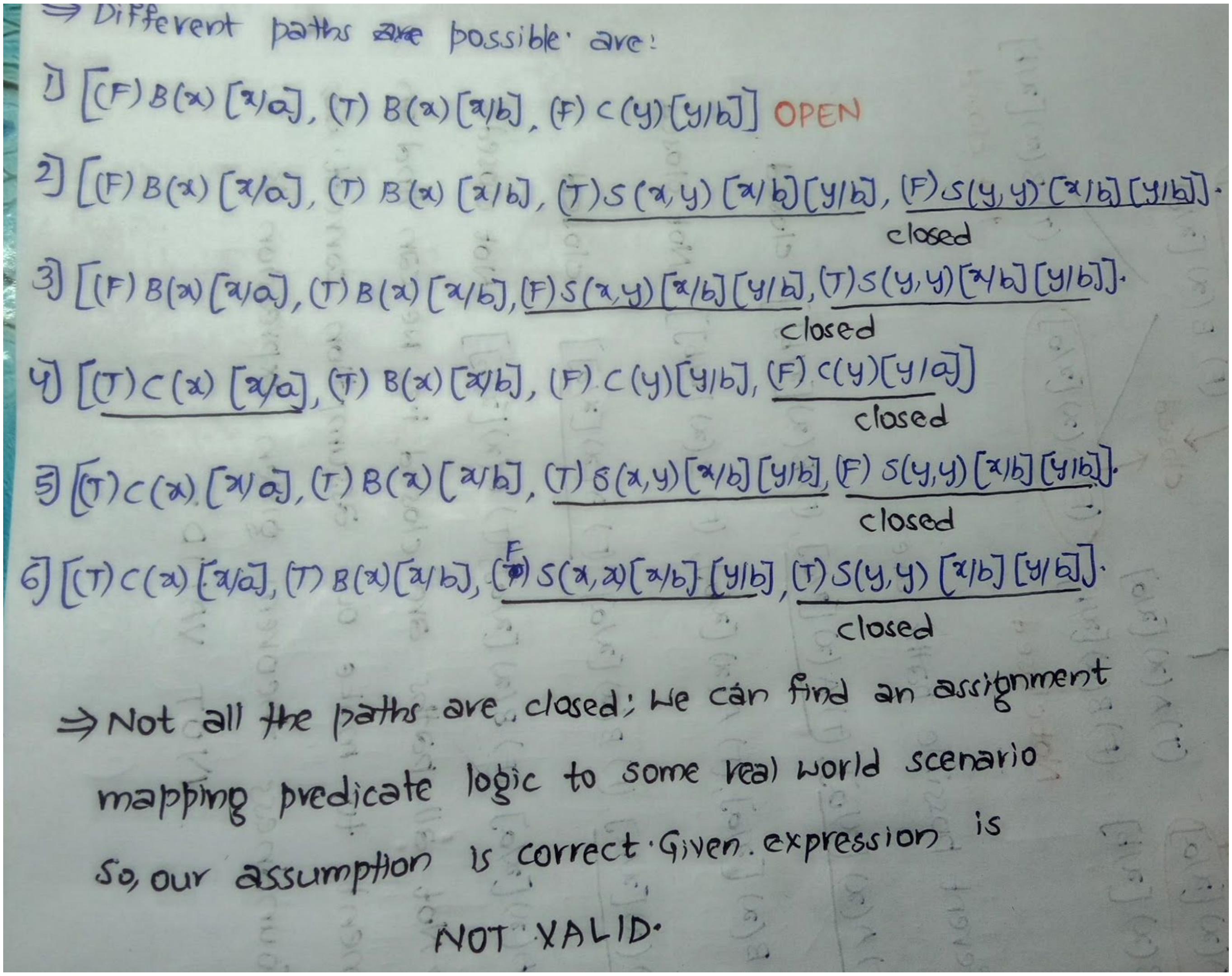
```
print("The reduced DNF is:- ")
print(reduced_dnf)
```

****

Q2)(a)


Q2] a]
(F) (∃x (B(x) ∧ ∀y (C(y) ⇒ (S(x, y) ⇔ ¬S(y,y))))) ⇒ ∃x (B(x) ∧ ¬C(x))
(T) (∃x(B(x) ∧ ∀y (C(y) ⇒ (S(x,y) ⇔ ¬S(y, y))))
(F) ∃x(B(x) ∧ ¬C(x))
(F) (B(x) ∧ ¬C(x)) [x/a]
(F) B(x) [x/a]
(F) ¬C(x) [x/a]
(T) (B(x) ∧ ∀y (C(y) ⇒ (S(x,y) ⇔ ¬S(y,y)))) [x/b]
(T) C(x) [x/a]
(T) (B(x) ∧ ∀y (C(y) ⇒ (S(x,y) ⇔ ¬S(y,y)))) [x/b]
(T) B(x) [x/b]
(T) (C(y) ⇒ (S(x,y) ⇔ ¬S(y,y))) [x/b] [y/b]
(T) B(x) [x/b]
(T) (C(y) ⇒ (S(x,y) ⇔ ¬S(y,y))) [x/b] [y/b]
(F) C(y) [y/b]
OPEN
(T) (S(x,y) ⇔ ¬S(y,y)) [x/b] [y/b]
(F) C(y) [y/b]
(T) (S(x,y) ⇔ ¬S(y,y)) [x/b] [y/b]
(T) S(x,y) [x/b] [y/b]
(T) ¬S(y,y) [x/b] [y/b]
(F) S(y,y) [x/b] [y/b]
closed
(F) S(x,y) [x/b] [y/b]
(F) ¬S(y,y) [x/b] [y/b]
(T) S(y,y) [x/b] [y/b]
closed
As it is ∀y, we can use
(F) C(y) [y/a]
closed
(T) S(x,y) [x/b] [y/b]
(F) S(y,y) [x/b] [y/b]
closed
(F) S(x,y) [x/b] [y/b]
(T) S(y,y) [x/b] [y/b]
closed
NOT VALID.

Explanation for 2(a)



⇒ Different paths ~~are~~ possible· are:

1] $[(F)B(x)[x/a], (T)B(x)[x/b], (F)C(y)[y/b]]$ OPEN

2] $[(F)B(x)[x/a], (T)B(x)[x/b], \underline{(T)S(x,y)[x/b][y/b]}, \underline{(F)S(y,y)[x/b][y/b]}]$.

closed

3] $[(F)B(x)[x/a], (T)B(x)[x/b], \underline{(F)S(x,y)[x/b][y/b]}, \underline{(T)S(y,y)[x/b][y/b]}]$.

closed

4] $[\underline{(T)C(x)[x/a]}, (T)B(x)[x/b], (F)C(y)[y/b], \underline{(F)C(y)[y/a]}]$

closed

5] $[(T)C(x)[x/a], (T)B(x)[x/b], \underline{(T)S(x,y)[x/b][y/b]}, \underline{(F)S(y,y)[x/b][y/b]}]$.

closed

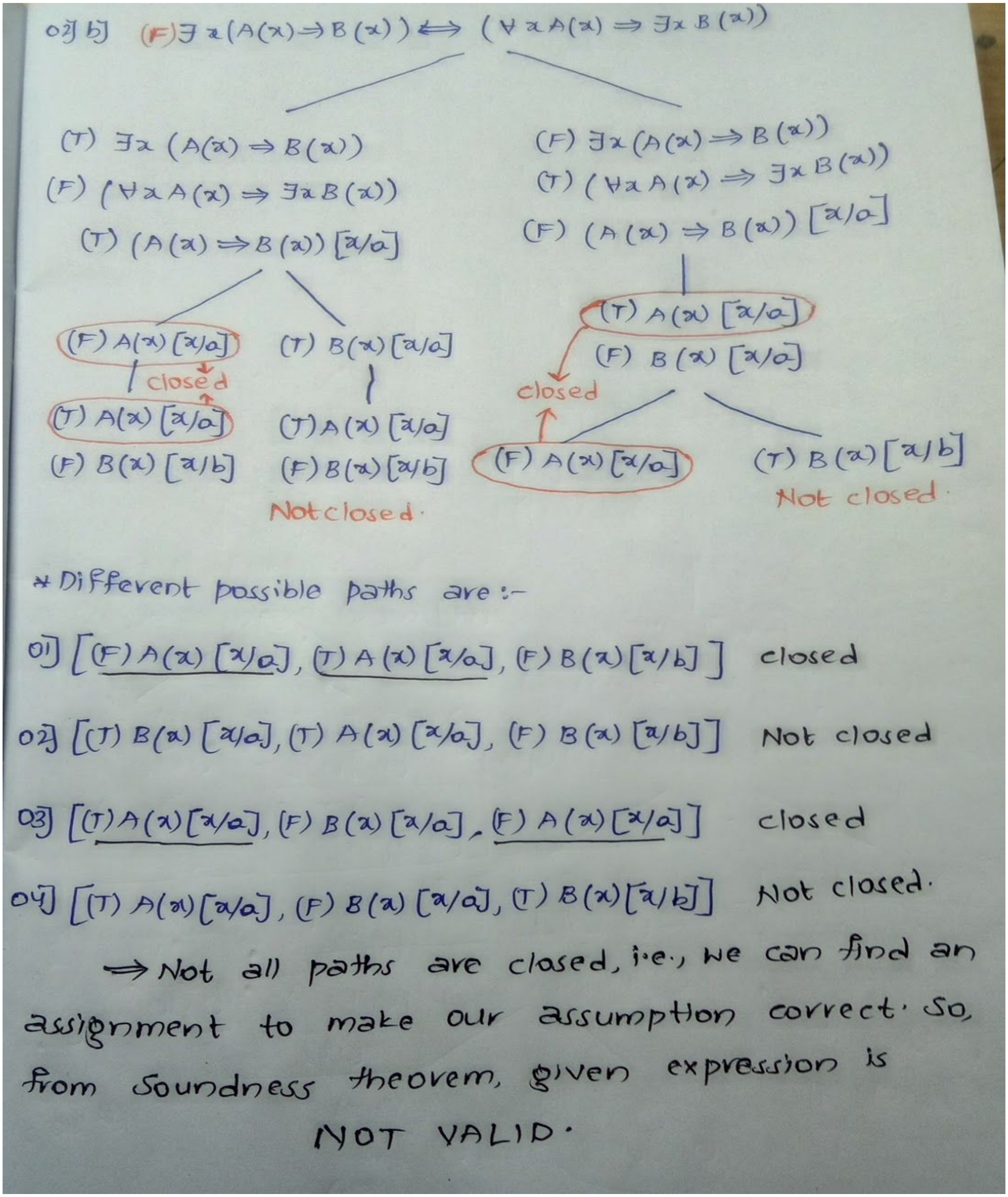
6] $[(T)C(x)[x/a], (T)B(x)[x/b], \underline{\overset{F}{(\ )}S(x,x)[x/b][y/b]}, \underline{(T)S(y,y)[x/b][y/b]}]$.

closed

⇒ Not all the paths are closed; we can find an assignment mapping predicate logic to some real world scenario so, our assumption is correct· Given expression is

NOT VALID·

Q2)(b)



02] b] $(F)\ \exists x(A(x) \Rightarrow B(x)) \Leftrightarrow (\forall x A(x) \Rightarrow \exists x B(x))$

$(T)\ \exists x\ (A(x) \Rightarrow B(x))$

$(F)\ (\forall x A(x) \Rightarrow \exists x B(x))$

$(T)\ (A(x) \Rightarrow B(x))\ [x/a]$

$(F)\ A(x)\ [x/a]$

closed

$(T)\ A(x)\ [x/a]$

$(F)\ B(x)\ [x/b]$

$(T)\ B(x)\ [x/a]$

$(T)\ A(x)\ [x/a]$

$(F)\ B(x)\ [x/b]$

Notclosed.

$(F)\ \exists x\ (A(x) \Rightarrow B(x))$

$(T)\ (\forall x A(x) \Rightarrow \exists x B(x))$

$(F)\ (A(x) \Rightarrow B(x))\ [x/a]$

$(T)\ A(x)\ [x/a]$

$(F)\ B(x)\ [x/a]$

closed

$(F)\ A(x)\ [x/a]$

$(T)\ B(x)\ [x/b]$

Not closed.

* Different possible paths are :-

01] $[\underline{(F)\ A(x)\ [x/a]}, \underline{(T)\ A(x)\ [x/a]}, (F)\ B(x)\ [x/b]]$ closed

02] $[(T)\ B(x)\ [x/a], (T)\ A(x)\ [x/a], (F)\ B(x)\ [x/b]]$ Not closed

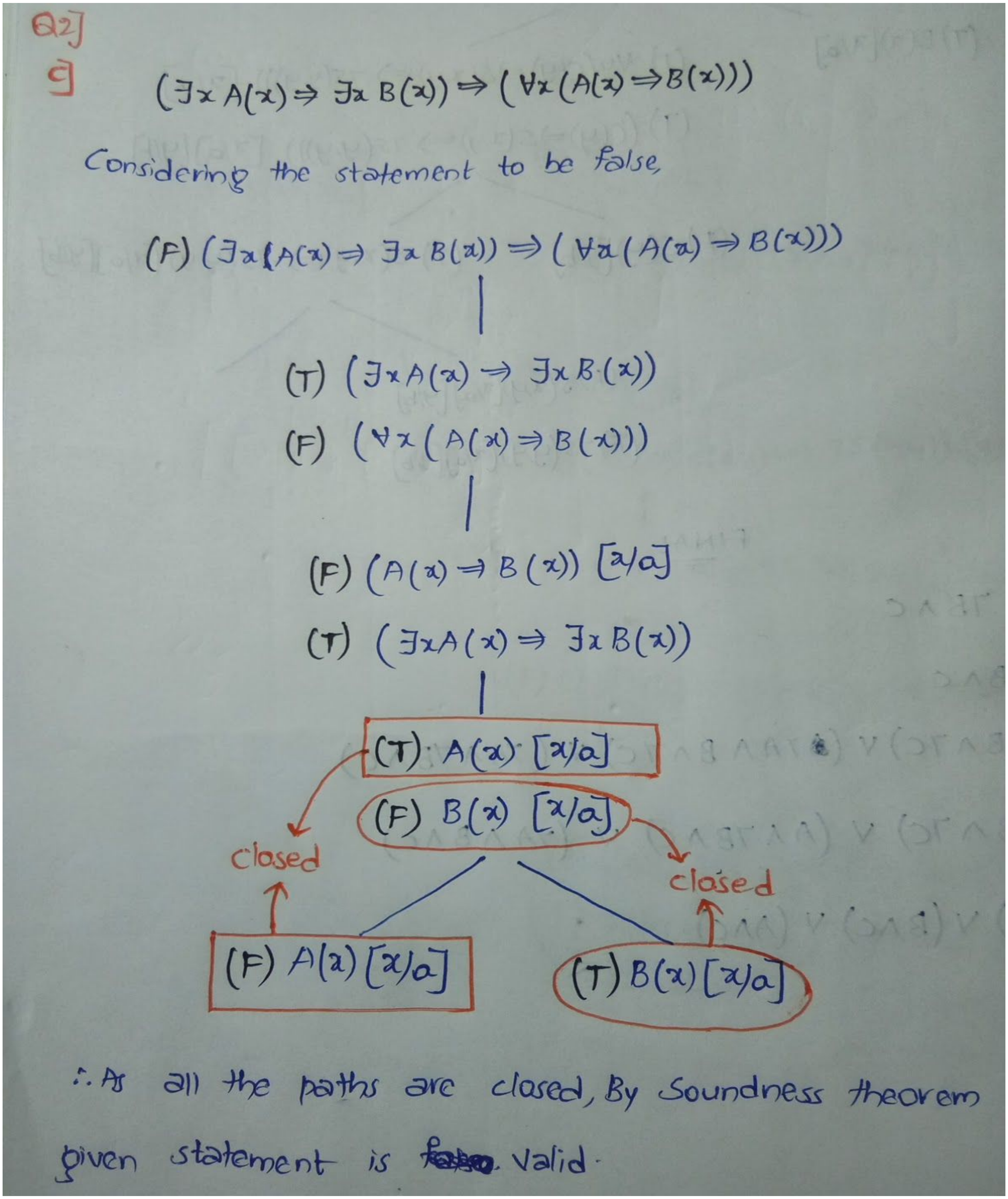
03] $[\underline{(T)\ A(x)\ [x/a]}, (F)\ B(x)\ [x/a], \underline{(F)\ A(x)\ [x/a]}]$ closed

04] $[(T)\ A(x)\ [x/a], (F)\ B(x)\ [x/a], (T)\ B(x)\ [x/b]]$ Not closed.

⇒ Not all paths are closed, i.e., we can find an assignment to make our assumption correct. So, from soundness theorem, given expression is

NOT VALID.

Q2)(c)



Q2]

c] $(\exists x\, A(x) \Rightarrow \exists x\, B(x)) \Rightarrow (\forall x(A(x) \Rightarrow B(x)))$

Considering the statement to be false,

$(F)\ (\exists x(A(x) \Rightarrow \exists x\, B(x)) \Rightarrow (\forall x(A(x) \Rightarrow B(x)))$

$(T)\ (\exists x\, A(x) \Rightarrow \exists x\, B(x))$

$(F)\ (\forall x(A(x) \Rightarrow B(x)))$

$(F)\ (A(x) \Rightarrow B(x))\ [x/a]$

$(T)\ (\exists x A(x) \Rightarrow \exists x\, B(x))$

$(T)\ A(x)\ [x/a]$

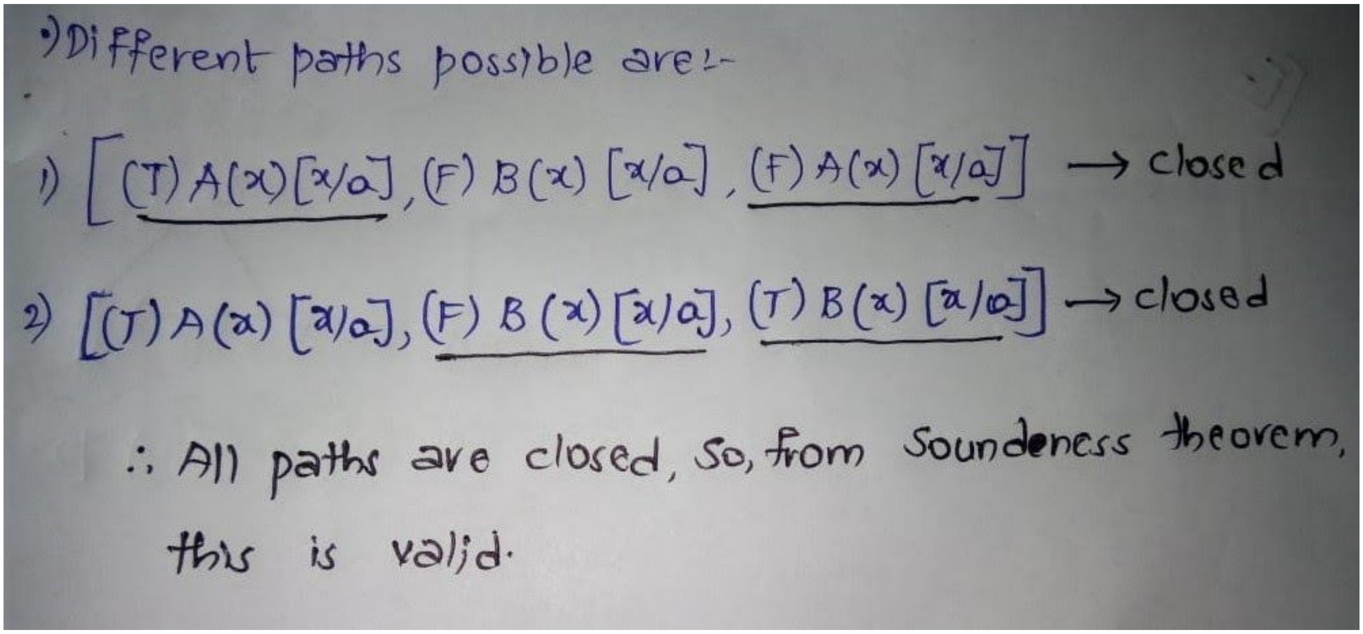
$(F)\ B(x)\ [x/a]$

$(F)\ A(x)\,[x/a]$ — closed

$(T)\ B(x)\,[x/a]$ — closed

∴ As all the paths are closed, By Soundness theorem given statement is ~~false~~ valid.

Explanation for Q2)(c)



•) Different paths possible are:-

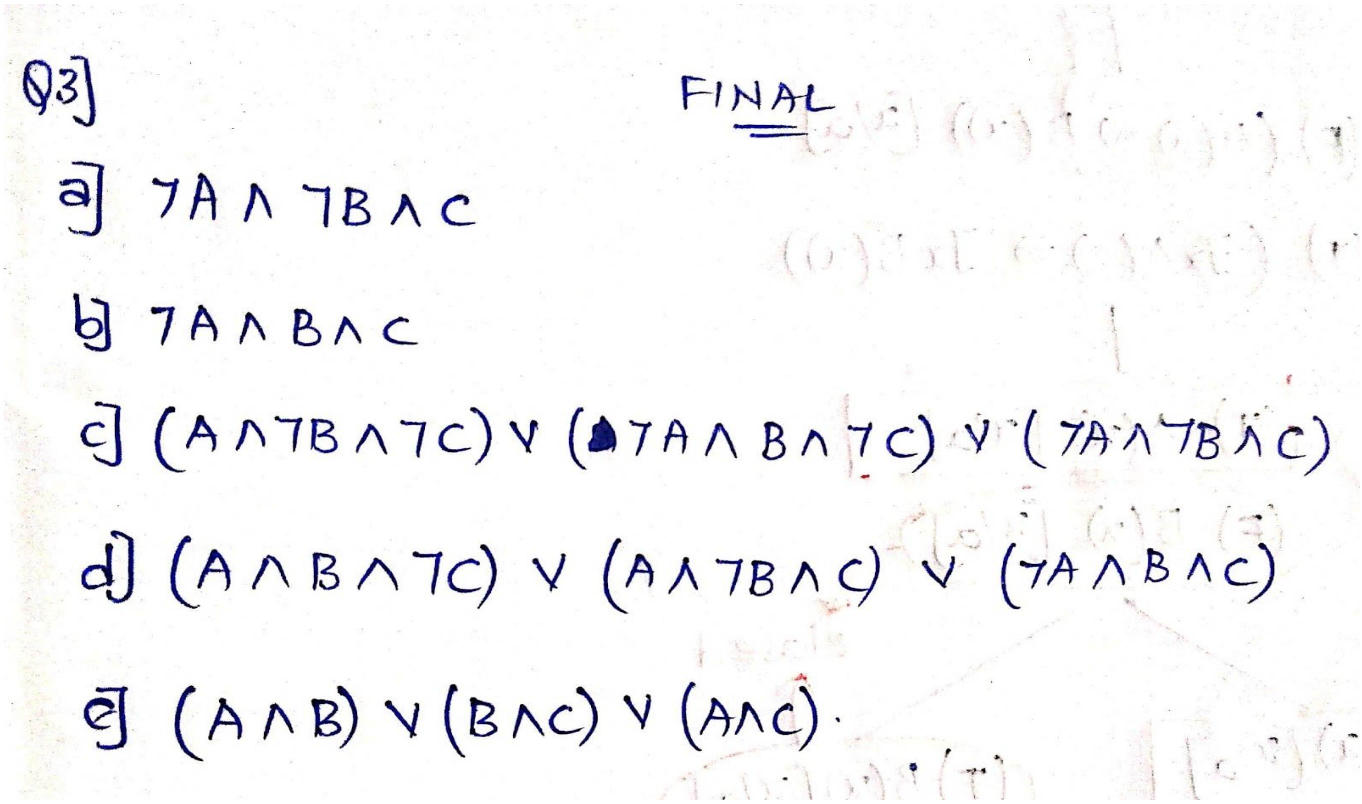
1) $\left[\underline{(T)\,A(x)[x/a]},\ (F)\,B(x)\,[x/a],\ \underline{(F)\,A(x)\,[x/a]}\right] \rightarrow$ closed

2) $\left[(T)\,A(x)\,[x/a],\ \underline{(F)\,B(x)\,[x/a]},\ \underline{(T)\,B(x)\,[x/a]}\right] \rightarrow$ closed

∴ All paths are closed, So, from Soundeness theorem, this is valid.

****

Q3)



Q3]

FINAL

a] $\neg A \land \neg B \land C$

b] $\neg A \land B \land C$

c] $(A \land \neg B \land \neg C) \lor (\neg A \land B \land \neg C) \lor (\neg A \land \neg B \land C)$

d] $(A \land B \land \neg C) \lor (A \land \neg B \land C) \lor (\neg A \land B \land C)$

e] $(A \land B) \lor (B \land C) \lor (A \land C)$.

****

Q4)

```
# '#' for Disjunction
# '.' for Conjunction
# '`' for Negation
# '*' for Implication
# '$' for Bi-implication
# Use '+' to indicate it as True
# Use '-' to indicate as False

## Alpha-Beta logic is written in this method,Based on the
#  boolean function, two or one path is returned along with the
#  expressions(along with values of expressions) in the paths.
def actualSplit(expr,splitIndex,booleanType,exprValue):
    # exprValue indicates whether the expression is with True or False
    # expr is the actual expression which we have to split
    # booleanType is the boolean operation to be considered for split
    # splitIndex indicates on which index we have to split the expression
    # As we do not know into how many paths the expression gets splitted,we maintain
    # two lists to store the items of two paths if any.
    list1 = []
    list2 = []
    # If operation is implication
    if booleanType == '*':
        # If the value of expression is False
        if exprValue == '-':
            str1,str2 = splitOn(expr,splitIndex)
            str1 = deleteExtraBrackets(str1)
            str2 = deleteExtraBrackets(str2)
            # When implication is False, We know that the first part is True
            # AND second part is False and both has to be in same path
            # so we have to add them into same list
            str1 = '+'+str1
            str2 = '-'+str2
            list1.append(str1)
            list1.append(str2)
        # If the value of expression is True
```

```
        else:
            str1,str2 = splitOn(expr,splitIndex)
            str1 = deleteExtraBrackets(str1)
            str2 = deleteExtraBrackets(str2)
            # When implication is True, We know that, the first part can be False
            # OR the second part can be True, These split into two different paths,
            # So we have to add them into different lists
            str1 = '-'+str1
            str2 = '+'+str2
            list1.append(str1)
            list2.append(str2)
    #If the operation is Disjunction
    if booleanType == '#':
        #If the value of expression is False
        if exprValue == '-':
            str1,str2 = splitOn(expr,splitIndex)
            str1 = deleteExtraBrackets(str1)
            str2 = deleteExtraBrackets(str2)
            # When disjunction is False, it means that both of them have to be False,
            # As both needed to be False, There will be only single path for this,
            # We will be adding them into same list
            str1 = '-'+str1
            str2 = '-'+str2
            list1.append(str1)
            list1.append(str2)
        # If the value of expression is True
        else:
            str1,str2 = splitOn(expr,splitIndex)
            str1 = deleteExtraBrackets(str1)
            str2 = deleteExtraBrackets(str2)
            # When the Disjunction is True, it means that anyone of the both needs to
            # be True, As as one of it being True is sufficient, We have to add them
            # into seperate lists
            str1 = '+'+str1
            str2 = '+'+str2
            list1.append(str1)
            list2.append(str2)
    #If the operation is Conjunction
    if booleanType == '.':
```

```
        #If the value of expression is False
        if exprValue == '-':
            str1,str2 = splitOn(expr,splitIndex)
            str1 = deleteExtraBrackets(str1)
            str2 = deleteExtraBrackets(str2)
            # When the Conjunction is False, it means that anyone of the both needs
            #to
            # be False, As as one of it being False is sufficient, We have to add them
            # into seperate lists
            str1 = '-'+str1
            str2 = '-'+str2
            list1.append(str1)
            list2.append(str2)
        #If the value of expression is True
        if exprValue == '+':
            str1,str2 = splitOn(expr,splitIndex)
            str1 = deleteExtraBrackets(str1)
            str2 = deleteExtraBrackets(str2)
            # When Conjunction is True, it means that both of them have to be True,
            # As both needed to be True, There will be only single path for this,
            # We will be adding them into same list
            str1 = '+'+str1
            str2 = '+'+str2
            list1.append(str1)
            list1.append(str2)


    return list1,list2


## This method splits the expression based on the index passed.
def splitOn(expr,i):
    str1 = ""
    str2 = ""
    print(expr)
    print(type(expr))
    str1 = expr[:i]
    str2 = expr[i+1:len(expr)]
    return str1,str2
```

```
## After Splitting, Sometimes we would be having extra brackets,
#  this method removes those extra brackets in the expression passed to it.
def deleteExtraBrackets(str1):
        open_brackets = 0
        closed_brackets = 0
        for i in range(len(str1)):
                if str1[i] == '(':
                        open_brackets+=1
                elif str1[i] == ')':
                        closed_brackets+=1
        if open_brackets == closed_brackets:
                return str1
        elif open_brackets+1 == closed_brackets:
                if str1[len(str1)-1] == ')':
                        str1 = str1[:-1]
                        return str1
        elif closed_brackets+1 == open_brackets:
                if str1[0] == '(':
                        str1 = str1[1:]
                        return str1

## Every expression contains '-' or '+' in the beginning, representing whether
#  they are 'False' or 'True' respectively. We seperate this value from
#  expression in this method.
def seperateValueFromExpression(expr):
        # print(expr)
        # print(type(expr))
        # print(expr[0])
        if expr[0] == '+':
                value = '+'
                expr = expr[1:]
        else:
                value = '-'
                expr = expr[1:]
        return value,expr

## Method for getting the splitIndex and SplitSymbol from Postfix expression
def getSplitIndex(postfixExpression):
```

```
        size = len(postfixExpression)
        symbol = postfixExpression[size-1][0]
        index = postfixExpression[size-1][1]
        return symbol,index

def isSimple(expr):
        if len(expr) == 2:
                return True
        else:
                return False

def isListSimple(l1):
        flag = 0   # 0 means simple as of now
        for i in l1:
                if isSimple(i) == False:
                        flag = 1
                        return False
        if flag == 0:
                return True

def checkForClosed(ans_tt):
        closed = True
        for ll in ans_tt:
                row_done = 0
                for i in range(len(ll)-1):
                        flag = 0        #represents whether the match found or not
                        sign = ll[i][0]
                        literal = ll[i][1]
                        if sign == '+':
                                wanted_sign = '-'
                        else:
                                wanted_sign = '+'
                        for j in range(i+1,len(ll)):
                                if ll[j] == wanted_sign+literal :
                                        flag = 1
                                        row_done = 1
                                        break
                        if flag == 1:
                                break
```

```
        if row_done == 0:
            closed = False
    return closed



##Class Definition for converting Infix expression to Postfix expression
##Reference:-
## geeksforgeeks.org/stack-set-2-infix-to-postfix/
#  Our postfix expression contains symbols along with the index of that
#  particular symbol in the initial expression. We use these indexes for
#  splitting.
class InfixToPostfix:

    ##For initialising the class variables
    def __init__(self,capacityOfStack):
        self.topOfStack = -1
        self.capacityOfStack = capacityOfStack
        #stack for conversion
        self.stackArray = []
        #for storing the output_expr
        self.output_expr = []
        #for precedent setting
        self.op_precedence = {'#':1, '.':1, '*':2, '$':2, '`':3}

    ##For checking whether the stack is empty or not
    def isEmpty(self):
        return True if self.topOfStack ==-1 else False

    ##For getting the top value of the stack
    def peek(self):
        dummy = self.stackArray[-1]
        return dummy[0]

    ##For removing the top most element of the stack
    def pop(self):
        #checking whether the stack is empty or not
        if not self.isEmpty():
            self.topOfStack -= 1
```

```
        return self.stackArray.pop()
    #if stack is empty return this character which says that the
    #stack is empty
    else:
        return "$"

##For inserting elements into the Stack
def push(self,op):
    duplicate_list = []
    #incrementing the value of top indeed increments the capacity of stack
    self.topOfStack +=1
    #inserting new element
    self.stackArray.append(op)


##Checking whether the character passed is Operand or not
def isOperand(self,ch):
    return ch.isalpha()


##Checking whether the operator precedence is strictly less than the
##operator present at the top of the stack
def notGreater(self,i):
    try:
        a = self.op_precedence[i]
        b = self.op_precedence[self.peek()]
        return True if a <= b else False
    except KeyError:
        return False

##Function to convert an infix expression to
##Postfix expression
def infixToPostfix(self,exp):

    #Traversing through the expression
    for i in range(len(exp)):
        if self.isOperand(exp[i]):
            duplicate_list1 = []
            duplicate_list1.append(exp[i])
```

```
                duplicate_list1.append(i)
                self.output_expr.append(duplicate_list1)

            #if the current symbol is open parenthesis we are pushing it
            #on to the stack
            elif exp[i] == '(':
                duplicate_list1 = []
                duplicate_list1.append(exp[i])
                duplicate_list1.append(i)
                self.push(duplicate_list1)

            #when current symbol is closed parenthesis
            elif exp[i] == ')':
                #popping all the elements of the stack still we encounter a
                #open parenthesis
                while ((not self.isEmpty()) and self.peek() != '('):
                    a = self.pop()
                    self.output_expr.append(a)
                if (not self.isEmpty() and self.peek() != '('):
                    return -1
                else:
                    self.pop()

            else:
                while(not self.isEmpty() and self.notGreater(exp[i])):
                    self.output_expr.append(self.pop())
                duplicate_list1 = []
                duplicate_list1.append(exp[i])
                duplicate_list1.append(i)
                self.push(duplicate_list1)

        while not self.isEmpty():
            self.output_expr.append(self.pop())

        #printing the final converted expression
        #print(self.output_expr)
        return self.output_expr

b = input("Enter the consequence:-  ")
```

```
a = input("Enter the statements:- ")
expr = '('+a+'*'+b+')'
#expr = "((a*(b*c))*((a*b)*(a*c)))"
print("The combined statement is:- ")
print(expr)
# For proving validity, we assume the given expression as False,
expr = '-'+expr
print(expr)
initial_path = []
initial_path.append(expr)
final_tree = []
final_tree.append(initial_path)
print("The final tree list is:- ")
print(final_tree)

def myMain(final_tree):
    for list1 in final_tree:
        flag1 = 0
        print(list1)
        for i in range(len(list1)):
            if isSimple(list1[i]) == False:
                value,expression = seperateValueFromExpression(list1[i])
                print("The value of the expression is:-  "+value)
                print("The expression after deleting its value is:- "+str(expression))
                obj = InfixToPostfix(len(expression))
                postexpr = obj.infixToPostfix(expression)
                print("The postfix expression is:- ")
                print(postexpr)
                symbol,index = getSplitIndex(postexpr)
                print("The symbol of split is:- "+symbol)
                print("The index of split is:- "+str(index))
                p1,p2 = actualSplit(str(expression),index,str(symbol),str(value))
                print("The path1 is:- ")
                print(p1)
                print("The path2 is:- ")
                print(p2)
                if len(p1) == 0 and len(p2) == 0:
                    continue;
```

```
            elif len(p2) == 0:
                for ele in p1:
                    list1.append(ele)
                list1.remove(list1[i])
            else:
                dummy1 = []
                for j in range(len(list1)):
                    if j != i:
                        dummy1.append(list1[j])
                for ele in p1:
                    dummy1.append(ele)

                dummy2 = []
                for j in range(len(list1)):
                    if j != i:
                        dummy2.append(list1[j])
                for ele in p2:
                    dummy2.append(ele)
                list1.clear()
                list1.append(dummy1)
                list1.append(dummy2)
                final_tree.remove(list1)
                flag1 = 1
                break

        print("The list1 is ")
        print(list1)
    if flag1 == 1:
        for dad in list1:
            if dad not in final_tree:
                final_tree.append(dad)
    #final_tree.append(list1)
    print("The final tree is ")
    print(final_tree)
# Trying for recursion
for lists in final_tree:
    if isListSimple(lists) == False:
        myMain(final_tree)
return final_tree
```

```
ans_tree = myMain(final_tree)
print()
print()
print()
print("All possible paths are:- ")
for ll in ans_tree:
        print(ll)
print()
print()
print()

Nikhil = checkForClosed(ans_tree)
if Nikhil == True:
        print("All paths are closed..!")
        print("Our assumption is wrong..!")
        print(b+" is logical consequence of "+a)
        print("!!!!!!! TRUE !!!!!!!!")
else:
        print("All paths are not closed...!")
        print(b+" is not a logical consequence of "+a)
        print("!!!!!!! FALSE !!!!!!!!")
```

****