

3장 프로그래밍 실습_파이썬 코드

[프로그래밍 실습 1]

```
EXP_SIZE = 60          # 합성명제의 길이를 위한 상수
OPERATOR_NUMBER = 10   # 명제 변수의 개수를 위한 상수
MAX_SIZE_TABLE = 26     # 논리표의 크기를 위한 상수

size = 0               # 연산자
keys = [''] * MAX_SIZE_TABLE # 연산자를 저장하기 위한 문자형 배열
values = [0] * MAX_SIZE_TABLE # 비연산자를 저장하기 위한 배열(0으로 초기화)
NUM_OF_OPERAND = 0     # 비연산자의 개수를 위한 변수
NUM_OF_OPERATOR = 0    # 연산자의 개수를 위한 변수

result_table = []      # 결과 논리표를 저장하기 위한 메모리 공간
expression_result = [] # 계산된 합성명제를 저장하기 위한 메모리 공간

class Operator:
    def __init__(self, c = "", headarrive = 0, next = None):
        self.op = [''] * EXP_SIZE
        self.HeadArrive = headarrive
        self.Next = next

    def Push(head, c):
        Temp = head
        if head.HeadArrive == 0:
            head.Next = None # 스택의 탑 노드이므로 Next는 None(C 코드에서는 Null)
            head.HeadArrive = 1
            head.op = c
            return head # 헤드값을 리턴
        else:
            while Temp.Next != None: # 스택의 탑 노드까지 찾아감
                Temp = Temp.Next

            NewNode = Operator() # 새로운 노드를 할당
            NewNode.op = c
            Temp.Next = NewNode
            head.HeadArrive += 1
            # 새로운 노드를 리턴

    def Pop(head):
        Temp = head
        if head.Next == None:
            if head.HeadArrive == 0:
                print("잘못된 Pop연산.\n")
                return None
            head.HeadArrive = 0
            return head.op # 스택의 탑값인 헤드를 리턴
        else:
            while Temp.Next.Next != None:
                Temp = Temp.Next
            Temp1 = Temp.Next
            Temp.Next = None
            head.HeadArrive -= 1
            return Temp1.op # 제거될 노드의 주소와 탑 노드 직전 노드의 주소를 리턴
```

```

def Peek(head):
    Temp = head
    if head.Next == None:
        if head.HeadArrive == 0:
            # printf("잘못된 Peek연산.\n")
            return '\0'
        return head.op # 스택의 탑값인 헤드를 리턴
    else:
        while Temp.Next != None:
            Temp = Temp.Next
        return Temp.op # 제거될 노드의 주소와 탑 노드 직전 노드의 주소를 리턴

class Value:
    def __init__(self, c = "", value =[0], headarrive = 0, next = None):
        self.exp = [''] * EXP_SIZE for _ in range(EXP_SIZE)]
        self.value = [0] * (1 << OPERATOR_NUMBER)
        self.HeadArrive = headarrive
        self.Next = next

    def Push(head, c, value):
        Temp = head
        if head.HeadArrive == 0:
            head.Next = None # 스택의 탑 노드이므로 Next는 None(C 코드에서는 Null)
            head.HeadArrive = 1
            head.exp = c
            for i in range(0, 1 << NUM_OF_OPERAND):
                head.value[i] = value[i]
            return head # 헤드값을 리턴
        else:
            while Temp.Next != None: # 스택의 탑 노드까지 찾아감
                Temp = Temp.Next

            NewNode = Value(c, 0) # 새로운 노드를 할당
            Temp.Next = NewNode
            NewNode.exp = c
            for i in range(0, 1 << NUM_OF_OPERAND):
                NewNode.value[i] = value[i]
            head.HeadArrive += 1
            # 새로운 노드를 리턴

    def Pop(head):
        Temp = head
        if head.Next == None:
            if head.HeadArrive == 0:
                printf("잘못된 Pop연산.\n")
                return None
            head.HeadArrive = 0
            return head # 스택의 탑값인 헤드를 리턴
        else:
            while Temp.Next.Next != None:
                Temp = Temp.Next
            Temp1 = Temp.Next
            Temp.Next = None
            head.HeadArrive -= 1
            return Temp1 # 제거될 노드의 주소와 탑 노드 직전 노드의 주소를 리턴

def check_and_despace(exp, num): # 명제에 허가되지 않는 특수 문자가 없는지 확인하는 함수

```

```

hash_table = [0] * 26
i = len(exp) - 1
while i >= 0:
    if not (exp[i].isalpha() or exp[i] == '|' or exp[i] == '&' or exp[i] == '!' or exp[i]
== '>' or exp[i] == '(' or exp[i] == ')'):
        return False
    elif exp[i].isalpha() and not hash_table[ord(exp[i]) - 65]:
        hash_table[ord(exp[i]) - 65] = 1
        num -= 1
    elif exp[i] == '|' or exp[i] == '&' or exp[i] == '!' or exp[i] == '>':
        global NUM_OF_OPERATOR
        NUM_OF_OPERATOR += 1

    i -= 1
if num == 0:
    return True
return False

def get_index(key):    # 명제 변수와 논리 연산자들의 인덱스를 반환하는 함수
    for i in range(0,size):
        if keys[i] == key:
            return i
    return -1

def insert(key, value):    # 명제 변수와 논리 연산자들을 인덱싱하는 함수
    global size
    index = get_index(key)
    if index == -1:
        keys[size] = key
        values[size] = value
        size += 1
    else:
        values[index] = value

def add_binary(a, b):    # 논리표에 있는 명제 변수들의 논릿값을 초기화하기 위한 함수
    len1 = len(a)
    len2 = len(b)
    carry = 0
    i = len1 - 1
    j = len2 - 1
    max_len = max(len1, len2)
    result = [''] * (max_len + 1)

    while i >= 0 or j >= 0:
        _sum = carry
        if i >= 0:
            _sum += int(a[i])
            i -= 1
        if j >= 0:
            _sum += int(b[j])
            j -= 1

        carry = _sum >> 1
        result[max_len] = str(_sum & 1)
        max_len -= 1

    if carry:
        result = ['1'] + result

```

```

        return ''.join(result)

def create_initial_table(num, letters):    # 결과 논리표를 만들고 초기화된 논릿값들로 채우는 함수
    for i in range(num):
        insert(letters[i], i)

    total_row = 1 << num
    arr = [[0] * (NUM_OF_OPERATOR + NUM_OF_OPERAND) for _ in range(total_row)]

    binary = ['0'] * num

    for i in range(total_row):
        for j in range(num):
            if binary[j] == '1':
                arr[i][j] = 1
            else:
                arr[i][j] = 0
            binary = add_binary(binary, '1')

    return arr

def main_operation(expression):    # 주 작업을 진행하는 메인 함수
    global expression_result, result_table
    operator_stack = Operator()
    value_stack = Value()
    operator_stack.top = -1
    value_stack.top = -1

    precedence = [
        [1,1,1,1,1],
        [1,1,0,0,0],
        [1,1,0,0,0],
        [1,1,0,0,0],
        [1,1,0,0,0]
    ]

    operator_string = ['!', '&', '>', '|']
    operator_hash = [0] * 128
    operator_hash[ord('!')] = 0
    operator_hash[ord('&')] = 1
    operator_hash[ord('>')] = 2
    operator_hash[ord('|')] = 3

    index = 0
    count = 0

    letters = [''] * (NUM_OF_OPERAND * 2)
    hash = [0] * 26

    letters[count] = '\0'

    total_row = 1 << NUM_OF_OPERAND
    popped_2 = [0] * total_row
    popped_1 = [0] * total_row
    value_vector = [0] * total_row

    for i in range(len(operator_string)):

```

```

insert(operator_string[i], i)

for char in expression:
    if char.isalpha() and hash(ord(char) - 65) == 0:
        letters[count] = char
        hash[ord(char) - 65] = 1
        count += 1

result_table = create_initial_table(NUM_OF_OPERAND, letters)

expression_result = [[''] * (EXP_SIZE) for _ in range(NUM_OF_OPERATOR + NUM_OF_OPERAND)]

Op_Stack = Operator()
Val_Stack = Value()

for i in range(0, NUM_OF_OPERAND):
    expression_result[i] = letters[i]

index = 0
result_index = NUM_OF_OPERAND

while index < len(expression):
    current = expression[index]
    if current == '!' and expression[index + 1].isalpha(): # '!' 뒤에 명제 변수가 올때
        index += 1
        current = expression[index] # '!' 뒤에 올 명제 변수를 읽음

        for i in range(0, total_row):
            value_vector[i] = result_table[i][get_index(current) - 4] # 4 = { !, &, >, | }
            value_vector[i] = 0 if (value_vector[i] == 1) else 1 # 논릿값을 부정
            result_table[i][result_index] = value_vector[i]

        temp_string = ""
        temp_string = temp_string + current
        expression_result[result_index] = temp_string
        Val_Stack.Push(temp_string, value_vector)
        result_index += 1

    elif current == '(':
        Op_Stack.Push(current)
    elif current == ')': # ')' 문자를 읽은 다음 연산자 두 개를 팝(pop)시킴
        temp_string = Op_Stack.Pop()
        if Op_Stack.Peek() == '(': # 스택의 탑에 있는 '('를 만날 때까지 돌림

            if temp_string == '!': # 스택에 있는 연산자가 '!'일 때

                Popped_Val = Val_Stack.Pop() # 탑에 있는 피연산자를 스택에서 꺼냄
                exp_1 = Popped_Val.exp

                for i in range(0, total_row):
                    result_table[i][result_index] = 0 if (Popped_Val.value[i] == 1)
else 1 # 논릿값을 부정
                    popped_1[i] = result_table[i][result_index]

                temp_string = temp_string + exp_1
                temp_string = Op_Stack.Pop() + temp_string + current # '('를 팝(pop)시킴
                expression_result[result_index] = temp_string
                Val_Stack.Push(temp_string, popped_1)
                result_index += 1

```

```

else: # 스택에 있는 연산자가 '!'가 아닐 때
    temp_from_stack = temp_string

    Popped_Val = Val_Stack.Pop() # 첫 번째 피연산자를 스택에서 꺼냄
    exp_2 = Popped_Val.exp
    for i in range(0, total_row):
        popped_2[i] = Popped_Val.value[i]

    Popped_Val = Val_Stack.Pop() # 두 번째 피연산자를 스택에서 꺼냄
    exp_1 = Popped_Val.exp
    for i in range(0, total_row):
        popped_1[i] = Popped_Val.value[i]

    exp_1 = exp_1 + temp_from_stack
    exp_1 = exp_1 + exp_2

    if temp_from_stack == '&':
        for i in range(0, total_row):
            value_vector[i] = 1 if popped_1[i] == 1 and popped_2[i] == 1
else 0

    elif temp_from_stack == '>':
        for i in range(0, total_row):
            if popped_1[i] == 1 and popped_2[i] == 0:
                value_vector[i] = 0
            else:
                value_vector[i] = 1
    elif temp_from_stack == '|':
        for i in range(0, total_row):
            value_vector[i] = 1 if popped_1[i] == 1 or popped_2[i] == 1
else 0

    for i in range(0, total_row):
        result_table[i][result_index] = value_vector[i]

    exp_1 = Op_Stack.Pop() + exp_1 + current # '('를 팝(pop)시킴
    expression_result[result_index] = exp_1
    result_index += 1

    Val_Stack.Push(exp_1, value_vector)

else: # 스택의 탑에 있는 연산자가 '('가 아닐 때
    Popped_Val = Val_Stack.Pop() # 탑에 있는 피연산자를 스택에서 꺼냄
    exp_1 = Popped_Val.exp

    for i in range(0, total_row):
        result_table[i][result_index] = 0 if (Popped_Val.value[i] == 1) else 1
        # 논릿값을 부정

        popped_1[i] = result_table[i][result_index]

    temp_string = temp_string + exp_1
    expression_result[result_index] = temp_string
    Val_Stack.Push(temp_string, popped_1)
    result_index += 1

```

```

temp_from_stack = Op_Stack.Pop()    # 스택의 탑에 있는 연산자를 Pop()

Popped_Val = Val_Stack.Pop()    # 첫 번째 피연산자를 스택에서 꺼냄
exp_2 = Popped_Val.exp
for i in range(0, total_row):
    popped_2[i] = Popped_Val.value[i]

Popped_Val = Val_Stack.Pop()    # 두 번째 피연산자를 스택에서 꺼냄
exp_1 = Popped_Val.exp
for i in range(0, total_row):
    popped_1[i] = Popped_Val.value[i]

exp_1 = exp_1 + temp_from_stack
exp_1 = exp_1 + exp_2

if temp_from_stack == '&':
    for i in range(0, total_row):
        value_vector[i] = 1 if popped_1[i] == 1 and popped_2[i] == 1    else
0
elif temp_from_stack == '>':
    for i in range(0, total_row):
        if popped_1[i] == 1 and popped_2[i] == 0:
            value_vector[i] = 0
        else:
            value_vector[i] = 1
elif temp_from_stack == '|':
    for i in range(0, total_row):
        value_vector[i] = 1 if popped_1[i] == 1 or popped_2[i] == 1    else 0

for i in range(0, total_row):
    result_table[i][result_index] = value_vector[i]

exp_1 = Op_Stack.Pop() + exp_1 + current    # '('를 팝(pop)시킴
expression_result[result_index] = exp_1
result_index += 1

Val_Stack.Push(exp_1, value_vector)

elif current.isalpha():    # 해당 문자가 명제 변수일 경우
    for i in range(0, total_row):
        value_vector[i] = result_table[i][get_index(current) - 4]    # 4 = | { !, &,
>, | } |
    temp_string = current
    Val_Stack.Push(temp_string, value_vector)

else:    # 해당 문자가 논리 연산자일 경우
    if precedence[get_index(Op_Stack.Peek()) + 1][get_index(current) + 1] == 0 :
        # 스택이 더 높음
        if Op_Stack.Peek() == '!':    # 스택에 있는 연산자가 '!'일 때
            temp_string = Op_Stack.Pop()    # "!"
            Popped_Val = Val_Stack.Pop()    # 탑에 있는 피연산자를 스택에서 꺼냄
            exp_1 = Popped_Val.exp

            for i in range(0, total_row):

```

```

    result_table[i][result_index] = 0 if (Popped_Val.value[i] == 1) else
1  # 논릿값을 부정
    popped_1[i] = result_table[i][result_index]

    temp_string = temp_string + exp_1
    expression_result[result_index] = temp_string
    Val_Stack.Push(temp_string, popped_1)
    result_index += 1
    Op_Stack.Push(current)

else:    # 스택에 있는 연산자가 '!'가 아닐 때

    temp_from_stack = Op_Stack.Pop()

    Popped_Val = Val_Stack.Pop()    # 첫 번째 피연산자를 스택에서 꺼냄
    exp_2 = Popped_Val.exp
    for i in range(0, total_row):
        popped_2[i] = Popped_Val.value[i]

    Popped_Val = Val_Stack.Pop()    # 두 번째 피연산자를 스택에서 꺼냄
    exp_1 = Popped_Val.exp
    for i in range(0, total_row):
        popped_1[i] = Popped_Val.value[i]

    exp_1 = exp_1 + temp_from_stack
    exp_1 = exp_1 + exp_2

    if temp_from_stack == '&':
        for i in range(0, total_row):
            value_vector[i] = 1 if popped_1[i] == 1 and popped_2[i] == 1 else
0
    elif temp_from_stack == '>':
        for i in range(0, total_row):
            if popped_1[i] == 1 and popped_2[i] == 0:
                value_vector[i] = 0
            else:
                value_vector[i] = 1
    elif temp_from_stack == '|':
        for i in range(0, total_row):
            value_vector[i] = 1 if popped_1[i] == 1 or popped_2[i] == 1 else
0

    for i in range(0, total_row):
        result_table[i][result_index] = value_vector[i]

    expression_result[result_index] = exp_1
    result_index += 1

    Val_Stack.Push(exp_1, value_vector)

    Op_Stack.Push(current)

    elif precedence[get_index(Op_Stack.Peek()) + 1][get_index(current) + 1] == 1:
        # input이 스택보다 높음
        Op_Stack.Push(current)
    index += 1

```



```

# Operator 스택에 남아 있는 것을 팝(pop)시킴
while Op_Stack.HeadArrive != 0:

    if Op_Stack.Peek() == '!':    # 스택에 있는 연산자가 '!'일 때
        temp_string = Op_Stack.Pop()    # "!"
        Popped_Val = Val_Stack.Pop()    # 탑에 있는 피연산자를 스택에서 꺼냄
        exp_1 = Popped_Val.exp

        for i in range(0, total_row):
            result_table[i][result_index] = 0 if (Popped_Val.value[i] == 1) else 1
                                                    # 논릿값을 부정

            popped_1[i] = result_table[i][result_index]

        temp_string = temp_string + exp_1
        expression_result[result_index] = temp_string
        Val_Stack.Push(temp_string, popped_1)
        result_index += 1

    else :
        temp_from_stack = Op_Stack.Pop()

        popped_value = Val_Stack.Pop()
        exp_2 = popped_value.exp
        popped_2 = popped_value.value[:total_row]

        popped_value = Val_Stack.Pop()
        exp_1 = popped_value.exp
        popped_1 = popped_value.value[:total_row]

        exp_1 = exp_1 + temp_from_stack
        exp_1 = exp_1 + exp_2

        if temp_from_stack == '&':
            for i in range(0, total_row):
                value_vector[i] = 1 if popped_1[i] == 1 and popped_2[i] == 1 else 0
        elif temp_from_stack == '>':
            for i in range(0, total_row):
                if popped_1[i] == 1 and popped_2[i] == 0:
                    value_vector[i] = 0
                else:
                    value_vector[i] = 1
        elif temp_from_stack == '|':
            for i in range(0, total_row):
                value_vector[i] = 1 if popped_1[i] == 1 or popped_2[i] == 1 else 0

        for i in range(0, total_row):
            result_table[i][result_index] = value_vector[i]

        expression_result[result_index] = exp_1
        result_index += 1

        Val_Stack.Push(exp_1, value_vector)

def display_and_write_on_file():
    outputFile = open("Truth Table.txt", "w")

    spaces = [0] * (NUM_OF_OPERAND + NUM_OF_OPERATOR)
    total_row = 1 << NUM_OF_OPERAND

```

```

print
print("\u250c", end = "");
# outputFile.write("\u250c");

for j in range(0, (NUM_OF_OPERAND + NUM_OF_OPERATOR)):
    spaces[j] = len(expression_result[j]) + 2

    for i in range(0, spaces[j]):
        print("\u2500", end = "")
        # outputFile.write("\x2500")

    if j != (NUM_OF_OPERAND + NUM_OF_OPERATOR) - 1:
        print("\u252c", end = "")
        # outputFile.write("\x252c")

print("\u2510\n\u2502", end = "")
# outputFile.write("\x2510\n\x2502")

for j in range(0, (NUM_OF_OPERAND + NUM_OF_OPERATOR)):
    print(' ', expression_result[j], ' ', end = "", sep = "")
    outputFile.write(" " + expression_result[j] + " ")
    print("\u2502", end = "")
    # outputFile.write("\x2502")

print("\n\u251c", end = "")
# outputFile.write("\n\x251c")
outputFile.write("\n")

for j in range(0, (NUM_OF_OPERAND + NUM_OF_OPERATOR)):

    for i in range(0, spaces[j]):
        print("\u2500", end = "")
        # outputFile.write("\x2500")

    if j != (NUM_OF_OPERAND + NUM_OF_OPERATOR) - 1:
        print("\u253c", end = "")
        # outputFile.write("\x253c")

print("\u2524", end = "")
# outputFile.write("\x2524")

for i in range(0, total_row):
    print()
    outputFile.write("\n")
    print("\u2502", end = "")
    # outputFile.write("\x2502")

    for j in range(0, (NUM_OF_OPERAND + NUM_OF_OPERATOR)):
        if spaces[j] % 2 == 1:
            for m in range(0, spaces[j] // 2):
                print(" ", end = "")
                outputFile.write(" ")
            if result_table[i][j] == 1:
                print("T", end = "");
                outputFile.write("T");
            elif result_table[i][j] == 0:
                print("F", end = "");

```

```

        outputFile.write("F");
        for m in range(0, spaces[j] // 2):
            print(" ", end = "")
            outputFile.write(" ")

    else :
        for m in range(0, (spaces[j] // 2) - 1):
            print(" ", end = "")
            outputFile.write(" ")
        if result_table[i][j] == 1:
            print("T", end = "");
            outputFile.write("T");
        elif result_table[i][j] == 0:
            print("F", end = "");
            outputFile.write("F");
        for m in range(0, spaces[j] // 2):
            print(" ", end = "")
            outputFile.write(" ")

    print("\u2502", end = "")
    # outputFile.write("\x2502")

    if i != total_row - 1:
        print("\n\u251c", end = "")
        # outputFile.write("\n\x251c")
        # outputFile.write("\n")

        for k in range(0, (NUM_OF_OPERAND + NUM_OF_OPERATOR)):
            for m in range(0, spaces[k]):
                print("\u2500", end = "")
                # outputFile.write("\x2500")

            if k != (NUM_OF_OPERAND + NUM_OF_OPERATOR) - 1:
                print("\u253c", end = "")
                # outputFile.write("\x253c")

        print("\u2524", end = "")
        # outputFile.write("\x2524")

    print("\n\u2514", end = "")
    # outputFile.write("\n\x2514")
    # outputFile.write("\n")

    for k in range(0, (NUM_OF_OPERAND + NUM_OF_OPERATOR)):
        for m in range(0, spaces[k]):
            print("\u2500", end = "")
            # outputFile.write("\x2500")

        if k != (NUM_OF_OPERAND + NUM_OF_OPERATOR) - 1:
            print("\u2534", end = "")
            # outputFile.write("\x2534")

    print("\u2518", end = "")
    # outputFile.write("\x2518")

    print("\n")
    outputFile.close()

```

```

if __name__ == "__main__":
    NUM_OF_OPERAND = 2
    expression = input("합성명제를 입력하라.: ")
    expression = expression.replace(' ', '')
    if not check_and_despace(expression, NUM_OF_OPERAND):
        print("잘못된 합성명제이다!\a")
    else:
        main_operation(expression)
        display_and_write_on_file()

# !(!(P&Q)&(!(P!Q)))
# !(P & Q) & (P ! Q)
# !(!(!(!(P&Q))))))
# !(!(!(!(P&Q)!P>!(P!Q))))))

```

[프로그래밍 실습 2]

```

EXP_SIZE = 50                # 합성명제의 길이를 위한 상수
OPERATOR_NUMBER = 10        # 명제 변수의 개수를 위한 상수
MAX_SIZE_TABLE = 26         # 논리표의 크기를 위한 상수

size = 0                    # 연산자
keys = [''] * MAX_SIZE_TABLE # 연산자를 저장하기 위한 문자형 배열
values = [0] * MAX_SIZE_TABLE # 비연산자를 저장하기 위한 배열(0으로 초기화)
NUM_OF_OPERAND = 0          # 비연산자의 개수를 위한 변수
NUM_OF_OPERATOR = 0         # 연산자의 개수를 위한 변수

result_table = []           # 결과 논리표를 저장하기 위한 메모리 공간
expression_result = []      # 계산된 합성명제를 저장하기 위한 메모리 공간

class Operator:
    def __init__(self, c = "", headarrive = 0, next = None):
        self.op = [''] * EXP_SIZE
        self.HeadArrive = headarrive
        self.Next = next

    def Push(head, c):
        Temp = head
        if head.HeadArrive == 0:
            head.Next = None # 스택의 탑 노드이므로 Next는 None(C 코드에서는 Null)
            head.HeadArrive = 1
            head.op = c
            return head      # 헤드값을 리턴
        else:
            while Temp.Next != None: # 스택의 탑으로 찾아감
                Temp = Temp.Next

            NewNode = Operator(c, 0) # 새로운 노드를 할당
            Temp.Next = NewNode
            head.HeadArrive += 1

    def Pop(head):
        Temp = head
        if head.Next == None:
            if head.HeadArrive == 0:
                print("잘못된 Pop연산.\n")

```

```

        return None
    head.HeadArrive = 0
    return head.op    # 스택의 탑값인 헤드를 리턴
else:
    while Temp.Next.Next != None:
        Temp = Temp.Next
    Temp1 = Temp.Next
    Temp.Next = None
    head.HeadArrive -= 1
    return Temp1.op[head.HeadArrive]    # 제거될 노드의 주소와 탑 노드 직전 노드의 주
    소를 리턴

def Peek(head):
    Temp = head
    if head.Next == None:
        if head.HeadArrive == 0:
            # printf("잘못된 Peek연산.\n")
            return '\0'
        return head.op    # 스택의 탑값인 헤드를 리턴
    else:
        while Temp.Next != None:
            Temp = Temp.Next
        return Temp    # 제거될 노드의 주소와 탑 노드 직전 노드의 주소를 리턴

class Value:
    def __init__(self, c = "", value =[0], headarrive = 0, next = None):
        self.exp = [[''] * EXP_SIZE for _ in range(EXP_SIZE)]
        self.value = [0] * (1 << OPERATOR_NUMBER)
        self.HeadArrive = headarrive
        self.Next = next

    def Push(head, c, value):
        Temp = head
        if head.HeadArrive == 0:
            head.Next = None    # 스택의 탑 노드이므로 Next는 None(C 코드에서는 Null)
            head.HeadArrive = 1
            head.exp = c
            for i in range(0, 1 << NUM_OF_OPERAND):
                head.value[i] = value[i]
            return head    # 헤드값을 리턴
        else:
            while Temp.Next != None:    # 스택의 탑 노드까지 찾아감
                Temp = Temp.Next

            NewNode = Value(c, 0)    # 새로운 노드를 할당
            Temp.Next = NewNode
            NewNode.exp = c
            for i in range(0, 1 << NUM_OF_OPERAND):
                NewNode.value[i] = value[i]
            head.HeadArrive += 1
            # 새로운 노드를 리턴

    def Pop(head):
        Temp = head
        if head.Next == None:
            if head.HeadArrive == 0:
                printf("잘못된 Pop연산.\n")
                return None

```

```

        head.HeadArrive = 0
        return head    # 스택의 탑값인 헤드를 리턴
    else:
        while Temp.Next.Next != None:
            Temp = Temp.Next
        Temp1 = Temp.Next
        Temp.Next = None
        head.HeadArrive -= 1
        return Temp1    # 제거될 노드의 주소와 탑 노드 직전 노드의 주소를 리턴

def check_and_despace(exp, num):
    hash_table = [0] * 26
    i = len(exp) - 1
    while i >= 0:
        if not (exp[i].isalpha() or exp[i] == '|' or exp[i] == '&' or exp[i] == '!' or exp[i]
        == '>' or exp[i] == '(' or exp[i] == ')'):
            return False
        elif exp[i].isalpha() and not hash_table[ord(exp[i]) - 65]:
            hash_table[ord(exp[i]) - 65] = 1
            num -= 1
        elif exp[i] == '|' or exp[i] == '&' or exp[i] == '!' or exp[i] == '>':
            global NUM_OF_OPERATOR
            NUM_OF_OPERATOR += 1

        i -= 1
    if num == 0:
        return True
    return False

def get_index(key):
    for i in range(0, size):
        if keys[i] == key:
            return i
    return -1

def insert(key, value):
    global size
    index = get_index(key)
    if index == -1:
        keys[size] = key
        values[size] = value
        size += 1
    else:
        values[index] = value

def get(key):
    index = get_index(key)
    if index == -1:
        return -1
    return values[index]

def add_binary(a, b):
    len1 = len(a)
    len2 = len(b)
    carry = 0
    i = len1 - 1
    j = len2 - 1
    max_len = max(len1, len2)

```

```

result = [''] * (max_len + 1)

while i >= 0 or j >= 0:
    _sum = carry
    if i >= 0:
        _sum += int(a[i])
        i -= 1
    if j >= 0:
        _sum += int(b[j])
        j -= 1

    carry = _sum >> 1
    result[max_len] = str(_sum & 1)
    max_len -= 1

if carry:
    result = ['1'] + result

return ''.join(result)

def create_initial_table(num, letters):
    for i in range(num):
        insert(letters[i], i)

    total_row = 1 << num
    arr = [[0] * (NUM_OF_OPERATOR + NUM_OF_OPERAND) for _ in range(total_row)]

    binary = ['0'] * num

    for i in range(total_row):
        for j in range(num):
            if binary[j] == '1':
                arr[i][j] = 1
            else:
                arr[i][j] = 0
            binary = add_binary(binary, '1')

    return arr

def main_operation(expression):
    global expression_result, result_table
    operator_stack = Operator()
    value_stack = Value()
    operator_stack.top = -1
    value_stack.top = -1

    # 우선순위표
    precedence = [
        [1,1,1,1,1],
        [1,1,0,0,0],
        [1,1,0,0,0],
        [1,1,0,0,0],
        [1,1,0,0,0],
        [1,1,0,0,0]
    ]

    operator_string = ['!', '&', '>', '|']
    operator_hash = [0] * 128
    operator_hash[ord('!')] = 0

```

```

operator_hash[ord('&')] = 1
operator_hash[ord('>')] = 2
operator_hash[ord('!')] = 3

index = 0
count = 0

letters = [''] * (NUM_OF_OPERAND * 2)
hash = [0] * 26

letters[count] = '\0'

total_row = 1 << NUM_OF_OPERAND
popped_2 = [0] * total_row
popped_1 = [0] * total_row
value_vector = [0] * total_row

for i in range(len(operator_string)):
    insert(operator_string[i], i)

for char in expression:
    if char.isalpha() and hash[ord(char) - 65] == 0:
        letters[count] = char
        hash[ord(char) - 65] = 1
        count += 1

result_table = create_initial_table(NUM_OF_OPERAND, letters)

expression_result = [[''] * (EXP_SIZE) for _ in range(NUM_OF_OPERATOR + NUM_OF_OPERAND)]

Op_Stack = Operator()
Val_Stack = Value()

for i in range(0, NUM_OF_OPERAND):
    expression_result[i] = letters[i]

index = 0
result_index = NUM_OF_OPERAND

while index < len(expression):
    current = expression[index]

    if current.isalpha():
        # 해당 문자가 명제 변수일 경우
        for i in range(0, total_row):
            value_vector[i] = result_table[i][get_index(current) - 4] # 4 = ! { !, &,
>, ! } !
            temp_string = current
            Val_Stack.Push(temp_string, value_vector)

    else:
        # 해당 문자가 논리 연산자일 경우
        if current == '!':
            index += 1
            current = expression[index] # '!' 뒤에 올 명제 변수를 읽음

            for i in range(0, total_row):
                value_vector[i] = result_table[i][get_index(current) - 4] # 4 = ! { !,
&, >, ! } !

                value_vector[i] = 0 if (value_vector[i] == 1) else 1 # 논릿값을 부정
                result_table[i][result_index] = value_vector[i]

```



```

temp_string = "!"
temp_string = temp_string + current
expression_result[result_index] = temp_string
Val_Stack.Push(temp_string, value_vector)
result_index += 1

elif precedence[get_index(Op_Stack.Peek()) + 1][get_index(current) + 1] == 0 :
    # 스택이 더 높음

temp_from_stack = Op_Stack.Pop()

Popped_Val = Val_Stack.Pop()    # 첫번째 피연산자를 스택에서 꺼냄
exp_2 = Popped_Val.exp
for i in range(0, total_row):
    popped_2[i] = Popped_Val.value[i]

Popped_Val = Val_Stack.Pop()    # 두번째 피연산자를 스택에서 꺼냄
exp_1 = Popped_Val.exp
for i in range(0, total_row):
    popped_1[i] = Popped_Val.value[i]

exp_1 = exp_1 + temp_from_stack
exp_1 = exp_1 + exp_2

if temp_from_stack == '&':
    for i in range(0, total_row):
        value_vector[i] = 1 if popped_1[i] == 1 and popped_2[i] == 1 else 0
elif temp_from_stack == '>':
    for i in range(0, total_row):
        if popped_1[i] == 1 and popped_2[i] == 0:
            value_vector[i] = 0
        else:
            value_vector[i] = 1
elif temp_from_stack == '|':
    for i in range(0, total_row):
        value_vector[i] = 1 if popped_1[i] == 1 or popped_2[i] == 1 else 0

for i in range(0, total_row):
    result_table[i][result_index] = value_vector[i]

expression_result[result_index] = exp_1
result_index += 1

Val_Stack.Push(exp_1, value_vector)

Op_Stack.Push(current)

elif precedence[get_index(Op_Stack.Peek()) + 1][get_index(current) + 1] == 1:
    # input이 스택보다 더 높음
    Op_Stack.Push(current)
index += 1

# Operator 스택에 남아 있는 것을 팝(pop)시킴
while Op_Stack.HeadArrive != 0:
    temp_from_stack = Op_Stack.Pop()

```

```

popped_value = Val_Stack.Pop()
exp_2 = popped_value.exp
popped_2 = popped_value.value[:total_row]

popped_value = Val_Stack.Pop()
exp_1 = popped_value.exp
popped_1 = popped_value.value[:total_row]

exp_1 = exp_1 + temp_from_stack
exp_1 = exp_1 + exp_2

if temp_from_stack == '&':
    for i in range(0, total_row):
        value_vector[i] = 1 if popped_1[i] == 1 and popped_2[i] == 1 else 0
elif temp_from_stack == '>':
    for i in range(0, total_row):
        if popped_1[i] == 1 and popped_2[i] == 0:
            value_vector[i] = 0
        else:
            value_vector[i] = 1
elif temp_from_stack == '|':
    for i in range(0, total_row):
        value_vector[i] = 1 if popped_1[i] == 1 or popped_2[i] == 1 else 0

for i in range(0, total_row):
    result_table[i][result_index] = value_vector[i]

expression_result[result_index] = exp_1
result_index += 1

Val_Stack.Push(exp_1, value_vector)

```

Truth Table.txt에 저장된 결과 논리표를 출력

```

def display_and_write_on_file():
    outputFile = open("Truth Table.txt", "w")

    spaces = [0] * (NUM_OF_OPERAND + NUM_OF_OPERATOR)
    total_row = 1 << NUM_OF_OPERAND

    print
    print("\u250c", end = "");
    # outputFile.write("\u250c");

    for j in range(0, (NUM_OF_OPERAND + NUM_OF_OPERATOR)):
        spaces[j] = len(expression_result[j]) + 2

        for i in range(0, spaces[j]):
            print("\u2500", end = "")
            # outputFile.write("\x2500")

        if j != (NUM_OF_OPERAND + NUM_OF_OPERATOR) - 1:
            print("\u252c", end = "")
            # outputFile.write("\x252c")

    print("\u2510\n\u2502", end = "")
    # outputFile.write("\x2510\n\x2502")

    for j in range(0, (NUM_OF_OPERAND + NUM_OF_OPERATOR)):
        print(' ', expression_result[j], ' ', end = "", sep = "")

```

```

        outputFile.write(" " + expression_result[j] + " ")
        print("\u2502", end = "")
        # outputFile.write("\x2502")

    print("\n\u251c", end = "")
    # outputFile.write("\n\x251c")
    outputFile.write("\n")

    for j in range(0, (NUM_OF_OPERAND + NUM_OF_OPERATOR)):

        for i in range(0, spaces[j]):
            print("\u2500", end = "")
            # outputFile.write("\x2500")

        if j != (NUM_OF_OPERAND + NUM_OF_OPERATOR) - 1:
            print("\u253c", end = "")
            # outputFile.write("\x253c")

    print("\u2524", end = "")
    # outputFile.write("\x2524")

    for i in range(0, total_row):
        print()
        outputFile.write("\n")
        print("\u2502", end = "")
        # outputFile.write("\x2502")

        for j in range(0, (NUM_OF_OPERAND + NUM_OF_OPERATOR)):
            if spaces[j] % 2 == 1:
                for m in range(0, spaces[j] // 2):
                    print(" ", end = "")
                    outputFile.write(" ")
                if result_table[i][j] == 1:
                    print("T", end = "");
                    outputFile.write("T");
                elif result_table[i][j] == 0:
                    print("F", end = "");
                    outputFile.write("F");
                for m in range(0, spaces[j] // 2):
                    print(" ", end = "")
                    outputFile.write(" ")

            else :
                for m in range(0, (spaces[j] // 2) - 1):
                    print(" ", end = "")
                    outputFile.write(" ")
                if result_table[i][j] == 1:
                    print("T", end = "");
                    outputFile.write("T");
                elif result_table[i][j] == 0:
                    print("F", end = "");
                    outputFile.write("F");
                for m in range(0, spaces[j] // 2):
                    print(" ", end = "")
                    outputFile.write(" ")

        print("\u2502", end = "")
        # outputFile.write("\x2502")

```

```

        if i != total_row - 1:
            print("\n\u251c", end = "")
            # outputFile.write("\n\u251c")
            # outputFile.write("\n")

            for k in range(0, (NUM_OF_OPERAND + NUM_OF_OPERATOR)):
                for m in range(0, spaces[k]):
                    print("\u2500", end = "")
                    # outputFile.write("\x2500")

                if k != (NUM_OF_OPERAND + NUM_OF_OPERATOR) - 1:
                    print("\u253c", end = "")
                    # outputFile.write("\x253c")

            print("\u2524", end = "")
            # outputFile.write("\x2524")

        print("\n\u2514", end = "")
        # outputFile.write("\n\u2514")
        # outputFile.write("\n")

        for k in range(0, (NUM_OF_OPERAND + NUM_OF_OPERATOR)):
            for m in range(0, spaces[k]):
                print("\u2500", end = "")
                # outputFile.write("\x2500")

            if k != (NUM_OF_OPERAND + NUM_OF_OPERATOR) - 1:
                print("\u2534", end = "")
                # outputFile.write("\x2534")

        print("\u2518", end = "")
        # outputFile.write("\x2518")

        print("\n")
        outputFile.close()

if __name__ == "__main__":
    NUM_OF_OPERAND = int(input("명제 변수의 개수를 입력하라.: "))
    expression = input("합성명제를 입력하라.: ")

    expression = expression.replace(' ', '')
    if not check_and_despace(expression, NUM_OF_OPERAND):
        print("잘못된 합성명제이다!\a")
    else:
        main_operation(expression)
        display_and_write_on_file()

```

[프로그래밍 실습 3]

```

from __future__ import print_function
from sys import stdin

def printf(str, *args):

```

```
print(str % args, end = '')

for i in range(1, 2001):
    Per_num = 0          # 배수들의 합을 위한 변수
    j = 1                # 배수를 위한 변수
    # 반복문으로 i에 대한 배수들의 합을 계산
    while j < i:
        if(i % j) == 0:
            Per_num += j
        j += 1
    # 계산된 합과 i값이 같으면 완전수라고 출력
    if Per_num == i:
        printf("%5d 는 완전수이다!\n" % i)

stdin.readline()
```