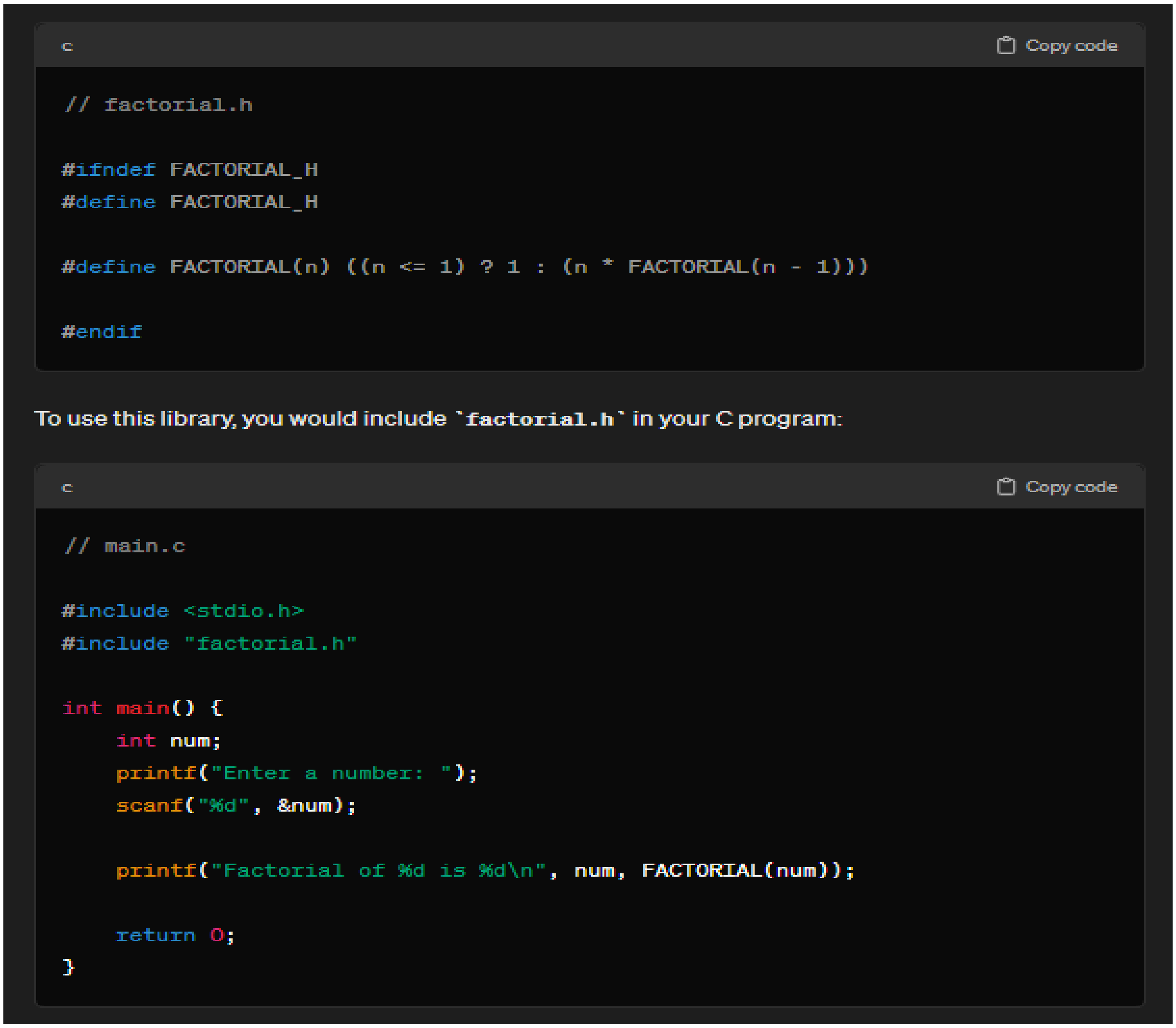
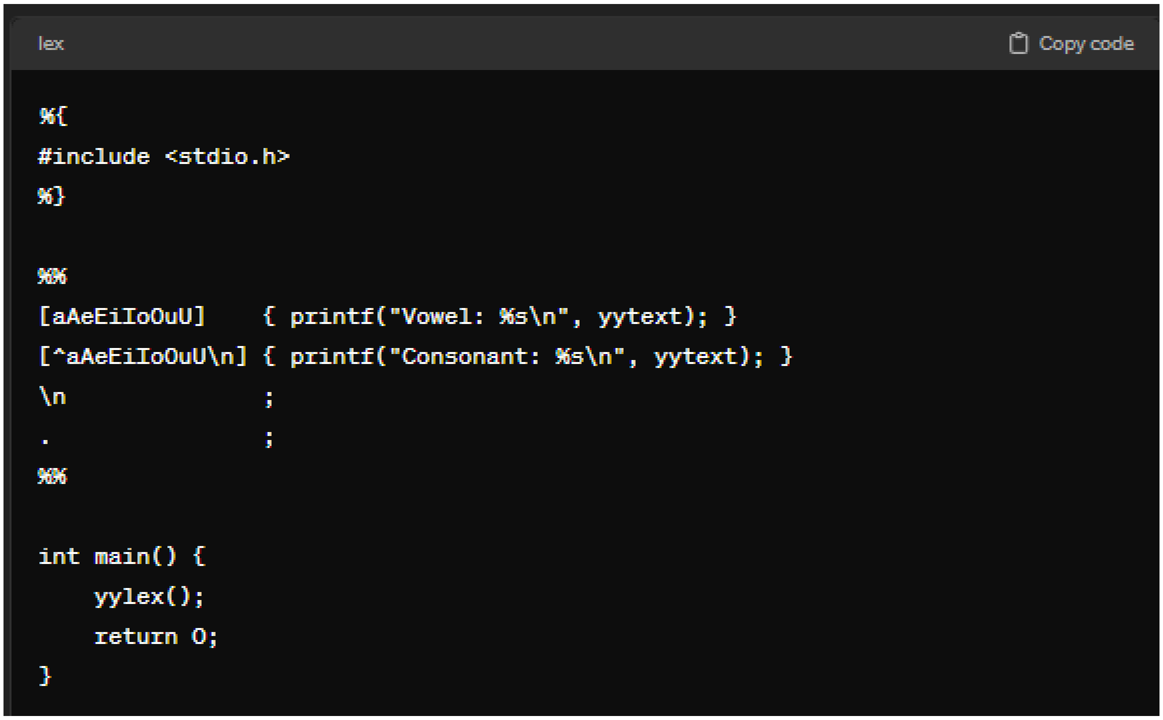
1. **WAP to create your own 'C' library using macros to find factorial of a number [05 marks]**
2. **Write a Lex program to count the number of vowels and consonants of a token [05 marks]**
3. **Write a program to display any given 3AC statement in Quadruples form [05 marks]**



```c
// factorial.h

#ifndef FACTORIAL_H
#define FACTORIAL_H

#define FACTORIAL(n) ((n <= 1) ? 1 : (n * FACTORIAL(n - 1)))

#endif
```

To use this library, you would include `factorial.h` in your C program:

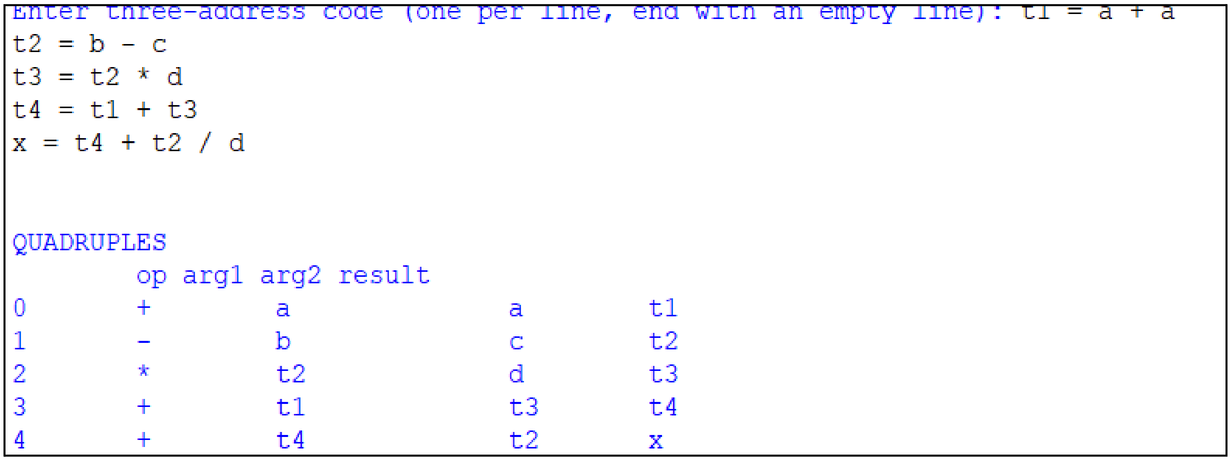
```c
// main.c

#include <stdio.h>
#include "factorial.h"

int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);

    printf("Factorial of %d is %d\n", num, FACTORIAL(num));

    return 0;
}
```



```
%{
#include <stdio.h>
%}

%%
[aAeEiIoOuU]    { printf("Vowel: %s\n", yytext); }
[^aAeEiIoOuU\n] { printf("Consonant: %s\n", yytext); }
\n              ;
.               ;
%%

int main() {
    yylex();
    return 0;
}
```

```
def check_3ac(operation, dict_3ac):
    for key, value in dict_3ac.items():
        if value == operation:
            return key
    return False

def convert_to_quadruples(address_code_dict):
    quadruple_list = dict()
    list_no = 0

    for key, value in address_code_dict.items():
        items = value.split()
        if len(items) <= 2:
            i = value.find('=')
            quadruple_list[list_no] = {"op": "=", "arg1": items[0], "arg2": " ", "result": key}
        else:
            for item in ['-', '+', '*', '/']:
                if item in value:
                    i = value.find(item)
                    quadruple_list[list_no] = {"op": items[1], "arg1": items[0], "arg2": items[2],
"result": key}
                    break
        list_no += 1

    return quadruple_list
```

```
# Take three-address code as input
three_ac = input("Enter three-address code (one per line, end with an empty line): ")
three_ac_dict = dict()

while three_ac:
    key, value = three_ac.split(" = ")
    three_ac_dict[key] = value
    three_ac = input()

# Convert three-address code to quadruples
quadruples = convert_to_quadruples(three_ac_dict)

# Print the quadruples
print("\nQUADRUPLES")
print("\top arg1 arg2 result")
for key, value in quadruples.items():
    print(f"{key}\t{value['op']}\t {value['arg1']}\t\t{value['arg2']}\t {value['result']}")
```

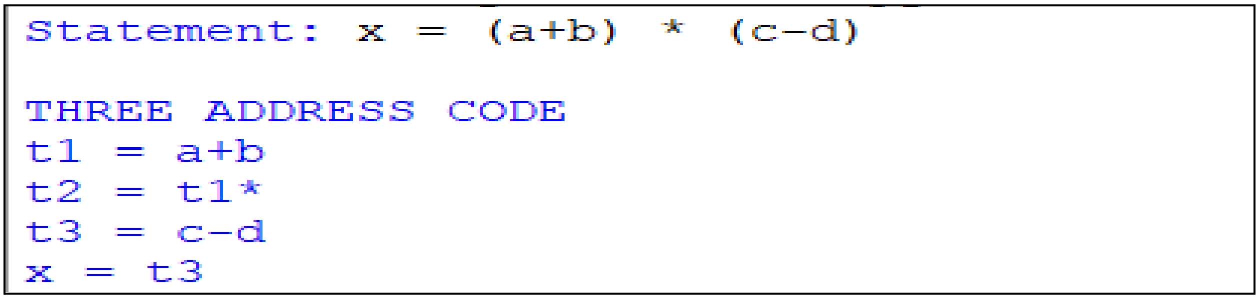


```
Enter three-address code (one per line, end with an empty line): t1 = a + a
t2 = b - c
t3 = t2 * d
t4 = t1 + t3
x = t4 + t2 / d


QUADRUPLES
        op arg1 arg2 result
0       +        a              a        t1
1       -        b              c        t2
2       *        t2             d        t3
3       +        t1             t3       t4
4       +        t4             t2       x
```

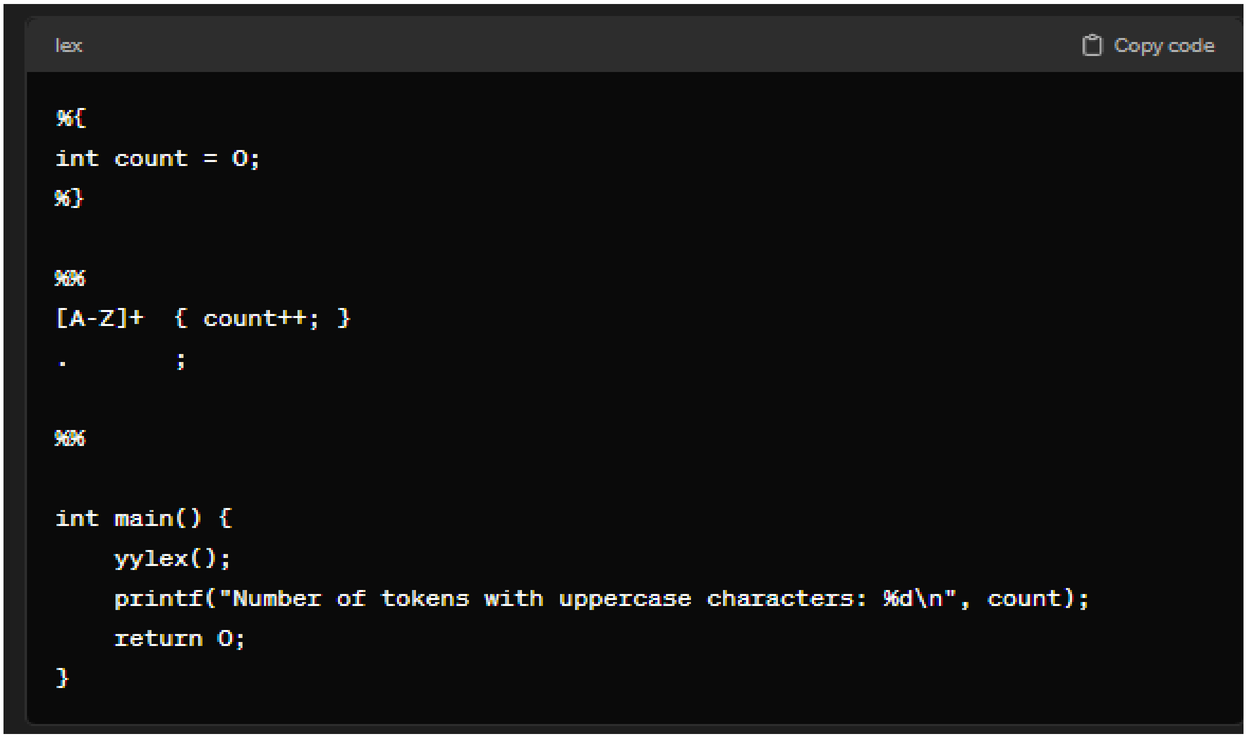
**1. Write a program to convert the given computation into three address code. [10 marks]**
**2. Write a Lex program to count the number of tokens with uppercase characters. [05 marks]**

x = (a+b) * (c-d);

```
def precedence(c):
    if c == '/' or c == '*':
        return 2
    elif c == '+' or c == '-':
        return 1
    else:
        return -1

def infix_to_postfix(s):
    result = []
    stack = []
    assignment_operator = False
    assignment = ""
    for i in range(len(s)):
        c = s[i]
        if ('a' <= c <= 'z') or ('A' <= c <= 'Z') or ('0' <= c <= '9'):
            result.append(c)
        elif c == '(':
            stack.append(c)
        elif c == ')':
            while stack and stack[-1] != '(':
                result.append(stack.pop())
            stack.pop()
        elif c == '=':
            assignment_operator = True
            assignment = result.pop()
        else:
            while stack and (precedence(s[i]) <= precedence(stack[-1])):
                result.append(stack.pop())
            stack.append(c)
    while stack:
        result.append(stack.pop())
    if assignment_operator:
        result.extend([assignment, '='])
    return result

def check_3ac(operation, dict_3ac):
    for key, value in dict_3ac.items():
        if value == operation:
            return key
    return False

def convert_to_3ac(expression):
    count = 0
    three_ac_dict = dict()
    operand = list()
    for i, item in enumerate(expression):
        if item == '+' or item == '-' or item == '/' or item == '*':
            op2 = operand.pop()
            op1 = operand.pop()
            in_3ac = check_3ac(f'{op1}{item}{op2}', three_ac_dict)
            if not in_3ac:
                count += 1
                three_ac_dict[f't{count}'] = f'{op1}{item}{op2}'
                operand.append(f't{count}')
            else:
                operand.append(in_3ac)
        elif item == '=':
            three_ac_dict[operand.pop()] = f't{count}'
        else:
            operand.append(item)
    return three_ac_dict

statement = input("Statement: ")  # x=a+a*(b-c)+(b-c)/d
postfix = infix_to_postfix(statement)
address_codes = convert_to_3ac(postfix)
print("\nTHREE ADDRESS CODE")
for key, value in address_codes.items():
    print(f'{key} = {value}')
```



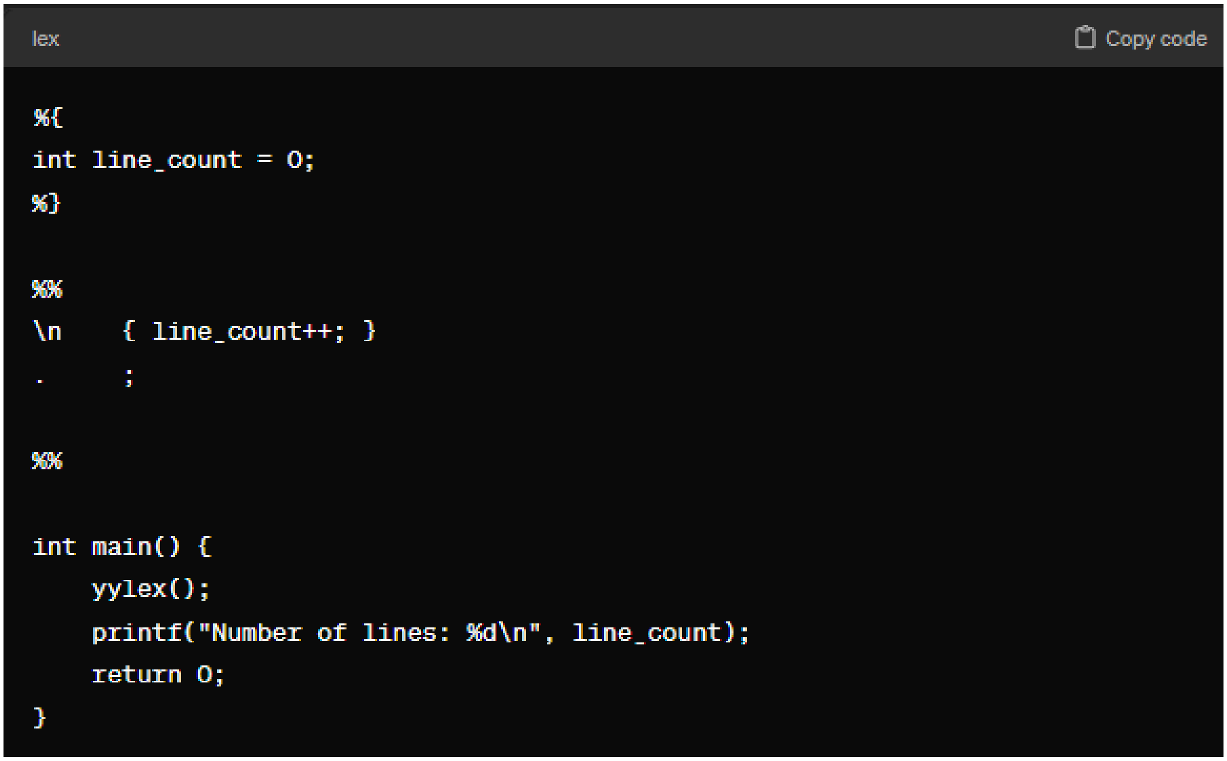
```
Statement: x = (a+b) * (c-d)

THREE ADDRESS CODE
t1 = a+b
t2 = t1*
t3 = c-d
x = t3
```



```
%{
int count = 0;
%}

%%
[A-Z]+  { count++; }
.       ;

%%

int main() {
    yylex();
    printf("Number of tokens with uppercase characters: %d\n", count);
    return 0;
}
```

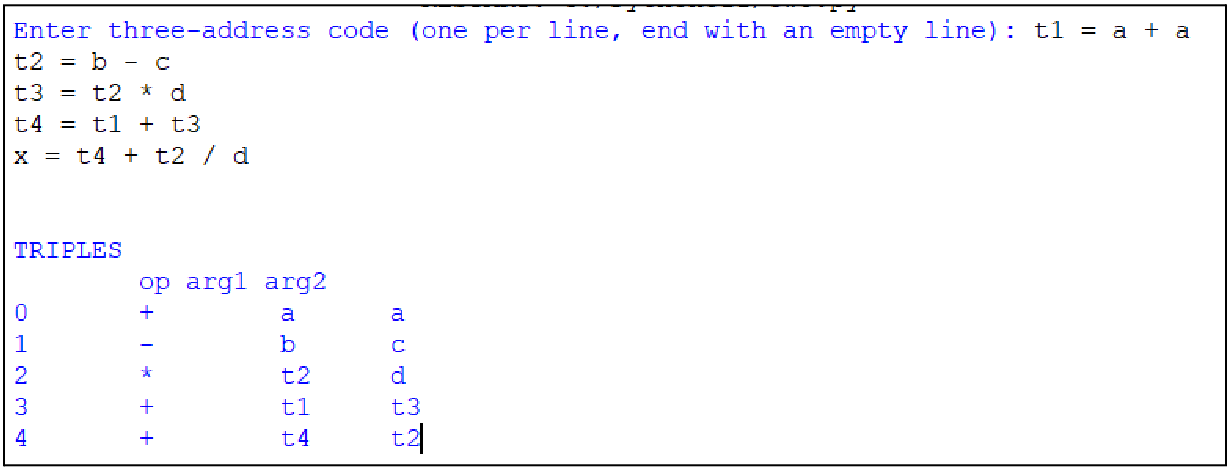
**1. Write a program to create your own 'C' library using macros that can find the area of square and rectangle [05 marks]**
**2. Write a Lex program to print the number of lines in the source program. [05 marks]**
**3. Consider the following Three address code and display Triples**
**f=c+d**
**e=a-f**
**g=b*e**

```
#include <stdio.h>
// Macros to calculate the area of a square and rectangle
#define SQUARE_AREA(side) ((side) * (side))
#define RECTANGLE_AREA(length, width) ((length) * (width))

int main() {
 // Test values
 float square_side = 5.0;
 float rectangle_length = 6.0;
 float rectangle_width = 4.0;

 // Calculate areas using macros
 float square_area = SQUARE_AREA(square_side);
 float rectangle_area = RECTANGLE_AREA(rectangle_length, rectangle_width);

 // Print the results
 printf("Area of square with side %.2f: %.2f\n", square_side, square_area);
 printf("Area of rectangle with length %.2f and width %.2f: %.2f\n", rectangle_length,
rectangle_width, rectangle_area);

 return 0;
}
```



```
%{
int line_count = 0;
%}

%%
\n    { line_count++; }
.     ;

%%

int main() {
    yylex();
    printf("Number of lines: %d\n", line_count);
    return 0;
}
```

```
def check_3ac(operation, dict_3ac):
    for key, value in dict_3ac.items():
        if value == operation:
            return key
    return False

def convert_to_triples(address_code_dict):
    triples_dict = dict()
    dict_no = 0

    for key, value in address_code_dict.items():
        items = value.split()
        if len(items) <= 2:
            triples_dict[dict_no] = {'op': '=', 'arg1': key, 'arg2': items[0]}
        else:
            for item in ['-', '+', '*', '/']:
                if item in value:
                    i = value.find(item)
                    triples_dict[dict_no] = {'op': items[1], 'arg1': items[0], 'arg2': items[2]}
                    break
        dict_no += 1

    return triples_dict

# Take three-address code as input
three_ac = input("Enter three-address code (one per line, end with an empty line): ")
three_ac_dict = dict()

while three_ac:
    key, value = three_ac.split(" = ")
    three_ac_dict[key] = value
    three_ac = input()

# Convert three-address code to triples
triples = convert_to_triples(three_ac_dict)

# Print the triples
print("\nTRIPLES")
print("\top arg1 arg2")
for key, value in triples.items():
    print(f"{key}\t{value['op']}\t {value['arg1']}\t{value['arg2']}")
```

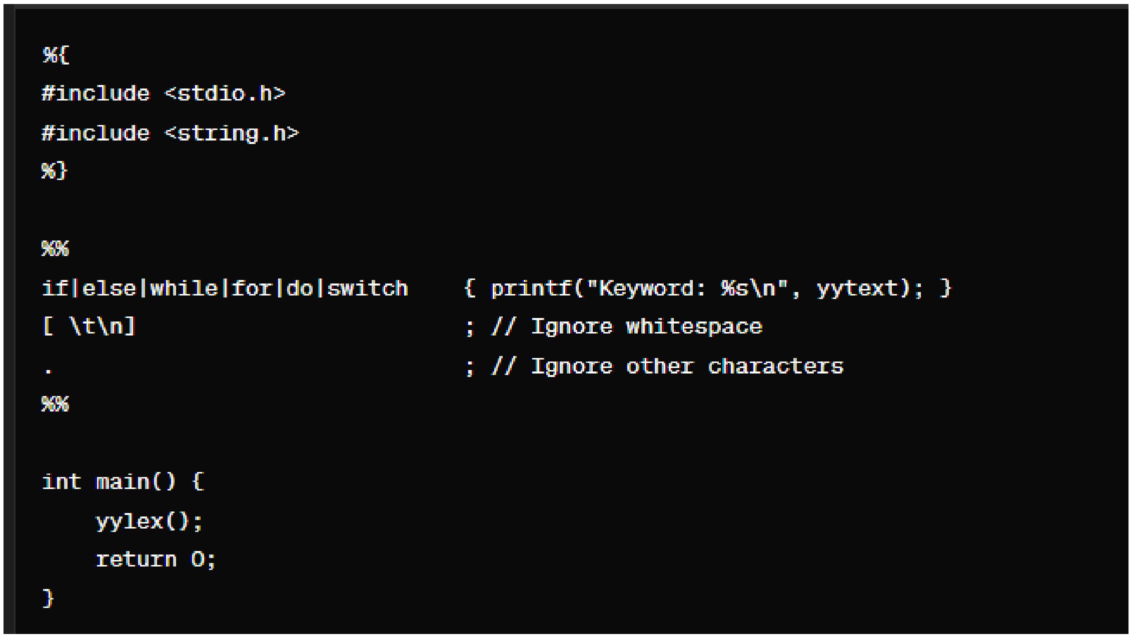


```
Enter three-address code (one per line, end with an empty line): t1 = a + a
t2 = b - c
t3 = t2 * d
t4 = t1 + t3
x = t4 + t2 / d


TRIPLES
        op arg1 arg2
0       +        a       a
1       -        b       c
2       *        t2      d
3       +        t1      t3
4       +        t4      t2
```

**1. Consider the following program, Display the Pass-1 of the Program [15 marks]**
**START 501**
**A DS 1**
**B DS 1**
**C DS 1**
**READ A**
**READ B**
**MOVER AREG, A**
**ADD AREG, B**
**MOVEM AREG, C**
**PRINT C**
**END**
**—------------------------------------------------------------------------------------------------------------------------**

**1. Write a program to remove left recursion from a given context free grammar. [10 marks]**
**Nonterminal ={S,L} terminal={( , x, , ,) }**
**S (L)/x**
**L L,S/S**
**2. Write a lex program to identify all the keywords (if, else, while etc.) [05 marks]**

```
def add_production(production_str):
    """
    Adds a production rule to the grammar.
    """
    production_str = production_str.replace(" ", "").replace("\n", "")
    lhs, rhs = production_str.split("->")
    rhs_alternatives = rhs.split("|")
    grammar[lhs] = rhs_alternatives
    return grammar

def remove_direct_left_recursion(grammar, non_terminal):
    """
    Removes direct left recursion from a given non-terminal.
    """
    alpha = []
    beta = []
    for rhs in grammar[non_terminal]:
        if rhs[0] == non_terminal:
            alpha.append(rhs[1:] + [f"{non_terminal}'"])
        else:
            beta.append([*rhs, f"{non_terminal}'"])
    grammar[non_terminal] = beta
    grammar[f"{non_terminal}'"] = alpha or [["epsilon"]]
    return grammar

def remove_indirect_left_recursion(grammar, non_terminal):
    """
    Removes indirect left recursion from a given non-terminal.
    """
    new_rhs = []
    for rhs in grammar[non_terminal]:
        if rhs[0] in grammar and rhs[0] != non_terminal:
            for inner_rhs in grammar[rhs[0]]:
                new_rhs.append(inner_rhs + rhs[1:])
        else:
            new_rhs.append(rhs)
    grammar[non_terminal] = new_rhs
    return grammar

def remove_left_recursion(grammar):
    """
    Removes left recursion from the given grammar.
    """
    new_grammar = {}
    for i, (non_terminal, rhs_list) in enumerate(grammar.items(), start=1):
        new_non_terminal = f"A{i}"
        new_grammar[new_non_terminal] = rhs_list
        grammar = remove_direct_left_recursion(new_grammar, new_non_terminal)
        grammar = remove_indirect_left_recursion(new_grammar, new_non_terminal)
        new_grammar[non_terminal] = [x[:-1] for x in new_grammar[new_non_terminal]]
    return new_grammar

# Usage
n = int(input("Enter the number of production rules: "))
grammar = {}
for _ in range(n):
    production_rule = input("Enter a production rule: ")
    add_production(production_rule)

result = remove_left_recursion(grammar)
for non_terminal, rhs_list in result.items():
    print(f"{non_terminal} -> {' | '.join(".join(rhs) for rhs in rhs_list)}")
```

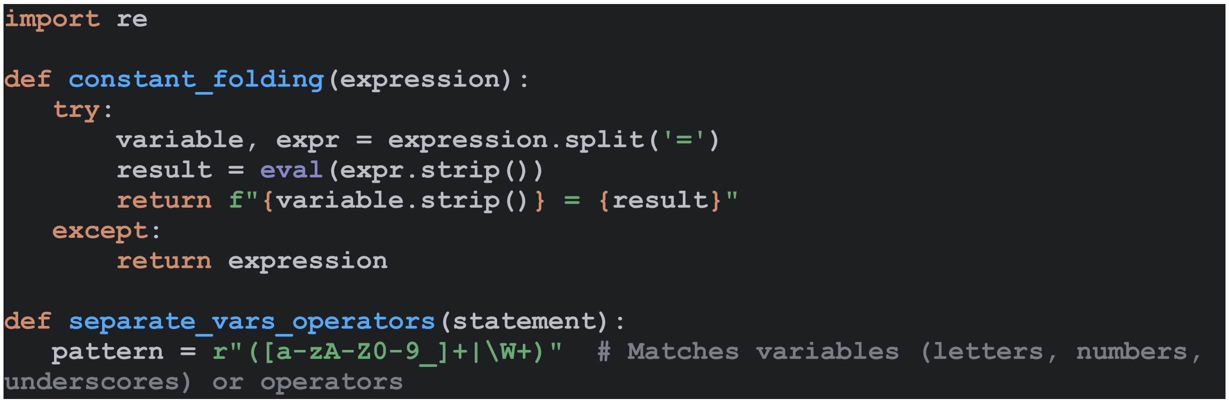


```
%{
#include <stdio.h>
#include <string.h>
%}

%%
if|else|while|for|do|switch    { printf("Keyword: %s\n", yytext); }
[ \t\n]                        ; // Ignore whitespace
.                              ; // Ignore other characters
%%

int main() {
    yylex();
    return 0;
}
```

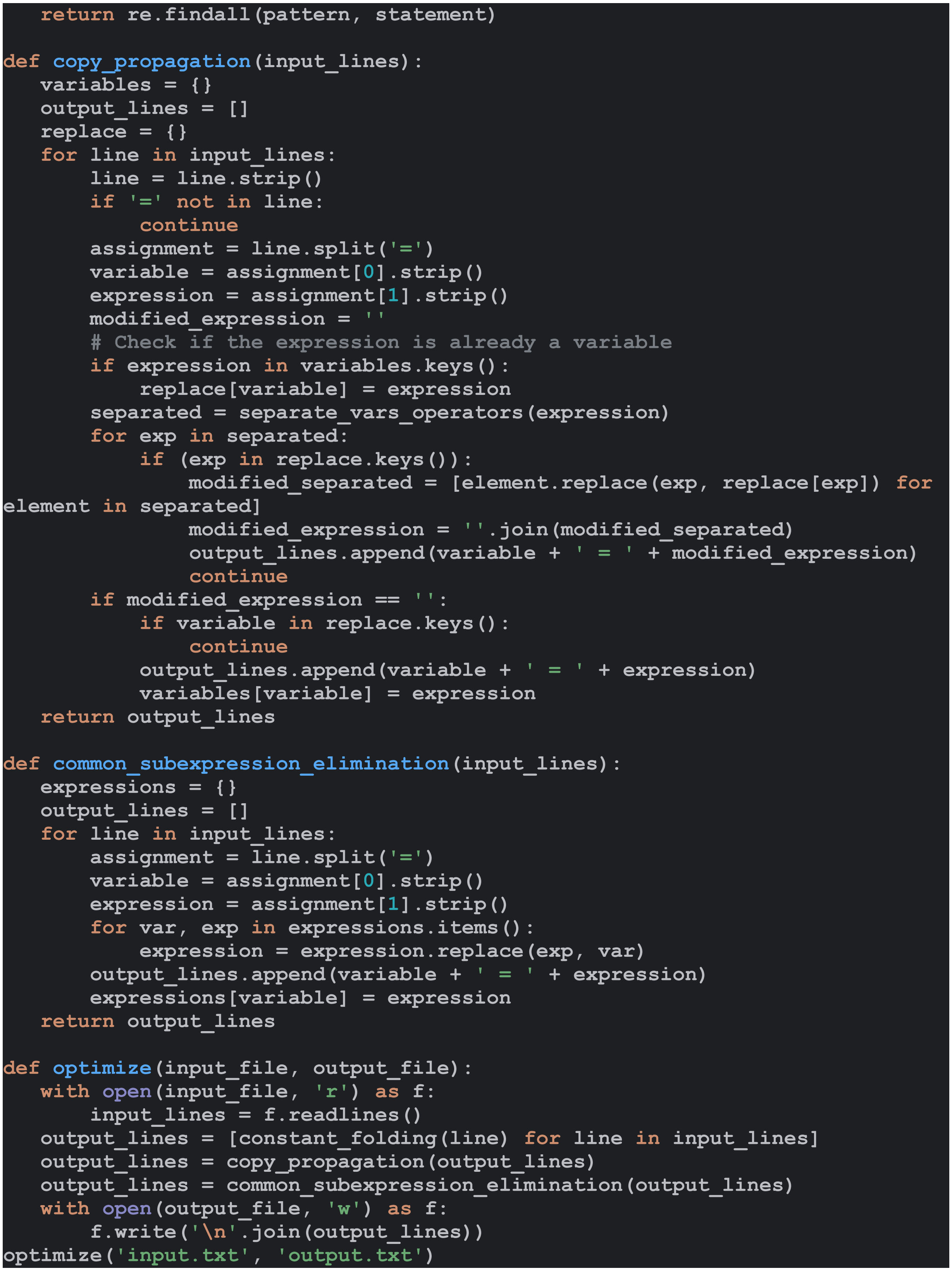
1. **Write a program to optimize the given three address code. [10 marks]**

**T1= 5*3+10 // Constant folding**
**T3=T1 //Copy propagation**
**T2=T1+T3**
**T5=4*T2 // common sub-expression elimination**
**T6=4*T2+100**

2. **Write a program to create your own ‘C’ library using macros to print greatest of two numbers [05 marks]**

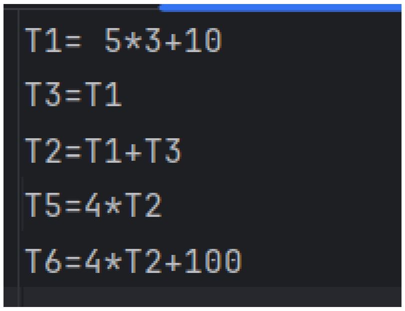


```
import re

def constant_folding(expression):
   try:
       variable, expr = expression.split('=')
       result = eval(expr.strip())
       return f"{variable.strip()} = {result}"
   except:
       return expression

def separate_vars_operators(statement):
   pattern = r"([a-zA-Z0-9_]+|\W+)"  # Matches variables (letters, numbers, underscores) or operators
```



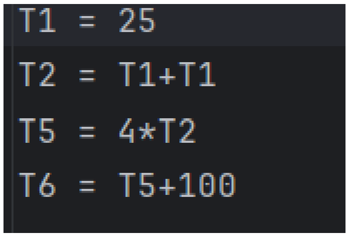
```
   return re.findall(pattern, statement)

def copy_propagation(input_lines):
   variables = {}
   output_lines = []
   replace = {}
   for line in input_lines:
       line = line.strip()
       if '=' not in line:
           continue
       assignment = line.split('=')
       variable = assignment[0].strip()
       expression = assignment[1].strip()
       modified_expression = ''
       # Check if the expression is already a variable
       if expression in variables.keys():
           replace[variable] = expression
       separated = separate_vars_operators(expression)
       for exp in separated:
           if (exp in replace.keys()):
               modified_separated = [element.replace(exp, replace[exp]) for
element in separated]
               modified_expression = ''.join(modified_separated)
               output_lines.append(variable + ' = ' + modified_expression)
               continue
       if modified_expression == '':
           if variable in replace.keys():
               continue
           output_lines.append(variable + ' = ' + expression)
           variables[variable] = expression
   return output_lines

def common_subexpression_elimination(input_lines):
   expressions = {}
   output_lines = []
   for line in input_lines:
       assignment = line.split('=')
       variable = assignment[0].strip()
       expression = assignment[1].strip()
       for var, exp in expressions.items():
           expression = expression.replace(exp, var)
       output_lines.append(variable + ' = ' + expression)
       expressions[variable] = expression
   return output_lines

def optimize(input_file, output_file):
   with open(input_file, 'r') as f:
       input_lines = f.readlines()
   output_lines = [constant_folding(line) for line in input_lines]
   output_lines = copy_propagation(output_lines)
   output_lines = common_subexpression_elimination(output_lines)
   with open(output_file, 'w') as f:
       f.write('\n'.join(output_lines))
optimize('input.txt', 'output.txt')
```

**INPUT.TXT:**

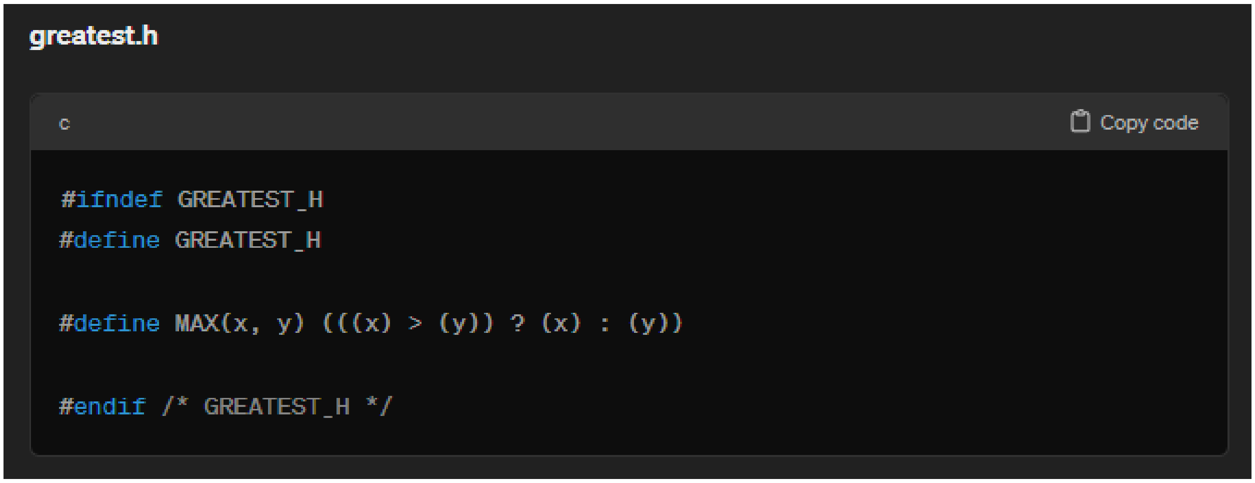


```
T1= 5*3+10
T3=T1
T2=T1+T3
T5=4*T2
T6=4*T2+100
```

**OUTPUT.TXT**

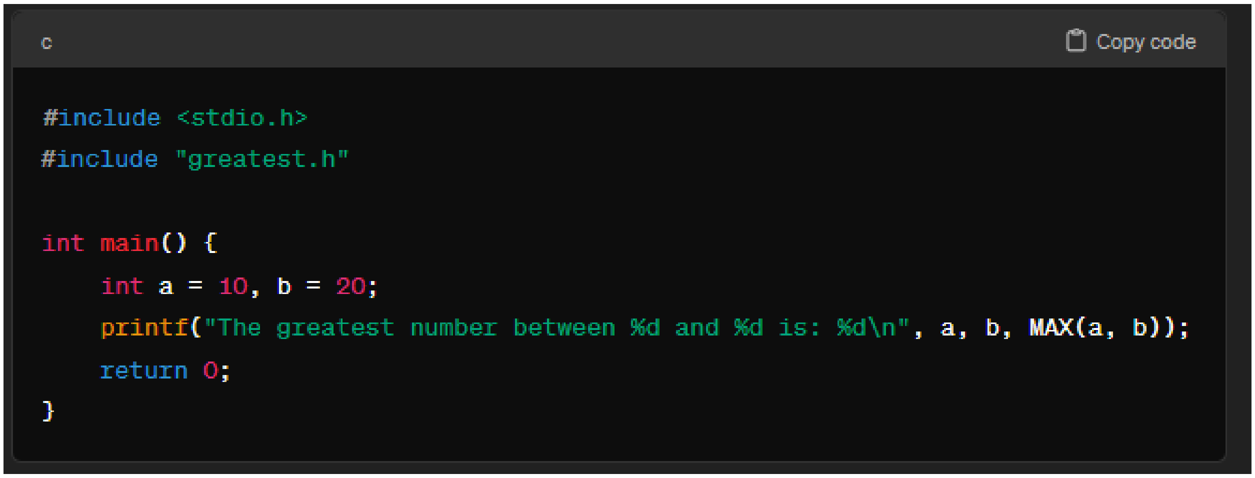


```
T1 = 25
T2 = T1+T1
T5 = 4*T2
T6 = T5+100
```



greatest.h

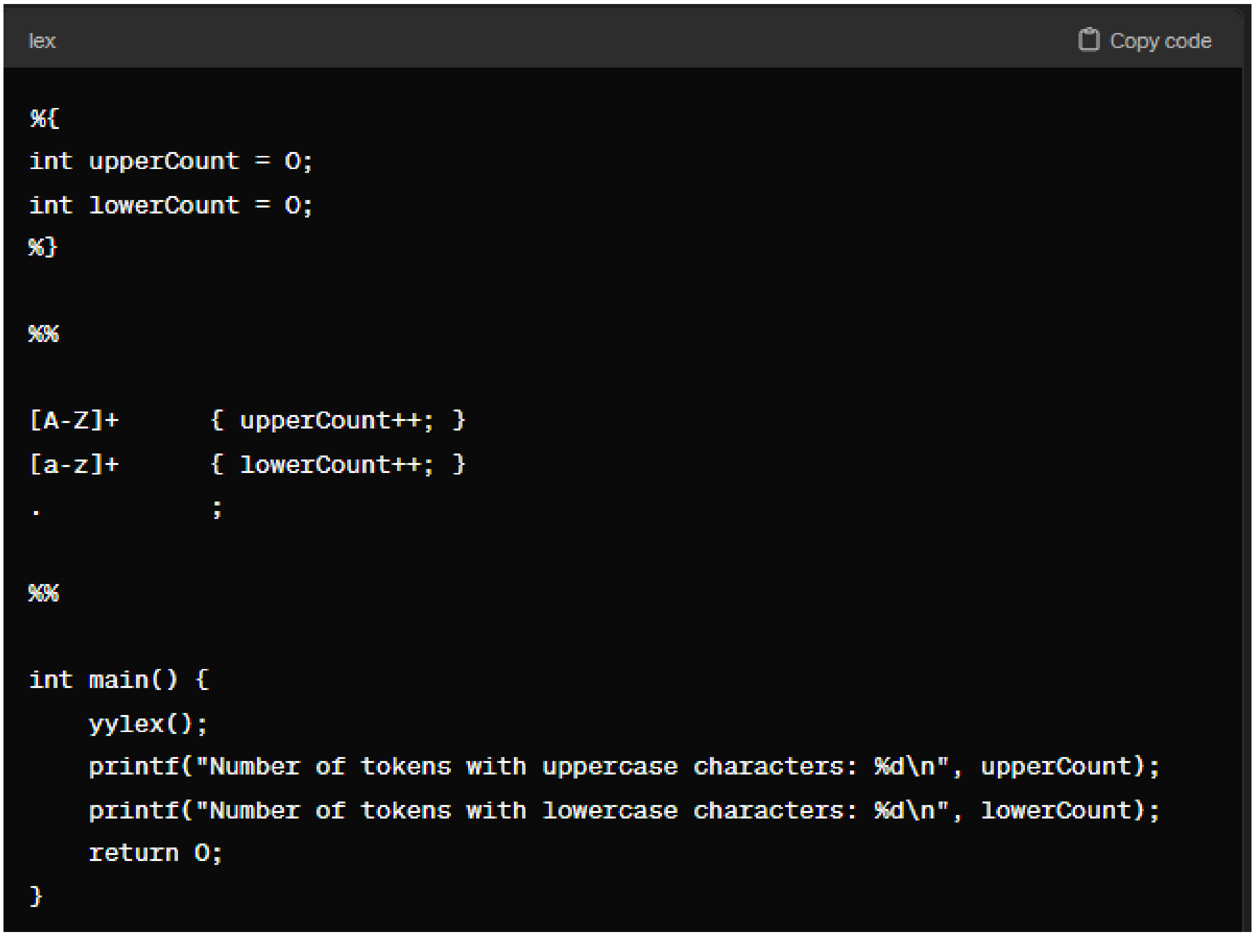
```c
#ifndef GREATEST_H
#define GREATEST_H

#define MAX(x, y) (((x) > (y)) ? (x) : (y))

#endif /* GREATEST_H */
```



```c
#include <stdio.h>
#include "greatest.h"

int main() {
    int a = 10, b = 20;
    printf("The greatest number between %d and %d is: %d\n", a, b, MAX(a, b));
    return 0;
}
```

---

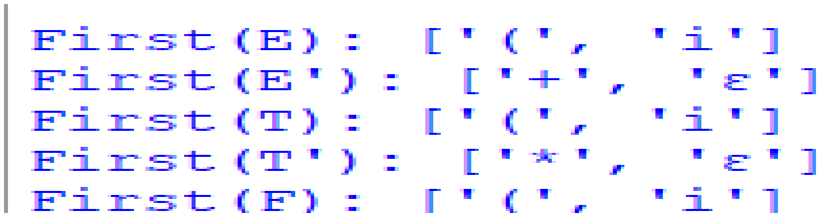
**1. Write a Lex program to count the number of tokens with uppercase and lowercase characters.**
**2. Write a program to print FIRST() for the following grammar [10 marks]**
**E->TE’**
**E’->+TE’/ε**
**T->FT’**
**T’->*FT’/ε**
**F->(ε)/id**



```lex
%{
int upperCount = 0;
int lowerCount = 0;
%}

%%

[A-Z]+      { upperCount++; }
[a-z]+      { lowerCount++; }
.           ;

%%

int main() {
    yylex();
    printf("Number of tokens with uppercase characters: %d\n", upperCount);
    printf("Number of tokens with lowercase characters: %d\n", lowerCount);
    return 0;
}
```

```
gram = {
    "E":["TE'"],
    "E'":["+TE'", "ε"],
    "T":["FT'"],
    "T'":["*FT'", "ε"],
    "F":["(ε)", "id"]
}


def removeDirectLR(gramA, A):
    """gramA is dictonary"""
    temp = gramA[A]
    tempCr = []
    tempInCr = []
    for i in temp:
        if i[0] == A:
            #tempInCr.append(i[1:])
            tempInCr.append(i[1:]+[A+"'"])
        else:
            #tempCr.append(i)
            tempCr.append(i+[A+"'"])
    tempInCr.append(["e"])
    gramA[A] = tempCr
    gramA[A+"'"] = tempInCr
    return gramA


def checkForIndirect(gramA, a, ai):
    if ai not in gramA:
        return False
    if a == ai:
        return True
    for i in gramA[ai]:
        if i[0] == ai:
            return False
        if i[0] in gramA:
            return checkForIndirect(gramA, a, i[0])
    return False

def rep(gramA, A):
    temp = gramA[A]
    newTemp = []
    for i in temp:
        if checkForIndirect(gramA, A, i[0]):
            t = []
            for k in gramA[i[0]]:
                t=[]
                t+=k
                t+=i[1:]
                newTemp.append(t)

        else:
            newTemp.append(i)
    gramA[A] = newTemp
    return gramA

def rem(gram):
    c = 1
    conv = {}
    gramA = {}
    revconv = {}
    for j in gram:
        conv[j] = "A"+str(c)
        gramA["A"+str(c)] = []
        c+=1

    for i in gram:
        for j in gram[i]:
            temp = []
            for k in j:
                if k in conv:
                    temp.append(conv[k])
                else:
                    temp.append(k)
            gramA[conv[i]].append(temp)


    #print(gramA)
    for i in range(c-1,0,-1):
        ai = "A"+str(i)
        for j in range(0,i):
            aj = gramA[ai][0][0]
            if ai!=aj :
                if aj in gramA and checkForIndirect(gramA,ai,aj):
                    gramA = rep(gramA, ai)

    for i in range(1,c):
        ai = "A"+str(i)
        for j in gramA[ai]:
            if ai==j[0]:
                gramA = removeDirectLR(gramA, ai)
                break

    op = {}
    for i in gramA:
        a = str(i)
        for j in conv:
            a = a.replace(conv[j],j)
        revconv[i] = a

    for i in gramA:
        l = []
        for j in gramA[i]:
            k = []
            for m in j:
                if m in revconv:
                    k.append(m.replace(m,revconv[m]))
                else:
                    k.append(m)
            l.append(k)
```

```
                        op[revconv[i]] = 1

            return op

result = rem(gram)


def first(gram, term):
            a = []
            if term not in gram:
                        return [term]
            for i in gram[term]:
                        if i[0] not in gram:
                                    a.append(i[0])
                        elif i[0] in gram:
                                    a += first(gram, i[0])
            return a

firsts = {}
for i in result:
            firsts[i] = first(result,i)
            print(f'First({i}):',firsts[i])
```



```
First(E): ['(', 'i']
First(E'): ['+', 'ε']
First(T): ['(', 'i']
First(T'): ['*', 'ε']
First(F): ['(', 'i']
```

**1. Write a program to create your own ‘C’ library using macros for conversions. [05 marks] (binary ⬄ decimal, binary ⬄ hexadecimal)**

**2. Write a program to print FOLLOW() for the following grammar [10 marks]**

**E->TE’**

**E’->+TE’/ε**

**T->FT’**

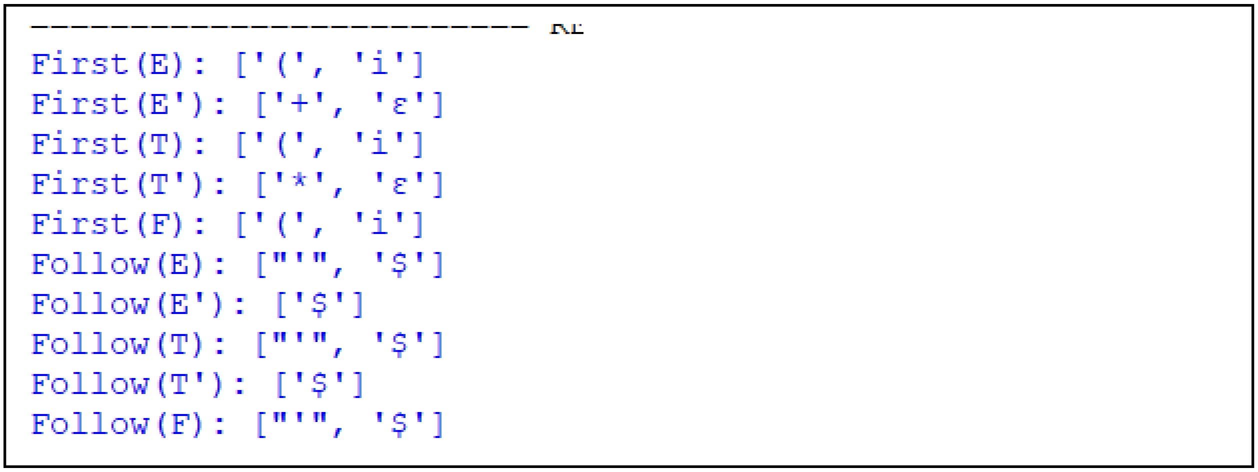
**T’->*FT’/ε**

**F->(ε)/id**

```
#include <stdio.h>
#include <math.h>
```

```
// Macro for converting binary to decimal
#define BINARY_TO_DECIMAL(binary) ({                    \
    int decimal = 0, i = 0;                             \
    while (binary) {                                    \
        decimal += (binary % 10) * pow(2, i++);         \
        binary /= 10;                                   \
    }                                                   \
    decimal;                                            \
})

// Macro for converting decimal to binary
#define DECIMAL_TO_BINARY(decimal) ({                   \
    int binary = 0, i = 1;                              \
    while (decimal) {                                   \
        binary += (decimal % 2) * i;                    \
        i *= 10;                                        \
        decimal /= 2;                                   \
    }                                                   \
    binary;                                             \
})

// Macro for converting binary to hexadecimal
#define BINARY_TO_HEXADECIMAL(binary) ({                \
    int hexadecimal = 0, i = 0;                         \
    while (binary) {                                    \
        hexadecimal = (hexadecimal * 16) + (binary % 10000); \
        binary /= 10000;                                \
        i++;                                            \
    }                                                   \
    hexadecimal;                                        \
})

// Macro for converting hexadecimal to binary
#define HEXADECIMAL_TO_BINARY(hexadecimal) ({           \
    int binary = 0, i = 1;                              \
    while (hexadecimal) {                               \
        binary += (hexadecimal % 16) * i;               \
        i *= 10000;                                     \
        hexadecimal /= 16;                              \
    }                                                   \
    binary;                                             \
})

int main() {
    int binary = 10101010;
    int decimal = BINARY_TO_DECIMAL(binary);
    int hexadecimal = BINARY_TO_HEXADECIMAL(binary);

    printf("Binary: %d\n", binary);
    printf("Decimal: %d\n", decimal);
    printf("Hexadecimal: %X\n", hexadecimal);

    binary = DECIMAL_TO_BINARY(decimal);
    hexadecimal = DECIMAL_TO_BINARY(hexadecimal);

    printf("Binary: %d\n", binary);
    printf("Hexadecimal: %X\n", hexadecimal);

    return 0;
}
```

```
# #example for direct left recursion
# gram = {"A":["Aa","Ab","c","d"]
# }
#example for indirect left recursion
gram = {
        "E":["TE'"],
        "E'":["+TE'", "ε"],
        "T":["FT'"],
        "T'":["*FT'", "ε"],
        "F":["(ε)", "id"]
}

def removeDirectLR(gramA, A):
        """gramA is dictonary"""
        temp = gramA[A]
        tempCr = []
        tempInCr = []
        for i in temp:
                if i[0] == A:
                        #tempInCr.append(i[1:])
                        tempInCr.append(i[1:]+[A+"'"])
                else:
                        #tempCr.append(i)
                        tempCr.append(i+[A+"'"])
        tempInCr.append(["e"])
        gramA[A] = tempCr
        gramA[A+"'"] = tempInCr
        return gramA


def checkForIndirect(gramA, a, ai):
    if ai not in gramA:
        return False
    if a == ai:
        return True
    for i in gramA[ai]:
        if i[0] == ai:
            return False
        if i[0] in gramA:
            return checkForIndirect(gramA, a, i[0])
    return False

def rep(gramA, A):
    temp = gramA[A]
    newTemp = []
    for i in temp:
        if checkForIndirect(gramA, A, i[0]):
            t = []
            for k in gramA[i[0]]:
                t=[]
                t+=k
                t+=i[1:]
                newTemp.append(t)

        else:
            newTemp.append(i)
    gramA[A] = newTemp
    return gramA

def rem(gram):
    c = 1
    conv = {}
    gramA = {}
    revconv = {}
    for j in gram:
        conv[j] = "A"+str(c)
        gramA["A"+str(c)] = []
        c+=1

    for i in gram:
        for j in gram[i]:
            temp = []
            for k in j:
                if k in conv:
                    temp.append(conv[k])
                else:
                    temp.append(k)
            gramA[conv[i]].append(temp)


    #print(gramA)
    for i in range(c-1,0,-1):
        ai = "A"+str(i)
        for j in range(0,i):
            aj = gramA[ai][0][0]
            if ai!=aj :
                if aj in gramA and checkForIndirect(gramA,ai,aj):
                    gramA = rep(gramA, ai)

    for i in range(1,c):
        ai = "A"+str(i)
        for j in gramA[ai]:
            if ai==j[0]:
                gramA = removeDirectLR(gramA, ai)
                break

    op = {}
    for i in gramA:
        a = str(i)
        for j in conv:
            a = a.replace(conv[j],j)
        revconv[i] = a

    for i in gramA:
        l = []
        for j in gramA[i]:
            k = []
            for m in j:
                if m in revconv:
                    k.append(m.replace(m,revconv[m]))
                else:
                    k.append(m)
            l.append(k)
        op[revconv[i]] = l

    return op

result = rem(gram)


def first(gram, term):
    a = []
    if term not in gram:
        return [term]
    for i in gram[term]:
        if i[0] not in gram:
            a.append(i[0])
        elif i[0] in gram:
            a += first(gram, i[0])
    return a

firsts = {}
for i in result:
    firsts[i] = first(result,i)
    print(f'First({i}):',firsts[i])
#   temp = follow(result,i,i)
#   temp = list(set(temp))
#   temp = [x if x != "e" else "$" for x in temp]
#   print(f'Follow({i}):',temp)

def follow(gram, term):
    a = []
    for rule in gram:
        for i in gram[rule]:
            if term in i:
                temp = i
                indx = i.index(term)
                if indx+1!=len(i):
                    if i[-1] in firsts:
                        a+=firsts[i[-1]]
                    else:
                        a+=[i[-1]]
                else:
                    a+=["e"]
                if rule != term and "e" in a:
                    a+= follow(gram,rule)
    return a

follows = {}
for i in result:
    follows[i] = list(set(follow(result,i)))
    if "e" in follows[i]:
        follows[i].pop(follows[i].index("e"))
    follows[i]+=["$"]
    print(f'Follow({i}):',follows[i])
```



```
--------------------------
First(E): ['(', 'i']
First(E'): ['+', 'ε']
First(T): ['(', 'i']
First(T'): ['*', 'ε']
First(F): ['(', 'i']
Follow(E): ["'", '$']
Follow(E'): ['$']
Follow(T): ["'", '$']
Follow(T'): ['$']
Follow(F): ["'", '$']
```

—------------------------------------------------------------------------------------------------------------------------

**1. Write a program to remove Left Factoring from the given grammar [05 marks]**

**A —-> bE+acF | bE+F**

**2. Consider the following Three address code and display Quadruples & Triples [10 marks]**

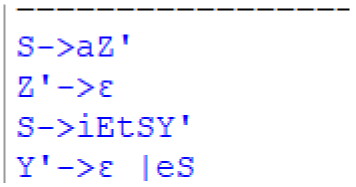
**f=c+d**

**e=a-f**

**g=b*e**

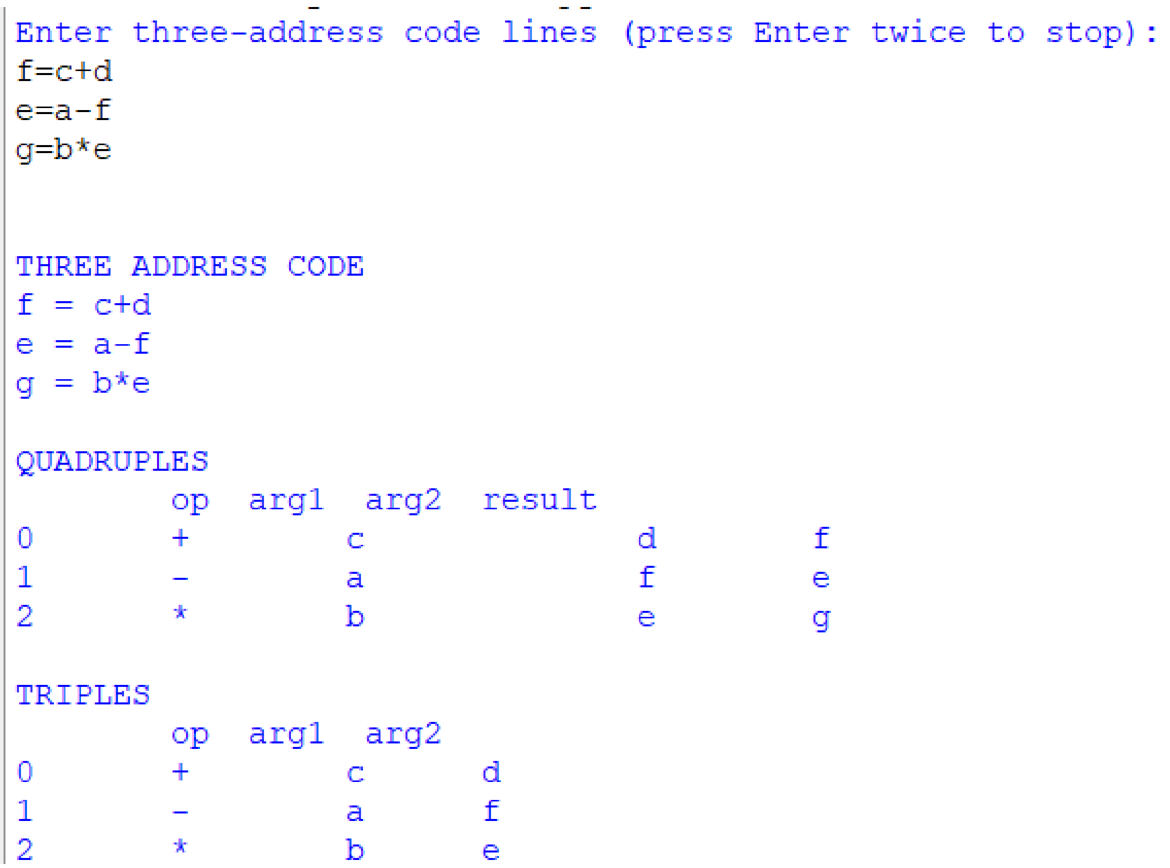
```
from itertools import takewhile

s= "S->iEtS|iEtSeS|a"

def groupby(ls):
    d = {}
    ls = [ y[0] for y in rules ]
    initial = list(set(ls))
    for y in initial:
        for i in rules:
            if i.startswith(y):
                if y not in d:
                    d[y] = []
                d[y].append(i)
    return d

def prefix(x):
    return len(set(x)) == 1


starting=""
rules=[]
common=[]
alphabetset=["A'","B'","C'","D'","E'","F'","G'","H'","I'","J'","K'","L'","M'","N'","O'","P'","Q'","R'","S'","T'","U'","V'","W'","X'","Y'","Z'"]
```

```
s = s.replace(" ", "").replace("	", "").replace("\n", "")

while(True):
    rules=[]
    common=[]
    split=s.split("->")
    starting=split[0]
    for i in split[1].split("|"):
        rules.append(i)

#logic for taking commons out
    for k, l in groupby(rules).items():
        r = [l[0] for l in takewhile(prefix, zip(*l))]
        common.append(''.join(r))
#end of taking commons
    for i in common:
        newalphabet=alphabetset.pop()
        print(starting+"->"+i+newalphabet)
        index=[]
        for k in rules:
            if(k.startswith(i)):
                index.append(k)
        print(newalphabet+"->",end="")
        for j in index[:-1]:
            stringtoprint=j.replace(i,"", 1)+"|"
            if stringtoprint=="|":
                print("\u03B5","|",end="")
            else:
                print(j.replace(i,"", 1)+"|",end="")
        stringtoprint=index[-1].replace(i,"", 1)+"|"
        if stringtoprint=="|":
            print("\u03B5","",end="")
        else:
            print(index[-1].replace(i,"", 1)+"",end="")
        print("")
    break
```



```
------------------
S->aZ'
Z'->ε
S->iEtSY'
Y'->ε |eS
```

```
def convert_to_quadruples(address_code_dict):
    quadruple_list = dict()
    list_no = 0
    for key, value in address_code_dict.items():
        if len(value) <= 2:
            i = value.find('=')
            quadruple_list[list_no] = {"op": "=", "arg1": value, "arg2": "   ", "result": key}
            list_no += 1
        else:
            for item in ['-', '+', '*', '/']:
                if item in value:
                    i = value.find(item)
                    quadruple_list[list_no] = {"op": value[i], "arg1": value[:i], "arg2": value[i + 1:],
                                               "result": key}
                else:
                    continue
            list_no += 1
    return quadruple_list

def convert_to_triples(quadruples_dict):
    triples_dict = dict()
    dict_no = 0
    for key1, value1 in quadruples_dict.items():
        if value1['op'] == "=":
            triples_dict[dict_no] = {'op': value1['op'], 'arg1': value1['result'],
                                     'arg2': f"({int(value1['arg1'][-1]) - 1})" if "t" in value1['arg1'] else
value1['arg1']
                                     }
        else:
            triples_dict[dict_no] = {'op': value1['op'], 'arg1': f"({int(value1['arg1'][-1]) - 1})" if "t"
in value1['arg1'] else value1['arg1'],
                                     'arg2': f"({int(value1['arg2'][-1]) - 1})" if "t" in value1['arg2'] else
value1['arg2']
                                     }
        dict_no += 1
    return triples_dict

three_address_code = {}
print("Enter three-address code lines (press Enter twice to stop):")
while True:
    line = input()
    if not line:
        break
    parts = line.split('=')
    variable = parts[0].strip()
    expression = ''.join(parts[1:]).strip()
    three_address_code[variable] = expression

print("\nTHREE ADDRESS CODE")
for key, value in three_address_code.items():
    print(f'{key} = {value}')

quadruples = convert_to_quadruples(three_address_code)
print("\nQUADRUPLES")
print("\top  arg1  arg2  result")
for key, value in quadruples.items():
    print(f"{key}\t{value['op']}\t {value['arg1']}\t\t{value['arg2']}\t {value['result']}")

triples = convert_to_triples(quadruples)
print("\nTRIPLES")
print("\top  arg1  arg2")
for key, value in triples.items():
    print(f"{key}\t{value['op']}\t {value['arg1']}\t{value['arg2']}")
```



```
Enter three-address code lines (press Enter twice to stop):
f=c+d
e=a-f
g=b*e


THREE ADDRESS CODE
f = c+d
e = a-f
g = b*e

QUADRUPLES
        op  arg1  arg2  result
0       +        c              d        f
1       -        a              f        e
2       *        b              e        g

TRIPLES
        op  arg1  arg2
0       +        c      d
1       -        a      f
2       *        b      e
```

**1. WAP to implement Two Pass Macro Processor for the following [15 marks]**

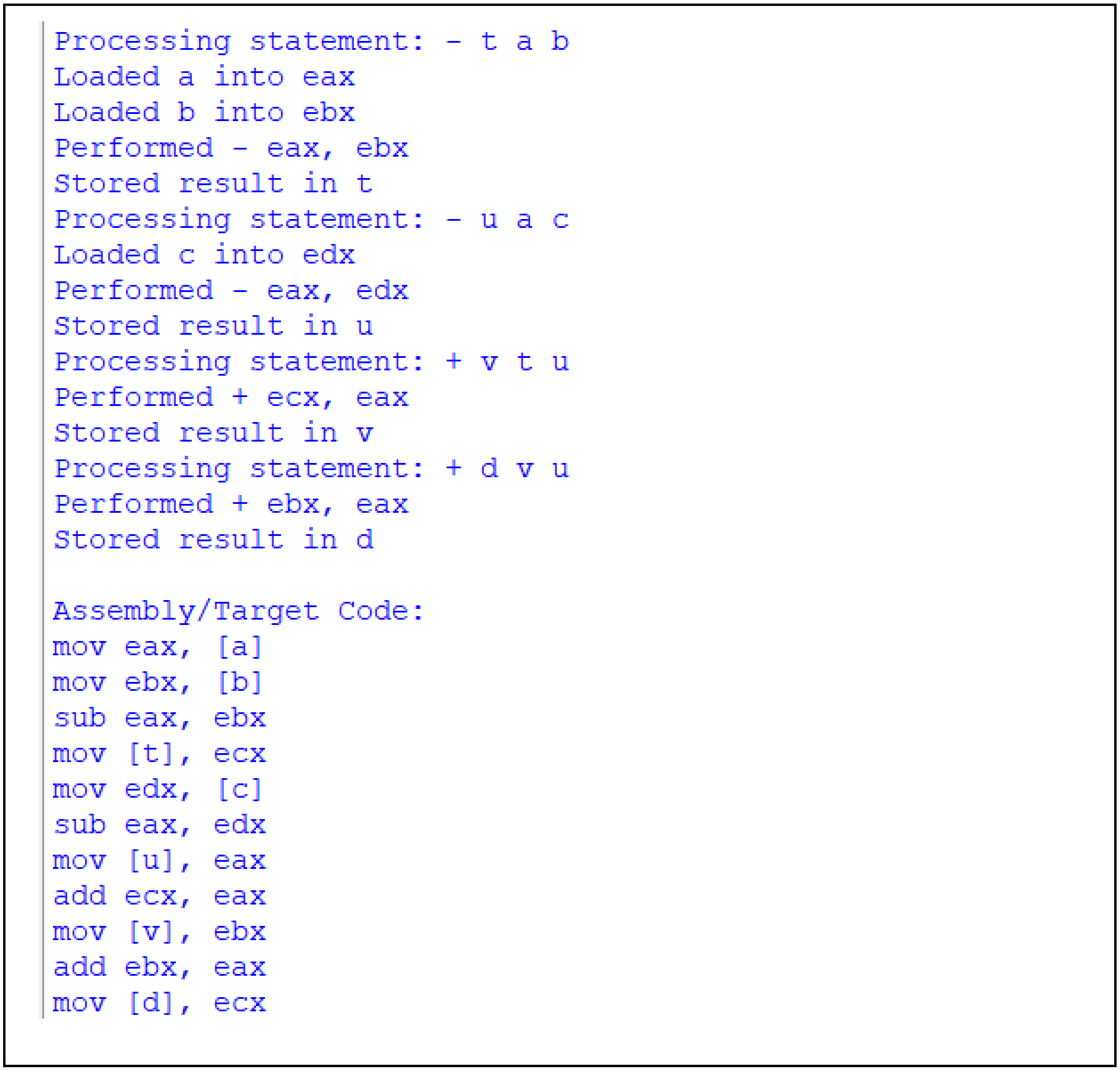
**MACRO**
**ADD &ARG1, &ARG2**
**L 1, &ARG1**
**A 1, &ARG2**
**MEND**
**MACRO**
**SUB &ARG3, &ARG4**
**L 1, &ARG3**
**S 1, &ARG4**
**MEND**
**ADD DATA1, DATA2**
**SUB DATA1, DATA2**
**DATA1 DC F’9’**
**DATA2 DC F’5’**
**END**

---

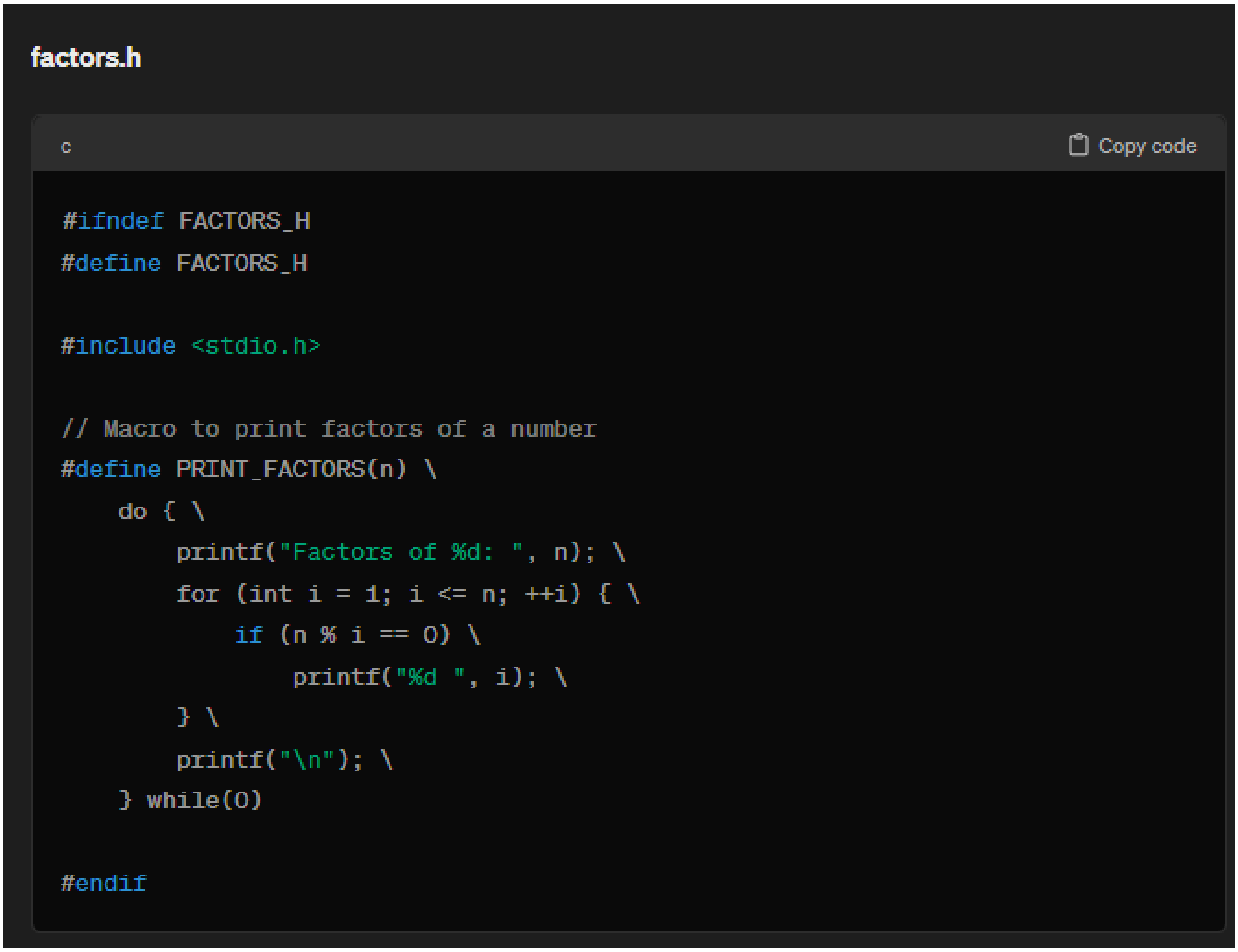
**1. Write a program to display assembly / target code for the following 3AC statements [10 marks]**
**t = a - b**
**u = a - c**
**v = t + u**
**d = v + u**

**2. Write a program to create your own ‘C’ library using macros to print factors of any number**

```
def generate_assembly_code(three_address_code):
    assembly_code = []
    registers = ['eax', 'ebx', 'ecx', 'edx']
    register_map = {}
    register_counter = 0

    for statement in three_address_code:
        print(f"Processing statement: {statement}")
        operation, result, operand1, operand2 = statement.split()

        # Load operands into registers
        if operand1 not in register_map:
            register_map[operand1] = registers[register_counter]
            assembly_code.append(f"mov {registers[register_counter]}, [{operand1}]")
            print(f"Loaded {operand1} into {registers[register_counter]}")
            register_counter = (register_counter + 1) % len(registers)
        if operand2 not in register_map:
            register_map[operand2] = registers[register_counter]
            assembly_code.append(f"mov {registers[register_counter]}, [{operand2}]")
            print(f"Loaded {operand2} into {registers[register_counter]}")
            register_counter = (register_counter + 1) % len(registers)

        # Perform operation
        if operation == '-':
            assembly_code.append(f"sub {register_map[operand1]}, {register_map[operand2]}")
            print(f"Performed {operation} {register_map[operand1]}, {register_map[operand2]}")
        elif operation == '+':
            assembly_code.append(f"add {register_map[operand1]}, {register_map[operand2]}")
            print(f"Performed {operation} {register_map[operand1]}, {register_map[operand2]}")

        # Store result
        register_map[result] = registers[register_counter]
        assembly_code.append(f"mov [{result}], {registers[register_counter]}")
        print(f"Stored result in {result}")
        register_counter = (register_counter + 1) % len(registers)

    return assembly_code

# Example 3AC statements
three_address_code = [
    "- t a b",
    "- u a c",
    "+ v t u",
    "+ d v u"
]

# Generate assembly code
assembly_code = generate_assembly_code(three_address_code)

# Print assembly code
print("\nAssembly/Target Code:")
for line in assembly_code:
    print(line)
```

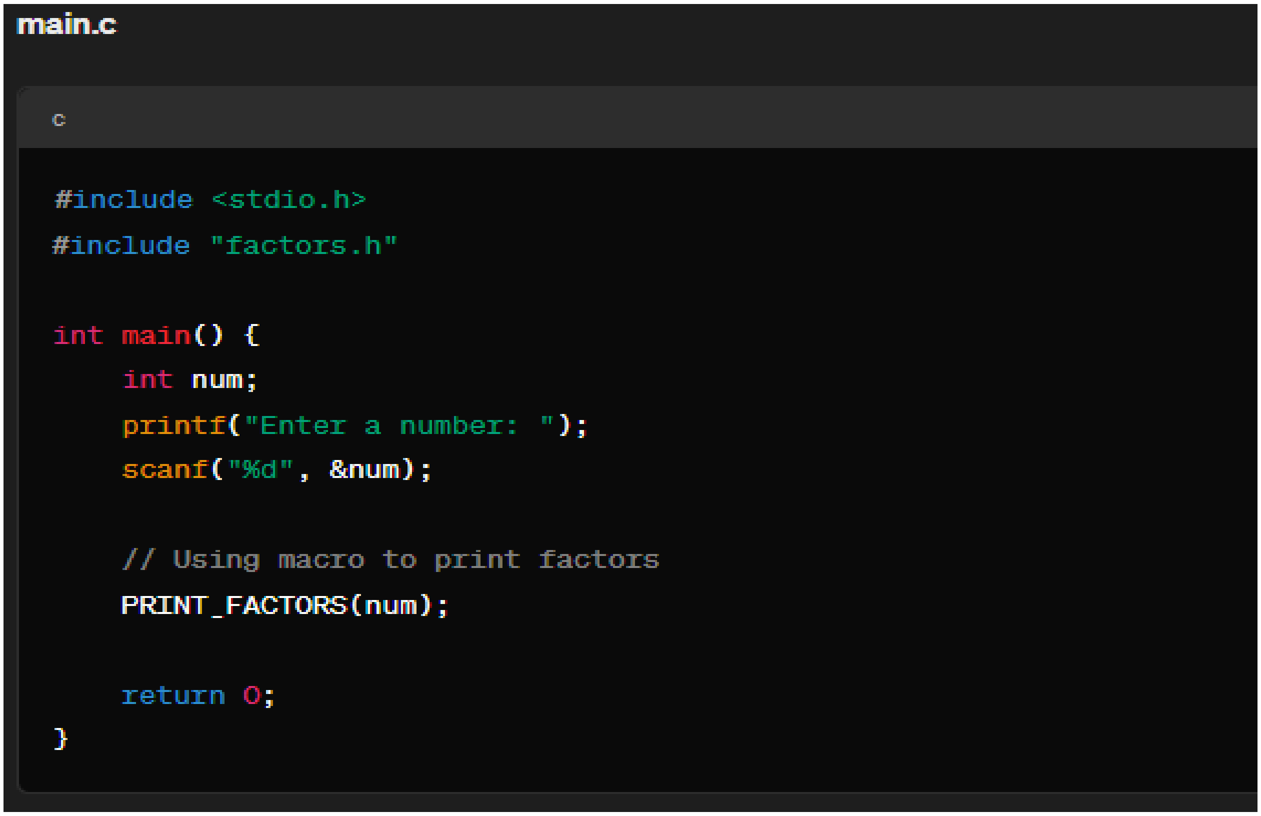


```
Processing statement: - t a b
Loaded a into eax
Loaded b into ebx
Performed - eax, ebx
Stored result in t
Processing statement: - u a c
Loaded c into edx
Performed - eax, edx
Stored result in u
Processing statement: + v t u
Performed + ecx, eax
Stored result in v
Processing statement: + d v u
Performed + ebx, eax
Stored result in d

Assembly/Target Code:
mov eax, [a]
mov ebx, [b]
sub eax, ebx
mov [t], ecx
mov edx, [c]
sub eax, edx
mov [u], eax
add ecx, eax
mov [v], ebx
add ebx, eax
mov [d], ecx
```



factors.h

```c
#ifndef FACTORS_H
#define FACTORS_H

#include <stdio.h>

// Macro to print factors of a number
#define PRINT_FACTORS(n) \
    do { \
        printf("Factors of %d: ", n); \
        for (int i = 1; i <= n; ++i) { \
            if (n % i == 0) \
                printf("%d ", i); \
        } \
        printf("\n"); \
    } while(0)

#endif
```



main.c

c

```c
#include <stdio.h>
#include "factors.h"

int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);

    // Using macro to print factors
    PRINT_FACTORS(num);

    return 0;
}
```

—---------------------------------------------------------------------------------------------------------------------------

**1. Write a program to optimize the given three address code. [10 marks] DONE**

**T1= 5*3+10 // Constant folding**
**T3=T1 //Copy propagation**
**T2=T1+T3**
**T5=4*T2 // common sub-expression elimination**
**T6=4*T2+100**

**2. Write a program to create your own ‘C’ library using macros to print greatest of two Numbers DONE**