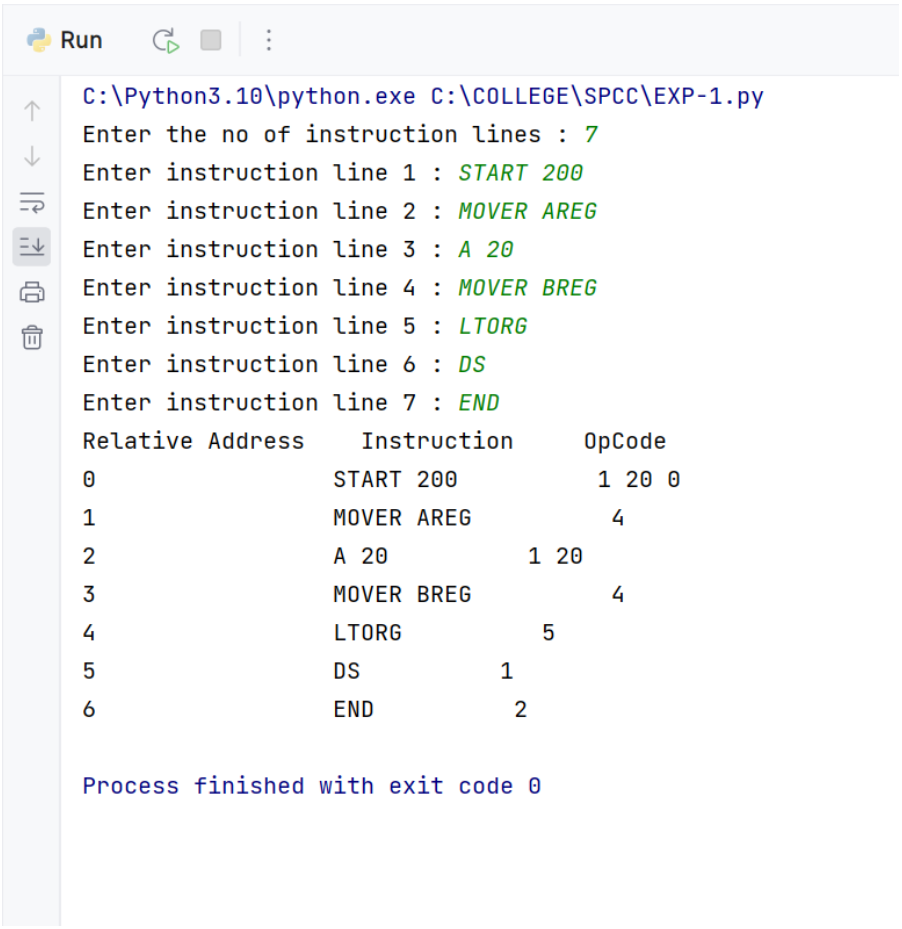
**// Expt. 1 Single Pass Assembler Implementation**

```
from sys import exit

motOpCode = {
        "STOP": 0,
        "ADD": 1,
        "SUB": 2,
        "MULT": 3,
        "MOVER": 4,
        "MOVEM": 5,
        "COMP": 6,
        "BC": 7,
        "DIV": 8,
        "READ": 9,
        "PRINT": 10,
        "START": 1,
        "END": 2,
        "EQU": 3,
        "ORIGIN": 4,
        "LTORG": 5,
        "DS": 1,
        "DC": 2,
        "AREG": 1,
        "BREG": 2,
        "CREG": 3,
        "DREG": 4,
        "A": 1,
        "B": 2,
}

motSize = {
        "STOP": 1,
        "ADD": 1,
        "SUB": 1,
        "MULT": 1,
        "MOVER": 1,
        "MOVEM": 1,
        "COMP": 1,
        "BC": 1,
        "DIV": 1,
        "READ": 1,
        "PRINT": 1,
        "START": 1,
        "END": 1,
        "EQU": 1,
        "ORIGIN": 1,
        "LTORG": 1,
        "DS": 1,
        "DC": 1,
        "AREG": 1,
        "BREG": 1,
        "CREG": 1,
        "DREG": 1,
        "A": 1,
        "B": 1,
}

l = []
relativeAddress = []
machineCode = []
RA = 0
current = 0
count = 0
n = int(input("Enter the no of instruction lines : "))for i in range(n):
```

```
        instructions = input("Enter instruction line {} : ".format(i + 1))l.append(instructions)
l = [x.upper() for x in l]   # Converting all the instructions to upper case
for i in range(n):x = l[i]
        if " " in x:
                s1 = ''.join(x) a, b =
                s1.split()
                if a in motOpCode:

                        value = motOpCode.get(a)size =
                        motSize.get(a) previous = size
                        RA += current current = previous
                        relativeAddress.append(RA) if
                        b.isalpha() is True:
                                machineCode.append(str(value)) else:
                                temp = list(b)
                                for i in range(len(temp)):if count == 2:
                                                temp.insert(i, ' ')count = 0
                                        else:
                          count = count + 1s = ''.join(temp)
                                machineCode.append(str(value) + " " + s)
                   else:

                        print("Instruction is not in Op Code Table.")exit(0)   # EXIT if
                        Mnemonics is not in MOT
        else:

                if x in motOpCode:
                        value = motOpCode.get(x)size =
                        motSize.get(x) previous = size
                        RA += current current =
                        previous
                        relativeAddress.append(RA)
                        machineCode.append(value)
                else:
                        print("Instruction is not in Op Code Table.")exit(0)

print("Relative Address Instruction OpCode")
for i in range(n):
        print("{}  {}          {}".format(relativeAddress[i], l[i], machineCode[i]))
```

**Output :**


Run
C:\Python3.10\python.exe C:\COLLEGE\SPCC\EXP-1.py
Enter the no of instruction lines : 7
Enter instruction line 1 : START 200
Enter instruction line 2 : MOVER AREG
Enter instruction line 3 : A 20
Enter instruction line 4 : MOVER BREG
Enter instruction line 5 : LTORG
Enter instruction line 6 : DS
Enter instruction line 7 : END
Relative Address Instruction OpCode
0 START 200 1 20 0
1 MOVER AREG 4
2 A 20 1 20
3 MOVER BREG 4
4 LTORG 5
5 DS 1
6 END 2
Process finished with exit code 0

**// Expt. 2 Two Pass Assembler Implementation**

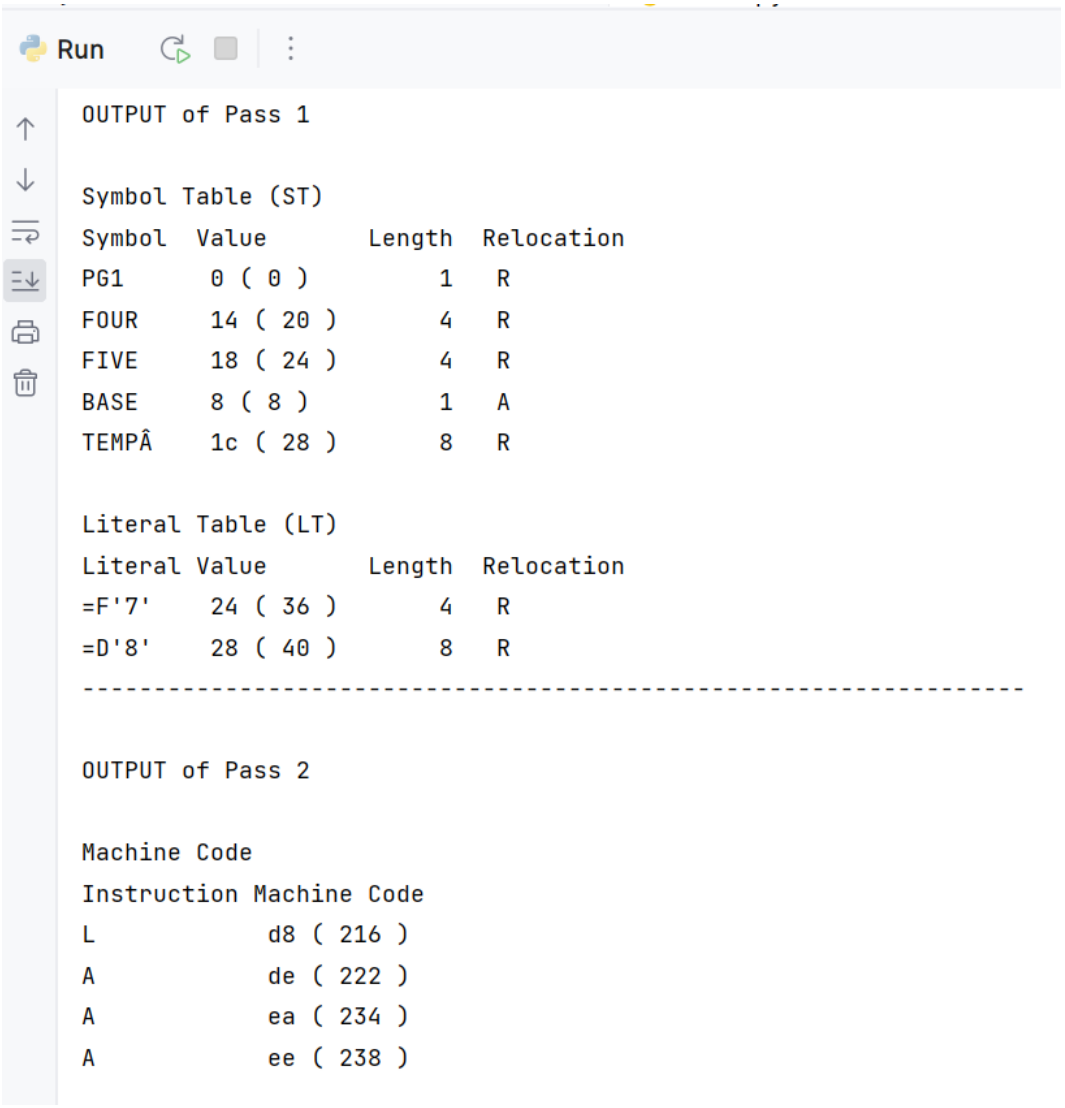
```
with open('EXP.txt') as t:data = []
        for line in t.readlines(): data.append(line.split())
# print(data)

symbols = []value
= 0


def contains(string): string =
        list(string)for i in string:
                if i == "F":
                        return 4 elif i
                == "D":
                        return 8
        return 1


def contains_literal(string):string =
        list(string)
        if "=" in string:return
                True


for j, i in enumerate(data):

        if len(i) == 2 and i[0].lower() == "using":value = 0
                continue

        if len(i) == 2:value +=
                4
        if j == 1:
                value = 0
                continue

        if len(i) == 3:
                length = contains(i[2]) if i[1].lower()
                == "eqv":
                        symbols.append([i[0], int(i[2]), length, 'A'])base = int(i[2])
                else:
                        symbols.append([i[0], value, length, "R"])if (length != 4):
                                value += lengthelse:
                                value += 4

print("OUTPUT of Pass 1\n\nSymbol Table (ST)")
print("Symbol\tValue\t\tLength\tRelocation") for i in symbols:
        print(i[0], "\t", hex(i[1])[2:], '(', i[1], ')', "\t\t", i[2], "\t",i[3])

literals = []lvalue = value

for j, i in enumerate(data):if len(i) == 2:
                if contains_literal(i[1]):
                        a = list((i[1].split('='))[1])length =
                        contains(a[0])
                        literals.append([(i[1].split(','))[1], lvalue, length, "R"])if (length != 4):
                                lvalue += lengthelse:
                                lvalue += 4
                        # print(a)
print("\nLiteral Table (LT)")
print("Literal\tValue\t\tLength\tRelocation") for i in literals:
        print(i[0], "\t", hex(i[1])[2:], '(', i[1], ')', "\t\t", i[2], "\t",i[3])
```

```
main = symbols + literals
mot = [['L', int('58', 16)], ['ST', int('50', 16)], ['A', int('5A', 16)]]


def getOpHex(op):for i in
        mot:
                if i[0] == op:return i[1]
        return


def getOpOperand(op):

        for i in main:
                if i[0] == op:return i[1]
        return


print("      ----------------------------------------------------------------------------------------------------------------------------------------------------------------")
print("\nOUTPUT of Pass 2\n\nMachine Code")
print("Instruction\tMachine Code")

one = 100
for i, j in enumerate(data[2:], 1):if len(j) == 2:
                final = getOpHex(j[0]) + getOpOperand(j[1].split(',')[1]) + one +
base
                print(j[0], '\t\t\t', hex(final)[2:], '(', final, ')')

bases = []
for i in range(0, 16):if (i == base):
                bases.append(['Y', 000000])else:
                bases.append(['N', None])
print("\nBase Table (BT)")
print("Base Availability Indicator Contents")for j, i in
enumerate(bases):
        if (i[1] == 0):
                print(j, "\t", i[0], "\t\t\t\t\t\t", str(i[1]) * 6)else:
                print(j, "\t", i[0])
```

**OUTPUT:**

**INPUT**

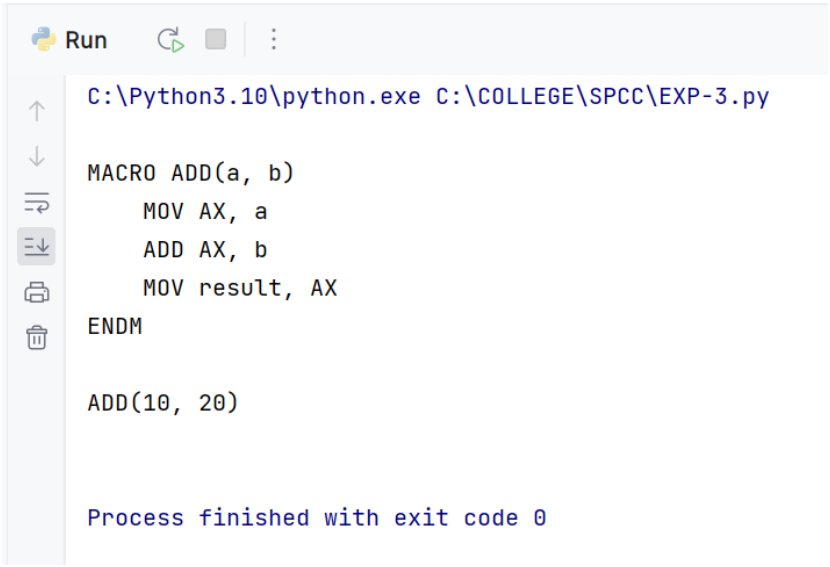
```
PG1 START 0
   USING *,BASE
   L 1,FOUR
   A 1,FIVE
   A 1,=F'7'
   A 1,=D'8'
   ST 1,TEMP
FOUR DC F'4'
FIVE DC F'5'
BASE EQV 8
TEMP DC '1'D
   END
```


Run
OUTPUT of Pass 1
Symbol Table (ST)
Symbol Value Length Relocation
PG1 0 ( 0 ) 1 R
FOUR 14 ( 20 ) 4 R
FIVE 18 ( 24 ) 4 R
BASE 8 ( 8 ) 1 A
TEMPÂ 1c ( 28 ) 8 R
Literal Table (LT)
Literal Value Length Relocation
=F'7' 24 ( 36 ) 4 R
=D'8' 28 ( 40 ) 8 R
OUTPUT of Pass 2
Machine Code
Instruction Machine Code
L d8 ( 216 )
A de ( 222 )
A ea ( 234 )
A ee ( 238 )

**// Expt. 3 Two Pass Macro Processor Implementation**

```
import re

# Regular expressions for macro definition and macro call DEFINITION_REGEX = r'^\s*MACRO\s+(\w+)\s+(.*)$'
CALL_REGEX = r'(\w+)\((.*)\)'

class Macro:
def init (self, name, parameters, code): self.name = name
self.parameters = parameters self.code = code

class TwoPassMacroProcessor: def init (self):
self.macros = {}

def first_pass(self, source):
# Extract macro definitions and build macro table
lines = source.split('\n') i = 0
while i < len(lines):
match = re.match(DEFINITION_REGEX, lines[i]) if match:
name, parameters = match.groups()
parameters = [p.strip() for p in parameters.split(',')] code = []
i += 1
while i < len(lines) and not re.match(DEFINITION_REGEX,

lines[i]):

code.append(lines[i]) i += 1
macro = Macro(name, parameters, code) self.macros[name] = macro
else:
i += 1

def second_pass(self, source):
# Replace macro calls with expanded code

lines = source.split('\n') for i in range(len(lines)):
match = re.search(CALL_REGEX, lines[i]) if match:
name, arguments = match.groups()
arguments = [a.strip() for a in arguments.split(',')] macro = self.macros.get(name)
if macro:
expanded_code = macro.code.copy() for j in range(len(arguments)):
expanded_code = [re.sub(r'\b' + macro.parameters[j]
+ r'\b', arguments[j], line) for line in expanded_code]
lines[i:i+1] = expanded_code return '\n'.join(lines)

def process(self, source): self.first_pass(source)
return self.second_pass(source) source = '''
MACRO ADD(a, b) MOV AX, a ADD AX, b
MOV result, AX ENDM

ADD(10, 20) '''

processor = TwoPassMacroProcessor()
result = processor.process(source) print(result)
```

**Output :**

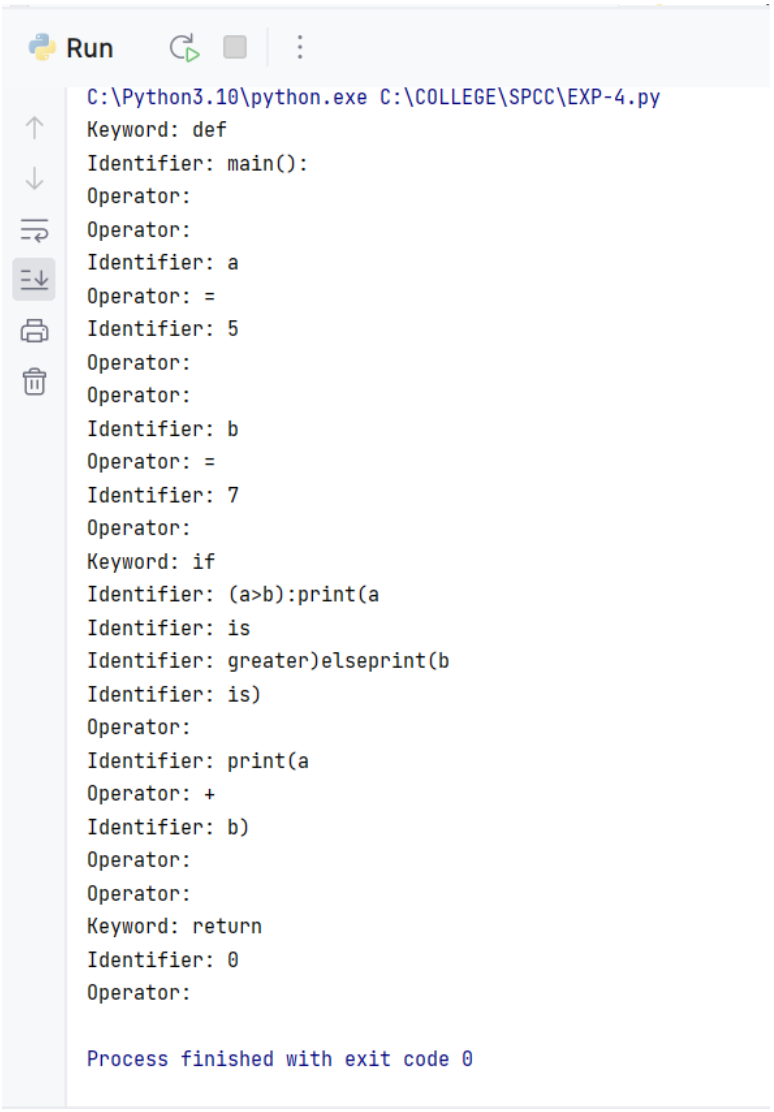



```
MOV AX, 10
ADD AX, 20
MOV result, AX
```

**// Expt. 4 Lexical Analyzer Implementation**

```
KEYWORDS = ["for", "while", "if", "else", "def", "return", "in", "not","and", "or", "print", "range", "input"]
FUNCTIONS = ["len", "int", "str", "float", "bool", "list", "tuple", "dict","set", "sorted", "max", "min"]
OPERATORS = ["+", "-", "", "/", "%", "//", "", "+=", "-=", "=", "/=", "=",
"==", "!=", "<", ">", "<=", ">=", "not", "in", "and", "or"]

def parse_code(code):for line
        in code:
                parts = line.split(" ")for part in
                parts:
                        if part in KEYWORDS: print("Keyword: " + part)
                        elif part in FUNCTIONS: print("Function: " + part)
                        elif part in OPERATORS: print("Operator: " + part)
                        else:
                                print("Identifier: " + part)

code = [
        "def main():","    a =
        5",
        "    b = 7",
        " if (a>b):"
        "print(a is greater)""else"
        "print(b is)"
        "    print(a + b)","    return
        0",
        ""]

parse_code(code)
```

**Output :**

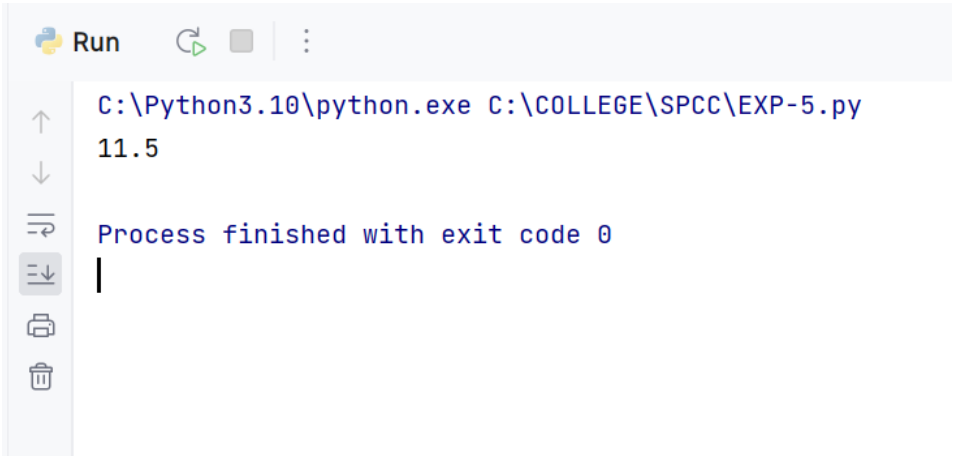

Run
C:\Python3.10\python.exe C:\COLLEGE\SPCC\EXP-4.py
Keyword: def
Identifier: main():
Operator:
Operator:
Identifier: a
Operator: =
Identifier: 5
Operator:
Operator:
Identifier: b
Operator: =
Identifier: 7
Operator:
Keyword: if
Identifier: (a>b):print(a
Identifier: is
Identifier: greater)elseprint(b
Identifier: is)
Operator:
Identifier: print(a
Operator: +
Identifier: b)
Operator:
Operator:
Keyword: return
Identifier: 0
Operator:
Process finished with exit code 0

**// Expt. 5 Parser Techniques Implementation**

```
import re

class Token:
        def init (self, token_type, value):self.token_type =
                token_type self.value = value

class Parser:
        def __init__(self, text):
                self.tokens = self.tokenize(text)self.pos = 0

        def parse(self):

                return self.expr()

        def tokenize(self, text):token_exprs
                = [
                        (r'\d+', 'INT'),
                        (r'\+', 'PLUS'),
                        (r'-', 'MINUS'),
                        (r'\*', 'MULTIPLY'),
                        (r'/', 'DIVIDE'),
                        (r'\(', 'LPAREN'),
                        (r'\)', 'RPAREN'),
                        (r'\s', None)   # skip whitespace
                ]
                tokens = []pos =
                0
                while pos < len(text):match =
                        None
                        for token_expr in token_exprs: pattern, token_type
                                = token_exprregex = re.compile(pattern)
                                match = regex.match(text, pos) if match:
                                        value = match.group(0)if
                                        token_type:
                                                token = Token(token_type, value)
                                                tokens.append(token)
                                        breakif
                        not match:
                                raise ValueError(f'Invalid input at position {pos}')else:
                                pos = match.end(0)return
                tokens

        def consume(self, token_type):
                if self.pos < len(self.tokens) and self.tokens[self.pos].token_type
== token_type:
                        self.pos += 1else:
                        raise ValueError(f'Expected token type {token_type} at position
{self.pos}')

        def factor(self):
                token = self.tokens[self.pos]if
                token.token_type == 'INT':
                        self.consume('INT') return
                        int(token.value)
                elif token.token_type == 'LPAREN':
                        self.consume('LPAREN')
                        value = self.expr()
                        self.consume('RPAREN')return
                        value
```

```
    def term(self):
        value = self.factor()
        while self.pos < len(self.tokens):token =
            self.tokens[self.pos]
            if token.token_type == 'MULTIPLY':
                self.consume('MULTIPLY')
                value *= self.factor()
            elif token.token_type == 'DIVIDE':
                self.consume('DIVIDE')

                value /= self.factor()else:
                break
        return value

    def expr(self):
        value = self.term()
        while self.pos < len(self.tokens):token =
            self.tokens[self.pos] if token.token_type ==
            'PLUS':
                self.consume('PLUS')value +=
                self.term()
            elif token.token_type == 'MINUS':
                self.consume('MINUS')
                value -= self.term()else:
                break
        return value

text = '2 * (3 + 4) - 5 / 2'parser =
Parser(text)
result = parser.parse() print(result)   # Output:
12.5
```

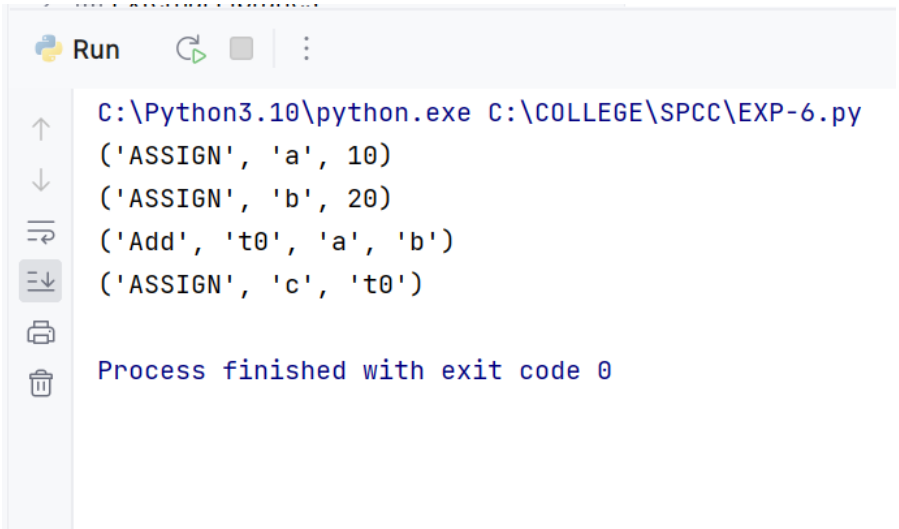
**Output :**



**// Expt. 6 Implement Intermediate code generation phase of compiler**

```
import ast

class IntermediateCodeGenerator(ast.NodeVisitor): def init (self):
self.instructions = [] self.temp_count = 0

def new_temp(self):
temp = f"t{self.temp_count}" self.temp_count += 1
return temp

def visit_Assign(self, node): target = node.targets[0].id value = self.visit(node.value)
self.instructions.append(('ASSIGN', target, value))

def visit_BinOp(self, node): left = self.visit(node.left)
right = self.visit(node.right)

op = node.op. class . name temp = self.new_temp()
self.instructions.append((op, temp, left, right)) return temp

def visit_Num(self, node): return node.n

def visit_Name(self, node): return node.id

def visit_Print(self, node):
value = self.visit(node.values[0]) self.instructions.append(('PRINT', value))

def generate_intermediate_code(source_code): ast_tree = ast.parse(source_code)
icg = IntermediateCodeGenerator() icg.visit(ast_tree)
return icg.instructions

# Example usage source_code = """ a = 10
b = 20
c = a + b print(c) """
instructions = generate_intermediate_code(source_code) for instruction in instructions:
print(instruction)
```

**Output :**



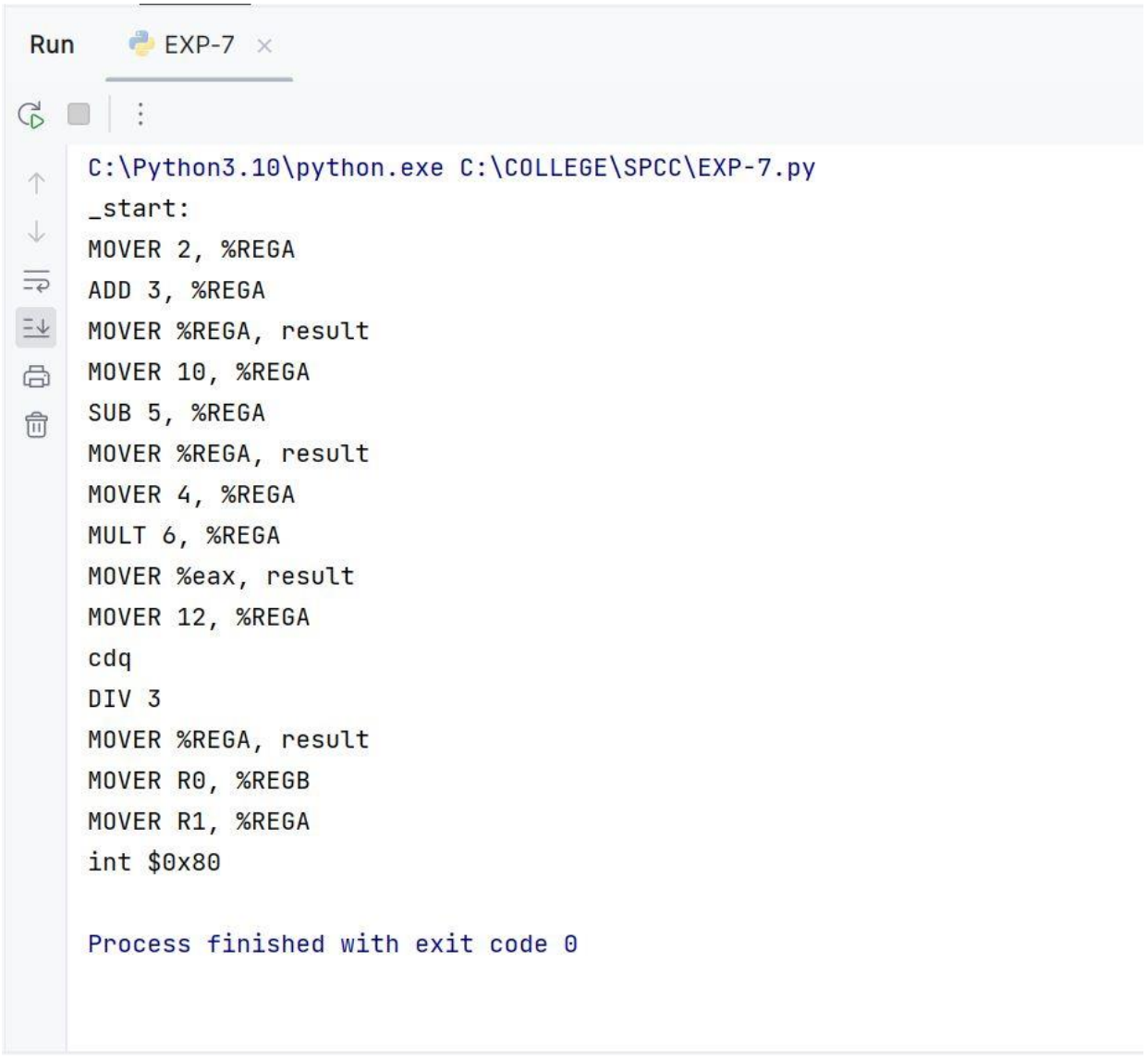
**// Expt. No. 7 Code Generation Algorithm Implementation**

```
class CodeGenerator:
    def _init_(self, intermediate_code):
        self.intermediate_code = intermediate_code
        self.generated_code = []

    def generate_code(self):
        self.generated_code.append('_start:')

        for instruction in self.intermediate_code:
            opcode = instruction['opcode']
            operands = instruction['operands']

            if opcode == 'ADD':
                self.add(operands)
            elif opcode == 'SUB':
                self.sub(operands)
            elif opcode == 'MULT':
                self.mult(operands)
            elif opcode == 'DIV':
                self.div(operands)
            else:
                raise ValueError(f"Invalid opcode '{opcode}'")

        self.generated_code.append('MOVER R0, %REGB')
        self.generated_code.append('MOVER R1, %REGA')
        self.generated_code.append('int $0x80')

    def add(self, operands):
        op1, op2, result = operands
        self.generated_code.append(f"MOVER {op1}, %REGA")
        self.generated_code.append(f"ADD {op2}, %REGA")
        self.generated_code.append(f"MOVER %REGA, {result}")

    def sub(self, operands):
        op1, op2, result = operands
        self.generated_code.append(f"MOVER {op1}, %REGA")
        self.generated_code.append(f"SUB {op2}, %REGA")
        self.generated_code.append(f"MOVER %REGA, {result}")

    def mult(self, operands):
        op1, op2, result = operands
        self.generated_code.append(f"MOVER {op1}, %REGA")
        self.generated_code.append(f"MULT {op2}, %REGA")
        self.generated_code.append(f"MOVER %eax, {result}")

    def div(self, operands):
        op1, op2, result = operands
        self.generated_code.append(f"MOVER {op1}, %REGA")
        self.generated_code.append(f"cdq")
        self.generated_code.append(f"DIV {op2}")
        self.generated_code.append(f"MOVER %REGA, {result}")


# Example intermediate code
intermediate_code = [
    {'opcode': 'ADD', 'operands': [2, 3, 'result']},
    {'opcode': 'SUB', 'operands': [10, 5, 'result']},
    {'opcode': 'MULT', 'operands': [4, 6, 'result']},
    {'opcode': 'DIV', 'operands': [12, 3, 'result']}
```

]

```
# Generate x86 assembly code
code_generator = CodeGenerator(intermediate_code)
code_generator.generate_code()
assembly_code = '\n'.join(code_generator.generated_code)

# Print generated x86 assembly code
print(assembly_code)
```

**Output :**



**// Expt. 8 LEX and YACC Implement**

/* 8.3 *Parser Using Lex And Yacc* */

***// LEX FILE***

```
%{
#include "y.tab.h"
extern int yylval;
%}

%%
[\t]+     ;
[0-9]+   {yylval=atoi(yytext); return(num);}
\n        {return 0;}
.         {return(yytext[0]);}
%%


int yywrap()
{
return 1;
}
```

***// YACC FILE***

```
%{
#include<stdio.h>
#include<stdlib.h>
%}
%start s
%token num
%left '+' '-'
%left '*' '/'
%%

s:E       {printf("Result=%d\n",$1);};
E:E'+'E  {$$=$1+$3;}
 |E'-'E {$$=$1-$3;}
 |E'*'E {$$=$1*$3;}
 |E'/'E {$$=$1/$3;}
 |num {$$=$1;}
%%


main()
{
 printf("Enter the expression:\n");
 yyparse();
}
int yyerror()
{
 return(1);
}
```

**----------OUTPUT----------**

**[**root@localhost Desktop]# yacc -d 38.y
[root@localhost Desktop]# lex spexp3.l
[root@localhost Desktop]# cc -o abc lex.yy.c y.tab.c -ll

[root@localhost Desktop]# ./abc

Enter the expression:
4+6-7+8
Result=11

[root@localhost Desktop]# ./abc

Enter the expression:
8*9/3
Result=24

[root@localhost Desktop]# ./abc

Enter the expression:
6*5-7
Result=23