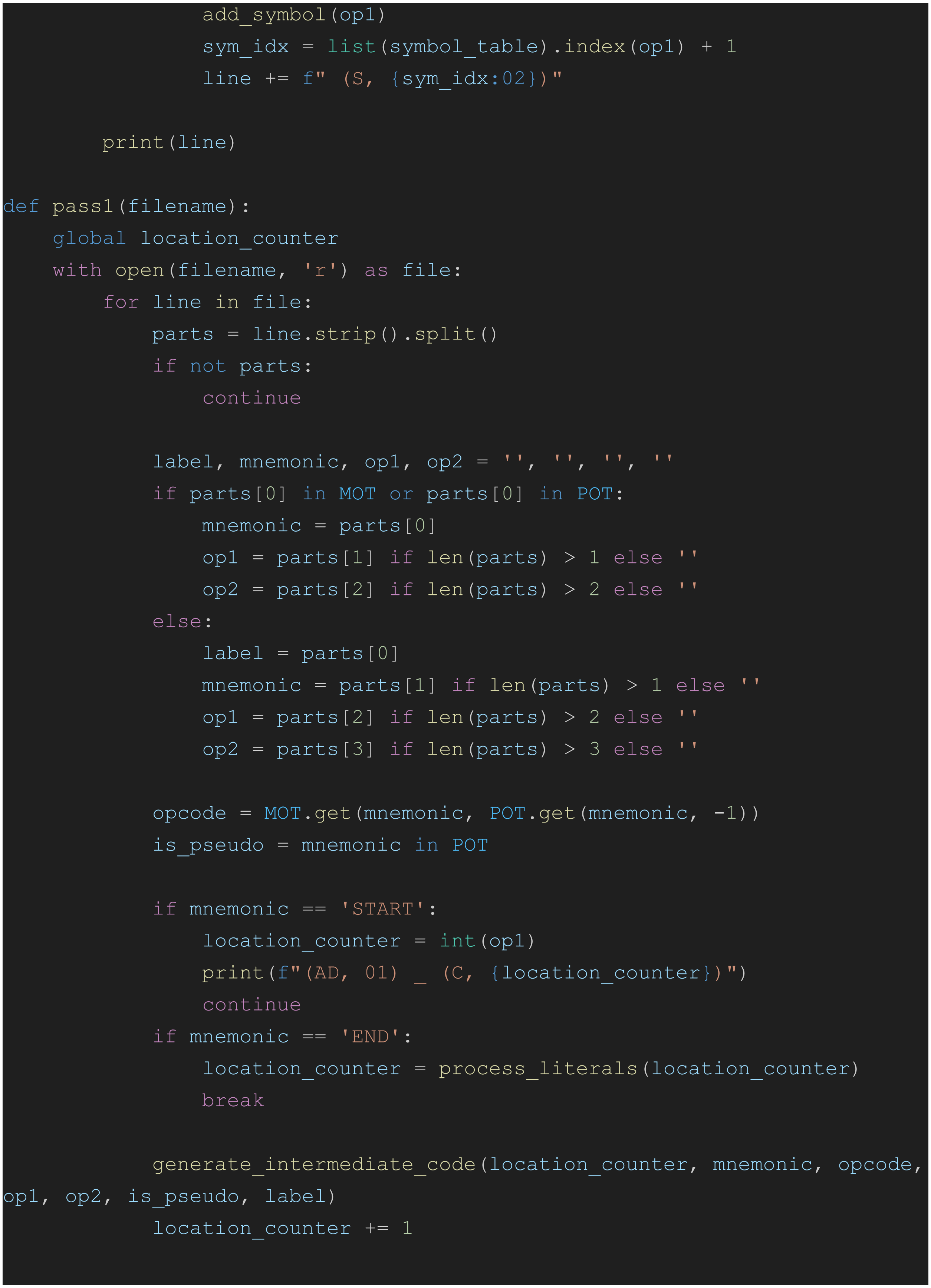
SPCC pracs exam

Q) Exp 1: Design & Implementation of Pass-1 of Two Pass Assembler.

Code:

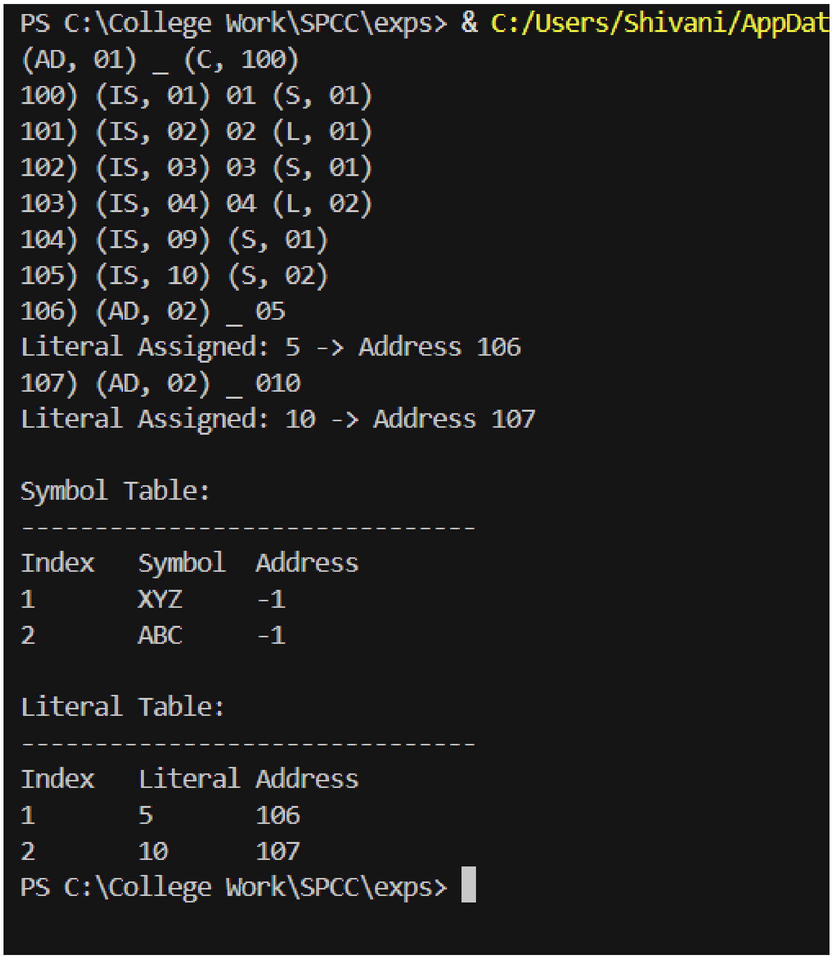
```
MOT = {
    "MOVER": 1, "MOVEM": 2, "ADD": 3, "SUB": 4, "MULT": 5,
    "DIV": 6, "BC": 7, "COMP": 8, "PRINT": 9, "READ": 10
}

POT = {"START": 1, "END": 2, "EQU": 3, "ORIGIN": 4, "LTORG": 5}
registers = {"AREG": 1, "BREG": 2, "CREG": 3, "DREG": 4}

symbol_table, literal_table = {}, []
location_counter = 0

def get_register_opcode(reg):
    return registers.get(reg, -1)

def add_symbol(symbol, address=-1):
    if symbol not in symbol_table:
        symbol_table[symbol] = address

def add_literal(value):
    if not any(lit["value"] == value for lit in literal_table):
        literal_table.append({"value": value, "address": -1})

def process_literals(current_address):
    for lit in literal_table:
        if lit["address"] == -1:
            lit["address"] = current_address
            print(f"{current_address:03}) (AD, 02) _ 0{lit['value']}")
            print(f"Literal Assigned: {lit['value']} -> Address
{current_address}")
            current_address += 1
    return current_address

def print_tables():
    print("\nSymbol Table:\n--------------------------------")
    print("Index\tSymbol\tAddress")
    for i, (sym, addr) in enumerate(symbol_table.items(), 1):
        print(f"{i}\t{sym}\t{addr}")
    print("\nLiteral Table:\n--------------------------------")
    print("Index\tLiteral\tAddress")
    for i, lit in enumerate(literal_table, 1):
        print(f"{i}\t{lit['value']}\t{lit['address']}")

def generate_intermediate_code(addr, mnemonic, opcode, op1, op2,
is_pseudo, label):
    if label:
        add_symbol(label, addr)

    if is_pseudo:
        if mnemonic in ("START", "ORIGIN"):
            print(f"(AD, {opcode:02}) _ (C, {op1})")
        else:
            print(f"(AD, {opcode:02}) _")
    else:
        line = f"{addr:03}) (IS, {opcode:02})"
        reg1 = get_register_opcode(op1)
        reg2 = get_register_opcode(op2)

        if reg1 != -1:
            line += f" {reg1:02}"
            if op2:
                if op2.startswith('#'):
                    add_literal(op2[1:])
                    lit_idx = len(literal_table)
                    line += f" (L, {lit_idx:02})"
                else:
                    add_symbol(op2)
                    sym_idx = list(symbol_table).index(op2) + 1
                    line += f" (S, {sym_idx:02})"
        elif op1:
            if op1.startswith('#'):
                add_literal(op1[1:])
                lit_idx = len(literal_table)
                line += f" (L, {lit_idx:02})"
            else:
```



```
                add_symbol(op1)
                sym_idx = list(symbol_table).index(op1) + 1
                line += f" (S, {sym_idx:02})"

        print(line)

def pass1(filename):
    global location_counter
    with open(filename, 'r') as file:
        for line in file:
            parts = line.strip().split()
            if not parts:
                continue

            label, mnemonic, op1, op2 = '', '', '', ''
            if parts[0] in MOT or parts[0] in POT:
                mnemonic = parts[0]
                op1 = parts[1] if len(parts) > 1 else ''
                op2 = parts[2] if len(parts) > 2 else ''
            else:
                label = parts[0]
                mnemonic = parts[1] if len(parts) > 1 else ''
                op1 = parts[2] if len(parts) > 2 else ''
                op2 = parts[3] if len(parts) > 3 else ''

            opcode = MOT.get(mnemonic, POT.get(mnemonic, -1))
            is_pseudo = mnemonic in POT

            if mnemonic == 'START':
                location_counter = int(op1)
                print(f"(AD, 01) _ (C, {location_counter})")
                continue
            if mnemonic == 'END':
                location_counter = process_literals(location_counter)
                break

            generate_intermediate_code(location_counter, mnemonic, opcode,
op1, op2, is_pseudo, label)
            location_counter += 1
```

```
if __name__ == "__main__":
    pass1("C:/College Work/SPCC/exps/exps pracs code python/input.txt")
    print_tables()
```

Output:

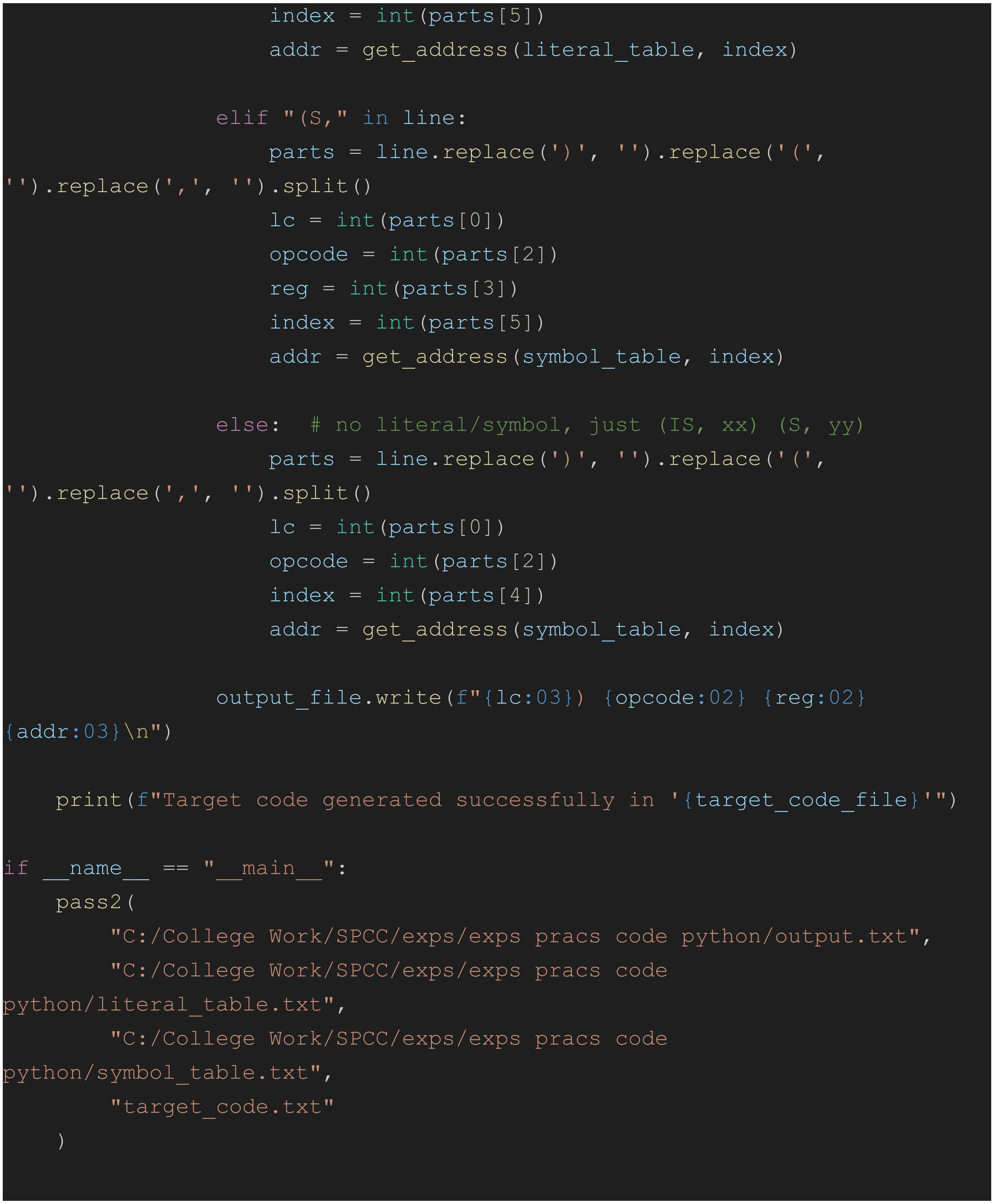


```
PS C:\College Work\SPCC\exps> & C:/Users/Shivani/AppDat
(AD, 01) _ (C, 100)
100) (IS, 01) 01 (S, 01)
101) (IS, 02) 02 (L, 01)
102) (IS, 03) 03 (S, 01)
103) (IS, 04) 04 (L, 02)
104) (IS, 09) (S, 01)
105) (IS, 10) (S, 02)
106) (AD, 02) _ 05
Literal Assigned: 5 -> Address 106
107) (AD, 02) _ 010
Literal Assigned: 10 -> Address 107

Symbol Table:
------------------------------
Index   Symbol  Address
1       XYZ     -1
2       ABC     -1

Literal Table:
------------------------------
Index   Literal Address
1       5       106
2       10      107
PS C:\College Work\SPCC\exps> 
```

Q) Exp 2: Design and Implementation of Pass 2 of Two Pass Assembler.

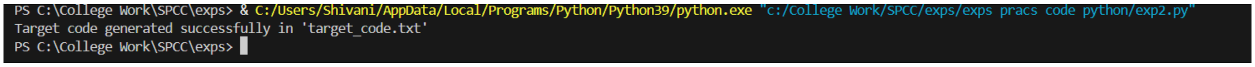
Code:

```
def load_table(filepath):
    table = []
    with open(filepath, 'r') as file:
        next(file)  # skip header line
        for line in file:
            parts = line.strip().split()
            if len(parts) == 3:
                _, value, address = parts
                table.append({'value': value, 'address': int(address)})
    return table

def get_address(table, index):
    if 0 <= index - 1 < len(table):
        return table[index - 1]['address']
    return -1

def pass2(intermediate_file, symbol_table_file, literal_table_file,
target_code_file):
    symbol_table = load_table(symbol_table_file)
    literal_table = load_table(literal_table_file)

    with open(intermediate_file, 'r') as input_file,
open(target_code_file, 'w') as output_file:
        for line in input_file:
            if "(AD" in line:
                continue  # skip assembler directives

            if "(IS" in line:
                lc, opcode, reg, index, addr = 0, 0, 0, -1, 0

                if "(L," in line:
                    parts = line.replace(')', '').replace('(',
'').replace(',', '').split()
                    lc = int(parts[0])
                    opcode = int(parts[2])
                    reg = int(parts[3])
```



```
                    index = int(parts[5])
                    addr = get_address(literal_table, index)

                elif "(S," in line:
                    parts = line.replace(')', '').replace('(',
'').replace(',', '').split()
                    lc = int(parts[0])
                    opcode = int(parts[2])
                    reg = int(parts[3])
                    index = int(parts[5])
                    addr = get_address(symbol_table, index)

                else:  # no literal/symbol, just (IS, xx) (S, yy)
                    parts = line.replace(')', '').replace('(',
'').replace(',', '').split()
                    lc = int(parts[0])
                    opcode = int(parts[2])
                    index = int(parts[4])
                    addr = get_address(symbol_table, index)

                output_file.write(f"{lc:03}) {opcode:02} {reg:02}
{addr:03}\n")

    print(f"Target code generated successfully in '{target_code_file}'")

if __name__ == "__main__":
    pass2(
        "C:/College Work/SPCC/exps/exps pracs code python/output.txt",
        "C:/College Work/SPCC/exps/exps pracs code
python/literal_table.txt",
        "C:/College Work/SPCC/exps/exps pracs code
python/symbol_table.txt",
        "target_code.txt"
    )
```

Output will be generated and stored, target.txt.



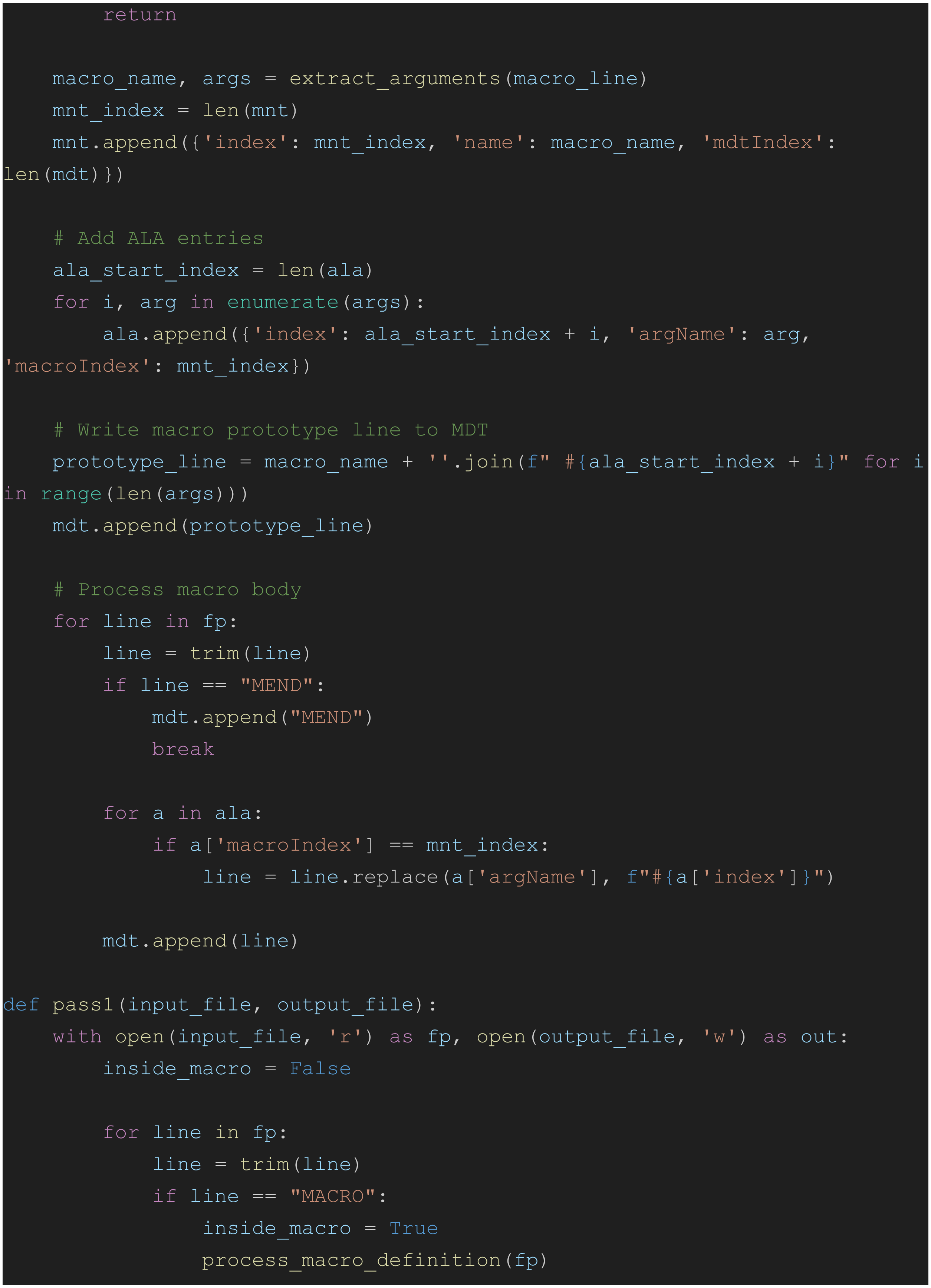
```
PS C:\College Work\SPCC\exps> & C:/Users/Shivani/AppData/Local/Programs/Python/Python39/python.exe "c:/College Work/SPCC/exps/exps pracs code python/exp2.py"
Target code generated successfully in 'target_code.txt'
PS C:\College Work\SPCC\exps>
```

target.txt

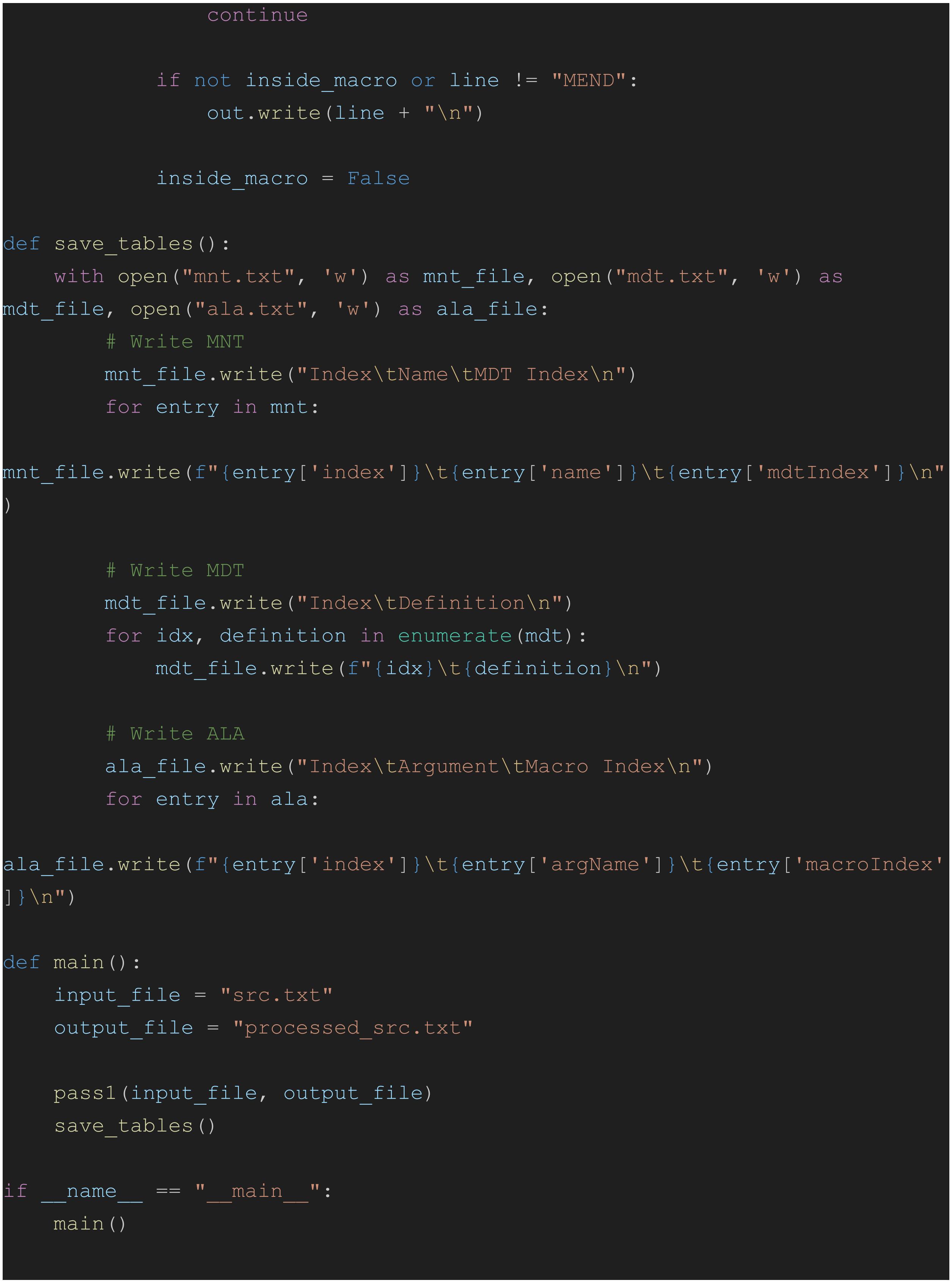
```
200) 04 01 209
201) 05 02 205
202) 04 01 210
204) 05 03 206
205) 02 02 211
206) 03 01 207
```

Q) Exp 3 - Design & Implementation of Pass 1 of Two Pass Macro Processor.

```
import os

MAX_LINE = 100
MAX_MACRO = 50
MAX_ARGS = 10

mnt = []
mdt = []
ala = []

def trim(s):
    return s.strip()

def extract_arguments(line):
    parts = line.strip().split()
    args = []
    if len(parts) > 1:
        arg_part = ' '.join(parts[1:])
        args = [arg.strip() for arg in arg_part.split(',') if
arg.strip().startswith('&')]
    return parts[0], args

def process_macro_definition(fp):
    macro_line = fp.readline()
    if not macro_line:
```



```python
        return

    macro_name, args = extract_arguments(macro_line)
    mnt_index = len(mnt)
    mnt.append({'index': mnt_index, 'name': macro_name, 'mdtIndex': len(mdt)})

    # Add ALA entries
    ala_start_index = len(ala)
    for i, arg in enumerate(args):
        ala.append({'index': ala_start_index + i, 'argName': arg, 'macroIndex': mnt_index})

    # Write macro prototype line to MDT
    prototype_line = macro_name + ''.join(f" #{ala_start_index + i}" for i in range(len(args)))
    mdt.append(prototype_line)

    # Process macro body
    for line in fp:
        line = trim(line)
        if line == "MEND":
            mdt.append("MEND")
            break

        for a in ala:
            if a['macroIndex'] == mnt_index:
                line = line.replace(a['argName'], f"#{a['index']}")

        mdt.append(line)

def pass1(input_file, output_file):
    with open(input_file, 'r') as fp, open(output_file, 'w') as out:
        inside_macro = False

        for line in fp:
            line = trim(line)
            if line == "MACRO":
                inside_macro = True
                process_macro_definition(fp)
```



```
                continue

            if not inside_macro or line != "MEND":
                out.write(line + "\n")

            inside_macro = False

def save_tables():
    with open("mnt.txt", 'w') as mnt_file, open("mdt.txt", 'w') as
mdt_file, open("ala.txt", 'w') as ala_file:
        # Write MNT
        mnt_file.write("Index\tName\tMDT Index\n")
        for entry in mnt:

mnt_file.write(f"{entry['index']}\t{entry['name']}\t{entry['mdtIndex']}\n"
)

        # Write MDT
        mdt_file.write("Index\tDefinition\n")
        for idx, definition in enumerate(mdt):
            mdt_file.write(f"{idx}\t{definition}\n")

        # Write ALA
        ala_file.write("Index\tArgument\tMacro Index\n")
        for entry in ala:

ala_file.write(f"{entry['index']}\t{entry['argName']}\t{entry['macroIndex'
]}\n")

def main():
    input_file = "src.txt"
    output_file = "processed_src.txt"

    pass1(input_file, output_file)
    save_tables()

if __name__ == "__main__":
    main()
```

Output:

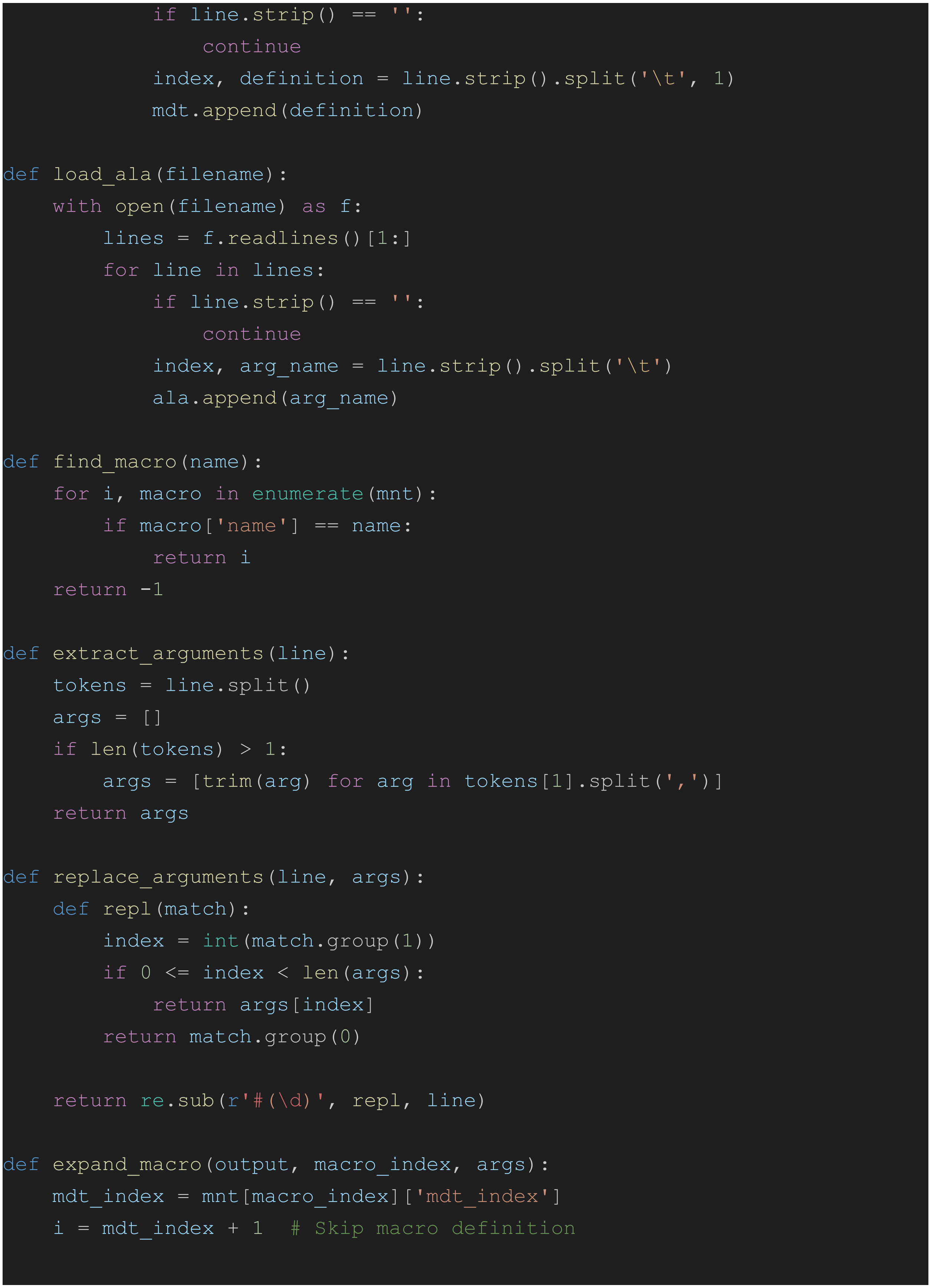
processed_src.txt

```
START 100
INCR A,B
END
```

Q) Exp 4: Design & Implementation of Pass 2 of Two Pass Macro Processor.

Code:

```python
import re

MAX_LINE = 100
MAX_MACRO = 50
MAX_ARGS = 10

mnt = []
mdt = []
ala = []

def trim(s):
    return s.strip()

def load_mnt(filename):
    with open(filename) as f:
        lines = f.readlines()[1:]  # Skip header
        for line in lines:
            if line.strip() == '':
                continue
            parts = line.strip().split('\t')
            if len(parts) == 3:
                index, name, mdt_index = parts
                mnt.append({'name': name, 'mdt_index': int(mdt_index)})

def load_mdt(filename):
    with open(filename) as f:
        lines = f.readlines()[1:]
        for line in lines:
```



```
            if line.strip() == '':
                continue
            index, definition = line.strip().split('\t', 1)
            mdt.append(definition)

def load_ala(filename):
    with open(filename) as f:
        lines = f.readlines()[1:]
        for line in lines:
            if line.strip() == '':
                continue
            index, arg_name = line.strip().split('\t')
            ala.append(arg_name)

def find_macro(name):
    for i, macro in enumerate(mnt):
        if macro['name'] == name:
            return i
    return -1

def extract_arguments(line):
    tokens = line.split()
    args = []
    if len(tokens) > 1:
        args = [trim(arg) for arg in tokens[1].split(',')]
    return args

def replace_arguments(line, args):
    def repl(match):
        index = int(match.group(1))
        if 0 <= index < len(args):
            return args[index]
        return match.group(0)

    return re.sub(r'#(\d)', repl, line)

def expand_macro(output, macro_index, args):
    mdt_index = mnt[macro_index]['mdt_index']
    i = mdt_index + 1  # Skip macro definition
    while i < len(mdt) and mdt[i] != "MEND":
        line = mdt[i]
        tokens = line.split()
        if tokens:
            nested_macro_index = find_macro(tokens[0])
            if nested_macro_index != -1:
                nested_args = extract_arguments(line)
                nested_args = [replace_arguments(arg, args) for arg in nested_args]
                expand_macro(output, nested_macro_index, nested_args)
            else:
                expanded_line = replace_arguments(line, args)
                output.write(expanded_line + '\n')
        i += 1

def pass2(input_file, output_file):
    with open(input_file) as inp, open(output_file, 'w') as out:
        in_macro_def = False
        for line in inp:
            line = trim(line)
            if line == '':
                continue
            if line == "MACRO":
                in_macro_def = True
                continue
            if line == "MEND":
                in_macro_def = False
                continue
            if in_macro_def:
                continue
            if line == "END":
                out.write("END\n")
                break

            tokens = line.split()
            macro_index = find_macro(tokens[0]) if tokens else -1
            if macro_index != -1:
                args = extract_arguments(line)
                expand_macro(out, macro_index, args)
            else:
                out.write(line + '\n')

if __name__ == "__main__":
    load_mnt("MNT.txt")
    load_mdt("MDT.txt")
    load_ala("ALA.txt")
    pass2("input.txt", "output.txt")
    print("Macro expansion complete. Check 'output.txt'.")
```

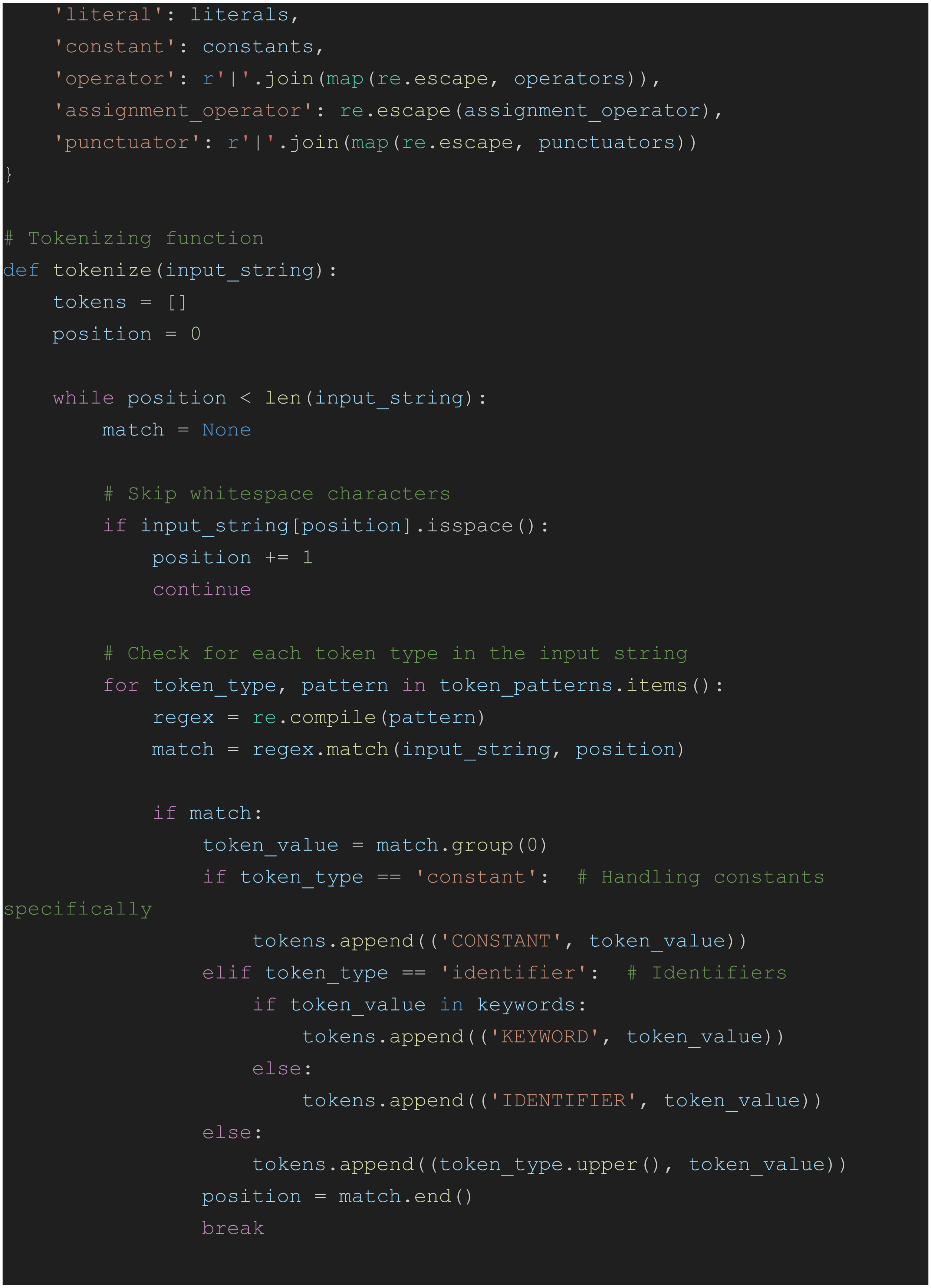
Output:
output.txt

```
START
LOAD VALUE1
ADD VALUE2
LOAD 5
ADD 10
END
```

Q) Exp 5: Study and Implementation of Lexical Analyzer.

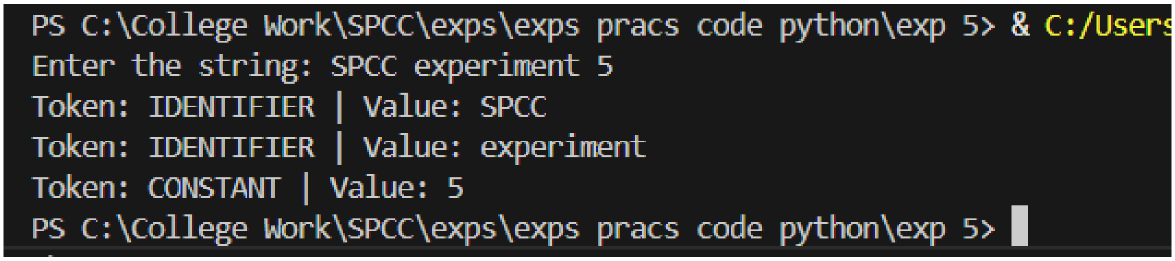
Code:

```
import re

# Define patterns for different token types using regex
keywords = {'if', 'else', 'while', 'for', 'return', 'int', 'float'}
punctuators = {',', ';', '(', ')', '{', '}'}
assignment_operator = '='
operators = {'+', '-', '*', '/', '%'}

constants = r'\d+(\.\d+)?'  # Matches integers or floating-point numbers
identifiers = r'[a-zA-Z_][a-zA-Z0-9_]*'  # Identifiers start with a letter or underscore
literals = r'\"[^\"]*\"'  # Matches strings enclosed in double quotes

# Regular expressions for each token
token_patterns = {
    'keyword': r'\b(?:' + '|'.join(keywords) + r')\b',
    'identifier': identifiers,
```



```
    'literal': literals,
    'constant': constants,
    'operator': r'|'.join(map(re.escape, operators)),
    'assignment_operator': re.escape(assignment_operator),
    'punctuator': r'|'.join(map(re.escape, punctuators))
}

# Tokenizing function
def tokenize(input_string):
    tokens = []
    position = 0

    while position < len(input_string):
        match = None

        # Skip whitespace characters
        if input_string[position].isspace():
            position += 1
            continue

        # Check for each token type in the input string
        for token_type, pattern in token_patterns.items():
            regex = re.compile(pattern)
            match = regex.match(input_string, position)

            if match:
                token_value = match.group(0)
                if token_type == 'constant':  # Handling constants
specifically
                    tokens.append(('CONSTANT', token_value))
                elif token_type == 'identifier':  # Identifiers
                    if token_value in keywords:
                        tokens.append(('KEYWORD', token_value))
                    else:
                        tokens.append(('IDENTIFIER', token_value))
                else:
                    tokens.append((token_type.upper(), token_value))
                position = match.end()
                break
        # If no match is found, print the unhandled character and raise an
error
        if not match:
            print(f"Unmatched character at position {position}:
{input_string[position]}")
            raise SyntaxError(f"Invalid character at position {position}")

    return tokens

# Input string for testing
input_string = input("Enter the string: ")

# Tokenize the input
try:
    tokens = tokenize(input_string)

    # Output the tokens
    for token in tokens:
        print(f"Token: {token[0]} | Value: {token[1]}")
except SyntaxError as e:
    print(e)
```

Output:

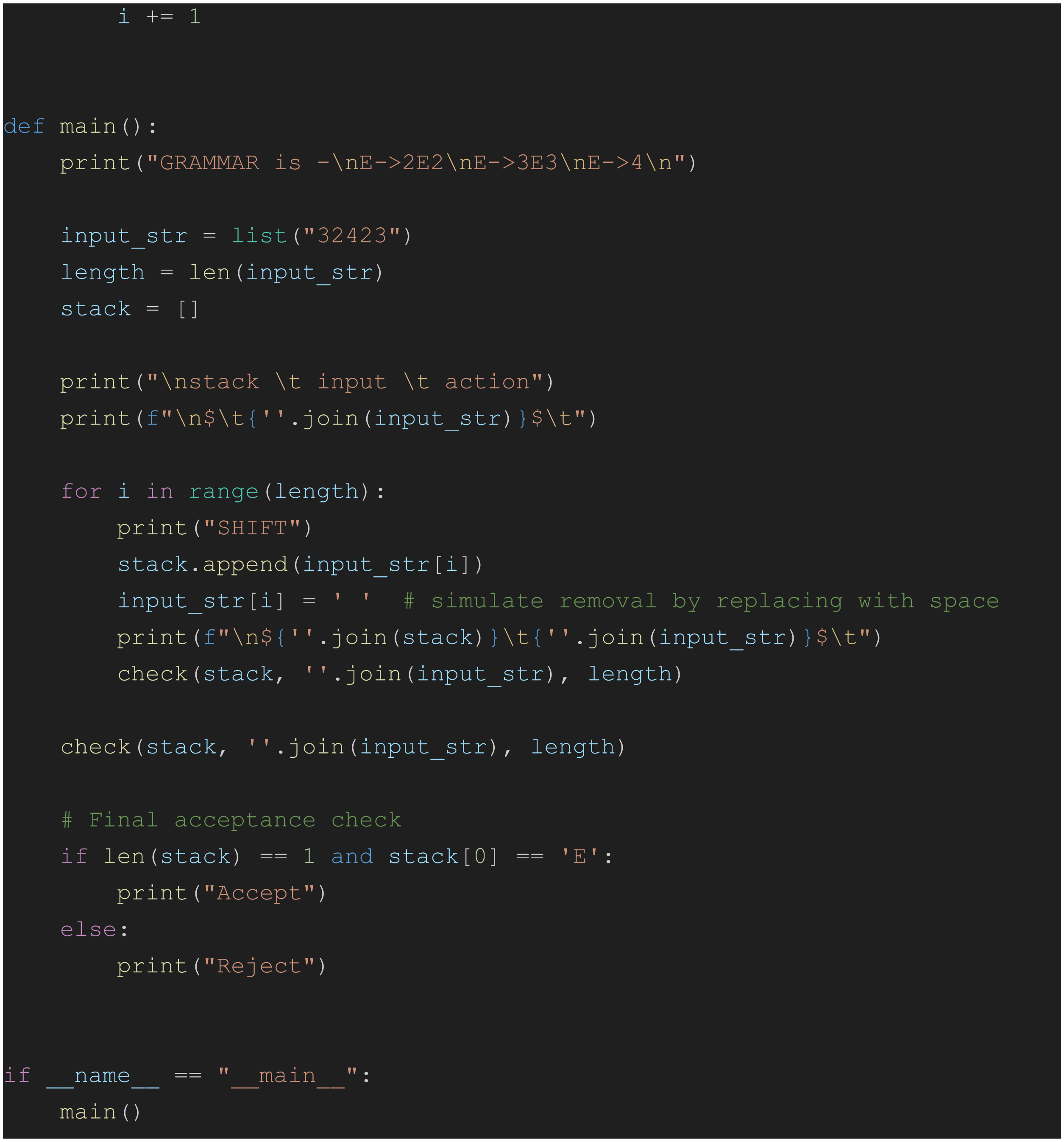


```
PS C:\College Work\SPCC\exps\exps pracs code python\exp 5> & C:/Users
Enter the string: SPCC experiment 5
Token: IDENTIFIER | Value: SPCC
Token: IDENTIFIER | Value: experiment
Token: CONSTANT | Value: 5
PS C:\College Work\SPCC\exps\exps pracs code python\exp 5> 
```

Q) Exp 6: Implementation of Shift Reduce Parser (SRP)

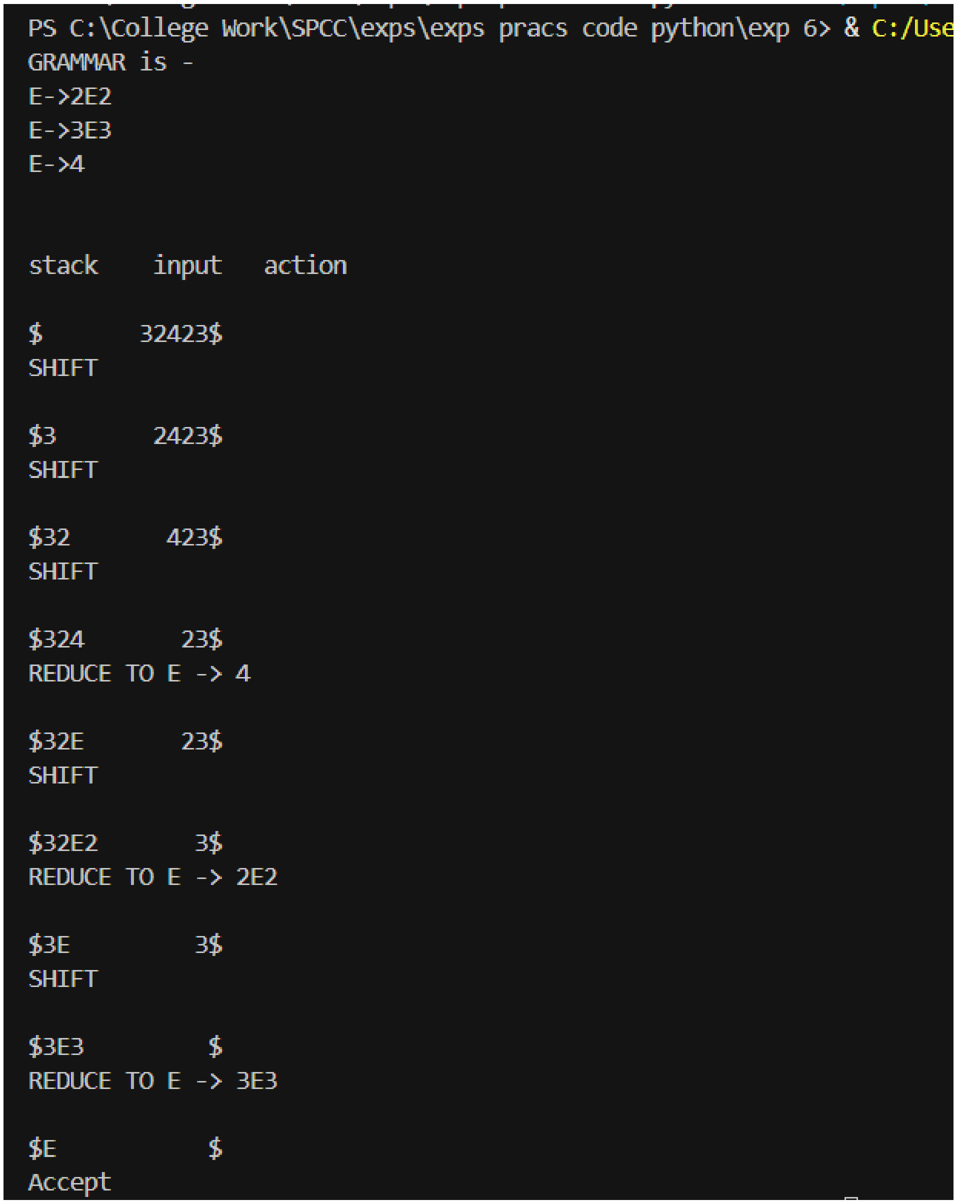
Code:

```
def check(stack, input_str, length):
    action = "REDUCE TO E -> "
    i = 0

    while i < len(stack):
        # Check for E -> 4
        if stack[i] == '4':
            print(f"{action}4")
            stack[i] = 'E'
            print(f"\n${''.join(stack)}\t{input_str}$\t")
        i += 1

    i = 0
    while i < len(stack) - 2:
        # Check for E -> 2E2
        if stack[i] == '2' and stack[i + 1] == 'E' and stack[i + 2] ==
'2':
            print(f"{action}2E2")
            stack[i] = 'E'
            del stack[i + 1]
            del stack[i + 1]
            print(f"\n${''.join(stack)}\t{input_str}$\t")
            i = -1  # Restart scan
        i += 1

    i = 0
    while i < len(stack) - 2:
        # Check for E -> 3E3
        if stack[i] == '3' and stack[i + 1] == 'E' and stack[i + 2] ==
'3':
            print(f"{action}3E3")
            stack[i] = 'E'
            del stack[i + 1]
            del stack[i + 1]
            print(f"\n${''.join(stack)}\t{input_str}$\t")
            i = -1  # Restart scan
```



```python
        i += 1


def main():
    print("GRAMMAR is -\nE->2E2\nE->3E3\nE->4\n")

    input_str = list("32423")
    length = len(input_str)
    stack = []

    print("\nstack \t input \t action")
    print(f"\n$\t{''.join(input_str)}$\t")

    for i in range(length):
        print("SHIFT")
        stack.append(input_str[i])
        input_str[i] = ' '  # simulate removal by replacing with space
        print(f"\n${''.join(stack)}\t{''.join(input_str)}$\t")
        check(stack, ''.join(input_str), length)

    check(stack, ''.join(input_str), length)

    # Final acceptance check
    if len(stack) == 1 and stack[0] == 'E':
        print("Accept")
    else:
        print("Reject")


if __name__ == "__main__":
    main()
```

Output:

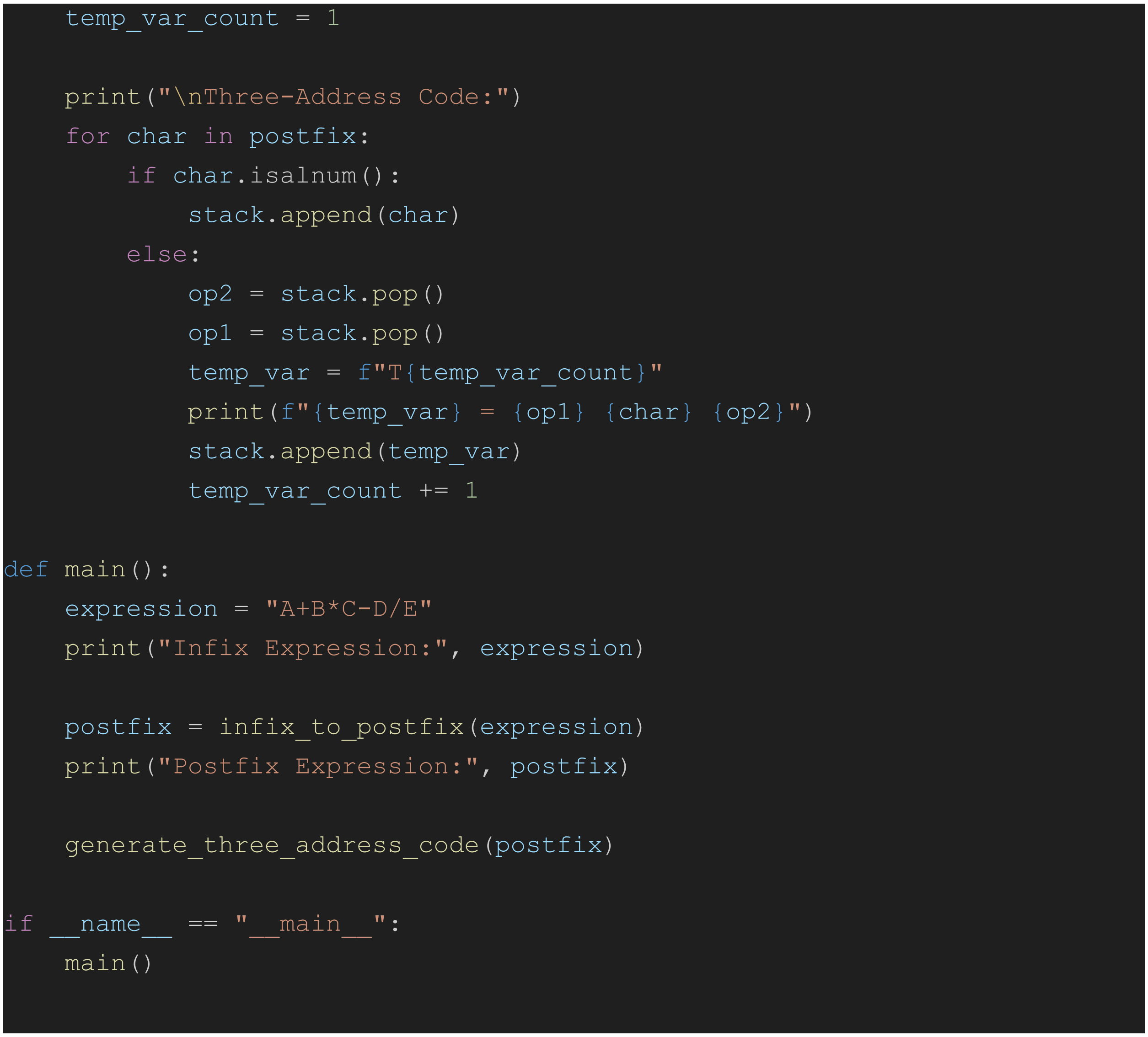


```
PS C:\College Work\SPCC\exps\exps pracs code python\exp 6> & C:/Use
GRAMMAR is -
E->2E2
E->3E3
E->4


stack    input   action

$       32423$
SHIFT

$3       2423$
SHIFT

$32       423$
SHIFT

$324       23$
REDUCE TO E -> 4

$32E       23$
SHIFT

$32E2       3$
REDUCE TO E -> 2E2

$3E         3$
SHIFT

$3E3         $
REDUCE TO E -> 3E3

$E           $
Accept
```

Q) Exp 7: Implementation of Intermediate Code Generation (ICG)

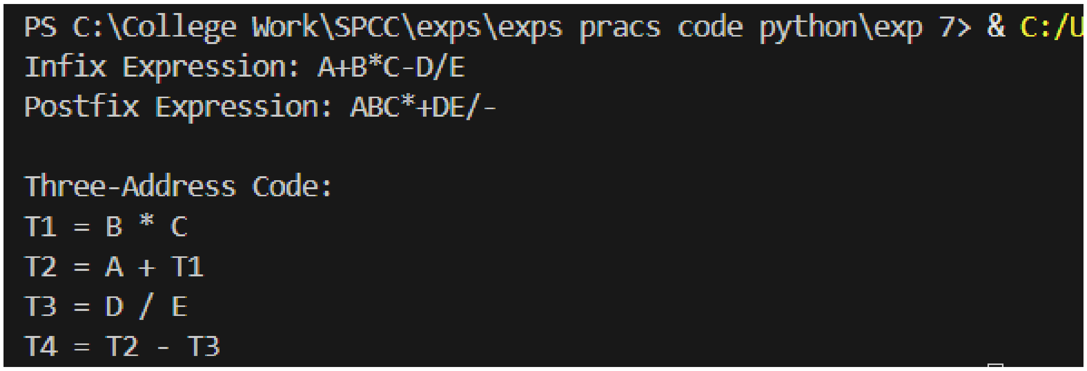
Code:

```
def precedence(op):
    if op == '^':
        return 3
    elif op in ('*', '/'):
        return 2
    elif op in ('+', '-'):
        return 1
    else:
        return -1

def infix_to_postfix(expression):
    stack = []
    postfix = []

    for char in expression:
        if char.isalnum():  # Operand
            postfix.append(char)
        elif char == '(':
            stack.append(char)
        elif char == ')':
            while stack and stack[-1] != '(':
                postfix.append(stack.pop())
            if stack:  # Pop the '('
                stack.pop()
        else:  # Operator
            while stack and precedence(char) <= precedence(stack[-1]):
                postfix.append(stack.pop())
            stack.append(char)

    while stack:
        postfix.append(stack.pop())

    return ''.join(postfix)

def generate_three_address_code(postfix):
    stack = []
```



```
    temp_var_count = 1

    print("\nThree-Address Code:")
    for char in postfix:
        if char.isalnum():
            stack.append(char)
        else:
            op2 = stack.pop()
            op1 = stack.pop()
            temp_var = f"T{temp_var_count}"
            print(f"{temp_var} = {op1} {char} {op2}")
            stack.append(temp_var)
            temp_var_count += 1

def main():
    expression = "A+B*C-D/E"
    print("Infix Expression:", expression)

    postfix = infix_to_postfix(expression)
    print("Postfix Expression:", postfix)

    generate_three_address_code(postfix)

if __name__ == "__main__":
    main()
```

Output:



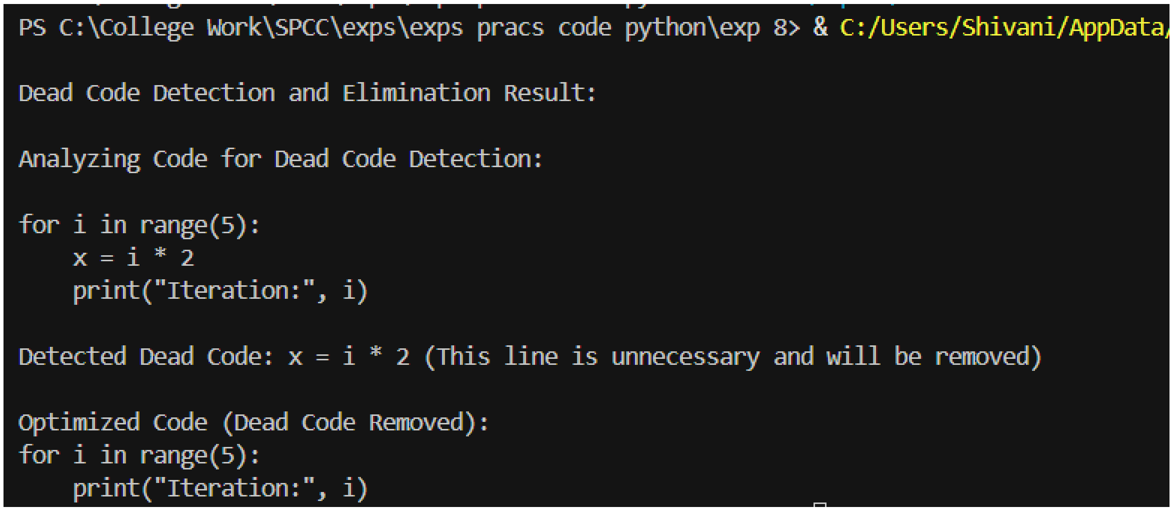
```
PS C:\College Work\SPCC\exps\exps pracs code python\exp 7> & C:/U
Infix Expression: A+B*C-D/E
Postfix Expression: ABC*+DE/-

Three-Address Code:
T1 = B * C
T2 = A + T1
T3 = D / E
T4 = T2 - T3
```

Q) Exp 8: Implementation of code optimization phase of compiler.

Code:

```
import re

def detect_and_eliminate_dead_code():
    user_code = """
for i in range(5):
    x = i * 2
    print("Iteration:", i)
    """

    print("\nAnalyzing Code for Dead Code Detection:")
    print(user_code)

    lines = user_code.strip().split('\n')
    optimized_code = []
    dead_code_detected = False

    # Define patterns for unused assignments
    dead_code_patterns = [
        r"\b(z|unused_variable|x)\s*=.*"  # Matches assignment to unused
vars like x
    ]

    for line in lines:
        if any(re.search(pattern, line) for pattern in
dead_code_patterns):
            if "print" not in line:
                print(f"Detected Dead Code: {line.strip()} (This line is
unnecessary and will be removed)")
                dead_code_detected = True
                continue  # Skip adding dead code
        optimized_code.append(line)

    optimized_code_str = '\n'.join(optimized_code)

    if not dead_code_detected:
        print("No dead code detected.")
    print("\nOptimized Code (Dead Code Removed):")
    print(optimized_code_str)

    return optimized_code_str

if __name__ == "__main__":
    print("\nDead Code Detection and Elimination Result:")
    detect_and_eliminate_dead_code()
```

Output:



```
PS C:\College Work\SPCC\exps\exps pracs code python\exp 8> & C:/Users/Shivani/AppData

Dead Code Detection and Elimination Result:

Analyzing Code for Dead Code Detection:

for i in range(5):
    x = i * 2
    print("Iteration:", i)

Detected Dead Code: x = i * 2 (This line is unnecessary and will be removed)

Optimized Code (Dead Code Removed):
for i in range(5):
    print("Iteration:", i)
```

Q) Exp 9: Study and implementation of lexical analyzer using LEX Tool and parser using YACC Tool

Q) Exp 10: Case study on Optimization Techniques in Compilers.

Both exps in zip folder as docs