ANNAMALAI  UNIVERSITY



# FACULTY OF ENGINEERING AND TECHNOLOGY

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**B.E. Information Technology**

**V - SEMESTER**

## 22ITCP508 – SYSTEM SOFTWARE AND COMPILER DESIGN LAB

# LABORATORY RECORD

**(July 2024 – December 2024)**

**Name:** _____

**Register No.:** _____

ANNAMALAI  UNIVERSITY

# FACULTY OF ENGINEERING AND TECHNOLOGY

## DEPARTMENT OF INFORMATION TECHNOLOGY

**B.E. Information Technology**

**V-SEMESTER**

## 22ITCP508 – SYSTEM SOFTWARE AND COMPILER DESIGN LAB

Certified that this is the bona fide record of work done by

Mr./Ms._____

Reg. No._____ of B.E. (Information Technology) in the

**22ITCP508 – SYSTEM SOFTWARE AND COMPILER DESIGN LAB** during the odd semester of the

academic year 2024 – 2025.

Place: Annamalai Nagar
Date: / / 2024.

**Staff in-charge**

**Internal Examiner** **External Examiner**

## DEPARTMENT OF INFORMATION TECHNOLOGY

## B.E INFORMATION TECHNOLOGY

VISION:

To produce globally competent, quality technocrats, to inculcate values of leadership and research qualities and to play a vital role in the socio-economic progress of the nation.

MISSION:

- To partner with the University community to understand the information technology needs of faculty, staff and students.
- To develop dynamic IT professionals with globally competitive learning experience by providing high class education.
- To involve graduates in understanding need-based Research activities and disseminate the knowledge to develop entrepreneur skills.

### PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

| PEO | PEO Statements |
|---|---|
| PEO1 | To offer students with **core competence** in mathematical, scientific and basic engineering rudiments necessary to prepare, analyze and solve hardware/software engineering problems and/or also to pursue advanced study or research. |
| PEO2 | To educate students with good **scope** of knowledge in core areas of IT and related engineering so as to comprehend engineering trade-offs, analyze, design, and synthesize data and technical concepts to create novel products and solutions for the real-life problems. |
| PEO3 | To instil in students to maintain high **proficiency** and ethical standards, effective oral and written communication skills, to work as part of teams on multidisciplinary projects and diverse professional environments, and relate engineering issues to the society, global economy and to emerging technologies. |
| PEO4 | To deliver our graduates with **learning environment** awareness of the life-long learning needed for a successful professional career and to introduce them to written ethical codes and guidelines, perform excellence, leadership and demonstrate good citizenship. |

## COURSE OBJECTIVES

1. Be familiar with the main features of C language
2. Understands Assembler and loader paradigm
3. Apply the concepts of parser

## COURSE OUTCOMES:

At the end of this course, the students will be able to

1. Understand and use of assembler and loader.
2. Understand and define the role of lexical analyzer, use of regular expression and transition diagrams.
3. Understand and use Context free grammar, and parse tree construction.

# CONTENTS

**Ex.No: 1**

**Date:**

## IMPLEMENTATION OF SIMPLE TEXT EDITOR

**AIM:**

To write a program for implementing Simple Text editor using Python.

**ALGORITHM:**

1. Initialize an empty string variable for content.
2. Display available commands: NEW, OPEN, SAVE, INSERT, DELETE, HELP, QUIT.
3. Prompt the user for their choice.
4. If "NEW" is chosen, create a new file and set content to an empty string.
5. If "OPEN" is chosen, prompt for a filename, read and display the file content if it exists.
6. If "SAVE" is chosen, prompt for a filename and save the current content to it.
7. If "INSERT" is chosen, prompt for the text type (word/sentence/character) and position, then update the content.
8. If "DELETE" is chosen, prompt for the text type and positions to delete, then update the content.
9. If "HELP" is chosen, display the help menu with available commands.
10. If "QUIT" is chosen, print an exit message and terminate the program.

**PROGRAM:**

```
import os

def new_file():
    return ""

def open_file():
    filename = input("Enter the filename to open: ")
```

```
    if os.path.exists(filename):
        with open(filename, 'r') as file:
            content = file.read()
        print(f"File '{filename}' opened successfully.")
        return content
    else:
        print(f"File '{filename}' not found.")
        return ""

def save_file(content):
    filename = input("Enter the filename to save: ")
    with open(filename, 'w') as file:
        file.write(content)
    print(f"File saved as '{filename}'.")

def help_menu():
    """Display the help menu"""
    print("\n--- HELP MENU ---")
    print("1. NEW: Create a new text.")
    print("2. OPEN: Open an existing file.")
    print("3. SAVE: Save the current text to a file.")
    print("4. INSERT: Insert text (word/sentence) or character.")
    print("5. DELETE: Delete text (word/sentence) or character.")
    print("6. QUIT: Exit the editor.")
    print("------------------\n")

def insert_text(content):
    insert_type = input("Insert (word/sentence/character): ").lower()
    if insert_type == "word" or insert_type == "sentence":
        position = int(input("Enter the position to insert at: "))
        text_to_insert = input(f"Enter the {insert_type} to insert: ")
        content = content[:position] + text_to_insert + content[position:]
    elif insert_type == "character":
        position = int(input("Enter the position to insert the character: "))
        char_to_insert = input("Enter the character to insert: ")
        content = content[:position] + char_to_insert + content[position:]
    else:
        print("Invalid insert type.")
    return content

def delete_text(content):
```

```
    delete_type = input("Delete (word/sentence/character): ").lower()
    if delete_type == "word" or delete_type == "sentence":
        start_pos = int(input(f"Enter the start position of the {delete_type} to delete: "))
        end_pos = int(input(f"Enter the end position of the {delete_type} to delete: "))
        content = content[:start_pos] + content[end_pos:]
    elif delete_type == "character":
        position = int(input("Enter the position of the character to delete: "))
        content = content[:position] + content[position+1:]
    else:
        print("Invalid delete type.")
    return content

def main():
    content = ""
    while True:
        print("\nOptions: NEW, OPEN, SAVE, INSERT, DELETE, HELP, QUIT")
        choice = input("Enter your choice: ").lower()

        if choice == "new":
            content = new_file()
            print("New text file created.")
        elif choice == "open":
            content = open_file()
            print("Current content:\n", content)
        elif choice == "save":
            save_file(content)
        elif choice == "insert":
            content = insert_text(content)
            print("Updated content:\n", content)
        elif choice == "delete":
            content = delete_text(content)
            print("Updated content:\n", content)
        elif choice == "help":
            help_menu()
        elif choice == "quit":
            print("Exiting the text editor.")
            break
        else:
            print("Invalid choice. Type 'HELP' to see available commands.")

main()
```

**Sample Input and Output:**

```
Options: NEW, OPEN, SAVE, INSERT, DELETE, HELP, QUIT
Enter your choice: open
Enter the filename to open: new.txt
File 'new.txt' opened successfully.
Current content:
 hello

Options: NEW, OPEN, SAVE, INSERT, DELETE, HELP, QUIT
Enter your choice: insert
Insert (word/sentence/character): word
Enter the position to insert at: 6
Enter the word to insert: world
Updated content:
 helloworld

Options: NEW, OPEN, SAVE, INSERT, DELETE, HELP, QUIT
Enter your choice: save
Enter the filename to save: new.txt
File saved as 'new.txt'.

Options: NEW, OPEN, SAVE, INSERT, DELETE, HELP, QUIT
Enter your choice: quit
Exiting the text editor.
```

**RESULT:**

Thus the program for implementation of Simple Text Editor is executed and verified.

**Ex.No: 2**

**Date:**

## IMPLEMENTATION OF PASS ONE OF TWO PASS SIC ASSEMBLER.

**AIM:**

To write a Python program for implementing PASS ONE of the TWO PASS assembler.

.

**ALGORITHM:**

1. Define a dictionary to map opcodes to their corresponding instruction codes.

2. Read the assembly source code from a file, line by line.

3. Initialize the location counter (`locctr`) to the specified starting address.

4. Create an empty list for the intermediate file and a dictionary for the symbol table (`symtab`).

5. Iterate through each line of the source code:
   - Skip comments (lines starting with a dot).
   - Process the line to extract the label, opcode, and operand.

6. If the opcode is `START`, set `locctr` to the specified starting address and store the entry in the intermediate file.

7. If a label is present, check for duplicates in the symbol table and add the label with its address to the table.

8. Append the current `locctr`, label, opcode, and operand to the intermediate file.

9. Increment by 3 for standard instructions (`LDA`, `ADD`, etc.), or based on the operand size for `WORD`, `RESW`, and `RESB`.

10. Write the contents of the intermediate file and symbol table to separate output files once processing is complete.

**Program :**

```
INSTRUCTION_SET = {
   'LDA': 0x00, 'LDX': 0x04, 'STA': 0x0C, 'STX': 0x10,
   'ADD': 0x18, 'SUB': 0x1C, 'MUL': 0x20, 'DIV': 0x24
}

def process_line(line):
   parts = line.split()
   if len(parts) == 3:
      label, opcode, operand = parts
   elif len(parts) == 2:
      label = None
      opcode, operand = parts

   elif len(parts) == 1:  # opcode only
      label = None
      opcode = parts[0]
      operand = None
   else:
      raise SyntaxError("Invalid instruction format")
   return label, opcode, operand

def pass_one_sic_assembler(source_code, start_address=0):
   locctr = start_address
   intermediate_file = []
   symtab = {}

   for line in source_code:
      line = line.strip()

      if line.startswith("."):
         continue

      label, opcode, operand = process_line(line)


      if opcode == 'START':
         locctr = int(operand, 16)
         intermediate_file.append((locctr, label, opcode, operand))
```

```
            continue


        if label:
            if label in symtab:
                raise ValueError(f"Duplicate symbol: {label}")
            symtab[label] = locctr


        intermediate_file.append((locctr, label, opcode, operand))


        if opcode in INSTRUCTION_SET:
            locctr += 3
        elif opcode == 'WORD':
            locctr += 3
        elif opcode == 'RESW':
            locctr += 3 * int(operand)
        elif opcode == 'RESB':
            locctr += int(operand)
        elif opcode == 'END':
            break
        else:
            raise ValueError(f"Invalid opcode: {opcode}")

    return intermediate_file, symtab

source_code = None
with open('program.asm','r') as f:
    source_code = f.readlines()

intermediate_file, symtab = pass_one_sic_assembler(source_code)

with open("intermediate.txt", "w") as int_file:
    for entry in intermediate_file:
        locctr, label, opcode, operand = entry
        int_file.write(f"{locctr:04X}  {label or ''}  {opcode}  {operand or ''}\n")

with open("symtab.txt", "w") as symtab_file:
    for symbol, address in symtab.items():
        symtab_file.write(f"{symbol}  {address:04X}\n")
```

**print("Pass One Complete. Intermediate file and SYMTAB created.")**

**Program.asm**

```
COPY START 1000
FIRST LDA ALPHA
    STA BETA
    SUB 1
ALPHA WORD 1
BETA RESW 1
    END FIRST
```

**Intermediate.txt :**

```
1000  COPY  START  1000
1000  FIRST  LDA  ALPHA
1003   STA  BETA
1006   SUB  1
1009  ALPHA  WORD  1
100C  BETA  RESW  1
100F   END  FIRST
```

**Symtab.txt**

```
FIRST  1000
ALPHA  1009
BETA  100C
```

**RESULT:**

Thus the program for implementation of PASS ONE of the TWO PASS assembler is executed and verified.

Ex.No: 3

**Date:**

## IMPLEMENTATION OF PASS TWO OF TWO PASS SIC ASSEMBLER.

**AIM:**

To write a Python program for implementing PASS TWO of the TWO PASS assembler.

.

**ALGORITHM:**

1. Create a dictionary mapping instruction mnemonics to their corresponding hexadecimal opcode values.

2. Set up variables for the object code, header record, text records, and end record.

3. Loop through each entry in the intermediate file, which contains location counter values, labels, opcodes, and operands.

4. If the opcode is `START`, extract the start address and program name, and create the header record.

5. For each opcode in the instruction set:
   - Retrieve the corresponding hexadecimal opcode and shift it to the left by 16 bits to prepare for 3-byte format.
   - Look up the operand address in the symbol table; use 0 if the operand is not found.
   - Combine the opcode and operand addresses to form the complete instruction object code.

6. Ignore these opcodes as they do not generate object code.

7. If the opcode is `END`, create the end record using the address from the symbol table.

8. Format the text record with the start address, length of the text records, and concatenate all object codes into a single string.

9. Determine the length of the program using the difference between the final location counter and the start address.

10. Save the header record, text record, and end record to an object code file. Print a message indicating that Pass Two is complete and the object code has been generated.

**PROGRAM:**

```
INSTRUCTION_SET = {
    'LDA': 0x00, 'LDX': 0x04, 'STA': 0x0C, 'STX': 0x10,
    'ADD': 0x18, 'SUB': 0x1C, 'MUL': 0x20, 'DIV': 0x24
}

def pass_two_sic_assembler(intermediate_file, symtab):
    object_code = []
    header_record = None
    text_records = []
    end_record = None

    for line in intermediate_file:
        locctr, label, opcode, operand = line


        if opcode == 'START':
            start_address = int(operand, 16)
            program_name = label or "DEFAULT"
            header_record = f"H{program_name:<6}{start_address:06X}{0:06X}"
            continue

        if opcode in INSTRUCTION_SET:
            opcode_hex = INSTRUCTION_SET[opcode] << 16  # shift to 24 bits as 3 bytes

            if operand in symtab:
                operand_address = symtab[operand]
            else:
                operand_address = 0  # If operand is not found, use 0 (error case should be handled)

            instruction_obj_code = opcode_hex + operand_address

            object_code_hex = f"{instruction_obj_code:06X}"

            text_records.append(object_code_hex)

        elif opcode == 'RESW' or opcode == 'RESB':
            continue
```

```
        elif opcode == 'END':
            end_record = f"E{symtab[operand]:06X}"


    text_record_str = "T{:<06X}{:<02X}".format(start_address, len(text_records) * 3) +
''.join(text_records)

    program_length = locctr - start_address
    header_record = f"H{program_name:<6}{start_address:06X}{program_length:06X}"


    with open("object_code.txt", "w") as obj_file:
        obj_file.write(header_record + "\n")
        obj_file.write(text_record_str + "\n")
        obj_file.write(end_record + "\n")

    print("Pass Two Complete. Object code generated.")

intermediate_file = [
    (0x1000, 'COPY', 'START', '1000'),
    (0x1000, 'FIRST', 'LDA', 'FIVE'),
    (0x1003, None, 'ADD', 'BETA'),
    (0x1006, None, 'SUB', 'GAMMA'),
    (0x1009, None, 'STA', 'ALPHA'),
    (0x100C, 'SECOND', 'LDX', 'ALPHA'),
    (0x100F, None, 'STX', 'DELTA'),
    (0x1012, None, 'RESW', '1'),
    (0x1015, None, 'RESB', '5'),
    (0x101A, None, 'END', 'FIRST')
]

# Sample symbol table (from Pass One)
symtab = {
    'COPY': 0x1000,
    'FIRST': 0x1000,
    'SECOND': 0x100C,
    'FIVE': 0x1018,
    'BETA': 0x101B,
    'GAMMA': 0x101E,
    'ALPHA': 0x1021,
```

**'DELTA': 0x1024**
**}**

**pass_two_sic_assembler(intermediate_file, symtab)**

**Sample Input and Output:**

Pass Two Complete. Object code generated.

**Object_code.txt**

HCOPY 00100000001A
T1000001200101818101B1C101E0C1021041021101024
E001000

**RESULT:**

**Thus the program for implementation of PASS TWO of the TWO PASS assembler is executed and verified.**

**Ex.No: 4**

**Date:**

## IMPLEMENTATION OF PASS ONE & TWO OF TWO PASS MACRO PROCESSOR.

**AIM:**

To write a Python program for implementing PASS ONE & TWO of the TWO PASS macro processor.

.

**ALGORITHM:**

1. Create `pass_one` to build the MNT and MDT from the source code.

2. Set up MNT, MDT, and a counter.

3. Loop through the source code lines.

4. If a line starts with `MACRO`, extract the name and parameters, adding them to MNT and MDT.

5. If a line is `MEND`, mark the end of the macro and add it to MDT.

6. While inside a macro, add each line to MDT.

7. After processing, return the MNT and MDT.

8. Create `pass_two` to expand macro calls.

9. For each line, check for macro calls in the MNT, replace parameters with arguments, and append expanded lines.

10. Return the expanded code and print MNT, MDT, and the expanded output.

**PROGRAM:**

```
import re

def pass_one(source_code):

    mnt = {}  # Macro Name Table
    mdt = []  # Macro Definition Table
    mdtc = 0  # Macro Definition Table Counter
    inside_macro = False
    macro_name = None

    for line in source_code:
        tokens = line.split()
        if not tokens:
            continue

        if tokens[0] == "MACRO":
            inside_macro = True
            macro_name = tokens[1] if len(tokens) > 1 else None

            if macro_name is not None:
                mnt[macro_name] = {"mdt_index": mdtc, "params": tokens[2:]}  # Collect params
                mdt.append(line)
                mdtc += 1
            else:
                print("Error: Macro name is missing.")

        elif tokens[0] == "MEND":
            inside_macro = False
            mdt.append(line)
            mdtc += 1
        elif inside_macro:
            mdt.append(line)
            mdtc += 1
        else:
            pass

    return mnt, mdt
```

```
def pass_two(source_code, mnt, mdt):

    expanded_code = []

    for line in source_code:
        if line.startswith("MACRO"):
            continue

        macro_call = re.match(r"(\w+)\s*(.*)", line)
        if macro_call:
            macro_name = macro_call.group(1)
            if macro_name in mnt:
                macro_args = [arg.strip() for arg in macro_call.group(2).split(",")]
                param_names = mnt[macro_name]["params"]
                mdt_index = mnt[macro_name]["mdt_index"]

                for i in range(mdt_index + 1, len(mdt)):
                    if mdt[i].startswith("MEND"):
                        break
                    expanded_line = mdt[i]
                    for param, arg in zip(param_names, macro_args):
                        expanded_line = expanded_line.replace(param, arg)
                    expanded_code.append(expanded_line)
            else:
                expanded_code.append(line)
        else:
            expanded_code.append(line)

    return expanded_code

source_code = [
    "MACRO ADD &X, &Y",  # Macro definition with name
    "LOAD &X",
    "ADD &Y",
    "STORE &X",
    "MEND",
    "START",
    "ADD A, B",  # Macro call
```

```
    "END"
]

mnt, mdt = pass_one(source_code)
expanded_code = pass_two(source_code, mnt, mdt)
print("MNT:", mnt)
print("MDT:", mdt)
print("Expanded Code:", expanded_code)
```

**Sample Input and Output:**

MNT: {'ADD': {'mdt_index': 0, 'params': ['&X,', '&Y']}}

MDT: ['MACRO ADD &X, &Y', 'LOAD &X', 'ADD &Y', 'STORE &X', 'MEND']

Expanded Code: ['LOAD &X', 'LOAD &X', 'ADD &Y', 'STORE &X', 'STORE &X',

'MEND', 'START', 'LOAD &X', 'ADD B', 'STORE &X', 'END']

**RESULT:**

**Thus the program for implementation of PASS ONE & TWO of the TWO PASS Macro Processor is executed and verified.**

**Ex. No: 5**

**Date:**

## Implementation of Lexical Analyzer for 'if' Statement

**Aim:**

To write a Python program to implement lexical analyzer for 'if' statement.

**Algorithm:**

1. Read an input string from the user.
2. Define token patterns using regular expressions, token types, and addresses.
3. Compile the token patterns into regular expressions.
4. Initialize position to start at the beginning of the string and create an empty list to store tokens.
5. Loop through the input string, checking each position for matching token patterns.
6. Match a token pattern at the current position using the compiled regular expressions.
7. Add the token and its address to the token list if a match is found.
8. Advance the position based on the length of the matched token.
9. Handle errors by raising a syntax error if no match is found at any position.
10. Return the list of tokens with their associated addresses once the entire string is processed.

**Program:**

```
import re

token_patterns = [
   (r'\s+', None, None),  # Whitespace (ignored)
   (r'if', 'IF', '<1,1>'),
   (r'[a-zA-Z_][a-zA-Z_0-9]*', 'VARIABLE', '<2,#address>'),
   (r'[0-9]+', 'NUMBER', '<3,#address>'),
   (r';', 'SEMICOLON', '<4,4>'),
   (r'\(', 'LPAREN', '<5,0>'),
   (r'\)', 'RPAREN', '<5,1>'),
   (r'\{', 'LBRACE', '<6,0>'),
   (r'\}', 'RBRACE', '<6,1>'),
   (r'>=', 'GTE', '<620,620>'),
   (r'>', 'GT', '<62,62>'),
   (r'<=', 'LTE', '<600,600>'),
```

```
    (r'<', 'LT', '<60,60>'),
    (r'!=', 'NEQ', '<330,330>'),
    (r'!', 'NOT', '<33,33>'),
    (r'==', 'EQ', '<610,610>'),
    (r'=', 'ASSIGN', '<61,61>'),
]

token_regex = [(re.compile(pattern), token_type, address) for pattern, token_type,
address in token_patterns]

def lex(input_string):
    position = 0
    tokens = []

    while position < len(input_string):
        match = None
        for token_re, token_type, address in token_regex:
            match = token_re.match(input_string, position)
            if match:
                if token_type:  # If not a whitespace
                    token = match.group(0)
                    if token_type == 'VARIABLE':
                        address = address.replace('#address', hex(id(token)))
                    elif token_type == 'NUMBER':
                        address = address.replace('#address', token)
                    tokens.append((token_type, token, address))
                break
        if not match:
            raise SyntaxError(f"Illegal character: {input_string[position]}")
        else:
            position = match.end(0)

    return tokens

input_string = input("enter the input string")

tokens = lex(input_string)
for token in tokens:
    token_type, value, address = token
    print(f"{value} {address}")
```

**Sample Input & Output:**

Enter the input string: if(a>=b) max=a;

| | |
|---|---|
| if | <1,1> |
| ( | <5,0> |
| a | <2,0960> |
| >= | <620,620> |
| b | <2,09c4> |
| ) | <5,1> |
| max | <2,0A28> |
| = | <61,61> |
| a | <2,0A8c> |
| ; | <4,4> |

**Result:**

The above Python program was successfully executed and verified.

**Ex. No: 6**

**Date:**

## Implementation of Lexical Analyzer for Arithmetic Expression

**Aim:**

To write a Python program to implement lexical analyzer for Arithmetic Expression.

**Algorithm:**

1. Define token patterns: Specify token types (e.g., `VARIABLE`, `NUMBER`, `SEMICOLON`) with associated regex patterns and address placeholders.
2. Compile the token patterns into regex objects to prepare for efficient matching.
3. Initialize position and tokens list: Start reading from the beginning of the `input_string` and create an empty list `tokens` to store results.
4. Loop through the input string while the position is within the string's length.
5. Match tokens: For each position, try to match the input string with the token regex patterns. If a match occurs:
   - If the token is a `VARIABLE` or 'CONSTANT' , replace the address placeholder with the memory address of the token.
6. Store the token: Append the token type, matched value, and address to the `tokens` list.
7. Handle unmatched input: If no pattern matches the input, raise a syntax error indicating an illegal character.
8. Move to the next part: Update the `position` to the end of the matched token and continue.
9. Return tokens: After processing the entire string, return the list of tokens with their values and addresses.
10. Display tokens: Iterate through the tokens and print their values and addresses.

**Program:**

```
import re

token_patterns = [
   (r'\s+', None, None),  # Whitespace (ignored)
   (r'[a-zA-Z_][a-zA-Z_0-9]*', 'VARIABLE', '<1,#address>'),
   (r'[0-9]+', 'NUMBER', '<2,#address>'),
   (r';', 'SEMICOLON', '<3,3>'),
   (r'=', 'ASSIGN', '<4,4>'),
   (r'\+', 'PLUS', '<43,43>'),
   (r'\+=', 'PLUSEQ', '<430,430>'),
   (r'-', 'MINUS', '<45,45>'),
   (r'-=', 'MINUSEQ', '<450,450>'),
```

```
    (r'\*', 'MULT', '<42,42>'),
    (r'\*=', 'MULTEQ', '<420,420>'),
    (r'/', 'DIV', '<47,47>'),
    (r'/=', 'DIVEQ', '<470,470>'),
    (r'%', 'MOD', '<37,37>'),
    (r'%=', 'MODEQ', '<370,370>'),
    (r'\^', 'XOR', '<94,94>'),
    (r'\^=', 'XOREQ', '<940,940>')
]

token_regex = [(re.compile(pattern), token_type, address) for pattern, token_type,
address in token_patterns]

def lex(input_string):
    position = 0
    tokens = []

    while position < len(input_string):
        match = None
        for token_re, token_type, address in token_regex:
            match = token_re.match(input_string, position)
            if match:
                if token_type:  # If not a whitespace
                    token = match.group(0)
                    if token_type == 'VARIABLE':
                        address = address.replace('#address', hex(id(token)))
                    elif token_type == 'NUMBER':
                        address = address.replace('#address', token)
                    tokens.append((token_type, token, address))
                break
        if not match:
            raise SyntaxError(f"Illegal character: {input_string[position]}")
        else:
            position = match.end(0)

    return tokens


input_string = "x = 10 + y;"
tokens = lex(input_string)
for token in tokens:
```

```
token_type, value, address = token
print(f"{value} {address}")
```

**Sample Input & Output:**

Enter the Input String: a=a*2+b/c;

| | |
|---|---|
| a | <1,08CE> |
| = | <4,4> |
| a | <1,08CE> |
| * | <42,42> |
| 2 | <2,04E9> |
| + | <43,43> |
| b | <1,0932> |
| / | <47,47> |
| c | <1,0996> |
| ; | <3,3> |

**Result:**

The above Python program was successfully executed and verified.

**Ex. No: 7**

**Date:**

# Implementation of Lexical Analyzer using Lex Tool

**Aim:**

To write a Python program to implement Lexical Analyzer using Lex Tool.

**Algorithm:**

1. Define a function to check if a character is alphanumeric or a specific special character.
2. Read the input file to obtain its content.
3. Process the content:
   For each character in the content:
   a. If it's alphanumeric or a special character, write it to an intermediate file.
   b. If it's a newline, write a specific marker to the intermediate file.
   c. For any other character, write it with a specific format to the intermediate file.
4. Read operator and keyword files to create a mapping of operators and a set of keywords.
5. Initialize a line counter for tracking current line numbers during analysis.
6. Perform lexical analysis on the intermediate file:
   a. Split each line into tokens.
   b. For each token:
      I. Print the current line number when encountering a specific marker.
      II. Check if the token is an operator, keyword, constant, or identifier, and print its type accordingly .
7. Complete the analysis after processing all lines and tokens.

**Program:**

```
import re

def is_alpha_or_digit_or_special(c):
    return c.isalnum() or c in ['[', ']', '.']

def process_file(input_file, intermediate_file, oper_file, key_file):
    with open(input_file, "r") as fi:
        content = fi.read()
    with open(intermediate_file, "w") as fo:
        for c in content:
            if is_alpha_or_digit_or_special(c):
                fo.write(c)
            else:
```

```
            if c == '\n':
                fo.write("\n\t$\t")
            else:
                fo.write(f"\n\t{c}\t")

with open(intermediate_file, "r") as fi, \
     open(oper_file, "r") as fop, \
     open(key_file, "r") as fk:

    fop_dict = {}
    for line in fop:
        parts = line.split()
        if len(parts) == 2:
            fop_dict[parts[0]] = parts[1]

    fk_set = set()
    for line in fk:
        parts = line.split()
        if len(parts) > 0:
            fk_set.add(parts[0])

    print("\n Lexical Analysis")
    i = 1
    print(f"\n Line: {i}\n")
    i += 1

    tokenlist = []
    for line in fi:
        tokens = re.split(r'\s+', line.strip())
        tokenlist.append(tokens)
        for token in tokens:
            if token == "$":
                print(f"\n Line: {i}\n")
                i += 1
            elif token in fop_dict:
                print(f"\t{token} : {fop_dict[token]}")
            elif token in fk_set:
                print(f"\t{token} : Keyword")
            else:
                if token.isdigit():
                    print(f"\t{token} : Constant")
```

```
            else:
                print(f"\t{token} : Identifier")

input_file = "input.c"
intermediate_file = "inter-py.c"
oper_file = "oper.c"
key_file = "key.c"
process_file(input_file, intermediate_file, oper_file, key_file)
```

## Sample Input and Output:

## Input:

**Key.C:**

int
void
main
char
if
for
while
else
printf
scanf
FILE
include
stdio.h
conio.h
iostream.h

**Oper.C:**

( open para )
closepara {
openbrace }
closebrace <
lesser
> greater
" doublequote
' singlequote
: colon
; semicolon
# preprocessor
= equal
== asign

% percentage
^ bitwise
& reference
* star
+ add
- sub
\ backslash
/ slash

**Input.C:**

```
#include  "stdio.h"
#include "conio.h"
void main()
{
int a=10,b,c;
a=b*c;
getch();
}
```

**Output**

Enter the File Name: Input.C

Line: 1
# : Preprocessor
include : keyword
< : lesser
stdio.h : keyword
> : greater

Line: 2
# : Preprocessor
include : keyword
< : lesser
conio.h : keyword
> : greater

Line: 3
void :  keyword

main : keyword
( : openpara
) : closepara

Line: 4
{ : openbrace

Line: 5
int : keyword
a : identifier =
: equal
10  : constant
,  :  identifier
b : identifier

,  : identifier c
: identifier  ; :
semicolon

Line: 6
a : identifier
= : equal
b : identifier
* : star
c : identifier ;
: semicolon

Line: 8
} : closebrace

**Result:**

The above Python program was successfully executed and verified.

**Ex. No: 8**

**Date:**

# Construction of NFA from Regular Expression

**Aim:**

To write a C program to construct a Non Deterministic Finite Automata (NFA) from Regular Expression.

**Algorithm:**

1. Start the Program.
2. Enter the regular expression R over alphabet E.
3. Decompose the regular expression R into its primitive components
4. For each component construct finite automata.
5. To construct components for the basic regular expression way that corresponding to that way compound regular expression.
6. Stop the Program.

**Program:**

```
import matplotlib.pyplot as plt
import matplotlib.patches as patches

def draw_star(ax, x1, y1, x2, y2):
    width = x2 - x1
    ellipse = patches.Ellipse(((x1 + x2) / 2, y2 - 10), width, 70, angle=0, color='black',
fill=False)
    ax.add_patch(ellipse)
    ax.text(x1 - 2, y2 - 17, 'v', fontsize=12)
    ax.plot([x2 + 10, x2 + 30], [y2, y2], 'k-')
    ax.text(x1 - 15, y1 - 3, '>', fontsize=12)
    ax.add_patch(patches.Circle((x1 - 40, y1), 10, color='black', fill=False))
    ax.add_patch(patches.Circle((x1 - 80, y1), 10, color='black', fill=False))
    ax.plot([x1 - 30, x1 - 10], [y2, y2], 'k-')
    ax.text(x2 + 25, y2 - 3, '>', fontsize=12)
    ax.text(x2 + 15, y2 - 9, chr(238), fontsize=12)
    ax.text(x1 - 25, y1 - 9, chr(238), fontsize=12)
    ax.text((x2 - x1) / 2 + x1, y1 - 30, chr(238), fontsize=12)
    ax.text((x2 - x1) / 2 + x1, y1 + 30, chr(238), fontsize=12)
```

```
    ax.add_patch(patches.Ellipse(((x1 + x2) / 2, y2 + 10), width + 80, 70, angle=180,
color='black', fill=False))
    ax.text(x2 + 37, y2 + 14, '^', fontsize=12)
    return x1 - 40, y1

def draw_basis(ax, x1, y1, label):
    ax.add_patch(patches.Circle((x1, y1), 10, color='black', fill=False))
    ax.plot([x1 + 30, x1 + 10], [y1, y1], 'k-')
    ax.text(x1 + 20, y1 - 10, label, fontsize=12)
    ax.text(x1 + 23, y1 - 3, '>', fontsize=12)

def draw_slash(ax, x1, y1, x2, y2, x3, y3, x4, y4):
    c1 = max(x1, x3)
    c2 = max(x2, x4)
    ax.plot([x1 - 10, c1 - 40], [y1, (y3 - y1) / 2 + y1 - 10], 'k-')
    ax.text(x1 - 15, y1 - 3, '>', fontsize=12)
    ax.text(x3 - 15, y4 - 3, '>', fontsize=12)
    ax.add_patch(patches.Circle((c1 - 40, (y4 - y2) / 2 + y2), 10, color='black', fill=False))
    ax.text(c1 - 40, (y4 - y2) / 2 + y2 + 25, chr(238), fontsize=12)
    ax.text(c1 - 40, (y4 - y2) / 2 + y2 - 25, chr(238), fontsize=12)
    ax.plot([x2 + 10, c2 + 40], [y2, (y4 - y2) / 2 + y2 - 10], 'k-')
    ax.plot([x3 - 10, c1 - 40], [y3, (y3 - y1) / 2 + y2 + 10], 'k-')
    ax.add_patch(patches.Circle((c2 + 40, (y4 - y2) / 2 + y2), 10, color='black', fill=False))
    ax.text(c2 + 40, (y4 - y2) / 2 + y2 - 25, chr(238), fontsize=12)
    ax.text(c2 - 40, (y4 - y2) / 2 + y2 + 25, chr(238), fontsize=12)
    ax.text(c2 + 35, (y4 - y2) / 2 + y2 - 15, '^', fontsize=12)
    ax.text(c1 + 35, (y4 - y2) / 2 + y2 + 10, '^', fontsize=12)

def draw_nfa(regex):
    x1, y1 = 200, 200
    stx, endx, sty, endy = [], [], [], []
    pos = 0
    i = 0
    len_regex = len(regex)

    fig, ax = plt.subplots()
    ax.set_aspect('equal')
    ax.axis('off')

    while i < len_regex:
        if regex[i].isalpha():
```

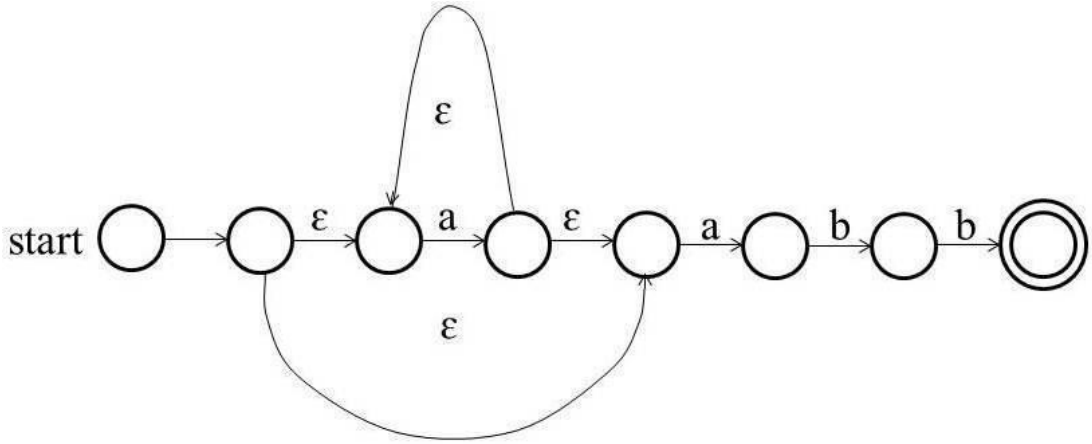
```
            if i + 1 < len_regex and regex[i + 1] == '*':
                x1 += 40
            draw_basis(ax, x1, y1, regex[i])
            stx.append(x1)
            endx.append(x1 + 40)
            sty.append(y1)
            endy.append(y1)
            x1 += 40
            pos += 1
        elif regex[i] == '*':
            x1, y1 = draw_star(ax, stx[pos - 1], sty[pos - 1], endx[pos - 1], endy[pos - 1])
            x1 += 40
        elif regex[i] == '/':
            cx1, cy1 = stx[pos - 1], sty[pos - 1]
            cx2, cy2 = endx[pos - 1], endy[pos - 1]
            i += 1
            while i < len_regex and regex[i] in ' ()':
                i += 1
            cx3, cy3 = stx[pos - 1], sty[pos - 1]
            cx4, cy4 = endx[pos - 1], endy[pos - 1]
            draw_slash(ax, cx1, cy1, cx2, cy2, cx3, cy3, cx4, cy4)
            x1 += 40
        i += 1

    ax.add_patch(patches.Circle((x1, y1), 13, color='black', fill=False))
    ax.plot([x1 - 30, x1 - 10], [y1, y1], 'k-')
    ax.text(x1 - 100, y1 - 10, 'start', fontsize=12)
    ax.text(x1 - 15, y1 - 3, '>', fontsize=12)

    plt.show()

# Example usage
regex = input("Enter the regular expression: ")
draw_nfa(regex)
```

**Sample Input & Output:**




**Result:**

The above Python program was successfully executed and verified.

**Ex.No: 9**

**Date:**

## Implementation of Shift Reduce Parsing Algorithm

**Aim:**

To write a C program to implement the shift-reduce parsing algorithm.

**Algorithm:**

**Grammar:**

E->E+E

E->E*E

E->E/E

E->a/b

**Method:**

| Stack | Input Symbol | Action |
|---|---|---|
| $ | id1*id2$ | shift |
| $id1 | *id2 $ | shift * |
| $* | id2$ | shift id2 |
| $id2 | $ | shift |
| $ | $ | accept |

Shift: Shifts the next input symbol onto the stack.

Reduce: Right end of the string to be reduced must be at the top of the stack.

Accept: Announce successful completion of parsing.

Error: Discovers a syntax error and call an error recovery routine.

**Program:**

```
class ProductionRule:
    def __init__(self, left, right):
        self.left = left
        self.right = right

rules = [
    ProductionRule('E', 'E+E'),
    ProductionRule('E', 'E/E'),
    ProductionRule('E', 'E*E'),
    ProductionRule('E', 'a'),
    ProductionRule('E', 'b')
]
stack = ""
input_string = input("\nEnter the input string: ").strip()
i = 0
while True:

    if i < len(input_string):
        ch = input_string[i]
        i += 1
        stack += ch
        print(f"{stack}\t{input_string[i:]}\tShift {ch}")
    reduction_made = False
    for rule in rules:
        while rule.right in stack:
            stack = stack.replace(rule.right, rule.left, 1)
            print(f"{stack}\t{input_string[i:]}\tReduce {rule.left}->{rule.right}")
            reduction_made = True
    if not reduction_made:
        if stack == rules[0].left and i == len(input_string):
            print("\nAccepted")
            break
        if i == len(input_string):
            print("\nNot Accepted")
            break
```

**Sample Input & Output:**

Enter the input string: a*a+b

```
a      *a+b    Shift a
E      *a+b    Reduce E->a
E*     a+b     Shift *
E*a    +b      Shift a
E*E    +b      Reduce E->a
E*E+   b       Shift +
E+     b       Reduce E->E*E
E+b            Shift b
E+E            Reduce E->b
E              Reduce E->E+E
```

Accepted

**Result:**

The above Python program was successfully executed and verified.

**Ex.No: 10**

**Date:**

## Implementation of Operator Precedence Parser

**Aim:**

To write a Python program to implement Operator Precedence Parser.

**Algorithm:**

Input: String of terminals from the operator grammar

Output: Sequence of shift reduce step1

1. Define a class to represent production rules with a left side and a right side.

2. Create a list of production rules for the grammar that includes operations like addition, subtraction, multiplication, division, and parentheses.

3. Implement a function to apply reductions based on the production rules by replacing the right side with the left side in the current stack.

4. Create a function to parse the input expression by initializing an empty stack and adding an end-of-input marker.

5. Process the expression character by character, adding each character to the stack and printing the current state.

6. Check for possible reductions by applying the production rules to the stack until no further reductions can be made.

7. Determine acceptance by checking if the final stack is equal to the start symbol and if the end marker follows the last character.

8. Output the result as "Accepted" or "Rejected" based on the final state of the stack.

**Program:**

```
class ProductionRule:
    def __init__(self, left, right):
        self.left = left
        self.right = right

rules = [
    ProductionRule('E', 'E+E'),
    ProductionRule('E', 'E-E'),
    ProductionRule('E', 'E*E'),
    ProductionRule('E', 'E/E'),
    ProductionRule('E', 'E^E'),
    ProductionRule('E', '(E)'),
    ProductionRule('E', 'i')
]

def apply_reduction(stack, rule):
    pos = stack.find(rule.right)
    if pos != -1:
        return stack[:pos] + rule.left + stack[pos + len(rule.right):]
    return stack

def parse_expression(expression):
    stack = ""
    expression += '$'
    i = 0
    while True:
        if i < len(expression):
            stack += expression[i]
            print(f"{stack}\t{expression[i+1:]}\tShift {expression[i]}")
            i += 1

        reduction_made = False
```

```
        for rule in reversed(rules):
            new_stack = apply_reduction(stack, rule)
            while new_stack != stack:
                stack = new_stack
                print(f"{stack}\t{expression[i:]}\tReduce {rule.left}->{rule.right}")
                reduction_made = True
                new_stack = apply_reduction(stack, rule)

        if i == len(expression) - 1:
            if stack == 'E' and expression[i] == '$':
                print("\nAccepted")
                return
            else:
                print("\nRejected")
                return

input_string = input("\nEnter the Arithmetic Expression End with $: ").strip()
parse_expression(input_string)
```

**Sample Input & Output:**

Enter the Arithmetic Expression End with $: i-(i*i)

```
i       -(i*i)$ Shift i
E       -(i*i)$ Reduce E->i
E-      (i*i)$  Shift -
E-(     i*i)$   Shift (
E-(i    *i)$    Shift i
E-(E    *i)$    Reduce E->i
E-(E*   i)$     Shift *
E-(E*i  )$      Shift i
E-(E*E  )$      Reduce E->i
E-(E    )$      Reduce E->E*E
E-(E)   $       Shift )
E-E     $       Reduce E->(E)
E       $       Reduce E->E-E
```

Accepted

**Result:**

The above Python program was successfully executed and verified.

**Ex.No: 11**

**Date:**

# Implementation of Recursive Descent Parser

**Aim:**

To write a Python program to implement Recursive Descent Parser.

**Algorithm:**

Input: Context Free Grammar without last recursion and an input string from the grammar.

Output: Sequence of productions rules used to derive the sentence.

1. Set ip_ptr to 0 to track the current position in the input string.
2. Create a function that increments ip_ptr by 1.
3. Implement a function that handles the E non-terminal, calling t and e_prime.
4. Create a function that checks for a '+' and calls t and itself recursively if found; otherwise, produces ε.
5. Implement a function that handles the T non-terminal, calling f and t_prime.
6. Create a function that checks for '*' and calls f and itself recursively if found; otherwise, produces ε.
7. Implement a function that checks if the current symbol is an identifier or a '('. If it is an identifier, process it; if it's '(', parse the expression and check for a matching ')'.
8. Check for invalid symbols and print an error message if encountered.
9. Print the grammar rules before accepting input.
10. Prompt the user to enter the input symbol.
11. Call the e function to begin the parsing process.

**Program:**

```
ip_ptr = 0

def advance():
    global ip_ptr
    ip_ptr += 1

def e():
    print("\n\tE  -> T E'")
    t()
    e_prime()

def e_prime():
    if ip_ptr < len(input_string) and input_string[ip_ptr] == '+':
        print("\n\tE' -> + T E'")
        advance()
        t()
        e_prime()
    else:
        print("\n\tE' -> ε")

def t():
    print("\n\tT  -> F T'")
    f()
    t_prime()

def t_prime():
    if ip_ptr < len(input_string) and input_string[ip_ptr] == '*':
        print("\n\tT' -> * F T'")
        advance()
        f()
        t_prime()
    else:
        print("\n\tT' -> ε")

def f():
    if ip_ptr < len(input_string) and input_string[ip_ptr] in ('i', 'j'):
        print("\n\tF  -> id")
        advance()
```

```
    elif ip_ptr < len(input_string) and input_string[ip_ptr] == '(':
        advance()
        e()
        if ip_ptr < len(input_string) and input_string[ip_ptr] == ')':
            advance()
            print("\n\tF  -> (E)")
        else:
            print("\n\tSyntax Error: Missing ')'")
            exit(1)
    else:
        print("\n\tSyntax Error: Invalid symbol in F")
        exit(1)

print("\n\tGRAMMAR WITHOUT RECURSION")
print("\n\tE  -> T E'\n\tE' -> + T E'/ε\n\tT  -> F T'\n\tT' -> * F T'/ε\n\tF  -> (E)/id")
input_string = input("\n\tEnter the Input Symbol: ")

print("\n\tSequence of Production Rules")
e()
```

**Sample Input & Output:**

GRAMMAR WITHOUT RECURSION

E -> T E'
E' -> + T E'/ε
T -> F T'
T' -> * F T'/ε
F -> (E)/id

Enter the Input Symbol: i*j

Sequence of Production Rules

E -> T E'

T -> F T'

F -> id

T' -> * F T'

F -> id

T' -> ε

E' -> ε

**Result:**

The above Python program was successfully executed and verified.

**Ex.No: 12**

**Date:**

# Implementation of Code Optimization Techniques

**Aim:**

To write a Python program to implement Code Optimization Techniques.

**Algorithm:**

Input: Set of „L‟ values with corresponding „R‟ values.

Output: Intermediate code & Optimized code after eliminating common expressions.

1. Identify Dead Code:
    i. Check for operations that produce values that are never used.
    ii. Remove these operations to simplify the code.

2. Gather all operations in a structured format for analysis.

3. Look for expressions that are repeated across different operations.

4. Replace Duplicates:
    1. For each duplicate expression found, replace it with a single instance.
    2. Update any references in the remaining operations to point to this instance.

5. Create a new list to store operations after dead code and duplicate eliminations.

6. Print the final list of operations that are free from duplicates and dead code.

**Program:**

```
class Operation:
    def __init__(self, left, right):
        self.left = left
        self.right = right

# Input number of operations
n = int(input("Enter the Number of Values: "))

# Read operations
ops = [Operation(input("left: "), input("\tright: ")) for _ in range(n)]

# Print Intermediate Code
print("\nIntermediate Code")
for op in ops:
    print(f"{op.left} = {op.right}")

# Dead Code Elimination
pr = []
for i in range(n - 1):
    temp = ops[i].left
    if any(temp in op.right for op in ops):
        pr.append(Operation(ops[i].left, ops[i].right))

pr.append(Operation(ops[n - 1].left, ops[n - 1].right))

# Print after Dead Code Elimination
print("\nAfter Dead Code Elimination\n")
for op in pr:
    print(f"{op.left}\t= {op.right}")

# Common Subexpression Elimination
for m in range(len(pr)):
    tem = pr[m].right
    for j in range(m + 1, len(pr)):
        if pr[j].right in tem:
            t = pr[j].left
            pr[j].left = pr[m].left
            for i in range(len(pr)):
```

```
            if t in pr[i].right:
                a = pr[i].right.index(t)
                pr[i].right = pr[i].right[:a] + pr[m].left + pr[i].right[a + 1:]

# Print after Common Subexpression Elimination
print("\nEliminate Common Expression")
for op in pr:
    print(f"{op.left}\t= {op.right}")

optimized_pr = []
for i in range(len(pr)):
    if any(op.left == pr[i].left and op.right == pr[i].right for op in optimized_pr):
        continue
    optimized_pr.append(pr[i])

# Print Optimized Code
print("\nOptimized Code")
for op in optimized_pr:
    print(f"{op.left} = {op.right}")
```

**Sample Input & Output:**

```
Enter the Number of Values: 5
Left:  a        right:  9
Left:  b        right:  c+d
Left:  e        right:  c+d
Left:  f        right:  b+e
Left:  r        right:  f

Intermediate Code
a=9
b=c+d
e=c+d
f=b+e
r=:f

After Dead Code Elimination
b =c+d
e    =c+d
f    =b+e
r    =:f

Eliminate Common Expression
b    =c+d
b    =c+d
f    =b+b
r    =:f

Optimized Code
b=c+d
f=b+b
r=:f
```

**Result:**

The above Python program was successfully executed and verified.

**Ex.No: 13**

**Date:**

# Implementation of Code Generator

**Aim:**

To write a Python program to implement Simple Code Generator.

**Algorithm:**

Input: Set of three address code sequence.

Output: Assembly code sequence

a. Create a set of registers to store variables and mark them as available for use.

b. Continuously read each line of input representing a mathematical operation.

c. For each line, identify the variables, operator, and the result of the operation.

d. Register Assignment:
   i. Check if any register is free for each variable in the operation.
   ii. If a variable doesn't have a register assigned, allocate one to it.

e. For each variable, generate a command to move the variable's value into the assigned register.

f. Based on the operator (e.g., `+`, `-`, `*`, `/`), generate the corresponding command to perform the operation on the variables in the registers.

g. Assign the result of the operation to a register and move the result to the register if necessary.

h. Once an operation is complete, mark the registers used by the variables as free, so they can be reused in subsequent operations.

**Program:**

```
import re

class Register:
    def __init__(self):
        self.var = ""
        self.alive = False

def get_register(var, registers):
    for i in range(10):
        if not registers[i].alive:
            registers[i].var = var
            registers[i].alive = True
            return i
    return -1

def get_var(exp):
    match = re.match(r"([a-zA-Z]+)", exp)
    return match.group(0) if match else ""

def process_code(lines):
    registers = [Register() for _ in range(10)]

    for line in lines:
        line = re.sub(r'\s*([+\-*/])\s*', r' \1 ', line)
        parts = line.split()

        if len(parts) < 5:
            continue

        var1 = get_var(parts[2])
        operator = parts[3]
        var2 = get_var(parts[4])
        result = parts[0]

        reg1 = get_register(var1, registers)
        reg2 = get_register(var2, registers)
        reg_result = get_register(result, registers)

        if reg1 != -1:
            print(f"Mov R{reg1}, {var1}")
```

```
        if reg2 != -1:
            print(f"Mov R{reg2}, {var2}")

        if operator == '+':
            print(f"Add R{reg1}, R{reg2}")
        elif operator == '-':
            print(f"Sub R{reg1}, R{reg2}")
        elif operator == '*':
            print(f"Mul R{reg1}, R{reg2}")
        elif operator == '/':
            print(f"Div R{reg1}, R{reg2}")

        if reg_result != -1:
            print(f"Mov R{reg_result}, R{reg1}")

        if reg1 != -1:
            registers[reg1].alive = False
        if reg2 != -1:
            registers[reg2].alive = False

lines = []
line = ''
while line != 'exit':
    line = input()
    lines.append(line)

process_code(lines)
```

**Sample Input & Output:**

Enter the Three Address Code:

a=b+c
c=a*c
exit

The Equivalent Assembly Code is:

```
Mov R0,b
Add  c,R0
Mov a,R0
Mov R1,a
Mul  c,R1
Mov c,R1
```

**Result:**

The above Python program was successfully executed and verified.

**Ex.No: 14**

**Date:**

## IMPLEMENTATION OF SYMBOL TABLE

**AIM:**

To write a program for implementing Symbol Table using Python.

**ALGORITHM:**

1. Start the program for performing insert, display, delete, search and modify option in symbol table
2. Define the structure of the Symbol Table
3. Enter the choice for performing the operations in the symbol Table
4. If the entered choice is 1, search the symbol table for the symbol to be inserted. If the symbol is already present, it displays "Duplicate Symbol". Else, insert the symbol and the corresponding address in the symbol table.
5. If the entered choice is 2, the symbols present in the symbol table are displayed.
6. If the entered choice is 3, the symbol to be deleted is searched in the symbol table.
7. If it is not found in the symbol table it displays "Label Not found". Else, the symbol is deleted.
8. If the entered choice is 5, the symbol to be modified is searched in the symbol table. The label or address or both can be modified.

**PROGRAM:**

```
class SymbolTableNode:
    def __init__(self, label, addr):
        self.label = label
        self.addr = addr
        self.next = None

class SymbolTable:
    def __init__(self):
        self.first = None
        self.last = Non    e
        self.size = 0

    def insert(self):
        label = input("Enter the label: ")
        if self.search(label):
            print("The label already exists. Duplicate cannot be inserted")
        else:
            addr = int(input("Enter the address: "))
            new_node = SymbolTableNode(label, addr)
            if self.size == 0:
                self.first = new_node
                self.last = new_node
            else:
                self.last.next = new_node
                self.last = new_node
            self.size += 1

    def display(self):
        print("LABEL\tADDRESS")
        current = self.first
        while current is not None:
            print(f"{current.label}\t{current.addr}")
            current = current.next

    def search(self, label):
        current = self.first
        while current is not None:
            if current.label == label:
                return True
            current = current.next
```

```
        return False

    def modify(self):
        choice = int(input("What do you want to modify?\n1. Only the label\n2. Only the
address of a particular label\n3. Both the label and address\nEnter your choice: "))
        label = input("Enter the old label: ")

        if not self.search(label):
            print("No such label")
            return

        if choice == 1:
            new_label = input("Enter the new label: ")
            current = self.first
            while current is not None:
                if current.label == label:
                    current.label = new_label
                    break
                current = current.next
        elif choice == 2:
            new_addr = int(input("Enter the new address: "))
            current = self.first
            while current is not None:
                if current.label == label:
                    current.addr = new_addr
                    break
                current = current.next
        elif choice == 3:
            new_label = input("Enter the new label: ")
            new_addr = int(input("Enter the new address: "))
            current = self.first
            while current is not None:
                if current.label == label:
                    current.label = new_label
                    current.addr = new_addr
                    break
                current = current.next

    def delete(self):
        label = input("Enter the label to be deleted: ")
```

```
        if not self.search(label):
            print("Label not found")
            return

        if self.first.label == label:
            self.first = self.first.next
            if self.size == 1:
                self.last = None
        else:
            prev = None
            current = self.first
            while current is not None:
                if current.label == label:
                    if self.last == current:
                        self.last = prev
                    if prev is not None:
                        prev.next = current.next
                    break
                prev = current
                current = current.next

        self.size -= 1

table = SymbolTable()
while True:
    print("\nSYMBOL TABLE IMPLEMENTATION")
    print("1. INSERT")
    print("2. DISPLAY")
    print("3. DELETE")
    print("4. SEARCH")
    print("5. MODIFY")
    print("6. END")
    op = int(input("Enter your option: "))
    if op == 1:
        table.insert()
        table.display()
    elif op == 2:
        table.display()
    elif op == 3:
        table.delete()
        table.display()
```

```
elif op == 4:
    label = input("Enter the label to be searched: ")
    if table.search(label):
        print("The label is already in the symbol table")
    else:
        print("The label is not found in the symbol table")
elif op == 5:
    table.modify()
    table.display()
elif op == 6:
    break
```

**OUTPUT:**

SYMBOL TABLE IMPLEMENTATION

1.INSERT

2.DISPLAY

3.DELETE

4.SEARCH

5.MODIFY

6.END

Enter your option:1

Enter the label: A

Enter the address: 10

LABEL ADDRESS

A 10

SYMBOL TABLE IMPLEMENTATION

1.INSERT

2.DISPLAY

3.DELETE

4.SEARCH

5.MODIFY

6.END

Enter your option:1

Enter the label: B

Enter the address: 11

LABEL ADDRESS
A 10

B 11

SYMBOL TABLE IMPLEMENTATION

1.INSERT

2.DISPLAY

3.DELETE

4.SEARCH

5.MODIFY

6.END

Enter your option:2

LABEL ADDRESS
A 10

B 11

SYMBOL TABLE IMPLEMENTATION

1.INSERT

2.DISPLAY

3.DELETE

4.SEARCH

5.MODIFY

6.END

Enter your option:3

Enter the label to be deleted: A LABEL

ADDRESS

B 11

SYMBOL TABLE IMPLEMENTATION

1.INSERT

2.DISPLAY

3.DELETE

4.SEARCH

5.MODIFY

6.END

Enter your option:4

Enter the label to be searched: A

The label is not found in the symbol table.

SYMBOL TABLE IMPLEMENTATION

1.INSERT

2.DISPLAY

3.DELETE

4.SEARCH

5.MODIFY

6.END

Enter your option:4

Enter the label to be searched: B

The label is already in the symbol table.

SYMBOL TABLE IMPLEMENTATION

1.INSERT

2.DISPLAY

3.DELETE

4.SEARCH

5.MODIFY

6.END

Enter your option:5

what do you want to modify?

1.only the label

2.only the address of a particular label

3.both the label and address

Enter your choice:1

Enter the old label

B

Enter the new label

A

| LABEL | ADDRESS |
|---|---|
| a | 11 |

SYMBOL TABLE IMPLEMENTATION

1.INSERT

2.DISPLAY

3.DELE

TE

4.SEAR

CH

5.MODI

FY

6.END

Enter your option:6

**RESULT:**

Thus the program for implementation of Symbol Table is executed and verified