Practical 3

Aim :

Practical Definition

Implementation of a Lexical Analyzer for C Language Compiler

Objective

To design and implement a lexical analyser, the first phase of a compiler, for the C programming language. The lexical analyser should perform the following tasks: (1) tokenizing the input string (2) removing comments (3) removing white spaces (4) entering identifiers into the symbol table (5) generating lexical errors.

Language Constraint

The program can be implemented in any programming language

Input requirement

- Accept a C source code file.
- The input can contain keywords, identifiers, constants, strings, punctuation, operators, comments, and white spaces.

Expected output

- Tokenized output categorizing tokens into six types: keyword, identifier, constant, string, punctuation, and operator.
- Symbol table with all identified identifiers stored.
- Detection and reporting of lexical errors
- Modified source code

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>


#define MAX_TOKENS 1000

#define MAX_LENGTH 100
```

D23DCS140

```
const char *keywords[] = {"int", "char", "return", "void", "long", "float", "struct", "scanf", "printf"};
const int num_keywords = 9;

const char *operators[] = {"+", "-", "*", "/", "=", "==", "<", ">", "<=", ">="};
const int num_operators = 10;

const char punctuations[] = {';', ',', '(', ')', '{', '}', '[', ']'};

typedef struct { char type[20], value[MAX_LENGTH]; } Token;
Token tokens[MAX_TOKENS];
int token_count = 0;

char symbol_table[MAX_TOKENS][MAX_LENGTH];
int symbol_count = 0;

void add_token(const char *type, char *value) {
    strcpy(tokens[token_count].type, type);
    strcpy(tokens[token_count].value, value);
    token_count++;
}

int is_keyword(char *word) {
    for (int i = 0; i < num_keywords; i++) if (strcmp(word, keywords[i]) == 0) return 1;
    return 0;
}

int is_operator(char *op) {
```

```
    for (int i = 0; i < num_operators; i++) if (strcmp(op, operators[i]) == 0) return 1;
    return 0;
}

int is_punctuation(char ch) {
    for (int i = 0; i < sizeof(punctuations); i++) if (ch == punctuations[i]) return 1;
    return 0;
}

int is_identifier(char *word) {
    if (!isalpha(word[0]) && word[0] != '_') return 0;
    for (int i = 1; word[i]; i++) if (!isalnum(word[i]) && word[i] != '_') return 0;
    return 1;
}

int is_constant(char *word) {
    for (int i = 0; word[i]; i++) if (!isdigit(word[i])) return 0;
    return 1;
}

int should_ignore(char *identifier) {
    return (strcmp(identifier, "main") == 0 || strcmp(identifier, "demo") == 0 ||
strcmp(identifier, "program") == 0);
}

void add_symbol(char *identifier) {
    if (should_ignore(identifier)) return;
    for (int i = 0; i < symbol_count; i++) if (strcmp(symbol_table[i], identifier) == 0) return;
```

```
    strcpy(symbol_table[symbol_count++], identifier);
}

void tokenize(char *code) {
    char buffer[MAX_LENGTH];
    int i = 0, j = 0;

    while (code[i]) {
        if (isspace(code[i])) { i++; continue; }

        if (isalpha(code[i]) || code[i] == '_') {  // Identifier or Keyword
            j = 0;
            while (isalnum(code[i]) || code[i] == '_') buffer[j++] = code[i++];
            buffer[j] = '\0';
            if (is_keyword(buffer)) add_token("Keyword", buffer);
            else { add_token("Identifier", buffer); add_symbol(buffer); }
        }
        else if (isdigit(code[i])) {  // Constant
            j = 0;
            while (isdigit(code[i])) buffer[j++] = code[i++];
            buffer[j] = '\0';
            add_token("Constant", buffer);
        }
        else if (code[i] == '"' || code[i] == '\'') {  // String
            char quote = code[i++];
            j = 0;
            buffer[j++] = quote;
            while (code[i] != quote && code[i]) buffer[j++] = code[i++];
```

```
            buffer[j++] = quote;
            buffer[j] = '\0';
            i++;
            add_token("String", buffer);
        }
        else if (is_punctuation(code[i])) {  // Punctuation
            buffer[0] = code[i++];
            buffer[1] = '\0';
            add_token("Punctuation", buffer);
        }
        else {  // Operator
            j = 0;
            while (!isalnum(code[i]) && !isspace(code[i]) && !is_punctuation(code[i])) buffer[j++]
= code[i++];
            buffer[j] = '\0';
            if (is_operator(buffer)) add_token("Operator", buffer);
        }
    }
}

void print_results() {
    printf("TOKENS\n");
    for (int i = 0; i < token_count; i++) printf("%s: %s\n", tokens[i].type, tokens[i].value);

    printf("\nSYMBOL TABLE\n");
    for (int i = 0; i < symbol_count; i++) printf("%d) %s\n", i+1, symbol_table[i]);
}
```

D23DCS140

```
// Student structure and salary calculation part
struct student {
   int id;
   float cgpa;
   long int bs, da, hra, gs;
};

// Function prototype for 'add' function
void add(int x, int y);

int main() {
   // Part 1: Tokenizing code
   char code[] =
      "/* Demo Program */\n"
      "void main() {\n"
      "    int a = 5;\n"
      "    char b = 'x';\n"
      "    return a + b;\n"
      "}";

   tokenize(code);
   print_results();

   // Part 2: Salary calculation and student structure
   struct student s;

   // Take basic salary as input
   printf("\nEnter basic salary: ");
```

```
    scanf("%ld", &s.bs);

    // Calculate allowances
    s.da = s.bs * 0.40;
    s.hra = s.bs * 0.20;
    s.gs = s.bs + s.da + s.hra;

    // Display salary slip
    printf("\n\nBasic Salary: %ld", s.bs);
    printf("\nDA: %ld", s.da);
    printf("\nHRA: %ld", s.hra);
    printf("\nGross Salary: %ld", s.gs);

    // Initialize student data
    s.id = 10;
    s.cgpa = 8.7;

    // Display student information
    printf("\n\nStudent ID: %d", s.id);
    printf("\nCGPA: %.2f", s.cgpa);

    // Function call for addition
    int a = 10, b = 20;
    add(a, b);

    return 0;
}
```

D23DCS140

```
// Add function definition
void add(int x, int y) {
   int sum = x + y;
   printf("\nSum of %d and %d is: %d", x, y, sum);
}
```

```
TOKENS
Identifier: Demo
Identifier: Program
Keyword: void
Identifier: main
Punctuation: (
Punctuation: )
Punctuation: {
Keyword: int
Identifier: a
Operator: =
Constant: 5
Punctuation: ;
Keyword: char
Identifier: b
Operator: =
String: 'x'
Punctuation: ;
Keyword: return
Identifier: a
Operator: +
Identifier: b
Punctuation: ;
Punctuation: }
```

```
SYMBOL TABLE
1) Demo
2) Program
3) a
4) b

Enter basic salary: 100000


Basic Salary: 100000
DA: 40000
HRA: 20000
Gross Salary: 160000

Student ID: 10
CGPA: 8.70
Sum of 10 and 20 is: 30

...Program finished with exit code 0
Press ENTER to exit console.
```