

|  |  |
| --- | --- |
| Student Name | Siddhi Wani |
| SRN No | 202200906 |
| Roll No | 02 |
| Program | Computer Engg |
| Year | Third Year |
| Division | C |
| Subject | System Programming Lab |
| Assignment No | 6 |

**Statement:** Design a lexical analyzer for ‘C’ language.

# Code:

#include <stdbool.h> #include <stdio.h> #include <string.h> #include <stdlib.h> #include <ctype.h>

// Constants

#define MAX\_TOKENS 1000

#define MAX\_IDENTIFIER\_LENGTH 50

#define MAX\_FILE\_SIZE 10000

// Token types enum for better type identification typedef enum {

TOKEN\_KEYWORD, TOKEN\_IDENTIFIER, TOKEN\_INTEGER, TOKEN\_REAL, TOKEN\_OPERATOR, TOKEN\_DELIMITER, TOKEN\_INVALID

} TokenType;

// Structure to store token information typedef struct {

TokenType type; char\* lexeme; int lineNumber;

} Token;

// Structure to store symbol table entry typedef struct {

char name[MAX\_IDENTIFIER\_LENGTH]; char type[20];

int lineNumber;

} SymbolEntry;

// Structure to store source code information typedef struct { char\* buffer; int position; int currentLine;

} SourceCode;

// Global variables

SourceCode source = {NULL, 0, 1}; SymbolEntry symbolTable[MAX\_TOKENS]; int symbolCount = 0;

Token tokenTable[MAX\_TOKENS]; int tokenCount = 0;

char constantTable[MAX\_TOKENS][MAX\_IDENTIFIER\_LENGTH]; int constantCount = 0;

// Function declarations void readFile(const char\* filename); void displayMenu(); void processChoice(int choice); void displaySymbolTable(); void displayTokenTable(); void displayConstantTable(); void clearTables();

// Initialize source code buffer from file void readFile(const char\* filename) { FILE\* file = fopen(filename, "r"); if (file == NULL) { printf("Error opening file: %s\n", filename); return;

}

// Allocate buffer for file content source.buffer = (char\*)malloc(MAX\_FILE\_SIZE); if (source.buffer == NULL) { printf("Memory allocation failed\n"); fclose(file); return;

}

// Read file content size\_t bytesRead = fread(source.buffer, 1, MAX\_FILE\_SIZE - 1, file); source.buffer[bytesRead] = '\0'; source.position = 0; source.currentLine = 1;

fclose(file);

}

// Free source code buffer void freeSource() { if (source.buffer != NULL) { free(source.buffer); source.buffer = NULL;

}

}

// Check if character is a delimiter bool isDelimiter(char ch) { const char\* delimiters = " \t\n,;(){}[]<>=+-\*/"; return strchr(delimiters, ch) != NULL;

}

// Check if character is an operator bool isOperator(char ch) { const char\* operators = "+-\*/<>="; return strchr(operators, ch) != NULL;

}

// Check if string is a keyword bool isKeyword(const char\* str) {

const char\* keywords[] = {

"auto", "break", "case", "char", "const", "continue",

"default", "do", "double", "else", "enum", "extern",

"float", "for", "goto", "if", "int", "long",

"register", "return", "short", "signed", "sizeof", "static",

"struct", "switch", "typedef", "union", "unsigned", "void", "volatile", "while"

};

int num\_keywords = sizeof(keywords) / sizeof(keywords[0]); for (int i = 0; i < num\_keywords; i++) { if (strcmp(str, keywords[i]) == 0) return true;

} return false;

}

// Other helper functions remain the same as in previous version bool isValidIdentifier(const char\* str) { if (str[0] == '\0' || isdigit(str[0])) return false; for (int i = 0; str[i] != '\0'; i++) { if (!(isalnum(str[i]) || str[i] == '\_')) return false;

} return true;

}

bool isInteger(const char\* str) {

if (str[0] == '\0') return false; int i

= 0; if (str[0] == '-' || str[0] == '+')

i++; for (; str[i] != '\0'; i++) { if (!isdigit(str[i])) return false;

} return true;

}

bool isReal(const char\* str) { if (str[0] == '\0') return false;

bool hasDecimal = false;

int i = 0; if (str[0] == '-' || str[0]

== '+') i++; for (; str[i] != '\0';

i++) { if (str[i] == '.') { if (hasDecimal) return false; hasDecimal = true;

} else if (!isdigit(str[i])) return false;

}

return hasDecimal;

}

// Extract substring from source char\* extractSubstring(int start, int end) { int length = end - start + 1; char\* substr = (char\*)malloc(length + 1); if (substr == NULL) { printf("Memory allocation failed\n"); return NULL;

}

strncpy(substr, &source.buffer[start], length); substr[length] = '\0'; return substr;

}

// Add entry to symbol table

void addToSymbolTable(const char\* name, const char\* type, int line) {

if (symbolCount < MAX\_TOKENS) { strncpy(symbolTable[symbolCount].name, name, MAX\_IDENTIFIER\_LENGTH - 1);

strncpy(symbolTable[symbolCount].type, type, 19); symbolTable[symbolCount].lineNumber = line; symbolCount++;

}

}

// Add entry to constant table

void addToConstantTable(const char\* constant) { if (constantCount < MAX\_TOKENS)

{ strncpy(constantTable[constantCount], constant, MAX\_IDENTIFIER\_LENGTH - 1); constantCount++;

}

}

// Get next token from source Token getNextToken() {

Token token = {TOKEN\_INVALID, NULL, source.currentLine};

while (source.buffer[source.position] != '\0' && isspace(source.buffer[source.position])) { if (source.buffer[source.position] == '\n') { source.currentLine++;

}

source.position++;

}

if (source.buffer[source.position] == '\0') { return token;

}

if (isOperator(source.buffer[source.position])) { char\* operator = extractSubstring(source.position, source.position); token.type = TOKEN\_OPERATOR;

token.lexeme = operator; source.position++;

// Add to token table if (tokenCount < MAX\_TOKENS) { tokenTable[tokenCount++] = token;

} return token;

}

if (isDelimiter(source.buffer[source.position]) &&

!isspace(source.buffer[source.position])) { char\* delimiter = extractSubstring(source.position, source.position); token.type = TOKEN\_DELIMITER;

token.lexeme = delimiter; source.position++;

// Add to token table if (tokenCount < MAX\_TOKENS) { tokenTable[tokenCount++] = token;

} return token;

}

int start = source.position; while (source.buffer[source.position] != '\0' &&

!isDelimiter(source.buffer[source.position])) { source.position++;

}

char\* lexeme = extractSubstring(start, source.position - 1); token.lexeme = lexeme;

if (isKeyword(lexeme)) { token.type = TOKEN\_KEYWORD;

}

else if (isValidIdentifier(lexeme)) { token.type = TOKEN\_IDENTIFIER; addToSymbolTable(lexeme, "IDENTIFIER", source.currentLine);

}

else if (isInteger(lexeme)) { token.type = TOKEN\_INTEGER; addToConstantTable(lexeme);

}

else if (isReal(lexeme)) { token.type = TOKEN\_REAL; addToConstantTable(lexeme);

}

// Add to token table

if (tokenCount < MAX\_TOKENS) { tokenTable[tokenCount++] = token;

} return token;

}

// Display menu options void displayMenu() { printf("\n=== Lexical Analyzer Menu ===\n"); printf("1. Analyze C File\n"); printf("2.

Display Symbol Table\n"); printf("3. Display Token Table\n"); printf("4. Display Constant Table\n"); printf("5. Clear Tables\n"); printf("6. Exit\n"); printf("Enter your choice (1-6): ");

}

// Process the C file void analyzeFile(const char\* filename) { clearTables(); readFile(filename);

if (source.buffer == NULL) { return;

}

Token token; do { token

= getNextToken(); if (token.lexeme != NULL) { free(token.lexeme);

}

} while (token.lexeme != NULL);

printf("\nFile analysis complete!\n"); freeSource();

}

// Display symbol table void displaySymbolTable() { printf("\n=== Symbol Table ===\n"); printf("%-20s %- 15s %-10s\n", "Name", "Type", "Line");

printf(" \n");

for (int i = 0; i < symbolCount; i++) { printf("%-20s %-15s %-10d\n", symbolTable[i].name, symbolTable[i].type, symbolTable[i].lineNumber);

}

}

// Display token table void displayTokenTable() { printf("\n=== Token Table ===\n"); printf("%-20s %-15s

%-10s\n", "Lexeme", "Type", "Line");

printf(" \n");

const char\* typeStrings[] = {

"KEYWORD", "IDENTIFIER", "INTEGER", "REAL",

"OPERATOR", "DELIMITER", "INVALID"

};

for (int i = 0; i < tokenCount; i++) { printf("%-20s %-15s %-10d\n", tokenTable[i].lexeme, typeStrings[tokenTable[i].type], tokenTable[i].lineNumber);

}

}

// Display constant table void displayConstantTable() { printf("\n=== Constant Table ===\n");

printf("%-20s\n", "Value"); printf("----

----------------\n"); for (int i = 0; i <

constantCount; i++) { printf("%- 20s\n", constantTable[i]);

}

}

// Clear all tables void clearTables() { symbolCount = 0; tokenCount = 0;

constantCount = 0; printf("\nAll tables cleared!\n");

}

// Process menu choice void processChoice(int choice) { char filename[100]; switch (choice) { case 1:

printf("Enter C file name: "); scanf("%s", filename); analyzeFile(filename); break;

case 2:

displaySymbolTable();

break; case 3: displayTokenTable(); break; case 4: displayConstantTable(); break; case 5: clearTables(); break; case 6:

printf("\nThank you for using the Lexical Analyzer!\n"); exit(0);

default:

printf("\nInvalid choice! Please try again.\n");

}

}

// Main function int main() {

int choice;

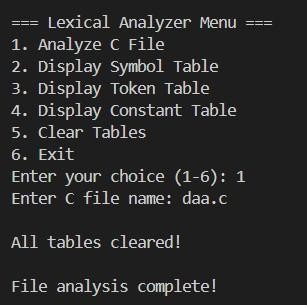
while (1) { displayMenu(); scanf("%d", &choice); processChoice(choice);

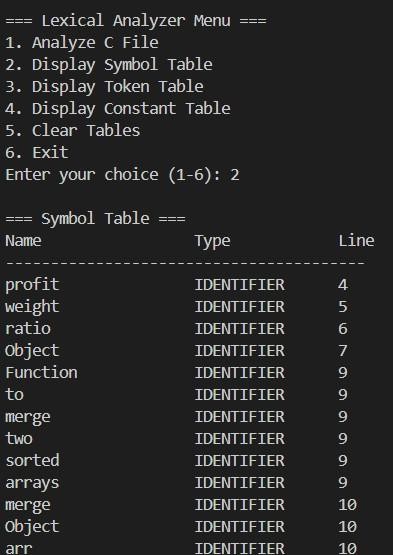
}

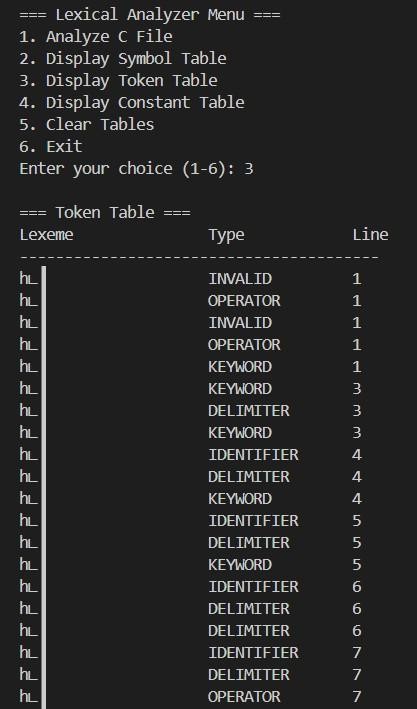
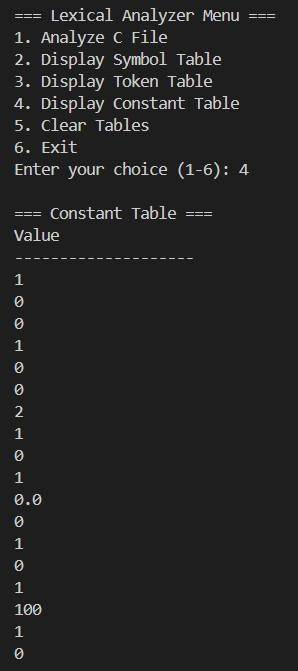
return 0;

}

# Output:





# Conclusion:

A lexical analyzer for the C language is crucial in the compilation process, as it reads the source code and breaks it down into meaningful tokens, such as keywords, operators, identifiers, and symbols. This step reduces the complexity for the parser by simplifying the input stream. The lexical analyzer must handle various language elements like whitespace, comments, and syntax errors. By efficiently identifying and classifying tokens, it serves as the foundation for further stages of compilation, ensuring the C code is processed accurately and efficiently.