# Faculty of Technology and Engineering

## Computer Science and Engineering

### Practical

| | | | | | |
|---|---|---|---|---|---|
| Academic Year | : | 2025-26 | Semester | : | 6 |
| Course code | : | CSE312 | Course name | : | Design of language processing |

## Practical - 2

**1. Objective:**
Implementation of a Lexical Analyzer for C Language Compiler

**2. Program Code:**

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

#define MAX_IDENTIFIER_LENGTH 100
#define MAX_SYMBOL_TABLE_SIZE 1000
#define MAX_TOKEN_LENGTH 100
#define MAX_ERRORS 100

// Token types
typedef enum {
   TOKEN_KEYWORD,
   TOKEN_IDENTIFIER,
   TOKEN_CONSTANT,
   TOKEN_STRING,
   TOKEN_PUNCTUATION,
   TOKEN_OPERATOR,
   TOKEN_COMMENT,
   TOKEN_WHITESPACE,
   TOKEN_ERROR
} TokenType;

// Token structure
typedef struct {
   TokenType type;
   char lexeme[MAX_TOKEN_LENGTH];
   int line;
```

```
    int column;
} Token;

// Symbol table entry
typedef struct {
    char identifier[MAX_IDENTIFIER_LENGTH];
    int count;
} SymbolTableEntry;

// Symbol table
typedef struct {
    SymbolTableEntry entries[MAX_SYMBOL_TABLE_SIZE];
    int size;
} SymbolTable;

// Lexical error structure
typedef struct {
    char message[256];
    int line;
    int column;
} LexicalError;

// Global variables
SymbolTable symbolTable = {.size = 0};
LexicalError errors[MAX_ERRORS];
int errorCount = 0;
int currentLine = 1;
int currentColumn = 1;

// C keywords
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"
};
const int keywordCount = sizeof(keywords) / sizeof(keywords[0]);

// Function prototypes
int isKeyword(const char *str);
void addToSymbolTable(const char *identifier);
void addError(const char *message, int line, int column);
void printToken(Token token);
void printSymbolTable();
void printErrors();
Token getNextToken(FILE *file);
void processFile(const char *filename);
void resetAnalyzer();

// Check if a string is a keyword
int isKeyword(const char *str) {
```

```
    for (int i = 0; i < keywordCount; i++) {
        if (strcmp(str, keywords[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

// Add identifier to symbol table
void addToSymbolTable(const char *identifier) {
    // Check if identifier already exists
    for (int i = 0; i < symbolTable.size; i++) {
        if (strcmp(symbolTable.entries[i].identifier, identifier) == 0) {
            symbolTable.entries[i].count++;
            return;
        }
    }

    // Add new identifier
    if (symbolTable.size < MAX_SYMBOL_TABLE_SIZE) {
        strcpy(symbolTable.entries[symbolTable.size].identifier, identifier);
        symbolTable.entries[symbolTable.size].count = 1;
        symbolTable.size++;
    }
}

// Add lexical error
void addError(const char *message, int line, int column) {
    if (errorCount < MAX_ERRORS) {
        sprintf(errors[errorCount].message, "%s", message);
        errors[errorCount].line = line;
        errors[errorCount].column = column;
        errorCount++;
    }
}

// Print token
void printToken(Token token) {
    const char *typeNames[] = {
        "Keyword", "Identifier", "Constant", "String",
        "Punctuation", "Operator", "Comment", "Whitespace", "Error"
    };

    if (token.type != TOKEN_WHITESPACE && token.type != TOKEN_COMMENT) {
        printf("%s: %s\n", typeNames[token.type], token.lexeme);
    }
}

// Print symbol table
void printSymbolTable() {
    printf("\n========================================\n");
```

```
    printf("SYMBOL TABLE ENTRIES\n");
    printf("=========================================\n");
    if (symbolTable.size == 0) {
        printf("(No identifiers found)\n");
    } else {
        for (int i = 0; i < symbolTable.size; i++) {
            printf("%d) %s\n", i + 1, symbolTable.entries[i].identifier);
        }
    }
    printf("=========================================\n");
}

// Print lexical errors
void printErrors() {
    if (errorCount > 0) {
        printf("\n=========================================\n");
        printf("LEXICAL ERRORS\n");
        printf("=========================================\n");
        for (int i = 0; i < errorCount; i++) {
            printf("%s\n", errors[i].message);
        }
        printf("=========================================\n");
    } else {
        printf("\n=========================================\n");
        printf("No lexical errors found!\n");
        printf("=========================================\n");
    }
}

// Reset analyzer state
void resetAnalyzer() {
    symbolTable.size = 0;
    errorCount = 0;
    currentLine = 1;
    currentColumn = 1;
}

// Get next token from file
Token getNextToken(FILE *file) {
    Token token;
    token.line = currentLine;
    token.column = currentColumn;
    int c = fgetc(file);

    // Skip whitespace
    while (c != EOF && isspace(c)) {
        if (c == '\n') {
            currentLine++;
            currentColumn = 1;
        } else {
            currentColumn++;
```

```
    }
    c = fgetc(file);
}

if (c == EOF) {
    token.type = TOKEN_ERROR;
    strcpy(token.lexeme, "EOF");
    return token;
}

token.line = currentLine;
token.column = currentColumn;

// Handle comments
if (c == '/') {
    int next = fgetc(file);
    if (next == '/') {
        // Single-line comment
        token.type = TOKEN_COMMENT;
        int idx = 0;
        token.lexeme[idx++] = c;
        token.lexeme[idx++] = next;
        while ((c = fgetc(file)) != EOF && c != '\n') {
            if (idx < MAX_TOKEN_LENGTH - 1) {
                token.lexeme[idx++] = c;
            }
            currentColumn++;
        }
        token.lexeme[idx] = '\0';
        if (c == '\n') {
            currentLine++;
            currentColumn = 1;
        }
        return token;
    } else if (next == '*') {
        // Multi-line comment
        token.type = TOKEN_COMMENT;
        int idx = 0;
        token.lexeme[idx++] = c;
        token.lexeme[idx++] = next;
        currentColumn += 2;

        int prev = 0;
        while ((c = fgetc(file)) != EOF) {
            if (idx < MAX_TOKEN_LENGTH - 1) {
                token.lexeme[idx++] = c;
            }
            currentColumn++;
            if (c == '\n') {
                currentLine++;
                currentColumn = 1;
```

```
            }
            if (prev == '*' && c == '/') {
                break;
            }
            prev = c;
        }
        token.lexeme[idx] = '\0';
        return token;
    } else {
        ungetc(next, file);
    }
}

// Handle identifiers and keywords
if (isalpha(c) || c == '_') {
    int idx = 0;
    token.lexeme[idx++] = c;
    currentColumn++;

    while ((c = fgetc(file)) != EOF && (isalnum(c) || c == '_')) {
        if (idx < MAX_TOKEN_LENGTH - 1) {
            token.lexeme[idx++] = c;
        }
        currentColumn++;
    }
    token.lexeme[idx] = '\0';

    if (c != EOF) {
        ungetc(c, file);
    }

    if (isKeyword(token.lexeme)) {
        token.type = TOKEN_KEYWORD;
    } else {
        token.type = TOKEN_IDENTIFIER;
        addToSymbolTable(token.lexeme);
    }
    return token;
}

// Handle numbers (constants)
if (isdigit(c)) {
    int idx = 0;
    token.type = TOKEN_CONSTANT;
    token.lexeme[idx++] = c;
    currentColumn++;

    int hasDecimal = 0;
    int hasError = 0;

    while ((c = fgetc(file)) != EOF) {
```

```
        if (isdigit(c)) {
            if (idx < MAX_TOKEN_LENGTH - 1) {
                token.lexeme[idx++] = c;
            }
            currentColumn++;
        } else if (c == '.' && !hasDecimal) {
            hasDecimal = 1;
            if (idx < MAX_TOKEN_LENGTH - 1) {
                token.lexeme[idx++] = c;
            }
            currentColumn++;
        } else if (isalpha(c)) {
            // Invalid: number followed by letter (like 7H)
            hasError = 1;
            if (idx < MAX_TOKEN_LENGTH - 1) {
                token.lexeme[idx++] = c;
            }
            currentColumn++;
            // Continue reading the invalid token
            while ((c = fgetc(file)) != EOF && (isalnum(c) || c == '_')) {
                if (idx < MAX_TOKEN_LENGTH - 1) {
                    token.lexeme[idx++] = c;
                }
                currentColumn++;
            }
            if (c != EOF) {
                ungetc(c, file);
            }
            break;
        } else {
            break;
        }
    }

    token.lexeme[idx] = '\0';

    if (hasError) {
        token.type = TOKEN_ERROR;
        char errorMsg[100];
        sprintf(errorMsg, "%s invalid lexeme", token.lexeme);
        addError(errorMsg, token.line, token.column);
    }

    if (c != EOF && !hasError) {
        ungetc(c, file);
    }
    return token;
}

// Handle string literals
if (c == '"') {
```

```
    int idx = 0;
    token.type = TOKEN_STRING;
    token.lexeme[idx++] = c;
    currentColumn++;

    while ((c = fgetc(file)) != EOF && c != '"') {
        if (c == '\\') {
            if (idx < MAX_TOKEN_LENGTH - 1) {
                token.lexeme[idx++] = c;
            }
            currentColumn++;
            c = fgetc(file);
            if (c == EOF) break;
        }
        if (idx < MAX_TOKEN_LENGTH - 1) {
            token.lexeme[idx++] = c;
        }
        currentColumn++;
        if (c == '\n') {
            currentLine++;
            currentColumn = 1;
        }
    }

    if (c == '"') {
        token.lexeme[idx++] = c;
        currentColumn++;
    }
    token.lexeme[idx] = '\0';
    return token;
}

// Handle character literals
if (c == '\'') {
    int idx = 0;
    token.type = TOKEN_STRING;
    token.lexeme[idx++] = c;
    currentColumn++;

    while ((c = fgetc(file)) != EOF && c != '\'') {
        if (c == '\\') {
            if (idx < MAX_TOKEN_LENGTH - 1) {
                token.lexeme[idx++] = c;
            }
            currentColumn++;
            c = fgetc(file);
            if (c == EOF) break;
        }
        if (idx < MAX_TOKEN_LENGTH - 1) {
            token.lexeme[idx++] = c;
        }
```

```
        currentColumn++;
    }

    if (c == '\'') {
        token.lexeme[idx++] = c;
        currentColumn++;
    }
    token.lexeme[idx] = '\0';
    return token;
}

// Handle operators and punctuation
const char *twoCharOps[] = {"==", "!=", "<=", ">=", "&&", "||", "++", "--",
                            "+=", "-=", "*=", "/=", "%=", "->", "<<", ">>"};
const int twoCharOpsCount = sizeof(twoCharOps) / sizeof(twoCharOps[0]);

int next = fgetc(file);
char twoChar[3] = {c, next, '\0'};
int isTwoChar = 0;

for (int i = 0; i < twoCharOpsCount; i++) {
    if (strcmp(twoChar, twoCharOps[i]) == 0) {
        isTwoChar = 1;
        break;
    }
}

if (isTwoChar) {
    token.type = TOKEN_OPERATOR;
    strcpy(token.lexeme, twoChar);
    currentColumn += 2;
    return token;
} else {
    if (next != EOF) {
        ungetc(next, file);
    }
}

// Single character operators and punctuation
if (strchr("+-*/%=<>!&|^~", c)) {
    token.type = TOKEN_OPERATOR;
    token.lexeme[0] = c;
    token.lexeme[1] = '\0';
    currentColumn++;
    return token;
}

if (strchr("(){}[];,.:?", c)) {
    token.type = TOKEN_PUNCTUATION;
    token.lexeme[0] = c;
    token.lexeme[1] = '\0';
```

```
    currentColumn++;
    return token;
  }

  // Invalid character
  token.type = TOKEN_ERROR;
  sprintf(token.lexeme, "%c", c);
  char errorMsg[100];
  sprintf(errorMsg, "%c invalid lexeme", c);
  addError(errorMsg, currentLine, currentColumn);
  currentColumn++;
  return token;
}

// Process file
void processFile(const char *filename) {
  FILE *file = fopen(filename, "r");
  if (!file) {
    printf("Error: Cannot open file '%s'\n", filename);
    printf("Please make sure the file exists in the current directory.\n");
    return;
  }

  resetAnalyzer();

  printf("\n========================================\n");
  printf("TOKENS\n");
  printf("========================================\n");

  Token token;
  do {
    token = getNextToken(file);
    if (strcmp(token.lexeme, "EOF") != 0) {
      printToken(token);
    }
  } while (strcmp(token.lexeme, "EOF") != 0);

  fclose(file);

  printSymbolTable();
  printErrors();
}

int main() {
  char filename[256];
  char choice;

  printf("========================================\n");
  printf("  LEXICAL ANALYZER\n");
  printf("========================================\n");
  printf("  DEBDOOT MANNA 23CS043\n");
```

```
    printf("=========================================\n\n");

    do {
        printf("Enter the C source file name: ");
        scanf("%s", filename);

        processFile(filename);

        printf("\nDo you want to analyze another file? (y/n): ");
        scanf(" %c", &choice);
        printf("\n");

    } while (choice == 'y' || choice == 'Y');

    printf("Thank you for using the Lexical Analyzer!\n");

    return 0;
}
```

## 3.Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

❯ gcc -o 3 3.c -Wall
 debdootmanna@Debdoots-MacBook-Air  ~  College  Sem 6 by college  DLP
❯ ./3
==========================================
  LEXICAL ANALYZER
==========================================
  DEBDOOT MANNA 23CS043
==========================================

Enter the C source file name: 3test1.c

==========================================
TOKENS
==========================================
Keyword: int
Identifier: main
Punctuation: (
Punctuation: )
Punctuation: {
Keyword: int
Identifier: a
Operator: =
Constant: 5
Punctuation: ,
Error: 7H
Punctuation: ;
Keyword: char
Identifier: b
Operator: =
String: 'x'
Punctuation: ;
Keyword: return
Identifier: a
Operator: +
Identifier: b
Punctuation: ;
Punctuation: }

==========================================
SYMBOL TABLE ENTRIES
==========================================
1) main
2) a
3) b
==========================================

==========================================
LEXICAL ERRORS
==========================================
7H invalid lexeme
==========================================

Do you want to analyze another file? (y/n): y

Enter the C source file name: 3test2.c

==========================================
TOKENS
==========================================
Keyword: void
Identifier: main
Punctuation: (
Punctuation: )
Punctuation: {
Keyword: long
Keyword: int
Identifier: bs
Punctuation: ,
Identifier: da
Punctuation: ,
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

debdootmanna@Debdoots-MacBook-Air ~ College Sem 6 by college DLP
./3
Identifier: hra
Punctuation: ,
Identifier: gs
Punctuation: ;
Identifier: scanf
Punctuation: (
String: "%ld"
Punctuation: ,
Operator: &
Identifier: bs
Punctuation: )
Punctuation: ;
Identifier: da
Operator: =
Identifier: bs
Operator: *
Constant: 0.40
Punctuation: ;
Identifier: hra
Operator: =
Identifier: bs
Operator: *
Constant: 0.20
Punctuation: ;
Identifier: gs
Operator: =
Identifier: bs
Operator: +
Identifier: da
Operator: +
Identifier: hra
Punctuation: ;
Identifier: printf
Punctuation: (
String: "\nbs : %ld"
Punctuation: ,
Identifier: bs
Punctuation: )
Punctuation: ;
Identifier: printf
Punctuation: (
String: "\nda : %ld"
Punctuation: ,
Identifier: da
Punctuation: )
Punctuation: ;
Identifier: printf
Punctuation: (
String: "\nhra : %ld"
Punctuation: ,
Identifier: hra
Punctuation: )
Punctuation: ;
Identifier: printf
Punctuation: (
String: "\ngs : %ld"
Punctuation: ,
Identifier: gs
Punctuation: )
Punctuation: ;
Punctuation: }

=========================================
SYMBOL TABLE ENTRIES
=========================================
1) main
2) bs
3) da
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS
debdootmanna@Debdoots-MacBook-Air ~ College Sem 6 by college DLP
❯ ./3
4) hra
5) gs
6) scanf
7) printf
==========================================

==========================================
No lexical errors found!
==========================================

Do you want to analyze another file? (y/n): y

Enter the C source file name: 3test3.c

==========================================
TOKENS
==========================================
Keyword: struct
Identifier: student
Punctuation: {
Keyword: int
Identifier: id
Punctuation: ;
Keyword: float
Identifier: cgpa
Punctuation: ;
Punctuation: }
Keyword: void
Identifier: main
Punctuation: (
Punctuation: )
Punctuation: {
Identifier: student
Identifier: s
Punctuation: ;
Identifier: s
Punctuation: .
Identifier: id
Operator: =
Constant: 10
Punctuation: ;
Identifier: s
Punctuation: .
Identifier: cgpa
Operator: =
Constant: 8.7
Punctuation: ;
Punctuation: }

==========================================
SYMBOL TABLE ENTRIES
==========================================
1) student
2) id
3) cgpa
4) main
5) s
==========================================

==========================================
No lexical errors found!
==========================================

Do you want to analyze another file? (y/n): n

Thank you for using the Lexical Analyzer!
debdootmanna@Debdoots-MacBook-Air ~ College Sem 6 by college DLP
❯
```