Fr. Conceicao Rodrigues College of Engineering
Department of Computer Engineering

Academic Term : Jan-May 2024 - 25

Class : T.E. (Computer - A)
Subject Name : System Programming and Compiler Construction Subject Code : (CPC601)

Practical No:	**02**
Title:	**Lexical Analyzer**
Date of Performance:	11/02/2025
Date of Submission:	19/02/2025
Roll No:	9913
Name of the Student:	Mark Lopes

Evaluation:

Sr. No	Rubric	Grade
1	Time Line (2)	
2	Output(3)	
3	Code optimization (2)	
4	Postlab (3)	

Signature of the Teacher :

FR. CONCEICAO RODRIGUES COLLEGE OF ENGINEERING

Department of Computer Engineering

System Programming and Compiler Construction

VI Semester (Computer) **Academic Year: 2024- 25**

Experiment No 2

Aim: Write a program to implement Lexical analyzer

Learning Objective: Converting a sequence of characters into a sequence of tokens.

Theory:

THE ROLE OF LEXICAL ANALYZER

The lexical analyzer is the first phase of a compiler. Its main task is to read the input

characters and produce as output a sequence of tokens that the parser uses for syntax analysis. Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

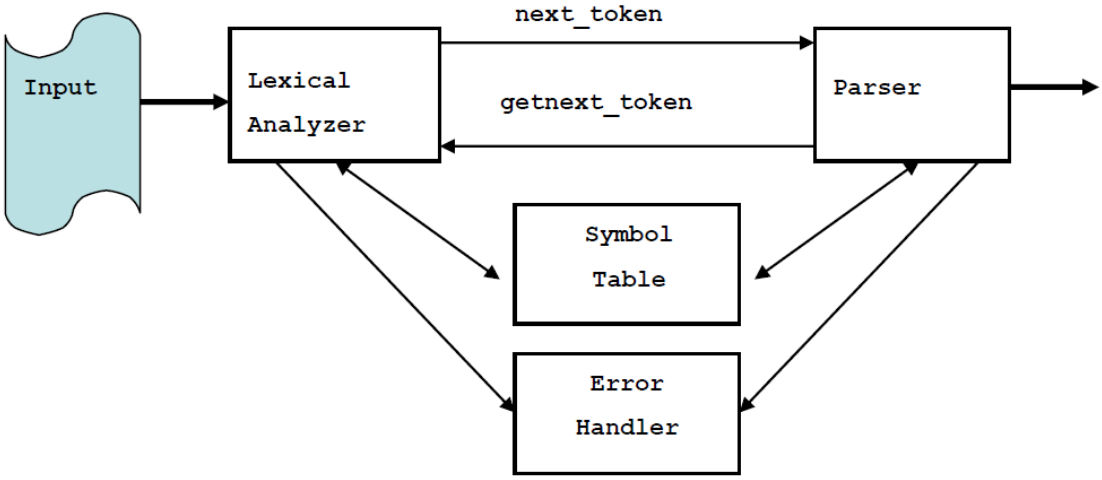

Figure 4.1 Interaction of Lexical Analyzer with Parser

Since the lexical analyzer is the part of the compiler that reads the source text, it may also perform certain secondary tasks at the user interface. One such task is stripping out from the source program comments and white spaces in the form of blank, tab, and new line characters. Another is correlating error messages from the compiler with the source program. Sometimes lexical analyzers are divided into a cascade of two phases first called "scanning" and the second "lexical analysis". The scanner is responsible for doing simple tasks, while the lexical analyzer proper does the more complex operations.

Implementation Details

1. Read the high level language as source program
2. Convert source program in to categories of tokens such as Identifiers, Keywords, Constants, Literals and Operators.

Test cases:

1. Input undefined token

Code:

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

#define MAX_TOKEN_LEN 100
#define MAX_TOKENS 1000

// Token type arrays
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 char operators[] = {
    '+', '-', '*', '/', '%',  // arithmetic
    '=', '<', '>', '!',       // relational
    '&', '|', '^', '~',       // bitwise
    ':', '?'                  // others
};
```

```c
const char delimiters[] = {
    '(', ')', '{', '}', '[', ']',
    ';', ',', '.'
};

const char* data_types[] = {
    "int", "char", "float", "double", "void",
    "long", "short", "signed", "unsigned"
};

// Arrays to store unique tokens
char identifiers[MAX_TOKENS][MAX_TOKEN_LEN];
char numbers[MAX_TOKENS][MAX_TOKEN_LEN];
char operators_found[MAX_TOKENS];
char delimiters_found[MAX_TOKENS];

int id_count = 0, num_count = 0, op_count = 0, del_count = 0;

// Helper function to check if a string is in an array
int exists_in_array(char arr[MAX_TOKENS][MAX_TOKEN_LEN], int count, char* str) {
    for (int i = 0; i < count; i++) {
        if (strcmp(arr[i], str) == 0) return 1;
    }
    return 0;
}
```

```c
// Helper function to check if a character is in an array
int exists_in_char_array(char arr[MAX_TOKENS], int count, char ch) {
    for (int i = 0; i < count; i++) {
        if (arr[i] == ch) return 1;
    }
    return 0;
}

// Check if token is a keyword
int is_keyword(char* str) {
    for (int i = 0; i < sizeof(keywords) / sizeof(char*); i++) {
        if (strcmp(str, keywords[i]) == 0) return 1;
    }
    return 0;
}

// Check if token is a data type
int is_data_type(char* str) {
    for (int i = 0; i < sizeof(data_types) / sizeof(char*); i++) {
        if (strcmp(str, data_types[i]) == 0) return 1;
    }
    return 0;
}
```

```c
// Check if character is an operator
int is_operator(char ch) {
    for (int i = 0; i < sizeof(operators); i++) {
        if (ch == operators[i]) return 1;
    }
    return 0;
}

// Check if character is a delimiter
int is_delimiter(char ch) {
    for (int i = 0; i < sizeof(delimiters); i++) {
        if (ch == delimiters[i]) return 1;
    }
    return 0;
}

// Helper function to print a line separator
void print_separator(int width, char style) {
    printf("+");
    for (int i = 0; i < width-1; i++) {
        printf("%c", style);
    }
    printf("+");
    for (int i = 0; i < width-1; i++) {
        printf("%c", style);
    }
```

```
    printf("+\n");
}

// Helper function to print wrapped text in a column
void print_wrapped_text(const char* text, int width) {
    int len = strlen(text);
    int pos = 0;
    int chars_remaining = len;
    int first_line = 1;

    while (chars_remaining > 0) {
        int chars_to_print = chars_remaining;
        if (chars_to_print > width - 2) {
            chars_to_print = width - 2;
            // Find last space before width limit
            while (chars_to_print > 0 && text[pos + chars_to_print] != ' ' && text[pos + chars_to_print] != ',') {
                chars_to_print--;
            }
            if (chars_to_print == 0) chars_to_print = width - 2; // If no space found, force break
        }
```

```
        if (first_line) {
            first_line = 0;
        } else {
            printf("| %*s | ", width-2, "");
        }

        printf("%-*.*s%*s |\n",
               chars_to_print, chars_to_print,
               text + pos,
               width - chars_to_print - 2, "");

        pos += chars_to_print;
        while (pos < len && (text[pos] == ' ' || text[pos] == ',')) pos++;
        chars_remaining = len - pos;
    }
}

int main() {
    printf("#########################################################################################################\n");
    printf("########################          LEXICAL ANALYSIS           ############################\n");
    printf("#########################################################################################################\n");
    FILE *file_ptr;
    char ch, token[MAX_TOKEN_LEN];
    int i = 0;
```

```
    file_ptr = fopen("fibo.c", "r");
    if (NULL == file_ptr) {
        printf("File can't be opened\n");
        return EXIT_FAILURE;
    }

    while ((ch = fgetc(file_ptr)) != EOF) {
        if (isspace(ch)) continue;

        if (isalpha(ch) || ch == '_') {
            i = 0;
            token[i++] = ch;
            while ((ch = fgetc(file_ptr)) != EOF && (isalnum(ch) || ch == '_')) {
                token[i++] = ch;
            }
            token[i] = '\0';
            ungetc(ch, file_ptr);

            if (!is_keyword(token) && !is_data_type(token) && !exists_in_array(identifiers, id_count, token)) {
                strcpy(identifiers[id_count++], token);
            }
        }
        else if (isdigit(ch)) {
            i = 0;
```

```
            token[i++] = ch;
            while ((ch = fgetc(file_ptr)) != EOF && (isdigit(ch) || ch == '.')) {
                token[i++] = ch;
            }
            token[i] = '\0';
            ungetc(ch, file_ptr);

            if (!exists_in_array(numbers, num_count, token)) {
                strcpy(numbers[num_count++], token);
            }
        }
        else if (is_operator(ch)) {
            if (!exists_in_char_array(operators_found, op_count, ch)) {
                operators_found[op_count++] = ch;
            }
        }
        else if (is_delimiter(ch)) {
            if (!exists_in_char_array(delimiters_found, del_count, ch)) {
                delimiters_found[del_count++] = ch;
            }
        }
    }
```

```
    fclose(file_ptr);

    int column_width = 50;

    // Print table header with dashes
    print_separator(column_width, '-');
    printf("| %-*s | %-*s |\n", column_width-2, "Category", column_width-2, "Value");
    print_separator(column_width, '_');  // Using underscore after Category

    // Print IDENTIFIER
    printf("| %-*s | ", column_width-2, "IDENTIFIER");
    char id_buffer[MAX_TOKENS * MAX_TOKEN_LEN] = {0};
    for (int j = 0; j < id_count; j++) {
        strcat(id_buffer, identifiers[j]);
        if (j < id_count - 1) strcat(id_buffer, ", ");
    }
    print_wrapped_text(id_buffer, column_width);
    print_separator(column_width, '_');  // Add separator after each category

    // Print OPERATOR
    printf("| %-*s | ", column_width-2, "OPERATOR");
    char op_buffer[MAX_TOKENS * 2] = {0};
    for (int j = 0; j < op_count; j++) {
        char temp[3] = {operators_found[j], 0};
        strcat(op_buffer, temp);
```

```c
        if (j < num_count - 1) strcat(num_buffer, ", ");
    }
    print_wrapped_text(num_buffer, column_width);
    print_separator(column_width, '_');  // Add separator after each category

    // Print DELIMITER
    printf("| %-*s | ", column_width-2, "DELIMITER");
    char del_buffer[MAX_TOKENS * 2] = {0};
    for (int j = 0; j < del_count; j++) {
        char temp[3] = {delimiters_found[j], 0};
        strcat(del_buffer, temp);
        if (j < del_count - 1) strcat(del_buffer, ", ");
    }
    print_wrapped_text(del_buffer, column_width);
    print_separator(column_width, '-');  // Using dash for final separator

    // Calculate and print total tokens
    int keyword_count = sizeof(keywords) / sizeof(keywords[0]);
    int datatype_count = sizeof(data_types) / sizeof(data_types[0]);
    int total_tokens = id_count + num_count + op_count + del_count + keyword_count + datatype_count;
    printf("\nTotal number of tokens: %d\n", total_tokens);

    return 0;
}
```

Output:

```
################################################################################################
########################          LEXICAL ANALYSIS          ############################
################################################################################################
+-------------------------------------------------+-------------------------------------------------+
| Category                                        | Value                                           |
+_________________________________________________+_________________________________________________+
| IDENTIFIER                                      | include, stdio, h, printFib, n, If, the, number, |
|                                                 | of, terms, is, smaller, than, printf, Invalid,  |
|                                                 | Number, First, two, series, prev1, prev2, loop, |
|                                                 | that, prints, fibonacci, i, Print, current, term |
|                                                 | and, update, previous, curr, d, main, Printing, |
|                                                 | first                                           |
+_________________________________________________+_________________________________________________+
| OPERATOR                                        | <, >, /, =, +, %                                |
+_________________________________________________+_________________________________________________+
| NUMBER                                          | 1, 0, 2, 9                                      |
+_________________________________________________+_________________________________________________+
| DELIMITER                                       | ., (, ), {, ;, }, ,                             |
+-------------------------------------------------+-------------------------------------------------+

Total number of tokens: 95
```

Post Lab Questions:

1. Explain the role of automata theory in compiler design.
2. What are the errors that are handled by Lexical analysis phase?

TE CompSA
Mark Lopes
9913

19/02/25 SPCC - Postlab - 2

Q.1 Role of automata theory in compiler design.

→ Automata theory is fundamental in compiler design, enabling efficient parsing and framing of program languages.

1. Lexical analysis :- The compiler first phase uses finite Automata to recognise tokens, keywords, identifiers, generators. Regular expression define token patterns which are converted into DFA for fast scanning.

2. Syntax analysis :- The syntax analyzer ensures the program follows the language. Grammar is content free grammar. Parser uses PDA for nested structures.

3. Semantic analysis and code generation :- Automata theory helps build abstract syntax tree ensure type correctness and generate intermediate code.

4. Optimization of code generation :- Automata helps in control flow analysis register allocation and instruction scheduling for efficient execution.

5. Runtime execution :- Virtual machines use automata for interpreting and executing compiled code.

Q.2 What are the errors that are handled by lexical analysis phase?

→ 1. Invalid characters :- Unrecognized symbols or illegal characters that do not match any defined token patterns.

2. Unterminated strings or comments :- Strings that start but don't end.

3. Improper number format :- Numeric literals with invalid formats like floating point number with multiple decimal points or characters.

4. Identifier length :- Identifiers that exceeds max. allocated allowed target length set by the language specification.

5. Illegal escape sequence :- Improper use of escape sequence in strings.