PART A

(PART A : TO BE REFFERED BY STUDENTS)

Experiment No. 5 (A)

Aim : To implement Lexical Analyzer for given language using Lex tool.

Objective: Develop a program to implement lexical analyzer:

a. Using Lex tool
b. Using finite automata

Outcome: Students are able to design and implement lexical analyzer for given language.

Theory:

The very first phase of compiler is lexical analysis. The lexical analyzer read the input characters and generates a sequence of tokens that are used by parser for syntax analysis. The figure 7.1 summarizes the interaction between lexical analyzer and parser.

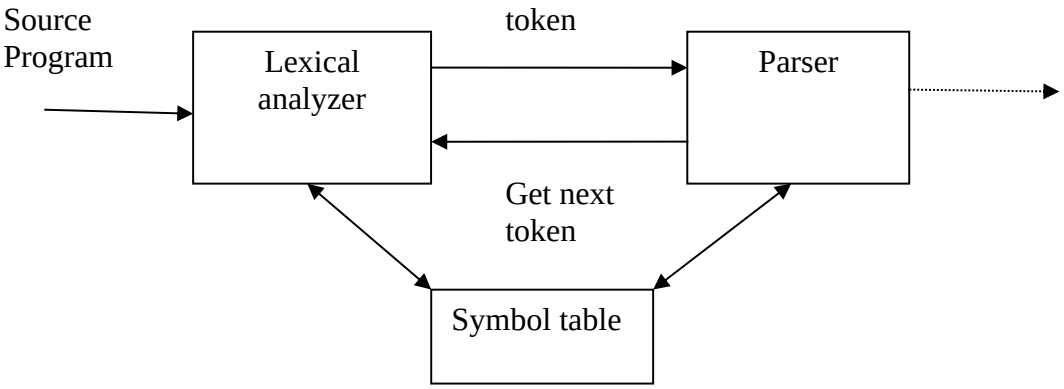


FIGURE: Interaction between lexical analyzer and parser

The lexical analyzer is usually implemented as subroutine or co-routine of the parser. When the "get next token" command received from parser, the lexical analyzers read input characters until it identifies next token.

Lexical analyzer also performs some secondary tasks at the user interface, such as stripping out comments and white spaces in the form of blank, tab and newline characters. It also correlates error messages from compiler to source program. For example lexical analyzer may keep track of number of newline characters and correlate the line number with an error message.

In some compilers, a lexical analyzer may create a copy of source program with error messages marked in it.

An important notation used to specify patterns is a regular expression. Each pattern matches a set of strings. Regular expression will serve as a name for set of strings.

A *recognizer* for a language is a program that takes as input a string *x* and answer "yes" if *x* is a sentence of the language and "no" otherwise.

We compile a regular expression into a recognizer by constructing a generalized transition diagram called a *finite automaton*.

We will design lexical analysis for the language which consists of strings containing “abb” as substring.

Thus, L= {“abb”, “babb”, “abbabba”,…}

The DFA for this language is represented in transition table below:

Q \ Σ	a	b
q0	q1	q0
q1	q1	q2
q2	q1	q3
q3	q3	q3

Now we will simulate this DFA to generate lexical analyzer for the language L.

LEX :

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

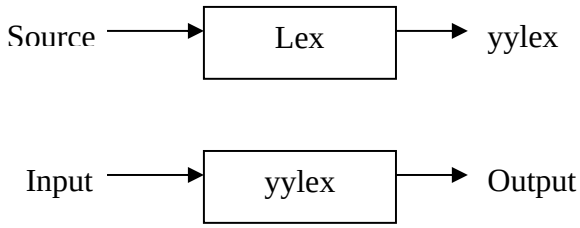


Fig : An overview of Lex

PART B

(PART B : TO BE COMPLETED BY STUDENTS)

(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded at the end of the practical)

Roll. No.B48	Name: Aryan Unhale
Class: TE-COMPS B	Batch:B3
Date of Experiment:11/3/25	Date of Submission:4/4/25
Grade:	

B.1 Software Code written by student:

Exp5.l:

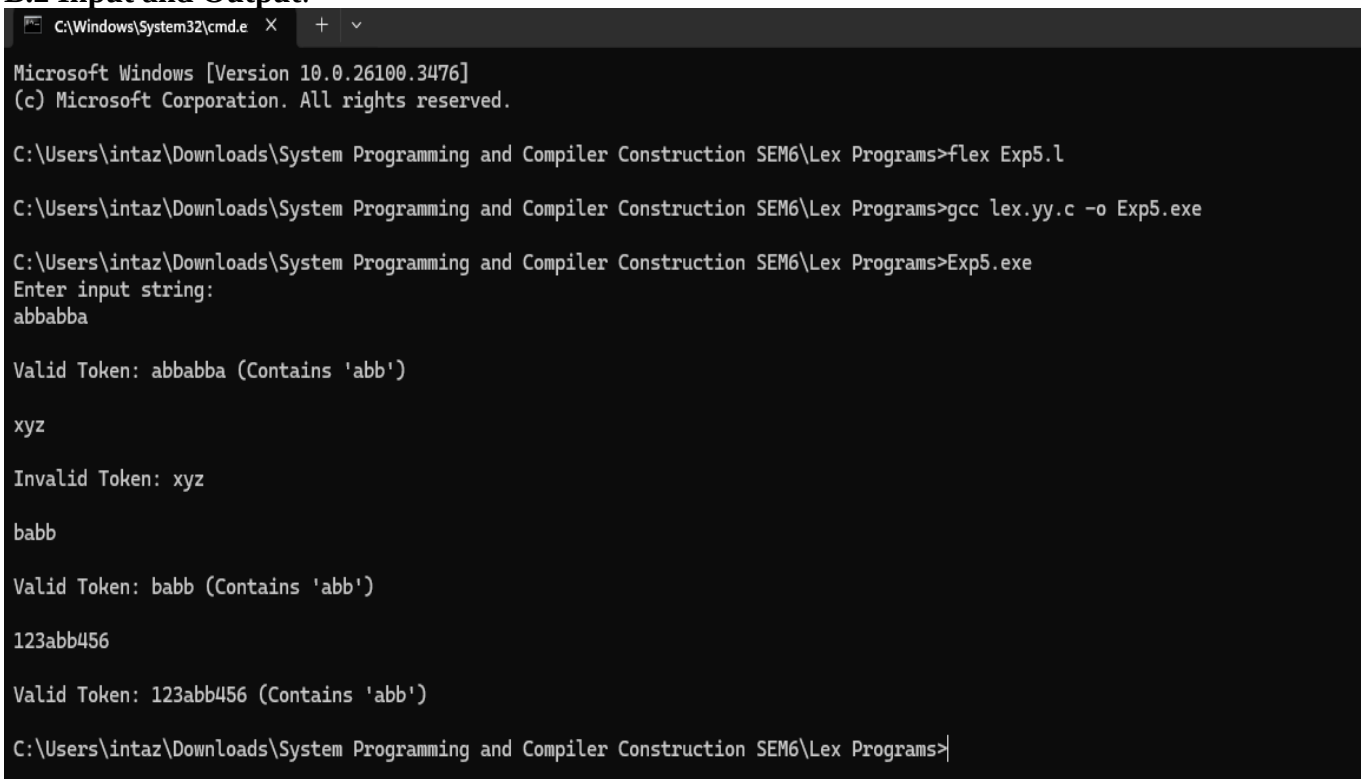
```
%{
#include <stdio.h>
%}

%%
.*abb.* { printf("\nValid Token: %s (Contains 'abb')", yytext); }
[a-zA-Z0-9]+ { printf("\nInvalid Token: %s", yytext); }
[ \t\n] { /* Ignore whitespace */ }
. { printf("\nUnknown Token: %s", yytext); }
%%

int main() {
    printf("Enter input string:\n");
    yylex();
    return 0;
}

int yywrap() {
    return 1;
}
```

B.2 Input and Output:



B.3 Observations and learning:

· Lex successfully generates a lexical analyzer that processes input strings.
· The generated program correctly identifies and classifies tokens based on the presence of "abb" as a substring.
· The modified regex .*abb.* ensures that words containing "abb" (anywhere) are recognized as valid.
· Strings without "abb" are marked as invalid tokens.
· The Lex tool automatically generates the C source code (lex.yy.c), which is then compiled and executed.
· The program correctly ignores whitespace and unrecognized characters.

B.4 Conclusion:

Lex can be used to efficiently implement a lexical analyzer for recognizing patterns in a given language.
Using regular expressions, we can define patterns to identify tokens within an input stream.
The experiment demonstrated how Lex translates pattern rules into a functional lexical analyzer, which can be integrated into a compiler.
This implementation serves as the first phase of a compiler (Lexical Analysis) and can be extended to handle more complex syntax structures.

B.5 Question of Curiosity

1. What is token?

A token is the smallest element of a program that is meaningful to the compiler. Tokens can be classified as follows:

- Keywords
- Identifiers
- Constants
- Strings
- Special Symbols
- Operators

2. What is the role of lexical analyzer?

- The lexical analysis is the first phase of the compiler where a lexical analyser operate as an interface between the source code and the rest of the phases of a compiler. It reads the input characters of the source program, groups them into lexemes, and produces a sequence of tokens for each lexeme. The tokens are sent to the parser for syntax analysis.
- If the lexical analyzer is located as a separate pass in the compiler it can need an intermediate file to locate its output, from which the parser would then takes its input. It can eliminate the need for the intermediate file, the lexical analyzer and the syntactic analyser (parser) are often grouped into the same pass where the lexical analyser operates either under the control of the parser or as a subroutine with the parser.
- The lexical analyzer also interacts with the symbol table while passing tokens to the parser. Whenever a token is discovered, the lexical analyzer returns a representation for that token to the parser. If the token is a simple construct including parenthesis, comma, or a colon, then it returns an integer program. If the token is a more complex items including an identifier or another token with a value, the value is also passed to the parser.
- Lexical analyzer separates the characters of the source language into groups that logically belong together, called tokens.

3. What is the output of Lexical analyzer?

The lexical token is a string containing a unit of grammar used in the programming language expressed as a series of characters, and it is generated by the lexical analyzer. The Lexical Analyzer's job and procedure is to tokenize the programme by dividing it into valid tokens and deleting all white space characters.

4. Which errors can be detected by Lexical Analyzer ?

During the lexical analysis phase this type of error can be detected. Lexical error is a sequence of characters that does not match the pattern of any token. Lexical phase error is found during the execution of the program.

Lexical phase error can be:

- Spelling error.
- Exceeding length of identifier or numeric constants.
- Appearance of illegal characters.
- To remove the character that should be present.
- To replace a character with an incorrect character.
- Transposition of two characters.