**Compiler Construction Lab 5**

**Lab Manual: Lex Tool**

**LEX Lexical analyzer generator:**

Lex is a computer program that generates lexical analyzers ("scanners" or "lexers"). Lex is commonly used with the yacc parser generator. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

1. A lexer or scanner is used to perform lexical analysis, or the breaking up of an input stream into meaningful units, or tokens.

2. For example, consider breaking a text file up into individual words.

3. Lex: a tool for automatically generating a lexer or scanner given a lex specification (.l file).

**Structure of a Lex file:**

The structure of a Lex file is intentionally similar to that of a yacc file; files are divided up into three sections, separated by lines that contain only two percent signs, as follows:

Definition section:

%%

Rules section:

%%

C code section:

<statements>

* The definition section is the place to define macros and to import header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.
* The rules section is the most important; it associates patterns with C statements. Patterns are regular expressions. When the lexer sees some text in the input matching a given pattern, it executes the associated C code. This is the basis of how Lex operates.
* The C code section contains C statements and functions copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs, it is more convenient to place this code in a separate file and link it in at compile time.

**Description:**

The lex command reads File or standard input, generates a C language program, and writes it to a file named lex.yy.c. This file, lex.yy.c, is a compilable C language program. A C++ compiler also can compile the output of the lex command. The -C flag renames the output file to lex.yy.C for the C++ compiler. The C++ program generated by the lex command can use either STDIO or IOSTREAMS. If the cpp define \_CPP\_IOSTREAMS is true during a C++ compilation, the program uses IOSTREAMS for all I/O. Otherwise, STDIO is used.

The lex command uses rules and actions contained in the File to generate a program, lex.yy.c, which can be compiled with the cc command. The compiled lex.yy.c can then receive input, break the input into the logical pieces defined by the rules in the File, and run program fragments contained in the actions in File.

The generated program is a C language function called yylex. The lex command stores the yylex function in a file named lex.yy.c. You can use the yylex function alone to recognize simple one-word input, or you can use it with other C language programs to perform more difficult input analysis functions. For example, you can use the lex command to generate a program that simplifies an input stream before sending it to a parser program generated by the yacc command. The yylex function analyzes the input stream using a program structure called a finite state machine. This structure allows the program to exist in only one state (or condition) at a time. There is a finite number of states allowed. The rules in File determine how the program moves from one state to

another. If you do not specify a File, the lex command reads standard input. It treats multiple files as a single file.

**Regular Expression Basics:**

. : matches any single character except \n

\* : matches 0 or more instances of the preceding regular expression

+ : matches 1 or more instances of the preceding regular expression

? : matches 0 or 1 of the preceding regular expression

| : matches the preceding or following regular expression

[ ] : defines a character class

() : groups enclosed regular expression into a new regular expression

“…”: matches everything within the “ “ literally

**Lab Tasks**

1. **Task: Identify Integers and Operators**
   * Create a Lex specification that recognizes integers (sequences of digits) and basic arithmetic operators (**+**, **-**, **\***, **/**).
   * Generate the lexical analyzer code using Lex.
   * Write a small program to test the generated lexical analyzer and ensure it correctly identifies integers and operators.
2. **Task: Ignore Whitespace and Comments**
   * Extend the Lex specification to ignore whitespace (spaces and tabs).
   * Add rules to handle single-line comments (starting with **//**) and ignore them.
   * Generate the updated lexical analyzer code using Lex.
   * Verify that the modified lexical analyzer correctly ignores whitespace and comments in a sample program.