**Assignment No : 07**

**TITLE :** Lexical analyzer for sample language using LEX.

**PROBLEM STATEMENT :**

Write a program using Lex specifications to implement lexical analysis phase of compiler to generate tokens of subset of ‘Java’ program.

**OBJECTIVE :**

* To understand first phase of compiler: Lexical Analysis.
* To learn and use compiler writing tools.
* Understand the importance and usage of LEX automated tool.

**THEORY :**

**Introduction:**

LEX stands for Lexical Analyzer.LEX is a UNIX utility which generates the lexical analyzer. LEX is a tool for generating scanners. Scanners are programs that recognize lexical patterns in text. These lexical patterns (or regular expressions) are defined in a particular syntax. A matched regular expression may have an associated action. This action may also include returning a token. When Lex receives input in the form of a file or text, it attempts to match the text with the regular expression. It takes input one character at a time and continues until a pattern is matched. If a pattern can be matched, then Lex performs the associated action (which may include returning a token). If, on the other hand, no regular expression can be matched, further processing stops and Lex displays an error message. Lex and C are tightly coupled. A lex file (files in Lex have the .l extension eg: first.l ) is passed through the lex utility, and produces output files in C (lex.yy.c). The program lex.yy.c basically consists of a transition diagram constructed from the regular expressions of first.l These file is then compiled object program a.out, and lexical analyzer transforms an input streams into a sequence of tokens as show in fig 1.1.

To generate a lexical analyzer two important things are needed. Firstly it will need a precise specification of the tokens of the language. Secondly it will need a specification of the action to be performed on identifying each token.



1. **LEX Specifications:**

The Structure of lex programs consists of three parts:



* **Definition Section :**

The Definition Section includes declarations of variables, start conditions regular definitions, and manifest constants (A manifest constant is an identifier that is declared to represent a constant e.g. # define PIE 3.14).

There are three things that can go in the definitions section:

* C code: Any indented code between %{ and %} is copied to the C file. This is typically used for defining file variables, and for prototypes of routines that are defined in the code segment.
* Definitions: A definition is very much like # define cpp directive. For example

letter [a-zA-Z]+

digit [0-9]+

These definitions can be used in the rules section: one could start a rule

{letter}{printf("n Wordis = %s",yytext);}

* State definitions: If a rule depends on context, it‟s possible to introduce states and incorporate those in the rules. A state definition looks like %s STATE, and by default a state INITIAL is already given.
* **Rule Section:**

Second section is for translation rules which consist of regular expression and action with respect to it. The translation rules of a Lex program are statements of the form:

p1 {action 1}

p2 {action 2}

p3 {action 3}

... ...

... ...

pn {action n}

Where, each p is a regular expression and each action is a program fragment describing what action the lexical analyzer should take when a pattern p matches a lexeme. In Lex the actions are written in C.

* **Auxiliary Function(User Subroutines):**

Third section holds whatever auxiliary procedures are needed by the actions. If the lex program is to be used on its own, this section will contain a main program. If you leave this section empty you will get the default main as follow:

int main()

{

yylex();

return 0;

}

In this section we can write a user subroutines its option to user e.g. yylex() is a unction automatically get called by compiler at compilation and execution of lex program or we can call that function from the subroutine section.

1. **Built - in Functions:**

|  |  |  |
| --- | --- | --- |
| **No.** | **Function** | **Meaning** |
| 1 | yylex() | The function that starts the analysis. It is automatically generated by Lex. |
| 2 | yywrap() | This function is called when end of file (or input) is encountered. If yywrap() returns 0, the scanner continues scanning, while if it returns 1 the scanner returns a zero token to report the end of file. |
| 3 | yyless(int n) | This function can be used to push back all but first „n‟ characters of the read Token. |
| 4 | yymore() | This function tells the lexer to append the next token to the current token. |
| 5 | yyerror() | This function is used for displaying any error message. |

1. **Built - in Variables:**

|  |  |  |
| --- | --- | --- |
| **No.** | **Variables** | **Meaning** |
| 1 | yyin | Of the type FILE\*. This point to the current file being parsed by the lexer. It is standard input file that stores input source program. |
| 2 | yyout | Of the type FILE\*. This point to the location where the output of the lexer will be written. By default, both yyin and yyout point to standard input and output. |
| 3 | yytext | The text of the matched pattern is stored in this variable (char\*) i.e. When lexer matches or recognizes the token from input token the lexeme stored in null terminated string called yytext.  OR  This is global variable which stores current token |
| 4 | yyleng | Gives the length of the matched pattern. (yyleng stores the length or number of character in the input string)The value in yyleng is same as strlen() functions. |
| 5 | yylineno | Provides current line number information. (May or may not be supported by the lexer.) |
| 6 | yylval | This is a global variable used to store the value of any token. |

1. **Regular Expression:**

|  |  |  |
| --- | --- | --- |
| **No.** | **RE** | **Meaning** |
| 1 | a | Matches a |
| 2 | abc | Matches abc |
| 3 | [abc] | Matches a or b or c |
| 4 | [a-f] | Matches a,b,c,d,e or f |
| 5 | [0-9] | Matches any digit |
| 6 | X+ | Matches one or more of x |
| 7 | X\* | Matches zero or more of x |
| 8 | [0-9]+ | Matches any integer |
| 9 | (…) | Grouping an expression into a single unit |
| 10 | | | Alteration ( or) |
| 11 | (b|c) | Is euivalent to [a-c]\* |
| 12 | X? | X is optional (0 or 1 occurrence) |
| 13 | If(def)? | Matches if or ifdef |
| 14 | [A-Za-z] | Matches any alphabetical character |
| 15 | . | Matches any character except new line |
| 16 | \. | Matches the . character |
| 17 | \n | Matches the new character |
| 18 | \t | Matches the tab character |
| 19 | \\ | Matches the \ character |
| 20 | [ \t] | Matches either a space or tab character |
| 21 | [^a-d] | Matches any character other than a,b,c and d |
| 22 | $ | End of the line |

1. **Steps to Execute the program:**

$ lex filename.l (eg: first.l)

$cc lex.yy.c–ll or gcc lex.yy.c–ll

$./a .out

**Algorithm:**

1. Start the program.
2. Lex program consists of three parts.
   1. Declaration         %%
   2. Translation rules %%
   3. Auxilary procedure.
3. The declaration section includes declaration of variables, maintest, constants and regular definitions.
4. Translation rule of lex program are statements of the form
   1. P1 {action}
   2. P2 {action}
   3. …
   4. …
   5. Pn {action}
5. Write a program in the vi editor and save it with .l extension.
6. Compile the lex program with lex compiler to produce output file as lex.yy.c. eg  $ lex filename.l   
        $ cc lex.yy.c -ll
7. Compile that file with C compiler and verify the output.

**CONCLUSION:**

Thus,we have understood Lexical Analyzer and implemented an application for lexical anaylzer to perform and scan the program and generate tokens of subset of java.

**FAQ**

1. What Is A Compiler And phases of compiler?
2. What is lex specification?
3. What is difference between LEX and YACC?
4. What is Regular Expression?
5. Why Lexical And Syntax Analysers Are Separated Out?
6. What is LEX?
7. How to run Lex program?
8. What is yytext,yyin,yyout,yywrap(),yylex() ?
9. Define: Token,lexemes,pattern.