**PRACTICAL-15**

Aim: Implement the lexical analyzer using JLex, flex or other lexical Analyzer generating tools.

Theory:

**Lexical analyser-** First phase of compiler (also known as scanner); it converts the high level language code into a sequence of tokens and can be implemented with the Deterministic Finite Automata. The output is a series of tokens sent to the syntax analyser.

**Lexical Analyser generating tools-** These are the tools that aid the construction of lexical analyser. Some of these are as follows:

JFlex: It is a lexical analyser generator for Java written in Java.

Flex (fast lexical analyser): It is a lexical analyser generating tool written in C. It is more flexible and produce faster codes.

Procedure:

Input: LEX specification file for tokens

Output: Produces source code for lexical analyser with name lex.yy.c and displays token in the input file.

1. Start
2. Open a new file in text editor.
3. Create a Lex specification file that accepts- keywords, identifiers, constants, operators, and relational operators in following format:
4. %{

Definition of constant /header files

%}

1. Regular Expressions

%%

Transition rules

%%

1. Auxiliary Procedure (main( ) function)
2. Save file with extension .l
3. Call lex tool on the terminal e.g. [root@localhost]# lex mylex.l.

This lex tool will convert “.l” file into “.c” language code file i.e., lex.yy.c

1. Compile this file using C/C++ compiler
2. Run file
3. After processing the output will be displayed on the file.
4. Stop

Code:

// \*\*\*\*\*\*\*\*\* LEX Program to identify Mini language Tokens \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

DIGIT [0-9]

LETTER [A-Z a-z]

DELIM [ \t\n]

WS { DELIM }+

ID {(LETTER)[LETTER/DIGIT]}+

INTEGER {DIGIT}+

%%

{WS} { printf("\n WS special characters \n"); }

{ID} { printf("\n Identifiers \n"); }

{DIGIT} {printf("\n Intgers\n"); }

if { printf("\n Keywords\n"); }

else { printf("\n keywords\n"); }

">" { printf("\n Relational Operators\n"); }

"<" { printf("\n Relational Operators \n"); }

"<=" { printf("\n Relational Operators \n"); }

"=>" { printf("\n Relational Operators \n"); }

"=" { printf("\n Relational Operators \n"); }

"!=" { printf("\n Logical Operators \n"); }

"&&" { printf("\n Logical Operators \n"); }

"||" { printf("\n Logical Operators \n"); }

"!" { printf("\n Logical Operators \n"); }

"+" { printf("\n Arthmetic Operator\n"); }

"-" { printf("\n Arthmetic Operator\n"); }

"\*" { printf("\n Arthmetic Operator\n"); }

"/" { printf("\n Arthmetic Operator\n"); }

"%" { printf("\n Arthmetic Operator\n"); }

%%

int yywrap(){ }

int main()

{

Printf(‘’ Enter the text : ’’)

yylex();

return 0 ;

}

Output:

[root@localhost]# lex lexprog.l

[root@localhost]# cc lex.yy.c

[root@localhost]# ./a.out lexprog

Test Cases:

|  |  |
| --- | --- |
| INPUT | OUTPUT |
| if | Keyword |
| \* | Arithmetic operator |
| <= | Relational operator |
| && | Logical Operator |