NIKIT GOKHE
Class – SY Comp D1
Roll No. 224024
GR No. 21810522

ASSIGNMENT NO 2

AIM:

Write a program using LEX specifications to implement lexical analysis phase of compiler to generate tokens from a given input .java file .

THEORY:

Lex is officially known as a ”Lexical Analyzer”. It’s main job is to break up an input stream into more usable elements. Or in, other words, to identify the ”interesting bits” in a text file.For exam- ple, if you are writing a compiler for the C programming language, the symbols ( ) ; all have significance on their own. The letter a usually appears as part of a keyword or variable name, and is not interesting on it’s own. Instead, we are interested in the whole word. Spaces and newlines are completely uninteresting, and we want to ignore them completely, unless they appear within quotes ”like this”All of these things are handled by the Lexical Analyser.

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and parti- tioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can generate analyzers in either C or Ratfor, a language which can be translated auto- matically to portable Fortran. It is available on the PDP-11

UNIX, Honeywell GCOS, and IBM OS systems. This manual, however, will only discuss generating analyzers in C on the UNIX sys- tem, which is the only supported form of Lex under UNIX Version 7. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

Running Lex:

>lex filename.l
>gcc lex.yy.c -lfl
>./a.out

To compile a lex program, do the following: Use the lex program to change the specification file into a C language program. The resulting program is in the lex.yy.c file. Use the cc command with the -ll flag to compile and link the program with a library of lex subroutines. The resulting executable program is in the a.out file.

1 Algorithm:

Step1: Start
Step2:Read input .java file
Step3 :Start reading from input file and generate token according to rules and print corresponding o/p
Step4 :
Example:
If word read is “class” print token as keyword
Step 5: keep reading till eof
Step6 :Stop

CODE:

```
%{
%}
DIGIT           [0-9]
NUMBER          {DIGIT}+
TEXT            [A-Za-z]

STRINGLITERAL   \"[^\n]*\"
KEYWORDS
  abstract|continue|goto|assert|this|implements|throw|break|throws|insta
```

```
nceof|return|transient|extends|try|catch|final|interface|static|finally|strictfp|v
olatile|class|native|super|const|new|synchronized
    ACCESS              "public"|"private"|"protected"|"default"
    PREPROCESSOR  import|package
    DATATYPE
       boolean|protected|double|byte|int|short|void|char|long|float
    CONDITIONAL     if|switch|else|case
    ITERATIVE           while|for|do
    ACCESSSPECIFIER             default|private|public|
    SC                          ";"
    IDENTIFIER          [A-Za-z$_]({DIGIT}|{TEXT}|_|$)*
    ARITH_OP            "+"|"-"|"/"|"%"|"*";
    LOGICAL_OP        "&&"|"||"|"!"|"!="
    REL_OP              "<"|">"|"<="|">="|"=="
    UNARY               "++"|"--"

    %%

    [ \n\t]+                    ;
    {PREPROCESSOR}                      { printf("%s\t==> PREPROCESSOR\n",yytext); }
    {CONDITIONAL}               { printf("%s\t==> CONDITIONAL\n",yytext); }
    {ITERATIVE}                 { printf("%s\t==> ITERATIVE\n",yytext); }
    {DATATYPE}                  { printf("%s\t==> DATATYPE\n",yytext); }
    {ACCESS}            { printf("%s\t==> ACCESS SPECIFIER\n",yytext); }
    {KEYWORDS}                  { printf("%s\t==> KEYWORDS\n",yytext); }
    {STRINGLITERAL}             { printf("%s\t==> STRINGLITERAL\n",yytext); }
    {IDENTIFIER}                { printf("%s\t==> IDENTIFIER\n",yytext); }
    {NUMBER}                    { printf("%s\t==> CONSTANT INTEGER\n",yytext); }
    {SC}                        { printf("%s\t==> DELIMITER\n",yytext); }
    {UNARY}                     { printf("%s\t==> UNARY OP\n",yytext); }
    {ARITH_OP}                  { printf("%s\t==> ARITHMETIC OPERATOR\n",yytext); }
    {LOGICAL_OP}                { printf("%s\t==> LOGICAL OP\n",yytext); }
    {REL_OP}            { printf("%s\t==> RELATIONAL OP\n",yytext); }
    "="                         { printf("%s\t==> ASSIGNMENT OP\n",yytext); }
    "{"                         { printf("%s\t==> BLOCK BEGIN\n",yytext); }
    "}"                         { printf("%s\t==> BLOCK END\n",yytext); }
    "("                         { printf("%s\t==> PARANTHESIS BEGIN\n",yytext); }
    ")"                         { printf("%s\t==> PARENTHESIS END\n",yytext); }
    .                           ;

    %%
```

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

int main()
{
   yyin = fopen("input.java","r");

   yylex();

   return 0;
}
```

OUTPUT

```
class   ==> KEYWORDS
input   ==> IDENTIFIER
{       ==> BLOCK BEGIN
public  ==> ACCESS SPECIFIER
static  ==> KEYWORDS
void    ==> DATATYPE
main    ==> IDENTIFIER
(       ==> PARANTHESIS BEGIN
String  ==> IDENTIFIER
args    ==> IDENTIFIER
)       ==> PARENTHESIS END
{       ==> BLOCK BEGIN
System  ==> IDENTIFIER
out     ==> IDENTIFIER
println ==> IDENTIFIER
(       ==> PARANTHESIS BEGIN
"This is going to go in a flex program."        ==> STRINGLITERAL
)       ==> PARENTHESIS END
;       ==> DELIMITER
a       ==> IDENTIFIER
=       ==> ASSIGNMENT OP
_b12    ==> IDENTIFIER
+       ==> ARITHMETIC OPERATOR
$c123   ==> IDENTIFIER
;       ==> DELIMITER
if      ==> CONDITIONAL
(       ==> PARANTHESIS BEGIN
i       ==> IDENTIFIER
==      ==> RELATIONAL OP
0       ==> CONSTANT INTEGER
)       ==> PARENTHESIS END
d       ==> IDENTIFIER
=       ==> ASSIGNMENT OP
10      ==> CONSTANT INTEGER
;       ==> DELIMITER
else    ==> CONDITIONAL
c       ==> IDENTIFIER
=       ==> ASSIGNMENT OP
10      ==> CONSTANT INTEGER
;       ==> DELIMITER
PUBLIC  ==> IDENTIFIER
;       ==> DELIMITER
}       ==> BLOCK END
}       ==> BLOCK END
```