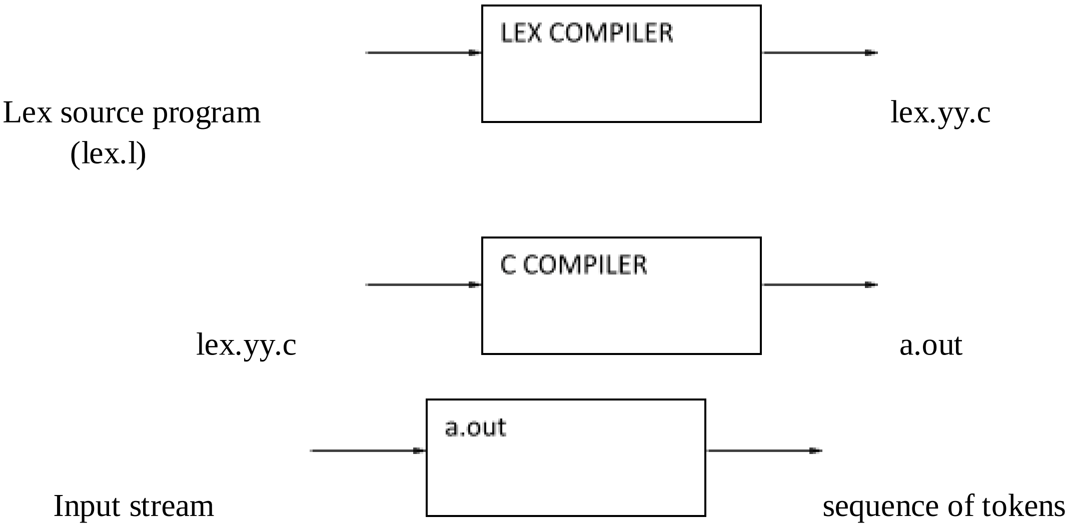
# Experiment 04 : Lex Tool

**Learning Objective**: Students should be able to build a Lexical analyzer using LEX / Flex tool.

**Tools:** Open Source tool (Ubuntu , LEX tool), Notepad++

## Theory:

**LEX :**A tool widely used to specify lexical analyzers for a variety of languages .We refer to the tool as Lex compiler , and to its input specification as the Lex language.



**Steps for creating a lexical analyzer with Lex**

### Lex specifications:

A Lex program (the .l file ) consists of three parts:

declarations

%%

translation rules

%%

auxiliary procedures

1. The declarations section includes declarations of variables,manifest constants(A manifest constant is an identifier that is declared to represent a constant e.g. # define PIE 3.14), and regular definitions.
2. The translation rules of a Lex program are statements of the form : p1 {action 1} p2 {action 2} p3 {action 3}

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.

3. In Lex the actions are written in C.
4. The third section holds whatever auxiliary procedures are needed by the actions.Alternatively these procedures can be compiled separately and loaded with the lexical analyzer.

**How does this Lexical analyzer work?**

The lexical analyzer created by Lex behaves in concert with a parser in the following manner. When activated by the parser, the lexical analyzer begins reading its remaining input , one character at a time, until it has found the longest prefix of the input that is matched by one of the regular expressions p. Then it executes the corresponding action. Typically the action will return control to the parser. However, if it does not, then the lexical analyzer proeeds to find more lexemes, until an action causes control to return to the parser. The repeated search for lexemes until an explicit return allows the lexical analyzer to process white space and comments conveniently.

The lexical analyzer returns a single quantity, the token, to the parser. To pass an attribute value with information about the lexeme, we can set the global variable yylval.

e.g. Suppose the lexical analyzer returns a single token for all the relational operators, in which case the parser won't be able to distinguish between " <=",">=","<",">","==" etc. We can set yylval appropriately to specify the nature $man flex
of the operator.

Note: To know the exact syntax and the various symbols that you can use to write the regular expressions visit the manual page of FLEX in LINUX :

**The two variables** *yytext* **and** *yyleng*

Lex makes the lexeme available to the routines appearing in the third section through two variables *yytext*and *yyleng*

1. *yytext* is a variable that is a pointer to the first character of the lexeme.
2. *yyleng*is an integer telling how long the lexeme is.

A lexeme may match more than one patterns. How is this problem resolved?

Take for example the lexeme *if*. It matches the patterns for both *keyword if* and *identifier*. If the pattern for *keyword if* precedes the pattern for *identifier* in the *declaration* list of the lex program the conflict is resolved in favor of the keyword. In general this ambiguity-resolving strategy makes it easy to reserve keywords by listing them ahead of the pattern for identifiers.

The Lex's strategy of selecting the longest prefix matched by a pattern makes it easy to resolve other conflicts like the one between "<" and "<=".

In the lex program, a *main()* function is generally included as: *main(){*

*yyin=fopen(***filename,***"r"); while(yylex());*

*}*

Here ***filename*** corresponds to input file and the *yylex* routine is called which returns the tokens.

**Lex Syntax and Example**

Lex is short for "lexical analysis". Lex takes an input file containing a set of lexical analysis rules or regular expressions. For output, Lex produces a C function which when invoked, finds the next match in the input stream.

1. Format of lex input:

(beginning in col. 1)　　　declarations
　　　　%%
　　　　*token-rules*

　　　　%%
　　　　*aux-procedures*

2. Declarations:
  a) string sets; name character-class
  b) standard C; %{　　-- c declarations --
　　　　%}

3. Token rules:　　　　*regular-expression* { *optional C-code* }

  a) if the expression includes a reference to a character class, enclose the class name in brackets { }
  b) regular expression operators;

* , +　　--closure, positive closure
" " or \　　--protection of special chars
|　　--or
^ --beginning-of-line anchor
()--grouping
$ --end-of-line anchor
? --zero or one
. --any char (except \n)
{ref}　　--reference to a named character class (a definition)
[ ]　　--character class
[^ ]　　--not-character class

4. Match rules:　　Longest match is preferred. If two matches are equal length, the first match is preferred. Remember, lex partitions, it does not attempt to find nested matches. Once a character becomes part of a match, it is no longer considered for other matches.

5. Built-in variables:　yytext　-- ptr to the matching lexeme. (char *yytext;) yyleng　　-- length of matching lexeme (yytext). Note: some systems use yyleng

6. **Aux Procedures:** C functions may be defined and called from the C-code of token rules or from other functions. Each lex file should also have a yyerror() function to be called when lex encounters an error condition.

7. Example header file: tokens.h

```
        #define NUM     1           // define constants used by lexyy.c

        #define ID      2           // could be defined in the lex rule file

        #define PLUS    3

        #define MULT    4

        #define ASGN    5

        #define SEMI    6
```

7. Example lex file

```
D       [0-9]                   /* note these lines begin in col. 1 */
A       [a-zA-Z]
  %{
  #include “tokens.h”
  %}
  %%
{D}+              return (NUM);       /* match integer numbers */
{A}({A}|{D})*                   return (ID);        /* match identifiers */
  "+"   return (PLUS);        /* match the plus sign (note protection) */ "*"
        return (MULT);        /* match the multsign (note protection
                                                  again) */
: =             return (ASGN);      /* match the assignment string */
;                 return (SEMI);      /* match the semi colon */
.               ;                   /* ignore any unmatched chars */
  %%

voidyyerror ()                  /* default action in case of error in yylex()
   {

                printf (" error\n");
  exit(0);
   }

                voidyywrap () { }             /* usually only needed for some Linux systems */
```

8. Execution of lex:

(to generate the *yylex()* function file and then compile a user program)

```
  (MS)      c:> flexrulefile                    (Linux)       $
                                                lexrulefile
```

lexproduceslex.yy

flexproduceslexyy.c .c

The produced .c file contains this function: intyylex()

9. User program:
(The above scanner file must be linked into the project)

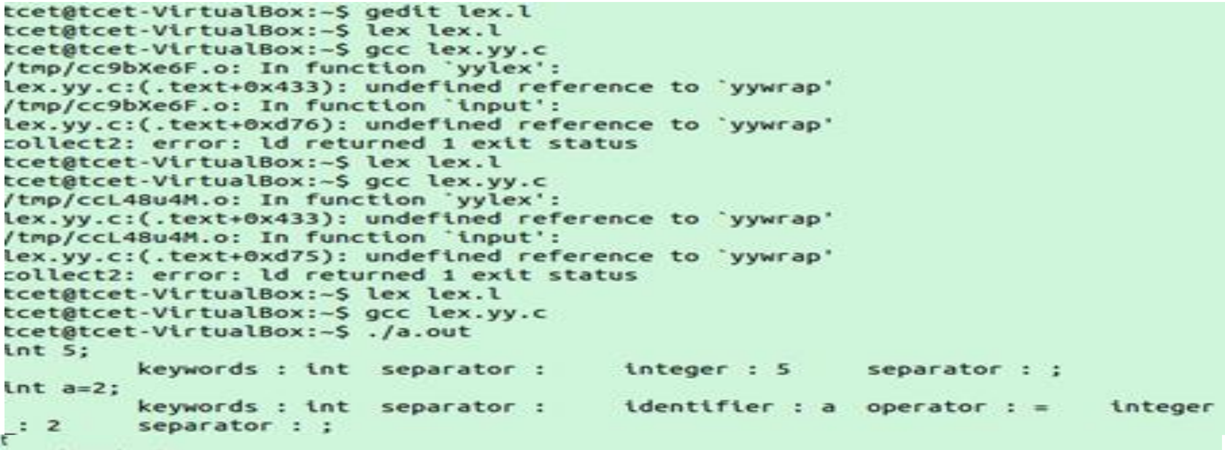
```
#include <stdio.h>
#include “tokens.h”

intyylex ();                          // scanner prototype
extern char* yytext;
main ()
{    int n; while ( n = yylex() )     // call scanner until it returns 0 for
     EOF
         printf (" %d %s\n", n, yytext);   // output the token code and lexeme
string
}
```

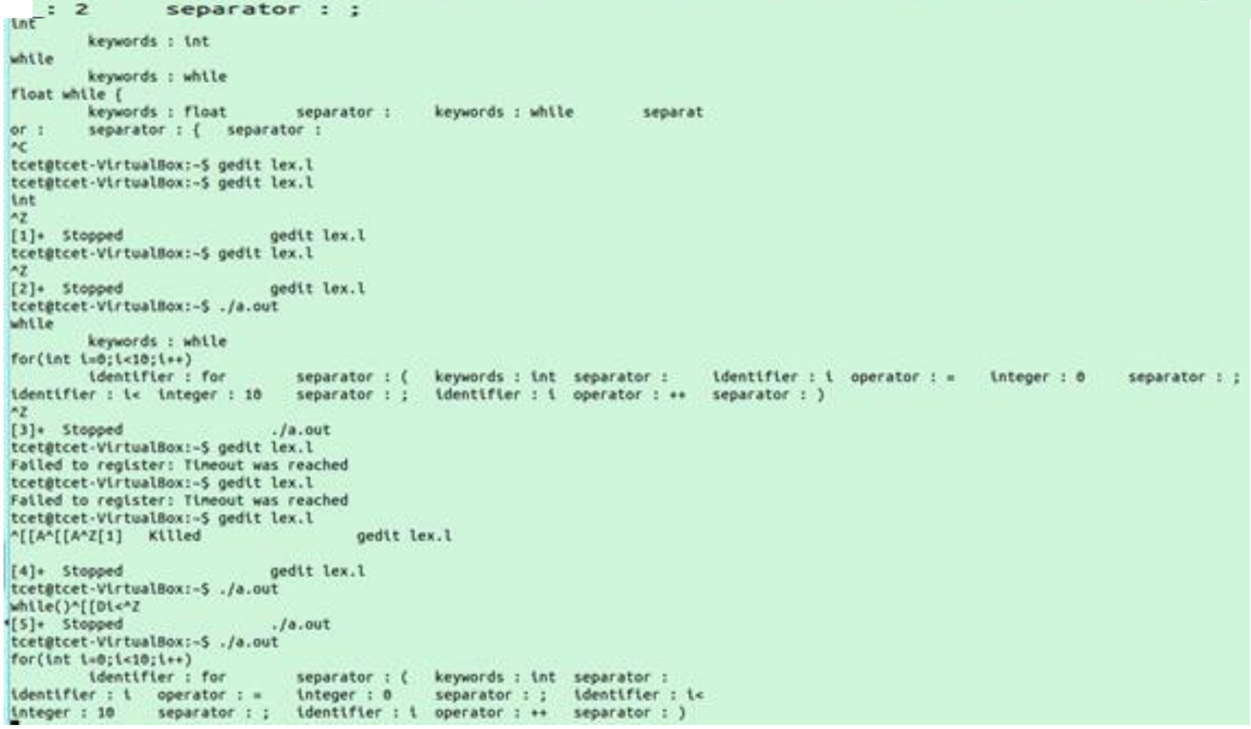
**Design:**

```
{
int n = 0 ; %}
%%
"while"|"if"|"else" {n++;printf("\t keywords: %s", yytext);}
"int"|"float" {n++; pr printf("\t keywords: %s", yytext);}
[a-zA-Z_][a-zA-Z0-9 -9_]* {n++; printf("\t identifier: %s", yytext);}
"<="|"=="|"="|"++"'" "|"-"|"*"|"+" {n++; printf("\t operator: %s", yytext);}
[(){}, ;] {n++; printf("\t separator: %s", yytext);}
[0-9]*"."[0-9]+ {n++;printf("\t float: %s", yytext);}
[0-9]+ {n++; printf("\t integer : %s", yytext);}
%%
int main()
{
yylex();
printf("\n total no. of token = %d\n", n);
}
int yywrap() {
return 0;
}
```

**Result and Discussion:**



```
tcet@tcet-VirtualBox:~$ gedit lex.l
tcet@tcet-VirtualBox:~$ lex lex.l
tcet@tcet-VirtualBox:~$ gcc lex.yy.c
/tmp/cc9bXe6F.o: In function `yylex':
lex.yy.c:(.text+0x433): undefined reference to `yywrap'
/tmp/cc9bXe6F.o: In function `input':
lex.yy.c:(.text+0xd76): undefined reference to `yywrap'
collect2: error: ld returned 1 exit status
tcet@tcet-VirtualBox:~$ lex lex.l
tcet@tcet-VirtualBox:~$ gcc lex.yy.c
/tmp/ccL48u4M.o: In function `yylex':
lex.yy.c:(.text+0x433): undefined reference to `yywrap'
/tmp/ccL48u4M.o: In function `input':
lex.yy.c:(.text+0xd75): undefined reference to `yywrap'
collect2: error: ld returned 1 exit status
tcet@tcet-VirtualBox:~$ lex lex.l
tcet@tcet-VirtualBox:~$ gcc lex.yy.c
tcet@tcet-VirtualBox:~$ ./a.out
int 5;
        keywords : int  separator :     integer : 5     separator : ;
int a=2;
        keywords : int  separator :     identifier : a  operator : =    integer
_: 2    separator : ;
```



```
int
        keywords : int
while
        keywords : while
float while {
        keywords : float        separator :     keywords : while        separat
or :    separator : {   separator :
^C
tcet@tcet-VirtualBox:~$ gedit lex.l
tcet@tcet-VirtualBox:~$ gedit lex.l
int
^Z
[1]+  Stopped                 gedit lex.l
tcet@tcet-VirtualBox:~$ gedit lex.l
^Z
[2]+  Stopped                 gedit lex.l
tcet@tcet-VirtualBox:~$ ./a.out
while
        keywords : while
for(int i=0;i<10;i++)
        identifier : for        separator : (   keywords : int  separator :     identifier : i  operator : =    integer : 0     separator : ;
identifier : i<  integer : 10    separator : ;   identifier : i  operator : ++   separator : )
^Z
[3]+  Stopped                 ./a.out
tcet@tcet-VirtualBox:~$ gedit lex.l
Failed to register: Timeout was reached
tcet@tcet-VirtualBox:~$ gedit lex.l
Failed to register: Timeout was reached
tcet@tcet-VirtualBox:~$ gedit lex.l
^[[A^[[A^Z[1]   Killed                  gedit lex.l

[4]+  Stopped                 gedit lex.l
tcet@tcet-VirtualBox:~$ ./a.out
while()^[[Di<^Z
[5]+  Stopped                 ./a.out
tcet@tcet-VirtualBox:~$ ./a.out
for(int i=0;i<10;i++)
        identifier : for        separator : (   keywords : int  separator :
identifier : i   operator : =    integer : 0     separator : ;   identifier : i<
integer : 10     separator : ;   identifier : i  operator : ++   separator : )
```

**Learning Outcomes:** The student should have the ability to

LO1 **Summarize** different Compiler Construction tools.

LO2: ***Describ*****e** the structure of Lex specification.

LO3: ***Apply*** LEX Compiler for Automatic Generation of Lexical Analyzer.

LO4: Construct Lexical analyzer using open source tool for compiler design

**Course Outcomes**: Upon completion of the course students will be able to Illustrate the working of the compiler and handwritten /automatic lexical analyzer..

**Conclusion:**

Lex is a tool or a computer program that generates Lexical Analyzers (converts the stream of characters into tokens). The Lex tool itself is a compiler. The Lex compiler takes the input and transforms that input into input patterns. It is commonly used with YACC(Yet Another Compiler Compiler). It was written by Mike Lesk and Eric Schmidt.

For Faculty Use

| | | | | |
|---|---|---|---|---|
| **Correction Parameters** | **Formative Assessment [40%]** | **Timely completion of Practical [ 40%]** | **Attendance / Learning Attitude [20%]** | |
| **Marks Obtained** | | | | |