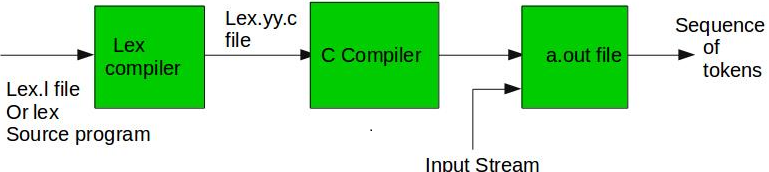
**FLEX (fast lexical analyzer generator)** is a tool/computer program for generating lexical analyzers (scanners or lexers).

The function **yylex()** is automatically generated by the flex when it is provided with a **.l file** and this yylex() function is expected by parser to call to retrieve tokens from current/this token stream.

**Note:** The function yylex() is the main flex function that runs the Rule Section and extension (.l) is the extension used to save the programs.



**Step 1:** An input file describes the lexical analyzer to be generated named lex.l is written in lex language. The lex compiler transforms lex.l to C program, in a file that is always named lex.yy.c.   
**Step 2:** The C compiler compile lex.yy.c file into an executable file called a.out.   
**Step 3:** The output file a.out take a stream of input characters and produce a stream of tokens.

**Program Structure:**

In the input file, there are 3 sections:

**1. Definition Section:** The definition section contains the declaration of variables, regular definitions, manifest constants. In the definition section, text is enclosed in **“%{ %}”** brackets. Anything written in this brackets is copied directly to the file **lex.yy.c**

**Syntax:**

%{

// Definitions

%}

**2. Rules Section:** The rules section contains a series of rules in the form: *pattern action* and pattern must be unintended and action begin on the same line in {} brackets. The rule section is enclosed in **“%% %%”**.

**Syntax:**

%%

pattern action

%%

| **Pattern** | **It can match with** |
| --- | --- |
| [0-9] | all the digits between 0 and 9 |
| [0+9] | either 0, + or 9 |
| [0, 9] | either 0, ‘, ‘ or 9 |
| [0 9] | either 0, ‘ ‘ or 9 |
| [-09] | either -, 0 or 9 |
| [-0-9] | either – or all digit between 0 and 9 |
| [0-9]+ | one or more digit between 0 and 9 |
| [^a] | all the other characters except a |
| [^A-Z] | all the other characters except the upper case letters |
| a{2, 4} | either aa, aaa or aaaa |
| a{2, } | two or more occurrences of a |
| a{4} | exactly 4 a’s i.e, aaaa |
| . | any character except newline |
| a\* | 0 or more occurrences of a |
| a+ | 1 or more occurrences of a |
| [a-z] | all lower case letters |
| [a-zA-Z] | any alphabetic letter |
| w(x | y)z | wxz or wyz |

**3. User Code Section:** This section contains C statements and additional functions. We can also compile these functions separately and load with the lexical analyzer.

Basic Program Structure:

%{

// Definitions

%}

%%

Rules

%%

User code section

**How to run the program:**   
To run the program, it should be first saved with the extension **.l or .lex**. Run the below commands on terminal in order to run the program file.

**Step 1:** lex filename.l or lex filename.lex depending on the extension file is saved with   
**Step 2:** gcc lex.yy.c   
**Step 3:** ./a.out   
**Step 4:** Provide the input to program in case it is required

**Note:** Press **Ctrl+D** or use some **rule** to stop taking inputs from the user.

Flex takes a program written in a combination of Flex and C, and it writes out a file (called **lex.yy.c**) that holds a definition of function yylex(), with the following prototype.

int yylex(void);

**The file to read**

yylex reads from the file stored in variable yyin:

FILE\* yyin;

It is up to you to open a file for reading and store it into yyin before you call yylex.

Each time your program calls yylex, it returns the next token (an integer token code).

**Handling end of file**

When yylex is finished, it call function yywrap(). If yywrap() returns 1, then yylex returns 0 to its caller. That means "end of file".

If yywrap returns 0, then yylex assumes that you have stored a different file into yyin, and it starts reading that file.

# Lex Program to count number of words

/\*lex program to count number of words\*/

%{

#include<stdio.h>

#include<string.h>

int i = 0;

%}

/\* Rules Section\*/

%%

([a-zA-Z0-9])\* {i++;} /\* Rule for counting

number of words\*/

"\n" {printf("%d\n", i); i = 0;}

%%

int yywrap(void){}

int main()

{

// The function that starts the analysis

yylex();

return 0;

}

# Lex program to count the number of lines, spaces and tabs

/\*lex code to count the number of lines,

tabs and spaces used in the input\*/

%{

#include<stdio.h>

int lc=0, sc=0, tc=0, ch=0; /\*Global variables\*/

%}

/\*Rule Section\*/

%%

\n lc++; //line counter

([ ])+ sc++; //space counter

\t tc++; //tab counter

. ch++; //characters counter

%%

int main()

{

// The function that starts the analysis

yylex();

printf("\nNo. of lines=%d", lc);

printf("\nNo. of spaces=%d", sc);

printf("\nNo. of tabs=%d", tc);

printf("\nNo. of other characters=%d", ch);

}

# Lex Program to Identify and Count Positive and Negative Numbers

**Examples:**

Input : -52

Output :

negative number=-52

number of positive numbers = 0

number of negative numbers = 1

Input : 63

Output :

positive number = 63

number of positive numbers = 0

number of negative numbers = 1

/\* Lex program to Identify and Count

Positive and Negative Numbers \*/

%{

int positive\_no = 0, negative\_no = 0;

%}

/\* Rules for identifying and counting

positive and negative numbers\*/

%%

^[-][0-9]+ {negative\_no++;

printf("negative number = %s\n",

yytext);} // negative number

[0-9]+ {positive\_no++;

printf("positive number = %s\n",

yytext);} // positive number

%%

/\*\*\* use code section \*\*\*/

int yywrap(){}

int main()

{

yylex();

printf ("number of positive numbers = %d,"

"number of negative numbers = %d\n",

positive\_no, negative\_no);

return 0;

}

# Lex program to identify the identifier

**Example:**

gfg : valid identifier

123 : invalid identifier

\_abc12 : valid identifier

#abc : invalid identifier

/\*lex code to determine whether input is an identifier or not\*/

% {

#include <stdio.h>

%

}

/ rule section % %

// regex for valid identifiers

^[a - z A - Z \_][a - z A - Z 0 - 9 \_] \* printf("Valid Identifier");

// regex for invalid identifiers

^[^a - z A - Z \_] printf("Invalid Identifier");

.;

% %

main()

{

yylex();

}