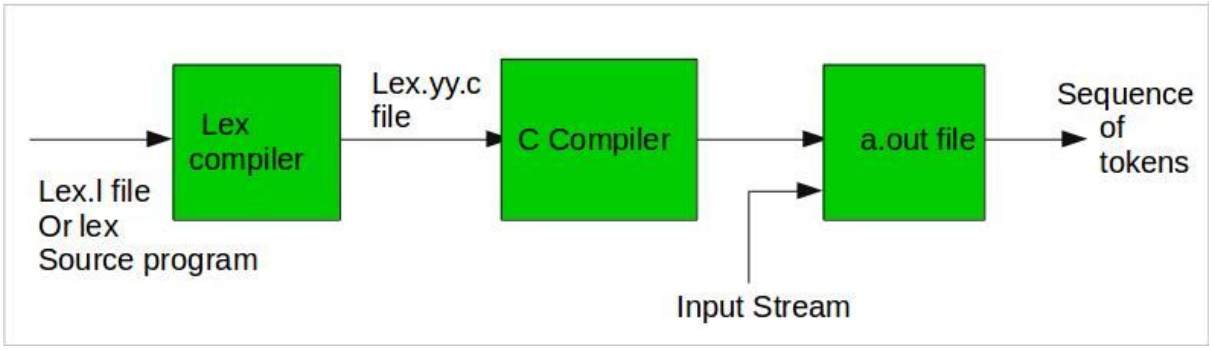
Flex (Fast Lexical Analyzer Generator)

FLEX (fast lexical analyzer generator) is a tool/computer program for generating lexical analyzers (scanners or lexers) written by Vern Paxson in C around 1987. It is used together with Berkeley Yacc parser generator or GNU Bison parser generator. Flex and Bison both are more flexible than Lex and Yacc and produces faster code.
Bison produces parser from the input file provided by the user. 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 which 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 complier 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

%%
```

Examples: Table below shows some of the pattern matches.

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 contain 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.

Example: Count the number of characters and number of lines in the input
/ Declaring two counters one for number of lines other for number of characters */*
%{
#include<stdio.h>
int no_of_lines = 0;
int no_of_chars = 0;
%}
*/***rule 1 counts the number of lines, rule 2 counts the number of characters and rule 3 specifies when to stop taking input***/*
%%
\n ++no_of_lines;
. ++no_of_chars;
end return 0;
%%
*/*** User code section***/*
int yywrap(){}
int main(int argc, char **argv)

```
{
yylex();
printf("number of lines = %d, number of chars = %d\n", no_of_lines, no_of_chars );
return 0;
}
```

The yyvariables: The following variables are offered by LEX to aid the programmer in designing sophisticated lexical analyzers. These variables are accessible in the LEX program and are automatically declared by LEX in *lex.yy.c*.

- **yyin:** yyin is a variable of the type FILE* and points to the input file. yyin is defined by LEX automatically. If the programmer assigns an input file to yyin in the auxiliary functions section, then yyin is set to point to that file. Otherwise LEX assigns yyin to stdin(console input).

- **yytext:** *yytext* is of type *char** and it contains the *lexeme* currently found. A **lexeme** is a sequence of characters in the input stream that matches some pattern in the Rules Section. (In fact, it is the first matching sequence in the input from the position pointed to by *yyin*.) Each invocation of the function *yylex()* results in *yytext* carrying a pointer to the lexeme found in the input stream by *yylex()*. The value of yytext will be overwritten after the next *yylex()* invocation.

- **yyleng:** *yyleng* is a variable of the type int and it stores the length of the lexeme pointed to by *yytext*.

The yyfunctions

- **yylex():** yylex() is a function of return type int. LEX automatically defines yylex() in lex.yy.c but does not call it. The programmer must call yylex() in the Auxiliary functions section of the LEX program. LEX generates code for the definition of yylex() according to the rules specified in the Rules section. When yylex() is invoked, it reads the input as pointed to by yyin and scans through the input looking for a matching pattern. When the input or a part of the input matches one of the given patterns, yylex() executes the corresponding action associated with the pattern as specified in the Rules section. In the above example, since there is no explicit definition of yyin, the input is taken from the console. If a match is found in the input for the pattern number, yylex() executes the corresponding action , i.e. return atoi(yytext). As a result yylex() returns the number matched. The value returned by yylex() is stored in the variable num. The value stored in this variable is then printed on screen using printf().

yylex() continues scanning the input till one of the actions corresponding to a matched pattern executes a return statement or till the end of input has been encountered. In case of the above example, yylex() terminates immediately after executing the rule because it consists of a return statement.

Note that if none of the actions in the Rules section executes a return statement, yylex() continues scanning for more matching patterns in the input file till the end of the file.

In the case of console input, yylex() would wait for more input through the console. The user will have to input ctrl+d in the terminal to terminate yylex(). If yylex() is called more than once, it simply starts scanning from the position in the input file where it had returned in the previous call.

Example:

```
/* Declarations */
%%
{number} {return atoi(yytext);}
%%
int main()
{
        int num = yylex();
        printf("Found: %d",num);
        return 1;
}
```

Sample Input/Output :
I: 42
O: Found: 42

- **yywrap():** LEX declares the function yywrap() of return-type int in the file lex.yy.c . LEX does not provide any definition for yywrap(). yylex() makes a call to yywrap() when it encounters the end of input. If yywrap() returns zero (indicating false) yylex() assumes there is more input and it continues scanning from the location pointed to by yyin. If yywrap() returns a non-zero value (indicating true), yylex() terminates the scanning process and returns 0 (i.e. "wraps up"). If the programmer wishes to scan more than one input file using the generated lexical analyzer, it can be simply done by setting yyin to a new input file in yywrap() and return 0.As LEX does not define yywrap() in lex.yy.c file but makes a call to it under yylex(), the programmer must define it in the Auxiliary functions section or provide %option noyywrap in the declarations section. This options removes the call to yywrap() in the lex.yy.c file. Note that, it is mandatory to either define yywrap() or indicate the absence using the %option feature. If not, LEX will flag an error.

  Example:

```
%{
        #include<stdio.h>
        char *file1;
%}
%%
[0-9]+  printf("number");
%%

int yywrap()
{
        FILE *newfile_pointer;
        char *file2="input_file_2.l";
        newfile_pointer = fopen("input_file_2.l","r");
        if(strcmp(file1,file2)!=0)
        {
                file1=file2;
                yyin = newfile_pointer;
                return 0;
        }
        else
                return 1;
}
int main()
{
        file1="input_file.l";
        yyin = fopen("input_file.l","r");
        yylex();
        return 1;
}
```

When yylex() finishes scanning the first input file, input_file.l yylex() invokes yywrap(). The above definition of yywrap() sets the input file pointer to input_file_2.l and returns 0 . As a result, the scanner continues

scanning in input_file_2.l . When yylex() calls yywrap() on encountering EOF of input_file_2.l, yywrap() returns 1 and thus yylex() ceases scanning.