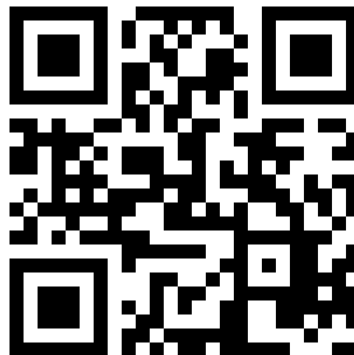
FUTURE VISION BIE

One Stop for All Study Materials & Lab Programs

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page



Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU, Currently for CSE – Computer Science Engineering...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

1. INTRODUCTION TO LEX

Lex and YACC helps you write programs that transforms structured input. Lex generates C code for lexical analyzer whereas YACC generates Code for Syntax analyzer. Lexical analyzer is build using a tool called LEX. Input is given to LEX and lexical analyzer is generated.

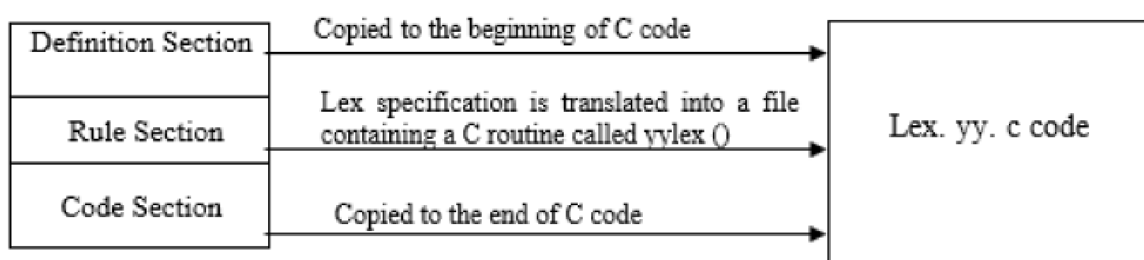
Lex is a UNIX utility. It is a program generator designed for lexical processing of character input streams. Lex generates C code for lexical analyzer. It uses the **patterns** that match **strings in the input** and converts **the strings** to tokens. Lex helps you by taking a set of descriptions of possible tokens and producing a C routine, which we call a lexical analyzer. The token descriptions that Lex uses are known as regular expressions.

1.1 Steps in writing LEX Program:

1st Step: Using gedit create a file with extension l. For example: prg1.l
2nd Step: lex prg1.l
3rd Step: cc lex.yy.c –ll
4th Step: ./a.out

1.2 Structure of LEX source program:

```
{definitions}
%%
{rules}
%%
{user subroutines/code section}
```



%% is a delimiter to the mark the beginning of the Rule section. The second %% is optional, but the first is required to mark the beginning of the rules. The definitions and the code /subroutines are often omitted.

Lex variables

yyin	Of the type FILE*. This points to the current file being parsed by the lexer.
yyout	Of the type FILE*. This points to the location where the output of the lexer will be written. By default, both yyin and yyout point to standard input and

	output.
yytext	The text of the matched pattern is stored in this variable (char*).
yyleng	Gives the length of the matched pattern.
yylineno	Provides current line number information. (May or may not be supported by the lexer.)

Lex functions

yylex()	The function that starts the analysis. It is automatically generated by Lex.
yywrap()	This function is called when end of file (or input) is encountered. If this function returns 1, the parsing stops. So, this can be used to parse multiple files. Code can be written in the third section, which will allow multiple files to be parsed. The strategy is to make yyin file pointer (see the preceding table) point to a different file until all the files are parsed. At the end, yywrap() can return 1 to indicate end of parsing.
yyless(int n)	This function can be used to push back all but first 'n' characters of the read token.
yymore()	This function tells the lexer to append the next token to the current token.

1.3 Regular Expressions

It is used to describe the pattern. It is widely used to in lex. It uses meta language. The character used in this meta language are part of the standard ASCII character set. An expression is made up of symbols. Normal symbols are characters and numbers, but there are other symbols that have special meaning in Lex. The following two tables define some of the symbols used in Lex and give a few typical examples.

Character	Meaning
A-Z, 0-9, a-z	Characters and numbers that form part of the pattern.
.	Matches any character except \n.
-	Used to denote range. Example: A-Z implies all characters from A to Z.
[ ]	A character class. Matches any character in the brackets. If the first character is ^ then it indicates a negation pattern. Example: [abC] matches either of a, b, and C.
*	Match zero or more occurrences of the preceding pattern.
+	Matches one or more occurrences of the preceding pattern.(no empty string). Ex: [0-9]+ matches "1","111" or "123456" but not an empty string.
?	Matches zero or one occurrences of the preceding pattern. Ex: -?[0-9]+ matches a signed number including an optional leading minus.
$	Matches end of line as the last character of the pattern.
{ }	1) Indicates how many times a pattern can be present. Example: A{1,3} implies one to three occurrences of A may be present. 2) If they contain name, they refer to a substitution by that name. Ex: {digit}
\	Used to escape meta characters. Also used to remove the special meaning of characters as defined in this table.

<table>
<tr><td></td><td>Ex: \n is a newline character, while “*” is a literal asterisk.</td></tr>
<tr><td>^</td><td>Negation.</td></tr>
<tr><td>|</td><td>Matches either the preceding regular expression or the following regular expression.
Ex: cow|sheep|pig matches any of the three words.</td></tr>
<tr><td>"< symbols>"</td><td>Literal meanings of characters. Meta characters hold.</td></tr>
<tr><td>/</td><td>Look ahead. Matches the preceding pattern only if followed by the succeeding expression. Example: A0/1 matches A0 only if A01 is the input.</td></tr>
<tr><td>( )</td><td>Groups a series of regular expressions together into a new regular expression.
Ex: (01) represents the character sequence 01. Parentheses are useful when building up complex patterns with *,+ and |</td></tr>
</table>

Examples of regular expressions

Regular expression	Meaning
joke[rs]	Matches either jokes or joker.
A{1,2}shis+	Matches AAshis, Ashis, AAshi, Ashi.
(A[b-e])+	Matches zero or one occurrences of A followed by any character from b to e.
[0-9]	0 or 1 or 2 or………9
[0-9]+	1 or 111 or 12345 or …At least one occurrence of preceding exp
[0-9]*	Empty string (no digits at all) or one or more occurrence.
-?[0-9]+	-1 or +1 or +2 …..
[0.9]*\.[0.9]+	0.0,4.5 or .31415 But won’t match 0 or 2

Examples of token declarations

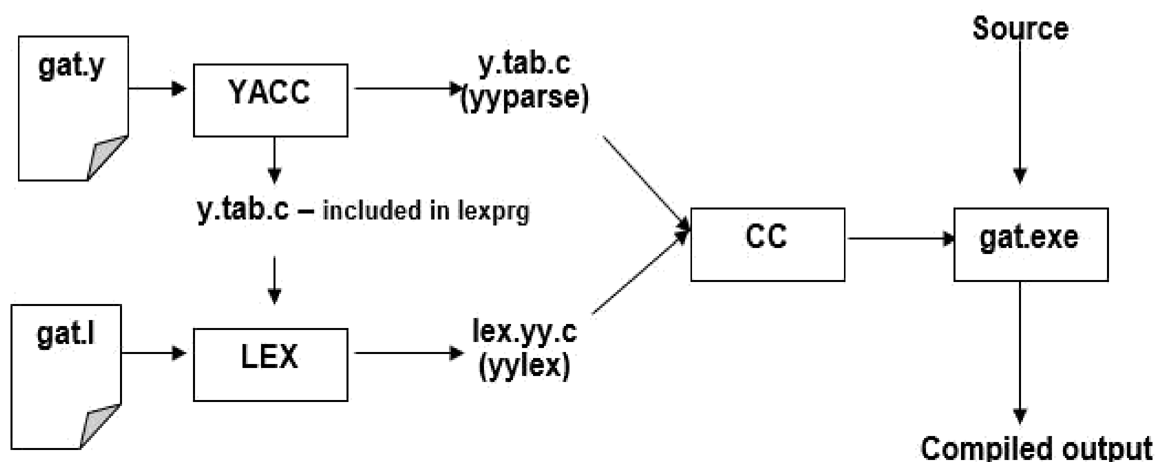
Token	Associated expression	Meaning
number	([0-9])+	1 or more occurrences of a digit
chars	[A-Za-z]	Any character
Blank	" "	A blank space
Word	(chars)+	1 or more occurrences of chars
Variable	(chars)+(number)*(chars)*(number)*	

2. INTRODUCTION TO YACC

YACC provides a general tool for imposing structure on the input to a computer program. The input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. YACC prepares a specification of the input process. YACC generates a function to control the input process. This function is called a parser.

The name is an acronym for “Yet Another Compiler Compiler”. YACC generates the code for the parser in the C programming language. YACC was developed at AT& T for the Unix operating system. YACC has also been rewritten for other languages, including Java, Ada.

The function parser calls the lexical analyzer to pick up the tokens from the input stream. These tokens are organized according to the input structure rules .The input structure rule is called as grammar. When one of the rule is recognized, then user code supplied for this rule (user code is action) is invoked. Actions have the ability to return values and makes use of the values of other actions.



2.1 Steps in writing YACC Program:

1st Step: Using gedit editor create a file with extension y. For example: gedit prg1.y
2nd Step: YACC –d prg1.y
3rd Step: lex prg1.l
4th Step: cc y.tab.c lex.yy.c -ll
5th Step: /a.out

When we run YACC, it generates a parser in file y.tab.c and also creates an include file y.tab.h. To obtain tokens, YACC calls yylex. Function yylex has a return type of int, and returns the token. Values associated with the token are returned by lex in variable yylval.

2.2 Structure of YACC source program:

Basic Specification:

Every YACC specification file consists of three sections. The declarations, Rules (of grammars), programs. The sections are separated by double percent "%%" marks. The % is generally used in YACC specification as an escape character.

The general format for the YACC file is very similar to that of the Lex file.

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

%% is a delimiter to the mark the beginning of the Rule section.

Definition Section

%union	It defines the Stack type for the Parser. It is a union of various datas/structures/ Objects
%token	These are the terminals returned by the yylex function to the YACC. A token can also have type associated with it for good type checking and syntax directed translation. A type of a token can be specified as %token <stack member>tokenName. Ex: %token NAME NUMBER
%type	The type of a non-terminal symbol in the Grammar rule can be specified with this.The format is %type <stack member>non-terminal.
%noassoc	Specifies that there is no associatively of a terminal symbol.
%left	Specifies the left associatively of a Terminal Symbol
%right	Specifies the right associatively of a Terminal Symbol.
%start	Specifies the L.H.S non-terminal symbol of a production rule which should be taken as the starting point of the grammar rules.
%prec	Changes the precedence level associated with a particular rule to that of the following token name or literal

Rules Section

The rules section simply consists of a list of grammar rules. A grammar rule has the form:

A: BODY

A represents a nonterminal name, the colon and the semicolon are YACC punctuation and BODY represents names and literals. The names used in the body of a grammar rule may represent tokens or nonterminal symbols. The literal consists of a character enclosed in single quotes.

Names representing tokens must be declared as follows in the declaration sections:

%token name1 name2…

Every name not defined in the declarations section is assumed to represent a non-terminal symbol. Every non-terminal symbol must appear on the left side of at least one rule. Of all the no terminal symbols, one, called the start symbol has a particular importance. The parser is designed to recognize the start symbol. By default the start symbol is taken to be the left hand side of the first grammar rule in the rules section.

With each grammar rule, the user may associate actions to be. These actions may return values, and may obtain the values returned by the previous actions. Lexical analyzer can return values for tokens, if desired. An action is an arbitrary C statement. Actions are enclosed in curly braces.

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
yytext
yyleng

3.1 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).

Example:

```
1
2      /* Declarations */
3   %%
4      /* Rules */
5   %%
6
7   main(int argc, char* argv[])
8   {
9           if(argc > 1)
10          {
11                  FILE *fp = fopen(argv[1], "r");
12                  if(fp)
13                          yyin = fp;
14          }
15          yylex();
16          return 1;
17  }
```

yyin_usage_example.c hosted with ♡ by **GitHub** **view raw**

Excercise:

In the generated *lex.yy.c* file, the following code segment can be found under the definition of *yylex()*.

```
if( ! yyin )
yyin = stdin;
```

Try to locate this code segment in the file *lex.yy.c*. What could be the consequences of removing this code segment from *lex.yy.c* before compiling it for generating the lexical analyzer? The above statement indicates that if the programmer does not define yyin, then yylex() by default sets *yyin* to the console input. Hence, any re-definition for *yyin* must be made before invoking *yylex()*. (This will be explained in detail later).

3.2 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.

Example:

```
1   %option noyywrap
2   %{
3           #include <stdlib.h>
```

```
4          #include <stdio.h>
5   %}
6
7   number [0-9]+
8
9   %%
10
11  {number} {printf("Found : %d\n",atoi(yytext));}
12
13  %%
14
15  int main()
16  {
17          yylex();
18          return 1;
19  }
```

yytext_example.l hosted with ♡ by **GitHub** **view raw**

In the above example, if a lexeme is found for the pattern defined by number then corresponding action is executed . Consider the following sample i/o,

Sample Input/Output:

```
I: 25
O: Found : 25
```

In this case when *yylex()* is called, the input is read from the location given by *yyin* and a string "25" is found as a match to 'number'. This location of this string in the memory is pointed to by *yytext*. The corresponding action in the above rule uses a built-in function *atoi()* to convert the string "25" (of type char*) to the integer 25 (of the type int) and then prints the result on the screen. Note that the header file "stdlib.h" is called in the auxiliary declarations section in order to invoke *atoi()* in the actions part of the rule.

NOTE: The lexeme found by LEX is stored in some memory allocated by LEX which can be accessed through the character pointer yytext.

NOTE: The *%option noyywrap* is used to inform the compiler that the function *yywrap()* has not been defined. We will see what this function does later on.

Exercise:

Suggest a modification in the above example to check whether a number found is even or odd.

3.3 yyleng

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

Example:

```
/* Declarations */
%%
/* Rules */
%%
{number} printf("Number of digits = %d",yyleng);
```

Sample Input/Output

```
I: 1234
O: Number of digits = 4
```

top ↑

The yyfunctions

yylex()
yywrap()

4.1 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.

NOTE: That yylex() need not necessarily be invoked in the Auxiliary Functions Section of LEX program when used with YACC.

Example:

```
1   /* Declarations */
2
3   %%
4
5   {number} {return atoi(yytext);}
6
7   %%
8
9   int main()
10  {
11          int num = yylex();
12          printf("Found: %d",num);
13          return 1;
14  }
```

yylex_usage_example.c hosted with ♡ by **GitHub** **view raw**

Sample Input/Output :

```
I: 42
O: Found: 42
```

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.

Exercise:

What would be the outputs of the lexical analyzer generated by the example LEX programs under section 3.2 and 4.1 for the following input :
25
32
44
Would both the outputs be the same? If not, explain why.

4.2 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:

```
1   %{
2           #include<stdio.h>
3           char *file1;
4   %}
5
6   %%
7
8   [0-9]+  printf("number");
9
10  %%
11
12  int yywrap()
13  {
14          FILE *newfile_pointer;
15          char *file2="input_file_2.l";
16          newfile_pointer = fopen("input_file_2.l","r");
17          if(strcmp(file1,file2)!=0)
18          {
19                  file1=file2;
20                  yyin = newfile_pointer;
21                  return 0;
22          }
23          else
24                  return 1;
25  }
26
27  int main()
28  {
29          file1="input_file.l";
30          yyin = fopen("input_file.l","r");
31          yylex();
32          return 1;
33  }
```

yywrap_usage_example.c hosted with ♡ by **GitHub** **view raw**

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.

Exercise:

Suggest a modification in the above example LEX program to make the generated lexical analyzer read input

-> Initially from the console and then from a file input_file.l
-> Initially from a file input_file.l and then from the console