A Short Tutorial on LEX and YACC

The LNM Institute of Information Technology

Praadeep Reddy

Raj Koushik Reddy

Contents

PREFACE

Objectives

This tutorial on LEX and YACC gives a brief introduction on how to use these tools for Lexical and Syntactical analysis of a given input. The structure and working of LEX and YACC programs has been discussed in this tutorial. Few examples have been used to explain the concepts more clearly to the reader .The material provided in this tutorial gives the reader enough insight to write a few basic programs in LEX and YACC and understand their usage.

Acknowledgements

We would like to thank Gaurav Somani Sir for inspiring and guiding us in writing this tutorial.

Prerequisites

Basic knowledge of C/C++ programming language.

A Linux machine installed with flex and bison packages. (Flex and Bison are open source versions of Lex and Yacc respectively).

LEX

Lex is a tool used for lexical analysis. Lex generates lexical analyzers. The main primary function of Lexical analyzer is scanning the source program or document and decomposing them into tokens. This process is called tokenization. Tokens are words which are defined to a particular pattern of symbols. Whenever a token is encountered corresponding to it some action can be taken. Lex is often used along a syntax analyzer such as yacc , which makes it a powerful tool for analyzing and interpreting a given input. This section concentrates on working of Lex as a standalone tool. The working of Lex along with Yacc is described in latter sections.

1.1 The basic structure of a Lex program

%{

Preprocessor Directives

Glogal Declarations

%}

Definitions

%%

Rules

%%

User code

1.2 Preprocessor Directives, Global declarations

In this section we can write global declarations and include preprocessor directives such as #include, #define, #if etc. This section follows same rules as defining them in C language. The code present in this part will be embedded as it is into the file "yylex.c", which is the C file generated after compilation of lex program.

Example

```
%{
#include <stdio.h>                    /* C global variables, directives etc */
#include <string.h>
    int a,b,c;
    char d[100];
%}
```

1.3 Definitions

Definitions are used for substitution of some patterns with text and using them in rules section. The word pattern represents occurrence of specified characters in certain sequence. It also contains some Meta characters for making pattern recognition easier. Given below is table 1.3.1 of Meta characters and how they are used to match patterns.

Metacharacter	**Matches**
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
$	end of line
a|b	a or b
(ab)+	one or more copies of ab (grouping)
"a+b"	literal "a+b"

[ ]	character class (atleast one of characters inside it)
\	literal meaning to next character (generally used for representing metacharacters)

Table 1.3.1 Metacharacters and what they match.

Given below is Table 1.3.2 of Examples for detecting of patterns using sequence of characters.

Sequence of characters	**Some of example Patterns matched**
Abc	Abc
.	A a b c 1 2 3 (any signle character except newline)
.a	1a 2a aa ba ca (any many)
abc*	ab abc abcc (ab for single time and c any number of times)
abc+	abc abcc (ab for single time and c atleast one time)
abc?	ab abc
a(bc)*	a abc abcbc
[abc]	One of : a, b, c
[abc]*	Any pattern of a,b,c (each of a,b,c can occur any number of times)
\.	.(literal occurrence of dot)
[a-z]	Any letter from :a-z (only in lower case)
[a\-z]	One of : a, -, z
[a-z]*	Any word of lower case english alphabets of any size
a|b	One of : a,b
[a|b]	One of : a, |, b

\t	Single white space
[\t]+	white space (any size)
\[	Occurrence of [
[A-Z]	Any letter from :A-Z (only in upper case)

Table 1.3.2 Detecting of patterns using sequence of characters.

Example

%}

word [a-z]+

%%

In the above example defines [a-z]+ pattern to as “word”. Now instead of using [a-z]+ we can use “word” in place of it . You might also get a doubt why use of this instead of pattern. The example pattern given above is simple that we can write it as many time as required but when we see some complex patterns like [a-zA-Z0-9\.\[\]\{\(\)\}\'\”\;] this it becomes difficult for us to write it repeatedly so we use definitions .

1.4: Rules

In Rules we define what action to take at occurrence of patterns.

%%

pattern action

%%

The pattern ends at the first non-escaped whitespace character after that comes action part.If the action is empty, then when the pattern is matched the input token is discarded.

Matching the input

If more than one pattern in the rules matches the given input lex behaves in the following manner

1. First the longest match is chosen and corresponding action is taken accordingly.

2. If two patterns match and have same length then the rule which is listed first is applied.

If no rule is matched then the next character copied to standard output

Example1:

Input: xyz

Rule: [a-z]+

In the above example “x” or ”y” or ”z” or ”xy” or “yz” or “xyz” match the rule pattern but token takes the longest matched pattern only.ie “‘xyz”.

Token: xyz

Example2:

Input: xyz

Rule1: “xyz”   printf (“Rule1 chosen,”);

Rule2: [a-z]+   printf (“Rule2 chosen”);

The output will print “Rule1 chosen” not “Rule2 chosen” even though both tokens match “xyz” because Rule1 comes first.

Example3 :

```
%%
[a-z] +         printf(“word found”);
%%
```

Whenever it finds a pattern which matches with [a-z]+ i.e. any word of lower case English alphabets. It prints “word found”.

The text it has scanned each time is stored in a lex string variable **yytext .**

Example4 :

```
%%
[a-z] +         printf(“%s”,yytext);
```

%%

If lex finds a pattern which matches with [a-z]+ it prints matched pattern.

1.5: User code

In this section user can write any c code which such as functions, declarations etc. which will be used as it is without any changes.

To invoke lexical analysis we call **yylex( )** function.

yylex( ) return value indicates type of token found

Example:

```
%%
int main(void) {
yylex();
return 0;
}
```

The above example takes input from standard input.

To take input from a file we use **yyin( )** function. This makes input pointer point to content of file.

```
%%
int main(void) {
yyin=fopen(filename,”r”);
yylex();
return 0;
}
```

Similar to **yyin( )** there is a **yyout( )** which helps us to output to a file.

```
%%
int main(void) {
yyout=fopen(filename,”w”);
yylex();
return 0;
}
```

The Declarations sections, definition section, rules, user code section are optional in a lex program. The only compulsory thing is “%%”.A simplest lex program can contain only “%%”.

1.6 Understanding Lex usage with few Examples

Example 1: ex1.lex

```
%{
#include <stdio.h>                                  /*Declarations, directives section*/
%}
word [a-zA-Z][a-zA-Z0-9]*                           /*Defining “word” to the given pattern*/
%%
[0123456789]+       printf("NUMBER\n");             /*Pattern followed by its respective action*/
 {word}             printf("WORD\n");
%%
                                                    /*user code section empty*/
```

Compilation

To compile lex program in terminal type

```
 lex ex1.lex
```

This generates a “lex.yy.c” file which is the lexical analyzer written in C language.

Then compile lex.yy.c

```
gcc lex.yy.c -o ex1 -ll
```

Here “-ll” is used for including default library main function as we don’t have main and yywrap( ) function in our user code section.

yywrap( ) function will be automatically called whenever lex encounters **EOF**. We can return 1 to stop analysis or can return 0 and make input pointer point to other input file. We can use this to take input from multiple files.

Executing ex1 after compilation

```
./ex1
```

Input the text using standard input and it displays output on standard output. The above example checks whether the given input is number or word.

Example 2: ex2.lex

```
%{
#include <stdio.h>
%}
word          [a-zA-Z][a-zA-Z0-9]*              /*Defining  word to the corresponding pattern*/
%%
[0123456789]+         fprintf(yyout,"NUMBER\n");                          /*pattern and action*/
{word}                      fprintf(yyout,"WORD\n");
%%
int main(int argc, char*argv[ ]) {
yyin=fopen(argv[1],"r");
yyout=fopen(argv[2],"w");
yylex();                                                        /*strating lexical analysis*/
return 0;
}
int yywrap()
{
   return 1;
}
```

Compilation

```
lex ex2.lex
gcc lex.yy.c -o ex2
./ex2 inputfile outputfile
```

Note: Here in this example we won’t use –ll while compiling because we have **yywrap()** and **main()** function used in our user code .

inputfile and outputfile are file names of input and output files respectively we are passing them as arguments to a c program.

Sample input file: **1.in**

```
Kpkr

123

1234kpkr
```

Sample output file: 1.out

```
WORD

NUMBER

NUMBER

WORD
```

Example showing how to take Input from Multiple files

As mentioned above we can take input from multiple files by returning 0 from yywrap( ) and make input pointer point to second input file. Given below is an example of such kind

Example3: ex3.lex

```
%{

#include <stdio.h>

int a=0;

%}

word [a-zA-Z][a-zA-Z0-9]*

%%

[0123456789]+          fprintf(yyout,"NUMBER\n");

{word}                 fprintf(yyout,"WORD\n");

%%

int main(int argc, char*argv[ ]) {

yyin=fopen(“1.in”,"r");

yyout=fopen(“1.out”,"w");
yylex();
return 0;
}
int yywrap()
{
if(a==0){
yyin=fopen("2.in","r");
a=1;
return 0;
}
return 1;
}
```

Sample input file1: 1.in

```
Kpkr
123
1234kpkr
```

Sample input file2: 2.in

```
111
123
```

Sample output file: 1.out

```
WORD
NUMBER
NUMBER
WORD
NUMBER
NUMBER
```

Example showing how to Output to Multiple files

Example4: ex4.lex

```
%{
#include <stdio.h>
int a=0;
%}
word [a-zA-Z][a-zA-Z0-9]*
%%
[0123456789]+          fprintf(yyout,"NUMBER\n");
{word}                 fprintf(yyout,"WORD\n");
%%
int main(int argc, char*argv[]) {
yyin=fopen(“1.in”,"r");
yyout=fopen(“1.out”,"w");
yylex();
return 0;
}
int yywrap()
{
if(a==0){
yyin=fopen("2.in","r");
yyout=fopen("2.out","w");
a=1;
return 0;
}
return 1;
}
```

Sample input file1: 1.in

```
Kpkr
123
```

1234kpkr

Sample input file2: 2.in

111

123

Sample output file1: 1.out

WORD

NUMBER

NUMBER

WORD

Sample output file2: 2.out

NUMBER

NUMBER

YACC: YET ANOTHER COMPILER COMPILER.

For doing syntax analysis (or parsing) of any program Yacc is used. Yacc specification file contains grammar rules for checking syntax. Yacc takes tokens as inputs which are passed by lex program. Yacc cannot function independently without a lexical analyzer.

2.1 The structure of yacc specification file.

%{ **Preprocessor Directives** **Glogal Declarations** **%}** **Definitions** **%%** **Rules** **%%** **User code**	**(OR)**	**%{** **Preprocessor Directives** **Glogal Declarations** **User code** **%}** **Definitions** **%%** **Rules**

The preprocessor directives, declaration section, user code of Yacc program follow same principles as Lex specification file.

2.2 Definitions:

The definition part we declare the tokens that are returned by Lex .We also have to specify data types for those tokens which carry values along with them.

The values can be retrieved using "$" symbol which will be discussed in rules section.

Example:

```
%{

#include<stdio.h>

int a,b,c;

void yyerror(const char* str)

{                                   /*checks for syntax errors in input*/
fprintf(stderr,”error: %s\n”,str);                    /* yyerror ( ) use is explained below */

}

main( ){

yyparse( ) ;                                          /*It starts the yacc rules section */

}

%}

%token TOKEN1 TOKEN2                                  /*Declaration of TOKENS */

%union

{

   int number;                                        /* number symbolizes int data type */

  char* string;                                       /*string symbolizes char* data type */

}

%token <string> TOKEN3       /*Declaring data type of TOKEN3 as string which is char* data type */

%token <number> TOKEN4    /*Declaring data type of TOKEN4 as number which is int data type */

%%
```

yyerror ( )

Whenever the syntax in the input file is not matched with the defined rules of yacc, it passes an error message **yyerror(**) function. In the above example whenever **yyerror( )** function is called it prints the error message on standard output.

2.2 Rules:

This section tells us how to understand the input and what actions are to be taken for each sentence. It is somewhat similar to rules section in the lex program. In lex file we specify sequence of characters as a pattern, but here we specify sequence of tokens, literal strings and other patterns.

Example:

```
 commands:

|commands TOKEN1
```

;

This says that commands pattern can be either empty or sequence of commands and TOKEN1

The inner commands again represent the same, so it matches with any of the patterns.

(Empty)

TOKEN1

TOKEN1 TOKEN1

TOKEN1 TOKEN1 TOKEN1

… … … … … … … … …

There can be infinite number of such patterns because it is in a loop.

Example: Use of literal strings

commands:

|commands ‘,’ TOKEN1

;

This says that commands pattern can be either empty or sequence of commands , literal string (,) and TOKEN1

The inner commands again represents the same so it matches with any of the patterns :

(empty)

, TOKEN1

, TOKEN1 , TOKEN1

, TOKEN1 , TOKEN1 , TOKEN1

… … … … … … … … …

There can be infinite number of such patterns because it is in a loop .

2.3 Compilation:

Compilation procedure is given after Example1. On compilation of Yacc program it generates three files **y.tab.h, y.tab.c, y.output.**

y.tab.h is the header file which contains the details of tokens declared in yacc file. This is included in the declaration section of lex file.

y.tab.c contains the C code equivalent of yacc program. This is similar to what **yylex.c** is to lex program.

2.4 Using Lex and Yacc in combination.

Example1.lex

```
%{

#include <stdio.h>

#include <string.h>

#include "y.tab.h"

%}

%%

logon|logoff              yylval.number=!strcmp(yytext,"logon"); return STATE;

message                  return MESSAGE;             /*Returns MESSAGE if “message” is encountered*/

send                     return SEND;                 /*Returns  SEND if “send” is encountered*/

[a-zA-Z0-9]+             yylval.string=strdup(yytext);return WORD; /*Returns WORD if any string is encountered*/


\n                        ;   /* ignore end of line */

[ \t]+                    ;  /* ignore whitespace */

%%
```

Example1.yacc

```
%{

#include <stdio.h>

#include <string.h>

int ln=0;

char* user;

void yyerror(const char *str)
{
        fprintf(stderr,"error: %s\n",str);
}

int yywrap()
{
        return 1;
}
main()
{
        yyparse();                          /*start parsing of input*/
}

%}
%token SEND MESSAGE
%union
{
        int number;
        char *string;
}

%token <number> STATE
%token <string> WORD

%%
commands:        /* empty */
        | commands  command
        ;

command:
        loge
        |
        SEND MESSAGE  {if(ln==1){printf("\t'%s' sent msg\n",user);}else{printf("please logon\n ");}}
        ;
loge:
        STATE WORD
        {
              if($1==1){                                                    /*$1 is the numerical value of STATE*/
              if(ln==1){printf("\t'%s' logged out\n",user);}
               printf("\t'%s' logged in\n",$2);
               ln=1;
               user=$2;
              }
              else{printf("\t'%s' logged out\n",$2);ln=0;}
        }
        ;
%%
```

Compilation

```
lex example1.lex
yacc –dv example1.yacc
cc lex.yy.c y.tab.c –o example1
./example1 <sample.in> sample.out
```

Input:

```
logon user1
send message
logoff user1
send message
logoff user2
logon user2
logon user3
```

Output:

```
'user1' logged in
        'user1' sent msg
        'user1' logged out
please logon
        'user2' logged out
        'user2' logged in
        'user2' logged out
        'user3' logged in
```

Working:

yylval is used to store some value in tokens .

In above example in yylval.number, number is the data type which we declared in yacc file

These values can be retrieved using $ followed by position of the token in the rule.

Lex Part:

Whenever lexical analyzer encounters logon or logoff string it returns a token named STATE and it also carries a value 0 or 1. If pattern is logon then the value is 1 and for logoff value is 0.

Yacc Part:

Using an integer constant "ln" which states the log of a user. If the user is logged on ln is set to 1 and 0 if logged off.

Note: You can give a value to a rule by assigning a value to "$$".

Example

```
%}
%union
{
    char *string;
}

%token <string> WORD
```

```
%type <string> str            /* declaring data type for a sequence of TOKENS to store values using “$$”*/
%%
commands: /* empty */
       | commands  command
       ;
command:str {printf(“”)}
;
str: WORD   {$$=$1;}

;
```

References

Lex - A Lexical Analyzer Generator   M. E. Lesk and E. Schmidt

S. C. Johnson, Yacc: Yet Another Compiler Compiler, Computing Science Technical Report No. 32, 1975, Bell Laboratories, Murray Hill, NJ 07974.

Lex and YACC primer/HOWTO PowerDNS BV (bert hubert <bert@powerdns.com>)
http://tldp.org/HOWTO

http://www.cse.iitk.ac.in/users/dsand/cs335/Mlex_tutorial.pdf

http://staff.science.uva.nl/~andy/compiler/yacc-intro.pdf