Jumpstart your Yacc...and Lex too!
An intro to Lex and Yacc

Ashish Bansal
Software Engineer, Sapient Corporation
November 2000

Lex and Yacc are two very important and powerful tools on UNIX. In fact, they are so powerful that building compilers for FORTRAN or C is child's play if you are fluent in Lex and Yacc. Ashish Bansal discusses these tools in sufficient detail for you to write your own language and its compiler! He covers regular expressions, declarations, matching patterns, variables, Yacc grammar, and parser code. At the end, he explains how to tie Lex and Yacc together.

Contents:

Lex stands for Lexical Analyzer. Yacc stands for Yet Another Compiler Compiler. Let's start with Lex.

Lex

Lex is a tool for generating scanners. Scanners are programs that recognize lexical patterns in text. These lexical patterns (or regular expressions) are defined in a particular syntax, which I will discuss in a minute.

A matched regular expression may have an associated action. This action may also include returning a token. When Lex receives input in the form of a file or text, it attempts to match the text with the regular expression. It takes input one character at a time and continues until a pattern is matched. If a pattern can be matched, then Lex performs the associated action (which may include returning a token). If, on the other hand, no regular expression can be matched, further processing stops and Lex displays an error message.

Lex and C are tightly coupled. A *.lex* file (files in Lex have the extension *.lex*) is passed through the lex utility, and produces output files in C. These file(s) are compiled to produce an executable version of the lexical analyzer.

Regular expressions in Lex

A regular expression is a pattern description using a meta language. 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.

Defining regular expressions in Lex

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.
?	Matches *zero or one* occurrences of the preceding pattern.
$	Matches end of line as the last character of the pattern.
{ }	Indicates how many times a pattern can be present. Example: A{1,3} implies one or three occurrences of A may be present.
\	Used to escape meta characters. Also used to remove the special meaning of characters as defined in this table.
^	Negation.
|	Logical OR between expressions.
"<some symbols>"	Literal meanings of characters. Meta characters hold.
/	Look ahead. Matches the preceding pattern only if followed by the succeeding expression. Example: A0/1 matches A0 only if A01 is the input.
( )	Groups a series of regular expressions.

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.

Tokens in Lex are declared like variable names in C. Every token has an associated expression. (Examples of tokens and expression are given in the following table.) Using the examples in our tables, we'll build a word-counting program. Our first task will be to show how tokens are declared.

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)*	

Programming in Lex
Programming in Lex can be divided into three steps:

1. Specify the pattern-associated actions in a form that Lex can understand.
2. Run Lex over this file to generate C code for the scanner.
3. Compile and link the C code to produce the executable scanner.

Note: If the scanner is part of a parser developed using Yacc, only steps 1 and 2 should be performed. Read the sections on Yacc and tying Lex and Yacc together for further help with this particular problem.

Now let's look at the kind of program format that Lex understands. A Lex program is divided into three sections: the first section has global C and Lex declarations, the second section has the patterns (coded in C), and the third section has supplemental C functions. main(), for example, would typically be found in the third section. These sections are delimited by %%. So, to get back to the word-counting Lex program, let's look at the composition of the various program sections.

Global C and Lex declarations
In this section we can add C variable declarations. We will declare an integer variable here for our word-counting program that holds the number of words counted by the program. We'll also perform token declarations of Lex.

Declarations for the word-counting program

```
%{
int wordCount = 0;
%}
chars [A-za-z\'\'\.\"]
numbers ([0-9])+
delim [" "\n\t]
whitespace {delim}+
words {chars}+
%%
```

The double percent sign implies the end of this section and the beginning of the second of the three sections in Lex programming.

Lex rules for matching patterns
Let's look at the Lex rules for describing the token that we want to match. (We'll use C to define what to do when a token is matched.) Continuing with our word-counting program, here are the rules for matching tokens.

Lex rules for the word-counting program

```
{words} { wordCount++; /*
increase the word count by one*/ }

{whitespace} { /* do
nothing*/ }

{numbers} { /* one may
want to add some processing here*/ }

%%
```

C code

The third and final section of programming in Lex covers C function declarations (and occasionally the main function) Note that this section has to include the yywrap() function. Lex has a set of functions and variables that are available to the user. One of them is yywrap. Typically, yywrap() is defined as shown in the example below. We'll explore this topic under Advanced Lex.

C code section for the word-counting program

```
void main()

{

yylex(); /* start the
analysis*/

printf(" No of words:
%d\n", wordCount);

}

int yywrap()

{

return 1;

}
```

In the preceding sections we've discussed the basic elements of Lex programming, which should help you in writing simple lexical analysis programs. In the Advanced Lex section we'll cover the functionality Lex provides, so that you can write more complex programs.

Putting it all together

The *.lex* file is Lex's scanner. It is presented to the Lex program as:

```
$ lex <file name.lex>
```

This produces the lex.yy.c file, which can be compiled using a C compiler. It can also be used with a parser to produce an executable, or you can include the Lex library in the link step with the option –ll.

Here are some of Lex's flags:

- ***-c* Indicates C actions and is the default.**
- ***-t* Causes the lex.yy.c program to be written instead to standard output.**
- ***-v* Provides a two-line summary of statistics.**
- ***-n* Will not print out the -v summary.**

Advanced Lex
Lex has several functions and variables that provide different information and can be used to build programs that can perform complex functions. Some of these variables and functions, along with their uses, are listed in the following tables. For an exhaustive list, please refer to the Lex or Flex manual (see Resources later in this article).

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.

This completes our discussion of Lex. Now let's move onto Yacc...

Yacc

Yacc stands for Yet Another Compiler Compiler. The GNU equivalent of Yacc is called Bison. It is a tool that translates any grammar that describes a language into a parser for that language. It is written in Backus Naur form (BNF). By convention, a Yacc file has the suffix .y. The Yacc compiler is invoked from the compile line as:

```
$ yacc <options>
<filename ending with .y>
```

Before going further, let's review what a grammar is. In the previous section, we saw that Lex recognizes tokens from an input sequence. If you are looking at a sequence of tokens, you may want to perform a particular action on an occurrence of this sequence. The specification of valid sequences in such cases is called a grammar. The Yacc grammar file contains this grammar specification. It also covers what you want to do when the sequence is matched.

To clarify this concept a bit further, let's take the English language for an example. The set of tokens might be: noun, verb, adjective, and so on. To form a grammatically correct sentence using these tokens, your construction must conform to certain rules. A simple sentence might be noun verb or noun verb noun. (I care. See spot run.)

So in our case, the tokens themselves come from the language (Lex), and the sequences allowed for these tokens (the grammar) is specified in Yacc.

There are four steps involved in creating a compiler in Yacc:

1. **Generate a parser from Yacc by running Yacc over the grammar file.**
2. **Specify the grammar:**
   - **Write the grammar in a .y file (also specify the actions here that are to be taken in C).**
   - **Write a lexical analyzer to process input and pass tokens to the parser. This can be done using Lex.**
   - **Write a function that starts parsing by calling yyparse().**
   - **Write error handling routines (like yyerror()).**
3. **Compile code produced by Yacc as well as any other relevant source files.**
4. **Link the object files to appropriate libraries for the executable parser.**

Writing a grammar in Yacc

As with Lex, a Yacc program is also divided into three sections separated by double percent signs. These are: declarations, grammar rules, and C code. We'll use an

Terminal and non-terminal symbols

Terminal symbol: Represents a class of syntactically equivalent tokens. Terminal symbols are of three types:

> *Named token*: These are defined via the *%token* identifier. By convention, these are all upper case.
>
> *Character token*: A character constant written in the same format as in C. For example, '+' is a character token.
>
> *Literal string token*: is written like a C string constant. For example, "<<" is a literal string token.

The lexer returns named tokens.

Non-terminal symbol: Is a symbol that is a group of non-terminal and terminal symbols. By convention, these are all lower case. In the example, file is a non-terminal while NAME is a terminal symbol.

example of parsing a file of the format name = age in years in order to illustrate the grammar specifications. We are assuming the file has multiple names and ages each separated by white space. As we look at each section of the Yacc program, we'll write a grammar file for our example case.

Declarations for C and Yacc
C declarations may define types and variables used in the actions, as well as macros. Header files may also be included. Each Yacc declaration part declares the names of both the terminal and non-terminal symbols (tokens), and may also describe operator precedence and data types for various symbols. The lexer (Lex) generally returns these tokens. All such tokens must be declared in the Yacc declarations.

In the file-parsing example we are interested in the tokens: name, equal sign, and age. Name is an all-character value. Age is numeric. So the declarations section would look like this:

Declarations for the file-parsing example

```
%

#typedef char* string; /*
to specify token types as char* */

#define YYSTYPE string /*
a Yacc variable which has the value of returned token */

%}

%token NAME EQ AGE

%%
```

The YYSTYPE may strike you as a bit odd. But like Lex, Yacc also has a set of variables and functions that are available to the user for extending functionality. YYSTYPE defines the type of yylval (another Yacc variable) used to copy values from the lexer to the parser or to Yacc. The default type is int. Since a string of characters will be copied from the lexer, the type has been redefined to char*. For a detailed discussion on Yacc variables, please refer to the Yacc manual (see Resources).

Yacc grammar rules
Yacc grammar rules take on the following general form:

```
result: components { /*
action to be taken in C */ }

;
```

In this example, result is the non-terminal symbol the rule describes. Components are various terminal and non-terminal symbols put together by the rule. Components can be followed by the action to be performed if that particular sequence is matched. Consider the following example:

```
param : NAME EQ NAME {
printf("\tName:%s\tValue(name):%s\n", $1,$3);}

    | NAME EQ VALUE{
    printf("\tName:%s\tValue(value):%s\n",$1,$3);}

;
```

If the sequence NAME EQ NAME is matched in the above example, the action in the corresponding { } brackets is taken. Another useful thing that comes into play here is the use of $1 and $3, which refer to the values of the tokens NAME and NAME (or VALUE for the second line). The lexer returns these values through a Yacc variable called yylval. Lex code for the token NAME would look like this:

```
char [A-Za-z]

name {char}+

%%

{name} { yylval = strdup(yytext);
return NAME; }
```

The rules section for our file-parsing example would look like this:

Grammar for the file-parsing

```
file : record file

| record

;

record: NAME EQ AGE {
printf("%s is now %s years old!!!", $1, $3);}

;

%%
```

Additional C code

Now let's move to the last section of the grammar file, the additional C code. (This section is optional, for those of you who want to skip it:) A function like main() calls the yyparse() function (the Yacc equivalent of Lex's yylex() function). Generally, Yacc expects that code for yyerror(char msg) also be provided. yyerror(char msg) is called whenever the parser encounters an error. The error message is passed as a parameter. A simple yyerror(char*) might look like this:

```
int yyerror(char* msg)

{

printf("Error: %s
encountered at line number:%d\n", msg, yylineno);

}
```

yylineno provides line number information.

The main function for our file-parsing example would also be included in this section:

Additional C code

```
void main()

{

    yyparse();

}

int yyerror(char* msg)

{

printf("Error: %s
encountered \n", msg);
```

To generate code, the following command may be used:

```
$ yacc –d <filename.y>
```

This generates the output files y.tab.h and y.tab.c and can be compiled using any standard C compiler on UNIX (gcc, for example).

Other common options that can be used on the command line

- ***'-d' ,'--defines'* : Write an extra output file containing macro definitions for: the token type names that are defined in the grammar, the semantic value type *YYSTYPE*, and a few external variable declarations. If the parser output file is named 'name.c', then the '-d' file is named 'name.h'. If you want to put the *yylex* definition in a separate source file you need 'name.h', because *yylex* has to be able to refer to token type codes as well as the *yylval* variable.**
- ***'-b file-prefix' ,'--file-prefix=prefix'* : Specify a prefix to use for all Yacc output file names. Chose the names as if the input file were named 'prefix.c'.**
- ***'-o outfile' ,'--output-file=outfile'* : Specify the name outfile for the parser file. The other**

output files' names are constructed from outfile as described under the '-d' options.

The Yacc library is usually automatically included in the compile step. But it can also be explicitly included to specify the *–ly* option during the compile step. In this case the compile command line would become:

```
$ cc <source file
names> -ly
```

Tying Lex and Yacc together
So far we've talked about Lex and Yacc separately. Now let's see how they can be used in conjunction with each other.

A program generally calls the *yylex()* function each time it returns a token. It stops doing this either at the end of the file or at an incorrect token.

A Yacc-generated parser calls *yylex()* to obtain tokens. *yylex()* can be generated by Lex or written from scratch. For the Lex-generated lexer to be used with Yacc, a token has to be returned every time a pattern is matched in Lex. The general form of action on matching a pattern in Lex would therefore be:

```
{pattern} { /* do smthg*/
return TOKEN_NAME; }
```

Yacc then obtains the returned tokens. When Yacc compiles a *.y* file with a –d option, a header file is generated, which has *#define* for each of the tokens. If Lex and Yacc are being used together, the header file must be included in the C declaration section of the corresponding Lex *.lex* file.

Let's get back to our name and age file-parsing example and look at the code for the Lex and Yacc files.

Name.y - The grammar file

```
%

typedef char* string;

#define YYSTYPE string

%}

%token NAME EQ AGE

%%

file : record file

| record
```

```
;

record : NAME EQ AGE {
printf("%s is %s years old!!!\n", $1, $3); }

;

%%

int main()

{

yyparse();

return 0;

}

int yyerror(char *msg)

{

printf("Error
encountered: %s \n", msg);

}
```

Name.lex - Lex file for the parser

```
%{

#include "y.tab.h"

#include <stdio.h>

#include <string.h>

extern char* yylval;

%}

char [A-Za-z]

num [0-9]

eq [=]
```

```
name {char}+

age {num}+

%%

{name} { yylval = strdup(yytext);
return NAME; }

{eq} { return EQ; }

{age} { yylval = strdup(yytext);
return AGE; }

%%

int yywrap()

{

return 1;

}
```

As a point of reference, let's list *y.tab.h*, the header file generated by Yacc.

y.tab.h - Yacc-generated header

```
# define NAME 257

# define EQ 258

# define AGE 259
```

This completes our discussion on Lex and Yacc. Which language do you want to compile today?

Resources

- **Lex and Yacc, Levine, Mason and Branson, O'Reilly and Associates Inc, 2nd Ed.**
- **Program Development in UNIX, J. T. Shen, Prentice-Hall India.**
- **Compilers: Principles, Techniques and Tools, Ahoo, Sethi and Ullman, Addison-Wesley Pub. Co., Nov, 1985.**
- **Bison Manual from Gnu.**
- **Flex Manual from Gnu.**
- **Info on Bison, (GNU's Yacc).**
- **MKS Lex and Yacc release notes.**

- **Tutorial on Lex and Yacc.**
- **Tutorials on Lex and Yacc and compiler writing.**
- **Tutorial on Java version of Lex, called Jlex.**
- **Example of formalizing a grammar for use with Lex & Yacc.**

About the author
Ashish Bansal has a bachelors degree in Electronics and Communications Engineering from the Institute of Technology, Banaras Hindu University, Varanasi, India. He is currently working as a software engineer with Sapient Corporation. He can be reached at abansal@sapient.com.