Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X
IMPACT FACTOR: 6.017

IJCSMC, Vol. 6, Issue. 7, July 2017, pg.77 – 82

Automatic Scanning and Parsing using LEX and YACC

Manju[1], Rajesh Kumar[2]

[1]Department of Computer Science & Applications, CRM Jat College, Hisar, Haryana, India
[2]Department of Computer Science & Applications, CRM Jat College, Hisar, Haryana, India
[1] duhan.manju@gmail.com; [2] rajtaya@kuk.ac.in

Abstract— This study is conducted in the area of automatic ''translator'' generator. The implementation of a basic LEX-style lexer generator or YACC- style parser generator requires only textbook knowledge. Lex is a lexical analyzer generator. Lex has been widely used to specify lexical analyzers for a variety of languages. Lex takes as input a stream of characters (regular expressions) that describe tokens, creates a Deterministic Finite Automata that recognizes that tokens, and then creates C code that implements that Deterministic Finite Automata. Yacc is a tool for building parsers. The specification of yacc is based on some set of rules called grammar that describe the syntax of a language, yacc turns the specification into a syntax analyzer. In this paper the author introduce the basic features and describe the essential design of LEX and YACC and explain how it can fit into Scheme.

Keywords— LEX, Parsing, Regular Expression, Scanning, YACC

I. INTRODUCTION

A compiler or interpreter for a programming language is often decomposed into two parts: First Read the source program and discover its structure. Second Process this structure, e.g. to generate the target program. Lex and Yacc can generate program fragments that solve the first task. The task of discovering the source structure again is decomposed into subtasks:

(a) Split the source file into tokens (Lex) shown in the figure 1.
(b) Find the hierarchical structure of the program (Yacc)

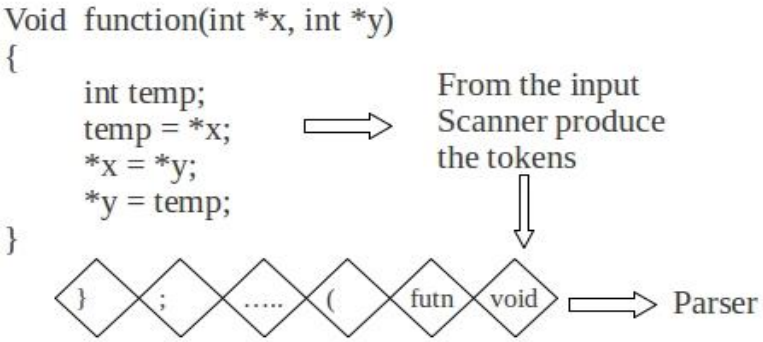

Figure: 1

In this paper the author refer to the tool as Lex Compiler, and to its input specification as the Lex language. Lex is generally used in the manner depicted in the Figure 2. First, a specification of a lexical analyzer is prepared by creating a program called lex.l in the lex language. Then, the lex.l is run through the Lex compiler to produce a C program lex.yy.c . The program lex.yy.c consists of a tabular representation of a transition diagram constructed from the regular expressions of the lex.l, together with a standard routine that uses the table to recognize lexemes. The actions associated with the regular expressions in lex.l are pieces of C code and are carried over directly to lex.yy.c. Finally, lex.yy.c is run through the C compiler to produce an object program which is the lexical analyzer that transforms the input stream into a sequence of tokens.

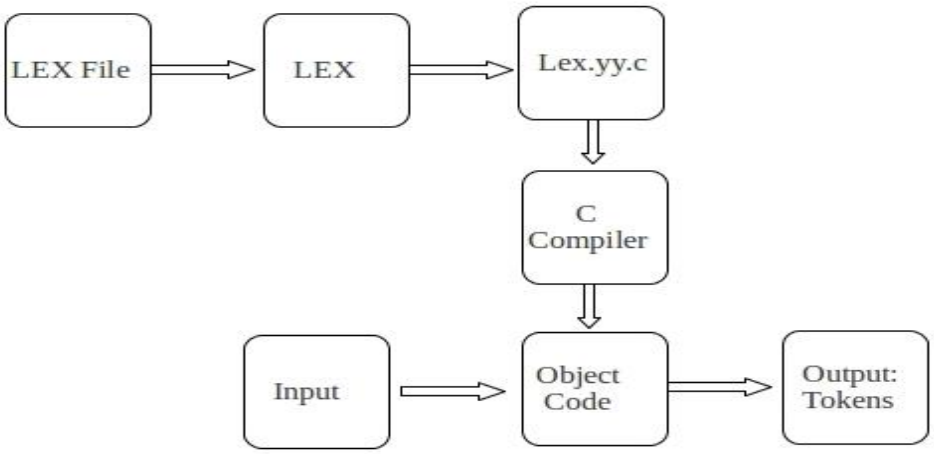

Figure: 2

A pure syntax analyzer merely checks whether or not an input string conforms to the syntax of the language. It can go beyond pure syntax analysis by attaching code in C or C++ to a grammar rule such code is called an action, and is executed whenever the rule is applied during syntax analysis. Thus, a desk calculator(discuss in next section) might use actions to evaluate an expression, and a compiler front end might use actions to emit intermediate code. Yacc allows us to automatically build parsers from LALR(1)[5] grammars without necessarily learning the underlying theory. The figure 3 describe how Yacc is used in combination with LEX.

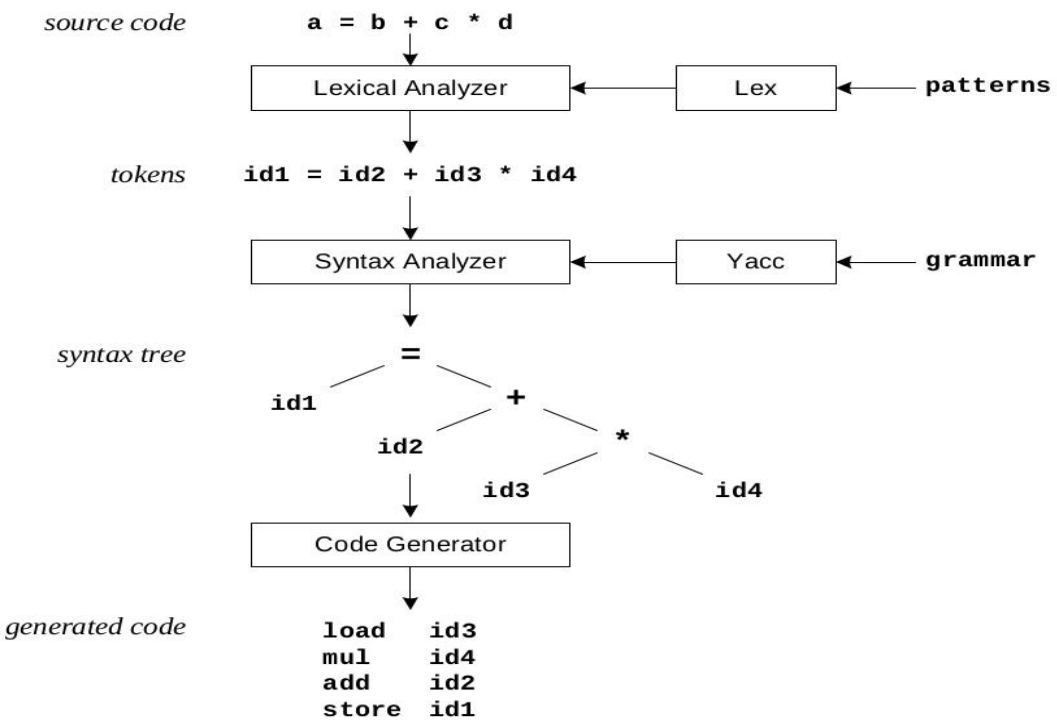

Figure: 3

II. PREVIOUS WORK

Lex & Yacc were developed at Bell Laboratories in the 70's .Yacc was developed as the first of the two by Stephen C. Johnson. Lex was designed by Mike E. Lesk and Eric Schmidt to work with Yacc. Lex [3], a tool for building lexical analyzers, works in harmony with yacc. It can be easily used to produce quite complicated lexical analyzers, but there remain some languages (Fortran, for example) whose lexical analyzers must be crafted by hand. Textbooks on compilers[4] provide more information on the behavior and construction of parsers than will be covered here. The algorithms underlying yacc are also discussed in a survey of LR parsing [5]. A feature that sets yacc apart - its ability to build fast compact LR parsers from ambiguous grammars -is based on the theory developed in [6]. Byacc was written around 1990 by Robert Corbett who is the original author of bison. Byacc is noted in Lex & Yacc by John Levine[7] for its compatibility with the original `yacc` program. Later, in Flex & Bison[7], John Levine states that because Corbett's implementation was faster than Bell (AT&T) `yacc`, and because it was distributed under the BSD license, "it quickly became the most popular version of yacc".

III. LEX IMPLEMENTATION

The brief explanation of lex file is discuss with the help of this below example :
File name say Lex.l

```
% %
" hello world"        printf(" Hello \n");
.                              ;
% %
```

This above program prints Hello anytime the string "hello world" is encountered. '.' does nothing for any other character. The following commands shows the behavior of above said program.

$ lex lex.l Process the lex file to generate a scanner and gets lex.yy.c file .
$ gcc lex.yy.c -ll Compile the scanner and grab main() from the lex library (-ll is option).
$ a.out Run the scanner taking input from standard input.

The author discuss how the existing LEX- and YACC-like systems for Scheme fit into the language. LEX works on some fundamentals of regular expressions. First, it reads the regular expressions which describe the languages that can be recognized by finite automata. Each token regular expression is then translated into a corresponding non- deterministic finite automaton (NFA) using Thompson's construction [2] . The NFA is then converted into an equivalent deterministic finite automaton (DFA). The DFA is then minimized to reduce the

number of states. Finally, the code driven by DFA[1] tables is emitted. Yacc/Bison source program specification (accept LALR grammars)[5] are described in next section.

IV. REGULAR EXPRESSIONS

Regular expressions allow us look for patterns in our data. The term "Regular Expression" (now commonly abbreviated to "RegExp" or even "RE") simply refers to a pattern that follows the rules of syntax outlined . Unix utilities such as sed and egrep use the same notation for finding patterns in text . Some example of regular expressions:

.	any character except newline
x	match the character 'x'
rs	the regular expression r followed by the regular expression s, called "concatenation"
r|s	either an r or an s
(r)	match an r, parentheses are used to provide grouping.
r*	zero or more r's, where r is any regular expression
r+	one or more r's
[xyz]	a "character class"; in this case, the pattern matches either an 'x', a 'y', or a 'z'.

These are the characters that are given special meaning within a regular expression, which can be used with the help of backslash:
. * ? + [ ] ( ) { } ^ $ | \
Any other characters automatically assume their literal meanings.

V. YACC DESCRIPTION

Yacc converts a context–free grammar and translation code into a set of tables for an LR(1) parser and translator. The grammar may be ambiguous, specified precedence rules are used to break ambiguities.
The output file of yacc is y.tab.c (C code that implements the parsing function int yyparse(void), yyparse returns 0 on a successful parse, and non-zero on a unsuccessful parse) must be compiled by the C compiler to produce a program yyparse. This program must be loaded with a lexical analyzer function, yylex (void). The yyerror() function is called by YACC if it finds an error. The function yywrap() can be used to continue reading from another file. It is called at EOF and another file can be opened, and return 0. Or it can return 1, indicating that this is truly the end. User can write their own main() function in YACC file, which calls yyparse() at one point. Yacc create yyparse() function and ends up in y.tab.c . A stream of token/value pairs is read by yyparse(), which needs to be supplied. The said function can be coded by user, or Lex can do it automatically.

VI. YACC ENVIRONMENT

YACC can parse input streams consisting of tokens with certain values. This clearly describes the relation YACC has with Lex, YACC has no idea what 'input streams' are, it needs preprocessed tokens .

YACC: **Y**et **A**nother **C**ompiler **C**ompiler (alternate software are available i.e)

- Bison: GNU Software

[1] ANTLR: **AN** other **T**ool for **L**anguage **R**ecognition

[2] Berkeley Yacc (**byacc**)

The specification of yacc.y file :

```
%{
 header          /*  (user supplied code within declarations */
%}
                 /*declarations */
%%
 Grammar rules
%%
 trailer         /*definition */
```


The above said format will be explained with the help of a simple calculator Program called cal.y.

TABLE I

```
/* Program Name :  Yacc.y */

%{
/* C declaration used in action */
void yyerror (char *s);
#include <stdio.h>
#include <stdlib.h>
int symbols[52];
int symVal(char symbol);
void updateSymVal(char symbol,
int val);
%}
%union {int num; char id;}
%start   line
%token  print
%token  exit_command
%token  <num> number
%token  <id> identifier
%type    <num> line exp term
%type    <id>  assignment
%%
```

```
/* inputs corresponding  actions*/
line : assignment ';'          {;}
|              exit_command              ';'
{exit(EXIT_SUCCESS);}
|              print          exp         ';'
{printf("printing %d\n",$2);}
| line assignment ';' {;}
|         line        print        exp        ';'
{printf("printing %d\n",$3);}
|line              exit_command              ';'
{exit(EXIT_SUCCESS);}
      ;
assignment
:           identifier          '='          exp
{updateSymVal($1,$3);}
      ;
exp  : term                    {$$ = $1;}
      | exp '+' term{$$ = $1 + $3;}
      | exp '-' term {$$ = $1 - $3;}
      ;
term : number      {$$ = $1;}
      | identifier    {$$ = symbolVal($1);}
       ;
%%
/*  c code */
Int SymIndex(char token)
{
           int idx = -1;
           if(islower(token))
           {
           idx = token - 'a' + 26;
           }
```

```
else if(isupper(token))
  {
           idx = token - 'A' ;
   }
           return idx;
}
int symbolVal(char symbol)
{
int                    bucket=
SymIndex(symbol);
return symbols[bucket];
}
voidupdateSymlVal(char
symbol, int val)
{
int          bucket          =
comSymIndex(symbol);
symbols[bucket] = val;
}
int main (void){
           int i;
           for(i=0; i<52; i++){
                      symbols[i] =
0;
}
           return yyparse ();
}
void yyerror (char *s) {fprintf
(stderr, "%s\n", s);}
```

When yacc is applied to above said specification, the output is a file of C code, called y.tab.c (the name might differ due to local file-system conventions).

In header part of a yacc specification includes C declaration and yacc definition (% start, %token, %union, %type). The declaration part represents a context free grammar. Action associated with each production rules are entered in braces. The third part contains valid C code that support the language processing, symbol table implementation and functions that might be called by actions associated with the productions in the second part.

A program say yacc.y can then be compiled into a file a.out by the following UNIX® system commands: yacc yacc.y -d (generates y.tab.c and header file y.tab.h) & gcc lex.yy.c y.tab.c (compiles executable program into file a.out).

VII. DEBUGGING LEX AND YACC

Lex has facilities that enable debugging. The code generated by lex in file lex.yy.c includeszdebugging statements that are enabled by specifying command-line option "-d". Debug output in flex (a GNU version of lex) may be toggled on and off by setting yy_flex_debug. Output includes the rule applied and corresponding matched text Yacc has facilities that enable debugging. The code generated by yacc in file y.tab.c includes debugging statements that are enabled by defining YYDEBUG and setting it to a non-zero value. This may also be done by specifying command-line option "-t". With YYDEBUG properly set, debug output may be toggled on and off by setting yydebug. Output includes tokens scanned and shift/reduce actions[5].

VIII. CONCLUSIONS

In this paper lax and yacc have been discussed. Lex and yacc produce the input to the parser. A parser consumes the output of a lexer, that produces a stream of terminals. Yacc expects the lexer to be a function of no

arguments (a thunk) that returns two values: the next terminal symbol, and the value of the symbol, which will be passed to the action associated with a production. At the end of the input, the lexer should return nil. So parser can be generated according to the languages. In case of Sanskrit, relations are critically examined from the point of view of feasibility of building a rule based parser. The first one, as is well known, as an almost exhaustive grammar for any natural language with meticulous details yet small enough to memorize

REFERENCES

[1] A. Aho, R. Sethi, and J. D. Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley, 1986.
[2] [K. Thompson. Programming Techniques: Regular expression search algorithm. Communications of the ACM, 11(6):419– 422, June 1968.
[3] Lesk, M.E. and Schmidt, M. Lex — A Lexical Analyzer Generator. In Unix Programmer's Manual, Tenth.
[4] Aho, A.V., Sethi, R., and Ullman, J.D. Compilers: Principles, Techniques, and Tools. Addison-Wesley.
[5] Aho, A.V. and Johnson, S.C. LR parsing. ACM Computing Surveys 6, 2 (1974), 99-124. Reading, Mass, 1986. Edition, AT&T Bell Laboratories, 1989.
[6] Aho, A.V., Johnson, S.C., and Ullman, J.D. Deterministic parsing of ambiguous grammars. CACM 18, 8 (1975), 441-452.
[7] Levine, John R., Tony Mason and Doug Brown [1992]. Lex & Yacc. O'Reilly & Associates, Inc. Sebastopol, California.

