Simple Calculator Compiler Using Lex and YACC

Mohit Upadhyaya
M.Tech Scholar, Department of Computer Science &Engg
Jodhpur National University
Jodhpur, India
e-mail: mohit.upadhyaya@gmail.com

***Abstract*— This paper contains the details of how one can develop the simple compiler for procedural lanaguage using Lex (Lexical Analyzer Generator) and YACC (Yet Another Compiler-Compiler). Lex tool helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-scripts type transformations and for segmenting input in preparation for a parsing routine. Lex tool source is the table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. On the other hand YACC tool receives input of the user grammar. Starting from this grammar it generates the C source code for the parser. YACC invokes Lex to scan the source code and uses the tokens returned by Lex to build a syntax tree. With the help of YACC and Lex tool one can write their own compiler**.

Keywords- compiler, lex, yacc, LALR, regular expressions, scanner, parser, optimization, pattern

I. INTRODUCTION

Before 1975 writing a compiler was a very time-consuming process. Then Lesk [1975] [6] and Johnson [1975] published papers on lex and yacc. These utilities greatly simplify compiler writing. Implementation details for lex and yacc may be found in Aho [1986]. Lex and yacc are available from.

- Mortice Kern Systems (MKS), at www.mks.com,
- GNU flex and bison, at www.gnu.org,
- Cygwin, at www.cygwin.com

Lex will read your patterns and generate C code for a lexical analyzer or scanner. The patterns in the diagram is a file you create with a text editor. Lex will read your patterns and generate C code for lexical analyzer or scanner. The lexical analyzer matches strings in the input, based on your patterns, and converts the strings to tokens. Tokens are numerical representation of strings, and simplify processing. When lexical analyzer finds identifiers in the input stream it enters them in a symbol table. The symbol table may also contain other information such as data type and location of each variable in memory. All subsequent references to identifiers refer to the appropriate symbol table index.

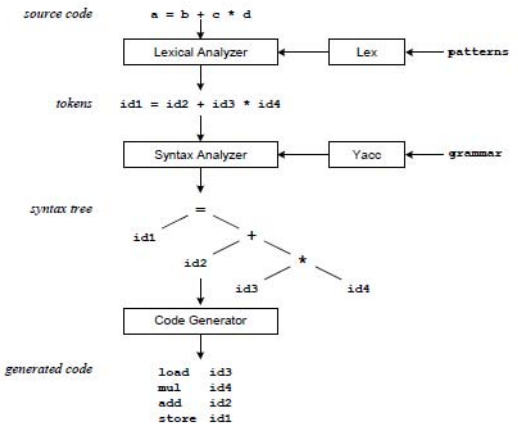

Figure 1: Compilation Sequence

Yacc will read your grammar and generate C code for a syntax analyzer or parser. The grammar in the above diagram is a text file you create with a text editor. The syntax analyzer uses grammar rules allow it to analyze tokens from the lexical analyzer and create a syntax tree. The syntax tree imposes a hierarchical structure of the tokens. For example, operator precedence and associativity are apparent in the syntax tree. The next step, code generation, does a depth-first walk of the syntax tree to generate code. Some compilers produce machine code, while others, as shown above, output assembly language code.

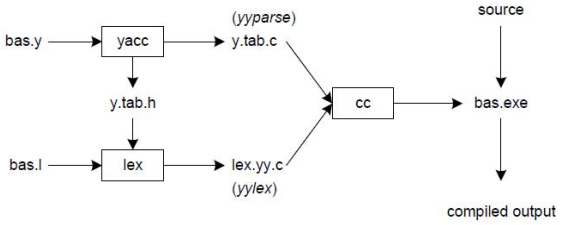

Figure 2: Building Compiler with Lex and Yacc

Figure 2 illustrates the file naming conventions used by lex and yacc [3]. We'll assume our goal is to write a Basic Compiler. First, we need to specify all pattern matching rules for lex (bas.l) and grammar rules for yacc (bas.y). Commands to create our compiler, bas.exe, are listed below:

```
yacc –d bas.y                 # create y.tab.h, y.tab.c
lex bas.l                     # create lex.yy.c
cc lex.yy.c y.tab.c –obas.exe # compile/link
```

978-1-4244-8679-3/11/$26.00 ©2011 IEEE

Yacc reads the grammar descriptions in bas.y and generates a syntax analyzer (parser), that includes function yyparse, in file y.tab.c. Included in file bas.y are token declarations. The –d option causes yacc to generate definitions for tokens and place them in file y.tab.h. Lex reads the pattern descriptions in bas.l, includes file y.tab.h, and generates a lexical analyzer, that includes function yylex, in file lex.yy.c. Finally, the lexer and parser are compiled and linked together to create executable bas.exe. From main we call yyparse to run the compiler. Function yyparse automatically calls yylex to obtain each token.

II. LEXICAL ANALYZER GENERATOR (LEX)

With regular expressions we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser. Initially we will simply print the matched string rather than return a token value. The following represents a simple pattern, composed of a regular expression that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

letter(letter|digit)*

This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

- repetition, expressed by the "*" operator
- alternation, expressed by the "|" operator
- concatenation

Any regular expression expressions may be expressed as a finite state automaton (FSA) [2]. We can represent an FSA using states, and transitions between states. There is one start state and one or more final or accepting states.This is the technique used by lex. Regular expressions are translated by lex to a computer program that mimics an FSA. Using the next input character and current state the next state is easily determined by indexing into a computer-generated state table [3].

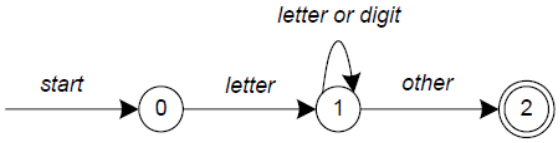

Figure 3: Finite State Automaton

Regular expressions in lex are composed of metacharacters (Table I)[3].

TABLE I. PATTERN MATCHING PRIMITIVES

Meta Character	Matches
.	Any character except new line
\n	Newline
*	Zero or more copies of the preceding expressin
+	One or more copies of the preceding expression
?	Zero or one copy of preceding expression
^	Beginning of line
$	End of line
a|b	a or b
(ab)+	One or more copies of **ab**
"a+b"	Literal "a+b"
[ ]	Character class

Within a character class normal operators lose their meaning. Two operators allowed in a character class are the hyphen ("-") and circumflex ("^"). When used between two characters the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string, the longest match wins. In case both matches are the same length, then the first pattern listed is used [2].

... definitions/declaration ...

%%

.. rules/translation rules ...

%%

... subroutines ...

Input to Lex is divided into three sections with %% dividing the sections. The declarations section includes declarations of variables, manifest constants (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions. The translation rules each have the form Pattern {Action}. Each pattern is a regular expression, which may use the regular definitions of the declarations section. The actions are fragments of code, typically written in C, although many variants of Lex using other languages have been created. The third section holds whatever additional functions are used in the actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyzer. Here is the example with defaults explicitly coded:

```
%%
        /* match everything except new line*/
        .       ECHO;
        /* match newline */
        \n      ECHO
%%
        int yywrap (void) {
                return 1;
        }
        int main (void) {
                yylex( );
```

```
    return 0;
}
```

Two patterns have been specified in the rules section. Each pattern must begin in column one. This is followed by whitespace (space, tab or newline) and an optional action associated with the pattern. The action may be a single C statement, or multiple C statements, enclosed in braces. Anything not starting in column one is copied verbatim to the generated C file. We may take advantage of this behavior to specify comments in our lex file. In this example there are two patterns, "." and "\n", with an ECHO action associated for each pattern. Several macros and variables are predefined by lex. ECHO is a macro that writes code matched by the pattern. This is the default action for any unmatched strings. Typically, ECHO is defined as:

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```

Variable *yytext* is a pointer to the matched string (NULL-terminated) and yyleng is the length of the matched string. Variable *yyout* is the output file and defaults to stdout. Function *yywrap* is called by lex when input is exhausted. Return 1 if you are done or 0 if more processing is required. Every C program requires a main function. In this case we simply call yylex that is the main entry-point for lex. Some implementations of lex include copies of main and *yywrap* in a library thus eliminating the need to code them explicitly. This is why our first example, the shortest lex program, functioned properly [3].

TABLE II. LEX PREDEFINED VARIABLES

Name	Function
int yylex(void)	Call to invoke lexer, returns token
char *yytext	Pointer to matched string
yyleng	Length of matched string
yylval	Value associated with token
int yywrap(void)	Wrapup, return 1 if done , 0 if not done
FILE *yyout	Output file
FILE *yyin	Input file
INITIAL	Initial start condition
BEGIN	Condition switch start condition
ECHO	Write matched string

III. YET ANOTHER COMPILER COMPILER (YACC)

Grammars for yacc are described using a variant of Backus Naur Form (BNF) [3]. This technique, pioneered by John Backus and Peter Naur, was used to describe ALGOL60. A BNF grammar can be used to express context-free languages. Most constructs in modern programming languages can be represented in BNF. For example, the grammar for an expression that multiplies and adds numbers is

$$E \rightarrow E + E \quad \text{- r1}$$

$$E \rightarrow E * E \quad \text{- r2}$$

$$E \rightarrow id \quad \text{-r3}$$

Three productions have been specified. Terms that appear on the left-hand side (lhs) of a production, such as E (expression) are nonterminals [2]. Terms such as id (identifier) are terminals (tokens returned by lex) and only appear on the right-hand side (rhs) of a production [2]. This grammar specifies that an expression may be the sum of two expressions, the product of two expressions, or an identifier. We can use this grammar to generate expressions:

$$E \rightarrow E * E \quad (r2)$$

$$\rightarrow E * z \quad (r3)$$

$$\rightarrow E + E * z \quad (r1)$$

$$\rightarrow E + y * z \quad (r3)$$

$$\rightarrow x + y * z \quad (r3)$$

At each step we expanded a term and replace the lhs of a production with the corresponding rhs. The numbers on the right indicate which rule applied. To parse an expression we need to do the reverse operation. Instead of starting with a single nonterminal (start symbol) and generating an expression from a grammar we need to reduce an expression to a single non-terminal, is known as bottom-up or shift-reduce parsing and uses a stack for storing terms. Here is the same derivation but in reverse order [2]:

i.	. x + y * z	shift
ii.	x . + y * z	reduce (r3)
iii.	E . + y * z	shift
iv.	E + . y * z	reduce (r3)
v.	E + E . * z	shift
vi.	E + E * . z	reduce (r3)
vii.	E + E * E .	reduce (r2) emit multiple
viii.	E + E.	reduce(r1) emit add
ix.	E.	accept

Terms to the left of the dot are on the stack while remaining input is to the right of the dot. We start by shifting tokens onto the stack. When the top of the stack matches the rhs of a production we replace the matched tokens on the stack with the lhs of the production. In other words the matched tokens of the rhs are popped off the stack, and the lhs of the production is pushed on the stack. The matched tokens are known as a handle and we are reducing the handle to the lhs of the production. This process continues until we have shifted all input to the stack and only the starting non-terminal remains on the stack. In step 1 we shift the x to the stack. Step 2 applies rule r3 to the stack to change x to E. We continue shifting and reducing until a single non-terminal, the start symbol, remains in the stack. In step 9, when we reduce rule r2, we emit the multiply instruction. Similarly the add instruction is emitted in step 10. Consequently multiply has a higher precedence than addition. Input to yacc is divided into three sections. The definitions section consists

of token declarations and C code bracketed by "%{" and "%}". The BNF grammar is placed in the rules section and user subroutines are added in the subroutines section.

```
... definitions ...

%%

... rules ...

%%

... subroutines ...
```

IV. DESIGN SIMPLE CALCULATOR COMPILER

In this part we described some simple methodology of designing a calculator using lex and yacc tool to understand their relationship and easiness of these tool. Initially take the basic functionality of calculator which is addition and subtraction. So the line of code to yacc which identify the type of value being defined as:

```
%token          INTEGER
```

This definition declares an INTEGER token [3]. Yacc generates a parser in file y.tab.c and an include file, y.tab.h:

```
#ifndef YYSTYPE

# define YYSTYPE int

# endif

# define INTEGER 258

 extern YYSTYPE yylval;
```

Lex includes this file and utilizes the definitions for token values. To obtain tokens yacc calls yylex [3]. Function yylex has a return type of int that returns a token. Values associated with the token are returned by lex in variable yylval. For example,

```
[0-9] + {

            yylval = atoi(yyytext);

            return INTEGER);

            }
```

would store the value of the integer in yylval, and return token INTEGER to yacc. The type of yylval is determined by YYSTYPE. Since the default type is integer this works well in this case. Token values 0-255 are reserved for character values. For example, if you had a rule such as

```
[-+]        return *yytext;     /* return operator */
```

The character value for minus or plus is returned. Note that we placed the minus sign first so that it wouldn't be mistaken for a range designator [3]. Generated token values typically start around 258 because lex reserves several values for end-of-file and error processing. Here is the complete lex input specification for our calculator [3]:

```
%{
        #include <stdlib.h>
        #include "calculator.h"
        #include "y.tab.h"
        void yyerror(char *);
%}

%%
[a-z]     {
            yylval.sIndex = *yytext - 'a';
            return VARIABLE;
          }
0         {
            yylval.iValue = atoi(yytext);
            return INTEGER;
          }

[1-9][0-9]*         {
                            yylval.iValue = atoi(yytext);
                            return INTEGER;
                    }

[-()<>=+*/;{}.]    {
                            return *yytext;
                    }
">="            return GE;
"<="            return LE;
"=="            return EQ;
"!="            return NE;
"while"         return WHILE;
"if"            return IF;
"else"          return ELSE;
"print"         return PRINT;

[ \t\n]+        ;       /* ignore whitespace */
.               yyerror("Unknown character");
%%
int yywrap(void) {
   return 1;
}
```

Internally yacc maintains two stacks in memory; a parse stack and a value stack [1,3]. The parse stack contains terminals and non-terminals that represent the current parsing state. The value stack is an array of YYSTYPE elements and associates a value with each element in the parse stack. For example when lex returns an INTEGER token yacc shifts this token to the parse stack. At the same time the corresponding yylval is shifted to the value stack. The parse and value stacks are always synchronized so finding a value related to a token on the stack is easily accomplished. Here is the yacc input specification for our calculator [3]:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "calculator.h"

/* prototypes */
```

```
nodeType *opr(int oper, int nops, ...);
nodeType *id(int i);
nodeType *con(int value);
void freeNode(nodeType *p);
int ex(nodeType *p);
int yylex(void);
void yyerror(char *s);
int sym[26];                    /* symbol table */
%}
%union {
    int iValue;                 /* integer value */
    char sIndex;                /* symbol table index */
    nodeType *nPtr;             /* node pointer */
}
%token <iValue> INTEGER
%token <sIndex> VARIABLE
%token WHILE IF PRINT
%nonassoc IFX
%nonassoc ELSE
%left GE LE EQ NE '>' '<'
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS
%type <nPtr> stmt expr stmt_list

%%
program:
        function                { exit(0); }
        ;
function:
          function stmt         { ex($2); freeNode($2); }
        | /* NULL */
        ;
stmt:
          ';'                   { $$ = opr(';', 2, NULL, NULL); }
        | expr ';'              { $$ = $1; }
        | PRINT expr ';'        { $$ = opr(PRINT, 1, $2); }
        | VARIABLE '=' expr ';'         { $$ = opr('=', 2, id($1), $3); }
        | WHILE '(' expr ')' stmt       { $$ = opr(WHILE, 2, $3, $5); }
        | IF '(' expr ')' stmt %prec IFX { $$ = opr(IF, 2, $3, $5); }
        | IF '(' expr ')' stmt ELSE stmt { $$ = opr(IF, 3, $3, $5, $7); }
        | '{' stmt_list '}'             { $$ = $2; }
        ;
stmt_list:
          stmt                  { $$ = $1; }
        | stmt_list stmt        { $$ = opr(';', 2, $1, $2); }
        ;
expr:
          INTEGER               { $$ = con($1); }
        | VARIABLE              { $$ = id($1); }
        | '-' expr %prec UMINUS { $$ = opr(UMINUS, 1, $2); }
        | expr '+' expr         { $$ = opr('+', 2, $1, $3); }
        | expr '-' expr         { $$ = opr('-', 2, $1, $3); }
        | expr '*' expr         { $$ = opr('*', 2, $1, $3); }
        | expr '/' expr         { $$ = opr('/', 2, $1, $3); }
        | expr '<' expr         { $$ = opr('<', 2, $1, $3); }
        | expr '>' expr         { $$ = opr('>', 2, $1, $3); }
        | expr GE expr          { $$ = opr(GE, 2, $1, $3); }
        | expr LE expr          { $$ = opr(LE, 2, $1, $3); }
        | expr NE expr          { $$ = opr(NE, 2, $1, $3); }
        | expr EQ expr          { $$ = opr(EQ, 2, $1, $3); }
        | '(' expr ')'          { $$ = $2; }
        ;
%%
#define SIZEOF_NODETYPE ((char *)&p->con - (char *)p)
nodeType *con(int value) {
    nodeType *p;
    size_t nodeSize;
    /* allocate node */
    nodeSize = SIZEOF_NODETYPE + sizeof(conNodeType);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");
    /* copy information */
    p->type = typeCon;
    p->con.value = value;
    return p;
}
nodeType *id(int i) {
    nodeType *p;
    size_t nodeSize;
    /* allocate node */
    nodeSize = SIZEOF_NODETYPE + sizeof(idNodeType);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");

    /* copy information */
    p->type = typeId;
    p->id.i = i;

    return p;
}
nodeType *opr(int oper, int nops, ...) {
    va_list ap;
    nodeType *p;
    size_t nodeSize;
    int i;

    /* allocate node */
    nodeSize = SIZEOF_NODETYPE + sizeof(oprNodeType) +
        (nops - 1) * sizeof(nodeType*);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");

    /* copy information */
    p->type = typeOpr;
    p->opr.oper = oper;
    p->opr.nops = nops;
    va_start(ap, nops);
    for (i = 0; i < nops; i++)
        p->opr.op[i] = va_arg(ap, nodeType*);
    va_end(ap);
    return p;
}
void freeNode(nodeType *p) {
    int i;

    if (!p) return;
    if (p->type == typeOpr) {
        for (i = 0; i < p->opr.nops; i++)
            freeNode(p->opr.op[i]);
    }
    free (p);
}
```

```
void yyerror(char *s) {
   fprintf(stdout, "%s\n", s);
}
int main(void) {
   yyparse();
   return 0;
}
```

The rules section resembles the BNF grammar discussed earlier. The left-hand side of a production, or nonterminal, is entered left-justified and followed by a colon. This is followed by the right-hand side of the production. Actions associated with a rule are entered in braces. With left-recursion, we have specified that a program consists of zero or more expressions. Each expression terminates with a newline. When a newline is detected we print the value of the expression. When we apply the rule

expr: expr '+' expr {$$ = $1 + $3;}

we replace the right-hand side of the production in the parse stack with the left-hand side of the same production. In this case we pop "expr '+' expr" and push "expr". We have reduced the stack by popping three terms off the stack and pushing back one term. We may reference positions in the value stack in our C code by specifying "$1" for the first term on the right-hand side of the production, "$2" for the second, and so on. "$$" designates the top of the stack after reduction has taken place. The above action adds the value associated with two expressions, pops three terms off the value stack, and pushes back a single sum. As a consequence the parse and value stacks remain synchronized.

Numeric values are initially entered on the stack when we reduce from INTEGER to expr. After INTEGER is shifted to the stack we apply the rule

expr: INTEGER {$$ = $1;}

The INTEGER token is popped off the parse stack followed by a push of expr. For the value stack we pop the integer value off the stack and then push it back on again. In other words we do nothing. In fact this is the default action and need not be specified. Finally, when a newline is encountered, the value associated with expr is printed. In the event of syntax errors yacc calls the user-supplied function yyerror. If you need to modify the interface to yyerror then alter the canned file that yacc includes to fit your needs. The last function in our yacc specification is main … in case you were wondering where it was. Parentheses may be used to over-ride operator precedence, and single-character variables may be specified in assignment statements. We may specify %left, for left-associative or %right for right associative [2]. The last definition listed has the highest precedence. Consequently multiplication and division have higher precedence than addition and subtraction. All four operators are left-associative. Using this simple technique we are able to disambiguate our grammar. The %nonassoc indicates no associativity is implied. It is frequently used in conjunction with %prec to specify precedence of a rule. The following illustrates sample input and calculator output:

```
user  : 3* (4+5)
calc  : 27
user  : x = 3* (4+5)
user  : y = 5
user  : x
calc  : 27
user  : y
calc  : 5
user  : x + 2*y
calc  : 37
```

V. CONCLUSION

With the lex and yacc tool one can create its own compiler, wherever one required. It is basically procedural language compiler tools and to support object oriented one need to work on structure of C language to support object oriented which makes the compiler quite complex. To use lex and yacc on UNIX is easy as compared to other operating system. Gcc is the basic compiler to generate the executable from lex and yacc compiled files. By studying these tools one can understand the basic structure of the compiler designed in C and go forward from it.

REFERENCES

[1] Ruoming Pang, Vern Paxson, Robin Sommer, Larry Peterson "binpac: A yacc for Writing Application Protocol parsers", Proceedings of the 6th ACM SIGCOMM conference on Internet measurement , ACM New York, NY, USA, 2006 pp 289-300.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, "Compiler Principles, Technique, Tools", Addison-Wesley, Reading, Massachusetts, pp 432-600.

[3] Tom Neimann, "A Compact Guide To Lex & Yacc", at epaperpress.

[4] Dick Grune, Henri E. Bal, Ceriel J. H. Jacobs," Modern Compiler Design", Vrije University, Amsterdam and Koen G. Langendoen: Delft University.

[5] S.C. Johnson,"Yacc: Yet another compiler compiler," *tech. rep.,* Bell Telephone Laboratories, 1975.

[6] M.E. Lesk and E. Schmidt,"Lex_a lexical analyzer generator," *tech. rep.,* Bell Telephone Laboratories, 1975.