LEX: LEXICAL ANALYZER GENERATOR

Compiled by
Dr. R. Leela Velusamy
Associate Professor

INTRODUCTION:

Lex takes specifications of character strings (regular expressions) as input and generates a program in `C' that recognizes regular expressions. Associated with a regular expression, an action may also be specified. The action is a code in `C'. When a regular expression is recognized, the corresponding action will be performed. i.e. the corresponding `C' code will be executed.

The program generated by LEX is named yylex. The yylex Program takes a stream of characters as input, recognizes it as a stream of various tokens as specified by the regular expressions given in the lex source.

r.e. specifications -----> lex ------> yylex

input -----> yylex -----> output

The sequence of UNIX commands from lex source to output is:

vi source

lex source

cc lex.yy.c -ll

a.out

LEX SOURCE

The general format of Lex source is

{definitions}
%%
{Rules}
%%
{user subroutines}

The absolute minimum Lex program is

%% which copies the input to the output unchanged.

LEX REGULAR EXPRESSIONS

The operator characters are "\[]^-?'*+|()$/{}%<>

eg. 1. To replace all `a's in the input stream by `A'

```
%%
a   printf ("A");
%%
```

(or)

```
%%
"a" printf ("A");
%%
```

Note: All the strings in the input, which are not matched, will be copied to the output. i.e. the input "It is a cat" will result in "It is A cAt"

2. The operators other than alphanumeric characters should be preceded by \ eg. 2 To suppress all spaces & tabs in the input.

```
%%
" “;
"\t";
%%
```

CHARACTER CLASSES:

Classes of characters can be specified using the operator pair []. eg. 3 To suppress the sign characters + and -.

```
%%
[+-] ;
%%
```

The construction [+ -] matches either + or -. i.e. the input +3-5 will produce 1235.

Note: The characters \, - and ^ are special within the square brackets.

eg. 4: To recognize lower and uppercase alphabets.

```
%%
```

```
[a-zA-Z]
%%
```

The minus character `-' is used to indicate range within square brackets.

Note: If it is desired to include the character - in a character class, it should be first or last. i.e [- + 0-9] matches all the digits and the two signs.

e.g. 5: To match all characters except "c", "s" or "e".

```
%%
[^cse]
%%
```

Note the ^ operator must appear as the first character within square brackets.

eg. 6 To recognize all non-alphabet characters

```
%%
[^a-zA-Z]
%%
```

eg.7: To match any character except new line

```
%%
.
%%
```

eg.8: To match ASCII characters from octal 40 to octal 176.

```
%%
[\40-\176]
%%
```

Optional Expressions:

The operator? indicates an optional element of an expression.

eg. 9: To match a digit or a sign followed by a digit.

```
%%
[+-]?[0-9]
%%
```

Repeated Expressions:

The operator * is used to indicate that zero or more instances of an element.

The operator + is used to indicate that one or more instances of an element.

eg.10: To recognize all strings of upper case letters.

```
%%
[A-Z]+   printf("\n a string of upper case letter
                has been found \n");
%%
```

eg.11: To recognize an identifier.

```
%%
[A-Za-z][A-Za-z0-9]*
%%
```

Alteration:

The operator | (vertical bar) is used to indicate alteration.

eg.12: To replace "sem" or "semester" by "year".

```
%%
sem|semester    printf ("year");
%%
```

Grouping

The parentheses are used for grouping.

eg.13:

To recognize even number of b's which may (or may not) be preceded by either a string of alphabets or a string of digits.

```
%%
([a-zA-Z]+|[0-9]+)?(bb)*
%%
```

The above matches' strings such as "bbbb", "csebb", "bbb" "12bbbb" but not "a12bb".

Context Sensitivity:

If the first character of an expression is ^, the expression is only matched at the beginning of a line. If the last character is $1 the expression is only matched at the end of a line.

The / operator is used to indicate trailing context.

eg. 14: To replace ";" by "." if it appears at the end of a line.

```
%%
";"$   printf (".");
%%
```

The above will produce a++; b=b+c. for the input

a++;b=b+c;

eg.15: To identify integral are fractional part of a fixed-point number.

```
%%
[0-9]+/"."  printf ("\n integral part: %s", yytext);
[0-9]+      printf ("\n fractional part: %s", yytext);
"."         ;
%%
```

If the input is 721.19, the above will produce the output

integral part : 721

real part : 19

Note: The string of matched characters is available in a character array, yytext [].

Definitions: The curly braces {} are used to enclose a predefined name. The name is defined in the definition part of the lex source.

eg. 16: To recognize integers.

```
digit  [0-9]

%%
{digit}+       printf ("integer");
%%
```

eg. 17: To recognize real numbers.

```
D       [0-9]
E       [DEde][-+]?{D}+
%%
{D}+"."{D}*({E})?|
{D}*"."{D}+({E}))?|
{D}+{E}          printf ("real");
%%
```

Note: A vertical bar is used in the first two rules. The meaning is, the action specified for the third rule is applicable for the first two rules also.

Repetitions:

eg. 18: To recognize 2 to 3 consecutive occurrence of "a".

```
%%
a{2,3}    printf ("\n a has occurred twice or thrice \n);
%%
```

eg. 19: To recognize a word in which three a's are followed by two b's.

```
%%
(a{3})(b{2})  printf ("\n aaabb has been found \n");
%%
```

eg.20: To count number of alphanumeric words in the input.

```
alphanum   [a-zA-Z0-9]

int  w=0;
%%
{alphanum}+w+f;
\n      printf ("\n No of words : %d\n", w);
%%
```

eg.21: To list the alphanumeric words and the number of characters in them.

```
    alphanum    [a-zA-Z0-9]
    int  i=0;
%%
{alphanum} + {i++;  printf ("\n%d.%s %d", i, yytext,yyleng);}
\n     exit ();
%%
```

Note: 1. yyleng gives the number of characters in matched string.

2. The declaration must not start at column 1.

eg.22: To recognize a string of characters between quotation marks (").

To include a " in a string. it must be preceded by a \.

```
%%
\"[^"]*  {
          if (yytext [yyleng-1] == `\\`)
            yymore();
          else
             printf ("\n %s \n", yytext+1);
        }
%%
```

The input "result = " yields result =

The input "We can include \" in a string " yields we can include " in a string.

Note: yymore() can be called to indicate that the next expression recognized is to be taken to the end of this input.

eg.23: A fortran expression 12.EQ.I does not contain a real number.

How to recognize the integer in such a fortran expression ?

```
%%
[0-9]+"."EQ {
```

```
            printf ("\n a dot follows an integer \n");
            yyless (yyleng-3);
        }
"."EQ       printf ("\n equality operator");
%%
```

Note: yyless (n) may be called to indicate that not all the characters matched by the currently successful expression are wanted right now.

It also retains n characters in yytext [].

eg.24: To print all the integers which or divisible by either 3 or 5.

```
%%
 int no;
[0-9]+  {
          no = atoi (yytext);
          is_div_by_3 (no);
          is_div_by_5 (no);
        }
%%
is_div_by_3 (n)
int n;
  {
   if  (n%3  ==  0)
        printf  ("\n%d is divisible by 3\n", n);
  }
  is_div_by_5 (n)
  int n;
    {
     if (n%5 == 0)
     printf ("\n %d is divisible by 5\n", n);
    }
```

Some I/O routines:

1. input () returns the next input character.

2. output(c) writes the character "C" on the output.

3. input (c) pushes the character "C" back onto the input stream to be read later by input ().

LEX EXERCISES

1. To recognize a Hexadecimal number and convert it into the equivalent decimal value. The Hexadecimal number starts with #.

2. To recognize a character constant. Note that the escape character \ may be used within quote marks in some cases like

'\n' , '\t', '\\' , '\"

3. To recognize a complex number and separate out the real and imaginary parts.

For example 42+i31 and aa-i2 are valid complex numbers.

4. In a language * is used as the escape character within a string constant. Write Lex source code to recognize such strings and convert them into equivalent C strings.

5. To recognize, validate and change a date in American convention to that in British convention.

******

YACC: YET ANOTHER COMPILER COMPILER

INTRODUCTION:

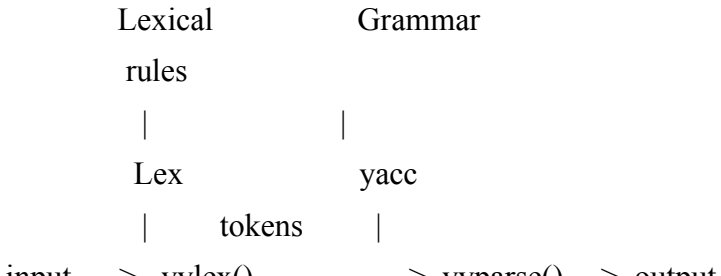
The yacc generates a function called yyparse(), which does LALR parsing ie. It accepts a stream of tokens as inputs, validates the syntactic structure of the input specified by grammar rules and performs actions corresponding to the rules.

The parser yyparse () calls the user supplied (or LEX generated low level input routine yylex () to pick up the basic items (called tokens) from the input stream.

These tokens are organized according to the input structure rules called grammar rules. When a rule is recognized, the user code (supplied for this rule, an action) is invoked.

LEX WITH YACC:

The interaction between LEX and YACC is given pictorially below: -



Note: - Instead of LEX generating yylex (), the user can write his own yylex ().

THE YACC SPECIFICATION:

Declarations
%%
Rules
%%
Programs

The declaration section may be empty, and if the program section is omitted, the second %% mark may also be omitted.

RULES SECTION:

This section is made up of one or more grammar rules.

A grammar rule has the form

A: BODY;

Where "A" represents a nonterminal name and "BODY" represents a sequence of zero more names and literals.

Names may be of arbitrary length and may be made up of letters, dots, underscores and no initial digits. Uppercase and lowercase letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols. A literal

consists of a character is enclosed in single quotes. The backslash (\) is an escape character within literals. Each non-terminal must appear at least once on LHS.

If there are several grammar rules with the same left hand side, the vertical bar (|) can be used to avoid rewriting the left-hand side. In addition the semicolon at the end of a rule can be dropped before a vertical bar.

eg 1:-

```
exp  :  exp '+' exp;
exp  :  CONST;
```

In the above example there are two rules. Exp is a non-terminal, '+' is a literal and CONST is a token returned by lexical analyzer. Since the same exp appears on LHS of both the rules, the above can be written as

```
exp  : exp '+' exp
     | CONST;
```

DECLARATIONS SECTION: -

All the variables can be declared in this section and they should be enclosed within %{and %}

eg 2:-

```
%{
int i,j;
struct & {
        int   a;
        float b;
      } S;
%}
%%
  rules
%%
```

Names representing tokens must be declared. This is done by writing % token name1 name2 name3 . . .

This follows the variable declarations.

eg 3:-

```
%{
   int i,j;
%}
%% token CONST NUM
```

By default the start symbol is taken to be the left hand side of the first grammar rules in the rules section. It is also possible to declare the start symbol explicitly in the declarations section using

%Start keyword.

% Start term

Defines term as the start symbol.

ACTIONS:

A grammar rule may be associated with actions to be performed each time the rule is recognized. On a rule, nonterminals, tokens and literals are treated as objects. The value of the nonterminal on LHS is stored in a dummy variable $$. The object appearing in position i on RHS of the rule is accessed through the dummy variable $i.

eg:-

```
exp : exp '+' exp   { $$ = $1+ $3; }
```

There are four objects $$,$1,$2 and $3. The action $$ = $1 + $3 adds the values of the two exp's on the RHS and associate the result with the LHS exp. Thus an action can return a value and at the same time it can make use of the values returned by the previous actions.

HOW LEX RETURNS TOKENS AND VALUES?

The Lexical Analyzer returns a token to the parser by the action,

```
return();
```

eg 4:-

Lex specification

```
%%
  [0-9]+ return (CONST);
%%
```

When a string of digits is recognized the token CONST is returned to the parser. Note that the name CONST should be declared as token in the delcaration section of the yacc specification.

The Lexical Analyzer assigns a value to a token and returns it to the parser through a built-in external variable called yylval.

eg 5 :- To recognize a binary number and convert it into decimal.

Lex specification

```
%%
[01]    {
          yylval = yytext[0]- '0';
          return(BIT);
        }
\n      return (0);
%%
```

Yacc specification

```
%token BIT
%%
decimal : bin.num {printf("\n decimal value : %d\n",$1);}
        ;
bin.num : bin.num BIT { $$ = $1 * 2 + $2; }
        | BIT       { $$ = $1; }
        ;
%%
```

Note:-

1. By default the type of yylval and all names in the grammar rules is integer. However the type can be changed. How?

2. If no action is specified, the value of the first object in RHS will be assigned to the object in LHS. So in the above example, the last action $$ = $1 can be dropped.

3. To terminate the whole process, the Lexical analyzer must return the value zero to the parser. So in the above example the last lexical rule explicitly returns "0".

However if a file is used as input (instead of giving input through keyboard) no explicit return of NULL is needed. The EOF will automatically return NULL.

AMBIGUITIES

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example consider the following rules.

```
exp  : expr '+'  expr
     | expr '-'  expr
     | expr '*'  expr
     | expr '/'  expr
     |     '-'  expr
     ;
```

Now if the input is

expr + expr - expr

the above grammar allows this input to be structured as either

(expr + expr) - expr

or as

expr + (expr - expr).

(The first is called "left association" and the second "right association")

Similarly the input "expr + expr * expr", can also be structured in two ways. Here apart from association, the precedence also comes into picture.

Unless the precedence and associativity are properly taken care, unambiguous grammar cannot be written.

Fortunately yacc provides certain features to take care of precedence and associativity. The precedence's and associativities are attached to tokens in the declarations sections. This is done by a series of lines beginning with a yacc keyword %left, %right or %nonassoc followed by a list of tokens. Thus:

```
% left '+' '-'
% left '*' '/'
```

describes the precedence and associativity of the four arithmetic operations.

Note:-

1. The declarations describing associativity are listed in order of increasing precedence.

2. Sometimes an unary operator and a binary operator have the same symbolic representation but different precedence's. An example is unary and binary minus. Unary minus is given the same precedence as multiplication while binary minus has lower precedence than multiplication.

The keyword %prec changes the precedence level associated with a particular grammar rule.

For example

```
% left '+' '-'
% left '*' '/'
%%
exp : exp '+'  exp
   | exp '-'  exp
   | exp '*'  exp
   | exp '/'  exp
   | '-' exp % prec '*'
   | NUM;
%%
```

might be used to give unary minus the same precedence as multiplication.

For a clear and better understanding of the above, try out the following example.

eg 6:- A CALCULATOR

Lex specification

```
digit [0-9]
%%
{digit}+    {yylval = atoi(yytext);
        return(NO);
        }
"+"  return ('+');
```

```
"-"  return ('-');
"*"  return ('*');
"/"  return ('/');
"("  return ('(');
")"  return (')');
\n   return(0);
```

Yacc specification

```
%token NO
%left "+" "-"
%left "*" "/"
%%
expression : exp {printf("\n Value : %d\n",$1);}
exp        | exp '+' exp {$$ = $1 + $3;}
exp        | exp '-' exp {$$ = $1 - $3;}
exp        | exp '*' exp {$$ = $1 * $3;}
exp        | exp '/' exp {$$ = $1 / $3;}
exp        | '-' exp %prec '*' {$$ = -$2;}
exp        | '(' exp ')' {$$ = $2;}
exp        | NO
%%
#include "lex.yy.c"
```

SUPPORT FOR ARBITRARY VALUE TYPES

By default, all the objects and yylval are of type integer. To associate arbitrary types to the objects and yylval, a union must be declared by the user. This union is declared in the declaration section and it follows the variable declarations enclosed in %{ and %}

For example

```
% union
{
  int ival;
  float fval;
}
```

can be declared by the user.

Now the external variable yylval has the type equal to this union. Thus we can manipulate yylval.ival and yylval.fval.

Alternatively, the union may be defined in a header file and a typdef is used to define the variable YYSTYPE to represent this union. Thus the header file might have

```
typedef union
  {
    body of union....
  }
  YYSTYPE;
```

instead. The header file must be included in the declarations section by use of %{ and %}

Once YYSTYPE is defined, the union member names must be associated with the various terminal and non-terminal names. The construction is used top indicate a union member name. If this follows one of the

keywords %token, %left and %nonassoc the union member name is

associated with the tokens listed. Thus saying

```
%left  '+' '-'
```

Causes any reference to values returned by these tokens to be

tagged with the union member name optype.

The keyword %type is used to associate union member names with nonterminals.

Thus % type exp stat

associates the union member node type with the nonterminals "exp" and

"stat".

For example consider the following :

```
%union
{
int ival;
float fval;
}
%  INTEGER
```

```
% REAL
% type var
```

The above indicates that the tokens INTEGER and REAL are of types ival and fval (i.e. int & float) respectively and the nonterminal var is a type fval ie.float. For a clear and better understanding, try out the following example.

eg.7 :- To evaluate a REAL or INTEGER expression. i.e. the input can be an expression of only fixed-point numbers or of only integer numbers. In case of mixed type expression an error message must be printed.

Lex specification

```
digit       [0-9]
%%
{digit}+    {yylval.ival = atoi(yytext);
        return(INTEGER);}
{digit}+"."{digit}*     {yylval.fval = atof(yytext);
                return(REAL);}
"+"       return('+');
"-"       return('-');
"*"       return('*');
"/"       return('/');
"("       return('(');
")"       return(')');
\n        return(0);
```

Yacc specification

```
%{
#include
#include
#include
typedef struct s
          {
          int ivalue;
          float fvalue;
```

```
            int type;
            } ETYPE;
%}

%union
{
    int       ival;
    float     fval;
    ETYPE     eval;
}
%token  <ival> INTEGER
%token  <fval> REAL
%type   <eval> exp
%left '+' '-'
%left '*' '/'
%%
stat      : exp
              { if($1.type==0) printf("\nvalue : %d\n",$1.ivalue);
                else           printf("\nvalue : %f\n",$1.fvalue);
              };
exp       : exp '+' exp
              { if($1.type  != $3.type)
                   yyerror("type mismatch error");
              $$.type = $1.type;
              if($1.type == 0)
                   $$.ivalue = $1.ivalue + $3.ivalue;
              else
                   $$.fvalue = $1.fvalue + $3.fvalue;
              }
          | exp '-' exp
              { if($1.type  != $3.type)
                   yyerror("type mismatch error");
              $$.type = $1.type;
              if($1.type == 0)
                   $$.ivalue = $1.ivalue - $3.ivalue;
              else
```

```
        $$.fvalue = $1.fvalue - $3.fvalue;
    }
| exp '*' exp
    { if($1.type != $3.type)
        yyerror(" type mismatch error");
    $$.type = $1.type;
    if($1.type == 0)
        $$.ivalue = $1.ivalue * $3.ivalue;
    else
        $$.fvalue = $1.fvalue * $3.fvalue;
    }
| exp '/' exp
    { if($1.type != $3.type)
        yyerror("type mismatch error");
    $$.type = $1.type;
    if($1.type == 0)
        $$.ivalue = $1.ivalue / $3.ivalue;
    else
        $$.fvalue = $1.fvalue / $3.fvalue;
    }
| '-'exp %prec '*'
    {
    $$.type = $2.type;
    if($2.type == 0)
        $$.ivalue = -$2.ivalue ;
    else
        $$.fvalue = -$2.fvalue ;
    }
| '('exp ')'
    {
    $$.type = $2.type;
    if($2.type == 0)
        $$.ivalue = $2.ivalue;
    else
        $$.fvalue = $2.fvalue;
    }
```

```
        | INTEGER
            {
            $$.type = 0;
            $$.ivalue=$1;
            }
        | REAL
            {
            $$.type = 1;
            $$.fvalue = $1;
            };
%%
#include "lex.yy.c"
```

ERROR HANDLING

The yacc provides a function called yyerror(). yyerror is invoked whenever an error is detected in the input. If the user does not explicitly call yyerror, the message "syntax error" will be printed on the screen when an error is detected.

A user can also call yyerror by passing source string. In that case, the string passed by the user will be printed. The user can also write his own yyerror routine. For example the user may want to print the error message as well as the line number at which the error has occurred.

For example

```
yyerror (s)
char *s;
{
    printf("\n line no %d : %s,l,s);
}
```

will print error message along with the line number. Here the variable l is global and can be declared in the declaration of the yacc specification. The function yyerror () can be written in the programs section of the yacc specification.

It is to be noted that when an error is encountered the behavior of the system may not be in our control. But the user may want the system to behave in a particular way after detecting an error. For example the user may want to collect all the errors in his input program. To facilitate this, yacc provides some feature. The token name "error" is reserved for error handling. The parser pops its stack until it enters a state where the token "error" is legal. It then behaves as if the token "error" is legal. It then behaves as if the token "error" was the current look-ahead token and performs the action encountered.

For example consider the following :

```
%%
state  :  exp   ';'
       |  error ';'
       ;
```

If an error is detected in a statement then all the tokens up to ';' are consumed. The normal processing continues with the next line.

Eg 8:- A language is a set of expressions enclosed in curly braces. The expressions are separated by ';'. Write lex & yacc specifications, to list out all errors along with the line numbers in the input.

Lex specification

```
digit       [0-9]
%%
{digit}+    {yylval = atoi(yytext);
             return(NO);
             }
"+"   return('+');
"-"   return('-');
"*"   return('*');
"/"   return('/');
"("   return('(');
")"   return(')');
"{"   return('{');
"}"   return('}');
";"   return(';');
"$"   return(0);
\n    l++;
```

Yacc specification

```
%{
int l=1;
%}
%token NO
%left "+" "-"
%left "*" "/"
%%
prog      : '{' slist '}'
slist     : slist stat
          | stat;
stat      : exp ';'
          | error ';';
exp       : exp '+' exp { $$ = $1 + $3; }
exp       : exp '-' exp { $$ = $1 - $3; }
exp       : exp '*' exp { $$ = $1 * $3; }
exp       : exp '/' exp { $$ = $1 / $3; }
exp       : '-' exp % prec '*' {$$ = -$2; }
exp       : '(' exp ')' {$$ = $2; }
exp       : NO
%%
#include "lex.yy.c"
yyerror(s)
char *s;
{
    printf("\nline %d : %s\n",l,s);
}
    int main(void )
    {
      yyparse();
      return 0;
    }
```

PREPARATION AND EXECUTION

In yacc specification, #include "lex.yy.c" is typed in programs section to enable parser to call lexical analyzer. The sequence of UNIX commands to prepare specifications and run programs is

```
vi lspec  /* Enter lex  specification */
vi yspec  /* Enter yacc specification */
lex lspec
yacc yspec
cc y.tab.c -ly -ll
a.out (or) a.out   <
```

User can call yyparse() from any 'C' program

For example

```
main ()
{
--------------
  yyparse ();
--------------
}
```

is possible.

EXERCISE

Write yacc and lex specifications to generate three address codes for a sequence of assignment statements. A variable may be declared as integer or as float.

Generate code as a linked list of records, each containing one three-address code.