Lex

Introduction:

Lex & YACC are the tools designed for writers of compilers & interpreters.
Applications of Lex & YACC are:

1. The desktop calculator bc.
2. The tools eqn & pic, typesetting pre-processors for mathematical equations & complex pictures.
3. PCC,the Portable C Compiler & GCC, the GNU C Compiler.
4. A menu compiler
5. A SQL data base language syntax checker.
6. The lex program itself.

Lex & yacc helps us write programs that transform structured input. In programs with structured input, two tasks occur again & again.

1. Dividing the input into meaningful units(tokens).
2. Establishing or discovering the relationship among the tokens.

Terms used:

1. Lexical Analysis or Lexing: It is the process of dividing an input into units or tokens.
2. Lexical Analyzer or Lexer or scanner: It is the process of taking a set of descriptions of possible tokens & producing a C routine.
3. Lex Specification: The set of descriptions you give to lex is called as lex specification.
4. Parsing: It is the process of establishing the relationship among the tokens.
5. Grammar: It is the list of rules that define the relationship among the tokens that the program understands.

Two Rules to remember of Lex:

1. Lex will always match the longest (number of characters) token possible.
   Ex: Input: abc
   Then [a-z]+ matches abc rather than a or ab or bc.
2. If two or more possible tokens are of the same length, then the token with the regular expression that is defined first in the lex specification is favored.
   Ex:
   [a-z]+ {printf("Hello");}
   [hit] {printf("World");}
   Input: hit
   output: Hello

The Structure of LEX:

```
%{
Definition section
%}
%%
Rules section
%%
User Subroutine section
```

- The **Definition** section is the place to define macros and to import header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file. It is bracketed with %{ and %}.
- The **Rules** section is the most important section; Each rule is made up of two parts: a pattern and a action separated by whitespace. The lexer that lex generates will execute the action when it recognizes the pattern. Patterns are simply regular expressions. When the lexer sees some text in the input matching a given pattern, it executes the associated C code. It is bracketed with %% & %%.
- The **User Subroutine section** in which all the required procedures are defined. It contains main in which C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section.
- Example:

```
%{
#include <stdio.h>
%}

%%
[0-9]+ { printf("Saw an integer: %s\n", yytext); }
. { ;}
%%

main( )
{
        printf("enter some input that consists of an integer number\n");
        yylex();
}

int yywrap()
{
        return 1;
}
```

<u>Running Lex program:</u>

```
[student@localhost ~]$ lex 1a.l
[student@localhost ~]$ cc lex.yy.c
[student@localhost ~]$ ./a.out
enter some input that consists of an integer number
hello 2345
Saw an integer :2345
```

Execution Step Explanation:

First line runs lex over the lex specification & generates a file, lex.yy.c which contains C code for the lexer.

Second line, compiles the C file.

Third line, executes the C file.

Recognizing Words with Lex:

Lex program that recognizes different parts of speech.

```
%{
        #include<stdio.h>
%}

%%
[\t  ]+     ;
is |
am |
was |
were |
had        { printf("%s: is a verb\n",yytext); }

very |
simply |
gently |
calmly   { printf("%s: is a adverb\n",yytext); }

to |
behind |
from |
below |
above    { printf("%s: is a preposition\n",yytext); }

if |
then |
and |
or |
but      { printf("%s: is a conjunction\n",yytext); }

their |
my |
your |
his |
her       { printf("%s: is adjective\n",yytext); }

I |
we |
he |
she |
they       { printf("%s: is a pronoun\n",yytext); }

oops |
hurrey |
yahoo |
wow    { printf("%s: is a interjection\n",yytext); }

[a-zA-Z]+   { printf("%s: don't recognize, might be noun\n",yytext); }
. | [\n]        {ECHO;}
%%

main( )
{
        printf("enter some English words\n");
        yylex( );
}

int yywrap()
{
        return 1;
}
```

Output :

```
enter some English words
I am going to college
I :is a pronoun
am : is a verb
going : don't recognize, might be noun
to : is a preposition
college : don't recognize, might be noun
```

Parser-Lexer Communication:

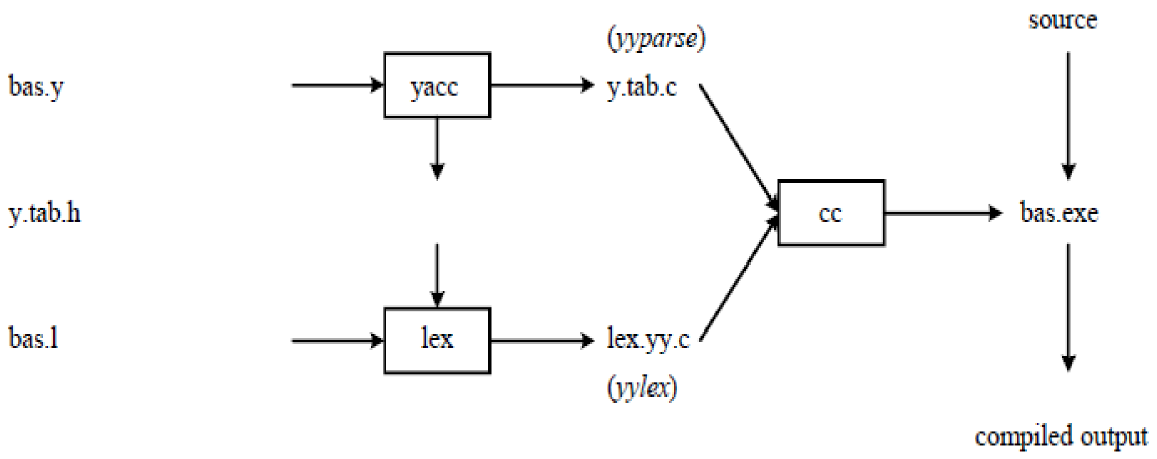


Figure 1-2: Building a Compiler with Lex/Yacc

In the process of compilation the lexical analyzer and parser work together. That means when parser requires string of tokens it invokes lexical analyzer, in turn lexical analyzer supplies tokens to the parser.

When we use LEX and YACC together the YACC becomes high-level routine. It calls the yylex() function of lexer in order to identify and collect the tokens. These generated tokens can then be demanded by the YACC parser for the further arrangement. The parser collects the sufficient number of tokens and builds the parse tree. Not all the tokens that are useful to parser. Some tokens can be ignored for efficient compilation process. Generally, any token can be agreed by LEX and YACC using token code. The token code can be defined using a macro #define. YACC writes the token codes in a separate header file called y.tab.h as follows:
#define NAME 259
The token code 0 represents the logical end of the input.

Regular Expressions:

Definition: A Regular Expression is a pattern description using a "meta" language, a language that you use to describe particular patterns of interest.
Characters that form the regular expressions:

Regular Expression	Description
.	Matches any single character except \n.
*	Match with *zero* or more occurrences of the preceding pattern or expression. Ex: [0-9]*
+	Matches with *one* or more occurrences of the preceding pattern. Ex: [a-z]+
?	Match with *zero* or one occurrences of the preceding pattern or expression. Ex: -?[0-9]* : starts with an - sign
^	1. Matches the beginning of a line as first character. Ex: ^verb : starts with a verb word 2. Used as for negation in Character class. Ex: [^0-9]+ : Except 0-9
[ ]	A character class. Matches *any* character in the brackets. - Used to denote range. Example: [A-Z] implies all characters from A to Z.
$	Matches end of line as the last character of the pattern. Ex: a+b$
{ }	Indicates how many times a pattern can be present. Ex: A{1,3} : implies one or three occurrences of A may be present.
|	Logical OR between expressions. i.e. another alternative. Ex : cow | horse
\	Used to escape meta characters. Also used to remove the special meaning of characters as defined in this table. Ex: \" [a-z]+ \" : String should be included in " ".

“ ”	The string written in quotes matches literally. Ex: “hello”
/	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. Ex: ([0-9]+) | ([0-9]*\.[0-9]+)

LEX Actions:

There are various LEX actions that can be used for ease of programming using LEX tool.

Action	**Description**
BEGIN	It indicates the start state. The lexical analyzer starts at state 0.
ECHO	It emits the input as it is.
Char *yytext	When lexer matches or recognizes the token from input token then the lexeme is stored in a null terminated string called yytext.
FILE *yyin	It is the standard input file.
FILE *yyout	It is the standard output file.
int yyleng	It stores the length or number of characters in the input string.
yylex( )	This is an important function. As soon as call to yylex( ) is encountered, scanner starts scanning the source program.
yywrap( )	It calls when the scanner encounters end of file.
yylval	It gives the value associated with the token.

Regular Expression exampls:

1. Identifier → [a-zA-Z][a-zA-Z0-9]*
2. Decimal Number → [0-9]*\.[0-9]+
3. -ve Integer → [-][0-9]+ or -?[0-9]+
4. +ve fraction → +?[0-9]*\.[0-9]+

Lex vs. Hand-written Lexers:

Lex Version:

Lex program for a simple command language that handles commands, numbers, strings, comments & new lines ignoring white space.

```
%{
        #include<stdio.h>
        #define NUMBER 400
        #define COMMENT 401
        #define STRINGS 402
        #define COMMAND 403
%}
%%
[\t ]+    ;
[0-9]+ | [0-9]*\.[0-9]+    { return NUMBER;}
#.*                        { return COMMENT;}
\"[^ \"\n]*\"                  { return STRINGS;}
[a-zA-Z][a-zA-Z0-9]+           { return COMMAND;}
\n                         {return '\n';}
%%
main( )
{
        int val;
        while( val = yylex( ))
        printf("value is %d\n",val);
}
```

Output :

```
"hello"
value is 402
```

C Version :

```
#include<stdio.h>
#include<ctype.h>
#define NUMBER 400
#define COMMENT 401
#define STRINGS 402
#define COMMAND 403

main( )
{
        int val;
        while( val = lexer( ))
        printf("value is %d\n",val);
}
```

```
lexer( )
{
        int c;
        while (( c=getchar( )) == ' ' || c == '\t')
                ;
        if(c == EOF)
                return 0;
        if(c == '.' || isdigit(c)) /* number */
        {
                while((c=getchar( )) != EOF && isdigit(c));
                if(c == '.')
                        while((c=getchar( )) != EOF && isdigit(c));
                ungetc(c,stdin);
                return NUMBER;
        }
        if( c == '#')  /* comment */
        {
                int index=1;
                while((c=getchar( )) != EOF && c != '\n');
                ungetc(c,stdin);
                return COMMENT;
        }
        if(c == ' ” ' )  /*literal text */
        {
                int index=1;
                while((c=getchar( )) != EOF && c != ' “ ' && c != '\n' );
                if(c == ' \n ')
                        ungetc(c,stdin)
                return STRINGS;
        }
        if(isalpha(c)) /* command */
        {
                int index=1;
                while((c=getchar( )) != EOF && isalnum(c));
                ungetc(c,stdin);
                return COMMAND;
        }
        return c;
}
/*Output : pwd
value is 403
*/
```

Here ungetc - push byte back into input stream.

int ungetc(int *c*, FILE **stream*);

The *ungetc()* function pushes the byte specified by *c* (converted to an **unsigned char**) back onto the input stream pointed to by *stream*.

A word counting program:

Lex program to count the number of spaces, words, lines & characters in a given input file.

```
%{
        #include<stdio.h>
        int sc=0,wc=0,lc=0,cc=0;
%}

%%
[\n]                    { lc++; }
[  \t]                  { sc++; }
[^\t\n  ]+        { wc++;  cc+=yyleng;}
%%

main(int argc ,char* argv[ ])
{
        if(argc==2)
        {
                yyin=fopen(argv[1],"r");
        }
        else
        {
                printf("Enter the input\n");
                yyin=stdin;
        }
        yylex();
        printf("The number of lines=%d\n",lc);
        printf("The number of spaces=%d\n",sc);
        printf("The number of words=%d\n",wc);
        printf("The number of characters are=%d\n",cc);
}


int yywrap( )
{
        return 1;
}
```

Output:

```
Enter the input
hello welcome to exam
good bye
The number of lines=2
The number of spaces=4
The number of words=6
The number of characters are=25
```

LEX Programs

1. Lex Program to find and display keywords, numbers and words in a given statement

```
%{
#include<stdio.h>
#include<string.h>
%}

%%
if {printf("\n%s: is a keyword", yytext);}
else {printf("\n%s: is a keyword", yytext);}
[0-9]+ {printf("\n%s: is a number", yytext);}
[a-zA-Z]+ {printf("\n%s: is a word", yytext);}
.|\n {ECHO;}
%%

int main()
{
        printf("Enter statement\n");
        yylex();
}
int yywrap( )
{
        return 1;
}
```

OUTPUT:

```
maheshgh@maheshgh:~/LexYacc$ ./a.out
Enter statement
Mahesh Huddar if 10 else HIT Nidasoshi

Mahesh: is a word
Huddar: is a word
if: is a keyword
10: is a number
else: is a keyword
HIT: is a word
Nidasoshi: is a word
```

2. Lex Program to find and display keywords, numbers and words in main function in a given statement

```
%{
#include<stdio.h>
#include<string.h>
int i=0,j=0,k=0,l;
char word[20][20],key[20][20], num[20][20];
%}

%%
if {strcpy(key[i],yytext); i++;}
else {strcpy(key[i],yytext); i++;}
[0-9]+ {strcpy(num[j],yytext); j++;}
[a-zA-Z]+ {strcpy(word[k],yytext); k++;}
.|\n {ECHO;}
%%

int main()
{
        printf("Enter statement\n");
        yylex();
        printf("\nKeywords are:");
        for (l=0; l<i;l++)
        {
                printf("%s\t",key[l]);
        }
        printf("\nNumbers are:");
        for (l=0; l<j;l++)
        {
                printf("%s\t",num[l]);
        }
        printf("\nWords are:");
        for (l=0; l<k;l++)
        {
                printf("%s\t",word[l]);
        }
        printf("\n");
}

int yywrap( )
{
        return 1;
}
```

OUTPUT:

```
maheshgh@maheshgh:~/LexYacc$ ./a.out
```

```
Enter statement
Mahesh Huddar if 10 else HIT Nidasoshi

Keywords are:if         else
Numbers are:10
Words are:Mahesh      Huddar          HIT      Nidasoshi
```

3. Lex Program to find and display Numbers, Comments, Identifiers and strings in a given statement

```
%{
#include<stdio.h>
%}

%%
[\t ]+ ;
[0-9]+|[0-9]*\.[0-9]+ { printf("\n%s is NUMBER", yytext);}
#.* { printf("\n%s is COMMENT", yytext);}
[a-zA-Z][a-zA-Z0-9]+ { printf("\n%s is IDENTIFIER", yytext);}
"\.*" { printf("\n%s is STRING", yytext);}
\n { ECHO;}
%%

int main()
{
        while( yylex());
}

int yywrap( )
{
        return 1;
}
```

OUTPUT:

```
maheshgh@maheshgh:~/LexYacc$ ./a.out
10 #Mahesh

10 is NUMBER
#Mahesh  is COMMENT
```

4. Lex Program to count numbers of lines, words, spaces and characters in a given statement

```
%{
#include<stdio.h>
int sc=0,wc=0,lc=0,cc=0;
%}
```

```
%%
[\n] { lc++; cc+=yyleng;}
[  \t] { sc++; cc+=yyleng;}
[^\t\n ]+ { wc++;  cc+=yyleng;}
%%

int main(int argc ,char* argv[ ])
{
        printf("Enter the input\n");
        yylex();
        printf("The number of lines=%d\n",lc);
        printf("The number of spaces=%d\n",sc);
        printf("The number of words=%d\n",wc);
        printf("The number of characters are=%d\n",cc);
}

int yywrap( )
{
        return 1;
}
```

OUTPUT:

```
maheshgh@maheshgh:~/LexYacc$ ./a.out
Enter the input
Mahesh Huddar
Associate Professor
CSE, HIT, Nidasoshi
The number of lines=3
The number of spaces=4
The number of words=7
The number of characters are=54
```

5. Lex Program to count numbers of lines, words, spaces and characters in a given statement or input file

```
%{
#include<stdio.h>
int sc=0,wc=0,lc=0,cc=0;
%}

%%
[\n] { lc++; cc+=yyleng;}
[  \t] { sc++; cc+=yyleng;}
[^\t\n ]+ { wc++;  cc+=yyleng;}
%%
```

```
int main(int argc ,char* argv[])
{
        if(argc==2)
        {
                yyin=fopen(argv[1],"r");
        }
        else
        {
                printf("Enter the input\n");
                yyin=stdin;
        }
        yylex();
        printf("The number of lines=%d\n",lc);
        printf("The number of spaces=%d\n",sc);
        printf("The number of words=%d\n",wc);
        printf("The number of characters are=%d\n",cc);
}

int yywrap( )
{
        return 1;
}
```

OUTPUT:

```
maheshgh@maheshgh:~/LexYacc$ ./a.out input.txt
The number of lines=3
The number of spaces=2
The number of words=5
The number of characters are=32
```

6. Lex program to display the input as it is.

```
%{
        #include<stdio.h>
%}

%%
.       ECHO;
%%

int main( )
{
        printf("Enter some input\n");
        yylex();
}
```

```
int yywrap( )
{
        return 1;
}
```

OUTPUT:

```
maheshgh@maheshgh:~/LexYacc$ ./a.out
Enter some input
Mahesh Huddar
Mahesh Huddar
```

7. Lex program to convert the substring abc to ABC from the given input string.

```
%{
        #include<stdio.h>
        int i ;
%}

%%
[a-zA-Z\n ]* { for(i=0;i<=yyleng;i++)
                {
                if(( yytext[i]=='a') && (yytext[i+1]=='b') && (yytext[i+2]=='c'))
                        {
                                yytext[i]='A';
                                yytext[i+1]='B';
                                yytext[i+2]='C';
                        }
                }
                printf("%s",yytext);
            }
.   ECHO;
%%

main(void)
{
        printf("Enter some string that consists of abc as substring\n");
        yylex();
}
int yywrap( )
{
        return 1;
}
```

OUTPUT:

```
maheshgh@maheshgh:~/LexYacc$ ./a.out
Enter some string that consists of abc as substring
```

```
HIT CSE abc VTU Belagavi
HIT CSE ABC VTU Belagavi
```

8. Lex program to count the number of comment lines in a given C program. Also eliminate them and copy the resulting program into separate file.

```
%{
#include<stdio.h>
int nc=0;
%}

%%
"/*"[a-zA-Z0-9\n\t ]*"*/"  {nc++;}
"//"[a-zA-Z0-9\t ]*"\n"   {nc++;}
%%

int main(int argc ,char* argv[])
{
        if(argc==2)
        {
                yyin=fopen(argv[1],"r");
        }
        else
        {
                printf("Enter the input\n");
                yyin=stdin;
        }
        yyout=fopen("output.c","w");
        yylex( );
        printf("The number of comment lines=%d\n",nc);
        fclose(yyin);
        fclose(yyout);
}

int yywrap( )
{
        return 1;
}
```

OUTPUT:

```
maheshgh@maheshgh:~/LexYacc$ ./a.out input.c
The number of comment lines=4
```

9. Program to recognize a valid arithmetic expression and to recognize the identifiers and operators present. Print them separately.

```
%{
#include<stdio.h>
#include<string.h>
int flag=0,i=0,j,k=0;
char operand[20][20],oparator[20][20];
%}

%%
[a-zA-Z0-9]+  {flag++; strcpy(operand[i],yytext);  i++;}
[-+*/]  {flag--;  strcpy(oparator[k],yytext);    k++;}
%%

int main(int argc, char* argv[])
{
        printf("enter an arithmetic expression\n");
        yylex();

        if(flag!=1)
                printf("invalid expression\n");
        else
        {
                printf("valid expression\n");

                printf("the operands are\t");
                for(j=0;j<i;j++)
                        printf("%s\t",operand[j]);

                printf("\nThe operators are\t");
                for(j=0;j<k;j++)
                        printf("%s\t",oparator[j]);

                printf("\n");
        }
}

int yywrap( )
{
        return 1;
}
```

OUTPUT:

```
maheshgh@maheshgh:~/LexYacc$ ./a.out
enter an arithmetic expression
a+b*c
```

```
valid expression
the operands are        a        b        c
The operators are       +        *

maheshgh@maheshgh:~/LexYacc$ ./a.out
enter an arithmetic expression
a+b*

invalid expression
```

10. Program to recognize a valid arithmetic expression and to recognize the identifiers only numbers and operators only + and * present. Print them separately.

```
%{
#include<stdio.h>
#include<string.h>
int flag=0,i=0,j,k=0;
char operand[20][20],oparator[20][20];
%}

%%
[0-9]+  {flag++; strcpy(operand[i],yytext);  i++;}
[+*]    {flag--;    strcpy(oparator[k],yytext);   k++;}
%%

int main(int argc, char* argv[])
{
        printf("enter an arithmetic expression\n");
        yylex();

        if(flag!=1)
                printf("invalid expression\n");
        else
        {
                printf("valid expression\n");

                printf("the operands are\t");
                for(j=0;j<i;j++)
                        printf("%s\t",operand[j]);

                printf("\nThe operators are\t");
                for(j=0;j<k;j++)
                        printf("%s\t",oparator[j]);

                printf("\n");
        }
}

int yywrap( )
{
        return 1;
}
```

OUTPUT:

```
maheshgh@maheshgh:~/LexYacc$ ./a.out
enter an arithmetic expression
a+b

valid expression
the operands are        a        b
The operators are       +
maheshgh@maheshgh:~/LexYacc$ ./a.out
enter an arithmetic expression
a+b*90

valid expression
the operands are        a        b        90
The operators are       +        *
```

11. Program to recognize whether a given sentence is simple or compound.

```
%{
        #include<stdio.h>
        int flag=0;
%}

%%
and |
or |
but |
because |
if |
then |
nevertheless  { flag=1; }
.  ;
\n  { return 0; }
%%

int main( )
{
        printf("Enter the sentence:\n");
        yylex();
        if(flag==0)
                printf("Simple sentence\n");
        else
                printf("compound sentence\n");
}

int yywrap( )
{
        return 1;
}
```

OUTPUT:

```
maheshgh@maheshgh:~/LexYacc$ ./a.out
Enter the sentence:
HSIT is accrediated by both NBA and NAAC
compound sentence
```

12. Lex program to count the number of vowels & consonants from the given input string.

```
%{
        #include<stdio.h>
        int vow=0, con=0;
%}

%%
[ \t\n]+    ;
[aeiouAEIOU]+    {vow++;}
[^aeiouAEIOU]     {con++;}
%%

int main( )
{
        printf("Enter some input string:\n");
        yylex();
        printf("Number of vowels=%d\n",vow);
        printf("Number of consonants=%d\n",con);
}

int yywrap( )
{
        return 1;
}
```

OUTPUT:

```
maheshgh@maheshgh:~/LexYacc$ ./a.out
```

```
Enter some input string:
HIT Nidasoshi VTU Belagavi
Number of vowels=10
Number of consonants=13
```

13. Lex program to identify the capital words from the given input string.

```
%{
	#include<stdio.h>
%}

%%
[A-Z]+[\t\n ] { printf("%s",yytext); }
.  ;
%%

main( )
{
	printf("Enter some string with capital words in between\n");
	yylex();
}

int yywrap( )
{
	return 1;
}
```

OUTPUT:

```
maheshgh@maheshgh:~/LexYacc$ ./a.out
Enter some string with capital words in between
HSIT Nidasoshi VTU Belagavi
HSIT VTU
```

14. Lex program that recognizes different parts of speech.

```
%{
	#include<stdio.h>
%}

%%
[\t  ]+     ;
is |
am |
was |
were |
had       { printf("%s: is a verb\n",yytext); }

very |
simply |
gently |
calmly   { printf("%s: is a adverb\n",yytext); }

to |
behind |
from |
below |
above    { printf("%s: is a preposition\n",yytext); }

if |
then |
and |
or |
but     { printf("%s: is a conjunction\n",yytext); }

their |
my |
your |
his |
her      { printf("%s: is adjective\n",yytext); }

I |
we |
he |
she |
they      { printf("%s: is a pronoun\n",yytext); }

oops |
hurrey |
yahoo |
wow    { printf("%s: is a interjection\n",yytext); }

[a-zA-Z]+   { printf("%s: don't recognize, might be noun\n",yytext); }
.|\n        {ECHO;}
%%

int main( )
{
        printf("Enter some English words\n");
        yylex( );
}
```

```
int yywrap( )
{
        return 1;
}
```

OUTPUT:

```
maheshgh@maheshgh:~/LexYacc$ ./a.out
Enter some English words
I am going to college
I: is a pronoun
am: is a verb
going: don't recognize, might be noun
to: is a preposition
college: don't recognize, might be noun
```

15. Lex Program to recognize and count the number of keywords in a given input file.

```
%{
#include<stdio.h>
#include<string.h>
int count_key=0;
%}

%%
int |
float |
char |
double |
switch |
if |
while |
do  { count_key++;}
.|\n   {;}
%%

int main ( int argc,char *argv[] )
{
        if(argc==2)
                yyin=fopen(argv[1],"r");
        else
        {
                printf("\nEnter the input:\n");
                yyin=stdin;
        }
        yylex();
        printf("\nNumber of keywords = %d\n",count_key);
}

int yywrap( )
{
        return 1;
}
```

OUTPUT:

```
maheshgh@maheshgh:~/LexYacc$ ./a.out input.c

Number of keywords = 4
```

16. Lex Program to recognize and count the number of identifiers in a given input file.

```
%{
    #include<stdio.h>
    #include<string.h>
    char kw[10][10]={"int","float","char","double","switch","long","while","do","for"};
    int num_of_kw=9;
    int count_id=0;
%}

%%
[a-zA-Z_][a-zA-Z0-9_]* {
                                int flag=0, i;
                                for(i=0;i<num_of_kw;i++)
                                {
                                        if(strcmp(yytext,kw[i])==0)
                                                flag=1;
                                }
                                if( flag==0 )
                                        count_id++;
                        }
.|\n ;
%%

int main (int argc,char *argv[])
{
        if(argc==2)
                yyin=fopen(argv[1],"r");
        else
        {
                printf("\nEnter the input:\n");
                yyin=stdin;
        }
        yylex();
        printf("\nNumber of identifiers = %d\n",count_id);
}

int yywrap( )
{
        return 1;
}
```

OUTPUT:

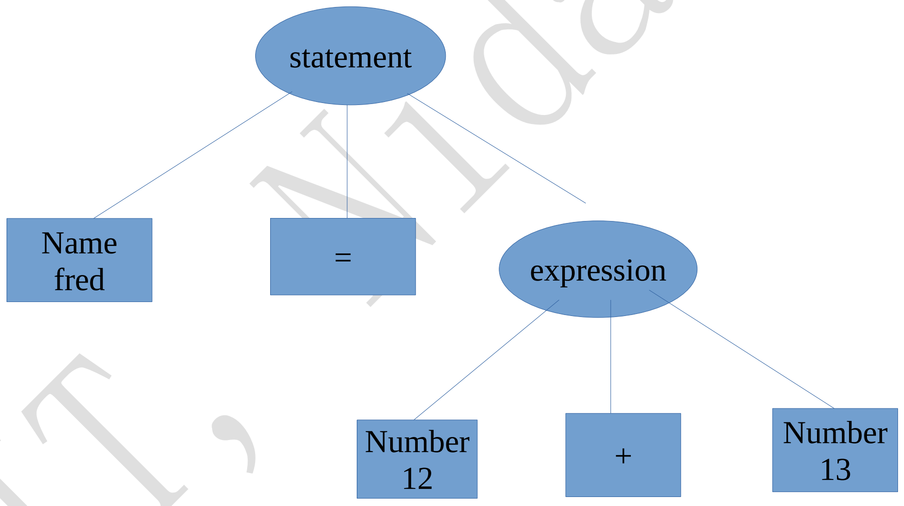
```
maheshgh@maheshgh:~/LexYacc$ ./a.out input.c

void main()
{
        int a;
        float b;
        fun();
}

Number of identifiers = 12


maheshgh@maheshgh:~/LexYacc$ ./a.out

Enter the input:
void main()
{
        int a;
        float b;
        fun();
}

Number of identifiers = 5
```

YACC – Yet Another Compiler-Compiler

Definition: YACC is a tool which makes use of grammar for checking syntax of the given input.

Grammar: It is a series of rules that the parser uses to recognizes syntactically valid input.
Ex: Statement → NAME = expression
expression → NUMBER + NUMBER
| NUMBER – NUMBER
The symbols to the left of the → is known as Left-hand side (LHS) of the rule , & the symbols to the right of the → are called as Right-hand side (RHS) of the rule.

Terminal Symbols or Tokens → Symbols that actually appear in the input & are returned by the lexer are called as terminal symbols or tokens.
Non-Terminal Symbols → Symbols that appear on the left-hand side of some rule are called as non-Terminal symbols.
Terminal & non-Terminal symbols must be different.
The usual way to represent a parsed sentence is as a tree.
Ex: For the above said grammar, the parse tree for the input fred=12+13.



Every grammar include a start symbol, the one that has to be at the root of the parse tree. In the above example statement is the start symbol.

Recursive Rule: Rules can refer directly or indirectly to themselves.
Advantage:
It makes it possible to parse arbitrarily long input sequences.
Ex: A → a A | a
This grammar has a non-Terminal A which again defines A. Using above grammar we can obtain the sequence like a, aa, aaa, aaaaa,.............etc.

Shift / Reduce Parsing :

Shift reduce parser attempts to construct parse tree from leaves to root. This works on the bottom-up approach. A shift reduce parser requires the following data structures:

- Input Buffer → to store input string.
- Stack → For string & accessing the LHS & RHS of rules.

The parser performs following basic operations:

1. **Shift** → As the parser reads tokens, each time it reads a token that doesn't complete a rule it pushes the token on an internal stack. This action is called as **Shift.**

2. **Reduce** → When the parser has found all the symbols that constitute the RHS of a rule, it pops the RHS symbol off the stack & pushes LHS onto the stack. This action is called as **Reduce**.

3. **Accept** → If the stack contains start symbol only & input buffer is empty at the same time then that action is called as **Accept**.

4. **Error** → A situation in which parser cannot either shift or reduce the symbols, it cannot even perform the accept action is called as **error**.

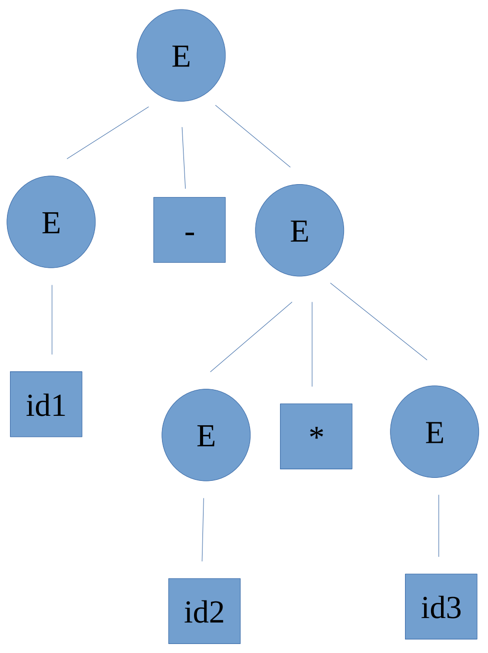
Problems on Shift/Reduce Parsing:

1. Consider the grammar

E → E – E
E → E * E
E → id

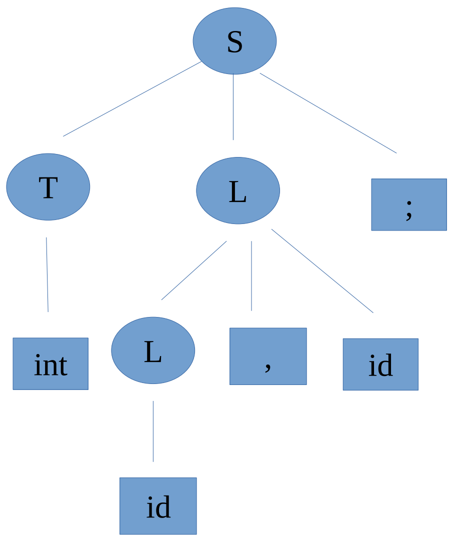
Perform shift-reduce parsing of the input string "id1- id2*id3" & draw the parse tree.

STACK	INPUT BUFFER	ACTION
$	id1- id2*id3$	shift
$id1	- id2*id3$	Reduce by E → id
$E	- id2*id3$	shift
$E-	id2*id3$	shift
$E-id2	*id3$	Reduce by E → id
$E-E	*id3$	shift
$E-E*	id3$	Shift
$E-E*id3	$	Reduce by E → id
$E-E*E	$	Reduce by E → E * E
$E-E	$	Reduce by E → E – E
$E	$	Accept



2. Consider the grammar
S → TL;
T → int | float
L → L,id | id
Parse the input string int id,id; using Shift-reduce parser. & draw its parse tree.

STACK	INPUT BUFFER	ACTION
$	int id,id;$	shift
$int	id,id;$	Reduce by T → int
$T	id,id;$	shift
$Tid	,id;$	Reduce by L → id
$TL	,id;$	Shift
$TL,	id;$	shift
$TL,id	;$	Reduce by L → L,id
$TL	;$	Shift
$TL;	$	Reduce by S → TL;
$S	$	Accept



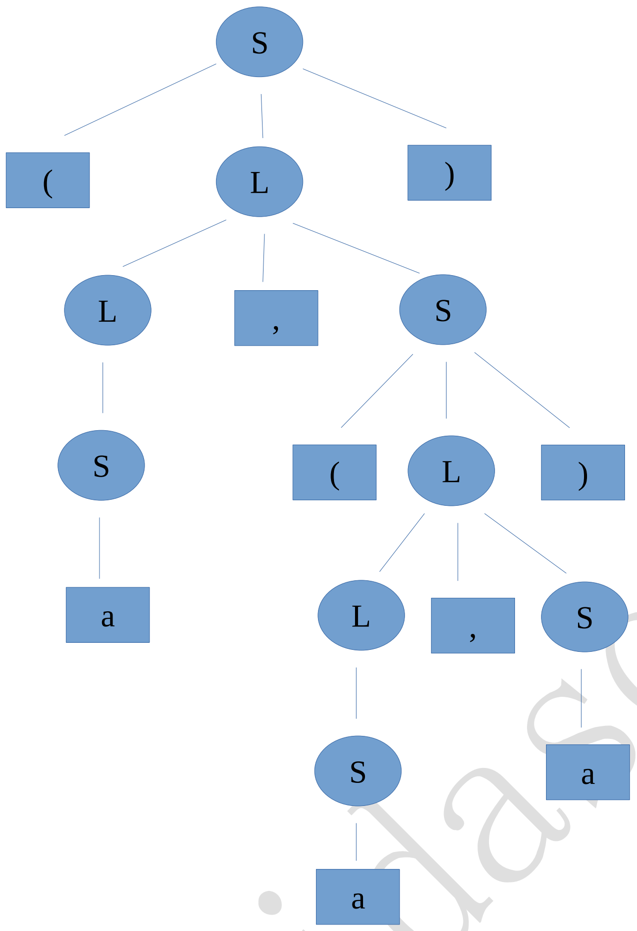
3. Consider the grammar

S → (L) | a

L → L,S | S

Parse the input string (a, (a, a)) using shift-reduce parser & draw its parse tree.

STACK	INPUT BUFFER	ACTION
$	(a, (a, a))$	Shift
$(	a, (a, a))$	Shift
$(a	, (a, a))$	Reduce by S → a
$(S	, (a, a))$	Reduce by L → S
$(L	, (a, a))$	Shift
$(L,	(a, a))$	Shift
$(L,(	a, a))$	Shift
$(L,(a	, a))$	Reduce by S → a
$(L,(S	, a))$	Reduce by L → S
$(L,(L	, a))$	Shift
$(L,(L,	a))$	Shift
$(L,(L,a	))$	Reduce by S → a
$(L,(L,S	))$	Reduce by L → L,S
$(L,(L	))$	Shift
$(L,(L)	)$	Reduce by S → (L)
$(L,S	)$	Reduce by L → L,S
$(L	)$	Shift
$(L)	$	Reduce by S → (L)
$S	$	Accept

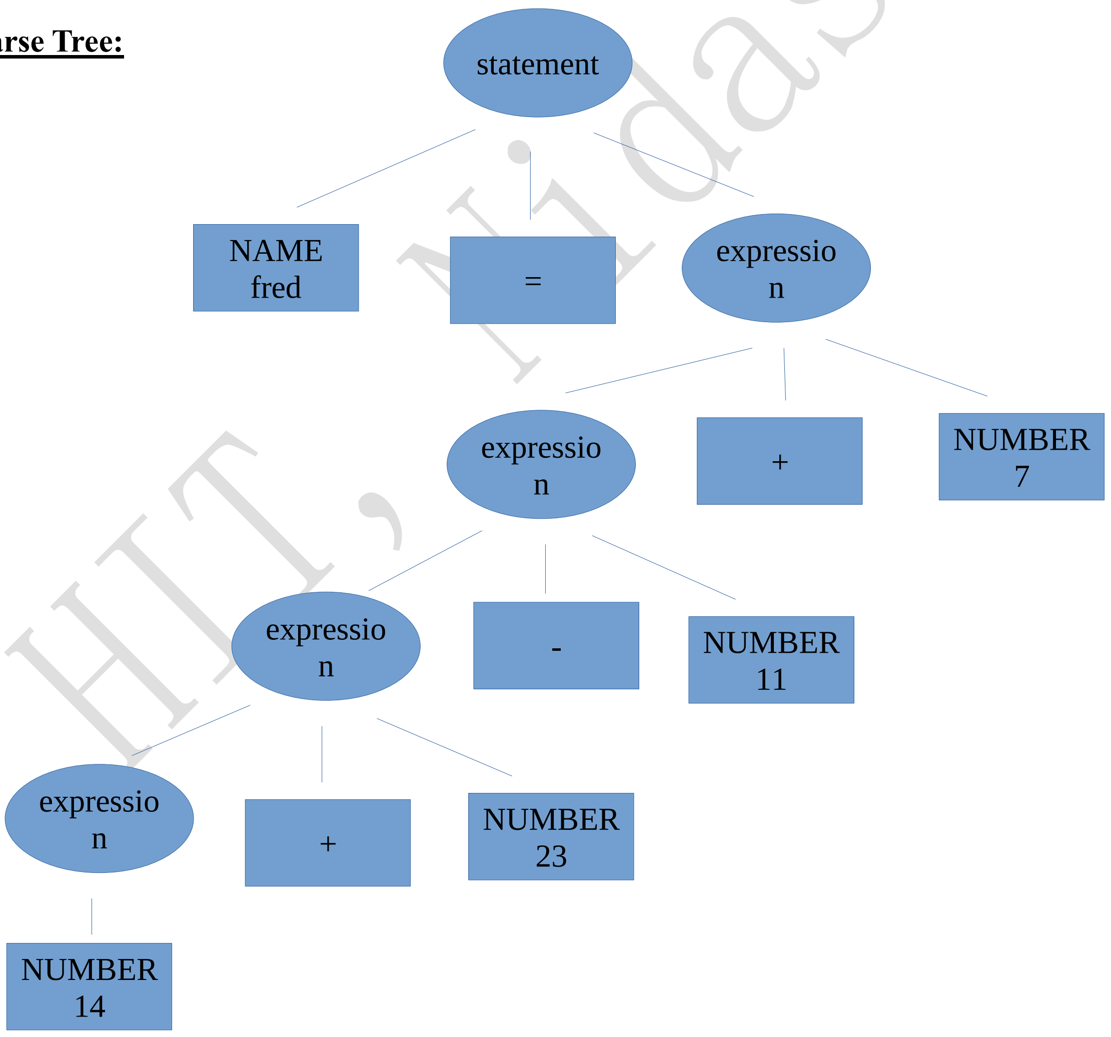


4. Consider the grammar:
statement → NAME = expression
expression → NUMBER
| expression + NUMBER
| expression – NUMBER
NAME → fred | john | scott
Parse the input string fred =14+23-11+7 & draw its parse tree.

STACK	INPUT BUFFER	ACTION
$	fred=14+23-11+7$	Shift
$fred	=14+23-11+7$	Reduce NAME → fred
$NAME	=14+23-11+7$	Shift
$NAME=	14+23-11+7$	Shift
$NAME=14	+23-11+7$	Reduce expression → NUMBER(14)
$NAME=expression	+23-11+7$	Shift
$NAME=expression +	23-11+7$	Shift

$NAME=expression +23	-11+7$	Reduce by expression → expression+ NUMBER(23)
$ NAME=expression	-11+7$	Shift
$NAME=expression -	11+7$	Shift
$NAME=expression -11		Reduce by expression → expression- NUMBER(11)
$ NAME=expression		Shift
$ NAME=expression +		Shift
$ NAME=expression +7	$	Reduce by expression → expression+ NUMBER(7)
$ NAME=expression	$	Reduce by statement → NAME=expression
$statement	$	Accept

Parse Tree:



What YACC cannot parse?

1. It cannot deal with ambiguous grammar, ones in which the same input can match more than one parse tree.
   Ex: E → E – E
   E → E * E
   E → id
   The input string id1- id2*id3 we get two parse trees.

2. It cannot deal with grammars that need more than one token of lookahead to tell whether it has matched a rule.
   Ex: phrase → cart_animal AND CART
   | work_animal AND PLOW
   cart_animal → HORSE | GOAT
   work_animal → HORSE | OX
   If the input is "HORSE AND CART", it cannot tell whether HORSE is a cart_animal or a work_animal
   until it sees CART, and YACC cannot look that far ahead.

Structure of the YACC program:

```
%{
Definition section
%}
%%
Rules section
%%
User Subroutine section
```

- **Definition Section :** It includes declarations of the tokens used in the grammar, the types of values used on the parser stack. It also includes a literal block, C code enclosed in %{ %}. It handles control information for the yacc - generated parser & generally sets up execution environment in which the parser will operate. Token declaration & type declaration can be done as follows:
  %token A B
  %type<dval> expr
  Single quoted characters can also be used as tokens without declaring them. '+', '-' etc.
- **Rules Section :** It simply consists of a list of grammar rules. Grammar consists of following format:
  LHS : RHS
  ;
  - It uses a colon between left & right hand sides of a rule & semicolon at the end of the each rule.

  - The symbol on the LHS of the first rule in the grammar is normally the start symbol.
  - Every symbol in a yacc parser has a value.
  - Whenever the parser reduces a rule, it executes user C code associated with the rule, known as rule's
    action.
  - The action appears in braces after the end of the rule, before the semicolon or vertical bar.
  - The action code can refer to the values of RHS symbols as $1, $2 ….......etc. & can set the value of the
    LHS by setting $$.
- **User – subroutine section :** The **User Subroutine section** in which all the required procedures are defined. It contains **main( )** in which C statements and functions that are copied verbatim to the generated source file. It mainly includes yyparse( ) & yyerror( ) functions. Yyparse( ) is the parser generated by yacc so our main program repeatedly calls until the lexer input file runs out. If the input contains invalid list of tokens or no matching rule then yyerror( ) routine is called.

Compiling & Running A Simple Parser:

1. lex 1a.l → It compiles a lex file & generates a C file called as lex.yy.c.
2. yacc -d 1a.y → It compiles a yacc file & generates two files : a C file known as y.tab.c & a header file y.tab.h.
3. cc lex.yy.c y.tab.c → Compiles both c files together.
4. ./a.out → Executes the C programs.

Arithmetic Expressions & Ambiguity :

Ambiguous Grammar : It is a grammar which defines two or more different rules for the same input. It includes two types of conflicts.

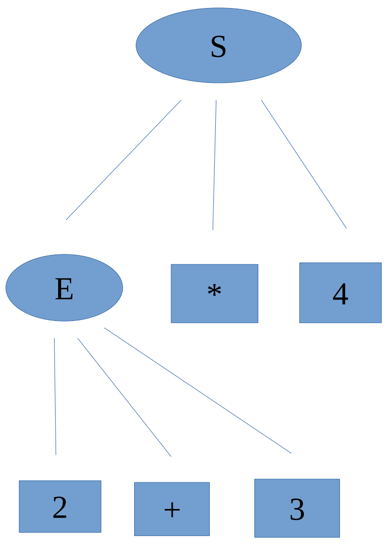
Two Types of Conflicts:

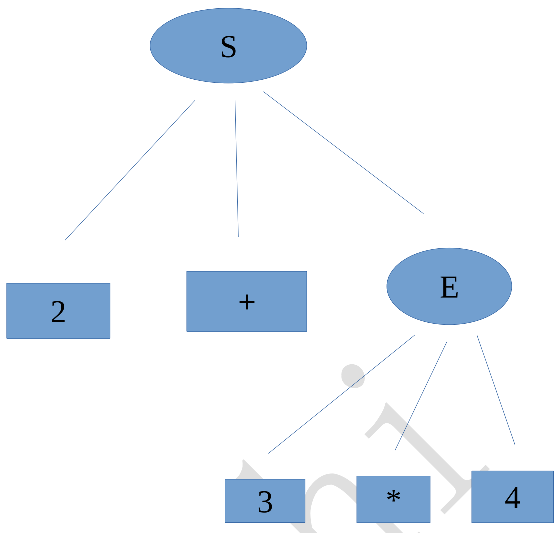
i) Shift/reduce conflict
ii)reduce/reduce conflict
i)shift/reduce conflict: It occurs when there are two possible parses for an input string and one of the parses completes a rule(reduce) and one doesn't(shift).

Consider the following grammar
S→ E
E → E '+' E
  | E '–' E
  | E '*' E
  | E '/' E
  | NUMBER

This is extremely ambiguous. For the input 2+3*4 we get two parse trees.

1. (2+3)*4
2. 2+(3*4).





ii)reduce/reduce conflict: It occurs when the same token could complete two different rules.

Ex:

```
%%
prog : proga | progb
        ;
proga : 'X';
progb : 'X';
```

An "X" could either be a proga or progb.

Resolving Ambiguities:

To resolve this ambiguities, Precedence & Associativity of the operators must be specified.

1. **Precedence :** It controls which operators to execute first in an expression. Operators are grouped into levels of precedence from lowest to highest.
2. **Associativity :** It controls the grouping of operators at the same precedence level. Operators may group to left associativity or right associativity.

<u>Two ways to specify Precedence & Associativity:</u>

- **Implicit way :** Rewrite the grammar using separate non-terminal symbols for each precedence level. Assuming the usual precedence & left associativity for everything we could rewrite the grammar like :

```
E : E '+' mulexp
  | E '-' mulexp
  | mulexp
  ;
mulexp : mulexp '*' primary
       | mulexp '/' primary
       | primary
       ;
primary : '(' E ')'
        | NUMBER
        ;
```

- **Explicit Way :** The following lines can be added to the definition section of YACC.
  %left '+' '-'
  %left '*' '/'
  Each of these declarations defines a level of precedence. These statements tell yacc that '+" & '-' are left associative & at the lowest level, '*' & '/' are left associative and at a higher level. Yacc assigns each rule the precedence of the rightmost token on the RHS. When Yacc encounters a shift/reduce conflict, it consults the table of precedences.

YACC Programs:

1. Program to recognize a valid arithmetic expression that uses operators +, -, * and /.

Lex part :

```
%{
        #include "y.tab.h"
%}

%%
[a-zA-Z]                { return ALPHA; }
[0-9]+                  { return NUM; }
[\t\n]+                  ;
.                       { return yytext[0]; }
%%
```

Yacc part :

```
%{
#include<stdio.h>

#include<stdlib.h>

#include<ctype.h>

int yylex();

int yyerror();
%}

%name parse

%token NUM ALPHA
%left '+' '-'
%left '*' '/ '
%left '(' ')'
```

```
%%
expr:'+'expr
   |'-'expr
   |expr'+'expr
   |expr'-'expr
   |expr'*'expr
   |expr'/'expr
   |'('expr')'
   |NUM
   |ALPHA
   ;
%%


int main( )
{
  printf("enter an arithmetic expression\n");
  yyparse();
  printf("arithmetic expression is valid\n");
  return 0;
}

int yyerror( )
{
  printf("\n arithmetic expression is invalid");
  exit(0);
}

yywrap()
{
        return 1;
}
```

Output:

```
enter an arithmetic expression
a+b
arithmetic expression is valid
```

2. Program to recognize a valid variable, which starts with a letter, followed by any number of letters or digits.

Lex Part:

```
%{
        #include "y.tab.h"
%}
```

```
%%
[a-zA-Z]        return L;
[0-9]           return D;
[\t\n]          return 0;
.               { printf("Invalid variable"); exit(0); }
%%
```

Yacc Part:

```
%{
#include<stdio.h>

#include<stdlib.h>

#include<ctype.h>

int yylex();

int yyerror();
%}

%name parse

%token L D


%%
S : V      { printf("Valid variable\n"); exit(0); }
   ;
V : L | V   L | V   D
  ;
%%


main( )
{
        printf("Enter variable\n");
        yyparse();
}
yyerror( )
{
        printf("Invalid variable\n");
}

int yywrap( )
```

```
{
        return                                                   1;
}
```

Output:

```
Enter variable
empno
Valid variable
```

3. Program to evaluate an arithmetic expression involving operators +, -, * and /.

Lex part :

```
%{
#include<stdio.h>
#include"y.tab.h"
extern int yylval;
%}
%%
[0-9]+ { yylval=atoi(yytext); return NUM;}
[\t\n ] ;
. return yytext[0];
%%
```

Yacc Part:

```
%{
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
int yylex();
int yyerror();
%}
%name parse
%token NUM
%left '+' '-'
%left '*' '/'
```

```
%%
expr: e { printf("Result:%d\n",$$); return 0; }
;
e:e'+'e {$$=$1+$3;}
| e'-'e {$$=$1-$3;}
| e'*'e {$$=$1*$3;}
| e'/'e {$$=$1/$3;}
| '('e')' {$$=$2;}
| NUM {$$=$1;}
;
%%

int main()
{
        printf("\n Enter the Arithmetic Expression:\n");
        yyparse();
        printf("\nValid Expression\n");
}

int yyerror()
{
        printf("\n Invalid Expression\n");
        exit(0);
}

int yywrap( )
{
        return 0;
}
```

Output:

```
Enter the arithmetic expression
10+6
Valid expression and the value is 16
```

4. Program to recognize strings 'aaab', 'abbb', 'ab' and 'a' using the grammar ($a^n b^n$, n>= 0).

Lex part :

```
%{
        #include "y.tab.h"
%}

%%
a               return A;
b               return B;
[ ]+            return empty;
\n              return *yytext;
.               { printf( " Invalid String \n" ); exit(0);  }
%%
```

Yacc Part :

```
%{
        #include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
int yylex();
int yyerror();
%}
%name parse
%token   A   B  empty

%%
S1 : S  '\n'        {  printf( "Valid String \n" ); exit(0);  }
   ;
S : A  B
  | A  S  B
  | empty
  ;
%%

main( )
{
        printf("Enter string:\n");
        yyparse();
}
```

```
yyerror( )
{
        printf("Invalid String \n ");
}

int yywrap( )
{
        return                                                  1;
}
```

Output:

```
enter the string
aaabbb
valid string
```

5. Program to recognize the grammar ($a^n b$, n>=10).

Lex part :

```
%{
        #include"y.tab.h"
%}


%%
a {return A;}
b {return B;}
\n {return yytext[0];}
. {return 0;}
%%
```

Yacc Part :

```
%{
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
int yylex();
int yyerror();
%}
```

```
%name parse
%token A B
%%
str:S'\n' {printf("Valid string\n"); exit(0);}
;
S:A S
|B
;
%%

int main()
{
        printf("Enter the string:\n");
        yyparse();
}

int yyerror()
{
        printf("Invalid string\n");
        exit(0);
}

int yywrap()
{
        return 0;
}
```

Output:

```
enter string
aaaaaaaaaab
Valid string
```

6. Write YACC program to recognize valid identifier, operators and keywords in the given text (C program) file.

LEX Part:

```
%{
#include <stdio.h>
#include "y.tab.h"
extern int yylval;
%}
%%
[ \t] ;
[+|-|*|/|=|<|>] {printf("%s is an operator\n",yytext);return OP;}
[0-9]+ {return DIGIT;}
int |
char |
bool |
float |
void |
for |
do |
while |
if |
else |
return |
main {printf("%s is a keyword\n",yytext);return KEY;}
[a-zA-Z][a-zA-Z0-9]* {printf("%s is an identifier\n",yytext);return ID;}
. ;
%%
```

YACC Part:

```
%{
#include <stdio.h>
#include <stdlib.h>
```

```
int yylex();
int yyerror();
extern int yylex();
extern FILE *yyin;
int id=0, dig=0, key=0, op=0;
%}

%name parse
%token DIGIT ID KEY OP

%%
input: input DIGIT { dig++; }
| input ID { id++; }
| input KEY { key++; }
| input OP {op++;}
| DIGIT { dig++; }
| ID { id++; }
| KEY { key++; }
| OP { op++;}
;
%%

int main(int argc, char* argv[])
{
        if(argc==2)
        {
                yyin=fopen(argv[1],"r");
        }
        else
        {
                printf("Input file Missing\n");
                exit(0);
        }
        yyparse();

        printf("\nNumbers = %d", dig);
        printf("\nKeywords = %d", key);
        printf("\nIdentifiers = %d", id);
        printf("\noperators = %d\n", op);
}
int yyerror()
{
        printf("Parse Error! ");
        exit(0);
}

int yywrap()
{
        return 1;
}
```