`

EXPERIMENT NO: 10

1. Title:
   Write a program using YACC specifications to implement syntax analysis phase of compiler to recognize simple and compound sentences given in input file.

2. Objectives:
   - To understand LEX & YACC Concepts
   - To implement LEX Program & YACC program
   - To study about Lex & Yaac specification
   - To know important about Lexical analyzer and Syntax analysis

3. Problem Statement:
   Write a program using YACC specifications to implement syntax analysis phase of compiler to recognize simple and compound sentences given in input file.

4. Outcomes:

After completion of this assignment students will be able to:
   - Understand the concept of LEX & YACC Tool
   - Understand the lexical analysis & Syntax analysis part
   - Understand the Simple and Compound sentense.

5. Software Requirements:
   FLEX, YACC (LEX & YACC)

6. Hardware Requirement:

   - M/C Lenovo Think center M700 Ci3,6100,6th Gen. H81, 4GB RAM ,500GB HDD

7. Theory Concepts:

Yacc(Yet Another Compiler-Compiler) is a computer program for the Unix operating system developed by Stephen C. Johnson. It is a Look Ahead Left-to-Right (LALR) parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code, specifically a LALR parser, based on an analytic grammar written in a notation similar to Backus–Naur Form (BNF). Yacc is supplied as a standard utility on BSD and AT&T Unix. GNU-based Linux distributions include Bison, a forward-compatible Yacc replacement.

Yacc is one of the automatic tools for generating the parser program. Basically Yacc is a LALR parser generator. The Yacc can report conflicts or ambiguities (if at all) in the form of error messages. LEX and Yacc work together to analyse the program syntactically.

Yacc is officially known as a “parser”. Its job is to analyze the structure of the input stream, and operate of the “big picture”. In the course of it’s normal work, the parser also verifies that the input is syntactically sound.

YACC stands for “Yet Another Compiler Compiler” which is a utility available from Unix.

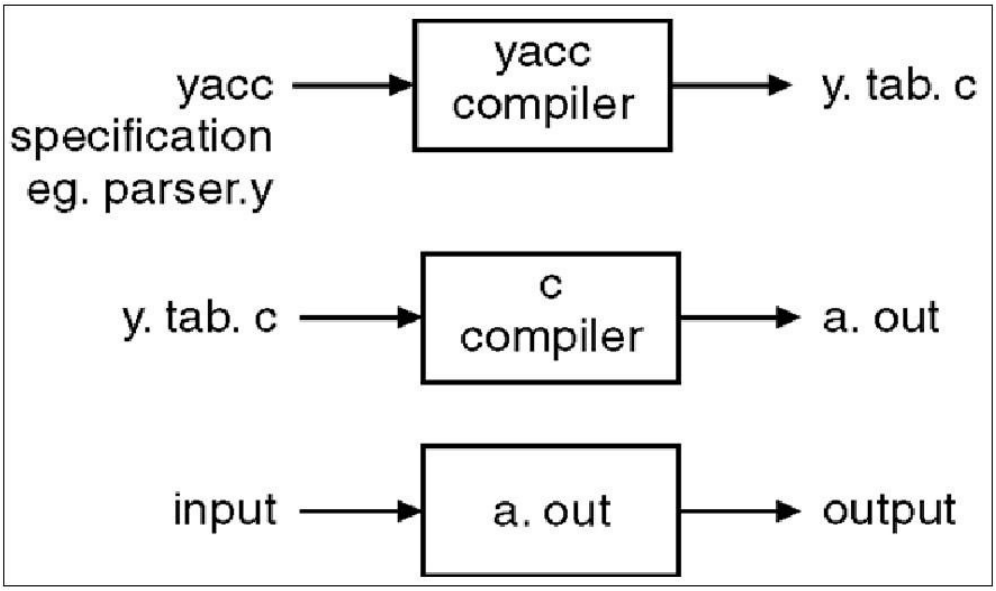


Fig:-YACC: Parser Generator Model

Structure of a yacc file:

A yacc file looks much like a lex file:

...definitions..

%%

...rules...

%%

...code...

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) 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 one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

date : month_name day ',' year ;

Here, date, month_name, day, and year represent structures of interest in the input process; presumably, month_name, day, and year are defined elsewhere. The comma ``,'' is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon

Merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input July 4, 1776 might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a terminal symbol, while the structure recognized by the parser is called a nonterminal symbol. To avoid confusion, terminal symbols will usually be referred to as tokens.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name : 'J' 'a' 'n'  ;
month_name  :  'F' 'e' 'b'   ;

        …….

month_name  :  'D' 'e' 'c'   ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and month_name would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a month name was seen; in this case, month name would be a token.

Literal characters such as ``,'' must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is relatively easy to add to the above example the rule

```
date : month '/' day '/' year  ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be ``slipped in'' to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self-contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases represent design errors; the latter cases can

often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules.

Basic Specifications:

Names refer to either tokens or non-terminal symbols. Yacc requires token names to be declared as such. In addition, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well.

Thus, every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent ``%%" marks. (The percent ``%" is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also;

thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A: BODY;
```

A represents a non-terminal name, and BODY represents a sequence of zero or more names and Literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ``.", underscore ``_", and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or non-terminal symbols.

A literal consists of a character enclosed in single quotes ``'". As in C, the backslash ``\" is an escape character within literals, and all the C escapes are recognized. Thus

```
'\n'   newline
'\r'   return
'\''   single quote ``'"
```

'\\' backslash ``\''
'\t' tab
'\b' backspace
'\f' form feed
'\xxx' ``xxx'' in octal

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

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. Thus the grammar rules

```
A  :  B C D ;
A  :  E F ;
A  :  G ;
```

can be given to Yacc as

```
A  :  B C  D
   |  E  F
   |  G
   ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a non-terminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty :  ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token name1 name2 . . .
```

in the declarations section. Every name not defined in the declarations section is assumed to represent a non-terminal symbol. Every non-terminal symbol must appear on the left side of at least one rule.

Of all the non-terminal symbols, one, called the start symbol, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start  symbol
```

The end of the input to the parser is signaled by a special token, called the end-marker. If the tokens up to, but not including, the end-marker form a structure which matches the start symbol, the parser function returns to its caller after the end-marker is seen; it accepts the input. If the end-marker is seen in any other context, it is an error.

Actions

`

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces ``{" and ``}". For example,

```
A     :     '(' B ')'
                {     hello( 1, "abc" );  }
```

and

```
XXX   :     YYY ZZZ
                {     printf("a message\n");
                      flag = 25;  }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol ``dollar sign" ``$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable ``$$" to some value. For example, an action that does nothing but return the value 1 is

```
{ $$ = 1;  }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A     :     B C D  ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr  :     '(' expr ')'  ;
```

The value returned by this rule is usually the value of the expr in parentheses. This can be indicated by

```
expr  :     '('  expr  ')'      { $$ = $2 ; }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A     :     B   ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by

the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A     :     B
                  { $$ = 1;  }
            C
                  { x = $2; y = $3;  }
      ;
```

the effect is to set x to 1, and y to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new non-terminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
$ACT   :     /* empty */
                  { $$ = 1;  }
      ;

A     :     B $ACT C
                  { x = $2; y = $3;  }
      ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function node, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

```
expr   :     expr '+'  expr
                  { $$ = node( '+', $1, $3 );  }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks ``%{'' and ``%}''. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{ int variable = 0;  %}
```

could be placed in the declarations section, making variable accessible to all of the actions. The Yacc parser uses only names beginning in ``yy''; the user should avoid such names.

Translating, Compiling and Executing A Yacc Program

The Lex program file consists of Lex specification and should be named <file name>.l and the Yacc program consists of Yacc sepecification and should be named <file name>.y. following command may be issued to generate the parser

Lex <file name>.l

Yacc –d <file name>.y

cc lex.yy.c y.tab.c –ll

./a.out

Yacc reads the grammar description in <file name>.yand generates a parser, function yyparse, in file y.tab.c . the –d option causes yacc to generate the definitions for tokens that are declared in the <file name>.y and palce them in file y.tab.h. Lex reads the pattern descriptions in <file name>.l, includes file y.tab.h, and generates a lexical analyzer, function yylex, in the file lex.yy.c

Finally, the lexer and the parser are compiled and linked (-ll) together to form the output file, a.out(by default).

The execution of the parser begins from the main function, which will be ultimately call yyparse() to run the parser. Function yyparse() automatically calls yylex() whenever it is in need of token .

Lexical Analyzer for YACC

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called yylex. The function returns an integer, the token number, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable yylval.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the ``# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
        . . .
        c = getchar();
        . . .
        switch( c ) {
                . . .
```

```
case '0':
case '1':
  . . .
case '9':
        yylval = c-'0';
        return( DIGIT );
        . . .
        }
. . .
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names if or while will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name error is reserved for error handling, and should not be used naively.

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

When Yacc generates, the parser(by default y.tab.c, which is C file), it will assign token numbers for all the tokens defined in Yacc program.Token numbers will be assigned using"#define"and will be copied, by default, to y.tab.h file. The lexical analyzer will reasd from this file or any furthe use.

Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient. This is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
  expr : expr OP  expr
```

and

```
  expr : UNARY  expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar with many parsing conflicts. You specify as disambiguating rules the precedence or binding strength of all the operators and the associativity of the binary operators. This information is sufficient to allow yacc to resolve the parsing conflicts in accordance with these rules and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with the yacc keywords %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus

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

describes the precedence and associativity of the four arithmetic operators. + and - are left associative and have lower precedence than * and /, which are also left associative. The keyword %right is used to describe right associative operators. The keyword %nonassoc is used to describe operators, like the operator .LT. in FORTRAN, that may not associate with themselves. That is, because

```
A .LT. B .LT. C
```

is invalid in FORTRAN, .LT. would be described with the keyword %nonassoc in yacc

As an example of the behavior of these declarations, the description

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

%%

expr : expr '=' expr
     | expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | NAME
     ;
```

might be used to structure the input

```
a = b = c * d - e - f * g
```

as follows

```
a = ( b = ( ((c * d) - e) - (f * g) ) )
```

in order to achieve the correct precedence of operators. When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator

have the same symbolic representation but different precedences. An example is unary and binary minus.

Unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword %prec changes the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, the rules

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

%%


expr : expr '+' expr

     | expr '-' expr

     | expr '*' expr

     | expr '/' expr

     | '-' expr %prec '*'

     | NAME

     ;
```

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

A token declared by %left, %right, and %nonassoc need not, but may, be declared by %token as well.

Precedences and associativities are used by yacc to resolve parsing conflicts. They give rise to the following disambiguating rules:

1. Precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule. It is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.

3. When there is a reduce-reduce or shift-reduce conflict, and either the input symbol or the grammar rule has no precedence and associativity, then the two default disambiguating rules given in the preceding section are used, and the conflicts are reported.

4. If there is a shift-reduce conflict and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action -- shift or reduce -- associated with the higher precedence. If precedences are equal, then associativity is used. Left associative implies reduce, right associative implies shift, nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift-reduce and reduce-reduce conflicts reported by yacc This means that mistakes in the specification of precedences may disguise errors in the input grammar.

The yyerror() Function

The yyerror function is called when Yacc encounters an invalid synatx. Whenver an invalid syntax finds error, it will move to already predefined error state. Moving to error state maens shifting (shift/reduce) to error, which is areserved token name for error handling.that is, any move to error state will cause to call function yyerror.the yyerror() is passed a single string of type char* as argument. The basic yyerror() function is like this:

```
yyerror(char* err)

{

  fprintf(stderr,"%s\n",err);

}
```

The above function just prints the error message when we call the function by passsing the argument.

A compound sentence is a sentence that has at least two independent clauses joined by a comma, semicolon or conjunction. An independent clause is a clause that has a subject and verb and forms a complete thought.

An example of a compound sentence is, 'This house is too expensive, and that house is too small.' This sentence is a compound sentence because it has two independent clauses, 'This house is too expensive' and 'that house is too small' separated by a comma and the conjunction 'and.'

Compound Sentences and Meaning

When independent clauses are joined with coordinators (also called coordinating conjunctions) commas and semicolons, they do more than just join the clauses. They add meaning and flow to your writing. First let's look at the coordinators you can use to join independent clauses. They are:

- For
- And
- Nor
- But
- Or
- Yet
- So

Note that they form the handy mnemonic FANBOYS. The three you will use most often are 'and,' 'but' and 'or.'

Here's an example of how coordinating conjunctions add meaning:

'I think you'd enjoy the party, but I don't mind if you stay home.'

In this sentence, the coordinator 'but' shows a clear relationship between the two independent clauses, in this case, that the speaker is making a suggestion that the person being addressed isn't expected to follow it. Without the coordinator 'but,' the relationship isn't apparent, making the writing choppy and the meaning less clear:

'I think you'd enjoy the party. I don't mind if you stay home.'

You can also join independent clauses with a semicolon (;), which looks something like a cross between a colon and a comma. If you join clauses with a semicolon, you add an abrupt pause, creating a different kind of effect, as shown in the sentence below:

'He said he didn't mind if I stayed home; it soon became clear he wasn't being honest.'

You should use a semicolon when the independent clauses are related, but contrast in a way that you want to stand out. In the sentence above, the contrast is that the person being talked about in the first clause sounded honest when he said he didn't mind if the speaker stayed home, but in the second clause, the speaker is telling you that the person being talked about was not honest. You could just as easily have written the sentence using a coordinating conjunction:

'He said he didn't mind if I stayed home, but it soon became clear he wasn't being honest.'

The sentence still means the same as before, but using the coordinator 'but' softens the impact of the second clause.

Comparing Sentence Types

Sentences give structure to language, and in English, they come in four types: simple, compound, complex and compound-complex. When you use several types together, your writing is more interesting. Combining sentences effectively takes practice, but you'll be happy with the result.

1. The simple sentence is an independent clause with one subject and one verb.

   For example: we are the indian.

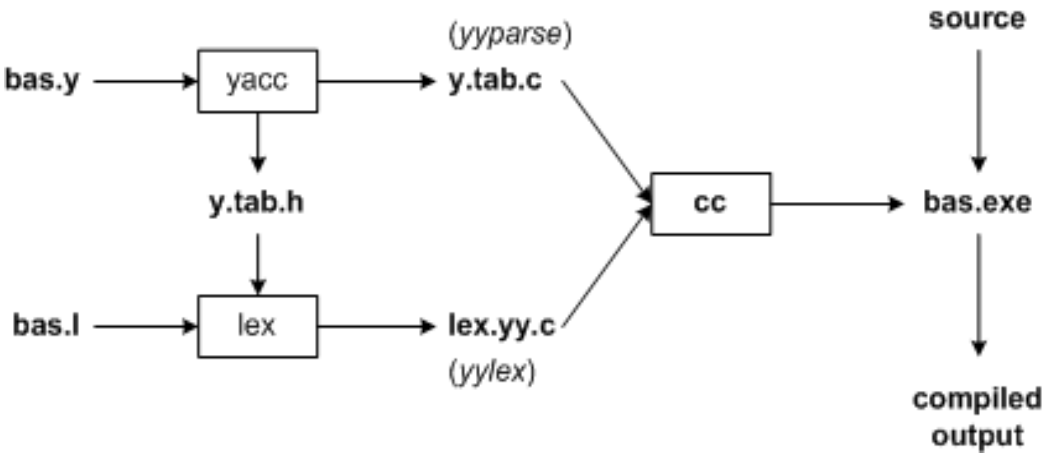
2. The Compound sentence is two or more independent clause, joined with comma, semicolon & conjuction.

   For example: I am student and indian

Application:

· YACC is used to generate parsers, which are an integral part of compiler.

8. Design (architecture) :



9. Algorithms(procedure) :

Note: you should write algorithm & procedure as per program/concepts

10. Flowchart :

Note: you should draw flowchart as per algorithm/procedure

11. Conclusion:

Thus, I have studied lexical analyzer, syntax analysis and implemented Lex & Yacc application for Syntax analyzer to validate the given infix expression.

References :

https://en.wikipedia.org/wiki/Lex_(software)
http://epaperpress.com/lexandyacc/prl.html
https://www.ibm.com/developerworks/library/l-lexyac/index.html
http://epaperpress.com/lexandyacc/pry2.html

Oral Questions: [Write short answer]

1. What is Lex & Yacc .
2. What is Compiler and phases of compiler.
3. What is Lex & Yacc specification.
4. What is the difference between Lex and YACC.
5. What is Regular Expression & grammer.
6. How to run a Lex & Yacc program.
7. What is yytext, yyin, yyout.
8. What is yywrap().
9. What is yylex().
10. What is yyparse().
11. Define token, lexemes, pattern & symbol error?
12. What is left, right & no associativity.
13. What is use of $$?
14. What is yylval.