EXPERIMENT NO : 09

1. Title:
   Write a program using YACC specifications to implement syntax analysis phase of compiler to validate type and syntax of variable declaration in Java.

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 validate infix expression & arithmetic expression in Java.

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
   - It can be used for data mining and checking(validation) concepts.

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.

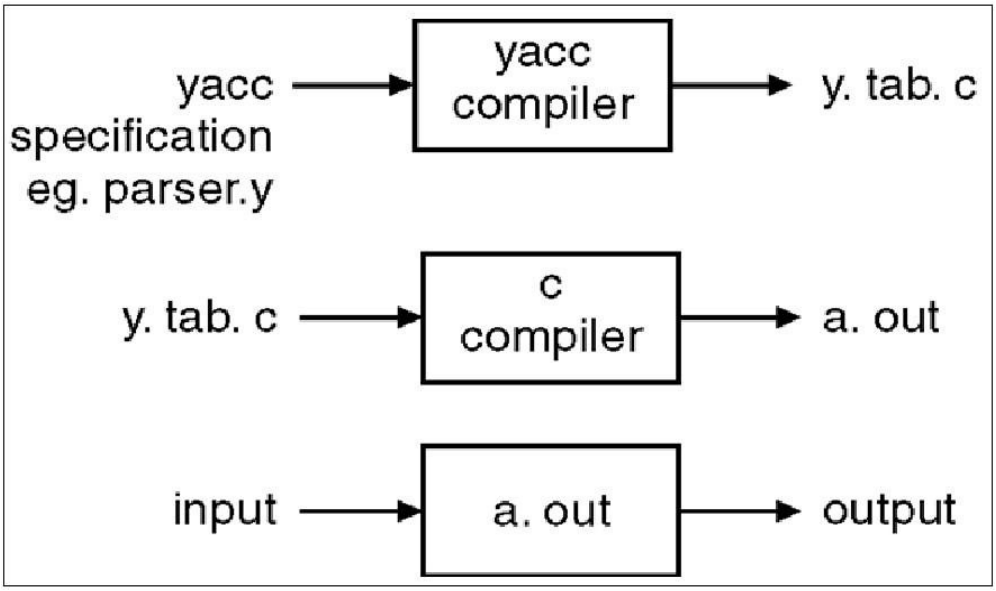


Fig:-YACC: Parser Generator Model

Structure of a yacc file:

A yacc file looks much like a lex file:

...definitions..

%%

...rules...

%%

...code...

Definitions As with lex, all code between %{ and %} is copied to the beginning of the resulting C file. Rules As with lex, a number of combinations of pattern and action. The patterns are now those of a context-free grammar, rather than of a regular grammar as was the 3 case with lex code. This can be very Elaborate, but the main ingredient is the call to yyparse, the grammatical parse.

Input to yacc is divided into three sections. The definitions section consists of token declarations and C code bracketed by "%{" and "%}". The BNF grammar is placed in the rules section and user subroutines are added in the subroutines section.
This is best illustrated by constructing a small calculator that can add and subtract numbers. We'll begin by examining the linkage between lex and yacc. Here is the definitions section for the yacc input file:

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

```
1  E -> E + E
2  E -> E * E
3  E -> id
```

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

```
E -> E * E       (r2)
  -> E * z       (r3)
  -> E + E * z   (r1)
  -> E + y * z   (r3)
  -> x + y * z   (r3)
```

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

```
 1  . x + y * z shift
 2  x . + y * z reduce(r3)
 3  E . + y * z shift
 4  E + . y * z shift
 5  E + y . * z reduce(r3)
 6  E + E . * z  shift
 7  E + E * . z  shift
 8  E + E * z .  reduce(r3)
 9  E + E * E .  reduce(r2) emit multiply
10  E + E .      reduce(r1) emit add
11  E .          accept
```

Terms to the left of the dot are on the stack while remaining input is to the right of the dot. We start by shifting tokens onto the stack. When the top of the stack matches the rhs of a production we replace the matched tokens on the stack with the lhs of the production. In other words the matched tokens of the rhs are popped off the stack, and the lhs of the production is pushed on the stack. The matched tokens are known as a handle and we are reducing the handle to the lhsof the production. This process continues until we have shifted all input to the stack and only the starting nonterminal remains on the stack. In step 1 we shift the x to the stack. Step 2 applies rule r3 to the stack to

change x to E. We continue shifting and reducing until a single nonterminal, the start symbol, remains in the stack. In step 9, when we reduce rule r2, we emit the multiply instruction. Similarly the add instruction is emitted in step 10. Consequently multiply has a higher precedence than addition.

Consider the shift at step 6. Instead of shifting we could have reduced and apply rule r1. This would result in addition having a higher precedence than multiplication. This is known as a shift-reduce conflict. Our grammar is ambiguous because there is more than one possible derivation that will yield the expression. In this case operator precedence is affected. As another example, associativity in the rule

```
E -> E + E
```

is ambiguous, for we may recurse on the left or the right. To remedy the situation, we could rewrite the grammar or supply yacc with directives that indicate which operator has precedence. The latter method is simpler and will be demonstrated in the practice section.

The following grammar has a reduce-reduce conflict. With an id on the stack we may reduce to T or E.

```
E -> T
E -> id
T -> id
```

Yacc takes a default action when there is a conflict. For shift-reduce conflicts yacc will shift. For reduce-reduce conflicts it will use the first rule in the listing. It also issues a warning message whenever a conflict exists. The warnings may be suppressed by making the grammar unambiguous. Several methods for removing ambiguity will be presented in subsequent sections.

```
... definitions
%%
... rules...
%%
... subroutines
```

Input to yacc is divided into three sections. The definitions section consists of token declarations and C code bracketed by "%{" and "%}". The BNF grammar is placed in the rules section and user subroutines are added in the subroutines section.

This is best illustrated by constructing a small calculator that can add and subtract numbers. We'll begin by examining the linkage between lex and yacc. Here is the definitions section for the yacc input file:

```
%token INTEGER
```

This definition declares an INTEGER token. Yacc generates a parser in file y.tab.c and an include file y.tab.h

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define INTEGER 258
extern YYSTYPE yylval;
```

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

```
[0-9]+  {
          yylval = atoi(yytext);
          return INTEGER;
        }
```

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

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

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

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

%%

[0-9]+  {
          yylval = atoi(yytext);
          return INTEGER;
        }

[-+\n]   return *yytext;

[ \t]    ; /* skip whitespace */

.        yyerror("invalid character");
```

```
%%

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

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

```
%{
  #include <stdio.h>
  int yylex(void);
  void yyerror(char *);
%}


%token INTEGER

%%

program:
    program expr '\n'    { printf("%d\n", $2); }
    |
    ;

expr:
    INTEGER             { $$ = $1; }
    | expr '+' expr     { $$ = $1 + $3; }
    | expr '-' expr     { $$ = $1 - $3; }
    ;

%%

void yyerror(char *s) {
  fprintf(stderr, "%s\n", s);
}

int main(void) {
  yyparse();
  return 0;
}
```

The rules section resembles the BNF grammar discussed earlier. The left-hand side of a production, or nonterminal, is entered left-justified and followed by a colon. This is followed by the right-hand side of the production. Actions associated with a rule are entered in braces.

With left-recursion we have specified that a program consists of zero or more expressions. Each expression terminates with a newline. When a newline is detected we print the value of the expression. When we apply the rule

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

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

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

```
expr: INTEGER            { $$ = $1; }
```

The INTEGER token is popped off the parse stack followed by a push of expr. For the value stack we pop the integer value off the stack and then push it back on again. In other words we do nothing. In fact this is the default action and need not be specified. Finally, when a newline is encountered, the value associated with expr is printed.

In the event of syntax errors yacc calls the user-supplied function yyerror. If you need to modify the interface to yyerror then alter the canned file that yacc includes to fit your needs. The last function in our yacc specification is main... in case you were wondering where it was. This example still has an ambiguous grammar. Although yacc will issue shift-reduce warnings it will still process the grammar using shift as the default operation.

In this section we will extend the calculator from the previous section to incorporate some new functionality. New features include arithmetic operators multiply and divide. Parentheses may be used to over-ride operator precedence, and single-character variables may be specified in assignment statements. The following illustrates sample input and calculator output:

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

```
calc:  5
user: x + 2*y
calc:  37
```

The lexical analyzer returns VARIABLE and INTEGER tokens. For variables yylval specifies an index to the symbol table sym. For this program sym merely holds the value of the associated variable. When INTEGER tokens are returned, yylval contains the number scanned. Here is the input specification for lex:

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

%%

  /* variables */
[a-z]    {
           yylval = *yytext - 'a';
           return VARIABLE;
         }

  /* integers */
[0-9]+   {
           yylval = atoi(yytext);
           return INTEGER;
         }

  /* operators */
[-+()=/*\n] { return *yytext; }

  /* skip whitespace */
[ \t]     ;

  /* anything else is an error */
.          yyerror("invalid character");

%%

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

The input specification for yacc follows. The tokens for INTEGER and VARIABLE are utilized by yacc to create #defines in y.tab.h for use in lex. This is followed by definitions for the arithmetic operators. We may specify %left, for left-associative or %right for right associative. The last

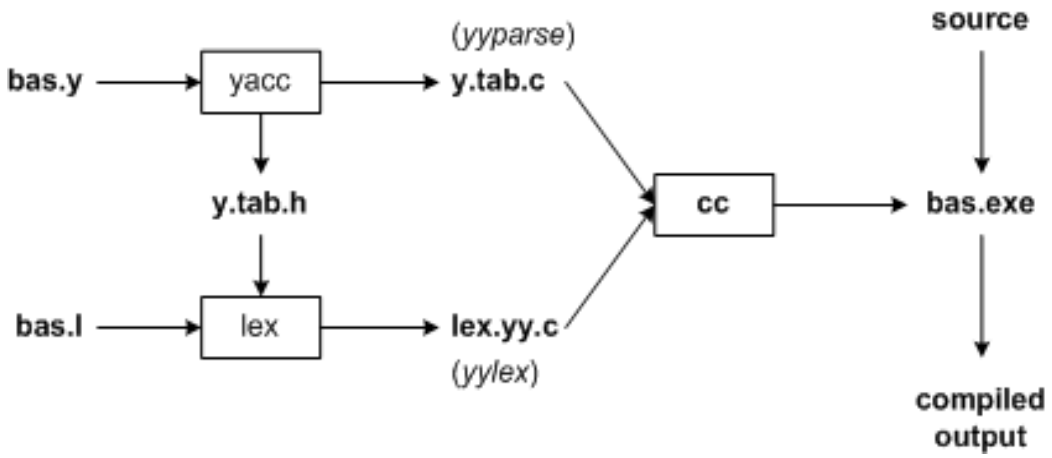
definition listed has the highest precedence. Consequently multiplication and division have higher precedence than addition and subtraction. All four operators are left-associative. Using this simple technique we are able to disambiguate our grammar.

```
%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'

%{
  void yyerror(char *);
  int yylex(void);
  int sym[26];
%}

%%

program:
      program statement '\n'
      |
      ;

statement:
      expr              { printf("%d\n", $1); }
      | VARIABLE '=' expr    { sym[$1] = $3; }
      ;

expr:
      INTEGER
      | VARIABLE           { $$ = sym[$1]; }
      | expr '+' expr     { $$ = $1 + $3; }
      | expr '-' expr     { $$ = $1 - $3; }
      | expr '*' expr     { $$ = $1 * $3; }
      | expr '/' expr     { $$ = $1 / $3; }
      | '(' expr ')'      { $$ = $2; }
      ;

%%

void yyerror(char *s) {
  fprintf(stderr, "%s\n", s);
}

int main(void) {
  yyparse();
  return 0;
}
```

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.