Name: Sanyukta Tamhankar

Roll: PA58

Batch: A4

Lab Assignment 8

Title: Parser for Arithmetic Grammar using YACC.

Aim: Write a program using LEX and YACC to create Parser for Arithmetic Grammar ---

-Design Calculator.

Objective:

1. To understand Yacc Tool.
2. To study how to use Yacc tool for implementing Parser.
3. To understand the compilation and execution of *. y file.

Theory: Write in brief for following:

1. Introduction to Yacc –

A parser generator is a program that takes as input a specification of a syntax, and produces as output a procedure for recognizing that language. Historically, they are also called compiler-compilers.
YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator. YACC was originally designed for being complemented by Lex.

2. Study of *. y file(specification of y file)

Input File:
YACC input file is divided into three parts.

/* definitions */

....

%%

/* rules */

....

%%

/* auxiliary routines */

....

Input File: Definition Part:

- The definition part includes information about the tokens used in the syntax definition:

%token NUMBER

%token ID

-
- Yacc automatically assigns numbers for tokens, but it can be overridden by

%token NUMBER 621

-
- Yacc also recognizes single characters as tokens. Therefore, assigned token numbers should not overlap ASCII codes.

- The definition part can include C code external to the definition of the parser and variable declarations, within **%{** and **%}** in the first column.

- It can also include the specification of the starting symbol in the grammar:

%start nonterminal

-

Input File: Rule Part:

- The rules part contains grammar definition in a modified BNF form.

- Actions is C code in { } and can be embedded inside (Translation schemes).

Input File: Auxiliary Routines Part:

- The auxiliary routines part is only C code.
- It includes function definitions for every function needed in rules part.
- It can also contain the main() function definition if the parser is going to be run as a program.

4.Compilation and Execution Process-

For Compiling YACC Program:

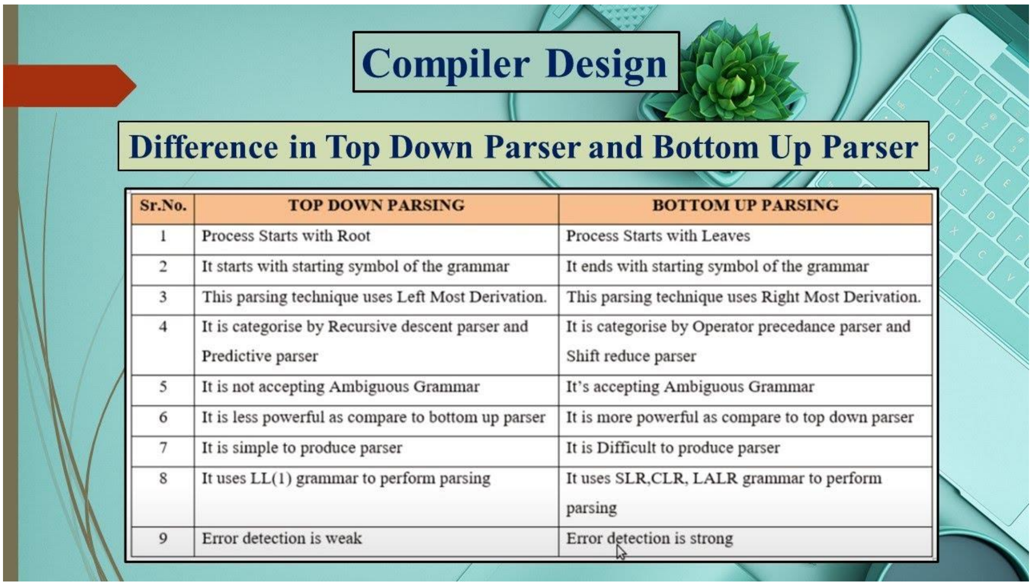
1. Write lex program in a file file.l and yacc in a file file.y
2. Open Terminal and Navigate to the Directory where you have saved the files.
3. type lex file.l
4. type yacc file.y
5. type cc lex.yy.c y.tab.h -ll
6. type ./a.out

Input: Source specification (*. y) file for arithmetic expression statements.

Output: Result of Arithmetic Expression

FAQs:

1. Differentiate between top down and bottom-up parsers.



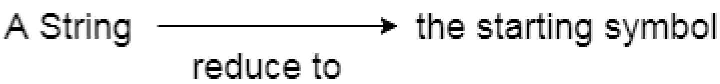
Compiler Design

Difference in Top Down Parser and Bottom Up Parser

Sr.No.	TOP DOWN PARSING	BOTTOM UP PARSING
1	Process Starts with Root	Process Starts with Leaves
2	It starts with starting symbol of the grammar	It ends with starting symbol of the grammar
3	This parsing technique uses Left Most Derivation.	This parsing technique uses Right Most Derivation.
4	It is categorise by Recursive descent parser and Predictive parser	It is categorise by Operator precedance parser and Shift reduce parser
5	It is not accepting Ambiguous Grammar	It's accepting Ambiguous Grammar
6	It is less powerful as compare to bottom up parser	It is more powerful as compare to top down parser
7	It is simple to produce parser	It is Difficult to produce parser
8	It uses LL(1) grammar to perform parsing	It uses SLR,CLR, LALR grammar to perform parsing
9	Error detection is weak	Error detection is strong

2. Explain working of shift-reduce parser.

  - hift reduce parsing is a process of reducing a string to the start symbol of a grammar.
  - Shift reduce parsing uses a stack to hold the grammar and an input tape to hold the string.



  - Sift reduce parsing performs the two actions: shift and reduce. That's why it is known as shift reduces parsing.
  - At the shift action, the current symbol in the input string is pushed to a stack.
  - At each reduction, the symbols will replaced by the non-terminals. The symbol is the right side of the production and non-terminal is the left side of the production.

3. Explain how communication between LEX and YACC is carried out.

As the two generated analysers then have to **communicate** certain information in both directions, **Lex and Yacc** know and use **each**

other's conventions. For example, the parser generated by **Yacc** calls yylex **each** time it needs a token. ... (The parser's own main function is named yyparse .)

4. How YACC resolves ambiguities within given grammar.

   A reduce/reduce conflict is **resolved** by choosing the conflicting production listed first in the **Yacc** specification. A shift/reduce conflict is **resolved** in favor of shift. Note that this rule correctly **resolves** the shift/reduce conflict arising from the dangling-else **ambiguity**.

CODE & OUTPUT:

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

%%
[0-9]+    {yylval.intval=atoi(yytext);
                 return NUMBER; }

"sin"       {return SIN; }

"cos"        {return COS; }

"tan"        {return TAN; }

[a-z]+      {strcpy(yylval.fchar,yytext);
                  return NAME; }

[\t ];
\n    return 0;

.        {return yytext[0]; }
%%

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

```
        #include<stdlib.h>
        #include<math.h>
        #include<stdio.h>
%}

%union{
                char fchar;
                double fval;
                int intval;
};

%token   SIN
%token   COS
%token   TAN
%token   <fchar>NAME
%token   <intval>NUMBER
%type     <fval>exp
%left       '+' ,  '-'
%left       '*' ,  '/'
%left       '^' ,  ' '
%%

stmt:  NAME'='exp { printf("=%f\t\n" ,$3); }
           |  exp  { printf("=%f\n",$1); };

exp  :   exp'+'exp { $$ = $1 + $3; }
           |exp'-'exp { $$ = $1 - $3; }
           |exp'*'exp { $$ = $1 * $3; }
           |SIN' 'exp { $$ = sin ($3*3.14/180); }
           |COS' 'exp { $$ = cos ($3*3.14/180); }
           |TAN' 'exp { $$ = tan ($3*(22/7)/180); }

           | exp'/'exp {
                             if($3==0)
                             {
                                     printf("\nDivide by zero.");
                             }
                             else
                              {
                                  $$ = $1 / $3;
                              }
                         }
           | NUMBER { $$ = $1; };
```

```
%%

void yyerror(char *error)
  {
     printf("%s",error);
   }

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