Name : Gaurav Thakkar
Class : CSE 1
Roll : 2183505
Batch : B

Assignment Number: 7

TITLE: Generating abstract syntax tree using LEX and YACC.

PROBLEM STATEMENT:
Generating abstract syntax tree using LEX and YACC.

OBJECTIVE:

1. To understand and generate an Abstract Syntax Tree.
2. To apply algorithmic strategies, Software Engineering and Testing while solving problems.
3. To develop problem solving abilities using Mathematical Modelling.

THEORY:

Lex is a computer program that generates lexical analyzers. It reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.
The structure of a Lex file is divided into three sections, separated by lines that contain only two percent signs, as follows:

```
Definition section
%%
Rules section
%%
C code section
```

Yacc is a computer program for the Unix operating system. The name is an acronym for "Yet Another Compiler Compiler". It is a LALR parser generator. Yacc produces only a parser (phrase analyzer); for full syntactic analysis this requires an external lexical analyzer to perform the first tokenization stage (word analysis), which is then followed by the parsing stage proper.

A YACC source program is structurally similar to a LEX one.

```
Declarations
%%
Rules
%%
routines
```

The declaration section may contain the following items.

Declarations of tokens. Yacc requires token names to be declared as such using the keyword % token.

Declaration of the start symbol using the keyword %start

C declarations: included files, global variables, types.

C code between %{ and %}.

The rule section is compulsary in a yacc file.
A rule has the form:

```
nonterminal : sentential form
            | sentential form
             ................
            | sentential form
            ;
```

Actions may be associated with rules and are executed when the associated sentential form is matched.

The subroutine section is an optional section and can be omitted.

An abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in not representing every detail appearing in the real syntax. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches.

This distinguishes abstract syntax trees from concrete syntax trees, traditionally designated parse trees, which are often built by a parser during the source code translation and compiling process. Once built, additional information is added to the AST by means of subsequent processing. Abstract syntax trees are data structures widely used in compilers, due to their property of representing the structure of program code. An AST is usually the result of the syntax analysis phase of a compiler. It often serves as an intermediate representation of the program through several stages that the compiler requires, and has a strong impact on the final output of the compiler. The AST is used intensively during semantic analysis, where the compiler checks for correct usage of the elements of the program and the language. The compiler also generates symbol tables based on the AST during semantic analysis. A complete traversal of the tree allows verification of the correctness of the program.

IMPLEMENTATION DETAILS / DESIGN LOGIC:

(Algorithm/Flow Charts/Pseudo Code/DFD/UML diagrams)

For lex:

1. Declare the header file.
2. End the declaration with %}.
3. Match RE to digit as input.
4. If match found, convert string to integer and store it in yylval. Returrn token NUMBER.
5. Similarly the other inputs with the RE and return corresponding valid token.
6. End rule section with %%.
7. If scanner encounters end of file, yywrap returns 1 else returns 0. 8. End.

For yacc:

1. Include header files.

2. Define struct node with fields left , right int pointers and token char pointer.
3. End declaration with %}.
4. Define tokens NUMBER, PLUS, MINUS, TIMES, LEFT_PARENTHESIS, RIGHT_PARENTHESIS and END.
5. If final expression evaluates , print tree
6. Declare the grammar. It is in the form of a rule and the corresponding action. It starts with the start symbol on the LHS and a string of non-terminals on the RHS are produced. The LHS can produce one or more than one strings of non-terminals on the RHS which are separated by pipes.
7. End rules section with %%.

8. Declare main and call yyparse().
9. Define mknode and printtree functions.
10. End.

Input : The input is of the kind Input 1 = (9+8*7+9-2).

Output : The abstract tree is :(- (+ (+ 9 (* 8 7)) 9) 2) .

MATHEMATICAL MODEL:

S is the system perspective.

S = {I, O, F, Success, Failure}

I= Input given.

O= Output generated.

F= Functions used.

Success= Desired output generated.

Failure= Desired output not generated or forced exit due to system error.

I= {i1, i2, i3….,in}

Where, i1 = exp

i2 = exp + exp

i3 = exp * exp

O = Generation of Abstract syntax Tree

F = {yywrap(),yyparse( ), yyerror( ), mknode( ),printtree()}

yyparse()= {Calls yylex() to obtain each token, whenever it needs one}

yyerror()= {Used to print the error message when an error is occurred in parsing of input}

yywrap()= {Function is called when scanner encounters end of file. If yywrap() returns 0, then scanner continues scanning. When yywrap() returns 1, that means end of file is encountered}

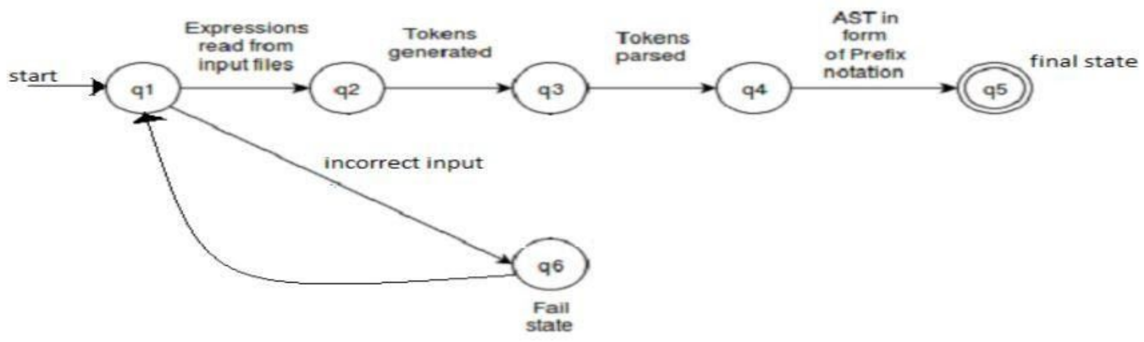
mknode()= {Creates a node with the field operator having operator as label, and two pointers to left and right}

printtree()= {Used to print the resultant abstract syntax tree}

Success = Correct AST generated.

Failure = Correct AST not generated.

State Diagram:



q1 – Start state . Input file is read.

q2 - Tokens are generated.

q3 – Tokens are parsed.

q4 – Abstract syntax tree is created and displayed.

q5 – Final state.

q6 – Failure state.

Execution Commands :

```
> yacc -d ast_y.y
> lex ast.l
> gcc y.tab.c lex.yy.c
> ./a.out
```

CODE:-

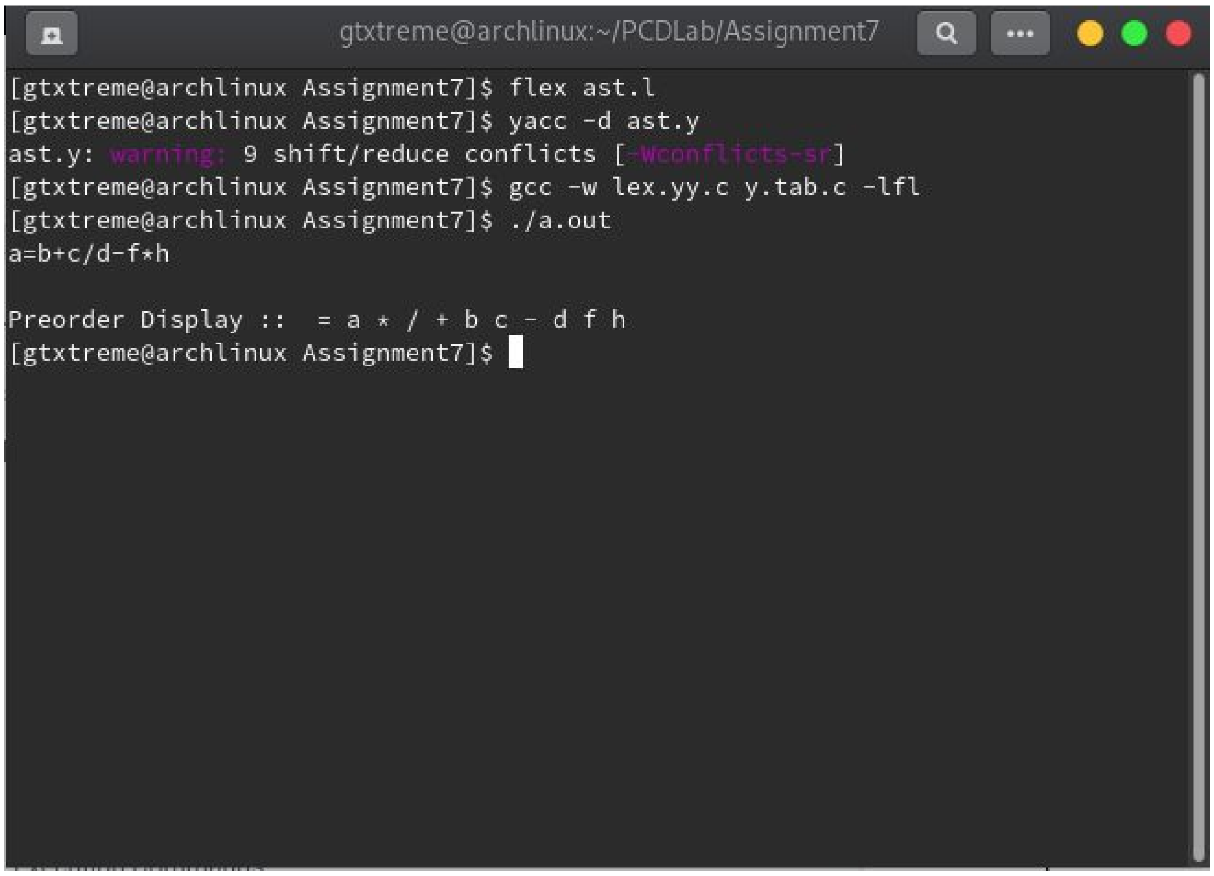
Lex File :

```
%{
#include"y.tab.h"
extern char yyval;
void yyerror(char *error);
%}

L [a-zA-Z]
%%

{L} {yylval.charval=(char)yytext[0];return ID;}
.         {
                  return *yytext;
          }
[\t];
[\n] return 0;

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

YACC File :

```
%{
#include<stdio.h>
#include<math.h>
#include<ctype.h>
#include<stdlib.h>
extern int yylex();
extern int yyparse();

typedef union type{
          int intval;
          char charval;
}type;

typedef struct node{
	type data;
	struct node *lptr,*rptr;
}node;
node* mknode(char dt,node* ln,node* rn);
node* mkleaf(char dt,node* ln,node* rn);
int op;
%}
%union{
	int intval;
	char charval;
	struct node *nptr;
	}
%token <intval>NUM
%token <charval>ID
%type <nptr>exp
%left '*' '/'
%left '+' '-'
%right '^'
%%
stmt	:exp	{	printf("\nPreorder Display :: ");
				display($1);
				printf("\n");

		}
	;
exp	:ID		{
			$$=mkleaf($1,NULL,NULL);
			}
	|exp'='exp	{
			$$=mknode('=',$1,$3);
			}
	|exp'+'exp	{
			$$=mknode('+',$1,$3);
			}
	|exp'-'exp	{

			$$=mknode('-',$1,$3);
			}
	|exp'*'exp	{
			$$=mknode('*',$1,$3);
			}
	|exp'/'exp	{
			$$=mknode('/',$1,$3);
			}
	;
%%
void yyerror(char *error){
	printf("\n%s",error);
}
node* mkleaf(char dt,node* ln,node* rn)
{
        node* temp=(node *)malloc(sizeof(node));
        temp->data.charval=dt;
        temp->lptr=ln;
        temp->rptr=rn;
        return temp;
}
node* mknode(char dt,node* ln,node* rn)
{
        node* temp=(node *)malloc(sizeof(node));
        temp->data.charval=dt;
        temp->lptr=ln;
        temp->rptr=rn;
        return temp;
}
void display(node* root){

        if(root!=NULL){

                if(root->lptr==NULL && root->rptr==NULL){

                        printf(" %c",root->data.charval);
                }
                else{
                        printf(" %c",root->data.charval);
                }
                display(root->lptr);
                display(root->rptr);
        }
}

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

OUTPUT:



```
[gtxtreme@archlinux Assignment7]$ flex ast.l
[gtxtreme@archlinux Assignment7]$ yacc -d ast.y
ast.y: warning: 9 shift/reduce conflicts [-Wconflicts-sr]
[gtxtreme@archlinux Assignment7]$ gcc -w lex.yy.c y.tab.c -lfl
[gtxtreme@archlinux Assignment7]$ ./a.out
a=b+c/d-f*h

Preorder Display ::  = a * / + b c - d f h
[gtxtreme@archlinux Assignment7]$ 
```

Conclusion: Successful generation of abstract syntax tree using lex and yacc with the desired outcomes generated.