ASSIGNMENT NO: 2.

Title : Representing ambiguous grammar using LEX & YACC.

Problem Statement :

Implement a parser for an expression grammar using YACC & LEX for the subset of C. Cross check your o/p with standford LEX and YACC.

Objective:

- To understand basic syntax of YACC specifications, built-in functions & variables.
- Be proficient on writing grammars to specify syntax.
- Be able to use YACC to generate parsers.

S/w package :- . 64 bit latest Computer
. 128 MB RAM
. 40 GB HDD
. Keyboard.

Theory :

Ambiguity :

A grammar is said to be an ambiguous grammar if there is some string that it can generate in more than one way (i.e., the string has more than one parse tree or more than one leftmost derivation.)

The Context free grammar

$A \rightarrow A + A \mid A - A \mid a$.

is ambiguous since there are two leftmost derivations for the string a+a+a:

$A \rightarrow A+A$	$A \rightarrow A+A$.
$\rightarrow a+A$	$\rightarrow A+A+A$.
$\rightarrow a+A+$ A	$\rightarrow a+A+A$.
$\rightarrow a+a+A$.	$\rightarrow a+a+A$.
$\rightarrow a+a+a$	$\rightarrow a+a+a$.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, This will result in shift/reduce conflict. When there are shift/reduce or reduce/reduce conflicts, YACC still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a disambiguating rule.

- In a shift/reduce conflict, the default is to do shift.
- In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule (in the input sequence).

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.

The precedence and associatives are attached to tokens in the declaration section. This is done by a series of lines beginning with a Yacc keyword: %left, %right or %nonassoc followed by a list of tokens. All the tokens on the same lines are assumed to have the same precedence level & associativity; the lines are listed in the order of increasing precedence or binding strength.

Thus, %left '+' '-'
%left '*' '/'.

describes the precedence & associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star & slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators that may not associate with themselves.

Passing Values between Actions

To get values generated by other actions, an action can use the yacc parameter keywords that begins with a dollar sign ($1, $2, ...). These keywords refer to the values returned by the components of the right side of a rule, reading from left to right. For example, if the rule is:

A: B CD;

then $1 has the value returned by the rule that

recognized B, $2 has the value returned by the rule that recognized C, $3 the value returned by the rule that recognized D. To return a value, the action sets the pseudo-variable $$ to some value. For eg, the foll" action returns a value of 1:

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

Algorithm:

1. Write lex code to separate out the tokens from arithmetic expression.
2. Write YACC code to check the syntax of the arithmetic expression.
3. Accept a input arithmetic expression for the user.
4. If input is as per syntax, evaluate the expression

Test I/p:

Arithmetic Expression : 10+10*50.

Test o/p:-

Syntax of Arithmetic Expression is correct.
Evaluated Arithmetic Expression: 510.

Conclusion:-

Hence, implemented a parser for an expression grammar using YACC & LEX for the subset of C.