BANGLADESH UNIVERSITY OF PROFESSIONALS

FACULTY OF SCIENCE & TECHNOLOGY

DEPT. OF COMPUTER SCIENCE & ENGINEERING (CSE)

Lab Report
NO - 03

COURSE NAME: Compiler Laboratory **CODE:** CSE-3104

TITL E : Calculator using YACC

SUBMITTED TO:

Rumana Yasmin	Sayma Alam Suha
Lecturer	Lecturer
Department of CSE	Department of CSE
FST, BUP	FST, BUP

SUBMITTED BY:

NAME	**:** Zuairia Kabir
ROLL	**:** 2252421013
SECTION	**:** A
LEVEL/TERM	**:** 3rd

SEMESTER: 1st

DATE OF SUBMISSION: December 19, 2024

Experiment number:03

Title: Calculator using YACC

Introduction: Bison is a widely used parser generator that is part of the GNU project. It is designed to convert a formal grammar, specified using Yacc-like syntax, into a parser. The parser generated by Bison can then be used to process input according to the rules defined in the grammar, typically for applications like compilers or interpreters. Bison is particularly known for its ability to handle context-free grammars and produce efficient parsers.

Bison takes in a grammar specification written in a high-level syntax, which describes how valid sequences of tokens (e.g., numbers, operators) can be combined into larger structures (e.g., expressions, statements). It then generates C or C++ code for the parser, which can be integrated with a lexer (such as one generated by Flex) to perform lexical analysis. The parser interprets or processes these sequences and often produces some output or performs computations based on the input.

Bison supports LR parsing (left-to-right, rightmost derivation), allowing it to handle a wide range of grammars, including many that are not suitable for simple recursive descent parsers. It also includes advanced features, such as error handling, precedence rules, and semantic actions, making it a powerful tool for building language processors. Bison is commonly used in the development of compilers, interpreters, and other software that needs to process structured input.

Objective:

- To understand the Basics of Parser Generation
- To implement Arithmetic Expression Parsing
- To integrate Bison with Lexical Analysis

Problem:

Implement a YACC code for the given expressions: +, -, *, /

Solution Code:

Lex:

```
%{
#include "calc.tab.h"
%}
%%
[0-9]+ {yylval = atoi(yytext); return ID;}
[+\-*/] {return yytext[0];}
[ \t]* {}
"\n" { return '\n'; }
. { return yytext[0]; }
```

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

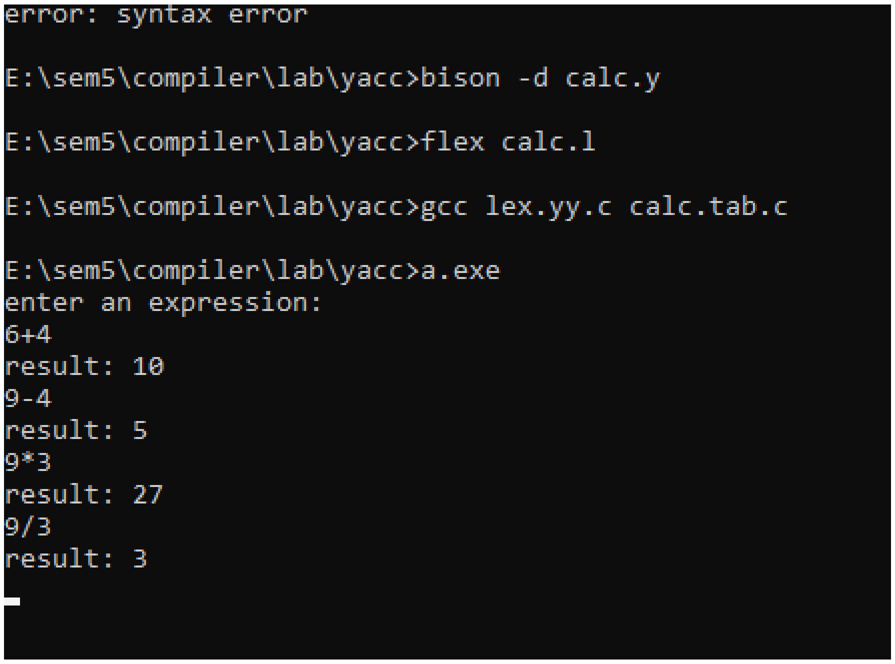
YACC:

```
%{
#include<stdio.h>
#include<stdlib.h>
int yylex();
void yyerror(const char *s);
%}
%token ID
%left '+' '-'
%left '*' '/'
%%
input: /* empty */
     | input expr '\n' {}
     ;
expr : expr '+' expr {printf("result: %d\n",$1 + $3);$$ = $1 + $3;}
     | expr '-' expr {printf("result: %d\n",$1 - $3);$$ = $1 - $3;}
     | expr '*' expr {printf("result: %d\n",$1 * $3);$$ = $1 * $3;}
     | expr '/' expr {printf("result: %d\n",$1 / $3);$$ = $1 / $3;}
     | ID {$$ = $1;};
%%

int main() {
 printf("enter an expression: \n");
 yyparse();
 return 0;
}

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

Output :



```
error: syntax error

E:\sem5\compiler\lab\yacc>bison -d calc.y

E:\sem5\compiler\lab\yacc>flex calc.l

E:\sem5\compiler\lab\yacc>gcc lex.yy.c calc.tab.c

E:\sem5\compiler\lab\yacc>a.exe
enter an expression:
6+4
result: 10
9-4
result: 5
9*3
result: 27
9/3
result: 3
```

Discussion: This program implements a simple calculator using Bison and Flex, which together form a compiler for arithmetic expressions. The primary purpose of this code is to demonstrate the process of parsing and evaluating basic mathematical operations, such as addition, subtraction, multiplication, and division. The calculator can evaluate expressions containing numbers and operators, respecting operator precedence and associativity rules.

The program begins by defining grammar rules using Bison, where expressions are recursively parsed based on their structure. Each rule corresponds to an arithmetic operation, and the program computes the result by performing operations on the values of the operands ($1 and $3 represent the left and right operands, while $$ represents the result). The expr rule defines how different operators (+, -, *, /) and the ID (used for numbers) are evaluated.

The input is read from the user, and after parsing the expression, the corresponding result is printed to the console. If an error occurs during parsing, the program displays an error message via the yyerror function. Following this process described above, a arithmetic calculator was built.