Laboratory 8

Title of the Laboratory Exercise: Program to test the syntax of a simple expression and evaluate an arithmetic expression involving operating +, -, * and /

1. Introduction and Purpose of Experiment

Students learn to

2. Aim and Objectives

Aim

- To write a program to

Objectives

At the end of this lab, the student will be able to

- Define

3. Experimental Procedure

Students are required to carry out the following steps:

- Algorithm
- Write the Lex program
- Write yacc program
- Compile and execute the program (steps)
- Complete the documentation for the given problem

4. Presentation of Results

calc.l

```
%{
   /* Definition section */

  #include <stdio.h>
  #include "calc.tab.h"
  extern int yylval;

%}

%option noyywrap
```

```
/* Rule Section */
%%
[0-9]+   {
          yylval=atoi(yytext);
          return NUMBER;
         }
[\t]     ;

[\n]     return 0;

.        return yytext[0];

%%
```

calc.y

```
%{
   /* Definition section */
   #include <stdio.h>
   int flag=0;

   int yylex(void);
   void yyerror(char* s);
%}

%token NUMBER

%left '+' '-'

%left '*' '/' '%'

%left '(' ')'

/* Rule Section */
%%

ArithmeticExpression : E {
                           printf("\nResult=%d\n", $$);
                           return 0;
                        }
                      ;
 E    :  E'+'E    { printf("E + E\n"); $$ = $1 + $3; }
      |  E'-'E    { printf("E + E\n"); $$ = $1 - $3; }
      |  E'*'E    { printf("E * E\n"); $$ = $1 * $3; }
      |  E'/'E    { printf("E / E\n"); $$ = $1 / $3; }
      |  E'%'E    { printf("E % E\n"); $$ = $1 % $3; }
      |  '('E')'  { printf("(E)\n");   $$ = $2; }
      |  NUMBER   { printf("NUMBER\n");  $$ = $1; }
      ;
```

```
%%
```

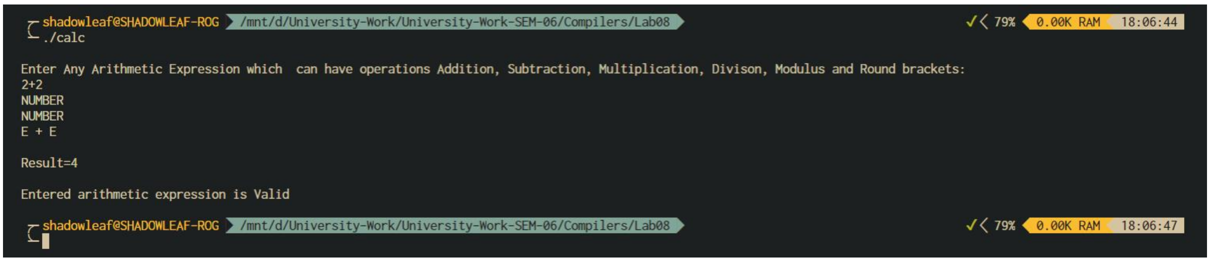
main.c

```c
#include <stdio.h>

#include "calc.tab.h"

extern int flag;
//driver code
int main() {
    printf("\nEnter Any Arithmetic Expression which  can have operations Addition, Su
btraction, Multiplication, Divison, Modulus and Round brackets:\n");

    yyparse();
    if (flag == 0) {
        printf("\nEntered arithmetic expression is Valid\n\n");
    }

    return 0;
}

void yyerror(char* s) {
    printf("\nEntered arithmetic expression is Invalid: %s\n", s);
    flag = 1;
}
```

Makefile

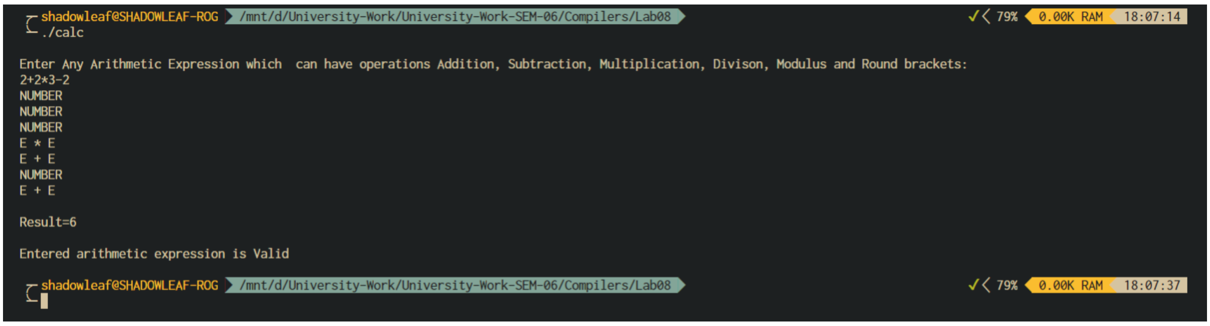
```makefile
all: calc

calc.tab.c calc.tab.h: calc.y
    bison -d calc.y

lex.yy.c: calc.l calc.tab.h
    flex calc.l

calc: main.c lex.yy.c calc.tab.c calc.tab.h
    gcc -w -o calc main.c calc.tab.c lex.yy.c

clean:
    rm calc calc.tab.c calc.tab.h lex.yy.c
```

5. Analysis and Discussions



```
shadowleaf@SHADOWLEAF-ROG /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab08    ✓ 79% 0.00K RAM 18:06:44
./calc

Enter Any Arithmetic Expression which  can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:
2+2
NUMBER
NUMBER
E + E

Result=4

Entered arithmetic expression is Valid
shadowleaf@SHADOWLEAF-ROG /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab08    ✓ 79% 0.00K RAM 18:06:47
```

Figure 0-1 OUTPUT 1



```
shadowleaf@SHADOWLEAF-ROG /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab08    ✓ 79% 0.00K RAM 18:07:14
./calc

Enter Any Arithmetic Expression which  can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:
2+2*3-2
NUMBER
NUMBER
NUMBER
E * E
E + E
NUMBER
E + E

Result=6

Entered arithmetic expression is Valid
shadowleaf@SHADOWLEAF-ROG /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab08    ✓ 79% 0.00K RAM 18:07:37
```

Figure 0-2 OUTPUT 2

```
shadowleaf@SHADOWLEAF-ROG /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab08    ✓ 79% 0.00K RAM 18:07:53
./calc

Enter Any Arithmetic Expression which  can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:
2-(2*2)
NUMBER
NUMBER
NUMBER
E * E
(E)
E + E

Result=-2

Entered arithmetic expression is Valid
shadowleaf@SHADOWLEAF-ROG /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab08    ✓ 79% 0.00K RAM 18:07:59
```

Figure 0-3 OUTPUT 3

6. Conclusions

Yacc provides operator precedence and associativity rules for eliminating ambiguity and resolving shift-reduce conflicts

Example on precedence and associativity of operators:

```
%nonassoc RELOP

%left ADDOP

%left MULOP

%right EXPOP
```

The order of declarations defines precedence of operators

RELOP has least precedence and EXPOP has the highest

ADDOP has higher precedence than RELOP

%left declarations means left-associative

%right declarations means right-associative

%nonassoc declarations means non-associative

The operator precedence and associativity resolve conflicts

Given the two productions:

```
E : E op1 E ;
E : E op2 E ;
```

- Suppose E op1 E is on top of parser stack and next token is op2
- If op2 has a higher precedence than op1, we shift
- If op2 has a lower precedence than op1, we reduce
- If op2 has an equal precedence to op1, we use associativity
    - If op1 and op2 are left-associative, we reduce
    - If op1 and op2 are right-associative, we shift
    - If op1 and op2 are non-associative, we have a syntax error

7. Comments

a. Limitations of Experiments

- Bison only supports BNF, which makes grammars more complicated.
- Bison supports two parsing algorithms that cover all ranges of performance and languages. It gives cryptic error messages

b. Limitations of Results

The solution program made for this experiment is limited to only one expression at a time, and the input is restricted to stdin.

c. Learning happened

We learnt how to use bison rules to define a grammar to make a simple calculator with operator associativity and precedence.

d. Recommendations

define the association properly and make the program more generic by taking command line arguments to parse a given input file that contains the expressions to be evaluated.

Component	Max Marks	Marks Obtained
Viva	6	
Results	7	
Documentation	7	
Total	20	