NAME: SATYAJIT GHANA REG NO: 17ETCS002159

Laboratory 6

Title of the Laboratory Exercise: Program to perform addition, subtraction, multiplication, division and power. Note: Without Precedence.

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
- Compile and execute the program (steps)
- Complete the documentation for the given problem

4. Presentation of Results

lab06.l

```
%{
   /* Definition section */

  #include <iostream>
  #include <string>
  #define YYSTYPE int
  #include "lab06.tab.h"
  extern YYSTYPE yylval;

%}
```

```
%option noyywrap

/* Rule Section */
%%

[0-9]+  { yylval = atoi(yytext); return NUMBER; }
[ ] ;
[-+\n] return *yytext;
.                 { return yytext[0]; }

%%
```

lab06.y

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

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

   #define DEBUG
   #undef DEBUG
%}

%token NUMBER

%left '+' '-'
%left '*' '/' '%'
%left '(' ')'

/* Rule Section */
%%

program : program E '\n' { std::cout << "\b\b:= " << $2 << '\n'; }
        | { }
        ;
 E    :  E'+'E    {
                    #ifdef DEBUG
                    printf("E + E\n");
                    #endif
                    $$ = $1 + $3;
                  }
      |  E'-'E    {
                    #ifdef DEBUG
                    printf("E + E\n");
                    #endif
                    $$ = $1 - $3;
                  }
      |  E'*'E    {
                    #ifdef DEBUG
                    printf("E * E\n");
                    #endif
                    $$ = $1 * $3;
                  }
      |  E'/'E    {
                    #ifdef DEBUG
                    printf("E / E\n");
                    #endif
                    $$ = $1 / $3;
                  }
      |  E'%'E    {
                    #ifdef DEBUG
                    printf("E % E\n");
                    #endif
                    $$ = $1 % $3;
                  }
      |  '('E')'  {
                    #ifdef DEBUG
                    printf("(E)\n");
                    #endif
                    $$ = $2;
                  }
      |  NUMBER   {
                    #ifdef DEBUG
                    printf("NUMBER\n");
                    #endif
                    $$ = $1;
                  }
      ;

%%
```

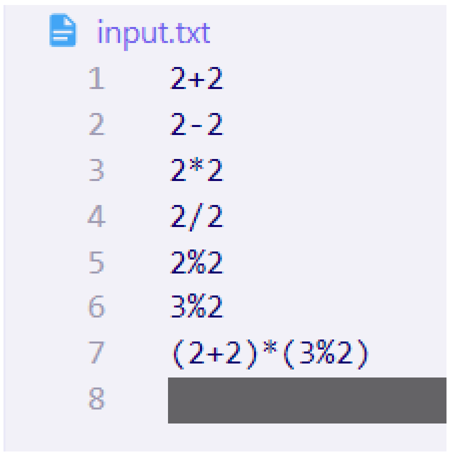
main.cpp

```
#include <stdio.h>

#include <iostream>

#include "lab06.tab.h"

extern int flag;
extern FILE* yyin;

auto main(int argc, char* argv[]) -> int {
    if (argc > 1) {
        yyin = fopen(argv[1], "r");
    }
    yyparse();
    if (flag == 0) {
        std::cout << "parse successful" << '\n';
    }
    return 0;
}

void yyerror(char* s) {
    std::cout << "Error Parsing: " << s << '\n';
    flag = 1;
}
```

Makefile

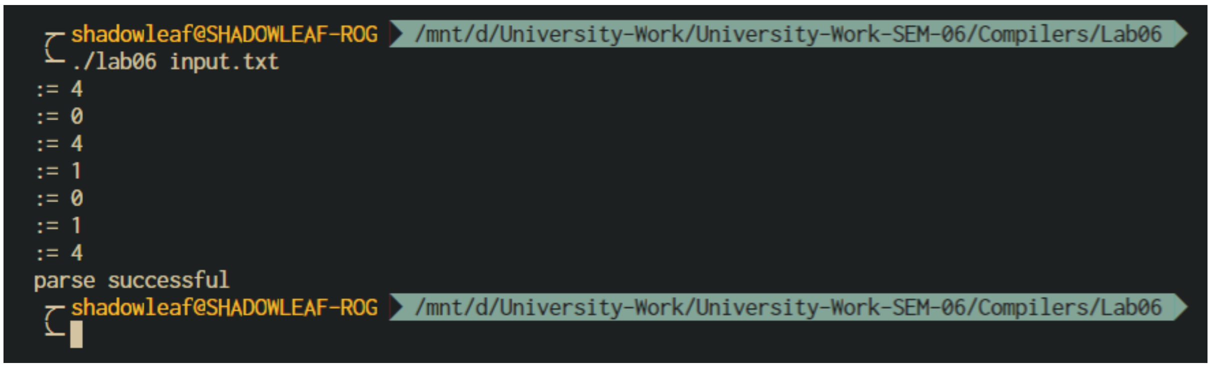
```
# name of the files and the program
NAME = lab06

# compiler setup
CC = g++ -g
LEX = flex
YACC = bison -d

all: ${NAME}

${NAME}.tab.c ${NAME}.tab.h: ${NAME}.y
    ${YACC} ${NAME}.y

lex.yy.c: ${NAME}.l ${NAME}.tab.h
    ${LEX} ${NAME}.l

${NAME}: main.cpp lex.yy.c ${NAME}.tab.c ${NAME}.tab.h
    ${CC} -o ${NAME} main.cpp lex.yy.c ${NAME}.tab.c ${NAME}.tab.h

clean:
    rm -f lex.yy.c lex.yy.o ${NAME} ${NAME}.tab.c ${NAME}.tab.h
    rm -f ${NAME}.tab.o
```

5. Analysis and Discussions



```
input.txt
2+2
2-2
2*2
2/2
2%2
3%2
(2+2)*(3%2)

```

Figure 0-1 INPUT file



```
┌ shadowleaf@SHADOWLEAF-ROG  /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab06
└ ./lab06 input.txt
:= 4
:= 0
:= 4
:= 1
:= 0
:= 1
:= 4
parse successful
┌ shadowleaf@SHADOWLEAF-ROG  /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab06
└ 
```

Figure 0-2 OUTPUT

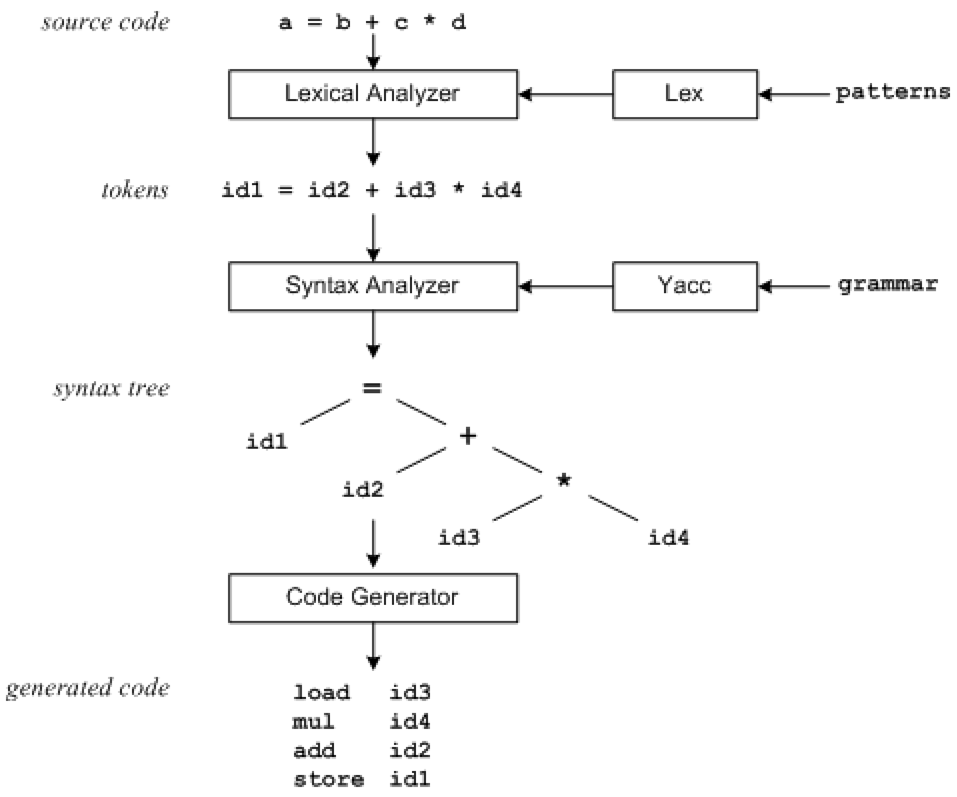


Figure 0-3 Overview of the program

The patterns in the above diagram is a file you create with a text editor. Lex will read your patterns and generate C code for a lexical analyzer or scanner. The lexical analyzer matches strings in the input, based on your patterns, and converts the strings to tokens. Tokens are numerical representations of strings, and simplify processing.

When the lexical analyzer finds identifiers in the input stream it enters them in a symbol table. The symbol table may also contain other information such as data type (integer or real) and location of each variable in memory. All subsequent references to identifiers refer to the appropriate symbol table index.

6. Conclusions

The yacc (yet another compiler compiler) utility provides a general tool for imposing structure on the input to a computer program. Before using yacc, you prepare a specification that includes:

A set of rules to describe the elements of the input

Code to be invoked when a rule is recognized

Either a definition or declaration of a low-level scanner to examine the input yacc then turns the specification into a C-language function that examines the input stream. This function, called a parser, works by calling the low-level scanner.

The scanner, called a lexical analyzer, picks up items from the input stream. The selected items are known as tokens. Tokens are compared to the input construct rules, called grammar rules.

When one of the rules is recognized, the code you have supplied for the rule is invoked. This code is called an action. Actions are fragments of C-language code. They can return values and use values returned by other actions.

Rules Section.

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.

7. Comments

   a. Limitations of Experiments

The operator precedence and associativity resolve conflicts

Given the two productions:

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

   b. Limitations of Results

Reduce-reduce conflict caused by the limitation of `LALR(1)`

```
S : X B c | Y B d ;
X : A ;
Y : A ;
```

c. Learning happened

An ambiguous grammar provides a shorter specification

- Can be more natural than any equivalent unambiguous grammar
- Produces more efficient parsers for real programming languages

d. Recommendations

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