

Compilers Laboratory

B. Tech. 6th Semester

Batch: 2017



Name: SATYAJIT GHANA

Reg No: 17ETCS002159

Department: Computer Science and Engineering

Faculty of Engineering & Technology

Ramaiah University of Applied Sciences

Ramaiah University of Applied Sciences

Private University Established in Karnataka State by Act No. 15 of 2013

Faculty	Engineering & Technology
Programme	B. Tech. in Computer Science and Engineering
Course	Compilers Laboratory
Year/Semester	2017/6 th Semester
Course Code	CSC312A

List of Experiments

LEX PROGRAMS

1. Program to count the number of vowels and consonants in a given string.
2. Program to find the longest word in a given string.
3. Program to count no of:
 - a. +positive and –negative integers
 - b. +positive and –negative fractions
4. Program to count the number of characters, words, spaces, end of lines in a given input file.
5. Program to count the no of ‘scanf’ and ‘printf’ statements in a C program. Replace them with ‘readf’ and ‘writef’ statements respectively.
6. Program to perform addition, subtraction, multiplication, division and power. Note: Without Precedence.

YACC & LEX PROGRAMS

7. Program to recognize a valid variable, which starts with a letter, followed by any number of letters or digits.
8. Program to test the syntax of a simple expression and evaluate an arithmetic expression involving operating +, -, * and /
9. Program to recognize strings ‘aaab’, ‘abbb’, ‘ab’ and ‘a’ using grammar ($a^n b^m$, $n \geq 0$, $m \geq 0$)

Laboratory 1

Title of the Laboratory Exercise: Program to count the number of vowels and consonants in a given string

1. Introduction and Purpose of Experiment

Students learn to use Lex program to find out vowels and consonants in a given string

2. Aim and Objectives

Aim

- To write a program to count the number of vowels and consonants in a given string

Objectives

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

- Define regular expression for vowels and consonants
- Count the number of vowels and consonants

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

Algorithm:

count_vowels_consonants() :

1. if `regex_match([aeiouAEIOU])`
2. `vowel++`
3. else if `regex_match([a-zA-Z])`
4. `consonant++`

5. print vowels and consonants

main.cpp

```
#include <iostream>

int vowel_cnt = 0;
int consonant_cnt = 0;

extern int yylex();

int main(int argc, char* argv[]) {
    std::cout << "Enter your stream of characters"
               << "\n";

    yylex();
    std::cout << "Found " << vowel_cnt << " Vowels and " << consonant_cnt << " Consonants"
               << "\n";

    return 0;
}
```

tokens.l

```
%{
#include <iostream>

extern int vowel_cnt;
extern int consonant_cnt;

void yyerror(const char *s);

}%

%option noyywrap

%%

[aeiouAEIOU]    { std::cout << "[VOWEL: " << yytext << "]; vowel_cnt++; }
[a-zA-Z]        { std::cout << "[CONSONANT: " << yytext << "]; consonant_cnt++; }
\n              { std::cout << "\nEND PARSE [NEWLINE]\n\n"; return 1; }
.               { std::cout << "[UNRECOGNIZED: "<< yytext << " ]"; }

%%
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0.0)
project(Lab01 VERSION 0.1.0)

include(CTest)
enable_testing()

# setup FLEX
FIND_PACKAGE(FLEX)
```

```

FLEX_TARGET(Scanner tokens.l ${CMAKE_CURRENT_BINARY_DIR}/tokens.cpp)

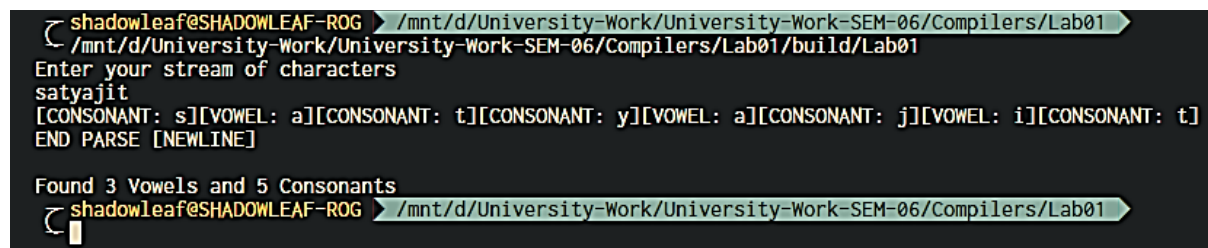
add_executable(Lab01 main.cpp tokens.l ${FLEX_Scanner_OUTPUTS})
target_include_directories(Lab01 PUBLIC ${CMAKE_CURRENT_BINARY_DIR})

set(CPACK_PROJECT_NAME ${PROJECT_NAME})
set(CPACK_PROJECT_VERSION ${PROJECT_VERSION})
include(CPack)

```

5. Analysis and Discussions

OUTPUT:



```

shadowleaf@SHADOWLEAF-ROG > /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab01
/mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab01/build/Lab01
Enter your stream of characters
satyajit
[CONSONANT: s][VOWEL: a][CONSONANT: t][CONSONANT: y][VOWEL: a][CONSONANT: j][VOWEL: i][CONSONANT: t]
END PARSE [NEWLINE]

Found 3 Vowels and 5 Consonants
shadowleaf@SHADOWLEAF-ROG > /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab01

```

Figure 0-1 Output for "satyajit"

Explanation:

The regex for matching the vowels is `[aeiouAEIOU]` which will match all upper case and lower-case vowels, and the rest of the letters `[a-zA-Z]` will be consonants, i.e. if the vowel regex does not match and the `a-zA-Z` does, then it is a consonant. Once the regex matches, the corresponding count is incremented. When a newline is encountered, the parsing is stopped and the output is displayed.

6. Conclusions

Flex is a tool for generating scanners: programs which recognize lexical patterns in text. The flex input file consists of three sections, separated by a line with just `%%` in it:

```

definitions
%%
rules
%%
user code

```

The definitions section contains declarations of simple name definitions to simplify the scanner specification, and declarations of start conditions, which are explained in a later section.

Name definitions have the form:

```
name definition
```

The rules section of the flex input contains a series of rules of the form:

```
pattern  action
```

where the pattern must be unindented and the action must begin on the same line.

When the generated scanner is run, it analyzes its input looking for strings which match any of its patterns. If it finds more than one match, it takes the one matching the most text (for trailing context rules, this includes the length of the trailing part, even though it will then be returned to the input). If it finds two or more matches of the same length, the rule listed first in the flex input file is chosen.

Once the match is determined, the text corresponding to the match (called the token) is made available in the global character pointer `yytext`, and its length in the global integer `yylen`. The action corresponding to the matched pattern is then executed (a more detailed description of actions follows), and then the remaining input is scanned for another match.

If no match is found, then the default rule is executed: the next character in the input is considered matched and copied to the standard output.

For a really detailed documentation refer to: <http://dinosaur.compilertools.net/flex/manpage.html>

7. Comments

a. Limitations of Experiments

The experiment does not define the input file i.e. either STDIN or some other user file, the buffer size isn't specified either.

b. Limitations of Results

The results are limited to very few test cases, which could have been increased to justify the regex used in the program.

c. Learning happened

The concept of regex to detect patterns was learnt, along with lex, which was used to perform some action with the matched strings.

d. Recommendations

Perform extensive testing on different forms of regex to better understand the different forms of regex that match the same string.

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

Laboratory 2

Title of the Laboratory Exercise: Program to find the longest word in a given string.

1. Introduction and Purpose of Experiment

Students learn to use Lex program to find out the longest word in a given string.

2. Aim and Objectives

Aim

- To write a program to find the longest word in a given string

Objectives

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

- Define regular expression for words
- Find the longest word in a given string

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

```
tokens.l
%{

#include <iostream>

extern int maxlen;
extern std::string maxstring;

%}

%option noyywrap

%%
```



```

[a-zA-Z0-9_\\+\\*\\/]+ {
    std::string curr_string(yytext);
    std::cout << "[" << curr_string << " : " << curr_string.length() << "
] \n";

    // if the maxstring length is less than the curr_string then store th
is
    // curr_string
    if ( maxstring.length() <= curr_string.length() ) {
        maxstring = curr_string;
    }
}

[ \\t\\r] { }
"\\n" { return 1; }
. { std::cout << "[UNRECOGNIZED]\\n"; }

```

main.cpp

```

#include <iostream>
#include <string>

extern int yylex();

std::string maxstring = "";

auto main(int argc, char* argv[]) -> int {

    std::cout << "enter stream of characters" << "\\n";

    yylex();

    std::cout << "max string: " << maxstring << ", len: " << maxstring.length() << "\\n";

    return 0;
}

```

5. Analysis and Discussions

```

shadowleaf@SHADOWLEAF-ROG > /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab02
./build/Lab02
enter stream of characters
my name is satyajit ghana
[my : 2]
[name : 4]
[is : 2]
[satyajit : 8]
[ghana : 5]
max string: satyajit, len: 8
shadowleaf@SHADOWLEAF-ROG > /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab02

```

Figure 0-1 OUTPUT

Algorithm:

find_maxstring():

1. max_string = ""
2. for each character string in stdin
3. match regex [a-zA-Z0-9_+*\./]+
4. curr_string = yytext
5. if (max_string.length < curr_string.length)
6. max_string = curr_string
7. display max_string

6. Conclusions**Lex Regular Expressions.**

A LEX regular expression is a word made of text characters (letters of the alphabet, digits, ...)

operators: " \ { } [] ^ \$ < > ? . * + | () /

A Character Class is a class of characters specified using the operator pair []. The expression

[ab] matches the string a or b.

Within square brackets most operators are ignored except the three special characters \ - ^ are which used as follows

(a) the escape character \ as above,

(b) the minus character - which is used for ranges like in digit [0-9]

(c) the *hat* character ^ as first character after the opening square bracket, it is used for complemented matches like in

NOTabc [^abc]

7. Comments

a. Limitations of Experiments

The experiment does not define the characters that are included in a string; hence a few assumptions are made, alphabet + numbers + some special characters are included in the string

b. Limitations of Results

The solution program written takes input from stdin, which is not generalized, i.e. the program should also take input from file and parse the file to find the max string.

Dynamic resizing of the input buffer is slow, as it entails rescanning all the text matched so far by the current (generally huge) token. Due to both buffering of input and read-ahead, you cannot intermix calls to `<stdio.h>` routines, such as, `getchar()`, with flex rules and expect it to work.

c. Learning happened

We learnt how to define regular expression for words and find the longest word in a given string

d. Recommendations

The characters that consist a string should be well defined in the experiment.

Try to use standard C++ string functions for operations with `yytext` to avoid input buffer resizing problems with `<stdio.h>`.

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

NAME: SATYAJIT GHANA

REG NO: 17ETCS002159

Laboratory 3

1. Title of the Laboratory Exercise: Program to count no of:
 - a. +positive and –negative integers
 - b. +positive and –negative fractions

2. Introduction and Purpose of Experiment

.

2. Aim and Objectives

Aim

Objectives

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

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

main.cpp

```
#include <iostream>

extern int yylex();

int dposcnt = 0, dnegcnt = 0, fposcnt = 0, fnegcnt = 0;

auto main(int argc, char* argv[]) -> int {
    std::cout << "enter different numbers" << '\n';

    if (yylex() == 0)
```

```

        std::cout << "\nparsed successfully" << '\n';
    else
        std::cerr << "error parsing" << '\n';

    std::cout << '\n';

    std::cout << "Positive Decimals : " << dposcnt << '\n'
        << "Negative Decimals : " << dnegcnt << '\n'
        << "Positive Fractions : " << fposcnt << '\n'
        << "Negative Fractions : " << fnegcnt << '\n';

    return 0;
}

```

lab03.1

```

%{

#include <iostream>

extern int dposcnt, dnegcnt, fposcnt, fnegcnt;

%}

%option noyywrap

white      [ \n\t]+
digit      [0-9]
integer    [digit]+
exponent   [eE] [+]?{integer}

%%

\+?[digit]+      { std::cout << "found +ve decimal : " << yytext << '\n'; dpos
cnt++; }
\+?[digit]+\.{digit}* { std::cout << "found +ve fraction : "<< yytext << '\n'; fpos
cnt++; }
-[digit]+        { std::cout << "found -
ve decimal : " << yytext << '\n'; dnegcnt++; }
-[digit]+\.{digit}* { std::cout << "found -
ve fraction : "<< yytext << '\n'; fnegcnt++; }
{white}          { ; } // eat whitespaces

```

Makefile

```

# name of the files and the program
NAME = lab03

# compiler setup
CC = g++

```

```

LEX = flex

all: ${NAME}

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

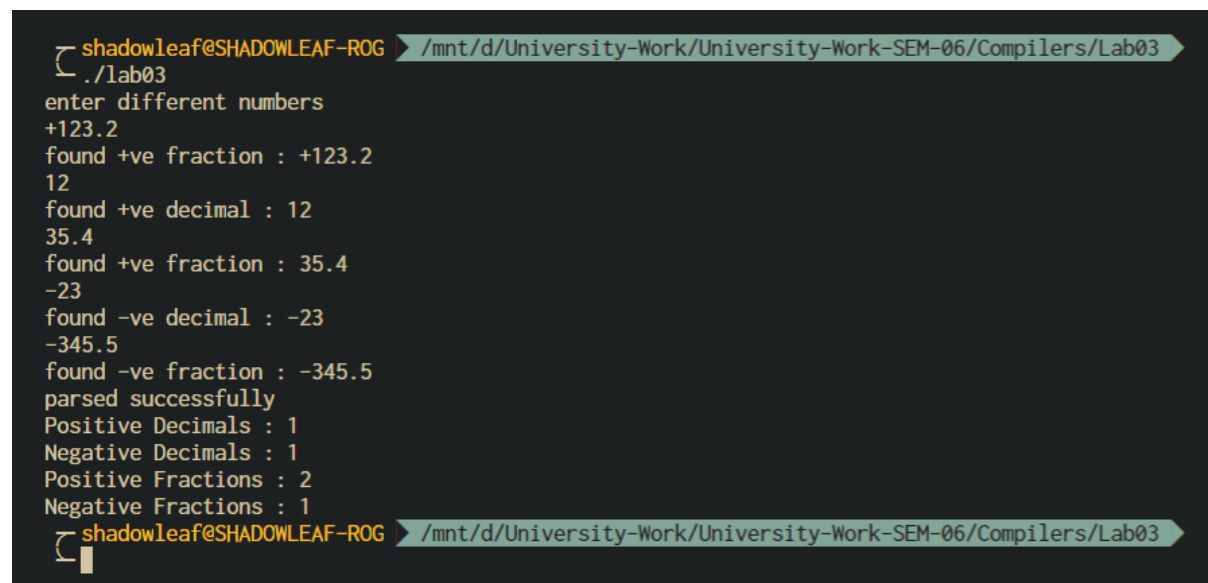
lex.yy.c: ${NAME}.l
    ${LEX} ${NAME}.l

clean:
    rm -f lex.yy.c ${NAME}

run:
    @./${NAME}

```

5. Analysis and Discussions



```

shadowleaf@SHADOWLEAF-ROG /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab03
./lab03
enter different numbers
+123.2
found +ve fraction : +123.2
12
found +ve decimal : 12
35.4
found +ve fraction : 35.4
-23
found -ve decimal : -23
-345.5
found -ve fraction : -345.5
parsed successfully
Positive Decimals : 1
Negative Decimals : 1
Positive Fractions : 2
Negative Fractions : 1
shadowleaf@SHADOWLEAF-ROG /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab03

```

Figure 0-1 OUTPUT

count_numbers()

1. for each character in character stream
2. if regex_match \+?{digit}+
3. dposcnt++
4. if regex_match \+?{digit}+\.{digit}*
5. fposcnt++
6. if regex_match -{digit}+
7. dnegcnt++
8. if regex_match -{digit}+\.{digit}*

9. fnegcnt++
10.display dposcnt, fposcnt, dnegcnt, fnegcnt

6. Conclusions

NOTE:

A Real Number can be regex matched by

$-?([0-9]+) | ([0-9]*\.[0-9]+) ([eE] [-+]? [0-9]+)?$

where \. denotes a literal period.

Context Sensitivity. LEX provides some support for contextual grammatical rules.

- If ^ is the first character in an expression, then this expression will only be matched at the beginning of a line.
- If \$ is the last character in an expression, then this expression will only be matched at the end of a line.
- If r and s are two LEX regular expressions then r/s is another LEX regular expression.
- It matches r if and only if it is followed by an s.
- It is called a trailing context.
- After use in this context, s is then returned to the input before the action is executed. So the action only sees the text matched by r
- Left context is handled by means of start conditions which we will talk about later.

7. Comments

a. Limitations of Experiments

The experiment does not define the notations to be used for fractional numbers and hence few assumptions were made. Fractions like .10, -.20 are not allowed for this experiment.

b. Limitations of Results

The solution program written for this experiment does not accommodate for that fact that fractional number can also be represented in the exponent form, i.e. $2.5e-20$ to represent 2.5×10^{-20} . In simple words scientific notation is not considered in this program. Refer to the Note in Conclusion for a solution regex for matching real numbers.

c. Learning happened

We learnt how to write regex expressions to match numbers and count the positive and negative numbers from a given stream of characters.

d. Recommendations

The experiment should define the decimal and fractional number notations to be used.

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

Laboratory 4

Title of the Laboratory Exercise: Program to count the number of characters, words, spaces, end of lines in a given input file.

1. Introduction and Purpose of Experiment

2. Aim and Objectives

Aim

- To write a program to

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

lab04.1

```
%{  
  
#include <iostream>  
  
extern int charcnt, wordcnt, spacecnt, eolcnt ;  
  
%}  
  
%option noyywrap  
  
white      [ \n\t]+
```

```

digit      [0-9]
integer    [digit]+
exponent   [eE] [+]?{integer}
letter     [a-zA-Z]
eol        [\n]

%x         WORD

%%

[\n]       { eolcnt++; }
[ \t]      { spacecnt++; }
[a-zA-Z]   { BEGIN(WORD); charcnt++; }
<WORD>[a-zA-Z] { charcnt++; }
<WORD>[\n]  { BEGIN(INITIAL); wordcnt++; eolcnt++; charcnt++; }
<WORD>[ \t] { BEGIN(INITIAL); wordcnt++; spacecnt++; charcnt++; }
<WORD>[^a-zA-Z] { BEGIN(INITIAL); wordcnt++; charcnt++; }
.          { charcnt++; }

```

main.cpp

```

#include <stdio.h>

#include <iostream>

extern int yylex();
extern FILE* yyin;

int charcnt = 0, wordcnt = 0, spacecnt = 0, eolcnt = 0;

auto main(int argc, char* argv[]) -> int {
    FILE* file;

    if (argc > 1) {
        file = fopen(argv[1], "r");
        yyin = file;
    }

    if (yylex() == 0)
        std::cout << "parsed successfully" << '\n';
    else
        std::cerr << "error parsing" << '\n';

    std::cout << "character count = " << charcnt << '\n'
              << "word count = " << wordcnt << '\n'
              << "space count = " << spacecnt << '\n'
              << "EOL count = " << eolcnt << '\n';

    return 0;
}

```

```
}
```

Makefile

```
# name of the files and the program
NAME = lab04

# compiler setup
CC = g++
LEX = flex

all: ${NAME}

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

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

clean:
    rm -f lex.yy.c ${NAME}

run:
    @./${NAME}
```

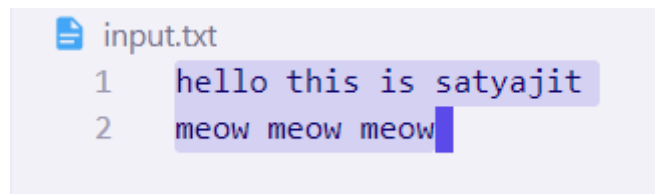
Algorithm:

count_context()

1. for each lexeme
2. if regex_match [\n]
3. eolcnt++
4. if regex_match [\t]
5. spacecnt++
6. if regex_match [a-zA-Z]
7. state = WORD
8. charcnt++
9. if state == WORD and regex_match [a-zA-Z]
10. charcnt++
11. if state == WORD and regex_match [\n]
12. state = INITIAL
13. wordcnt++; eolcnt++; charcnt++;
14. if state == WORD and regex_match [\t]
15. state = INITIAL
16. wordcnt++; spacecnt++; charcnt++;

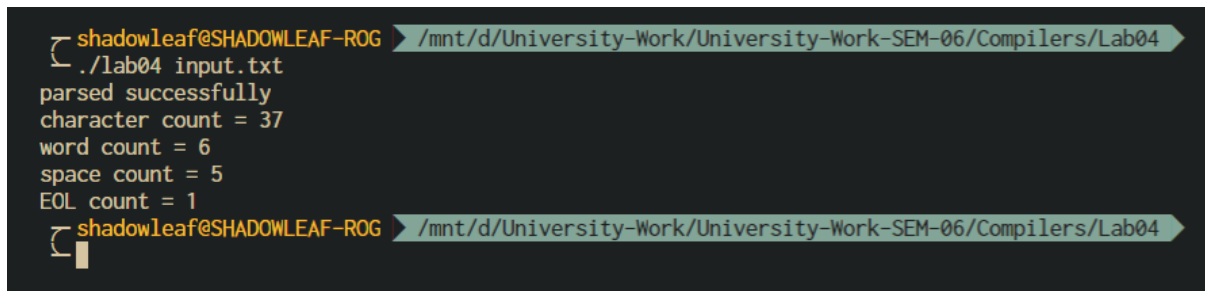
```
17.    if state == WORD and regex_match [^a-zA-Z]
18.        state = INITIAL
19.        wordcnt++; charcnt++;
20.    else
21.        charcnt++
22. display wordcnt, spacecnt, eolcnt, charcnt
```

5. Analysis and Discussions



```
input.txt
1  hello this is satyajit
2  meow meow meow
```

Figure 0-1 INPUT file



```
shadowleaf@SHADOWLEAF-ROG /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab04
./lab04 input.txt
parsed successfully
character count = 37
word count = 6
space count = 5
EOL count = 1
shadowleaf@SHADOWLEAF-ROG /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab04
```

Figure 0-2 OUTPUT

BEGIN

The action:

BEGIN newstate;

switches the state (start condition) to newstate. If the string newstate has not been declared previously as a start condition in the Definitions section, the results are unspecified. The initial state is indicated by the digit '0' or the token INITIAL.

ECHO

Write the contents of the string yytext on the output.

6. Conclusions

Flex provides a mechanism for conditionally activating rules. Any rule whose pattern is prefixed with '<sc>' will only be active when the scanner is in the start condition named sc. For example,

```
<STRING>[^"]*      { /* eat up the string body ... */  
    ...  
}
```

will be active only when the scanner is in the STRING start condition, and

```
<INITIAL,STRING,QUOTE>\.      { /* handle an escape ... */  
    ...  
}
```

will be active only when the current start condition is either INITIAL, STRING, or QUOTE.

Any line beginning with a '%' followed by a word beginning with either 'x' or 'X' shall define a set of exclusive start conditions. When the generated scanner is in a %s state, patterns with no state specified shall be also active; in a %x state, such patterns shall not be active.

The rest of the line, after the first word, shall be considered to be one or more <blank>-separated names of start conditions. Start condition names shall be constructed in the same way as definition names. Start conditions can be used to restrict the matching of regular expressions to one or more states as described in Regular Expressions in lex.

7. Comments

a. Limitations of Experiments

Lex cannot be used to recognize nested structures such as parentheses. Nested structures are handled by incorporating a stack. Whenever we encounter a "(" we push it on the stack. When a ")" is encountered we match it with the top of the stack and pop the stack. However, lex only has states and transitions between states. Since it has no stack it is not well suited for parsing nested structures.

b. Limitations of Results

The lexical analyzer is not very efficient and its maintenance can be complicated. The number of states, their start and end conditions will increase as we add more context information to the transitions.

c. Learning happened

We learnt how to use Lex states, and use this to count the number of words, lines, spaces and characters.

d. Recommendations

Try not to complicate the states used in lex, and define the problem statement more clearly to make it easy to write the lex rules.

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

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

lab06.1

```
%{  
    /* 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
                        }
        ;
```

```

        #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}

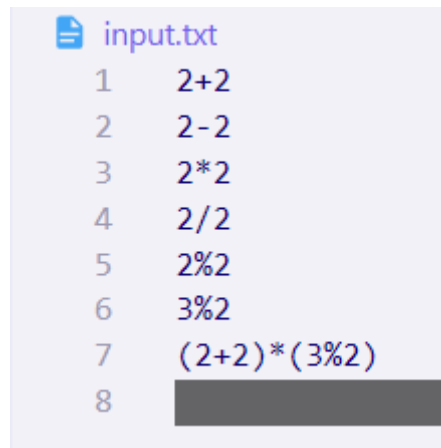
${NAME}.tab.c ${NAME}.tab.h: ${NAME}.y
    ${YACC} ${NAME}.y

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

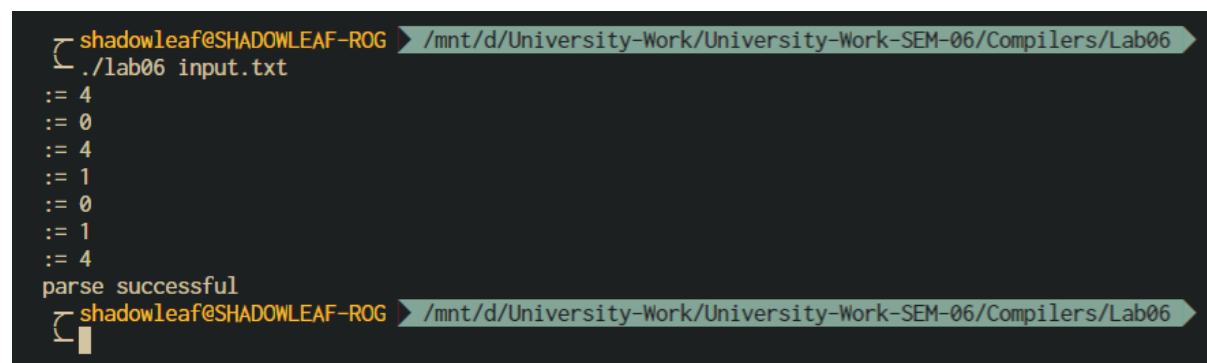
clean:
    rm -f lex.yy.c lex.yy.o ${NAME} ${NAME}.tab.c ${NAME}.tab.h
    rm -f ${NAME}.tab.o
```

5. Analysis and Discussions

A screenshot of a text editor showing a file named 'input.txt'. The file contains eight lines of arithmetic expressions, each preceded by a line number from 1 to 8. The expressions are: 1. 2+2, 2. 2-2, 3. 2*2, 4. 2/2, 5. 2%2, 6. 3%2, 7. (2+2)*(3%2), and 8. A dark grey rectangular box is positioned at the end of line 8.

```
input.txt
1    2+2
2    2-2
3    2*2
4    2/2
5    2%2
6    3%2
7    (2+2)*(3%2)
8    
```

Figure 0-1 INPUT file

A screenshot of a terminal window showing the execution of a program. The prompt is 'shadowleaf@SHADOWLEAF-ROG' and the directory is '/mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab06'. The command './lab06 input.txt' is entered. The output consists of eight lines of values: 4, 0, 4, 1, 0, 1, 4, and 'parse successful'.

```
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

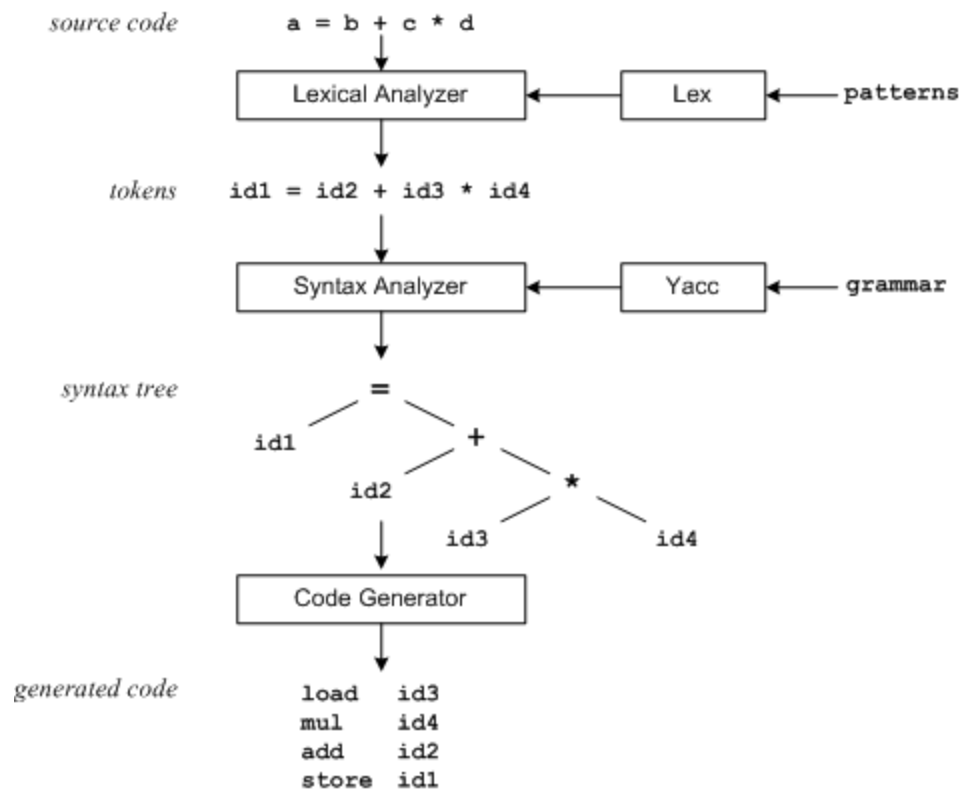


Figure 0-3 Overview of the program

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:

$E : E \text{ op1 } E ;$

$E : E \text{ op2 } E ;$

b. Limitations of Results

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

$S : X B c \mid Y B d ;$

$X : A ;$

$Y : A ;$

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	

Laboratory 7

Title of the Laboratory Exercise: Program to recognize a valid variable, which starts with a letter, followed by any number of letters or digits.

1. Introduction and Purpose of Experiment

Students learn to

2. Aim and Objectives

Aim

- To write a program

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

tokens.l

```
%{  
#include "parser.tab.h"  
%}  
  
%option noyywrap  
  
%%  
  
[0-9]+      { return DIGIT; }  
[a-zA-Z]+  { return LETTER; }  
[ \t]      { ; }
```



```
\n      { return 0; }
.      { return yytext[0]; }

%%
```

parser.y

```
%{

%}

%token DIGIT LETTER

%%

start  : LETTER s

s      : LETTER s
      | DIGIT s
      | /* empty */
      ;

%%
```

main.c

```
#include <stdio.h>
#include <stdlib.h>

#include "parser.tab.h"

int main(int argc, char* argv[]) {
    printf("Enter String : ");
    yyparse();
    printf("valid identifier\n");
    return 0;
}

void yyerror(char* s) {
    printf("invalid: %s\n", s);
    exit(0);
}
```

Makefile

```
all: ident

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

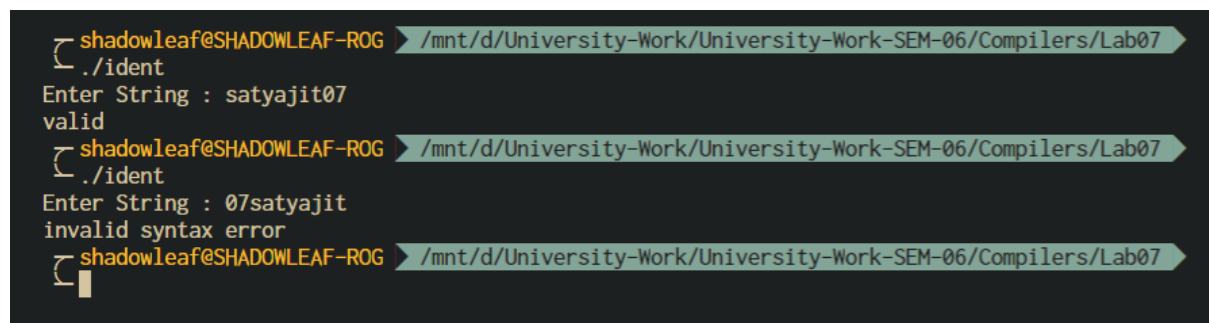
lex.yy.c: tokens.l parser.tab.h
```

```
flex tokens.l

ident: main.c lex.yy.c parser.tab.c parser.tab.h
      gcc -o ident main.c parser.tab.c lex.yy.c

clean:
      rm ident parser.tab.c parser.tab.h lex.yy.c
```

5. Analysis and Discussions



```
shadowleaf@SHADOWLEAF-ROG > /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab07
└─ ./ident
Enter String : satyajit07
valid
shadowleaf@SHADOWLEAF-ROG > /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab07
└─ ./ident
Enter String : 07satyajit
invalid syntax error
shadowleaf@SHADOWLEAF-ROG > /mnt/d/University-Work/University-Work-SEM-06/Compilers/Lab07
└─
```

Figure 0-1 OUTPUT

6. Conclusions

As Bison reads tokens, it pushes them onto a stack along with their semantic values. The stack is called the parser stack. Pushing a token is traditionally called shifting.

The function `yyparse` is implemented using a finite-state machine. The values pushed on the parser stack are not simply token type codes; they represent the entire sequence of terminal and nonterminal symbols at or near the top of the stack. The current state collects all the information about previous input which is relevant to deciding what to do next.

Each time a lookahead token is read, the current parser state together with the type of lookahead token are looked up in a table. This table entry can say, "Shift the lookahead token." In this case, it also specifies the new parser state, which is pushed onto the top of the parser stack. Or it can say, "Reduce using rule number n." This means that a certain number of tokens or groupings are taken off the top of the stack, and replaced by one grouping. In other words, that number of states are popped from the stack, and one new state is pushed.

The basic way to declare a token type name (terminal symbol) is as follows:

```
%token name
```

Bison will convert this into a definition in the parser, so that the function `yylex` (if it is in this file) can use the name `name` to stand for this token type's code.

7. Comments

a. Limitations of Experiments

Yacc is inflexible in some ways:

- good error handling is hard (basically, its algorithm is only defined to parse a correct string correctly, otherwise, all bets are off; this is one of the reasons that GCC moved to a hand-written parser)
- context-dependency is hard to express, whereas with a hand-written recursive descent parser you can simply add a parameter to the functions

Furthermore, `lex/yacc` object code is often bigger than a hand-written recursive descent parser (source code tends to be the other way round).

b. Limitations of Results

The solution program written here throws a syntax error when the grammar is not matched, the function of a lexer should be to point the line number and column number of the syntax error, which this program does not, a major limitation of the result obtained.

c. Learning happened

We learnt the basics of YACC and LEX to convert an FSM grammar rules to program that can validate strings which start with letter followed by any number of letters or numbers.

d. Recommendations

Add the functionality to display the error along with line and column number for better debugging.

Component	Max Marks	Marks Obtained
Viva	6	
Results	7	

NAME: SATYAJIT GHANA

REG NO: 17ETCS002159

Documentation	7	
Total	20	

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

```
#include <stdio.h>

#include "calc.tab.h"

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

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

    return 0;
}

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

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, Division, 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
./calc

```

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, Division, 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
./calc

```

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, Division, 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
./calc

```

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:

$$E : E \text{ op1 } E ;$$
$$E : E \text{ op2 } E ;$$

- Suppose $E \text{ op1 } E$ 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	