Date:

Experiment – 1

Aim:

Design a Lexical analyzer for the given language. The lexical analyzer should ignore redundant spaces, tabs, and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value.

Description:

Lexical Analyzer Overview

A Lexical Analyzer (or Lexer) is the first phase of a compiler, responsible for converting a sequence of characters into tokens. It reads the source code, removes unnecessary spaces, tabs, new lines, and comments, and outputs meaningful tokens for further processing in the compilation pipeline.

Key Features of the Lexical Analyzer:

- Tokenization: Breaks the input into meaningful symbols like keywords, identifiers, operators, and literals.
- Whitespace & Comment Handling: Ignores redundant spaces, tabs, new lines, and comments.
- Identifier Length Restriction: Limits identifier length to a reasonable number of characters.

Key Definitions

- Token: The smallest meaningful unit of code (e.g., keywords, identifiers, literals).
- Lexeme: The actual character sequence in the source code that forms a token.
- Pattern: A rule that defines the structure of lexemes (e.g., [a-zA-Z_][a-zA-Z0-9_]* for identifiers).
- Whitespace Handling: Ignores spaces, tabs, and new lines that are not inside string literals.
- Comment Handling: Removes single-line (// comment) and multi-line (/* comment */) comments.

Programs:

Lexical Analyzer for C Source Code

Lexical Analysis.c

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int isKeyword(char *str) {
    char k[32][10] = {
        "auto", "break", "case", "char", "const",
"continue", "default", "do",
        "double", "else", "enum", "extern", "float", "for",
"goto", "if", "int", "long", "register",
        "return", "short", "signed", "sizeof", "static",
"struct", "switch", "typedef", "union",
        "unsigned", "void", "volatile", "while"
    };
    int i;
    for (i = 0; i < 32; i++)
        if (strcmp(k[i], str) == 0)
            return 1;
    return 0;
}

int isFunction(char *str) {
    if (strcmp(str, "main") == 0 || strcmp(str, "printf") ==
0)
        return 1;
    return 0;
}

int main() {
    int kc, lno = 1, sno = 0;
    char fn[20], c, buf[30];
    FILE *fp;

    printf("\nEnter the file name: ");
    scanf("%s", fn);

    printf("\n\nS.No         Token            Lexeme
           Line No");
    fp = fopen(fn, "r");
```

```
    if (fp == NULL) {
        printf("\nError opening file.");
        return 1;
    }

    while ((c = fgetc(fp)) != EOF) {
        if (isalpha(c)) {
            buf[kc = 0] = c;
            while (isalnum(c = fgetc(fp))) {
                buf[++kc] = c;
            }
            buf[++kc] = '\0';
            if (isKeyword(buf))
                printf("\n%4d        keyword           %20s
   %7d", ++sno, buf, lno);
            else if (isFunction(buf))
                printf("\n%4d        function          %20s
   %7d", ++sno, buf, lno);
            else
                printf("\n%4d       identifier         %20s
   %7d", ++sno, buf, lno);
        } else if (isdigit(c)) {
            buf[kc = 0] = c;
            while (isdigit(c = fgetc(fp)))
                buf[++kc] = c;
            buf[++kc] = '\0';
            printf("\n%4d         number           %20s
%7d", ++sno, buf, lno);
        }

        if (c == '(' || c == ')')
            printf("\n%4d       parenthesis         %6c
             %7d", ++sno, c, lno);
        else if (c == '{' || c == '}')
            printf("\n%4d         brace                %6c
             %7d", ++sno, c, lno);
        else if (c == '[' || c == ']')
            printf("\n%4d       array
index         %6c                  %7d", ++sno, c, lno);
        else if (c == ',' || c == ';')
            printf("\n%4d       punctuation         %6c
             %7d", ++sno, c, lno);
        else if (c == '"') {
            kc = -1;
```

```
            while ((c = fgetc(fp)) != '"' && c != EOF)
                buf[++kc] = c;
            buf[++kc] = '\0';
            printf("\n%4d         string            %20s
 %7d", ++sno, buf, lno);
        } else if (c == ' ') {
            c = fgetc(fp);
        } else if (c == '\n') {
            ++lno;
        } else {
            printf("\n%4d         operator            %6c
             %7d", ++sno, c, lno);
        }
    }

    fclose(fp);
    return 0;
}
```

Test.c

```
#include <stdio.h>
int main() {
    int x = 10, y = 20;
    float z = 3.14;
    if (x < y) {
        printf("x is smaller\n");
    }
    return 0;
}
```

Output:

```
PS C:\Users\DELL\OneDrive\Desktop\22501A0533> gcc .\LexicalAnalysis.c
PS C:\Users\DELL\OneDrive\Desktop\22501A0533> .\a.exe

 Enter the file name: .\test.c


 S.No         Token             Lexeme                  Line No
    1          operator              #                        1
    2        identifier                   include             1
    3        identifier                     stdio             1
    4          operator              .                        1
    5        identifier                         h             1
    6          operator              >                        1
    7         keyword                         int             2
    8        identifier                       ain             2
    9        parenthesis             (                        2
   10        parenthesis             )                        2
   11         keyword                         int             3
   12          number                           0             3
   13        punctuation             ,                        3
   14          number                           0             3
   15        punctuation             ;                        3
   16         keyword                       float             4
   17          operator              .                        4
   18          number                          14             4
   19        punctuation             ;                        4
   20         keyword                          if             5
   21        identifier                         x             5
   22        parenthesis             )                        5
   23         function                     printf             6
   24        parenthesis             (                        6
   25          string                x is smaller\n            6
   26        parenthesis             )                        6
   27        punctuation             ;                        6
   28          brace                 }                        7
   29         keyword                      return             8
   30        punctuation             ;                        8
   31          brace                 }                        9
PS C:\Users\DELL\OneDrive\Desktop\22501A0533>
```

Date:

Experiment – 2

Aim:

(a) Implement the lexical analyzer using LEX program for the regular expression

RE's: a(a+b)*

(b) Implement the LEX program to implement

RE's: (a+b)*abb(a+b)*

Description:

A lexical analyzer (lexer) processes input strings based on given regular expressions (REs) and classifies them into valid and invalid tokens. Here, we implement LEX programs for two REs:

1. **a(a+b)*** – Strings must start with 'a' and can be followed by any combination of 'a' and 'b'. Ex: {a,ab,aa,aab,aabb,abba,…}
2. **(a+b)*abb(a+b)*** – Strings must contain 'abb' as a substring, surrounded by any combination of 'a' and 'b'. Ex: {abb,aabb,abbb,abba,babbb,…}

Steps to run:

1. Go to the directory, where the program is there
2. Execute the command: flex filename.l
3. Execute the command: gcc ./lex.yy.c
4. Execute the command: ./a.exe
5. Output is Generated

Program:

(a) RE: a(a+b)*

```
%{
    #include <stdio.h>
    int result = 0;
%}
pattern a[a|b]*[\n]
%%
{pattern} { printf("String is valid \n"); }
. { printf("String is not valid \n"); }
%%
int yywrap() {
    return 1;
}
int main()
{
    printf("Enter the String to Automata: ");
    yylex();
}
```

Output:

```
PS C:\Users\DELL\OneDrive\Desktop\22501A0533> flex ./calculate.l
PS C:\Users\DELL\OneDrive\Desktop\22501A0533> gcc ./lex.yy.c -o output
PS C:\Users\DELL\OneDrive\Desktop\22501A0533> ./output.exe
Enter the String to Automata: abbbababababaa
String is valid
b
String is not valid

PS C:\Users\DELL\OneDrive\Desktop\22501A0533>
```

(b) RE: (a+b)*abb(a+b)*

```
%{
    #include <stdio.h>
    int result = 0;
%}
pattern [a|b]*abb[a|b]*[\n]
%%
{pattern} { printf("String is valid \n"); }
. { printf("String is not valid \n"); }
%%
int yywrap() {
    return 1;
}
int main()
{
    printf("Enter the String to Automata: ");
    yylex();
}
```

Output:

```
PS C:\Users\DELL\OneDrive\Desktop\22501A0533> flex ./calculate.l
PS C:\Users\DELL\OneDrive\Desktop\22501A0533> gcc ./lex.yy.c -o output
PS C:\Users\DELL\OneDrive\Desktop\22501A0533> ./output.exe
Enter the String to Automata: abba
String is valid
b
String is not valid

PS C:\Users\DELL\OneDrive\Desktop\22501A0533>
```

(c) RE: (a+b)*a

```
%{
    #include <stdio.h>
    int result = 0;
%}
pattern [a|b]*a[\n]

%%
{pattern} { printf("String is valid \n"); }

. { printf("String is not valid \n"); }

%%
int yywrap() {
    return 1;
}
int main()
{
    printf("Enter the String to Automata: ");
    yylex();
}
```

Output:

```
PS C:\Users\DELL\OneDrive\Desktop\22501A0533> flex ./calculate.l
PS C:\Users\DELL\OneDrive\Desktop\22501A0533> gcc ./lex.yy.c -o output
PS C:\Users\DELL\OneDrive\Desktop\22501A0533> ./output.exe
Enter the String to Automata: babbabab
String is not valid
String is not valid
String is not valid
String is not valid
String is not valid
String is not valid
String is not valid
String is not valid

abbba
String is valid
PS C:\Users\DELL\OneDrive\Desktop\22501A0533>
```

(d) RE: a(a+b)*b

```
%{
    #include <stdio.h>
    int result = 0;
%}
pattern a[a|b]*b[\n]
%%
{pattern} { printf("String is valid \n"); }
. { printf("String is not valid \n"); }
%%
int yywrap() {
    return 1;
}
int main()
{
    printf("Enter the String to Automata: ");
    yylex();
}
```

Output:

```
PS C:\Users\DELL\OneDrive\Desktop\22501A0533> flex ./calculate.l
PS C:\Users\DELL\OneDrive\Desktop\22501A0533> gcc ./lex.yy.c -o output
PS C:\Users\DELL\OneDrive\Desktop\22501A0533> ./output.exe
Enter the String to Automata: abababababab
String is valid
ab
String is valid
bba
String is not valid
String is not valid
String is not valid

PS C:\Users\DELL\OneDrive\Desktop\22501A0533>
```

Date:

Experiment – 3

Aim:

(a) Implement the lexical analyzer using JLEX, FLEX or LEX or other lexical analyzer generating stools.

(b) Implement the lexical analyzer Program to count no of +ve and –ve integers using LEX

Description:

A lexical analyzer (lexer) processes input text and converts it into meaningful tokens using tools like **JLEX, FLEX, or LEX**. These tools generate lexers that scan input, recognize patterns using regular expressions, and classify them into tokens such as keywords, identifiers, numbers, and operators. The lexer plays a crucial role in compilers and interpreters by breaking down source code into a structured format for further parsing and syntax analysis. The implementation involves defining token patterns in a .l file (for LEX/FLEX) or .lex (for JLEX), running the lexer generator to produce C or Java code, compiling it, and executing the lexer to analyze input.

A LEX program can be implemented to count the number of positive and negative integers in an input stream. It identifies numbers using regular expressions and increments counters based on whether a number is positive or negative. The program follows these steps:

- **Read input**: The lexer scans each token from user input or a file.
- **Recognize numbers**: A pattern is defined to detect positive integers ([1-9][0-9]*) and negative integers (- [1-9][0-9]*).
- **Count occurrences**: A counter is maintained for each category.
- **Display results**: After processing all input, the program prints the count of positive and negative numbers.
  This implementation is useful in data processing applications where classification of numerical data is needed.

Program:

(a) Lexical Analyzer Implementation using JLEX, FLEX, or LEX

Lexical Analyzer.l

```
%{

#include <stdio.h>

#include <stdlib.h>


char *word[] = {"keyword", "identifier", "operator",
"preprocessor", "comment", "invalid literal", "reserved",
"number", "string"};

void display(int);

%}


keyword
"int"|"char"|"short"|"void"|"long"|"if"|"else"|"case"|"for"|"d
o"|"while"|"break"|"auto"|"static"|"const"|"enum"|"struct"

reserved
"main"|"FILE"|"printf"|"scanf"|"puts"|"putc"|"getc"|"pow"

comments  "//".*|"/\\*".*"\\*/"

operator  "."|"{"|"}"|"("|")"|"["|"]"|"->"|"+"|"-
"|"*"|"/"|"|"|"="|"+="|"-
="|"*="|"/="|"%="|"&&"|"||"|"!"|"~"|";"

preprocessor  "#".*

string    "\"".*"\""

identifier  [a-zA-Z_][a-zA-Z0-9_]*

number    [0-9]+(\.[0-9]+)?

%%


{comments}   { display(4); }

{preprocessor}   { display(3); }

{reserved}   { display(6); }
```

```
{keyword}    { display(0); }

{operator}   { display(2); }

{string}     { display(8); }

{identifier} { display(1); }

{number}     { display(7); }

[\n\t' ']    {};  // ignore whitespace characters

.            { display(5); }  // invalid literal for anything
else

%%


void display(int n) {

    printf("\n%s --> %s\n", yytext, word[n]);

}


int yywrap() {

    return 1;

}


int main(int argc, char **argv) {

    if (argc > 1) {

        yyin = fopen(argv[1], "r");

        if (!yyin) {

            printf("Could not open %s \n", argv[1]);

            exit(0);

        }

    }

    yylex();

    return 0;

}
```

U.c

```
#include<stdio.h>
int main()
{
    printf("Hello World!!!");
    return 0;
}
```

Output:

```
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533> gcc .\lex.yy.c
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533> .\a.exe .\U.c

#include<stdio.h> --> preprocessor

int --> keyword

main --> reserved

( --> operator

) --> operator

{ --> operator

printf --> reserved

( --> operator

"Hello World!!!" --> string

) --> operator

; --> operator

return --> identifier

0 --> number

; --> operator

} --> operator
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533>
```

(b) Counting Positive and Negative Integers using LEX

Numbre Analyzer.c

```
%{
#include <stdio.h>
#include <stdlib.h>
// Counters for different types of numbers
int posint = 0, negint = 0, posfraction = 0, negfraction = 0;
%}
%%
[-][0-9]+(\.[0-9]+)?   { if (strchr(yytext, '.'))
negfraction++; else negint++; }  // Matches negative integers
and fractions
[+]?[0-9]+(\.[0-9]+)?  { if (strchr(yytext, '.'))
posfraction++; else posint++; }  // Matches positive integers
and fractions
[ \t\n]+               { /* Ignore whitespace */ }
.                      { /* Ignore other characters */ }
%%
int yywrap() {
    return 1; // Indicates end of file
}
int main(int argc, char *argv[]) {
    // Check for valid command-line arguments
    if (argc != 2) {
        printf("Usage: <./a.out> <sourcefile>\n");
        exit(0);
    }
    // Open the input file
    yyin = fopen(argv[1], "r");
    if (!yyin) {
        printf("Error: Could not open file %s\n", argv[1]);
```

```
        exit(1);
    }
    // Run the lexical analyzer
    yylex();
    // Print the results
    printf("Number of +ve integers = %d\n", posint);
    printf("Number of -ve integers = %d\n", negint);
    printf("Number of +ve fractions = %d\n", posfraction);
    printf("Number of -ve fractions = %d\n", negfraction);

    fclose(yyin); // Close the file
    return 0;
}
```

Input.txt

12

-45

0.67

-0.89

+100

-200.25

Output:

```
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533> flex ./Numberanalyzer.l
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533> gcc .\lex.yy.c
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533> .\a.exe .\input.txt
Number of +ve integers = 2
Number of -ve integers = 1
Number of +ve fractions = 1
Number of -ve fractions = 2
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533>
```

Date:

Experiment – 4

Aim:

(a) Implement the lexical analyzer Program to count the number of vowels and consonants in a given string.

(b) Implement the lexical analyzer Program to count the number of characters, words, spaces, end of lines in a given input file.

Description:

A lexical analyzer can be implemented using LEX to process input text and categorize characters based on predefined patterns. It scans input, identifies specific characters or words, and counts occurrences based on defined rules.

- Counting Vowels and Consonants: The program reads a given string and classifies each letter as a vowel (a, e, i, o, u in both uppercase and lowercase) or a consonant (any other alphabetic character). It maintains separate counters for vowels and consonants and prints the count at the end. This is useful in text analysis and linguistic processing.
- Counting Characters, Words, Spaces, and End-of-Lines: The program processes an input file and counts different textual elements.
  - Characters: Every non-whitespace symbol is counted.
  - Words: Identified by sequences of alphanumeric characters separated by spaces or newlines.
  - Spaces: Explicitly counted to track word separation.
  - End of Lines (EOLs): Counted to measure the number of lines in the file. This program is helpful in text editors, document analysis, and processing tools that require basic text statistics.

Program:

(a) Counting Vowels and Consonants using LEX

```
%{
#include <stdio.h>
#include <stdlib.h>
// Counters for vowels and consonants
int vowels = 0;
int cons = 0;
%}
%%
[aeiouAEIOU]    { vowels++; }   // Matches vowels
[a-zA-Z]        { cons++; }     // Matches consonants
[ \t\n]+        { /* Ignore whitespace, tabs, and newlines */ }
.               { /* Ignore other characters */ }
%%
int yywrap() {
    return 1; // Indicates end of input
}
int main() {
    printf("Enter the string (end input with Ctrl+D):\n");
    // Call the lexer
    yylex();
    // Print results
    printf("\nNumber of vowels = %d\n", vowels);
    printf("Number of consonants = %d\n", cons);

    return 0;
}
```

Output:

```
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533> flex .\checkvowel.l
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533> gcc .\lex.yy.c
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533> .\a.exe
Enter the string (end input with Ctrl+D):
aeioubcdf

Number of vowels = 5
Number of consonants = 4
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533>
```

(b) Counting Characters, Words, Spaces, and Lines using LEX

textAnalyzer.l

```
%{
#include<stdio.h>
int c=0, w=0, s=0, l=0;
%}

WORD [^ \t\n,\.:]+
EOL [\n]
BLANK [ ]

%%

{WORD}   {w++; c = c + yyleng;}
{BLANK}  {s++;}
{EOL}    {l++;}
.        {c++;}

%%

int yywrap()
{
   return 1;
```

```
}

int main(int argc, char *argv[])
{
   if(argc != 2)
   {
      printf("Usage: <./a.out> <sourcefile>\n");
      exit(0);
   }
   yyin = fopen(argv[1], "r");
   yylex();
   printf("No of characters = %d\nNo of words = %d\nNo of spaces = %d\nNo of lines = %d", c, w, s, l);
   return 0;
}
```

input.txt

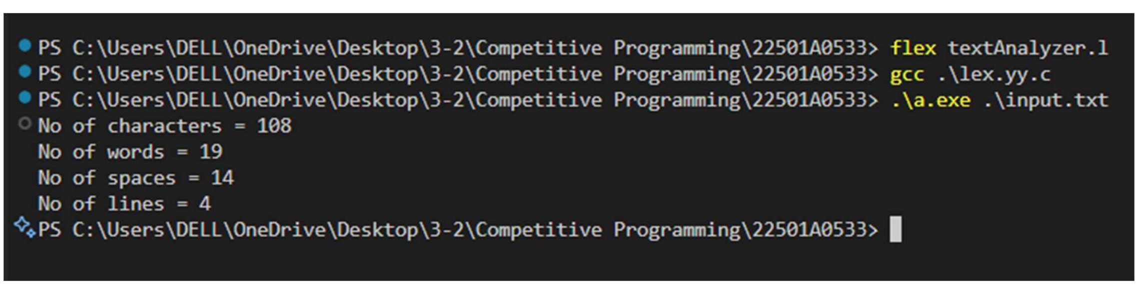
Prasad V Potluri Siddhartha Institute of Technology

III B.tech CSE Section-1 Students

Compiler Design Lab

Simple Lex programs

Output:



```
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533> flex textAnalyzer.l
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533> gcc .\lex.yy.c
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533> .\a.exe .\input.txt
No of characters = 108
No of words = 19
No of spaces = 14
No of lines = 4
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533>
```

Date:

Experiment – 5

Aim:

Implement a C program to calculate FIRST and FOLLOW sets of given Grammar

Description:

In **compiler design**, **FIRST** and **FOLLOW** sets are essential for **parsing** and **syntax analysis** in **context-free grammars (CFGs)**.

- **FIRST Set**: The set of terminals that appear at the **beginning** of any string derived from a non-terminal.
- **FOLLOW Set**: The set of terminals that can appear **immediately after** a non-terminal in some derivation.

This program takes a **user-defined grammar**, computes the **FIRST** and **FOLLOW** sets, and displays them. It helps in **LL(1) parsing table construction**, making it crucial for **syntax analysis** in **compilers**.

Program:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void followfirst(char, int, int);
void follow(char c);
void findfirst(char, int, int);

int count = 8, n = 0, m = 0;
char calc_first[10][100];
char calc_follow[10][100];
char production[10][10];
```

```
char f[10], first[10];
int k;
char ck;
int e;

int main(int argc, char **argv) {
    int jm = 0, km = 0;
    int i, choice;
    char c, ch;
    int kay;
    int ptr = -1;
    char done[count];

    // Initialize productions
    strcpy(production[0], "E=TR");
    strcpy(production[1], "R=+TR");
    strcpy(production[2], "R=#");
    strcpy(production[3], "T=FY");
    strcpy(production[4], "Y=*FY");
    strcpy(production[5], "Y=#");
    strcpy(production[6], "F=(E)");
    strcpy(production[7], "F=i");

    // Initialize calc_first and calc_follow
    for(k = 0; k < count; k++) {
        for(kay = 0; kay < 100; kay++) {
            calc_first[k][kay] = '!';
            calc_follow[k][kay] = '!';
        }
```

```
    }

    // Calculate FIRST sets
    int point1 = 0, point2, xxx;
    for(k = 0; k < count; k++) {
        c = production[k][0];
        point2 = 0;
        xxx = 0;

        // Skip if already calculated FIRST set
        for(kay = 0; kay <= ptr; kay++) {
            if(c == done[kay]) {
                xxx = 1;
                break;
            }
        }

        if (xxx == 1)
            continue;

        findfirst(c, 0, 0);
        ptr += 1;
        done[ptr] = c;

        // Print FIRST set
        printf("\nFirst(%c) = { ", c);
        calc_first[point1][point2++] = c;
        for(i = 0 + jm; i < n; i++) {
            int lark = 0, chk = 0;
```

```
            for(lark = 0; lark < point2; lark++) {
                if (first[i] == calc_first[point1][lark]) {
                    chk = 1;
                    break;
                }
            }
            if(chk == 0) {
                printf("%c, ", first[i]);
                calc_first[point1][point2++] = first[i];
            }
        }
        printf("}\n");
        jm = n;
        point1++;
    }

    printf("\n-----------------------------------------------
\n\n");

    // Calculate FOLLOW sets
    char donee[count];
    ptr = -1;
    point1 = 0;
    int land = 0;
    for(e = 0; e < count; e++) {
        ck = production[e][0];
        point2 = 0;
        xxx = 0;

        // Skip if already calculated FOLLOW set
```

```
for(kay = 0; kay <= ptr; kay++) {
    if(ck == donee[kay]) {
        xxx = 1;
        break;
    }
}

if (xxx == 1)
    continue;

land += 1;
follow(ck);
ptr += 1;
donee[ptr] = ck;

// Print FOLLOW set
printf("Follow(%c) = { ", ck);
calc_follow[point1][point2++] = ck;
for(i = 0 + km; i < m; i++) {
    int lark = 0, chk = 0;
    for(lark = 0; lark < point2; lark++) {
        if (f[i] == calc_follow[point1][lark]) {
            chk = 1;
            break;
        }
    }
    if(chk == 0) {
        printf("%c, ", f[i]);
        calc_follow[point1][point2++] = f[i];
```

```
            }
        }
        printf("}\n\n");
        km = m;
        point1++;
    }

    return 0;
}

void follow(char c) {
    int i, j;
    if(production[0][0] == c) {
        f[m++] = '$';
    }

    for(i = 0; i < 10; i++) {
        for(j = 2; j < 10; j++) {
            if(production[i][j] == c) {
                if(production[i][j+1] != '\0') {
                    followfirst(production[i][j+1], i, j+2);
                }
                if(production[i][j+1] == '\0' && c !=
production[i][0]) {
                    follow(production[i][0]);
                }
            }
        }
    }
}
```

```
void findfirst(char c, int q1, int q2) {
    int j;
    if (!(isupper(c))) {
        first[n++] = c;
    }

    for(j = 0; j < count; j++) {
        if(production[j][0] == c) {
            if(production[j][2] == '#') {
                if(production[q1][q2] == '\0') {
                    first[n++] = '#';
                }
                else if(production[q1][q2] != '\0' && (q1 != 0
|| q2 != 0)) {
                    findfirst(production[q1][q2], q1, (q2+1));
                }
                else {
                    first[n++] = '#';
                }
            }
            else if(!isupper(production[j][2])) {
                first[n++] = production[j][2];
            }
            else {
                findfirst(production[j][2], j, 3);
            }
        }
    }
}
```

```
void followfirst(char c, int c1, int c2) {
    int k;
    if(!(isupper(c))) {
        f[m++] = c;
    } else {
        int i = 0, j = 1;
        for(i = 0; i < count; i++) {
            if(calc_first[i][0] == c) {
                break;
            }
        }

        while(calc_first[i][j] != '!') {
            if(calc_first[i][j] != '#') {
                f[m++] = calc_first[i][j];
            } else {
                if(production[c1][c2] == '\0') {
                    follow(production[c1][0]);
                } else {
                    followfirst(production[c1][c2], c1, c2+1);
                }
            }
            j++;
        }
    }
}
```

Output:

```
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533\22501A0533> gcc ./firstandfollow.c
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533\22501A0533> .\a.exe

 First(E) = { (, i, }

 First(R) = { +, #, }

 First(T) = { (, i, }

 First(Y) = { *, #, }

 First(F) = { (, i, }

-----------------------------------------------

 Follow(E) = { $, ),  }

 Follow(R) = { $, ),  }

 Follow(T) = { +, $, ),  }

 Follow(Y) = { +, $, ),  }

 Follow(F) = { *, +, $, ),  }

PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533\22501A0533>
```

Date:

Experiment – 6

Aim:

Design Predictive parser for the given language.

Description:

A **Predictive Parser** is a type of **recursive descent parser** that **does not require backtracking**. It uses a **LOOKAHEAD symbol** to determine which production to apply. This is achieved by precomputing the **FIRST** and **FOLLOW** sets of the grammar.

The parser follows these steps:

1. **Eliminating Left Recursion** (if any exists).
2. **Left Factoring the Grammar** (to ensure it is in LL(1) form).
3. **Constructing the Parsing Table** using **FIRST** and **FOLLOW** sets.
4. **Parsing Input Strings** using a stack-based approach.

This parser is designed to process a given **context-free grammar (CFG)** and validate whether an input string belongs to the language.

Program:

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

char input[20];
int len, ln = 0, err = 0;

void E();
void E1();
void T();
void T1();
```

```
void F();
void match(char topChar);

void E() {
    T();
    E1();
}

void E1() {
    if (*input == '+') {
        match('+');
        T();
        E1();
    } else {
        return;
    }
}

void T() {
    F();
    T1();
}

void T1() {
    if (*input == '*') {
        match('*');
        F();
        T1();
    } else {
```

```
        return;
    }
}



void F() {
    if (*input == '(') {
        match('(');
        E();
        match(')');
    } else {
        match('i');
    }
}



void match(char topChar) {
    if (*input == topChar) {
        printf("\n%s popped %c", input, topChar);
        ln++;
        strcpy(input, &input[1]); // Pops matched input symbol
from input
    } else {
        printf("\nError: '%c' was expected but not found.",
topChar);
        err++;
    }
}
```

```c
int main() {
    printf("Enter the Input: ");
    scanf("%s", input); // Using scanf instead of gets for safety

    len = strlen(input);
    input[len] = '$';  // Append `$` to mark the end of input
    input[len + 1] = '\0';

    E();

    if (err == 0 && ln == len) {
        printf("\n\nString parsed successfully!!!\n");
    } else {
        printf("\n\nString is not parsed successfully.\nErrors occurred or input contains invalid characters.\n\n");
    }

    return 0;
}
```

Output:

```
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533\22501A0533> gcc .\predictiveparser.c
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533\22501A0533> .\a.exe
Enter the Input: i+i*i

i+i*i$ popped i
+i*i$ popped +
i*i$ popped i
*i$ popped *
i$ popped i

String parsed successfully!!!
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533\22501A0533>
```

Date:

Experiment – 7

Aim:

To implement Shift Reduce Parsing algorithm.

Description:

The **Shift-Reduce Parsing Algorithm** is a **bottom-up parsing technique** that **reduces** a given input string to the **start symbol** using a **stack-based approach**. It follows these steps:

1. **Shift**: Move symbols from the input to the stack one by one.
2. **Reduce**: Replace a sequence of symbols in the stack with a non-terminal based on the grammar rules.
3. **Repeat**: Continue shifting and reducing until the stack contains only the start symbol (E) and the input is empty.
4. **Accept or Reject**: If the stack has only E and the input is empty, the string is **accepted**; otherwise, it is **rejected**.

The provided C program implements **Shift-Reduce Parsing** for an **expression-based grammar** (E -> E + E | E * E | E / E | a | b). It prints the **stack contents, remaining input, and actions taken at each step**, showing how the parsing progresses.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char ip_sym[15], stack[15];
int ip_ptr = 0, st_ptr = 0, len, i;
char temp[2], temp2[2];
char act[15];
void check();
void main()
{
    printf("\n\t\t SHIFT REDUCE PARSER\n");
    printf("\n GRAMMER\n");
    printf("\n E->E+E\n E->E/E");
    printf("\n E->E*E\n E->a/b");
    printf("\n enter the input string:\t");
    gets(ip_sym);
    printf("\n\t stack implementation table");
```

```
    printf("\n stack \t\t input symbol\t\t action");
    printf("\n________\t\t____________\t\t____________\n");
    printf("\n $\t\t%s$\t\t\t--", ip_sym);
    strcpy(act, "shift ");
    temp[0] = ip_sym[ip_ptr];
    temp[1] = '\0';
    strcat(act, temp);
    len = strlen(ip_sym);
    for (i = 0; i <= len - 1; i++)
    {
        stack[st_ptr] = ip_sym[ip_ptr];
        stack[st_ptr + 1] = '\0';
        ip_sym[ip_ptr] = ' ';
        ip_ptr++;
        printf("\n $%s\t\t%s$\t\t\t%s", stack, ip_sym, act);
        strcpy(act, "shift ");
        temp[0] = ip_sym[ip_ptr];
        temp[1] = '\0';
        strcat(act, temp);
        check();
        st_ptr++;
    }
    st_ptr++;
    check();
}
void check()
{
    int flag = 0;
    temp2[0] = stack[st_ptr];
    temp2[1] = '\0';
    if ((!strcmp(temp2, "a")) || (!strcmp(temp2, "b")))
    {
        stack[st_ptr] = 'E';
        if (!strcmp(temp2, "a"))
            printf("\n $%s\t\t%s$\t\t\tE->a", stack, ip_sym);
        else
            printf("\n $%s\t\t%s$\t\t\tE->b", stack, ip_sym);
        flag = 1;
    }
    if ((!strcmp(temp2, "+")) || (strcmp(temp2, "*")) || (!strcmp(temp2,
"/")))
    {
        flag = 1;
    }
    if ((!strcmp(stack, "E+E")) || (!strcmp(stack, "E\E")) || (!strcmp(stack,
"E*E")))
    {
        strcpy(stack, "E");
```

```
        st_ptr = 0;
        if (!strcmp(stack, "E+E"))
            printf("\n $%s\t\t%s$\t\t\tE->E+E", stack, ip_sym);
        else if (!strcmp(stack, "E\E"))
            printf("\n $%s\t\t%s$\t\t\tE->E\E", stack, ip_sym);
        else if (!strcmp(stack, "E*E"))
            printf("\n $%s\t\t%s$\t\t\tE->E*E", stack, ip_sym);
        else
            printf("\n $%s\t\t%s$\t\t\tE->E+E", stack, ip_sym);
        flag = 1;
    }
    if (!strcmp(stack, "E") && ip_ptr == len)
    {
        printf("\n $%s\t\t%s$\t\t\tACCEPT", stack, ip_sym);
        exit(0);
    }
    if (flag == 0)
    {
        printf("\n%s\t\t\t%s\t\t reject", stack, ip_sym);
        exit(0);
    }
    return;
}
```

Output:

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   POSTMAN CONSOLE   COMMENTS

PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533> ./srparser.exe

                SHIFT REDUCE PARSER

 GRAMMER

 E->E+E
 E->E/E
 E->E*E
 E->a/b
 enter the input string:       a+b

        stack implementation table
 stack          input symbol          action
 --------              -----------          -----------

 $              a+b$                  --
 $a              +b$                  shifta
 $E              +b$                  E->a
 $E+              b$                  shift+
 $E+b              $                  shiftb
 $E+E              $                  E->b
 $E                $                  E->E+E
 $E                $                  ACCEPT
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533>
```

Experiment – 8

Aim:

Design LALR bottom-up parser for the given language. (Implementation of calculator using YACC)

Description:

The given code implements an **LALR (Look-Ahead LR) Bottom-Up Parser** using **YACC (Yet Another Compiler Compiler)** and **Lex (Lexical Analyzer Generator)** to evaluate **arithmetic expressions**. This parser follows **shift-reduce parsing** to evaluate mathematical operations with correct precedence and associativity.

How It Works

1. **Lexical Analysis (Lex File - ical.lex)**
    - It identifies numbers and operators in the input.
    - Converts digits into tokens (NUMBER).
    - Ignores tabs and spaces.
2. **Syntax Analysis (YACC File - ical.y)**
    - Defines grammar rules for arithmetic expressions.
    - Implements operator precedence using %left.
    - Uses shift-reduce parsing to compute expressions step by step.
    - If the input follows the grammar, it is **valid**; otherwise, an error is thrown.
3. **Execution Flow**
    - The user enters an arithmetic expression.
    - Lex tokenizes the input.
    - YACC applies the rules to evaluate the expression.
    - The result is displayed if the input is valid.

Program:

ical.y

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

int flag = 0;
int yylex();
void yyerror(char *s);
%}

%token NUMBER

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

%%

ArithmeticExpression:
    E {
        printf("\nResult = %d\n", $1);
        return 0;
    }
    ;

E:
    E '+' E { $$ = $1 + $3; }
    | E '-' E { $$ = $1 - $3; }
    | E '*' E { $$ = $1 * $3; }
    | E '/' E { $$ = $1 / $3; }
```

```
    | E '%' E { $$ = $1 % $3; }
    | '(' E ')' { $$ = $2; }
    | NUMBER { $$ = $1; }
    ;

%%

// Driver Code
int main() {
    printf("\nEnter an arithmetic expression (Add, Sub, Mul, Div, Mod, 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\n\n");
    flag = 1;
}
```

ical.l:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "ical.tab.h"

extern int yylval;
%}

%%
```

```
[0-9]+ {
    yylval = atoi(yytext);
    return NUMBER;
}

[\t] ;

[\n] { return 0; }

. { return yytext[0]; }

%%

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

Output:

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   COMMENTS

PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533> bison -d ical.y
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533> flex ical.lex
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533> gcc -o parser ical.tab.c lex.yy.c
PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533> ./parser

Enter an arithmetic expression (Add, Sub, Mul, Div, Mod, Brackets):
3+3*8

Result = 27

Entered arithmetic expression is Valid

PS C:\Users\DELL\OneDrive\Desktop\3-2\Competitive Programming\22501A0533>
```

Date:

Experiment – 9

Aim:

Convert the BNF rules into YACC form and write code to generate abstract syntax tree.

Description:

Backus-Naur Form (BNF) is a notation for describing the syntax of languages. **YACC (Yet Another Compiler Compiler)** is a tool used to generate a parser based on these grammar rules.

A parser typically processes an input according to the grammar rules and builds an **Abstract Syntax Tree (AST)**, which represents the structure of the parsed input.

Steps Involved:

1. **Convert BNF to YACC format**
    - Rewrite the BNF grammar using **YACC syntax** (tokens, precedence, and rules).
2. **Generate an Abstract Syntax Tree (AST)**
    - Define a **structure for AST nodes** in C.
    - Modify the YACC actions to **create AST nodes** instead of just evaluating expressions.
3. **Use Lex (Flex) for Tokenization**
    - Define tokens in a **Lex file** to work with YACC.
4. **Implement AST Traversal**
    - Print the tree in **preorder/postorder** for verification.

Programs:

int.l

```
%{
#include"y.tab.h"
#include<stdio.h>
#include<string.h>
int LineNo=1;
%}

identifier [a-zA-Z][_a-zA-Z0-9]*
number [0-9]+|([0-9]*\.[0-9]+)

%%
main\(\) return MAIN;
if return IF;
else return ELSE;
while return WHILE;
int | char | float return TYPE;

{identifier} {strcpy(yylval.var,yytext); return VAR;}
{number} {strcpy(yylval.var,yytext); return NUM;}

< |> |>= |<= |== {strcpy(yylval.var,yytext); return RELOP;}

[ \t] ;
\n LineNo++;
. return yytext[0];

%%
```

int.y

```
%{
#include<string.h>
#include<stdio.h>

struct quad{
    char op[5];
    char arg1[10];
    char arg2[10];
    char result[10];
} QUAD[30];

struct stack{
    int items[100];
    int top;
} stk;

int Index=0, tIndex=0, StNo, Ind, tInd;
extern int LineNo;
%}

%union {
    char var[10];
}

%token <var> NUM VAR RELOP
%token MAIN IF ELSE WHILE TYPE
%type <var> EXPR ASSIGNMENT CONDITION IFST ELSEST WHILELOOP
```

```
%left '-' '+'
%left '*' '/'

%%

PROGRAM : MAIN BLOCK ;

BLOCK: '{' CODE '}' ;

CODE: BLOCK
    | STATEMENT CODE
    | STATEMENT
;

STATEMENT: DESCT ';'
    | ASSIGNMENT ';'
    | CONDST
    | WHILEST
;

DESCT: TYPE VARLIST ;

VARLIST: VAR ',' VARLIST
    | VAR
;

ASSIGNMENT: VAR '=' EXPR {
    strcpy(QUAD[Index].op, "=");
    strcpy(QUAD[Index].arg1, $3);
```

```
    strcpy(QUAD[Index].arg2, "");
    strcpy(QUAD[Index].result, $1);
    strcpy($$, QUAD[Index++].result);
}
;

EXPR: EXPR '+' EXPR { AddQuadruple("+", $1, $3, $$); }
    | EXPR '-' EXPR { AddQuadruple("-", $1, $3, $$); }
    | EXPR '*' EXPR { AddQuadruple("*", $1, $3, $$); }
    | EXPR '/' EXPR { AddQuadruple("/", $1, $3, $$); }
    | '-' EXPR { AddQuadruple("UMIN", $2, "", $$); }
    | '(' EXPR ')' { strcpy($$, $2); }
    | VAR
    | NUM
;

CONDST: IFST {
    Ind = pop();
    sprintf(QUAD[Ind].result, "%d", Index);
    Ind = pop();
    sprintf(QUAD[Ind].result, "%d", Index);
}
| IFST ELSEST
;

IFST: IF '(' CONDITION ')' {
    strcpy(QUAD[Index].op, "==");
    strcpy(QUAD[Index].arg1, $3);
    strcpy(QUAD[Index].arg2, "FALSE");
```

```
    strcpy(QUAD[Index].result, "-1");
    push(Index);
    Index++;
}
BLOCK {
    strcpy(QUAD[Index].op, "GOTO");
    strcpy(QUAD[Index].arg1, "");
    strcpy(QUAD[Index].arg2, "");
    strcpy(QUAD[Index].result, "-1");
    push(Index);
    Index++;
};

ELSEST: ELSE {
    tInd = pop();
    Ind = pop();
    push(tInd);
    sprintf(QUAD[Ind].result, "%d", Index);
}
BLOCK {
    Ind = pop();
    sprintf(QUAD[Ind].result, "%d", Index);
};

CONDITION: VAR RELOP VAR {
    AddQuadruple($2, $1, $3, $$);
    StNo = Index - 1;
}
| VAR
```

```
| NUM
;

WHILEST: WHILELOOP {
    Ind = pop();
    sprintf(QUAD[Ind].result, "%d", StNo);
    Ind = pop();
    sprintf(QUAD[Ind].result, "%d", Index);
}
;

WHILELOOP: WHILE '(' CONDITION ')' {
    strcpy(QUAD[Index].op, "==");
    strcpy(QUAD[Index].arg1, $3);
    strcpy(QUAD[Index].arg2, "FALSE");
    strcpy(QUAD[Index].result, "-1");
    push(Index);
    Index++;
}
BLOCK {
    strcpy(QUAD[Index].op, "GOTO");
    strcpy(QUAD[Index].arg1, "");
    strcpy(QUAD[Index].arg2, "");
    strcpy(QUAD[Index].result, "-1");
    push(Index);
    Index++;
}
;
```

```
%%

extern FILE *yyin;

int main(int argc, char *argv[]) {
    FILE *fp;
    int i;

    if (argc > 1) {
        fp = fopen(argv[1], "r");
        if (!fp) {
            printf("\n File not found");
            exit(0);
        }
        yyin = fp;
    }

    yyparse();

    printf("\n\n\t\t ----------------------------""\n\t\t Pos
Operator Arg1 Arg2 Result" "\n\t\t --------------------");

    for (i = 0; i < Index; i++) {
        printf("\n\t\t %d\t %s\t %s\t %s\t %s", i, QUAD[i].op,
QUAD[i].arg1, QUAD[i].arg2, QUAD[i].result);
    }

    printf("\n\t\t -----------------------");
    printf("\n\n");
```

```
    return 0;
}

void push(int data) {
    stk.top++;
    if (stk.top == 100) {
        printf("\n Stack overflow\n");
        exit(0);
    }
    stk.items[stk.top] = data;
}

int pop() {
    int data;
    if (stk.top == -1) {
        printf("\n Stack underflow\n");
        exit(0);
    }
    data = stk.items[stk.top--];
    return data;
}

void AddQuadruple(char op[5], char arg1[10], char arg2[10],
char result[10]) {
    strcpy(QUAD[Index].op, op);
    strcpy(QUAD[Index].arg1, arg1);
    strcpy(QUAD[Index].arg2, arg2);
    sprintf(QUAD[Index].result, "t%d", tIndex++);
    strcpy(result, QUAD[Index++].result);
}
```

```
yyerror() {
    printf("\n Error on line no:%d", LineNo);
}
```

test.c

```
main() {
    int a, b, c;
    if (a < b) {
        a = a + b;
    }
    while (a < b) {
        a = a + b;
    }
    if (a <= b) {
        c = a - b;
    }
    else {
        c = a + b;
    }
}
```

Output:

Pos	Operator	Arg1	Arg2	Result
0	<	a	b	t0
1	==	t0	FALSE	5
2	+	a	b	t1
3	==	t1		5
4	GOTO			
5	<	a	b	t2
6	==	t2	FALSE	10
7	+	a	b	t3
8	=	t3		a
9	GOTO			5
10	<=	a	b	t4
11	==	t4	FALSE	15
12	-	a	b	t5
13	=	t5		c
14	GOTO			17
15	+	a	b	t6
16	=	t6		c

Date:

Experiment – 10

Aim:

Implement a Machine Code for a given Intermediate Code.

Description:

In **compiler design**, the **Intermediate Code** (IC) is generated after syntax and semantic analysis. This IC is architecture-independent and must be **converted into Machine Code (MC)** to run on a real system.

Steps in Translation Process:

1. **Intermediate Code Representation:**
   - The compiler first generates an intermediate representation, such as **Three Address Code (TAC)**, **Quadruples**, or **Triples**.
2. **Instruction Selection:**
   - Convert each **IC operation** to a corresponding **assembly-level instruction**.
   - Optimize by selecting **efficient instructions** based on the target architecture.
3. **Register Allocation & Optimization:**
   - Assign **registers** efficiently to minimize memory usage.
   - Implement techniques like **Graph Coloring** for register allocation.
4. **Code Generation:**
   - Convert optimized **assembly instructions** into **binary machine code**.
   - Handle **instruction formats, opcodes, and addressing modes**.
5. **Final Assembly & Linking:**
   - Convert the assembly into **object code** and link it with required libraries.
   - Generate the **final executable binary** for execution.

Programs:

MachineEvaluator.c

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

int label[20];
int no = 0;

int check_label(int k);

int main() {
    FILE *fp1, *fp2;
    char fname[20], op[10], ch;
    char operand1[8], operand2[8], result[8];
    int i = 0, j = 0;

    printf("\nEnter filename of the intermediate code: ");
    scanf("%s", fname);

    fp1 = fopen(fname, "r");
    fp2 = fopen("target.txt", "w");

    if (fp1 == NULL || fp2 == NULL) {
        printf("\nError opening the file\n");
        exit(1);
    }

    while (fscanf(fp1, "%s", op) != EOF) {
```

```
        fprintf(fp2, "\n");


        i++;
        if (check_label(i))
            fprintf(fp2, "\nlabel#%d:", i);


        if (strcmp(op, "print") == 0) {
            fscanf(fp1, "%s", result);
            fprintf(fp2, "\n\tOUT %s", result);
        }
        else if (strcmp(op, "goto") == 0) {
            fscanf(fp1, "%s %s", operand1, operand2);
            fprintf(fp2, "\n\tJMP label#%s", operand2);
            label[no++] = atoi(operand2);
        }
        else if (strcmp(op, "[]=") == 0) {
            fscanf(fp1, "%s %s %s", operand1, operand2,
result);
            fprintf(fp2, "\n\tSTORE %s, %s[%s]", result,
operand1, operand2);
        }
        else if (strcmp(op, "uminus") == 0) {
            fscanf(fp1, "%s %s", operand1, result);
            fprintf(fp2, "\n\tLOAD -%s, R1", operand1);
            fprintf(fp2, "\n\tSTORE R1, %s", result);
        }
        else {
            switch (op[0]) {
                case '*':
```

```
                    fscanf(fp1, "%s %s %s", operand1,
operand2, result);
                    fprintf(fp2, "\n\tLOAD %s, R0", operand1);
                    fprintf(fp2, "\n\tLOAD %s, R1", operand2);
                    fprintf(fp2, "\n\tMUL R1, R0");
                    fprintf(fp2, "\n\tSTORE R0, %s", result);
                    break;
                case '+':
                    fscanf(fp1, "%s %s %s", operand1,
operand2, result);
                    fprintf(fp2, "\n\tLOAD %s, R0", operand1);
                    fprintf(fp2, "\n\tLOAD %s, R1", operand2);
                    fprintf(fp2, "\n\tADD R1, R0");
                    fprintf(fp2, "\n\tSTORE R0, %s", result);
                    break;
                case '-':
                    fscanf(fp1, "%s %s %s", operand1,
operand2, result);
                    fprintf(fp2, "\n\tLOAD %s, R0", operand1);
                    fprintf(fp2, "\n\tLOAD %s, R1", operand2);
                    fprintf(fp2, "\n\tSUB R1, R0");
                    fprintf(fp2, "\n\tSTORE R0, %s", result);
                    break;
                case '/':
                    fscanf(fp1, "%s %s %s", operand1,
operand2, result);
                    fprintf(fp2, "\n\tLOAD %s, R0", operand1);
                    fprintf(fp2, "\n\tLOAD %s, R1", operand2);
                    fprintf(fp2, "\n\tDIV R1, R0");
                    fprintf(fp2, "\n\tSTORE R0, %s", result);
                    break;
```

```
                case '%':
                    fscanf(fp1, "%s %s %s", operand1,
operand2, result);
                    fprintf(fp2, "\n\tLOAD %s, R0", operand1);
                    fprintf(fp2, "\n\tLOAD %s, R1", operand2);
                    fprintf(fp2, "\n\tMOD R1, R0");
                    fprintf(fp2, "\n\tSTORE R0, %s", result);
                    break;
                case '=':
                    fscanf(fp1, "%s %s", operand1, result);
                    fprintf(fp2, "\n\tSTORE %s, %s", operand1,
result);
                    break;
                case '>':
                    fscanf(fp1, "%s %s %s", operand1,
operand2, result);
                    fprintf(fp2, "\n\tLOAD %s, R0", operand1);
                    fprintf(fp2, "\n\tJGT R0, %s, label#%s",
operand2, result);
                    label[no++] = atoi(result);
                    break;
                case '<':
                    fscanf(fp1, "%s %s %s", operand1,
operand2, result);
                    fprintf(fp2, "\n\tLOAD %s, R0", operand1);
                    fprintf(fp2, "\n\tJLT R0, %s, label#%s",
operand2, result);
                    label[no++] = atoi(result);
                    break;
            }
        }
```

```
    }

    fclose(fp2);
    fclose(fp1);

    // Display generated target code
    fp2 = fopen("target.txt", "r");
    if (fp2 == NULL) {
        printf("Error opening target file\n");
        exit(1);
    }

    while ((ch = fgetc(fp2)) != EOF) {
        printf("%c", ch);
    }

    fclose(fp2);
    return 0;
}

int check_label(int k) {
    for (int i = 0; i < no; i++) {
        if (k == label[i])
            return 1;
    }
    return 0;
}
```

int.txt

```
= t1 2
[]= a 0 1
[]= a 1 2
[]= a 2 3
* t1 6 t2
+ a[2] t2 t3
- a[2] t1 t2
/ t3 t2 t2
uminus t2 t2
print t2
goto t2 t3
= t3 99
uminus 25 t2
* t2 t3 t3
uminus t1 t1
+ t1 t3 t4
print t4
```

Output:

```
PS D:\B-Tech\PVP\3rd Year\2nd Sem\01 Compiler Design\Lab\Github\Exp - 10 Implementing Machine Code> gcc .\MachineEvaluator.c -o output
PS D:\B-Tech\PVP\3rd Year\2nd Sem\01 Compiler Design\Lab\Github\Exp - 10 Implementing Machine Code> .\output.exe

Enter filename of the intermediate code: int.txt


        STORE t1, 2

        STORE 1, a[0]

        STORE 2, a[1]

        STORE 3, a[2]

        LOAD t1, R0
        LOAD 6, R1
        MUL R1, R0
        STORE R0, t2

        LOAD a[2], R0
        LOAD t2, R1
        ADD R1, R0
        STORE R0, t3

        LOAD a[2], R0
        LOAD t1, R1
        SUB R1, R0
        STORE R0, t2

        LOAD t3, R0
        LOAD t2, R1
        DIV R1, R0
        STORE R0, t2

        LOAD -t2, R1
        STORE R1, t2

        OUT t2

        JMP label#t3

        STORE t3, 99

        LOAD -25, R1
        STORE R1, t2

        LOAD t2, R0
        LOAD t3, R1
        MUL R1, R0
        STORE R0, t3

        LOAD -t1, R1
        STORE R1, t1

        LOAD t1, R0
        LOAD t3, R1
        ADD R1, R0
        STORE R0, t4
```