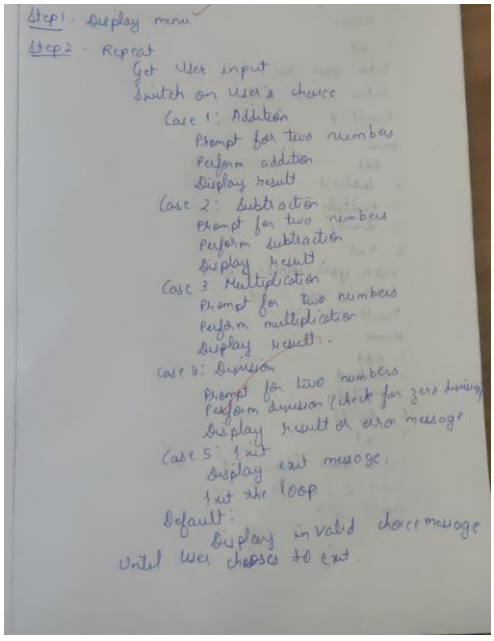
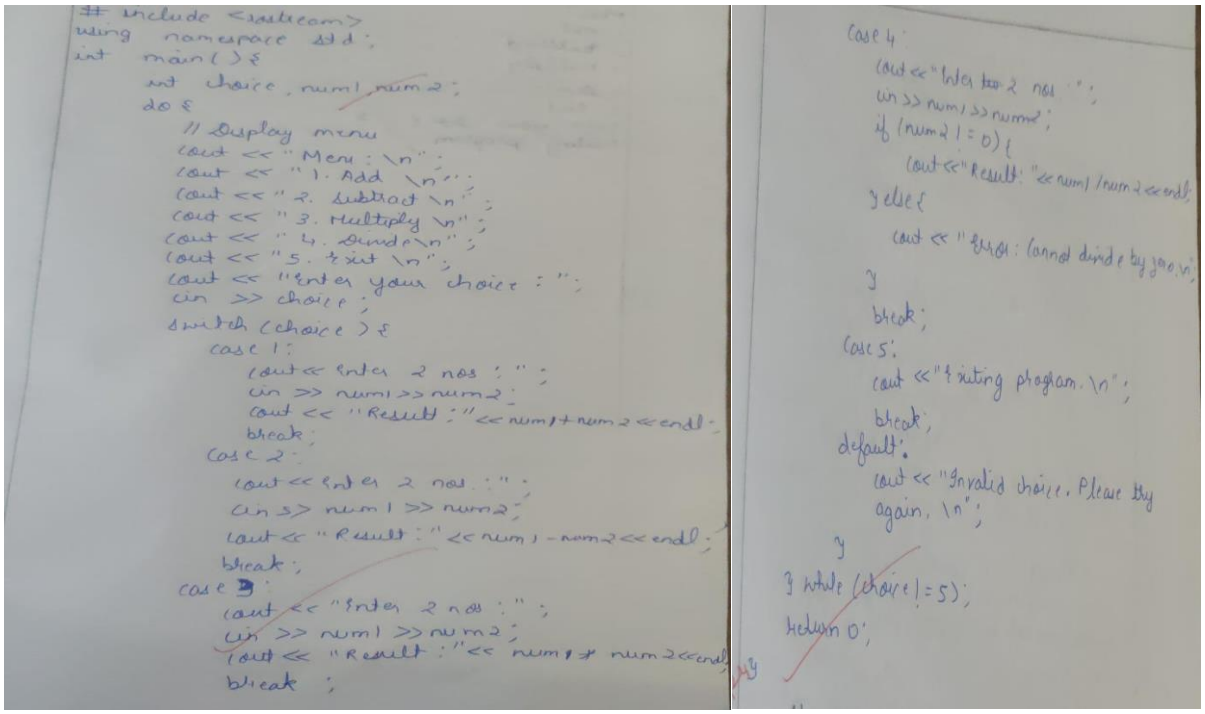
**CD LAB**

## EXP 1: a) arithmetic operations

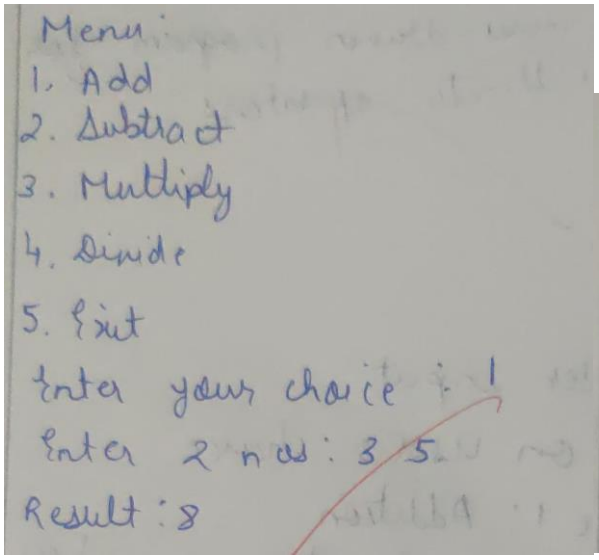
## Algo:

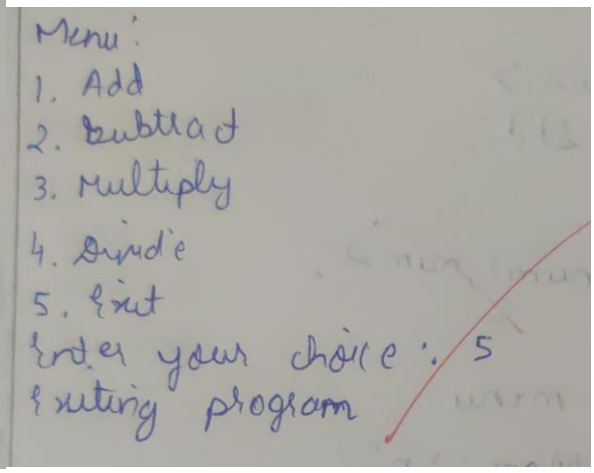



## Program:



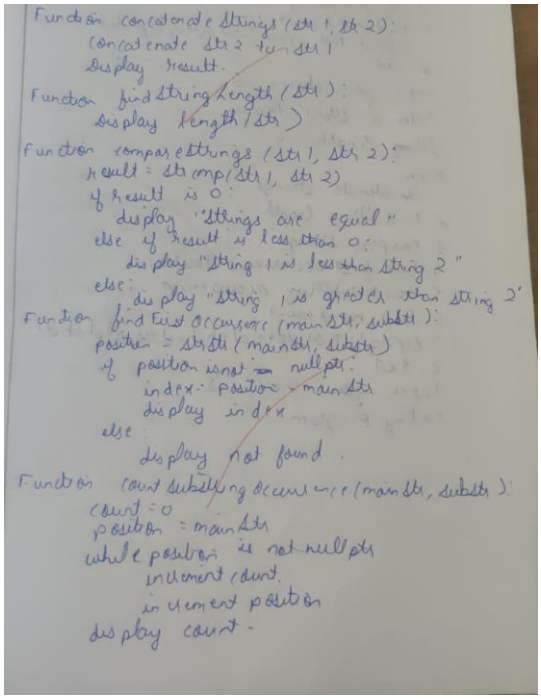
**Output:**

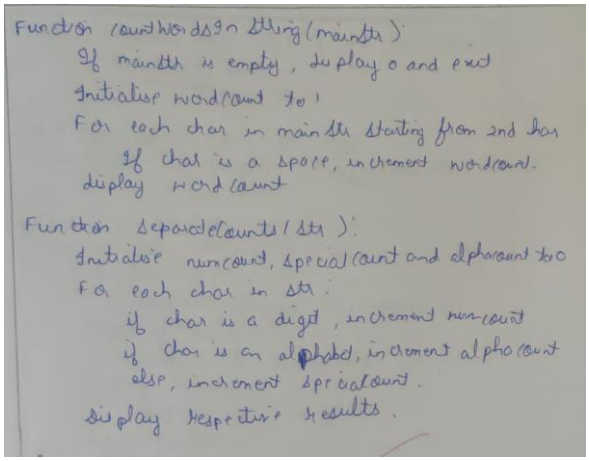


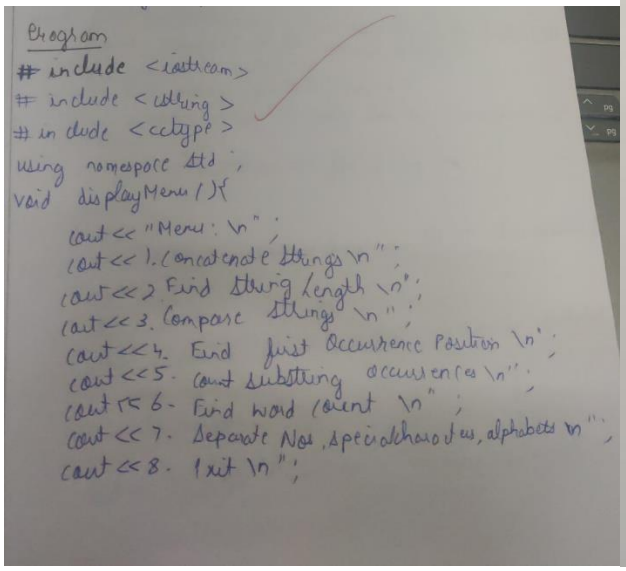

**b) string operations**

**Algo:**

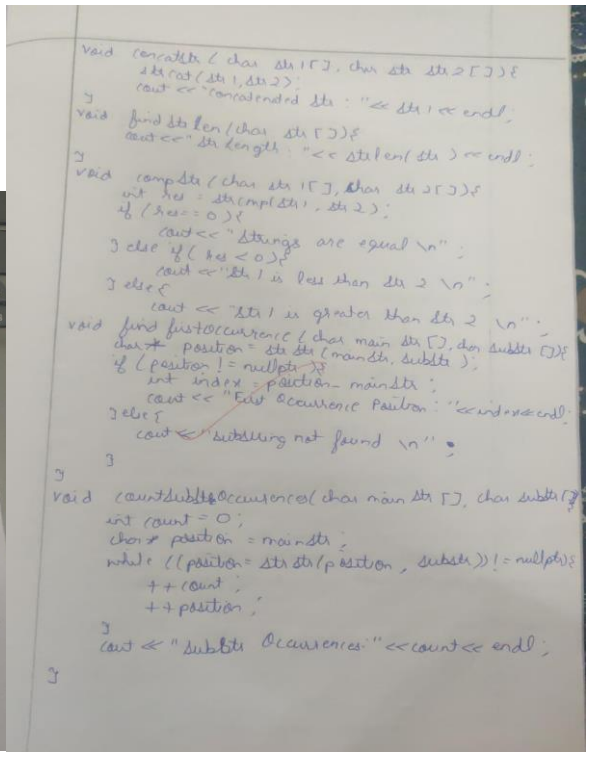


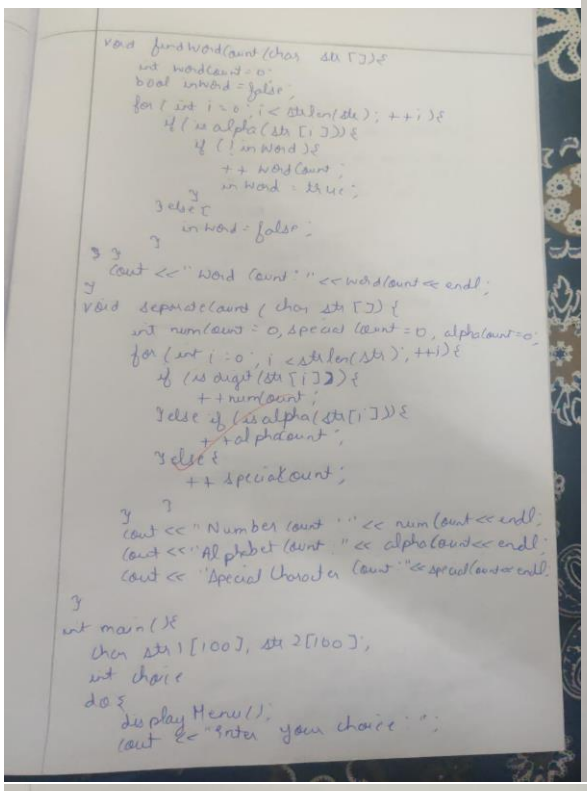
**Program:**



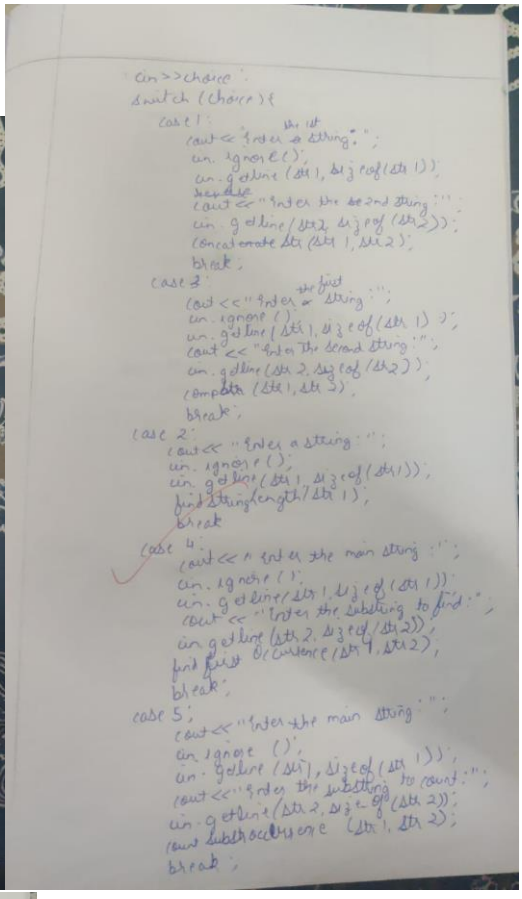
```
Program
#include <iostream>
#include <string>
#include <cctype>
using namespace std;
void displayMenu(){
    cout << "Menu: \n";
    cout << 1. Concatenate Strings \n";
    cout << 2. Find String Length \n";
    cout << 3. Compare Strings \n";
    cout << 4. Find first Occurrence Position \n";
    cout << 5. Count Substring Occurrences \n";
    cout << 6. Find word count \n";
    cout << 7. Separate Nos, Special characters, alphabets \n";
    cout << 8. Exit \n";
```



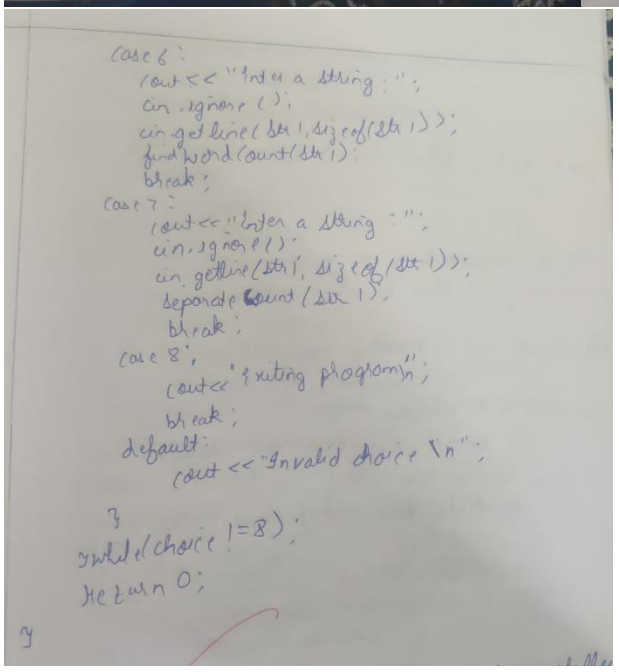
```
void concatStr(char str1[], char str2[]){
    strcat(str1, str2);
    cout << "concatenated Str: " << str1 << endl;
}
void findStrLen(char str[]){
    cout << "Str length: " << strlen(str) << endl;
}
void compStr(char str1[], char str2[]){
    int res = strcmp(str1, str2);
    if (res == 0){
        cout << "Strings are equal \n";
    } else if (res < 0){
        cout << "Str1 is less than Str2 \n";
    } else {
        cout << "Str1 is greater than Str2 \n";
}
void findFirstOccurrence(char main Str[], char subStr[]){
    char* position = strstr(mainStr, subStr);
    if (position != nullptr){
        int index = position - mainStr;
        cout << "First Occurrence Position: " << index << endl;
    } else {
        cout << "SubString not found \n";
    }
}
void countSubStrOccurrences(char mainStr[], char subStr[]){
    int count = 0;
    char* position = mainStr;
    while ((position = strstr(position, subStr)) != nullptr){
        ++count;
        ++position;
    }
    cout << "SubStr Occurrences: " << count << endl;
}
```



```
void findWordCount(char str[]){
    int wordCount = 0;
    bool inWord = false;
    for (int i = 0; i < strlen(str); ++i){
        if (isalpha(str[i])){
            if (!inWord){
                ++wordCount;
                inWord = true;
            }
        }else{
            inWord = false;
        }
    }
    cout << "Word Count: " << wordCount << endl;
}
void separateCount(char str[]){
    int numCount = 0, specialCount = 0, alphaCount = 0;
    for (int i = 0; i < strlen(str); ++i){
        if (isdigit(str[i])){
            ++numCount;
        }else if (isalpha(str[i])){
            ++alphaCount;
        }else{
            ++specialCount;
        }
    }
    cout << "Number Count: " << numCount << endl;
    cout << "Alphabet Count: " << alphaCount << endl;
    cout << "Special Character Count: " << specialCount << endl;
}
int main(){
    char str1[100], str2[100];
    int choice;
    do{
        displayMenu();
        cout << "Enter your choice: ";
```

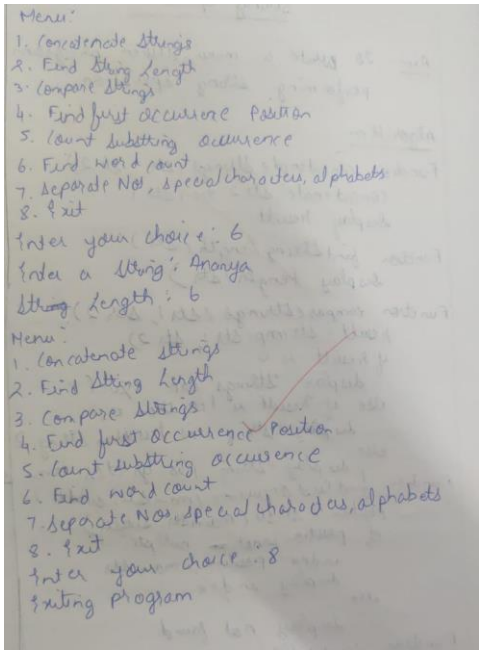


```
cin >> choice;
switch (choice){
    case 1:
        cout << "Enter the 1st string: ";
        cin.ignore();
        cin.getline(str1, sizeof(str1));
        cin.ignore();
        cout << "Enter the second string: ";
        cin.getline(str2, sizeof(str2));
        concatenate(str1, str2);
        break;
    case 2:
        cout << "Enter the first string: ";
        cin.ignore();
        cin.getline(str1, sizeof(str1));
        cout << "Enter the second string: ";
        cin.getline(str2, sizeof(str2));
        compare(str1, str2);
        break;
    case 3:
        cout << "Enter a string: ";
        cin.ignore();
        cin.getline(str1, sizeof(str1));
        findStringLength(str1);
        break;
    case 4:
        cout << "Enter the main string: ";
        cin.ignore();
        cin.getline(str1, sizeof(str1));
        cout << "Enter the substring to find: ";
        cin.getline(str2, sizeof(str2));
        findFirstOccurrence(str1, str2);
        break;
    case 5:
        cout << "Enter the main string: ";
        cin.ignore();
        cin.getline(str1, sizeof(str1));
        cout << "Enter the substring to count: ";
        cin.getline(str2, sizeof(str2));
        countSubstringOccurrence(str1, str2);
        break;
```

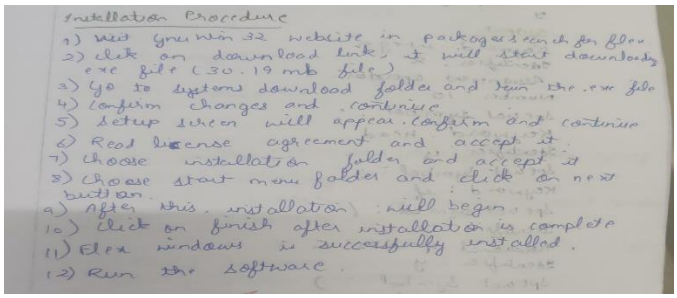


```
    case 6:
        cout << "Enter a string: ";
        cin.ignore();
        cin.getline(str1, sizeof(str1));
        findWordCount(str1);
        break;
    case 7:
        cout << "Enter a string: ";
        cin.ignore();
        cin.getline(str1, sizeof(str1));
        separateCount(str1);
        break;
    case 8:
        cout << "Exiting program\n";
        break;
    default:
        cout << "Invalid choice\n";
    }
}while(choice != 8);
return 0;
}
```

**Output:**




**EXP 2: flex installation**




**EXP 3: implementation of scanner by specifying regular expression**

**Algo:**

1) Define Tokens: Specify patterns for various tokens like keywords, operators, identifiers, and numbers using regular expressions.
2) Lexical Analysis:
   Scan the input string.
   Match patterns for tokens based on the defined regular expressions.
   Print the matched token along with its type using printf.
3) Main Function: Start the lexical analysis by calling yylex().
4) Handle Whitespace: Skip whitespace characters.
5) Error Handling: Ignore any symbols or characters that do not match the specified tokens.
6) End of File: When the end of the input is reached, return 0.

This algorithm describes the basic steps taken by the Lex program to tokenize input strings and identify various types of tokens based on their patterns.

**Program:**

```
%option noyywrap
%{
#include <stdio.h>
%}

%%

integer|read|display|if|else|then|while|for|to|step { printf("Keyword: %s\n", yytext); }
"<="|">="|"<"|">"|"=="|"#" { printf("Relational Operator: %s\n", yytext); }
"+"|"-"|"*"|"/" { printf("Arithmetic Operator: %s\n", yytext); }
"++" { printf("Increment Operator: %s\n", yytext); }
"--" { printf("Decrement Operator: %s\n", yytext); }
"=" { printf("Assignment Operator: %s\n", yytext); }
"("|")"|"{"|"}"|","|";" { printf("Special Symbol: %s\n", yytext); }
[a-zA-Z][a-zA-Z0-9_]* { printf("Identifier: %s\n", yytext); }
[0-9]+ { printf("Number: %s\n", yytext); }
[ \t\n]+ { /* Skip whitespace */ }
. { /* Ignore any other symbols */ }

%%

int main() {
   yylex();
   return 0;
}
```

**Output:**

Sample Input:

```arduino
integer x = 10;
read y;
if (x < y) {
    display "x is less than y";
} else {
    display "x is greater than or equal to y";
}
```

Output:

```yaml
Keyword: integer
Identifier: x
Assignment Operator: =
Number: 10
Special Symbol: ;
Keyword: read
Identifier: y
Special Symbol: ;
Keyword: if
Special Symbol: (
Identifier: x
Relational Operator: <
Identifier: y
Special Symbol: )
Special Symbol: {
Keyword: display
String: "x is less than y"
Special Symbol: ;
Special Symbol: }
Keyword: else
Special Symbol: {
Keyword: display
String: "x is greater than or equal to y"
Special Symbol: ;
Special Symbol: }
```

**EXP 4: Implementation of scanner to scan a file by specifying Regular Expressions.**

**Write a lex program to extract the following tokens:**

| | | |
|---|---|---|
| Keywords | : | integer read display if else then while for to step |
| Relational Operators | : | < <= > >= == # |
| Arithmetic Operators | : | + - * / |
| Increment Operator | : | ++ |
| Decrement Operators | : | -- |
| Assignment Operator | : | = |
| Special Symbols | : | ( ) { } , ; |
| Identifiers | : | (Variables) |
| Numbers | : | (Integer numbers) |
| White space | : | (Eliminate) |
| Any other symbols | : | (Eliminate) |

**Input:** Create the following files and extract the tokens

**file1.star**

```
integer num1, num2, sum;
read num1, num2;
sum = num1 + num2;
```

```
display sum;
```

**file2.star**

```
integer a, b;
read a, b;
if (a > b) then
 {
        display a;
 }
else
{
        display b;
}
```

**file3.star**

```
integer i, n;
read n;
i=1;
while (i <= n)
 {
        display i;
        i++;
 }
```

**file4.star**

```
integer i, n;
read n;
for i = 1 to n step 2
{
        display a;
}
```

**Algo:**

The provided Flex code is essentially a lexical analyzer (lexer) for a programming language. It identifies various tokens such as keywords, identifiers, operators, and symbols. Below is a very short algorithmic description of its functionality:

1. Include necessary header files, especially `stdio.h`.

2. Define patterns using regular expressions to recognize different tokens in the input.

3. For each recognized token, print its corresponding type or classification along with its value.

4. Ignore whitespace and newline characters.

5. Open the input file provided as an argument.

6. Set Flex to read from the file instead of stdin.

7. Start parsing the input file.

8. Close the input file once parsing is complete.

9. Provide a `main()` function to execute the lexer.

This lexer is primarily designed to tokenize a programming language, recognizing keywords (`if`, `for`, `while`, etc.), relational operators (`<`, `<=`, `==`, etc.), arithmetic operators (`+`, `-`, `*`, `/`), increment operators (`++`, `--`, `=`), integers, identifiers, and symbols (`{`, `}`, `(`, `)`, etc.). Unrecognized tokens are flagged as errors.

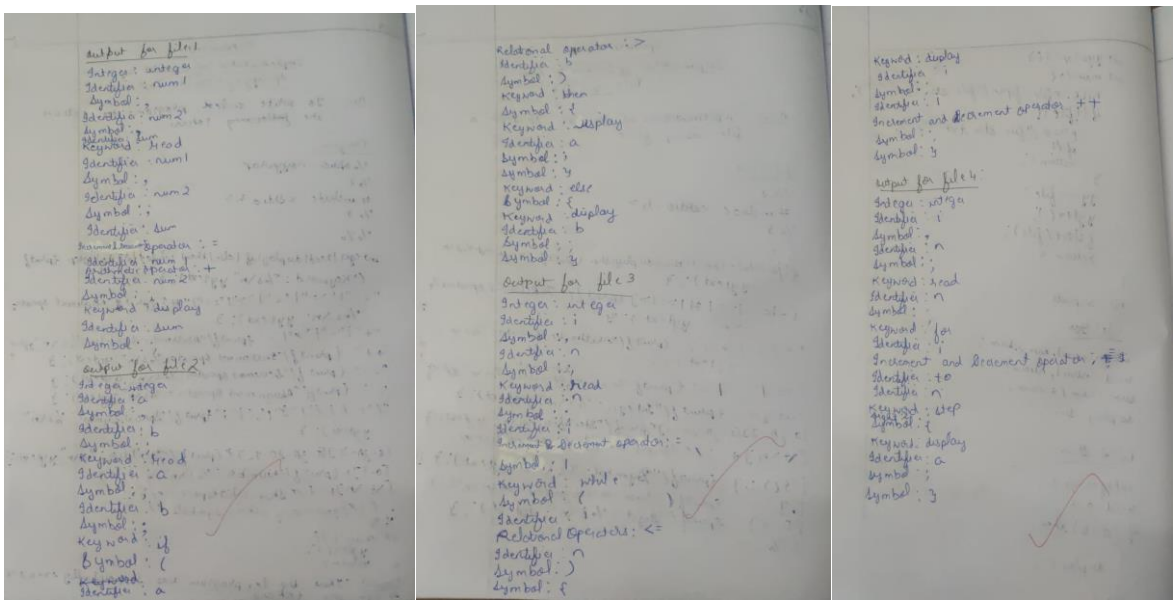
In summary, the lexer reads a file character by character, identifying tokens based on predefined patterns, and printing their types along with their values.

**Program:**

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


%%
if|for|while|read|then|else|display|step { printf("KEYWORD %s\n",yytext); }
[<|<=|==|#|>=|>]   { printf("%s: RELATIONAL operators\n",yytext); }
"+"|"-"|"*"|"/" { printf("Arithmetic Operator: %s\n", yytext); }
"++"|"--"|"=" { printf("Increment Operator: %s\n", yytext); }

"integer"          { printf("INTEGER %s\n",yytext); }
[a-zA-Z][a-zA-Z0-9]*  { printf("IDENTIFIER %s\n",yytext); }
[ \t\n]+         ; /* Ignore whitespace and newline */
[{}();,]         { printf("%s: SYMBOL\n",yytext); }

[.]              { printf("ERROR: Unrecognized token\n"); }
[0-9]            { printf("Digit: %s\n",yytext); }

%%

int yywrap() {}
int main() {
   // Open the file provided as argument
   FILE *file = fopen("file4.star.txt", "r");
   if (!file) {
      perror("file4.star.txt");
```

```
        return 1;
    }

    // Set Flex to read from the file instead of stdin
    yyin = file;

    // Start parsing
    yylex();

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

**Output:**



**EXP 5: implementation of parser**

**Algo:**

The provided code implements a parser for arithmetic expressions. Below is a very short algorithmic description of its functionality:

1. Define a lexer using Flex to recognize tokens in the input, particularly numbers and newline characters.

2. Link the lexer with a parser generated by Bison, denoted by `#include "parser.tab.h"`.

3. Specify parsing rules and precedence using Bison syntax.

4. Define the arithmetic expression grammar, associativity, and token types.

5. Implement parsing rules for arithmetic expressions, including addition, subtraction, multiplication, and division operations.

6. Define a `main()` function to initiate the parsing process.

7. Implement an error-handling function `yyerror()` to handle parsing errors.

8. When the program is executed, it parses the input arithmetic expression and prints the result.

In summary, the parser reads input expressions, parses them according to defined rules, and evaluates arithmetic expressions while considering operator precedence and associativity. It utilizes Flex for lexical analysis and Bison for parsing.

**Program:**

Lex file

```
%{
#include <stdio.h>
#include "parser.tab.h"
%}

%option noyywrap

%%

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

%%
```

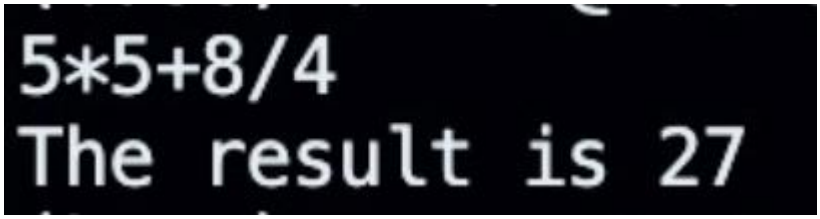
Parser file

```
%{
#include <stdio.h>
void yyerror(const char *s);
int yylex(void);
```

```
int yyparse(void);
%}

%union {
   int num;
}
%token <num> NUMBER
%left '+' '-'
%left '' '/' // Changed to left associativity for '' and '/'
%type <num> AE
%type <num> E

%%

AE : E { printf("The result is %d\n", $$); }
E  : E '' E { $$ = $1 * $3; } // Higher precedence for '' and '/'
   | E '/' E { $$ = $1 / $3; } // Higher precedence for '*' and '/'
   | E '+' E { $$ = $1 + $3; }
   | E '-' E { $$ = $1 - $3; }
   | NUMBER  { $$ = $1; }
   ;

%%

int main() {
   yyparse();
   return 0;
}
```

```
void yyerror(const char *s) {
   printf("Error: %s\n", s);
}
```

**Output:**




**EXP 6: implementation of predictive parser**

**Algo:**

The provided code generates a predictive parsing table for a given context-free grammar. Here's a very short algorithmic description of its functionality:

1. Initialize the predictive parsing table with empty strings.

2. Define the production rules, non-terminals, terminals, first sets, and follow sets for the grammar.

3. For each non-terminal symbol, fill the predictive parsing table:

   a. For each terminal symbol in its first set, add the corresponding production rule.

   b. If epsilon (`@`) is in the first set, add the production rule to each terminal symbol in its follow set.

4. Add the row and column headers for terminals and non-terminals to the table.

5. Print the predictive parsing table.

Algorithm:

1. Initialize the parsing table with empty strings.

2. Iterate through each production rule:

   - For each non-terminal symbol `X`, add corresponding production rules to table[X, a], where `a` is each terminal symbol in FIRST(X).

   - If `@` is in FIRST(X), add corresponding production rules to table[X, b], where `b` is each terminal symbol in FOLLOW(X).

3. Populate the table with terminal and non-terminal symbols.

4. Print the parsing table.

This algorithm constructs a predictive parsing table based on the given grammar and symbols.

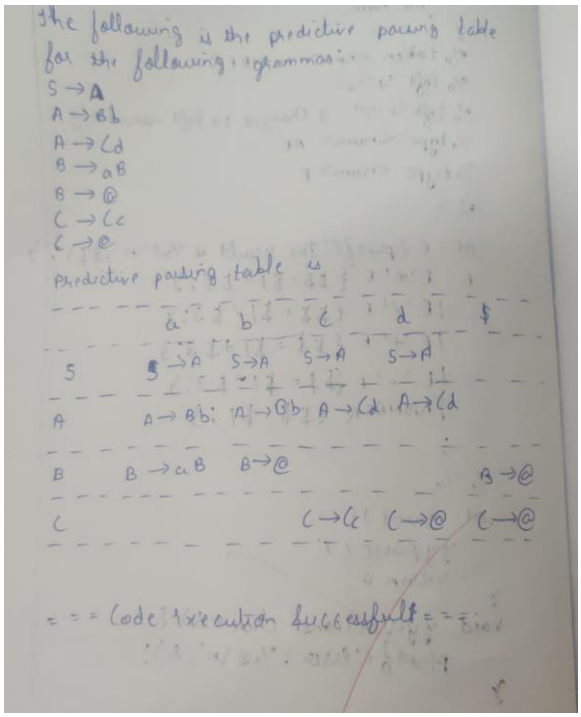
**Program:**

```
#include<stdio.h>
#include<string.h>
char prol[7][10]={"S","A","A","B","B","C","C"};
char pror[7][10]={"A","Bb","Cd","aB","@","Cc","@"};
char prod[7][10]={"S->A","A->Bb","A->Cd","B->aB","B->@","C->Cc","C->@"};
char first[7][10]={"abcd","ab","cd","a@","@","c@","@"};
char follow[7][10]={"$","$","$","a$","b$","c$","d$"};
char table[5][6][10];
int numr(char c)
{
switch(c){
case 'S': return 0;
case 'A': return 1;
case 'B': return 2;
case 'C': return 3;
case 'a': return 0;
case 'b': return 1;
case 'c': return 2;
case 'd': return 3;
case '$': return 4;
}
return(2);
}
void main()
```

```
{
int i,j,k;
for(i=0;i<5;i++)
for(j=0;j<6;j++)
strcpy(table[i][j]," ");
printf("\nThe following is the predictive parsing table for the following grammar:\n");
for(i=0;i<7;i++)
printf("%s\n",prod[i]);
printf("\nPredictive parsing table is\n");
fflush(stdin);
for(i=0;i<7;i++){
k=strlen(first[i]);
for(j=0;j<10;j++)
if(first[i][j]!='@')
strcpy(table[numr(prol[i][0])+1][numr(first[i][j])+1],prod[i]);
}
for(i=0;i<7;i++){
if(strlen(pror[i])==1)
{
if(pror[i][0]=='@')
{
k=strlen(follow[i]);
for(j=0;j<k;j++)
strcpy(table[numr(prol[i][0])+1][numr(follow[i][j])+1],prod[i]);
}
}
}
strcpy(table[0][0]," ");
strcpy(table[0][1],"a");
```

```
strcpy(table[0][2],"b");
strcpy(table[0][3],"c");
strcpy(table[0][4],"d");
strcpy(table[0][5],"$");
strcpy(table[1][0],"S");
strcpy(table[2][0],"A");
strcpy(table[3][0],"B");
strcpy(table[4][0],"C");
printf("\n----------------------------------------------------\n");
for(i=0;i<5;i++)
for(j=0;j<6;j++){
printf("%-10s",table[i][j]);
if(j==5)
printf("\n----------------------------------------------------\n");
}
}
```

**Output:**



**EXP 7: implementation of SLR parser**

**Algo:**

Here's a very concise algorithm for the Simple LR Parser implementation:

1. Read the grammar rules from the input file (`tab6.txt`) and store them.
2. Initialize the LR(0) states.
3. Construct the LR(0) items for the initial state.
4. Compute the closure for each LR(0) item.
5. For each LR(0) item in a state, compute the transition to other states based on the symbols after the dot.
6. Repeat steps 4 and 5 until no new states can be added.
7. Generate the DFA table for transitions between states.
8. Display the DFA table.

This algorithm outlines the steps involved in constructing LR(0) items, computing closures, transitions, and generating the DFA table for the Simple LR Parser.

**Program:**

```
// C code to Implement SLR Parser
/* C program to implement Simple LR Parser. */

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

int i,j,k,m,n=0,o,p,ns=0,tn=0,rr=0,ch=0;
char cread[15][10],gl[15],gr[15][10],temp,templ[15],tempr[15][10],*ptr,temp2[5];
char dfa[15][10];
```

```
struct states
{
   char lhs[15],rhs[15][10];
   int n;//state number
}I[15];

int compstruct(struct states s1,struct states s2)
{
   int t;
   if(s1.n!=s2.n)
      return 0;
   if( strcmp(s1.lhs,s2.lhs)!=0 )
      return 0;
   for(t=0;t<s1.n;t++)
      if( strcmp(s1.rhs[t],s2.rhs[t])!=0 )
         return 0;
   return 1;
}

void moreprod()
{
   int r,s,t,l1=0,rr1=0;
   char *ptr1,read1[15][10];

   for(r=0;r<I[ns].n;r++)
   {
      ptr1=strchr(I[ns].rhs[l1],'.');
      t=ptr1-I[ns].rhs[l1];
```

```
if( t+1==strlen(I[ns].rhs[l1]) )
{
   l1++;
   continue;
}
temp=I[ns].rhs[l1][t+1];
l1++;
for(s=0;s<rr1;s++)
   if( temp==read1[s][0] )
      break;
if(s==rr1)
{
   read1[rr1][0]=temp;
   rr1++;
}
else
   continue;

for(s=0;s<n;s++)
{
   if(gl[s]==temp)
   {
      I[ns].rhs[I[ns].n][0]='.';
      I[ns].rhs[I[ns].n][1]='\0';
      strcat(I[ns].rhs[I[ns].n],gr[s]);
      I[ns].lhs[I[ns].n]=gl[s];
      I[ns].lhs[I[ns].n+1]='\0';
      I[ns].n++;
   }
```

```
        }
    }
}

void canonical(int l)
{
    int t1;
    char read1[15][10],rr1=0,*ptr1;
    for(i=0;i<I[l].n;i++)
    {
        temp2[0]='.';
        ptr1=strchr(I[l].rhs[i],'.');
        t1=ptr1-I[l].rhs[i];
        if( t1+1==strlen(I[l].rhs[i]) )
            continue;

        temp2[1]=I[l].rhs[i][t1+1];
        temp2[2]='\0';

        for(j=0;j<rr1;j++)
            if( strcmp(temp2,read1[j])==0 )
                break;
        if(j==rr1)
        {
            strcpy(read1[rr1],temp2);
            read1[rr1][2]='\0';
            rr1++;
        }
        else
```

```
    continue;

for(j=0;j<I[0].n;j++)
{
    ptr=strstr(I[l].rhs[j],temp2);
    if( ptr )
    {
        templ[tn]=I[l].lhs[j];
        templ[tn+1]='\0';
        strcpy(tempr[tn],I[l].rhs[j]);
        tn++;
    }
}

for(j=0;j<tn;j++)
{
    ptr=strchr(tempr[j],'.');
    p=ptr-tempr[j];
    tempr[j][p]=tempr[j][p+1];
    tempr[j][p+1]='.';
    I[ns].lhs[I[ns].n]=templ[j];
    I[ns].lhs[I[ns].n+1]='\0';
    strcpy(I[ns].rhs[I[ns].n],tempr[j]);
    I[ns].n++;
}

moreprod();
for(j=0;j<ns;j++)
{
```

```
    //if ( memcmp(&I[ns],&I[j],sizeof(struct states))==1 )
    if( compstruct(I[ns],I[j])==1 )
    {
        I[ns].lhs[0]='\0';
        for(k=0;k<I[ns].n;k++)
            I[ns].rhs[k][0]='\0';
        I[ns].n=0;
        dfa[l][j]=temp2[1];
        break;
    }
}
if(j<ns)
{
    tn=0;
    for(j=0;j<15;j++)
    {
        templ[j]='\0';
        tempr[j][0]='\0';
    }
    continue;
}

dfa[l][j]=temp2[1];
printf("\n\nI%d :",ns);
for(j=0;j<I[ns].n;j++)
    printf("\n\t%c -> %s",I[ns].lhs[j],I[ns].rhs[j]);
//getch();
ns++;
tn=0;
```

```
        for(j=0;j<15;j++)
        {
            templ[j]='\0';
            tempr[j][0]='\0';
        }
    }
}

int main()
{
    FILE *f;
    int l;
    //clrscr();

    for(i=0;i<15;i++)
    {
        I[i].n=0;
        I[i].lhs[0]='\0';
        I[i].rhs[0][0]='\0';
        dfa[i][0]= '\0';
    }

    f=fopen("tab6.txt","r");
    while(!feof(f))
    {
        fscanf(f,"%c",&gl[n]);
        fscanf(f,"%s\n",gr[n]);
        n++;
    }
```

```
printf("THE GRAMMAR IS AS FOLLOWS\n");
for(i=0;i<n;i++)
   printf("\t\t\t\t%c -> %s\n",gl[i],gr[i]);

I[0].lhs[0]='Z';
strcpy(I[0].rhs[0],".S");
I[0].n++;
l=0;
for(i=0;i<n;i++)
{
   temp=I[0].rhs[l][1];
   l++;
   for(j=0;j<rr;j++)
      if( temp==cread[j][0] )
         break;
   if(j==rr)
   {
      cread[rr][0]=temp;
      rr++;
   }
   else
      continue;
   for(j=0;j<n;j++)
   {
      if(gl[j]==temp)
      {
         I[0].rhs[I[0].n][0]='.';
         strcat(I[0].rhs[I[0].n],gr[j]);
```

```
            I[0].lhs[I[0].n]=gl[j];
            I[0].n++;
        }
    }
}
ns++;

printf("\nI%d :\n",ns-1);
for(i=0;i<I[0].n;i++)
    printf("\t%c -> %s\n",I[0].lhs[i],I[0].rhs[i]);

for(l=0;l<ns;l++)
    canonical(l);

printf("\n\n\t\tPRESS ANY KEY FOR TABLE");
//getch();
//clrscr();

printf("\t\t\t\nDFA TABLE IS AS FOLLOWS\n\n\n");
for(i=0;i<ns;i++)
{
    printf("I%d : ",i);
    for(j=0;j<ns;j++)
        if(dfa[i][j]!='\0')
            printf("'%c'->I%d | ",dfa[i][j],j);
    printf("\n\n\n");
}
printf("\n\n\n\t\tPRESS ANY KEY TO EXIT");
//getch();
```

}

// Input File tab6.txt For SLR Parser:

// S S+T

// S T

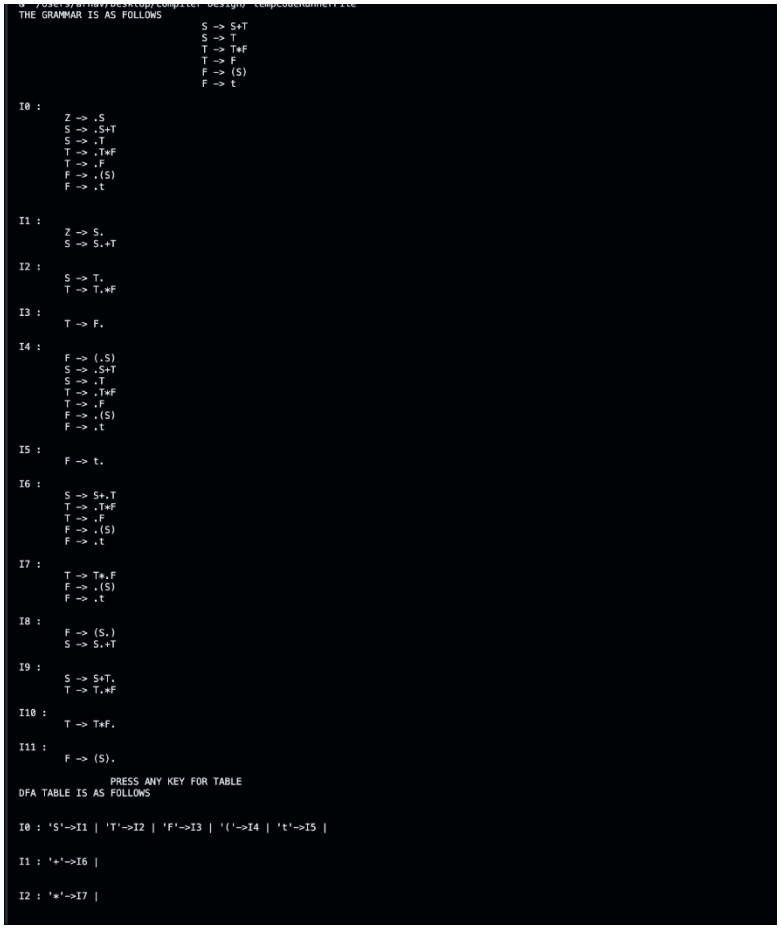
// T T*F

// T F

// F (S)

// F t

**Output:**



```
I3 :

I4 : 'T'->I2 | 'F'->I3 | '('->I4 | 't'->I5 | 'S'->I8 |

I5 :

I6 : 'F'->I3 | '('->I4 | 't'->I5 | 'T'->I9 |

I7 : '('->I4 | 't'->I5 | 'F'->I10 |

I8 : 'F'->I0 | '+'->I6 | ')'->I11 |

I9 : ')'->I1 | '*'->I7 |

I10 :

I11 :
```

**EXP 8: INTRODUCTION TO BASIC JAVA - PROGRAMS IN JAVA**

**a. Write a Java Program to print the message.**

**Algo:**

1) Define a public class named PrintMessage.
2) Define a public static void main method.
3) Inside the main method, use System.out.println to print the message "EX.7 INTRODUCTION TO BASIC JAVA!".

**Program:**

```
public class PrintMessage {
   public static void main(String[] args) {
      System.out.println("EX.7 INTRODUCTION TO BASIC JAVA!");
   }
}
```

**Output:**

```
java -cp /tmp/RJqqIvsYv5/PrintMessage
EX.7 INTRODUCTION TO BASIC JAVA!

=== Code Execution Successful ===
```

**b. Write a Java Program to get the value from keyboard and print.**

**Algo:**

1) Import the Scanner class from java.util package.
2) Define a public class named InputOutput.
3) Define a public static void main method.

4) Inside the main method, create a Scanner object to read input from the keyboard.
5) Prompt the user to enter a value.
6) Read an integer value from the keyboard using scanner.nextInt().
7) Print the value entered by the user.

**Program:**

```
import java.util.Scanner;


public class InputOutput {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      System.out.print("Enter a value: ");
      int value = scanner.nextInt();
      System.out.println("You entered: " + value);
   }
}
```

**Output:**


**c. Write a Java Program to add two integer number.**

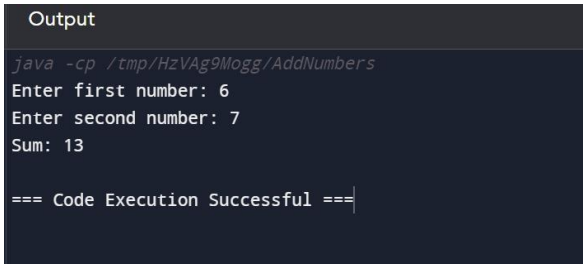
**Algo:**

1) Import the Scanner class from java.util package.
2) Define a public class named AddNumbers.
3) Define a public static void main method.
4) Inside the main method, create a Scanner object to read input from the keyboard.
5) Prompt the user to enter the first number.
6) Read the first integer number from the keyboard using scanner.nextInt().
7) Prompt the user to enter the second number.
8) Read the second integer number from the keyboard using scanner.nextInt().
9) Add the two numbers.
10) Print the sum.

**Program:**

```
import java.util.Scanner;


public class AddNumbers {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter first number: ");
        int num1 = scanner.nextInt();
        System.out.print("Enter second number: ");
        int num2 = scanner.nextInt();
        int sum = num1 + num2;
        System.out.println("Sum: " + sum);
    }
}
```

**Output:**




**d. Write a Java Program to implement Calculator Program.**

**Algo:**

1) Import the Scanner class from java.util package.
2) Define a public class named Calculator.
3) Define a public static void main method.
4) Inside the main method, create a Scanner object to read input from the keyboard.
5) Prompt the user to enter the first number.
6) Read the first double number from the keyboard using scanner.nextDouble().
7) Prompt the user to enter the operator (+, -, *, /).
8) Read the operator character from the keyboard using scanner.next().charAt(0).
9) Prompt the user to enter the second number.
10) Read the second double number from the keyboard using scanner.nextDouble().
11) Perform the arithmetic operation based on the operator entered by the user.

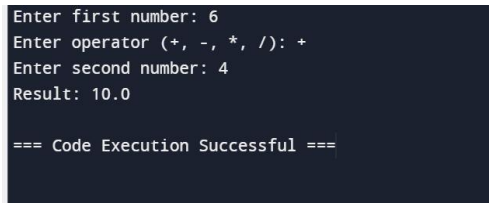
12) Print the result.

**Program:**

```
import java.util.Scanner;


public class Calculator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter first number: ");
        double num1 = scanner.nextDouble();
        System.out.print("Enter operator (+, -, *, /): ");
        char operator = scanner.next().charAt(0);
        System.out.print("Enter second number: ");
        double num2 = scanner.nextDouble();

        double result = 0;
        switch (operator) {
            case '+':
                result = num1 + num2;
                break;
            case '-':
                result = num1 - num2;
                break;
            case '*':
                result = num1 * num2;
                break;
            case '/':
                if (num2 != 0) {
                    result = num1 / num2;
                } else {
```

```
                System.out.println("Error: Division by zero!");
                return;
            }
            break;
        default:
            System.out.println("Invalid operator!");
            return;
    }

    System.out.println("Result: " + result);
  }
}
```

**Output:**




**EXP 9: To traverse syntax tree and perform arithmetic operations**

**Algo:**

1. Define a structure `Node` with data, left child, and right child.

2. Implement a function `constructSyntaxTree(postfix)` to construct a syntax tree from a postfix expression.

- Initialize an empty stack.

- Iterate through each character `c` in the postfix expression.

- If `c` is an operand, push a new node with `c` onto the stack.

- If `c` is an operator:

  - Pop two nodes from the stack as left and right operands.

  - Create a new node with `c` as data and set its left and right children to the popped nodes.

  - Push the new node onto the stack.

- After processing all characters, return the root of the syntax tree (the top of the stack).

3. Implement a function `evaluateSyntaxTree(root)` to recursively evaluate the syntax tree.

- If `root` is null, return 0.

- If `root` contains an operand, return its integer value.

- Recursively evaluate the left and right subtrees.

- Perform the operation on the left and right subtree values based on the operator stored in `root`.

4. In the `main()` function:

- Prompt the user to enter a postfix expression.

- Construct the syntax tree.

- Evaluate the syntax tree.

- Print the result.

**Program:**

```
#include <iostream>
#include <string>
#include <stack>
#include <cctype>

using namespace std;

// Node for syntax tree
struct Node {
    char data;
    Node* left;
    Node* right;

    Node(char data) : data(data), left(nullptr), right(nullptr) {}
};

// Function to check if a character is an operator
```

```
bool isOperator(char c) {
    return c == '+' || c == '-' || c == '*' || c == '/';
}

// Function to construct syntax tree from postfix expression
Node* constructSyntaxTree(const string& postfix) {
    stack<Node*> stack;

    for (char c : postfix) {
        if (isalnum(c)) {
            stack.push(new Node(c));
        } else if (isOperator(c)) {
            Node* rightOperand = stack.top();
            stack.pop();
            Node* leftOperand = stack.top();
            stack.pop();

            Node* newNode = new Node(c);
            newNode->left = leftOperand;
            newNode->right = rightOperand;
            stack.push(newNode);
        }
    }

    return stack.top();
}

// Function to perform arithmetic operation based on operator
int performOperation(char operation, int operand1, int operand2) {
```

```
    switch (operation) {
        case '+':
            return operand1 + operand2;
        case '-':
            return operand1 - operand2;
        case '*':
            return operand1 * operand2;
        case '/':
            return operand1 / operand2;
        default:
            cerr << "Invalid operator!" << endl;
            return 0;
    }
}

// Function to evaluate syntax tree recursively
int evaluateSyntaxTree(Node* root) {
    if (!root)
        return 0;

    if (isalnum(root->data)) {
        return root->data - '0'; // Convert char to int
    }

    int leftValue = evaluateSyntaxTree(root->left);
    int rightValue = evaluateSyntaxTree(root->right);

    return performOperation(root->data, leftValue, rightValue);
}
```

```
int main() {
    string postfixExpression;
    cout << "Enter a postfix expression: ";
    cin >> postfixExpression;

    Node* syntaxTreeRoot = constructSyntaxTree(postfixExpression);

    int result = evaluateSyntaxTree(syntaxTreeRoot);
    cout << "Result: " << result << endl;

    return 0;
}
```

**Output:**


**EXP 10: Intermediate code generation for if and while constructs**

**Algo:**

1. **Generate Label Function:**
   - Implement a function to generate unique labels.

2. **Generate Intermediate Code for If-Else:**
   - Concatenate condition `E`, `TRUE` label, `FALSE` label, `S1_CODE`, a jump to `S_NEXT`, `FALSE` label, and `S2_CODE`.

3. **Generate Intermediate Code for If:**
   - Concatenate condition `E`, `TRUE` label, `S_NEXT` label, `S1_CODE`, and `TRUE` label.

4. **Generate Intermediate Code for While:**

   - Concatenate `BEGIN` label, condition `E`, `TRUE` label, `NEXT` label, `S1_CODE`, a jump to `BEGIN`, and `NEXT` label.

**Program:**

```
#include <iostream>

#include <string>

#include <sstream>


using namespace std;


// Function to generate new label

string newLabel() {

   static int labelCounter = 0;

   stringstream ss;

   ss << "L" << labelCounter++;

   return ss.str();

}


// Function to generate intermediate code for if construct

string generateIf(string E, string S1_CODE, string S_NEXT) {

   string TRUE = newLabel();

   string FALSE = S_NEXT;

   string code = E + " TRUE: " + TRUE + "\n" +

            " FALSE: " + FALSE + "\n" +

            S1_CODE + "\n" +

            TRUE + ": \n";

   return code;

}


// Function to generate intermediate code for if-else construct
```

```
string generateIfElse(string E, string S1_CODE, string S2_CODE, string S_NEXT) {
    string TRUE = newLabel();
    string FALSE = newLabel();
    string code = E + " TRUE: " + TRUE + "\n" +
                  " FALSE: " + FALSE + "\n" +
                  S1_CODE + "\n" +
                  "goto " + S_NEXT + "\n" +
                  FALSE + ": \n" +
                  S2_CODE + "\n";
    return code;
}

// Function to generate intermediate code for while construct
string generateWhile(string E, string S1_CODE) {
    string BEGIN = newLabel();
    string TRUE = newLabel();
    string NEXT = newLabel();
    string code = BEGIN + ": \n" +
                  E + " TRUE: " + TRUE + "\n" +
                  " FALSE: " + NEXT + "\n" +
                  S1_CODE + "\n" +
                  "goto " + BEGIN + "\n" +
                  NEXT + ": \n";
    return code;
}

int main() {
    // Example usage:
    string E = "if (condition)";
```

```
    string S1_CODE = "cout << \"Condition is true\";";
    string S2_CODE = "cout << \"Condition is false\";";
    string S_NEXT = "end;";

    // Generate intermediate code for if-else construct
    string ifElseCode = generateIfElse(E, S1_CODE, S2_CODE, S_NEXT);
    cout << "Intermediate code for if-else:\n" << ifElseCode << endl;

    // Generate intermediate code for if construct
    string ifCode = generateIf(E, S1_CODE, S_NEXT);
    cout << "Intermediate code for if:\n" << ifCode << endl;

    // Generate intermediate code for while construct
    string whileCode = generateWhile(E, S1_CODE);
    cout << "Intermediate code for while:\n" << whileCode << endl;

    return 0;
}
```

**Output:**

```
Intermediate code for if-else:
if (condition) TRUE: L0
 FALSE: L1
cout << "Condition is true";
goto end;
L1:
cout << "Condition is false";

Intermediate code for if:
if (condition) TRUE: L2
 FALSE: end;
cout << "Condition is true";
L2:

Intermediate code for while:
L3:
if (condition) TRUE: L4
 FALSE: L5
cout << "Condition is true";
goto L3
L5:
```