Compiler Design

Exp 1. Write a simple calculator program in C/C++/JAVA.

```
# include <iostream>
using namespace std;

int main() {

  char op;
  float num1, num2;

  cout << "Enter operand 1: ";
  cin >> num1;
  cout << "Enter operand 2: ";
  cin >> num2;
  cout << "Enter operator: (+, -, *, /) : ";
  cin >> op;

  cout<<"Result: ";

  switch(op) {

    case '+':
      cout << num1 << " + " << num2 << " = " << num1 + num2;
      break;

    case '-':
      cout << num1 << " - " << num2 << " = " << num1 - num2;
      break;

    case '*':
      cout << num1 << " * " << num2 << " = " << num1 * num2;
      break;

    case '/':
      cout << num1 << " / " << num2 << " = " << num1 / num2;
      break;

    default:
      cout << "Error! operator is not correct";
      break;
  }

  return 0;
}
```

Output

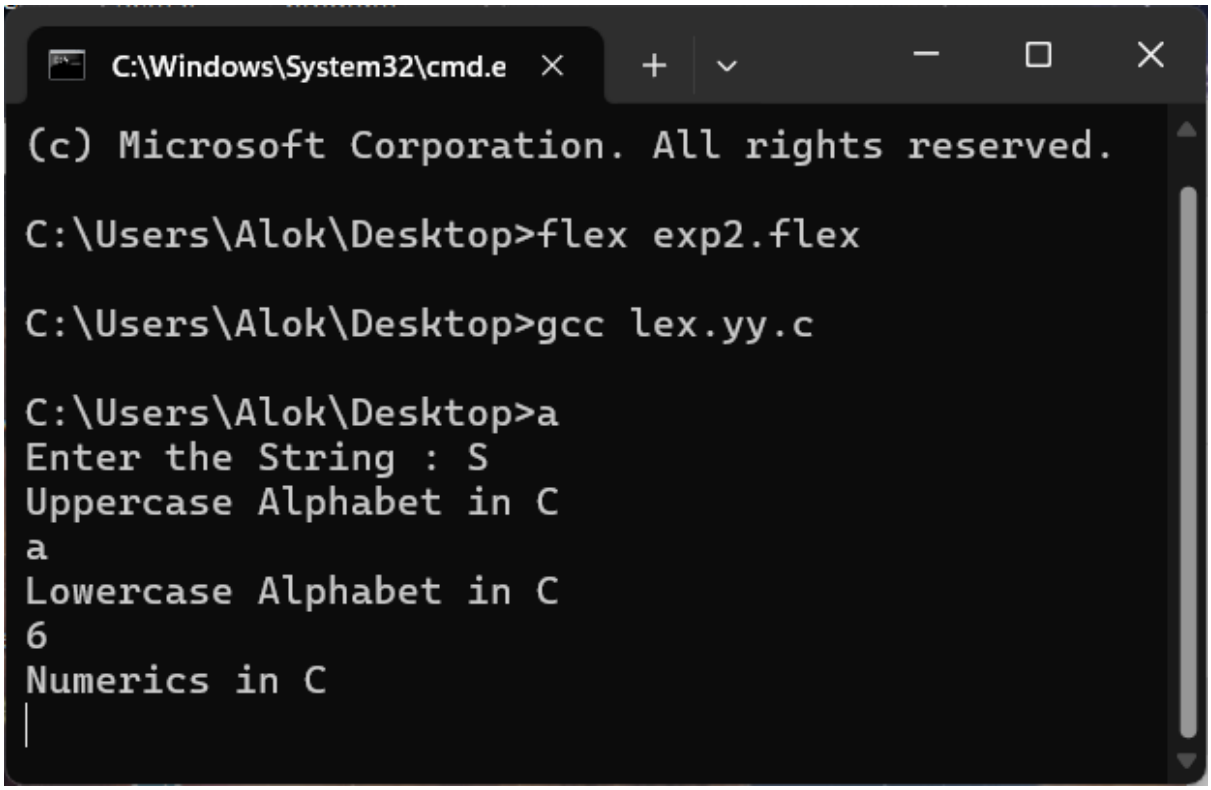
```
/tmp/tbG9EaeXab.o
Enter operand 1: 8
Enter operand 2: 2
Enter operator: (+, -, *, /) : *
Result: 8 * 2 = 16
```

Scanner & Parser

Exp 2. Write a program using FLEX.

filename: exp2.flex

```
%{
%}
%%
[0-9] {printf("Numerics in C");}
[a-z] {printf("Lowercase Alphabet in C");}
[A-Z] {printf("Uppercase Alphabet in C");}
%%
void main()
{
printf("Enter the String : ");yylex();
}
int yywrap(){}
```



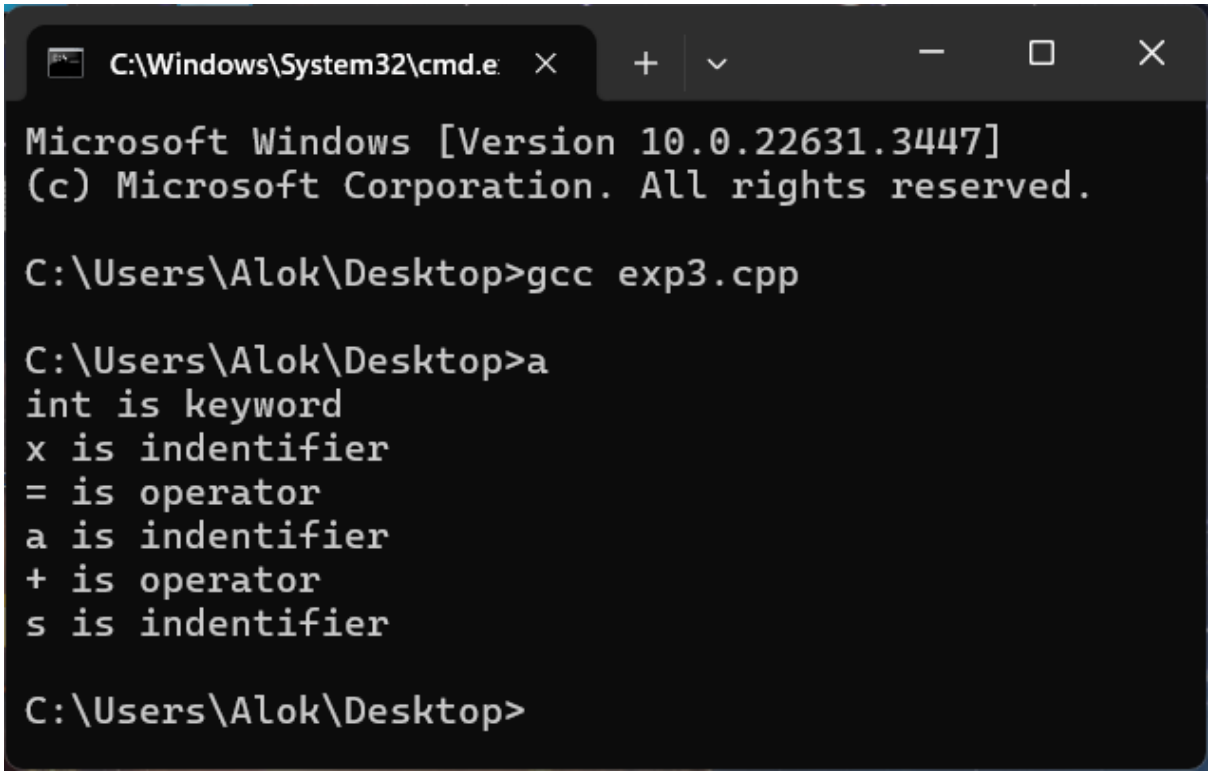
Exp 3. Implementation of scanner by specifying Regular Expressions.

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

int isKeyword(char buffer[]){
    char keywords[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, flag = 0;
    for(i = 0; i < 32; ++i){
        if(strcmp(keywords[i], buffer) == 0){
            flag = 1;
            break;
        }
    }
    return flag;
}
int main(){
    char ch, buffer[15], operators[] = "+-*/%=";
    FILE *fp;
    int i,j=0;
    fp = fopen("program.txt","r");
    if(fp == NULL){
        printf("error while opening the file\n");
        exit(0);
    }
    while((ch = fgetc(fp)) != EOF){
        for(i = 0; i < 6; ++i)
            if(ch == operators[i])
                printf("%c is operator\n", ch);
        if(isalnum(ch))
            buffer[j++] = ch;
        else if((ch == ' ' || ch == '\n') && (j != 0)){
            buffer[j] = '\0';
            j = 0;
            if(isKeyword(buffer) == 1)
                printf("%s is keyword\n", buffer);
            else
                printf("%s is indentifier\n", buffer);
        }
    }
    fclose(fp);
    return 0;
}
```

Create a file named program.txt with any expression such as:

int x = a + s;

Save the file and close it before executing the cpp program, you can compile and run directly from your editor - CodeBlocks or use Command prompt.



Exp 4. Write a program using BISON.

Open the folder named liked win_flex_bison-2.5.25 , create two files named test.l and test.y and run the commands as given in the output image:

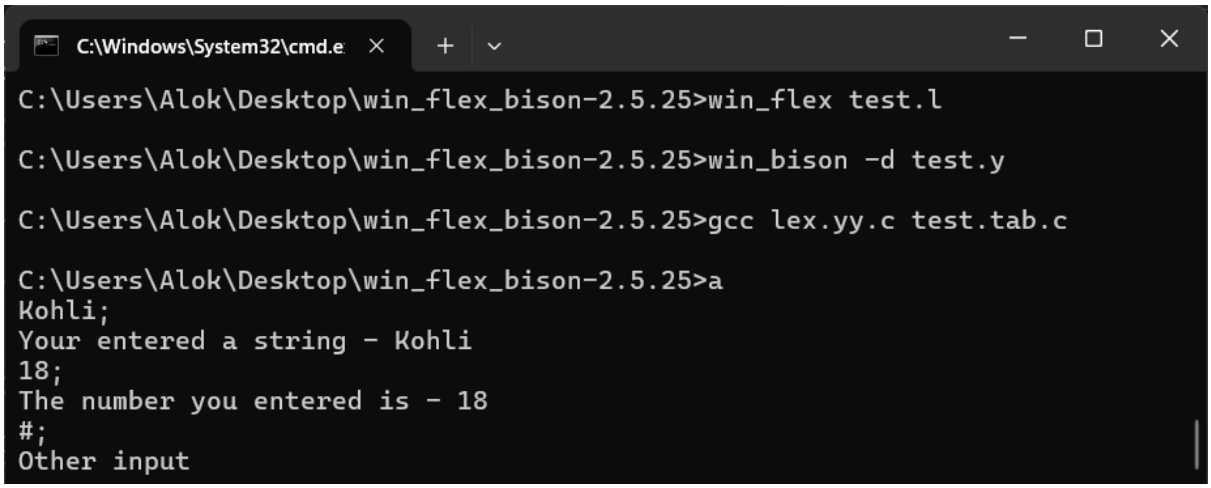
File 1 Name - test.l

```
%{
#include <stdio.h>
#include <string.h>
#include "test.tab.h"
void showError();
%}


numbers     ([0-9])+
alpha       ([a-zA-Z])+

%%

{alpha}                                 {sscanf(yytext, "%s", yylval.name); return (STRING);}
{numbers}       {yylval.number = atoi(yytext); return (NUM);}
";"                     {return (SEMICOLON);}
.                       {showError(); return(OTHER);}

%%


void showError(){
   printf("Other input");
}
int yywrap(){
   return 1;
}
```

File 2 Name - test.y

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

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

%}

%token STRING NUM OTHER SEMICOLON

%type <name> STRING
%type <number> NUM

%union{
            char name[20];
   int number;
}

%%
```

```
prog:
  stmts
;

stmts:
                | stmt SEMICOLON stmts

stmt:
                STRING {
                                printf("Your entered a string - %s", $1);
                }
                | NUM {
                                printf("The number you entered is - %d", $1);
                }
                | OTHER
;

%%

int yyerror(char *s)
{
        printf("Syntax Error on line %s\n", s);
        return 0;
}

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



Exp 5. Write a program for Top Down Parsing - predictive parsing table (Removal of Left recursion/Left factoring and Compute FIRST & FOLLOW).

```
#include <iostream>
#include <vector>
#include <map>
#include <set>

using namespace std;

class Grammar {
private:
    map<char, vector<string>> productions;
    set<char> nonTerminals;
    set<char> terminals;

public:
    Grammar(map<char, vector<string>> prod) {
        productions = prod;
        for (auto const &entry : prod) {
            nonTerminals.insert(entry.first);
            for (auto const &prod : entry.second) {
                for (char symbol : prod) {
                    if (!isupper(symbol) && symbol != '|') {
                        terminals.insert(symbol);
                    }
                }
            }
        }
    }

    void eliminateLeftRecursion() {
        map<char, vector<string>> newProductions;

        for (char nonTerminal : nonTerminals) {
            vector<string> alpha, beta;
            for (string prod : productions[nonTerminal]) {
                if (prod[0] == nonTerminal) {
                    alpha.push_back(prod.substr(1));
                } else {
                    beta.push_back(prod);
                }
            }

            if (!alpha.empty()) {
                char newNonTerminal = nonTerminal + 1;
                newProductions[newNonTerminal] = alpha;
                for (string &prod : beta) {
                    prod += newNonTerminal;
                }
                newProductions[nonTerminal] = beta;
            } else {
```

```
                newProductions[nonTerminal] = productions[nonTerminal];
            }
        }

        productions = newProductions;
    }

    void eliminateLeftFactoring() {
        map<char, vector<string>> newProductions;

        for (char nonTerminal : nonTerminals) {
            map<char, vector<string>> commonPrefixes;
            for (string prod : productions[nonTerminal]) {
                char prefix = prod[0];
                if (commonPrefixes.find(prefix) != commonPrefixes.end()) {
                    commonPrefixes[prefix].push_back(prod.substr(1));
                } else {
                    commonPrefixes[prefix] = {prod.substr(1)};
                }
            }

            for (auto const &entry : commonPrefixes) {
                if (entry.second.size() > 1) {
                    char newNonTerminal = nonTerminal + 1;
                    newProductions[nonTerminal].push_back(string(1,
entry.first) + newNonTerminal);
                    newProductions[newNonTerminal] = entry.second;
                } else {
                    newProductions[nonTerminal].push_back(string(1,
entry.first) + entry.second[0]);
                }
            }
        }

        productions = newProductions;
    }

    map<char, set<char>> constructFirst() {
        map<char, set<char>> first;

        for (char nonTerminal : nonTerminals) {
            first[nonTerminal] = {};
        }

        bool updated = true;
        while (updated) {
            updated = false;
            for (auto const &entry : productions) {
                char nonTerminal = entry.first;
                for (string prod : entry.second) {
                    char symbol = prod[0];
                    if (!isupper(symbol) || symbol == '|') {
                        if (symbol != '#' && first[nonTerminal].find(symbol)
== first[nonTerminal].end()) {
                            first[nonTerminal].insert(symbol);
```

```
                            updated = true;
                        }
                    } else {
                        bool allHaveEpsilon = true;
                        for (char s : prod) {
                            if (s != nonTerminal && first[s].find('#') ==
first[s].end()) {
                                allHaveEpsilon = false;
                                if (first[nonTerminal].find(s) ==
first[nonTerminal].end()) {
                                    first[nonTerminal].insert(s);
                                    updated = true;
                                }
                                break;
                            }
                        }
                        if (allHaveEpsilon && first[nonTerminal].find('#') ==
first[nonTerminal].end()) {
                            first[nonTerminal].insert('#');
                            updated = true;
                        }
                    }
                }
            }
        }

        return first;
    }

    map<char, set<char>> constructFollow(map<char, set<char>> first) {
        map<char, set<char>> follow;
        for (char nonTerminal : nonTerminals) {
            follow[nonTerminal] = {};
        }


        bool updated = true;
        while (updated) {
            updated = false;
            for (auto const &entry : productions) {
                char nonTerminal = entry.first;
                for (string prod : entry.second) {
                    for (size_t i = 0; i < prod.size(); ++i) {
                        char symbol = prod[i];
                        if (isupper(symbol) && symbol != '|') {
                            if (i + 1 < prod.size() && !isupper(prod[i + 1])
&& prod[i + 1] != '|') {
                                if (follow[symbol].find(prod[i + 1]) ==
follow[symbol].end()) {
                                    follow[symbol].insert(prod[i + 1]);
                                    updated = true;
                                }
                            } else {
                                bool allHaveEpsilon = true;
                                for (size_t j = i + 1; j < prod.size(); ++j) {
```

```
                                    char s = prod[j];
                                    if (!isupper(s) || s == '|') {
                                        if (follow[symbol].find(s) ==
follow[symbol].end()) {
                                            follow[symbol].insert(s);
                                            updated = true;
                                        }
                                        allHaveEpsilon = false;
                                        break;
                                    } else {
                                        for (char f : first[s]) {
                                            if (f != '#' &&
follow[symbol].find(f) == follow[symbol].end()) {
                                                follow[symbol].insert(f);
                                                updated = true;
                                            }
                                        }
                                        if (first[s].find('#') ==
first[s].end()) {
                                            allHaveEpsilon = false;
                                            break;
                                        }
                                    }
                                }
                                if (allHaveEpsilon && follow[symbol].find('#')
== follow[symbol].end()) {
                                    for (char f : follow[nonTerminal]) {
                                        if (follow[symbol].find(f) ==
follow[symbol].end()) {
                                            follow[symbol].insert(f);
                                            updated = true;
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }

            return follow;
        }

        void displayProductions() {
            for (auto const &entry : productions) {
                cout << entry.first << " -> ";
                for (string prod : entry.second) {
                    cout << prod << " | ";
                }
                cout << endl;
            }
        }

        void displayFirst(map<char, set<char>> first) {
```

```
        for (auto const &entry : first) {
            cout << "First(" << entry.first << ") = { ";
            for (char f : entry.second) {
                cout << f << " ";
            }
            cout << "}" << endl;
        }
    }

    void displayFollow(map<char, set<char>> follow) {
        for (auto const &entry : follow
        ) {
            cout << "Follow(" << entry.first << ") = { ";
            for (char f : entry.second) {
                cout << f << " ";
            }
            cout << "}" << endl;
        }
    }
};

int main() {
    map<char, vector<string>> productions = {
        {'E', {"E+T", "T"}},
        {'T', {"T*F", "F"}},
        {'F', {"(E)", "id"}}
    };

    Grammar grammar(productions);

    cout << "Original Productions:" << endl;
    grammar.displayProductions();

    grammar.eliminateLeftRecursion();
    cout << "\nProductions after left recursion elimination:" << endl;
    grammar.displayProductions();

    grammar.eliminateLeftFactoring();
    cout << "\nProductions after left factoring:" << endl;
    grammar.displayProductions();

    auto first = grammar.constructFirst();
    cout << "\nFirst sets:" << endl;
    grammar.displayFirst(first);

    auto follow = grammar.constructFollow(first);
    cout << "\nFollow sets:" << endl;
    grammar.displayFollow(follow);

    return 0;
}
```

Output

```
/tmp/YOzQJQJUY5.o
Original Productions:
E -> E+T | T |
F -> (E) | id |
T -> T*F | F |

Productions after left recursion elimination:
E -> TF |
F -> (E) | id |
T -> FU |
U -> *F |

Productions after left factoring:
E -> TF |
F -> (E) | id |
T -> FU |

First sets:
First(E) = { T }
First(F) = { ( i }
First(T) = { F }

Follow sets:
Follow(E) = { ) }
Follow(F) = { ) }
Follow(T) = { ( i }
Follow(U) = { ( i }


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

Exp 6. Write a program for Bottom Up Parsing - SLR Parsing.

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

struct ProductionRule
{
    char left[10];
    char right[10];
};

int main()
{
    char input[20], stack[50], temp[50], ch[2], *token1, *token2, *substring;
    int i, j, stack_length, substring_length, stack_top, rule_count = 0;
    struct ProductionRule rules[10];

    stack[0] = '\0';

    printf("\nEnter the number of production rules: ");
    scanf("%d", &rule_count);

    printf("\nEnter the production rules (in the form 'left->right'): \n");
    for (i = 0; i < rule_count; i++)
    {
        scanf("%s", temp);
        token1 = strtok(temp, "->");
        token2 = strtok(NULL, "->");
        strcpy(rules[i].left, token1);
        strcpy(rules[i].right, token2);
    }

    printf("\nEnter the input string: ");
    scanf("%s", input);

    printf("\nStack \t Input \t Action\n");
    i = 0;

    bool flag = true;

    while (1)
    {
        if (i < strlen(input))
        {
            ch[0] = input[i];
            ch[1] = '\0';
            i++;
            strcat(stack, ch);
            printf("%s\t", stack);
            for (int k = i; k < strlen(input); k++)
            {
```

```
                printf("%c", input[k]);
            }
            printf("\tShift %s\n", ch);
        }

        for (j = 0; j < rule_count; j++)
        {
            substring = strstr(stack, rules[j].right);
            if (substring != NULL)
            {
                stack_length = strlen(stack);
                substring_length = strlen(substring);
                stack_top = stack_length - substring_length;
                stack[stack_top] = '\0';
                strcat(stack, rules[j].left);
                printf("%s\t", stack);
                for (int k = i; k < strlen(input); k++)
                {
                    printf("%c", input[k]);
                }
                printf("\tReduce %s->%s\n", rules[j].left, rules[j].right);

                if (strcmp(stack, rules[0].left) == 0 && i == strlen(input))
                {
                    printf("\nAccepted");
                    flag = false;
                    break;
                } else{
                    j = -1;
                }
            }
        }


        if (i == strlen(input))
        {
            if(flag){
                printf("\nNot Accepted");
            }
            break;
        }
    }

    return 0;
}
```

Note: This program was for Shift-Reduce

```
Enter the number of production rules: 4

Enter the production rules (in the form 'left->right'):
S->(L)
S->a
L->L,S
L->S

Enter the input string: (a,(a,a))

Stack    Input   Action
(       a,(a,a))        Shift (
(a      ,(a,a)) Shift a
(S      ,(a,a)) Reduce S->a
(L      ,(a,a)) Reduce L->S
(L,     (a,a))  Shift ,
(L,(    a,a))   Shift (
(L,(a   ,a))    Shift a
(L,(S   ,a))    Reduce S->a
(L,(L   ,a))    Reduce L->S
(L,(L,  a))     Shift ,
(L,(L,a ))      Shift a
(L,(L,S ))      Reduce S->a
(L,(L   ))      Reduce L->L,S
(L,(L)  )       Shift )
(L,S    )       Reduce S->(L)
(L      )       Reduce L->L,S
(L)             Shift )
S               Reduce S->(L)

Accepted
```

```
Enter the number of production rules: 4

Enter the production rules (in the form 'left->right'):
E->E+E
E->E*E
E->(E)
E->x

Enter the input string: x+x*x

Stack    Input   Action
x       +x*x    Shift x
E       +x*x    Reduce E->x
E+      x*x     Shift +
E+x     *x      Shift x
E+E     *x      Reduce E->x
E       *x      Reduce E->E+E
E*      x       Shift *
E*x             Shift x
E*E             Reduce E->x
E               Reduce E->E*E

Accepted
```

Intermediate Code Generation:

Exp 7. Introduction to basic Java - Programs in java

File Name and Class name should be same, eg. Grades.java

```java
import java.util.Scanner;
public class Grades {
    public static void main(String args[]) {
        // Array to store subject names
        String[] subjects = {"Complier Design", "AI", "OPEN-ELECTIVE", "DBMS",
"PE-B", "PE-D", "IAF", "DBMS-LAB", "CD-LAB", "AI-LAB"};
        int marks[] = new int[10];
        int i;
        float total=0, avg;
        Scanner scanner = new Scanner(System.in);
        for(i=0; i<10; i++) {
            System.out.print("Enter Marks of " + subjects[i] + ": ");
            marks[i] = scanner.nextInt();
            total = total + marks[i];
        }
        scanner.close();
        avg = total/10;
        System.out.println("Total : " + total);
        System.out.println("Average : " + avg);
        System.out.print("The student Grade is: ");
        if(avg>90)
            System.out.print("O");
        else if(avg>80)
            System.out.print("A+");
        else if(avg>70)
           System.out.print("A");
        else if(avg>60)
            System.out.print("B+");
        else if(avg>50)
            System.out.print("B");
        else
            System.out.print("F");
    }
}
```

Command to execute a Java program: java filename.java

java Grades.java

Output

```
java -cp /tmp/iio8Knxxu1/Grades
Enter Marks of Complier Design: 99
Enter Marks of AI: 95
Enter Marks of OPEN-ELECTIVE: 96
Enter Marks of DBMS: 95
Enter Marks of PE-B: 85
Enter Marks of PE-D: 65
Enter Marks of IAF: 75
Enter Marks of DBMS-LAB: 85
Enter Marks of CD-LAB: 35
Enter Marks of AI-LAB: 95
Total : 825.0
Average :82.5
The student Grade is: A+
=== Code Execution Successful ===
```

Exp 8. Write a program to traverse syntax trees and perform action arithmetic operations.

```
import java.util.Stack;

public class SyntaxTree {
    public static class Node {
        String value;
        Node left;
        Node right;

        Node(String value) {
            this.value = value;
            this.left = null;
            this.right = null;
        }
    }

    public static Node buildSyntaxTree(String[] postfixExpr) {
        Stack<Node> stack = new Stack<>();

        for (String token : postfixExpr) {
            if (isOperator(token)) {
                Node rightNode = stack.pop();
                Node leftNode = stack.pop();
                Node operatorNode = new Node(token);
                operatorNode.left = leftNode;
                operatorNode.right = rightNode;
                stack.push(operatorNode);
            } else {
                stack.push(new Node(token));
            }
        }
        return stack.pop();
    }

    public static double evaluateSyntaxTree(Node root) {
        if (root == null)
            return 0;

        if (isNumeric(root.value))
            return Double.parseDouble(root.value);

        double leftValue = evaluateSyntaxTree(root.left);
        double rightValue = evaluateSyntaxTree(root.right);

        switch (root.value) {
            case "+":
                return leftValue + rightValue;
            case "-":
                return leftValue - rightValue;
            case "*":
                return leftValue * rightValue;
```

```
            case "/":
                if (rightValue == 0) {
                    throw new ArithmeticException("Division by zero");
                }
                return leftValue / rightValue;
            default:
                throw new IllegalArgumentException("Invalid operator: " +
root.value);
        }
    }

    private static boolean isNumeric(String str) {
        try {
            Double.parseDouble(str);
            return true;
        } catch (NumberFormatException e) {
            return false;
        }
    }

    private static boolean isOperator(String str) {
        return str.equals("+") || str.equals("-") || str.equals("*") ||
str.equals("/");
    }

    public static void main(String[] args) {
        // Example postfix expression: "5 3 + 2 *"
        String[] postfixExpr = {"5", "3", "+", "2", "*"};

        Node root = buildSyntaxTree(postfixExpr);

        double result = evaluateSyntaxTree(root);
        System.out.println("Result: " + result);
    }
}
```

Command to execute a Java program: java filename.java

java SyntaxTree.java

```
Output
java -cp /tmp/uzFDBCmTKl/SyntaxTree
Result: 16.0

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

Exp 9. Write an Intermediate code generation for If/While.

```java
public class If_While_Code {

    // Function to generate code for the if statement
    public static String generateIfCode(String condition, String action) {
        String code = "if (" + condition + ") {\n";
        code += "\t" + action + "\n";
        code += "}";
        return code;
    }

    // Function to generate code for the while loop
    public static String generateWhileCode(String condition, String action) {
        String code = "while (" + condition + ") {\n";
        code += "\t" + action + "\n";
        code += "}";
        return code;
    }

    public static void main(String[] args) {

        // Example input: if (x > 8) { y = x * 2; }
        String if_condition = "x > 8";
        String if_action = "y = x * 2;";

        String ifCode = generateIfCode(if_condition, if_action);
        System.out.println("Generated code for If statement:\n" + ifCode);

        // Example input: while (i < 10) { System.out.println(i); i++; }
        String while_condition = "i < 10";
        String while_action = "System.out.println(i); i++;";

        String whileCode = generateWhileCode(while_condition, while_action);
        System.out.println("Generated code for While loop:\n" + whileCode);
    }
}
```

A different code on next page

Exp 9 Different : IF-While CODE Generation:

```
import java.util.*;

public class If_While_Code2 {

    private static int labelCounter = 0;

    public static void main(String[] args) {
        String javaCode = "int x = 5;\n" +
                          "if (x > 0) {\n" +
                          "    System.out.println(\"x is positive\");\n" +
                          "}\n" +
                          "else {\n" +
                          "    System.out.println(\"x is non-positive\");\n" +
                          "}\n" +
                          "while (x > 0) {\n" +
                          "    x--;\n" +
                          "}\n";
        String intermediateCode = generateIntermediateCode(javaCode);
        System.out.println(intermediateCode);
    }

    public static String generateIntermediateCode(String javaCode) {
        StringBuilder intermediateCode = new StringBuilder();
        String[] lines = javaCode.split("\n");

        for (String line : lines) {
            line = line.trim();
            if (line.startsWith("if")) {
                intermediateCode.append(generateIfIntermediateCode(line));
            } else if (line.startsWith("while")) {
                intermediateCode.append(generateWhileIntermediateCode(line));
            } else {
                // Handle other statements if necessary
                intermediateCode.append(line).append("\n");
            }
        }

        return intermediateCode.toString();
    }

    public static String generateIfIntermediateCode(String ifStatement) {
        String condition = ifStatement.substring(ifStatement.indexOf("(") + 1,
ifStatement.indexOf(")")).trim();
        String trueLabel = getNextLabel();
        String falseLabel = getNextLabel();
        String intermediateCode = String.format("if (%s) goto %s;\n",
condition, falseLabel);
        intermediateCode += "  // True branch\n";
        intermediateCode += String.format("goto %s;\n", trueLabel);
        intermediateCode += String.format("%s:\n", falseLabel);
        intermediateCode += "  // False branch\n";
        intermediateCode += String.format("%s:\n", trueLabel);
```

```
        return intermediateCode;
    }

    public static String generateWhileIntermediateCode(String whileStatement)
{
        String condition =
whileStatement.substring(whileStatement.indexOf("(") + 1,
whileStatement.indexOf(")")).trim();
        String startLabel = getNextLabel();
        String endLabel = getNextLabel();
        String intermediateCode = String.format("%s:\n", startLabel);
        intermediateCode += String.format("if (!(%s)) goto %s;\n", condition,
endLabel);
        intermediateCode += String.format("goto %s;\n", startLabel);
        intermediateCode += String.format("%s:\n", endLabel);
        return intermediateCode;
    }

    public static String getNextLabel() {
        return "L" + labelCounter++;
    }
}
```

Code Generation:

Exp 10. Introduction to MIPS Assembly language- (Teach spim mips simulator).

```
.data
num1: .word 10
num2: .word 6
.text
main:
   # load num1 into $a0
   lw $a0, num1
   # load num2 into $a1
   lw $a1, num2
   # add num1 and num2
   add $t0, $a0, $a1      # Store the result in $t0
   # print result
   move $a0, $t0          # Move the result from $t0 to $a0 for printing

   li $v0, 1              # syscall code for print_int
   syscall

   # exit program
   li $v0, 10             # syscall code for exit
   syscall
```

Download Mars MIPS Simulator:
https://courses.missouristate.edu/KenVollmar/mars/MARS_4_5_Aug2014/Mars4_5.jar

Copy the Code and Save the File
Press F3 – Assemble the File
Press F5 – Run the Program

Exp 11. Write a program to generate machine code for a simple statement.

```
import java.io.FileOutputStream;
import java.io.IOException;
public class SimpleStatementMachineCode {
    public static void main(String[] args) throws IOException {
        //Create a byte array to store the machine code
        byte[] machineCode = new byte[4];
        //Set the opcode for the MIPS instruction in the byte array
        machineCode[0] = (byte) 0x0;
        //Set the source register for the first operand in the byte array
        machineCode[1] = (byte) 0x0;
        //Set the source register for the second operand in the byte array
        machineCode[2] = (byte) 0x80;
        //Set the destination register in the byte array
        machineCode[3] = (byte) 0x21;
        //Write the machine code to a file
        try (FileOutputStream fos = new FileOutputStream("machine_code.bin")){
            fos.write(machineCode);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Exp 12. Write a program to generate machine code for an indexed assignment statement.

```
import java.io.FileOutputStream;
import java.io.IOException;
public class GenerateMachineCodeIndexedAssignment {
    public static void main(String[] args) throws IOException {
        // Create a byte array to store the machine code
        byte[] machineCode = new byte[6];
        // Set the opcode for the MIPS instruction in the byte array
        machineCode[0] = (byte) 0x2b;
        // Set the source register in the byte array
        machineCode[1] = (byte) 0x04;
        // Set the base register in the byte array
        machineCode[2] = (byte) 0x00;
        // Set the offset in the byte array
        machineCode[3] = (byte) 0x00;
        machineCode[4] = (byte) 0x00;
        machineCode[5] = (byte) 0x18;

        // Write the machine code to a file
        try (FileOutputStream fos = new FileOutputStream("machine_code.bin"))
{
            fos.write(machineCode);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```