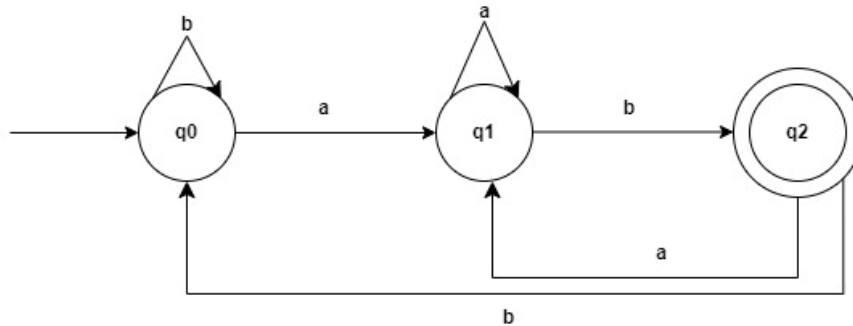


Lab 1:**Date: 2080/02/09****Write a program to implement DFA that accepts string ending with ab.**

Introduction:

*fig: DFA accepting the string ending with ab*

States	a	b
q0	q1	q0
q1	q1	q2
q2	q1	q0

Initial State: q0

Final State: q2

Input: a b

Code:

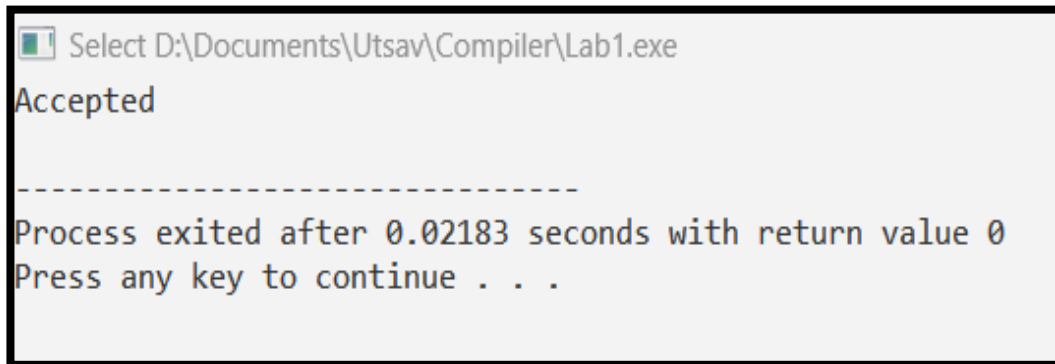
```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

char state[3];
bool flag;
int i;
bool dfaEndingWithab(const char* checkString) {
    strcpy(state, "q0");
    int length = strlen(checkString);

    for (i = 0; i < length; i++) {
        if (checkString[i] == 'a' && strcmp(state, "q0") == 0)
            strcpy(state, "q1");
        if (checkString[i] == 'b' && strcmp(state, "q0") == 0)
            strcpy(state, "q0");
        if (checkString[i] == 'a' && strcmp(state, "q1") == 0)
            strcpy(state, "q1");
        if (checkString[i] == 'b' && strcmp(state, "q1") == 0 && i ==
(length - 1))
            flag = true;
        if (checkString[i] == 'b' && strcmp(state, "q1") == 0)
            strcpy(state, "q2");
        if (checkString[i] == 'a' && strcmp(state, "q2") == 0)
            strcpy(state, "q1");
        if (checkString[i] == 'b' && strcmp(state, "q2") == 0)
            strcpy(state, "q0");
        else
            flag = false;
    }
    return flag;
}

int main() {
    const char* checkString = "bbbabbbbab";
    dfaEndingWithab(checkString);
    if (flag == true)
        printf("Accepted\n");
    else
        printf("Rejected\n");
    return 0;
}
```

Output:



```
Select D:\Documents\Utsav\Compiler\Lab1.exe
Accepted

-----
Process exited after 0.02183 seconds with return value 0
Press any key to continue . . .
```

Write a program for comment validation.

Introduction:

In programming, a comment is a piece of text that is used to provide explanations, descriptions, or notes within the source code of a program. Comments are ignored by the compiler or interpreter and do not affect the execution of the program. They are purely for the benefit of human readers, including the programmers themselves and other developers who may need to understand or modify the code in the future.

Comment validation refers to the process of verifying the correctness and adherence of comments in the source code. It involves checking if the comments follow certain rules or guidelines set by the programming language or the coding standards of a particular project.

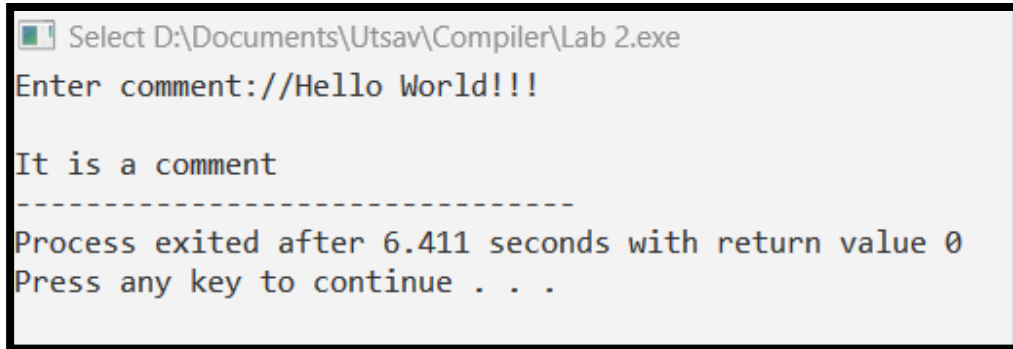
Comment validation can include various aspects, such as:

1. **Comment format:** Ensuring that comments are correctly formatted and use the appropriate syntax. For example, in C/C++/Java, single-line comments start with `"/"` and multi-line comments are enclosed between `"/"` and `"/"`.
2. **Comment placement:** Verifying that comments are placed in appropriate locations within the code, providing relevant information about the associated code block or explaining its purpose.
3. **Comment content:** Checking if the comments accurately describe the code's functionality, logic, or intent. Comments should be clear, concise, and meaningful, helping other developers understand the code more easily.
4. **Comment consistency:** Ensuring that comments are consistent throughout the codebase, following a consistent style and level of detail. This can be specified by project-specific coding standards or style guides.

Code:

```
#include<stdio.h>
#include<conio.h>
int main()
{
    char com[30];
    int i=2, a=0;
    printf("Enter comment:");
    gets(com);
    if(com[0]=='/')
    {
        if(com[1]=='/')
            printf("\nIt is a comment");
        else if(com[1]=='*')
        {
            for(i=2;i<=30;i++)
            {
                if(com[i]=='*'&&com[i+1]=='/')
                {
                    printf("\nIt is a comment");
                    a=1;
                    break;
                }
                else
                    continue;
            }
            if(a==0)
                printf("\nIt is not a comment");
        }
        else
            printf("\nIt is not a comment");
    }
    else
        printf("\nIt is not a comment");
    return 0;
}
```

Output:



```
Select D:\Documents\Utsav\Compiler\Lab 2.exe
Enter comment://Hello World!!!

It is a comment
-----
Process exited after 6.411 seconds with return value 0
Press any key to continue . . .
```

Lab 3:**Date: 2080/02/12****Write a program to recognize string under 'a*', 'a*b+', 'abb'.**

Introduction:

A regular expression, often abbreviated as regex or regexp, is a sequence of characters that forms a search pattern. It is a powerful tool used for pattern matching and manipulating strings. Regular expressions are supported by many programming languages, text editors, and command-line tools.

1. 'a*': This expression represents zero or more occurrences of the character 'a'. Therefore, the strings accepted by this expression can be any combination of 'a' characters, including an empty string. For example, it accepts strings like "", "a", "aa", "aaa", and so on.
2. 'a*b+': This expression represents a sequence that starts with zero or more 'a' characters, followed by one or more 'b' characters. In other words, it matches any string that begins with 'a' (optional), followed by one or more 'b' characters. For example, it accepts strings like "b", "ab", "aab", "aaab", and so on.
3. 'abb': This expression matches the exact string "abb". It only accepts the string "abb" and no other variations or additional characters.

Code:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int main()
{
    char s[20], c;
    int state=0, i=0;
    printf("Enter a string:");
    gets(s);
    while(s[i]!='\0')
    {
        switch(state)
        {
            case 0:
                c=s[i++];
                if(c=='a')
                    state=1;
                else if(c=='b')
                    state=2;
                else
                    state=6;
                break;
            case 1:
                c=s[i++];
                if(c=='a')
                    state=3;
                else if(c=='b')
                    state=4;
                else
                    state=6;
                break;
            case 2:
                c=s[i++];
                if(c=='a')
                    state=6;
                else if(c=='b')
                    state=2;
                else
                    state=6;
                break;
            case 3:
```

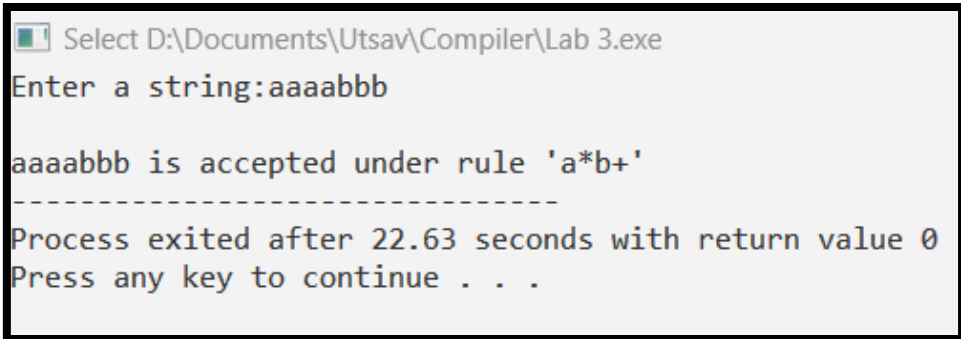


```

        c=s[i++];
        if(c=='a')
            state=3;
        else if(c=='b')
            state=2;
        else
            state=6;
            break;
    case 4:
        c=s[i++];
        if(c=='a')
            state=6;
        else if(c=='b')
            state=5;
        else
            state=6;
            break;
    case 5:
        c=s[i++];
        if(c=='a')
            state=6;
        else if(c=='b')
            state=2;
        else
            state=6;
            break;
    case 6:
        printf("\n%s is not recognized", s);
        exit(0);
    }
}
if(state==1)
    printf("\n%s is accepted under rule 'a'", s);
else if((state==2)||(state==4))
    printf("\n%s is accepted under rule 'a*b+', s);
else if(state==5)
    printf("\n%s is accepted under rule 'abb'", s);
return 0;
}

```

Output:



```
Select D:\Documents\Utsav\Compiler\Lab 3.exe
Enter a string:aaaabbb

aaaabbb is accepted under rule 'a*b+'
-----
Process exited after 22.63 seconds with return value 0
Press any key to continue . . .
```

Lab 4:

Date: 2080/02/16

Write a program to test whether a given identifier is valid or not.

Introduction:

Identifiers in computer programming are like names given to different parts of a program, such as variables, functions, or classes. They help programmers understand and refer to these parts easily. Here are some simple rules for valid identifiers:

1. **Character Set:** Identifiers can use letters (uppercase and lowercase), digits, and underscores (_). Some languages may allow additional characters like dollar signs (\$) or periods (.), but they have specific uses.

Example: validIdentifier, myVariable, calculate_sum

2. **Starting Character:** Identifiers must start with a letter (uppercase or lowercase) or an underscore (_). They cannot begin with a digit or special character.

Example: firstName, _total_count, calculateAverage

3. **Length:** Identifiers can be any length, but it's best to choose concise names that are still meaningful. Some languages may have limits on identifier length.

Example: maxScore, num1, totalNumberOfStudents

4. **Case Sensitivity:** Most programming languages consider uppercase and lowercase letters as different. So, "name" and "Name" are distinct identifiers.

Example: temperature, UserName, isValid

5. **Reserved Keywords:** Identifiers cannot have the same name as reserved keywords or language-specific keywords. These are words with predefined meanings in the programming language.

Example: if, while, function, class

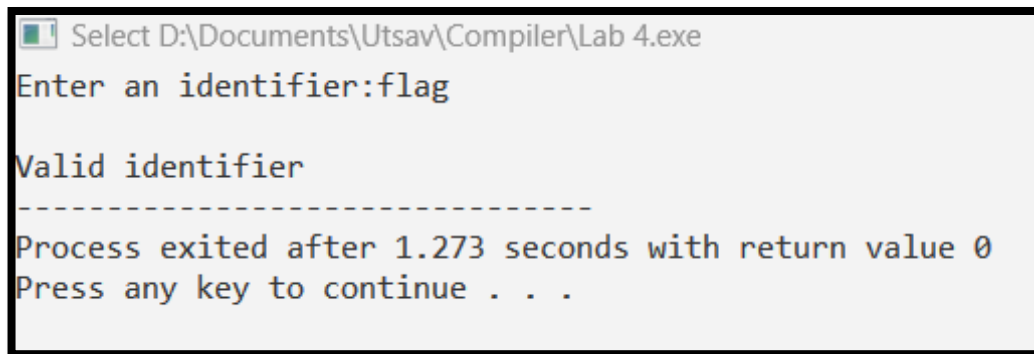
6. **Special Characters:** Some programming languages may restrict the use of spaces, hyphens, or other symbols in identifiers.

Example: studentName, total_marks, calculateSum

Code:

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
int main()
{
    char a[10];
    int flag, i=1;
    printf("Enter an identifier:");
    gets(a);
    if(isalpha(a[0]) || a[0]=='_')
        flag=1;
    else
        printf("Not a valid identifier");
    while(a[i]!='\0')
    {
        if(!isdigit(a[i]) && !isalpha(a[i]) && a[i] != '_')
        {
            flag=0;
            break;
        }
        i++;
    }
    if(flag==1)
        printf("\nValid identifier");
    else
        printf("\nNot a valid identifier");
    return 0;
}
```

Output:



```
Select D:\Documents\Utsav\Compiler\Lab 4.exe
Enter an identifier:flag
Valid identifier
-----
Process exited after 1.273 seconds with return value 0
Press any key to continue . . .
```

Lab 5:**Date: 2080/02/17****Write a program for Lexical Analyzer.**

Introduction:

A lexical analyzer, also called a lexer or scanner, is a vital part of a compiler or interpreter for programming languages. Its main job is to break down the source code into small, meaningful units called tokens. Tokens represent different parts of the program, like keywords, names, numbers, symbols, and operators.

The lexical analyzer works by examining the source code character by character. It ignores spaces, comments, and other unimportant elements and focuses on identifying and categorizing tokens. It follows predefined rules and patterns to recognize and extract tokens accurately.

To do this, the lexical analyzer uses techniques like regular expressions or finite automata to match patterns and find tokens in the source code. It scans the code and generates a stream of tokens as output.

This stream of tokens is then used by other parts of the compiler or interpreter to understand the structure and meaning of the program. The tokens provide important information about the program's syntax and semantics, helping with further analysis and translation processes.

inputfile.txt contains:

```
void main()
{
    int a = 10;
    int b = 5;
    int mul = a*b;
}
```

Code:

```
#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("inputfile.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 identifier\n", buffer);
        }
    }
    fclose(fp);
    return 0;
}
```


Output:

```
Select D:\Documents\Utsav\Compiler\Lab 5.exe
void is keyword
main is identifier
int is keyword
a is identifier
= is operator
10 is identifier
int is keyword
b is identifier
= is operator
5 is identifier
int is keyword
mul is identifier
= is operator
* is operator
ab is identifier

-----
Process exited after 0.01911 seconds with return value 0
Press any key to continue . . .
```

Lab 6:**Date: 2080/02/18****Write a program to find out first and follow of given grammar.**

Introduction:

nullable():

In the context of formal language theory and parsing, nullable() is a function or attribute associated with a grammar production or a symbol. It determines whether a symbol or production can derive the empty string (ϵ) or not. If a symbol or production is nullable, it means that it can produce the empty string as a valid derivation. The nullable() function is often used in various parsing algorithms, such as constructing nullable sets or calculating first and follow sets.

firstpos():

firstpos() is a function used in compiler design and syntax analysis, specifically in the construction of the syntax tree or parse tree. It is associated with a node in the syntax tree and represents the set of positions (or indexes) in the input string where the first character of that node's corresponding construct (such as a nonterminal or terminal) can appear. The firstpos() function is crucial for determining the first positions of nodes in the syntax tree, which are later used in constructing various parsing tables or for further analysis.

followpos():

followpos() is another function used in compiler design and syntax analysis, particularly in the construction of the syntax tree or parse tree. It is associated with a node in the syntax tree and represents the set of positions (or indexes) in the input string where the next character following the construct represented by that node can appear. The followpos() function is used to calculate the follow positions of nodes in the syntax tree, which are important for constructing parsing tables and performing further parsing or analysis operations.

Code for firstpos():

```
#include<stdio.h>
#include<ctype.h>
void FIRST(char[ ], char );
void addToResultSet(char[ ], char);
int numOfProductions;
char productionSet[10][10];
int main()
{
    int i;
    char choice;
    char c;
    char result[20];
    printf("How many number of productions ? :");
    scanf(" %d", &numOfProductions);

    for(i=0; i <numOfProductions; i++)
    {
        printf("Enter productions Number %d : ",i+1);
        scanf(" %s", productionSet[i]);
    }
    do
    {
        printf("\nFind the FIRST of :");
        scanf(" %c", &c);

        FIRST(result, c);
        printf("\nFIRST(%c)= { ",c);
        for(i=0;result[i]!='\0';i++)

            printf(" %c ",result[i]);
        printf("}\n");
        printf("Continue(y/n) : ");
        scanf(" %c", &choice);
    }while(choice=='y'||choice=='Y');
}
void FIRST(char* Result, char c)
{
    int i, j, k;
    char subResult[20];
    int foundEpsilon;
    subResult[0]='\0';
    Result[0]='\0';
```

```

    if(!(isupper(c)))
    {
        addToResultSet(Result, c);
        return ;
    }

    for(i=0; i<numOfProductions; i++)
    {
        if(productionSet[i][0]==c)
        {
            if(productionSet[i][2]=='$')
                addToResultSet(Result, '$');

            else
            {
                j=2;
                while(productionSet[i][j]!='\0')
                {
                    foundEpsilon=0;
                    FIRST(subResult,
productionSet[i][j]);
                    for(k=0; subResult[k]!='\0'; k++)
                        addToResultSet(Result,
subResult[k]);

                    for(k=0; subResult[k]!='\0'; k++)
                    {
                        if(subResult[k]=='$')
                        {
                            foundEpsilon=1;
                            break;
                        }
                    }

                    if(!foundEpsilon)
                        break;

                    j++;
                }
            }
        }
    }

    return;
}

```

```
void addToResultSet(char Result[ ], char val)
{
    int k;
    for(k=0 ;Result[k]!='\0';k++)
        if(Result[k]==val)
            return;
    Result[k]=val;
    Result[k+1]='\0';
}
```

Output:

```
Select D:\Documents\Utsav\Compiler\Lab 6.1.exe
How many number of productions ? :6
Enter productions Number 1 : S=ABCDE
Enter productions Number 2 : A=a
Enter productions Number 3 : B=b
Enter productions Number 4 : C=c
Enter productions Number 5 : D=d
Enter productions Number 6 : E=e

Find the FIRST of :S

FIRST(S)= { a }
Continue(y/n) : y

Find the FIRST of :A

FIRST(A)= { a }
Continue(y/n) : y

Find the FIRST of :B

FIRST(B)= { b }
Continue(y/n) : n

-----
Process exited after 119.3 seconds with return value 110
Press any key to continue . . .
```

Code for followpos():

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
int n,p,i=0,j=0;
char a[10][10],Result[10];
char subResult[20];
void follow(char* Result,char c);
void first(char* Result,char c);
void addToResultSet(char[ ], char);
int main()
{
    int i;
    int choice;
    char c, ch;
    printf("Enter the no. of productions: ");
    scanf("%d", &n);
    printf("Enter %d productions\n", n);
    for(i=0;i<n;i++)
        scanf("%s", a[i]);
    do
    {
        printf("Find FOLLOW of -->");
        scanf(" %c", &c);
        follow(Result, c);
        printf("FOLLOW(%c) = { ", c);
        for(i=0;Result[i]!='\0';i++)
            printf(" %c ", Result[i]);
        printf(" }\n");
        printf("Press 1 to continue");
        scanf("%d", &choice);
    } while(choice==1);
}
void follow(char* Result, char c)
{
    int k;
    subResult[0]='\0';
    Result[0]='\0';
    if(a[0][0]==c) addToResultSet(Result,'$');
    for(i=0;i<n;i++)
    {
        for(j=2;j<strlen(a[i]);j++)
```

```

        if(a[i][j]==c)
        {

            if(a[i][j+1]!='\0')first(subResult,a[i][j+1]);

            if(a[i][j+1]=='\0'&& c!=a[i][0])

                follow(subResult,a[i][0]);
                                for(k=0;subResult[k]!='\0';k++)

                addToResultSet(Result,subResult[k]);
                                }
        }
    }
}

void first(char* R, char c)
{
    int k, m;
    if(!(isupper(c))&&c!='#')
        addToResultSet(R, c);
    for(k=0;k<n;k++)
    {
        if(a[k][0]==c)
        {
            if(a[k][2]=='#'&&c!=a[i][0])
                follow(R, a[i][0]);
            else if(!(isupper(a[k][2]))&&a[k][2]!='#')
                addToResultSet(R, a[k][2]);
            else first(R, a[k][2]);
                                for(m=0;R[m]!='\0';m++)
                                    addToResultSet(Result, R[m]);
        }
    }
}

void addToResultSet(char Result[], char val)
{
    int k;
    for(k=0 ;Result[k]!='\0';k++)
        if(Result[k]==val)
            return;

    Result[k]=val;
    Result[k+1]='\0';
}

```


Output:

```
Select D:\Documents\Utsav\Compiler\Lab 6.2.exe
Enter the no. of productions: 6
Enter 6 productions
S=ABCDE
A=a
B=b
C=c
D=d
E=e
Find FOLLOW of -->S
FOLLOW(S) = { $ }
Press 1 to continue1
Find FOLLOW of -->C
FOLLOW(C) = { d }
Press 1 to continue2

-----
Process exited after 41.19 seconds with return value 2
Press any key to continue . . .
```

Lab 7:

Date: 2080/02/22

Write a program to implement LL (1) Parser.

Introduction:

LL(1) parsing is a top-down parsing technique used in the field of formal language theory and compiler design. It is widely employed in the analysis and translation of programming languages. The term "LL" stands for "Left-to-right, Leftmost derivation," indicating the order in which the parsing process proceeds.

In LL(1) parsing, the parsing algorithm starts from the leftmost symbol of the input string and proceeds in a left-to-right fashion. It aims to construct a leftmost derivation of the input string by expanding nonterminal symbols based on the production rules defined in the grammar.

The distinguishing feature of LL(1) parsing is the one-symbol lookahead property. This means that the parsing decision at each step is determined by examining the current nonterminal being expanded and the next symbol in the input string. By considering only one symbol ahead, the parsing process remains deterministic and avoids ambiguity.

To facilitate this parsing technique, a parsing table, often referred to as an LL(1) parsing table, is constructed based on the grammar of the language. The parsing table guides the parser in selecting the appropriate production rule to apply at each parsing step.

Production:

$$S \rightarrow E$$
$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid \text{id}$$

Code:

```
#include<stdio.h>
#include<string.h>
#include<process.h>
char s[20],stack[20];
int main()
{
    char m[5][6][4]={ "tb"," ","","tb"," "," "," "+tb"," "," "
    ", "n","n","fc"," "," ","fc"," "," ","n","*fc"," a","n","n","i"," ","
    ", "(e)"," "," "};
    int
size[5][6]={ 2,0,0,2,0,0,0,3,0,0,1,1,2,0,0,2,0,0,0,1,3,0,1,1,1,0,0,3,0,0};
    int i,j,k,n,str1,str2;
    printf("Enter the input string: ");
    scanf("%s",s);
    strcat(s,"$");
    n=strlen(s);
    stack[0]='$';
    stack[1]='e';
    i=1;
    j=0;
    printf("\nStack          Input\n");
    printf("_____ \n");
    while((stack[i]!='$')&&(s[j]!='$'))
    {
        if(stack[i]==s[j])
        {
            i--;
            j++;
        }
        switch(stack[i])
        {
            case 'e': str1=0;
            break;
            case 'b': str1=1;
            break;
            case 't': str1=2;
            break;
            case 'c': str1=3;
            break;
            case 'f': str1=4;
            break;

        }
    }
```

```

switch(s[j])
{
    case 'i': str2=0;
    break;
    case '+': str2=1;
    break;
    case '*': str2=2;
    break;
    case '(': str2=3;
    break;
    case ')': str2=4;
    break;
    case '$': str2=5;
    break;
}
if(m[str1][str2][0]=='\0')
{
    printf("\nERROR");
    exit(0);
}
else if(m[str1][str2][0]=='n')
    i--;
else if(m[str1][str2][0]=='i')
    stack[i]='i';
else
{
    for(k=size[str1][str2]-1;k>=0;k--)
    {
        stack[i]=m[str1][str2][k];
        i++;
    }
    i--;
}
for(k=0;k<=i;k++)
    printf("%c",stack[k]);
printf(" ");
for(k=j;k<=n;k++)
    printf("%c",s[k]);
printf(" \n ");
}
printf("\nAccept");
return 0;
}

```

Output:

```
Select D:\Documents\Utsav\Compiler\Lab 7.exe
Enter the input string: id+id*id

Stack      Input
-----
$bt        id+id*id$
$bcf       id+id*id$
$bci       id+id*id$
$b         d+id*id$
$          d+id*id$

Accept
-----
Process exited after 4.456 seconds with return value 0
Press any key to continue . . .
```

Lab 8:**Date: 2080/02/23****Write a program to implement LL (1) Parser.**

Introduction:

LR(0) parsing is a method used in compiler design and formal language theory to analyze and understand the structure of a programming language. It works from the input string towards the root of the parse tree. The process involves using a table and a stack to determine the appropriate actions to take at each step.

The parsing table is created based on the grammar rules of the language being parsed. This table helps guide the parsing process by telling us what action to take (either shifting or reducing) or which state to transition to based on the current state and the next symbol in the input.

The "LR" in LR(0) means that we start parsing from the leftmost symbol of the input and continue in a left-to-right manner. We build the rightmost derivation of the parse tree in reverse, starting from the bottom. The "0" in LR(0) refers to the fact that we don't look ahead at the next input symbol when making parsing decisions. We only consider the current state to determine the next action.

inputfile.txt Contains:

S S+T

S T

T T*F

T F

F (S)

F t

Code:

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

int i,j,k,m,n=0,o,p,ns=0,tn=0,rr=0,ch=0;
char
read[15][10],gl[15],gr[15][10],temp,templ[15],tempr[15][10],*ptr,temp
p2[5],dfa[15][15];

struct states
{
    char lhs[15],rhs[15][10];
    int n;
}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]=NULL;
        strcat(I[ns].rhs[I[ns].n],gr[s]);
        I[ns].lhs[I[ns].n]=gl[s];
        I[ns].lhs[I[ns].n+1]=NULL;
        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]=NULL;

        for(j=0;j<rr1;j++)
            if( strcmp(temp2,read1[j])==0 )

```



```

        break;
    if(j==rr1)
    {
        strcpy(read1[rr1],temp2);
        read1[rr1][2]=NULL;
        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]=NULL;
            strcpy(temp1[tn],I[l].rhs[j]);
            tn++;
        }
    }

    for(j=0;j<tn;j++)
    {
        ptr=strchr(temp1[j],'.');
        p=ptr-temp1[j];
        temp1[j][p]=temp1[j][p+1];
        temp1[j][p+1]='.';
        I[ns].lhs[I[ns].n]=templ[j];
        I[ns].lhs[I[ns].n+1]=NULL;
        strcpy(I[ns].rhs[I[ns].n],temp1[j]);
        I[ns].n++;
    }

    moreprod();
    for(j=0;j<ns;j++)
    {
        if( compstruct(I[ns],I[j])==1 )
        {
            I[ns].lhs[0]=NULL;
            for(k=0;k<I[ns].n;k++)
                I[ns].rhs[k][0]=NULL;
            I[ns].n=0;
            dfa[l][j]=temp2[1];

```

```

        break;
    }
}
if(j<ns)
{
    tn=0;
    for(j=0;j<15;j++)
    {
        templ[j]=NULL;
        tempr[j][0]=NULL;
    }
    continue;
}

dfa[l][j]=temp2[1];
printf("\n\nI%d :",ns);
for(j=0;j<I[ns].n;j++)
    printf("\n%c -> %s",I[ns].lhs[j],I[ns].rhs[j]);
getch();
ns++;
tn=0;
for(j=0;j<15;j++)
{
    templ[j]=NULL;
    tempr[j][0]=NULL;
}
}
}

void main()
{
    FILE *f;
    int l;

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

    f=fopen("inputfile.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("%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==read[j][0] )
            break;
    if(j==rr)
    {
        read[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("%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 DFA TABLE");
getch();

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

Output:

```
Select D:\Documents\Utsav\Compiler\Lab 8.exe
THE GRAMMAR IS AS FOLLOWS
S -> S+T
S -> T
T -> T*F
T -> F
F -> (S)
F -> t

I0 :
Z -> .S
S -> .S+T
S -> .T
T -> .T*F
T -> .F
F -> .(S)
F -> .t

I1 :
Z -> S.
S -> S.+T

I2 :
S -> T.
T -> T.*F

I3 :
T -> F.

I4 :
F -> (.S)
S -> .S+T
S -> .T
T -> .T*F
T -> .F
F -> .(S)
F -> .t

I5 :
F -> t.

I6 :
S -> S+.T
T -> .T*F
T -> .F
F -> .(S)
F -> .t

I7 :
T -> T*.F
F -> .(S)
F -> .t

I8 :
F -> (S.)
S -> S.+T

I9 :
S -> S+T.
T -> T.*F

I10 :
T -> T*F.

I11 :
F -> (S).
```

```
PRESS ANY KEY FOR DFA TABLE I0 :  
  
'S' -> I1 |  
'T' -> I2 |  
'F' -> I3 |  
'(' -> I4 |  
't' -> I5 |  
  
I1 :  
'+' -> I6 |  
  
I2 :  
'*' -> I7 |  
  
I3 :  
  
I4 :  
'T' -> I2 |  
'F' -> I3 |  
'(' -> I4 |  
't' -> I5 |  
'S' -> I8 |  
  
I5 :  
  
Select D:\Documents\Utsav\Compiler\Lab 8.exe  
  
I6 :  
'F' -> I3 |  
'(' -> I4 |  
't' -> I5 |  
'T' -> I9 |  
  
I7 :  
'(' -> I4 |  
't' -> I5 |  
'F' -> I10 |  
  
I8 :  
'+' -> I6 |  
'-' -> I11 |  
  
I9 :  
'*' -> I7 |  
  
I10 :  
  
I11 :  
  
PRESS ANY KEY TO EXIT
```

Lab 9:

Date: 2080/02/24

Write a program to implement Shift Reduce Parser.

Introduction:

A shift-reduce parser is a technique used in compiler design and formal language theory to analyze the structure of a program or language expression. It works by scanning the input from left to right and making decisions on whether to shift input symbols onto a stack or reduce symbols on the stack based on a set of rules called the grammar.

Think of the parser as a worker who is reading a sentence word by word and trying to understand its meaning. The worker has an empty container (stack) in which they can store words temporarily. They also have a set of rules (grammar) that guide them on how to interpret the words and their relationships.

The parser starts with an empty stack and reads the input symbols (words) one by one. For each symbol, the parser decides whether to shift it onto the stack or to reduce a set of symbols on the stack based on the grammar rules. Shifting means putting the symbol on top of the stack, while reducing means replacing a group of symbols on the stack with a single symbol according to a grammar rule.

The parsing process continues until the parser either encounters an error or successfully constructs a valid structure (parse tree). An error can occur if there is a mismatch between the expected symbol and the actual symbol in the input, or if the parser cannot find a valid action to take based on the current state and input symbol combination. In such cases, the parser may report a syntax error and stop analyzing the input.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.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("SHIFT REDUCE PARSER\n");
    printf("\nGRAMMER\n");
    printf("\n1)E->E+E\n2)E->E/E");
    printf("\n3)E->E*E\n4)E->a/b");
    printf("\nenter the input symbol:\t");
    gets(ip_sym);
    printf("\n\tstack 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 %s \t\t %s",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 %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((!strcmpi(temp2,"a"))||(!strcmpi(temp2,"b")))
        {
            stack[st_ptr]='E';
            if(!strcmpi(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((!strcmpi(temp2,"+"))||(strcmpi(temp2,"*"))||(strcmpi(temp2,
"/"))))
        {
            flag=1;
        }
        if((!strcmpi(stack,"E+E"))||(strcmpi(stack,"E\E"))||(strcmpi(st
ack,"E*E")))
        {
            strcpy(stack, "E");
            st_ptr=0;
            if(!strcmpi(stack,"E+E"))
                printf("\n $%s\t\t%s$\t\t\tE->E+E", stack,
ip_sym);
            else
                if(!strcmpi(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(!strcmpi(stack,"E")&&ip_ptr==len)
        {
            printf("\n $%s\t\t%s$\t\t\tACCEPT",stack,ip_sym);
            getch();
            exit(0);
        }
    }

```

```
if(flag==0)
{
    printf("\n%s\t\t%s\t\t reject",stack,ip_sym);
    exit(0);
}
return;
}
```

Output:

```
Select D:\Documents\Utsav\Compiler\Lab 9.exe
SHIFT REDUCE PARSER

GRAMMER
1)E->E+E
2)E->E/E
3)E->E*E
4)E->a/b
enter the input symbol: a+b

      stack implementation table
stack      input symbol      action
-----
$          a+b$              --
$a         +b$              shift a
$E         +b$              E->a
$E+        b$              shift +
$E+b       $               shift b
$E+E       $               E->b
$E         $               E->E*E
$E         $               ACCEPT
-----
Process exited after 2.888 seconds with return value 0
Press any key to continue . . .
```

Lab 10:

Date: 2080/02/25

Write a program for intermediate code generator.

Introduction:

Intermediate code generation is a step in the compilation process where we create a simplified version of the source code. The purpose of this step is to make the later stages of compilation, like optimization and code generation for the target machine, easier.

Intermediate code acts as a bridge between the original high-level source code and the low-level target code. It is a representation of the source code that is more abstract and independent of a specific machine. The intermediate code captures the main meaning and structure of the source code while removing some of the language-specific details.

The process of generating intermediate code involves analyzing the source code and creating a representation that is easier to work with and optimize. This representation can take different forms, such as a tree-like structure called an abstract syntax tree, a simplified code representation with three-addresses, a set of instructions called quadruples, or a bytecode format.

In simpler terms, intermediate code generation is a way to transform the source code into a form that is easier to manipulate and optimize. It helps simplify the compilation process and prepares the code for further steps, such as making it faster and more efficient for the specific machine it will run on.

Code:

```
#include<stdio.h>
#include<string.h>
#include<process.h>
int i=1,j=0,no=0,tmpch=90;
char str[100],left[15],right[15];
void findopr();
void explore();
void fleft(int);
void fright(int);
struct exp
{
    int pos;
    char op;
}k[15];
int main()
{
    printf("Enter the Expression :");
    scanf("%s", str);
    printf("The intermediate code:\t\t Expression\n");
    findopr();
    explore();
    return 0;
}
void findopr()
{
    for(i=0;str[i]!='\0';i++)
        if(str[i]==':')
        {
            k[j].pos=i;
            k[j++].op=':';
        }
    for(i=0;str[i]!='\0';i++)
        if(str[i]=='/')
        {
            k[j].pos=i;
            k[j++].op='/';
        }
    for(i=0;str[i]!='\0';i++)
        if(str[i]=='*')
        {
            k[j].pos=i;
            k[j++].op='*';
        }
}
```

```

    }
    for(i=0;str[i]!='\0';i++)
        if(str[i]=='+')
        {
            k[j].pos=i;
            k[j++].op='+';
        }
    for(i=0;str[i]!='\0';i++)
    {
        if(str[i]=='-')
        {
            k[j].pos=i;
            k[j++].op='-';
        }
    }
}
void explore()
{
    i=1;
    while(k[i].op!='\0')
    {
        fleft(k[i].pos);
        fright(k[i].pos);
        str[k[i].pos]=tmpch--;
        printf("\t%c := %s%c%s\t\t", str[k[i].pos], left, k[i].op,
right);
        for(j=0;j <strlen(str);j++)
            if(str[j]!='$')
                printf("%c", str[j]);
        printf("\n");
        i++;
    }
    fright(-1);
    if(no==0)
    {
        fleft(strlen(str));
        printf("\t%s := %s", right, left);
        exit(0);
    }
    printf("\t%s := %c", right, str[k[--i].pos]);
}
void fleft(int x)
{
    int w=0, flag=0;

```

```

        x--;
        while(x!= -1 &&str[x]!='+'
&&str[x]!='*'&&str[x]!='='&&str[x]!='\0'&&str[x]!='-'&& str[x]!='/'
&& str[x]!=':')
        {
            if(str[x]!='$'&& flag==0)
            {
                left[w++]=str[x];
                left[w]='\0';
                str[x]='$';
                flag=1;
            }
            x--;
        }
    }
void fright(int x)
{
    int w=0,flag=0;
    x++;
    while(x!= -1 && str[x]!='+
'&&str[x]!='*'&&str[x]!='\0'&&str[x]!='='&&str[x]!=':'&& str[x]!='-
'&& str[x]!='/')
    {
        if(str[x]!='$'&& flag==0)
        {
            right[w++]=str[x];
            right[w]='\0';
            str[x]='$';
            flag=1;
        }
        x++;
    }
}

```

Output:

```
Select D:\Documents\Utsav\Compiler\Lab 10.exe
Enter the Expression :a+b+c-d-e
The intermediate code:      Expression
      Z := b+c              a+Z-d-e
      Y := Z-d              a+Y-e
      X := Y-e              a+X
      a := X
-----
Process exited after 4.111 seconds with return value 0
Press any key to continue . . .
```


Lab 11:

Date: 2080/02/26

Write a program for final code generator.

Introduction:

Final code generation is a crucial phase in the compilation process where the optimized form of the code is generated. The input for this phase is the intermediate code, which represents the essential semantics and structure of the source code. The goal of final code generation is to produce assembly code that carries out the operations defined in the intermediate code.

The process of final code generation involves several steps. Firstly, the intermediate code is analyzed to identify optimization opportunities. Various optimization techniques can be applied at this stage to improve the efficiency and performance of the generated code. Common optimization techniques include constant folding, dead code elimination, register allocation, and loop optimization. These optimizations aim to minimize the number of instructions, reduce redundant computations, and utilize hardware resources effectively.

Once the intermediate code has been optimized, the next step is to generate assembly code. Assembly code is a low-level representation of the program that can be directly executed by the target machine. It consists of mnemonic instructions that correspond to specific operations supported by the machine architecture. The assembly code is designed to efficiently execute the operations defined in the intermediate code.

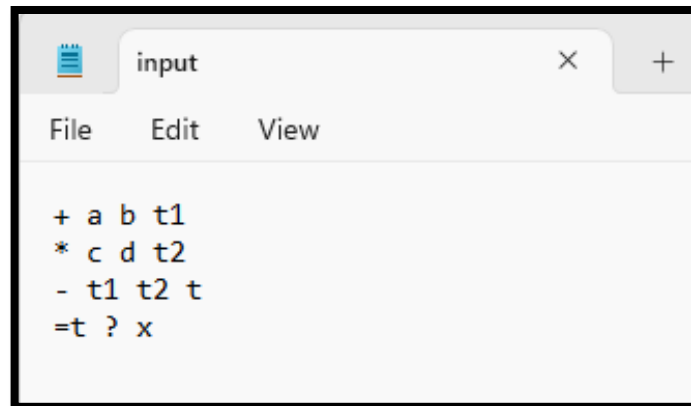
During the code generation process, the optimized intermediate code is translated into a sequence of assembly instructions. This involves mapping high-level constructs to their corresponding assembly instructions, managing memory and register allocation, and handling control flow constructs such as conditionals and loops. The generated assembly code should be semantically equivalent to the original program and correctly implement the operations specified in the intermediate code.

Code:

```
#include<stdio.h>
#include<string.h>
char op[2],arg1[5],arg2[5],result[5];
int main()
{
    FILE *fp1,*fp2;
    fp1=fopen("input.txt","r");
    fp2=fopen("output.txt","w");
    while(!feof(fp1))
    {
        fscanf(fp1,"%s%s%s%s",op,arg1,arg2,result);
        if(strcmp(op,"+")==0)
        {
            fprintf(fp2,"\n MOV R0,%s",arg1);
            fprintf(fp2,"\n ADD R0,%s",arg2);
            fprintf(fp2,"\n MOV %s,R0",result);
        }
        if(strcmp(op,"*")==0)
        {
            fprintf(fp2,"\n MOV R0,%s",arg1);
            fprintf(fp2,"\n MUL R0,%s",arg2);
            fprintf(fp2,"\n MOV %s, R0",result);
        }
        if(strcmp(op,"-")==0)
        {
            fprintf(fp2,"\n MOV R0,%s",arg1);
            fprintf(fp2,"\n SUB R0,%s",arg2);
            fprintf(fp2,"\n MOV %s,R0",result);
        }
        if(strcmp(op,"/")==0)
        {
            fprintf(fp2,"\n MOV R0,%s",arg1);
            fprintf(fp2,"\n DIV R0,%s",arg2);
            fprintf(fp2,"\n MOV %s,R0",result);
        }
        if(strcmp(op,"")==0)
        {
            fprintf(fp2,"\n MOV R0,%s",arg1);
            fprintf(fp2,"\n MOV %s,R0",result);
        }
    }
    fclose(fp1);fclose(fp2);return 0;}
```

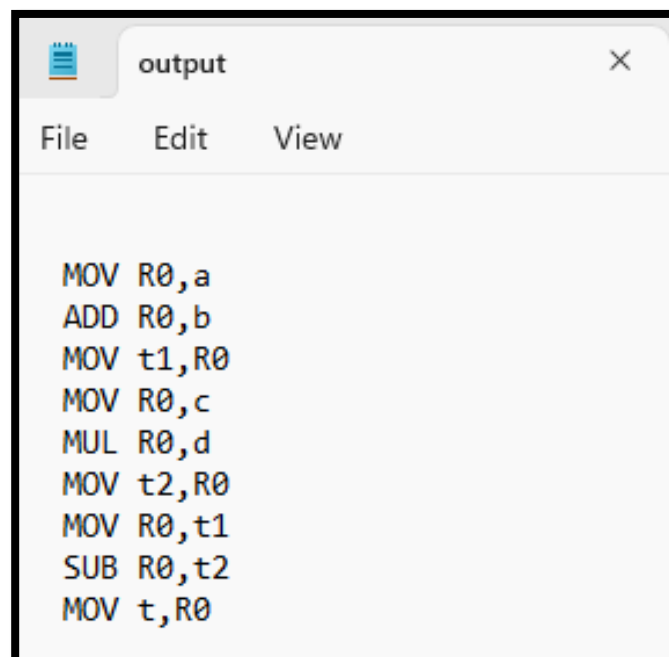
Output:

input.txt:



```
+ a b t1
* c d t2
- t1 t2 t
=t ? x
```

output.txt



```
MOV R0,a
ADD R0,b
MOV t1,R0
MOV R0,c
MUL R0,d
MOV t2,R0
MOV R0,t1
SUB R0,t2
MOV t,R0
```