Lab 1:

Write a program to implement DFA that accepts string ending with ab.

Introduction:

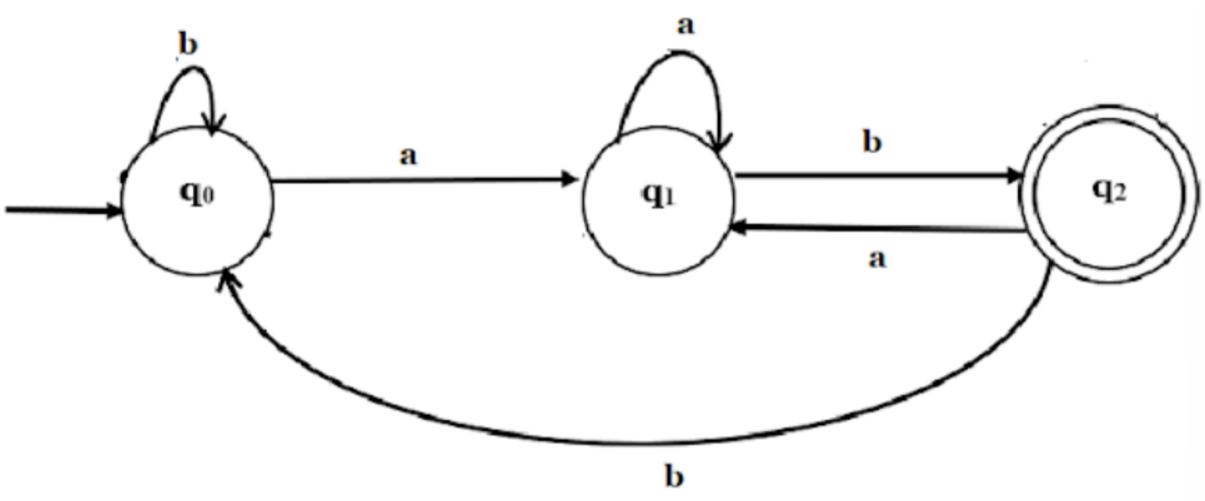


Fig: Transition diagram

States	**a**	**b**
➔ q0	q1	q0
q1	q1	q2
*q2	q1	q0

Initial State: q0

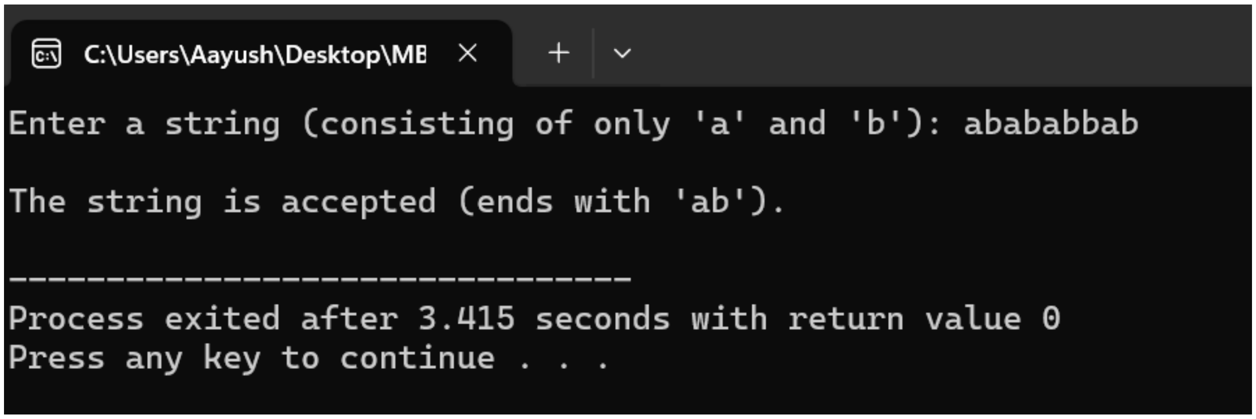
Final State: q2

Input: a b

Code:

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


enum State { q0, q1, q2 };

int main() {
    char input[100];
    enum State currentState = q0;
    int i;

    printf("Enter a string (consisting of only 'a' and 'b'): ");
    scanf("%s", input);

    for (i = 0; i < strlen(input); i++) {
        char symbol = input[i];

        switch (currentState) {
            case q0:
                if (symbol == 'a')
                    currentState = q1;
                else if (symbol == 'b')
                    currentState = q0;
                else {
                    printf("Invalid character: %c\n", symbol);
                    return 1;
                }
                break;

            case q1:
                if (symbol == 'a')
                    currentState = q1;
                else if (symbol == 'b')
                    currentState = q2;
                else {
                    printf("Invalid character: %c\n", symbol);
                    return 1;
                }
                break;

            case q2:
                if (symbol == 'a')
                    currentState = q1;
                else if (symbol == 'b')
                    currentState = q0;
                    else {
                        printf("Invalid character: %c\n", symbol);
                        return 1;
                    }
                    break;

                case q2:
                    if (symbol == 'a')
                        currentState = q1;
                    else if (symbol == 'b')
                        currentState = q0;


            printf("\nThe string is rejected (does not end with 'ab').\n");

        return 0;
    }
```

Output:



```
C:\Users\Aayush\Desktop\MB

Enter a string (consisting of only 'a' and 'b'): abababbab

The string is accepted (ends with 'ab').

--------------------------------
Process exited after 3.415 seconds with return value 0
Press any key to continue . . .
```

Lab 2:

Write a program for comment validation.

Introduction:

In the world of programming, comments act as the silent guides embedded within the code. They don’t influence how a program runs or behaves, but they play a vital role in communicating the intent, logic, and structure of the code to human readers.

Whether it’s a short explanation, a note to self, or documentation for a future collaborator, comments ensure that code remains understandable, maintainable, and meaningful.

Comment validation refers to the process of examining and ensuring that these in-code notes follow the expected norms and guidelines. It goes beyond just checking for syntax—it’s about verifying that the comments genuinely serve their purpose in enhancing code readability and comprehension.

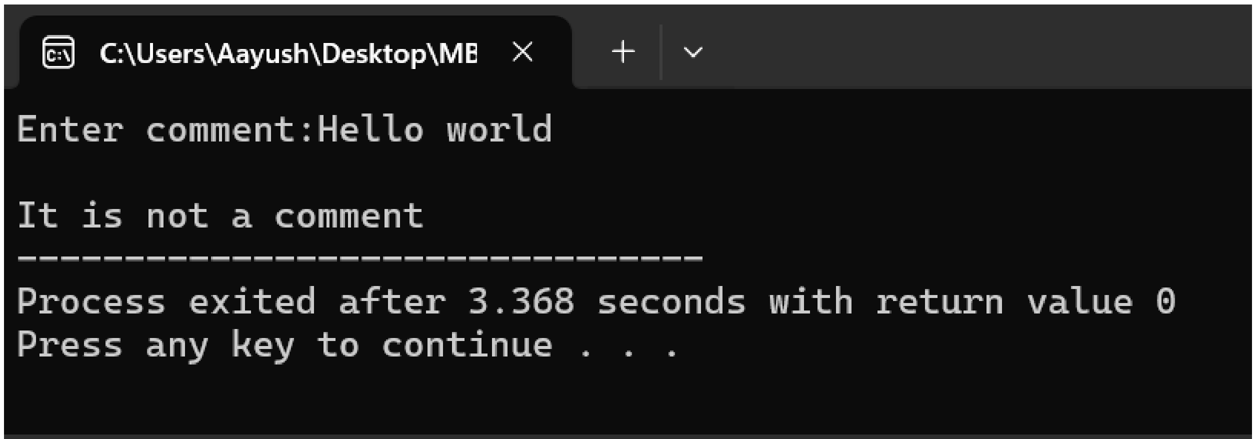
Comment validation can include various aspects, such as:

1. Proper Syntax and Structure: 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. Relevant 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. Meaningful 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. Stylistic 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.

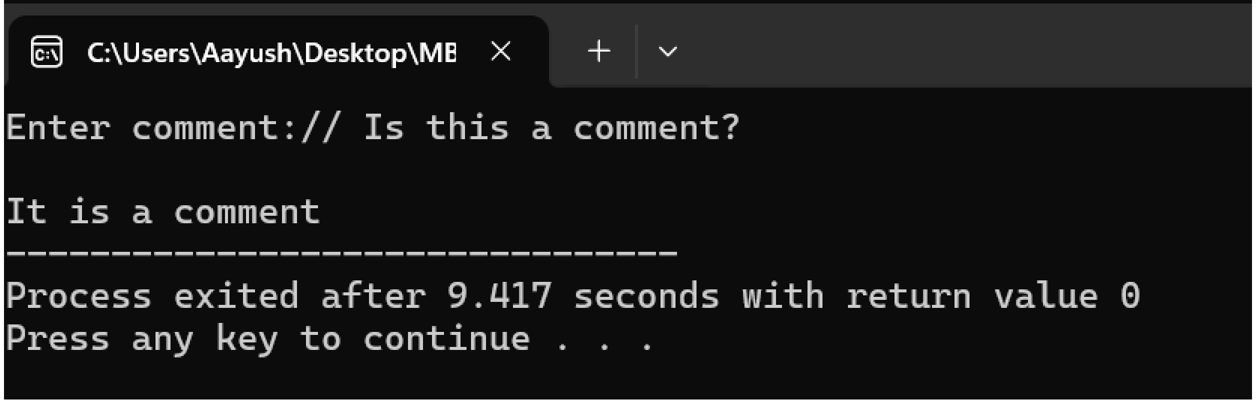
Code:

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

Output:



```
Enter comment:Hello world

It is not a comment
--------------------------------
Process exited after 3.368 seconds with return value 0
Press any key to continue . . .
```



```
Enter comment:// Is this a comment?

It is a comment
--------------------------------
Process exited after 9.417 seconds with return value 0
Press any key to continue . . .
```

Lab 3:

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.

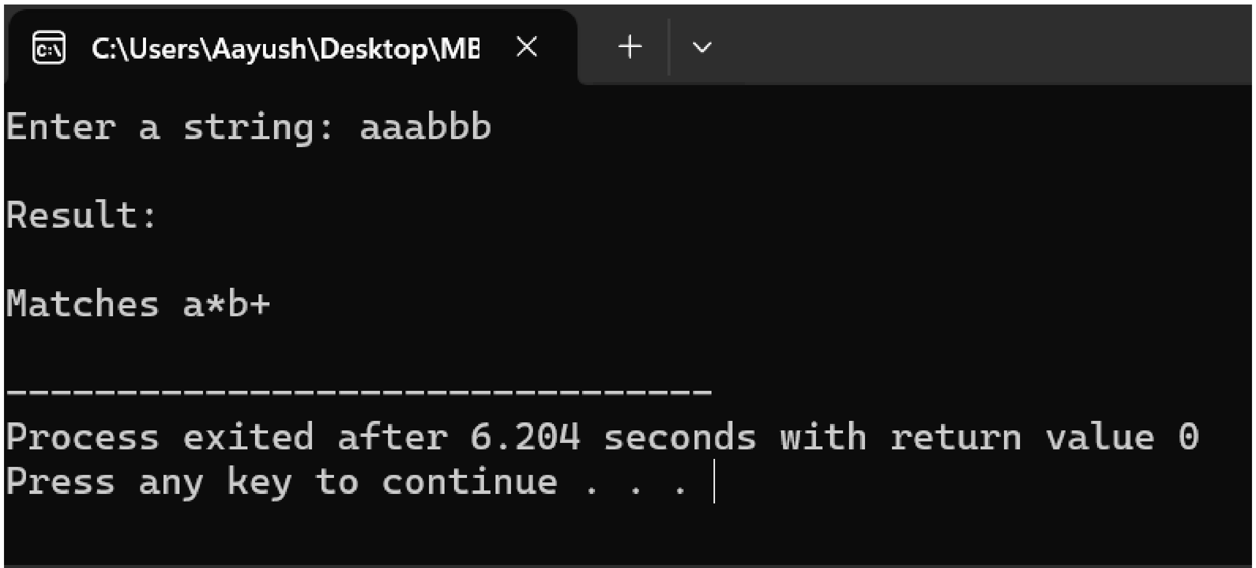
2. 'a*b+': This expression represents a sequence that starts with zero or more 'a' characters, followed by one or more 'b' characters. In other words, it matches any string that begins with 'a' (optional), followed by one or more 'b' characters. For example, it accepts strings like "b", "ab", "aab", "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.

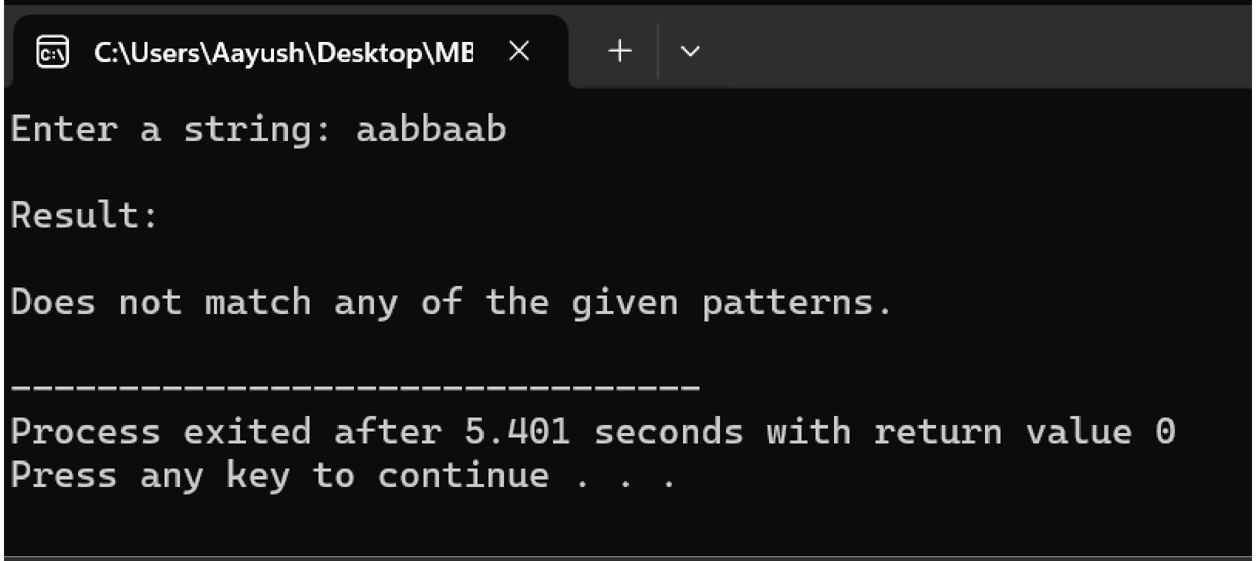
Code:

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

        // check if string matches a*
        bool isAStar(char str[]) {
                int i;
             for (i = 0; str[i]; i++) {
                 if (str[i] != 'a')
                      return false;
                      }
                        return true;
                     }

        // check if string matches a*b+
        bool isAStarBPlus(char str[]) {
                        int i = 0;

                        while (str[i] == 'a') i++;

                        int bCount = 0;

                        while (str[i] == 'b') {
                           bCount++;
                           i++;
                        }

                        return (str[i] == '\0' && bCount >= 1);
                     }

                     // check if string is exactly "abb"
                     bool isExactlyAbb(char str[]) {
                        return (strcmp(str, "abb") == 0);
                     }

                     int main() {
                        char input[100];
                        printf("Enter a string: ");
                        scanf("%s", input);
bool matchAStar = isAStar(input);
    bool matchAStarBPlus = isAStarBPlus(input);
    bool matchExactlyAbb = isExactlyAbb(input);

    printf("\nResult:\n");
    if (matchAStar)
        printf("\nMatches a*\n");
    if (matchAStarBPlus)
        printf("\nMatches a*b+\n");
    if (matchExactlyAbb)
        printf("\nMatches abb\n");
    if (!matchAStar && !matchAStarBPlus && !matchExactlyAbb)
        printf("\nDoes not match any of the given patterns.\n");

    return 0;
}
```

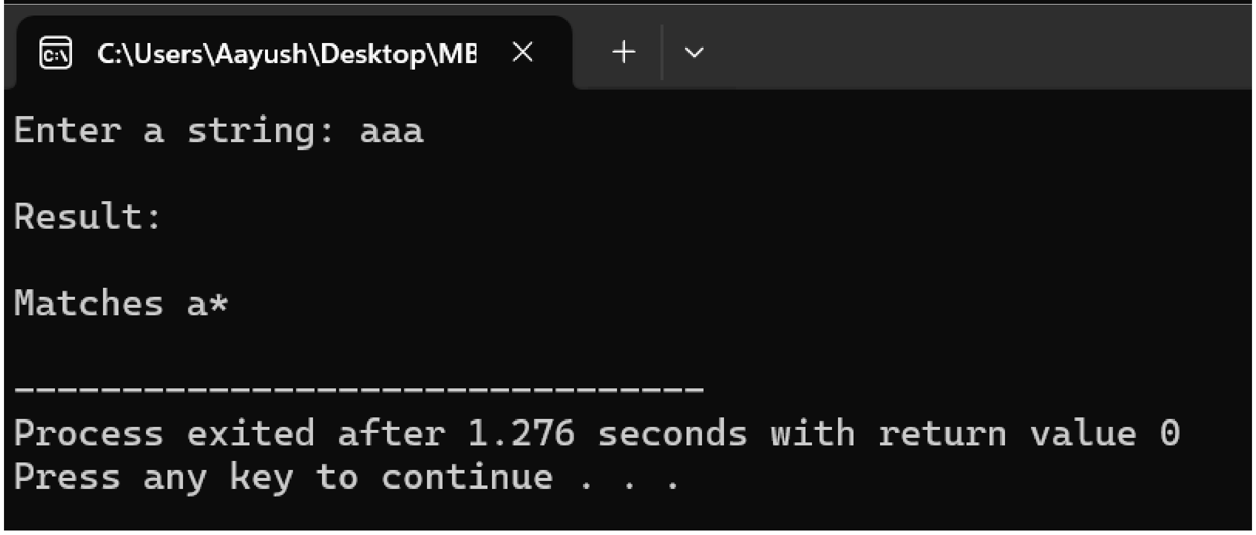
Output:



```
Enter a string: aaabbb

Result:

Matches a*b+

--------------------------------
Process exited after 6.204 seconds with return value 0
Press any key to continue . . .
```



```
Enter a string: aabbaab

Result:

Does not match any of the given patterns.

--------------------------------
Process exited after 5.401 seconds with return value 0
Press any key to continue . . .
```



```
Enter a string: aaa

Result:

Matches a*

--------------------------------
Process exited after 1.276 seconds with return value 0
Press any key to continue . . .
```

Lab 4:

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

Introduction:

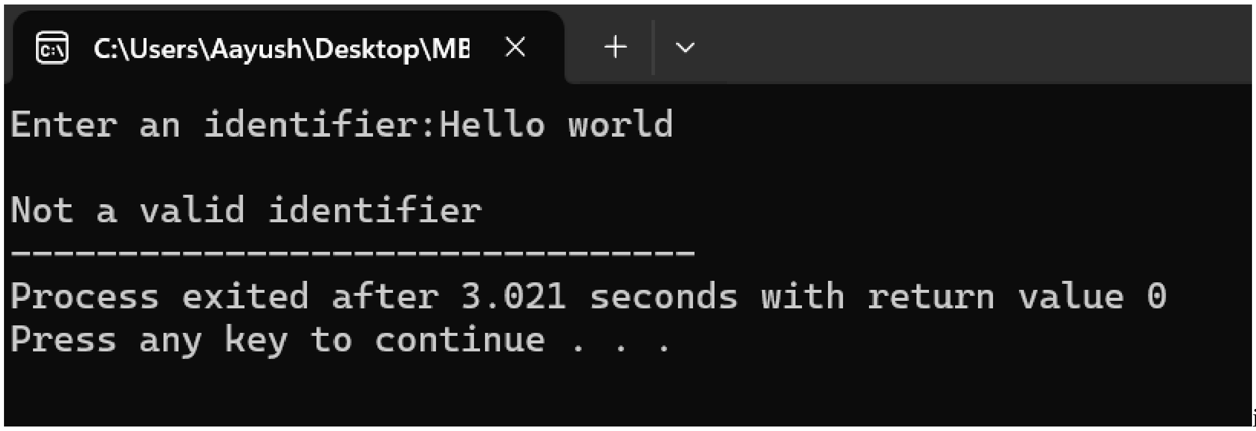
Identifiers in computer programming are like names given to different parts of a program, such as variables, functions, or classes. They help programmers understand and refer to these parts easily. Here are some simple rules for valid identifiers:

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;
}
```

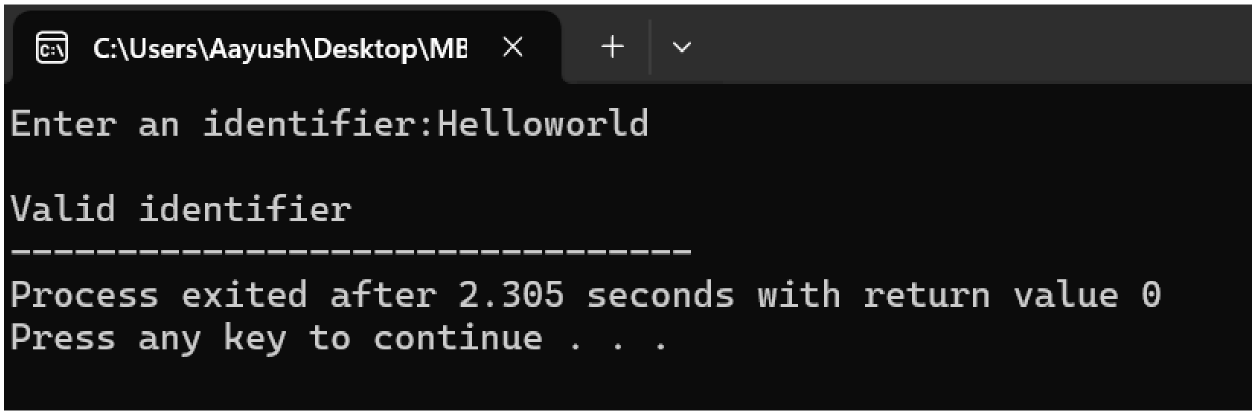
Output:



C:\Users\Aayush\Desktop\MB

```
Enter an identifier:Hello world

Not a valid identifier
--------------------------------
Process exited after 3.021 seconds with return value 0
Press any key to continue . . .
```

id



C:\Users\Aayush\Desktop\MB

```
Enter an identifier:Helloworld

Valid identifier
--------------------------------
Process exited after 2.305 seconds with return value 0
Press any key to continue . . .
```

Lab 5:

Write a program for Lexical Analyzer.

Introduction:

A lexical analyzer, also called a lexer or scanner, is a fundamental part of a compiler that performs the first phase of the compilation process. Its primary job is to read the source code (the code written by the programmer) and break it down into meaningful sequences called tokens.

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:

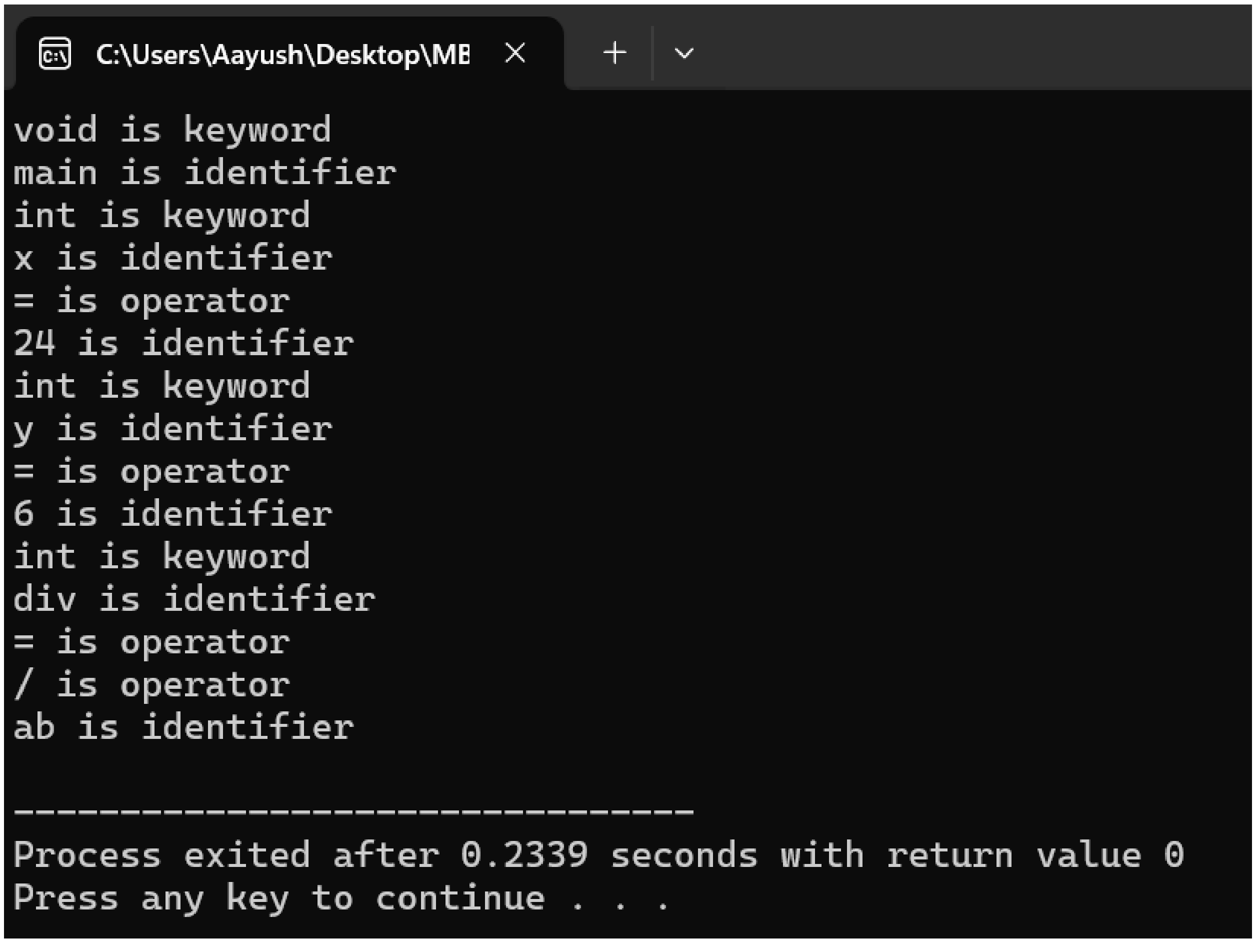
```
void main()
{
        int x = 24;
        int y = 6;
        int div = a/b;
}
```

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:



```
void is keyword
main is identifier
int is keyword
x is identifier
= is operator
24 is identifier
int is keyword
y is identifier
= is operator
6 is identifier
int is keyword
div is identifier
= is operator
/ is operator
ab is identifier

--------------------------------
Process exited after 0.2339 seconds with return value 0
Press any key to continue . . .
```

Lab 6:

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():

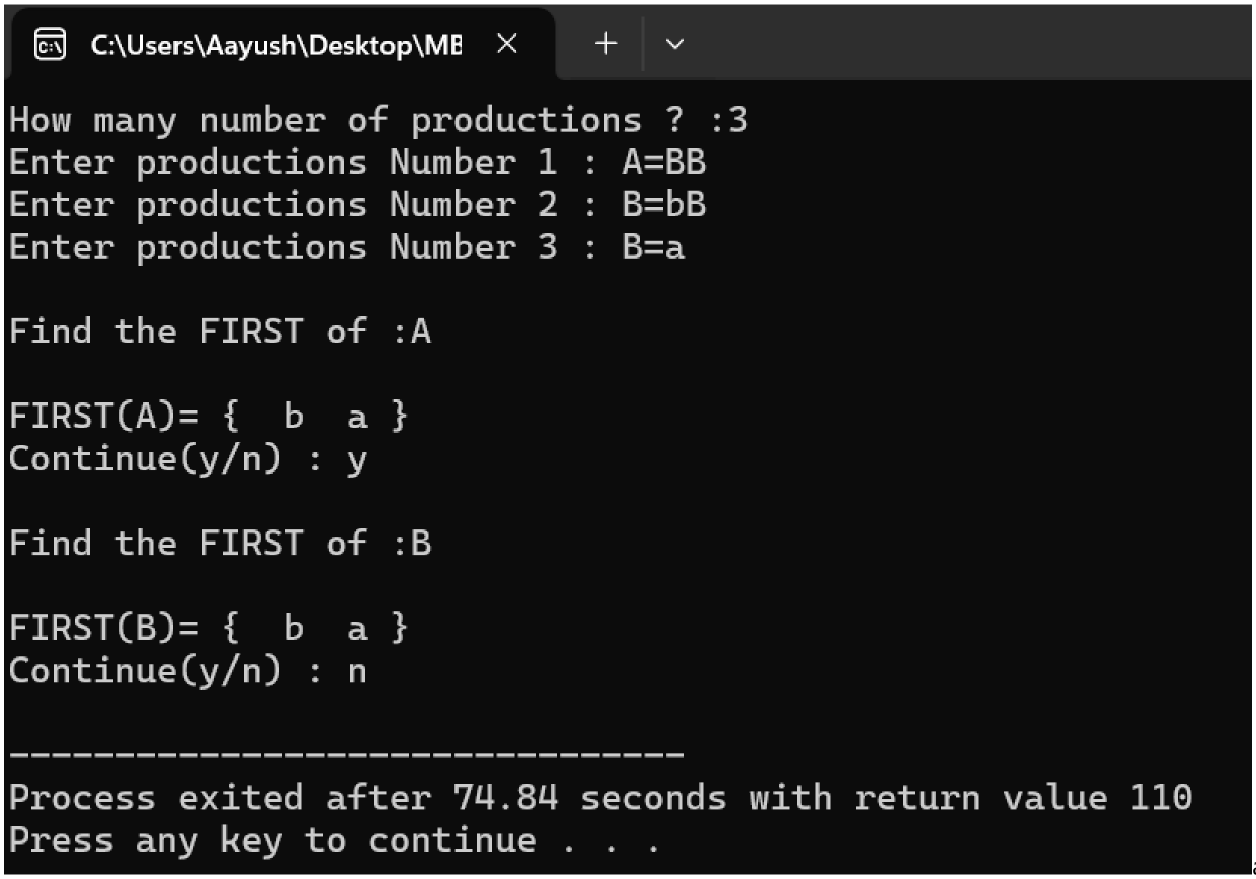
followpos() is another function used in compiler design and syntax analysis, particularly in the construction of the syntax tree or parse tree. It is associated with a node in the syntax tree and represents the set of positions (or indexes) in the input string where the next character following the construct 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:



```
How many number of productions ? :3
Enter productions Number 1 : A=BB
Enter productions Number 2 : B=bB
Enter productions Number 3 : B=a

Find the FIRST of :A

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

Find the FIRST of :B

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

--------------------------------
Process exited after 74.84 seconds with return value 110
Press any key to continue . . .
```

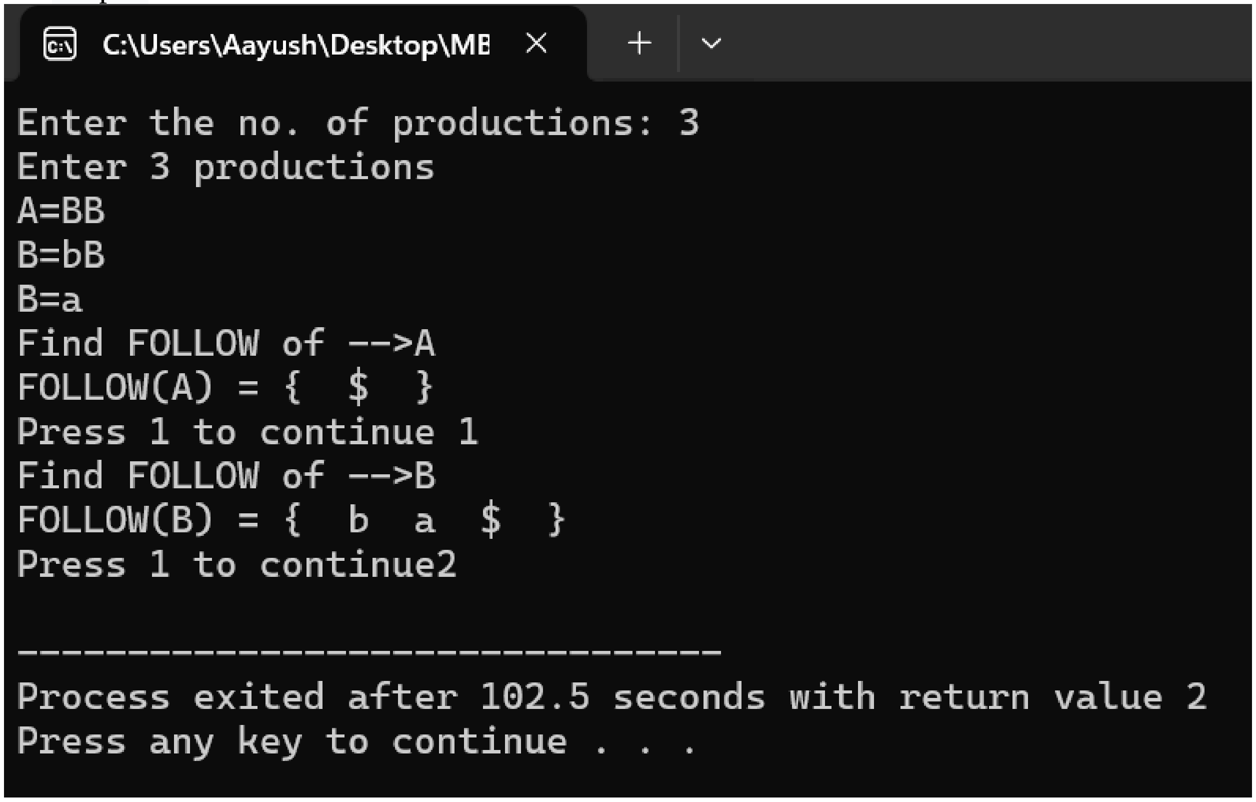
a

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:



C:\Users\Aayush\Desktop\MB

```
Enter the no. of productions: 3
Enter 3 productions
A=BB
B=bB
B=a
Find FOLLOW of -->A
FOLLOW(A) = {  $  }
Press 1 to continue 1
Find FOLLOW of -->B
FOLLOW(B) = {  b  a  $  }
Press 1 to continue2

--------------------------------
Process exited after 102.5 seconds with return value 2
Press any key to continue . . .
```

Lab 7:

Write a program to implement LL (1) Parser.

Introduction:

LL(1) parsing is a top-down parsing technique used in compiler design. It reads the input from Left to right (the first L) and constructs a Leftmost derivation (the second L) of the sentence. The 1 indicates that it uses one lookahead symbol at a time to make parsing decisions.

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 → E

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

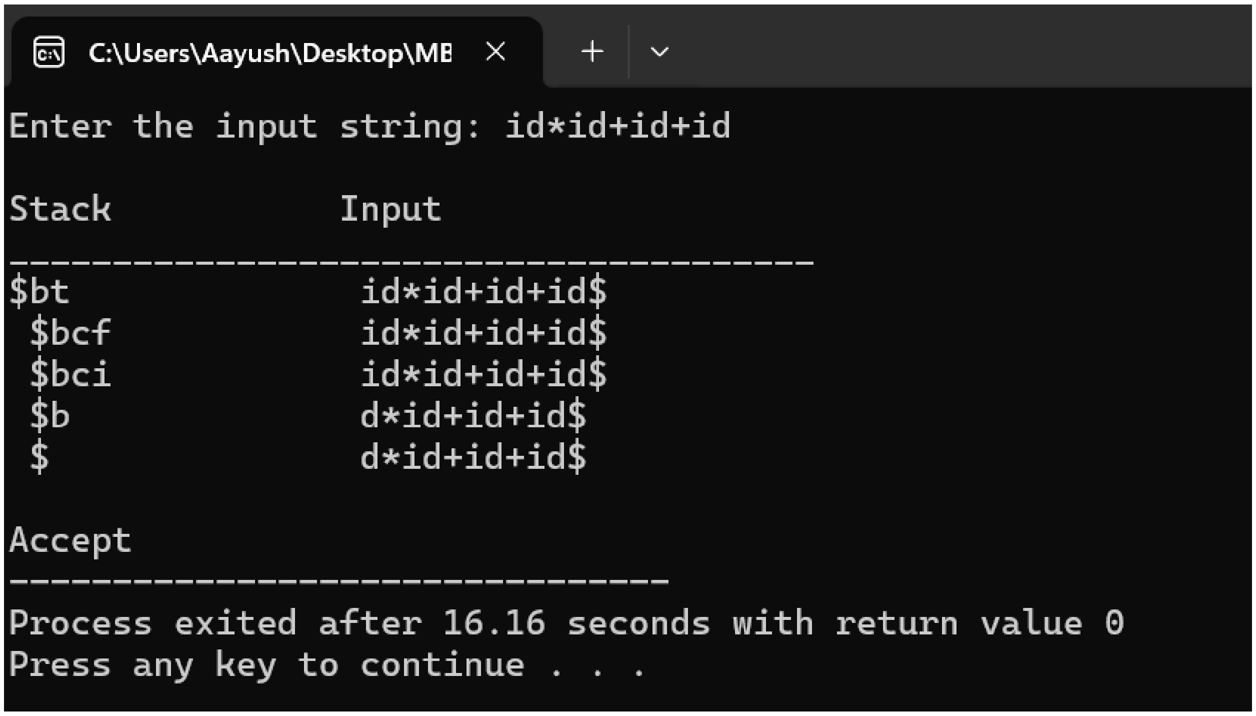
F → (E) | id

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:



```
Enter the input string: id*id+id+id

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

Accept
--------------------------------
Process exited after 16.16 seconds with return value 0
Press any key to continue . . .
```

Lab 8:

Write a program to implement LR (0) 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.

Textfile2.txt Contains:

```
S S+T
S T
T T*F
T F
F (S)
F t
```

Code:

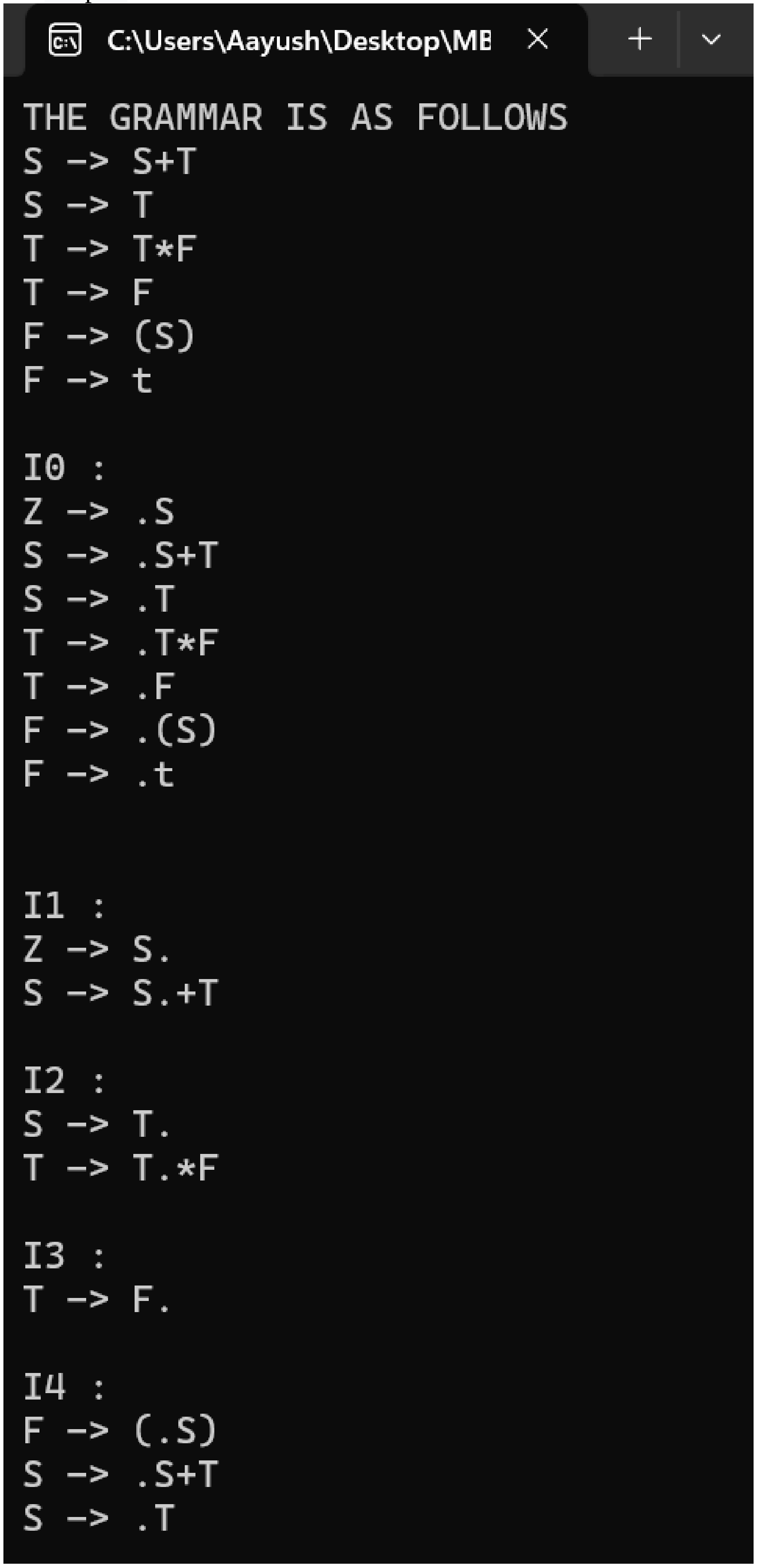
```
#include<stdio.h>
#include<string.h>

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,tem
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(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]=NULL;
            strcpy(I[ns].rhs[I[ns].n],tempr[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]);
```

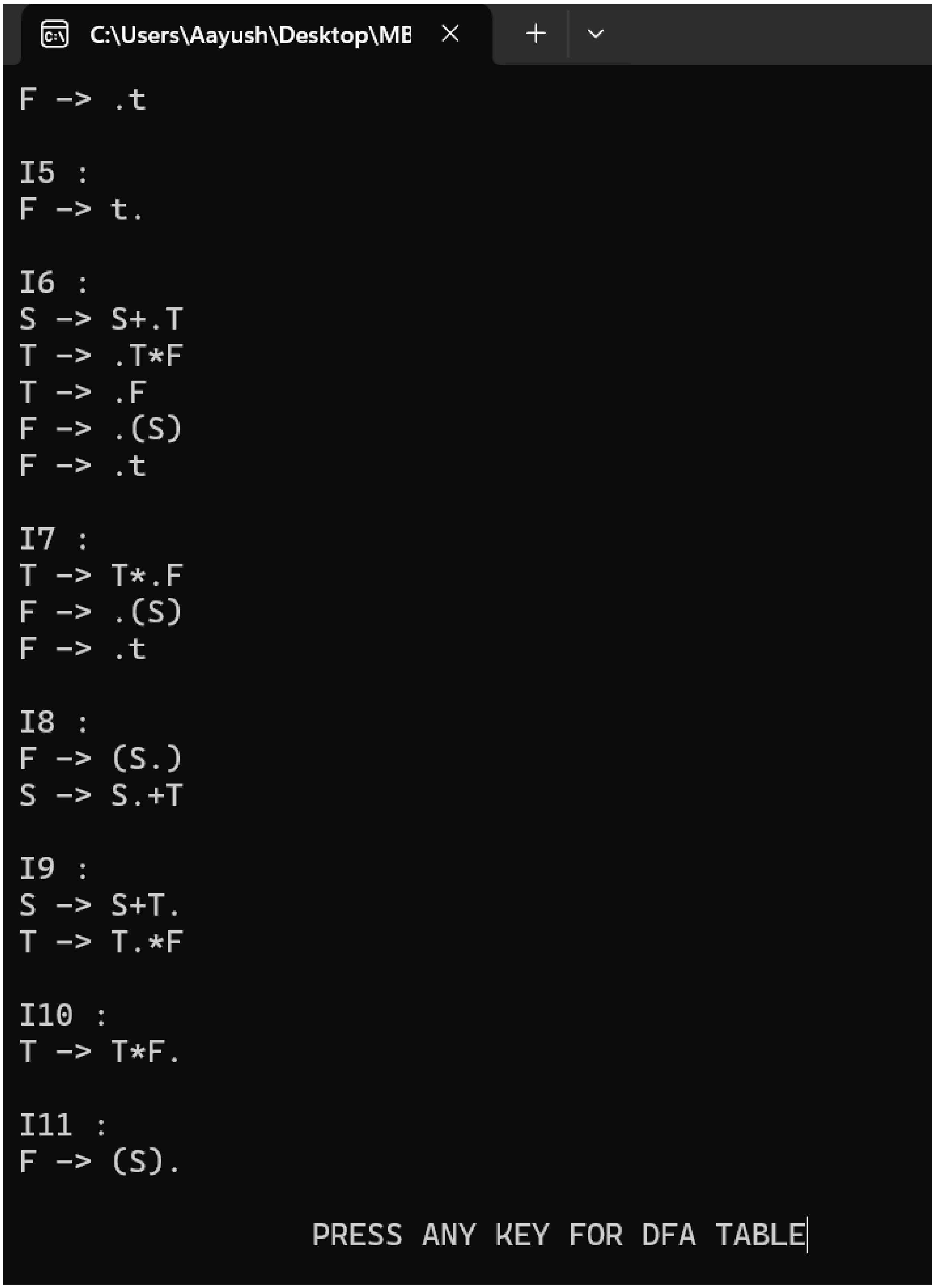
```
    for(l=0;l<ns;l++)
        canonical(l);

    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\n");
    }
    printf("\n\n\n\t\tPRESS ANY KEY TO EXIT");
    getch();
}
```

Output:



C:\Users\Aayush\Desktop\MB

```
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
```



```
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
```

Lab 9:

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 is a bottom-up technique used in syntax analysis, where the goal is to create a parse tree for a given input based on grammar rules. The process works by reading a stream of tokens (the input), the then working backwards through the grammar rules to discover how the input can be generated.

1. Input Buffer: This stores the string or sequence of tokens that needs to be parsed.

2. Stack: The parser uses a stack to keep track of which symbols or parts of the parse it has already processed. As it processes the input, symbols are pushed onto and popped off the stack.

3. Parsing Table: Similar to a predictive parser, a parsing table helps the parser decide what action to take next.

Shift-reducing parsing works by processing the input left to right and gradually building up a parse tree by shifting tokens onto the stack and reducing them using grammar rules, until it reaches the start symbol of the grammar.

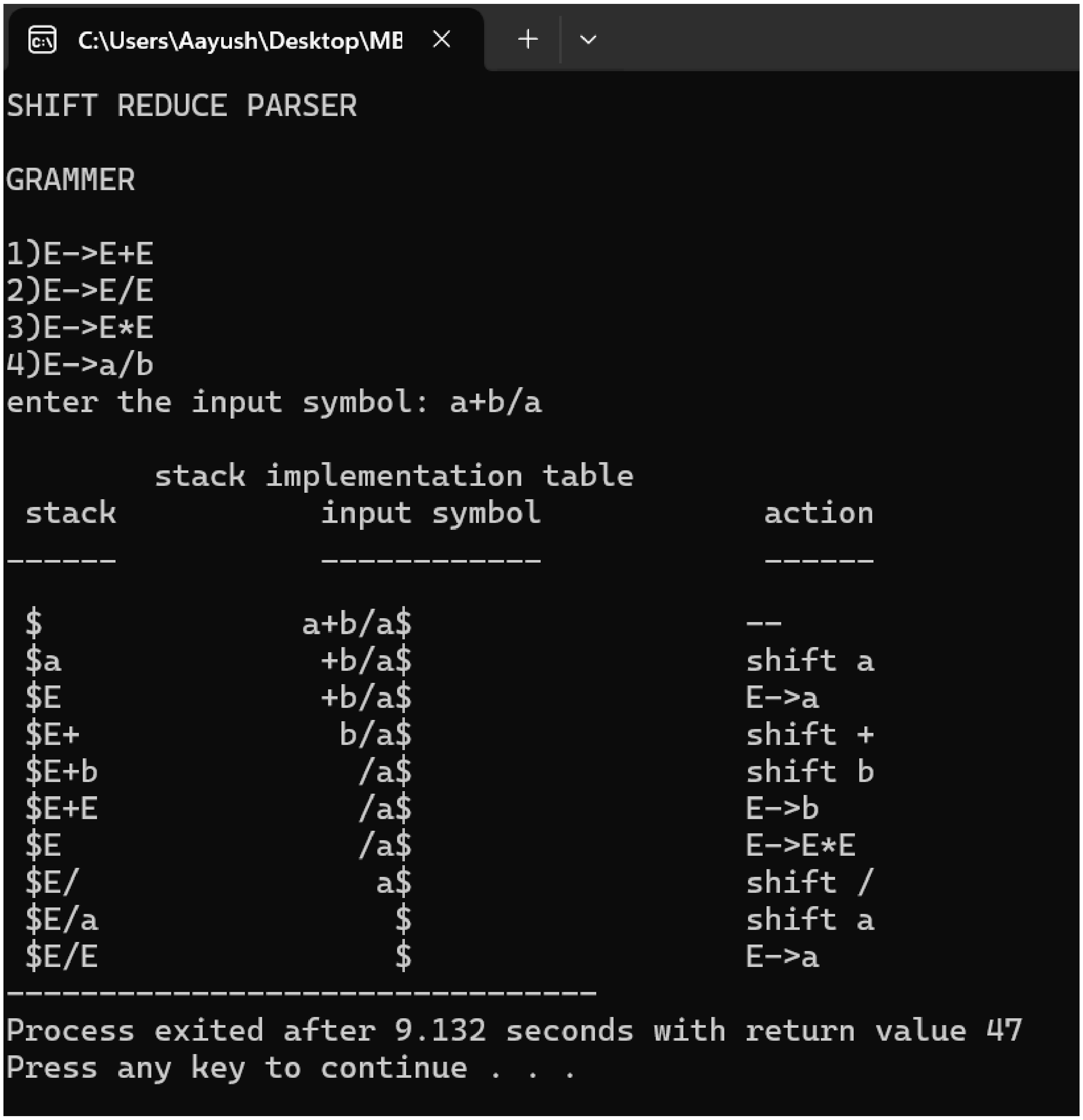
Code:

```
#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\t--",ip_sym);
        strcpy(act,"shift ");
        temp[0]=ip_sym[ip_ptr];
        temp[1]='\0';
        strcat(act,temp);
        len=strlen(ip_sym);
        for(i=0;i<=len-1;i++)
        {
                stack[st_ptr]=ip_sym[ip_ptr];
                stack[st_ptr+1]='\0';
                ip_sym[ip_ptr]=' ';
                ip_ptr++;
                printf("\n $%s\t\t%s$\t\t\t%s",stack,ip_sym,act);
                strcpy(act,"shift ");
                temp[0]=ip_sym[ip_ptr];
                temp[1]='\0';
                strcat(act,temp);
                check();
                st_ptr++;
        }
        st_ptr++;
```

```
		check();
	}
	void check()
	{
		int flag=0;
		temp2[0]=stack[st_ptr];
		temp2[1]='\0';
		if((!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\t%s\t\t reject",stack,ip_sym);
                exit(0);
        }
        return;
}
```

Output:



C:\Users\Aayush\Desktop\MB

```
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/a

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

 $              a+b/a$                  --
 $a              +b/a$                  shift a
 $E              +b/a$                  E->a
 $E+              b/a$                  shift +
 $E+b              /a$                  shift b
 $E+E              /a$                  E->b
 $E                /a$                  E->E*E
 $E/                a$                  shift /
 $E/a                $                  shift a
 $E/E                $                  E->a
--------------------------------
Process exited after 9.132 seconds with return value 47
Press any key to continue . . .
```

Lab 10:

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 primary advantage of generating intermediate code is portability. By using a machine-independent intermediate code, the same front end of the compiler can be used for different target machines, reducing the need for a full native compiler for each machine.

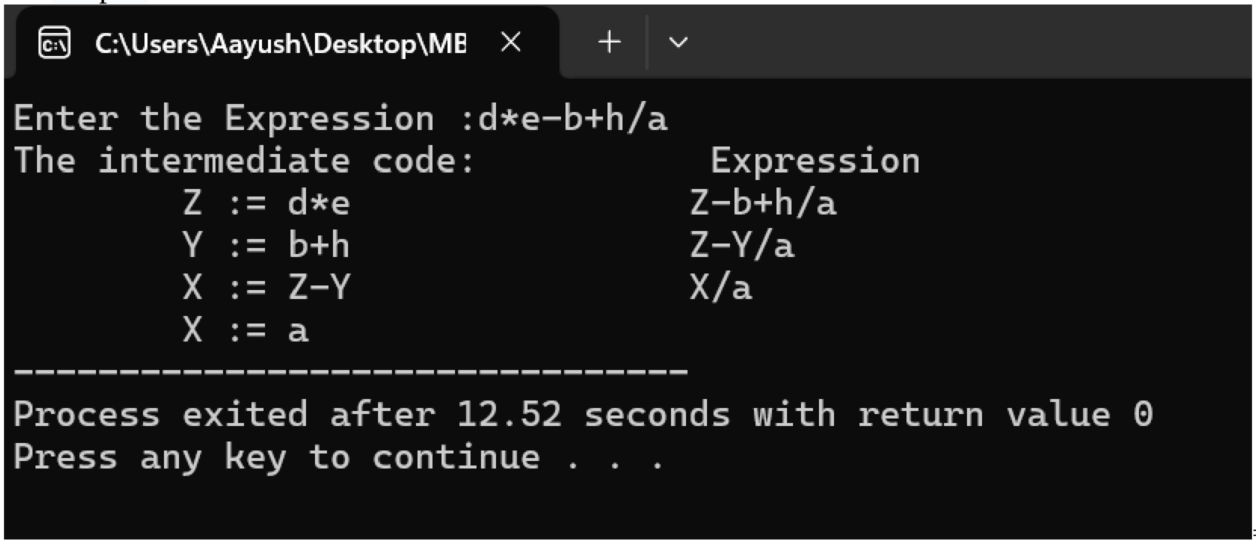
Another significant benefit is the ease of optimization. Intermediate code allows for various code optimization techniques to be applied, improving the performance and efficiency of the generated code. Additionally, intermediate code can be reused to generate code for other platforms or languages, enhancing code reuse and maintainability.

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:


C:\Users\Aayush\Desktop\MB

```
Enter the Expression :d*e-b+h/a
The intermediate code:          Expression
        Z := d*e                Z-b+h/a
        Y := b+h                Z-Y/a
        X := Z-Y                X/a
        X := a
--------------------------------
Process exited after 12.52 seconds with return value 0
Press any key to continue . . .
```

Lab 11:

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[3], arg1[10], arg2[10], result[10];

int main() {
    FILE *fp1, *fp2;

    fp1 = fopen("input.txt", "r");
    fp2 = fopen("output.txt", "w");

    if (fp1 == NULL || fp2 == NULL) {
        printf("Error opening file.\n");
        return 1;
    }
    while (fscanf(fp1, "%s%s%s%s", op, arg1, arg2, result) ==
4) {
        if (strcmp(op, "+") == 0) {
            fprintf(fp2, "MOV R0, %s\n", arg1);
            fprintf(fp2, "ADD R0, %s\n", arg2);
            fprintf(fp2, "MOV %s, R0\n", result);
        } else if (strcmp(op, "-") == 0) {
            fprintf(fp2, "MOV R0, %s\n", arg1);
            fprintf(fp2, "SUB R0, %s\n", arg2);
            fprintf(fp2, "MOV %s, R0\n", result);
        } else if (strcmp(op, "*") == 0) {
            fprintf(fp2, "MOV R0, %s\n", arg1);
            fprintf(fp2, "MUL R0, %s\n", arg2);
            fprintf(fp2, "MOV %s, R0\n", result);
        } else if (strcmp(op, "/") == 0) {
            fprintf(fp2, "MOV R0, %s\n", arg1);
            fprintf(fp2, "DIV R0, %s\n", arg2);
            fprintf(fp2, "MOV %s, R0\n", result);
        } else if (strcmp(op, "=") == 0) {
            fprintf(fp2, "MOV R0, %s\n", arg1);
            fprintf(fp2, "MOV %s, R0\n", result);
        }
    fclose(fp1);
    fclose(fp2);
    printf("Assembly code generated in output.txt\n");
    return 0;
}
```

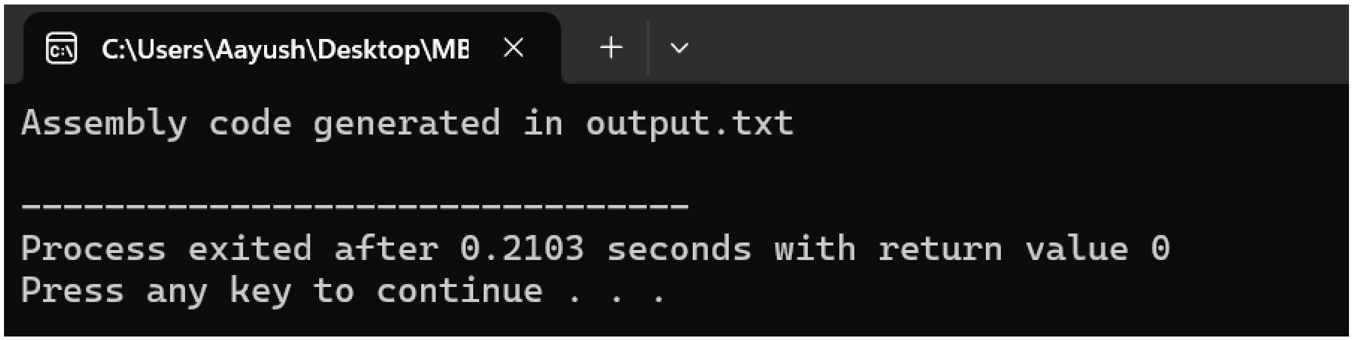
input.txt:

```
+ 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
MOV R0, t
MOV x, R0
```

Output



```
Assembly code generated in output.txt

--------------------------------
Process exited after 0.2103 seconds with return value 0
Press any key to continue . . .
```

Compiler Design and Construction

Lab Report

Submitted by

Aayush Basnet
BSc. CSIT | Batch of 2078
Symbol: 29131

Submitted To

Department of Computer Science and Information Technology
Madan Bhandari Memorial College
Baneshor, Kathmandu

Date:

Madan Bhandari Memorial College

Department of Computer Science and Information Technology (B.Sc.CSIT)

New Baneshor, Kathmandu

Practical Cover Page

Subject: - Symbol: -

Name:-	Semester :-	Batch :- 2078

SN	Topic	Date of Practical	Signature	Remarks