

# **DELHI TECHNOLOGICAL UNIVERSITY**



## **Compiler Design Lab File**

**Submitted by:**

Sameer Ahmed

2K19/CO/336

A5(G2)

**Submitted to:**

Dr. Pawan S. Mehra

# INDEX

S. No.	EXPERIMENT	Date
1	Write a program to convert Non-Deterministic Finite Automata (NFA) to Deterministic Finite Automata (DFA).	13/01/2022
2	Write a program for acceptance of string by DFA	20/01/2022
3	Write a program to find different tokens in a program	27/01/2022
4	To implement lexical analyser	03/02/2022
5	Write a program to implement recursive descent parser	24/02/2022
6	Write a program to implement left factoring.	10/03/2022
7	Write a program to convert left recursive grammar to right recursive grammar	24/03/2022
8	Write a program to compute <u>First</u> and <u>Follow</u> .	31/03/2022
9	Write a program to construct LL(1) parsing table	07/04/2022

# EXPERIMENT - 1

## AIM

Write a program to convert Non-Deterministic Finite Automata (NFA) to Deterministic Finite Automata (DFA).

## THEORY

### Deterministic Finite Automaton

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called **Deterministic Automaton**. As it has a finite number of states, the machine is called **Deterministic Finite Machine** or **Deterministic Finite Automaton**.

### Formal Definition of a DFA

A DFA can be represented by a 5-tuple  $(Q, \Sigma, \delta, q_0, F)$  where –

- $Q$  is a finite set of states.
- $\Sigma$  is a finite set of symbols called the alphabet.
- $\delta$  is the transition function where  $\delta: Q \times \Sigma \rightarrow Q$
- $q_0$  is the initial state from where any input is processed ( $q_0 \in Q$ ).
- $F$  is a set of final state/states of  $Q$  ( $F \subseteq Q$ ).

### Non-Deterministic Finite Automaton

In NDFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called **Non-deterministic Automaton**. As it has finite number of states, the machine is called **Non-deterministic Finite Machine** or **Non-deterministic Finite Automaton**.

### Formal Definition of an NDFA

An NDFA can be represented by a 5-tuple  $(Q, \Sigma, \delta, q_0, F)$  where –

- $Q$  is a finite set of states.
- $\Sigma$  is a finite set of symbols called the alphabets.

- $\delta$  is the transition function where  $\delta: Q \times \Sigma \rightarrow 2^Q$   
(Here the power set of  $Q$  ( $2^Q$ ) has been taken because in case of NFA, from a state, transition can occur to any combination of  $Q$  states)
- $q_0$  is the initial state from where any input is processed ( $q_0 \in Q$ ).
- $F$  is a set of final state/states of  $Q$  ( $F \subseteq Q$ ).

### Converting NFA to DFA

In NFA, when a specific input is given to the current state, the machine goes to multiple states. It can have zero, one or more than one move on a given input symbol. On the other hand, in DFA, when a specific input is given to the current state, the machine goes to only one state. DFA has only one move on a given input symbol.

Let,  $M = (Q, \Sigma, \delta, q_0, F)$  is an NFA which accepts the language  $L(M)$ . There should be equivalent DFA denoted by  $M' = (Q', \Sigma', q_0', \delta', F')$  such that  $L(M) = L(M')$ .

### Steps for converting NFA to DFA:

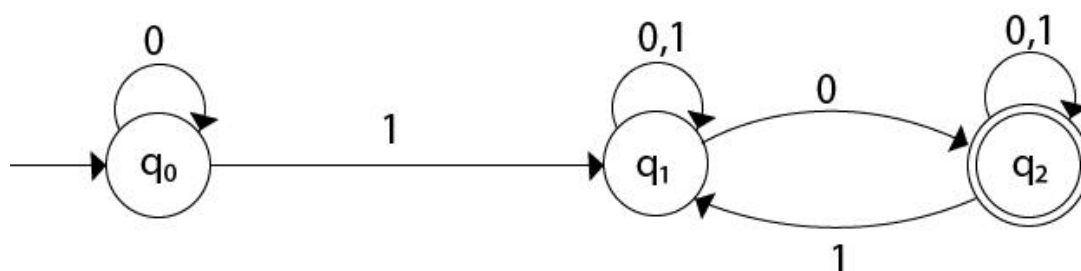
**Step 1:** Initially  $Q' = \phi$

**Step 2:** Add  $q_0$  of NFA to  $Q'$ . Then find the transitions from this start state.

**Step 3:** In  $Q'$ , find the possible set of states for each input symbol. If this set of states is not in  $Q'$ , then add it to  $Q'$ .

**Step 4:** In DFA, the final state will be all the states which contain  $F$  (final states of NFA)

### Example :



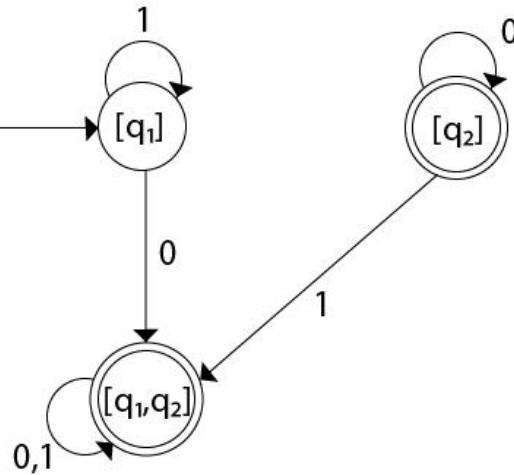
NFA Transition Table:

State	0	1
$\rightarrow q_0$	$q_0$	$q_1$
$q_1$	$\{q_1, q_2\}$	$q_1$
$*q_2$	$q_2$	$\{q_1, q_2\}$

DFA Transition Table :

State	0	1
$\rightarrow [q_0]$	$[q_0]$	$[q_1]$
$[q_1]$	$[q_1, q_2]$	$[q_1]$
$*[q_2]$	$[q_2]$	$[q_1, q_2]$
$*[q_1, q_2]$	$[q_1, q_2]$	$[q_1, q_2]$

DFA Transition Diagram



## CODE

```

#include<iostream>
#include<unordered_map>
#include<unordered_set>
#include<vector>
#include<queue>
#include<math.h>

using namespace std ;

string unionStates(string str, int alphabet, unordered_map<string, vector<string> >
map)
{
    string ans = "" ;

    for(int i = 0 ; i < str.length() ; i++)
    {
        string ch ;
        ch += str[i] ;
        ans += map[ch][alphabet] ;
    }
}

```

```

    return ans ;
}

int main()
{
    int states , alphabets ;

    cout << "Enter number of States : " << endl ;
    cin >> states ;

    // take input for states
    cout<<"\nEnter " <<states<<" states : \n";
    string sa[states] ;
    for (int i = 0; i < states; i++)
    {
        cin >> sa[i] ;
    }

    cout << "\nEnter number of inputs : " << endl ;
    cin >> alphabets ;

    unordered_map<string, vector<string> > map ;

    cout<<"\n\nEnter the transitions (Enter X for null) : \n";

    for (int s = 0; s < states; s++)
    {
        for(int a = 0 ; a < alphabets ; a++)
        {
            cout<<"\nState\tInput";
            cout << " \n" << sa[s] << "\t" << a << endl ;
            string transition ;
            cout<<"Trnasition to : ";
            cin >> transition ;

```

```

        if(transition == "X")
            transition = "";

        map[sa[s]].push_back(transition) ;
    }
}

int max = (int)(pow(2,states) + 0.5) ;
unordered_map<string,vector<string> > nfa ;

string is, fs ;
cout << "\nEnter initial and final states : " << endl ;
cin >> is >> fs ;

queue<string> q ;
unordered_set<string> visited ;

q.push(is) ;

while(!q.empty())
{
    string present = q.front() ;
    q.pop() ;

    if(visited.count(present))
        continue ;

    visited.insert(present) ;

    for (int a = 0; a < alphabets; a++)
    {
        string ans = unionStates(present, a, map) ;
        nfa[present].push_back(ans) ;
        q.push(ans) ;
    }
}

```



```

}

cout << "\n\nFINAL ANSWER" << endl ;

unordered_map<string,vector<string> >::iterator i = nfa.begin();

for( ;i!=nfa.end() ;i++)
{
    cout << i->first << "\t\t" ;

    for (int a = 0; a < alphabets; a++)
    {
        cout << i->second[a] << "\t";
    }

    cout << endl ;
}

return 0 ;
}

```

## OUTPUT

Enter number of States :  
3

Enter 3 states :  
A B C

Enter number of inputs :  
2

Enter the transitions (Enter X for null) :

State Input  
A 0  
Trnasition to : AB

State Input  
A 1  
Trnasition to : A

State Input  
B 0  
Trnasition to : C

State Input  
B 1  
Trnasition to : C

State Input  
C 0  
Trnasition to : X

State Input  
C 1  
Trnasition to : X

Enter initial and final states :  
A C

FINAL ANSWER

AC	AB	A
ABC	ABC	AC
AB	ABC	AC
A	AB	A

-----  
Process exited after 14.92 seconds with return value 0  
Press any key to continue . . . |

## LEARNING OUTCOME

- What is Deterministic Finite Automaton
- Formal definition of Deterministic Finite Automaton
- What is acceptance
- Acceptance by Deterministic Finite Automaton
- Check if a string is accepted or rejected by a DFA

# EXPERIMENT - 2

## AIM

Write a program for acceptance of string by DFA

## THEORY

### Deterministic Finite Automaton

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called **Deterministic Automaton**. As it has a finite number of states, the machine is called **Deterministic Finite Machine** or **Deterministic Finite Automaton**.

### Formal Definition of a DFA

A DFA can be represented by a 5-tuple  $(Q, \Sigma, \delta, q_0, F)$  where –

- $Q$  is a finite set of states.
- $\Sigma$  is a finite set of symbols called the alphabet.
- $\delta$  is the transition function where  $\delta: Q \times \Sigma \rightarrow Q$
- $q_0$  is the initial state from where any input is processed ( $q_0 \in Q$ ).
- $F$  is a set of final state/states of  $Q$  ( $F \subseteq Q$ ).

### Acceptance by DFA

A string  $w$  is accepted by a DFA  $\langle Q, \Sigma, q_0, \delta, A \rangle$ , if and only if  $\delta^*(q_0, w) \in A$ . That is, a string is accepted by a DFA if and only if the DFA starting at the initial state ends in an accepting state after reading the string. If the DFA does not end on a final state, then the string is rejected.

A language  $L$  is accepted by a DFA  $\langle Q, \Sigma, q_0, \delta, A \rangle$ , if and only if  $L = \{ w \mid \delta^*(q_0, w) \in A \}$ . That is, the language accepted by a DFA is the set of strings accepted by the DFA. If any one string in the set is not accepted by the DFA then the Language is rejected by the DFA.

A DFA  $= (Q, \Sigma, s, F, T)$ , accepts a string  $w$  iff  $T(s, w) \in F$

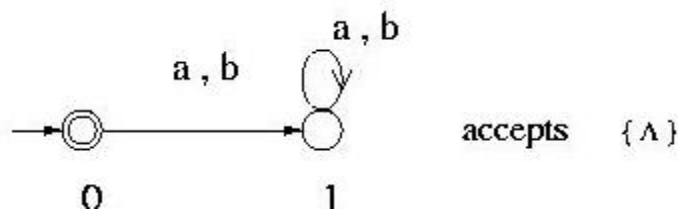
The language of the automaton  $A$  is

$$L(A) = \{w \mid A \text{ accepts } w\}.$$

More formally,

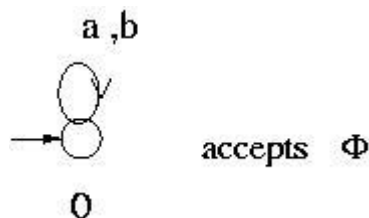
$$L(A) = \{w \mid T(\text{Start}(A), w) \in \text{Final}(A)\}$$

**Example 1 :**



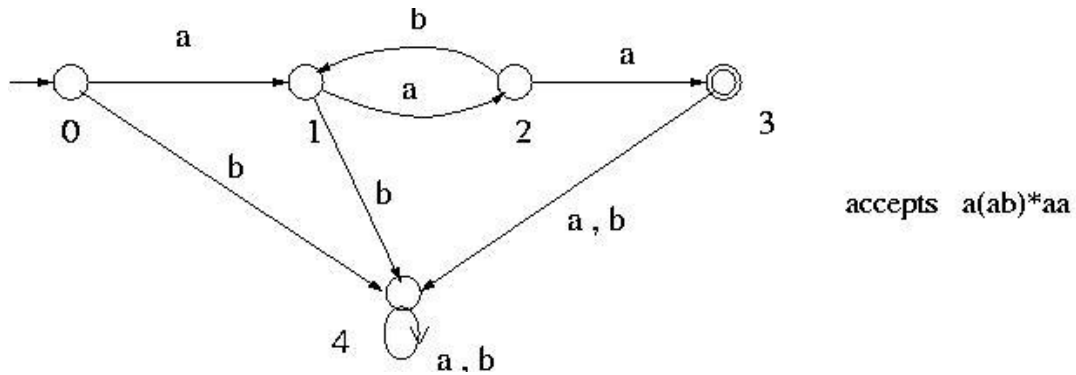
This DFA accepts  $\{\Lambda\}$  because it can go from the initial state to the accepting state (also the initial state) without reading any symbol of the alphabet i.e. by reading an empty string  $\Lambda$ . It accepts nothing else because any non-empty symbol would take it to state 1, which is not an accepting state, and it stays there.

**Example 2 :**



This DFA does not accept any string because it has no accepting state. Thus the language it accepts is the empty set  $\Phi$ .

**Example 3 :**



This DFA has a cycle: 1 - 2 - 1 and it can go through this cycle any number of times by reading substring ab repeatedly.

To find the language it accepts, first from the initial state go to state 1 by reading one a. Then from state 1 go through the cycle 1 - 2 - 1 any number of times by reading substring ab any number of times to come back to state 1. This is represented by  $(ab)^*$ . Then from state 1 go to state 2 and then to state 3 by reading aa. Thus a string that is accepted by this DFA can be represented by  $a(ab)^*aa$ .

### CODE

```

#include<stdio.h>
#include<conio.h>

int ninputs;

int check(char,int );    //function declaration
int dfa[10][10];
char c[10], string[10];
int main()
{
    int nstates, nfinals;
    int f[10];
    int i,j;
    printf("Enter the number of states in the DFA : ");
    scanf("%d",&nstates);
    printf("\nEnter the number of input symbol : ");
    scanf("%d",&ninputs);

```

```

printf("\nEnter the input symbols\t");
for(i=0; i<ninputs; i++)
{
    printf("\n\n %d input\t", i+1);
    printf("%c",c[i]=getch());
}
printf("\n\nEnter number of final states\t");
scanf("%d",&nfinals);

for(i=0;i<nfinals;i++)
{
    printf("\n\nFinal state %d : q",i+1);
    scanf("%d",&f[i]);
}

printf("-----");
printf("\n\nDefine transition rule as (initial state, input symbol ) = final state\n");
for(i=0; i<nstates; i++)
{
    for(j=0; j<ninputs; j++)
    {
        printf("\n(q%d , %c ) = q",i,c[j]);
        scanf("%d",&dfa[i][j]);
    }
}

do
{
    i=0;
    int final=0, s=0;
    printf("\n\nEnter Input String : ");
    scanf("%s",string);

    while(string[i]!='\0')

```

```

if((s=check(string[i++],s))<0)
    break;

for(i=0 ;i<nfinals ;i++)
    if(f[i] ==s )
        final=1;
        if(final==1)
            printf("\nVALID STRING");
        else
            printf("\nINVALID STRING");
        getch();

printf("\n\nDo you want to continue? (y/n) ");
} while(getch()=='y');

//  getch();
}

int check(char b,int d)
{
    int j;
    for(j=0; j<ninputs; j++)
        if(b==c[j])
            return(dfa[d][j]);
    return -1;
}

```

## OUTPUT

**Screenshot 1 –  
(DFA to accept Binary strings that ends with “01”)**

Enter the number of states in the DFA : 3

Enter the number of input symbol : 2

Enter the input symbols

1 input            0

2 input            1

Enter number of final states    1

Final state 1 : q2

-----  
Define transition rule as (initial state, input symbol ) = final state

(q0 , 0 ) = q1

(q0 , 1 ) = q0

(q1 , 0 ) = q1

(q1 , 1 ) = q2

(q2 , 0 ) = q1

(q2 , 1 ) = q0

Enter Input String : 101001

VALID STRING

Do you want to continue? (y/n)

-----  
Process exited after 31.49 seconds with return value 0  
Press any key to continue . . . |



**(DFA to accept strings containing exactly two '0' over input alphabets  $\Sigma = \{0, 1\}$ )**

```
Enter the number of states in the DFA : 4
Enter the number of input symbol : 2
Enter the input symbols
1 input      0
2 input      1
Enter number of final states
2
Final state 1 : q1
Final state 2 : q2
-----
Define transition rule as (initial state, input symbol ) = final state
(q0 , 0 ) = q1
(q0 , 1 ) = q0
(q1 , 0 ) = q2
(q1 , 1 ) = q1
(q2 , 0 ) = q3
(q2 , 1 ) = q2
(q3 , 0 ) = q3
(q3 , 1 ) = q3

Enter Input String : 11001111
VALID STRING
Do you want to continue? (y/n)
-----
Process exited after 33.41 seconds with return value 0
Press any key to continue . . .
```

## LEARNING OUTCOME

- What is Deterministic Finite Automaton
- Formal definition of Deterministic Finite Automaton
- What is acceptance
- Acceptance by Deterministic Finite Automaton
- Check if a string is accepted or rejected by a DFA

# EXPERIMENT - 3

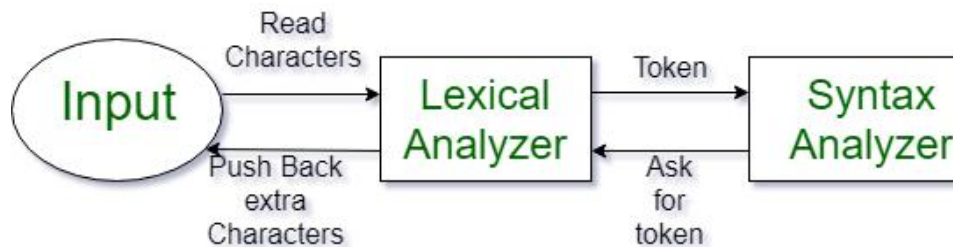
## AIM

Write a program to find different tokens in a program.

## THEORY

Lexical Analysis is the first phase of the compiler also known as a scanner. It converts the High level input program into a sequence of Tokens.

- Lexical Analysis can be implemented with the Deterministic finite Automata.
- The output is a sequence of tokens that is sent to the parser for syntax analysis



## Token

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

### Example of tokens:

Type token (id, number, real, . . . )

Punctuation tokens (IF, void, return, . . . )

Alphabetic tokens (keywords)

Keywords; Examples-for, while, if etc.

Identifier; Examples-Variable name, function name, etc.

Operators; Examples '+', '++', '-' etc.

Separators; Examples ',', ';' etc

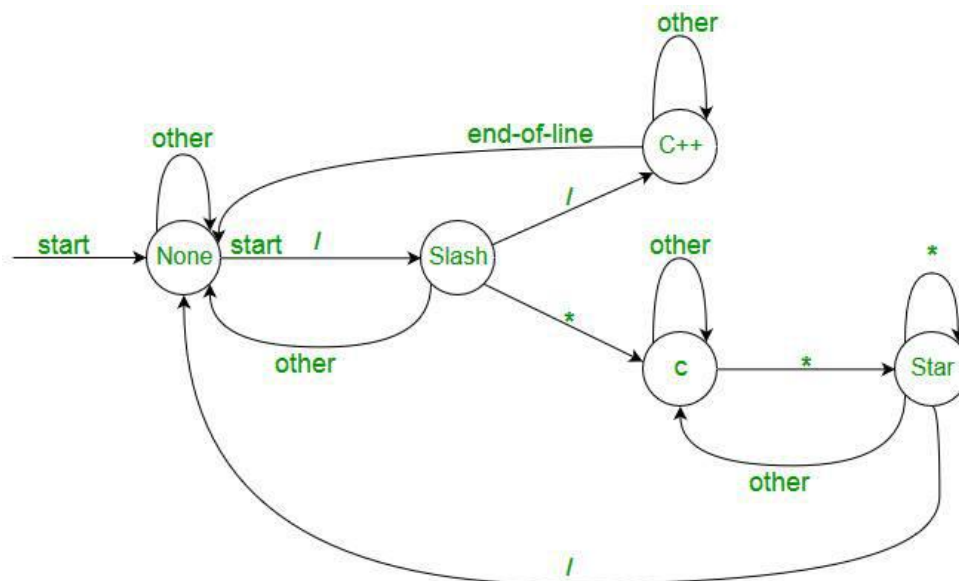
### Example of Non-Tokens:

Comments, preprocessor directive, macros, blanks, tabs, newline, etc.

**Lexeme:** The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme. eg- "float", "abs\_zero\_Kelvin", "=", "-", "273", ";;".

### How Lexical Analyzer functions

1. Tokenization i.e. Dividing the program into valid tokens.
2. Remove white space characters.
3. Remove comments.
4. It also provides help in generating error messages by providing row numbers and column numbers.



The lexical analyzer identifies the error with the help of the automation machine and the grammar of the given language on which it is based like C, C++, and gives row number and column number of the error.

Suppose we pass a statement through lexical analyzer –

`a = b + c ;` It will generate token sequence like this:

`id=id+id;` Where each id refers to it's variable in the symbol table referencing all details

**Example :**

```
int max(int i);
```

Lexical analyzer first read **int** and finds it to be valid and accepts as token  
**max** is read by it and found to be a valid function name after reading (  
**int** is also a token , then again **i** as another token and finally ;

**Answer:** Total number of tokens **7 : int, max, (, int, i, ), ;**

### **CODE**

```
#include<iostream>
#include <bits/stdc++.h>
using namespace std;

// Returns 'true' if the character is a DELIMITER.
bool isDelimiter(char ch)
{
    if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
        ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
        ch == '[' || ch == ']' || ch == '{' || ch == '}')
        return (true);
    return (false);
}

// Returns 'true' if the character is an OPERATOR.
bool isOperator(char ch)
{
    if (ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == '>' || ch == '<' ||
        ch == '=')
        return (true);
    return (false);
}
```

```
}
```

```
// Returns 'true' if the string is a VALID IDENTIFIER.
```

```
bool validIdentifier(char* str)
```

```
{  
    if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||  
        str[0] == '3' || str[0] == '4' || str[0] == '5' ||  
        str[0] == '6' || str[0] == '7' || str[0] == '8' ||  
        str[0] == '9' || isDelimiter(str[0]) == true)  
        return (false);  
    return (true);  
}
```

```
// Returns 'true' if the string is a KEYWORD.
```

```
bool isKeyword(char* str)
```

```
{  
    if (!strcmp(str, "if") || !strcmp(str, "else") ||  
        !strcmp(str, "while") || !strcmp(str, "do") ||  
        !strcmp(str, "break") ||  
        !strcmp(str, "continue") || !strcmp(str, "int")  
        || !strcmp(str, "double") || !strcmp(str, "float")  
        || !strcmp(str, "return") || !strcmp(str, "char")  
        || !strcmp(str, "case") || !strcmp(str, "char")  
        || !strcmp(str, "sizeof") || !strcmp(str, "long")  
        || !strcmp(str, "short") || !strcmp(str, "typedef")  
        || !strcmp(str, "switch") || !strcmp(str, "unsigned")  
        || !strcmp(str, "void") || !strcmp(str, "static")  
        || !strcmp(str, "struct") || !strcmp(str, "goto"))  
        return (true);  
    return (false);  
}
```

```
// Returns 'true' if the string is an INTEGER.
```

```
bool isInteger(char* str)
```

```
{
```

```

int i, len = strlen(str);

if (len == 0)
    return (false);
for (i = 0; i < len; i++) {
    if (str[i] != '0' && str[i] != '1' && str[i] != '2'
        && str[i] != '3' && str[i] != '4' && str[i] != '5'
        && str[i] != '6' && str[i] != '7' && str[i] != '8'
        && str[i] != '9' || (str[i] == '-' && i > 0))
        return (false);
}
return (true);
}

```

// Returns 'true' if the string is a REAL NUMBER.

```

bool isRealNumber(char* str)
{
    int i, len = strlen(str);
    bool hasDecimal = false;

    if (len == 0)
        return (false);
    for (i = 0; i < len; i++) {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
            && str[i] != '3' && str[i] != '4' && str[i] != '5'
            && str[i] != '6' && str[i] != '7' && str[i] != '8'
            && str[i] != '9' && str[i] != '.' ||
            (str[i] == '-' && i > 0))
            return (false);
        if (str[i] == '.')
            hasDecimal = true;
    }
    return (hasDecimal);
}

```

// Extracts the SUBSTRING.

```
char* subString(char* str, int left, int right)
{
    int i;
    char* subStr = (char*)malloc(
        sizeof(char) * (right - left + 2));

    for (i = left; i <= right; i++)
        subStr[i - left] = str[i];
    subStr[right - left + 1] = '\0';
    return (subStr);
}
```

// Parsing the input STRING.

```
void parse(char* str)
{
    int left = 0, right = 0;
    int len = strlen(str);

    while (right <= len && left <= right) {
        if (isDelimiter(str[right]) == false)
            right++;

        if (isDelimiter(str[right]) == true && left == right) {
            if (isOperator(str[right]) == true)
                printf("'"%c" IS AN OPERATOR\n", str[right]);

            right++;
            left = right;
        } else if (isDelimiter(str[right]) == true && left != right
            || (right == len && left != right)) {
            char* subStr = subString(str, left, right - 1);

            if (isKeyword(subStr) == true)
                printf("'"%s" IS A KEYWORD\n", subStr);
        }
    }
}
```

```

        else if (isInteger(subStr) == true)
            printf("'%'s' IS AN INTEGER\n", subStr);

        else if (isRealNumber(subStr) == true)
            printf("'%'s' IS A REAL NUMBER\n", subStr);

        else if (validIdentifier(subStr) == true
            && isDelimiter(str[right - 1]) == false)
            printf("'%'s' IS A VALID IDENTIFIER\n", subStr);

        else if (validIdentifier(subStr) == false
            && isDelimiter(str[right - 1]) == false)
            printf("'%'s' IS NOT A VALID IDENTIFIER\n", subStr);
        left = right;
    }
}
return;
}

// DRIVER FUNCTION
int main()
{
    // maximum length of string is 100 here
    char str[100];
    cout<<"Enter code : ";
    cin.getline(str,100);
    cout<<"\n";
    parse(str); // calling the parse function

    return (0);
}

```



## OUTPUT

---

```
Enter code : A = B - C * 5.0
```

```
'A' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'B' IS A VALID IDENTIFIER
'-' IS AN OPERATOR
'C' IS A VALID IDENTIFIER
'*' IS AN OPERATOR
'5.0' IS A REAL NUMBER
```

```
-----
Process exited after 40.88 seconds with return value 0
Press any key to continue . . .
```

## LEARNING OUTCOME

- Lexical Analysis is the first phase of the compiler also known as a scanner.
- It converts the High level input program into a sequence of Tokens.
- A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.
- The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme.
- C++ program to identify token in a program.

# EXPERIMENT - 4

## AIM

To implement lexical analyser.

## THEORY

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

## CODE

```
#include<bits/stdc++.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
using namespace std;

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], b[30], logical_op[] =
        "><", math_op[] = "+-*/=", numer[] = ".0123456789", other[] = ";\n\{\}\[\]'\":";

```

```

ifstream fin("lexicalinput.txt");
int mark[1000] = {
    0
};
int i, j = 0, kc = 0, ic = 0, lc = 0, mc = 0, nc = 0, oc = 0, aaa = 0;
vector < string > k;
vector < char > id;
vector < char > lo;
vector < char > ma;
vector < string > nu;
vector < char > ot;
if (!fin.is_open()) {
    cout << "error while opening the file\n";
    exit(0);
}
while (!fin.eof()) {
    ch = fin.get();
    for (i = 0; i < 12; ++i) {
        if (ch == other[i]) {
            int aa = ch;
            if (mark[aa] != 1) {
                ot.push_back(ch);
                mark[aa] = 1;
                ++oc;
            }
        }
    }
}
for (i = 0; i < 5; ++i) {
    if (ch == math_op[i]) {
        int aa = ch;
        if (mark[aa] != 1) {
            ma.push_back(ch);
            mark[aa] = 1;
            ++mc;
        }
    }
}

```

```

    }
    for (i = 0; i < 2; ++i) {
        if (ch == logical_op[i]) {
            int aa = ch;
            if (mark[aa] != 1) {
                lo.push_back(ch);
                mark[aa] = 1;
                ++lc;
            }
        }
    }
    if (ch == '0' || ch == '1' || ch == '2' || ch == '3' || ch == '4' || ch == '5' || ch == '6' ||
ch == '7' ||
        ch == '8' || ch == '9' || ch == '.' || ch == ' ' || ch == '\n' || ch == ';') {
        if (ch == '0' || ch == '1' || ch == '2' || ch == '3' || ch == '4' || ch == '5' || ch == '6' ||
ch == '7' ||
            ch == '8' || ch == '9' || ch == '.') b[aaa++] = ch;
        if ((ch == ' ' || ch == '\n' || ch == ';') && (aaa != 0)) {
            b[aaa] = '\0';
            aaa = 0;
            char arr[30];
            strcpy(arr, b);
            nu.push_back(arr);
            ++nc;
        }
    }
    if (isalnum(ch)) {
        buffer[j++] = ch;
    } else if ((ch == ' ' || ch == '\n') && (j != 0)) {
        buffer[j] = '\0';
        j = 0;
        if (isKeyword(buffer) == 1) {
            k.push_back(buffer);
            ++kc;
        } else {
            if (buffer[0] >= 97 && buffer[0] <= 122) {

```

```

        if (mark[buffer[0] - 'a'] != 1) {
            id.push_back(buffer[0]);
            ++ic;
            mark[buffer[0] - 'a'] = 1;
        }
    }
}
}
}
fin.close();
printf("Keywords: ");
for (int f = 0; f < kc; ++f) {
    if (f == kc - 1) {
        cout << k[f] << "\n";
    } else {
        cout << k[f] << ", ";
    }
}
printf("Identifiers: ");
for (int f = 0; f < ic; ++f) {
    if (f == ic - 1) {
        cout << id[f] << "\n";
    } else {
        cout << id[f] << ", ";
    }
}
printf("Math Operators: ");
for (int f = 0; f < mc; ++f) {
    if (f == mc - 1) {
        cout << ma[f] << "\n";
    } else {
        cout << ma[f] << ", ";
    }
}
printf("Logical Operators: ");
for (int f = 0; f < lc; ++f) {

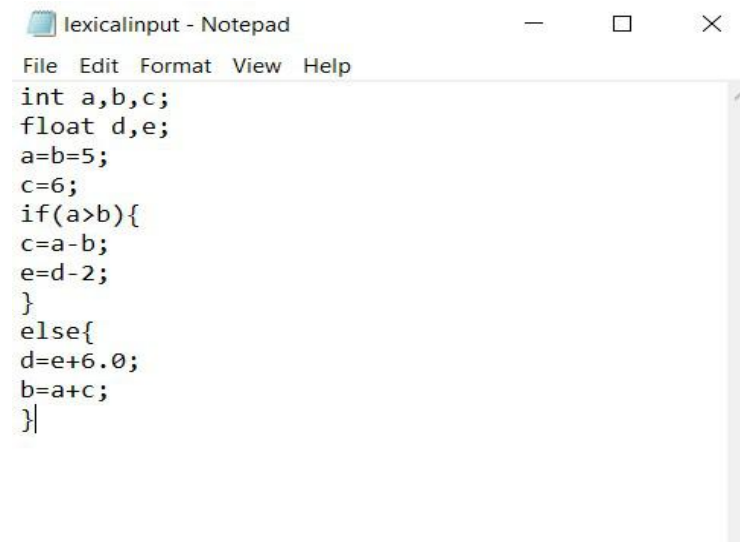
```

```

    if (f == lc - 1) {
        cout << lo[f] << "\n";
    } else {
        cout << lo[f] << ", ";
    }
}
printf("Numerical Values: ");
for (int f = 0; f < nc; ++f) {
    if (f == nc - 1) {
        cout << nu[f] << "\n";
    } else {
        cout << nu[f] << ", ";
    }
}
printf("Others: ");
for (int f = 0; f < oc; ++f) {
    if (f == oc - 1) {
        cout << ot[f] << "\n";
    } else {
        cout << ot[f] << " ";
    }
}
return 0;
}

```

## OUTPUT



```
lexicalinput - Notepad
File Edit Format View Help
int a,b,c;
float d,e;
a=b=5;
c=6;
if(a>b){
c=a-b;
e=d-2;
}
else{
d=e+6.0;
b=a+c;
}|
```

Keywords: int, float, else  
Identifiers: a, d, c, i, e, b  
Math Operators: =, -, +  
Logical Operators: >  
Numerical Values: 5, 6, 2, 6.0  
Others: , ; ( ) { }

-----  
Process exited after 0.06849 seconds with return value 0  
Press any key to continue . . .

## LEARNING OUTCOME

- What is Deterministic Finite Automaton
- Formal definition of Deterministic Finite Automaton
- What is acceptance
- Acceptance by Deterministic Finite Automaton
- Check if a string is accepted or rejected by a DFA



# EXPERIMENT - 5

## AIM

Write a program to implement recursive descent parser.

## THEORY

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and nonterminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require backtracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing.

Consider the grammar used before for simple arithmetic expressions:

$$\begin{aligned}P &\rightarrow E \\E &\rightarrow E + T \mid E - T \mid T \\T &\rightarrow T * S \mid T / S \mid S \\S &\rightarrow F \wedge S \mid F \\F &\rightarrow ( E ) \mid \text{char}\end{aligned}$$

The above grammar won't work for recursive descent because of the left recursion in the second and third rules. (The recursive function for E would immediately call E recursively, resulting in an indefinite recursive regression.)

In order to eliminate left recursion, one simple method is to introduce new notation: curly brackets, where {xx} means "zero or more repetitions of xx", and parentheses () used for grouping, along with the or-symbol: |. Because of the many metasymbols, it is a good idea to enclose all terminals in single quotes. Also put a '\$' at the end. The resulting grammar looks as follows :

$$\begin{aligned}P &\rightarrow E \$ \\E &\rightarrow T \{ ('+' | '-') T \} \\T &\rightarrow S \{ ('*' | '/') S \}\end{aligned}$$

$$S \rightarrow F \wedge S \mid F$$

$$F \rightarrow (' E ') \mid \text{char}$$

Now the grammar is suitable for creation of a recursive descent parser. Notice that this is a different grammar that describes the same language, that is the same sentences or strings of terminal symbols. A given sentence will have a similar parse tree to one given by the previous grammar, but not necessarily the same parse tree.

One could alter the first grammar in other ways to make it work for recursive descent. For example, one could write the rule for E as:

$$E \rightarrow T '+' E \mid T$$

## CODE

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

char input[100];
int i,l;

void main()
{
    printf("\nRecursive descent parsing for the grammar shown below:\n");
    printf("\nE->TE'\nE' -> +TE'/@\nT->FT'\nT' -> *FT'/@\nF->(E)/ID\n");
    printf("\nEnter the string to be checked:");
    gets(input);

    if(E())
    {
        if(input[i+1]=='\0')
            printf("\nString is accepted");
        else
            printf("\nString is not accepted");
    }

    else
        printf("\nString not accepted");
}
```

```

    getch();
}

E()
{
    if(T())
    {
        if(EP())
            return(1);
        else
            return(0);
    }

    else
        return(0);
}

EP()
{
    if(input[i]=='+')
    {
        i++;
        if(T())
        {
            if(EP())
                return(1);
            else
                return(0);
        }
        else
            return(0);
    }

    else
        return(1);
}

T()
{

```

```

    if(F())
    {
        if(TP())
            return(1);
        else
            return(0);
    }

    else
        return(0);
}

```

```

TP()
{
    if(input[i]=='*')
    {
        i++;
        if(F())
        {
            if(TP())
                return(1);
            else
                return(0);
        }
        else
            return(0);
    }
}

```

```

    else
        return(1);
}

```

```

F()
{
    if(input[i]=='(')
    {
        i++;
        if(E())
        {
            if(input[i]==')')

```

```

        {
            i++;
            return(1);
        }
        else
            return(0);
    }
    else
        return(0);
}

else if(input[i]>='a'&&input[i]<='z' || input[i]>='A'&&input[i]<='Z')
{
    i++;
    return(1);
}

else
    return(0);
}

```

## OUTPUT

Recursive descent parsing for the grammar shown below:

```

E->TE'
E' ->+TE' / @
T->FT'
T' ->*FT' / @
F->(E) / ID

```

Enter the string to be checked: a+b\*c

String is accepted

## LEARNING OUTCOME

- Recursive descent is a top-down parsing technique
- The input is read from left to right
- Implementation of Recursive Descent Parser in C++

# EXPERIMENT - 6

## AIM

Write a program to implement left factoring.

## THEORY

A grammar is said to be left factored when it is of the form: –

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \dots \mid \alpha\beta_n \mid \gamma$$

i.e the productions start with the same terminal (or set of terminals).

On seeing the input  $\alpha$  we cannot immediately tell which production to choose to expand A. Hence, left factoring is a grammar transformation that is useful for producing grammar suitable for predictive or top-down parsing. When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen to make the right choice.

For the grammar,

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \dots \mid \alpha\beta_n \mid \gamma$$

The equivalent left factored grammar will be: –

$$\begin{aligned} A &\rightarrow \alpha A' \mid \gamma \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \dots \mid \beta_n \end{aligned}$$

## CODE

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

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

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

```
struct production
```

```
{
```

```
    int length;
```

```
    char prod[10][10];
```

```
} typedef production;
```

```
struct grammer
```

```
{
```

```
    int number;
```

```
    production nonT[26];
```

```
} typedef grammer;
```

```
grammer gram;
```

```
void printProd(production *);
```

```
void printGram();
```

```
int getF(char *str)
```

```
{
```

```
    int i = 1;
```

```
    while (i < strlen(str) && str[i - 1] != '>') i++;
```

```
    while (i < strlen(str) && str[i] == ' ') i++;
```

```
    return i;
```

```
}
```

```
void setProd(char *str)
```

```
{
```

```
    production *produce = &gram.nonT[str[0] - 'A'];
```

```
    int prod = 0, j;
```

```
    for (int i = getF(str); i < strlen(str); i++)
```

```
    {
```

```
        for (j = 0; i < strlen(str) && str[i] != '|'; i++)
```

```
            if (str[i] != ' ')
```

```
                produce->prod[prod][j++] = str[i];
```

```
            produce->prod[prod++][j++] = '\0';
```

```
    }
```

```
    produce->length = prod;
```

```
}
```

```
void copy(char *src, char *dest, int i)
```

```
{
```

```
    int j = i + 1;
```

```
    do
```

```
    {
```

```
        dest[j - i - 1] = src[j];
```

```
    } while (src[j++] != '\0');
```

```
    if (dest[0] == '\0')
```



```

{
    dest[0] = 'e';
    dest[1] = '\0';
}
}

```

```

bool factorProd(production *prod)
{
    bool prev[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
    bool next[10] = {0};
    char c[128] = {0}, ch;
    int j = 0, maxx;
    while (true)
    {
        maxx = 0;
        for (int i = 0; i < 10; i++)
            if (prev[i] && strlen(prod->prod[i]) > j)
                c[prod->prod[i][j]]++;

        for (int i = 0; i < 128; i++)
        {
            if (c[i] > maxx) maxx = c[i], ch = i;
            c[i] = 0;
        }
    }
}

```

```

    if (maxx < 2) break;

    for (int i = 0; i < 10; i++)
        prev[i] = j < strlen(prod->prod[i]) && prod->prod[i][j] == ch;
    j++;
}

if (j == 0) return false;

int newNonT = 0, p_no = 0;
while (gram.nonT[newNonT].length != 0) newNonT++;

production *newP = &gram.nonT[newNonT];

for (int i = 0; i < 10; i++)
    if (prev[i] && prod->prod[i][0] != '\0')
    {
        copy(prod->prod[i], newP->prod[p_no], j - 1);
        prod->prod[i][j] = newNonT + 'A';
        prod->prod[i][j + 1] = '\0';
        p_no++;
    }

gram.number++;
newP->length = p_no;

```

```

int i = 0, num = 0;
while (prev[i] == false) i++;
for (i++; i < 10; i++)
    if (prev[i] && prod->prod[i][0] != '\0')
        prod->prod[i][0] = '\0', num++;

prod->length -= num;
return true;
}

void leftFactor()
{
    for (int i = 0; i < 26; i++)
    {
        if (gram.number == 26)
        {
            printf("No new production can be formed.\n");
            return;
        }
        if (gram.nonT[i].length >= 2)
            while (factorProd(&gram.nonT[i]))
            {
                printf("\n\nAfter left factoring production from %c:", i + 'A');
                printGram();
            }
    }
}

```

```

    }
}

int main()
{
    char str[100];

    printf("Enter the number of productions: ");
    scanf("%d", &(gram.number));

    printf("\nEnter productions in the format:\n");
    printf("A -> aAb | aAab | e\n");
    printf("(Spaces will be skipped, e is null)\n");

    char c;
    for (int i = 0; i < gram.number; i++)
    {
        scanf("%c", &c);
        scanf("%[^\\n]s", str);
        setProd(str);
    }
    printf("\nInput Grammer:");
    printGram();

    leftFactor();

```

```

    printf("\n\nFinally... ..");
    printGram();
    printf("\n");
    return 0;
}

void printProd(production *prod)
{
    printf("%s ", prod->prod[0]);
    for (int i = 1; i < 10; i++)
        if (prod->prod[i][0] != '\0')
            printf("| %s ", prod->prod[i]);
}

void printGram()
{
    for (int i = 0; i < 26; i++)
        if (gram.nonT[i].length)
        {
            printf("\n%c -> ", i + 'A');
            printProd(&gram.nonT[i]);
        }
}

```

## OUTPUT

```
Enter the number of productions: 1

Enter productions in the format:
A -> aAb | aAab | e
(Spaces will be skipped, e is null)
S->aSSbS|aSaSb|abb|b

Input Grammer:
S -> aSSbS | aSaSb | abb | b

After left factoring production from S:
A -> SbS | aSb
S -> aSA | abb | b

After left factoring production from S:
A -> SbS | aSb
B -> SA | bb
S -> aB | b

Finally... ..
A -> SbS | aSb
B -> SA | bb
S -> aB | b

Process returned 0 (0x0)   execution time : 60.106 s
Press any key to continue.
```

## LEARNING OUTCOME

- We have learned about Factoring and what is “Left Factoring”.
- We have learned how to identify a “Left Factoring” in a given language and write a program for the same.

# EXPERIMENT - 7

## AIM

Write a program to convert left recursive grammar to right recursive grammar

## THEORY

The production is left-recursive if the leftmost symbol on the right side is the same as the non terminal on the left side. For example,  $\text{expr} \rightarrow \text{expr} + \text{term}$ .

For each rule which contains a left-recursive option,

$$A \rightarrow A\alpha \mid \beta$$

introduce a new nonterminal  $A'$  and rewrite the rule as

$$A \rightarrow \beta A' \quad A' \rightarrow \alpha A' \mid \epsilon$$

Thus the production:

$$E \rightarrow E + T \mid T$$

is left-recursive with "E" playing the role of "A", "+" T" playing the role of  $\alpha$ , and "T" playing the role of  $\beta$ . Introducing the new nonterminal  $E'$ , the production can be replaced by:

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \end{aligned}$$

For example, the left-recursive grammar is:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ E &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

We can redefine E and T without left-recursion as:

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \mid \epsilon \\ T &\rightarrow F T' \end{aligned}$$

$$T \rightarrow * FT' \mid E$$
$$F \rightarrow (E) \mid id$$

## CODE

```
#include <bits/stdc++.h>
#include <iostream>
#define pb push_back
using namespace std;

set<string> prod[26], ans[52];
bool used[26];

int h(char ch)
{
    return ch-65;
}

char rh(int x)
{
    return x+65;
}

bool isTerminal(char ch)
{
    if(ch=='^'||(ch>=65&&ch<=90))
        return false;
    return true;
}

void convert_to_right(int i, set<string> conv, set<string> S)
{
    for(set<string>::iterator j=S.begin();j!=S.end();j++)
    {
        string x= *j+rh(i);
        x+="";
    }
}
```



```

        ans[i].insert(x);
    }

    for(set<string>::iterator j=conv.begin();j!=conv.end();j++)
    {
        string x=(*j).substr(1)+rh(i);
        x+="";
        ans[i+26].insert(x);
    }

    ans[i+26].insert("^");
}

int main()
{
    int m;
    cout<<"\nEnter number of productions: ";
    cin>>m;
    cout<<"\nEnter production rules:\n";

    int i, st;
    string s;

    for(i=0;i<m;i++)
    {
        cin>>s;
        int x=h(s[0]);
        used[x]=1;
        int j=1;
        while(s[j]!='-'||s[j]!='>')
            j++;
        while(j<s.size())
        {
            string v="";
            while(s[j]!=' ')j++;

```

```

        while(j<s.size()&& s[j]!='|')
        {
            v+=s[j];
            j++;
        }
        prod[x].insert(v);
        j++;
    }
}

for(i=0;i<26;i++)
{
    if(prod[i].empty())
        continue;
    set<string> S=prod[i], conv, aux;
    set<string>::iterator it;
    for(it=S.begin();it!=S.end();it++)
    {
        string v=*it;
        int j=h(v[0]);
        if(j>=0&&j<i)
        {
            //modify this production
            set<string>::iterator k=ans[j].begin();
            while(k!=ans[j].end())
            {
                aux.insert(*k+v.substr(1));
                k++;
            }
        }
        else
            aux.insert(v);
    }

    for(it=aux.begin();it!=aux.end();it++)

```

```

{
    string v=*it;
    int j=h(v[0]);
    if(j==i)
        conv.insert(v);
    else
        ans[i].insert(v);
}

if(conv.size()==0)
    continue;

S=ans[i];
ans[i].clear();
convert_to_right(i, conv, S);
}

cout<<"Productions after removal of left Recursion\n";
for(i=0;i<26;i++)
{
    if(ans[i].empty())continue;
    cout<<rh(i)<<"-> ";

    for(set<string>::iterator j=ans[i].begin();j!=ans[i].end();j++)
    {
        cout<<*j;
        set<string>::iterator k=j;
        k++;
        if(k!=ans[i].end())
            cout<<" | ";
    }
    cout<<endl;

    if(ans[i+26].empty())
        continue;
}

```

```

        cout<<rh(i)<<"-> ";

        for(set<string>::iterator j=ans[i+26].begin();j!=ans[i+26].end();j++)
        {
            cout<<*j;
            set<string>::iterator k=j;
            k++;
            if(k!=ans[i+26].end())
                cout<<" | ";
        }
        cout<<endl;
    }
}

```

## OUTPUT

Enter number of productions: 2

Enter production rules:

A->ABd|Aa|a

B->Be|b

Productions after removal of left Recursion

A-> aA'

A' -> BdA' | ^ | aA'

B-> bB'

B' -> ^ | eB'

# EXPERIMENT - 8

## AIM

Write a program to compute First and Follow.

## THEORY

First(y) is the set of terminals that begin the strings derived from y. Follow(A) is the set of terminals that can appear to the right of A. First and Follow are used in the construction of the parsing table.

$A \rightarrow abc / def / ghi$

$First(A) = \{ a, d, g \}$

### ➤ To compute First :-

- X is a terminal  $First(X) = \{X\}$
- $X \rightarrow \epsilon$  is a production add  $\epsilon$  to  $First(X)$
- X is a non-terminal and  $X \rightarrow Y_1 Y_2 \dots Y_k$  is a production place z in  $First(X)$  if z is in  $First(Y_i)$  for some i and  $\epsilon$  is in all of  $First(Y_1) \dots First(Y_{i-1})$

### ➤ To compute Follow :-

- Place \$ in  $Follow(S)$ , where S is the start symbol and \$ is the end-of-input marker.
- There is a production  $A \rightarrow \alpha B \beta$  everything in  $First(\beta)$  except for  $\epsilon$  is placed in  $Follow(B)$ .
- There is a production  $A \rightarrow \alpha B$ , or a production  $A \rightarrow \alpha B \beta$  where  $First(\beta)$  contains  $\epsilon$  everything in  $Follow(A)$  is placed in  $Follow(B)$ .

So, now lets see C code to compute First and Follow.

## CODE

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

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

void findfirst(char, int, int);

int count, n = 0;

char calc_first[10][100];

char calc_follow[10][100];
int m = 0;

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

int main(int argc, char **argv)
{
    int jm = 0;
    int km = 0;
    int i, choice;
    char c, ch;
    count = 8;

    strcpy(production[0], "S=AR");
    strcpy(production[1], "R=+AR");
    strcpy(production[2], "R=#");
```

```

strcpy(production[3], "A=FY");
strcpy(production[4], "Y=*FY");
strcpy(production[5], "Y=#");
strcpy(production[6], "F=(S)");
strcpy(production[7], "F=i");

int kay;
char done[count];
int ptr = -1;

for(k = 0; k < count; k++) {
    for(kay = 0; kay < 100; kay++) {
        calc_first[k][kay] = '!';
    }
}
int point1 = 0, point2, xxx;

for(k = 0; k < count; k++)
{
    c = production[k][0];
    point2 = 0;
    xxx = 0;

    for(kay = 0; kay <= ptr; kay++)
        if(c == done[kay])
            xxx = 1;

    if (xxx == 1)
        continue;

    findfirst(c, 0, 0);
    ptr += 1;

    done[ptr] = c;
    printf("\n First(%c) = { ", c);
    calc_first[point1][point2++] = c;

```

```

for(i = 0 + jm; i < n; i++) {
    int lark = 0, chk = 0;

    for(lark = 0; lark < point2; lark++) {

        if (first[i] == calc_first[point1][lark])
        {
            chk = 1;
            break;
        }
    }
    if(chk == 0)
    {
        printf("%c, ", first[i]);
        calc_first[point1][point2++] = first[i];
    }
}
printf("{}");
jm = n;
point1++;
}
printf("\n\n");
printf("-----\n\n");
char donee[count];
ptr = -1;

for(k = 0; k < count; k++) {
    for(kay = 0; kay < 100; kay++) {
        calc_follow[k][kay] = '!';
    }
}
point1 = 0;
int land = 0;
for(e = 0; e < count; e++)
{

```



```

ck = production[e][0];
point2 = 0;
xxx = 0;

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

if (xxx == 1)
    continue;
land += 1;

follow(ck);
ptr += 1;

donee[ptr] = ck;
printf(" Follow(%c) = { ", ck);
calc_follow[point1][point2++] = ck;

for(i = 0 + km; i < m; i++) {
    int lark = 0, chk = 0;
    for(lark = 0; lark < point2; lark++)
    {
        if (f[i] == calc_follow[point1][lark])
        {
            chk = 1;
            break;
        }
    }
    if(chk == 0)
    {
        printf("%c, ", f[i]);
        calc_follow[point1][point2++] = f[i];
    }
}
printf(" }\n");

```

```

        km = m;
        point1++;
    }
}

void follow(char c)
{
    int i, j;

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

                if(production[i][j+1] == '\0' && c != production[i][0])
                {
                    follow(production[i][0]);
                }
            }
        }
    }
}

```

```

void findfirst(char c, int q1, int q2)
{
    int j;

```

```

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

```

```

void followfirst(char c, int c1, int c2)
{
    int k;

    if(!(isupper(c)))

```

```

        f[m++] = c;
    else
    {
        int i = 0, j = 1;
        for(i = 0; i < count; i++)
        {
            if(calc_first[i][0] == c)
                break;
        }

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

```

## OUTPUT

### Output

```
First(S) = { (, i, }  
First(R) = { +, #, }  
First(A) = { (, i, }  
First(Y) = { *, #, }  
First(F) = { (, i, }
```

```
Follow(S) = { $, ), }  
Follow(R) = { $, ), }  
Follow(A) = { +, $, ), }  
Follow(Y) = { +, $, ), }  
Follow(F) = { *, +, $, ), }
```

## LEARNING OUTCOME

- We have learned and briefly grasped the concept of "First and Follow".
- We have also learned how to compute the "First and Follow" for the given grammar.

# EXPERIMENT - 9

## AIM

Write a program to construct LL(1) parsing table

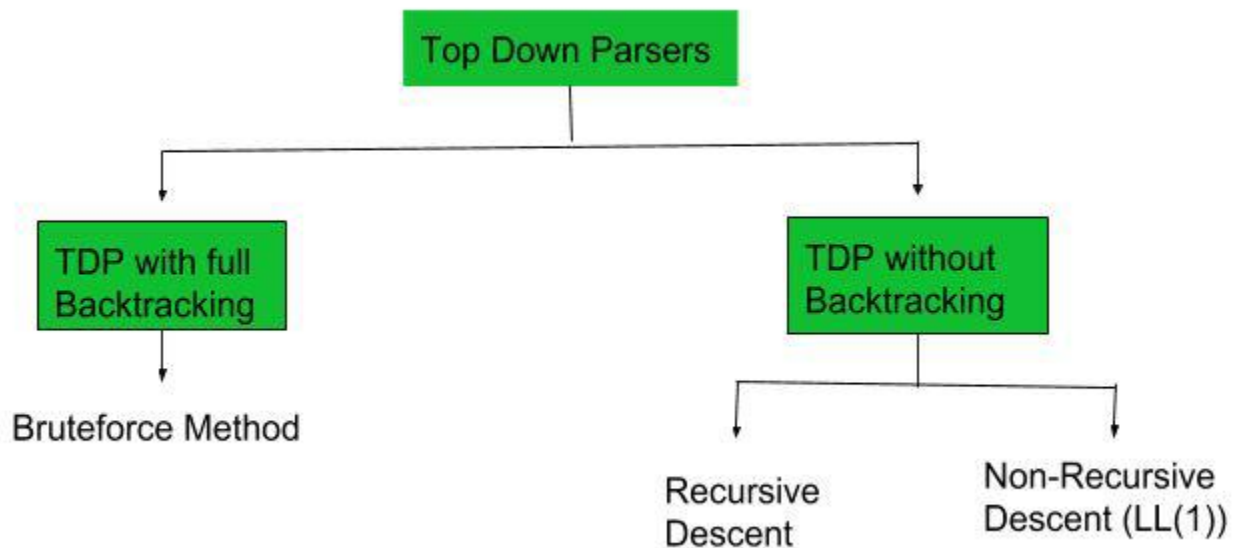
## THEORY

A top-down parser builds the parse tree from the top down, starting with the start non-terminal. There are two types of Top-Down Parsers:

Top-Down Parser with Backtracking

Top-Down Parsers without Backtracking

Top-Down Parsers without backtracking can further be divided into two parts:



In this article, we are going to discuss Non-Recursive Descent which is also known as LL(1) Parser.

LL(1) Parsing:

Here the 1st L represents that the scanning of the Input will be done from Left to Right manner and the second L shows that in this parsing technique we are going to

use Left most Derivation Tree. And finally, the 1 represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.

Algorithm to construct LL(1) Parsing Table:

Step 1: First check for left recursion in the grammar, if there is left recursion in the grammar remove that and go to step 2.

Step 2: Calculate First() and Follow() for all non-terminals.

First(): If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.

Follow(): What is the Terminal Symbol which follows a variable in the process of derivation.

Step 3: For each production  $A \rightarrow \alpha$ . ( $A$  tends to alpha)

Find  $\text{First}(\alpha)$  and for each terminal in  $\text{First}(\alpha)$ , make entry  $A \rightarrow \alpha$  in the table.

If  $\text{First}(\alpha)$  contains  $\epsilon$  (epsilon) as terminal than, find the  $\text{Follow}(A)$  and for each terminal in  $\text{Follow}(A)$ , make entry  $A \rightarrow \alpha$  in the table.

If the  $\text{First}(\alpha)$  contains  $\epsilon$  and  $\text{Follow}(A)$  contains \$ as terminal, then make entry  $A \rightarrow \alpha$  in the table for the \$.

To construct the parsing table, we have two functions:

In the table, rows will contain the Non-Terminals and the column will contain the Terminal Symbols. All the Null Productions of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of the First set.

Now, let's understand with an example.

Example-1:

Consider the Grammar:

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow id \mid (E)$

\* $\epsilon$  denotes epsilon

Find their First and Follow sets:

	First	Follow
$E \rightarrow TE'$	{ id, ( }	{ \$, ) }
$E' \rightarrow +TE'/\epsilon$	{ +, $\epsilon$ }	{ \$, ) }
$T \rightarrow FT'$	{ id, ( }	{ +, \$, ) }
$T' \rightarrow *FT'/\epsilon$	{ *, $\epsilon$ }	{ +, \$, ) }
$F \rightarrow id/(E)$	{ id, ( }	{ *, +, \$, ) }

Now, the LL(1) Parsing Table is:

	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		



	id	+	*	(	)	\$
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

As you can see that all the null productions are put under the Follow set of that symbol and all the remaining productions are lie under the First of that symbol.

Note: Every grammar is not feasible for LL(1) Parsing table. It may be possible that one cell may contain more than one production.

Let's see with an example.

Example-2:  
Consider the Grammar

$S \rightarrow A \mid a$

$A \rightarrow a$

Find their First and Follow sets:

	First	Follow
$S \rightarrow A/a$	{ a }	{ \$ }
$A \rightarrow a$	{ a }	{ \$ }

Parsing Table:

	a	\$
S	$S \rightarrow A, S \rightarrow a$	

a                      \$

A    A → a

Here, we can see that there are two productions into the same cell. Hence, this grammar is not feasible for LL(1) Parser.

### CODE

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

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

int count,n=0;
char calc_first[10][100];
char calc_follow[10][100];
int m=0;
char production[10][10], first[10];
char f[10];
int k;
char ck;
int e;

int main(int argc,char **argv)
{
    int jm=0;
    int km=0;
    int i,choice;
    char c,ch;
    printf("How many productions ? :");
    scanf("%d",&count);
    printf("\nEnter %d productions in form A=B where A and B are grammar symbols :\n\n",count);
```

```

for(i=0;i<count;i++)
{
    scanf("%s%c",production[i],&ch);
}
int kay;
char done[count];
int ptr = -1;
for(k=0;k<count;k++){
    for(kay=0;kay<100;kay++){
        calc_first[k][kay] = '!';
    }
}
int point1 = 0,point2,xxx;
for(k=0;k<count;k++)
{
    c=production[k][0];
    point2 = 0;
    xxx = 0;
    for(kay = 0; kay <= ptr; kay++)
        if(c == done[kay])
            xxx = 1;
    if (xxx == 1)
        continue;
    findfirst(c,0,0);
    ptr+=1;
    done[ptr] = c;
    printf("\n First(%c)= { ",c);
    calc_first[point1][point2++] = c;
    for(i=0+jm;i<n;i++){
        int lark = 0,chk = 0;
        for(lark=0;lark<point2;lark++){
            if (first[i] == calc_first[point1][lark]){
                chk = 1;
                break;
            }
        }
    }
}

```

```

        if(chk == 0){
            printf("%c, ",first[i]);
            calc_first[point1][point2++] = first[i];
        }
    }
    printf("\n");
    jm=n;
    point1++;
}
printf("\n");
printf("-----\n\n");
char donee[count];
ptr = -1;
for(k=0;k<count;k++){
    for(kay=0;kay<100;kay++){
        calc_follow[k][kay] = '!';
    }
}
point1 = 0;
int land = 0;
for(e=0;e<count;e++)
{
    ck=production[e][0];
    point2 = 0;
    xxx = 0;
    for(kay = 0; kay <= ptr; kay++)
        if(ck == donee[kay])
            xxx = 1;
    if (xxx == 1)
        continue;
    land += 1;
    follow(ck);
    ptr+=1;
    donee[ptr] = ck;
    printf(" Follow(%c) = { ",ck);
    calc_follow[point1][point2++] = ck;
}

```

```

        for(i=0+km;i<m;i++){
            int lark = 0,chk = 0;
            for(lark=0;lark<point2;lark++){
                if (f[i] == calc_follow[point1][lark]){
                    chk = 1;
                    break;
                }
            }
            if(chk == 0){
                printf("%c, ",f[i]);
                calc_follow[point1][point2++] = f[i];
            }
        }
        printf(" }\n\n");
        km=m;
        point1++;
    }
    char ter[10];
    for(k=0;k<10;k++){
        ter[k] = '!';
    }
    int ap,vp,sid = 0;
    for(k=0;k<count;k++){
        for(kay=0;kay<count;kay++){
            if(!isupper(production[k][kay]) && production[k][kay] != '#' &&
production[k][kay] != '=' && production[k][kay] != '\0'){
                vp = 0;
                for(ap = 0;ap < sid; ap++){
                    if(production[k][kay] == ter[ap]){
                        vp = 1;
                        break;
                    }
                }
            }
            if(vp == 0){
                ter[sid] = production[k][kay];
                sid ++;
            }
        }
    }

```



```

                                tem[ct++] = calc_first[zap][tuna];
                                }
                                else
                                break;
                                }
                                break;
                                }
                                }
                                tem[ct++] = '_';
                                }
                                k++;
                                }
                                int zap = 0,tuna;
                                for(tuna = 0;tuna<ct;tuna++){
                                    if(tem[tuna] == '#'){
                                        zap = 1;
                                    }
                                    else if(tem[tuna] == '_'){
                                        if(zap == 1){
                                            zap = 0;
                                        }
                                        else
                                            break;
                                    }
                                    else{
                                        first_prod[ap][destiny++] = tem[tuna];
                                    }
                                }
                                }
                                char table[land][sid+1];
                                ptr = -1;
                                for(ap = 0; ap < land ; ap++){
                                    for(kay = 0; kay < (sid + 1) ; kay++){
                                        table[ap][kay] = '!';
                                    }
                                }

```

```

for(ap = 0; ap < count ; ap++){
    ck = production[ap][0];
    xxx = 0;
    for(kay = 0; kay <= ptr; kay++){
        if(ck == table[kay][0])
            xxx = 1;
    }
    if (xxx == 1)
        continue;
    else{
        ptr = ptr + 1;
        table[ptr][0] = ck;
    }
}

for(ap = 0; ap < count ; ap++){
    int tuna = 0;
    while(first_prod[ap][tuna] != '\0'){
        int to,ni=0;
        for(to=0;to<sid;to++){
            if(first_prod[ap][tuna] == ter[to]){
                ni = 1;
            }
        }
        if(ni == 1){
            char xz = production[ap][0];
            int cz=0;
            while(table[cz][0] != xz){
                cz = cz + 1;
            }
            int vz=0;
            while(ter[vz] != first_prod[ap][tuna]){
                vz = vz + 1;
            }
            table[cz][vz+1] = (char)(ap + 65);
        }
        tuna++;
    }
}

```



```

}
for(k=0;k<sid;k++){
    for(kay=0;kay<100;kay++){
        if(calc_first[k][kay] == '!'){
            break;
        }
        else if(calc_first[k][kay] == '#'){
            int fz = 1;
            while(calc_follow[k][fz] != '!'){
                char xz = production[k][0];
                int cz=0;
                while(table[cz][0] != xz){
                    cz = cz + 1;
                }
                int vz=0;
                while(ter[vz] != calc_follow[k][fz]){
                    vz = vz + 1;
                }
                table[k][vz+1] = '#';
                fz++;
            }
            break;
        }
    }
}
}
for(ap = 0; ap < land ; ap++){
    printf("\t\t\t %c\t\t",table[ap][0]);
    for(kay = 1; kay < (sid + 1) ; kay++){
        if(table[ap][kay] == '!')
            printf("\t\t");
        else if(table[ap][kay] == '#')
            printf("%c=#\t\t",table[ap][0]);
        else{
            int mum = (int)(table[ap][kay]);
            mum -= 65;
            printf("%s\t\t",production[mum]);
        }
    }
}

```

[illegible]

```

        i_ptr++;
        printf("POP ACTION\n");
    }
    else{
        printf("\nString Not Accepted by LL(1) Parser !!\n");
        exit(0);
    }
}
else{
    for(i=0;i<sid;i++){
        if(ter[i] == her)
            break;
    }
    char produ[100];
    for(j=0;j<land;j++){
        if(him == table[j][0]){
            if (table[j][i+1] == '#'){
                printf("%c=#\n",table[j][0]);
                produ[0] = '#';
                produ[1] = '\0';
            }
            else if(table[j][i+1] != '!'){
                int mum = (int)(table[j][i+1]);
                mum -= 65;
                strcpy(produ,production[mum]);
                printf("%s\n",produ);
            }
            else{
                printf("\nString Not Accepted by LL(1)
Parser !!\n");
                exit(0);
            }
        }
    }
}
int le = strlen(produ);
le = le - 1;
if(le == 0){

```

```

        continue;
    }
    for(j=le;j>=2;j--){
        s_ptr++;
        stack[s_ptr] = produ[j];
    }
}

printf("\n\t\t\t=====
=====
=====\\n");
if (input[i_ptr] == '\\0'){
    printf("\\t\\t\\t\\t\\t\\t\\tYOUR STRING HAS BEEN ACCEPTED !!\\n");
}
else
    printf("\\n\\t\\t\\t\\t\\t\\t\\tYOUR STRING HAS BEEN REJECTED !!\\n");
printf("\\t\\t\t=====
=====
=====\\n");
}

```

void follow(char c)

```

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

```

```

    }
}

}

}

}

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

```

```

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

```

```

B=#
C=CC
C=#

First(S) = { a, b, c, d, }
First(B) = { a, #, }
First(C) = { c, #, }

-----

Follow(S) = { $, }
Follow(B) = { b, }
Follow(C) = { d, }

```

S		b	d	a	c	
	S		S=Bb	S=Cd	S=Bb	S=Cd
	B		B=#	B=aB		
	C		C=#		C=CC	

Stack	Input	Action
\$S	aaaaab\$	S=Bb
\$Bb	aaaaab\$	B=aB
\$Bba	aaaaab\$	POP ACTION
\$Bb	aaaaab\$	B=aB
\$Bba	aaaab\$	POP ACTION
\$Bb	aaaab\$	B=aB
\$Bba	aaab\$	POP ACTION
\$Bb	aaab\$	B=aB
\$Bba	aab\$	POP ACTION
\$Bb	aab\$	B=aB
\$Bba	ab\$	POP ACTION
\$Bb	ab\$	B=#
\$b	b\$	POP ACTION
\$	\$	POP ACTION

YOUR STRING HAS BEEN ACCEPTED !!

## LEARNING OUTCOME

- 79