Lab File Compiler Construction



[CSE304**]**

DEPARTMENT OF

COMPUTER SCIENCE AND ENGINEERING

BACHELOR OF TECHNOLOGY IN

COMPUTER SCIENCE AND ENGINEERING

## Submitted To: Submitted By:

Dr. Shilpi Sharma Shaurya Guliani Associate Professor A2305219086

CSE Department, ASET 6CSE2X

AMITY SCHOOL OF ENGINEERING AND TECHNOLOGY AMITY UNIVERSITY UTTAR PRADESH

NOIDA-201301

**Index**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **S.N**  **o** | **Title** | **Page no** | **Date of experiment** | **Date of submission** | **Mark s** | **Remarks** |
| **1.** | **To check whether a string is a comment or not. If it is accepted by a\*, a\*b+, or abb.** | **4** |  |  |  |  |
| **2.** | 1. **Write a program to convert given infix expression to postfix expression.** 2. **Write a program to convert given infix expression to prefix expression.** | **9** |  |  |  |  |
| **3.** | **Write a program to find number of tokens in the expression and to print identified tokens.** | **15** |  |  |  |  |
| **4.** | **Consider the following regular expressions (or Language):**  **a) 0\*1\*(0+1)**  **b) L = {w is interpreted as a binary string (ignoring the leading 0’s) is divisible by 5} for example:**  **L = {null, 0, 00,**  **000, 0101, 101,**  **1010, …}**  **Write separate programs for each of the regular expressions mentioned above.** | **21** |  |  |  |  |
| **5.** | **Write a program to eliminate Left Recursion.** | **30** |  |  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **6.** | **Write a program for Recursive Descent Parsing.** | **34** |  |  |  |  |
| **7.** | **Write a program for Shift Reduce Parser.** | **39** |  |  |  |  |
| **8.** | **Write a program to implement LL(1) parser.** | **44** |  |  |  |  |
| **9.** | **Write a program which accepts regular grammar without null productions and .** | **46** |  |  |  |  |

## Experiment – 1

**Aim:** To check whether a string is a comment or not. If it is accepted by a\*, a\*b+, or abb.

**Software used:** Visual Studio Code

### Theory:

A regular expression (sometimes called a rational expression) is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings, or string matching,

i.e. “find and replace”-like operations.

Regular expressions are a generalized way to match patterns with sequences of characters. It is used in every programming language like C++, Java and Python.

Regex are used in *Google analytics* in URL matching in supporting search and replace in most popular editors like Sublime, Notepad++, Brackets, Google Docs and Microsoft word. Regular expressions are written with the help of following symbols:

* + **Repeaters : \* , + and { } :** These symbols act as repeaters and tell the computer that the preceding character is to be used for more than just one time.
  + **The asterisk symbol ( \* ):** It tells the computer to match the preceding character (or set of characters) for 0 or more times (upto infinite).
  + **The Plus symbol ( + ):** It tells the computer to repeat the preceding character (or set of characters) for atleast one or more times(upto infinite).
  + **Wildcard – ( . ):** The dot symbol can take place of any other symbol, that is why it is called the wildcard character.
  + **Optional character – ( ? )** : This symbol tells the computer that the preceding character may

or may not be present in the string to be matched.

* + **The curly braces {…}:** It tells the computer to repeat the preceding character (or set of characters) for as many times as the value inside this bracket.

### Code:

#include <iostream> #include <string>

using namespace std;

int IsComment(string s, int len)

{

if (s[0] == '/' && s[1] == '/')

{

cout<<"\nThe string is a comment"<<endl; return 0;

}

else if (s[0] == '/' && s[1] == '\*' && s[len-2] == '\*' && s[len-1] == '/')

{

cout<<"\nThe string is a comment"<<endl; return 0;

}

else

{

cout<<"\nString is not a comment"<<endl; return -1;

}

}

void a1(string s, int len)

{

if (len == 0)

{

cout<<"The string is accepted by a\*"<<endl;

}

else {

int i = 0; while(s[i] == 'a')

{

i++;

}

if(i<len-1)

{

cout<<"\nThe string is not accepted by a\*"<<endl;

}

else

{

cout<<"\nThe string is accepted by a\*"<<endl;

}

}

}

void a2(string s, int len)

{

int i = 0;

if(len == 0)

cout<<"\nThe string is not accepted by a\*b+"<<endl; else{

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

if(i < len-1 && s[i+1] != 'b')

cout<<"\nString is not accepted by a\*b+"<<endl;

else{

while(s[i] == 'b') i++;

if(i < len-1)

cout<<"\nString is not accepted by a\*b+"<<endl; else

cout<<"\nString is accepted by a\*b+"<<endl;

}

}

}

void a3(string s, int len)

{

string s1 = "abb"; if(s == s1)

cout<<"\nThe string is accepted by abb."<<endl; else

cout<<"\nThe string is not accepted by abb."<<endl;

}

int main()

{

int len = 0;

cout<<"\nEnter the string you want to check :"<<endl; string s;

cin>>s;

len = s.length();

int x = IsComment(s, len); if(x != 0 )

{

a1(s, len);

a2(s, len);

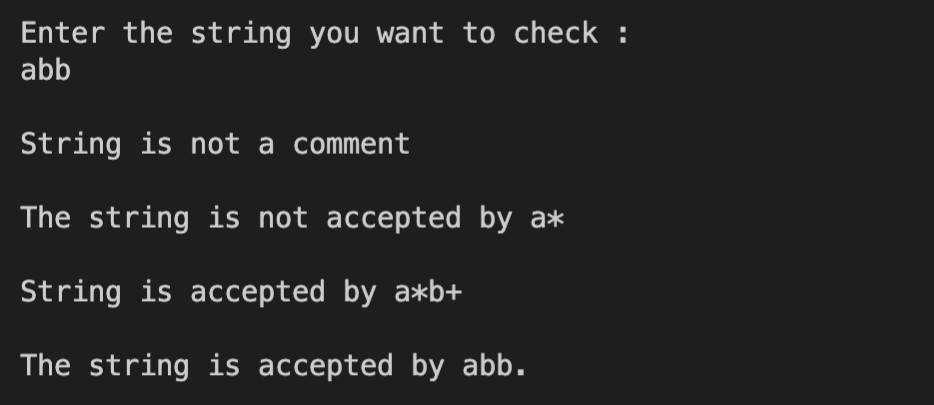
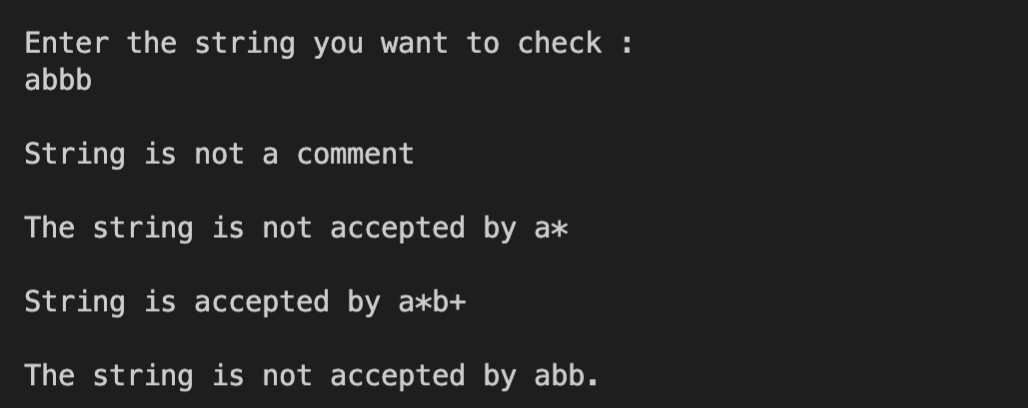
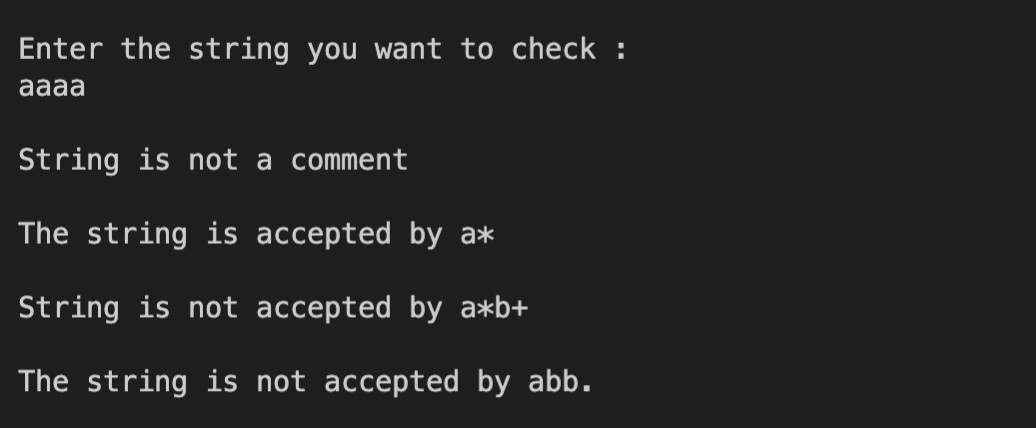
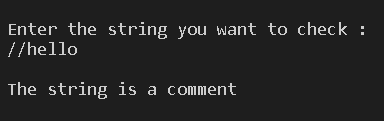
a3(s, len);

}

cout<<endl;

}

**Output:**



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science & Engineering  Amity University, Noida (UP) | | | | |
| Programme | B. Tech CSE | | Course Name | Compiler Construction |
| Course Code | [CSE304] | | Semester | 6 |
| Student Name | Shaurya Guliani | | Enrollment No. | A2305219086 |
| Marking Criteria | | | | |
| Criteria | Total Marks | Marks Obtained | | Comments |
| Concept (A) | 2 |  | |  |
| Implementation (B) | 2 |  | |  |
| Performance (C) | 2 |  | |  |
| Total | 6 |  | |  |

# Experiment-2

**Aim:** a) Write a program to convert the given infix expression to postfix expression.

b) Write a program to convert the given infix expression to prefix expression.

**Software Used:** VS Code.

## Theory:

While we use infix expressions in our day to day lives. Computers have trouble understanding this format because they need to keep in mind rules of operator precedence and brackets. For writing complex mathematical expressions, we generally prefer parentheses to make them more readable. In computers, expressions with various parentheses add unnecessary work while solving. So, to minimize the computational works, various notations have been made namely Infix, Postfix and Prefix. Prefix and Postfix expressions are easier for a computer to understand and evaluate. Given two operands 𝑎 and 𝑏 ,and an operator 𝑂,

the infix notation implies that 𝑂 will be placed in between 𝑎 and 𝑏 i.e. 𝑎𝑂𝑏 When the

operator is placed after both operands i.e., 𝑎𝑏𝑂, it is called postfix notation. And when the operator is placed before the operands i.e., 𝑂𝑎𝑏, the expression is said to be in prefix notation. Given any infix expression, we can obtain the equivalent prefix and postfix format.

## Code:

1. **Infix to Postfix.** #include<iostream> #include<stack>

using namespace std; bool isOperator(char c)

{

if(c=='+'||c=='-'||c=='\*'||c=='/'||c=='^'){ return true;

}else{

return false;

}

}

int precedence(char c)

{

if(c == '^') return 3;

else if(c == '\*' || c == '/') return 2;

else if(c == '+' || c == '-') return 1;

else

return -1;

}

string InfixToPostfix(stack<char> s, string infix)

{

string postfix;

for(int i=0;i<infix.length();i++){

if((infix[i] >= 'a' && infix[i] <= 'z')||(infix[i] >= 'A' && infix[i] <= 'Z')){ postfix+=infix[i];

}else if(infix[i] == '('){ s.push(infix[i]);

}else if(infix[i] == ')'){ while((s.top()!='(') && (!s.empty())){

char temp=s.top(); postfix+=temp; s.pop();

}

if(s.top()=='('){ s.pop();

}

} else if(isOperator(infix[i])){ if(s.empty()){

s.push(infix[i]);

}else{

if(precedence(infix[i])>precedence(s.top())){ s.push(infix[i]);

}else if((precedence(infix[i])==precedence(s.top()))&&(infix[i]=='^')){ s.push(infix[i]);

}else{

while((!s.empty())&&( precedence(infix[i])<=precedence(s.top()))){ postfix+=s.top();

s.pop();

}

s.push(infix[i]);

}

}

}

}

while(!s.empty()){ postfix+=s.top(); s.pop();

}

return postfix;

}

int main()

{

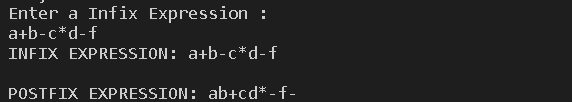
string infix\_exp, postfix\_exp; cout<<"Enter a Infix Expression :"<<endl; cin>>infix\_exp;

stack <char> stack;

cout<<"INFIX EXPRESSION: "<<infix\_exp<<endl; postfix\_exp = InfixToPostfix(stack, infix\_exp); cout<<endl<<"POSTFIX EXPRESSION: "<<postfix\_exp; return 0;

}

## Output:



1. **Infix to Prefix** #include <iostream> #include<stack> #include<algorithm> using namespace std; bool check\_char(char c){

if(c=='+'||c=='-'||c=='\*'||c=='/'||c=='^'){ return true;

}

else{

return false;

}

}

int check\_prec(char c){ if(c=='^')

return 3;

else if(c=='\*'||c=='/') return 2;

else if(c=='+'||c=='-') return 1;

else return -1;

}

string InfixtoPrefix(stack<char> s, string infix){ string prefix; reverse(infix.begin(),infix.end());

for(int i =0;i<infix.length();i++){ if(infix[i]=='('){

infix[i]=')';

}

else if (infix[i]== ')'){

infix[i]='(';

}

}

for(int i =0;i<infix.length();i++){

if((infix[i] >= 'a' && infix[i] <= 'z' || infix[i] >= 'A' && infix[i] <= 'Z')){

prefix += infix[i];

}

else if(infix[i]=='('){

s.push(infix[i]);

}

else if(infix[i]==')'){

while((s.top()!= '(') && (!s.empty())){ char temp =s.top();

prefix += temp; s.pop();

}

if(s.top()=='('){ s.pop();

}

}

else if(check\_char(infix[i])){

if(s.empty()){

s.push(infix[i]);

}

else{

if(check\_prec(infix[i])>check\_prec(s.top())){ s.push(infix[i]);

}

else if(check\_prec(infix[i])==check\_prec(s.top()) && (infix[i]=='^'))

{

while((check\_prec(infix[i])== check\_prec(s.top())) && (infix[i]=='^'))

{

prefix+=s.top(); s.pop();

}

s.push(infix[i]);

}

else if(check\_prec(infix[i])==check\_prec(s.top())){ s.push(infix[i]);

}

else {

while((!s.empty())&& (check\_prec(infix[i]))<check\_prec(s.top())){

prefix += s.top(); s.pop();

}

s.push(infix[i]);

}

}

}

}

while(!s.empty()){ prefix += s.top(); s.pop();

}

reverse(prefix.begin(),prefix.end()); return prefix;

}

int main()

{

string infix,prefix;

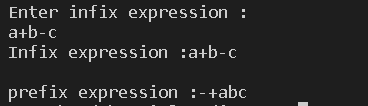
cout<<"Enter infix expression :"<<endl; cin>>infix;

stack<char>stack;

cout<<"Infix expression :"<<infix<<endl; prefix = InfixtoPrefix(stack,infix); cout<<endl<<"prefix expression :"<<prefix; return 0;

}

**Output:**



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science & Engineering  Amity University, Noida (UP) | | | | |
| Programme | B. Tech CSE | | Course Name | Compiler Construction |
| Course Code | [CSE304] | | Semester | 6 |
| Student Name | Shaurya Guliani | | Enrollment No. | A2305219086 |
| Marking Criteria | | | | |
| Criteria | Total Marks | Marks Obtained | | Comments |
| Concept (A) | 2 |  | |  |
| Implementation (B) | 2 |  | |  |
| Performance (C) | 2 |  | |  |
| Total | 6 |  | |  |

# EXPERIMENT-3

**Aim:** Write a program to find the number of tokens in the expression and to print identified tokens.

**Software Used:** VS Code.

## Theory:

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what a token can be, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators, and punctuations symbols can be considered as tokens.

### Example of tokens:

* Type token (id, number, real, . . .)
* Punctuation tokens (IF, void, return, . . .)
* Alphabetic tokens (keywords) Keywords: for, while, if etc.

Identifier: Variable name, function name, etc. Operators: '+', '++', '-' etc.

Separators: ',' ';' etc

### Example of Non-Tokens:

Comments, pre-processor directive, macros, blanks, tabs, newline, etc.

## Code:

#include<iostream> #include<conio.h> #include<string> #include<process.h> #include<ctype.h> using namespace std;

int main()

{

string program,str; string tokens[100]; string type[100];

int c,i,count=0; cout<<"Enter the program"; getline(cin,program);

for(i=0;i<program.length();i++)

{

str=""; c=0;

if(program[i]==32)

{

continue;

}

else if(program[i]=='"')

{

str=str+program[i]; i++;

while(program[i]!='"')

{

str=str+program[i]; i++;

}

str=str+program[i]; tokens[count]=str; type[count]="String"; count++;

}

else if(isdigit(program[i]))

{

while(isdigit(program[i]))

{

str=str+program[i]; i++;

}

tokens[count]=str; type[count]="Constant"; i--;

count++;

}

else if(isalpha(program[i]))

{

while(isalnum(program[i]))

{

str=str+program[i]; i++;

}

tokens[count]=str; if(str=="auto"||str=="break"||str=="case"||str=="char"|| str=="const"||str=="continue"||str=="default"||str=="do"|| str=="double"||str=="else"||str=="enum"||str=="extern"|| str=="float"||str=="for"||str=="goto"||str=="if"|| str=="int"||str=="long"||str=="register"||str=="return"|| str=="short"||str=="signed"||str=="sizeof"||str=="static"|| str=="struct"||str=="switch"||str=="typedef"||str=="union"||

str=="unsigned"||str=="void"||str=="volatile"||str=="while"||str=="cin"||str=="cout")

{

type[count]="Keyword";

}

else

{

type[count]="Identifier";

}

count++; i--;

}

else if(program[i]=='~')

{

tokens[count]="~"; type[count]="Operator"; i++;

count++;

}

else

if(program[i]=='+'||program[i]=='-'||program[i]=='\*'||program[i]=='/'||program[i]=='='||progra m[i]=='%')

{

str=str+program[i]; c++;

i++;

if(program[i]=='=')

{

str=str+program[i]; i++;

}

else

if(program[i-1]=='+'&&program[i]=='+')

{

str=str+program[i];

i++;

}

else

if(program[i-1]=='-'&&program[i]=='-')

{

str=str+program[i]; i++;

}

type[count]="Operator"; tokens[count]=str; count++;

i--;

}

else if(program[i]=='&'||program[i]=='|'||program[i]=='!')

{

str=str+program[i]; c++;

i++;

if((program[i-1]=='&'||program[i-1]=='|') && program[i]==program[i-1])

{

str=str+program[i]; i++;

}

else

if (program[i-1]=='!'&&program[i]=='=')

{

str=str+program[i]; i++;

}

type[count]="Operator"; tokens[count]=str; count++;

i--;

}

else if(program[i]=='<'||program[i]=='>')

{

str=str+program[i]; c++;

i++;

if(program[i]=='='||(program[i-1]=='<'&&program[i]=='<')||(program[i-1]=='>'&&program[i]

=='>'))

{

str=str+program[i]; i++;

}

type[count]="Operator"; tokens[count]=str; count++;

i--;

}

else if(program[i]==','||program[i]=='<'||program[i]=='>'||program[i]=='{'|| program[i]=='}'||program[i]==';'||program[i]=='{'||program[i]=='}'|| program[i]=='('||program[i]==')'||program[i]=='#')

{

str=str+program[i]; tokens[count]=str; type[count]="Punctuator";

count++;

}

}

cout<<"No. of Tokens are: "<<count<<endl; for(i=0;i<count;i++)

{

cout<<tokens[i]<<"--->"<<type[i]<<endl;

}

getch(); return 0;

}

## Output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science & Engineering  Amity University, Noida (UP) | | | | |
| Programme | B. Tech CSE | | Course Name | Compiler Construction |
| Course Code | [CSE304] | | Semester | 6 |
| Student Name | Shaurya Guliani | | Enrollment No. | A2305219086 |
| Marking Criteria | | | | |
| Criteria | Total Marks | Marks Obtained | | Comments |
| Concept (A) | 2 |  | |  |
| Implementation (B) | 2 |  | |  |
| Performance (C) | 2 |  | |  |
| Total | 6 |  | |  |

**Experiment – 4**

**Aim:** Consider the following regular expressions (or Language): c) 0\*1\*(0+1)

d) L = {w is interpreted as a binary string (ignoring the leading 0’s) is divisible by 5} for example:

L = {null, 0, 00, 000, 0101, 101, 1010, …}

Write separate programs for each of the regular expressions mentioned above.

**Software used:** Visual Studio Code

### Theory:

A regular expression (sometimes called a rational expression) is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings, or string matching,

i.e. “find and replace”-like operations.

Regular expressions are a generalized way to match patterns with sequences of characters. It is used in every programming language like C++, Java and Python.

Regex are used in *Google analytics* in URL matching in supporting search and replace in most popular editors like Sublime, Notepad++, Brackets, Google Docs and Microsoft word. Regular expressions are written with the help of following symbols:

* + **Repeaters : \* , + and { } :** These symbols act as repeaters and tell the computer that the preceding character is to be used for more than just one time.
  + **The asterisk symbol ( \* ):** It tells the computer to match the preceding character (or set of characters) for 0 or more times (upto infinite).
  + **The Plus symbol ( + ):** It tells the computer to repeat the preceding character (or set of characters) for atleast one or more times(upto infinite).
  + **Wildcard – ( . ):** The dot symbol can take place of any other symbol, that is why it is called the wildcard character.
  + **Optional character – ( ? )** : This symbol tells the computer that the preceding character may

or may not be present in the string to be matched.

* + **The curly braces {…}:** It tells the computer to repeat the preceding character (or set of characters) for as many times as the value inside this bracket.

### s

**Code:**

### Given regular expression: 0\*1\*(0+1)

#include <iostream> #include <string.h> using namespace std;

void Q0(string,int); void Q1(string,int); void Q2(string,int);

void Q3(string,int);

void Q0(string S, int n){ if(n == S.length())

{

cout<<" String not accepted "; return;

}

if(S[n]=='0')

{

cout<<" Q2 -> ";

Q2(S, n+1);

}

if(S[n] == '1')

{

cout<<" Q1 -> ";

Q1(S, n+1);

}

}

void Q1(string S, int n)

{

if(n == S.length())

{

cout<<" String accepted "; return;

}

if(S[n] == '1')

{

cout<<" Q1 -> ";

Q1(S, n+1);

}

if(S[n] == '0')

{

cout<<" Q3 -> ";

Q3(S, n+1);

}

}

void Q2(string S, int n)

{

if(n == S.length())

{

cout<<" String accepted "; return;

}

if(S[n] == '0')

{

cout<<" Q2 -> ";

Q2(S, n+1);

}

if(S[n] == '1')

{

cout<<" Q1 -> ";

Q1(S, n+1);

}

}

void Q3(string S, int n)

{

if(n == S.length())

{

cout<<" String accepted "; return;

}

if(S[n] == '1' || S[n] == '0')

{

cout<<" Dead State -> "; cout<<" String not accepted "; return;

}

}

int main(){

string S; cout<<endl;

cout<<"Enter the string which you want to check : "; cin>>S;

cout<<endl;

cout<<"The transition of states is as follows: "<<endl; cout<<""<<endl;

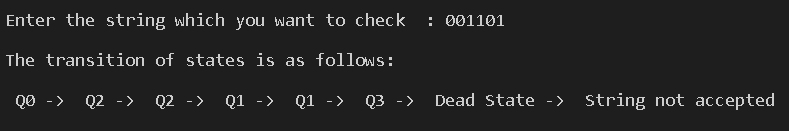
cout<<" Q0 -> ";

Q0(S, 0);

cout<<" "<<endl; cout<<endl; return 0;

}

### Output:



1. **The given language is :**

### L = {w is interpreted as a binary string (ignoring the leading 0’s) is divisible by 5} for example:

**L = {null, 0, 00, 000, 0101, 101, 1010, …}**

#include <iostream> #include <string.h> using namespace std;

void Q0(string,int); void Q1(string,int); void Q2(string,int); void Q3(string,int); void Q4(string,int);

void Q0(string S, int n){

if(n == S.length())

{

cout<<" String is accepted "; return;

}

if(S[n] == '0')

{

cout<<" Q0 -> ";

Q0(S, n+1);

}

if(S[n] == '1')

{

cout<<" Q1 -> ";

Q1(S, n+1);

}

}

void Q1(string S, int n)

{

if(n == S.length())

{

cout<<" String is not accepted ";

return;

}

if(S[n] == '1')

{

cout<<" Q3 -> ";

Q3(S, n+1);

}

if(S[n] == '0')

{

cout<<" Q2 -> ";

Q2(S, n+1);

}

}

void Q2(string S, int n)

{

if(n == S.length())

{

cout<<" String is not accepted "; return;

}

if(S[n] == '1')

{

cout<<" Q0 -> ";

Q0(S, n+1);

}

if(S[n] == '0')

{

cout<<" Q4 -> ";

Q4(S, n+1);

}

}

void Q3(string S, int n)

{

if(n == S.length())

{

cout<<" Invalid String "; return;

}

if(S[n] == '1')

{

cout<<" Q2 -> ";

Q2(S, n+1);

}

if(S[n] == '0')

{

cout<<" Q1 -> ";

Q1(S, n+1);

}

}

void Q4(string S, int n)

{

if(n == S.length())

{

cout<<" String not accepted "; return;

}

if(S[n] == '1')

{

cout<<" Q4 -> ";

Q4(S, n+1);

}

if(S[n] == '0')

{

cout<<" Q3 -> ";

Q3(S, n+1);

}

}

int main(){

string S;

cout<<endl;

cout<<"Enter the string which you want to check : "; cin>>S;

cout<<endl;

cout<<"The transition of states is as follows: "<<endl; cout<<endl;

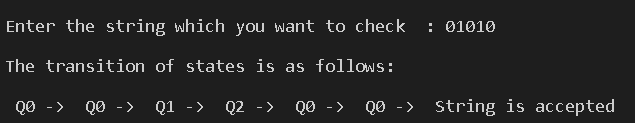
cout<<" Q0 -> ";

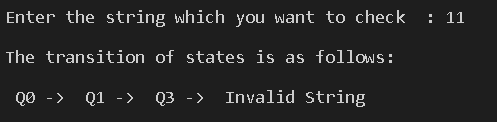
Q0(S, 0);

cout<<"\n"<<endl; return 0;

}

**Output:**





|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science & Engineering  Amity University, Noida (UP) | | | | |
| Programme | B. Tech CSE | | Course Name | Compiler Construction |
| Course Code | [CSE304] | | Semester | 6 |
| Student Name | Shaurya Guliani | | Enrollment No. | A2305219086 |
| Marking Criteria | | | | |
| Criteria | Total Marks | Marks Obtained | | Comments |
| Concept (A) | 2 |  | |  |
| Implementation (B) | 2 |  | |  |
| Performance (C) | 2 |  | |  |
| Total | 6 |  | |  |

# EXPERIMENT – 5

**Aim:** Write a program to eliminate Left Recursion.

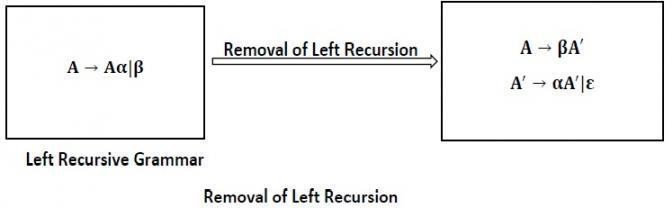
**Software Used:** VS Code.

## Theory:

A grammar in the form G = (V, T, S, P) is said to be in left recursive form if it has the production rules of the form A → A α |β.

* + In the production rule above, the variable in the left side occurs at the first position on the right-side production, due to which the left recursion occurs.
  + If we have a left recursion in our grammar, then it leads to infinite recursion, due to which we cannot generate the given string.

Left Recursion can be eliminated by introducing new non-terminal A such that.



This type of recursion is also called **Immediate Left Recursion**.

The general form for left recursion is

A → Aα1|Aα2| …. |Aαm|β1|β2| ….. βn can be replaced by

A → β1A′|β2A′| … . . | … . . |βnA′

A → α1A′|α2A′| … . . |αmA′|ε

**Code:** #include<iostream> #include<string> using namespace std; int main(){

string ip,op1,op2,temp; int sizes[10] = {};

char c; int n,j,l;

cout<<"Enter the Parent : "; cin>>c;

ip.push\_back(c); op1 += ip + "\'->";

ip += "->";

op2+=ip;

cout<<"number of productions : "; cin>>n;

for(int i=0;i<n;i++){

cout<<"Enter Production "<<i+1<<" : "; cin>>temp;

sizes[i] = temp.size(); ip+=temp;

if(i!=n-1)

ip += "|";

}

cout<<"Production Rule : "<<ip<<endl; for(int i=0,k=3;i<n;i++){

if(ip[0] == ip[k]){

cout<<"Production "<<i+1<<" has left recursion."<<endl; if(ip[k] != '#'){

for(l=k+1;l<k+sizes[i];l++) op1.push\_back(ip[l]); k=l+1; op1.push\_back(ip[0]);

op1 += "\'|";

}

}else{

cout<<"Production "<<i+1<<" does not have left recursion."<<endl; if(ip[k] != '#'){

for(j=k;j<k+sizes[i];j++) op2.push\_back(ip[j]); k=j+1; op2.push\_back(ip[0]); op2 += "\'|";

}else{

op2.push\_back(ip[0]); op2 += "\'";

}

}

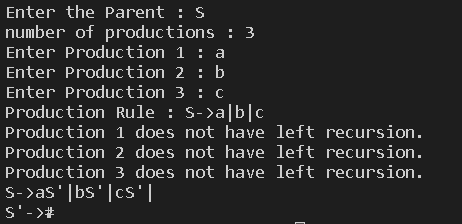
}

op1 += "#";

cout<<op2<<endl; cout<<op1<<endl; return 0;

}

**Output:**



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science & Engineering  Amity University, Noida (UP) | | | | |
| Programme | B. Tech CSE | | Course Name | Compiler Construction |
| Course Code | [CSE304] | | Semester | 6 |
| Student Name | Shaurya Guliani | | Enrollment No. | A2305219086 |
| Marking Criteria | | | | |
| Criteria | Total Marks | Marks Obtained | | Comments |
| Concept (A) | 2 |  | |  |
| Implementation (B) | 2 |  | |  |
| Performance (C) | 2 |  | |  |
| Total | 6 |  | |  |

# EXPERIMENT – 6

**Aim:** Write a program for Recursive Descent Parser.

**Software Used:** VS Code.

## Theory:

Parsing is the process to determine whether the start symbol can derive the program or not. If the Parsing is successful then the program is a valid program otherwise the program is invalid.

There are generally two types of Parsers:

### Top-Down Parsers:

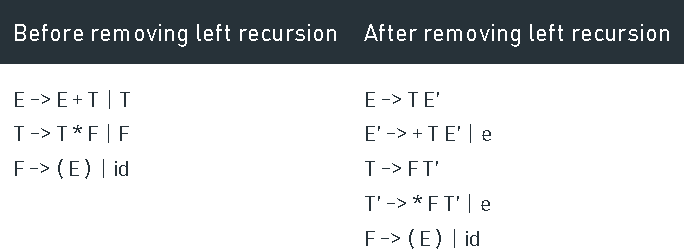
* + In this Parsing technique we expand the start symbol to the whole program.
  + Recursive Descent and LL parsers are the Top-Down parsers.

### Bottom-Up Parsers:

* + In this Parsing technique we reduce the whole program to start symbol.
  + Operator Precedence Parser, LR(0) Parser, SLR Parser, LALR Parser and CLR Parser are the Bottom-Up parsers.

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 non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. 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 backtracking is known as **predictive parsing**.

This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.



\*\*Here e is Epsilon

For Recursive Descent Parser, we are going to write one program for every variable.

## Code:

#include <iostream> #include <stdlib.h> using namespace std; int count = 0;

void E();

void Ed();

void T();

void Td();

void F();

string expr;

int main() { cin >> expr;

int l = expr.length(); expr += "$";

E();

if (l == count)

cout << "Accepted" << endl; else

cout << "Rejected" << endl;

}

void E() {

cout << "E->TE'" << endl; T();

Ed();

}

void Ed() {

if (expr[count] == '+') { count++;

cout << "E'->+TE'" << endl; T();

Ed();

}

else if (expr[count] == '-') { count++;

cout << "E'->-TE'" << endl; T();

Ed();

}

else {

cout << "E'->null" << endl;

}

}

void T() {

cout << "T->FT'" << endl; F();

Td();

}

void Td() {

if (expr[count] == '\*') { count++;

cout << "T'->\*FT'" << endl;

F();

Td();

}

else if (expr[count] == '/') { count++;

cout << "T'->/FT'" << endl; F();

Td();

}

else {

cout << "T'->null" << endl;

}

}

void F() {

if (isalpha(expr[count])) { count++;

cout << "F->id" << endl;

} else if (isdigit(expr[count])) { count++;

cout << "F->digit" << endl;

} else if (expr[count] == '(') { count++;

cout << "F->(E)" << endl; E();

if (expr[count] != ')') {

cout << "Rejected" << endl; exit(0);

}

count++;

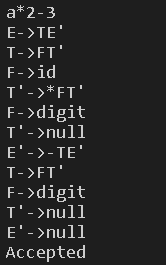
} else {

cout << "Rejected" << endl; exit(0);

}

}

**Output:**



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science & Engineering  Amity University, Noida (UP) | | | | |
| Programme | B. Tech CSE | | Course Name | Compiler Construction |
| Course Code | [CSE304] | | Semester | 6 |
| Student Name | Shaurya Guliani | | Enrollment No. | A2305219086 |
| Marking Criteria | | | | |
| Criteria | Total Marks | Marks Obtained | | Comments |
| Concept (A) | 2 |  | |  |
| Implementation (B) | 2 |  | |  |
| Performance (C) | 2 |  | |  |
| Total | 6 |  | |  |

# EXPERIMENT – 7

**Aim:** Write a program for Shift Reducer Parser.

**Software Used:** VS Code.

## Theory:

Shift Reduce Parser is a type of Bottom-Up Parser. It generates the Parse Tree from Leaves to the Root. In Shift Reduce Parser, the input string will be reduced to the starting symbol. This reduction can be produced by handling the rightmost derivation in reverse, i.e., from starting symbol to the input string.

Shift Reduce Parser requires two Data Structures

* + Input Buffer
  + Stack

There are the various steps of Shift Reduce Parsing which are as follows −

* + It uses a stack and an input buffer.
  + Insert $ at the bottom of the stack and the right end of the input string in Input Buffer.



* + **Shift** − Parser shifts zero or more input symbols onto the stack until the handle is on top of the stack.
  + **Reduce** − Parser reduce or replace the handle on top of the stack to the left side of production, i.e., R.H.S. of production is popped, and L.H.S is pushed.
  + **Accept** − Step 3 and Step 4 will be repeated until it has detected an error or until the stack includes start symbol (S) and input Buffer is empty, i.e., it contains $.



## Code:

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

int z = 0, i = 0, j = 0, c = 0;

char a[16], ac[20], stk[15], act[10]; void check()

{

strcpy(ac,"REDUCE TO E -> "); for(z = 0; z < c; z++)

{

if(stk[z] == '4')

{

printf("%s4", ac);

stk[z] = 'E';

stk[z + 1] = '\0'; printf("\n$%s\t%s$\t", stk, a);

}

}

for(z = 0; z < c - 2; z++)

{

if(stk[z] == '2' && stk[z + 1] == 'E' &&

stk[z + 2] == '2')

{

printf("%s2E2", ac);

stk[z] = 'E';

stk[z + 1] = '\0';

stk[z + 2] = '\0'; printf("\n$%s\t%s$\t", stk, a); i = i - 2;

}

}

for(z = 0; z < c - 2; z++)

{

if(stk[z] == '3' && stk[z + 1] == 'E' &&

stk[z + 2] == '3')

{

printf("%s3E3", ac);

stk[z]='E';

stk[z + 1]='\0';

stk[z + 1]='\0'; printf("\n$%s\t%s$\t", stk, a); i = i - 2;

}

}

return ;

}

int main()

{

cout<<"GRAMMAR is -\nE->2E2 \nE->3E3 \nE->4\n"; strcpy(a,"32423");

c=strlen(a); strcpy(act,"SHIFT");

cout<<"\nstack \t input \t action"; cout<<"\n$\t"<<a<<"\t";

for(i = 0; j < c; i++, j++)

{

cout<< act; stk[i] = a[j];

stk[i + 1] = '\0';

a[j]=' ';

cout<<"\n$"<<"\t"<<a<<"$\t"; check();

}

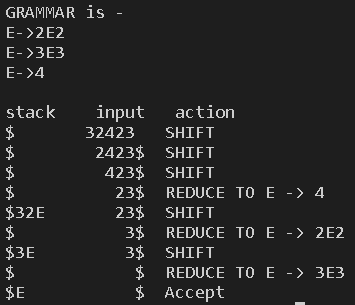
check();

if(stk[0] == 'E' && stk[1] == '\0') cout<<"Accept\n";

else //else reject cout<<"Reject\n";

}

**Output:**



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science & Engineering  Amity University, Noida (UP) | | | | |
| Programme | B. Tech CSE | | Course Name | Compiler Construction |
| Course Code | [CSE304] | | Semester | 6 |
| Student Name | Shaurya Guliani | | Enrollment No. | A2305219086 |
| Marking Criteria | | | | |
| Criteria | Total Marks | Marks Obtained | | Comments |
| Concept (A) | 2 |  | |  |
| Implementation (B) | 2 |  | |  |
| Performance (C) | 2 |  | |  |
| Total | 6 |  | |  |

# EXPERIMENT – 8

**Aim:** Write a program to implement LL(1) parser on production.

**Software Used:** VS Code.

## 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:

1. Top-Down Parser with Backtracking
2. 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.

## Code:

#include<stdio.h> #include<string.h> #define TSIZE 128

// table[i][j] stores

// the index of production that must be applied on

// ith varible if the input is

// jth nonterminal

int table[100][TSIZE];

// stores all list of terminals

// the ASCII value if use to index terminals

// terminal[i] = 1 means the character with

// ASCII value is a terminal char terminal[TSIZE];

// stores all list of terminals

// only Upper case letters from 'A' to 'Z'

// can be nonterminals

// nonterminal[i] means ith alphabet is present as

// nonterminal is the grammar char nonterminal[26];

// structure to hold each production

// str[] stores the production

// len is the length of production struct product {

char str[100]; int len;

}pro[20];

// no of productions in form A->ß int no\_pro;

char first[26][TSIZE]; char follow[26][TSIZE];

// stores first of each production in form A->ß char first\_rhs[100][TSIZE];

// check if the symbol is nonterminal int isNT(char c) {

return c >= 'A' && c <= 'Z';

}

// reading data from the file void readFromFile() {

FILE\* fptr;

fptr = fopen("text.txt", "r"); char buffer[255];

int i; int j;

while (fgets(buffer, sizeof(buffer), fptr)) { printf("%s", buffer);

j = 0;

nonterminal[buffer[0] - 'A'] = 1;

for (i = 0; i < strlen(buffer) - 1; ++i) { if (buffer[i] == '|') {

++no\_pro;

pro[no\_pro - 1].str[j] = '\0'; pro[no\_pro - 1].len = j;

pro[no\_pro].str[0] = pro[no\_pro - 1].str[0]; pro[no\_pro].str[1] = pro[no\_pro - 1].str[1]; pro[no\_pro].str[2] = pro[no\_pro - 1].str[2]; j = 3;

}

else {

pro[no\_pro].str[j] = buffer[i];

++j;

if (!isNT(buffer[i]) && buffer[i] != '-' && buffer[i] != '>') { terminal[buffer[i]] = 1;

}

}

}

pro[no\_pro].len = j;

++no\_pro;

}

}

void add\_FIRST\_A\_to\_FOLLOW\_B(char A, char B) { int i;

for (i = 0; i < TSIZE; ++i) {

if (i != '^')

follow[B - 'A'][i] = follow[B - 'A'][i] || first[A - 'A'][i];

}

}

void add\_FOLLOW\_A\_to\_FOLLOW\_B(char A, char B) { int i;

for (i = 0; i < TSIZE; ++i) {

if (i != '^')

follow[B - 'A'][i] = follow[B - 'A'][i] || follow[A - 'A'][i];

}

}

void FOLLOW() { int t = 0;

int i, j, k, x;

while (t++ < no\_pro) {

for (k = 0; k < 26; ++k) {

if (!nonterminal[k]) continue; char nt = k + 'A';

for (i = 0; i < no\_pro; ++i) {

for (j = 3; j < pro[i].len; ++j) {

if (nt == pro[i].str[j]) {

for (x = j + 1; x < pro[i].len; ++x) { char sc = pro[i].str[x];

if (isNT(sc)) { add\_FIRST\_A\_to\_FOLLOW\_B(sc, nt); if (first[sc - 'A']['^'])

continue;

}

else {

follow[nt - 'A'][sc] = 1;

}

break;

}

if (x == pro[i].len) add\_FOLLOW\_A\_to\_FOLLOW\_B(pro[i].str[0], nt);

}

}

}

}

}

}

void add\_FIRST\_A\_to\_FIRST\_B(char A, char B) { int i;

for (i = 0; i < TSIZE; ++i) {

if (i != '^') {

first[B - 'A'][i] = first[A - 'A'][i] || first[B - 'A'][i];

}

}

}

void FIRST() { int i, j;

int t = 0;

while (t < no\_pro) {

for (i = 0; i < no\_pro; ++i) {

for (j = 3; j < pro[i].len; ++j) { char sc = pro[i].str[j];

if (isNT(sc)) {

add\_FIRST\_A\_to\_FIRST\_B(sc, pro[i].str[0]); if (first[sc - 'A']['^'])

continue;

}

else {

first[pro[i].str[0] - 'A'][sc] = 1;

}

break;

}

if (j == pro[i].len) first[pro[i].str[0] - 'A']['^'] = 1;

}

++t;

}

}

void add\_FIRST\_A\_to\_FIRST\_RHS B(char A, int B) { int i;

for (i = 0; i < TSIZE; ++i) {

if (i != '^')

first\_rhs[B][i] = first[A - 'A'][i] || first\_rhs[B][i];

}

}

// Calculates FIRST(ß) for each A->ß void FIRST\_RHS() {

int i, j; int t = 0;

while (t < no\_pro) {

for (i = 0; i < no\_pro; ++i) {

for (j = 3; j < pro[i].len; ++j) { char sc = pro[i].str[j];

if (isNT(sc)) { add\_FIRST\_A\_to\_FIRST\_RHS B(sc, i); if (first[sc - 'A']['^'])

continue;

}

else {

first\_rhs[i][sc] = 1;

}

break;

}

if (j == pro[i].len) first\_rhs[i]['^'] = 1;

}

++t;

}

}

int main() { readFromFile();

follow[pro[0].str[0] - 'A']['$'] = 1; FIRST();

FOLLOW();

FIRST\_RHS();

int i, j, k;

// display first of each variable printf("\n");

for (i = 0; i < no\_pro; ++i) {

if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) { char c = pro[i].str[0];

printf("FIRST OF %c: ", c); for (j = 0; j < TSIZE; ++j) {

if (first[c - 'A'][j]) { printf("%c ", j);

}

}

printf("\n");

}

}

// display follow of each variable printf("\n");

for (i = 0; i < no\_pro; ++i) {

if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) { char c = pro[i].str[0];

printf("FOLLOW OF %c: ", c); for (j = 0; j < TSIZE; ++j) {

if (follow[c - 'A'][j]) { printf("%c ", j);

}

}

printf("\n");

}

}

// display first of each variable ß

// in form A->ß printf("\n");

for (i = 0; i < no\_pro; ++i) { printf("FIRST OF %s: ", pro[i].str); for (j = 0; j < TSIZE; ++j) {

if (first\_rhs[i][j]) {

printf("%c ", j);

}

}

printf("\n");

}

// the parse table contains '$'

// set terminal['$'] = 1

// to include '$' in the parse table terminal['$'] = 1;

// the parse table do not read '^'

// as input

// so we set terminal['^'] = 0

// to remove '^' from terminals terminal['^'] = 0;

// printing parse table printf("\n");

printf("\n\t\*\*\*\*\*\* LL(1) PARSING TABLE \*\*\*\*\*\*\*\n"); printf("\t \n");

printf("%-10s", "");

for (i = 0; i < TSIZE; ++i) {

if (terminal[i]) printf("%-10c", i);

}

printf("\n"); int p = 0;

for (i = 0; i < no\_pro; ++i) {

if (i != 0 && (pro[i].str[0] != pro[i - 1].str[0])) p = p + 1;

for (j = 0; j < TSIZE; ++j) {

if (first\_rhs[i][j] && j != '^') { table[p][j] = i + 1;

}

else if (first\_rhs[i]['^']) {

for (k = 0; k < TSIZE; ++k) {

if (follow[pro[i].str[0] - 'A'][k]) { table[p][k] = i + 1;

}

}

}

}

}

k = 0;

for (i = 0; i < no\_pro; ++i) {

if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {

printf("%-10c", pro[i].str[0]); for (j = 0; j < TSIZE; ++j) {

if (table[k][j]) {

printf("%-10s", pro[table[k][j] - 1].str);

}

else if (terminal[j]) { printf("%-10s", "");

}

}

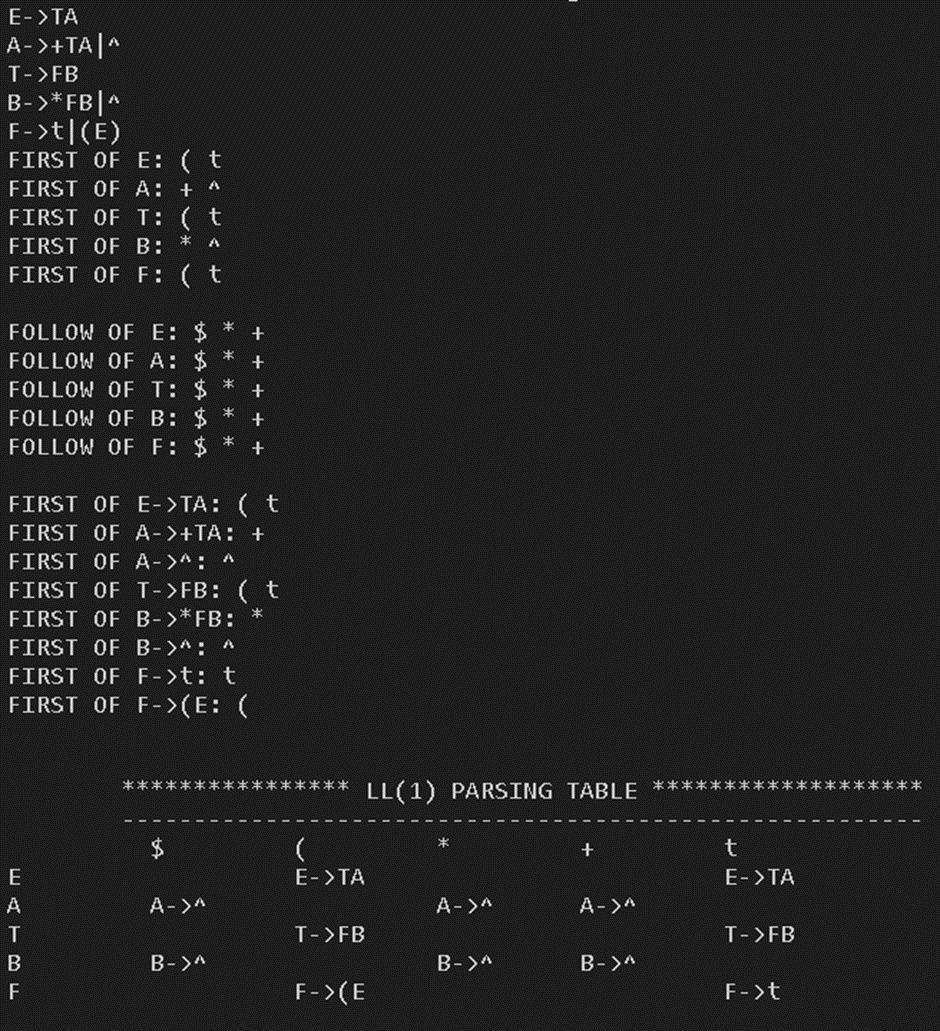
++k;

printf("\n");

}

}

}

**Output:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science & Engineering  Amity University, Noida (UP) | | | | |
| Programme | B. Tech CSE | | Course Name | Compiler Construction |
| Course Code | [CSE304] | | Semester | 6 |
| Student Name | Shaurya Guliani | | Enrollment No. | A2305219086 |
| Marking Criteria | | | | |
| Criteria | Total Marks | Marks Obtained | | Comments |
| Concept (A) | 2 |  | |  |
| Implementation (B) | 2 |  | |  |
| Performance (C) | 2 |  | |  |
| Total | 6 |  | |  |

# EXPERIMENT – 9

**Aim:** Write a program which accepts a regular grammar with no left recursion, and no null- production rules, and then it accepts a string and reports whether the string is accepted by the grammar or not.

**Software Used:** VS Code.

## Theory:

Regular grammar generates regular language. They have a single non-terminal on the

left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a non-terminal.

The productions must be in the form –

**A** ⇢ **xB A** ⇢ **x A** ⇢ **Bx**

where A, B ∈ Variable(V) and x ∈ T\* i.e., string of terminals. Types of regular grammar:

### Left linear grammar (LLG) –

In LLG, the productions are in the form if all the productions are of the form A ⇢ Bx

A ⇢ x

where A, B ∈ V and x ∈ T\*

### Right linear grammar (RLG) –

In RLG, the productions are in the form if all the productions are of the form A ⇢ xB

A ⇢ x

where A, B ∈ V and x ∈ T\*

**Code:** #include<iostream> #include<stack>

using namespace std; string l;

string z; int flag; int j=0; int E();

int Edash();

void match(string t);

int main(){

cout<<"enter the input string = "; cin>>l;

E();

if (flag == 1) {

cout<<"Parsing unSuccessful"; } else if (l[j] == '$'){

cout<<"Parsing Successful"; }} int E(){

if (l[j] == 'i') {

match("i");

Edash(); }

}

void match(string t){ z=l[j];

if(z==t){ j=j+1;

}else flag=1;}

int Edash(){ if(l[j]=='+'){

match("+");

match("i");

Edash();

}else return 0;}

## Output:



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science & Engineering  Amity University, Noida (UP) | | | | |
| Programme | B. Tech CSE | | Course Name | Compiler Construction |
| Course Code | [CSE304] | | Semester | 6 |
| Student Name | Shaurya Guliani | | Enrollment No. | A2305219086 |
| Marking Criteria | | | | |
| Criteria | Total Marks | Marks Obtained | | Comments |
| Concept (A) | 2 |  | |  |
| Implementation (B) | 2 |  | |  |
| Performance (C) | 2 |  | |  |
| Total | 6 |  | |  |