**INDEX**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **S.**  **No.** | **Category of exp.** | **Code** | **Exp.**  **No.** | **Name of experiment** | **Date of allotment of exp.** | **Date of**  **Evaluan** | **Max.**  **marks** | **Marks**  **Obt.** | **Sign.**  **Of**  **faculty** |
|  | **Mandato-ry**  **Exp.** | **LR (10)** | **1** | WAP to check whether string is accepted or not for entered grammar |  |  | **1** |  |  |
|  | **2** | WAP To convert infix expression into postfix expression |  |  | **1** |  |  |
|  | **3** | WAP To convert infix expression into prefix expression |  |  | **1** |  |  |
|  | **4** | WAP To count the number of tokens in an expression |  |  | **1** |  |  |
|  | **5** | WAP to convert Regular Expression to NFA. |  |  | **1** |  |  |
|  | **6** | WAP to convert NFA to DFA. |  |  | **1** |  |  |
|  | **7** | WAP to calculate Leading and Trailing. |  |  | **1** |  |  |
|  | **8** | WAP to calculate First and Follow |  |  | **1** |  |  |
|  | **9** | WAP to design a shift reduce parser for  E -> E + E | E \* E | (E) | id |  |  | **1** |  |  |
|  | **10** | WAP to design a stop down parser for E -> E + E | E \* E | (E) | id |  |  | **1** |  |  |
|  | **Design Based Open Ended experiment** | **PR (10)** |  |  |  |  | **10** |  |  |
|  | **Viva** | **Viva (5)** |  |  |  |  | **5** |  |  |

**EXPERIMENT – 1**

**DATE:**

**OBJECTIVE:** WAP to check whether string is accepted or not for entered grammar

**EQUIPMENT/SOFTWARE USED:**

|  |  |  |
| --- | --- | --- |
| S.No. | Hardware | Software |
| 1. | I7 Processor | OS (Windows 8) |
| 2. | 8 GB RAM | Microsoft Word |
| 3. | Keyboard | TURBO C/C++ |
| 4. | Mouse |  |
| 5. | Monitor |  |
| 6. | Printer |  |

**THEORY:**

In formal language theory, a grammar (when the context is not given, often called a formal grammar for clarity) is a set of production rules for strings in a formal language. The rules describe how to form strings from the language's alphabet that are valid according to the language's syntax. A grammar does not describe the meaning of the strings or what can be done with them in whatever context—only their form.

A grammar mainly consists of a set of rules for transforming strings. To generate a string in the language, one begins with a string consisting of only a single start symbol. The production rules are then applied in any order, until a string that contains neither the start symbol nor designated nonterminal symbols is produced. A production rule is applied to a string by replacing one occurrence of its left-hand side in the string by its right-hand side (cf. the operation of the theoretical Turing machine). The language formed by the grammar consists of all distinct strings that can be generated in this manner. Any particular sequence of production rules on the start symbol yields a distinct string in the language. If there are multiple ways of generating the same single string, the grammar is said to be ambiguous.

**PROCEDURE:**

#include<iostream.h>

#include<conio.h>

#include<string.h>

#include<stdio.h>

void main()

{

char string[20];

int state=0,count=0;

clrscr();

cout<<"the grammar is: S->aS, S->Sb, S->ab \n";

cout<<"enter the string to be checked \n";

gets(string);

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

{

switch(state)

{

case 0: if (string[count]=='a')

state=1;

else

state=3;

break;

case 1: if (string[count]=='a')

state=1;

else if(string[count]=='b')

state=2;

else

state=3;

break;

case 2: if (string[count]=='b')

state=2;

else

state=3;

break;

default: break;

}

count++;

if(state==3)

break;

}

if(state==2)

cout<<"string is accepted";

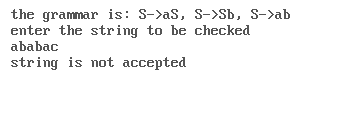
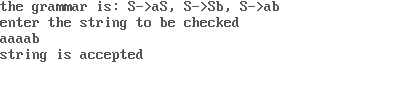
else

cout<<"string is not accepted";

getch();

}

**OUTPUT:**

****

**EXPERIMENT –2**

**DATE:**

**OBJECTIVE:** WAP To convert infix expression into postfix expression

**EQUIPMENT/SOFTWARE USED:**

|  |  |  |
| --- | --- | --- |
| S.No. | Hardware | Software |
| 1. | I7 Processor | OS (Windows 8) |
| 2. | 8 GB RAM | Microsoft Word |
| 3. | Keyboard | TURBO C/C++ |
| 4. | Mouse |  |
| 5. | Monitor |  |
| 6. | Printer |  |

**THEORY:**

Reverse Polish notation (RPN) is a mathematical notation in which every [operator](http://en.wikipedia.org/wiki/Operation_(mathematics)) follows all of its [operands](http://en.wikipedia.org/wiki/Operand), in contrast to [Polish notation](http://en.wikipedia.org/wiki/Polish_notation), which puts the operator in the prefix position. It is also known as postfix notation and is parenthesis-free as long as operator [arities](http://en.wikipedia.org/wiki/Arity) are fixed. The description "Polish" refers to the nationality of logician [Jan £ukasiewicz](http://en.wikipedia.org/wiki/Jan_%C5%81ukasiewicz), who invented (prefix) Polish notation in the 1920s.

The reverse Polish scheme was proposed in 1954 by Burks, Warren, and Wright and was independently reinvented by [F. L. Bauer](http://en.wikipedia.org/wiki/F._L._Bauer) and [E. W. Dijkstra](http://en.wikipedia.org/wiki/E._W._Dijkstra) in the early 1960s to reduce computer memory access and utilize the [stack](http://en.wikipedia.org/wiki/Stack_(data_structure)) to evaluate expressions. The algorithms and notation for this scheme were extended by [Australian](http://en.wikipedia.org/wiki/Australia) philosopher and computer scientist [Charles Hamblin](http://en.wikipedia.org/wiki/Charles_Leonard_Hamblin) in the mid-1950s.

In [computer science](http://en.wikipedia.org/wiki/Computer_science), postfix notation is often used in [stack-based](http://en.wikipedia.org/wiki/Stack-based_language) and [concatenative programming](http://en.wikipedia.org/wiki/Concatenative_programming) languages. It is also common in [dataflow](http://en.wikipedia.org/wiki/Dataflow_programming) and [pipeline](http://en.wikipedia.org/wiki/Pipeline_(software))-based systems, including[Unix pipelines](http://en.wikipedia.org/wiki/Unix_pipeline).

**PROCEDURE:**

#DEFINE SIZE 50

#include <ctype.h>

char s[SIZE];

int top = -1; /\* Global declarations \*/

push(char elem) { /\* Function for PUSH operation \*/

s[++top] = elem;

}

char pop() { /\* Function for POP operation \*/

return (s[top--]);

}

intpr(char elem) { /\* Function for precedence \*/

switch (elem) {

case '#':

return 0;

case '(':

return 1;

case '+':

case '-':

return 2;

case '\*':

case '/':

return 3;

}

}

main() { /\* Main Program \*/

charinfx[50], pofx[50], ch, elem;

inti = 0, k = 0;

clrscr();

printf("\n\nRead the Infix Expression ? ");

scanf("%s", infx);

push('#');

while ((ch = infx[i++]) != '\0') {

if (ch == '(')

push(ch);

else if (isalnum(ch))

pofx[k++] = ch;

else if (ch == ')') {

while (s[top] != '(')

pofx[k++] = pop();

elem = pop(); /\* Remove ( \*/

} else { /\* Operator \*/

while (pr(s[top]) >= pr(ch))

pofx[k++] = pop();

push(ch);

}

}

while (s[top] != '#') /\* Pop from stack till empty \*/

pofx[k++] = pop();

pofx[k] = '\0'; /\* Make pofx as valid string \*/

printf("\n\nGiven Infix Expn: %s Postfix Expn: %s\n", infx, pofx);

getch();

}

**OBSERVATIONS AND DISCUSSIONS:**

The output from the above code can be shown using below snapshot



**RESULT:**

The program is successfully written and created in c language

**EXPERIMENT –3**

**DATE:**

**OBJECTIVE: WAP To convert infix expression into prefix expression**

**EQUIPMENT/SOFTWARE USED:**

|  |  |  |
| --- | --- | --- |
| S.No. | Hardware | Software |
| 1. | I7 Processor | OS (Windows 8) |
| 2. | 8 GB RAM | microsoft word |
| 3. | Keyboard | TURBO C/C++ |
| 4. | Mouse |  |
| 5. | Monitor |  |
| 6. | Printer |  |

**THEORY:**-Polish notation, also known as Polish prefix notation or simply prefix notation, is a form of notation for logic, arithmetic, and algebra. Its distinguishing feature is that it places operators to the left of their operands. If the p[arity](http://en.wikipedia.org/wiki/Arity) of the operators is fixed, the result is a syntax lacking parentheses or other brackets that can still be parsed without ambiguity. The Polish logician Jan £ukasiewicz invented this notation in 1924 in order to simplify sentential logic.

The term Polish notation is sometimes taken (as the opposite of infix notation) to also include Polish postfix notation, or Reverse Polish notation, in which the operator is placed after the operands.[[1]](http://en.wikipedia.org/wiki/Prefix_notation#cite_note-1)

When Polish notation is used as a syntax for mathematical expressions by interpreters of programming languages, it is readily parsed into abstract syntax trees and can, in fact, define a one-to-one representation for the same. Because of this, Lisp (see below) and related programming languages define their entire syntax in terms of prefix notation (and others use postfix notation).

**PROCEDURE:**

#include<stdio.h>

#include<conio.h>

//Stack precedence function

int F(char symbol)

{

switch(symbol)

{

case '+' :

case '-' :

return 1;

case '\*':

case '^':

return 6;

case ')':

return 0;

case '#':

return -1;

default:

return 8;

}

}

//Input precedence function

int G(char symbol)

{

switch(symbol)

{

case '+' :

case '-' :

return 2;

case '\*':

return 4;

case '^':

return 5;

case '(':

return 0;

case ')':

return 9;

case '#':

return -1;

default:

return 7;

}

}

voidinfix\_prefix(char infix[], char prefix[])

{

int top, j, i;

char symbol, s[40];

top = -1;

s[++top] = '#';

j = 0;

strrev(infix);

for(i = 0;i <strlen(infix); i++)

{

symbol= infix[i];

while(F(s[top]) > G(symbol))

{

prefix[j] = s[top--];

j++;

}

if(F(s[top]) != G(symbol))

s[++top] = symbol;

else

top--;

}

while(s[top] != '#')

{

prefix[j++] = s[top--];

}

prefix[j] = '\0';

strrev(prefix);

}

void main()

{

char infix[20];

char prefix[20];

clrscr();

printf("/nEnter a valid infix expression\n");

scanf("%s",infix);

infix\_prefix(infix, prefix);

printf("\n\nThe prefix expression is\n");

printf("%s\n",prefix);

getch();

}

**OBSERVATIONS AND DISCUSSIONS:**

The output from the above code can be shown using below snapshot



**RESULT:**

The program is successfully written and created in c language

**EXPERIMENT – 4**

**DATE:**

**OBJECTIVE:WAP To count the number of tokens in an expression**

**EQUIPMENT/SOFTWARE USED:**

|  |  |  |
| --- | --- | --- |
| S.No. | Hardware | Software |
| 1. | I7 Processor | OS (Windows 8) |
| 2. | 8 GB RAM | Microsoft word |
| 3. | Keyboard | TURBO C/C++ |
| 4. | Mouse |  |
| 5. | Monitor |  |
| 6. | Printer |  |

**THEORY:-**A **token** is a string of one or more characters that is significant as a group. The process of forming tokens from an input stream of characters is called **tokenization**.

Tokens are identified based on the specific rules of the lexer. Some methods used to identify tokens include: [regular expressions](http://en.wikipedia.org/wiki/Regular_expression), specific sequences of characters known as a[flag](http://en.wikipedia.org/wiki/Flag_(computing)), specific separating characters called [delimiters](http://en.wikipedia.org/wiki/Delimiter), and explicit definition by a dictionary. Special characters, including punctuation characters, are commonly used by lexers to identify tokens because of their natural use in written and programming languages.

A lexical analyzer generally does nothing with combinations of tokens, a task left for a [parser](http://en.wikipedia.org/wiki/Parser). For example, a typical lexical analyzer recognizes parentheses as tokens, but does nothing to ensure that each "(" is matched with a ")".

**PROCEDURE:**

#include<stdio.h>

#include<ctype.h>

#include<conio.h>

#include<string.h>

int main()

{

charstr[50];

intlen;

inti,a=0,b=0,d=0,f=0,var=0,tokens=0,constant=0,oper=0;

clrscr();

printf("enter string :");

scanf("%s",str);

len=strlen(str);

for(i=0;i<len;i++)

{

if (isalpha(str[i]))

a++;

if (isdigit(str[i]))

{

while(isdigit(str[i]))

{

i++;

}

d++;

}

if(str[i]=='%'||str[i]=='\*'||str[i]=='/'||str[i]=='+'||str[i]=='-'||str[i]=='=')

f++;

else

b++;

}

var=a;

constant=d;

oper=f;

tokens=var+constant+oper;

printf("\ntotalvar:%d ",var);

printf("\ntotal constants:%d",constant);

printf("\ntotalopeators:%d",oper);

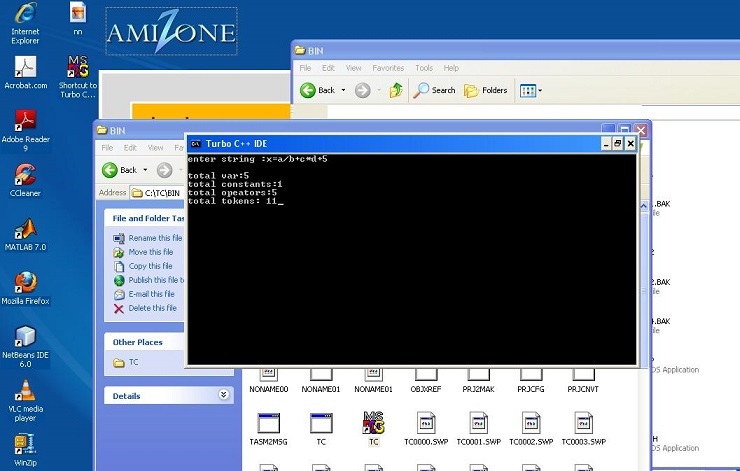
printf("\ntotal tokens: %d",tokens);

return 0;

getch();

}

**OBSERVATIONS AND DISCUSSIONS:**

\

**RESULT:**

The program is successfully written and created in c language

**EXPERIMENT-5**

**DATE:**

**OBJECTIVE:**WAP to convert Regular Expression to NFA.

**EQUIPMENT/SOFTWARE USED:**

|  |  |  |
| --- | --- | --- |
| S.No. | Hardware | Software |
| 1. | I7 Processor | OS (Windows 8) |
| 2. | 8 GB RAM | Microsoft word |
| 3. | Keyboard | TURBO C/C++ |
| 4. | Mouse |  |
| 5. | Monitor |  |
| 6. | Printer |  |

**THEORY:-** In [theoretical computer science](http://en.wikipedia.org/wiki/Theoretical_computer_science) and [formal language theory](http://en.wikipedia.org/wiki/Formal_language_theory), a **regular expression** (abbreviated **regex** or **regexp**) is a sequence of [characters](http://en.wikipedia.org/wiki/Character_(computing)) that forms a search pattern, mainly for use in [pattern matching](http://en.wikipedia.org/wiki/Pattern_matching) with [strings](http://en.wikipedia.org/wiki/String_(computer_science)), or [string matching](http://en.wikipedia.org/wiki/String_matching), i.e. "find and replace"-like operations. Each character in a regular expression is either understood to be a [metacharacter](http://en.wikipedia.org/wiki/Metacharacter) with its special meaning, or a regular character with its literal meaning. Together, they can be used to identify textual material of a given pattern, or process a number of instances of it that can vary from a precise equality to a very general similarity of the pattern. The pattern sequence itself is an expression that is a statement in a language designed specifically to represent prescribed targets in the most concise and flexible way to direct the automation of [text processing](http://en.wikipedia.org/wiki/Text_processing) of general text files, specific textual forms, or of random input strings.

In [automata theory](http://en.wikipedia.org/wiki/Automata_theory), a **nondeterministic finite automaton** (NFA), or nondeterministic finite state machine, is a [finite state machine](http://en.wikipedia.org/wiki/Finite_state_machine) that ***(1)*** does not require input symbols for state transitions and ***(2)*** is capable of transitioning to zero or two or more states for a given start state and input symbol.

**PROCEDURE:**

#include<stdio.h>

#include<string.h>

#include<conio.h>

int main()

{   clrscr();

    char reg[20];

    int q[20][3],i,j,len,a,b;

    for(a=0;a<20;a++)

    {

            for(b=0;b<3;b++)

            {

                q[a][b]=0;

            }

    }

    printf("Regular expression: \n");

    scanf("%s",reg);

    len=strlen(reg);

    i=0;

    j=1;

    while(i<len)

    {

            if(reg[i]=='a'&&reg[i+1]!='/'&&reg[i+1]!='\*')

            {

                q[j][0]=j+1;

                j++;

            }

            if(reg[i]=='b'&&reg[i+1]!='/'&&reg[i+1]!='\*')

            {

                q[j][1]=j+1;

                j++;

            }

            if(reg[i]=='e'&&reg[i+1]!='/'&&reg[i+1]!='\*')

            {

                q[j][2]=j+1;

                j++;

            }

            if(reg[i]=='a'&&reg[i+1]=='/'&&reg[i+2]=='b')

            {

                q[j][2]=((j+1)\*10)+(j+3);

                j++;

                q[j][0]=j+1;

                j++;

                q[j][2]=j+3;

                j++;

                q[j][1]=j+1;

                j++;

                q[j][2]=j+1;

                j++;

                i=i+2;

            }

            if(reg[i]=='b'&&reg[i+1]=='/'&&reg[i+2]=='a')

            {

                q[j][2]=((j+1)\*10)+(j+3);

                j++;

                q[j][1]=j+1;

                j++;

                q[j][2]=j+3;

                j++;

                q[j][0]=j+1;

                j++;

                q[j][2]=j+1;

                j++;

                i=i+2;

            }

            if(reg[i]=='a'&&reg[i+1]=='\*')

            {

                q[j][2]=((j+1)\*10)+(j+3);

                j++;

                q[j][0]=j+1;

                j++;

                q[j][2]=((j+1)\*10)+(j-1);

                j++;

            }

            if(reg[i]=='b'&&reg[i+1]=='\*')

            {

                q[j][2]=((j+1)\*10)+(j+3);

                j++;

                q[j][1]=j+1;

                j++;

                q[j][2]=((j+1)\*10)+(j-1);

                j++;

            }

            if(reg[i]==')'&&reg[i+1]=='\*')

            {

                q[0][2]=((j+1)\*10)+1;

                q[j][2]=((j+1)\*10)+1;

                j++;

            }

            i++;

    }

    printf("Transition function \n");

    for(i=0;i<=j;i++)

    {

            if(q[i][0]!=0)

                printf("\n q[%d,a]-->%d",i,q[i][0]);

            if(q[i][1]!=0)

                printf("\n q[%d,b]-->%d",i,q[i][1]);

            if(q[i][2]!=0)

            {

                if(q[i][2]<10)

                        printf("\n q[%d,e]-->%d",i,q[i][2]);

                else

                        printf("\n q[%d,e]-->%d & %d",i,q[i][2]/10,q[i][2]%10);

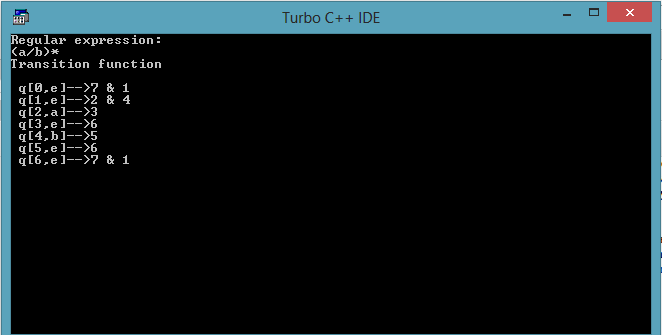
            }

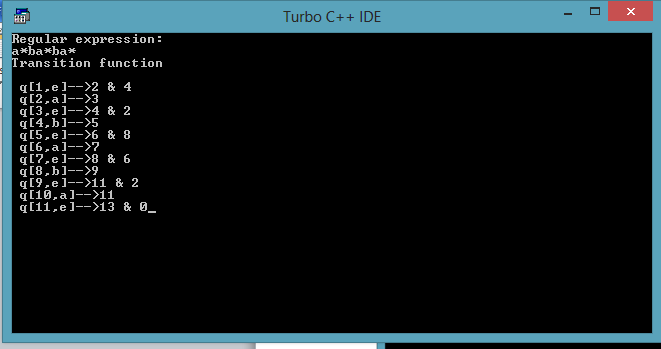
            getch();

    }

**OBSERVATIONS AND DISCUSSIONS:**

**The output from the above code can be shown using below snapshot.**

****



**RESULT:**

The program is successfully written and created in c language.

**EXPERIMENT-6**

**DATE:**

**OBJECTIVE:**WAP to convert NFA to DFA.

**EQUIPMENT/SOFTWARE USED:**

|  |  |  |
| --- | --- | --- |
| S.No. | Hardware | Software |
| 1. | I7 Processor | OS (Windows 8) |
| 2. | 8 GB RAM | Microsoft word |
| 3. | Keyboard | TURBO C/C++ |
| 4. | Mouse |  |
| 5. | Monitor |  |
| 6. | Printer |  |

**THEORY:-**In automata theory, a **nondeterministic finite automaton** (NFA), or nondeterministic finite state machine, is a finite state machine that ***(1)*** does not require input symbols for state transitions and ***(2)*** is capable of transitioning to zero or two or more states for a given start state and input symbol. This distinguishes it from a deterministic finite automaton (DFA), in which all transitions are uniquely determined and in which an input symbol is required for all state transitions. Although NFA and DFA have distinct definitions, all NFAs can be translated to equivalent DFAs using the subset construction algorithm,[1] i.e., constructed DFAs and their corresponding NFAs recognize the same formal language. Like DFAs, NFAs only recognize regular languages.

**Deterministic finite state machine**—is a finite state machine that accepts/rejects finite strings of symbols and only produces a unique computation (or run) of the automaton for each input string. 'Deterministic' refers to the uniqueness of the computation.

**PROCEDURE:**

#include<iostream.h>

#include<string.h>

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

char nfa[50][50],s[20],st[10][20],eclos[20],input[20];

int x,e,top=0,topd=0,n=0,ns,nos,in;

int checke(char a)

{

int i;

for(i=0;i<e;i++)

{

if(eclos[i]==a)

return i;

}

return -1;

}

int check(char a)

{

int i;

for(i=0;i<in;i++)

{

if(input[i]==a)

return i;

}

return -1;

}

void push(char a)

{

s[top]=a;

top++;

}

char pop()

{

top--;

return s[top];

}

void pushd(char \*a)

{

strcpy(st[topd],a);

topd++;

}

char \*popd()

{

topd--;

return st[topd];

}

int ctoi(char a)

{

int i=a-48;

return i;

}

char itoc(int a)

{

char i=a+48;

return i;

}

char \*eclosure(char \*a)

{

int i,j;

char c;

for(i=0;i<strlen(a);i++)

push(a[i]);

e=strlen(a);

strcpy(eclos,a);

while(top!=0)

{

c=pop();

for(j=0;j<ns;j++)

{

if(nfa[ctoi(c)][j]=='e')

{

if(check(itoc(j))==-1)

{

eclos[e]=itoc(j);

push(eclos[e]);

e++;

}

}

}

}

eclos[e]='\0';

return eclos;

}

void main()

{

int i,j,k,count;

char ec[20],a[20],b[20],c[20],dstates[10][10];

clrscr();

cout<<"Enter the number of states"<<endl;

cin>>ns;

for(i=0;i<ns;i++)

{

for(j=0;j<ns;j++)

{

cout<<"Move["<<i<<"]["<<j<<"]";

cin>>nfa[i][j];

if(nfa[i][j]!='-'&&nfa[i][j]!='e')

{

if((check(nfa[i][j]))==-1)

input[in++]=nfa[i][j];

}

}

}

topd=0;

nos=0;

c[0]=itoc(0);

c[1]='\0';

pushd(eclosure(c));

strcpy(dstates[nos],eclosure(c));

for(x=0;x<in;x++)

cout<<"\t"<<input[x];

cout<<"\n";

while(topd>0)

{

strcpy(a,popd());

cout<<a<<"\t";

for(i=0;i<in;i++)

{

int len=0;

for(j=0;j<strlen(a);j++)

{

int x=ctoi(a[j]);

for(k=0;k<ns;k++)

{

if(nfa[x][k]==input[i])

ec[len++]=itoc(k);

}

}

ec[len]='\0';

strcpy(b,eclosure(ec));

count=0;

for(j=0;j<=nos;j++)

{

if(strcmp(dstates[j],b)==0)

count++;

}

if(count==0)

{

if(b[0]!='\0')

{

nos++;

pushd(b);

strcpy(dstates[nos],b);

}

}

cout<<b<<"\t";

}

cout<<endl;

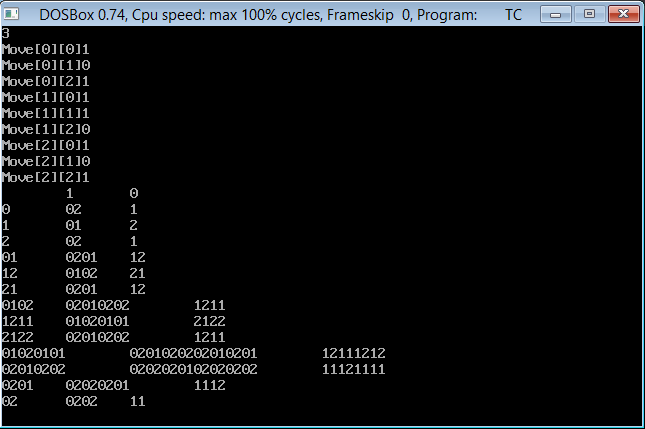
}

getch();

}

**OBSERVATIONS AND DISCUSSIONS:**

The output from the above code can be shown using below snapshot



**RESULT:**

The program is successfully written and created in c language.

**EXPERIMENT-7**

**DATE:**

**OBJECTIVE:**WAP to calculate Leading and Trailing.

**EQUIPMENT/SOFTWARE USED:**

|  |  |  |
| --- | --- | --- |
| S.No. | Hardware | Software |
| 1. | I7 Processor | OS (Windows 8) |
| 2. | 8 GB RAM | Microsoft word |
| 3. | Keyboard | TURBO C/C++ |
| 4. | Mouse |  |
| 5. | Monitor |  |
| 6. | Printer |  |

**THEORY :**

**ALGORITHM (LEADING)**

1. Start  
2. For each nonterminal A and terminal a **do** L(A,a):= **false;**

3. For each production of the form A->a or A->B **do** INSTALL(A,a);  
4. **While STACK** not empty **repeat** step 5& 6 5. Pop top pair (B,a) from STACK;  
6. For each production of the form A->B

**Do**INSTALL(A,a)

7. Stop

**Algorithm For INSTALL(A,a)**

1. Start  
2. If L(A,a) not present do step 3 and 4. 3 . Make L(A,a)=True  
4 . Push (A,a) onto stack  
5 . Stop

**ALGORITHM (TRAILING)**

1. Start  
   2. For each non terminal A and terminal a do L(A,a):=false;  
   3. For each production of the form A->a(alpha) or A-> Ba(alpha) do INSTALL(A,a) 4. While STACK not empty repeat 5 and 6  
   5. Pop top pair from stack  
   6. For each production of the form A->B(alpha) do INSTALL(A,a)  
   7. Stop

**Algorithm For INSTALL(A,a)**

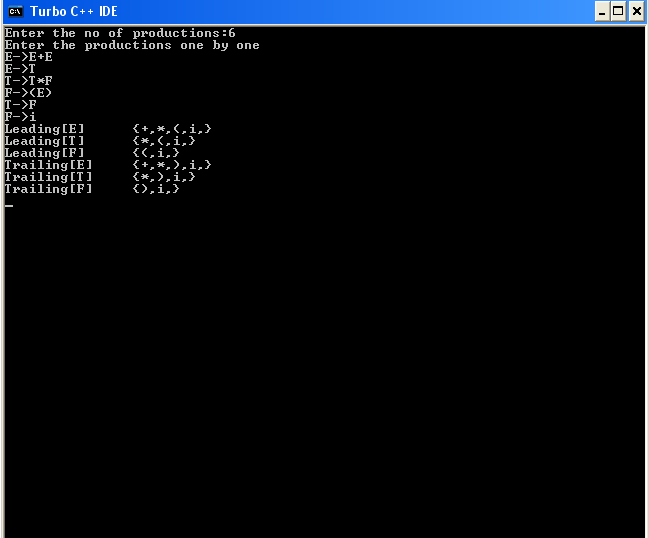
1. Start  
2. If L[A,a] not present repeat step 3 and 4 3. Make L(A,a)=True  
4. Push (A,a) onto stack  
5. Stop

**PROGRAM:**

#include<iostream.h>  
#include<string.h>  
#include<conio.h>  
intnt,t,top=0;  
char s[50],NT[10],T[10],st[50],l[10][10],tr[50][50];  
intsearchnt(char a)  
{  
int count=-1,i;  
for(i=0;i<nt;i++)  
{  
if(NT[i]==a)  
return i;  
}  
return count;  
}  
intsearchter(char a)  
{  
int count=-1,i;  
for(i=0;i<t;i++)  
{  
if(T[i]==a)  
return i;  
}  
return count;  
}  
void push(char a)  
{  
s[top]=a;  
top++;  
}  
char pop()  
{  
top--;  
return s[top];  
}  
void installl(inta,int b)   
  
{  
if(l[a][b]=='f')  
{  
l[a][b]='t';  
push(T[b]);  
push(NT[a]);  
}  
}  
void installt(inta,int b)  
{  
if(tr[a][b]=='f')  
{  
tr[a][b]='t';  
push(T[b]);  
push(NT[a]);  
}  
}  
void main()  
{  
inti,s,k,j,n;  
char pr[30][30],b,c;  
clrscr();  
cout<<"Enter the no of productions:";  
cin>>n;  
cout<<"Enter the productions one by one\n";  
for(i=0;i<n;i++)  
cin>>pr[i];  
nt=0;  
t=0;  
for(i=0;i<n;i++)  
{  
if((searchnt(pr[i][0]))==-1)  
NT[nt++]=pr[i][0];  
}  
for(i=0;i<n;i++)  
{  
for(j=3;j<strlen(pr[i]);j++)  
{  
if(searchnt(pr[i][j])==-1)  
{  
if(searchter(pr[i][j])==-1)  
T[t++]=pr[i][j];  
}  
}  
}  
for(i=0;i<nt;i++)  
{  
for(j=0;j<t;j++)  
l[i][j]='f';  
}  
for(i=0;i<nt;i++)  
{  
for(j=0;j<t;j++)  
  
tr[i][j]='f';  
}  
for(i=0;i<nt;i++)  
{  
for(j=0;j<n;j++)  
{  
if(NT[(searchnt(pr[j][0]))]==NT[i])  
{  
if(searchter(pr[j][3])!=-1)  
installl(searchnt(pr[j][0]),searchter(pr[j][3]));  
else  
{  
for(k=3;k<strlen(pr[j]);k++)  
{  
if(searchnt(pr[j][k])==-1)  
{  
installl(searchnt(pr[j][0]),searchter(pr[j][k]));  
break;  
}  
}  
}  
}  
}  
}  
while(top!=0)  
{  
b=pop();  
c=pop();  
for(s=0;s<n;s++)  
{  
if(pr[s][3]==b)  
installl(searchnt(pr[s][0]),searchter(c));  
}  
}  
for(i=0;i<nt;i++)  
{  
cout<<"Leading["<<NT[i]<<"]"<<"\t{";  
for(j=0;j<t;j++)  
{  
if(l[i][j]=='t')  
cout<<T[j]<<",";  
}  
cout<<"}\n";  
}  
  
top=0;  
for(i=0;i<nt;i++)  
{  
for(j=0;j<n;j++)  
{  
if(NT[searchnt(pr[j][0])]==NT[i])  
{  
if(searchter(pr[j][strlen(pr[j])-1])!=-1)  
installt(searchnt(pr[j][0]),searchter(pr[j][strlen(pr[j])-1]));  
else  
{  
for(k=(strlen(pr[j])-1);k>=3;k--)  
{  
if(searchnt(pr[j][k])==-1)  
{  
installt(searchnt(pr[j][0]),searchter(pr[j][k]));  
break;  
}  
}  
}  
}  
}  
}  
while(top!=0)  
{  
b=pop();  
c=pop();  
for(s=0;s<n;s++)  
{  
if(pr[s][3]==b)  
installt(searchnt(pr[s][0]),searchter(c));  
}  
}  
for(i=0;i<nt;i++)  
{  
cout<<"Trailing["<<NT[i]<<"]"<<"\t{";  
for(j=0;j<t;j++)  
{  
if(tr[i][j]=='t')  
cout<<T[j]<<",";  
}  
cout<<"}\n";  
}  
getch();

**OBSERVATIONS AND DISCUSSIONS:**

The output from the above code can be shown using below snapshot.



**RESULT:**

The program is successfully written and created in c language

**EXPERIMENT-8**

**DATE:**

**OBJECTIVE:**WAP to calculate First and Follow

**EQUIPMENT/SOFTWARE USED:**

|  |  |  |
| --- | --- | --- |
| S.No. | Hardware | Software |
| 1. | I7 Processor | OS (Windows 8) |
| 2. | 8 GB RAM | Microsoft word |
| 3. | Keyboard | TURBO C/C++ |
| 4. | Mouse |  |
| 5. | Monitor |  |
| 6. | Printer |  |

**THEORY:-**

**(A)FIRST**

**Definition**: For any string α of grammar symbols, we define **FIRST**(α) to be the set of terminals that occur as the first symbol in a string derived from α. So, if α⇒\*xQ for x a terminal and Q a string, then x is in FIRST(α). In addition if α⇒\*ε, then ε is in FIRST(α).

Unfortunately, the algorithms for computing FIRST and FOLLOW are not as simple to state as the definition suggests, in large part caused by ε-productions.

1. FIRST(a)={a} for all terminals a.
2. Initialize FIRST(A)=φ for all nonterminals A
3. If A → ε is a production, add ε to FIRST(A).
4. For each production A → Y1 ... Yn, add to FIRST(A) any terminal a satisfying
   1. a is in FIRST(Yi) and
   2. ε is in all previous FIRST(Yj).
5. Repeat this step until nothing is added.
6. FIRST of any string X=X1X2...Xn is initialized to φ and then
   1. add to FIRST(X) any non-ε symbol in FIRST(Xi) if ε is in all previous FIRST(Xj).
   2. add ε to FIRST(X) if ε is in every FIRST(Xj).

(**B)FOLLOW**

**Definition**: For any non terminal A, **FOLLOW**(A) is the set of terminals x, that can appear immediately to the right of A in a sentential form. Formally, it is the set of terminals x, such that S⇒\*αAxβ. In addition, if A can be the rightmost symbol in a sentential form, the endmarker $ is in FOLLOW(A).

Note that there might have been symbols between A and x during the derivation, providing they all derived ε and eventually x immediately follows A.

Initialize FOLLOW(S)=$ and FOLLOW(A)=φ for all other nonterminals A, and then apply the following three rules until nothing is add to any FOLLOW set.

i. For every production A → α B β, add all of FIRST(β) except ε to FOLLOW(B).

ii. For every production A → α B, add all of FOLLOW(A) to FOLLOW(B).

iii. For every production A → α B β where FIRST(β) contains ε, add all of FOLLOW(A) to FOLLOW(B).

**PROCEDURE:**

**(A)**

#include<stdio.h>  
#include<ctype.h>  
void FIRST(char );  
int count,n=0;  
char prodn[10][10], first[10];  
main()  
{int i,choice;  
char c,ch;  
printf("How many productions ? :");  
scanf("%d",&count);  
printf("Enter %d productions epsilon= $ :\n\n",count);  
for(i=0;i<count;i++)  
scanf("%s%c",prodn[i],&ch);  
do  
{n=0;  
printf("Element :");  
scanf("%c",&c);  
FIRST(c);  
printf("\n FIRST(%c)= { ",c);  
for(i=0;i<n;i++)  
printf("%c ",first[i]);  
printf("}\n");  
printf("press 1 to continue : ");  
scanf("%d%c",&choice,&ch);}  
while(choice==1);}  
void FIRST(char c)  
{int j;  
if(!(isupper(c)))first[n++]=c;  
for(j=0;j<count;j++)  
{if(prodn[j][0]==c)  
{if(prodn[j][2]=='$') first[n++]='$';  
else if(islower(prodn[j][2]))first[n++]=prodn[j][2];  
else FIRST(prodn[j][2]);}}}

**(B)**

#include<stdio.h>

#include<string.h>

int n,m=0,p,i=0,j=0;

char a[10][10],f[10];

void follow(char c);

void first(char c);

int main()

{int i,z;

char c,ch;

printf("Enter the no.of productions:");

scanf("%d",&n);

printf("Enter the productions(epsilon=$):\n");

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

scanf("%s%c",a[i],&ch);

do

{ m=0;

printf("Enter the element whose FOLLOW is to be found:");

scanf("%c",&c);

follow(c);

printf("FOLLOW(%c) = { ",c);

for(i=0;i<m;i++)

printf("%c ",f[i]);

printf(" }\n");

printf("Do you want to continue(0/1)?");

scanf("%d%c",&z,&ch);}

while(z==1);

return 0;}

void follow(char c)

{if(a[0][0]==c)f[m++]='$';

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

{for(j=2;j<strlen(a[i]);j++)

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

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

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

follow(a[i][0]);} }}}

void first(char c)

{int k;

if(!(isupper(c)))f[m++]=c;

for(k=0;k<n;k++)

{if(a[k][0]==c)

{if(a[k][2]=='$') follow(a[i][0]);

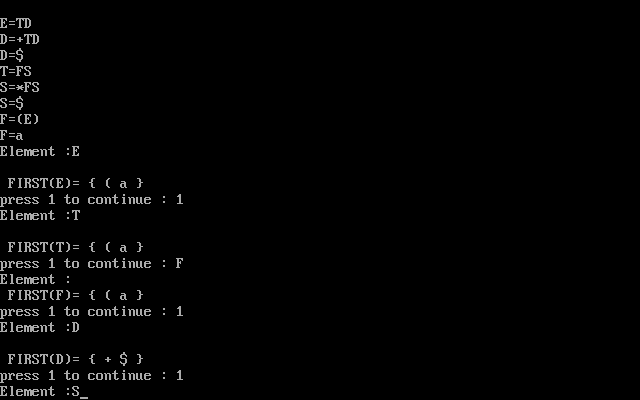
else if(islower(a[k][2]))f[m++]=a[k][2];

else first(a[k][2]);}}}

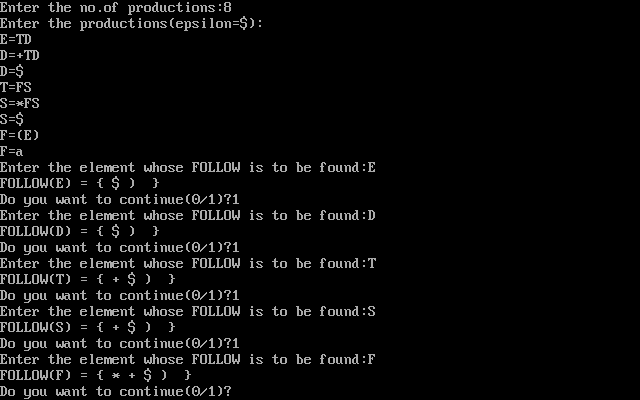
**OBSERVATIONS AND DISCUSSIONS:**

The output from the above code can be shown using below snapshot.

**(A)**



**(B)**

****

**RESULT:**

The program is successfully written and created in c language

**EXPERIMENT-9**

**DATE**:

**OBJECTIVE** : WAP to design a shift reduce parser for E -> E + E | E \* E | (E) | id

**EQUIPMENT/SOFTWARE USED :**

|  |  |  |
| --- | --- | --- |
| S.No. | Hardware | Software |
| 1. | I7 Processor | OS (Windows 8) |
| 2. | 8 GB RAM | Microsoft word |
| 3. | Keyboard | TURBO C/C++ |
| 4. | Mouse |  |
| 5. | Monitor |  |
| 6. | Printer |  |

**PROCEDURE :**

Shift-Reduce parsing is a category of efficient, table-driven [bottom-up parsing](http://en.wikipedia.org/wiki/Bottom-up_parsing) methods for computer languages and other notations formally defined by a [grammar](http://en.wikipedia.org/wiki/Grammar). The parsing methods most commonly used today, [LR parsing](http://en.wikipedia.org/wiki/LR_Parser) and its variations, are shift-reduce methods.[[1]](http://en.wikipedia.org/wiki/Shift-reduce_parser#cite_note-1) The [precedence parsers](http://en.wikipedia.org/wiki/Simple_precedence_parser) used before the invention of LR parsing are also shift-reduce methods. All shift-reduce parsers have similar outward effects, in the incremental order in which they build a parse tree or call specific output actions. The outward actions of an LR parser are best understood by ignoring the arcane mathematical details of how LR parser tables are generated, and instead looking at the parser as just some generic shift-reduce method.

A shift-reduce [parser](http://en.wikipedia.org/wiki/Parser) scans and parses the input text in one forward pass over the text, without backing up. (That forward direction is generally left-to-right within a line, and top-to-bottom for multi-line inputs.) The parser builds up the [parse tree](http://en.wikipedia.org/wiki/Parse_tree) incrementally, bottom up, and left to right, without guessing or backtracking. At every point in this pass, the parser has accumulated a list of subtrees or phrases of the input text that have been already parsed. Those subtrees are not yet joined together because the parser has not yet reached the right end of the syntax pattern that will combine them.

A shift-reduce parser works by doing some combination of Shift steps and Reduce steps.

* A Shift step advances in the input stream by one symbol. That shifted symbol becomes a new single-node parse tree.
* A Reduce step applies a completed grammar rule to some of the recent parse trees, joining them together as one tree with a new root symbol.

The parser continues with these steps until all of the input has been consumed and all of the parse trees have been reduced to a single tree representing an entire legal input.

**CODE :**

#include<stdio.h>

#include<conio.h>

#include<string.h>

char exp[30],stack[30],arr[30],temp[30];

int i,k=0,j,l,r,s;

void push(char exp[])

{

arr[i]=exp[k];

i++;

}

void dispinp()

{

printf("\t\t\t");

for(k=0;k<strlen(exp);k++)

printf("%c",exp[k]);

printf("$");

}

void dispstk()

{

printf("\n");

for(k=0;k<strlen(stack);k++)

printf("%c",stack[k]);

}

void assign()

{

stack[++j]=arr[i];

exp[i]=' '; 21

dispstk();

dispinp();

}

int main()

{

printf("\t\t\tSHIFT REDUCE PARSER\n");

printf("\nThe Production is: E->E+E/E\*E/d/a\n");

printf("\nEnter the string to be parsed:\n");

gets(exp);

printf("\nSTACK\t\t\tINPUT\t\t\tACTION\n");

printf("\n$");

dispinp();

printf("\t\t\tShift");

for(k=0;k<strlen(exp);k++)

push(exp);

l=strlen(exp);

stack[0]='$';

for(i=0;i<l;i++)

{

switch(arr[i])

{

case 'd':

assign();

printf("\t\t\tReduce by E->d");

stack[j]='E';

dispstk();

dispinp();

if(arr[i+1]!='\0') 22

printf("\t\t\tShift");

break;

case '+':

assign();

printf("\t\t\tShift");

break;

case '\*':

assign();

printf("\t\t\tShift");

break;

case '-':

assign();

printf("\t\t\tShift");

break;

default:

printf("\nError:String not accepted");

goto label;

}}

l=strlen(stack);

while(l>2)

{

r=0;

for(i=l-1;i>=l-3;i--)

{

temp[r]=stack[i];

r++;

}

temp[r]=NULL;

if((strcmp(temp,"E+E")==0)||(strcmp(temp,"E\*E")==0)||(strcmp(temp,"E\*E")==0))

{

for(i=l;i>l-3;i--)

stack[i]=' ';

stack[l-3]='E';

printf("\t\t\tReduce by E->");

for(i=0;i<strlen(temp);i++)

printf("%c",temp[i]);

dispstk();

dispinp();

l=l-2;

}

else

{

printf("\nError:String not accepted");

goto label;

}}

printf("\t\t\tAccept");

printf("\n\nString accepted");

label:

return 0;

getch();

}

**OBSERVATIONS AND DISCUSSIONS :**  The program is designed for shift reduce parser for the given grammar.



**RESULT :** A program that for calculates the Shift Reduce Parser of a grammar has been made.

**EXPERIMENT-10**

**DATE:**

**OBJECTIVE:** WAP to design a stop down parser for E -> E + E | E \* E | (E) | id

**EQUIPMENT/SOFTWARE US:ED**

|  |  |  |
| --- | --- | --- |
| S.No. | Hardware | Software |
| 1. | I7 Processor | OS (Windows 8) |
| 2. | 8 GB RAM | Microsoft word |
| 3. | Keyboard | TURBO C/C++ |
| 4. | Mouse |  |
| 5. | Monitor |  |
| 6. | Printer |  |

**PROCEDURE:**

The top down construction of a parse tree is done by starting with the root ,labeled with the starting non-terminal ,and repeatedly performing the following two steps-

1. at node n, labeled with non-terminal A,select one of the productions for A and construct

children at n for the symbols on the right side of the production

2. Find the next node at which the subtree is constructed.

For some grammars, the above steps can be implemented during a single left to right scan of the input string. The current token being scanned on the input is often called as the lookahead symbol . Initially the lookahead symbol is the first i.e the leftmost token of the input string.

Here we have assumed that , at the first attempt the parser would know

which production to use to get the right output, but in general, the selection of a production of a non-terminal may involve trial and error, that is we may have to try a production and backtrack to try another production if the first is found to be unsuitable. A production is unsuitable ,if after using the production, we cannot complete the tree to match the input string .we will discuss this parsing in the next section.

**DATA :**

#include<iostream.h>

#include<conio.h>

#include<string.h>

class parse

{

int nt,t,m[20][20],i,s,n,p1,q,k,j;

char p[30][30],n1[20],t1[20],ch,b,c,f[30][30],fl[30][30];

public:

int scant(char);

int scannt(char);

void process();

void input();

};

int parse::scannt(char a)

{

int c=-1,i;

for(i=0;i<nt;i++)

{

if(n1[i]==a)

{

return i;

}

}

return c;

}

int parse::scant(char b)

{

int c1=-1,j;

for(j=0;j<t;j++)

{

if(t1[j]==b)

{

return j;

}

}

return c1;

}

void parse::input()

{

cout<<"Enter the number of productions:";

cin>>n;

cout<<"Enter the productions one by one"<<endl;

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

cin>>p[i];

nt=0;

t=0;

}

void parse::process()

{

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

{

if(scannt(p[i][0])==-1)

n1[nt++]=p[i][0];

}

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

{

for(j=3;j<strlen(p[i]);j++)

{

if(p[i][j]!='e')

{

if(scannt(p[i][j])==-1)

{

if((scant(p[i][j]))==-1)

t1[t++]=p[i][j];

}

}

}

}

t1[t++]='$';

for(i=0;i<nt;i++)

{

for(j=0;j<t;j++)

m[i][j]=-1;

}

for(i=0;i<nt;i++)

{

cout<<"Enter first["<<n1[i]<<"]:";

cin>>f[i];

}

for(i=0;i<nt;i++)

{

cout<<"Enter follow["<<n1[i]<<"]:";

cin>>fl[i];

}

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

{

p1=scannt(p[i][0]);

if((q=scant(p[i][3]))!=-1)

m[p1][q]=i;

if((q=scannt(p[i][3]))!=-1)

{

for(j=0;j<strlen(f[q]);j++)

m[p1][scant(f[q][j])]=i;

}

if(p[i][3]=='e')

{

for(j=0;j<strlen(fl[p1]);j++)

m[p1][scant(fl[p1][j])]=i;

}

}

for(i=0;i<t;i++)

cout<<"\t"<<t1[i];

cout<<endl;

for(j=0;j<nt;j++)

{

cout<<n1[j];

for(i=0;i<t;i++)

{

cout<<"\t"<<" ";

if(m[j][i]!=-1)

cout<<p[m[j][i]];

}

cout<<endl;

}

}

void main()

{

clrscr();

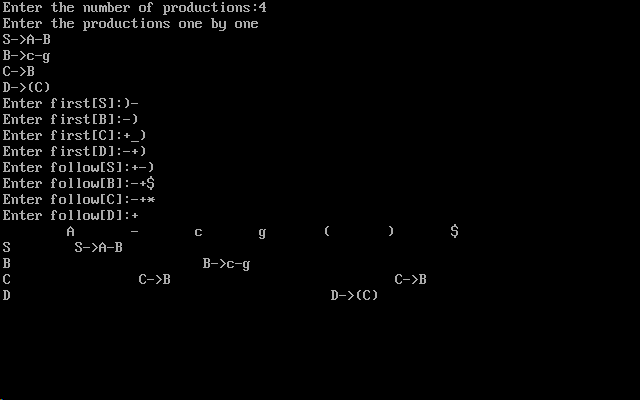
parse p;

p.input();

p.process();

getch();

}

**OBSERVATIONS AND DISCUSSIONS:**

**RESULT :**The program successfully runs through many test cases and every time, and gives the correct result for top down parser.