ASSIGNMENT-7

```
1       /*Remove left recursion and left factoring of a given grammar.*/
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<iostream>
using namespace std;
struct production
{
        char lf;
        char rt[10];
        int prod_rear;
        int fl;
};
struct production rule[20],rule_new[20];    //Creation of object
int b=-1,d,f,q,n,m=0,c=0,count_1=0;
char terminal[20],nonterm[20],alpha[10],extra[10],temp2[20];
char epsilon='^';
int main()  {
   char input[10][100],*l[10],*r[10],*temp[10],tempprod[10][20],productions[10][25][50];
   int k,n,i[10];
   int j=0,flag[10];
   char *spcl = "X";
   l[0] = spcl;
   printf("\nEnter no. of productions : ");
   scanf("%d",&n);
   printf("Enter the productions : \n");
   for(k=0;k<n;k++)
   {
            scanf("%s",input[k]);
            l[k] = strtok(input[k],"->");
            r[k] = strtok(NULL,"->");
            temp[k] = strtok(r[k],"|");
            while(temp[k])  {
               if(temp[k][0] == l[k][0])  {
                  flag[k] = 1;
                  sprintf(productions[k][i[k]++],"%s->%s%s\0",spcl,temp[k]+1,spcl);
               }
               else
                  sprintf(productions[k][i[k]++],"%s->%s%s\0",l[k],temp[k],spcl);
               temp[k] = strtok(NULL,"|");
            }

            sprintf(productions[k][i[k]++],"%s->^",spcl);
   }

   for(k=0;k<n;k++)
   {
            if(flag[k] == 0)
         {
```

```
                printf("\nThe given productions don't have Left Recursion");
                count_1 = i[k];
        }
           else
           {
                        printf("\nAfter removing left recursion, productions are : \n");
                    for(j=0;j<i[k];j++)  {
                        printf("\n%s",productions[k][j]);
                        count_1 = i[k];
                    }
                }
                printf("\n");
}
printf("\n");
        //Remove left factoring
        q = 1;
        alpha[0] = 'Y';
        int cnt,g,cnt3;
        n = count_1;
        for(cnt=0;cnt<n;cnt++)
        {
                //cout<<"\nProduction no. "<<cnt+1<<"  "<<productions[0][cnt];
                strcpy(temp2,productions[0][cnt]);
                rule[cnt].lf = temp2[0];
            int index = 0;
                 for(int posn = 3;posn<strlen(temp2);posn++)
                 {
                        rule[cnt].rt[index++] = temp2[posn];
                 }

            rule[cnt].prod_rear=strlen(rule[cnt].rt);
            rule[cnt].fl=0;
        }
for(int cnt1=0;cnt1<n;cnt1++)
{
 for(int cnt2=cnt1+1;cnt2<n;cnt2++)
 {
       if(rule[cnt1].lf==rule[cnt2].lf)
       {
         cnt=0;
         int p=-1;
         while((rule[cnt1].rt[cnt]!='\0')&&(rule[cnt2].rt[cnt]!='\0'))
         {
           if(rule[cnt1].rt[cnt]==rule[cnt2].rt[cnt])
           {
             extra[++p]=rule[cnt1].rt[cnt];
             rule[cnt1].fl=1;
             rule[cnt2].fl=1;
           }
           else
           {
```

```
            if(p==-1)
                  break;
            else
            {
                  int h=0,u=0;
                  rule_new[++b].lf=rule[cnt1].lf;
                  strcpy(rule_new[b].rt,extra);
                  rule_new[b].rt[p+1]=alpha[c];
                  rule_new[++b].lf=alpha[c];
                  for(g=cnt;g<rule[cnt2].prod_rear;g++)
                   rule_new[b].rt[h++]=rule[cnt2].rt[g];
                   rule_new[++b].lf=alpha[c];
                  for(g=cnt;g<=rule[cnt1].prod_rear;g++)
                   rule_new[b].rt[u++]=rule[cnt1].rt[g];
                   m=1;
                   break;
             }
            }
            cnt++;
          }
          if((rule[cnt1].rt[cnt]==0)&&(m==0))
          {
                int h=0;
                rule_new[++b].lf=rule[cnt1].lf;
                strcpy(rule_new[b].rt,extra);
                rule_new[b].rt[p+1]=alpha[c];
                rule_new[++b].lf=alpha[c];
                rule_new[b].rt[0]=epsilon;
                rule_new[++b].lf=alpha[c];
                for(int g=cnt;g<rule[cnt2].prod_rear;g++)
                rule_new[b].rt[h++]=rule[cnt2].rt[g];
          }
          if((rule[cnt2].rt[cnt]==0)&&(m==0))
          {
            int h=0;
            rule_new[++b].lf=rule[cnt1].lf;
            strcpy(rule_new[b].rt,extra);
            rule_new[b].rt[p+1]=alpha[c];
            rule_new[++b].lf=alpha[c];
            rule_new[b].rt[0]=epsilon;
            rule_new[++b].lf=alpha[c];
            for(int g=cnt;g<rule[cnt1].prod_rear;g++)
             rule_new[b].rt[h++]=rule[cnt1].rt[g];
          }
          c++;
          m=0;
        }
    }
}
cout<<"\n\nProduction rules after removing left factoring : \n";
```

```
  for(cnt3=0;cnt3<=b;cnt3++)
       {
              //cout<<"Production "<<cnt3+1<<" is: ";
              cout<<rule_new[cnt3].lf;
              cout<<"->";
              cout<<rule_new[cnt3].rt;
              cout<<endl;
       }
  for(int cnt4=0;cnt4<n;cnt4++)
  {
  if(rule[cnt4].fl==0)
  {
  //cout<<"Production "<<++cnt3<<" is: ";
  cout<<rule[cnt4].lf<<"->"<<rule[cnt4].rt<<endl;
  }
 }
```

```
mona@mona-VirtualBox:~/CD Lab$ g++ rec_fact.cpp -w
mona@mona-VirtualBox:~/CD Lab$ ./a.out

Enter no. of productions : 1
Enter the productions :
A->Aab|bc|bac

After removing left recursion, productions are :

X->abX
A->bcX
A->bacX
X->^



Production rules after removing left factoring :
A->b
->acX
->cX
X->abX
X->^
mona@mona-VirtualBox:~/CD Lab$
```

}

2 /*Find first and follow sets of a given grammar*/
/*To check whether the grammar is LL1 or not*/

```
#include<stdio.h>
 int n,m=0,p,i=0,j=0,npro,b;
 char a[10][10],f[10];
  int limit;
  void follow(char c);
void first(char c);

void Find_First(char* array, char ch);
void Array_Manipulation(char array[], char value);

void Find_First(char* array, char ch)
{
    int count1, j, k;
    char temporary_result[20];
    int x;
    temporary_result[0] = '\0';
    array[0] = '\0';
    if(!(isupper(ch)))
    {

        Array_Manipulation(array, ch);
        return ;
    }
    for(count1 = 0; count1 < limit; count1++)
    {
        if(a[count1][0] == ch)
        {
            if(a[count1][3] == '^')
            {

                Array_Manipulation(array, '^');
            }
            else
            {

                j = 3;
                while(a[count1][j] != '\0')
                {
                    x = 0;
                    Find_First(temporary_result, a[count1][j]);
                    for(k = 0; temporary_result[k] != '\0'; k++)
                    {
                        Array_Manipulation(array,temporary_result[k]);
                    }
                    for(k = 0; temporary_result[k] != '\0'; k++)
```

```
                    {
                        if(temporary_result[k] == '^')
                        {
                            x = 1;
                            break;
                        }
                    }
                    if(!x)
                    {
                        break;
                    }
                    j++;
                }
            }
        }
    }
    return;
}

void Array_Manipulation(char array[], char value)
{
    int temp;
    for(temp = 0; array[temp] != '\0'; temp++)
    {
        if(array[temp] == value)
        {
            return;
        }
    }
    array[temp] = value;
    array[temp + 1] = '\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][3]=='^')
    follow_fun(a[k][0]);
   else if(islower(a[k][3]))
    f[m++]=a[k][3];
   else first(a[k][3]);
 }
      }
     }
   void follow_fun(char c)
  {
```

```
  if(a[0][0]==c)
   f[m++]='$';
  for(b=0;b<npro;b++)
   {
    for(j=3;j<strlen(a[b]);j++)
     {
     //printf("\nINSIDE IF for %c",c);
      if(a[b][j]==c)
                        {
                          if(a[b][j+1]!='\0')
                                   first(a[b][j+1]);
                                  if(a[b][j+1]=='\0' && c!=a[b][0])
                                   follow_fun(a[b][0]);
                         }
     }
    }
  }


void main()
{
        char pro[10][10],first[10][10],follow[10][10],nt[10],ter[10],res[10][10][10],temp[10];
        int noter=0,nont=0,k,flag=0,count[10][10],row,col,l,index;
          char c,ch;
           char array[25];
        //clrscr();
        for(i=0;i<10;i++)
        {
                for(j=0;j<10;j++)
                {
                        count[i][j]=NULL;
                        for(k=0;k<10;k++)
                        {
                                res[i][j][k]=NULL;
                        }
                }
        }
        printf("Enter the no of productions:");
        scanf("%d",&npro);
        for(i=0;i<npro;i++)
        {
                //scanf("%s",pro[i]);
           scanf("%s%c",a[i],&ch);
                strcpy(pro[i],a[i]);
        }
        limit = npro;
        n = npro;
        for(i=0;i<npro;i++)
        {
                flag=0;
                for(j=0;j<nont;j++)
```

```
		{
			if(nt[j]==pro[i][0])

			flag=1;

		}
		if(flag==0)
		{
			nt[nont]=pro[i][0];
			nont++;
		}
}
for(i=0;i<nont;i++)
{
		m=0;
		Find_First(array, nt[i] );
		strcpy(first[i],array);

		m=0;
		 follow_fun(nt[i]);
		 strcpy(follow[i],f);
}
for(k=0;k<nont;k++)
{
		printf("\nFirst Value of %c:\t{ ", nt[k]);
		    for(i = 0; first[k][i] != '\0'; i++)
		    {
		        printf(" %c ", first[k][i]);
		    }
		    printf("}\n");

		 printf("Follow of %c:\t{ ",nt[k]);
		  for(i=0;i<m;i++)
			printf(" %c ",follow[k][i]);

}
for(i=0;i<nont;i++)
{
flag=0;
for(j=0;j<strlen(first[i]);j++)
{
for(k=0;k<noter;k++)
{
if(ter[k]==first[i][j])
{
flag=1;
}
}
if(flag==0)
{
if(first[i][j]!='^')
```

```
{
ter[noter]=first[i][j];
noter++;
}
}
}
}
for(i=0;i<nont;i++)
{
flag=0;
for(j=0;j<strlen(follow[i]);j++)
{
for(k=0;k<noter;k++)
{
if(ter[k]==follow[i][j])
{
flag=1;
}
}
if(flag==0)
{
ter[noter]=follow[i][j];
noter++;
}
}
}
for(i=0;i<nont;i++)
{
for(j=0;j<strlen(first[i]);j++)
{
flag=0;
if(first[i][j]=='^')
{
col=i;
for(m=0;m<strlen(follow[col]);m++)
{
for(l=0;l<noter;l++)
        {
        if(ter[l]==follow[col][m])
        {
        row=l;
        }
        }
        temp[0]=nt[col];
        temp[1]='-' ;
        temp[2]='>';
        temp[3]='^';
        temp[4]='\0';
        //printf("\ntemp %s",temp);
        strcpy(res[col][row],temp);
        count[col][row]+=1;
```

```
            for(k=0;k<10;k++){
            temp[k]=NULL;       }
            }

}
            else{
            for(l=0;l<noter;l++)
            {
            if(ter[l]==first[i][j])
            {
            row=l;
            }
            }
            for(k=0;k<npro;k++){
            if(nt[i]==pro[k][0])
            {
            col=i;
            if((pro[k][3]==first[i][j])&&(pro[k][0]==nt[col]))
            {
            strcpy(res[col][row],pro[k]);
            count[col][row]+=1;
            }
            else
            {
            if((isupper(pro[k][3]))&&(pro[k][0]==nt[col]))
            {
            flag=0;
            for(m=0;m<nont;m++)
            {
            if(nt[m]==pro[k][3]){index=m;flag=1;}
            }
            if(flag==1){
            for(m=0;m<strlen(first[index]);m++)
            {if(first[i][j]==first[index][m])
            {strcpy(res[col][row],pro[k]);
               count[col][row]+=1;}
            }
            }
            }}}}}
}}
printf("\n\nLL1 Table\n\n");
printf("----------------------\n\n");
flag=0;
for(i=0;i<noter;i++)
{
printf("\t%c",ter[i]);
}
for(j=0;j<nont;j++)
{
printf("\n\n%c",nt[j]);
for(k=0;k<noter;k++)
```

```
        {
        printf("\t%s",res[j][k]);
        if(count[j][k]>1){flag=1;}
        }
        }
        if(flag==1){printf("\nThe given grammar is not LL1\n");}
        else{printf("\nThe given grammar is LL1\n");}
}
```

```
mona@mona-VirtualBox:~/CD Lab$ gcc "LL(1)".c -w
mona@mona-VirtualBox:~/CD Lab$ ./a.out
Enter the no of productions:8
E->TZ
Z->+TZ
Z->^
T->FY
Y->*FY
Y->^
F->a
F->(E)

First Value of E:       {  a  ( }
Follow of E:    {  $  )
First Value of Z:       {  +  ^ }
Follow of Z:    {  $  )
First Value of T:       {  a  ( }
Follow of T:    {  +  $  )
First Value of Y:       {  *  ^ }
Follow of Y:    {  +  $  )
First Value of F:       {  a  ( }
Follow of F:    {  *  +  $  )

LL1 Table

---------------------

        a       (       +       *       $       )

E       E->TZ   E->TZ

Z                       Z->+TZ          Z->^    Z->^

T       T->FY   T->FY

Y                       Y->^    Y->*FY  Y->^    Y->^

F       F->a    F->(E)
The given grammar is LL1
```

ASSIGNMENT-8

```
/*Design a LR(0) Parser.*/

//Closure_goto
char items[30][100][100];
char augmented_grammar[100][100], terminals[10], nonterminals[10];
int no_of_productions = 0, no_of_states = 0, no_of_items[30], no_of_terminals = 0,
no_of_nonterminals = 0;

char FIRST[2][10][10];
char FOLLOW[10][10];


//Variables used only in this module.

int state_index = 0, goto_state_index = 0, closure_item_index = 0;

int check(char c) {
	int i;

	for(i = 0; i < no_of_terminals; i++)
		if(terminals[i] == c)
			return 1;

	return 0;
}

void generate_terminals() {
	int i, j;
	int index = 0;

	for(i = 0; i < no_of_productions; i++) {
		for(j = 0; augmented_grammar[i][j] != '>'; j++);
		j++;

		for(; augmented_grammar[i][j] != '\0'; j++) {
			if(augmented_grammar[i][j] < 65 || augmented_grammar[i][j] > 90) {
				if(!check(augmented_grammar[i][j])) {
					terminals[index] = augmented_grammar[i][j];
					no_of_terminals++;
					index++;
				}
			}
		}
	}

	terminals[index] = '$';
	no_of_terminals++;
	index++;
	terminals[index] = '\0';
```

```
}

int check2(char c, int index) {
	int i;

	for(i = 0; i < index; i++)
		if(nonterminals[i] == c)
			return 1;

	return 0;
}

void generate_nonterminals() {
	int i, index = 0;

	for(i = 0; i < no_of_productions; i++)
		if(!check2(augmented_grammar[i][0], index)) {
			nonterminals[index] = augmented_grammar[i][0];
			index++;
		}

	no_of_nonterminals = index;
	nonterminals[index] = '\0';
}

void initialize_items() {
	generate_terminals();
	generate_nonterminals();

	int i;

	for(i = 0; i < 30; i++)
		no_of_items[i] = 0;
}

void generate_item(char *s, char *t)
{
	int i;

	for(i = 0; i < 3; i++)
		t[i] = s[i];

	t[i] = '.';

	if(s[i] != '@')
		for(; i < strlen(s); i++)
			t[i+1] = s[i];

	t[i+1] = '\0';
}
```

```
int item_found(char *s) {		//Check for items in a state.
	int i;

	for(i = 0; i < closure_item_index; i++) {
		if(!strcmp(s, items[state_index][i]))		//If the strings match.
			return 1;
	}

	return 0;
}

int isterminal(char s) {
	int i;

	for(i = 0; i < no_of_terminals; i++)
		if(s == terminals[i])
			return 1;

	return 0;
}

void closure(char *s) {
	int i, j;

	for(i = 0; s[i] != '.'; i++);

	i++;

	if(!item_found(s)) {
		strcpy(items[state_index][closure_item_index], s);
		closure_item_index++;

//		printf("%s\n", items[state_index][closure_item_index-1]);
	}

	if(s[i] == s[0] && s[i-2] == '>')	//To avoid infinite loop due to left recursion.
		return;

	if(isterminal(s[i]))
		return;

	else	{		//Not a terminal
		for(j = 0; j < no_of_productions; j++) {
			char temp[100];

			if(augmented_grammar[j][0] == s[i]) {
				generate_item(augmented_grammar[j], temp);
				closure(temp);
			}
		}
	}
```

```
}

int Goto1(char s, char temp[][100]) {	//Find Goto on symbol s. GOTO(goto_state_index, s)
	int i, j;
	int n = 0;
	char t, temp2[100];

	if(s == '\0') {
		return n;
	}

	for(i = 0; i < no_of_items[goto_state_index]; i++) {
		strcpy(temp2, items[goto_state_index][i]);

		for(j = 0; temp2[j] != '.'; j++);

		if(temp2[j+1] == '\0')
			continue;

		if(temp2[j+1] == s) {
			t = temp2[j];
			temp2[j] = temp2[j+1];
			temp2[j+1] = t;

			strcpy(temp[n], temp2);
			n++;
		}
	}

	return n;
}

int state_found(char *s) {		//Checks for existance of same state.
	int i;

	for(i = 0; i < state_index; i++) {
		if(!strcmp(s, items[i][0]))		//Compare with the first item of each state.
			return 1;
	}

	return 0;
}

int transition_item_found(char * t_items, char s, int t_index) {
	int i;

	for(i = 0; i < t_index; i++)
		if(s == t_items[i])
			return 1;

	return 0;
```

```
}

void compute_closure_goto() {
	char temp[100][100], transition_items[100];
	int i, no_of_goto_items,j, transition_index = 0;

	generate_item(augmented_grammar[0], temp[0]);

	closure(temp[0]);

	no_of_items[state_index] = closure_item_index;
	closure_item_index = 0;

	state_index++;
	//state_index is 1 now.

	while(goto_state_index < 30) {
		transition_index = 0;
		transition_items[transition_index] = '\0';

		for(i = 0; i < no_of_items[goto_state_index]; i++) {
			for(j = 0; items[goto_state_index][i][j] != '.'; j++);
			j++;

			if(!transition_item_found(transition_items, items[goto_state_index][i][j],
transition_index)) {
				transition_items[transition_index] = items[goto_state_index][i][j];
				transition_index++;
			}
		}

		transition_items[transition_index] = '\0';

		for(i = 0; i < transition_index; i++) {
			int add_flag = 0;

			no_of_goto_items = Goto1(transition_items[i], temp);

			for(j = 0; j < no_of_goto_items; j++) {
				if(!state_found(temp[j])) {
					add_flag = 1;
					closure(temp[j]);
				}
				else
					break;
			}
			if(add_flag) {
				no_of_items[state_index] = closure_item_index;
				closure_item_index = 0;
				state_index++;
			}
```

```
                }

                goto_state_index++;
        }

        no_of_states = state_index;
}

void print() {
        int i, j;

        printf("\nNumber of states = %d.\n", no_of_states);

        for(i = 0; i < no_of_states; i++) {
                printf("\nItems in State %d...\n", i);

                for(j = 0; j < no_of_items[i]; j++)
                        printf("%s\n", items[i][j]);
        }
}



void start() {
        char str[100];

        printf("Enter number of productions:");
        scanf("%d", &no_of_productions);

        printf("Enter the individual production rules separately : \n");

        int i;
        for(i = 1; i <= no_of_productions; i++)
                scanf("%s", augmented_grammar[i]);

        printf("\nAugmented Grammar is...\n\n");

        strcpy(augmented_grammar[0], "Z->");
        str[0] = augmented_grammar[1][0];
        str[1] = '\0';
        strcat(augmented_grammar[0], str);

        no_of_productions++;

        for(i = 0; i < no_of_productions; i++)
                printf("%s\n", augmented_grammar[i]);

        initialize_items();

        compute_closure_goto();
```

```
	print();
}

struct Stack {	//Holds states.
	int states[100];
	int top;
} stack;

void push(int a) {
	stack.top++;
	stack.states[stack.top] = a;
}

void pop() {
	int a = stack.states[stack.top];
	stack.top--;
}

int get_top() {		//Returns top of stack state.
	return stack.states[stack.top];
}

void initialize_stack() {	//Initialize stack to have state 0 on top.
	stack.top = -1;

	push(0);
}

int get_int(char *s) {		//Get integer part of the strings found in table entries.
	int i, j;
	char temp[10];

	for(i = 0; s[i] != ':'; i++);
	i++;

	for(j = i; s[i] != '\0'; i++)
		temp[i-j] = s[i];

	temp[i-j] = '\0';

	return atoi(temp);
}

int get_length(char *production) {	//Returns length of string in the production body.
	int i, j;

	for(i = 0; production[i] != '>'; i++);
	i++;

	for(j = 0; production[i] != '\0'; i++, j++);
```

```
	return j;
}


//Start of functions meant only for displaying the result. (Doesn't affect the actual string parsing)

void get_stack_contents(char *t) {		//Stores stack contents in t.
	int i;
	char c[5];

	strcpy(t, "$");

	for(i = 0; i <= stack.top; i++) {
		int n = stack.states[i];
		sprintf(c, "%d", n);
		strcat(t, c);
	}
}

void get_remaining_input(char *string, int index, char *t) {		//Stores remaining Input string in t.
	int i, j;

	for(i = index, j = 0; string[i] != '\0'; i++, j++)
		t[j] = string[i];

	t[j] = '\0';
}

void print_contents(char *string, int index, char *matched_string) {		//Prints the required stuff.
	char t1[20], t2[20];

	get_stack_contents(t1);
	get_remaining_input(string, index, t2);

	printf("\t|  %-25s  |  %-25s  |   %25s  |  \t", t1, matched_string, t2);
}

//End of functions meant only for displaying the result.


void parse() {
	char string[100];
	char matched_string[100];

	initialize_stack();

	printf("\nEnter a string: ");
	scanf("%s", string);

	strcat(string, "$");		//Appending $ to end of input string.
	matched_string[0] = '\0';
```

```
	printf("\nThe reduction steps for the given string are as follows...\n\n");

	printf("\t| %-25s | %-25s | %25s | \t%-30s\n\n", "Stack", "Matched String", "Input String", "Action");

	int index = 0, m_index = 0;

	while(1) {
		char a = string[index];

		print_contents(string, index, matched_string);

		if(table.ACTION[get_top()][get_pos(0, a)][0] == 'S') {		//Shift Action. (Table entry starts with char 'S')
			int t = get_int(table.ACTION[get_top()][get_pos(0, a)]);
			push(t);//Push state t onto stack.
			index++;

			//Printing the result.
			char t1[20];
			char state[5];

			strcpy(t1, "Shift ");
			sprintf(state, "%d", t);
			strcat(t1, state);

			matched_string[m_index++] = a;
			matched_string[m_index] = '\0';

			printf("%-30s\n", t1);
		}

		else if(table.ACTION[get_top()][get_pos(0, a)][0] == 'R') {//Reduce Action.
			int i, j = get_int(table.ACTION[get_top()][get_pos(0, a)]);

			for(i = 0; i < get_length(augmented_grammar[j]); i++)	//Pop "length of string" times, w.r.t production 'j'.
				pop();

			int t = get_top();
			char A = augmented_grammar[j][0];	//Production head of 'j'th production. (non-terminal)

			push(table.GOTO[t][get_pos(1, A)]);	//Push state using GOTO of the table.

			//Printing the result.
			m_index -= get_length(augmented_grammar[j]);
			matched_string[m_index++] = A;
			matched_string[m_index] = '\0';
```

```
                        char t1[20];
                        strcpy(t1, "Reduce by ");
                        strcat(t1, augmented_grammar[j]);

                        printf("%-30s\n", t1);
                }

                else if(table.ACTION[get_top()][get_pos(0, a)][0] == 'a') {//Acceptance.
                        printf("%-30s\n", "Accept!!");
                        break;
                }

                else {                                                          //Error.
                        printf("%-30s\n", "Error!!\n\n");
                        printf("String doesn't belong to the language of the particular grammar!\n");
                        exit(0);
                }
        }

        printf("\nString accepted!\n");
}

// Parse
struct Stack {     //Holds states.
        int states[100];
        int top;
} stack;

void push(int a) {
        stack.top++;
        stack.states[stack.top] = a;
}

void pop() {
        int a = stack.states[stack.top];
        stack.top--;
}

int get_top() {             //Returns top of stack state.
        return stack.states[stack.top];
}

void initialize_stack() {   //Initialize stack to have state 0 on top.
        stack.top = -1;

        push(0);
}

int get_int(char *s) {              //Get integer part of the strings found in table entries.
        int i, j;
```

```
	char temp[10];

	for(i = 0; s[i] != ':'; i++);
	i++;

	for(j = i; s[i] != '\0'; i++)
		temp[i-j] = s[i];

	temp[i-j] = '\0';

	return atoi(temp);
}

int get_length(char *production) {		//Returns length of string in the production body.
	int i, j;

	for(i = 0; production[i] != '>'; i++);
	i++;

	for(j = 0; production[i] != '\0'; i++, j++);

	return j;
}


//Start of functions meant only for displaying the result. (Doesn't affect the actual string parsing)

void get_stack_contents(char *t) {		//Stores stack contents in t.
	int i;
	char c[5];

	strcpy(t, "$");

	for(i = 0; i <= stack.top; i++) {
		int n = stack.states[i];
		sprintf(c, "%d", n);
		strcat(t, c);
	}
}

void get_remaining_input(char *string, int index, char *t) {		//Stores remaining Input string in t.
	int i, j;

	for(i = index, j = 0; string[i] != '\0'; i++, j++)
		t[j] = string[i];

	t[j] = '\0';
}

void print_contents(char *string, int index, char *matched_string) {		//Prints the required stuff.
	char t1[20], t2[20];
```

```
	get_stack_contents(t1);
	get_remaining_input(string, index, t2);

	printf("\t| %-25s | %-25s |  %25s | \t", t1, matched_string, t2);
}

//End of functions meant only for displaying the result.


void parse() {
	char string[100];
	char matched_string[100];

	initialize_stack();

	printf("\nEnter a string: ");
	scanf("%s", string);

	strcat(string, "$");		//Appending $ to end of input string.
	matched_string[0] = '\0';

	printf("\nThe reduction steps for the given string are as follows...\n\n");

	printf("\t| %-25s | %-25s | %25s | \t%-30s\n\n", "Stack", "Matched String", "Input String", "Action");

	int index = 0, m_index = 0;

	while(1) {
		char a = string[index];

		print_contents(string, index, matched_string);

		if(table.ACTION[get_top()][get_pos(0, a)][0] == 'S') {		//Shift Action. (Table entry starts with char 'S')
			int t = get_int(table.ACTION[get_top()][get_pos(0, a)]);
			push(t);//Push state t onto stack.
			index++;

			//Printing the result.
			char t1[20];
			char state[5];

			strcpy(t1, "Shift ");
			sprintf(state, "%d", t);
			strcat(t1, state);

			matched_string[m_index++] = a;
			matched_string[m_index] = '\0';
```

```
                printf("%-30s\n", t1);
        }

        else if(table.ACTION[get_top()][get_pos(0, a)][0] == 'R') {//Reduce Action.
                int i, j = get_int(table.ACTION[get_top()][get_pos(0, a)]);

                for(i = 0; i < get_length(augmented_grammar[j]); i++)     //Pop "length of
string" times, w.r.t production 'j'.
                        pop();

                int t = get_top();
                char A = augmented_grammar[j][0];       //Production head of 'j'th
production. (non-terminal)

                push(table.GOTO[t][get_pos(1, A)]);       //Push state using GOTO of the
table.

                //Printing the result.
                m_index -= get_length(augmented_grammar[j]);
                matched_string[m_index++] = A;
                matched_string[m_index] = '\0';

                char t1[20];
                strcpy(t1, "Reduce by ");
                strcat(t1, augmented_grammar[j]);

                printf("%-30s\n", t1);
        }

        else if(table.ACTION[get_top()][get_pos(0, a)][0] == 'a') {//Acceptance.
                printf("%-30s\n", "Accept!!");
                break;
        }

        else {                                                          //Error.
                printf("%-30s\n", "Error!!\n\n");
                printf("String doesn't belong to the language of the particular grammar!\n");
                exit(0);
        }
    }

    printf("\nString accepted!\n");
}

//first_follow.h
int epsilon_flag = 0;

initialize_first_follow() {         //Initialize to null strings.
    int i;

    for(i = 0; i < no_of_terminals; i++)
```

```
		FIRST[0][i][0] = '\0';

	for(i = 0; i < no_of_nonterminals; i++) {
		FIRST[1][i][0] = '\0';
		FOLLOW[i][0] = '\0';
	}
}

void add_symbol(int flag, char *f, char *s) {		//Adds a symbol to FIRST or FOLLOW if it doesn't already exist in it.
	int i, j;
	int found;

	if(flag == 0) {		//For FIRST.
		for(i = 0; i < strlen(s); i++) {
			found = 0;

			for(j = 0; j < strlen(f); j++) {
				if(s[i] == f[j])
					found = 1;
			}

			if(!found) {
				char temp[2];
				temp[0] = s[i];
				temp[1] = '\0';
				strcat(f, temp);
			}
		}
	}

	else {		//For FOLLOW.
		for(i = 0; i < strlen(s); i++) {
			found = 0;

			if(s[i] == '@') {
				epsilon_flag = 1;
				continue;
			}

			for(j = 0; j < strlen(f); j++) {
				if(s[i] == f[j])
					found = 1;
			}

			if(!found) {
				char temp[2];
```

```
                                temp[0] = s[i];
                                temp[1] = '\0';
                                strcat(f, temp);
                        }
                }
        }
}

void first(char s) {
        if(isterminal(s)) {        //For terminals.
                FIRST[0][get_pos(0, s)][0] = s;
                FIRST[0][get_pos(0, s)][1] = '\0';
        }

        else {              //For non-terminals.
                int i, flag = 0;
                for(i = 0; i < no_of_productions; i++) {
                        if(augmented_grammar[i][0] == s) {              //Productions with head
as s.
                                int j;

                                for(j = 0; augmented_grammar[i][j] != '>'; j++);
                                j++;
                                char next_sym = augmented_grammar[i][j];

                                if(next_sym == '@') {  //Epsilon Production.
                                        add_symbol(0, FIRST[1][get_pos(1, s)], "@");
                                        flag = 1;
                                }

                                else {
                                        if(next_sym == s) {                //In case of left recursion,
to avoid infinite loop.
                                                if(flag)
                                                        next_sym =
augmented_grammar[i][++j];
                                                else
                                                        continue;
                                        }

                                        first(next_sym);                   //Recursive call, to find
FIRST of next symbol.

                                        if(isterminal(next_sym))        //Add first of next symbol
to first of current symbol.
                                                add_symbol(0, FIRST[1][get_pos(1, s)],
FIRST[0][get_pos(0, next_sym)]);
```

```
                                        else
                                                add_symbol(0, FIRST[1][get_pos(1, s)],
FIRST[1][get_pos(1, next_sym)]);
                                        }
                                }
                        }
                }
        }
}

void compute_first() {
        int i;

        for(i = 0; i < no_of_terminals; i++)
                first(terminals[i]);

        for(i = 0; i < no_of_nonterminals; i++)
                first(nonterminals[i]);

//      for(i = 0; i < no_of_nonterminals; i++)
//              printf("%s\n", FIRST[1][get_pos(1, nonterminals[i])]);
}



//FOLLOW

void follow(char s) {
        if(s == nonterminals[0])
                add_symbol(1, FOLLOW[0], "$");

        else if(s == nonterminals[1])
                add_symbol(1, FOLLOW[1], "$");

        int i, j;
        for(i = 0; i < no_of_productions; i++) {
                for(j = 3; j < strlen(augmented_grammar[i]); j++) {
                        epsilon_flag = 0;

                        if(augmented_grammar[i][j] == s) {
                                char next_sym = augmented_grammar[i][j+1];

                                if(next_sym != '\0') {          //If current symbol is not the last
symbol of production body.
                                        if(isterminal(next_sym))        //For terminals.
                                                add_symbol(1, FOLLOW[get_pos(1, s)],
FIRST[0][get_pos(0, next_sym)]);
```

```
                                        else {              //For non-terminals.
                                                add_symbol(1, FOLLOW[get_pos(1, s)],
FIRST[1][get_pos(1, next_sym)]);

                                                if(epsilon_flag) {          //If FIRST[next_sym] has
epsilon, find FOLLOW[next_sym].
                                                        follow(next_sym);
                                                        add_symbol(1, FOLLOW[get_pos(1, s)],
FOLLOW[get_pos(1, next_sym)]);
                                                }
                                        }
                                }

                                else {              //If current symbol is the last symbol of production
body.
                                        follow(augmented_grammar[i][0]);        //Follow of
production head.
                                        add_symbol(1, FOLLOW[get_pos(1, s)], FOLLOW[get_pos(1,
augmented_grammar[i][0])]);
                                }
                        }
                }
        }
}

compute_follow() {
        int i;

        for(i = 0; i < no_of_nonterminals; i++)
                follow(nonterminals[i]);

//      for(i = 0; i < no_of_nonterminals; i++)
//              printf("%s\n", FOLLOW[get_pos(1, nonterminals[i])]);
}


//Parsing Table.
struct Parsing_Table {    //Structure to represent the Parsing Table.
        char ACTION[30][100][100];
        int GOTO[30][100];
} table;

void initialize_table() {  //Initialize all entries to indicate Error.
        int i, j;

        for(i = 0; i < no_of_states; i++) {
                for(j = 0; j < no_of_terminals; j++)
                        strcpy(table.ACTION[i][j], "e");

                for(j = 0; j < no_of_nonterminals; j++)
                        table.GOTO[i][j] = -1;
```

```
	}
}

void print_table() {
	int i, j;

	printf("\n\nThe Parsing Table for the given grammar is...\n\n");

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

	for(i = 0; i < no_of_terminals; i++)
		printf("%10c", terminals[i]);

	printf(" | ");

	for(i = 1; i < no_of_nonterminals; i++)
		printf("%10c", nonterminals[i]);

	printf("\n\n");

	for(i = 0; i < no_of_states; i++) {
		printf("%10d | ", i);

		for(j = 0; j < no_of_terminals; j++) {
			if(!strcmp(table.ACTION[i][j], "e"))
				printf("%10s", ".");
			else
				printf("%10s", table.ACTION[i][j]);
		}

		printf(" | ");

		for(j = 1; j < no_of_nonterminals; j++) {
			if(table.GOTO[i][j] == -1)
				printf("%10s", ".");
			else
				printf("%10d", table.GOTO[i][j]);
		}

		printf("\n");
	}
}

void Goto(int i, int item, char *temp) {    //Computes goto for 'item'th item of 'i'th state.
	char t;

	strcpy(temp, items[i][item]);

	for(i = 0; temp[i] != '\0'; i++)
		if(temp[i] == '.') {
			t = temp[i];
```

```
                        temp[i] = temp[i+1];
                        temp[i+1] = t;
                        break;
                }
}

int get_state(char *t, int state) {        //Returns the state of a given item.
        int i, j;

        for(i = state; i < (no_of_states + state); i++) { //Start searching from current state and
then wrap around.
                for(j = 0; j < no_of_items[i % no_of_states]; j++) {
                        if(!strcmp(t, items[i % no_of_states][j]))
                                return i % no_of_states;
                }
        }

        printf("No match for string! (%s)\n", t);
}

int get_pos(int flag, char symbol) {    //Returns index of a terminal or a non-terminal from
the corresponding arrays.
        int i;

        if(flag == 0)
                for(i = 0; i < no_of_terminals; i++) {
                        if(terminals[i] == symbol)
                                return i;
                }
        else
                for(i = 0; i < no_of_nonterminals; i++) {
                        if(nonterminals[i] == symbol)
                                return i;
                }

        if(flag == 0)
                printf("Terminal not found in get_pos! (%c)\n", symbol);
        else
                printf("Non-terminal not found in get_pos! (%c)\n", symbol);
}

int get_production_no(char * item) {     //Given an item, it returns the production number of the
equivalent production.
        int i, j;

        char production[20];

        for(i = 0, j = 0; item[i] != '\0'; i++)
```

```
		if(item[i] != '.') {
			production[j] = item[i];
			j++;
		}

	if(j == 3) {			//If it's an epsilon production, the production won't have a body.
		production[j] = '@';
		j++;
	}

	production[j] = '\0';

	for(i = 0; i < no_of_productions; i++) {
		if(!strcmp(production, augmented_grammar[i]))
			return i;
	}

	printf("Production not found! (%s)\n", production);
}

void compute_action() {
	int i, item, j;
	char temp[100], symbol;

	for(i = 0; i < no_of_states; i++) {
		for(item = 0; item < no_of_items[i]; item++) {
			char *s = strchr(items[i][item], '.');		//Returns a substring starting with
'.'

			if(!s) {	//In case of error.
				printf("Item not found! State = %d, Item = %d\n", i, item);
				exit(-1);
			}

			if(strlen(s) > 1) {//dot is not at end of string. SHIFT ACTION!!
				if(isterminal(s[1])) {			//For terminals. Rule 1.
					if(strcmp(table.ACTION[i][get_pos(0,s[1])], "e")) {
	//Multiple entries conflict.
						printf("\n\nConflict(1): Multiple entries found for
(%d, %c)\n", i, s[1]);
						printf("\nGrammar is not in LR(0)!\n");
						exit(-1);
					}

					char state[3];

					Goto(i, item, temp);		//Store item in temp.
					j = get_state(temp, i);

					sprintf(state, "%d", j);
					strcpy(temp, "S:");
```

```
                                        strcat(temp, state);

                                        strcpy(table.ACTION[i][get_pos(0, s[1])], temp);
                                }

                                else {          //For non-terminals. Rule 4.

                                        Goto(i, item, temp);    //Store item in temp.
                                        j = get_state(temp, i);

                                        if(table.GOTO[i][get_pos(1, s[1])] == -1)  //To avoid multiple
entries.
                                                table.GOTO[i][get_pos(1, s[1])] = j;
                                }
                        }

                        else {  //dot is at end of string. Rule 2. REDUCE ACTION!!
                                char f[10], production_no[3];
                                int k, n;
                                n = get_production_no(items[i][item]);          //Get production
number from Augmented Grammar.

                                sprintf(production_no, "%d", n);
                                strcpy(temp, "R:");
                                strcat(temp, production_no);

                                strcpy(f, FOLLOW[get_pos(1, items[i][item][0])]);       //Get follow
of production head.
                                for(k = 0; f[k] != '\0'; k++) {
                                        if(strcmp(table.ACTION[i][get_pos(0, f[k])], "e")) {
        //Multiple entries conflict.
                                                printf("\n\nConflict(3): Multiple entries found for
(%d, %c)\n", i, f[k]);
                                                printf("\nGrammar is not in LR(0)!\n");
                                                exit(-1);
                                        }

                                        strcpy(table.ACTION[i][get_pos(0, f[k])], temp);
                                }
                        }
                }
        }

        strcpy(table.ACTION[1][get_pos(0, '$')], "acc");        //Accept-entry for item [S'->S.]
}

void create_parsing_table() {
        initialize_table();

        compute_action();
```

```
	print_table();
}

//End of Parsing Table.

//Parser.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#include"closure_goto.h"
#include"parsingtable.h"
#include"first_follow.h"
#include"parse.h"

int main() {
	start();  //Compute closure and goto.

	initialize_first_follow();
	compute_first();
	compute_follow();

	create_parsing_table();

	parse(); //Parse the input string.

	return 0;
}
```

```
mona@mona-VirtualBox:~/CD Lab/LR(0) Parser$ gcc parser.c -w
mona@mona-VirtualBox:~/CD Lab/LR(0) Parser$ ./a.out
Enter number of productions:3
Enter the individual production rules separately :
S->Aa
S->bAc
A->d

Augmented Grammar is...

Z->S
S->Aa
S->bAc
A->d

Number of states = 8.

Items in State 0...
Z->.S
S->.Aa
A->.d
S->.bAc

Items in State 1...
Z->S.

Items in State 2...
S->A.a

Items in State 3...
A->d.

Items in State 4...
S->b.Ac
A->.d

Items in State 5...
S->Aa.
Items in State 6...
S->bA.c

Items in State 7...
S->bAc.


The Parsing Table for the given grammar is...

                     a         b         c         d         $ |         S         A

        0 |          .       S:4         .       S:3         . |         1         2
        1 |          .         .         .         .       acc |         .         .
        2 |        S:5         .         .         .         . |         .         .
        3 |        R:3         .       R:3         .         . |         .         .
        4 |          .         .         .       S:3         . |         .         6
        5 |          .         .         .         .       R:1 |         .         .
        6 |          .         .       S:7         .         . |         .         .
        7 |          .         .         .         .       R:2 |         .         .

Enter a string: bdc

The reduction steps for the given string are as follows...

        |  Stack                     |  Matched String            |              Input String  |      A
ction

        |  $0                        |                            |                      bdc$  |      S
hift 4
        |  $04                       |  b                         |                       dc$  |      S
hift 3
        |  $043                      |  bd                        |                        c$  |      R
educe by A->d
        |  $046                      |  bA                        |                        c$  |      S
hift 7
        |  $0467                     |  bAc                       |                         $  |      R
educe by S->bAc
        |  $01                       |  S                         |                         $  |      A
ccept!!

String accepted!
mona@mona-VirtualBox:~/CD Lab/LR(0) Parser$
```

ASSIGNMENT-9

```
/*LR(1) PARSER  */
/* Defined grammar
E->E+T
E->T
T->T*F
T->F
F->(E)
F->i*/

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
void push(char *,int *,char);
char stacktop(char *);
void isproduct(char,char);
int ister(char);
int isnter(char);
int isstate(char);
void error();
void isreduce(char,char);
char pop(char *,int *);
void printt(char *,int *,char [],int);
void rep(char [],int);
struct action
{
        char row[6][5];
};
const struct action A[12]=
{
        {"sf","emp","emp","se","emp","emp"},
        {"emp","sg","emp","emp","emp","acc"},
        {"emp","rc","sh","emp","rc","rc"},
        {"emp","re","re","emp","re","re"},
        {"sf","emp","emp","se","emp","emp"},
        {"emp","rg","rg","emp","rg","rg"},
        {"sf","emp","emp","se","emp","emp"},
        {"sf","emp","emp","se","emp","emp"},
        {"emp","sg","emp","emp","sl","emp"},
        {"emp","rb","sh","emp","rb","rb"},
        {"emp","rb","rd","emp","rd","rd"},
        {"emp","rf","rf","emp","rf","rf"}
};
struct gotol
{
        char r[3][4];
};
const struct gotol G[12]=
{
        {"b","c","d"},
        {"emp","emp","emp"},
```

```
{"emp","emp","emp"},
{"emp","emp","emp"},
{"i","c","d"},
{"emp","emp","emp"},
{"emp","j","d"},
{"emp","emp","k"},
{"emp","emp","emp"},
{"emp","emp","emp"},
};
char ter[6]={'i','+','*',')','(','$'};
char nter[3]={'E','T','F'};
char states[12]={'a','b','c','d','e','f','g','h','m','j','k','l'};
char stack[100];
int top=-1;
char temp[10];
struct grammar
{
        char left;  char right[5]; };
const struct grammar rl[6]={
{'E',"e+T"},
{'E',"T"},
{'T',"T*F"},
{'T',"F"},
{'F',"(E)"},
{'F',"i"},
};
void main()
{
char inp[80],x,p,dl[80],y,bl='a';
int i=0,j,k,l,n,m,c,len;
printf(" Enter the input :");
scanf("%s",inp);
len = strlen(inp);
inp[len]='$';
inp[len+1]='\0';
push(stack,&top,bl);
printf("\n stack \t\t\t input");
printt(stack,&top,inp,i);
do
{
 x=inp[i];   p=stacktop(stack);
                                isproduct(x,p);
if(strcmp(temp,"emp")==0)
error();
if(strcmp(temp,"acc")==0)
break;
                                else
{
if(temp[0]=='s')
{
push(stack,&top,inp[i]);     push(stack,&top,temp[1]);
```

```
i++;
}
else
{
if(temp[0]=='r')
{
j=isstate(temp[1]);
strcpy(temp,rl[j-2].right);
dl[0]=rl[j-2].left;
dl[1]='\0';
n=strlen(temp);
for(k=0;k<2*n;k++)
pop(stack,&top);
for(m=0;dl[m]!='\0';m++)       push(stack,&top,dl[m]);
l=top;
y=stack[l-1];
isreduce(y,dl[0]);     for(m=0;temp[m]!='\0';m++)       push(stack,&top,temp[m]);
}
}
}
printt(stack,&top,inp,i);
}
while(inp[i]!='\0');
if(strcmp(temp,"acc")==0)
printf("\nThe string is accepted ");
else
printf("\nThe string is not accepted");

}
void push(char *s,int *sp,char item)
{
if(*sp==100)   printf("The stack is full ");
else
{
*sp=*sp+1;
s[*sp]=item;
}
}
char stacktop(char *s)
{
char i;  i=s[top];
return i;
}
void isproduct(char x,char p)
{
int k,l;
 k=ister(x);
l=isstate(p);
strcpy(temp,A[l-1].row[k-1]);
}
int ister(char x)
```

```
{
int i;
for(i=0;i<6;i++)
if(x==ter[i])
return i+1;
return 0;
}
int isnter(char x)
{
int i;
for(i=0;i<3;i++)
if(x==nter[i])
return i+1;
return 0;
}
int isstate(char p)
{
int i;
for(i=0;i<12;i++)
if(p==states[i])
return i+1;
return 0;
}
void error()
{
printf(" error in the input ");
exit(0);
}
void isreduce(char x,char p)
{
int k,l;   k=isstate(x);
l=isnter(p);
strcpy(temp,G[k-1].r[l-1]);
}
char pop(char *s,int *sp)
{
char item;   if(*sp==-1)
printf(" stack is empty ");
else
{
item=s[*sp];
*sp=*sp-1;
}
return item;
}
void printt(char *t,int *p,char inp[],int i)
{
int r;
printf("\n");
for(r=0;r<=*p;r++)
rep(t,r);
```

```
printf("\t\t\t");
for(r=i;inp[r]!='\0';r++)
printf("%c",inp[r]);
}
void rep(char t[],int r)
{
char c;
c=t[r];
switch(c)
{
        case 'a': printf("0");
                  break;
        case 'b': printf("1");
                  break;
        case 'c': printf("2");
                  break;
        case 'd': printf("3");
                  break;
        case 'e': printf("4");
                  break;
        case 'f': printf("5");
                  break;
        case 'g': printf("6");
                  break;
        case 'h': printf("7");
                  break;
        case 'm': printf("8");
                  break;
        case 'j': printf("9");
                  break;
        case 'k': printf("10");
                  break;
        case 'l': printf("11");
                  break;
        default  :printf("%c",t[r]);
                  break;
}
}
```

```
mona@mona-VirtualBox:~/CD Lab$ gcc "LR(1)".c
mona@mona-VirtualBox:~/CD Lab$ ./a.out
 Enter the input :i+i*i

 stack                   input
0                       i+i*i$
0i5                     +i*i$
0F3                     +i*i$
0T2                     +i*i$
0E1                     +i*i$
0E1+6                   i*i$
0E1+6i5                 *i$
0E1+6F3                 *i$
0E1+6T9                 *i$
0E1+6T9*7                       i$
0E1+6T9*7i5                     $
0E1+6T9*7F10                    $
0E1+6T9                 $
0E1                     $
The string is accepted mona@mona-VirtualBox:~/CD Lab$ 3
```