Compiler Design Laboratory

Assignments

Final Submitted BY

Hutesh Kumar Gauttam

Roll NO: 115CS0209

Section : S1

Program1: check string

%%

[a-zA-Z\_][\_a-zA-Z0-9]\* {printf("valid\n");}

.\* {printf("invalid\n");}

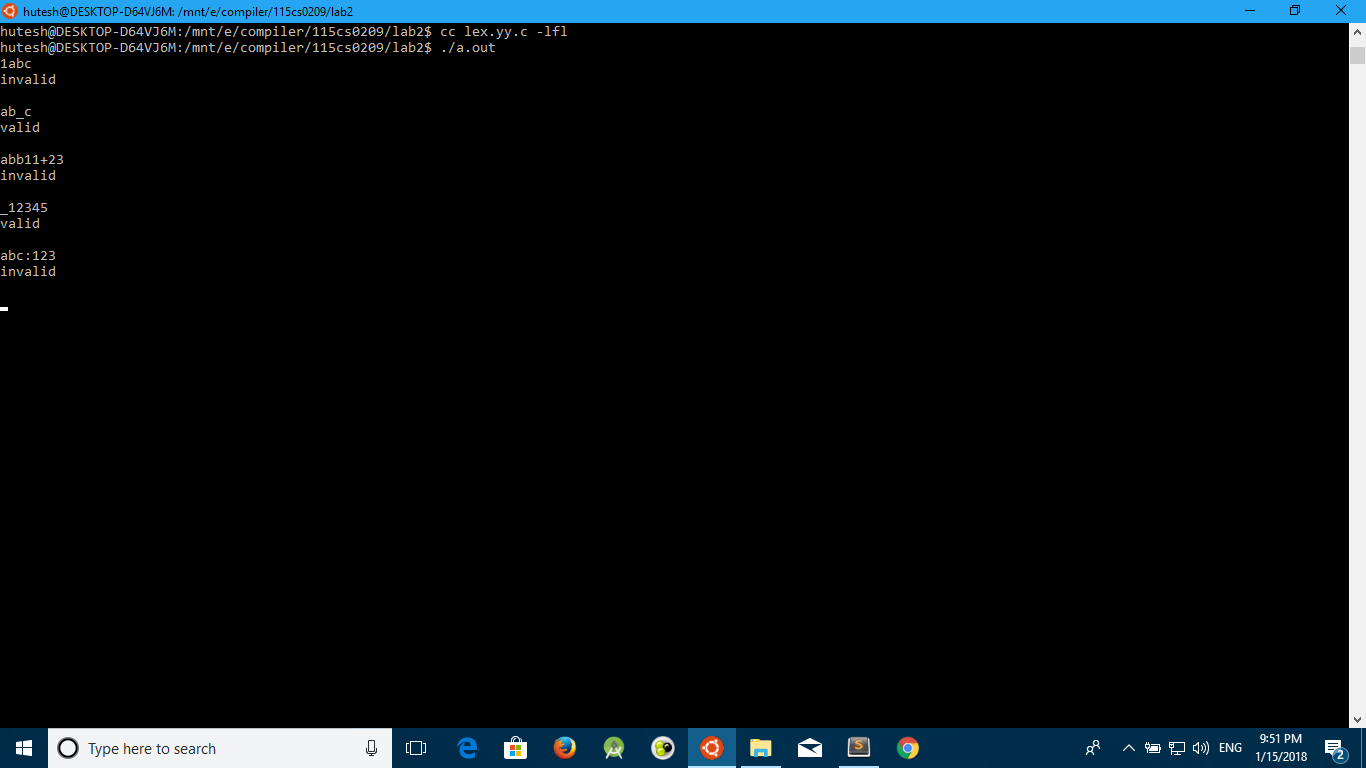
%%

main(int argc, char \*\*argv){

yylex();

}

Output:



Program2: for binary string validation

%%

(0|1(00|11)\*(01|10))(00|11|(01|10)(00|11)\*(01|10))\* {printf("valid\n");}

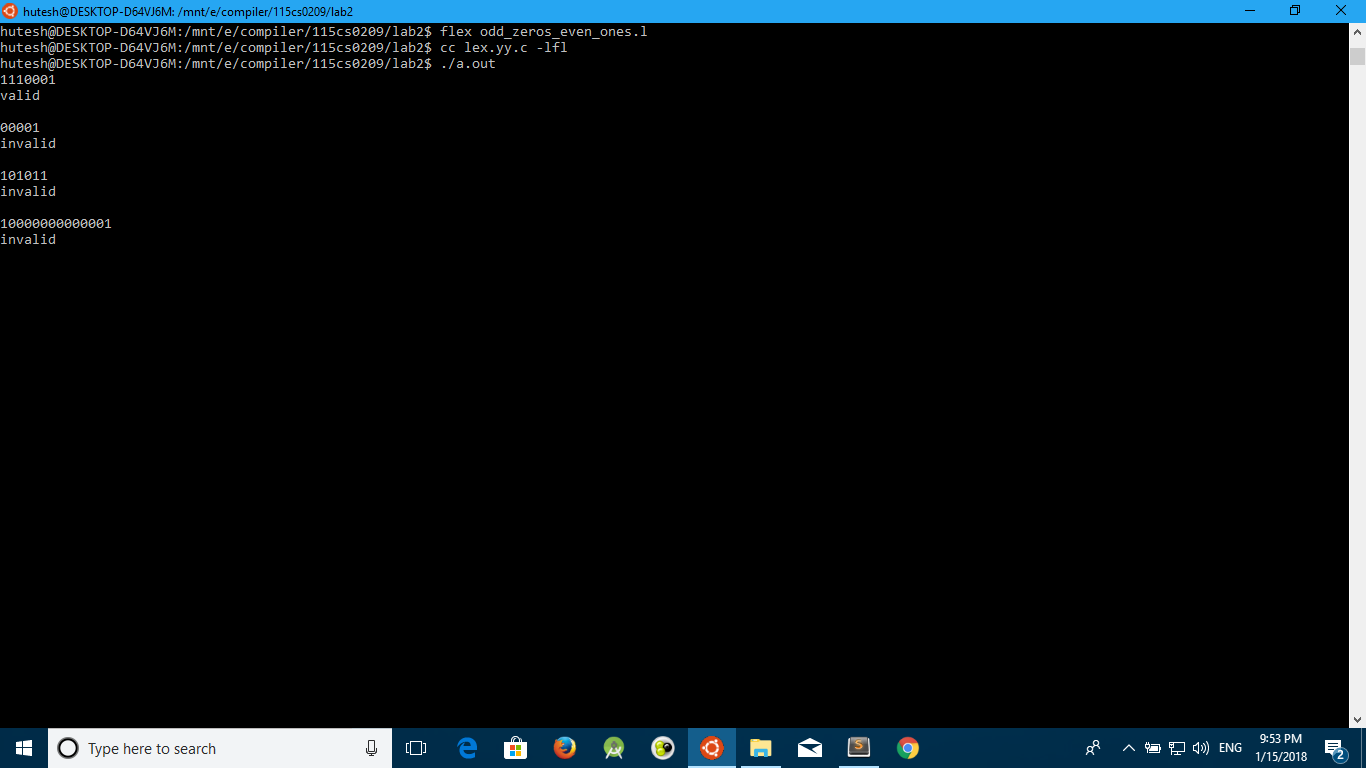
.\* {printf("invalid\n");}

%%

main(int argc, char \*\*argv){

yylex();

}



Program3: check octal and binary constant

%{

%};

%%

[B][0-1]\* {printf("UNSIGNED BINARY CONSTANT\n");}

[B][-][0-1]\* {printf("NEGETIVE BINARY CONSTANT\n");}

[B][+][0-1]\* {printf("POSITIVE BINARY CONSTANT\n");}

[O][0-7]\* {printf("UNSIGNED OCTAL CONSTANT\n");}

[O][-][0-7]\* {printf("NEGETIVE OCTAL CONSTANT\n");}

[O][+][0-7]\* {printf("POSITIVE OCTAL CONSTANT\n");}

[D][0-9]\* {printf("UNSIGNED DECIMAL CONSTANT\n");}

[D][-][0-9]\* {printf("NEGETIVE DECIMAL CONSTANT\n");}

[D][+][0-9]\* {printf("POSITIVE DECIMAL CONSTANT\n");}

[H][0-9]\*[A-F]\* {printf("UNSIGNED HEXADECIMAL CONSTANT\n");}

[H][-][0-9]\*[A-F]\* {printf("NEGETIVE HEXADECIMAL CONSTANT\n");}

[H][+][0-9]\*[A-F]\* {printf("POSITIVE HEXADECIMAL CONSTANT\n");}

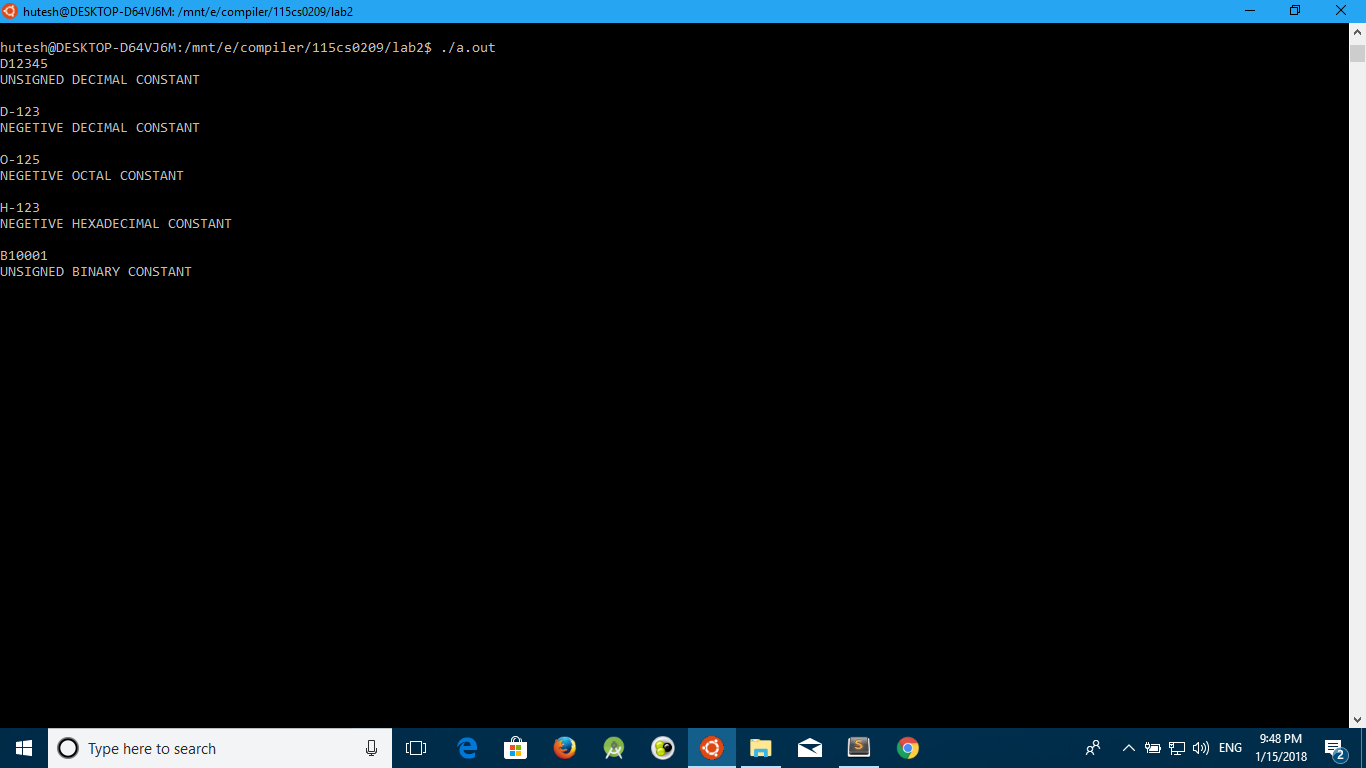
.\* {printf("invalid\n");}

%%

main(int argc, char \*\*argv){

yylex();

}



Program4: check it is string or not

%%

["].\*["] {printf("String constant\n");}

['].['] {printf("Character constant\n");}

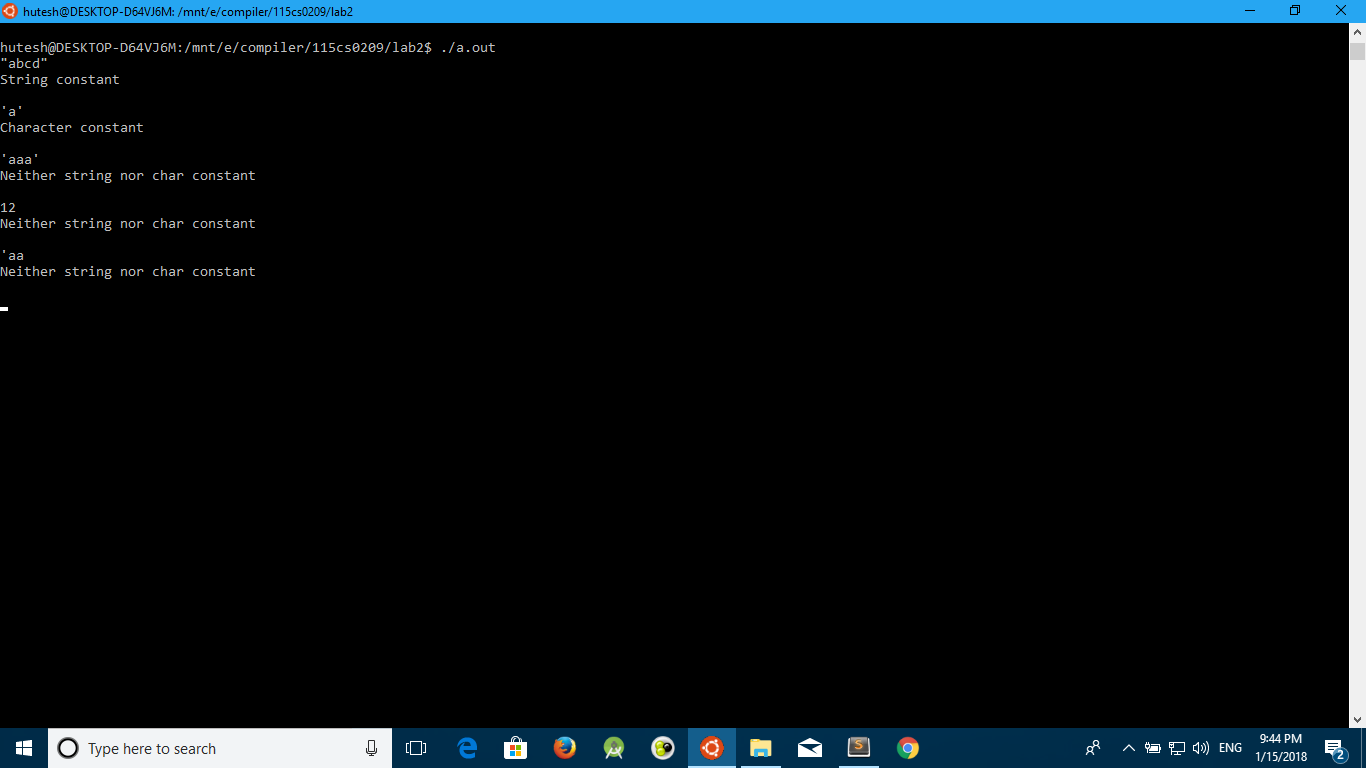
.\* {printf("Neither string nor char constant\n");}

%%

main(int argc, char \*\*argv){

yylex();

}



Lexical Analyser:

Code:

#include<bits/stdc++.h>

using namespace std;

int check(string s)

{

int l=s.length();

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

{

if(s[i]=='0'||s[i]=='1'||s[i]=='2'||s[i]=='3'||s[i]=='4'||s[i]=='5'||s[i]=='6'||s[i]=='7'||s[i]=='8'||s[i]=='9')

continue;

else

return 0;

}

return 1;

}

string temp="";

int main()

{

char s[1000];

cin.getline(s,1000);

map<string,int>code;

map<string,string>value;

code["begin"]=1;

code["end"]=1;

code["if"]=1;

code["else"]=1;

code["then"]=1;

code["identifier"]=1;

code["constant"]=1;

code["=="]=1;

code["!="]=1;

code[">="]=1;

code["<="]=1;

code["<"]=1;

code[">"]=1;

value["begin"]="-";

value["end"]="-";

value["if"]="-";

value["else"]="-";

value["then"]="-";

value["identifier"]="(6,sym\_tab\_ptr)";

value["constant"]="(7,sym\_tab\_ptr)";

value["=="]="(8,1)";

value["!="]="(8,2)";

value[">="]="(8,3)";

value["<="]="(8,4)";

value["<"]="(8,5)";

value[">"]="(8,6)";

int i=0;

int l =strlen(s);

cout<<"token"<<"\t\t"<<"code"<<"\t\t"<<"value"<<"\n";

while(i<=l)

{

if(s[i]==' '||s[i]=='\0')

{

if(code[temp]!=0)

cout<<temp<<"\t\t"<<code[temp]<<"\t\t"<<value[temp]<<"\n";

else

{

if(temp[0]!='0'||temp[0]!='0'||temp[0]!='0'||temp[0]!='0'||temp[0]!='0'||temp[0]!='0'||temp[0]!='0'||temp[0]!='0')

{

int p=check(temp);

if(p==1)

{

temp="constant";

cout<<temp<<"\t\t"<<code[temp]<<"\t\t"<<value[temp]<<"\n";

}

else

{

temp="identifier";

cout<<temp<<"\t\t"<<code[temp]<<"\t\t"<<value[temp]<<"\n";

}

}

else

cout<<"invalid IDENTIFIERS\n";

}

temp="";

}

else

{

temp=temp+s[i];

}

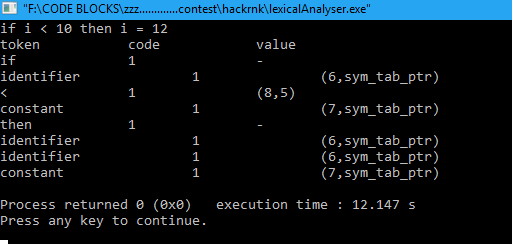
i++;

}

return 0;

}

OUTPUT:



Detect A Expression it can be generated from grammer or NOT:

Code:

#include<bits/stdc++.h>

using namespace std;

string s1="a";

string s2="e";

string s3="";

string s4="";

string s;

string Arr[]={"Aa","b"};

string Brr[]={"c"};

int checkParser(string s1,string s2,string s3,string s4,string input,int l,int p)

{

s=s1+s3+s4+s2;

if(s.length()>l)

return 0;

else if(s==input&&p==1)

return 1;

else if((s!=input)&&(s.length()==l))

return 0;

else if((s!=input)&&(s.length()<l&&p==1))

return 0;

else

{

for(int i=0;i<2;i++)

{

string s33=s3;

s33.erase(0,1);

s33=Arr[i]+s33;

for(int j=0;j<1;j++)

{

string s44=s4;

s44=s44.erase(0,1);

s44=Brr[j]+s44;

//cout<<s1+s33+s44+s2<<"\n";

if(checkParser(s1,s2,s33,s44,input,l,i))

return 1;

} }

}

return 0;

}

int main()

{

cout<<"Enter no of test cases\n";

int t;

cin>>t;

while(t--)

{

string input;

cout<<"Enter string to check that it can be generatd by grammer or not\n";

cin>>input;

int p=0;

int l=input.length();

if(checkParser(s1,s2,s3,s4,input,l,p))

cout<<"Can Generate This String"<<"\n";

else

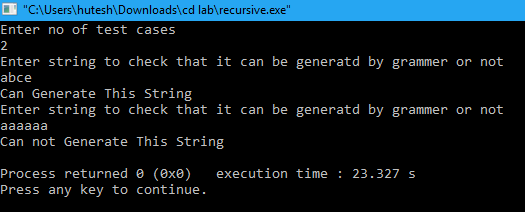
cout<<"Can not Generate This String"<<"\n";

}

return 0;

}

Output:



Remove Left Recursion:

Code:

#include<stdio.h>

#include<string.h>

#define SIZE 10

int main () {

char non\_terminal;

char beta,alpha;

int num;

int i;

char production[10][SIZE];

int index=3;

printf("Enter Number of Production : ");

scanf("%d",&num);

printf("Enter the grammar :\n");

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

scanf("%s",production[i]);

}

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

printf("\nGRAMMAR : : : %s",production[i]);

non\_terminal=production[i][0];

if(non\_terminal==production[i][index]) {

alpha=production[i][index+1];

printf(" is left recursive.\n");

while(production[i][index]!=0 && production[i][index]!='|') {

index++; }

if(production[i][index]!=0) {

beta=production[i][index+1];

printf("Grammar without left recursion:\n");

printf("%c->%c%c\'",non\_terminal,beta,non\_terminal);

printf("\n%c\'->%c%c\'|b\n",non\_terminal,alpha,non\_terminal);

}

else

printf(" can't be reduced\n");

}

else

printf(" is not left recursive.\n");

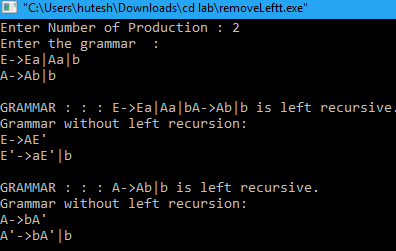
index=3;

}

getche();

}

OUTPUT:



Left Factoring:

Code:

#include <bits/stdc++.h>

using namespace std;

#define max 1000000000

int main()

{

long long int i,j,k,l,n,m=max,mini,ma=0;

string s[100],st,ch,sc="",result,fs,maxi,rs="";

vector<string>ss;

vector<string>sp;

cout<<"Enter No of Lines \n";

cin>>n;

for(i=1;i<=n;i++)

{

cin>>s[i]; //Grammer

}

for(i=1;i<=n;i++)

{

st=s[i];

sc="";

for(j=0;j<st.length();j++)

{

if(i==1)

{

fs=st[0];

}

if(st[j]=='=')

{

l=j;

}

}

if(i==1)

{

for(k=l+1;k<st.length();k++)

{

if(st[k]=='|')

{

ss.push\_back(sc);

sc="";

}

if(st[k]!='|')

{

ch=st[k];

sc=sc+ch;

}

}

ss.push\_back(sc);

}

}

for(k=0;k<ss.size();k++)

{

mini=ss[k].size();

m=min(m,mini);

maxi=ss[k];

}

cout<<"----------------------------------------------------\n";

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

char current = ss[0][i];

for (int j=1 ; j<ss.size(); j++) {

if (ss[j][i] != current) {

break;

}

result.push\_back(current);

}

}

for(j=0;j<ss.size();j++)

{

maxi=ss[j];

for(k=0;k<maxi.length();k++)

{

if(k>=result.length())

{

rs=rs+maxi[k];

}

}

if(j!=ss.size()-1)

{

rs=rs+'|';

}

}

cout<<fs<<"="<<result<<fs<<"'"<<endl;

cout<<fs<<"'"<<"="<<rs<<endl;

for(i=2;i<=n;i++){

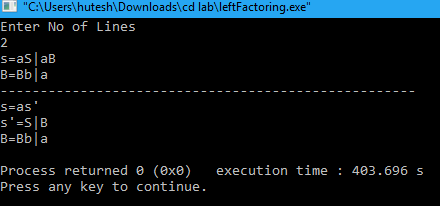
cout<<s[i]<<endl;

}

return 0;

}

OUTPUT:



Top Down Parsing:

Code:

#include<stdio.h>

#include<string.h>

void check(void);

void set\_value(void);

void get\_value(void);

void display\_output\_string(void);

int iptr=0,optr=0,current\_optr=0;

char output\_string[20],current\_output\_string[20],input\_string[20],temp\_string[20];

int main(){

printf("\nEnter the string to check From the grammer: ");

scanf("%s",input\_string);

check();

return 0;}

void check(void){

int flag=1,rule2\_index=1;

strcpy(output\_string,"S");

printf("\nThe output string in different stages are:\n");

while(iptr<=strlen(input\_string)){

if(strcmp(output\_string,temp\_string)!=0)

{

display\_output\_string();

}

if((iptr!=strlen(input\_string)) || (optr!=strlen(output\_string))){

if(input\_string[iptr]==output\_string[optr]){

iptr=iptr+1;

optr=optr+1;}

else{

if(output\_string[optr]=='S'){

memset(output\_string,0,strlen(output\_string));

strcpy(output\_string,"aBd");}

else if(output\_string[optr]=='B'){

set\_value();

if(rule2\_index==1&&input\_string[iptr]=='p'){

memset(output\_string,0,strlen(output\_string));

strcpy(output\_string,"apbd");}

else{

memset(output\_string,0,strlen(output\_string));

strcpy(output\_string,"add");}}

else if(output\_string[optr]=='b' && input\_string[iptr]=='d'){

rule2\_index=2;

get\_value();

iptr=iptr-1;}

else{

printf("\nThe given string, '%s' is NOT a VALID string.\n\n",input\_string);

break;}}}

else{

printf("\nThe given string, '%s' is VALID string.\n\n",input\_string);

break;}}}

void set\_value(void){ //setting values

current\_optr=optr;

strcpy(current\_output\_string,output\_string);

return;}

void get\_value(void){ // obtaining previous values

optr=current\_optr;

memset(output\_string,0,strlen(output\_string));

strcpy(output\_string,current\_output\_string);

return;}

void display\_output\_string(void){

printf("%s\n",output\_string);

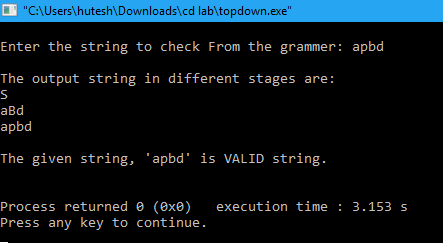
memset(temp\_string,0,strlen(temp\_string));

strcpy(temp\_string,output\_string);

return;

}

OUTPUT:



1. **Eliminate left factoring and left recursion from a given grammar.**

**PROGRAM:**

#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 (in the form X->Xy|Xx|x) : \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("For removal of left recursion, special character used is : %s",spcl);

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

cout<<"\nFor removal of left factoring, special character used is : Y";

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;

}

//Condition for left factoring

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;

}

}

}

//Display of Output

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)

{

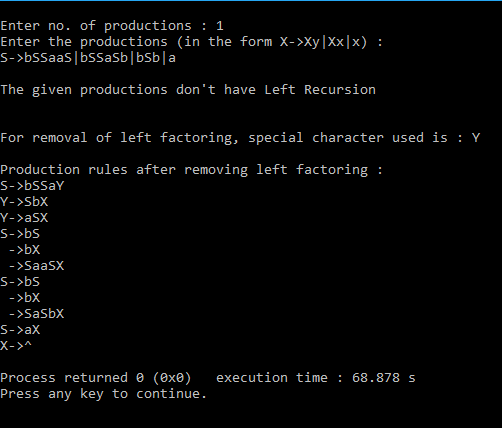
//cout<<"Production "<<++cnt3<<" is: ";

cout<<rule[cnt4].lf<<"->"<<rule[cnt4].rt<<endl;

}

}

}

****

1. **Find first and follow sets of a given grammar and check whether the grammar is LL1 or not.**

**PROGRAM:**

#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);

//For finding first

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];

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);

printf("Enter the productions in the form of X->xy and individual rules separately:\n");

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("LL1 Table\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");}}

Implementation OF LL(1):

#include<stdio.h>

#include<bits/stdc++.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);

//For finding first

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 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 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 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];

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);

printf("Enter the productions in the form of X->xy and individual rules separately:\n");

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("LL1 Table\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");}}}

Implementation of LR(0) Parser:

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

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("\nNo. of states = %d.\n", no\_of\_states);

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

printf("\n\nItems in State %d...\n\n", i);

for(j = 0; j < no\_of\_items[i]; j++)

printf("%s\n", items[i][j]);

}

}

void start() {

char str[100];

printf("How many productions:");

scanf("%d", &no\_of\_productions);

printf("Enter the productions...\n");

int i;

for(i = 1; i <= no\_of\_productions; i++)

scanf("%s", augmented\_grammar[i]);

printf("\n\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();

}

//#include"parsingtable.h"///////////////////////////////////////////////////////////////////////////////////////////////////////////////

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("\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();

}

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])]);

}

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");

}

int main() {

start(); //Compute closure and goto.

initialize\_first\_follow();

compute\_first();

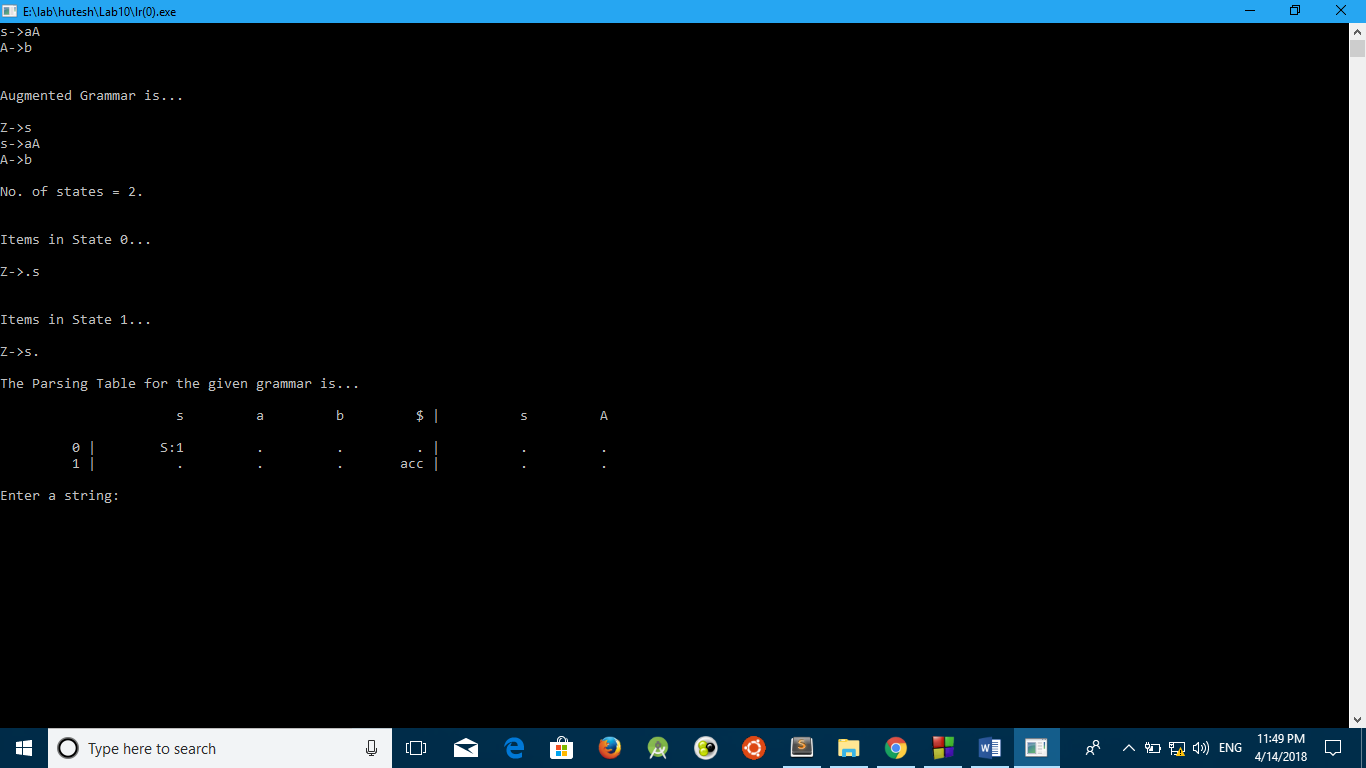
compute\_follow();

create\_parsing\_table();

parse(); //Parse the input string.

return 0;

}



Implementation of CLR(1):

#include<bits/stdc++.h>

using namespace std;

struct prod

{

string lhs;

string rhs;

};

int main()

{

struct prod p[3];

p[0].lhs="S";

p[0].rhs="AA";

p[1].rhs="aA";

p[1].lhs="A";

p[2].lhs="A";

p[2].rhs="b";

cout<<"The grammar for CLR(1) parser is\n"<<p[0].lhs<<" -> "<<p[0].rhs<<"\n"<<p[1].lhs<<" -> "<<p[1].rhs<<"\n"<<p[2].lhs<<" -> "<<p[2].rhs<<"\n";

string tbl[7][5];

int i,j;

for(i=0;i<7;i++)

for(j=0;j<5;j++)

tbl[i][j]="";

map<char,int> m1;

map<int,char> m2;

m1['a']=0;

m1['b']=1;

m1['$']=2;

m1['A']=3;

m1['S']=4;

m2[0]='a';

m2[1]='b';

m2[2]='$';

m2[3]='A';

m2[4]='S';

tbl[0][0]="S3";

tbl[0][1]="S4";

tbl[0][3]="G2";

tbl[0][4]="G1";

tbl[1][2]="A";

tbl[2][0]="S3";

tbl[2][1]="S4";

tbl[2][3]="G5";

tbl[3][0]="S3";

tbl[3][1]="S4";

tbl[3][3]="G6";

tbl[4][0]="R3";

tbl[4][1]="R3";

tbl[4][2]="R3";

tbl[5][0]="R1";

tbl[5][1]="R1";

tbl[5][2]="R1";

tbl[6][0]="R2";

tbl[6][1]="R2";

tbl[6][2]="R2";

cout<<"The parsing tbl for the above grammar is \n";

cout<<setw(10)<<"State";

for(i=0;i<5;i++)

cout<<setw(10)<<m2[i];

cout<<endl;

for(i=0;i<7;i++)

{

cout<<setw(10)<<i;

for(j=0;j<5;j++)

{

cout<<setw(10)<<tbl[i][j];

}

cout<<"\n\n\n";

}

while(1)

{

cout<<"\n\nEnter the string to check whether it belongs to the above grammar: ";

string s;

cin>>s;

s=s+"$";

stack<int> st;

st.push(0);

j=0;

int flag=0;

cout<<setw(20)<<"current state"<<setw(10)<<"input"<<setw(50)<<"Operations performed\n";

while(1)

{

int curr\_state=st.top();

cout<<setw(20)<<curr\_state<<setw(10)<<s[j];

if(tbl[curr\_state][m1[s[j]]].length()!=0)

{

if(tbl[curr\_state][m1[s[j]]][0]=='S')

{

cout<<setw(50)<<"Shift right";

st.push(s[j]);

st.push(tbl[curr\_state][m1[s[j]]][1]-'0');

j++;

}

else if(tbl[curr\_state][m1[s[j]]][0]=='R')

{

int l=tbl[curr\_state][m1[s[j]]][1]-'1';

int k,a,b;

for(k=0;k<2\*(p[l].rhs.length());k++)

{

st.pop();

}

a=st.top();

//cout<<"The current top state is "<<a<<endl;

st.push(p[l].lhs[0]);

//cout<<"THe goto operation performed is "<<p[l].lhs[0]<<endl;

if(tbl[a][m1[st.top()]].length()>1 && tbl[a][m1[st.top()]][0]=='G')

{

st.push(tbl[a][m1[st.top()]][1]-'0');

cout<<setw(50)<<"Reduce with production "<<p[l].lhs<<" -> "<<p[l].rhs<<" and goto state "<<st.top();

}

else break;

}

else if(tbl[curr\_state][m1[s[j]]][0]=='A')

{

flag=1;

break;

}

cout<<endl;

}

else

{

break;

}

}

if(flag==1)

cout<<"\nString is accepted\n";

else cout<<"\nString is not accepted\n";

char ch;

cout<<"Enter y to check another string else enter n"<<endl;

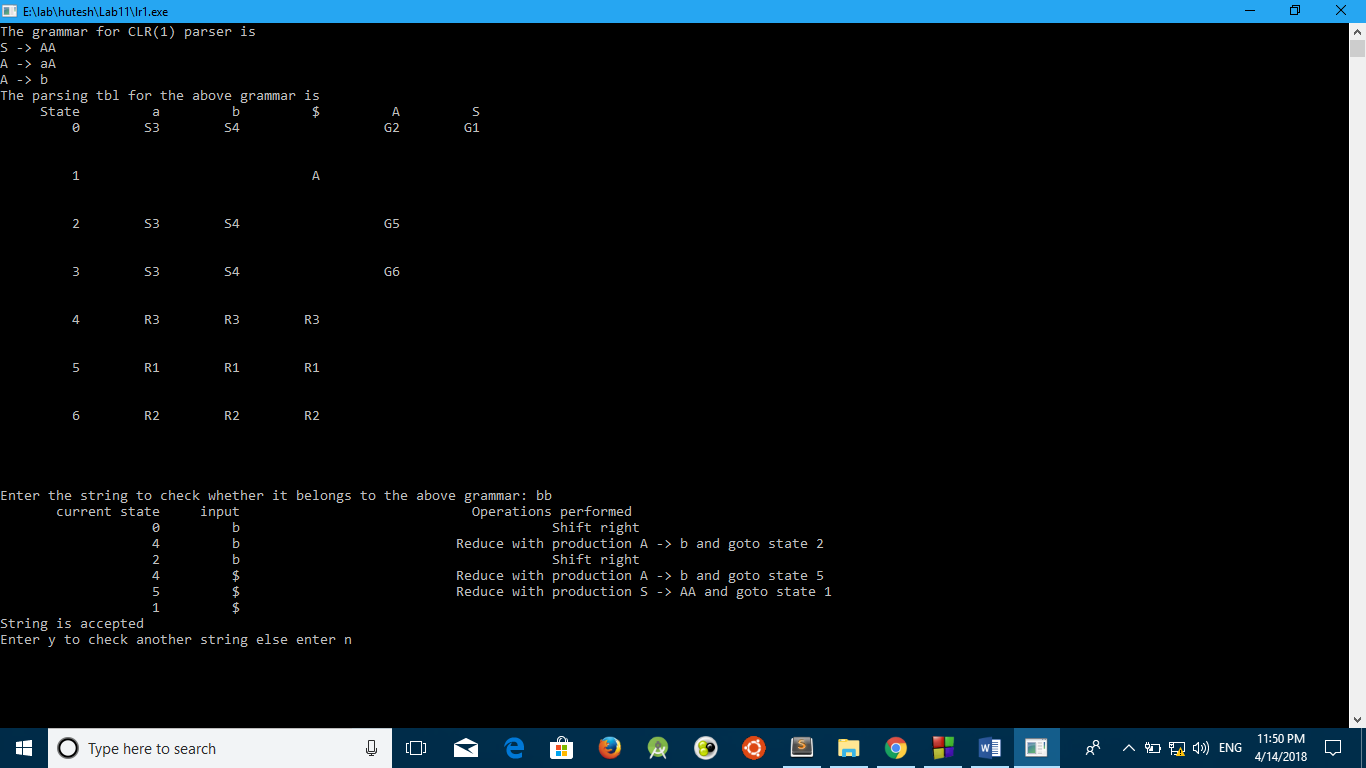
cin>>ch;

if(ch!='y')

break;

}

}



Implementation SYNTAX DIRECTED TRANSLATION ( SDT ):

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <ctype.h>

#include<string.h>

//#include<bits/stdc++.h>

//using namespace std;

// expression trees from pointer+struct+union+enum

const char \* expressionToParse;

char peek()

{

return \*expressionToParse;

}

char get()

{

return \*expressionToParse++;

}

int expression();

int number()

{

int result = get() - '0';

while (peek() >= '0' && peek() <= '9')

{

result = 10\*result + get() - '0';

}

return result;

}

int factor()

{

if (peek() >= '0' && peek() <= '9')

return number();

else if (peek() == '(')

{

get(); // '('

int result = expression();

get(); // ')'

return result;

}

else if (peek() == '-')

{

get();

return -factor();

}

return 0; // error

}

int term()

{

int result = factor();

while (peek() == '\*' || peek() == '/')

if (get() == '\*')

result \*= factor();

else

result /= factor();

return result;

}

int expression()

{

int result = term();

while (peek() == '+' || peek() == '-')

if (get() == '+')

result += term();

else

result -= term();

return result;

}

enum Etag {

constant, plus, minus, times

};

struct E {

enum Etag tag;

union {

int constant;

struct {

struct E \*e1;

struct E \*e2;

} plus;

struct {

struct E \*e1;

struct E \*e2;

} minus;

struct {

struct E \*e1;

struct E \*e2;

} times;

} Eunion;

};

// allocator

#define heapsize 100 // because C99 and gcc do not like the following:

// const int heapsize = 100; // fine in Xcode

struct E myheap[heapsize];

struct E \*freeptr = myheap;

struct E \*myalloc()

{

if(freeptr + 1 >= myheap + heapsize) {

fprintf(stderr, "Heap overflow.\n");

exit(1);

}

return freeptr++;

}

// expression tree construction

struct E \*makeconstant(int n)

{

struct E \*p;

p = myalloc();

p->tag = constant;

p->Eunion.constant = n;

return p;

}

struct E \*makeplus(struct E \*left,

struct E \*right)

{

struct E \*p;

p = myalloc();

p->tag = plus;

p->Eunion.plus.e1 = left;

p->Eunion.plus.e2 = right;

return p;

}

struct E \*makeminus(struct E \*left,

struct E \*right)

{

struct E \*p;

p = myalloc();

p->tag = minus;

p->Eunion.plus.e1 = left;

p->Eunion.plus.e2 = right;

return p;

}

struct E \*maketimes(struct E \*left,

struct E \*right)

{

struct E \*p;

p = myalloc();

p->tag = times;

p->Eunion.plus.e1 = left;

p->Eunion.plus.e2 = right;

return p;

}

// expression tree evaluation

int eval(struct E \*p)

{

switch(p->tag) {

case constant:

return p->Eunion.constant;

case plus:

return eval(p->Eunion.plus.e1)

+ eval(p->Eunion.plus.e2);

case minus:

return eval(p->Eunion.minus.e1)

- eval(p->Eunion.minus.e2);

case times:

return eval(p->Eunion.times.e1)

\* eval(p->Eunion.times.e2);

default:

fprintf(stderr, "Invalid tag for struct E.\n\n");

exit(1);

}

}

// scanning input

char input[100];

char \*pos = input;

char lookahead()

{

while(isspace(\*pos))

pos++;

return \*pos;

}

void syntaxerror()

{

int n = 0;

printf("Syntax error:\n");

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

while(n < pos - input) {

n++;

fprintf(stderr, " ");

}

fprintf(stderr, "^\n\n");

exit(1);

}

void match(char c)

{

if(lookahead() == c)

pos++;

else

syntaxerror();

}

void reset()

{

// re-use memory by resetting pointer to beginning of array

// no garbage

freeptr = myheap;

pos = input;

}

// forward declarations of parsing functions

struct E \*parseE();

struct E \*parseF();

struct E \*parseEprime(struct E\*);

struct E \*parseFprime(struct E\*);

// definitions of parsing functions

struct E \*parseP() // primary expression

{

char c;

struct E\* result = 0;

switch(c = lookahead()) {

case '0': case '1': case '2': case '3': case '4':

case '5': case '6': case '7': case '8': case '9':

match(c);

result = makeconstant(c - '0');

break;

case '(':

match('(');

result = parseE();

match(')');

break;

default:

syntaxerror();

}

return result;

}

struct E \*parseF() // factor

{

struct E\* resultOfP;

struct E\* result = NULL;

switch(lookahead()) {

case '0': case '1': case '2': case '3': case '4':

case '5': case '6': case '7': case '8': case '9':

case '(':

resultOfP = parseP();

result = parseFprime(resultOfP);

break;

default:

syntaxerror();

}

return result;

}

struct E \*parseFprime(struct E\* leftContext)

{

struct E \*resultOfP;

struct E \*result = NULL;

switch(lookahead()) {

case '\*':

match('\*');

resultOfP = parseP();

result = parseFprime(maketimes(leftContext, resultOfP));

break;

case ')': case '+': case '-': case '$':

result = leftContext;

break;

default:

syntaxerror();

}

return result;

}

struct E \*parseE()

{

struct E \*resultOfF;

struct E \*result = NULL;

switch(lookahead()) {

case '0': case '1': case '2': case '3': case '4':

case '5': case '6': case '7': case '8': case '9':

case '(':

resultOfF = parseF();

result = parseEprime(resultOfF);

break;

default:

syntaxerror();

}

return result;

}

struct E \*parseEprime(struct E\* leftContext)

{

struct E \*resultOfF;

struct E \*result = NULL;

switch(lookahead()) {

case '+':

match('+');

resultOfF = parseF();

result = parseEprime(makeplus(leftContext, resultOfF));

break;

case '-':

match('-');

resultOfF = parseF();

result = parseEprime(makeminus(leftContext, resultOfF));

break;

case ')': case '$':

result = leftContext;

break;

default:

syntaxerror();

}

return result;

}

int main(int argc, char \*argv[])

{

printf("\n\nExpression for evaluation:\n");

fgets(input, 80, stdin);

ToParse = input;

if (strlen(input) < 2) return 0;

input[strlen(input) - 1] = '$';

printf("Result = %d\n\n", expression());

reset();

}