**实验报告：LL (1)语法分析**

**一、实验目的**

通过设计、开发一个高级语言的LL（1）语法分析程序，实现对源程序的语法检查和结构分析，加深对相关课堂教学内容（包括自顶向下语法分析、First集、Follow集、Select集、文法等价变换）的理解，提高语法分析方法的实践能力。

**二、实验要求**

(1)理解语法分析在编译程序中的作用，以及它与词法分析程序的关系；

(2)掌握LL（1）语法分析方法的主要原理；

(3)理解LL（1）分析器模型；

(4)理解LL（1）语法分析方法对文法的要求；

(5)熟练掌握Select集合的求解方法和LL（1）分析表的构造方法；

(6)熟练掌握文法变换方法（消除左递归和提取公因子）。

**三、实验原理**

LL(1)分析法属于自顶向下分析方法，需解决的关键问题是在构建从文法开始符号到句子的推导序列时如何确定正确的产生式，即在LL(1)分析中，每当在符号栈的栈顶出现非终结符号时，要预测用哪个产生式的右部去替换该非终结符号。LL(1)分析方法对文法的要求与递归下降分析法一样。即对于文法中每个非终结符号A的全部产生式A?α 1 |α 2 |……|α n ，必须满足：Select(A?α i )∩Select(A?α j )=Φ,其中i≠j满足以上条件的文法称为LL（1）文法，在求解Select集的基础上可进一步构建LL（1）分析表，有了LL（1）分析表可调用LL（1）分析器模型实现语言的语法分析。LL(1)分析表的作用是对当前非终结符号和输入符号确定应该选择用哪个产生式进行推导。它的行对应文法的非终结符号，列对应终结符号，表中的值

有两种：一是产生式(或其编号)，一是错误处理动作。

LL(1)分析主要包括以下四个动作，其中X为符号栈栈顶元素，a为输入流当前字符:

a) 替换：当X∈V N 时选相应产生式的右部β去替换X。

b) 匹配：当X∈V T 时它与a进行匹配，其结果可能成功，也可能失败，如果成功则符号栈中将X退栈并将输入流指针向前移动一位，不成功则报错。

c) 成功：当格局为（空，空）时报告分析成功。

d) 报错：出错后，停止分析。

**S语言BNF：**

根据下列S语言的语法规则，进行语法分析

(1) <程序>→[<常量说明>][<变量说明>]<语句>

(2) <常量说明>→Const <常量定义>{，<常量定义>}；

(3) <常量定义>→<标识符>＝<无符号整数>

(4) <无符号整数>→<数字>{<数字>}

(5) <字母>→a|b|c| … |z

(6) <数字>→0|1|2| … |9

(7) <标识符>→<字母>{<字母>|<数字>}

(8) <变量说明>→Var <标识符>{，<标识符>}；

(9) <语句>→<赋值语句>|<条件语句>|<当循环语句>|<复合语句>|ε

(10) <赋值语句>→<标识符>＝<表达式>;

(11) <表达式>→[＋|－]<项>{<加法运算符><项>}

(12) <项>→<因子>{<乘法运算符><因子>}

(13) <因子>→<标识符>|<无符号整数>|‘(’<表达式>‘)’

(14) <加法运算符>→＋|－

(15) <乘法运算符>→\* |／

(16) <条件语句>→if <条件> then <语句>| if <条件> then <语句> else

<语句>

(17) <条件>→<表达式><关系运算符><表达式>

(18) <关系运算符>→＝＝|＜＝|＜|＞|＞＝|＜＞

(19) <当循环语句>→while <条件> do <语句>

(20) <复合语句>→begin <语句>{；<语句>} end

**四、实验步骤**

(1)根据S语言BNF形式的语法规则（见附件A），写出S语言的上下

文无关文法；

(2)求每个产生式的Select集：Select(A->)β)

= First(β)，当 ε∉First(β)

= (First(β)-{ε})∪Follow(A)，当ε∈First(β)

(3)判断文法是否为LL（1）的，若不是则用消除左递归和提取公因子等文法等价变换算法对文法进行变换，使其满足LL（1）文法的要求；

(4)根据Select集构建LL（1）分析表；

(5)调用LL（1）分析器模型，编程实现LL（1）分析程序；

(6)设计、实现能根据分析结果（产生式序列）构建源程序分析树的方法；

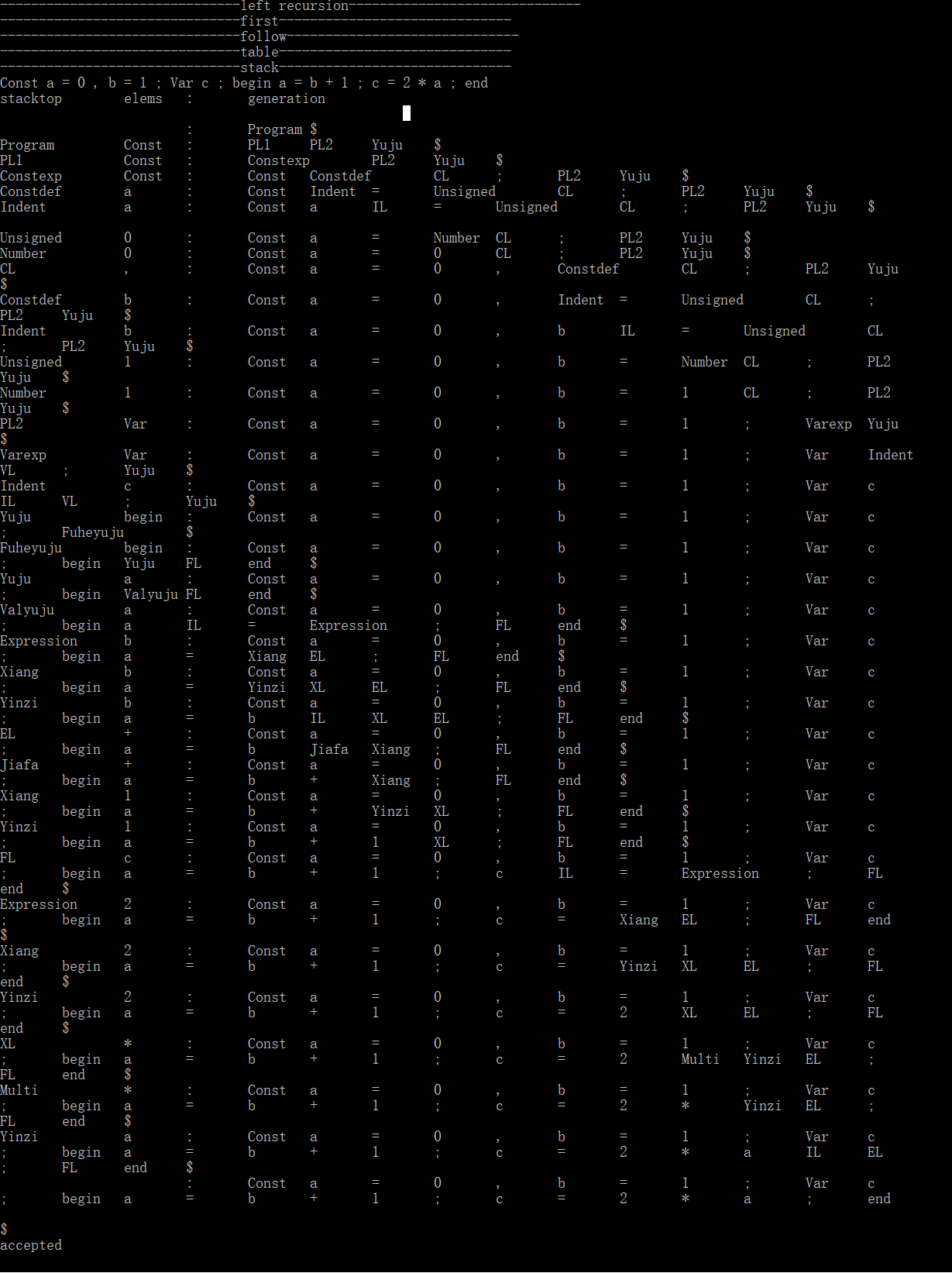
(7)撰写实验报告。

**五、实验结果**

1.输入样本：

Const a = 0 , b = 1 ; Var c ; begin a = b + 1 ; c = 2 \* a ; end

2.结果显示：



**六、源码**

#include<iostream>

#include<vector>

#include<map>

#include<set>

#include<string>

#include<cstdlib>

#include<fstream>

#include<sstream>

#include<stack>

#include<cstring>

using namespace std;

string parseTable[1000][1000];

string longest\_sub;

map<string,set<string> > first\_set;

map<string,set<string> > follow\_set;

vector<string> nonTerminals\_vector;

vector<string> Terminals\_vector;

map< string,vector<string> > grammar;

set<string> first\_expr(string);

vector<string> split(string s, char delim, vector<string> &elems);

bool is\_nonTerminal(string nonT);

bool has\_ep(string nonT);

void first();

void follow();

string getNonterminal(string line,int &linePos);

string getRHS(string line,int &linePos);

bool nonTerminal\_notExist(string nonTerminal);

bool check\_direct\_recursion(int i);

bool str\_match(int j,int i,int k);

void indirect\_recursion();

void direct\_recursion(int i);

void left();

bool check\_hash(string checkHash);

set<string> first\_expr(string s);

int lookTerminal\_vector(string s);

int parse();

int lookNonTerminal\_vector(string s);

void stack\_parsing();

int longest\_subsequence(string nonT);

void left\_fac();

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

{

cout << "-------------------------------left recursion------------------------------" << endl;

left();

vector<string>::iterator iter;

/\*

for(iter = nonTerminals\_vector.begin();iter != nonTerminals\_vector.end();iter++){

cout << \*iter << "-->" << longest\_subsequence(\*iter)<< endl;

}

\*/

/\*

cout << "-------------------------------left factor------------------------------" << endl;

left\_fac();

\*/

cout << "-------------------------------first------------------------------" << endl;

first();

cout << "-------------------------------follow------------------------------" << endl;

follow();

cout << "-------------------------------table------------------------------" << endl;

parse();

cout << "-------------------------------stack------------------------------" << endl;

while(1)

{

stack\_parsing();

}

return 0;

}

//stack\_parsing

void stack\_parsing()

{

string line;

getline(cin,line);

line+=" $";

vector<string> elems;

vector<string> st\_push;

vector<string> out\_elems;

split(line,' ',elems);

stack<string> stk;

stack<string> temp\_stk,temp1\_stk;

stk.push("$");

stk.push(nonTerminals\_vector[0]);

int i=0;

int l,m;

string temp;

bool flag=0;

l=lookNonTerminal\_vector("JL");

m=lookTerminal\_vector("else");

parseTable[l][m] = "else Yuju";

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

cout<<"elems"<<"\t"<<":"<<"\t";

cout<<"generation"<<endl<<endl;

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

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

cout<<stk.top()<<"\t"<<"$"<<endl;

while(stk.top()!="$")

{

if(stk.top()!=elems[i])

{

temp\_stk=stk;

while(!temp\_stk.empty())

{

temp\_stk.pop();

}

temp=stk.top();

stk.pop();

l=lookNonTerminal\_vector(temp);

m=lookTerminal\_vector(elems[i]);

if(l==-1 || m==-1)

{

flag=1;

break;

}

if(parseTable[l][m]=="#")

continue;

if(temp.length()>=8)

{

cout<<temp<<"\t";

cout<<elems[i]<<"\t"<<":"<<"\t";

}

else

{

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

cout<<elems[i]<<"\t"<<":"<<"\t";

}

split(parseTable[l][m],' ',st\_push);

int sl = st\_push.size();

for(int j=st\_push.size()-1;j>=0;j--)

{

stk.push(st\_push[j]);

}

//输出当前产生式

for(int j=0;j<out\_elems.size();j++)

cout<<out\_elems[j]<<"\t";

temp\_stk=stk;

while(!temp\_stk.empty())

{

temp = temp\_stk.top();

cout<<temp<<"\t";

temp\_stk.pop();

}

cout<<endl;

st\_push.erase(st\_push.begin(),st\_push.end());

}

else

{

temp\_stk=stk;

while(!temp\_stk.empty())

{

temp\_stk.pop();

}

temp=stk.top();

out\_elems.push\_back(temp);

stk.pop();

i++;

}

}

if(!flag)

{

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

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

for(int j=0;j<out\_elems.size();j++)

cout<<out\_elems[j]<<"\t";

cout<<endl;

temp\_stk=stk;

while(!temp\_stk.empty())

{

cout<<temp\_stk.top()<<"\t";

temp\_stk.pop();

}

cout<<endl;

if(elems[i]=="$")

cout << "accepted\n";

else

cout << "error\n";

}

else

cout << "error\n";

cout<<endl<<endl;

}

vector<string> split(string s, char delim, vector<string> &elems)

{

stringstream ss(s);

string item;

while(getline(ss, item, delim))

{

if(item != "")

elems.push\_back(item);

}

return elems;

}

//first

bool is\_nonTerminal(string nonT)

{

vector<string>::iterator iter;

for(iter = nonTerminals\_vector.begin();iter != nonTerminals\_vector.end();iter++)

{

if(\*iter == nonT)

return 1;

}

return 0;

}

bool has\_ep(string nonT)

{

vector<string>::iterator iter\_v;

for(iter\_v = grammar[nonT].begin();iter\_v != grammar[nonT].end();iter\_v++)

{

if(\*iter\_v == "#"){

return 1;

}

}

for(iter\_v = grammar[nonT].begin();iter\_v != grammar[nonT].end();iter\_v++)

{

int has;

vector<string> temp;

temp.clear();

vector<string>::iterator it\_temp;

split(\*iter\_v,' ',temp);

for(it\_temp = temp.begin();it\_temp != temp.end();it\_temp++)

{

has = 0;

if(is\_nonTerminal(\*it\_temp))

{

if(!has\_ep(\*it\_temp))

{

return 0;

}

else

has = 1;

}

else

break;

}

if(has)

return 1;

}

return 0;

}

void first()

{

vector<string>::iterator iter;

vector<string>::iterator iter\_v;

map<string,set<string> >::iterator it\_f;

int flag\_i = 1;

for(iter = nonTerminals\_vector.begin();iter != nonTerminals\_vector.end();iter++)

{

for(iter\_v = grammar[\*iter].begin();iter\_v != grammar[\*iter].end();iter\_v++)

{

vector<string> temp;

vector<string>::iterator it\_temp;

split(\*iter\_v,' ',temp);

for(it\_temp = temp.begin();it\_temp != temp.end();it\_temp++)

{

if(!is\_nonTerminal(\*it\_temp))

{

if(\*it\_temp != "")

{

if(first\_set[\*iter].find(\*it\_temp) == first\_set[\*iter].end()){

vector<string>::iterator it\_ter;

int exist = 0;

for(it\_ter = Terminals\_vector.begin();it\_ter != Terminals\_vector.end();it\_ter++)

{

if(\*it\_ter == \*it\_temp)

exist = 1;

}

if(!exist)

{

first\_set[\*it\_temp].insert(\*it\_temp);

Terminals\_vector.push\_back(\*it\_temp);

}

}

}

}

}

}

}

while(flag\_i)

{

flag\_i = 0;

for(iter = nonTerminals\_vector.begin();iter != nonTerminals\_vector.end();iter++)

{

if(has\_ep(\*iter))

{

if(first\_set[\*iter].find("#") == first\_set[\*iter].end())

flag\_i = 1;

first\_set[\*iter].insert("#");

}

for(iter\_v = grammar[\*iter].begin();iter\_v != grammar[\*iter].end();iter\_v++)

{

vector<string> temp;

vector<string>::iterator it\_temp;

split(\*iter\_v,' ',temp);

for(it\_temp = temp.begin();it\_temp != temp.end();it\_temp++)

{

if(is\_nonTerminal(\*it\_temp))

{

if(!has\_ep(\*it\_temp))

{

set<string>::iterator it\_s;

if(first\_set.find(\*it\_temp)!=first\_set.end())

{

for(it\_s = first\_set[\*it\_temp].begin();it\_s != first\_set[\*it\_temp].end();it\_s++)

{

if(first\_set[\*iter].find(\*it\_s) == first\_set[\*iter].end())

flag\_i = 1;

first\_set[\*iter].insert(\*it\_s);

}

}

break;

}

}

else

{

if(\*it\_temp != "")

{

if(first\_set[\*iter].find(\*it\_temp) == first\_set[\*iter].end())

flag\_i = 1;

first\_set[\*iter].insert(\*it\_temp);

break;

}

}

}

}

}

}

}

//follow

void follow()

{

follow\_set[nonTerminals\_vector[0]].insert("$");

vector<string>::iterator iter;

vector<string>::iterator iter\_v;

int flag\_i = 1;

while(flag\_i){

flag\_i = 0;

for(iter = nonTerminals\_vector.begin();iter != nonTerminals\_vector.end();iter++)

{

for(iter\_v = grammar[\*iter].begin();iter\_v != grammar[\*iter].end();iter\_v++)

{

vector<string> temp;

vector<string>::iterator it\_temp;

vector<string>::iterator it\_next;

split(\*iter\_v,' ',temp);

for(it\_temp = temp.begin(),it\_next=++temp.begin();it\_temp != temp.end();it\_temp++,it\_next++)

{

if(\*it\_temp != "")

{

if(is\_nonTerminal(\*it\_temp))

{

if(it\_next!=temp.end())

{

set<string>::iterator it\_s;

for(it\_s = first\_set[\*it\_next].begin();it\_s != first\_set[\*it\_next].end();it\_s++)

{

if(\*it\_s != "#")

{

if(follow\_set[\*it\_temp].find(\*it\_s) == follow\_set[\*it\_temp].end())

flag\_i = 1;

follow\_set[\*it\_temp].insert(\*it\_s);

}

}

set<string> first\_e;

vector<string>::iterator it\_v;

string tmp = "";

for(it\_v = it\_next;it\_v!=temp.end();it\_v++)

{

tmp += \*it\_v + " ";

}

first\_e = first\_expr(tmp);

if(first\_e.find("#") != first\_e.end())

{

set<string>::iterator it\_s;

if(follow\_set.find(\*iter)!=follow\_set.end())

{

for(it\_s=follow\_set[\*iter].begin();it\_s!=follow\_set[\*iter].end();it\_s++)

{

if(follow\_set[\*it\_temp].find(\*it\_s) == follow\_set[\*it\_temp].end())

flag\_i = 1;

follow\_set[\*it\_temp].insert(\*it\_s);

}

}

}

}

else{

set<string>::iterator it\_s;

if(follow\_set.find(\*iter)!=follow\_set.end())

{

for(it\_s=follow\_set[\*iter].begin();it\_s!=follow\_set[\*iter].end();it\_s++)

{

if(follow\_set[\*it\_temp].find(\*it\_s) == follow\_set[\*it\_temp].end())

flag\_i = 1;

follow\_set[\*it\_temp].insert(\*it\_s);

}

}

}

}

}

}

}

}

}

}

//left\_recursion

string getNonterminal(string line,int &linePos)

{

int i=0;

string str="";

while(line[i]!='-')

{

str+=line[i];

i++;

}

linePos=i+2;

return str;

}

string getRHS(string line,int &linePos)

{

int i=linePos;

int len=line.length();

string str="";

while(i!=len && line[i]!='|')

{

str+=line[i];

i++;

}

if(i!=len)

linePos=i+1;

else

linePos=i;

return str;

}

bool nonTerminal\_notExist(string nonTerminal)

{

int vec\_size=nonTerminals\_vector.size();

for(int i=0;i<vec\_size;i++)

{

if(nonTerminals\_vector[i]==nonTerminal)

return 0;

}

return 1;

}

bool str\_match(int j,int i,int k)

{

string str2=grammar[nonTerminals\_vector[i]][k];

if(str2.compare(0,nonTerminals\_vector[j].length(),nonTerminals\_vector[j]) == 0)

{

return 1;

}

return 0;

}

bool check\_direct\_recursion(int i)

{

int vec\_size=grammar[nonTerminals\_vector[i]].size();

for(int j=0;j<vec\_size;j++)

{

if(str\_match(i,i,j))

{

return 1;

}

}

return 0;

}

void direct\_recursion(int i)

{

int vec\_size=grammar[nonTerminals\_vector[i]].size();

if(check\_direct\_recursion(i))

{

string newNonTerminal=nonTerminals\_vector[i]+"\_dash";

nonTerminals\_vector.push\_back(newNonTerminal);

vector<string> temp\_vec;

for(int j=0;j<vec\_size;j++)

{

string originalRHS=grammar[nonTerminals\_vector[i]][j];

if(str\_match(i,i,j))

{

string changedRHS=originalRHS.substr(nonTerminals\_vector[i].length(),originalRHS.length()); //alpha1

grammar[newNonTerminal].push\_back(changedRHS+" "+newNonTerminal);

}

else

{

temp\_vec.push\_back(originalRHS+" "+newNonTerminal);

}

}

grammar[nonTerminals\_vector[i]].erase(grammar[nonTerminals\_vector[i]].begin(),grammar[nonTerminals\_vector[i]].end());

grammar[nonTerminals\_vector[i]].insert(grammar[nonTerminals\_vector[i]].begin(),temp\_vec.begin(),temp\_vec.end());

temp\_vec.erase(temp\_vec.begin(),temp\_vec.end());

grammar[newNonTerminal].push\_back("#");

}

}

void indirect\_recursion()

{

int i,j,k,l;

int vec\_size=nonTerminals\_vector.size();

for(i=0;i<vec\_size;i++)

{

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

{

int tmp=grammar[nonTerminals\_vector[i]].size();

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

{

if(str\_match(j,i,k))

{

string originalRHS=grammar[nonTerminals\_vector[i]][k];

string changedRHS;

changedRHS=originalRHS.substr(nonTerminals\_vector[j].length(),originalRHS.length());

grammar[nonTerminals\_vector[i]].erase(grammar[nonTerminals\_vector[i]].begin()+k);

int tmp1=grammar[nonTerminals\_vector[j]].size();

for(l=0;l<tmp1;l++)

{

grammar[nonTerminals\_vector[i]].push\_back(grammar[nonTerminals\_vector[j]][l]+changedRHS);

}

}

}

}

direct\_recursion(i);

}

}

void left()

{

string line;

ifstream fp;

fp.open("in");

int linePos;

while(getline (fp,line))

{

linePos=0;

string nonTerminal=getNonterminal(line,linePos);

if(nonTerminal\_notExist(nonTerminal))

{

nonTerminals\_vector.push\_back(nonTerminal);

}

string RHS;

int len=line.length();

while(linePos!=len)

{

RHS=getRHS(line,linePos);

grammar[nonTerminal].push\_back(RHS);

}

}

indirect\_recursion();

fp.close();

}

//parsing

int Terminal\_count;

int nonTerminal\_count;

bool check\_hash(string checkHash)

{

set<string>::iterator it;

it=first\_set[checkHash].find("#");

if(it!=first\_set[checkHash].end())

return 1;

return 0;

}

set<string> first\_expr(string s)

{

vector<string> elems;

split(s,' ',elems);

set<string> temp;

int i;

for(i=0;i<elems.size() && check\_hash(elems[i]);i++)

{

temp.insert(first\_set[elems[i]].begin(),first\_set[elems[i]].end());

}

if(i!=elems.size())

{

temp.insert(first\_set[elems[i]].begin(),first\_set[elems[i]].end());

}

return temp;

}

int lookTerminal\_vector(string s)

{

for(int i=0;i<Terminals\_vector.size();i++)

{

if(Terminals\_vector[i]==s)

return i;

}

return -1;

}

int parse()

{

Terminals\_vector.push\_back("$");

Terminal\_count=Terminals\_vector.size();

nonTerminal\_count=nonTerminals\_vector.size();

set<string> temp;

set<string> temp\_follow;

set<string>::iterator it;

set<string>::iterator it\_follow;

int l;

for(int i=0;i<nonTerminals\_vector.size();i++)

{

int len=grammar[nonTerminals\_vector[i]].size();

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

{

temp=first\_expr(grammar[nonTerminals\_vector[i]][j]);

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

{

if(\*it!="#")

{

l=lookTerminal\_vector(\*it);

parseTable[i][l]=grammar[nonTerminals\_vector[i]][j];

}

else

{

temp\_follow.insert(follow\_set[nonTerminals\_vector[i]].begin(),follow\_set[nonTerminals\_vector[i]].end());

for(it\_follow=temp\_follow.begin();it\_follow!=temp\_follow.end();it\_follow++)

{

l=lookTerminal\_vector(\*it\_follow);

parseTable[i][l]=grammar[nonTerminals\_vector[i]][j];

}

temp\_follow.erase(temp\_follow.begin(),temp\_follow.end());

}

}

}

}

}

int lookNonTerminal\_vector(string s)

{

for(int i=0;i<nonTerminals\_vector.size();i++)

{

if(nonTerminals\_vector[i]==s)

return i;

}

return -1;

}

//left factoring

int longest\_subsequence(string nonT)

{

int l\_size = grammar[nonT].size();

int old\_len = 0,new\_len = 0;

longest\_sub = "";

string seq = "";

for(int i=0;i<l\_size;i++){

for(int j=i+1;j<l\_size;j++){

new\_len = 0;

seq = "";

for(int k=0;k<grammar[nonT][i].size();k++)

{

if(grammar[nonT][i][k] == grammar[nonT][j][k])

{

seq += grammar[nonT][i][k];

new\_len++;

}

else

break;

}

if(old\_len < new\_len){

old\_len = new\_len;

longest\_sub = seq;

}

}

}

return old\_len;

}

/\*

void left\_fac()

{

int long\_sub;

for(int i=0;i<nonTerminals\_vector.size();i++)

{

int count = 0;

while(1)

{

longest\_sub = "";

long\_sub = longest\_subsequence(nonTerminals\_vector[i]);

string seq = longest\_sub;

longest\_sub = "";

if(long\_sub > 0)

{

string newNT = nonTerminals\_vector[i]+"\_fac";

for(int j=0;j<count;j++){

newNT += "\_fac";

}

vector<string> n\_push;

n\_push.clear();

int flag = 0;

for(int j=0;j<grammar[nonTerminals\_vector[i]].size();j++)

{

string str = grammar[nonTerminals\_vector[i]][j];

if(str.compare(0,seq.length(),seq)==0)

{

string old\_seq = grammar[nonTerminals\_vector[i]][j];

if(flag == 0)

{

grammar[nonTerminals\_vector[i]][j] = seq + newNT + " ";

flag = 1;

}

else

{

grammar[nonTerminals\_vector[i]].erase(grammar[nonTerminals\_vector[i]].begin()+j);

j--;

}

string change = old\_seq.substr(seq.length(),old\_seq.length());

if(change==""){

change = "#";

}

n\_push.push\_back(change);

}

}

nonTerminals\_vector.push\_back(newNT);

grammar[newNT] = n\_push;

count++;

}

else

break;

}

}

}

\*/