**广州大学学生实验报告**

**开课学院及实验室：**计算机科学与网络工程学院计算机科学与工程实验室 **2020年5月18日**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **学院** | **计算机科学与网络工程学院** | **年级/专业/班** | **计科174** | **姓名** | 李诗云 | **学号** | 1706100092 |
| **实验课程名称** | 编译原理实验 | | | | | **成绩** |  |
| **实验项目名称** | 实验二 语法分析(算术表达式的扩充) | | | | | **指导老师** | 吴昱 |

**实验二 语法分析(算术表达式的扩充)**

1. **实验目的**

掌握LR分析表的设计方法和语义加工程序的扩充

1. **实验要求**

参照算术表达式LR分析表的设计方法，设计扩充后的算术表达式LR分析表，并对原语义加工程序修改，加入新添的内容。

1. **实验内容及实验步骤**

自定义文法：

S->Eac | dEa | cBa

E->k

B->l

对上述文法使用LR(1)分析法进行语法分析，并检测

实验步骤：

（1）将原文法改写成拓广文法

S’->S

S->Eac

S->dEa

S->cBa

E->k

B->l

（2）求所有非终结符的first集和follow集

非终结符 FIRST集合 FOLLOW集合

B l a

E k a

S cdk #

（3）写出文法所有的项目

S->.Eac S->E.ac S->Ea.c S->Eac.

S->.dEa S->d.Ea S->dE.a S->dEa.

S->.cBa S->c.Ba S->cB.a S->cBa.

E->.k E->k.

B->.l B->l.

（4）根据closure和goto规则写出项目集规范族，并结合项目的活前缀构造DFA

项目集I0：

I0 -- E->I1

I0 -- S->I2

I0 -- c->I3

I0 -- d->I4

I0 -- k->I5

项目集I1：

I1 -- a->I6

项目集I3：

I3 -- B->I7

I3 -- l->I8

项目集I4：

I4 -- E->I9

I4 -- k->I5

项目集I6：

I6 -- c->I10

项目集I7：

I7 -- a->I11

项目集I9：

I9 -- a->I12

（5）构造LR分析表

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
| 项目集 | a | c | d | k | l | # | B | E | S |
| I0 |  | S3 | S4 | S5 |  |  |  | 1 | 2 |
| I1 | S6 |  |  |  |  |  |  |  |  |
| I2 |  |  |  |  |  | acc |  |  |  |
| I3 |  |  |  |  | S8 |  | 7 |  |  |
| I4 |  |  |  | S5 |  |  |  | 9 |  |
| I5 | R3 |  |  |  |  |  |  |  |  |
| I6 |  | S10 |  |  |  |  |  |  |  |
| I7 | S11 |  |  |  |  |  |  |  |  |
| I8 | R4 |  |  |  |  |  |  |  |  |
| I9 | S12 |  |  |  |  |  |  |  |  |
| I10 |  |  |  |  |  | R0 |  |  |  |
| I11 |  |  |  |  |  | R2 |  |  |  |
| I12 |  |  |  |  |  | R1 |  |  |  |

1. **实验代码及结果**

gramma.txt （第一行是所有的终结符）

acdkl

S->Eac

S->dEa

S->cBa

E->k

B->l

LR1.cpp

#include <fstream>

#include <string>

#include <vector>

#include <algorithm>

#include <set>

#include <stack>

#include <iomanip>

#include <sstream>

#include <string>

#include <cstring>

#include <map>

#define MAX\_Count 100 //产生式的最大数量

#include <iostream>

using namespace std;

template <typename CSS\_LR1> // 重载+号运算符

vector<CSS\_LR1> &operator+(vector<CSS\_LR1> &v1, vector<CSS\_LR1> &v2)

{

v1.insert(v1.end(), v2.begin(), v2.end());

return v1;

}

//非终结符及其对应的First集和follow集

struct vn\_Set

{

string vn\_name;

set<string> FIRST;

set<string> FOLLOW;

};

//产生式的结点结构

struct productnode

{

string start; //非终结符

vector<string> next;

};

struct CSS\_LR1

{

string start; //非终结符

vector<string> next; //First集合

int num;

vector<string> tail;

bool operator==(const CSS\_LR1 &rhs) //重载

{

return (start == rhs.start &&

next == rhs.next &&

num == rhs.num &&

tail == rhs.tail);

}

};

int CSSCount = 0;

productnode css[MAX\_Count]; //产生式

vn\_Set vn\_First[MAX\_Count]; //非终结符集的First集合

set<string> vn; //存放非终结符

set<string> vt; //存放终结符

int I\_count = 0; //记录LR1项目数

vector<CSS\_LR1> I[MAX\_Count]; //项目集

map<string, int> mark\_Follow; //用于标记Follow 避免嵌套

map<string, int> GOTO[MAX\_Count]; //goto表

map<string, string> ACTION[MAX\_Count]; //action表

//判断新生成的项目集是否已经存在

bool cmp\_vector(vector<CSS\_LR1> &v1, vector<CSS\_LR1> &v2)

{

if (v1.size() != v2.size())

return false;

for (int i = 0; i < v2.size(); i++)

{

CSS\_LR1 t;

t = v2[i];

vector<CSS\_LR1>::iterator result = find(v1.begin(), v1.end(), t);

if (result == v1.end())

return false;

}

return true;

}

//求a的First集合

set<string> get\_FIRST(string a)

{

set<string> T;

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

{

if (css[i].start == a)

{ // a->..

for (int j = 0; j < css[i].next.size(); j++)

{

if (vt.find(css[i].next[j]) != vt.end())

{ //是终结符开头

T.insert(css[i].next[j]);

break;

}

else

{

if (css[i].next[j] == css[i].start)

{

break;

}

set<string> U = get\_FIRST(css[i].next[j]);

T.insert(U.begin(), U.end());

if (U.find("$") != U.end())

{ //U中含有\*,继续查下个的first

if (j != css[i].next.size() - 1)

T.erase("$");

}

else

{

break;

}

}

}

}

}

return T;

}

//求a的follow集

set<string> get\_FOLLOW(string a)

{

set<string> T;

mark\_Follow[a]++;

if (mark\_Follow[a] >= 2)

{

return T;

}

set<string> temp;

if (a == css[0].start)

{

T.insert("#");

}

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

{

for (int j = 0; j < css[i].next.size(); j++)

{

//是非终结符，求FOLLOW集合

if (vt.find(css[i].next[j]) == vt.end() && a == css[i].next[j])

{

//S->...a

if (j == css[i].next.size() - 1 && a != css[i].start)

{

set<string> tt = get\_FOLLOW(css[i].start);

T.insert(tt.begin(), tt.end());

}

for (int k = j + 1; k < css[i].next.size(); k++)

{

//后面一个是终结符 S->..av..

if (vt.find(css[i].next[k]) != vt.end())

{

T.insert(css[i].next[k]);

break;

}

else

{

temp = get\_FIRST(css[i].next[k]);

//有$ S->..a B..

if (temp.find("$") != temp.end())

{

T.insert(temp.begin(), temp.end());

T.erase("$");

//S->..a B

if (k == css[i].next.size() - 1)

{

set<string> tt = get\_FOLLOW(css[i].start);

T.insert(tt.begin(), tt.end());

break;

}

}

else

{

T.insert(temp.begin(), temp.end());

break;

}

}

}

}

}

}

mark\_Follow[a] = 0;

return T;

}

//计算所有非终结符的first集和follow集，并打印输出

void PrintFirst\_and\_Follow()

{

set<string>::iterator it;

int count = 0;

cout << "=========================================" << endl;

cout << '\t' << "FIRST集合"

<< " "

<< "FOLLOW集合" << endl;

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

{

vn\_First[count].vn\_name = \*it;

vn\_First[count].FIRST = get\_FIRST(\*it);

mark\_Follow[\*it] = 0;

vn\_First[count].FOLLOW = get\_FOLLOW(\*it);

//输出FIRST

cout << vn\_First[count].vn\_name << '\t';

set<string>::iterator it;

for (it = vn\_First[count].FIRST.begin(); it != vn\_First[count].FIRST.end(); it++)

{

cout << \*it << "";

}

cout << '\t' << " ";

//输出FOLLOW

set<string>::iterator it1;

for (it1 = vn\_First[count].FOLLOW.begin(); it1 != vn\_First[count].FOLLOW.end(); it1++)

{

cout << \*it1 << " ";

}

cout << endl;

count++;

}

cout << "=========================================";

cout << endl<< endl;

}

void input()

{

ifstream inFile;

//打开文件

inFile.open("gramma.txt");

if (inFile)

{

cout << "FILE open successful" << endl;

}

else

cout << "file doesn't exist" << endl;

//读取文件

string temp;

getline(inFile, temp);

for (int j = 0; j < temp.length(); j++)

{

vt.insert(temp.substr(j, 1));

}

set<string>::iterator p;

cout << "终结符号:";

for (p = vt.begin(); p != vt.end(); p++)

{

cout << \*p << ",";

}

cout << endl;

int count = 0; //文件行数

while (getline(inFile, temp)) //按行读取文件内容

{

css[count].start = temp[0];

for (int j = 3; j < temp.length(); j++)

{

css[count].next.push\_back(temp.substr(j, 1));

}

vn.insert(css[count].start); //非终结符

cout << css[count].start << "->";

vector<string>::iterator it;

for (it = css[count].next.begin(); it != css[count].next.end(); it++)

{

cout << \*it;

}

cout << endl;

count++;

}

CSSCount = count;

}

//

bool find\_in\_vector(vector<CSS\_LR1> T, CSS\_LR1 p)

{

vector<CSS\_LR1>::iterator it;

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

{

if (\*it == p)

{

return true;

}

}

return false;

}

//求项目的闭包

vector<CSS\_LR1> CLOSURE(CSS\_LR1 I)

{

vector<CSS\_LR1> T;

//规约项目A->α.或者接受项目

if (I.num >= I.next.size())

{

return T;

}

else

{

string temp = I.next[I.num];

//点后面的是终结符 ,移进项目 A→α.aβ

if (vt.find(temp) != vt.end())

{

return T;

}

else //待约项目

{

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

{

if (css[i].start == temp)

{

CSS\_LR1 p;

p.start = css[i].start;

p.num = 0; //点在最前面

p.next = css[i].next;

set<string> f1;

for (int j = I.num + 1; j < I.next.size(); j++)

{

set<string> f2; //用于暂存first

if (vt.find(I.next[j]) != vt.end())

{

f2.insert(I.next[j]);

}

else

{

f2 = get\_FIRST(I.next[j]);

}

f1.insert(f2.begin(), f2.end());

if (f2.find("$") == f2.end())

{

break;

}

}

if (f1.size() == 0)

{

p.tail = I.tail;

}

else

{

vector<string> first\_tail;

if (f1.find("$") != f1.end())

{

f1.erase("$");

copy(f1.begin(), f1.end(), back\_inserter(first\_tail));

first\_tail.insert(first\_tail.end(), I.tail.begin(), I.tail.end());

}

else

{

copy(f1.begin(), f1.end(), back\_inserter(first\_tail));

}

p.tail = first\_tail;

}

if (!find\_in\_vector(T, p))

{

T.push\_back(p);

vector<CSS\_LR1> ol = CLOSURE(p);

vector<CSS\_LR1>::iterator z;

for (z = ol.begin(); z != ol.end(); z++)

{

if (find\_in\_vector(T, \*z)){}

else

{

T.push\_back(\*z);

}

}

}

}

}

}

}

return T;

}

//打印项目集

void showI(vector<CSS\_LR1> I)

{

vector<CSS\_LR1>::iterator it;

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

{

CSS\_LR1 p = \*it;

cout << p.start << "->";

vector<string>::iterator s;

for (int j = 0; j < p.next.size(); j++)

{

if (j == p.num)

cout << ".";

cout << p.next[j];

}

if (p.num == p.next.size())

cout << ".";

cout << ",";

for (int k = 0; k < p.tail.size(); k++)

{

cout << p.tail[k];

}

cout << endl;

}

}

//LR1分析

void LR1\_Analyse()

{

CSS\_LR1 p;

//初始项目 S’->.S ,#

p.start = css[0].start + "^";

p.num = 0; //点在最前面

p.tail.push\_back("#");

p.next.push\_back(css[0].start);

I[0] = CLOSURE(p); //求闭包后的I[0]

I[0].insert(I[0].begin(), p);

I\_count = 1;

//计算项目集

for (int i = 0; i < I\_count; i++)

{

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

cout << "项目集I" << i << endl;

//---------求ACTION的r部分--------------

vector<CSS\_LR1>::iterator t;

for (t = I[i].begin(); t != I[i].end(); t++)

{

CSS\_LR1 t2 = \*t;

if (t2.num == t2.next.size())

{

int num = 0;

for (int xp = 0; xp < CSSCount; xp++)

{

if (css[xp].start == t2.start && css[xp].next == t2.next)

{

num = xp;

break;

}

}

std::stringstream ss;

ss << num;

string s = ss.str();

for (int q = 0; q < t2.tail.size(); q++)

{

ACTION[i][t2.tail[q]] = "r" + s;

}

if (t2.num == 1 && t2.next[0] == css[0].start)

{

ACTION[i]["#"] = "acc";

}

}

}

set<string>::iterator it;

//每个非终结符

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

{

vector<CSS\_LR1> temp;

for (int j = 0; j < I[i].size(); j++)

{

CSS\_LR1 lr = I[i][j];

if (lr.num < lr.next.size() && lr.next[lr.num] == \*it)

{

vector<CSS\_LR1> t2;

lr.num++;

t2 = CLOSURE(lr);

t2.push\_back(lr);

temp = temp + t2;

}

}

if (temp.size() > 0)

{

int k;

for (k = 0; k < I\_count; k++)

{

if (cmp\_vector(I[k], temp))

{

break;

}

}

if (k == I\_count)

{

//产生了新的项目集

I[I\_count] = temp;

cout << " I" << i << " -- " << \*it << "->"

<< "I" << I\_count << endl

<< endl;

GOTO[i][\*it] = I\_count; //更新goto表

I\_count++;

}

else

{

//项目集已经存在，需要自己指向自己

cout << " I" << i << " -- " << \*it << "->"

<< "I" << k << endl

<< endl;

GOTO[i][\*it] = k;

}

}

}

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

{ //每个终结符

vector<CSS\_LR1> temp;

for (int j = 0; j < I[i].size(); j++)

{

CSS\_LR1 lr = I[i][j];

if (lr.num < lr.next.size() && lr.next[lr.num] == \*it)

{

vector<CSS\_LR1> t2;

lr.num++;

t2 = CLOSURE(lr); //闭包求出的结果不包含本身

t2.insert(t2.begin(), lr);

temp = temp + t2;

}

}

if (temp.size() > 0)

{

int k;

for (k = 0; k < I\_count; k++)

{

//判断项目集是否已经存在

if (cmp\_vector(I[k], temp))

{

break;

}

}

if (k == I\_count)

{

//产生了新的项目集

I[I\_count] = temp;

cout << " I" << i << " -- " << \*it << "->"

<< "I" << I\_count << endl

<< endl;

std::stringstream ss;

ss << I\_count;

string s = ss.str();

ACTION[i][\*it] = "S" + s; //更新AVTION表

I\_count++;

}

else

{

//项目集已经存在，需要自己指向自己

cout << " I" << i << " -- " << \*it << "->"

<< "I" << k << endl

<< endl;

std::stringstream ss;

ss << k;

string s = ss.str();

ACTION[i][\*it] = "S" + s;

}

}

}

}

}

void print\_line()

{

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

}

void print\_ACTION\_GOTO()

{

set<string>::iterator it;

print\_line();

cout << setw(27) << setiosflags(ios::right) << "ACTION";

cout << setw(20) << setiosflags(ios::left) << " GOTO" << endl;

print\_line();

cout << setw(8) << "项目集";

//打印action表头

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

{

cout << setw(8) << \*it;

}

cout << setw(8) << "#";

//打印goto表头

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

{

cout << setw(8) << \*it ;

}

cout << endl;

for (int j = 0; j < I\_count; j++)

{

cout << setw(5) << "I"<< j ;

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

{

cout << setw(8) << ACTION[j][\*it];

}

cout << setw(8) << ACTION[j]["#"];

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

{

if (GOTO[j][\*it]) //GOTO表为0

cout << setw(8) << GOTO[j][\*it];

else

{

cout << setw(8) << " ";

}

}

cout << endl;

}

print\_line();

}

//对栈容器进行输出,i=0,返回status中的字符串,i=1,返回sign中的字符串，i=2返回inputStr中的字符串

string vectTrancStr(int i, vector<int> status, vector<string> sign)

{

string buf;

int count = 0;

//输出状态栈

if (i == 0)

{

vector<int>::iterator it = status.begin();

//将数字转化为字符串

string str, tempStr;

for (it; it != status.end(); it++)

{

stringstream ss;

ss << \*it;

ss >> tempStr;

str += tempStr;

}

return str;

}

//输出符号栈

else if (i == 1)

{

vector<string>::iterator it = sign.begin();

for (it; it != sign.end(); it++)

{

buf += \*it;

count++;

}

}

string str(buf);

return str;

}

//输入具体文法进行分析

void Input\_Analyse()

{

vector<int> status; //定义状态栈

vector<string> sign; //定义符号栈

int step = 1; //步骤

string input;

cout << "请输入分析的字符串(无需以#结尾)：";

cin >> input; //输入待分析的句子

input = input + "#";

status.push\_back(0); //状态0入栈

sign.push\_back("#");

//输出初始栈状态

cout << setw(10) << "步骤" << setw(10) << "状态栈" << setw(10) << "符号栈" << setw(10) << "输入串" << setw(25) << "动作说明" << endl;

int s = 0; //初始状态

int oldStatus; //保存之前的状态

string input\_s; //获取初始符号

input\_s = input.substr(0, 1);

while (ACTION[s][input\_s] != "acc")

{

string str = ACTION[s][input\_s];

//如果str为空，报错并返回

if (str.size() == 0)

{

cout << "出错";

return;

}

//获取S或r后面的数字

stringstream ss;

ss << str.substr(1);

ss >> s; //新的状态号

//如果是移进

if (str.substr(0, 1) == "S")

{

cout << setw(10) << step << setw(10) << vectTrancStr(0, status, sign) << setw(10) << vectTrancStr(1, status, sign) << setw(10) << input << setw(10) << "A"

<< "CTION[" << status.back() << "," << input\_s << "]=S" << s << ","

<< "状态" << s << "入栈" << endl;

sign.push\_back(input\_s); //输入符号入栈

input.erase(0, 1);

status.push\_back(s); //将状态数字入栈

}

//如果是规约

else if (str.substr(0, 1) == "r")

{

string kaitou; //产生式的头部

kaitou = css[s].start;

int pop\_num = css[s].next.size(); //获取符号栈的出栈次数

string r;

stringstream ss;

ss << s;

ss >> r;

int oldStatus; //保存之前的状态

int status\_size = status.size();

oldStatus = status[status\_size - 1 - pop\_num];

s = GOTO[oldStatus][kaitou];

cout << setw(10) << step << setw(10) << vectTrancStr(0, status, sign) << setw(10) << vectTrancStr(1, status, sign) << setw(10) << input << setw(10) << (string) ":产生式" + r + (string) "归约,GOTO(" << oldStatus << "," << kaitou << ")=" << s << "入栈" << endl;

//对符号栈进行出栈和状态栈进行出栈

while (pop\_num--)

{

sign.pop\_back();

status.pop\_back();

}

sign.push\_back(kaitou); //再对产生式的开始符号入栈

status.push\_back(s); //再把新的状态入栈

}

step++; //步骤数加1

s = status.back(); //获取栈顶状态

input\_s = input.substr(0, 1); //获取输入的字符

}

cout << setw(10) << step << setw(10) << vectTrancStr(0, status, sign) << setw(10) << vectTrancStr(1, status, sign) << setw(10) << input << setw(10) << "A"

<< "cc:分析成功" << endl;

}

int main()

{

input(); //从文件中读取文法

PrintFirst\_and\_Follow(); //求First和Follow集合

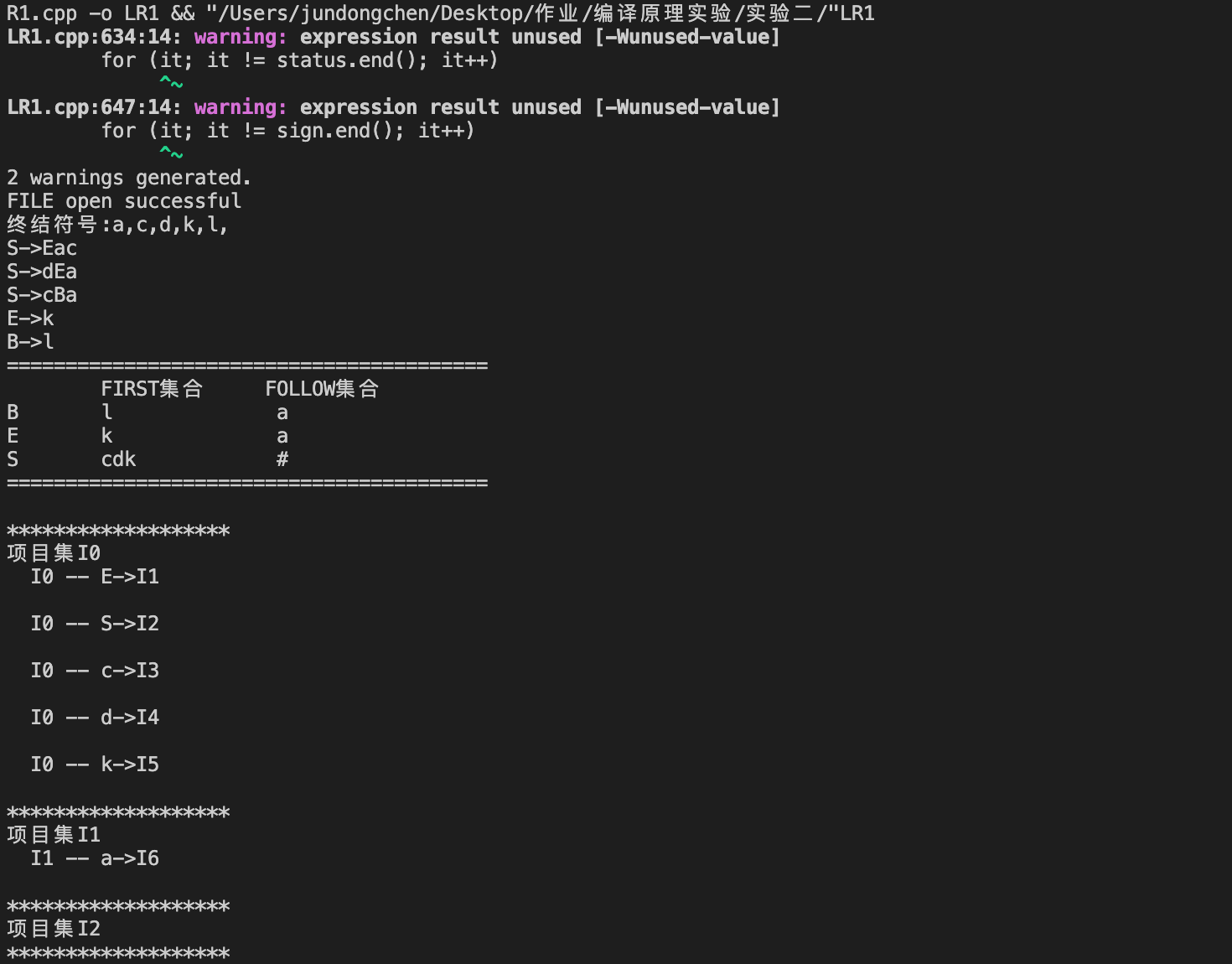
LR1\_Analyse();

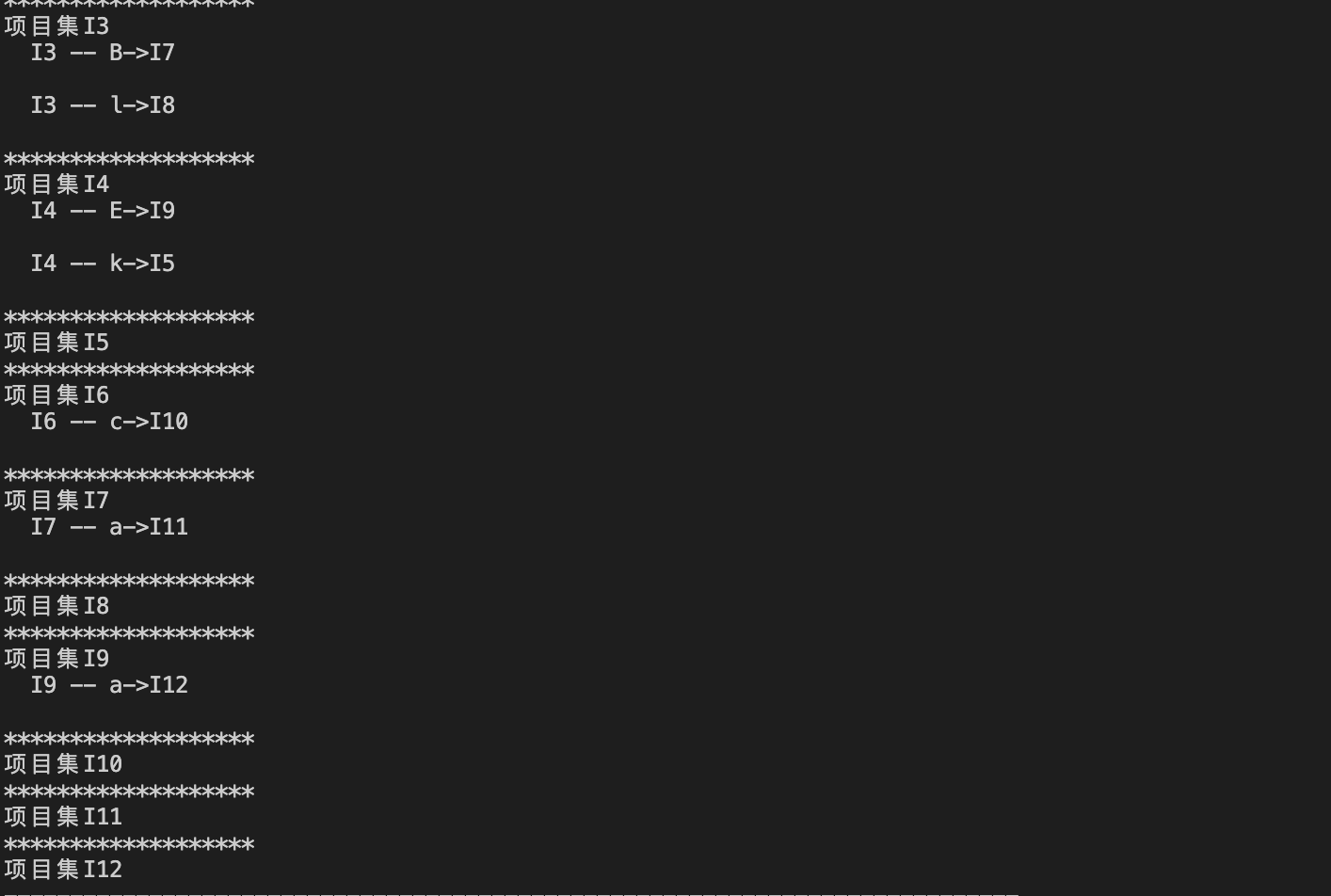
print\_ACTION\_GOTO(); //打印ACTION GOTO表

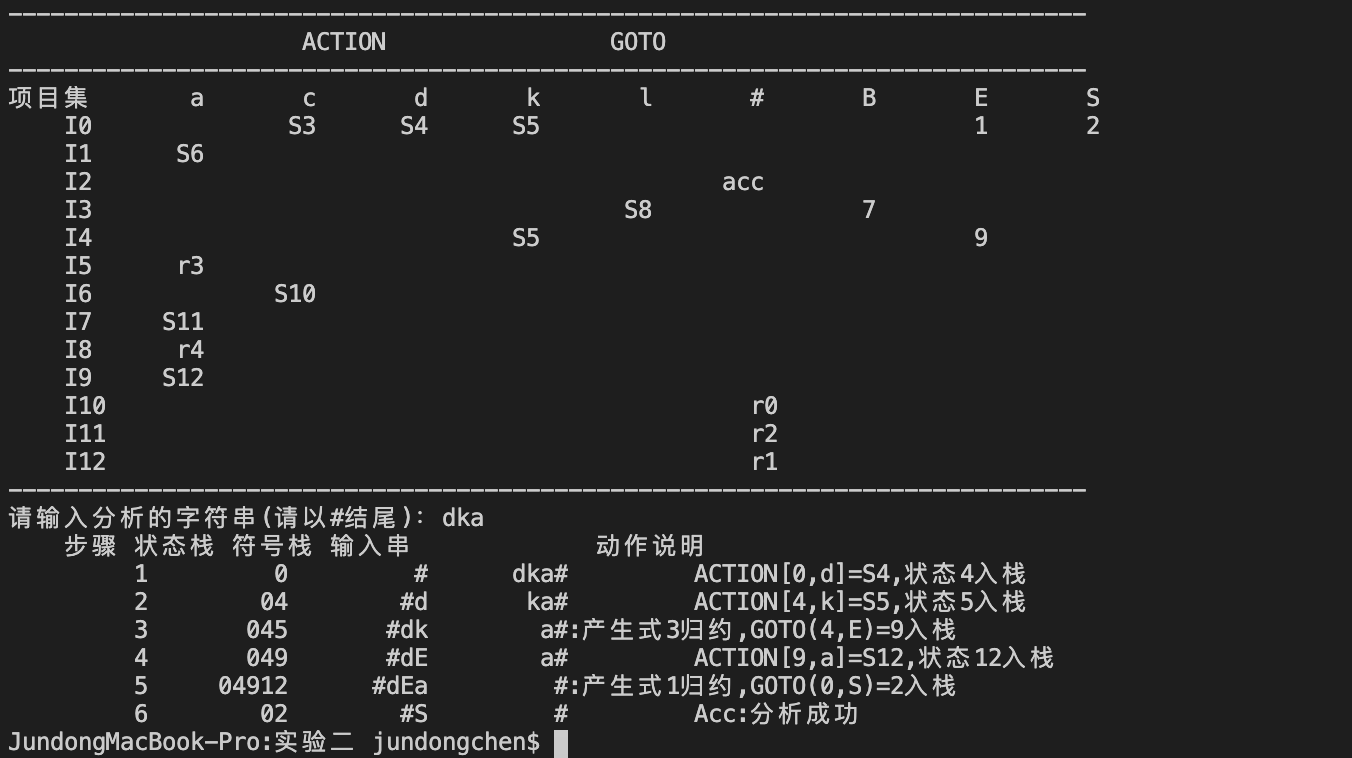
Input\_Analyse(); //输入具体文法开始分析

}

运行结果：







1. **实验总结**

本次实验有一定难度，在本次实验中我使用了LR(1)进行文法的自动化分析，也就是程序的输入是在文本中写入具体的文法产生式，然后根据产生式，求出对应非终结符的first集和follow集，再使用closure和goto语法生成项目集，最后再构建LR1分析表，再根据输入的字符串，进行查表分析，通过自底向上的不断进行查表，移进和归约动作，最后生成判断。在使用LR1过程中，我发现即使熟悉构建LR1分析表的构建过程，在程序实现过程还是很难的，期间也查了许多关于构建LR1分析表和分析过程的例子，最后终于写出来。