## **青岛科技大学实验报告**

姓名： 刘洋 专业： 计算机科学与技术 班级： 计算213

学号： 2108010316 课程： 编译原理 时间： 2024.05.19

实验项目： 实验3 LR(1)分析器的设计与实现

## 实验目的和要求

构造设计一个LR(1)分析器，即编制一个允许规范族有冲突的项目集用向前查看一个符号的办法来进行处理，并且能够解决存在的无效归约问题，以解决冲突的分析过程。

## 实验原理

**1. LR(1)项目集族的构造闭包函数(CLOSURE)求闭包的思想：**

以, # 属于初始项目集中，把 # 作为向前搜索符，包括活前缀为（若是有关S产生式的某一右部）要归约成S时，必须面临输入符为 # 才行。对初始项目, # 求闭包后，再用转换函数逐步求出整个文法的LR(1)项目集族。具体构造步骤如下：

（1）构造LR(1)项目集的闭包函数：① 假定I是一个项目集，I的任何项目都属于CLOSURE(I)；② 若有项目属于CLOSURE(I)，是文法中的产生式，∈，b∈FIRST(，则也属于CLOSURE(I)；③ 重复②，直到CLOSURE(I)不再增大为止。

（2）构造转换函数：GO(I, X) = CLOSURE(J)，其中I是LR(1)的项目集的状态，X是文法符号，J = {任何形如的项目| I}。

（3）对文法G’的LR(1)项目集族的构造以为初态集的初始项目，然后对其求闭包和转换函数，直到项目集不再增大为止。

**2. LR(1)分析表的构造：**

若已构造出某文法的LR(1)项目集族C:

C = {I0, I1, …, In}

其中Ik的k为分析器的状态，则动作(ACTION)表和状态转换(GOTO)表构造方法如下：

（1）若项目属于Ik，且GO(Ik, a) = Ij，其中a为终结符，则置ACTION[k, a] = Sj，Sj的含义是把符号a和状态j分别移入文法符号栈和状态栈。

（2）若项目属于Ik，则置ACTION[k, a] = rj。其中a为终结符，rj的含义为把当前栈顶符号串规约为A，（即用产生式规约），j为在文法中产生式的编号。

（3）若项目属于Ik，则置ACTION[k, #] = acc，表示“接受”。

（4）若GO(Ik, A) = Ij，其中A为非终结符，则置GOTO(k, A) = j，表示转入j状态，置当前文法符号栈顶为A，状态栈顶为j。

（5）凡不能用规则（1）~（4）填入分析表中的元素，均置为报错标志。

## 实验代码

1. #include <cstdio>  
   #include <vector>  
   #include <string>  
   #include <iostream>  
   #include <map>  
   #include <set>  
   #include <stack>  
   #include <queue>  
   #include <fstream>  
   #include <iomanip>  
   using namespace std;  
     
   // 产生式结构体，左部符号和右部符号串  
    struct Production {  
    char left;  
    vector<char> rigths;  
    /\* 重载== \*/  
    bool operator == (Production& rhs) const {  
    if (left != rhs.left)  
    return false;  
    for (int i = 0; i < rigths.size(); i++) {  
    if (i >= rhs.rigths.size())  
    return false;  
    if (rigths[i] != rhs.rigths[i])  
    return false;  
    }  
    return true;  
    }  
    };  
     
   // LR1项目  
    struct LR1Item {  
    Production p; // 产生式  
    int location; // 点的位置  
    char next; // 向前搜索符  
    };  
     
   // LR1项目集  
    struct LR1Items {  
    vector<LR1Item> items;  
    };  
     
   // LR1项目集规范族  
    struct CanonicalCollection {  
    vector<LR1Items> items; //项目集集合  
    vector< pair<int, char> > g[100]; // 保存DFA的图，first为转移到的状态序号，second是经什么转移  
    } CC;  
     
   // 文法结构体  
    struct Grammar {  
    int num; // 产生式数量  
    vector<char> T; // 终结符  
    vector<char> N; // 非终结符  
    vector<Production> prods; //产生式  
    }grammar;  
     
    map<char, set<string>> p; //产生式  
    map<char, set<char> > first, follow; //first集和follow集  
    map<char, int> done; //求first集是标记是否已经处理完  
    map<char, int> nul; //是否可以推导出空  
    set<char> vn, vt; //非终结符和终结符  
    queue< pair<LR1Items, int> > Q; //DFA队列， 用于存储待转移的有效项目集  
    pair<int, int> action[100][100]; //action表，first表示分析动作，0->ACC 1->S 2->R second表示转移状态或者产生式序号  
    int goton[100][100]; //goto表  
    string str; //待分析串  
    vector< pair<int, char> > ST; // 分析栈  
    const char N = '&';  
    const string blank = " ";  
     
   //判断c是否是终结符(set)  
    inline bool in\_vt(char c) {  
    if (vt.find(c) != vt.end()) return true;  
    return false;  
    }  
     
   //判断c是否是非终结符(set)  
    inline bool in\_vn(char c) {  
    if (vn.find(c) != vn.end()) return true;  
    return false;  
    }  
     
   //判断c是否是终结符(vector返回位置)  
    int isInT(char c) {  
    for (int i = 0; i < grammar.T.size(); i++) if (grammar.T[i] == c) return i + 1;  
    return 0;  
    }  
   //判断c是否是非终结符(vector返回位置)  
    int isInN(char c) {  
    for (int i = 0; i < grammar.N.size(); i++) if (grammar.N[i] == c) return i + 1;  
    return 0;  
    }  
     
   //判断是否为小写字母  
    inline bool is\_low(char c) {  
    if (!(c >= 'A' && c <= 'Z')) return true;  
    return false;  
    }  
     
   //检查一个串中是否含有中介符  
    bool check\_has\_vt(const string &now) {  
    int len = (int) now.size();  
    for (int i = 0; i < len; i++)  
    if (in\_vt(now[i])) return true;  
    return false;  
    }  
     
   //判断是否都能推导出空  
    int all\_null(const string &now) {  
    if (now == "null") return true;  
    int len = (int) now.size();  
    int f = 0;  
    for (int i = 0; i < len; i++)  
    if (nul[now[i]] == 0) {  
    return 0;  
    } else if (nul[now[i]] == -1) {  
    f = 1;  
    }  
    if (f) return -1;  
    return 1;  
    }  
     
   //判断所有非终结符能否推导出空  
    void get\_nul() {  
    map<char, set<string>> cur(p); //复制一份产生式(因为涉及到删除，所以为了保护原产生式，在复制的这份上进行删除操作)  
    set<char> now; //存放状态未定的非终结符  
    for (auto it = cur.begin(); it != cur.end(); ++it) {  
    nul[it->first] = -1; //状态置为待定  
    for (auto i = it->second.begin(); i != it->second.end();) {  
    string t = \*i;  
    if (t != "null") for (int j = 0, len = (int) t.size(); j < len; j++)  
    if (is\_low(t[j])) vt.insert(t[j]);  
    if (\*i == "null") {  
    /\*  
    该非终结符能推导出空串，将该非终结符的标志置为1，并删除该非终结符的所有产生式  
    此处不进行删除操作，而是用now存放那些状态未定的非终结符，后续中只去操作now中的元素，的这样该非终结符也就变相被”删除“了  
    \*/  
    nul[it->first] = 1;  
    break;  
    } else if (check\_has\_vt(\*i)) {  
    //若产生式中存在终结符，则删除该产生式  
    auto d = i;  
    ++i; //指针先后移，再删除  
    it->second.erase(\*d);  
    continue;  
    } else {  
    //若全为非终结符，则保留该产生式  
    ++i;  
    }  
    }  
    if ((int) it->second.size() > 0 && nul[it->first] != 1) {  
    // 如果该非终结符还存在右部为全为非终结符的产生式并且状态不为1，则加入now中进行后续处理  
    now.insert(it->first);  
    } else if (nul[it->first] == -1) {  
    // 如果该非终结符不存在右部为全为非终结符的产生式并且状态为待定，则它无法推导为空串，置为0  
    nul[it->first] = 0;  
    }  
    }  
    string bk = " ";  
    cout << "\n\n" << bk << "一次扫描后剩下的产生式:" << '\n' << '\n';  
    for (auto it = cur.begin(); it != cur.end(); ++it) {  
    for (auto i = it->second.begin(); i != it->second.end(); ++i) {  
    cout << blank << it->first << " -> "<< \*i << '\n';  
    }  
    }  
    bk = " ";  
    cout << "\n\n" << bk << "一次扫描后状态:" << '\n' << '\n';  
    for (auto it = nul.begin(); it != nul.end(); ++it) {  
    cout << blank << it->first << ": " << it->second << '\n';  
    }  
    cout << '\n';  
    int flag; //用于记录特征有无变化  
    do {  
    flag = 0; //刚开始时置为无变化  
    for (auto it = now.begin(); it != now.end();) {  
    int f = 0; //用于判断now中是否应该删除该非终结符  
    for (auto i = cur[\*it].begin(); i != cur[\*it].end();) {  
    int check = all\_null(\*i);  
    if (check == 1) {  
    // 若产生式右部的非终结符全都能推导出空串，则该非终结符能推导出空  
    nul[\*it] = 1;  
    flag = f = 1; // (1)有变化 (2)在now中删除该非终结符(删除该非终结的所有产生式)  
    break;  
    } else if (check == 0) {  
    // 若产生式右部的非终结符有的不能推导为空，则删除该产生式  
    flag = f = 1; // 在now中删除该非终结符(删除该非终结的所有产生式)  
    auto d = i;  
    ++i;  
    cur[\*it].erase(d);  
    } else {  
    // 若产生式右部的非终结符还处于待定状态，则暂时不管  
    ++i;  
    }  
    }  
    // 如果没有产生式了并且该非终结的状态还是待定，则将该非终结的状态置为0并且从now中删除  
    if ((int) cur[\*it].size() == 0 && nul[\*it] == -1) {  
    nul[\*it] = 0;  
    f = 1;  
    }  
    if (f) {  
    // 该非终结符能推导出空或者不能推导出空，则now中删除该非终结符  
    auto d = it;  
    ++it;  
    now.erase(\*d);  
    } else {  
    ++it;  
    }  
    }  
    } while (flag);  
    bk = " ";  
    cout << "\n" << bk << "最终状态:" << '\n' << '\n';  
    for (auto it = nul.begin(); it != nul.end(); ++it) {  
    cout << blank << it->first << ": " << it->second << '\n';  
    }  
    }  
     
   //求first集时进行dfs  
    set<char> dfs(char now) {  
    if (in\_vt(now)) { //如果是非终结符，则first集为本身  
    set<char> t;  
    t.insert(now);  
    return t;  
    } else {  
    if (done[now]) //如果该非终结符的first集已经处理过了，则直接返回即可，不必重复求解  
    return first[now];  
    for (auto it = p[now].begin(); it != p[now].end(); ++it) {  
    string r = \*it;  
    if (r == "null") { //如果能推导出空串，则将空串加入first集  
    first[now].insert(N);  
    } else {  
    int len = (int) r.size();  
    for (int i = 0; i < len; i++) {  
    if (in\_vn(r[i])) { //若为非终结符，进行继续进行搜索，利用搜索结果更新first集  
    auto h = dfs(r[i]); //dfs搜索该非终结符的first集  
    if (nul[r[i]] && (h.find(N) != h.end())) h.erase(N); //如果有空串，先剔除空串  
    for (auto j = h.begin(); j != h.end(); ++j) first[now].insert(\*j); //补充first集  
    if (nul[r[i]] != 1) { //如果当前字母不能推导为空串，则跳出，即没有空串加入  
    break;  
    } else if (i == len - 1) { //如果当前字母能推导为空串，且是最后一个，则在fisrt集中加上空串  
    first[now].insert(N);  
    }  
    } else { //若为终结符，则first集就是它自身  
    first[now].insert(r[i]);  
    break;  
    }  
    }  
    }  
    }  
    done[now] = 1; // 打上标记，表示该非终结符的first集已经处理过了，下次使用时直接返回即可，不必重复求解  
    return first[now];  
    }  
    }  
     
   //求first集  
    void get\_first() {  
    for (auto it = vn.begin(); it != vn.end(); ++it) dfs(\*it); //对每个非终结符进行dfs  
    string bk = " ";  
    cout << "\n\n" << bk << "First集为:" << '\n' << '\n';  
    for (auto it = first.begin(); it != first.end(); ++it) {  
    cout << blank << "FIRST(" << it->first << ") = {";  
    for (auto j = it->second.begin(); j != it->second.end();) {  
    cout << \*j++;  
    if (j != it->second.end()) cout << ",";  
    }  
    cout << "} \n";  
     
    }  
    }  
     
   //查找alpha串的FIRST集， 保存到FS集合中  
    void getFirstByAlphaSet(vector<char> &alpha, set<char> &FS) {  
    bool next = true; //当前符号是非终结符，且当前符号可以推出空，则还需判断下一个符号  
    int idx = 0;  
    while (idx < alpha.size() && next) {  
    next = false;  
    if (isInT(alpha[idx]) || alpha[idx] == N) { //当前符号是终结符或空，加入到FIRST集中  
    FS.insert(alpha[idx]);  
    } else {  
    char B = alpha[idx];  
    for (auto it = first[B].begin(); it != first[B].end(); it++) {  
    if (\*it == '&') { //当前符号FIRST集包含空，标记next为真  
    next = true;  
    continue;  
    } else { //把非空元素加入到FIRST集中  
    FS.insert(\*it);  
    }  
    }  
    }  
    idx++;  
    }  
    if (next) FS.insert(N); //如果到达产生式右部末尾next还为真，说明alpha可以推空，将空加入到FIRST集中  
    }  
     
   // 判断项目t否在项目集I中  
    bool isInLR1Items(LR1Items &I, LR1Item &t) {  
    for (auto it = I.items.begin(); it != I.items.end(); it++)  
    if (it->p == t.p && it->location == t.location && it->next == t.next) return true;  
    return false;  
    }  
     
   //求I的闭包  
    void closure(LR1Items &I) {  
    bool flag = true; //有变化判断有无变化  
    while (flag) {  
    flag = false;  
    LR1Items J;  
    J.items.assign(I.items.begin(), I.items.end());  
    for (auto it = J.items.begin(); it != J.items.end(); it++) { //枚举每个项目  
    LR1Item &L = \*it;  
    if (L.location < L.p.rigths.size()) { //非规约项目  
    char B = L.p.rigths[L.location]; //圆点右边的符号B  
    if (isInN(B)) { //如果B为非终结符, 把符合条件的LR1项目加入闭包中  
    set<char> FS; vector<char> alpha;  
    alpha.assign(L.p.rigths.begin() + L.location + 1, L.p.rigths.end());  
    alpha.push\_back(L.next);  
    getFirstByAlphaSet(alpha, FS); //求出B后面的FIRST集  
    for (int i = 0; i < grammar.prods.size(); i++) {  
    Production &P = grammar.prods[i];  
    if (P.left == B) {  
    //用B后面的FIRST集作为向前搜索符来生成新项目  
    for (auto it = FS.begin(); it != FS.end(); it++) {  
    LR1Item t;  
    t.location = 0;  
    t.next = \*it;  
    t.p.left = P.left;  
    t.p.rigths.assign(P.rigths.begin(), P.rigths.end());  
    if (!isInLR1Items(I, t)) {  
    flag = true; //有变化  
    I.items.push\_back(t); //将该项目加入项目集中  
    }  
    }  
    }  
    }  
    }  
    }  
    }  
    }  
    }  
     
   // 判断是否在项目集规范族中，若在返回序号  
    int isInCanonicalCollection(LR1Items &I) {  
    for (int i = 0; i < CC.items.size(); i++) {  
    LR1Items &J = CC.items[i];  
    bool flag = true;  
    if (J.items.size() != I.items.size()) {  
    flag = false;  
    continue;  
    }  
    //每个项目都在该项目集中，则认为这个两个项目集相等  
    for (auto it = I.items.begin(); it != I.items.end(); it++) {  
    LR1Item &t = \*it;  
    if (!isInLR1Items(J, t)) {  
    flag = false;  
    break;  
    }  
    }  
    if (flag) return i + 1;  
    }  
    return 0;  
    }  
     
   //转移函数，I为当前的项目集，J为转移后的项目集, 经X转移  
    void go(LR1Items &I, char X, LR1Items &J) {  
    for (auto it = I.items.begin(); it != I.items.end(); it++) {  
    LR1Item &L = \*it;  
    if (L.location < L.p.rigths.size()) { //非规约项目  
    char B = L.p.rigths[L.location];  
    if (B == X) { //如果能匹配，则点的位置加1并且新的项目加入项目集中  
    LR1Item t;  
    t.location = L.location + 1;  
    t.next = L.next;  
    t.p.left = L.p.left;  
    t.p.rigths.assign(L.p.rigths.begin(), L.p.rigths.end());  
    J.items.push\_back(t);  
    }  
    }  
    }  
    if (J.items.size() > 0) closure(J); //若J中有项目，则求其闭包  
    }  
     
   //打印某个项目集  
    void printLR1Items(LR1Items &I) {  
    for (auto it = I.items.begin(); it != I.items.end(); it++) {  
    LR1Item &L = \*it;  
    cout << blank << L.p.left << "->";  
    for (int i = 0; i < L.p.rigths.size(); i++) {  
    if (L.location == i)  
    cout << "·";  
    cout << L.p.rigths[i];  
    }  
    if (L.location == L.p.rigths.size())  
    cout << "·";  
    cout << "," << L.next << " \n";  
    }  
    }  
     
   //构建DFA和项目集规范族  
    void DFA() {  
    LR1Item t; //初始项目  
    t.location = 0; t.next = '#'; t.p.left = grammar.prods[0].left;  
    t.p.rigths.assign(grammar.prods[0].rigths.begin(), grammar.prods[0].rigths.end());  
    LR1Items I; //初始项目集  
    I.items.push\_back(t);  
    closure(I);  
    CC.items.push\_back(I); //将该项目集加入项目集规范族  
    Q.push(pair<LR1Items, int>(I, 0)); //把新加入的项目集加入待扩展队列中  
    while (!Q.empty()) {  
    LR1Items &S = Q.front().first;  
    int sidx = Q.front().second;  
    for (int i = 0; i < grammar.T.size(); i++) { //遍历每个终结符  
    LR1Items D;  
    go(S, grammar.T[i], D);  
    int idx;  
    if (D.items.size() > 0) { //若不为空  
    idx = isInCanonicalCollection(D); //查找是否已经在有效项目集族里  
    if (idx > 0) {  
    idx = idx - 1;  
    } else {  
    idx = (int) CC.items.size();  
    CC.items.push\_back(D); //将该项目集加入项目集规范族  
    Q.push(pair<LR1Items, int>(D, idx)); //将该项目集加入待扩展队列中  
    }  
    //从原状态到转移状态加一条边，边上的值为转移符号  
    CC.g[sidx].push\_back(pair<char, int>(grammar.T[i], idx));  
    }  
    }  
    for (int i = 0; i < grammar.N.size(); i++) { //遍历每个非终结符  
    LR1Items D;  
    go(S, grammar.N[i], D);  
    int idx;  
    if (D.items.size() > 0) {  
    idx = isInCanonicalCollection(D); //查找是否已经在有效项目集族里  
    if (idx != 0) {  
    idx = idx - 1;  
    } else {  
    idx = (int) CC.items.size();  
    CC.items.push\_back(D); //将该项目集加入项目集规范族  
    Q.push(pair<LR1Items, int>(D, idx)); //将该项目集加入待扩展队列中  
    }  
    //从原状态到转移状态加一条边，边上的值为转移符号  
    CC.g[sidx].push\_back(pair<char, int>(grammar.N[i], idx));  
    }  
    }  
    Q.pop(); //当前状态扩展完毕，弹出队列  
    }  
     
    const string bk = "\n\n ";  
    cout << bk << "项目集数量为: " << (int) CC.items.size() << '\n' << '\n';  
    for (int i = 0; i < (int)CC.items.size(); i++) {  
    cout << blank << "I" << i << ":" << '\n';  
    printLR1Items(CC.items[i]);  
    for (int j = 0; j < CC.g[i].size(); j++) {  
    pair<char, int> p= CC.g[i][j];  
    cout << blank << "识别 " << p.first << " 转移到 I" << p.second << '\n';  
    }  
    cout << '\n';  
    }  
    }  
   //生成LR1分析表  
    void productLR1AnalysisTabel() {  
    for (int i = 0; i < CC.items.size(); i++) {  
    LR1Items &LIt= CC.items[i];  
    /\* 构建action表 \*/  
    for (auto it = LIt.items.begin(); it != LIt.items.end(); it++) {  
    LR1Item &L = \*it;  
    /\* 非规约项目 \*/  
    if (L.location < L.p.rigths.size()) {  
    char a = L.p.rigths[L.location];  
    int j = isInT(a);  
    /\* a是终结符 \*/  
    if (j > 0) {  
    j = j - 1;  
    /\* 找到对应a的出边，得到其转移到的状态 \*/  
    for (int k = 0; k < CC.g[i].size(); k++) {  
    pair<char, int> p = CC.g[i][k];  
    if (p.first == a) {  
    action[i][j].first = 1; // 1->S  
    action[i][j].second = p.second; //转移状态  
    break;  
    }  
    }  
    }  
    } else { // 规约项目  
    /\* 接受项目 \*/  
    if (L.p.left == grammar.prods[0].left) {  
    if (L.next == '#')  
    action[i][grammar.T.size() - 1].first = 3;  
    } else {  
    /\* 终结符 \*/  
    int j = isInT(L.next) - 1;  
    /\* 找到产生式对应的序号 \*/  
    for (int k = 0; k < grammar.prods.size(); k++) {  
    if (L.p == grammar.prods[k]) {  
    action[i][j].first = 2;  
    action[i][j].second = k;  
    break;  
    }  
    }  
     
    }  
    }  
    }  
    /\* 构建goto表 \*/  
    for (int k = 0; k < CC.g[i].size(); k++) {  
    pair<char, int> p = CC.g[i][k];  
    char A = p.first;  
    int j = isInN(A);  
    /\* 终结符 \*/  
    if (j > 0) {  
    j = j - 1;  
    goton[i][j] = p.second; //转移状态  
    }  
    }  
    }  
    const string bk = " ";  
    //打印LR1分析表  
    cout << '\n' << "LR1分析表为：" << '\n';  
    cout << bk;  
    for (int i = 0; i < grammar.T.size() / 2; i++)  
    printf("\t");  
    printf("action");  
    for (int i = 0; i < grammar.N.size() / 2 + grammar.T.size() / 2 + 1; i++)  
    printf("\t");  
    printf(" goto\n");  
    printf("\t");  
    cout << bk;  
    for (int i = 0; i < grammar.T.size(); i++) {  
    printf("%c\t", grammar.T[i]);  
    }  
    printf("|\t");  
    for (int i = 1; i < grammar.N.size(); i++) {  
    printf("%c\t", grammar.N[i]);  
    }  
    printf("\n");  
    for (int i = 0; i < CC.items.size(); i++) {  
    cout << bk; printf("%d\t", i);  
    for (int j = 0; j < grammar.T.size(); j++) {  
    if (action[i][j].first == 1) {  
    printf("%c%d\t", 'S', action[i][j].second);  
    } else if (action[i][j].first == 2) {  
    printf("%c%d\t", 'R', action[i][j].second);  
    } else if (action[i][j].first == 3) {  
    printf("ACC\t");  
    } else {  
    printf("\t");  
    }  
    }  
    printf("|\t");  
    for (int j = 1; j < grammar.N.size(); j++) {  
    if (goton[i][j]) {  
    printf("%d\t", goton[i][j]);  
    } else {  
    printf("\t");  
    }  
     
    }  
    printf("\n");  
    }  
    }  
     
    void initGrammar() {  
    ifstream OutFile;  
    OutFile.open("/desktop/编译原理/实验/2108010316刘洋/LR1/production.txt");  
    OutFile >> grammar.num;  
    string s;  
    for (int i = 0; i < grammar.num; i++) {  
    OutFile >> s;  
    Production tmp;  
    tmp.left = s[0];  
    p[s[0]].insert(s.substr(3));  
    vn.insert(s[0]);  
    for (int j = 3; j < s.size(); j++) tmp.rigths.push\_back(s[j]);  
    grammar.prods.push\_back(tmp);  
    }  
    char ch;  
    OutFile >> ch;  
    while (ch != '#') {  
    grammar.N.push\_back(ch);  
    OutFile >> ch;  
    }  
    OutFile >> ch;  
    while (ch != '#') {  
    grammar.T.push\_back(ch);  
    OutFile >> ch;  
    }  
    grammar.T.push\_back('#'); //把#当作终结符  
    get\_nul(); //先扫一遍，判断每个非终结符是否可以推导为空  
    get\_first(); //求FIRST集  
    DFA(); //构建DFA  
    productLR1AnalysisTabel(); //构建LR1预测分析表  
    //读入待分析串并初始化分析栈  
    cout << "\n\n 请输入待分析串: ";  
    cin >> str;  
   // str = "cccdcd";  
    str += '#';  
    ST.push\_back(pair<int, char>(0, ' '));  
    }  
     
   // 分析程序  
    void process() {  
    int ip = 0;  
    int num = 1;  
    string bk = " ";  
    cout << "\n\n" << bk << "分析过程:" << '\n' << '\n';  
    cout << "\t\t" << "步骤" << "\t\t" << "状态栈" << "\t\t符号栈" << "\t\t" << "剩余输入串" << "\t\t" << "ACTION" << "\t\t" << "GOTO" << '\n';  
    do {  
    cout << "\t\t" << " " << num++;  
    cout << "\t\t"; for (auto it : ST) cout << it.first;  
    cout << "\t\t"; for (auto it : ST) cout << char(it.second);  
    cout << "\t"; string cur = str.substr(ip); cout << setw(15) << cur;  
    cout << "\t\t\t";  
    int s = ST.back().first;  
    char a = str[ip];  
    int j = isInT(a) - 1;  
    if (action[s][j].first == 1) { //移进  
    ST.push\_back(pair<int, char>(action[s][j].second, a));  
    ip = ip + 1;  
    cout << " S" << action[s][j].second << "\t\t";  
    } else if (action[s][j].first == 2) { // 规约  
    Production &P = grammar.prods[action[s][j].second];  
    cout << " r" << action[s][j].second << "\t\t ";  
    /\* 弹出并输出产生式 \*/  
   // printf("%c->", P.left);  
    for (int i = 0; i < P.rigths.size(); i++) {  
    ST.pop\_back();  
   // printf("%c", P.rigths[i]);  
    }  
    s = ST.back().first;  
    char A = P.left;  
    j = isInN(A) - 1;  
    ST.push\_back(pair<int, char>(goton[s][j], A));  
    cout << goton[s][j];  
    } else if (action[s][j].first == 3) { //接受  
    printf(" ACC\n");  
    return;  
    } else {  
    printf("error\n");  
    return;  
    }  
    cout << '\n';  
    } while(1);  
    }  
    int main() {  
    initGrammar();  
    process();  
    return 0;  
    }

## 实验结果

**1. 存放产生式的文件(data.txt)内容为：(格式解释：“4”表示有4条产生式，所以数字4后面有4条产生式，倒数第二行表示所有的非终结符，空格隔开，以‘#’结束，最后一行表示所有的终结符，空格隔开，以‘#’结束)**

4

A->S

S->CC

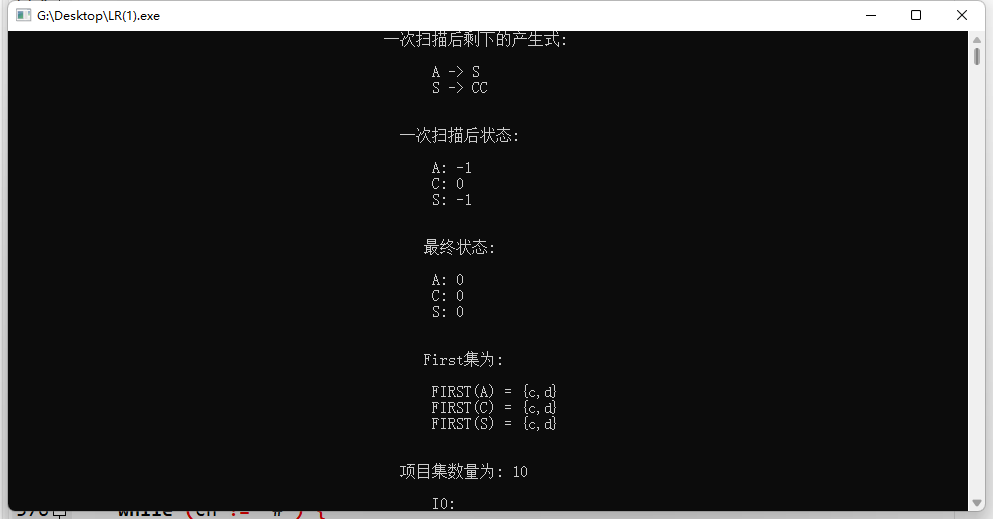
C->cC

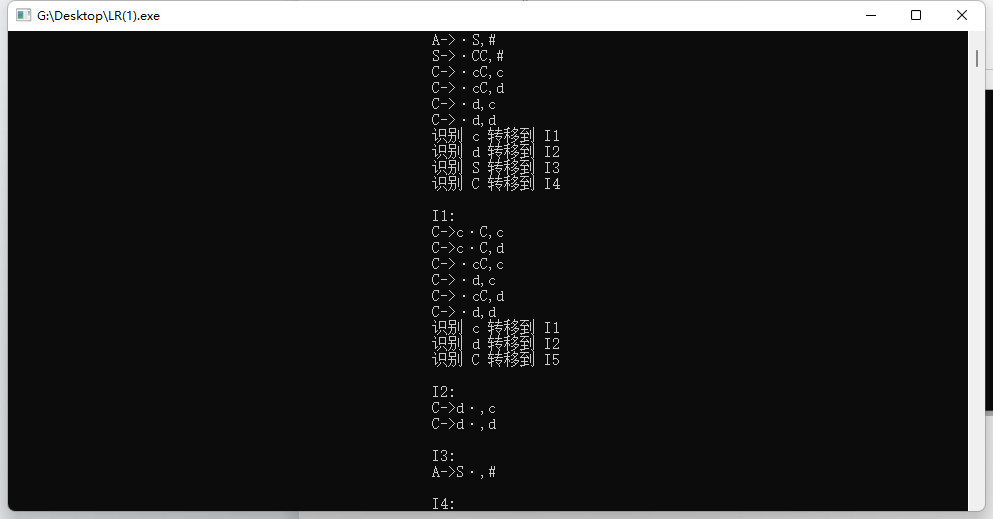
C->d

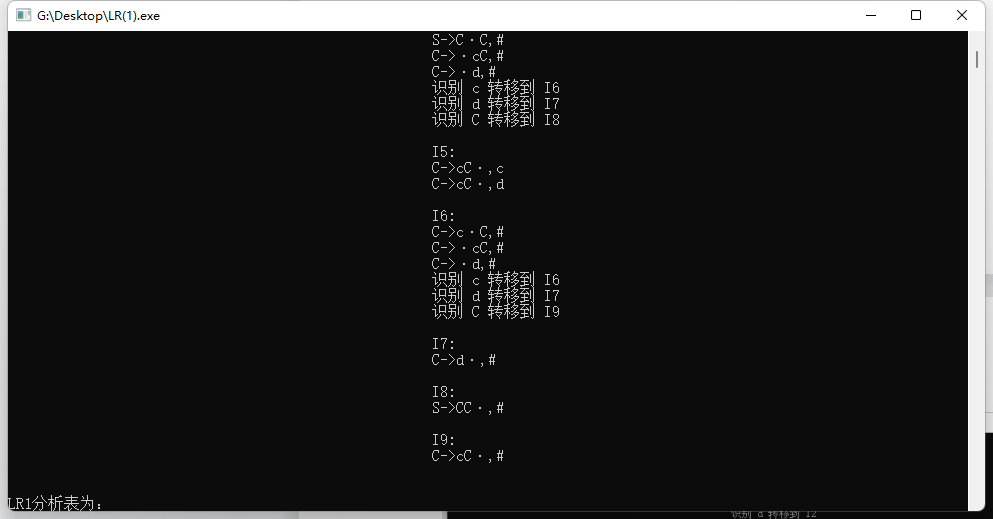
A S C #

c d #

**2. 读取产生式文件(data.txt)，构造FIRST集、项目集族和LR(1)分析表：**









## 实验总结

1．通过本次实验对LR(1)分析器的设计与实现，让我更加了解了LR(1)的分析表构造过程以及对输入串的分析过程。在完成实验的过程中，也是对知识的复习，通过将LR(1)分析法与其他LR分析法进行比较，让我更深刻地记住了它们之间的区别。

2．LR分析法小结：

LR(0)：到达终态就规约，由于大多数实用的程序设计语言的文法不能满足LR(0)文法的条件，也就是说很少有语言是LR(0)的。

SLR(1)：到达终态时先看FOLLOW集，与FOLLOW集对应的项目归约，其它报错。是SLR(1)文法容许LR(0)中有冲突的项目集可以用前查看一个符号的方法来进行处理，以解决冲突。因为只对有冲突的的状态才向前查看一个符号，以确定做哪种动作，所以称这种分析方法为简单的LR(1)分析法，用SLR(1)表示，则构造的分析表为SLR分析表，SLR的DFA与LR(0)的DFA是相同的。

LR(1)：将向前搜索符作为规约时查看的符号集合，用以代替SLR(1)中的FOLLOW集。

LALR(1)：若两个项目集相同则将它们合并为一个，即合并同心集

分析法的分析能力比较：LR(1)>LALR(1)>SLR(1)>LR(0)。