**编译原理**

班级：23计算机科学与计算机

姓名：盲灯 学号：

实验三 自底向上语法分析器的构建

1. 实验要求

运用自底向上语法分析方法，针对给定的上下文无关文法，给出实验方案。预估实验中可能出现的问题。

1. 实验方案

使用SLR(1)文法。逐步完成Augmented Grammar、First&Follow Set、DFA、Parse Table。

然后将分析表处理为程序可读数据，程序根据分析表的内容得出分析过程和结果。

1.Augmented Grammar

首先对原文法G构造增强文法 G'，引入新的开始符号 E'：

E' → E

E → E + T | T

T → T \* F | F

F → ( E ) | id

增强文法保证分析器有一个明确的 Accept 出口。

2.First 集 & Follow 集计算

First 集：

First(E) = First(T) = First(F) = { '(', id }

Follow 集：

Follow(E) = { ')', $, + }

Follow(T) = { ')', $, + }

Follow(F) = { ')', $, +, \* }

3.构造 LR(0) DFA

将每个 形如 A → α • β 的项作为状态；

初始项： E' → • E，通过 GOTO 操作不断扩展状态机；

使用 Closure 和 GOTO 算法生成 DFA 状态集。

4.构造 SLR(1) 语法分析表

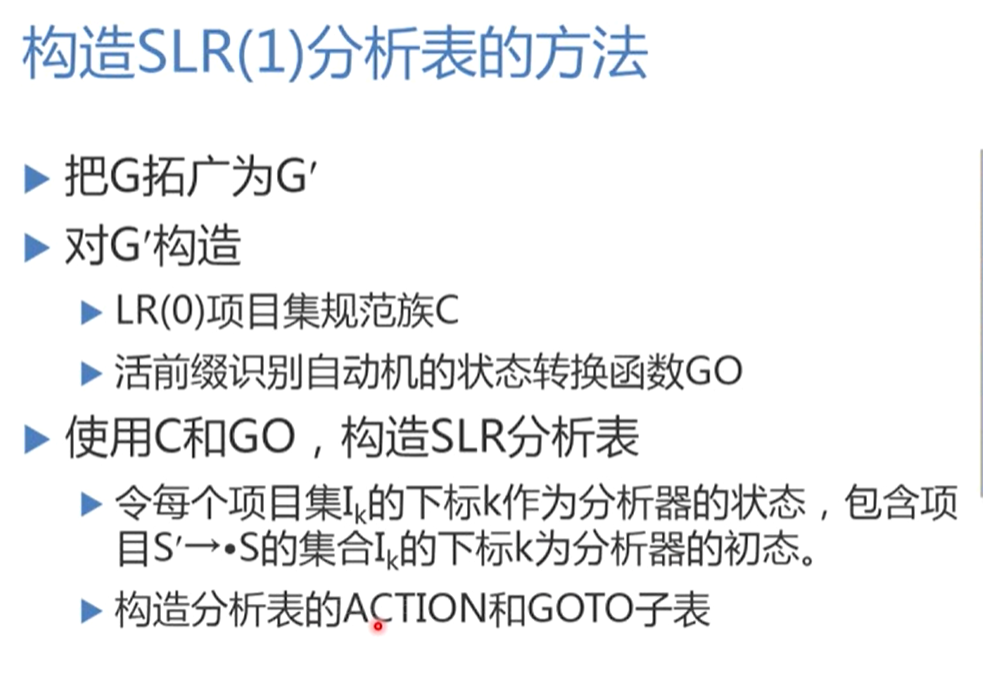
根据 DFA 中的状态项构造分析表：

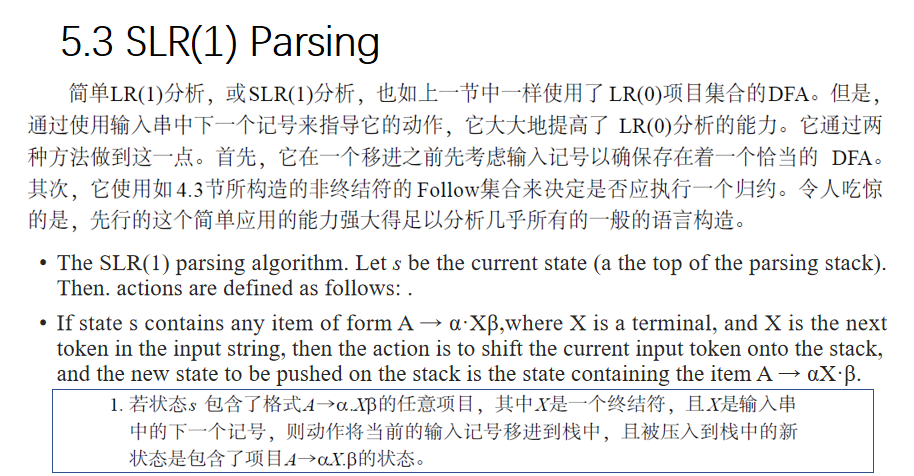
对于移进项 A → α • a β，定义 ACTION[state,a] = shift s'；

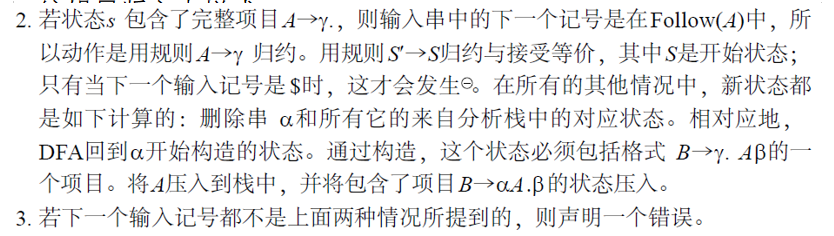
对于规约项 A → α •，在 Follow(A) 中定义 ACTION[state,a] = reduce A → α；

E' → E • 触发 Accept。

GOTO 表记录非终结符转移。







输入设计：

Parse Table、Augmented Grammar。

输出设计：

带有Parsing Stack、Input和Action的分析过程。

1. 分析表parsing table问题

理论和设计

1. 程序需要事先获得Augmented Grammar、Parse Table、Terminals、Non-terminals的具体内容，这些数据都要被处理为适当的格式，过程比较繁琐。
2. 可以将每个(non)terminal映射为数字，便于直接调用table(i,(non)terminal)。
3. 注意非终结符id的特殊处理。
4. 内容和步骤

1、考虑简单算术表达式文法G:

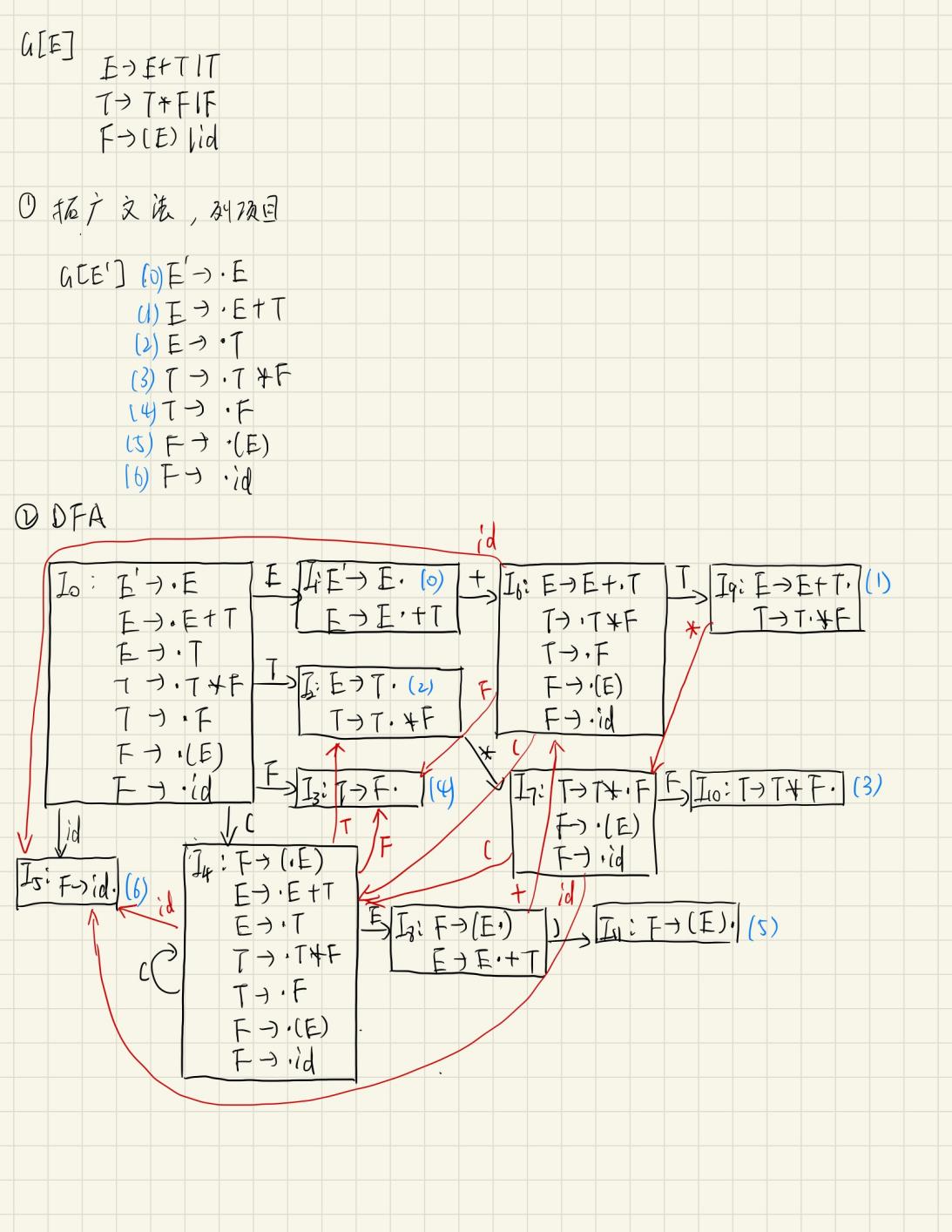
E→E + T | T

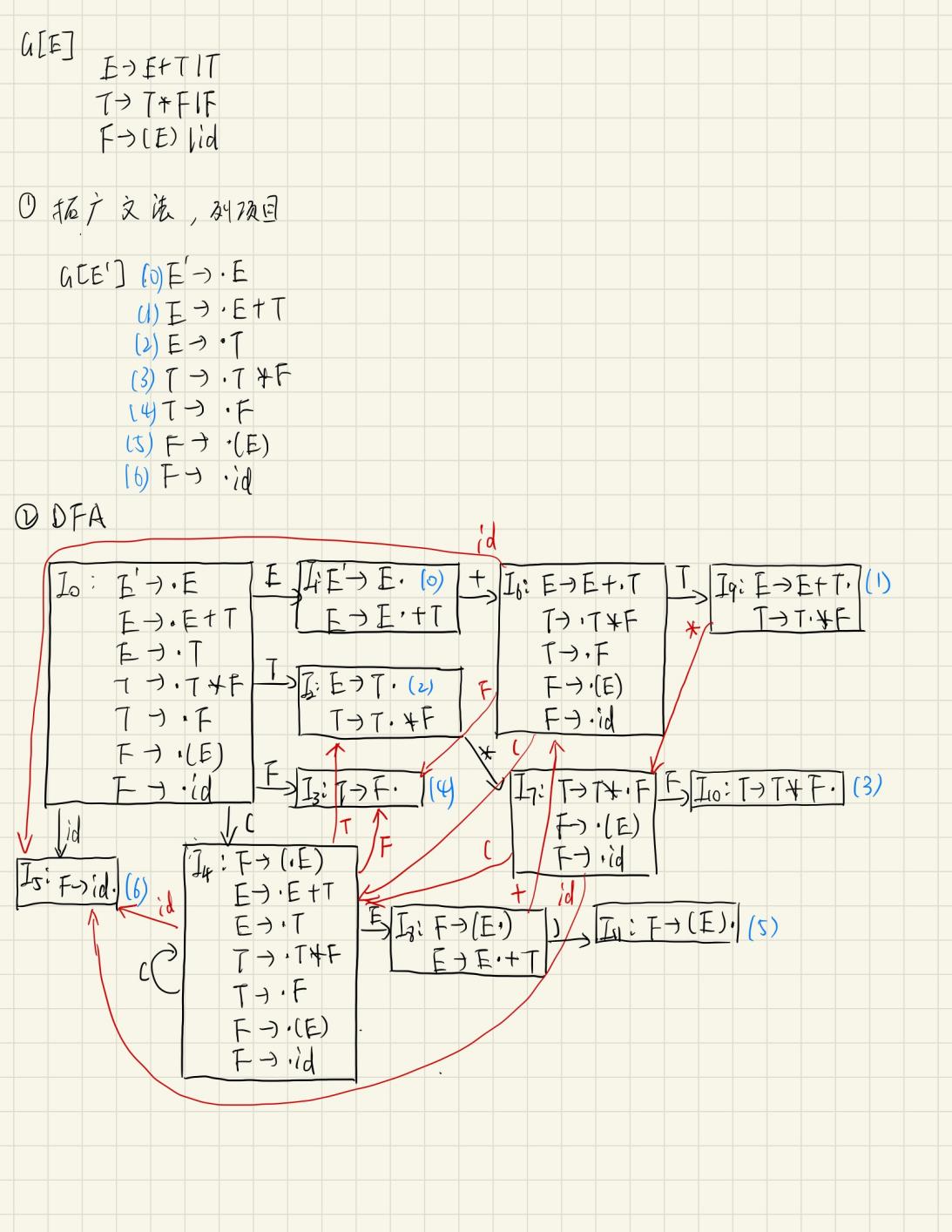
T→T \* F | F

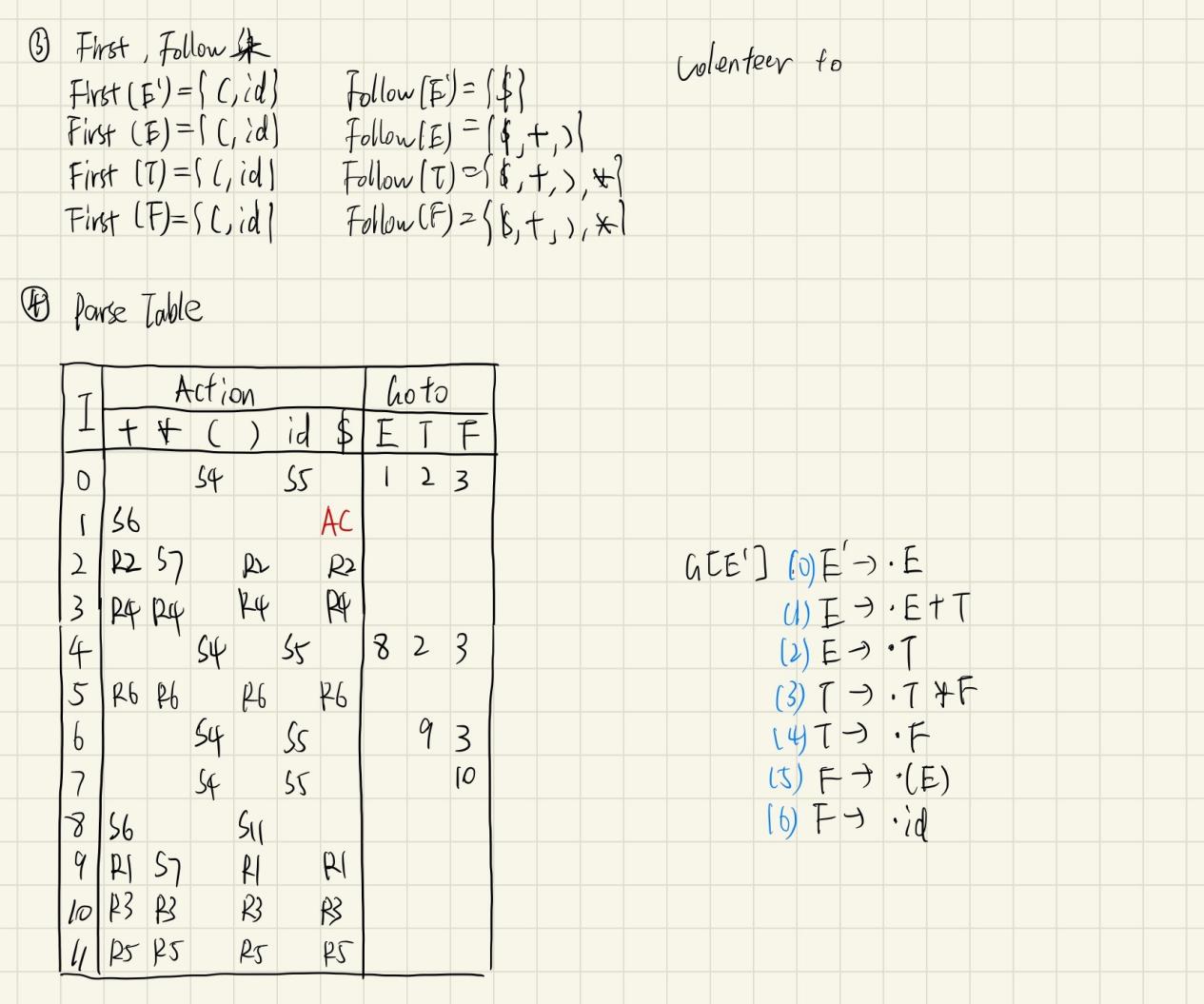
F→(E) | id

运用自底向上语法分析方法，对 (a+b)\*c+(d+e) 符号串进行语法分析。

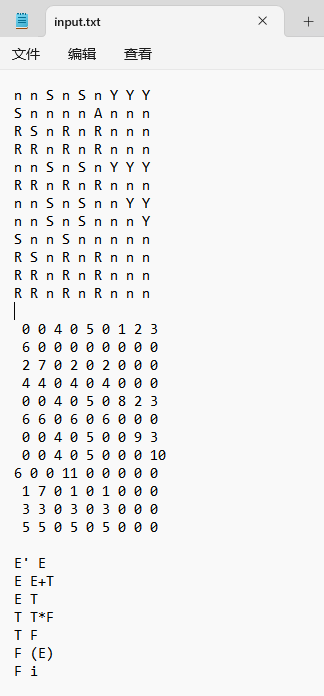
2、实验具体步骤







输入数据：



1. 实验结果:

1.代码

#include <iostream>

#include <iomanip>

#include <stack>

#include <string>

using namespace std;

int table[20][20][2];*// I,(non)terminals,action*

int mp[200];*//映射*

string terminals = "+\*()i$ETF";

string inputString = "(a+b)\*c+(d+e)";

stack<int> pStk, iStk; *//parse stack, input stack*

stack<int> reduceResult;*//存储规约结果, 用于分析树*

string reduce[7][2]; *//规约*

int step = 1;

namespace Graph { *//处理分析树*

    const int N = 1000, M = N \* 2;

    int nodemap[N] = { 0, 'E' };

    int depth[N];

    struct Edge {

        int to, nxt;

    }e[M];

    int adt, head[N];

    void add(int u, int v) {

        e[++adt] = { v,head[u] };

        head[u] = adt;

    }

    int fa[N];

    void dfs(int p1) {*//输出分析树*

        for (int i = 0; i < depth[p1]; i++) {

            cout << "   |";

        }

        cout << "--" << (char)nodemap[p1] << endl;

        for (int i = head[p1]; i != 0; i = e[i].nxt) {

            int p2 = e[i].to;

            if (p2 == fa[p1]) continue;

            fa[p2] = p1;

            dfs(p2);

        }

    }

    void parseTree() {

        int vst[1000] = { 0 };

        depth[1] = 1;

        int cnt = 2;

        while (!reduceResult.empty()) {*//读出规约结果*

            int rTop = reduceResult.top(); reduceResult.pop();

            int leftChar = reduce[rTop][0][0];*//产生式左部*

            string rightString = reduce[rTop][1];*//产生式右部*

            int oldCnt = cnt;

            for (int j = oldCnt - 1; j >= 1; j--) {*//从右往左匹配父节点*

                if (nodemap[j] == leftChar and !vst[j]) {

                    vst[j] = 1;

                    for (char k : rightString) {

                        nodemap[cnt] = k;*//给节点编号 加映射*

                        depth[cnt] = depth[j] + 1;

                        add(j, cnt);

                        cnt++;

                    }

                    break;

                }

            }

        }

        dfs(1);

    }

}

stack<int> reverse(stack<int> s) {

    stack<int> tmp;

    while (!s.empty()) {

        tmp.push(s.top());

        s.pop();

    }

    s = tmp;

    return s;

}

string reverse(string s) {*//禁止使用引用*

    reverse(s.begin(), s.end());

    return s;

}

void show(stack<int> ps, stack<int> is) {

    ps = reverse(ps);

    int width = 25;

    string str1, str2;

    int odd = 1;

    while (!ps.empty()) {

        if (odd == 1)  str1 += (char)ps.top();

        else str1 += to\_string(ps.top());

        odd \*= -1;

        ps.pop();

    }

    while (!is.empty()) {

        str2 += (char)is.top();

        is.pop();

    }

    string strBlank(width - str1.size() - str2.size(), ' ');

    cout << str1 << strBlank << str2;

}

void init() {

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

        mp[terminals[i]] = i; *// 映射 (non)terminals -> number*

        if (terminals[i] == 'i') {*// id = {a ~ z}*

            for (int j = 'a'; j <= 'z'; j++) {

                mp[j] = i;

            }

        }

    }

    for (int i = 0; i <= 11; i++) {

        for (int j = 0; j < terminals.size(); j++) {

            char c; cin >> c;

            table[i][j][0] = c;

        }

    }

    for (int i = 0; i <= 11; i++) {

        for (int j = 0; j < terminals.size(); j++) {

            int num; cin >> num;

            table[i][j][1] = num;

        }

    }

    for (int i = 0; i < 7; i++) {

        cin >> reduce[i][0] >> reduce[i][1];

        reduce[i][1] = reverse(reduce[i][1]);

    }

}

int parseTable() {

    pStk.push('$');

    pStk.push(0);

    iStk.push('$');

    for (int i = inputString.size() - 1; i >= 0; i--) {

        iStk.push(inputString[i]);

    }

    cout << "---------------------------"

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

    cout << "Step   Parsing Stack       Input    Action" << endl;

    while (!iStk.empty() and !pStk.empty()) {

        cout << step << "     ";

        if (step++ < 10) putchar(' ');

        show(pStk, iStk);

        int pTop = pStk.top(), iTop = iStk.top();

        int action[2] = { table[pTop][mp[iTop]][0],

                          table[pTop][mp[iTop]][1] };

        if (action[0] == 'A') {*// 接受*

            cout << "    Accept" << endl;

            return 200;

        } else if (action[0] == 'S') {*// 移进*

            cout << "    Shift " << action[1] << endl;

            pStk.push(iTop);

            pStk.push(action[1]);

            iStk.pop();

        } else if (action[0] == 'R') {*// 规约*

            reduceResult.push(action[1]);

            cout << "    Reduce " << action[1]

                << ": " + reduce[action[1]][0] + " -> "

                << reverse(reduce[action[1]][1]) << endl;

            for (auto i : reduce[action[1]][1]) {

                while (!pStk.empty()) {

                    int c = pStk.top();

                    pStk.pop();

                    if (c == i) break;

                    if (i == 'i') {

                        if ('a' <= c && c <= 'z') break;

                    }

                }

            }

            int pTop1 = pStk.top();

            pStk.push(reduce[action[1]][0][0]);

            int pTop2 = pStk.top();

            pStk.push(table[pTop1][mp[pTop2]][1]);

        } else {

            return 500;

        }

    }

    return 500;

}

void solve() {

    init();

    int res = parseTable();

    if (res == 200) {

        puts("\nParsing Success");

    } else {

        puts("\nParsing Failed");

    }

    Graph::parseTree();

}

int main() {

    FILE\* fp;

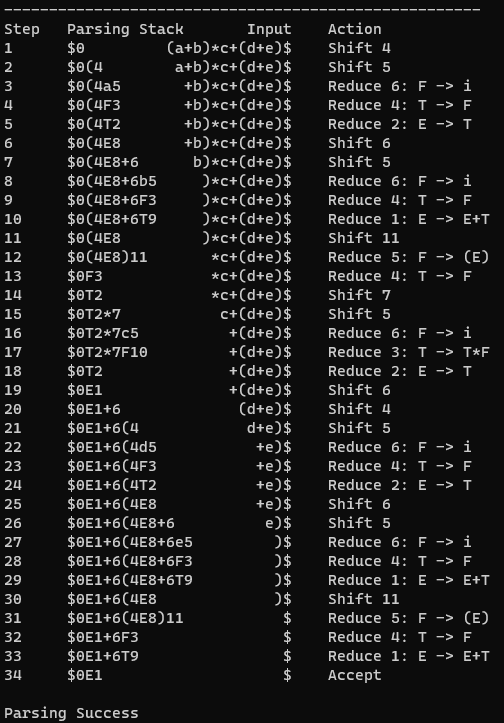
    freopen\_s(&fp, "input.txt", "r", stdin);

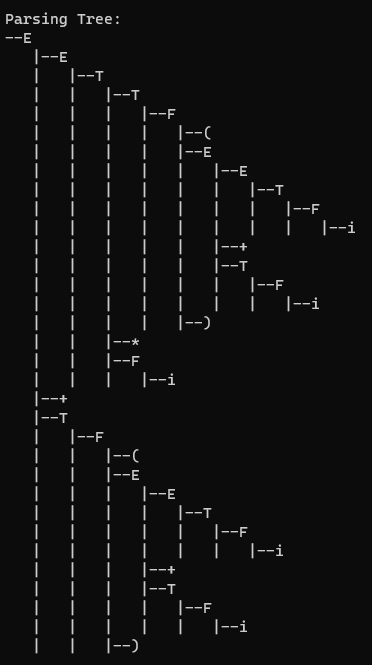
    solve();

    fclose(fp);

}

2.截图





1. 实验结论:

1 、实验结论

通过本次实验，我成功使用 SLR(1) 自底向上语法分析方法，针对简单算术表达式文法 G，完成了 Augmented Grammar、First/Follow 集、LR(0) DFA、Parsing Table 的构建，并基于分析表实现了程序可读的移进-规约分析器。程序能够正确解析输入串 (a+b)\*c+(d+e)，并输出完整的分析过程和对应的语法分析树，达到了预期效果。由此证明，本实验方案能够准确实现自底向上语法分析器的功能要求。

1. 分析和总结

（1）使用工具如 Yacc/Bison + Lex/Flex，可以非常高效地自动完成文法分析表构建，只需提供文法描述，工具即可生成完整的分析程序。这种方式效率高、出错率低，适合工程开发中使用。

（2）本次实验采用手动编写移进-规约分析器，虽然实现过程更繁琐，但能够深入理解 SLR(1) 分析器的核心算法，如 DFA 状态转移、First/Follow 计算、分析表填表策略等。通过实践掌握了分析表驱动语法分析过程的内部机制。

结论：

工具生成适合快速开发，手动编写有助于理解底层原理。二者结合使用效果最佳，手动实现的经验能帮助更好地掌握和调优自动生成的分析器。

1. 实验中出现的冲突及解决过程描述

（1）移进-归约冲突  
SLR(1) 方法可能在存在二义性文法时出现移进-归约冲突。通过合理设计 Follow 集，并调整文法优先级和结合性策略，解决了冲突，保证表中同一格不出现冲突动作。

（2）归约-归约冲突  
若某些 Follow 集交叉覆盖，可能导致归约-归约冲突。通过反复校验文法各产生式的设计，消除了归约-归约冲突，保证分析表一致性。

实际操作中遇到的问题

（3）First/Follow 集递归处理错误  
初始实现 First/Follow 集时未正确处理 ε 空产生式，导致 Follow 计算遗漏，造成分析器分析失败。通过完善递归终止条件，正确处理 ε 规则，解决了该问题。

（4）分析表编码错误  
最初将终结符和非终结符混合编号，导致程序在 GOTO 和 ACTION 查表时发生偏移，分析过程报错。通过重新明确映射关系，将终结符、非终结符分开编号，修正了表的设计。

（5）id 的特殊处理  
id 作为词法单元，需映射为整体 token，而非单字符 i。起初未区分，导致 id 归约失效。通过在映射阶段为 id 保留独立映射编号，正确解决了该问题，分析器能够正确识别并处理 id。