华 南 师 范 大 学 实 验 报 告

1 实验内容

设计一个应用软件，以实现SLR(1)分析生成器。

必做内容

(1)要提供一个文法输入编辑界面，让用户输入文法规则（可保存、打开存有文法规则的文件）
(2)求出文法各非终结符号的first集合与follow集合，并提供窗口以便用户可以查看这些集合结果。【可以采用表格的形式呈现】
(3)需要提供窗口以便用户可以查看文法对应的LR(0)DFA图。（可以用画图的方式呈现，也可用表格方式呈现该图点与边数据）
(4)要提供窗口以便用户可以查看该文法是否为SLR(1)文法。（如果非SLR(1)文法，可查看其原因）
(5)需要提供窗口以便用户可以查看文法对应的SLR(1)分析表。（如果该文法为SLR(1)文法时）【SLR(1)分析表采用表格的形式呈现】
(6)应该书写完善的软件文档
(7)应用程序应为Windows界面。

选做实验
(1)需要提供窗口以便用户输入需要分析的句子。
(2)需要提供窗口以便用户查看使用SLR(1)分析该句子的过程。【可以使用表格的形式逐行显示分析过程】

2 实验目的

设计一个应用软件，以实现SLR(1)分析生成器。

3 实验文档

3.1 实验文档：基于Qt的C++ SLR(1)文法分析生成器

3.2 引言

本实验旨在开发一个基于Qt的SLR(1)文法分析生成器，该软件能判断用户输入的文法是否为 SLR(1) 文法，并使用 SLR(1) 分析法进行文法分析。该程序能显示出文法各非终结符号的first集合与follow集合，生成文法的 LR(0)DFA 图以及对应的 SLR(1) 分析表。该程序还能对用户输入的句子进行分析，能显示出使用SLR(1)分析该句子的过程。SLR(1)文法分析生成器在软件工程规范下开发，具有高度的可维护性和扩展性。

3.3 数据结构与设计思路

核心过程由 **Solution.h** 中的 **Solution** 类来实现，简要声明如下：

```
class Solution {
public:
    Solution() = default;
    Solution(string str):str(str) {}
    void init();                // 初始化文法
    void clear();               // 清空文法
    void clearAnalysis();       // 清空分析栈和日志
    void Debug();               // 调试并运行
    void calcFirst();           // 求first集
    void calcFollow();          // 求follow集
    void calcDFA();             // 求DFA
    void judgeSLR1();           // 判断是否为 SLR(1) 文法
    void calcAnalysisTable();   // 求分析表
    void analysis();            // 分析句子

private:
    string str;                             // 输入的文法
    map<char, set<string>> grammar;         // 文法
    char start = ' ';                       // 文法开始符号
    set<char> non_terminal;                 // 非终结符号
    set<char> charSet;                      // 出现过的符号
    map<char, set<char>> first;             // first 集合
    map<char, set<char>> follow;            // follow 集合

    int idDFA = 0;                          // DFA 状态编号
    map<int, map<char, int>> DFA;           // DFA
    map<int, vector<Rule>> stateMap;        // 状态映射
    map<vector<Rule>, int> stateMapReverse; // 状态逆映射
    bool isSLR1 = false;                    // 判断是否为 SLR(1) 文法
    vector<Conflict> conflict;              // 冲突信息

    map<int, map<char, Action>> inputAnalysisTable; // 输入分析表
    map<int, map<char, int>> gotoAnalysisTable;     // GOTO 分析表

    string sentence;                        // 待分析的句子
    vector<StackNode> analysisStack;        // 分析栈
    vector<LogNode> analysisLog;            // 日志
};
```

在以上代码里使用到了几个自定义的数据结构，分别是：

- `Rule`：表示文法规则，包含产生式左部(**char first**)和右部(**string second**)以及当前运行位置(**int idx**)。
- 枚举类型 `ConflictType`：表示冲突类型，包括 `SHIFT_REDUCE_CONFLICT` 移进-归约冲突以及 `REDUCE_REDUCE_CONFLICT` 归约-归约冲突。
- `Conflict`：表示冲突信息，包括冲突类型(**ConflictType type**)、冲突产生式(**Rule rule**)、冲突状态编号(**int num**)。

- 枚举类型 `ActionType`：表示动作类型，包括 `SHIFT` 移进、`REDUCE` 归约以及 `ACCEPT` 接受。
- `Action`：表示动作，包括动作类型(**ActionType type**)、移进时进入的 **DFA** 状态编号(**int num**)以及归约时使用的规则(**Rule rule**)。
- `StackNode`：表示分析栈中的节点，用于对句子的分析，包括当前状态编号(**int state**)以及符号(**char symbol**)。
- `LogNode`：表示日志中的节点，用于记录分析过程，包括当前步骤编号(**int step**)、当前分析栈的字符串表示(**string analysisStackData**)、当前输入串(**string inputData**)、当前动作(**string action**)。

主要的数据结构

- `map<char, set<string>> grammar`：用于存储文法，其中 `char` 表示产生式左部，`set<string>` 表示产生式右部的集合。
- `map<char, set<char>> first, follow`：用于存储文法各非终结符号的 **first** 与 **follow** 集合。
- `map<int, map<char, int>> DFA`：用于存储 **DFA**，其中 `int` 表示 **DFA** 状态编号，`map<char, int>` 表示 **DFA** 状态转移表。
- `vector<Conflict> conflict`：用于存储文法中的冲突信息。
- `vector<LodNode> analysisLog`：用于存储分析过程中的日志信息。

软件的总体结构可以分为以下模块：

- 用户输入模块：用户输入模块负责对用户的输入进行预处理，转换为程序能够识别的格式，以方便后续处理。
- 文法处理模块：文法处理模块负责对用户输入的文法进行处理，包括求 **first** 集、**follow** 集、**DFA**、**SLR(1)**分析表等。
- 句子分析模块：句子分析模块负责对用户输入的句子进行分析，包括分析栈的维护、分析过程的记录等。

3.4 核心代码实现细节

3.4.1 用户输入模块

用户输入模块负责对用户的输入进行预处理，转换为程序能够识别的格式，以方便后续处理。用户输入的文法格式为：单一个大写字母作为非终结符号，非大写英文字母（除@外）作为终结符号，用@表示空串，默认左边出现的第一个大写字母为文法的开始符号。

```
E->E+T
E->T
T->a
```

程序需要对其进行预处理，将其转换为 `map<char, set<string>> grammar` 的形式，具体实现如下：

```cpp
void Solution::init() {
    istream *input = &cin;
    #if defined(INQT)
        stringstream ss(str);
        input = &ss;
    #endif
    string s;
    while (getline(*input, s)) {
        if (s == "#") break;
        s.erase(remove(s.begin(), s.end(), ' '), s.end());
        if (s.empty()) continue;
        char left = s[0];
        if (start == ' ') start = left;
        non_terminal.insert(left);
        charSet.insert(left);
        s = s.substr(3);
        // 处理可能存在的 |
        do {
            int idx = s.find('|');
            if (idx == string::npos) {
                grammar[left].insert(s);
                for (auto &c:s) if (c != '@') charSet.insert(c);
                break;
            }
            for (auto &c:s.substr(0, idx)) if (c != '@') charSet.insert(c);
            grammar[left].insert(s.substr(0, idx));
            s = s.substr(idx + 1);
        } while (true);
    }
    if (grammar[start].size() > 1) {    // 文法规则的扩充
        char new_start = 'Z';
        if (!non_terminal.count('S')) new_start = 'S';
        else while (non_terminal.count(new_start)) new_start--;
        non_terminal.insert(new_start);
        grammar[new_start].insert(string(1, start));
        start = new_start;
    }
}
```

本程序对 `|` 进行了特殊处理，使得用户不需要输入多条规则，而是可以用 `|` 连接多条规则。例如，用户可以输入 `E->E+T|T`，而不需要输入两条规则 `E->E+T` 和 `E->T`，简化了用户输入的复杂度。

这段代码还包括一些条件编译的部分，这是因为在编写程序的核心功能 **Solution** 时，我使用的是 **VsCode**，而在编写界面时，我使用的是 **Qt Creator**，两者的输入方式不同，因此使用条件编译可以很方便地切换输入方式，使得能轻松地在不同的 **IDE** 中进行调试。

3.4.2 求 first 集合

求 **first** 集合的算法在教材上已经以伪代码的形式给出：

程序清单4-5　为所有的非终结符A计算First (A)的算法

```
for all nonterminals A do First(A) := {};
while there are changes to any First(A) do
  for each production choice A → X1X2 . . . Xn do
    k := 1 ; Continue := true ;
    while Continue = true and k <= n do
      add First(Xk)−{ε} to First(A);
      if ε is not in First(Xk) then Continue := false ;
      k := k + 1 ;
    if Continue = true then add ε to First(A) ;
```

图1. 计算 first 集合的算法

算法的流程大致如下：

1. 初始化 first 集合，将所有非终结符号的 first 集合置为空集。
2. 重复以下步骤，直到 first 集合不再变化：
   (a) 对于每个产生式，如果产生式右部的第一个符号是终结符号，则将该终结符号加入产生式左部的 first 集合中。
   (b) 对于每个产生式，如果产生式右部的第一个符号是非终结符号，则将该非终结符号的 first 集合中的所有符号加入产生式左部的 first 集合中。
   (c) 对于每个产生式，如果产生式右部的第一个符号是非终结符号，并且该非终结符号的 first 集合中包含空串，则将产生式右部的第二个符号加入产生式左部的 first 集合中。
   (d) 对于每个产生式，如果产生式右部的所有符号都是非终结符号，并且所有非终结符号的 first 集合中都包含空串，则将空串加入产生式左部的 first 集合中。

根据以上算法，我实现了求 first 集的函数 `calcFirst()`，代码如下：

```cpp
void Solution::calcFirst() {
    for (auto &it:non_terminal) {
        first[it] = set<char>();
    }
    bool flag = true;
    while (flag) {  // 直到 first 集合不再变化
        flag = false;
        for (auto &[left, right]:grammar) {     // 遍历每个非终结符号
            for (auto &s:right) {               // 遍历每个产生式
                int k = 0, n = s.size();
                bool continue_flag = true;
                while (continue_flag && k < n) {
                    if (s[k] == '@') ;
                    else if (!isupper(s[k])) {  // 终结符号
                        flag |= first[left].insert(s[k]).second;
                        continue_flag = false;
                    } else {    // 非终结符号
                        for (auto &c:first[s[k]]) {     // 将 first[s[k]] 中的元素加入 first[left]
                            if (c != '@') {
                                flag |= first[left].insert(c).second;
                            }
```

```
                        }
                        if (first[s[k]].count('@') == 0) {  // 如果该非终结符号不含空串，则不再继续
                            continue_flag = false;
                        }
                    }
                    k++;
                }
                if (continue_flag) {    // 如果该产生式的每个符号都含有空串，则将空串加入first[left]
                    flag |= first[left].insert('@').second;
                }
            }
        }
    }
}
```

3.4.3 求 follow 集合

求 follow 集合的算法在教材上已经以伪代码的形式给出：

程序清单4-7 计算Follow集合的算法

Follow(start-symbol) := {$} ;
for *all nonterminals A* ≠ *start-symbol* **do** *Follow(A)* := {} ;
while *there are changes to any Follow sets* **do**
  for *each production* $A \rightarrow X_1X_2 \ldots X_n$ **do**
    for each X_i *that is a nonterminal* **do**
      add $First(X_{i+1}X_{i+2} \ldots X_n) - \{\varepsilon\}$ *to* $Follow(X_i)$
      (* *Note: if* $i=n$, *then* $X_{i+1}X_{i+2} \ldots X_n = \varepsilon$ *)
      if ε *is in* $First(X_{i+1}X_{i+2} \ldots X_n)$ **then**
        add Follow(A) to $Follow(X_i)$

图2. 计算 follow 集合的算法

算法的流程大致如下：

1. 初始化 follow 集合，将开始符号的 follow 集合置为文法结束符号 `$`。
2. 重复以下步骤，直到 follow 集合不再变化：
   (a) 对于每个产生式的右边的非终结符号，若
      i. 该非终结符号是产生式的最后一个符号，则将产生式左部的 follow 集合加入该非终结符号的 follow 集合中。
      ii. 该非终结符号的后一个符号是终结符号，则将该终结符号加入该非终结符号的 follow 集合中。
      iii. 该非终结符号的后一个符号是非终结符号，则将该非终结符号的后一个符号的 first 集合中的所有符号（除空串）加入该非终结符号的 follow 集合中。若该非终结符号的后一个符号的 first 集合中包含空串，则继续查看后一个符号，直到找到一个不含空串的符号。

根据以上算法，我实现了求 follow 集的函数 `calcFollow()`，代码如下：

```
void Solution::calcFollow() {
```

```
    for (auto &it:non_terminal) {
        follow[it] = set<char>();
    }
    follow[start].insert('$');      // 将 $ 加入 follow[start]
    bool flag = true;
    while (flag) {
        flag = false;
        for (auto &[left, right]:grammar) {     // 遍历每个非终结符号
            for (auto &s:right) {               // 遍历每个产生式
                // left -> X1 X2 ... Xn
                int k = 0, n = s.size();
                while (k < n) {
                    if (!isupper(s[k])) {k++; continue;}
                    // 非终结符号
                    int j = k + 1;
                    // Xk+1 Xk+2 ... Xn
                    while (j < n && isupper(s[j]) && first[s[j]].count('@')) {  // 该非终结符号的first集合含空串
                        for (auto &c:first[s[j]]) {
                            if (c != '@') {
                                flag |= follow[s[k]].insert(c).second;
                            }
                        }
                        j++;
                    }
                    if (j == n) {   // 该非终结符号的first集合含空串
                        for (auto &c:follow[left]) {
                            flag |= follow[s[k]].insert(c).second;
                        }
                    } else {    // 该非终结符号的first集合不含空串
                        if (!isupper(s[j])) {   // 终结符号
                            flag |= follow[s[k]].insert(s[j]).second;
                        } else {
                            for (auto &c:first[s[j]]) {
                                if (c != '@') {
                                    flag |= follow[s[k]].insert(c).second;
                                }
                            }
                        }
                    }
                    k++;
                }
            }
        }
    }
}
```

3.4.4 求 LR(0)DFA

我选择的算法是广度优先搜索 bfs 求 LR(0)DFA，算法的流程如下：

1. 初始化 DFA，将文法开始符号对应状态加入 DFA 中，并将其加入 bfs 队列中。

2. 重复以下步骤，直到 bfs 队列为空：

(a) 取出队首状态 **u**，遍历状态 **u** 中的每个产生式及其运行到的位置，为该 **DFA** 状态产生一条转移边，转移边的符号为产生式右部的下一个符号，转移的状态为该产生式右部下一个符号对应的 **DFA** 状态。若该 **DFA** 状态不存在，则新建一个 **DFA** 状态，并将其加入 **bfs** 队列中，否则直接将该 **DFA** 状态加入转移边中。

具体实现如下：

```cpp
void Solution::calcDFA() {
    auto addDependency = [&](vector<Rule> &v) { // 添加依赖
        bool flag = true;
        while (flag) {
            flag = false;
            vector<Rule> tmp;
            for (auto &[left, right, idx]:v) {
                if (idx == right.size() - 1) continue;  // .在最后
                if (!isupper(right[idx + 1])) continue; // .后面是终结符号
                char c = right[idx + 1];
                for (auto &s:grammar[c]) {
                    if (s == "@") tmp.push_back({c, "."});
                    else tmp.push_back({c, "." + s});
                }
            }
            for (auto &it:tmp) {
                if (find(v.begin(), v.end(), it) == v.end()) {
                    v.push_back(it);
                    flag = true;
                }
            }
        }
    };
    // DFA 初态
    DFA[idDFA] = map<char, int>();
    stateMap[idDFA] = {{start, "." + *grammar[start].begin()}};
    addDependency(stateMap[idDFA]);
    stateMapReverse[stateMap[idDFA]] = idDFA;

    queue<int> q;
    q.push(idDFA++);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        auto status = stateMap[u];
        map<char, vector<Rule>> temp;   // 临时存储
        for (auto &[left, right, idx]:status) {
            if (idx == right.size() - 1) continue;  // .在最后
            /* 添加新状态的产生式 */
        }
        for (auto &[nextC, needToAdd]:temp) {
            addDependency(needToAdd);   // 添加依赖
            /* 如果不存在该状态，则新建一个状态 */
            // 加边
            DFA[u][nextC] = stateMapReverse[needToAdd];
        }
    }
}
```

3.4.5 判断是否为 SLR(1) 文法

当且仅当对于任何状态s,以下的两个条件：

1. 对于在s中的任何项目$A \to \alpha.X\beta$，当X是一个终结符,且X在Follow (B)中时, s中没有完整的项目$B \to \gamma$。 [移进一归约冲突]

2. 对于在s中的任何两个完整项目$A \to \alpha$。和$B \to \beta$. Follow $(A)\cap$ Follow (B)为空。 [归约一归约冲突]

均满足时，文法为SLR(1)文法。

因此我们需要遍历 DFA 中的每一个状态，判断其是否存在冲突。具体实现如下：

```
void Solution::judgeSLR1() {
    isSLR1 = true;
    for (auto &[DFAidx, status]:stateMap) {
        map<char, vector<Rule>> shifting;   // 移进项
        vector<Rule> reduction; // 归约项
        /* 求出每个状态其中的移进项与归约项 */

        if (reduction.empty()) continue;
        // 移进-归约冲突
        for (auto &[left, right, idx]:reduction) {
            auto Follow = follow[left];
            for (auto [c, v]:shifting) {
                if (Follow.count(c)) {
                    /* 添加冲突信息 */
                }
            }
        }
        // 归约-归约冲突
        for (int i = 0; i < reduction.size(); i++) {
            for (int j = i + 1; j < reduction.size(); j++) {
                set<char> result;
                set_intersection(follow[reduction[i].first].begin(),
follow[reduction[i].first].end(),
                                 follow[reduction[j].first].begin(),
follow[reduction[j].first].end(),
                                 inserter(result, result.begin())); // 求 follow 交集
                if (result.empty()) continue;
                /* 添加冲突信息 */
            }
        }
    }
}
```

3.4.6 求 SLR(1) 分析表

我们在前面已经得到 LR(0)DFA 图以及每个 DFA 状态所包含的文法，因此求 SLR(1) 分析表就变得很简单了，只需要遍历 DFA 中的每一个状态，即可按部就班地得出 SLR(1) 分析表。具体实现如下：

```
void Solution::calcAnalysisTable() {
    // 填入移进项
    for (auto &[idx, m]:DFA) {
        for (auto &[c, next]:m) {

```

```
            if (isupper(c)) {
                gotoAnalysisTable[idx][c] = next;
            } else {
                inputAnalysisTable[idx][c].type = SHIFT;
                inputAnalysisTable[idx][c].num = next;
            }
        }
    }

    // 填入归约项
    for (auto &[DFAidx, status]:stateMap) {
        for (auto &[left, right, idx]:status) {
            if (idx == right.size() - 1) {  // 归约项
                if (left == start) {
                    inputAnalysisTable[DFAidx]['$'].type = ACCEPT;
                } else {
                    auto temp = right.substr(0, right.size() - 1);
                    if (temp.empty()) temp = "@";    // 空串
                    for (auto &c:follow[left]) {
                        inputAnalysisTable[DFAidx][c].type = REDUCE;
                        inputAnalysisTable[DFAidx][c].rule = {left, temp};
                    }
                }
            }
        }
    }
}
```

3.4.7 句子分析

本程序还实现了实验的选做内容——句子分析。句子分析的过程就是对分析栈的维护过程，具体实现如下：

```
void Solution::analysis() {
    int nowStep = 1;            // 当前步骤
    int analysisIdx = 0;        // 当前分析的位置
    sentence.push_back('$');
    analysisStack.emplace_back(0, '$');

    auto stack2str = [](const vector<StackNode> &v) {   // 分析栈转字符串
        string s;
        for (auto &[DFAidx, c]:v) {
            s += string(1, c) + " " + to_string(DFAidx) + " ";
        }
        return s;
    };

    while (true) {
        LogNode node(nowStep++, stack2str(analysisStack),
sentence.substr(analysisIdx), " ");
        int nowstate = analysisStack.back().state;
        char nowchar = sentence[analysisIdx];

        /* 如果没有该转移状态说明句子不符合文法规则，报错 */

```

```
        auto todo = inputAnalysisTable[nowstate][nowchar];
        if (todo.type == SHIFT) {    // 移进
            /* 输出移进信息 */
            analysisStack.emplace_back(todo.num, nowchar);
            analysisIdx++;
        } else if (todo.type == REDUCE) {    // 归约
            node.action = "用 " + string(1, todo.rule.first) + "->" +
todo.rule.second + " 归约";
            int len = todo.rule.second.size();
            if (len == 1 && todo.rule.second[0] == '@') len = 0;
            while (len--) analysisStack.pop_back();     // 弹出 len 个状态
            analysisStack.emplace_back(gotoAnalysisTable[analysisStack.back().state]
[todo.rule.first], todo.rule.first);    // 压入新状态
        } else {
            node.action = "接受";
        }
        analysisLog.push_back(node);
        if (todo.type == ACCEPT) break;
    }
}
```

以上就是本程序的核心算法，在可视化部分，只需要调用以上函数，将结果进行适当的转换，显示在界面上即可。

3.5 测试

选择以下文法规则与句子作为测试用例：

```
// 测试用例
E->E+n
E->n

// 句子分析
n+n+n
```

测试结果如下：

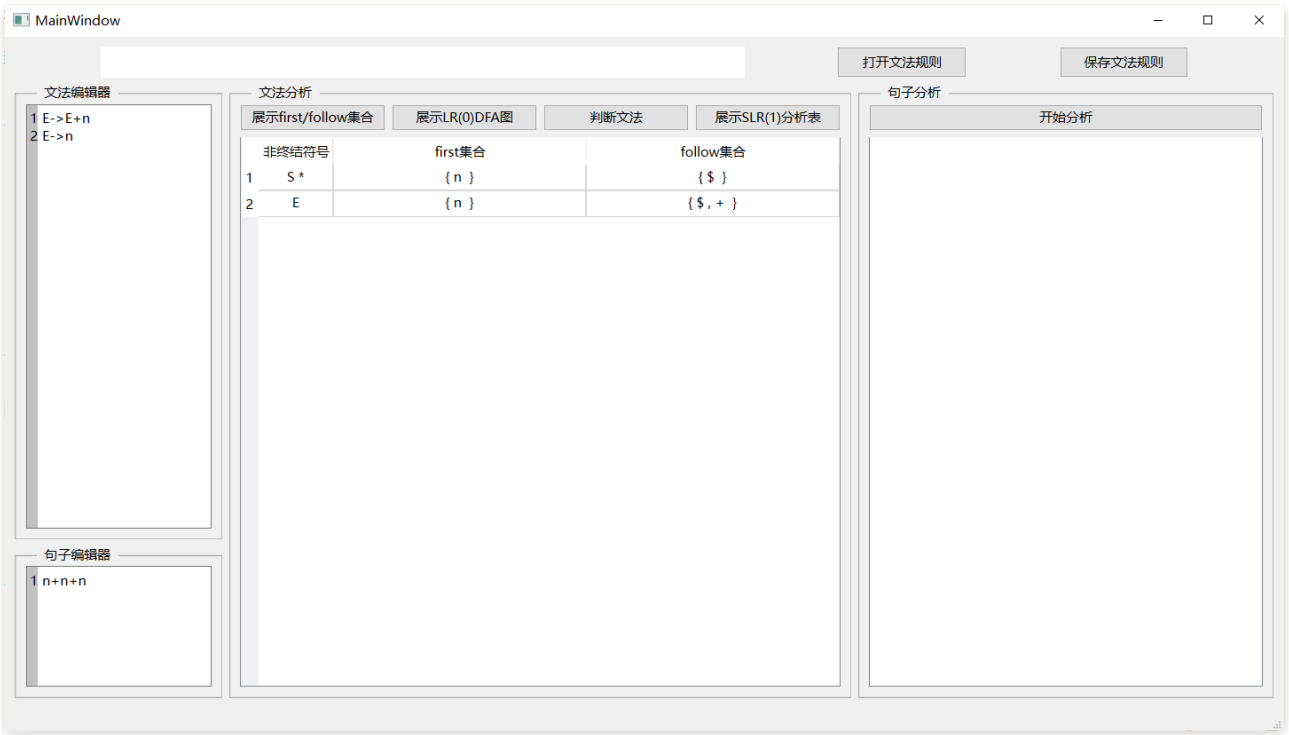

图3-1. 展示first/follow集合

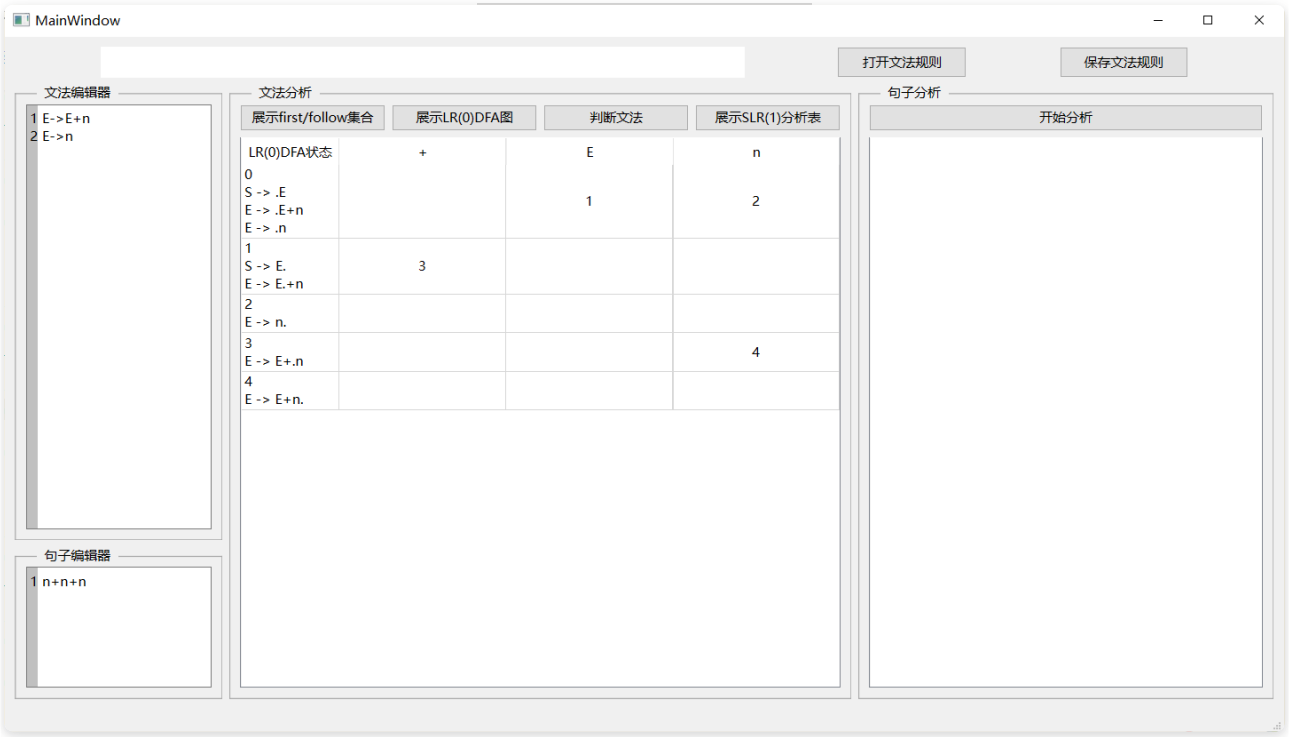

图3-2. 展示DFA

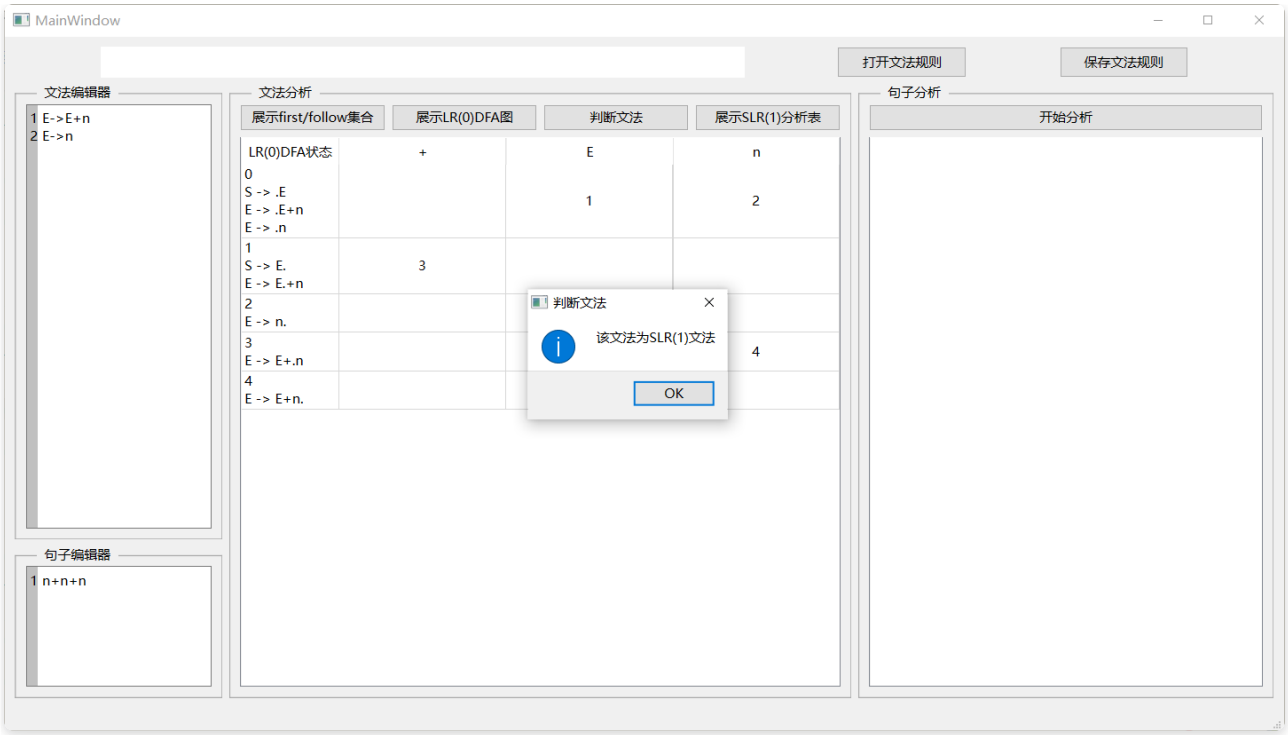

图3-3. 判断文法

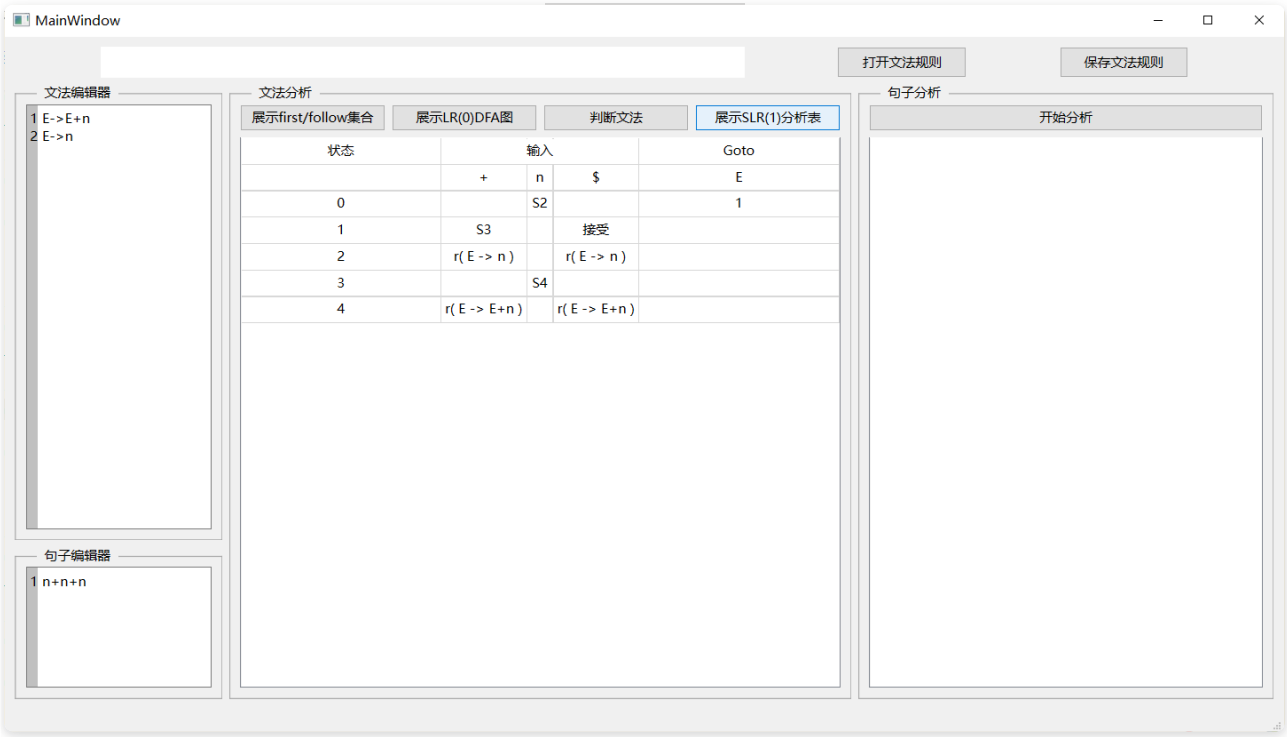

图3-4. 展示分析表

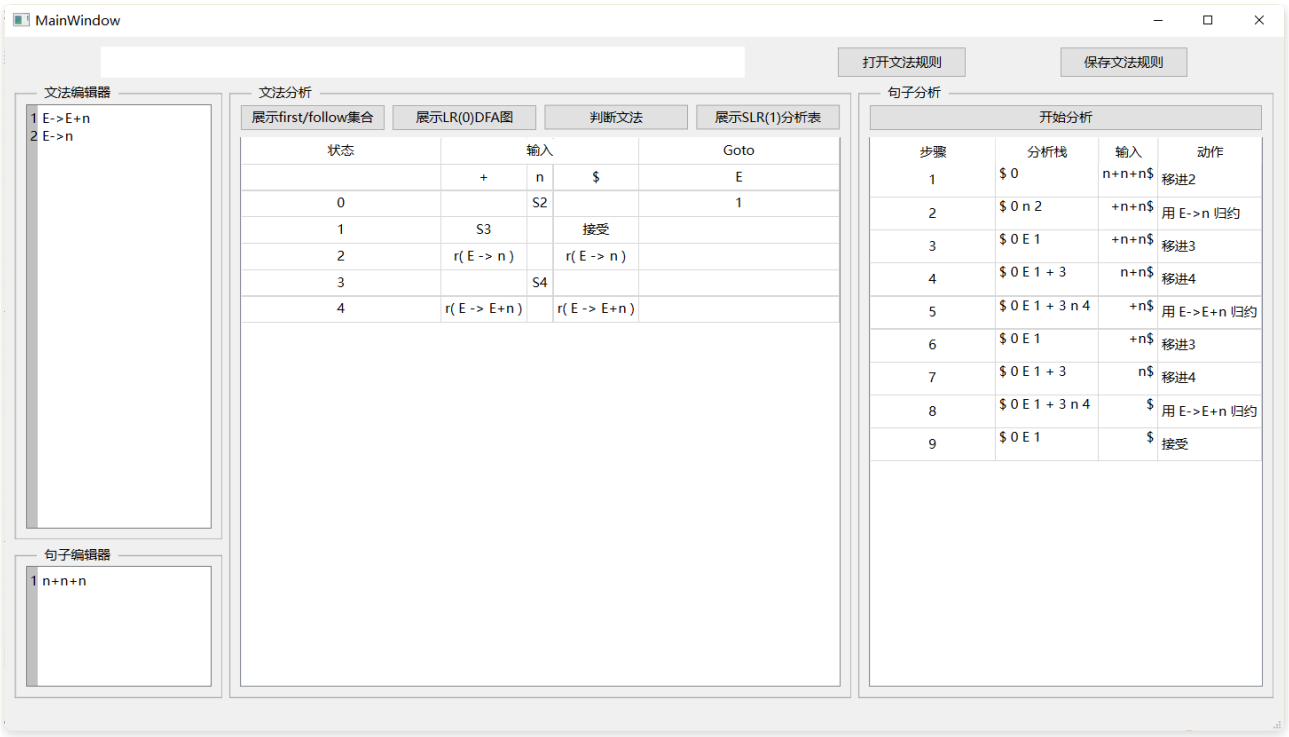

图3-5. 句子分析

将程序运行结果与教材上的结果进行对比：

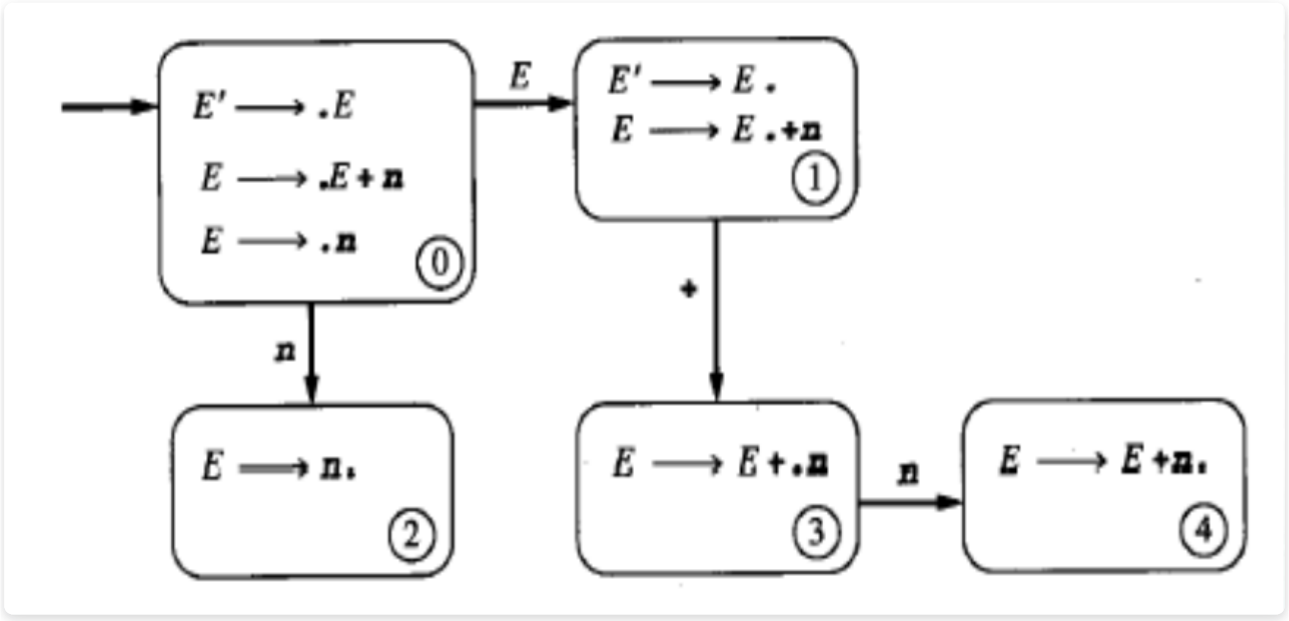

图3-6. 教材LR(0)DFA

状　态	输　入			Goto
	n	+	$	E
0	s2			1
1		s3	接受	
2		r(E→n)	r(E→n)	
3	s4			
4		r(E→E+n)	r(E→E+n)	

步骤	分析栈	输入	动作
1	$ 0	n + n + n $	移进2
2	$ 0 n 2	+ n + n $	用E→n归约
3	$ 0 E 1	+ n + n $	移进3
4	$ 0 E 1 + 3	n + n $	移进4
5	$ 0 E 1 + 3 n 4	+ n $	用E→E + n 归约
6	$ 0 E 1	+ n $	移进3
7	$ 0 E 1 + 3	n $	移进4
8	$ 0 E 1 + 3 n 4	$	用E→E + n 归约
9	$ 0 E 1	$	接受

图3-7. 教材SLR(1)分析表及句子分析

可以发现，本程序的结果与教材上的结果基本完全一致（除了某些 DFA 状态的节点编号不同外），说明本程序能正确地求出文法的 first 集、follow 集、DFA 图、SLR(1)分析表，并且能正确地判断文法是否为 SLR(1) 文法，能正确地对句子进行分析。

更多测试样例请看Testfile文件夹内的测试文档。

4　实验总结

这次实验让我深入理解了 SLR(1) 的文法分析方法。在实验过程中，我学会了如何求出文法的 first 集合与 follow 集合，以及如何求出文法的 LR(0)DFA 图，判断文法是否为 SLR(1) 文法，以及求出文法的 SLR(1) 分析表。不仅如此，我还完成了实验的选做部分——句子分析，能正确地对句子进行分析。这使我对编译原理中的 LR 分析法有了更为清晰的认识。

在实现过程中，我通过合理的数据结构设计和模块化的编程，将不同功能的代码分离，提高了代码的可维护性和扩展性。尤其LR(0)DFA图的构建和SLR(1)文法的判断等关键算法的实现，让我更熟练地运用了C++语言。

在项目中，我还强化了使用Qt框架来开发GUI应用程序的技能，能更轻松地开发基于qt的GUI程序。

5 参考文献

《Qt 学习之路 2》目录 - DevBean Tech World

编译原理教材

黄煜廉老师的ppt