北京郵電大學

实验报告：语法分析程序的设计与实现

——LR 分析方法

学院：计算机学院（国家示范性软件学院）

专业：计算机科学与技术

班级：2022211305

学号：2022211683

姓名：张晨阳

2024 年 11 月 21 号

目录

1.实验概述

1.1. 实验题目及要求

编写 LR 语法分析程序，实现对算术表达式的语法分析。要求所分析算数表达式由如下的文法产生。

$$E \rightarrow E+T \mid E-T \mid T$$

$$T \rightarrow T*F \mid T/F \mid F$$

$$F \rightarrow (E) \mid num$$

在对输入的算术表达式进行分析的过程中，依次输出所采用的产生式。

1. 构造识别该文法所有活前缀的 DFA。
2. 构造该文法的 LR 分析表。
3. 编程实现算法 4.3，构造 LR 分析程序。

1.2. 实验环境

- Windows 11
- Visual Studio Code
- C++17

2.程序设计说明

2.1. 程序结构设计概述

为了清晰地组织代码，本项目将各模块的功能划分到不同的文件中：

1. **Grammar.h 和 Grammar.cpp**

- **功能：**实现文法相关的功能，包括非终结符、终结符的管理，产生式的存储与操作，拓展文法的功能等。
- **主要内容：**
  - 存储文法的核心数据结构（非终结符、终结符、产生式、开始符号等）。
  - 提供文法操作接口，如添加产生式、拓展文法、计算产生式编号。

2. **DFA.h 和 DFA.cpp**

- **功能：**实现有限自动机的功能，包括闭包计算、状态转移、FIRST 集的计算等。
- **主要内容：**
  - 管理 DFA 的状态集合和状态转移。
  - 提供核心算法，如计算闭包和状态转移。

3. **LR1Parser.h 和 LR1Parser.cpp**

- **功能：**构造 LR(1) 分析器，包括 LR(1) 自动机和分析表的构造以及输入串的分析功能。
- **主要内容：**
  - 构造 LR(1) 自动机和分析表。
  - 使用分析表解析输入串，并提供详细的分析过程输出。

4. **主程序**

- **功能：**作为程序入口，创建文法、调用各模块进行 LR(1) 分析。

2.1.1.Grammar 文法类

功能概述

Grammar 类是文法相关操作的核心，负责存储文法信息并提供文法操作的功能。

其主要职责包括：

1. 存储文法的非终结符、终结符、产生式和开始符号。
2. 提供接口添加产生式并进行拓展文法操作。
3. 提供根据产生式编号查找产生式的方法。

类的设计

```
1. class Grammar {
2.    public:
3.     Grammar();
4.     void addProduction(const std::string& non_terminal, const
std::string& production);
5.     void expandGrammar();
6.     void printGrammar() const;
7.     std::string findProduction(const std::string& left, const
std::string& right) const;
8.
9.     std::set<std::string> nonTerminals;     // 非终结符集合
10.    std::set<std::string> terminals;        // 终结符集合
11.    std::map<std::string, std::vector<std::string>> productions;
12.    std::string startSymbol;                // 开始符号
13.    std::unordered_map<int, std::map<std::string, std::string>>
productionCount; // 产生式编号映射
14.    int count;                              // 产生式编号计数
15. };
```

主要方法

1. addProduction：
   - 用于添加产生式，并更新非终结符和终结符集合。
2. expandGrammar：
   - 拓展文法，添加新的开始符号及其产生式。
3. findProduction：
   - 根据产生式左部和右部查找对应的编号。

2.1.2.DFA 有限自动机

功能概述

DFA 类负责构建 LR(1) 自动机，其主要职责包括：

1. 管理 DFA 的状态集合和状态转移。
2. 提供闭包计算和状态转移功能。
3. 计算 FIRST 集，辅助 LR(1) 分析。

类的设计

```
1. class DFA {
2.   public:
3.     DFA();
4.std::string getFirst(const Grammar& grammar, const std::string& symbol);
5.     std::set<std::pair<std::pair<std::string, std::string>,
std::set<std::string>>>
6.     getClosure(const Grammar& grammar, const
std::set<std::pair<std::pair<std::string, std::string>,
std::set<std::string>>>& state);
7.     std::set<std::pair<std::pair<std::string, std::string>,
std::set<std::string>>>
8.     mergeStates(const std::set<std::pair<std::pair<std::string,
std::string>, std::set<std::string>>>& state);
9.     void addEdge(int start, int end, const std::string& symbol);
10.    void printState(int number) const;
11.    void showDFA() const;
12.    std::unordered_map<int, std::set<std::pair<std::pair<std::string,
std::string>, std::set<std::string>>>> states;
13.    std::unordered_map<int, std::unordered_map<int, std::string>>
transitions;
14.    int stateCount;    // 状态计数器，表示状态总数
15.   private:
16.bool isTerminal(const std::string& symbol,const Grammar& grammar) const;
17. };
```

主要方法

1. **getClosure**：计算 LR(1) 项的闭包。
2. **getFirst**：计算 FIRST 集。
3. **addEdge**：添加状态转移边。
4. **showDFA**：打印 DFA 的状态及其转移。

2.1.3.LR(1)分析器

功能概述

LR1Parser 类是程序的核心模块，负责：

1. 构造 LR(1) 自动机。
2. 构造 LR(1) 分析表。
3. 使用分析表解析输入串。

类的设计

```
class LR1Parser {
    public:
     LR1Parser();
     void computeFirst(Grammar& grammar);
     void computeLR1Automaton(Grammar& grammar, int number = 0);
     void computeAnalysisTable(Grammar& grammar);
     void parse(Grammar& grammar, const std::string& inputString);

    private:
     DFA dfa; // LR(1) 自动机
     std::unordered_map<int, std::unordered_map<std::string,
std::string>> analysisTable; // 分析表
};
```

主要方法

1. **computeFirst**：初始化 I0 状态。
2. **computeLR1Automaton**：构造 LR(1) 自动机。
3. **computeAnalysisTable**：构造 LR(1) 分析表。
4. **parse**：使用分析表解析输入串，并输出分析过程。

2.2. 程序具体逻辑说明

2.2.1.拓展文法

功能说明

拓展文法是 LR(1) 分析的第一步。通过为原始文法添加一个新的起始符号及其产生式，可以简化分析过程，统一起始点。

实现步骤

1. 创建一个新的起始符号，通常为原始起始符号后加一个单引号（如 E'）。
2. 为新起始符号添加一条产生式：$E' \rightarrow E$，其中 E 是原始文法的起始符号。
3. 将新起始符号设为当前文法的起始符号。
4. 更新产生式编号，方便后续规约时定位产生式。

关键代码

```
1. void Grammar::expandGrammar() {
2.     std::set<std::string> newNonTerminals;
3.     std::map<std::string, std::vector<std::string>> newProductions;
4.     std::string newStartSymbol = startSymbol + "'";
5.
6.     newNonTerminals.insert(newStartSymbol);
7.     newProductions[newStartSymbol] = {startSymbol};
8.     productionCount[0][newStartSymbol] = startSymbol;  // 编号为 0
9.     count = 1;                                          // 从 1 开始计数
10.
11.    // 第二步：保留原始输入顺序添加产生式
12.    for (const auto& non_terminal : nonTerminals) {
13.        newNonTerminals.insert(non_terminal);
14.
15.        if (productions.find(non_terminal) != productions.end()) {
16.            newProductions[non_terminal] = productions[non_terminal];
17.
18.            // 为每个产生式分配编号
19.            for (const auto& rightStr : productions[non_terminal]) {
20.                productionCount[count][non_terminal] = rightStr;
21.                count++;
22.            }
23.        }
24.    }
```

```
25.
26.     // 更新类成员
27.     nonTerminals = newNonTerminals;
28.     productions = newProductions;
29.     startSymbol = newStartSymbol;
30. }
```

2.2.2.计算 FIRST 集

功能说明

FIRST 集是 LR(1) 分析的核心部分之一，用于确定某个符号串可能生成的第一个终结符集合。

实现步骤

1. 如果符号是终结符，FIRST 集直接为该符号。
2. 如果符号是非终结符，FIRST 集为其所有产生式右部第一个符号的 FIRST 集。
3. 如果右部的第一个符号能够生成空（ε），则继续处理右部的下一个符号，直到遇到非空符号或右部结束。

关键代码

```
1. std::string DFA::getFirst(const Grammar& grammar, const std::string&
symbol) {
2.     if (isTerminal(symbol, grammar) || symbol == "$") {
3.         return symbol;  // 如果是终结符或结束符号，直接返回
4.     }
5.
6.     auto productions = grammar.productions.find(symbol);
7.
8.     if (productions != grammar.productions.end())
9.         for (const auto& right : productions->second) {
10.            if (isTerminal(std::string(1, right[0]), grammar))
11.                return std::string(1, right[0]);
12.            else if (right == "num")
13.                return "num";
14.            else if (right[0] == symbol[0])
15.                continue;
16.            else if (right[0] != symbol[0])
17.                return getFirst(grammar, std::string(1, right[0]));
18.        }
19.
20.    return "";
21. }
```

2.2.3.计算闭包 closure

功能说明

闭包是构造 LR(1) 自动机的关键步骤。给定一个 LR(1) 项集合，通过扩展非终结符的产生式，可以生成该集合的闭包。

实现步骤

1. 遍历当前集合中的每个项。
2. 如果某项的右部中，• 后是非终结符：添加该非终结符的所有产生式，且 lookahead 为当前项的 FIRST 集。
3. 循环执行上述步骤，直到集合不再增加新项。

关键代码

```
 1. std::set<std::pair<std::pair<std::string, std::string>,
std::set<std::string>>>
 2. DFA::getClosure(const Grammar& grammar, const
std::set<std::pair<std::pair<std::string, std::string>,
std::set<std::string>>>& state) {
 3.     auto closure = state;
 4.
 5.     bool addedNewItem = true;  // 标志位，用于检测是否有新项目被添加
 6.
 7.     while (addedNewItem) {
 8.         addedNewItem = false;
 9.
10.         std::set<std::pair<std::pair<std::string, std::string>,
std::set<std::string>>> newItems;
11.
12.         for (const auto& item : closure) {
13.             const auto& left = item.first.first;        // 获取产生式左部
14.             const auto& right_str = item.first.second;  // 获取产生式右部
15.             const auto& lookahead = item.second;
16.
17.             for (const auto& currentLookahead : lookahead) {
18.                 for (size_t i = 0; i < right_str.size(); ++i) {
19.                     if (right_str[i] == '@' && i + 1 < right_str.size())
{
20.                         std::string nextSymbol(1, right_str[i + 1]);
21.
```

```
22.                         if (grammar.nonTerminals.find(nextSymbol) !=
grammar.nonTerminals.end()) {
23.                             for (const auto& production :
grammar.productions.at(nextSymbol)) {
24.                                 std::string strAhead;
25.
26.                                 if (i + 1 == right_str.size() - 1)
27.                                     strAhead = currentLookahead;
28.                                 else
29.                                     strAhead = right_str.substr(i + 2) +
currentLookahead;
30.
31.                                 std::set<std::string> firstSet;
32.
33.                                 std::string symbol(1, strAhead[0]);
34.                                 std::string first = getFirst(grammar,
symbol);
35.                                 if (!first.empty())
36.                                     firstSet.insert(first);
37.
38.                                 std::pair<std::pair<std::string,
std::string>, std::set<std::string>> newItem =
39.                                     std::make_pair(std::make_pair(nextSym
bol, "@" + production), firstSet);
40.
41.                                 if (closure.find(newItem) ==
closure.end() && newItems.find(newItem) == newItems.end()) {
42.                                     newItems.insert(newItem);
43.                                     addedNewItem = true;  // 有新项目加入
44.                                 }
45.                             }
46.                         }
47.                     }
48.                 }
49.             }
50.         }
51.
52.         closure.insert(newItems.begin(), newItems.end());
53.     }
54.
55.     return closure;
56. }
```

2.2.4.构造 LR(1)自动机

功能说明

LR(1) 自动机的构造是分析表生成的基础。通过遍历所有状态，基于每个状态的项目集计算闭包和转移关系，可以生成完整的 LR(1) 自动机。

实现步骤

1. **初始化 I0 状态**：从文法的扩展开始，初始化状态 I0，计算其闭包，并将其作为自动机的起点。
2. **状态分类**：对于每个状态，根据 • 后的符号对项目进行分类。每类符号生成一个新的项目集。
3. **生成新状态**：对分类后的项目集，计算其闭包，并合并同类项。检查新状态是否已存在：1. 如果存在，仅添加转移边。2. 如果不存在，将其添加为新状态并记录转移。
4. **递归处理所有状态**：依次处理状态集中的每个状态，直到所有可能的状态都被计算完成。

关键代码解析

1. **初始化 I0 状态**

computeFirst(grammar) 用于初始化 I0：

```
if (number == 0) {
    computeFirst(grammar);  // 初始化 I0 状态
}
```

computeFirst 生成初始状态：$E' -> \cdot E\ ,\$$

2. **项目分类**

```
for (const auto& item : state) {
    const auto& right = item.first.second;
    size_t dotIndex = right.find('@');

    if (dotIndex != std::string::npos && dotIndex + 1 < right.size()) {
        std::string key(1, right[dotIndex + 1]);
        if (key == "n" && right.substr(dotIndex + 1) == "num") {
            key = "num";
        }
        classified[key].insert(item);
```

```
    }
}
```

- 对于当前状态的项目，找到 · 后面的符号。
- 如果是终结符或非终结符，将项目按符号分类存储在 classified 中。

3. 生成新状态

```
auto newState = std::set<std::pair<std::pair<std::string, std::string>, std::set<std::string>>>();

for (const auto& tpl : subset) {
    const auto& left = tpl.first.first;
    const auto& right_str = tpl.first.second;
    auto lookahead = tpl.second;

    size_t dotIndex = right_str.find('@');
    std::string newRightStr;

    if (dotIndex != std::string::npos && dotIndex + 1 < right_str.size()) {
        if (right_str.substr(dotIndex + 1) == "num") {
            newRightStr = "num@";
        } else {
            newRightStr = right_str.substr(0, dotIndex) + right_str[dotIndex + 1] + "@" + right_str.substr(dotIndex + 2);
        }
    }

    newState.insert({{left, newRightStr}, lookahead});
}
```

- 遍历分类后的项目集，将 • 向右移动生成新项目。
- 将新项目集合作为新状态的候选。

4. 计算闭包并合并

```
newState = dfa.getClosure(grammar, newState);
newState = dfa.mergeStates(newState);
```

- 通过 getClosure 计算新状态的闭包。
- 使用 mergeStates 合并相同左部的项目，优化状态结构。

5. 检查新状态是否已存在

```
bool exists = false;
int existingState = -1;

for (const auto& [key, value] : dfa.states) {
    if (value == newState) {
        exists = true;
        existingState = key;
        break;
    }
}

if (!exists) {
    dfa.states[dfa.stateCount] = newState;        // 添加新状态
    dfa.addEdge(number, dfa.stateCount, symbol);  // 添加状态转移边
    dfa.stateCount++;
} else {
    dfa.addEdge(number, existingState, symbol);  // 添加已有状态的转移边
}
```

- 如果新状态已经存在，则仅添加转移关系。
- 如果新状态不存在，则将其作为新状态加入，并更新状态计数器。

6. 递归处理下一个状态

```
computeLR1Automaton(grammar, number + 1);
```

- 递归处理当前状态的下一个状态，直到所有状态都被计算完成。

2.2.5.构造 LR(1) 分析表

功能说明

LR(1) 分析表是 LR(1) 语法分析的核心数据结构，用于指导分析器的操作。分析表分为两部分：

1. **ACTION 表**：定义在某个状态下遇到某个终结符时的操作（如 Shift, Reduce, Accept, 或 Error）。
2. **GOTO 表**：定义在某个状态下遇到某个非终结符时的转移。

实现步骤

1. **初始化分析表：**

- 对每个状态，将所有终结符的表项初始化为未定义值（如 ""），结束符 $ 的表项同样初始化。
- 对每个状态，将所有非终结符的表项初始化为未定义值。

2. **遍历每个状态的项目集：**

- 如果项目为规约项（right.back() == '@'）：
  - 如果是扩展文法的起始符号（如 $E' \to E \cdot$），设置 action 为 ACC（接受）。
  - 否则，添加 Reduce 操作。
- 根据状态的转移关系，添加 Shift 和 GOTO 操作。

3. **输出分析表**：打印分析表内容，包括每个状态对终结符的 ACTION 和非终结符的 GOTO。

关键代码解析

1. 初始化分析表

```
1. for (int i = 0; i < dfa.stateCount; ++i) {
2.     for (const auto& terminal : grammar.terminals) {
3.         analysisTable[i][terminal] = "";  // 初始化终结符表项
4.     }
5.     analysisTable[i]["$"] = "";          // 初始化结束符表项
6.     for (const auto& nonTerminal : grammar.nonTerminals) {
7.         analysisTable[i][nonTerminal] = "";  // 初始化非终结符表项
8.     }
9. }
```

- 对每个状态初始化所有符号的表项，确保未定义的表项以空字符串表示。

2. 填充 ACTION 表和 GOTO 表

- **规约项：**

```
if (right.back() == '@') {  // 如果是规约项
    if (left == grammar.startSymbol) {
        analysisTable[i]["$"] = "ACC";  // 接受状态
    } else {
        for (const auto& lookahead : item.second) {
            std::string prodNum = grammar.findProduction(left,
right.substr(0, right.size() - 1));
            analysisTable[i][lookahead] = "R" + prodNum;  // Reduce 操作
        }
    }
}
```

如果项目是扩展文法的结束项（如$E' \to E \cdot$），设置 ACC。

对其他规约项，查找产生式编号并设置 Reduce 操作。

- **状态转移（Shift 和 GOTO）：**

```
if (dfa.transitions.find(i) != dfa.transitions.end()) {
    for (const auto& [neighbor, transSymbol] : dfa.transitions[i]) {
        if (grammar.terminals.count(transSymbol)) {
           analysisTable[i][transSymbol] = "S"+std::to_string(neighbor);
        } else if (grammar.nonTerminals.count(transSymbol)) {
            analysisTable[i][transSymbol] = std::to_string(neighbor);
        }
    }
}
```

对于终结符转移，设置 Shift 操作。

对于非终结符转移，设置 GOTO 操作。

2.2.6.进行 LR(1)分析

功能说明

LR(1) 分析过程是通过分析表解析输入串的核心步骤。通过栈和分析表的配合，逐步完成对输入串的语法分析，并输出具体的分析过程。

实现步骤

1. 初始化：初始化栈，将初始状态 0 和起始符号（如 $）压入栈。将输入串加上结束符 $。

2. 分析循环：根据栈顶状态和当前输入符号，从分析表中获取对应的操作（ACTION 或 GOTO）。根据操作执行：

- **Shift**：将输入符号和目标状态压入栈。移动输入指针。
- **Reduce**：弹出栈顶若干符号（产生式右部的长度）。压入产生式左部，并根据 GOTO 表更新状态。
- **Accept**：如果操作为 ACC，表示分析成功。
- **Error**：如果无效操作（如空值），表示分析失败。

3. 输出分析过程：每个步骤打印当前栈状态、输入串剩余部分，以及执行的操作（Shift，Reduce，Accept，或 Error）。

关键代码

```
1. void LR1Parser::parse(Grammar& grammar,const std::string& inputString){
2.     std::vector<std::pair<int, std::string>> stack = {{0, "$"}};
3.     std::string input = inputString + "$";
4.     size_t ptr = 0;
5.
6.     std::cout << std::right << std::setw(23) << "Stack (States)"<< "\t"
7.               << std::setw(33) << "Input" << "\tAction" << std::endl;
8.
9.     while (true) {
10.        std::string stateStack, symbolStack;
11.        for (const auto& s : stack) {
12.            stateStack += std::to_string(s.first) + " ";
13.            symbolStack += s.second + " ";
14.        }
15.
16.        std::string symbol(1, input[ptr]);
17.        if (std::isdigit(input[ptr]))
```

```
18.             symbol = "num";
19.
20.         int state = stack.back().first;
21.         const std::string& action = analysisTable[state][symbol];
22.
23.         std::cout << "States:  " << stateStack << "\n";
24.         std::cout<<"Symbols: "<<std::left<<std::setw(28) << symbolStack
25.                  << std::right << std::setw(20) << input.substr(ptr)
26.                  << "\t" << action << std::endl;
27.
28.         if (action.empty()) {
29.             std::cout << "Error! Current state: " << state << ", symbol:
" << symbol << std::endl;
30.             break;
31.         }
32.
33.         if (action == "ACC") {
34.             std::cout << "Accepted!" << std::endl;
35.             break;
36.         }
37.
38.         if (action[0] == 'S') {
39.             int nextState = std::stoi(action.substr(1));  // 获取目标状态
40.             stack.emplace_back(nextState, symbol);        // 压入栈
41.             ptr++;                                       // 输入指针右移
42.         }
43.
44.         else if (action[0] == 'R') {
45.             int prodNum = std::stoi(action.substr(1));  // 获取产生式编号
46.             const auto& prod = grammar.productionCount[prodNum];
47.             const std::string& left = prod.begin()->first;
48.             const std::string& right = prod.begin()->second;
49.
50.             size_t popCount = (right == "num") ? 1 : right.size();
51.             for (size_t i = 0; i < popCount; ++i)
52.                 stack.pop_back();
53.
54.             state = stack.back().first;  // 获取当前栈顶状态
55.         stack.emplace_back(std::stoi(analysisTable[state][left]),left);
56.         }
57.     }
58. }
```

3.测试设计与分析

3.1. 测试方法及辅助信息说明

1. 确认原始输入文法 + 测试案例

在 main.cpp 文件中设置：

```
Grammar grammar;
grammar.addProduction("E", "E+T");
grammar.addProduction("E", "E-T");
grammar.addProduction("E", "T");
grammar.addProduction("T", "T*F");
grammar.addProduction("T", "T/F");
grammar.addProduction("T", "F");
grammar.addProduction("F", "(E)");
grammar.addProduction("F", "num");
grammar.startSymbol = "E";

std::vector<std::string> testInputs = {
    "1+2",            // 正常的输入串
    "1+2*(3-(4/5))",  // 含括号的复杂表达式
    "1+2*/3",         // 含错误的输入串
    "1+(2*9)+3)"      // 含未匹配括号的输入串
};
```

2. 编译生成可执行文件并执行

```
> g++ -o main Grammar.cpp DFA.cpp LR1Parser.cpp main.cpp
> ./main
```

3. 拓广文法

输出如下：

```
Original grammar:
E -> E+T | E-T | T
F -> (E) | num
T -> T*F | T/F | F

Expanded grammar:
0 : E' -> E
1 : E -> E+T
2 : E -> E-T
3 : E -> T
```

```
4 : F -> (E)
5 : F -> num
6 : T -> T*F
7 : T -> T/F
8 : T -> F
```

4. 构建 LR(1)项目集规范簇

结果如下：

```
LR(1) DFA:
I0:
E -> @E+T , $ + -
E -> @E-T , $ + -
E -> @T , $ + -
E' -> @E , $
F -> @(E) , $ * + - /
F -> @num , $ * + - /
T -> @F , $ * + - /
T -> @T*F , $ * + - /
T -> @T/F , $ * + - /
--- T ---> I5
--- ( ---> I4
--- E ---> I3
--- F ---> I1
--- num ---> I2
I1:
T -> F@ , $ * + - /
I2:
F -> num@ , $ * + - /
I3:
E -> E@+T , $ + -
E -> E@-T , $ + -
E' -> E@ , $
--- + ---> I7
--- - ---> I6
I4:
E -> @E+T , ) + -
E -> @E-T , ) + -
E -> @T , ) + -
F -> (@E) , $ * + - /
F -> @(E) , ) * + - /
F -> @num , ) * + - /
T -> @F , ) * + - /
T -> @T*F , ) * + - /
```

```
T -> @T/F , ) * + - /
--- T ---> I12
--- ( ---> I11
--- E ---> I10
--- F ---> I8
--- num ---> I9
I5:
E -> T@ , $ + -
T -> T@*F , $ * + - /
T -> T@/F , $ * + - /
--- * ---> I14
--- / ---> I13
I6:
E -> E-@T , $ + -
F -> @(E) , $ * + - /
F -> @num , $ * + - /
T -> @F , $ * + - /
T -> @T*F , $ * + - /
T -> @T/F , $ * + - /
--- ( ---> I4
--- T ---> I15
--- F ---> I1
--- num ---> I2
I7:
E -> E+@T , $ + -
F -> @(E) , $ * + - /
F -> @num , $ * + - /
T -> @F , $ * + - /
T -> @T*F , $ * + - /
T -> @T/F , $ * + - /
--- ( ---> I4
--- F ---> I1
--- T ---> I16
--- num ---> I2
I8:
T -> F@ , ) * + - /
I9:
F -> num@ , ) * + - /
I10:
E -> E@+T , ) + -
E -> E@-T , ) + -
F -> (E@) , $ * + - /
--- - ---> I19
--- ) ---> I17
```

```
--- + ---> I18
I11:
E -> @E+T , ) + -
E -> @E-T , ) + -
E -> @T , ) + -
F -> (@E) , ) * + - /
F -> @(E) , ) * + - /
F -> @num , ) * + - /
T -> @F , ) * + - /
T -> @T*F , ) * + - /
T -> @T/F , ) * + - /
--- T ---> I12
--- ( ---> I11
--- E ---> I20
--- F ---> I8
--- num ---> I9
I12:
E -> T@ , ) + -
T -> T@*F , ) * + - /
T -> T@/F , ) * + - /
--- * ---> I22
--- / ---> I21
I13:
F -> @(E) , $ * + - /
F -> @num , $ * + - /
T -> T/@F , $ * + - /
--- num ---> I2
--- F ---> I23
--- ( ---> I4
I14:
F -> @(E) , $ * + - /
F -> @num , $ * + - /
T -> T*@F , $ * + - /
--- num ---> I2
--- F ---> I24
--- ( ---> I4
I15:
E -> E-T@ , $ + -
T -> T@*F , $ * + - /
T -> T@/F , $ * + - /
--- * ---> I14
--- / ---> I13
I16:
E -> E+T@ , $ + -
```

```
T -> T@*F , $ * + - /
T -> T@/F , $ * + - /
--- * ---> I14
--- / ---> I13
I17:
F -> (E)@ , $ * + - /
I18:
E -> E+@T , ) + -
F -> @(E) , ) * + - /
F -> @num , ) * + - /
T -> @F , ) * + - /
T -> @T*F , ) * + - /
T -> @T/F , ) * + - /
--- ( ---> I11
--- T ---> I25
--- F ---> I8
--- num ---> I9
I19:
E -> E-@T , ) + -
F -> @(E) , ) * + - /
F -> @num , ) * + - /
T -> @F , ) * + - /
T -> @T*F , ) * + - /
T -> @T/F , ) * + - /
--- ( ---> I11
--- T ---> I26
--- F ---> I8
--- num ---> I9
I20:
E -> E@+T , ) + -
E -> E@-T , ) + -
F -> (E@) , ) * + - /
--- - ---> I19
--- ) ---> I27
--- + ---> I18
I21:
F -> @(E) , ) * + - /
F -> @num , ) * + - /
T -> T/@F , ) * + - /
--- num ---> I9
--- F ---> I28
--- ( ---> I11
I22:
F -> @(E) , ) * + - /
```

```
F -> @num , ) * + - /
T -> T*@F , ) * + - /
--- num ---> I9
--- F ---> I29
--- ( ---> I11
I23:
T -> T/F@ , $ * + - /
I24:
T -> T*F@ , $ * + - /
I25:
E -> E+T@ , ) + -
T -> T@*F , ) * + - /
T -> T@/F , ) * + - /
--- * ---> I22
--- / ---> I21
I26:
E -> E-T@ , ) + -
T -> T@*F , ) * + - /
T -> T@/F , ) * + - /
--- * ---> I22
--- / ---> I21
I27:
F -> (E)@ , ) * + - /
I28:
T -> T/F@ , ) * + - /
I29:
T -> T*F@ , ) * + - /
```

5. 构造 LR(1)分析表

输出如下：

LR(1) analysis table:

Analysis Table:

State	(	)	*	+	-	/	num	$	E	F	T
0	S4						S2		3	1	5
1			R8	R8	R8	R8		R8			
2			R5	R5	R5	R5		R5			
3				S7	S6			ACC			
4	S11						S9		10	8	12
5			S14	R3	R3	S13		R3			
6	S4						S2			1	15
7	S4						S2			1	16
8		R8	R8	R8	R8	R8					
9		R5	R5	R5	R5	R5					
10		S17		S18	S19						
11	S11						S9		20	8	12
12		R3	S22	R3	R3	S21					
13	S4						S2			23	
14	S4						S2			24	
15			S14	R2	R2	S13		R2			
16			S14	R1	R1	S13		R1			
17			R4	R4	R4	R4		R4			
18	S11						S9			8	25
19	S11						S9			8	26
20		S27		S18	S19						
21	S11						S9			28	
22	S11						S9			29	
23			R7	R7	R7	R7		R7			
24			R6	R6	R6	R6		R6			
25		R1	S22	R1	R1	S21					
26		R2	S22	R2	R2	S21					
27		R4	R4	R4	R4	R4					
28		R7	R7	R7	R7	R7					
29		R6	R6	R6	R6	R6					

3.2. 测试 1+2

测试目的

验证程序是否能够正确解析简单的加法表达式，且正确进行移进和规约操作。

测试预期

输入表达式 1+2，程序应通过分析表成功完成移进、规约，最终接受该输入。

测试输出

```
Testing input: 1+2
        Stack (States)                          Input       Action
States:  0
Symbols: $                                      1+2$        S2
States:  0 2
Symbols: $ num                                   +2$        R5
States:  0 1
Symbols: $ F                                     +2$        R8
States:  0 5
Symbols: $ T                                     +2$        R3
States:  0 3
Symbols: $ E                                     +2$        S7
States:  0 3 7
Symbols: $ E +                                    2$        S2
States:  0 3 7 2
Symbols: $ E + num                                $         R5
States:  0 3 7 1
Symbols: $ E + F                                  $         R8
States:  0 3 7 16
Symbols: $ E + T                                  $         R1
States:  0 3
Symbols: $ E                                      $         ACC
Accepted!
```

结果分析

程序成功识别并解析表达式 1+2。移进、规约及接受步骤清晰准确，分析表与分析过程完全匹配。

3.3. 测试 1+2*(3-(4/5))

测试目的

验证程序能否处理嵌套表达式，包括括号、加减法和乘除法的优先级，以及正确解析深层嵌套的表达式。

测试预期

输入表达式 1+2*(3-(4/5))，程序应按优先级依次完成括号中的表达式解析，并最终接受该输入。

测试输出

```
Testing input: 1+2*(3-(4/5))
        Stack (States)                          Input       Action
States:  0
Symbols: $                                  1+2*(3-(4/5))$      S2
States:  0 2
Symbols: $ num                               +2*(3-(4/5))$      R5
States:  0 1
Symbols: $ F                                 +2*(3-(4/5))$      R8
States:  0 5
Symbols: $ T                                 +2*(3-(4/5))$      R3
States:  0 3
Symbols: $ E                                 +2*(3-(4/5))$      S7
States:  0 3 7
Symbols: $ E +                                2*(3-(4/5))$      S2
States:  0 3 7 2
Symbols: $ E + num                              *(3-(4/5))$      R5
States:  0 3 7 1
Symbols: $ E + F                                *(3-(4/5))$      R8
States:  0 3 7 16
Symbols: $ E + T                                *(3-(4/5))$      S14
States:  0 3 7 16 14
Symbols: $ E + T *                               (3-(4/5))$      S4
States:  0 3 7 16 14 4
Symbols: $ E + T * (                              3-(4/5))$      S9
States:  0 3 7 16 14 4 9
Symbols: $ E + T * ( num                            -(4/5))$      R5
States:  0 3 7 16 14 4 8
Symbols: $ E + T * ( F                              -(4/5))$      R8
States:  0 3 7 16 14 4 12
Symbols: $ E + T * ( T                              -(4/5))$      R3
```

```
States:  0 3 7 16 14 4 10
Symbols: $ E + T * ( E                          -(4/5))$       S19
States:  0 3 7 16 14 4 10 19
Symbols: $ E + T * ( E -                         (4/5))$       S11
States:  0 3 7 16 14 4 10 19 11
Symbols: $ E + T * ( E - (                        4/5))$       S9
States:  0 3 7 16 14 4 10 19 11 9
Symbols: $ E + T * ( E - ( num                     /5))$       R5
States:  0 3 7 16 14 4 10 19 11 8
Symbols: $ E + T * ( E - ( F                       /5))$       R8
States:  0 3 7 16 14 4 10 19 11 12
Symbols: $ E + T * ( E - ( T                       /5))$       S21
States:  0 3 7 16 14 4 10 19 11 12 21
Symbols: $ E + T * ( E - ( T /                      5))$       S9
States:  0 3 7 16 14 4 10 19 11 12 21 9
Symbols: $ E + T * ( E - ( T / num                   ))$       R5
States:  0 3 7 16 14 4 10 19 11 12 21 28
Symbols: $ E + T * ( E - ( T / F                     ))$       R7
States:  0 3 7 16 14 4 10 19 11 12
Symbols: $ E + T * ( E - ( T                         ))$       R3
States:  0 3 7 16 14 4 10 19 11 20
Symbols: $ E + T * ( E - ( E                         ))$       S27
States:  0 3 7 16 14 4 10 19 11 20 27
Symbols: $ E + T * ( E - ( E )                        )$       R4
States:  0 3 7 16 14 4 10 19 8
Symbols: $ E + T * ( E - F                            )$       R8
States:  0 3 7 16 14 4 10 19 26
Symbols: $ E + T * ( E - T                            )$       R2
States:  0 3 7 16 14 4 10
Symbols: $ E + T * ( E                                )$       S17
States:  0 3 7 16 14 4 10 17
Symbols: $ E + T * ( E )                               $       R4
States:  0 3 7 16 14 24
Symbols: $ E + T * F                                   $       R6
States:  0 3 7 16
Symbols: $ E + T                                       $       R1
States:  0 3
Symbols: $ E                                           $       ACC
Accepted!
```

结果分析

程序成功解析复杂嵌套表达式，按照括号优先级解析子表达式后完成整体分析。移进和规约操作均正确，最终状态成功匹配接受状态。

3.4. 测试 1+2*/3

测试目的

验证程序能否检测非法输入并及时报错，尤其是操作符 */ 的语法错误。

测试预期

输入表达式 1+2*/3，程序应在检测到 */ 时，根据分析表判断为非法表达式，并输出错误信息。

测试输出

```
Testing input: 1+2*/3
        Stack (States)                          Input       Action
States:  0
Symbols: $                                    1+2*/3$       S2
States:  0 2
Symbols: $ num                                 +2*/3$       R5
States:  0 1
Symbols: $ F                                   +2*/3$       R8
States:  0 5
Symbols: $ T                                   +2*/3$       R3
States:  0 3
Symbols: $ E                                   +2*/3$       S7
States:  0 3 7
Symbols: $ E +                                  2*/3$       S2
States:  0 3 7 2
Symbols: $ E + num                               */3$       R5
States:  0 3 7 1
Symbols: $ E + F                                 */3$       R8
States:  0 3 7 16
Symbols: $ E + T                                 */3$       S14
States:  0 3 7 16 14
Symbols: $ E + T *                                /3$
Error! Current state: 14, symbol: /
```

结果分析

程序正确检测到非法操作符 */ 并中止分析，输出错误信息，定位问题位置和状态，符合预期。

3.5. 测试 1+(2*9)+3)

测试目的

验证程序能否检测括号匹配错误，并在遇到多余右括号时及时报错。

测试预期

输入表达式 1+(2*9)+3)，程序应在分析到多余的右括号时，根据分析表输出错误信息。

测试输出

```
Testing input: 1+(2*9)+3)
        Stack (States)                          Input       Action
States:  0
Symbols: $                                   1+(2*9)+3)$      S2
States:  0 2
Symbols: $ num                                +(2*9)+3)$      R5
States:  0 1
Symbols: $ F                                  +(2*9)+3)$      R8
States:  0 5
Symbols: $ T                                  +(2*9)+3)$      R3
States:  0 3
Symbols: $ E                                  +(2*9)+3)$      S7
States:  0 3 7
Symbols: $ E +                                 (2*9)+3)$      S4
States:  0 3 7 4
Symbols: $ E + (                                2*9)+3)$      S9
States:  0 3 7 4 9
Symbols: $ E + ( num                             *9)+3)$      R5
States:  0 3 7 4 8
Symbols: $ E + ( F                               *9)+3)$      R8
States:  0 3 7 4 12
Symbols: $ E + ( T                               *9)+3)$      S22
States:  0 3 7 4 12 22
Symbols: $ E + ( T *                              9)+3)$      S9
States:  0 3 7 4 12 22 9
Symbols: $ E + ( T * num                           )+3)$      R5
States:  0 3 7 4 12 22 29
Symbols: $ E + ( T * F                             )+3)$      R6
States:  0 3 7 4 12
Symbols: $ E + ( T                                 )+3)$      R3
States:  0 3 7 4 10
Symbols: $ E + ( E                                 )+3)$      S17
```

```
States:  0 3 7 4 10 17
Symbols: $ E + ( E )                                  +3)$       R4
States:  0 3 7 1
Symbols: $ E + F                                      +3)$       R8
States:  0 3 7 16
Symbols: $ E + T                                      +3)$       R1
States:  0 3
Symbols: $ E                                         +3)$       S7
States:  0 3 7
Symbols: $ E +                                        3)$       S2
States:  0 3 7 2
Symbols: $ E + num                                     )$
Error! Current state: 2, symbol: )
```

结果分析

程序检测到括号不匹配（多余的右括号），及时中止分析并输出错误信息。错误定位清晰，符合预期。

4.总结心得

通过本次 LR(1) 语法分析器的设计与实现，我对编译原理中语法分析部分，尤其是 LR(1) 分析的核心思想和实现细节，有了更加全面和深入的理解。这次实验让我深刻体会到了从理论到实践的复杂性与成就感。

本次实验的核心内容包括文法的拓展、FIRST 集的计算、闭包与状态转移的构造、分析表的生成，以及最终的 LR(1) 语法分析器的实现。每个步骤都相辅相成，为最终实现一个功能完整的 LR(1) 分析器奠定了坚实基础。

实验的第一步是对文法进行拓展，添加新的起始符号并更新产生式编号。这一步让我认识到文法设计中初始状态的一致性对于分析器实现的重要性。

随后在计算 FIRST 集和闭包时，我通过递归和集合操作，解决了符号的多重依赖问题，加深了对自动机构造中递归思想的理解。

构造 LR(1) 自动机和分析表是实验的难点，也是亮点。通过对状态集的分类与闭包操作，我成功地构造了一个完整的 LR(1) 自动机，并在分析表中清晰地定义了每个状态下的移进、规约和转移关系。这一步让我感受到编译原理中自动机与表驱动解析的巧妙结合。

调试过程中，我遇到了诸如状态转移错误、FIRST 集计算不完整、分析表填充不当等问题。这些问题虽然增加了实验的难度，但通过逐一分析与解决，我更加熟悉了 LR(1) 分析器的运行机制，也进一步锻炼了自己的调试能力和代码优化能力。

总的来说，这次实验不仅帮助我掌握了 LR(1) 分析器的设计与实现，更让我深刻理解了语法分析在编译器设计中的关键地位。这次实验为我未来进一步学习和实践编译器技术奠定了坚实的基础。