编译原理与技术实验二：语法分析程序的设计与实现（LR 分析方法）
实验报告

张梓良　2021212484

日期：2023 年 11 月 15 日

1　概述

1.1　实验内容及要求

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。

$$E \to E + T \mid E - T \mid T$$

$$T \to T * F \mid T/F \mid F$$

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

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。

1.2　实验环境

- cmake version 3.27.0-rc4
- gcc version 8.1.0
- Visual Studio Code 1.82.2
- OS: Windows_NT x64 10.0.22621

1.3　实验目的

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。

2　程序设计说明

该部分具体介绍了程序设计的思路，对于一些较为简单的函数设计直接给出了代码，其余较为复杂的函数设计给出了相应算法伪代码。

2.1 模块划分

语法分析程序分为三个模块：`grammar`，`table`，`main`，其中 `grammar` 模块定义了构造识别该文法所有活前缀的 DFA 的相关操作，`table` 模块定义了构建 LR(1) 文法分析表的操作，`main` 模块定义了 LR 分析控制程序并调用前两个模块以实现对输入的字符串分析。模块间的调用关系如 **图 1** 。

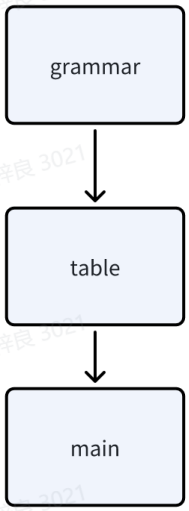

图 1: 模块调用关系

2.2 文法模块

grammar 模块定义了 Grammar 类：

```
class Grammar
{
public:
    Grammar() = default;

    /**
     * @brief 从文件流中读取文法
     * @param fin 文件流f
     */
    void GetGrammar(ifstream &fin);

    /**
     * @brief 构造拓广文法
     */
    void ExtendGrammar();

    /**
     * @brief 构造非终结符的FIRST集
     * @param nonterminal 非终结符
```

```
20      */
21     void BuildFirst(const string &noterminal);
22
23     /**
24      * @brief 构造所有非终结符的FIRST集
25      */
26     void First();
27
28     /**
29      * @brief 构造LR(1)项目集的闭包
30      * @param lr1s
31      */
32     void Closure(LR1s &lr1s);
33
34     /**
35      * @brief 构造LR(1)项目集的转移
36      * @param lr1s
37      * @param symbol
38      * @param newLR1s
39      */
40     void Go(const LR1s &lr1s, const string &symbol, LR1s &newLR1s);
41
42     /**
43      * @brief 构造识别该文法所有活前缀的DFA
44      */
45     void LivePreDFA();
46
47     /**
48      * @brief 判断是否为LR(1)文法
49      * @return true
50      * @return false
51      */
52     bool IsLR1();
53
54     unordered_set<string> nonterminals;                                  // 非终结
           符号集
55     unordered_set<string> terminals;                                     // 终结符
           号集
56     unordered_map<string, vector<vector<string>>> productions;           // 产生式
           集
57     string startSymbol;                                                  // 起始符
           号
58     unordered_map<string, unordered_set<string>> first;                  // 非终结
           符号的FIRST集
59     unordered_map<string, vector<unordered_set<string>>> candidateFirst; // 候选式
           的FIRST集
60     unordered_map<string, bool> firstDoned;                              // FIRST集
```

```
        是否已经构造完成
61      unordered_map<string, unordered_map<string, bool>> firstRelation;     // FIRST集
        的关系
62      vector<LR1s> DFAStates;                                               // DFA状
        态集/LR(1)项目集规范族
63      unordered_map<int, unordered_map<string, int>> DFAStateTrans;         // DFA状态
        转移
64  };
```

同时定义了以下数据结构：

```
1   // 自定义LR(1)项目的哈希函数
2   struct LR1sHash
3   {
4       size_t operator()(const pair<pair<string, vector<string>>, pair<int, string>> &
            p) const
5       {
6           // 计算第一部分的哈希值
7           size_t hash1 = hash<string>{}(p.first.first);
8           for (const auto &str : p.first.second)
9           {
10              hash1 ^= hash<string>{}(str);
11          }
12
13          // 计算第二部分的哈希值
14          size_t hash2 = hash<int>{}(p.second.first) ^ (hash<string>{}(p.second.
                second));
15
16          // 合并哈希值
17          return hash1 ^ (hash2 << 1);
18      }
19  };
20
21  typedef pair<pair<string, vector<string>>, pair<int, string>> LR1; // LR(1)项目
22  typedef unordered_set<LR1, LR1sHash> LR1s;                          // LR(1)项目集
```

Grammar 类实现了读入给定文法，构造拓广文法，构建非终结符的 FIRST 集，构造 LR(1) 项目集的闭包，构造 LR(1) 项目集的转移，构造识别该文法所有活前缀的 DFA 以及判断是否为 LR(1) 文法等方法。

具体处理流程如图2。

2.2.1 读入文法

通过方法 `void Grammar::GetGrammar(ifstream &fin)` 从输入文件流中读取文法。该函数依次读入：非终结符、终结符和文法规则。

函数首先读取输入文件流的第一行，并检查它是否与字符串“====Nonterminal Symbols====”匹配。如果匹配，函数进入一个循环，读取非终结符，直到遇到字符串 “======Terminal Sym-

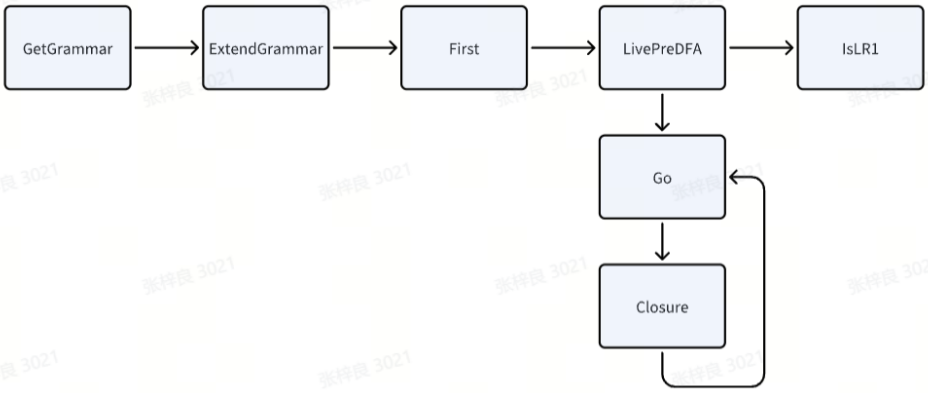

图 2: 处理流程图

bols=====”。遇到的第一个非终结符被指定为起始符号。

读取非终结符后，函数进入一个循环，读取终结符，直到遇到字符串 “=======Grammar Rules=======”。

最后，函数读取文法规则，直到到达输入文件流的末尾。

文法的存储格式如下：

```
====Nonterminal Symbols====
E T F
======Terminal Symbols=====
+ - * / ( ) num
=======Grammar Rules=======
E -> E + T | E - T | T
T -> T * F | T / F | F
F -> ( E ) | num
```

2.2.2 构造拓广文法

该算法接受一个上下文无关文法 $\mathbf{G} = (\mathbf{N}, \mathbf{T}, \mathbf{P}, S)$ 作为输入，输出为该文法的拓广文法 $\mathbf{G}' = (\mathbf{N} \cup \mathbf{S}', \mathbf{T}, \mathbf{P} \cup \{\mathbf{S}' \to \mathbf{S}\}, S')$。

实现代码如下：

```
1  void Grammar::ExtendGrammar()
2  {
3      string newStartSymbol = "S'";
4      nonterminals.insert(newStartSymbol);
5      productions[newStartSymbol].push_back({startSymbol});
6      startSymbol = newStartSymbol;
7
8      // 输出拓广文法
9      cout << "====After Extend====" << endl;
10     cout << "start symbol: " << startSymbol << endl;
```

```
    for (const auto &nonter : nonterminals)
    {
        cout << nonter << " -> ";
        for (auto it = productions[nonter].begin(); it != productions[nonter].end()
            ; it++)
        {
            for (const auto &sym : *it)
                cout << sym << " ";
            if (it != productions[nonter].end() - 1)
                cout << "| ";
        }
        cout << endl;
    }
    cout << endl;
}
```

2.2.3 构建非终结符的 FIRST 集

该算法接受一个上下文无关文法 $\mathbf{G} = (\mathbf{N}, \mathbf{T}, \mathbf{P}, S)$ 作为输入，输出为 FIRST 集，其中每个非终结符 A 都对应一个 FIRST 集合 FIRST(A)。

由于文法可能存在左递归，需特殊判断形如 $A \to A\ldots$ 和 $A \to B\ldots, B \to A\ldots$ 的 FIRST 集相互包含的产生式，避免死循环。为了避免死循环，定义变量`firstRelation`记录 FIRST 集的包含关系。算法最后再处理相互包含的 FIRST 集。

算法伪代码如下：

Algorithm 1: 构建非终结符的 FIRST 集

Input: $\mathbf{G} = (\mathbf{N}, \mathbf{T}, \mathbf{P}, S)$

Output: $\mathbf{FIRST}$

```
1  Function BuildFirst(A)
2      if firstDoned[A] = ture then return;
3      foreach A → α ∈ P do
4          if α = ε then
5              FIRST(α) ← {ε};
6          end
7          for i ← 1 to k do                                   // α = Y_1 Y_2 ... Y_k
8              if Y_i ∈ T then
9                  FIRST(α) ← FIRST(α) ∪ {Y_i};
10                 break;
11             end
12             BuildFirst(Y_i);
13             FIRST(α) = FIRST(α) ∪ FIRST(Y_i);
14             if ε ∉ FIRST(Y_i) then break;
15         end
16         if i = k then FIRST(α) ← FIRST(α) ∪ {ε};
17         FIRST(A) ← FIRST(A) ∪ FIRST(α);
18     end
19 end
20 foreach A ∈ N do
21     if firstDoned[A] = false then BuildFirst(A);
22 end
23 foreach A, B ∈ N do
24     if firstRelation[A, B] = true ∧ firstRelation[B, A] = true then
25         FIRST(A) ← FIRST(A) ∪ FIRST(B);
26         FIRST(B) ← FIRST(A);
27     end
28 end
```

2.2.4 构造 LR(1) 项目集的闭包

该算法接受一个 LR(1) 项目集 **I** 作为输入，输出为该项目集的闭包 $\mathbf{J} = \mathbf{closure}(\mathbf{I})$。

算法伪代码如下：

Algorithm 2: closure(I) 的构造过程

输入： 项目集合 I
输出： 集合 $J = closure(I)$
方法：
$J = I$
do $J_{\text{new}} = J$;
  for ($[A \to \alpha \cdot B\beta, a]$ 和文法 G 的每个产生式 $B \to \eta$)
    for ($FIRST(\beta a)$ 中的每个终结符号 b)
      if ($[B \to \cdot\eta, b] \notin J$)
        把 $[B \to \cdot\eta, b]$ 加入 J;
while ($J_{\text{new}} \neq J$);

2.2.5 构造 LR(1) 项目集的转移

转移函数的定义：$go(I, X) = \text{closure}(J)$，其中$J = \{[A \to \alpha X \cdot \beta, a] \mid$ 当$[A \to \alpha \cdot X\beta, a] \in I$时$\}$

实现代码如下：

```
void Grammar::Go(const LR1s &lr1s, const string &symbol, LR1s &newLR1s)
{
    for (auto const &lr1 : lr1s)
    {
        string nonterminal = lr1.first.first;            // 非终结符
        vector<string> pro = lr1.first.second;           // 产生式
        int index = lr1.second.first;                    // 圆点的位置
        string nextSymbols = lr1.second.second;          // 向前看符号
        if (index == pro.size() || pro[index] != symbol) // 圆点在产生式最后或者圆
            点后面不是symbol
            continue;
        // 加入新的LR(1)项目
        newLR1s.insert({{nonterminal, pro}, {index + 1, nextSymbols}});
    }
    Closure(newLR1s);
}
```

2.2.6 构造识别该文法所有活前缀的 DFA

该算法接受一个拓广文法 G' 作为输入，输出为该拓广文法的 LR(1) 项目集规范族。

算法伪代码如下：

Algorithm 3: 构造文法 G 的 $LR(1)$ 项目集规范族

输入： 拓广文法 G'

输出： G' 的 $LR(1)$ 项目集规范族

方法：

$C = \{\text{closure}(\{[S' \to \cdot S, \$]\})\}$

do

  for (每一个项目集 I 和每一个文法符号 X)

    if ($\text{go}(I, X)$ 不为空，且不在 C 中)

      把 $\text{go}(I, X)$ 加入 C 中;

while (没有新项目集加入 C 中)

2.2.7 判断是否为 LR(1) 文法

一个文法是 LR(1) 文法，当且仅当它的 LR(1) 项目集规范族中不含有任何冲突。

IsLR1函数遍历拓广文法的 LR(1) 项目集规范族中的每一个项目集，判断是否存在移进-规约冲突或规约-规约冲突。

实现代码如下：

```
bool Grammar::IsLR1()
{
    for (const auto &state : DFAStates)
    {
        for (const auto &lr1 : state)
        {
            if (lr1.second.first == lr1.first.second.size())
            {
                for (const auto &lr1t : state)
                {
                    int index = lr1t.second.first;
                    // 移进-规约冲突
                    if (index != lr1t.first.second.size() && lr1t.first.second[
                        index] == lr1.second.second)
                        return false;
                    // 规约-规约冲突
                    else if (lr1t != lr1 && index == lr1t.first.second.size() &&
                        lr1t.second.second == lr1.second.second)
                        return false;
                }
            }
        }
    }
```

```
    return true;
}
```

2.3 LR(1) 分析表模块

table 模块定义了 Table 类：

```
class Table
{
public:
    Table() = default;

    /**
     * @brief 构造LR(1)分析表
     * @param grammar LR(1)文法
     */
    void BuildTable(const Grammar &g);

    /**
     * @brief 输出LR(1)分析表
     */
    void OutputTable() const;

    /**
     * @brief  错误处理
     * @param state 当前状态
     * @param symbol 输入串的下一个符号
     * @param index 输入串的下一个符号的下标
     * @param sentence 输入串
     */
    void Error(const int state, string &symbol, const int index, string &sentence)
        const;

    unordered_set<string> nonterminals;                              // 非终结符号集
    unordered_set<string> terminals;                                 // 终结符号集
    vector<LR1s> DFAStates;                                          // DFA状态集/LR
        (1)项目集规范族
    unordered_map<int, unordered_map<string, int>> DFAStateTrans; // DFA状态转移
    vector<pair<string, vector<string>>> productions;                // 产生式集合
    unordered_map<int, unordered_map<string, int>> parsingTable;  // 预测分析表
};
```

该类实现了构造 LR(1) 分析表，输出分析表内容，错误处理等方法。

2.3.1 构造 LR(1) 分析表

该算法接受一个拓广文法 G' 作为输入，输出为该拓广文法的 LR(1) 分析表。

算法伪代码如下：

Algorithm 4: 构造 $LR(1)$ 分析表

输入： 拓广文法 G'

输出： 文法 G' 的分析表

方法如下：

1. 构造文法 G' 的 $LR(1)$ 项目集规范族 $C = \{I_0, I_1, \ldots, I_n\}$
2. 对于状态 i（代表项目集 I_i），分析动作如下：
   a) 若 $[A \to \alpha \cdot a\beta, b] \in I_i$，且 go$(I_i, a) = I_j$，则置 action$[i, a] = S_j$
   b) 若 $[A \to \alpha\cdot, a] \in I_i$，且 $A \neq S'$，则置 action$[i, a] = R_j$
   c) 若 $[S' \to S\cdot, \$] \in I_i$，则置 action$[i, \$] =$ ACC
3. 若对非终结符号 A，有 go$(I_i, A) = I_j$，则置 goto$[i, A] = j$
4. 凡是不能用上述规则填入信息的空白表项，均置上出错标志 *error*。
5. 分析程序的初态是包括 $[S' \to \cdot S, \$]$ 的有效项目集所对应的状态。

2.3.2 输出预测分析表

`OutputTable`函数按照以下格式输出预测分析表的内容：

```
====Productions====
(0) T -> F
(1) E -> T
(2) F -> num
(3) E -> E - T
(4) E -> E + T
(5) T -> T * F
(6) T -> T / F
(7) F -> ( E )

====Parsing Table====
I0: {action: [($, error) (num, S4) (-, error) (+, error) (*, error) (/,
↪  error) ((, S5) (), error) ], goto: [(F, 1) (E, 2) (T, 3) ]}
I1: {action: [($, R0) (num, error) (-, R0) (+, R0) (*, R0) (/, R0) ((, error)
↪  (), error) ], goto: [(F, error) (E, error) (T, error) ]}
I2: {action: [($, ACC) (num, error) (-, S6) (+, S7) (*, error) (/, error) ((,
↪  error) (), error) ], goto: [(F, error) (E, error) (T, error) ]}
I3: {action: [($, R1) (num, error) (-, R1) (+, R1) (*, S8) (/, S9) ((, error)
↪  (), error) ], goto: [(F, error) (E, error) (T, error) ]}
I4: {action: [($, R2) (num, error) (-, R2) (+, R2) (*, R2) (/, R2) ((, error)
↪  (), error) ], goto: [(F, error) (E, error) (T, error) ]}
```

```
I5: {action: [($, error) (num, S13) (-, error) (+, error) (*, error) (/,
↪  error) ((, S14) (), error) ], goto: [(F, 10) (E, 11) (T, 12) ]}
I6: {action: [($, error) (num, S4) (-, error) (+, error) (*, error) (/,
↪  error) ((, S5) (), error) ], goto: [(F, 1) (E, error) (T, 15) ]}
I7: {action: [($, error) (num, S4) (-, error) (+, error) (*, error) (/,
↪  error) ((, S5) (), error) ], goto: [(F, 1) (E, error) (T, 16) ]}
I8: {action: [($, error) (num, S4) (-, error) (+, error) (*, error) (/,
↪  error) ((, S5) (), error) ], goto: [(F, 17) (E, error) (T, error) ]}
I9: {action: [($, error) (num, S4) (-, error) (+, error) (*, error) (/,
↪  error) ((, S5) (), error) ], goto: [(F, 18) (E, error) (T, error) ]}
I10: {action: [($, error) (num, error) (-, R0) (+, R0) (*, R0) (/, R0) ((,
↪  error) (), R0) ], goto: [(F, error) (E, error) (T, error) ]}
I11: {action: [($, error) (num, error) (-, S19) (+, S20) (*, error) (/,
↪  error) ((, error) (), S21) ], goto: [(F, error) (E, error) (T, error) ]}
I12: {action: [($, error) (num, error) (-, R1) (+, R1) (*, S22) (/, S23) ((,
↪  error) (), R1) ], goto: [(F, error) (E, error) (T, error) ]}
I13: {action: [($, error) (num, error) (-, R2) (+, R2) (*, R2) (/, R2) ((,
↪  error) (), R2) ], goto: [(F, error) (E, error) (T, error) ]}
I14: {action: [($, error) (num, S13) (-, error) (+, error) (*, error) (/,
↪  error) ((, S14) (), error) ], goto: [(F, 10) (E, 24) (T, 12) ]}
I15: {action: [($, R3) (num, error) (-, R3) (+, R3) (*, S8) (/, S9) ((,
↪  error) (), error) ], goto: [(F, error) (E, error) (T, error) ]}
I16: {action: [($, R4) (num, error) (-, R4) (+, R4) (*, S8) (/, S9) ((,
↪  error) (), error) ], goto: [(F, error) (E, error) (T, error) ]}
I17: {action: [($, R5) (num, error) (-, R5) (+, R5) (*, R5) (/, R5) ((,
↪  error) (), error) ], goto: [(F, error) (E, error) (T, error) ]}
I18: {action: [($, R6) (num, error) (-, R6) (+, R6) (*, R6) (/, R6) ((,
↪  error) (), error) ], goto: [(F, error) (E, error) (T, error) ]}
I19: {action: [($, error) (num, S13) (-, error) (+, error) (*, error) (/,
↪  error) ((, S14) (), error) ], goto: [(F, 10) (E, error) (T, 25) ]}
I20: {action: [($, error) (num, S13) (-, error) (+, error) (*, error) (/,
↪  error) ((, S14) (), error) ], goto: [(F, 10) (E, error) (T, 26) ]}
I21: {action: [($, R7) (num, error) (-, R7) (+, R7) (*, R7) (/, R7) ((,
↪  error) (), error) ], goto: [(F, error) (E, error) (T, error) ]}
I22: {action: [($, error) (num, S13) (-, error) (+, error) (*, error) (/,
↪  error) ((, S14) (), error) ], goto: [(F, 27) (E, error) (T, error) ]}
I23: {action: [($, error) (num, S13) (-, error) (+, error) (*, error) (/,
↪  error) ((, S14) (), error) ], goto: [(F, 28) (E, error) (T, error) ]}
```

```
I24: {action: [($, error) (num, error) (-, S19) (+, S20) (*, error) (/,
↪  error) ((, error) (), S29) ], goto: [(F, error) (E, error) (T, error) ]}
I25: {action: [($, error) (num, error) (-, R3) (+, R3) (*, S22) (/, S23) ((,
↪  error) (), R3) ], goto: [(F, error) (E, error) (T, error) ]}
I26: {action: [($, error) (num, error) (-, R4) (+, R4) (*, S22) (/, S23) ((,
↪  error) (), R4) ], goto: [(F, error) (E, error) (T, error) ]}
I27: {action: [($, error) (num, error) (-, R5) (+, R5) (*, R5) (/, R5) ((,
↪  error) (), R5) ], goto: [(F, error) (E, error) (T, error) ]}
I28: {action: [($, error) (num, error) (-, R6) (+, R6) (*, R6) (/, R6) ((,
↪  error) (), R6) ], goto: [(F, error) (E, error) (T, error) ]}
I29: {action: [($, error) (num, error) (-, R7) (+, R7) (*, R7) (/, R7) ((,
↪  error) (), R7) ], goto: [(F, error) (E, error) (T, error) ]}
```

2.3.3 错误处理

定义了五种错误：

- Missing arithmetic object（缺少运算对象）
- Mismatched parentheses（括号不匹配）
- Missing operator（缺少运算符）
- Missing opening parenthese or operator（缺少左括号或运算符）
- Missing closing parenthese（缺少右括号）

Error函数能够判断具体错误类型并打印相应错误信息，同时适当恢复错误使得语法分析能够继续。

2.4 主函数模块

主函数的处理流程如下：

1. 用户输入文法文件名，打开该文件读取文法
2. 调用 grammar 模块的方法构造识别输入文法所有活前缀的 DFA
3. 调用 table 模块的方法构建 LR(1) 文法的分析表
4. 用户输入待预测分析的算数表达式，执行 LR 分析控制程序

2.4.1 LR 分析控制程序

该算法接受一个上下文无关文法 $\mathbf{G}$、它的 LR(1) 分析表以及一个符号串 ω 作为输入，输出为该符号串自底向上的分析。

算法伪代码如下：

Algorithm 5: LR 分析控制程序

Require: 文法 G 的一张分析表和一个输入符号串 ω

Ensure: 若 $\omega \in L(G)$，得到 ω 的自底向上的分析，否则报错

开始时，初始状态 S_0 在栈顶，ω 在输入缓冲区中。置 ip 指向第一个符号。
while true **do**
  S 是栈顶状态，a 是 ip 所指向的符号
  if action$[S, a] ==$ shift S' **then**
    把 a 和 S' 分别压入符号栈和状态栈
    推进 ip，使它指向下一个符号
  else if action$[S, a] ==$ reduce by $A \to \beta$ **then**
    从栈顶弹出 $|\beta|$ 个符号
    令 S' 是现在的栈顶状态
    把 A 和 goto$[S', A]$ 分别压入符号栈和状态栈
    输出 $A \to \beta$
  else if action$[S, a] ==$ accept **then**
    return
  else
    error()
  end if
end while

3 用户手册

运行`Syntax.exe`程序,输入文法文件名称,eg.`grammar1.txt`,注意文法文件应存储在`grammar`文件夹下。程序会依次输出拓广后的文法，FIRSR 集，识别拓广文法所有活前缀的 DFA 的状态以及转移，文法是否为 LR(1) 文法的判断以及 LR(1) 分析表的内容。

接着输入待分析的算数表达式，注意表达式中不应该包含空格，同时程序可以循环读入算数表达式并进行分析，通过输入"$" 以结束程序。

详细说明文档见`README.md`文件。

4 程序测试

程序分别对实验要求的算数表达式文法以及一个自选文法进行了测试。

4.1 算数表达式文法

读入给定的文法，程序输出相关预处理信息:

```
Please input the grammar file name: grammar1.txt
====After Extend====
```

```
start symbol: S'
S' -> E
F -> ( E ) | num
E -> E + T | E - T | T
T -> T * F | T / F | F

====FIRST Sets====
S': num (
F: num (
E: num (
T: num (

====DFA States====
I0:
F -> .( E ) , + * $ / -
F -> .num , / * - + $
T -> .F , - * $ / +
T -> .T / F , - + / $ *
E -> .T , + $ -
E -> .E + T , - + $
S' -> .E , $
T -> .T * F , + / - $ *
E -> .E - T , - + $

I1:
T -> F ., $ + / - *

I2:
E -> E .- T , $ - +
S' -> E ., $
E -> E .+ T , - + $

I3:
T -> T ./ F , * / - $ +
T -> T .* F , $ * - + /
E -> T ., - + $

I4:
```

```
F -> num ., $ + - / *

I5:
F -> .num , + * ) / -
E -> .E - T , + - )
E -> .E + T , + - )
F -> .( E ) , * / ) - +
T -> .F , ) * + - /
T -> .T * F , ) + - / *
T -> .T / F , ) - / + *
E -> .T , ) - +
F -> ( .E ) , + * $ / -

I6:
F -> .( E ) , * / + $ -
F -> .num , / + - * $
T -> .F , / * $ + -
T -> .T / F , / - + $ *
E -> E - .T , + $ -
T -> .T * F , + $ - * /

I7:
F -> .( E ) , * / $ + -
F -> .num , / + * $ -
T -> .F , / $ * + -
T -> .T / F , $ + / - *
E -> E + .T , $ - +
T -> .T * F , + $ - * /

I8:
F -> .( E ) , * $ + - /
F -> .num , $ - / * +
T -> T * .F , * $ - + /

I9:
F -> .( E ) , / $ * - +
F -> .num , * - + / $
T -> T / .F , / $ * - +
```

```
I10:
T -> F ., / - + ) *

I11:
E -> E .+ T , ) - +
F -> ( E .) , / - + * $
E -> E .- T , ) + -

I12:
T -> T ./ F , * ) + - /
T -> T .* F , / + ) * -
E -> T ., + - )

I13:
F -> num ., - / + ) *

I14:
F -> .num , + * ) / -
E -> .E - T , + - )
E -> .E + T , + - )
F -> .( E ) , * / ) - +
T -> .F , ) * + - /
T -> .T * F , ) + - / *
T -> .T / F , ) - / + *
E -> .T , ) - +
F -> ( .E ) , / * ) - +

I15:
T -> T .* F , / $ + * -
T -> T ./ F , * - / + $
E -> E - T ., - + $

I16:
T -> T .* F , / $ + * -
E -> E + T ., + $ -
T -> T ./ F , * - + $ /
```

```
I17:
T -> T * F ., - / * + $

I18:
T -> T / F ., + - * / $

I19:
F -> .( E ) , / * - + )
F -> .num , / - + ) *
T -> .F , * - + ) /
E -> E - .T , - ) +
T -> .T * F , - * ) + /
T -> .T / F , - / ) + *

I20:
F -> .( E ) , * / ) + -
F -> .num , / + ) * -
T -> .F , / * ) + -
E -> E + .T , ) - +
T -> .T / F , + / - ) *
T -> .T * F , + ) - * /

I21:
F -> ( E ) ., $ * / + -

I22:
F -> .num , + - / ) *
F -> .( E ) , + - ) / *
T -> T * .F , + / - ) *

I23:
F -> .( E ) , / * + ) -
F -> .num , * + - ) /
T -> T / .F , ) / * + -

I24:
F -> ( E .) , + ) / * -
E -> E .+ T , ) + -
```

```
E -> E .- T , ) + -

I25:
T -> T .* F , / + ) - *
T -> T ./ F , * ) - + /
E -> E - T ., ) - +

I26:
T -> T ./ F , * ) + / -
T -> T .* F , / + ) * -
E -> E + T ., - ) +

I27:
T -> T * F ., ) - + * /

I28:
T -> T / F ., - + ) * /

I29:
F -> ( E ) ., * / - + )



====DFA Transitions====
I26 -> / -> I23    I26 -> * -> I22
I25 -> / -> I23    I25 -> * -> I22
I24 -> ) -> I29    I24 -> - -> I19    I24 -> + -> I20
I23 -> ( -> I14    I23 -> F -> I28    I23 -> num -> I13
I6 -> num -> I4    I6 -> F -> I1    I6 -> ( -> I5    I6 -> T -> I15
I5 -> ( -> I14    I5 -> T -> I12    I5 -> F -> I10    I5 -> num -> I13
I5 -> E -> I11
I22 -> ( -> I14    I22 -> F -> I27    I22 -> num -> I13
I3 -> / -> I9    I3 -> * -> I8
I20 -> num -> I13    I20 -> F -> I10    I20 -> ( -> I14    I20 -> T -> I26
I7 -> num -> I4    I7 -> F -> I1    I7 -> ( -> I5    I7 -> T -> I16
I0 -> ( -> I5    I0 -> T -> I3    I0 -> F -> I1    I0 -> num -> I4
I0 -> E -> I2
I2 -> + -> I7    I2 -> - -> I6
I19 -> num -> I13    I19 -> F -> I10    I19 -> ( -> I14    I19 -> T -> I25
```

```
I8 -> ( -> I5    I8 -> F -> I17    I8 -> num -> I4
I9 -> ( -> I5    I9 -> F -> I18    I9 -> num -> I4
I11 -> ) -> I21    I11 -> - -> I19    I11 -> + -> I20
I12 -> / -> I23    I12 -> * -> I22
I14 -> ( -> I14    I14 -> T -> I12    I14 -> F -> I10    I14 -> num -> I13
I14 -> E -> I24
I15 -> / -> I9    I15 -> * -> I8
I16 -> / -> I9    I16 -> * -> I8

This grammar is LR(1) grammar.

====Productions====
(0) T -> F
(1) E -> T
(2) F -> num
(3) E -> E - T
(4) E -> E + T
(5) T -> T * F
(6) T -> T / F
(7) F -> ( E )

====Parsing Table====
I0: {action: [($, error) (num, S4) (-, error) (+, error) (*, error) (/,
↪  error) ((, S5) (), error) ], goto: [(F, 1) (E, 2) (T, 3) ]}
I1: {action: [($, R0) (num, error) (-, R0) (+, R0) (*, R0) (/, R0) ((, error)
↪  (), error) ], goto: [(F, error) (E, error) (T, error) ]}
I2: {action: [($, ACC) (num, error) (-, S6) (+, S7) (*, error) (/, error) ((,
↪  error) (), error) ], goto: [(F, error) (E, error) (T, error) ]}
I3: {action: [($, R1) (num, error) (-, R1) (+, R1) (*, S8) (/, S9) ((, error)
↪  (), error) ], goto: [(F, error) (E, error) (T, error) ]}
I4: {action: [($, R2) (num, error) (-, R2) (+, R2) (*, R2) (/, R2) ((, error)
↪  (), error) ], goto: [(F, error) (E, error) (T, error) ]}
I5: {action: [($, error) (num, S13) (-, error) (+, error) (*, error) (/,
↪  error) ((, S14) (), error) ], goto: [(F, 10) (E, 11) (T, 12) ]}
I6: {action: [($, error) (num, S4) (-, error) (+, error) (*, error) (/,
↪  error) ((, S5) (), error) ], goto: [(F, 1) (E, error) (T, 15) ]}
I7: {action: [($, error) (num, S4) (-, error) (+, error) (*, error) (/,
↪  error) ((, S5) (), error) ], goto: [(F, 1) (E, error) (T, 16) ]}
```

```
I8: {action: [($, error) (num, S4) (-, error) (+, error) (*, error) (/,
↪  error) ((, S5) (), error) ], goto: [(F, 17) (E, error) (T, error) ]}
I9: {action: [($, error) (num, S4) (-, error) (+, error) (*, error) (/,
↪  error) ((, S5) (), error) ], goto: [(F, 18) (E, error) (T, error) ]}
I10: {action: [($, error) (num, error) (-, R0) (+, R0) (*, R0) (/, R0) ((,
↪  error) (), R0) ], goto: [(F, error) (E, error) (T, error) ]}
I11: {action: [($, error) (num, error) (-, S19) (+, S20) (*, error) (/,
↪  error) ((, error) (), S21) ], goto: [(F, error) (E, error) (T, error) ]}
I12: {action: [($, error) (num, error) (-, R1) (+, R1) (*, S22) (/, S23) ((,
↪  error) (), R1) ], goto: [(F, error) (E, error) (T, error) ]}
I13: {action: [($, error) (num, error) (-, R2) (+, R2) (*, R2) (/, R2) ((,
↪  error) (), R2) ], goto: [(F, error) (E, error) (T, error) ]}
I14: {action: [($, error) (num, S13) (-, error) (+, error) (*, error) (/,
↪  error) ((, S14) (), error) ], goto: [(F, 10) (E, 24) (T, 12) ]}
I15: {action: [($, R3) (num, error) (-, R3) (+, R3) (*, S8) (/, S9) ((,
↪  error) (), error) ], goto: [(F, error) (E, error) (T, error) ]}
I16: {action: [($, R4) (num, error) (-, R4) (+, R4) (*, S8) (/, S9) ((,
↪  error) (), error) ], goto: [(F, error) (E, error) (T, error) ]}
I17: {action: [($, R5) (num, error) (-, R5) (+, R5) (*, R5) (/, R5) ((,
↪  error) (), error) ], goto: [(F, error) (E, error) (T, error) ]}
I18: {action: [($, R6) (num, error) (-, R6) (+, R6) (*, R6) (/, R6) ((,
↪  error) (), error) ], goto: [(F, error) (E, error) (T, error) ]}
I19: {action: [($, error) (num, S13) (-, error) (+, error) (*, error) (/,
↪  error) ((, S14) (), error) ], goto: [(F, 10) (E, error) (T, 25) ]}
I20: {action: [($, error) (num, S13) (-, error) (+, error) (*, error) (/,
↪  error) ((, S14) (), error) ], goto: [(F, 10) (E, error) (T, 26) ]}
I21: {action: [($, R7) (num, error) (-, R7) (+, R7) (*, R7) (/, R7) ((,
↪  error) (), error) ], goto: [(F, error) (E, error) (T, error) ]}
I22: {action: [($, error) (num, S13) (-, error) (+, error) (*, error) (/,
↪  error) ((, S14) (), error) ], goto: [(F, 27) (E, error) (T, error) ]}
I23: {action: [($, error) (num, S13) (-, error) (+, error) (*, error) (/,
↪  error) ((, S14) (), error) ], goto: [(F, 28) (E, error) (T, error) ]}
I24: {action: [($, error) (num, error) (-, S19) (+, S20) (*, error) (/,
↪  error) ((, error) (), S29) ], goto: [(F, error) (E, error) (T, error) ]}
I25: {action: [($, error) (num, error) (-, R3) (+, R3) (*, S22) (/, S23) ((,
↪  error) (), R3) ], goto: [(F, error) (E, error) (T, error) ]}
I26: {action: [($, error) (num, error) (-, R4) (+, R4) (*, S22) (/, S23) ((,
↪  error) (), R4) ], goto: [(F, error) (E, error) (T, error) ]}
```

```
I27: {action: [($, error) (num, error) (-, R5) (+, R5) (*, R5) (/, R5) ((,
↪  error) (), R5) ], goto: [(F, error) (E, error) (T, error) ]}
I28: {action: [($, error) (num, error) (-, R6) (+, R6) (*, R6) (/, R6) ((,
↪  error) (), R6) ], goto: [(F, error) (E, error) (T, error) ]}
I29: {action: [($, error) (num, error) (-, R7) (+, R7) (*, R7) (/, R7) ((,
↪  error) (), R7) ], goto: [(F, error) (E, error) (T, error) ]}
```

4.2 测试集 1

此测试集为一个简单的算术表达式，用于测试程序是否能够正常运行。

4.2.1 测试内容

```
(1+(2*3))/4
```

4.2.2 测试结果

```
F -> num
T -> F
E -> T
F -> num
T -> F
F -> num
T -> T * F
E -> T
F -> ( E )
T -> F
E -> E + T
F -> ( E )
T -> F
F -> num
T -> T / F
E -> T
success!
```

4.2.3 结果分析

经过验证，测试结果为算数表达式 `(1+(2*3))/4` 的最右推导的逆序，说明程序可以对简单的算数表达式正确分析。

4.3 测试集 2

此测试集为一个较为复杂的算术表达式，用于测试程序的鲁棒性。

4.3.1 测试内容

```
((1+(2*(3/4+5)-(6+7)/(8-9))*(10+11+12*13))/(14*(15+16)))
```

4.3.2 测试结果

```
F -> num
T -> F
E -> T
F -> num
T -> F
F -> num
T -> F
F -> num
T -> T / F
E -> T
F -> num
T -> F
E -> E + T
F -> ( E )
T -> T * F
E -> T
F -> num
T -> F
E -> T
F -> num
T -> F
E -> E + T
F -> ( E )
T -> F
F -> num
T -> F
E -> T
F -> num
T -> F
E -> E - T
```

```
F -> ( E )
T -> T / F
E -> E - T
F -> ( E )
T -> F
F -> num
T -> F
E -> T
F -> num
T -> F
E -> E + T
F -> num
T -> F
F -> num
T -> T * F
E -> E + T
F -> ( E )
T -> T * F
E -> E + T
F -> ( E )
T -> F
F -> num
T -> F
F -> num
T -> F
E -> T
F -> num
T -> F
E -> E + T
F -> ( E )
T -> T * F
E -> T
F -> ( E )
T -> T / F
E -> T
F -> ( E )
T -> F
E -> T
```

```
success!
```

4.3.3 结果分析

经过验证,测试结果为算数表达式((1+(2*(3/4+5)-(6+7)/(8-9))*(10+11+12*13))/(14*(15+16))) 的最右推导的逆序，说明程序可以对较为复杂的算数表达式正确分析。

4.4 测试集 3

此测试集为一个缺少右括号的算术表达式，用于测试程序是否能够发现该错误并处理。

4.4.1 测试内容

```
(1+2
```

4.4.2 测试结果

```
F -> num
T -> F
E -> T
Error: Missing closing parenthese!
F -> num
T -> F
E -> E + T
F -> ( E )
T -> F
E -> T
success!
```

4.4.3 结果分析

程序能够发现算数表达式缺少右括号的错误,并输出相应错误信息:Error: Missing closing parenthese!。同时程序能够适当恢复错误让语法分析继续执行下去。

4.5 测试集 4

此测试集为一个缺少左括号的算术表达式，用于测试程序是否能够发现该错误并处理。

4.5.1 测试内容

```
1+2)
```

4.5.2 测试结果

```
F -> num
T -> F
E -> T
Error: Mismatched parentheses!
F -> num
T -> F
E -> E + T
success!
```

4.5.3 结果分析

程序能够发现算数表达式缺少左括号的错误，并输出相应错误信息：`Error: Mismatched parentheses!`。同时程序能够适当恢复错误让语法分析继续执行下去。

4.6 测试集 5

此测试集为一个缺少运算符的算术表达式，用于测试程序是否能够发现该错误并处理。

4.6.1 测试内容

```
1(2+3)
```

4.6.2 测试结果

```
Error: Mismatched parentheses!
Error: Mismatched parentheses!
F -> num
T -> F
E -> T
Error: Mismatched parentheses!
F -> num
T -> F
E -> E + T
success!
```

4.6.3 结果分析

程序能够发现算数表达式缺少运算符的错误，并输出相应错误信息：`Error: Mismatched parentheses!`。这是因为程序对该类错误的恢复机制是：`1(2+3)-> 1+3`，程序会认为左右括号

均不匹配。同时程序能够适当恢复错误让语法分析继续执行下去。

4.7 测试集 6

此测试集为一个缺少运算对象的算术表达式，用于测试程序是否能够发现该错误并处理。

4.7.1 测试内容

```
*(2+3)
```

4.7.2 测试结果

```
Error: Missing arithmetic object!
F -> num
T -> F
F -> num
T -> F
E -> T
F -> num
T -> F
E -> E + T
F -> ( E )
T -> T * F
E -> T
success!
```

4.7.3 结果分析

程序能够发现算数表达式缺少运算对象的错误，并输出相应错误信息：`Error: Missing arithmetic object!`。同时程序能够适当恢复错误让语法分析继续执行下去。

4.8 自选文法

读入第四章讲义 P57 页练习中的文法，程序输出相关预处理信息：

```
Please input the grammar file name: grammar2.txt
====After Extend====
start symbol: S'
S' -> S
S -> ( L ) | a
L -> L , S | S
```

```
====FIRST Sets====
S': a (
S: a (
L: a (

====DFA States====
I0:
S -> .( L ) , $
S -> .a , $
S' -> .S , $

I1:
S' -> S ., $

I2:
S -> a ., $

I3:
S -> .( L ) , , )
L -> .S , , )
L -> .L , S , ) ,
S -> ( .L ) , $
S -> .a , ) ,

I4:
L -> S ., ) ,

I5:
L -> L ., S , , )
S -> ( L .) , $

I6:
S -> a ., , )

I7:
S -> .( L ) , , )
L -> .S , , )
```

```
S -> .a , , )
L -> .L , S , , )
S -> ( .L ) , ) ,

I8:
S -> .( L ) , , )
S -> .a , ) ,
L -> L , .S , , )

I9:
S -> ( L ) ., $

I10:
L -> L ., S , ) ,
S -> ( L .) , , )

I11:
L -> L , S ., ) ,

I12:
S -> ( L ) ., ) ,



====DFA Transitions====
I8 -> ( -> I7    I8 -> S -> I11    I8 -> a -> I6
I5 -> ) -> I9    I5 -> , -> I8
I7 -> ( -> I7    I7 -> a -> I6    I7 -> S -> I4    I7 -> L -> I10
I0 -> ( -> I3    I0 -> S -> I1    I0 -> a -> I2
I10 -> ) -> I12    I10 -> , -> I8
I3 -> ( -> I7    I3 -> a -> I6    I3 -> S -> I4    I3 -> L -> I5

This grammar is LR(1) grammar.

====Productions====
(0) S -> a
(1) L -> S
(2) S -> ( L )
(3) L -> L , S
```

```
====Parsing Table====
I0: {action: [($, error) (a, S2) ((, S3) (,, error) (), error) ], goto: [(S,
↪  1) (L, error) ]}
I1: {action: [($, ACC) (a, error) ((, error) (,, error) (), error) ], goto:
↪  [(S, error) (L, error) ]}
I2: {action: [($, R0) (a, error) ((, error) (,, error) (), error) ], goto:
↪  [(S, error) (L, error) ]}
I3: {action: [($, error) (a, S6) ((, S7) (,, error) (), error) ], goto: [(S,
↪  4) (L, 5) ]}
I4: {action: [($, error) (a, error) ((, error) (,, R1) (), R1) ], goto: [(S,
↪  error) (L, error) ]}
I5: {action: [($, error) (a, error) ((, error) (,, S8) (), S9) ], goto: [(S,
↪  error) (L, error) ]}
I6: {action: [($, error) (a, error) ((, error) (,, R0) (), R0) ], goto: [(S,
↪  error) (L, error) ]}
I7: {action: [($, error) (a, S6) ((, S7) (,, error) (), error) ], goto: [(S,
↪  4) (L, 10) ]}
I8: {action: [($, error) (a, S6) ((, S7) (,, error) (), error) ], goto: [(S,
↪  11) (L, error) ]}
I9: {action: [($, R2) (a, error) ((, error) (,, error) (), error) ], goto:
↪  [(S, error) (L, error) ]}
I10: {action: [($, error) (a, error) ((, error) (,, S8) (), S12) ], goto:
↪  [(S, error) (L, error) ]}
I11: {action: [($, error) (a, error) ((, error) (,, R3) (), R3) ], goto: [(S,
↪  error) (L, error) ]}
I12: {action: [($, error) (a, error) ((, error) (,, R2) (), R2) ], goto: [(S,
↪  error) (L, error) ]}
```

4.9 测试集 7

此测试集为自选文法推导出的一个表达式，用于测试程序是否能够对任意文法的表达式进行语法分析。

4.9.1 测试内容

```
(a,(a,(a,a)))
```

4.9.2 测试结果

```
S -> a
L -> S
S -> a
L -> S
S -> a
L -> S
S -> a
L -> L , S
S -> ( L )
L -> L , S
S -> ( L )
L -> L , S
S -> ( L )
success!
```

4.9.3 结果分析

经过验证，测试结果为表达式 (a,(a,(a,a))) 的最右推导的逆序，说明程序可以对任意文法的表达式进行语法分析。

5 实验总结

在本次实验中，我成功地编写了 LR(1) 语法分析程序，实现了对算术表达式语法分析的任务。通过实验，我加深了对 LR(1) 语法分析方法的理解，掌握了 LR 分析表的构造和 LR 预测分析程序的编写。这次实验让我更清晰地认识到了文法分析的重要性，也提高了我的编程能力和对语法分析的应用能力。

同时我还扩展了自己的语法分析程序的功能，使其能够应用于任意文法，只需修改读入的文法文件的内容即可。