编译原理与技术实验二：语法分析程序的设计与实现（LL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 LL(1) 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. 编程实现算法 4.2，为给定文法自动构造预测分析表。
2. 编程实现算法 4.1，构造 LL(1) 预测分析程序。

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` 模块定义了将输入文法转换为 LL(1) 文法的相关操作，`table` 模块定义了构建 LL(1) 文法的预测分析表的操作，`main` 模块定义了预测分析控制程序并调用前两个模块以实现对输入的字符串预测分析。模块间的调用关系如 **图 1** 。

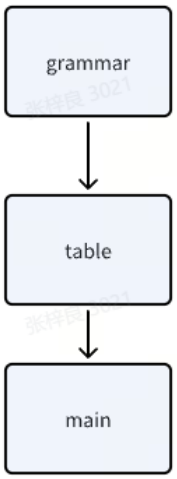

图 1: 模块调用关系

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
     */
    void GetGrammar(ifstream &fin);

    /**
     * @brief 消除左递归
     */
    void EliminateLeftRecursion();

    /**
     * @brief 提取左公因子
```

```
19          */
20         void ExtractLeftCommon();
21
22         /**
23          * @brief 构造非终结符的FIRST集
24          * @param nonterminal 非终结符
25          */
26         void BuildFirst(const string &noterminal);
27
28         /**
29          * @brief 构造非终结符的FOLLOW集
30          * @param noterminal 非终结符
31          */
32         void BuildFollow(const string &noterminal);
33
34         /**
35          * @brief 判断是否是LL(1)文法
36          * @return 如果是LL(1)文法，返回true；否则返回false
37          */
38         bool IsLL1() const;
39
40         /**
41          * @brief 将文法转换为LL(1)文法
42          */
43         void CovertGrammar();
44
45         unordered_set<string> nonterminals;                                  // 非终结符号集
46         unordered_set<string> terminals;                                     // 终结符号集
47         unordered_map<string, vector<vector<string>>> productions;           // 产生式集
48         string startSymbol;                                                  // 起始符号
49         unordered_map<string, vector<unordered_set<string>>> candidateFirst; // 候选式的FIRST集
50         unordered_map<string, unordered_set<string>> first;                  // 非终结符号的FIRST集
51         unordered_map<string, unordered_set<string>> follow;                 // 非终结符号的FOLLOW集
52         unordered_map<string, bool> firstDoned;                              // FIRST集是否已经构造完成
53         unordered_map<string, bool> followDoned;                             // FOLLOW集是否已经构造完成
54         unordered_map<string, unordered_map<string, bool>> followRelation;   // FOLLOW集的关系
55 };
```

该类实现了读入给定文法，消除左递归，提取左公因子，构建非终结符和候选式的 FIRST 集，构建非终结符的 FOLLOW 集，判断是否为 LL(1) 文法，将文法转换为 LL(1) 文法等方法。

具体处理流程如图2。

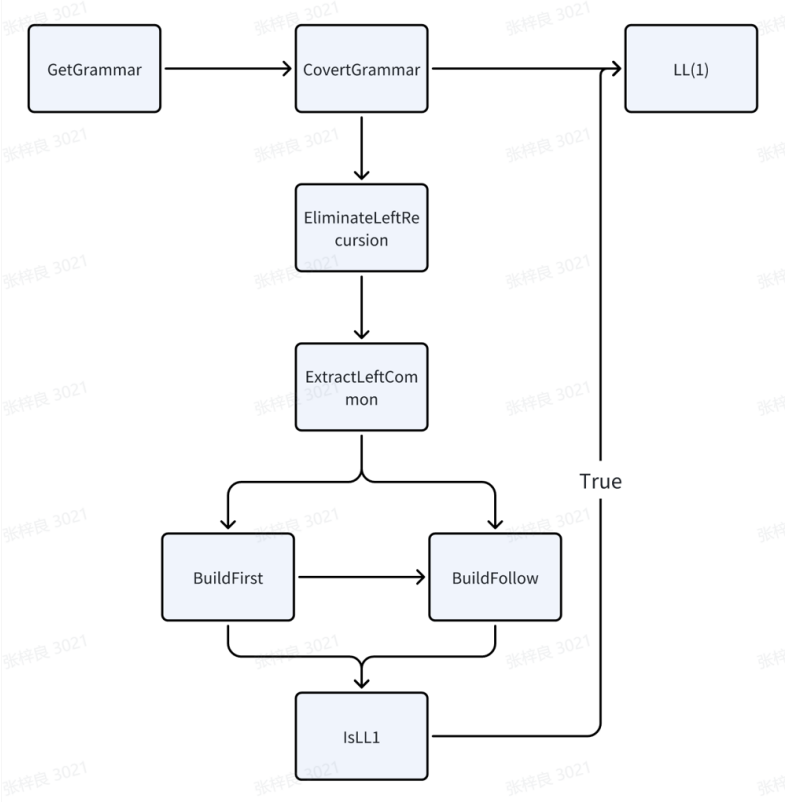

图 2: 处理流程图

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 Symbols=====”。遇到的第一个非终结符被指定为起始符号。

读取完非终结符后，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 消除左递归

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，输出为一个没有左递归的上下文无关文法 $\mathbf{G}' = (\mathbf{N}', \mathbf{T}, \mathbf{P}', S)$。

算法的伪代码如下：

Algorithm 1: 消除左递归

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

Output: $\mathbf{G}' = (\mathbf{N}', \mathbf{T}, \mathbf{P}', S)$

Function `EliminateLeftRecursion()`
- $\mathbf{G}' \leftarrow (\mathbf{N}, \mathbf{T}, \mathbf{P}', S)$;
- $\mathbf{P}' \leftarrow \varnothing$;
- **foreach** $A \to \alpha_1 | \alpha_2 | \ldots | \alpha_n \in \mathbf{P}$ **do**
  - **if** *any* α_i *starts with* A **then**
    - $\mathbf{N} \leftarrow \mathbf{N} \cup \{A'\}$;
    - $\mathbf{P}' \leftarrow \mathbf{P}' \cup \{A' \to \varepsilon\}$;
    - **foreach** $\alpha_i \in \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ **do**
      - $\alpha_i \leftarrow \alpha_i A'$;
      - **if** α_i *starts with* A **then**
        - $\alpha_i \leftarrow$ remove the leading A from α_i;
        - $\mathbf{P}' \leftarrow \mathbf{P}' \cup \{A' \to \alpha_i A'\}$;
      - **end**
      - **else**
        - $\mathbf{P}' \leftarrow \mathbf{P}' \cup \{A \to \alpha_i A'\}$;
      - **end**
    - **end**
  - **end**
  - **else**
    - $\mathbf{P}' \leftarrow \mathbf{P}' \cup \{A \to \alpha_1 | \alpha_2 | \ldots | \alpha_n\}$;
  - **end**
- **end**
- **return** $\mathbf{G}'$;

end

2.2.3 提取左公因子

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，输出为一个没有左公因子的上下文无关文法 $\mathbf{G}' = (\mathbf{N}', \mathbf{T}, \mathbf{P}', S)$。

算法伪代码如下：

Algorithm 2: 提取左公因子

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

Output: $\mathbf{G}' = (\mathbf{N}', \mathbf{T}, \mathbf{P}', S)$

```
Function ExtractLeftFactoring()
    G' ← (N, T, P', S);
    P' ← ∅;
    foreach A → α1|α2|...|αn ∈ P do
        while there exist common prefix C among {α1, α2, ..., αn} do
            N ← N ∪ {A'};
            P' ← P' ∪ {A → CA'};
            foreach αi ∈ {α1, α2, ..., αn} do
                if αi has prefix C then
                    αi ← remove C from the beginning of αi;
                    P' ← P' ∪ {A' → αi};
                end
                else
                    P' ← P' ∪ {A → αi};
                end
            end
        end
    end
    return G';
end
```

2.2.4 构建非终结符和候选式的 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，其中每个非终结符或候选式 α 都对应一个 FIRST 集合 FIRST(α)。

算法主要分为两个部分：

1. 利用递归调用，计算每个非终结符和候选式的 FIRST 集合；
2. 对于每个非终结符 A，如果 `fistDoned[A] = false`，则递归调用一次第一步以计算它的 FIRST 集合。

具体来说，算法对于每个产生式 $A \to \alpha$，将 α 拆分为 $Y_1Y_2 \dots Y_k$ 的形式，并依次处理每个 Y_i，直到遇到一个终结符或者某个 Y_i 的 FIRST 集合中不包含 ε。如果遍历完整个 α 之后，所有 Y_i 的 FIRST 集合都包含 ε，则将 ε 添加到 FIRST(α) 中。

根据计算出来的 FIRST(α)，对于每个产生式 $A \to \alpha$，将 FIRST(α) 添加到 FIRST(A) 中。

算法伪代码如下：

Algorithm 3: 构建非终结符和候选式的 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
```

2.2.5 构建非终结符的 FOLLOW 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 FOLLOW 集，其中每个非终结符 A 都对应一个 FOLLOW 集合 FOLLOW(A)。

该算法首先将 \$ 加入 S 的 FOLLOW 集，再遍历每一个非终结符号，通过判断 `followDoned` 是否为 `true` 判断该非终结符的 FOLLOW 集是否已经构建，若该非终结符的 FOLLOW 集还未

构建，则调用函数 `BuildFollow()` 构建该函数的 FOLLOW 集。

在构建过程中可能出现 A 的 FOLLOW 集包括 B 的 FOLLOW 集，同时 B 的 FOLLOW 集也包括 A 的 FOLLOW 集这样的情况，会导致递归函数 `BuildFollow()` 陷入死循环，因此需要对此情况进行特殊处理，此处引入一个变量 `followRelation` 来记录两个非终结符号的 FOLLOW 集合是否存在包含关系，`followRelation[A, B] = true` 代表 A 的 FOLLOW 集包括 B 的 FOLLOW 集。

`BuildFollow(B)` 函数的具体处理流程是：遍历所有产生式，若产生式的形式为：$A \to \alpha B \beta$，其中 $\beta = Y_1 Y_2 \dots Y_k$，则遍历 $Y_1 Y_2 \dots Y_k$。若 Y_i 为终结符，则将 Y_i 加入到 B 的 FOLLOW 集，结束遍历；若 Y_i 为非终结符，则将 Y_i 的 FIRST 集中除了 ε 以外的元素加入到 B 的 FOLLOW 集。若 Y_i 的 FIRST 集中有 ε 则继续遍历，否则结束遍历。

当遍历结束后，判断是否遍历到 β 的末尾或者 $\beta = \varepsilon$，若是，且 A 和 B 的 FOLLOW 集不互相包含，则调用 `BuildFollow(A)` 递归构造 A 的 FOLLOW 集，并将 A 的 FOLLOW 集加入到 B 的 FOLLOW 集中。

最后处理相互包含的特殊情况：若 A 和 B 的 FOLLOW 集互相包含，则将 A 和 B 的 FOLLOW 集都更新为二者的并。

算法伪代码如算法 4。

Algorithm 4: 构建非终结符的 FOLLOW 集

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

Output: $\mathbf{FOLLOW}$

Function `BuildFollow(`B`)`
  if followDoned$[B] = ture$ **then** return;
  foreach $A \in \mathbf{N}$ **do**
    foreach $A \to \alpha B \beta \in \mathbf{P}$ **do**
      for $i \leftarrow 1$ **to** k **do** // $\beta = Y_1 Y_2 \dots Y_k$
        if $Y_i \in \mathbf{T}$ **then**
          $\mathbf{FOLLOW}(B) \leftarrow \mathbf{FOLLOW}(B) \cup \{Y_i\}$;
          break;
        end
        $\mathbf{FOLLOW}(B) \leftarrow \mathbf{FOLLOW}(B) \cup (\mathbf{FIRST}(Y_i) - \{\varepsilon\})$;
        if $\varepsilon \notin \mathbf{FIRST}(Y_i)$ **then** break;
      end
      if $i = k \wedge A \neq B$ **then**
        followRelation$[B, A] = true$; // $\mathbf{FOLLOW}(A) \subseteq \mathbf{FOLLOW}(B)$
        if followRelation$[A, B] = false$ **then**
          `BuildFollow(`A`);`
        end
        $\mathbf{FOLLOW}(B) \leftarrow \mathbf{FOLLOW}(B) \cup \mathbf{FOLLOW}(A)$;
      end
    end
  end
  followDoned$[B] = true$;
end
$\mathbf{FOLLOW}(S) \leftarrow \{\$\}$;
foreach $A \in \mathbf{N}$ **do**
  if followDoned$[A] = false$ **then** `BuildFollow(`A`);`
end
foreach $A, B \in \mathbf{N}$ **do**
  if followRelation$[A, B] = true \wedge$ followRelation$[B, A] = true$ **then**
    $\mathbf{FOLLOW}(A) \leftarrow \mathbf{FOLLOW}(A) \cup \mathbf{FOLLOW}(B)$;
    $\mathbf{FOLLOW}(B) \leftarrow \mathbf{FOLLOW}(A)$;
  end
end

2.2.6 判断是否为 LL(1) 文法

一个文法是 LL(1) 文法，当且仅当它的每一个产生式 $A \to \alpha \mid \beta$，满足：

- FIRST(α) ∩ FIRST(β) = $\varnothing$ 并且
- 若 β 推导出 ε，则 FIRST(α) ∩ FOLLOW(A) = $\varnothing$

`IsLL1`函数遍历文法中的每个非终结符。对于每个非终结符，它遍历非终结符的候选式的 FIRST 集。对于每一对元候选式的 FIRST 集，函数检查两个候选式的 FIRST 集的交集是否非空。如果交集非空，则返回`false`。

函数还检查 ε 符号是否在候选式的 FIRST 集中。如果是，函数检查其他候选式的 FIRST 集和非终结符的 FOLLOW 集的交集是否非空。如果交集非空，则返回`false`。

如果所有的检查都没有失败，函数返回`true`，表示文法为 LL(1) 文法。

2.2.7 文法转换

`CovertGrammar`函数依次调用上述算法的函数，并输出相应转换信息，以实现将输入的文法转换为 LL(1) 文法的功能。

具体调用流程如图2。

2.3 预测分析表模块

table 模块定义了 Table 类：

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

    /**
     * @brief 构造预测分析表
     * @param grammar LL(1)文法
     */
    void buildTable(const Grammar &g);

    /**
     * @brief 输出预测分析表
     */
    void outputTable() const;

    /**
     * @brief  错误处理
     */
    void error(const string &nonter) const;

    unordered_map<string, unordered_map<string, vector<string>>> parsingTable; //
        预测分析表
```

```
23      unordered_set<string> nonterminals;                                    //
            非终结符号集
24      unordered_set<string> terminals;                                       //
            终结符号集
25  };
```

该类实现了构造预测分析表，输出预测分析表的内容，错误处理等方法。

2.3.1 构造预测分析表

构造预测分析表算法的伪代码如下：

输入：文法G
输出：文法G的预测分析表M
方法：

```
for (文法G的每个产生式 A→α ) {
    for (每个终结符号 a∈FIRST(α))
        把 A→α 放入 M[A, a] 中;
    if (ε∈FIRST(α))
        for (任何 b∈FOLLOW(A))
            把 A→α 放入 M[A, b] 中;
};
for (所有无定义的M[A, a])
    标上错误标志;
```

图 3: 构造预测分析表

BuildTable函数遍历语法中的每个非终结符。对于每个非终结符，它遍历该非终结符的产生式。函数获取当前产生式的 FIRST 集，对于 FIRST 集中的每个终结符，函数向分析表中添加一个条目，将非终结符和终结符映射到当前产生式。如果当前产生式的 FIRST 集包含 ε 符号，函数遍历非终结符的 FOLLOW 集。对于 FOLLOW 集中的每个终结符，函数向分析表中添加一个条目，将非终结符和终结符映射到当前产生式。

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：

```
====Parsing Table====
T' [ ( $: T' -> epsilon ) ( num: error ) ( -: T' -> epsilon ) ( +: T' ->
↪  epsilon ) ( *: T' -> * F T' ) ( /: T' -> / F T' ) ( (: error ) ( ): T' ->
↪  epsilon ) ]
E' [ ( $: E' -> epsilon ) ( num: error ) ( -: E' -> - T E' ) ( +: E' -> + T
↪  E' ) ( *: error ) ( /: error ) ( (: error ) ( ): E' -> epsilon ) ]
```

```
F [ ( $: error ) ( num: F -> num ) ( -: error ) ( +: error ) ( *: error ) (
↪  /: error ) ( (: F -> ( E ) ) ( ): error ) ]
E [ ( $: error ) ( num: E -> T E' ) ( -: error ) ( +: error ) ( *: error ) (
↪  /: error ) ( (: E -> T E' ) ( ): error ) ]
T [ ( $: error ) ( num: T -> F T' ) ( -: error ) ( +: error ) ( *: error ) (
↪  /: error ) ( (: T -> F T' ) ( ): error ) ]
```

2.3.3 错误处理

定义了四种错误：

- Missing arithmetic object（缺少运算对象）
- 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。

错误的具体处理流程如下：

1. 发现错误：(X 为栈顶元素，a 为 ip 指向的字符)
   (a) $X \in V_T$，但 $X \neq a$；
   (b) $X \in V_N$，但 M[X,a] 为 synch。
   (c) $X \in V_N$，但 M[X,a] 为空。
2. 处理方法：
   (a) 弹出 X；
   (b) 弹出 X；
   (c) 跳过 a。

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 模块的方法将文法转换为 LL(1) 文法
3. 调用 table 模块的方法构建 LL(1) 文法的预测分析表
4. 用户输入待预测分析的算数表达式，执行预测分析控制程序

2.4.1 预测分析控制程序

根据栈顶符号 X 和当前输入符号 a，分析动作有 4 种可能：

1. $X = a = \$$，宣告分析成功，停止分析；
2. $X = a \neq \$$，从栈顶弹出 X，输入指针前移一个位置；
3. $X \in V_T$，但 $X \neq a$，发现错误，调用错误处理程序，报告错误及进行错误恢复；
4. 若 $X \in V_N$，访问分析表 $M[X, a]$

(a) $M[X,a] = X \to Y_1Y_2...Y_n$ 先将 X 从栈顶弹出，然后把产生式的右部符号串按反序推入栈中（即按 $Y_n, Y_{n-1}, ...Y_2, Y_1$ 的顺序）；

(b) $M[X,a] = X \to \varepsilon$ 从栈顶弹出 X；

(c) $M[X,a] = error$ 调用错误处理程序。

算法的伪代码如下：

```
输入：输入符号串ω，文法G的预测分析表M。
输出：若ω ∈L(G)，则输出ω的最左推导，否则报告错误。
方法：分析开始时，$在栈底，文法开始符号S在栈顶，ω$在输入缓冲区中。
      置ip指向 ω$ 的第一个符号；
    do {                          // X是栈顶符号，a是ip所指向的符号
       if (X是终结符号或$) {
          if (X==a) { 从栈顶弹出X;   ip前移一个位置； }
          else  error();
       else                       // X是非终结符号
          if (M[X, a]=X→Y1Y2…Yk ) {
             从栈顶弹出X;  把Yk、Yk-1、…、Y2、Y1压入栈，Y1在栈顶;
             输出产生式X→Y1Y2…Yk ;
          }
          else  error();
    } while(X!=$)                 // 栈不空，继续
```

图 4: LL(1) 预测分析控制程序

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。程序会读入输入的文法，并将其转换成相应的 LL(1) 文法。程序会依次输出消除左递归后的文法，提取左公因子后的文法，FIRSR 集，FOLLOW 集，文法是否为 LL(1) 文法的判断以及预测分析表的内容。

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，程序将其转换为相应的 LL(1) 文法：

```
Please input the grammar file name: grammar.txt
====After Eliminate Left Recursion====
start symbol: E
```

```
E' -> epsilon | + T E' | - T E'
T' -> epsilon | * F T' | / F T'
F -> ( E ) | num
T -> F T'
E -> T E'

====After Extract Left Common====
start symbol: E
E' -> epsilon | + T E' | - T E'
T' -> epsilon | * F T' | / F T'
F -> ( E ) | num
T -> F T'
E -> T E'

====FIRST Sets====
E': - + epsilon
T': / * epsilon
F: num (
T: num (
E: num (

====FOLLOW Sets====
E': $ )
T': - + $ )
F: ) - + * $ /
T: ) $ + -
E: ) $

The grammar can be converted to a LL(1) grammar.

====Parsing Table====
E' [ ( ): E' -> epsilon ) ( num: error ) ( (: error ) ( $: E' -> epsilon ) (
↪  /: error ) ( -: E' -> - T E' ) ( *: error ) ( +: E' -> + T E' ) ]
T' [ ( ): T' -> epsilon ) ( num: error ) ( (: error ) ( $: T' -> epsilon ) (
↪  /: T' -> / F T' ) ( -: T' -> epsilon ) ( *: T' -> * F T' ) ( +: T' ->
↪  epsilon ) ]
F [ ( ): error ) ( num: F -> num ) ( (: F -> ( E ) ) ( $: error ) ( /: error
↪  ) ( -: error ) ( *: error ) ( +: error ) ]
```

```
T [ ( ): error ) ( num: T -> F T' ) ( (: T -> F T' ) ( $: error ) ( /: error
↪  ) ( -: error ) ( *: error ) ( +: error ) ]
E [ ( ): error ) ( num: E -> T E' ) ( (: E -> T E' ) ( $: error ) ( /: error
↪  ) ( -: error ) ( *: error ) ( +: 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

```
E -> T E'
T -> F T'
F -> ( E )
E -> T E'
T -> F T'
F -> num
T' -> epsilon
E' -> + T E'
T -> F T'
F -> ( E )
E -> T E'
T -> F T'
F -> num
T' -> * F T'
F -> num
T' -> epsilon
E' -> epsilon
T' -> epsilon
E' -> epsilon
T' -> / F T'
F -> num
T' -> epsilon
E' -> epsilon
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 的最左推导，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

```
E -> T E'
T -> F T'
F -> ( E )
E -> T E'
T -> F T'
F -> ( E )
E -> T E'
T -> F T'
F -> num
T' -> epsilon
E' -> + T E'
T -> F T'
F -> ( E )
E -> T E'
T -> F T'
F -> num
T' -> * F T'
F -> ( E )
E -> T E'
T -> F T'
F -> num
T' -> / F T'
F -> num
T' -> epsilon
E' -> + T E'
```

```
T -> F T'
F -> num
T' -> epsilon
E' -> epsilon
T' -> epsilon
E' -> - T E'
T -> F T'
F -> ( E )
E -> T E'
T -> F T'
F -> num
T' -> epsilon
E' -> + T E'
T -> F T'
F -> num
T' -> epsilon
E' -> epsilon
T' -> / F T'
F -> ( E )
E -> T E'
T -> F T'
F -> num
T' -> epsilon
E' -> - T E'
T -> F T'
F -> num
T' -> epsilon
E' -> epsilon
T' -> epsilon
E' -> epsilon
T' -> * F T'
F -> ( E )
E -> T E'
T -> F T'
F -> num
T' -> epsilon
E' -> + T E'
T -> F T'
```

```
F -> num
T' -> epsilon
E' -> + T E'
T -> F T'
F -> num
T' -> * F T'
F -> num
T' -> epsilon
E' -> epsilon
T' -> epsilon
E' -> epsilon
T' -> / F T'
F -> ( E )
E -> T E'
T -> F T'
F -> num
T' -> * F T'
F -> ( E )
E -> T E'
T -> F T'
F -> num
T' -> epsilon
E' -> + T E'
T -> F T'
F -> num
T' -> epsilon
E' -> epsilon
T' -> epsilon
E' -> epsilon
T' -> epsilon
E' -> epsilon
T' -> epsilon
E' -> epsilon
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)))的最左推导，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

```
E -> T E'
T -> F T'
F -> ( E )
E -> T E'
T -> F T'
F -> num
T' -> epsilon
E' -> + T E'
T -> F T'
F -> num
T' -> epsilon
E' -> epsilon
Error: Missing closing parenthese!
T' -> epsilon
E' -> epsilon
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

```
E -> T E'
T -> F T'
F -> num
T' -> epsilon
E' -> + T E'
T -> F T'
F -> num
T' -> epsilon
E' -> epsilon
Error: Missing opening parenthese or operator!
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: Missing opening parenthese or operator!`。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

```
E -> T E'
T -> F T'
F -> num
Error: Missing operator!
Error: Missing arithmetic object!
T' -> epsilon
E' -> + T E'
T -> F T'
```

```
F -> num
T' -> epsilon
E' -> epsilon
Error: Missing opening parenthese or operator!
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: Missing operator !。程序还输出了Error: Missing arithmetic object!和Error: Missing opening parenthese or operator!，这是因为程序对该类错误的恢复机制是：1(2+3)-> 1 op num op (2+3)。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!
E -> T E'
T -> F T'
F -> ( E )
E -> T E'
T -> F T'
F -> num
T' -> epsilon
E' -> + T E'
T -> F T'
F -> num
T' -> epsilon
E' -> epsilon
T' -> epsilon
E' -> epsilon
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，程序将其转换为相应的 LL(1) 文法：

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

====After Extract Left Common====
start symbol: S
L' -> epsilon | , S L'
S -> ( L ) | a
L -> S L'

====FIRST Sets====
L': , epsilon
S: a (
L: a (

====FOLLOW Sets====
L': )
S: ) , $
L: )

The grammar can be converted to a LL(1) grammar.

====Parsing Table====
L' [ ( $: error ) ( a: error ) ( (: error ) ( ,: L' -> , S L' ) ( ): L' ->
↪  epsilon ) ]
S [ ( $: error ) ( a: S -> a ) ( (: S -> ( L ) ) ( ,: error ) ( ): error ) ]
L [ ( $: error ) ( a: L -> S L' ) ( (: L -> S L' ) ( ,: error ) ( ): 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 -> ( L )
L -> S L'
S -> a
L' -> , S L'
S -> ( L )
L -> S L'
S -> a
L' -> , S L'
S -> ( L )
L -> S L'
S -> a
L' -> , S L'
S -> a
L' -> epsilon
L' -> epsilon
L' -> epsilon
success!
```

4.9.3 结果分析

经过验证，测试结果为表达式 (a,(a,(a,a))) 的最左推导，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 LL(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 LL(1) 语法分析方法的理解，掌握了预测分析表的构造和 LL(1) 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。