北京郵電大學

实验报告：语法分析程序的设计与实现

——LL 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 15 号

目录

1. 实验概述.....1
1.1. 实验题目及要求.....1
1.2. 实验环境.....1
2. 程序设计说明.....2
2.1. 程序结构设计概述.....2
2.1.1. Grammar 文法类.....3
2.1.2. LL(1)分析器.....4
2.2. 程序具体逻辑说明.....5
2.2.1. 消除左递归及左公因子.....5
2.2.2. 计算 FIRST 集.....6
2.2.3. 计算 FOLLOW 集.....7
2.2.4. 构造预测分析表.....9
2.2.5. 进行 LL(1)分析.....10
3. 测试设计与分析.....12
3.1. 测试方法及辅助信息说明.....12
3.2. 测试 1+2.....14
3.3. 测试 1+2*(3-4).....15
3.4. 测试 0+1+2*(3-(4/2)).....16
3.5. 测试 1+2*/(3-4).....18
4. 总结心得.....19

1.实验概述

1.1. 实验题目及要求

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→ E+T | E－T | T

T→ T*F | T/F | F

F→ (E) | 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. 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。

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

本程序实现了一个 LL(1) 语法分析器，主要包括两个核心模块：Grammar 文法类和 LL(1) 分析器。程序分为以下文件：

1. **Grammar.h / Grammar.cpp**：定义并实现 Grammar 类，用于表示和操作文法，包括添加产生式、设置起始符号，以及对文法进行左递归和左公因子的消除。
2. **LL1Parser.h / LL1Parser.cpp**：定义并实现 LL1Parser 类，用于 LL(1) 语法分析的核心功能，包括计算 FIRST 集、FOLLOW 集，构建预测分析表，并对输入表达式进行 LL(1) 语法分析。
3. **main.cpp**：程序入口文件，负责创建文法和 LL(1) 分析器对象，调用相关方法执行文法处理和分析工作，并输出结果。此文件用于测试文法和分析器的功能。

程序的主要步骤包括创建并初始化文法，消除左递归和左公因子，计算 FIRST 和 FOLLOW 集，构建预测分析表，最后使用 LL(1) 分析器对输入表达式进行语法分析。

下面将分别概述 Grammar 文法类和 LL1Parser LL(1) 分析器的结构和功能。

2.1.1.Grammar 文法类

Grammar 类用于定义和处理文法。其主要职责包括存储文法的非终结符、终结符、产生式，以及提供操作文法的功能。Grammar 类的主要功能和方法如下：

1. 数据成员：

- nonTerminals：表示文法的非终结符集合。
- terminals：表示文法的终结符集合。
- productions：表示文法的产生式集合，使用映射的方式存储，每个非终结符对应多个产生式。
- startSymbol：文法的起始符号。

2. 方法：

- addProduction(nonTerminal, production)：用于向文法中添加一个产生式，其中 nonTerminal 是产生式的左部，production 是右部。此方法会将产生式按左部的非终结符存储在 productions 映射中。
- eliminateLeftRecursion()：消除左递归的方法。对于某个非终结符的产生式，如果存在左递归，会进行处理，将左递归形式转换为等价的无左递归形式。
- eliminateLeftFactor()：消除左公因子的方法。对于某个非终结符的产生式，如果多个产生式具有相同的前缀（即左公因子），此方法会提取出左公因子，形成新的非终结符，重写产生式。
- printGrammar()：打印文法的产生式，用于显示文法的当前结构。

Grammar 类提供了消除左递归和左公因子的方法，以确保文法可以适用于 LL(1) 解析器。

2.1.2.LL(1)分析器

LL1Parser 类是程序的核心，用于实现 LL(1) 语法分析的各项功能，包括计算 FIRST 和 FOLLOW 集、构建预测分析表，以及对输入表达式进行解析。

主要功能如下：

1. **数据成员：**
   - first：用于存储每个符号的 FIRST 集。FIRST 集包含了文法中每个非终结符在推导时可以首先出现的终结符集合。
   - follow：用于存储每个非终结符的 FOLLOW 集。FOLLOW 集包含了每个非终结符在文法中可能跟随的终结符集合。
   - predictionsTable：预测分析表，用于存储每个非终结符和终结符的预测产生式。这是实现 LL(1) 分析的核心数据结构。
2. **公共方法：**
   - computeFirst(Grammar& grammar)：计算并存储文法中每个符号的 FIRST 集。
   - computeFollow(Grammar& grammar)：计算并存储文法中每个非终结符的 FOLLOW 集。
   - computePredictionsTable(Grammar& grammar)：根据 FIRST 集和 FOLLOW 集构建预测分析表。
   - isLL1(const Grammar& grammar) const：检查文法是否符合 LL(1) 的条件，即预测分析表中的每个单元格最多只有一个产生式。
   - parse(const Grammar& grammar, const std::string& inputString)：使用预测分析表对输入表达式进行 LL(1) 语法分析。该方法输出分析过程，显示栈状态、输入状态和推导步骤。
   - printFollow() 和 printPredictionsTable(Grammar& grammar)：分别用于输出 FOLLOW 集和预测分析表，便于调试和验证结果。

LL1Parser 类通过计算 FIRST 和 FOLLOW 集来判断文法的 LL(1) 性质，并生成预测分析表。其 parse 方法使用预测分析表对输入表达式进行解析，以模拟 LL(1) 分析过程。

2.2. 程序具体逻辑说明

2.2.1.消除左递归及左公因子

在 LL(1) 分析中，文法必须满足 LL(1) 的条件，即不能有左递归和左公因子。因此，首先需要对原始文法进行预处理，消除左递归和左公因子。Grammar 类的 eliminateLeftRecursion 和 eliminateLeftFactor 方法分别实现了这两个功能。

1. 消除左递归：

左递归指的是产生式中存在 $A \to A\alpha|\beta$ 形式，其中 A 是非终结符，α 和 β 是符号串。当存在左递归时，递归的形式会导致 LL(1) 分析器进入无限循环。

为了消除左递归，对于每一个非终结符 A 的左递归产生式，可以将其转换为无左递归形式。转换后的形式为：

$$A \to \beta A'$$

$$A' \to \alpha A' \mid \varepsilon$$

在 eliminateLeftRecursion 方法中，首先识别每个非终结符的左递归产生式，接着按照上述规则将其分解成新的无左递归形式，生成一个新的非终结符 A' 以及新的产生式集合。

2. 消除左公因子：

左公因子指的是多个产生式之间存在相同的前缀。例如， $A \to \alpha\beta 1|\alpha\beta 2$ 具有共同前缀 α。当存在左公因子时，无法确定 LL(1) 分析器应该选择哪条产生式。

为了消除左公因子，将相同前缀提取出来，重构产生式，形式如下：

$$A \to \alpha\, A'$$

$$A' \to \beta 1|\beta 2$$

在 eliminateLeftFactor 方法中，识别每个非终结符的产生式是否具有相同前缀，如果有，则将前缀提取为新的非终结符，并重新组织产生式结构。

具体代码实现可见附件 Grammar.cpp.

2.2.2.计算 FIRST 集

FIRST 集是 LL(1) 语法分析中的一个重要概念，用于确定一个非终结符在推导时最先可能出现的终结符集合。

计算逻辑

1. **初始化**：在 computeFirst 方法中，首先为每个非终结符初始化一个空集合 first[nonTerminal]，然后为每个终结符的 FIRST 集初始化为其自身。例如，终结符 a 的 FIRST(a) = {a}。
2. **递归计算**：

- calculateFirst 方法用于递归计算某个非终结符的 FIRST 集。它遍历该非终结符的每个产生式，检查产生式右部的第一个符号。
- 如果第一个符号是终结符，则直接将其加入 FIRST 集。例如，若有产生式 $A \to aB$，则 a ∈ FIRST(A)。
- 如果第一个符号是非终结符，则需要递归地计算该非终结符的 FIRST 集，并将其加入 FIRST(A)。例如，若有产生式 $A \to B$C，则需要将 FIRST(B) 加入 FIRST(A)。
- 如果 FIRST(B) 包含空串 ε，则继续查看 B 后的符号 C，将 FIRST(C) 中的所有非空符号加入 FIRST(A)。这一过程会递归地处理产生式右部的符号，直到不再出现 ε 或达到右部末尾。

3. **处理空串**：

- 若某个非终结符的产生式右部可以推导出空串 ε，则将 ε 加入该非终结符的 FIRST 集。例如，若 $A \to \varepsilon$ 或 $A \to B$，且 ε∈FIRST(B)，则 ε∈FIRST(A)。
- 在递归过程中，若某个符号的 FIRST 集中包含 ε，则需要从该符号的后续符号继续计算 FIRST 集。

代码逻辑

- computeFirst：负责初始化每个符号的 FIRST 集并调用 calculateFirst 方法，逐个计算每个非终结符的 FIRST 集。
- calculateFirst：具体负责递归计算 FIRST 集的内容。对每个产生式右部逐个字符进行分析，并处理终结符、非终结符以及空串的情况，确保所有符号的 FIRST 集都正确构建。

2.2.3.计算 FOLLOW 集

FOLLOW 集在 LL(1) 语法分析中用于确定每个非终结符在出现时可能跟随的终结符。FOLLOW 集的计算与 FIRST 集密切相关，它帮助确定非终结符在不同上下文中的作用。

LL1Parser 类的 computeFollow 方法用于计算文法中每个非终结符的 FOLLOW 集。

计算逻辑

1. 初始化：

- 在 computeFollow 方法中，首先为每个非终结符初始化一个空的 FOLLOW 集。
- 对起始符号的 FOLLOW 集进行特殊处理，默认包含终结符 $，表示输入结束。

2. 规则应用：

- FOLLOW 集的计算遵循以下规则：
  1. **规则 1：**对于产生式 $A \to \alpha B\beta$，将 $FIRST(\beta) - \{\varepsilon\}$ 加入 FOLLOW(B)。即在 A 的产生式中，如果 B 后面存在符号 β，那么 B 的 FOLLOW 集包含 β 的所有非空 FIRST 符号。
  2. **规则 2：**如果 B 是产生式 $A \to \alpha B$ 或 $A \to \alpha B\beta$ 且 $\varepsilon \in FIRST(\beta)$，则将 FOLLOW(A) 中的所有符号加入 FOLLOW(B)。即如果 B 位于产生式的结尾或其后续可以为空，则 FOLLOW(B) 包含 FOLLOW(A) 中的所有符号。

3. 迭代求解：

- FOLLOW 集的计算是一个迭代过程，直到所有 FOLLOW 集不再发生变化为止。每次迭代中，根据以上规则对每个产生式进行检查，并更新相应的 FOLLOW 集。
- 通过反复应用这些规则，逐步计算出每个非终结符的完整 FOLLOW 集。

代码逻辑

- computeFollow 方法负责整体的 FOLLOW 集计算。它为起始符号初始化 FOLLOW 集为 { $ }，然后在每轮迭代中遍历所有产生式。

- 在遍历过程中：
  - 对每个产生式右部的每个符号进行检查，判断其后续符号并根据规则更新 FOLLOW 集。
  - 如果 FOLLOW 集发生变化，继续进行迭代，直到没有进一步的更新为止。

举例说明

假设文法如下：

$$E \to TE'$$

$$E' \to +T\,E' \mid \varepsilon$$

$$T \to FT'$$

$$T' \to *\,F\,T' \mid \varepsilon$$

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

计算 FOLLOW 集的过程如下：

1. FOLLOW(E) 包含 $（因为 E 是起始符号）。
2. 对于 FOLLOW(E')，在 $E \to TE'$ 中，E' 位于末尾，因此 FOLLOW(E') 包含 FOLLOW(E)，即 $。
3. 对于 FOLLOW(T)，在 $E' \to +TE'$ 中，T 后面跟随 E'，因此 FOLLOW(T) 包含 FIRST(E')（即 + 和 ε），并且 FOLLOW(T) 包含 FOLLOW(E')（即 $）。
4. 对于 FOLLOW(F)，在 $T \to FT'$ 中，T' 跟随 F，所以 FOLLOW(F) 包含 FIRST(T')（即 * 和 ε），并且如果 T' 可以为空，则 FOLLOW(F) 包含 FOLLOW(T)。

最终得到的 FOLLOW 集用于构建预测分析表，以确定非终结符的分析规则。

具体的代码实现可见附件 LL1Parser.cpp

2.2.4.构造预测分析表

在 LL(1) 语法分析中，预测分析表用于决定在每个步骤选择哪个产生式来进行解析。LL1Parser 类的 computePredictionsTable 方法用于构建预测分析表，表中每个单元格记录了一个非终结符在遇到某个终结符时应使用的产生式。

构造逻辑

构造预测分析表的基本逻辑是利用 FIRST 和 FOLLOW 集，按照以下规则为每个非终结符和终结符对填充表项：

1. **FIRST 集规则：**
   - 对于每个产生式 $A \to \alpha$，将 FIRST(α) - {ε} 中的所有终结符加入 $M[A, a]$，并在相应表项中记录产生式 $A \to \alpha$。
   - 例如，如果 $A \to aB$ 且 $FIRST(aB)$ 包含终结符 a，则 $M[A, a] = A \to aB$。
2. **FOLLOW 集规则：**
   - 如果产生式 $A \to \alpha$ 可以推导出空串 ε（即 $\varepsilon \in FIRST(\alpha)$），则对 FOLLOW(A) 中的每个终结符 b，在 $M[A, b]$ 中记录 $A \to \alpha$。
   - 例如，如果 $A \to \alpha$ 且 $\varepsilon \in FIRST(\alpha)$，并且 FOLLOW(A) 包含 b，则 $M[A, b] = A \to \alpha$。
3. **错误同步：**
   - 如果某个非终结符 A 的 FOLLOW 集中有符号 c，但预测分析表 $M[A, c]$ 中没有相应的产生式，则在 $M[A, c]$ 中填入 "synch" 表示同步（即错误恢复）。这种情况下，分析器会在出错时跳过 A。

代码逻辑

- computePredictionsTable 方法通过遍历每个非终结符的产生式，构建预测分析表。
- 对于每个产生式，根据 FIRST 集填入对应的预测表项，并根据 FOLLOW 集填入可能的空串产生式。
- 最后一步，为每个非终结符的 FOLLOW 集中存在但没有在表中填充的终结符位置填入 "synch"，表示同步错误恢复。

2.2.5.进行 LL(1)分析

在完成 FIRST 集、FOLLOW 集的计算和预测分析表的构建后，便可以使用预测分析表对输入字符串进行 LL(1) 语法分析。LL1Parser 类的 parse 方法负责使用预测分析表对输入表达式进行解析，并输出分析过程的每一步。

分析逻辑

1. 初始化：

- 初始化分析栈，将结束标记 $ 和起始符号（文法的开始符号）依次压入栈。
- 初始化输入指针，用于逐步读取输入字符串中的字符。

2. 解析过程：

- 进入解析循环，每次循环中，检查分析栈的栈顶元素和当前输入符号。
- 根据栈顶符号类型（终结符或非终结符）和当前输入符号，执行以下操作：

  1. 栈顶是终结符：

  - 如果栈顶是终结符且与当前输入符号相同，说明匹配成功，弹出栈顶符号并移动输入指针。
  - 如果栈顶是终结符且与当前输入符号不同，说明出现错误，记录错误并从栈中移除栈顶符号（或进行其他错误恢复策略）。

  2. 栈顶是非终结符：

  - 当栈顶为非终结符时，根据栈顶符号和当前输入符号查找预测分析表。
  - 若表中存在对应的产生式，则将栈顶符号弹出，并将产生式右部（反序）压入栈。
  - 若表项为 "synch"，表示同步错误，记录错误并弹出栈顶符号，以恢复同步。
  - 若表项为空或不存在，表示解析表缺少该符号的处理规则，记录错误并跳过当前输入符号。

3. 终止条件：

- 当栈顶符号和当前输入符号均为 $ 时，表示解析完成，成功匹配整个输入。
- 若栈为空或输入未读完时遇到错误，则解析失败。

错误处理

- 在 LL(1) 解析过程中，若发现预测分析表缺少对应规则或出现同步标记

"synch"，则记录错误并弹出栈顶符号或跳过输入符号。

- 错误处理策略允许解析器在遇到简单错误时尽量恢复同步，继续解析剩余部分。

代码逻辑

- parse 方法通过栈和输入指针模拟 LL(1) 分析过程。
- 每一步都会根据栈顶符号和输入符号的组合，从预测分析表中查询并执行相应的产生式。
- 在分析过程中，输出当前栈、输入状态和解析操作的详细信息，便于调试和理解 LL(1) 分析的每一步。

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：

```
// 添加产生式
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");

// Step 5: 测试解析一组输入表达式
std::vector<std::string> testStrings = {
    "1+2",              // 简单的加法表达式
    "1+2*(3-4)",        // 带括号的复合表达式
    "0+1+2*(3-(4/2))",  // 复杂的嵌套表达式
    "1+2*/(3-4)",       // 包含错误的表达式
};
```

2. 编译生成可执行文件并执行

```
> g++ -o main main.cpp Grammar.cpp LL1Parser.cpp
> ./main
```

3. 消除左递归&左公因子信息

输出如下：

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

Grammar after eliminating left recursion and left factoring:
E -> TE'
E' -> +TE' | -TE' | ε
F -> (E) | num
T -> FT'
T' -> *FT' | /FT' | ε
```

4. FIRST 集和 FOLLOW 集信息

```
First Set:
First(E) = { ( num }
First(E') = { + - ε }
First(F) = { ( num }
First(T) = { ( num }
First(T') = { * / ε }

Follow Set:
Follow(E) = { $ ) }
Follow(E') = { $ ) }
Follow(F) = { $ ) * + - / }
Follow(T) = { $ ) + - }
Follow(T') = { $ ) + - }
```

5. 预测分析表输出

Predictions Table:

	(	)	*	+	-	/	num	$
E	E -> TE'	E -> synch					E -> TE'	E -> synch
E'		E' -> ε		E' -> +TE'	E' -> -TE'			E' -> ε
F	F -> (E)	F -> synch	F -> synch	F -> synch	F -> synch	F -> synch	F -> num	F -> synch
T	T -> FT'	T -> synch		T -> synch	T -> synch		T -> FT'	T -> synch
T'		T' -> ε	T' -> *FT'	T' -> ε	T' -> ε	T' -> /FT'		T' -> ε

3.2. 测试 1+2

测试目的：

该测试用例是一个最简单的加法表达式，目的是验证 LL(1) 分析器是否能够正确解析最基础的表达式结构，包括非终结符的推导和终结符的匹配。这是所有后续复杂测试的基础，确保基本功能无误。

测试预期：

- 正确匹配产生式，并顺利完成解析。
- 栈操作、输入指针移动、以及输出的推导步骤应符合 LL(1) 分析逻辑。
- 输出结果为 Complete! No error。

测试输出：

```
Parsing expression: 1+2
Stack                   Input                   Output
$E                      1+2$                    E -> TE'
$E'T                    1+2$                    T -> FT'
$E'T'F                  1+2$                    F -> num
$E'T'num                1+2$
$E'T'                   +2$                     T' -> ε
$E'                     +2$                     E' -> +TE'
$E'T+                   +2$
$E'T                    2$                      T -> FT'
$E'T'F                  2$                      F -> num
$E'T'num                2$
$E'T'                   $                       T' -> ε
$E'                     $                       E' -> ε
$                       $
Complete!       No error
```

经对比分析，该分析过程完全正确。

3.3. 测试 1+2*(3-4)

测试目的：

该测试用例是一个复合表达式，包含加法、乘法和括号。目的是验证 LL(1) 分析器是否能够正确处理操作符优先级，以及括号内嵌套表达式的解析。

测试预期：

- 能正确识别括号的匹配关系和优先级。
- 在遇到嵌套表达式时，能按文法规则逐步解析内部结构。
- 输出结果为 Complete! No error。

测试输出：

```
Parsing expression: 1+2*(3-4)
Stack                   Input                   Output
$E                      1+2*(3-4)$              E -> TE'
$E'T                    1+2*(3-4)$              T -> FT'
$E'T'F                  1+2*(3-4)$              F -> num
$E'T'num                1+2*(3-4)$
$E'T'                   +2*(3-4)$               T' -> ε
$E'                     +2*(3-4)$               E' -> +TE'
$E'T+                   +2*(3-4)$
$E'T                    2*(3-4)$                T -> FT'
$E'T'F                  2*(3-4)$                F -> num
$E'T'num                2*(3-4)$
$E'T'                   *(3-4)$                 T' -> *FT'
$E'T'F*                 *(3-4)$
$E'T'F                  (3-4)$                  F -> (E)
$E'T')E(                (3-4)$
$E'T')E                 3-4)$                   E -> TE'
$E'T')E'T               3-4)$                   T -> FT'
$E'T')E'T'F             3-4)$                   F -> num
$E'T')E'T'num           3-4)$
$E'T')E'T'              -4)$                    T' -> ε
$E'T')E'                -4)$                    E' -> -TE'
$E'T')E'T-              -4)$
$E'T')E'T               4)$                     T -> FT'
$E'T')E'T'F             4)$                     F -> num
$E'T')E'T'num           4)$
$E'T')E'T'              )$                      T' -> ε
$E'T')E'                )$                      E' -> ε
$E'T')                  )$
$E'T'                   $                       T' -> ε
$E'                     $                       E' -> ε
$                       $
Complete!       No error
```

经对比分析，该分析过程完全正确。

3.4. 测试 0+1+2*(3-(4/2))

测试目的：

该测试用例是一个复杂的嵌套表达式，包含多种操作符和层级较深的括号嵌套。目的是验证 LL(1) 分析器在面对复杂表达式时的鲁棒性和正确性。

测试预期：

- 能正确解析嵌套结构，并逐层消化括号内部的表达式。
- 能处理多种操作符优先级（加法、乘法、除法）和嵌套结构的混合。
- 输出结果为 Complete! No error。

测试输出：

```
Parsing expression: 0+1+2*(3-(4/2))
Stack                   Input                   Output
$E                      0+1+2*(3-(4/2))$        E -> TE'
$E'T                    0+1+2*(3-(4/2))$        T -> FT'
$E'T'F                  0+1+2*(3-(4/2))$        F -> num
$E'T'num                0+1+2*(3-(4/2))$
$E'T'                   +1+2*(3-(4/2))$         T' -> ε
$E'                     +1+2*(3-(4/2))$         E' -> +TE'
$E'T+                   +1+2*(3-(4/2))$
$E'T                    1+2*(3-(4/2))$          T -> FT'
$E'T'F                  1+2*(3-(4/2))$          F -> num
$E'T'num                1+2*(3-(4/2))$
$E'T'                   +2*(3-(4/2))$           T' -> ε
$E'                     +2*(3-(4/2))$           E' -> +TE'
$E'T+                   +2*(3-(4/2))$
$E'T                    2*(3-(4/2))$            T -> FT'
$E'T'F                  2*(3-(4/2))$            F -> num
$E'T'num                2*(3-(4/2))$
$E'T'                   *(3-(4/2))$             T' -> *FT'
$E'T'F*                 *(3-(4/2))$
$E'T'F                  (3-(4/2))$              F -> (E)
$E'T')E(                (3-(4/2))$
$E'T')E                 3-(4/2))$               E -> TE'
$E'T')E'T               3-(4/2))$               T -> FT'
$E'T')E'T'F             3-(4/2))$               F -> num
$E'T')E'T'num           3-(4/2))$
$E'T')E'T'              -(4/2))$                T' -> ε
$E'T')E'                -(4/2))$                E' -> -TE'
$E'T')E'T-              -(4/2))$
$E'T')E'T               (4/2))$                 T -> FT'
```

```
$E'T')E'T'F             (4/2))$             F -> (E)
$E'T')E'T')E(           (4/2))$
$E'T')E'T')E            4/2))$              E -> TE'
$E'T')E'T')E'T          4/2))$              T -> FT'
$E'T')E'T')E'T'F        4/2))$              F -> num
$E'T')E'T')E'T'num      4/2))$
$E'T')E'T')E'T'         /2))$               T' -> /FT'
$E'T')E'T')E'T'F/       /2))$
$E'T')E'T')E'T'F        2))$                F -> num
$E'T')E'T')E'T'num      2))$
$E'T')E'T')E'T'         ))$                 T' -> ε
$E'T')E'T')E'           ))$                 E' -> ε
$E'T')E'T')             ))$
$E'T')E'T'              )$                  T' -> ε
$E'T')E'                )$                  E' -> ε
$E'T')                  )$
$E'T'                   $                   T' -> ε
$E'                     $                   E' -> ε
$                       $
Complete!       No error
```

经对比分析，该分析过程完全正确。

3.5. 测试 1+2*/(3-4)

测试目的：

该测试用例包含一个语法错误，即乘法操作符 * 后直接跟随除法操作符 /。目的是验证 LL(1) 分析器的错误检测能力以及在错误出现后的同步机制。

测试预期

- 能正确检测到语法错误，并提示错误位置。
- 能通过同步机制跳过错误，继续解析剩余部分的结构。
- 输出结果为 Complete! 1 error(s)

测试输出：

```
Parsing expression: 1+2*/(3-4)
Stack                   Input                   Output
$E                      1+2*/(3-4)$             E -> TE'
$E'T                    1+2*/(3-4)$             T -> FT'
$E'T'F                  1+2*/(3-4)$             F -> num
$E'T'num                1+2*/(3-4)$
$E'T'                   +2*/(3-4)$              T' -> ε
$E'                     +2*/(3-4)$              E' -> +TE'
$E'T+                   +2*/(3-4)$
$E'T                    2*/(3-4)$               T -> FT'
$E'T'F                  2*/(3-4)$               F -> num
$E'T'num                2*/(3-4)$
$E'T'                   */(3-4)$                T' -> *FT'
$E'T'F*                 */(3-4)$
$E'T'F                  /(3-4)$                 Error: predictions_table[F, /] is synch
$E'T'                   /(3-4)$                 T' -> /FT'
$E'T'F/                 /(3-4)$
$E'T'F                  (3-4)$                  F -> (E)
$E'T')E(                (3-4)$
$E'T')E                 3-4)$                   E -> TE'
$E'T')E'T               3-4)$                   T -> FT'
$E'T')E'T'F             3-4)$                   F -> num
$E'T')E'T'num           3-4)$
$E'T')E'T'              -4)$                    T' -> ε
$E'T')E'                -4)$                    E' -> -TE'
$E'T')E'T-              -4)$
$E'T')E'T               4)$                     T -> FT'
$E'T')E'T'F             4)$                     F -> num
$E'T')E'T'num           4)$
$E'T')E'T'              )$                      T' -> ε
$E'T')E'                )$                      E' -> ε
$E'T')                  )$
$E'T'                   $                       T' -> ε
$E'                     $                       E' -> ε
$                       $
Complete!       1 error(s)
```

输出完全正确。且该输出结果显示了分析过程中的错误情况，也展示了错误恢复的能力，使得分析过程能够继续进行直到完成。

4.总结心得

通过本次 LL(1) 语法分析器的设计与实现，我对编译原理中的语法分析部分有了更加深入的理解，并切实感受到了将理论应用于实践的挑战与乐趣。

本次实验的核心内容包括文法的预处理、FIRST 和 FOLLOW 集的计算、预测分析表的构造以及 LL(1) 分析器的实现，每一步都紧密相连，为最终完成语法解析奠定了坚实的基础。

实验的第一步是对文法进行预处理，以消除左递归和左公因子，使其符合 LL(1) 文法的要求。这一步让我深刻体会到编译原理中文法形式化的重要性，也加深了我对递归结构和提取公共因子的理解。随后，我实现了 FIRST 和 FOLLOW 集的计算。在这一过程中，我深刻体会到递归定义与迭代算法的结合所带来的逻辑严密性，以及在实现复杂文法时自动化计算的重要性。

预测分析表的构造则是实验中的一个亮点，通过将 FIRST 和 FOLLOW 集的规则相结合，我成功地将抽象的文法规则转化为具体的解析策略。错误同步机制的引入进一步增强了解析器的鲁棒性，使其在面对语法错误时能够部分恢复，继续完成后续解析。

在调试过程中，我遇到了诸多问题，例如如何处理空串 ε，如何动态调整栈结构，以及如何优雅地实现错误恢复。这些问题虽然一度增加了实验的复杂度，但也正是在解决这些问题的过程中，我对 LL(1) 语法分析的原理和实践有了更深的理解。

总的来说，这次实验不仅帮助我们掌握了 LL(1) 分析器的设计与实现，还让我们感受到程序设计中逻辑严谨性的重要性，以及将理论转化为实践的挑战与成就感。