

编译原理程序设计报告

程序设计 2：算术表达式语法分析器

2021211110 杜抒泽

2023 年 11 月 11 日

目录

1. 实验要求	2
2. 程序设计说明	2
2.1. 实现的基本功能	2
2.2. 开发、构建与运行	2
2.3. 程序设计介绍与功能细节	3
2.3.1. 词法 Token 与语法 Symbol	3
2.3.2. 文法相关类型定义	3
2.3.3. 从字符串中获取文法	5
2.3.4. 文法相关算法	6
2.3.4.1. 消除左递归	6
2.3.4.2. 提取左公因子	7
2.3.4.3. 构造 FIRST 集	9
2.3.4.4. 构造 FOLLOW 集	10
2.3.4.5. 判断是否为 LL(1) 文法	11
2.3.4.6. 构造预测分析表	13
2.3.5. 语法分析与错误处理	14
3. 测试、运行结果与分析说明	15
3.1. 对于给定文法的变换与分析	15
3.2. 对于任给的算术表达式的语法分析	16
4. 总结	24

1. 实验要求

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

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

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

$$F \rightarrow (E) \mid \text{num}$$

实验要求：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。

方法 1：编写递归调用程序实现自顶向下的分析。

方法 2：编写 LL(1) 语法分析程序，要求如下。

- (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

2.1. 实现的基本功能

本次程序设计选取方法 2。使用 C++ 完成算术表达式语法分析器的编写。程序完成了：

1. 对给定文法进行消除左递归，提取左公因子，构造 FIRST 集、FOLLOW 集，判断是否是 LL(1) 文法。
2. 对给定的 LL(1) 文法构造预测分析表。
3. 对给定的表达式进行词法分析。
4. 基于构造的预测分析表对给定的表达式进行语法分析，输出每一步的栈和输入串状态，以及使用的产生式。

2.2. 开发、构建与运行

本次程序设计的项目名为 ExParser，采用的 C++ 标准为 C++20 (GNU++20)，使用 CMake 进行构建，在 ARM aarch 64 架构的 macOS Ventura 13.0 上编写、调试与测试。

若需在自己的环境中运行，而提供的可执行文件无法使用，可利用 CMake 与给出的 CMakeLists.txt 自行构建。

ExParser 是一个命令行工具。构建完成后，在命令行中键入 ./ExParser 即可运行，使用方式和运行截图见下文。

代码总行数为 927 sloc，全部代码随附在作业中提交。若发现有任何问题请随时联系我。

本地采用的编译工具链如下表：

工具	版本
CMake	3.26.4
g++-13	(Homebrew GCC 13.2.0) 13.2.0
lldb	16.0
ld	ld64-820.1

表 1：工具链信息

2.3. 程序设计介绍与功能细节

ExParser 是规范 C++ 项目。源码除 `main` 外还包含 `SimpleLexer`、`Parser`、`Util` 三个子模块，分别是简易词法分析器、语法分析器与常用工具代码库，它们全部位于 namespace `ep` 内。

`SimpleLexer` 实现了对于数字和符号的简单词法分析；`Parser` 模块实现了文法及其各种操作和基于预测分析表的预测分析程序；`Util` 模块用于辅助实现功能。以下对 `SimpleLexer` 和 `Parser` 模块进行介绍。

2.3.1. 词法 Token 与语法 Symbol

ExParser 提供的简易词法分析器模块 `SimpleLexer` 基于第一次程序设计的词法分析器修改而来，主旨是通过 `Lexer::next_token` 接口为语法分析器提供服务。

它能够识别的词法 token 种类及对应的语法 symbol 如下表：

名称	Token 种类	Symbol 种类	备注
整数	Integer	终结符 "num"	识别所有连续的十进制数位 (相当于正则表达式 <code>\d+</code>)
符号	Punctuator(punct)	终结符 punct	接受字符集 (,), +, -, *, /
空白字符	Whitespace	N/A (忽略)	
词法错误	Lex Error	N/A (不进入语法分析阶段)	

表 2：Token 种类与 Symbol 种类对应

2.3.2. 文法相关类型定义

如何选择合适的数据结构表达文法是本次程序设计的关键。为了为后续程序设计提供便利，`parser/grammar.h` 提供了一系列类型的定义或别名（省略成员函数）：

```

struct Symbol {
    std::string v{};
    enum Type { Terminator, NonTerminator } type;
};

using ProductionSet = std::pair<Symbol, std::set<std::vector<Symbol>>>;
using FirstSet = std::map<Symbol, std::set<Symbol>>;
using FollowSet = std::map<Symbol, std::set<Symbol>>;
using PredictionTable
    = std::map<Symbol, std::map<Symbol, std::vector<Symbol>>>;

struct Grammar {
    std::map<Symbol, std::set<std::vector<Symbol>>> productions{};
};

```

在 C++ 中，`std::map<A, B>` 可以看作数组，其下标是 A 类型的任意值，数组中的元素是 B 类型的值（其底层实际上可以看作 `std::set<std::pair<A, B>>`）。以下我们依次介绍这些类型：

在您提供的代码中，定义了几种数据结构来表示文法，这些是编写语法分析器时的关键组成部分。我们可以依次介绍这些类型：

1. `struct Symbol` 用于表示文法中的符号（终结符和非终结符）。
 - `std::string v` 用于存储符号的实际值。
 - `enum Type` 指示符号是终结符（`Terminator`）或非终结符（`NonTerminator`）。
2. `using ProductionSet = std::pair<Symbol, std::set<std::vector<Symbol>>>` 用于表示一组生成式。
 - 用 `std::vector<Symbol>` 来表示一个生成式的右侧，即一组有序的符号。
 - 用 `std::set<std::vector<Symbol>>` 来表示一组左侧相同的生成式。
 - 因此 `ProductionSet` 可用于表示一组生成式，其中 `first` 部分是生成式的左侧（非终结符），`second` 部分是一组生成式的右侧。
3. `using FirstSet = std::map<Symbol, std::set<Symbol>>` 与 `using FollowSet = std::map<Symbol, std::set<Symbol>>`
 - 根据定义，它们可以用于表示 FIRST 集合和 FOLLOW 集合。实际上 FIRST 和 FOLLOW 集合还可以扩充到 `std::map<std::vector<Symbol>, std::set<Symbol>>`，但是本次程序设计并未使用到。

4. `using PredictionTable = std::map<Symbol, std::map<Symbol, std::vector<Symbol>>>`
表示预测分析表。

- 它实际上是一个二维数组，数组的两维下标都是 `Symbol` 类型，数组的元素是 `std::vector<Symbol>` 类型，表示一个生成式的右侧。这个类型的定义是本次程序设计的核心，它是预测分析表的数据结构，也是预测分析程序的核心数据结构。

5. `struct Grammar` 用于表示文法。

- 成员 `std::map<Symbol, std::set<std::vector<Symbol>>> productions` 用于存储所有生成式。
- `Grammar` 类型包含一系列的成员函数，用于对文法进行操作，如消除左递归、提取左公因子、构造 FIRST 集、FOLLOW 集、判断是否是 LL(1) 文法、构造预测分析表等，是本次程序设计的核心。

这些结构体和类型别名是构建语法分析器的基础，帮助管理和使用文法规则。

2.3.3. 从字符串中获取文法

`Grammar :: from_str` 函数用于从字符串中获取文法。用于表示输入文法的文法 G 定义为 $G = (N, T, P, \text{Start})$ ，其中 $N = \{\text{Start}, \text{Line}, \text{ExtraLine}, \text{Lhs}, \text{Rhs}, \text{ExtraRhs}, \text{Str}, \text{ExtraStr}\}$ ， $T = \{[\text{LineFeed}], [\text{Space}], \text{Nonterminator}, \text{Terminator}, []\}$ ， P 定义为

$\text{Start} \rightarrow \text{Line ExtraLine}$

$\text{ExtraLine} \rightarrow [\text{LineFeed}] \text{ Line ExtraLine} \mid \epsilon$

$\text{Line} \rightarrow \text{Lhs} [\text{Space}] \rightarrow [\text{Space}] \text{ Rhs}$

$\text{Lhs} \rightarrow \text{NonTerminator}$

$\text{Rhs} \rightarrow \text{Str ExtraRhs}$

$\text{ExtraRhs} \rightarrow [\text{Space}] " | " [\text{Space}] \text{ Rhs} \mid \epsilon$

$\text{Str} \rightarrow \text{NonTerminator ExtraStr} \mid \text{Terminator ExtraStr}$

$\text{ExtraStr} \rightarrow [\text{Space}] \text{ NonTerminator ExtraStr} \mid [\text{Space}] \text{ Terminator ExtraStr} \mid \epsilon$

其中 $[\text{LineFeed}]$ 表示换行符， $[\text{Space}]$ 表示空格， NonTerminator 表示非终结符， Terminator 表示终结符。用语言描述即是，文法每行一组产生式，不能包含多余空行。每个产生式的左侧是一个非终结符，右侧是一组符号，每组符号内部的符号之间用空格分隔，各组符号之间用“ $|$ ”分隔。为了避免歧义，不能增加或减少任何空格。 NonTerminator 和 Terminator 可以包含任意字符，但不能重复；用“ \sim ”表示空串，规定空串属于 Terminator 。

例如，题给文法在这个文法下表示为：

```
E → E + T | E - T | T
T → T * F | T / F | F
F → ( E ) | num
```

2.3.4. 文法相关算法

2.3.4.1. 消除左递归

左递归是指在文法中，一个非终结符的生成式直接或间接地引用了它自身作为最左边的符号，例如 $A \rightarrow A\alpha | \beta$ 。这种形式的文法会导致递归下降解析器陷入无限递归，因此在进行语法分析之前需要消除左递归，消除左递归是构造预测分析表的第一步。

在 Grammar 类中，我们实现了 `Grammar::eliminate_left_recursion` 函数，用于消除文法中的直接左递归。具体步骤：

1. 遍历原始文法的所有生成式。每个生成式由左侧的非终结符 `lhs` 和右侧的生成式集合 `rhs_set` 组成。
2. 检查一个生成式是否是左递归的。如果不是左递归的，直接将该生成式加入新文法。如果是则处理左递归。
 3. 处理左递归生成式：
 - 创建一个新的非终结符 `new_lhs`，采用在原有非终结符的基础上添加一个特殊标记“'”。
 - 为新旧非终结符创建两个生成式集合 `new_rhs_set` 和 `old_rhs_set`。
 - 遍历原始的右侧生成式集合 `rhs_set`。如果生成式的最左边是原左侧非终结符（即形成左递归），则移除这个符号，并在生成式末尾添加新的非终结符 `new_lhs`，加入到 `new_rhs_set`；如果不是左递归，则在生成式末尾添加 `new_lhs`，加入到 `old_rhs_set`。
 - 将处理过的两类生成式集合添加到新文法中。对于有左递归的非终结符，还需要添加一个额外的生成式 $\text{new_lhs} \rightarrow \epsilon$ ，以确保语法的完整性。
4. 将原文法替换为新构建的文法。

这个过程实现了将左递归文法转换为等价的非左递归文法，同时保持语言的生成能力不变。

代码如下：

```
void Grammar::eliminate_left_recursion() {
    Grammar new_grammar{};

    for (const auto &[lhs, rhs_set] : productions) {
        if (!ep::is_left_recursive({lhs, rhs_set})) {
```

```

    new_grammar.push_productions(lhs, rhs_set);
    continue;
}

Symbol new_lhs{lhs.v + "", Symbol::NonTerminator};
std::set<std::vector<Symbol>> new_rhs_set{};
std::set<std::vector<Symbol>> old_rhs_set{};

for (auto &&rhs : rhs_set) {
    if (rhs.front() == lhs) {
        auto new_rhs = rhs;
        new_rhs.erase(new_rhs.begin());
        new_rhs.emplace_back(new_lhs);
        new_rhs_set.emplace(std::move(new_rhs));
    } else {
        auto old_rhs = rhs;
        old_rhs.emplace_back(new_lhs);
        old_rhs_set.emplace(std::move(old_rhs));
    }
}

new_grammar.push_productions(lhs, std::move(old_rhs_set));
new_grammar.push_productions(new_lhs, std::move(new_rhs_set));
new_grammar.push_production(new_lhs, {Symbol::empty_symbol()});
}

*this = std::move(new_grammar);
}

```

2.3.4.2. 提取左公因子

提取左公因子能够处理文法生成式中的公共前缀问题，消除从同一个非终结符开始的多个生成式间的公共前缀，能够消除解析器在选择正确生成式时的出现的歧义。

函数 `Grammar::extract_left_factoring` 用于提取左公因子。具体步骤：

1. 通过一个不断重复的循环进行检查和处理，直到一个完整的迭代中没有进行任何改变（意味着提取左公因子过程结束）。
2. 遍历文法的每个生成式，每个生成式由一个左侧的非终结符（`lhs`）和一个右侧生成式集合（`rhs_set`）组成。
3. 对于每组生成式创建一个新的生成式集合映射 `new_rhs_set_map`，以存储改写后的生成式。对于 `rhs_set` 中的每个生成式，去掉其最左边的符号，并将剩余部分作为新生成式的右侧，键为原始生成式的最左符号。

4. 提取左公因子：遍历 `new_rhs_set_map`。对于原始生成式的每个最左符号，检查它映射到的新生成式集合。如果一个最左符号只映射到一个生成式，直接将其加入新文法；如果一个最左符号映射到多个生成式，说明存在左因子需要被提取。为这些生成式创建一个新的非终结符 `new_lhs`，并更新文法，以此非终结符开始的新生成式替代原有的左因子。
5. 在每次循环的末尾，用新文法替换当前文法，如果在这个循环中进行了修改，则继续循环；否则说明提取左公因子过程结束。

代码如下：

```

void Grammar::extract_left_factoring() {
    for (bool changed;;) {
        changed = false;

        Grammar new_grammar{};
        for (const auto &[lhs, rhs_set] : productions) {
            std::map<Symbol, std::set<std::vector<Symbol>>> new_rhs_set_map{};

            for (const auto &rhs : rhs_set) {
                if (rhs.empty())
                    continue;

                auto new_rhs = rhs;
                new_rhs.erase(new_rhs.begin());
                new_rhs_set_map[rhs.front()].emplace(std::move(new_rhs));
            }

            int counter = 0;
            for (const auto &[symbol, new_rhs_set] : new_rhs_set_map) {
                if (new_rhs_set.size() > 1) {
                    ++counter;
                    Symbol new_lhs{
                        lhs.v + std::to_string(counter), Symbol::NonTerminator
                    };
                    new_grammar.push_productions(lhs, {symbol, new_lhs});
                    new_grammar.push_productions(new_lhs, new_rhs_set);
                } else {
                    std::vector<Symbol> new_rhs{symbol};
                    new_rhs.insert(
                        new_rhs.end(), new_rhs_set.begin()→begin(),
                        new_rhs_set.begin()→end()
                    );
                    new_grammar.push_production(lhs, std::move(new_rhs));
                }
            }
            changed |= counter > 0;
        }
    }
}

```

```
    }

    *this = std::move(new_grammar);

    if (!changed)
        break;
}

}
```

2.3.4.3. 构造 FIRST 集

对于文法 $G = (N, T, P, S)$, $\alpha \in (T \cup N)^*$, FIRST(α) 是指 α 能够推导的所有字符串的首符号的集合。形式化地,

$$\text{FIRST}(\alpha) = \left\{ a \in T \mid \alpha \xrightarrow{*} a\beta, a \in T, \beta \in (T \cup N)^* \right\} \cup \left\{ \varepsilon \mid \alpha \xrightarrow{*} \varepsilon \right\}$$

其中 ϵ 表示空串。FIRST 集合是文法分析的重要组成部分，它能够帮助我们判断一个符号串是否能够推导出空串，以及在预测分析表中选择正确的生成式。

函数 `Grammar::construct_first_set` 用于构造 FIRST 集。具体步骤：

- 对于每个终结符，将其加入到自身的 FIRST 集中。
 - 遍历文法的每个生成式，每个生成式由一个左侧的非终结符（lhs）和一个右侧生成式集合（rhs_set）组成。
 - 对于每个生成式 $A \rightarrow \alpha$ ，如果 $\alpha = \varepsilon$ ，则将 ε 加入 $\text{FIRST}(A)$ ；否则，如果 $\alpha = a\beta$ 且 $a \in T$ ，则将 a 加入 $\text{FIRST}(A)$ ；否则，如果 $\alpha = B\beta$ 且 $B \in N$ ，则将 $\text{FIRST}(B) - \{\varepsilon\}$ 加入 $\text{FIRST}(A)$ 。
对于 $A \rightarrow B_1B_2\dots B_n$ ($B_i \in N, 1 < i < n$)，若 $\varepsilon \in \bigcap_{i=1}^{k-1} \text{FIRST}(B_i)$ ，则将 $\text{FIRST}(B_k) - \{\varepsilon\}$ 加入 $\text{FIRST}(A)$ ；若 $\varepsilon \in \bigcap_{i=1}^{n-1} \text{FIRST}(B_i)$ ，则将 ε 加入 $\text{FIRST}(A)$ 。

代码如下：

```
FirstSet Grammar::build_first_set() const {
    FirstSet first_set{};

    auto terminators = get_terminators();

    for (const auto &terminator : terminators.first)
        first_set[terminator].emplace(terminator);
    if (terminators.second)
        first_set[Symbol::empty_symbol()].emplace(Symbol::empty_symbol());

    for (bool changed;;) {
        changed = false;
```

```

for (const auto &[lhs, rhs_set] : productions) {
    for (const auto &rhs : rhs_set) {
        if (rhs.empty()) {
            changed |= first_set[lhs].emplace(Symbol::empty_symbol()).second;
            continue;
        }

        auto old_size = first_set[lhs].size();
        for (const auto &symbol : rhs) {
            auto &first_set_rhs = first_set[symbol];
            first_set[lhs].insert(first_set_rhs.begin(), first_set_rhs.end());
            if (!first_set_rhs.contains(Symbol::empty_symbol()))
                break;
        }
        changed |= first_set[lhs].size() != old_size;
    }
}

if (!changed)
    break;
}

return first_set;
}

```

2.3.4.4. 构造 FOLLOW 集

对于文法 $G = (N, T, P, S)$, 对于 $A \in N$, $\text{FOLLOW}(A)$ 表示该文法的所有句型中可能在 A 之后出现的终结符号 $a \in T$ 或结束符号 \$ 的集合。形式化地,

$$\text{FOLLOW}(A) = \left\{ a \in T \mid S \xrightarrow{*} \alpha A a \beta, a \in T, \alpha, \beta \in (T \cup N)^* \right\}$$

函数 `Grammar::build_follow_set` 用于构造 FOLLOW 集。具体步骤:

1. 将 \$ 加入 $\text{FOLLOW}(S)$ 。
2. 通过一个不断重复的循环进行检查和处理, 直到一个完整的迭代中没有进行任何改变, 即 FOLLOW 集不再发生变化。每趟循环遍历文法的每个生成式。对于每个生成式, 遍历其右侧的所有非终结符作为下述的 B 。对于生成式 $A \rightarrow \alpha B \beta$, 将 $\text{FIRST}(\beta) - \{\epsilon\}$ 加入 $\text{FOLLOW}(B)$; 如果 $\epsilon \in \text{FIRST}(\beta)$, 则将 $\text{FOLLOW}(A)$ 加入 $\text{FOLLOW}(B)$ 。对于生成式 $A \rightarrow \alpha B$, 也将 $\text{FOLLOW}(A)$ 加入 $\text{FOLLOW}(B)$ 。

代码如下:

```

FollowSet Grammar::build_follow_set(
    FirstSet &first_set, const Symbol &start_symbol
) const {
    FollowSet follow_set{};

    auto terminators = get_terminators();

    follow_set[start_symbol].emplace("$", Symbol::Terminator);

    for (bool changed;;) {
        changed = false;

        for (const auto &[lhs, rhs_set] : productions) {
            for (const auto &rhs : rhs_set) {
                for (auto it = rhs.begin(); it != rhs.end(); ++it) {
                    if (it->type == Symbol::Terminator)
                        continue;

                    auto &follow_set_lhs = follow_set[*it];
                    auto old_size = follow_set_lhs.size();

                    auto it2 = std::next(it);
                    if (it2 != rhs.end()) {
                        auto &first_set_rhs = first_set[*it2];
                        for (const auto &symbol : first_set_rhs)
                            if (symbol != Symbol::empty_symbol())
                                follow_set_lhs.emplace(symbol);
                    }
                    if (it2 == rhs.end() ||
                        first_set[*it2].contains(Symbol::empty_symbol()))
                        follow_set_lhs.insert(
                            follow_set[lhs].begin(), follow_set[lhs].end()
                        );
                }

                changed |= follow_set_lhs.size() != old_size;
            }
        }

        if (!changed)
            break;
    }

    return follow_set;
}

```

2.3.4.5. 判断是否为 LL(1) 文法

对于文法 $G = (N, T, P, S)$, 若其满足对于任意两个生成式 $A \rightarrow \alpha$ 和 $A \rightarrow \beta$ 都有 $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$, 且若 $\alpha \xrightarrow{*} \varepsilon$, 则 $\text{FIRST}(\beta) \cap \text{FOLLOW}(A) = \emptyset$, 则称 G 是 LL(1) 文法。

函数 `Grammar::is_ll1_grammar` 用于判断文法是否是 LL(1) 文法。具体步骤：

1. 构造 FIRST 集和 FOLLOW 集。
2. 遍历文法的每组生成式 $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$ 。对于每对生成式 $A \rightarrow \alpha_i$ 和 $A \rightarrow \alpha_j$, 将其按照前述的 $A \rightarrow \alpha$ 和 $A \rightarrow \beta$ 进行测试是否满足。若不满足则直接返回错误。

代码如下。这个函数在文法不是 LL(1) 文法时返回错误信息，否则返回 `std::nullopt`。

```
std::optional<std::string>
Grammar::is_ll1(FirstSet &first_set, FollowSet &follow_set) const {
    for (const auto &[lhs, rhs_set] : productions) {
        for (const auto &rhs : rhs_set) {
            if (rhs.empty())
                continue;

            auto &first_set_rhs = first_set[rhs.front()];
            if (first_set_rhs.contains(Symbol::empty_symbol())) {
                auto &follow_set_lhs = follow_set[lhs];
                for (const auto &symbol : follow_set_lhs)
                    if (first_set_rhs.contains(symbol))
                        return /* error_info */;
            }
        }

        for (auto it1 = rhs_set.begin(); it1 != rhs_set.end(); ++it1) {
            for (auto it2 = std::next(it1); it2 != rhs_set.end(); ++it2) {
                auto &first_set_rhs1 = first_set[*it1->begin()];
                auto &first_set_rhs2 = first_set[*it2->begin()];

                std::set<Symbol> intersection{};
                std::set_intersection(
                    first_set_rhs1.begin(), first_set_rhs1.end(),
                    first_set_rhs2.begin(), first_set_rhs2.end(),
                    std::inserter(intersection, intersection.begin()))
            );
            if (!intersection.empty())
                return /* error_info */;
        }
    }
    return std::nullopt;
}
```

2.3.4.6. 构造预测分析表

LL(1) 预测分析表实际上是一个二维数组，第一维下标是非终结符号 $A \in N$ ，第二维下标是终结符号 $a \in T$ （包含结束符号 \$）。

LL(1) 预测分析表的构造方式如同教材中的伪代码。函数 `Grammar::build_prediction_table` 用于构造预测分析表。具体步骤：

1. 构造 FIRST 集和 FOLLOW 集。

2. 遍历文法的每个生成式 $A \rightarrow \alpha$ 。

- 对于所有 $a \in \text{FIRST}(\alpha) \cap T$ ，将 $A \rightarrow \alpha$ 加入 $M[A, a]$ 。
- 如果 $\varepsilon \in \text{FIRST}(\alpha)$ ，则对于所有 $b \in \text{FOLLOW}(A)$ ，将 $A \rightarrow \alpha$ 加入 $M[A, b]$ 。

代码如下：

```
PredictionTable Grammar::build_prediction_table(
    FirstSet &first_set, FollowSet &follow_set
) const {
    PredictionTable prediction_table{};

    for (const auto &[lhs, rhs_set] : productions) {
        for (const auto &rhs : rhs_set) {
            auto &first_set_rhs = first_set[rhs.front()];
            for (const auto &symbol : first_set_rhs)
                prediction_table[lhs][symbol] = rhs;

            if (first_set_rhs.contains(Symbol::empty_symbol()))
                for (const auto &symbol : follow_set[lhs])
                    prediction_table[lhs][symbol] = rhs;

            if (rhs.empty()) {
                for (const auto &symbol : follow_set[lhs])
                    prediction_table[lhs][symbol] = rhs;
                continue;
            }
        }
    }

    for (auto &[lhs, rhs_map] : prediction_table)
        for (auto it = rhs_map.begin(); it != rhs_map.end();)
            if (it->first == Symbol::empty_symbol())
                it = rhs_map.erase(it);
            else
                ++it;
}
```

```

    return prediction_table;
}

```

2.3.5. 语法分析与错误处理

语法分析器的主要功能是调用 `Lexer` 提供的接口获取 `token` 流并转换为 `symbol` 流，并根据预测分析表进行语法分析，判断输入的符号串是否符合文法规则。在 `Parser` 模块中，我们实现了 `Parser` 类，用于进行语法分析。

函数 `Parser::parse_expression(std::vector<Symbol> &symbol_stream)` 完成了根据预测分析表进行语法分析的功能，采取的算法与教材中伪代码一致：

1. 初始化栈 S ，将栈底符号 $\$$ 和初始符号压入栈中。
2. 当栈非空时，重复以下步骤：
 - 令 X 为栈顶符号， a 为输入符号流的第一个符号。
 - 如果 $X = a = \$$ ，则分析成功，返回。
 - 如果 $X = a \in T$ ，则弹出 X ，输入符号流前进。
 - 如果 $X \in T$ 但 $X \neq a$ ，遇到语法错误，此处采取的错误处理方式是显示错误并弹出栈顶符号。
 - 如果 $X \in N$ ，则访问预测分析表 $M[X, a]$
 - 如果 $M[X, a] = X \rightarrow \alpha$ ，则弹出 X ，将 α 逆序压入栈中。
 - 如果 $M[X, a] = \epsilon$ ，则弹出 X 。
 - 如果 $M[X, a]$ 被标记为错误（此处实现采取置空），则遇到语法错误，跳过剩余输入符号串中的若干个符号，直到可以继续进行分析为止。

代码如下（不包含输出部分）：

```

void Parser::parse_expression(std::vector<Symbol> &symbol_stream) {
    symbol_stream.emplace_back("$", Symbol::Terminator);

    std::vector<Symbol> stack;
    stack.emplace_back("$", Symbol::Terminator);
    stack.emplace_back("E", Symbol::NonTerminator);

    auto it = symbol_stream.begin();
    while (!stack.empty()) {
        auto top = stack.back();
        stack.pop_back();

```

```

if (top.type == Symbol::Terminator) {
    if (top == Symbol::empty_symbol())
        ; // 此处只输出，无其他操作，故意留空
    else if (top.v == it->v)
        ++it;
    else
        // 栈顶符号与输入符号不匹配：记录错误并弹出栈顶符号
} else {
    auto row = prediction_table_.at(top);
    for (; it != symbol_stream.end() && !row.count(*it); ++it)
        // 分析表中没有对应的生成式：记录错误并跳过输入符号流中的符号，直至可以继续分析
    if (it == symbol_stream.end())
        // 输入符号流已经结束但栈未空：记录错误并直接退出
        auto prediction = row.at(*it);
        stack.insert(stack.end(), prediction.rbegin(), prediction.rend());
    }
}
}
}

```

3. 测试、运行结果与分析说明

3.1. 对于给定文法的变换与分析

如前文给出，题给文法表示为

```

E → E + T | E - T | T
T → T * F | T / F | F
F → ( E ) | num

```

这是一个含有左递归和左公因子的文法。我们的程序需要对其进行变换，消除左递归和左公因子，得到等价的文法，并验证其为 LL(1) 文法，之后才能构造 LL(1) 预测分析表，并利用其进行语法分析。

程序在每一个算法运行后都会输出关于文法的信息。运行 [./ExParser](#)，得到下图输出。（注意程序中所有空串都使用“~”表示，因为测试发现非 ASCII 字符 ϵ 在计算宽度时会出现问题导致无法正常输出）

```

$ ./ExParser
-- Input grammar --
Terminators: { (, ), *, +, -, /, n }
NonTerminators: { E, F, T }
Productions:
E → E + T | E - T | T
F → ( E ) | n
T → F T' | T * F | T / F
}

-- Grammar eliminated left recursion --
Terminators: { (, ), *, +, -, /, n }
NonTerminators: { E, E', F, T, T' }
Productions:
E → T E'
E' → ~ | + T E' | - T E'
F → ( E ) | n
T → F T'
T' → ~ | * F T' | / F T'
}

-- Grammar extracted left factoring --
Terminators: { (, ), *, +, -, /, n }
NonTerminators: { E, E', F, T, T' }
Productions:
E → T E'
E' → ~ | + T E' | - T E'
F → ( E ) | n
T → F T'
T' → ~ | * F T' | / F T'
}

-- FIRST SET --
FIRST(~) = {~}
FIRST(()) = {}
FIRST(*) = {*}
FIRST(+) = {+}
FIRST(-) = {-}
FIRST(/) = {/}
FIRST(E) = {{, n}}
FIRST(E') = {~, +, -}
FIRST(F) = {{, n}}
FIRST(T) = {{, n}}
FIRST(T') = {~, *, /}
FIRST(n) = {n}

-- FOLLOW SET --
FOLLOW(E) = {$, )}
FOLLOW(E') = {$, )}
FOLLOW(F) = {$, ), *, +, -, /}
FOLLOW(T) = {$, ), +, -}
FOLLOW(T') = {$, ), +, -}

-- Grammar is validated to be LL(1) --

-- Prediction table --
|   |   $   |   (   |   )   |   *   |   +   |   ~   |   /   |   n   | |
|---|---|---|---|---|---|---|---|---|---|
| E |   (nul) |   E → T E' |   (nul) |   (nul) |   (nul) |   (nul) |   (nul) |   E → T E' |
| E' |   E' → ~ |   (nul) |   E' → ~ |   (nul) |   E' → + T E' |   E' → - T E' |   (nul) |   (nul) |
| F |   (nul) |   F → ( E ) |   (nul) |   (nul) |   (nul) |   (nul) |   (nul) |   F → n |
| T |   (nul) |   T → F T' |   (nul) |   (nul) |   (nul) |   (nul) |   (nul) |   T → F T' |
| T' |   T' → ~ |   (nul) |   T' → ~ |   T' → * F T' |   T' → ~ |   T' → ~ |   T' → / F T' |   (nul) |
|---|---|---|---|---|---|---|---|---|---|

```

Enter a line of expression, or 'q' to quit.

图 1：文法的变换与分析结果

预测分析表中的空表项（或错误表项）用 (nul) 表示。

容易验证，对于输入文法、消左递归后的文法、提取左公因子后的文法、FIRST 集、FOLLOW 集、预测分析表的输出结果都是正确的。测试的重点是以下的语法分析功能。

3.2. 对于任给的算术表达式的语法分析

预先准备了以下的测试样例，其中包含了正确、错误的从简单到复杂的算术表达式¹。

1. [接受] 1 + 1
2. [接受] (114 + 514) * 1919 / 810
3. [出错] 1 1
4. [出错] 1025 - (365) +
5. [出错] / 3722 * 525

¹不测试含有词法分析错误的输入，这不是本次程序设计的重点。如果出现未知的 token，程序只会输出一行 Lex Error。

6. [出错] $1 ++ 2 ** 3 // 4 -- 5$
7. [出错] $(9 +) 8$
8. [出错] $(309 * 23$
9. [出错] $408 / 11)$
10. [出错] $(- 0 * / 021 / 8 / 3 +) - 4 +) (4 * 58 ++ 9 (*) - 811 - 9 () /) 07$
 $* 23 (/ 38 / 35 + 23 * / 25 651 (+ 30 -) 72 / 0 /) + 5 -) 97 (1) / * -$
 $3 - * + 1 + * 5 (5 / 152 + - 1 + - (+ ()) / 2 8 * 10 * /)) * 8 * 3 + 70$
 $* 8 575 *) 0 + 22 - (/ 6 (** + 16 + 6$
11. [接受] $((31 * ((41 + 5) * (926 - 5)) - 3 + (4 / (7 - 1)) - 2) + (4 + ((9 - 2)$
 $* 3) / 6 - (1 + 8) / (7 - 1)) / ((3 * (4 + 2)) + (15 + (234 * (2 + (9 - 3))))$
 $- 2 * (8 - (6 / 3)) + (7 + 1) - (5 / (234 + 1)) + 9 / ((2 * 3) - (8 + 1)) + 4$
 $- (6 * 2))) / 0$

以下是每个例子的运行结果或部分运行结果。

1. [接受] $1 + 1$

1+1			
#	<Stack>	<Input>	<Action>
1	\$E	n+n\$	Initial
2	\$E'T	n+n\$	E → T E'
3	\$E'T'F	n+n\$	T → F T'
4	\$E'T'n	n+n\$	F → n
5	\$E'T'	+n\$	
6	\$E'~	+n\$	T' → ~
7	\$E'	+n\$	
8	\$E'T+	+n\$	E' → + T E'
9	\$E'T	n\$	
10	\$E'T'F	n\$	T → F T'
11	\$E'T'n	n\$	F → n
12	\$E'T'	\$	
13	\$E'~	\$	T' → ~
14	\$E'	\$	
15	\$~	\$	E' → ~
16	\$	\$	Accept

图 2: 例 1 - 接受

2. [接受] $(114 + 514) * 1919 / 810$

(114 + 514) * 1919 / 810	-- Parsing procedure --	
# <Stack>	<Input>	<Action>
1 \$E	(n+n)*n/n\$	Initial
2 \$E'T	(n+n)*n/n\$	E → T E'
3 \$E'T'F	(n+n)*n/n\$	T → F T'
4 \$E'T')E((n+n)*n/n\$	F → (E)
5 \$E'T')E	n+n)*n/n\$	
6 \$E'T')E'T	n+n)*n/n\$	E → T E'
7 \$E'T')E'T'F	n+n)*n/n\$	T → F T'
8 \$E'T')E'T'n	n+n)*n/n\$	F → n
9 \$E'T')E'T'	+n)*n/n\$	
10 \$E'T')E'~	+n)*n/n\$	T' → ~
11 \$E'T')E'	+n)*n/n\$	
12 \$E'T')E'T+	+n)*n/n\$	E' → + T E'
13 \$E'T')E'T	n)*n/n\$	
14 \$E'T')E'T'F	n)*n/n\$	T → F T'
15 \$E'T')E'T'n	n)*n/n\$	F → n
16 \$E'T')E'T')*n/n\$	
17 \$E'T')E'~)*n/n\$	T' → ~
18 \$E'T')E')*n/n\$	
19 \$E'T')~)*n/n\$	E' → ~
20 \$E'T'))*n/n\$	
21 \$E'T'	*n/n\$	
22 \$E'T'F*	*n/n\$	T' → * F T'
23 \$E'T'F	n/n\$	
24 \$E'T'n	n/n\$	F → n
25 \$E'T'	/n\$	
26 \$E'T'F/	/n\$	T' → / F T'
27 \$E'T'F	n\$	
28 \$E'T'n	n\$	F → n
29 \$E'T'	\$	
30 \$E'~	\$	T' → ~
31 \$E'	\$	
32 \$~	\$	E' → ~
33 \$	\$	Accept

图 3：例 2 - 接受

以下是每个例子的运行结果或部分运行结果。

3. [出错] 1 1

1 1	-- Parsing procedure --	
# <Stack>	<Input>	<Action>
1 \$E	nn\$	Initial
2 \$E'T	nn\$	E → T E'
3 \$E'T'F	nn\$	T → F T'
4 \$E'T'n	nn\$	F → n
5 \$E'T'	n\$	
6		Error: T' not match n
7 \$E'~	\$	T' → ~
8 \$E'	\$	
9 \$~	\$	E' → ~
10 \$	\$	
11		

图 4：例 3 - 出错

4. [出错] 1025 - (365) +

```

1025 - (365) +
-- Parsing procedure --
# | <Stack> | <Input> | <Action>
1 | $E | n-(n)+$ | Initial
2 | $E'T | n-(n)+$ | E → T E'
3 | $E'T'F | n-(n)+$ | T → F T'
4 | $E'T'n | n-(n)+$ | F → n
5 | $E'T' | -(n)+$ |
6 | $E'~ | -(n)+$ | T' → ~
7 | $E' | -(n)+$ |
8 | $E'T- | -(n)+$ | E' → - T E'
9 | $E'T | (n)+$ |
10 | $E'T'F | (n)+$ | T → F T'
11 | $E'T')E( | (n)+$ | F → ( E )
12 | $E'T')E | n)+$ |
13 | $E'T')E'T | n)+$ | E → T E'
14 | $E'T')E'T'F | n)+$ | T → F T'
15 | $E'T')E'T'n | n)+$ | F → n
16 | $E'T')E'T' | )+$ |
17 | $E'T')E'~ | )+$ | T' → ~
18 | $E'T')E' | )+$ |
19 | $E'T')~ | )+$ | E' → ~
20 | $E'T') | )+$ |
21 | $E'T' | +$ |
22 | $E'~ | +$ | T' → ~
23 | $E' | +$ |
24 | $E'T+ | +$ | E' → + T E'
25 | $E'T | $ |
26 | | | Error: T not match $
27 | | | Error: T not match ~

```

图 5：例 4 - 出错

5. [出错] / 3722 * 525

```

/ 3722 * 525
-- Parsing procedure --
# | <Stack> | <Input> | <Action>
1 | $E | /n*n$ | Initial
2 | | | Error: E not match /
3 | $E'T | n*n$ | E → T E'
4 | $E'T'F | n*n$ | T → F T'
5 | $E'T'n | n*n$ | F → n
6 | $E'T' | *n$ |
7 | $E'T'F* | *n$ | T' → * F T'
8 | $E'T'F | n$ |
9 | $E'T'n | n$ | F → n
10 | $E'T' | $ |
11 | $E'~ | $ | T' → ~
12 | $E' | $ |
13 | $~ | $ | E' → ~
14 | $ | $ |
15 | | | 

```

图 6：例 5 - 出错

6. [出错] 1 ++ 2 ** 3 // 4 -- 5

#	<Stack>	<Input>	<Action>
1	\$E	n++n**n//n--n\$	Initial
2	\$E'T	n++n**n//n--n\$	E → T E'
3	\$E'T'F	n++n**n//n--n\$	T → F T'
4	\$E'T'n	n++n**n//n--n\$	F → n
5	\$E'T'	++n**n//n--n\$	
6	\$E'~	++n**n//n--n\$	T' → ~
7	\$E'	++n**n//n--n\$	
8	\$E'T+	++n**n//n--n\$	E' → + T E'
9	\$E'T	+n**n//n--n\$	
10			Error: T not match +
11	\$E'T'F	n**n//n--n\$	T → F T'
12	\$E'T'n	n**n//n--n\$	F → n
13	\$E'T'	**n//n--n\$	
14	\$E'T'F*	**n//n--n\$	T' → * F T'
15	\$E'T'F	*n//n--n\$	
16			Error: F not match *
17	\$E'T'n	n//n--n\$	F → n
18	\$E'T'	//n--n\$	
19	\$E'T'F/	//n--n\$	T' → / F T'
20	\$E'T'F	/n--n\$	
21			Error: F not match /
22	\$E'T'n	n--n\$	F → n
23	\$E'T'	--n\$	
24	\$E'~	--n\$	T' → ~
25	\$E'	--n\$	
26	\$E'T-	--n\$	E' → - T E'
27	\$E'T	-n\$	
28			Error: T not match -
29	\$E'T'F	n\$	T → F T'
30	\$E'T'n	n\$	F → n
31	\$E'T'	\$	
32	\$E'~	\$	T' → ~
33	\$E'	\$	
34	\$~	\$	E' → ~
35	\$	\$	
36			

图 7: 例 6 - 出错

7. [出错] (9 +) 8

#	<Stack>	<Input>	<Action>
1	\$E	(n+)n\$	Initial
2	\$E'T	(n+)n\$	E → T E'
3	\$E'T'F	(n+)n\$	T → F T'
4	\$E'T')E((n+)n\$	F → (E)
5	\$E'T')E	n+)\$	
6	\$E'T')E'T	n+)\$	E → T E'
7	\$E'T')E'T'F	n+)\$	T → F T'
8	\$E'T')E'T'n	n+)\$	F → n
9	\$E'T')E'T'	+)\$	
10	\$E'T')E'~	+)\$	T' → ~
11	\$E'T')E'	+)\$	
12	\$E'T')E'T+	+)\$	E' → + T E'
13	\$E'T')E'T)n\$	
14			Error: T not match)
15	\$E'T')E'T'F	n\$	T → F T'
16	\$E'T')E'T'n	n\$	F → n
17	\$E'T')E'T'	\$	
18	\$E'T')E'~	\$	T' → ~
19	\$E'T')E'	\$	
20	\$E'T')~	\$	E' → ~
21	\$E'T')	\$	
22			Error:) not match \$
23	\$E'~	\$	T' → ~
24	\$E'	\$	
25	\$~	\$	E' → ~
26	\$	\$	
27			

图 8: 例 7 - 出错

8. [出错] (309 * 23

(309 * 23 -- Parsing procedure --		
# <Stack>	<Input>	<Action>
1 \$E	(n*n\$	Initial
2 \$E'T	(n*n\$	E → T E'
3 \$E'T'F	(n*n\$	T → F T'
4 \$E'T')E((n*n\$	F → (E)
5 \$E'T')E	n*n\$	
6 \$E'T')E'T	n*n\$	E → T E'
7 \$E'T')E'T'F	n*n\$	T → F T'
8 \$E'T')E'T'n	n*n\$	F → n
9 \$E'T')E'T'	*n\$	
10 \$E'T')E'T'F*	*n\$	T' → * F T'
11 \$E'T')E'T'F	n\$	
12 \$E'T')E'T'n	n\$	F → n
13 \$E'T')E'T'	\$	
14 \$E'T')E'~	\$	T' → ~
15 \$E'T')E'	\$	
16 \$E'T')~	\$	E' → ~
17 \$E'T')	\$	
18		Error:) not match \$
19 \$E'~	\$	T' → ~
20 \$E'	\$	
21 \$~	\$	E' → ~
22 \$	\$	
23		

图 9: 例 8 - 出错

9. [出错] 408 / 11)

408 / 11)		
-- Parsing procedure --		
# <Stack>	<Input>	<Action>
1 \$E	n/n)\$	Initial
2 \$E'T	n/n)\$	E → T E'
3 \$E'T'F	n/n)\$	T → F T'
4 \$E'T'n	n/n)\$	F → n
5 \$E'T'	/n)\$	
6 \$E'T'F/	/n)\$	T' → / F T'
7 \$E'T'F	n)\$	
8 \$E'T'n	n)\$	F → n
9 \$E'T')\$	
10 \$E'~)\$	T' → ~
11 \$E')\$	
12 \$~)\$	E' → ~
13 \$)\$	
14		Error: \$ not match)

图 10: 例 9 - 出错

10. [出错] (- 0 * / 021 / 8 / 3 +) - 4 +) (4 * 58 ++ 9 (*) - 811 - 9 () /) 07

* 23 (/ 38 / 35 + 23 * / 25 651 (+ 30 -) 72 / 0 /) + 5 -) 97 (1) / * -
 3 - * + 1 + * 5 (5 / 152 + - 1 + - (+ ()) / 2 8 * 10 * /)) * 8 * 3 + 70
 * 8 575 *) 0 + 22 - (/ 6 (** + 16 + 6

图 11: 例 10 - 出错 (请放大查看)

11. [接受] $((31 * ((41 + 5) * (926 - 5)) - 3 + (4 / (7 - 1)) - 2) + (4 + ((9 - 2) * 3) / 6 - (1 + 8) / (7 - 1)) / ((3 * (4 + 2)) + (15 + (234 * (2 + (9 - 3)))) - 2 * (8 - (6 / 3)) + (7 + 1) - (5 / (234 + 1)) + 9 / ((2 * 3) - (8 + 1)) + 4 - (6 * 2))) / 0^2$

图 12: 例 11 - 接受 (开头)

图 13: 例 11 - 接受 (结尾)

²因篇幅较长（足足 521 行），在此只给出运行结果的开头和结尾。

4. 总结

本次程序设计到此顺利完成。在程序设计过程中，我学到了许多自顶向下的语法分析方式相关的概念的方法，实现了一个名为 ExParser 的语法分析程序 ExParser 能够接受一个文法作为输入，对其进行消除左递归、提取左公因子、构造 FIRST 集、FOLLOW 集、判断是否是 LL(1) 文法、构造预测分析表等操作，并能够根据预测分析表进行语法分析，判断输入的符号串是否符合文法规则。

由于时间有限，本次实验的程序实现也存在一些不足之处。首先，程序的输入和输出都是通过标准输入输出流进行的，没有提供完善的 CLI。其次，由于采用的文法是给定的，因此程序中有些地方直接采取了硬编码的方式，而没有提供运行时的文法输入接口。这些是程序未来的改进方向。

感谢阅读。