《编译原理与设计》

语法分析程序的设计与实现

实验报告

班级 2014211304

学号 2014210336

姓名 杨炫越

日期 2016.11.14

1. 实验内容

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

$E \rightarrow E+T \mid E–T \mid T$
$T \rightarrow T*F \mid T/F \mid F$
$F \rightarrow (E) \mid \text{num}$

2. 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。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。

3. 开发环境

操作系统：Microsoft Windows 10.0.14393 (x64)
IDE：Microsoft Visual Studio Community 2015
编译器：MSVC++ 14.0
附加库：Boost 1.62.0

4. 设计思路

本语法分析程序首先**利用上次实验实现的词法分析程序**将输入串转化为 token 流，再在此基础上采用 LL(1)分析方法对输入算术表达式进行分析，并输出分析过程采用的产生式。流程如下：

LL(1)分析的关键在于构造预测分析表，然后使用分析表与一个分析栈进行联合控制，实现对输入符号串的自顶向上分析。构造预测分析表的前序工作依次如下（各部分算法细节详见 5. 程序实现）：

(1) 消除文法的左递归；
(2) 提取文法的左公因子；
(3) 构造文法的 FIRST 集；
(4) 构造文法的 FOLLOW 集。

此外，本程序的文法可配置，可通过修改.ini 文件来完善 **C 语言**文法规则。

5. 程序实现

源码：

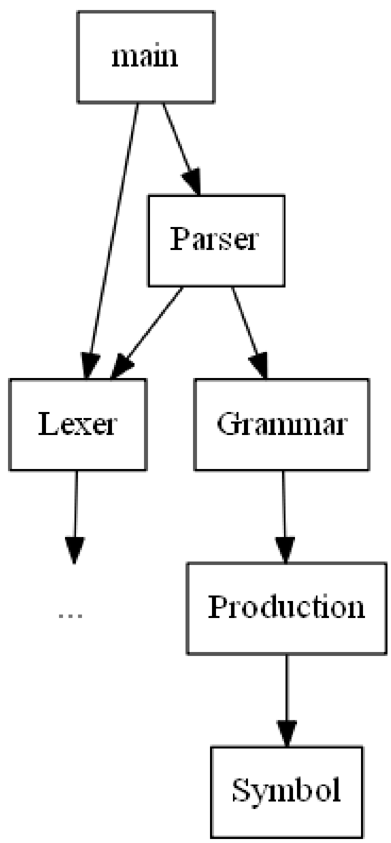
Local：/src_code

Online：https://github.com/YangXuanyue/Compiler

本语法分析程序采用了 C++来编写，程序中实现了一个`Parser`类作为语法分析器的对外接口，并为其重载了输入流操作符，可与上次实验实现的词法分析类`Lexer`连接如下：

```
Lexer lexer('\n'); //以换行符为输入串结尾
Parser parser;
cin >> lexer >> parser;
```

本程序的结构示意图如下所示：



其中 Lexer 分支为上次实验实现的词法分析器，其余各模块实现简述如下：

5.1. Symbol

该模块定义了文法符号的存储设计如下：

```
enum SymbolType {
    NONTERMINAL, //string
    TERMINAL //TokenType
};

typedef variant<string, TokenType> Symbol;
```

文法符号有非终结符（nonterminal）与终结符（terminal）两种，考虑到语法分析是在词法分析得到的 token 流上进行的，其文法的终结符即为各类 token，于是在本程序中采用字符串`std::string`来存储非终结符，采用`TokenType`来存储终结符。为统一，使用 Boost 库的`boost::variant`来存放两者，其可视为一个面向对象版本的联合类型 union，文档详见 [Boost.Variant]。

5.2. Production

该模块定义了产生式结构体大致如下：

```
struct Production {
    Symbol left;
    deque<Symbol> right;
};
```

其中产生式的左部为一个`Symbol`类型的非终结符。考虑到在文法的改造环节如消除左递归、提取左公因子中，需频繁对产生式右部文法符号串的头尾端进行插入删除等操作，为提高效率，采用了对头尾增删操作具有$\Theta(1)$时间复杂度的**双端队列**`std::deque`来存放产生式右部的文法符号串。

5.3. Grammar

该模块定义了文法类大致如下：

```
class Grammar {
private:
    //非终结符集与终结符集
    vector<Symbol> nonterminals, terminals;
    //起始符号
    Symbol start_symbol;
    //产生式集
    vector<Production> productions;
    //每个非终结符为左端的产生式序号集
    map<Symbol, set<int>> production_idxes;
    //每个产生式右端文法符号串的FIRST集
    vector<set<Symbol>> first_of_production;
    //FIRST集与FOLLOW集
    map<Symbol, set<Symbol>> first, follow;
    //对每个非终结符是否已构造完FIRST集与FOLLOW集的标志，方便递归
    map<Symbol, bool> has_constructed_first, has_constructed_follow;
    //判断两个非终结符的FOLLOW集是否互相包含，避免求FOLLOW集过程出现无限递归
    map<Symbol, map<Symbol, bool>> includes_follow_of;

public:
    Grammar();
    //从配置文件加载文法
    void load_from_ini();
    //消除左递归
    void remove_left_recursion();
    //提取左公因子
    void extract_common_left_factor();
    //对非终结符nonterminal构造FIRST集
    void construct_first(const Symbol& nonterminal);
    //对非终结符nonterminal构造FOLLOW集
    void construct_follow(const Symbol& nonterminal);
};
```

该模块实现了文法的加载、改造与 FIRST 集、FOLLOW 集的构造，此为构造 LL(1)预测分析表的前序工作。具体算法及实现简述如下：

5.3.1. 加载配置文件

本实验仅要求识别简单的算术表达式，而 C 语言的文法远不止于此，为便于后期拓展完善，本程序实现了文法的可配置化，配置文件为 src_code/Compiler/Parser/Grammar.ini，通过`load_from_ini()`函数读取该配置文件，并依据其格式完成文法的初始化。配置格式如下：

```
nonterminals = {
    E T F ...
}

terminals = {
    + / if for ...
}

start_symbol = {E}

productions = {
    {E -> E+T | E-T | ...}
    ...
}
```

要求：

(1) 输入的终结符是合法的 C 语言关键字、运算符或诸如 `num`、`id` 之类的 `token` 类型；

(2) 产生式以`{}`括起，同一非终结的不同产生式之间以`|`隔开，由于暂不支持转义，产生式中不可出现`{`、`}`或`|`，有待进一步改进。

本实验的文法配置如下：

```
nonterminals = {
    E T F
}

terminals = {
    + - * / ( ) num
}

start_symbol = {E}

productions = {
    {E -> E+T | E-T | T}
    {T -> T*F | T/F | F}
    {F -> (E) | num}
}
```

5.3.2. 消除左递归

若一个文法中存在非终结符A，对某个文法符号串α，存在推导

$$A \overset{+}{\Rightarrow} A\alpha,$$

则该文法存在左递归。

若对非终结A，有产生式$A \longrightarrow A\alpha \mid \beta$，则$A$是直接左递归的。消除直接左递归可对产生式作如下改造：

$$A \longrightarrow \beta A'$$
$$A' \longrightarrow \alpha A' \mid \varepsilon$$

为消除文法中的所有左递归，需将所有非终结排成一定顺序，依次将$A_i \longrightarrow A_j\beta (i > j)$中的$A_j$分别替换为$A_j$的所有产生式，再对$A_i$消除直接左递归。

在递归下降分析方法中，要求文法不能含有左递归，否则将可能出现死循环。而 LL(1)预测分析消除了递归下降分析的不确定性，也基于文法不能含有左递归的前提条件，否则在构造 FIRST 集的过程就会出现死循环。

本程序中消除左递归的算法实现如下：

```
void Grammar::remove_left_recursion() {
    set<Symbol> vis_nonterminals; //已对应产生式消除完左递归的非终结符集
    vector<Symbol> new_nonterminals; //新终结符集
    for (const auto& nonterminal : nonterminals) { //遍历非终结集
        set<int> new_production_idxes(production_idxes[nonterminal]);
        for (int i : production_idxes[nonterminal]) { //遍历非终结符nonterminal的所有产生式
            Symbol first_symbol(productions[i].right.front()); //产生式第一个符号
            //若firsr_symbol是一个已消除左递归的非终结符
            if (vis_nonterminals.find(first_symbol) != vis_nonterminals.end()) {
                new_production_idxes.erase(i);
                //将该产生式的第一个符号分别替换为该符号的所有产生式
                productions[i].right.pop_front();
                for (int j : production_idxes[first_symbol]) {
                    new_production_idxes.insert(productions.size());
                    Production new_production(productions[i]);
                    new_production.right.insert(
                        new_production.right.begin(),
                        productions[j].right.begin(),
                        productions[j].right.end()
                    );
                    productions.emplace_back(std::move(new_production));
                }
            }
        }
        production_idxes[nonterminal] = new_production_idxes;
        vector<int> left_recursive_production_idxes; //含左递归的产生式序号集
        for (int i : production_idxes[nonterminal]) { //遍历非终结符nonterminal的产生式
            const Symbol& first_symbol(productions[i].right.front());
            //若出现了左递归，即第一个符号是nonterminal
            if (first_symbol == nonterminal) {
```

```
            if (first_symbol == nonterminal) {
                //将该产生式从nonterminal的产生式集中删除
                new_production_idxes.erase(i);
                //再加入left_recursive_production_idxes中
                left_recursive_production_idxes.push_back(i);
            }
        }
        if (left_recursive_production_idxes.size()) { //若含有左递归
            production_idxes[nonterminal] = new_production_idxes;
            Symbol new_nonterminal(boost::get<string>(nonterminal) + "\'"); //新非终结符
            new_nonterminals.emplace_back(new_nonterminal);
            //消除nonterminal的直接左递归
            for (int i : production_idxes[nonterminal]) {
                productions[i].right.push_back(new_nonterminal);
            }
            production_idxes[nonterminal] = std::move(new_production_idxes);
            production_idxes[new_nonterminal].insert(productions.size());
            productions.emplace_back(new_nonterminal,
                                     std::move(deque<Symbol>{EPSILON}));
            for (int i : left_recursive_production_idxes) {
                productions[i].left = new_nonterminal;
                productions[i].right.pop_front();
                productions[i].right.push_back(new_nonterminal);
                production_idxes[new_nonterminal].insert(i);
            }
        }
        vis_nonterminals.insert(nonterminal);
    }
    for (auto& new_nonterminal : new_nonterminals) {
        nonterminals.emplace_back(std::move(new_nonterminal));
    }
}
```

对本实验中文法消除左递归后的输出如下：

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

5.3.3. 提取左公因子

若有产生式$A \longrightarrow \alpha\beta_1 \mid \alpha\beta_2$，则可提取左公因子将其改造为：

$$A \longrightarrow \alpha A_1$$
$$A_1 \longrightarrow \beta_1 \mid \beta_2$$

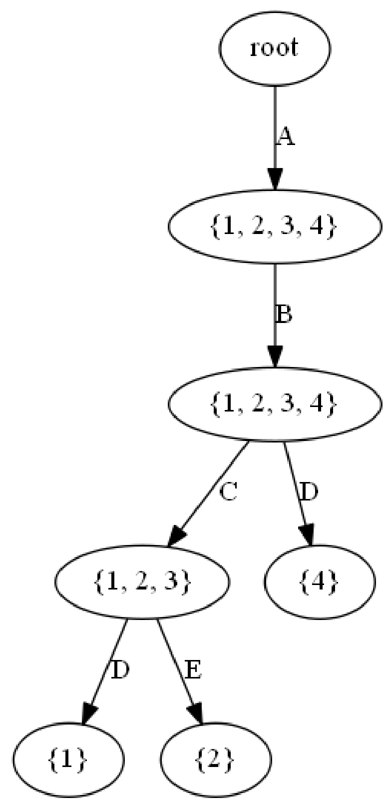
为消除预测分析时的不确定性，需对文法提取左公因子。提取左公因子是一个比较麻烦的问题，因一个非终结符的所有产生式可能含有不同的左公因子，并且对每个产生式都得提取出与其他若干个产生式最

长的左公因子。本程序使用了**Trie 树**这一数据结构高效地解决了这个问题，算法如下：

(1) 将所有产生式的右部都插入 Trie 树中，并在 Trie 树节点中保存所有经过该结点中的产生式序号。以文法

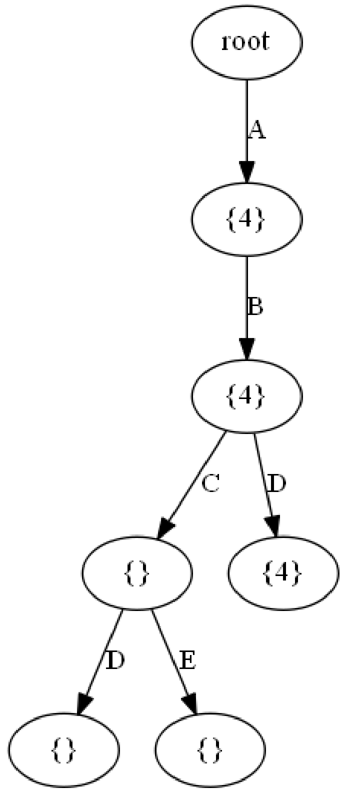
1. $S \longrightarrow ABCD$
2. $S \longrightarrow ABCE$
3. $S \longrightarrow ABC$
4. $S \longrightarrow ABD$

为例，插入 Trie 树后如下图：



(2) 对每一个产生式的右部，在其在 Trie 树上对应根到叶节点的路径中，找到第一个只有该产生式经过的结点，则其父节点中保存的产生式集即为与其有最长左公因子的产生式集。如对 1 号产生式，可查得与其有最长左公因子的产生式集为{1, 2, 3}。

(3) 将该产生式集中每一个产生式从 Trie 上删除，再对该集提取左公因子。对该例子得到：



并且产生式集{1, 2, 3}被改造为：

$$S \longrightarrow ABCS_1$$
$$S_1 \longrightarrow D \mid E \mid \varepsilon$$

(4) 对剩余的未被删除的产生式从步骤(1) 开始执行。

本程序中提取左公因子的算法实现如下：

```
void Grammar::extract_common_left_factor() {
    //记录对每个非终结符的产生式已提取了多少个左公因子,
    //其个数作为对应新非终结符后缀
    map<Symbol, int> new_nonterminal_suffixes;
    //一直循环直至所有左公因子被消除
    while (true) {
        bool has_common_left_factor(false);
        //将每个符号映射到一个整数
        map<Symbol, int> symbol_to_idx;
        int cur_idx(0);
        for (const auto& nonterminal : nonterminals) {
            symbol_to_idx[nonterminal] = cur_idx++;
        }
        for (const auto& terminal : terminals) {
            symbol_to_idx[terminal] = cur_idx++;
        }
        Trie<int, -1> trie(MAX_TRIE_SIZE, cur_idx);
        vector<Symbol> new_nonterminals;
```

```
    //maps each symbol in each production to its idx
    //for insertion in trie
    vector<vector<int>> int_mapped_productions(productions.size());
    for (const auto& nonterminal : nonterminals) {
        trie.clear();
        //对非终结符nonterminal的每个产生式，将其转化为整数映射值数组，插入到trie中
        for (int i : production_idxes[nonterminal]) {
            for (const auto& symbol : productions[i].right) {
                int_mapped_productions[i].push_back(symbol_to_idx[symbol]);
            }
            trie.insert<vector<int>, int>(int_mapped_productions[i], i);
        }
        set<int> remaining_production_idxes(production_idxes[nonterminal]);
        //clf = common left factor
        //in pair<vector<int>, int>:
        //vector<int> contanis indexes of productions shared clf
        //int indicates the length of their clf
        vector<pair<set<int>, int>> clf_shared_productions_sets;
        for (int i : production_idxes[nonterminal]) {
            if (remaining_production_idxes.find(i)
                != remaining_production_idxes.end()) {
                int cur(trie.root);
                int clf_len(0);
                //在该产生式对应trie树上路径中找第一个只有其经过的结点
                for (const auto& symbol : productions[i].right) {
                    int idx(symbol_to_idx[symbol]);
                    int nxt(trie.nodes[cur].next[idx]);
                    if (trie.nodes[nxt].vis_vals.size() == 1) {
                        if (trie.nodes[cur].vis_vals.size() > 1) {
                            //trie根节点到其父结点路径长度即为该产生式集的左公因子长度
                            clf_shared_productions_sets.emplace_back(
                                trie.nodes[cur].vis_vals, clf_len
                            );
                            //将经过其父节点的产生式从trie中删除
```

```
                            for (int i : clf_shared_productions_sets.back().first) {
                                remaining_production_idxes.erase(i);
                                trie.erase<vector<int>, int>(int_mapped_productions[i], i);
                            }
                            break;
                        }
                    }
                    cur = nxt;
                    ++clf_len;
                }
            }
        }
        if (clf_shared_productions_sets.size()) {
            has_common_left_factor = true;
            production_idxes[nonterminal] = std::move(remaining_production_idxes);
            //int new_nonterminal_suffix(1);
            for (auto& clf_shared_productions : clf_shared_productions_sets) {
                //对每一个含左公因子的产生式集都得增加一个新非终结符
                Symbol new_nonterminal(
                    boost::get<string>(nonterminal)
                    + "_" + lexical_cast<string>(new_nonterminal_suffixes[nonterminal]++)
                );
                new_nonterminals.emplace_back(new_nonterminal);
                Production new_production(nonterminal);
                int clf_len(clf_shared_productions.second);
                auto clf_shared_production_idxes(
                    std::move(clf_shared_productions.first)
                );
                bool has_init_new_production(false);
                //提取左公因子
                for (int i : clf_shared_production_idxes) {
                    productions[i].left = new_nonterminal;
                    for (int j(0); j < clf_len; ++j) {
                        if (!has_init_new_production) {
                            new_production.right.emplace_back(
```

```
                                    std::move(productions[i].right.front())
                                );
                            }
                            productions[i].right.pop_front();
                        }
                        has_init_new_production = true;
                        if (productions[i].right.empty()) {
                            productions[i].right.emplace_back(EPSILON);
                        }
                        production_idxes[new_nonterminal].insert(i);
                    }
                    production_idxes[nonterminal].insert(productions.size());
                    new_production.right.emplace_back(std::move(new_nonterminal));
                    productions.emplace_back(std::move(new_production));
                }
            }
        }
        //若新左公因子，即有新产生式，需继续循环
        if (has_common_left_factor) {
            for (auto& new_nonterminal : new_nonterminals) {
                nonterminals.emplace_back(std::move(new_nonterminal));
            }
        } else {
            break;
        }
    }
}
```

本实验中文法不含左公因子，以上述文法为例，程序输出为：

```
S -> A B S_1
S_0 -> D
S_0 -> E
S_0 -> epsilon
S_1 -> D
S_1 -> C S_0
```

5.3.4. 构造 FIRST 集

对任意文法符号串，其 FIRST 集为其可推导出的开头终结符号集合。

构造任意文法符号X的 FIRST 集$\mathrm{FIRST}(X)$可遍历X的产生式并利用如下规则：

(1) 若$X \in V_\mathrm{T}$，则$\mathrm{FIRST}(X) = \{X\}$；

(2) 若$X \in V_\mathrm{N}$，且有$X \to a \ldots (a \in V_\mathrm{T} \vee a = \varepsilon)$，则$\mathrm{FIRST}(X) = \mathrm{FIRST}(\mathrm{X}) \cup \{a\}$；

(3) 若有$X \to Y_1 Y_2 \ldots Y_k Y_{k+1} \ldots$，且有$\varepsilon \in \mathrm{FIRST}(Y_i)(i = 1, \ldots, k)$，则$\mathrm{FIRST}(X) = \mathrm{FIRST}(\mathrm{X}) \cup \{a \in \mathrm{FIRST}(Y_i) \mid a \neq \varepsilon\}$；
若$\forall Y_i, \varepsilon \in \mathrm{FIRST}(Y_i)$，$\mathrm{FIRST}(X) = \mathrm{FIRST}(\mathrm{X}) \cup \{\varepsilon\}$。

该程序中构造 FIRST 集的算法实现如下：

```cpp
//构造非终结符nonterminal的FIRST集
void Grammar::construct_first(const Symbol& nonterminal) {
    for (int i : production_idxes[nonterminal]) { //遍历nonterminal的产生式
        const auto& production(productions[i]);
        bool all_has_epsilon(true); //标记是否产生式所有符号的FIRST集都含epsilon
        for (const auto& symbol : production.right) { //遍历产生式的符号
            //若该符号是终结符，将其加入该产生式的FIRST集中并退出
            if (symbol.which() == TERMINAL) {
                first_of_production[i].insert(symbol);
                break;
            }
            //若还未构造该符号的FIRST集，递归构造该符号的FIRST集
            if (!has_constructed_first[symbol]) {
                construct_first(symbol);
            }
            //将该符号的FIRST集加入到该产生式的FIRST集中
            first_of_production[i].insert(
                first[symbol].begin(),
                first[symbol].end()
            );
            //若该符号的FIRST集中含epsilon，继续遍历，否则退出
            if (first[symbol].find(EPSILON) == first[symbol].end()) {
                all_has_epsilon = false;
                break;
            } else {
                //去掉epsilon
                first_of_production[i].erase(EPSILON);
            }
        }
        //若都含epsilon加入到，将epsilon加入到该产生式的FIRST集中
        if (all_has_epsilon) {
            first_of_production[i].insert(EPSILON);
        }
        //将该产生式的FIRST集加入到nonterminal的FIRST集中
        first[nonterminal].insert(
            first_of_production[i].begin(),
            first_of_production[i].end()
        );
    }
    //标记已构造nonterminal的FIRST集
    has_constructed_first[nonterminal] = true;
}
```

对该实验中文法构造 FIRST 集，输出如下：

```
first[E] = {(, num}
first[T] = {(, num}
first[F] = {(, num}
first[E'] = {+, -, epsilon}
first[T'] = {/, *, epsilon}
```

5.3.5. 构造 FOLLOW 集

对任意非终结符，其 FOLLOW 集是该文法所有句型中紧跟在其后的终结符或结尾符号 `end` 的集合。

构造任意文法符号X的 FOLLOW 集$\mathrm{FOLLOW}(X)$可遍历文法所有产生式并利用如下规则：

(1) 若X是起始符号，则$\mathrm{FOLLOW}(X) = \mathrm{FOLLOW}(X) \cup \{\mathrm{end}\}$；

(2) 若有$A \to \cdots X\beta$，则$\mathrm{FOLLOW}(X) = \mathrm{FOLLOW}(X) \cup \{a \in \mathrm{FIRST}(\beta) \mid a \neq \varepsilon\}$；

(3) 若有$A \to \cdots X$，或有$A \to \cdots X\beta$且$\varepsilon \in \mathrm{FIRST}(\beta)$，则$\mathrm{FOLLOW}(X) = \mathrm{FOLLOW}(X) \cup \mathrm{FOLLOW}(A)$。

由于两个非终结符的 FOLLOW 集完全有可能互相包含，此时若在求其中一个的 FOLLOW 直接递归求另一个的 FOLLOW 集，则会出现无穷递归的死循环。为解决这个问题，注意到

$$\left.\begin{aligned}\mathrm{FOLLOW}(A) \subseteq \mathrm{FOLLOW}(B)\\ \mathrm{FOLLOW}(B) \subseteq \mathrm{FOLLOW}(A)\end{aligned}\right\} \Longrightarrow \mathrm{FOLLOW}(A) = \mathrm{FOLLOW}(B),$$

故本程序利用二维数组`includes_follow_of`记录下两个非终结符的 FOLLOW 集的包含关系。以 A，B 为例，若在构造 A 的 FOLLOW 集时需用到 B 的 FOLLOW 集，则`includes_follow_of`[A][B]为 true，并递归对 B 构造 FOLLOW 集，此时若又需用到 A 的 FOLLOW 集，则`includes_follow_of`[B][A]为 true，但由于`includes_follow_of`[A][B]为 true，不会再递归构造 A 的 FOLLOW 集。最后由于`includes_follow_of`[A][B]与`includes_follow_of`[B][A]都为 true，可判断$\mathrm{FOLLOW}(A) = \mathrm{FOLLOW}(B)$，将$\mathrm{FOLLOW}(A)$与$\mathrm{FOLLOW}(B)$的并集赋给彼此即可。

该程序中构造 FOLLOW 集的算法实现如下：

```
//构造非终结符nonterminal的FOLLOW集
void Grammar::construct_follow(const Symbol& nonterminal) {
    //遍历文法所有产生式
    for (const auto& tmp_nonterminal : nonterminals) {
        for (int i : production_idxes[tmp_nonterminal]) {
            const auto& production(productions[i]);
            for (int j(0), k; j < production.right.size(); ++j) {
                if (production.right[j] == nonterminal) { //生成式中出现nonterminal
                    for (k = j + 1; k < production.right.size(); ++k) {
                        const auto& symbol(production.right[k]);
                        if (symbol.which() == TERMINAL) { //终结符直接加入并退出
                            follow[nonterminal].insert(symbol);
```

```cpp
                    break;
                }
                //依次加入后续符号的FIRST集除非该符号的FIRST集不含epsilon
                follow[nonterminal].insert(
                    first[symbol].begin(),
                    first[symbol].end()
                );
                if (first[symbol].find(EPSILON) == first[symbol].end()) {
                    break;
                } else {
                    follow[nonterminal].erase(EPSILON);
                }
            }
            //若无后续符号或后续符号串的FIRST集都含epsilon
            if (k == production.right.size()
                && production.left != nonterminal) {
                const auto& another_nonterminal(production.left);
                //标记nonterminal的FOLLOW集包含another_nonterminal的FOLLOW集
                includes_follow_of[nonterminal][another_nonterminal] = true;
                //若未标记another_nonterminal的FOLLOW集包含nonterminal的FOLLOW集
                if (!includes_follow_of[another_nonterminal][nonterminal]) {
                    //若还未构造another_nonterminal的FOLLOW集,
                    //递归构造another_nonterminal的FOLLOW集
                    if (!has_constructed_follow[another_nonterminal]) {
                        construct_follow(another_nonterminal);
                    }
                    //将another_nonterminal的FOLLOW集加入到nonterminal的FOLLOW集
                    follow[nonterminal].insert(
                        follow[another_nonterminal].begin(),
                        follow[another_nonterminal].end()
                    );
                }
            }
        }
    }
}
    //标记已构造nonterminal的FOLLOW集
    has_constructed_follow[nonterminal] = true;
}
```

最后还应处理 FOLLOW 集互相包含的情况：

```cpp
    for (const auto& nonterminal : nonterminals) {
        for (const auto& another_nonterminal : nonterminals) {
            if (includes_follow_of[nonterminal][another_nonterminal]
                && includes_follow_of[another_nonterminal][nonterminal]) {
                follow[nonterminal].insert(
                    follow[another_nonterminal].begin(),
                    follow[another_nonterminal].end()
                );
                follow[another_nonterminal] = follow[nonterminal];
            }
        }
    }
```

对该实验中文法构造 FOLLOW 集，输出如下：

```
follow[E] = {), end}
follow[T] = {+, -, ), end}
follow[F] = {+, -, /, *, ), end}
follow[E'] = {), end}
follow[T'] = {+, -, ), end}
```

5.4. Parser

该模块定义了语法分析器类大致如下：

```
class Parser {
    //重载流操作符让parser在lexer词法分析的结果上进行语法分析
    friend Parser& operator >> (const Lexer& lexer, Parser& parser);

private:
    Grammar grammar; //文法
    map<Symbol, map<Symbol, int>> parsing_table; //分析表

    void construct_parsing_table(); //构造分析表

public:
    enum {
        SYNCH = -1 //错误处理标志
    };

    Parser() {
        construct_parsing_table();
    }
};
```

5.4.1. 构造预测分析表

预测分析表是预测分析程序工作的依据，其是一个二维表，表项 `parsing_table`[nonterminal][terminal]是非终结符 nonterminal 遇到非终结符 terminal（可能为 `end`）时的分析动作指示，如采用某个产生式进行最左推导或错误提示。

构造预测分析表的算法实现如下：

```
void Parser::construct_parsing_table() {
    grammar.remove_left_recursion(); //消除左递归
    grammar.extract_common_left_factor(); //提取左公因子
    grammar.construct_first(); //构造FIRST集
    grammar.construct_follow(); //构造FOLLOW集
    for (const auto& nonterminal : grammar.nonterminals) { //遍历非终结符集
        for (int i : grammar.production_idxes[nonterminal]) { //遍历其产生式集
            bool has_epsilon = false; //标记是否为空产生式
            //对该产生式的FIRST集中的终结符，将该产生式填入对应表项中
            for (const auto& terminal : grammar.first_of_production[i]) {
                if (boost::get<TokenType>(terminal) != EPSILON) {
                    parsing_table[nonterminal][terminal] = i;
                } else {
                    has_epsilon = true;
                }
            }
```

```
                //若是空产生式
                if (has_epsilon) {
                    //对该非终结符FOLLOW集中的终结符，将该产生式填入对应表项中
                    for (const auto& terminal : grammar.follow[nonterminal]) {
                        parsing_table[nonterminal][terminal] = i;
                    }
                }
            }
            //对于该非终结符FOLLOW集中的终结符，若对应表项为空，填入用于错误处理的同步信息SYNCH
            for (const auto& terminal : grammar.follow[nonterminal]) {
                if (parsing_table[nonterminal].find(terminal)
                    == parsing_table[nonterminal].end()) {
                    parsing_table[nonterminal][terminal] = SYNCH;
                }
            }
        }
}
```

5.4.2. 预测分析程序

本非递归预测分析程序使用了一个输入缓冲区（词法分析得到的 `token_stream`）、一个分析栈（`parsing_stack`）与一张分析表（`parsing_table`），输出为每次最左推导使用的产生式。其中分析栈存放了待扫描 token 流的句型，每次依据栈顶符号与带扫描 token 流的第一个 token（终结符），有分析表得到当前的分析动作（移进扫描指针、使用产生式替换栈顶或给出错误提示）。具体实现如下：

```
Parser& operator >> (const Lexer& lexer, Parser& parser) {
    //经词法分析得到的token流
    const vector<Token>& token_stream(lexer.get_token_stream());
    //分析栈
    stack<Symbol> parsing_stack;
    //左句型
    pair<deque<Symbol>, deque<Symbol>> left_sentencial_form{
        {},{parser.grammar.get_start_symbol()}
    };
    //将结尾符号与文法起始符号压入分析栈
    parsing_stack.push(END);
    parsing_stack.push(parser.grammar.get_start_symbol());
    const vector<Production>& productions(parser.grammar.get_productions());
    //扫描token流
    for (int i(0); i < token_stream.size(); ) {
        out << "current left sentencial form:\n\t\t\t\t";
        for (const auto& symbol : left_sentencial_form.first) {
            print_symbol(out, symbol);
            out << " ";
        }
        for (const auto& symbol : left_sentencial_form.second) {
            print_symbol(out, symbol);
            out << " ";
        }
        out << endl;
        out << "current token stream:\n\t\t\t\t";
        for (int j(i); j < token_stream.size(); ++j) {
            print_symbol(out, token_stream[j].type);
```

```
            out << " ";
        }
        out << endl;
        out << "output:\n\t\t\t\t";
        const auto& token(token_stream[i]);
        //若当前分析栈顶为终结符
        if (parsing_stack.top().which() == TERMINAL) {
            //若该终结符与待输入token流第一个token匹配
            if (boost::get<TokenType>(parsing_stack.top()) == token.type) {
                //指针移进
                ++i;
            } else {
                //否则出现错误，给出错误信息：此处需要一个栈顶终结符类型的token（后续会弹栈）
                out << "error: ";
                print_symbol(out, parsing_stack.top());
                out << " expected\n";
            }
            //弹栈
            parsing_stack.pop();
            //更新当前左句型
            if (left_sentencial_form.second.size()) {
                left_sentencial_form.first.push_back(
                    left_sentencial_form.second.front()
                );
                left_sentencial_form.second.pop_front();
            }
        } else {
            //栈顶为非终结符，在分析表中查找对应动作
            auto res(parser.parsing_table[parsing_stack.top()].find(token.type));
            if (res != parser.parsing_table[parsing_stack.top()].end()) {
                //若表项非空白，弹栈
                Symbol nonterminal(std::move(parsing_stack.top()));
                parsing_stack.pop();
                left_sentencial_form.second.pop_front();
                if (res->second != Parser::SYNCH) {
                    //若表项不是同步信息SYNCH,
                    //将使用对应产生式压栈，并更新当前左句型
                    const Production& production(productions[res->second]);
                    out << production;
                    if (!(production.right.front().which() == TERMINAL
                          && boost::get<TokenType>(production.right.front())
                             == EPSILON)) {
                        for (int j(production.right.size() - 1); ~j; --j) {
                            parsing_stack.push(production.right[j]);
                        }
                        left_sentencial_form.second.insert(
                            left_sentencial_form.second.begin(),
                            production.right.begin(),
                            production.right.end()
                        );
                    }
                } else {
                    //若为SYNCH给出错误提示：此处需要一个栈顶非终结符能推导出的句子（已弹栈）
                    out << "error: " << nonterminal << " expected\n";
                }
            } else {
                //若表项为空白，将指针移进并给出错误提示：此处不需要该token
                out << "error: ";
                print_symbol(out, Symbol(token_stream[i].type));
                out << " unexpected\n";
                ++i;
            }
        }
        out << endl;
    }
    out << endl;
    return parser;
}
```

6. 程序测试

以下为测试样例，分为正确表达式与错误表达式两部分，样例格式如下：

```
（输入算术表达式）
current left sentencial form:
                        （当前左句型）
current token stream:
                        （当前待扫描 token 流） 注：已经词法分析将符号串转为 token 流
output:
                        （输出产生式）或（错误信息）
```

6.1. 正确表达式

```
1
current left sentencial form:
                        E
current token stream:
                        num end
output:
                        E -> T E'

current left sentencial form:
                        T E'
current token stream:
                        num end
output:
                        T -> F T'

current left sentencial form:
                        F T' E'
current token stream:
                        num end
output:
                        F -> num

current left sentencial form:
                        num T' E'
current token stream:
                        num end
output:

current left sentencial form:
                        num T' E'
current token stream:
                        end
output:
                        T' -> epsilon

current left sentencial form:
                        num E'
```

```
current token stream:
                        end
output:
                        E' -> epsilon

current left sentencial form:
                        num
current token stream:
                        end
output:
```

```
1+1
```

```
current left sentencial form:
                        E
current token stream:
                        num + num end
output:
                        E -> T E'

current left sentencial form:
                        T E'
current token stream:
                        num + num end
output:
                        T -> F T'

current left sentencial form:
                        F T' E'
current token stream:
                        num + num end
output:
                        F -> num

current left sentencial form:
                        num T' E'
current token stream:
                        num + num end
output:

current left sentencial form:
                        num T' E'
current token stream:
                        + num end
output:
                        T' -> epsilon

current left sentencial form:
                        num E'
current token stream:
                        + num end
output:
                        E' -> + T E'

current left sentencial form:
```

```
                        num + T E'
current token stream:
                        + num end
output:

current left sentencial form:
                        num + T E'
current token stream:
                        num end
output:
                        T -> F T'

current left sentencial form:
                        num + F T' E'
current token stream:
                        num end
output:
                        F -> num

current left sentencial form:
                        num + num T' E'
current token stream:
                        num end
output:

current left sentencial form:
                        num + num T' E'
current token stream:
                        end
output:
                        T' -> epsilon

current left sentencial form:
                        num + num E'
current token stream:
                        end
output:
                        E' -> epsilon

current left sentencial form:
                        num + num
current token stream:
                        end
output:
```

```
2.3+4.5e6
current left sentencial form:
                        E
current token stream:
                        num + num end
output:
                        E -> T E'

current left sentencial form:
```

```
                        T E'
current token stream:
                        num + num end
output:
                        T -> F T'

current left sentencial form:
                        F T' E'
current token stream:
                        num + num end
output:
                        F -> num

current left sentencial form:
                        num T' E'
current token stream:
                        num + num end
output:

current left sentencial form:
                        num T' E'
current token stream:
                        + num end
output:
                        T' -> epsilon

current left sentencial form:
                        num E'
current token stream:
                        + num end
output:
                        E' -> + T E'

current left sentencial form:
                        num + T E'
current token stream:
                        + num end
output:

current left sentencial form:
                        num + T E'
current token stream:
                        num end
output:
                        T -> F T'

current left sentencial form:
                        num + F T' E'
current token stream:
                        num end
output:
                        F -> num

current left sentencial form:
                        num + num T' E'
current token stream:
                        num end
```

```
output:

current left sentencial form:
                        num + num T' E'
current token stream:
                        end
output:
                        T' -> epsilon

current left sentencial form:
                        num + num E'
current token stream:
                        end
output:
                        E' -> epsilon

current left sentencial form:
                        num + num
current token stream:
                        end
output:
```

```
(1 + 3) * (3 / 2 + 4)
current left sentencial form:
                        E
current token stream:
                        ( num + num ) * ( num / num + num ) end
output:
                        E -> T E'

current left sentencial form:
                        T E'
current token stream:
                        ( num + num ) * ( num / num + num ) end
output:
                        T -> F T'

current left sentencial form:
                        F T' E'
current token stream:
                        ( num + num ) * ( num / num + num ) end
output:
                        F -> ( E )

current left sentencial form:
                        ( E ) T' E'
current token stream:
                        ( num + num ) * ( num / num + num ) end
output:

current left sentencial form:
                        ( E ) T' E'
current token stream:
                        num + num ) * ( num / num + num ) end
```

```
output:
                        E -> T E'

current left sentencial form:
                        ( T E' ) T' E'
current token stream:
                        num + num ) * ( num / num + num ) end
output:
                        T -> F T'

current left sentencial form:
                        ( F T' E' ) T' E'
current token stream:
                        num + num ) * ( num / num + num ) end
output:
                        F -> num

current left sentencial form:
                        ( num T' E' ) T' E'
current token stream:
                        num + num ) * ( num / num + num ) end
output:

current left sentencial form:
                        ( num T' E' ) T' E'
current token stream:
                        + num ) * ( num / num + num ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( num E' ) T' E'
current token stream:
                        + num ) * ( num / num + num ) end
output:
                        E' -> + T E'

current left sentencial form:
                        ( num + T E' ) T' E'
current token stream:
                        + num ) * ( num / num + num ) end
output:

current left sentencial form:
                        ( num + T E' ) T' E'
current token stream:
                        num ) * ( num / num + num ) end
output:
                        T -> F T'

current left sentencial form:
                        ( num + F T' E' ) T' E'
current token stream:
                        num ) * ( num / num + num ) end
output:
                        F -> num
```

```
current left sentencial form:
                        ( num + num T' E' ) T' E'
current token stream:
                        num ) * ( num / num + num ) end
output:

current left sentencial form:
                        ( num + num T' E' ) T' E'
current token stream:
                        ) * ( num / num + num ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( num + num E' ) T' E'
current token stream:
                        ) * ( num / num + num ) end
output:
                        E' -> epsilon

current left sentencial form:
                        ( num + num ) T' E'
current token stream:
                        ) * ( num / num + num ) end
output:

current left sentencial form:
                        ( num + num ) T' E'
current token stream:
                        * ( num / num + num ) end
output:
                        T' -> * F T'

current left sentencial form:
                        ( num + num ) * F T' E'
current token stream:
                        * ( num / num + num ) end
output:

current left sentencial form:
                        ( num + num ) * F T' E'
current token stream:
                        ( num / num + num ) end
output:
                        F -> ( E )

current left sentencial form:
                        ( num + num ) * ( E ) T' E'
current token stream:
                        ( num / num + num ) end
output:

current left sentencial form:
                        ( num + num ) * ( E ) T' E'
current token stream:
                        num / num + num ) end
output:
```

```
                        E -> T E'

current left sentencial form:
                        ( num + num ) * ( T E' ) T' E'
current token stream:
                        num / num + num ) end
output:
                        T -> F T'

current left sentencial form:
                        ( num + num ) * ( F T' E' ) T' E'
current token stream:
                        num / num + num ) end
output:
                        F -> num

current left sentencial form:
                        ( num + num ) * ( num T' E' ) T' E'
current token stream:
                        num / num + num ) end
output:

current left sentencial form:
                        ( num + num ) * ( num T' E' ) T' E'
current token stream:
                        / num + num ) end
output:
                        T' -> / F T'

current left sentencial form:
                        ( num + num ) * ( num / F T' E' ) T' E'
current token stream:
                        / num + num ) end
output:

current left sentencial form:
                        ( num + num ) * ( num / F T' E' ) T' E'
current token stream:
                        num + num ) end
output:
                        F -> num

current left sentencial form:
                        ( num + num ) * ( num / num T' E' ) T' E'
current token stream:
                        num + num ) end
output:

current left sentencial form:
                        ( num + num ) * ( num / num T' E' ) T' E'
current token stream:
                        + num ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( num + num ) * ( num / num E' ) T' E'
```

```
current token stream:
                        + num ) end
output:
                        E' -> + T E'

current left sentencial form:
                        ( num + num ) * ( num / num + T E' ) T' E'
current token stream:
                        + num ) end
output:

current left sentencial form:
                        ( num + num ) * ( num / num + T E' ) T' E'
current token stream:
                        num ) end
output:
                        T -> F T'

current left sentencial form:
                        ( num + num ) * ( num / num + F T' E' ) T' E'
current token stream:
                        num ) end
output:
                        F -> num

current left sentencial form:
                        ( num + num ) * ( num / num + num T' E' ) T' E'
current token stream:
                        num ) end
output:

current left sentencial form:
                        ( num + num ) * ( num / num + num T' E' ) T' E'
current token stream:
                        ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( num + num ) * ( num / num + num E' ) T' E'
current token stream:
                        ) end
output:
                        E' -> epsilon

current left sentencial form:
                        ( num + num ) * ( num / num + num ) T' E'
current token stream:
                        ) end
output:

current left sentencial form:
                        ( num + num ) * ( num / num + num ) T' E'
current token stream:
                        end
output:
                        T' -> epsilon
```

```
current left sentencial form:
                        ( num + num ) * ( num / num + num ) E'
current token stream:
                        end
output:
                        E' -> epsilon

current left sentencial form:
                        ( num + num ) * ( num / num + num )
current token stream:
                        end
output:
```

```
(3.2 + 6.9)
current left sentencial form:
                        E
current token stream:
                        ( num + num ) end
output:
                        E -> T E'

current left sentencial form:
                        T E'
current token stream:
                        ( num + num ) end
output:
                        T -> F T'

current left sentencial form:
                        F T' E'
current token stream:
                        ( num + num ) end
output:
                        F -> ( E )

current left sentencial form:
                        ( E ) T' E'
current token stream:
                        ( num + num ) end
output:

current left sentencial form:
                        ( E ) T' E'
current token stream:
                        num + num ) end
output:
                        E -> T E'

current left sentencial form:
                        ( T E' ) T' E'
current token stream:
                        num + num ) end
output:
```

```
                        T -> F T'

current left sentencial form:
                        ( F T' E' ) T' E'
current token stream:
                        num + num ) end
output:
                        F -> num

current left sentencial form:
                        ( num T' E' ) T' E'
current token stream:
                        num + num ) end
output:

current left sentencial form:
                        ( num T' E' ) T' E'
current token stream:
                        + num ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( num E' ) T' E'
current token stream:
                        + num ) end
output:
                        E' -> + T E'

current left sentencial form:
                        ( num + T E' ) T' E'
current token stream:
                        + num ) end
output:

current left sentencial form:
                        ( num + T E' ) T' E'
current token stream:
                        num ) end
output:
                        T -> F T'

current left sentencial form:
                        ( num + F T' E' ) T' E'
current token stream:
                        num ) end
output:
                        F -> num

current left sentencial form:
                        ( num + num T' E' ) T' E'
current token stream:
                        num ) end
output:

current left sentencial form:
                        ( num + num T' E' ) T' E'
```

```
current token stream:
                        ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( num + num E' ) T' E'
current token stream:
                        ) end
output:
                        E' -> epsilon

current left sentencial form:
                        ( num + num ) T' E'
current token stream:
                        ) end
output:

current left sentencial form:
                        ( num + num ) T' E'
current token stream:
                        end
output:
                        T' -> epsilon

current left sentencial form:
                        ( num + num ) E'
current token stream:
                        end
output:
                        E' -> epsilon

current left sentencial form:
                        ( num + num )
current token stream:
                        end
output:
```

((((((4))))))

```
current left sentencial form:
                        E
current token stream:
                        ( ( ( ( ( ( num ) ) ) ) ) ) end
output:
                        E -> T E'

current left sentencial form:
                        T E'
current token stream:
                        ( ( ( ( ( ( num ) ) ) ) ) ) end
output:
                        T -> F T'

current left sentencial form:
```

```
                        F T' E'
current token stream:
                        ( ( ( ( ( ( num ) ) ) ) ) ) end
output:
                        F -> ( E )

current left sentencial form:
                        ( E ) T' E'
current token stream:
                        ( ( ( ( ( ( num ) ) ) ) ) ) end
output:

current left sentencial form:
                        ( E ) T' E'
current token stream:
                        ( ( ( ( ( num ) ) ) ) ) ) end
output:
                        E -> T E'

current left sentencial form:
                        ( T E' ) T' E'
current token stream:
                        ( ( ( ( ( num ) ) ) ) ) ) end
output:
                        T -> F T'

current left sentencial form:
                        ( F T' E' ) T' E'
current token stream:
                        ( ( ( ( ( num ) ) ) ) ) ) end
output:
                        F -> ( E )

current left sentencial form:
                        ( ( E ) T' E' ) T' E'
current token stream:
                        ( ( ( ( ( num ) ) ) ) ) ) end
output:

current left sentencial form:
                        ( ( E ) T' E' ) T' E'
current token stream:
                        ( ( ( ( num ) ) ) ) ) ) end
output:
                        E -> T E'

current left sentencial form:
                        ( ( T E' ) T' E' ) T' E'
current token stream:
                        ( ( ( ( num ) ) ) ) ) ) end
output:
                        T -> F T'

current left sentencial form:
                        ( ( F T' E' ) T' E' ) T' E'
current token stream:
                        ( ( ( ( num ) ) ) ) ) ) end
```

```
output:
                        F -> ( E )

current left sentencial form:
                        ( ( ( E ) T' E' ) T' E' ) T' E'
current token stream:
                        ( ( ( ( num ) ) ) ) ) ) end
output:

current left sentencial form:
                        ( ( ( E ) T' E' ) T' E' ) T' E'
current token stream:
                        ( ( ( num ) ) ) ) ) ) end
output:
                        E -> T E'

current left sentencial form:
                        ( ( ( T E' ) T' E' ) T' E' ) T' E'
current token stream:
                        ( ( ( num ) ) ) ) ) ) end
output:
                        T -> F T'

current left sentencial form:
                        ( ( ( F T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        ( ( ( num ) ) ) ) ) ) end
output:
                        F -> ( E )

current left sentencial form:
                        ( ( ( ( E ) T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        ( ( ( num ) ) ) ) ) ) end
output:

current left sentencial form:
                        ( ( ( ( E ) T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        ( ( num ) ) ) ) ) ) end
output:
                        E -> T E'

current left sentencial form:
                        ( ( ( ( T E' ) T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        ( ( num ) ) ) ) ) ) end
output:
                        T -> F T'

current left sentencial form:
                        ( ( ( ( F T' E' ) T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        ( ( num ) ) ) ) ) ) end
output:
                        F -> ( E )
```

```
current left sentencial form:
                        ( ( ( ( ( E ) T' E' ) T' E' ) T' E' ) T' E' ) T'
E'
current token stream:
                        ( ( num ) ) ) ) ) ) end
output:

current left sentencial form:
                        ( ( ( ( ( E ) T' E' ) T' E' ) T' E' ) T' E' ) T'
E'
current token stream:
                        ( num ) ) ) ) ) ) end
output:
                        E -> T E'

current left sentencial form:
                        ( ( ( ( ( T E' ) T' E' ) T' E' ) T' E' ) T' E' )
T' E'
current token stream:
                        ( num ) ) ) ) ) ) end
output:
                        T -> F T'

current left sentencial form:
                        ( ( ( ( ( F T' E' ) T' E' ) T' E' ) T' E' ) T'
E' ) T' E'
current token stream:
                        ( num ) ) ) ) ) ) end
output:
                        F -> ( E )

current left sentencial form:
                        ( ( ( ( ( ( E ) T' E' ) T' E' ) T' E' ) T' E' )
T' E' ) T' E'
current token stream:
                        ( num ) ) ) ) ) ) end
output:

current left sentencial form:
                        ( ( ( ( ( ( E ) T' E' ) T' E' ) T' E' ) T' E' )
T' E' ) T' E'
current token stream:
                        num ) ) ) ) ) ) end
output:
                        E -> T E'

current left sentencial form:
                        ( ( ( ( ( ( T E' ) T' E' ) T' E' ) T' E' ) T'
E' ) T' E' ) T' E'
current token stream:
                        num ) ) ) ) ) ) end
output:
                        T -> F T'

current left sentencial form:
                        ( ( ( ( ( ( F T' E' ) T' E' ) T' E' ) T' E' ) T'
E' ) T' E' ) T' E'
```

```
current token stream:
                        num ) ) ) ) ) ) end
output:
                        F -> num

current left sentencial form:
                        ( ( ( ( ( ( num T' E' ) T' E' ) T' E' ) T' E' )
T' E' ) T' E' ) T' E'
current token stream:
                        num ) ) ) ) ) ) end
output:

current left sentencial form:
                        ( ( ( ( ( ( num T' E' ) T' E' ) T' E' ) T' E' )
T' E' ) T' E' ) T' E'
current token stream:
                        ) ) ) ) ) ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( ( ( ( ( num E' ) T' E' ) T' E' ) T' E' ) T'
E' ) T' E' ) T' E'
current token stream:
                        ) ) ) ) ) ) end
output:
                        E' -> epsilon

current left sentencial form:
                        ( ( ( ( ( ( num ) T' E' ) T' E' ) T' E' ) T'
E' ) T' E' ) T' E'
current token stream:
                        ) ) ) ) ) ) end
output:

current left sentencial form:
                        ( ( ( ( ( ( num ) T' E' ) T' E' ) T' E' ) T'
E' ) T' E' ) T' E'
current token stream:
                        ) ) ) ) ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( ( ( ( ( num ) E' ) T' E' ) T' E' ) T' E' )
T' E' ) T' E'
current token stream:
                        ) ) ) ) ) end
output:
                        E' -> epsilon

current left sentencial form:
                        ( ( ( ( ( ( num ) ) T' E' ) T' E' ) T' E' ) T'
E' ) T' E'
current token stream:
                        ) ) ) ) ) end
output:
```

```
current left sentencial form:
                        ( ( ( ( ( ( num ) ) T' E' ) T' E' ) T' E' ) T'
E' ) T' E'
current token stream:
                        ) ) ) ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( ( ( ( ( num ) ) E' ) T' E' ) T' E' ) T' E' )
T' E'
current token stream:
                        ) ) ) ) end
output:
                        E' -> epsilon

current left sentencial form:
                        ( ( ( ( ( ( num ) ) ) T' E' ) T' E' ) T' E' ) T'
E'
current token stream:
                        ) ) ) ) end
output:

current left sentencial form:
                        ( ( ( ( ( ( num ) ) ) T' E' ) T' E' ) T' E' ) T'
E'
current token stream:
                        ) ) ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( ( ( ( ( num ) ) ) E' ) T' E' ) T' E' ) T' E'
current token stream:
                        ) ) ) end
output:
                        E' -> epsilon

current left sentencial form:
                        ( ( ( ( ( ( num ) ) ) ) T' E' ) T' E' ) T' E'
current token stream:
                        ) ) ) end
output:

current left sentencial form:
                        ( ( ( ( ( ( num ) ) ) ) T' E' ) T' E' ) T' E'
current token stream:
                        ) ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( ( ( ( ( num ) ) ) ) E' ) T' E' ) T' E'
current token stream:
                        ) ) end
output:
```

```
                        E' -> epsilon

current left sentencial form:
                        ( ( ( ( ( ( num ) ) ) ) ) T' E' ) T' E'
current token stream:
                        ) ) end
output:

current left sentencial form:
                        ( ( ( ( ( ( num ) ) ) ) ) T' E' ) T' E'
current token stream:
                        ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( ( ( ( ( num ) ) ) ) ) E' ) T' E'
current token stream:
                        ) end
output:
                        E' -> epsilon

current left sentencial form:
                        ( ( ( ( ( ( num ) ) ) ) ) ) T' E'
current token stream:
                        ) end
output:

current left sentencial form:
                        ( ( ( ( ( ( num ) ) ) ) ) ) T' E'
current token stream:
                        end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( ( ( ( ( num ) ) ) ) ) ) E'
current token stream:
                        end
output:
                        E' -> epsilon

current left sentencial form:
                        ( ( ( ( ( ( num ) ) ) ) ) )
current token stream:
                        end
output:
```

```
((0.2-9) *(5*9/9+(10)))
```

```
current left sentencial form:
                        E
current token stream:
                        ( ( num - num ) * ( num * num / num +
( num ) ) ) end
```

```
output:
                        E -> T E'

current left sentencial form:
                        T E'
current token stream:
                        ( ( num - num ) * ( num * num / num +
( num ) ) ) end
output:
                        T -> F T'

current left sentencial form:
                        F T' E'
current token stream:
                        ( ( num - num ) * ( num * num / num +
( num ) ) ) end
output:
                        F -> ( E )

current left sentencial form:
                        ( E ) T' E'
current token stream:
                        ( ( num - num ) * ( num * num / num +
( num ) ) ) end
output:

current left sentencial form:
                        ( E ) T' E'
current token stream:
                        ( num - num ) * ( num * num / num + ( num ) ) )
end
output:
                        E -> T E'

current left sentencial form:
                        ( T E' ) T' E'
current token stream:
                        ( num - num ) * ( num * num / num + ( num ) ) )
end
output:
                        T -> F T'

current left sentencial form:
                        ( F T' E' ) T' E'
current token stream:
                        ( num - num ) * ( num * num / num + ( num ) ) )
end
output:
                        F -> ( E )

current left sentencial form:
                        ( ( E ) T' E' ) T' E'
current token stream:
                        ( num - num ) * ( num * num / num + ( num ) ) )
end
output:
```

```
current left sentencial form:
                        ( ( E ) T' E' ) T' E'
current token stream:
                        num - num ) * ( num * num / num + ( num ) ) )
end
output:
                        E -> T E'

current left sentencial form:
                        ( ( T E' ) T' E' ) T' E'
current token stream:
                        num - num ) * ( num * num / num + ( num ) ) )
end
output:
                        T -> F T'

current left sentencial form:
                        ( ( F T' E' ) T' E' ) T' E'
current token stream:
                        num - num ) * ( num * num / num + ( num ) ) )
end
output:
                        F -> num

current left sentencial form:
                        ( ( num T' E' ) T' E' ) T' E'
current token stream:
                        num - num ) * ( num * num / num + ( num ) ) )
end
output:

current left sentencial form:
                        ( ( num T' E' ) T' E' ) T' E'
current token stream:
                        - num ) * ( num * num / num + ( num ) ) ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( num E' ) T' E' ) T' E'
current token stream:
                        - num ) * ( num * num / num + ( num ) ) ) end
output:
                        E' -> - T E'

current left sentencial form:
                        ( ( num - T E' ) T' E' ) T' E'
current token stream:
                        - num ) * ( num * num / num + ( num ) ) ) end
output:

current left sentencial form:
                        ( ( num - T E' ) T' E' ) T' E'
current token stream:
                        num ) * ( num * num / num + ( num ) ) ) end
output:
                        T -> F T'
```

```
current left sentencial form:
                        ( ( num - F T' E' ) T' E' ) T' E'
current token stream:
                        num ) * ( num * num / num + ( num ) ) ) end
output:
                        F -> num

current left sentencial form:
                        ( ( num - num T' E' ) T' E' ) T' E'
current token stream:
                        num ) * ( num * num / num + ( num ) ) ) end
output:

current left sentencial form:
                        ( ( num - num T' E' ) T' E' ) T' E'
current token stream:
                        ) * ( num * num / num + ( num ) ) ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( num - num E' ) T' E' ) T' E'
current token stream:
                        ) * ( num * num / num + ( num ) ) ) end
output:
                        E' -> epsilon

current left sentencial form:
                        ( ( num - num ) T' E' ) T' E'
current token stream:
                        ) * ( num * num / num + ( num ) ) ) end
output:

current left sentencial form:
                        ( ( num - num ) T' E' ) T' E'
current token stream:
                        * ( num * num / num + ( num ) ) ) end
output:
                        T' -> * F T'

current left sentencial form:
                        ( ( num - num ) * F T' E' ) T' E'
current token stream:
                        * ( num * num / num + ( num ) ) ) end
output:

current left sentencial form:
                        ( ( num - num ) * F T' E' ) T' E'
current token stream:
                        ( num * num / num + ( num ) ) ) end
output:
                        F -> ( E )

current left sentencial form:
                        ( ( num - num ) * ( E ) T' E' ) T' E'
current token stream:
```

```
                        ( num * num / num + ( num ) ) ) end
output:

current left sentencial form:
                        ( ( num - num ) * ( E ) T' E' ) T' E'
current token stream:
                        num * num / num + ( num ) ) ) end
output:
                        E -> T E'

current left sentencial form:
                        ( ( num - num ) * ( T E' ) T' E' ) T' E'
current token stream:
                        num * num / num + ( num ) ) ) end
output:
                        T -> F T'

current left sentencial form:
                        ( ( num - num ) * ( F T' E' ) T' E' ) T' E'
current token stream:
                        num * num / num + ( num ) ) ) end
output:
                        F -> num

current left sentencial form:
                        ( ( num - num ) * ( num T' E' ) T' E' ) T' E'
current token stream:
                        num * num / num + ( num ) ) ) end
output:

current left sentencial form:
                        ( ( num - num ) * ( num T' E' ) T' E' ) T' E'
current token stream:
                        * num / num + ( num ) ) ) end
output:
                        T' -> * F T'

current left sentencial form:
                        ( ( num - num ) * ( num * F T' E' ) T' E' ) T'
E'
current token stream:
                        * num / num + ( num ) ) ) end
output:

current left sentencial form:
                        ( ( num - num ) * ( num * F T' E' ) T' E' ) T'
E'
current token stream:
                        num / num + ( num ) ) ) end
output:
                        F -> num

current left sentencial form:
                        ( ( num - num ) * ( num * num T' E' ) T' E' ) T'
E'
current token stream:
                        num / num + ( num ) ) ) end
```

```
output:

current left sentencial form:
                        ( ( num - num ) * ( num * num T' E' ) T' E' ) T'
E'
current token stream:
                        / num + ( num ) ) ) end
output:
                        T' -> / F T'

current left sentencial form:
                        ( ( num - num ) * ( num * num / F T' E' ) T'
E' ) T' E'
current token stream:
                        / num + ( num ) ) ) end
output:

current left sentencial form:
                        ( ( num - num ) * ( num * num / F T' E' ) T'
E' ) T' E'
current token stream:
                        num + ( num ) ) ) end
output:
                        F -> num

current left sentencial form:
                        ( ( num - num ) * ( num * num / num T' E' ) T'
E' ) T' E'
current token stream:
                        num + ( num ) ) ) end
output:

current left sentencial form:
                        ( ( num - num ) * ( num * num / num T' E' ) T'
E' ) T' E'
current token stream:
                        + ( num ) ) ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( num - num ) * ( num * num / num E' ) T' E' )
T' E'
current token stream:
                        + ( num ) ) ) end
output:
                        E' -> + T E'

current left sentencial form:
                        ( ( num - num ) * ( num * num / num + T E' ) T'
E' ) T' E'
current token stream:
                        + ( num ) ) ) end
output:

current left sentencial form:
```

```
                        ( ( num - num ) * ( num * num / num + T E' ) T'
E' ) T' E'
current token stream:
                        ( num ) ) ) end
output:
                        T -> F T'

current left sentencial form:
                        ( ( num - num ) * ( num * num / num + F T' E' )
T' E' ) T' E'
current token stream:
                        ( num ) ) ) end
output:
                        F -> ( E )

current left sentencial form:
                        ( ( num - num ) * ( num * num / num + ( E ) T'
E' ) T' E' ) T' E'
current token stream:
                        ( num ) ) ) end
output:

current left sentencial form:
                        ( ( num - num ) * ( num * num / num + ( E ) T'
E' ) T' E' ) T' E'
current token stream:
                        num ) ) ) end
output:
                        E -> T E'

current left sentencial form:
                        ( ( num - num ) * ( num * num / num + ( T E' )
T' E' ) T' E' ) T' E'
current token stream:
                        num ) ) ) end
output:
                        T -> F T'

current left sentencial form:
                        ( ( num - num ) * ( num * num / num + ( F T'
E' ) T' E' ) T' E' ) T' E'
current token stream:
                        num ) ) ) end
output:
                        F -> num

current left sentencial form:
                        ( ( num - num ) * ( num * num / num + ( num T'
E' ) T' E' ) T' E' ) T' E'
current token stream:
                        num ) ) ) end
output:

current left sentencial form:
                        ( ( num - num ) * ( num * num / num + ( num T'
E' ) T' E' ) T' E' ) T' E'
current token stream:
```

```
                         ) ) ) end
output:
                         T' -> epsilon

current left sentencial form:
                         ( ( num - num ) * ( num * num / num + ( num E' )
T' E' ) T' E' ) T' E'
current token stream:
                         ) ) ) end
output:
                         E' -> epsilon

current left sentencial form:
                         ( ( num - num ) * ( num * num / num + ( num ) T'
E' ) T' E' ) T' E'
current token stream:
                         ) ) ) end
output:

current left sentencial form:
                         ( ( num - num ) * ( num * num / num + ( num ) T'
E' ) T' E' ) T' E'
current token stream:
                         ) ) end
output:
                         T' -> epsilon

current left sentencial form:
                         ( ( num - num ) * ( num * num / num + ( num )
E' ) T' E' ) T' E'
current token stream:
                         ) ) end
output:
                         E' -> epsilon

current left sentencial form:
                         ( ( num - num ) * ( num * num / num + ( num ) )
T' E' ) T' E'
current token stream:
                         ) ) end
output:

current left sentencial form:
                         ( ( num - num ) * ( num * num / num + ( num ) )
T' E' ) T' E'
current token stream:
                         ) end
output:
                         T' -> epsilon

current left sentencial form:
                         ( ( num - num ) * ( num * num / num + ( num ) )
E' ) T' E'
current token stream:
                         ) end
output:
                         E' -> epsilon
```

```
current left sentencial form:
                        ( ( num - num ) * ( num * num / num +
( num ) ) ) T' E'
current token stream:
                        ) end
output:

current left sentencial form:
                        ( ( num - num ) * ( num * num / num +
( num ) ) ) T' E'
current token stream:
                        end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( num - num ) * ( num * num / num +
( num ) ) ) E'
current token stream:
                        end
output:
                        E' -> epsilon

current left sentencial form:
                        ( ( num - num ) * ( num * num / num +
( num ) ) )
current token stream:
                        end
output:
```

```
((2*(5 - 9)/8)*(9/(1/(9-9.36e5))))
current left sentencial form:
                        E
current token stream:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) ) ) ) end
output:
                        E -> T E'

current left sentencial form:
                        T E'
current token stream:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) ) ) ) end
output:
                        T -> F T'

current left sentencial form:
                        F T' E'
current token stream:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) ) ) ) end
output:
                        F -> ( E )
```

```
current left sentencial form:
                        ( E ) T' E'
current token stream:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) ) ) ) end
output:

current left sentencial form:
                        ( E ) T' E'
current token stream:
                        ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) ) ) ) end
output:
                        E -> T E'

current left sentencial form:
                        ( T E' ) T' E'
current token stream:
                        ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) ) ) ) end
output:
                        T -> F T'

current left sentencial form:
                        ( F T' E' ) T' E'
current token stream:
                        ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) ) ) ) end
output:
                        F -> ( E )

current left sentencial form:
                        ( ( E ) T' E' ) T' E'
current token stream:
                        ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) ) ) ) end
output:

current left sentencial form:
                        ( ( E ) T' E' ) T' E'
current token stream:
                        num * ( num - num ) / num ) * ( num / ( num
/ ( num - num ) ) ) ) end
output:
                        E -> T E'

current left sentencial form:
                        ( ( T E' ) T' E' ) T' E'
current token stream:
                        num * ( num - num ) / num ) * ( num / ( num
/ ( num - num ) ) ) ) end
output:
                        T -> F T'

current left sentencial form:
                        ( ( F T' E' ) T' E' ) T' E'
```

```
current token stream:
                        num * ( num - num ) / num ) * ( num / ( num
/ ( num - num ) ) ) ) end
output:
                        F -> num

current left sentencial form:
                        ( ( num T' E' ) T' E' ) T' E'
current token stream:
                        num * ( num - num ) / num ) * ( num / ( num
/ ( num - num ) ) ) ) end
output:

current left sentencial form:
                        ( ( num T' E' ) T' E' ) T' E'
current token stream:
                        * ( num - num ) / num ) * ( num / ( num /
( num - num ) ) ) ) end
output:
                        T' -> * F T'

current left sentencial form:
                        ( ( num * F T' E' ) T' E' ) T' E'
current token stream:
                        * ( num - num ) / num ) * ( num / ( num /
( num - num ) ) ) ) end
output:

current left sentencial form:
                        ( ( num * F T' E' ) T' E' ) T' E'
current token stream:
                        ( num - num ) / num ) * ( num / ( num /
( num - num ) ) ) ) end
output:
                        F -> ( E )

current left sentencial form:
                        ( ( num * ( E ) T' E' ) T' E' ) T' E'
current token stream:
                        ( num - num ) / num ) * ( num / ( num /
( num - num ) ) ) ) end
output:

current left sentencial form:
                        ( ( num * ( E ) T' E' ) T' E' ) T' E'
current token stream:
                        num - num ) / num ) * ( num / ( num / ( num
- num ) ) ) ) end
output:
                        E -> T E'

current left sentencial form:
                        ( ( num * ( T E' ) T' E' ) T' E' ) T' E'
current token stream:
                        num - num ) / num ) * ( num / ( num / ( num
- num ) ) ) ) end
output:
```

```
                        T -> F T'

current left sentencial form:
                        ( ( num * ( F T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        num - num ) / num ) * ( num / ( num / ( num
- num ) ) ) ) end
output:
                        F -> num

current left sentencial form:
                        ( ( num * ( num T' E' ) T' E' ) T' E' ) T'
E'
current token stream:
                        num - num ) / num ) * ( num / ( num / ( num
- num ) ) ) ) end
output:

current left sentencial form:
                        ( ( num * ( num T' E' ) T' E' ) T' E' ) T'
E'
current token stream:
                        - num ) / num ) * ( num / ( num / ( num -
num ) ) ) ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( num * ( num E' ) T' E' ) T' E' ) T' E'
current token stream:
                        - num ) / num ) * ( num / ( num / ( num -
num ) ) ) ) end
output:
                        E' -> - T E'

current left sentencial form:
                        ( ( num * ( num - T E' ) T' E' ) T' E' ) T'
E'
current token stream:
                        - num ) / num ) * ( num / ( num / ( num -
num ) ) ) ) end
output:

current left sentencial form:
                        ( ( num * ( num - T E' ) T' E' ) T' E' ) T'
E'
current token stream:
                        num ) / num ) * ( num / ( num / ( num -
num ) ) ) ) end
output:
                        T -> F T'

current left sentencial form:
                        ( ( num * ( num - F T' E' ) T' E' ) T' E' )
T' E'
current token stream:
```

```
                        num ) / num ) * ( num / ( num / ( num -
num ) ) ) ) end
output:
                        F -> num

current left sentencial form:
                        ( ( num * ( num - num T' E' ) T' E' ) T'
E' ) T' E'
current token stream:
                        num ) / num ) * ( num / ( num / ( num -
num ) ) ) ) end
output:

current left sentencial form:
                        ( ( num * ( num - num T' E' ) T' E' ) T'
E' ) T' E'
current token stream:
                        ) / num ) * ( num / ( num / ( num -
num ) ) ) ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( num * ( num - num E' ) T' E' ) T' E' )
T' E'
current token stream:
                        ) / num ) * ( num / ( num / ( num -
num ) ) ) ) end
output:
                        E' -> epsilon

current left sentencial form:
                        ( ( num * ( num - num ) T' E' ) T' E' ) T'
E'
current token stream:
                        ) / num ) * ( num / ( num / ( num -
num ) ) ) ) end
output:

current left sentencial form:
                        ( ( num * ( num - num ) T' E' ) T' E' ) T'
E'
current token stream:
                        / num ) * ( num / ( num / ( num -
num ) ) ) ) end
output:
                        T' -> / F T'

current left sentencial form:
                        ( ( num * ( num - num ) / F T' E' ) T' E' )
T' E'
current token stream:
                        / num ) * ( num / ( num / ( num -
num ) ) ) ) end
output:

current left sentencial form:
```

```
                        ( ( num * ( num - num ) / F T' E' ) T' E' )
T' E'
current token stream:
                        num ) * ( num / ( num / ( num - num ) ) ) )
end
output:
                        F -> num

current left sentencial form:
                        ( ( num * ( num - num ) / num T' E' ) T'
E' ) T' E'
current token stream:
                        num ) * ( num / ( num / ( num - num ) ) ) )
end
output:

current left sentencial form:
                        ( ( num * ( num - num ) / num T' E' ) T'
E' ) T' E'
current token stream:
                        ) * ( num / ( num / ( num - num ) ) ) ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( num * ( num - num ) / num E' ) T' E' )
T' E'
current token stream:
                        ) * ( num / ( num / ( num - num ) ) ) ) end
output:
                        E' -> epsilon

current left sentencial form:
                        ( ( num * ( num - num ) / num ) T' E' ) T'
E'
current token stream:
                        ) * ( num / ( num / ( num - num ) ) ) ) end
output:

current left sentencial form:
                        ( ( num * ( num - num ) / num ) T' E' ) T'
E'
current token stream:
                        * ( num / ( num / ( num - num ) ) ) ) end
output:
                        T' -> * F T'

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * F T' E' )
T' E'
current token stream:
                        * ( num / ( num / ( num - num ) ) ) ) end
output:

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * F T' E' )
T' E'
```

```
current token stream:
                        ( num / ( num / ( num - num ) ) ) ) end
output:
                        F -> ( E )

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( E ) T'
E' ) T' E'
current token stream:
                        ( num / ( num / ( num - num ) ) ) ) end
output:

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( E ) T'
E' ) T' E'
current token stream:
                        num / ( num / ( num - num ) ) ) ) end
output:
                        E -> T E'

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( T E' )
T' E' ) T' E'
current token stream:
                        num / ( num / ( num - num ) ) ) ) end
output:
                        T -> F T'

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( F T'
E' ) T' E' ) T' E'
current token stream:
                        num / ( num / ( num - num ) ) ) ) end
output:
                        F -> num

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num T'
E' ) T' E' ) T' E'
current token stream:
                        num / ( num / ( num - num ) ) ) ) end
output:

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num T'
E' ) T' E' ) T' E'
current token stream:
                        / ( num / ( num - num ) ) ) ) end
output:
                        T' -> / F T'

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num / F
T' E' ) T' E' ) T' E'
current token stream:
                        / ( num / ( num - num ) ) ) ) end
output:
```

```
current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num / F
T' E' ) T' E' ) T' E'
current token stream:
                        ( num / ( num - num ) ) ) ) end
output:
                        F -> ( E )

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( E ) T' E' ) T' E' ) T' E'
current token stream:
                        ( num / ( num - num ) ) ) ) end
output:

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( E ) T' E' ) T' E' ) T' E'
current token stream:
                        num / ( num - num ) ) ) ) end
output:
                        E -> T E'

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( T E' ) T' E' ) T' E' ) T' E'
current token stream:
                        num / ( num - num ) ) ) ) end
output:
                        T -> F T'

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( F T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        num / ( num - num ) ) ) ) end
output:
                        F -> num

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        num / ( num - num ) ) ) ) end
output:

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        / ( num - num ) ) ) ) end
output:
                        T' -> / F T'

current left sentencial form:
```

```
                        ( ( num * ( num - num ) / num ) * ( num /
( num / F T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        / ( num - num ) ) ) ) end
output:

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / F T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        ( num - num ) ) ) ) end
output:
                        F -> ( E )

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( E ) T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        ( num - num ) ) ) ) end
output:

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( E ) T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        num - num ) ) ) ) end
output:
                        E -> T E'

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( T E' ) T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        num - num ) ) ) ) end
output:
                        T -> F T'

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( F T' E' ) T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        num - num ) ) ) ) end
output:
                        F -> num

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num T' E' ) T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        num - num ) ) ) ) end
output:

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num T' E' ) T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        - num ) ) ) ) end
```

```
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num E' ) T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        - num ) ) ) ) end
output:
                        E' -> - T E'

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - T E' ) T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        - num ) ) ) ) end
output:

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - T E' ) T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        num ) ) ) ) end
output:
                        T -> F T'

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - F T' E' ) T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        num ) ) ) ) end
output:
                        F -> num

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - num T' E' ) T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        num ) ) ) ) end
output:

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - num T' E' ) T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        ) ) ) ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - num E' ) T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        ) ) ) ) end
output:
                        E' -> epsilon
```

```
current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        ) ) ) ) end
output:

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) T' E' ) T' E' ) T' E' ) T' E'
current token stream:
                        ) ) ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) E' ) T' E' ) T' E' ) T' E'
current token stream:
                        ) ) ) end
output:
                        E' -> epsilon

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) ) T' E' ) T' E' ) T' E'
current token stream:
                        ) ) ) end
output:

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) ) T' E' ) T' E' ) T' E'
current token stream:
                        ) ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) ) E' ) T' E' ) T' E'
current token stream:
                        ) ) end
output:
                        E' -> epsilon

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) ) ) T' E' ) T' E'
current token stream:
                        ) ) end
output:

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) ) ) T' E' ) T' E'
current token stream:
```

```
                        ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) ) ) E' ) T' E'
current token stream:
                        ) end
output:
                        E' -> epsilon

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) ) ) ) T' E'
current token stream:
                        ) end
output:

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) ) ) ) T' E'
current token stream:
                        end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) ) ) ) E'
current token stream:
                        end
output:
                        E' -> epsilon

current left sentencial form:
                        ( ( num * ( num - num ) / num ) * ( num /
( num / ( num - num ) ) ) )
current token stream:
                        end
output:
```

6.2. 错误表达式

```
((1)
```

```
current left sentencial form:
                        E
current token stream:
                        ( ( num ) end
output:
                        E -> T E'

current left sentencial form:
```

```
                        T E'
current token stream:
                        ( ( num ) end
output:
                        T -> F T'

current left sentencial form:
                        F T' E'
current token stream:
                        ( ( num ) end
output:
                        F -> ( E )

current left sentencial form:
                        ( E ) T' E'
current token stream:
                        ( ( num ) end
output:

current left sentencial form:
                        ( E ) T' E'
current token stream:
                        ( num ) end
output:
                        E -> T E'

current left sentencial form:
                        ( T E' ) T' E'
current token stream:
                        ( num ) end
output:
                        T -> F T'

current left sentencial form:
                        ( F T' E' ) T' E'
current token stream:
                        ( num ) end
output:
                        F -> ( E )

current left sentencial form:
                        ( ( E ) T' E' ) T' E'
current token stream:
                        ( num ) end
output:

current left sentencial form:
                        ( ( E ) T' E' ) T' E'
current token stream:
                        num ) end
output:
                        E -> T E'

current left sentencial form:
                        ( ( T E' ) T' E' ) T' E'
current token stream:
                        num ) end
```

```
output:
                        T -> F T'

current left sentencial form:
                        ( ( F T' E' ) T' E' ) T' E'
current token stream:
                        num ) end
output:
                        F -> num

current left sentencial form:
                        ( ( num T' E' ) T' E' ) T' E'
current token stream:
                        num ) end
output:

current left sentencial form:
                        ( ( num T' E' ) T' E' ) T' E'
current token stream:
                        ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( num E' ) T' E' ) T' E'
current token stream:
                        ) end
output:
                        E' -> epsilon

current left sentencial form:
                        ( ( num ) T' E' ) T' E'
current token stream:
                        ) end
output:

current left sentencial form:
                        ( ( num ) T' E' ) T' E'
current token stream:
                        end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( num ) E' ) T' E'
current token stream:
                        end
output:
                        E' -> epsilon

current left sentencial form:
                        ( ( num ) ) T' E'
current token stream:
                        end
output:
                        error: ) expected
```

```
current left sentencial form:
                        ( ( num ) ) T' E'
current token stream:
                        end
output:
                        T' -> epsilon

current left sentencial form:
                        ( ( num ) ) E'
current token stream:
                        end
output:
                        E' -> epsilon

current left sentencial form:
                        ( ( num ) )
current token stream:
                        end
output:
```

(1))

```
current left sentencial form:
                        E
current token stream:
                        ( num ) ) end
output:
                        E -> T E'

current left sentencial form:
                        T E'
current token stream:
                        ( num ) ) end
output:
                        T -> F T'

current left sentencial form:
                        F T' E'
current token stream:
                        ( num ) ) end
output:
                        F -> ( E )

current left sentencial form:
                        ( E ) T' E'
current token stream:
                        ( num ) ) end
output:

current left sentencial form:
                        ( E ) T' E'
current token stream:
                        num ) ) end
output:
                        E -> T E'
```

```
current left sentencial form:
                        ( T E' ) T' E'
current token stream:
                        num ) ) end
output:
                        T -> F T'

current left sentencial form:
                        ( F T' E' ) T' E'
current token stream:
                        num ) ) end
output:
                        F -> num

current left sentencial form:
                        ( num T' E' ) T' E'
current token stream:
                        num ) ) end
output:

current left sentencial form:
                        ( num T' E' ) T' E'
current token stream:
                        ) ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( num E' ) T' E'
current token stream:
                        ) ) end
output:
                        E' -> epsilon

current left sentencial form:
                        ( num ) T' E'
current token stream:
                        ) ) end
output:

current left sentencial form:
                        ( num ) T' E'
current token stream:
                        ) end
output:
                        T' -> epsilon

current left sentencial form:
                        ( num ) E'
current token stream:
                        ) end
output:
                        E' -> epsilon

current left sentencial form:
                        ( num )
current token stream:
```

```
                        ) end
output:
                        error: end expected
```

```
*1
current left sentencial form:
                        E
current token stream:
                        * num end
output:
                        error: * unexpected

current left sentencial form:
                        E
current token stream:
                        num end
output:
                        E -> T E'

current left sentencial form:
                        T E'
current token stream:
                        num end
output:
                        T -> F T'

current left sentencial form:
                        F T' E'
current token stream:
                        num end
output:
                        F -> num

current left sentencial form:
                        num T' E'
current token stream:
                        num end
output:

current left sentencial form:
                        num T' E'
current token stream:
                        end
output:
                        T' -> epsilon

current left sentencial form:
                        num E'
current token stream:
                        end
output:
                        E' -> epsilon

current left sentencial form:
                        num
current token stream:
```

```
                                end
output:
```

```
*1*+1
```

```
current left sentencial form:
                                E
current token stream:
                                * num * + num end
output:
                                error: * unexpected

current left sentencial form:
                                E
current token stream:
                                num * + num end
output:
                                E -> T E'

current left sentencial form:
                                T E'
current token stream:
                                num * + num end
output:
                                T -> F T'

current left sentencial form:
                                F T' E'
current token stream:
                                num * + num end
output:
                                F -> num

current left sentencial form:
                                num T' E'
current token stream:
                                num * + num end
output:

current left sentencial form:
                                num T' E'
current token stream:
                                * + num end
output:
                                T' -> * F T'

current left sentencial form:
                                num * F T' E'
current token stream:
                                * + num end
output:

current left sentencial form:
                                num * F T' E'
current token stream:
                                + num end
```

```
output:
                        error: + unexpected

current left sentencial form:
                        num * T' E'
current token stream:
                        + num end
output:
                        T' -> epsilon

current left sentencial form:
                        num * E'
current token stream:
                        + num end
output:
                        E' -> + T E'

current left sentencial form:
                        num * + T E'
current token stream:
                        + num end
output:

current left sentencial form:
                        num * + T E'
current token stream:
                        num end
output:
                        T -> F T'

current left sentencial form:
                        num * + F T' E'
current token stream:
                        num end
output:
                        F -> num

current left sentencial form:
                        num * + num T' E'
current token stream:
                        num end
output:

current left sentencial form:
                        num * + num T' E'
current token stream:
                        end
output:
                        T' -> epsilon

current left sentencial form:
                        num * + num E'
current token stream:
                        end
output:
                        E' -> epsilon
```

```
current left sentencial form:
                        num * + num
current token stream:
                        end
output:
```

```
1/(1+
```

```
current left sentencial form:
                        E
current token stream:
                        num / ( num + end
output:
                        E -> T E'

current left sentencial form:
                        T E'
current token stream:
                        num / ( num + end
output:
                        T -> F T'

current left sentencial form:
                        F T' E'
current token stream:
                        num / ( num + end
output:
                        F -> num

current left sentencial form:
                        num T' E'
current token stream:
                        num / ( num + end
output:

current left sentencial form:
                        num T' E'
current token stream:
                        / ( num + end
output:
                        T' -> / F T'

current left sentencial form:
                        num / F T' E'
current token stream:
                        / ( num + end
output:

current left sentencial form:
                        num / F T' E'
current token stream:
                        ( num + end
output:
                        F -> ( E )

current left sentencial form:
```

```
                        num / ( E ) T' E'
current token stream:
                        ( num + end
output:

current left sentencial form:
                        num / ( E ) T' E'
current token stream:
                        num + end
output:
                        E -> T E'

current left sentencial form:
                        num / ( T E' ) T' E'
current token stream:
                        num + end
output:
                        T -> F T'

current left sentencial form:
                        num / ( F T' E' ) T' E'
current token stream:
                        num + end
output:
                        F -> num

current left sentencial form:
                        num / ( num T' E' ) T' E'
current token stream:
                        num + end
output:

current left sentencial form:
                        num / ( num T' E' ) T' E'
current token stream:
                        + end
output:
                        T' -> epsilon

current left sentencial form:
                        num / ( num E' ) T' E'
current token stream:
                        + end
output:
                        E' -> + T E'

current left sentencial form:
                        num / ( num + T E' ) T' E'
current token stream:
                        + end
output:

current left sentencial form:
                        num / ( num + T E' ) T' E'
current token stream:
                        end
output:
```

```
                                error: T expected

current left sentencial form:
                                num / ( num + E' ) T' E'
current token stream:
                                end
output:
                                E' -> epsilon

current left sentencial form:
                                num / ( num + ) T' E'
current token stream:
                                end
output:
                                error: ) expected

current left sentencial form:
                                num / ( num + ) T' E'
current token stream:
                                end
output:
                                T' -> epsilon

current left sentencial form:
                                num / ( num + ) E'
current token stream:
                                end
output:
                                E' -> epsilon

current left sentencial form:
                                num / ( num + )
current token stream:
                                end
output:
```

```
1+2*-8
current left sentencial form:
                                E
current token stream:
                                num + num * - num end
output:
                                E -> T E'

current left sentencial form:
                                T E'
current token stream:
                                num + num * - num end
output:
                                T -> F T'

current left sentencial form:
                                F T' E'
current token stream:
                                num + num * - num end
```

```
output:
                        F -> num

current left sentencial form:
                        num T' E'
current token stream:
                        num + num * - num end
output:

current left sentencial form:
                        num T' E'
current token stream:
                        + num * - num end
output:
                        T' -> epsilon

current left sentencial form:
                        num E'
current token stream:
                        + num * - num end
output:
                        E' -> + T E'

current left sentencial form:
                        num + T E'
current token stream:
                        + num * - num end
output:

current left sentencial form:
                        num + T E'
current token stream:
                        num * - num end
output:
                        T -> F T'

current left sentencial form:
                        num + F T' E'
current token stream:
                        num * - num end
output:
                        F -> num

current left sentencial form:
                        num + num T' E'
current token stream:
                        num * - num end
output:

current left sentencial form:
                        num + num T' E'
current token stream:
                        * - num end
output:
                        T' -> * F T'

current left sentencial form:
```

```
                        num + num * F T' E'
current token stream:
                        * - num end
output:

current left sentencial form:
                        num + num * F T' E'
current token stream:
                        - num end
output:
                        error: F expected

current left sentencial form:
                        num + num * T' E'
current token stream:
                        - num end
output:
                        T' -> epsilon

current left sentencial form:
                        num + num * E'
current token stream:
                        - num end
output:
                        E' -> - T E'

current left sentencial form:
                        num + num * - T E'
current token stream:
                        - num end
output:

current left sentencial form:
                        num + num * - T E'
current token stream:
                        num end
output:
                        T -> F T'

current left sentencial form:
                        num + num * - F T' E'
current token stream:
                        num end
output:
                        F -> num

current left sentencial form:
                        num + num * - num T' E'
current token stream:
                        num end
output:

current left sentencial form:
                        num + num * - num T' E'
current token stream:
                        end
output:
```

```
                        T' -> epsilon

current left sentencial form:
                        num + num * - num E'
current token stream:
                        end
output:
                        E' -> epsilon

current left sentencial form:
                        num + num * - num
current token stream:
                        end
output:
```

7. 分析总结

从测试结果可看出，对于正确的表达式，语法分析输出结果也是正确的，而对于错误的表达式，本程序的错误处理措施也能在一定程度上对其进行恢复，当然也具有一定的局限性。

在编译流程中，语法分析是词法分析的后续环节，其分析的基础是词法分析得到的 token 流，故在本次试验中也利用了上次词法分析实验的成果，在其基础上加以拓展，延续了其设计思路。

编写语法分析程序的过程中遇到了更大的挑战，如如何表示文法产生式，如何高效且正确地提取左公因子，如何解决求 FOLLOW 集过程可能出现的死循环问题，这也相当于是对数据结构、算法及编程能力的一个锻炼。