编译原理程序设计报告

程序设计 3：LR(1) 算术表达式语法分析器

2021211110 杜抒泽

2023 年 12 月 30 日

目录

1 实验要求

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

$$
\begin{aligned}
&E \to E + T \mid E - T \mid T \\
&T \to T * F \mid T/F \mid F \\
&F \to (E) \mid \text{num}
\end{aligned}
$$

实验要求：在对输人的算术表达式进行分析的过程中，依次输出所采用的产生式。

方法 3：编写语法分析程序实现自底向上的分析，要求如下。

（1）构造识别该文法所有活前缀的 DFA。

（2）构造该文法的 LR 分析表。

（3）编程实现算法 4.3，构造 LR 分析程序。

方法 4：利用 YACC 自动生成语法分析程序，调用 LEX 自动生成的词法分析程序。

2 程序设计说明

2.1 实现的基本功能

本次程序设计选取方法 3。使用 C++ 完成算术表达式自底向上语法分析器的编写。程序完成了：

1. 对给定文法计算 FIRST 集，构造 LR(1) 项目集规范族及识别其所有活前缀的 DFA。
2. 构造该文法的的 LR(1) 分析表。
3. 对给定的表达式进行词法分析。
4. 根据词法分析的结果，基于构造的 LR(1) 分析表对给定的表达式进行语法分析，输出每一步的状态栈、符号栈和输入串状态，以及进行的操作。

2.2 开发、构建与运行

本次程序设计的项目名为 ExParserR，采用的 C++ 标准为 C++23（GNU++23），使用 CMake 进行构建，在 ARM aarch 64 架构的 macOS Ventura 13.0 上编写、调试与测试。

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

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

代码总行数为 11 sloc，全部代码随附在作业中提交。若发现有任何问题请随时联系我。本地采用的编译工具链如下表：

工具	版本
CMake	3.27.8 / 3.28.1
`g++-13`	13.2.0
`lldb`	16.0.6
`ld`	ld64-820.1

表 1　工具链信息

2.3 程序设计介绍与功能细节

ExParserR 是规范 C++ 项目。源码除 `main` 外还包含 `SimpleLexer`、`Parser`、`Util` 三个子模块，分别是简易词法分析器、LR(1) 语法分析器与常用工具代码库，它们全部位于 `namespace epr` 内。

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

2.3.1 词法 Token 与语法 Symbol

ExParserR 提供的简易词法分析器模块 `SimpleLexer` 基于前两次程序设计的词法分析器修改而来，主旨是通过 `Lexer::lex_effective` 接口为语法分析器提供服务。

它能够识别的词法元素 token 种类及对应的语法元素 symbol 如下表：

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

表 2　Token 种类与 Symbol 种类对应

2.3.2 文法与 LR(1) 相关类型定义

如何选择合适的数据结构表达文法与 LR(1) 分析表的构造过程中将会用到的类型是本次程序设计的关键。为了为后续程序设计提供便利，parser 模块提供了一系列类型的定义或别名（省略成员函数）：

2.3.2.1 文法元素与文法

```
struct Symbol {
  std::string name{};

  enum Type { Terminator, NonTerminator } type;
};

using FirstSet = std::map<Symbol, std::set<Symbol>>;

struct Grammar {
  inline static Symbol END_SYMBOL{"$", Symbol::Type::Terminator};

  FirstSet first_set{};

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

  std::map<std::pair<const Symbol, std::vector<Symbol>>, usize>
      production_index{};

  std::vector<std::pair<Symbol, std::vector<Symbol>>> production_list{};

  Symbol start_symbol;
};
```

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

1. `struct Symbol` 用于表示文法中的符号（终结符和非终结符）。
   - `std::string v` 用于存储符号的实际值。
   - `enum Type` 指示符号是终结符（`Terminator`）或非终结符（`NonTerminator`）。
2. `using FirstSet = std::map<Symbol, std::set<Symbol>>`
   - 它们用于表示 FIRST 集合和 FOLLOW 集合。实际上 FIRST 集合还可以扩充到 `std::map<std::vector<Symbol>, std::set<Symbol>>`，但是本次程序设计并未直接存储它们，而是在需要时动态计算。`struct Grammar` 提供了这个接口。
3. `struct Grammar` 用于表示文法。

- `Symbol start_symbol` 用于存储文法的开始符号。
- 成员 `std::map<std::pair<const Symbol, std::vector<Symbol>>, usize> production_index` 和 `std::vector<std::pair<Symbol, std::vector<Symbol>>> production_list` 用于存储文法的生成式。其中 `production_index` 用于快速查找生成式，`production_list` 用于按顺序遍历生成式。
- 成员 `std::map<Symbol, std::set<std::vector<Symbol>>> productions` 用于存储所有生成式。
- 成员 `FirstSet first_set` 用于缓存文法每个符号的 FIRST 集。
- 静态成员 `Symbol END_SYMBOL` 表示 LR(1) 分析时的结束符号。
- `struct Grammar` 还包含一系列的成员函数，用于对文法进行操作，如进行文法增广、构造 FIRST 集、计算非终结符序列的 FIRST 函数等。

2.3.2.2 LR(1) 项目与 LR(1) 项目集

```
struct Item {
  Symbol lookahead;
  std::vector<Symbol> rhs{};
  Symbol lhs;
  usize dot_pos{}; // range: [0, rhs.size()]
};

struct ItemSet {
  std::set<Item> items{};
};
```

`struct Item` 更侧重存储数据，提供较少的成员函数。反之，`struct ItemSet` 的字段简单，但提供了大量的成员函数，用于对项目集进行操作，包含了求项目集的闭包 closure 和转移 go 函数的算法等。

1. `struct Item` 用于表示 LR(1) 项目。
   - `Symbol lhs` 用于表示项目的左侧符号。
   - `std::vector<Symbol> rhs` 用于表示项目的右侧符号串。
   - `usize dot_pos` 用于表示项目的右侧圆点位置。
   - `Symbol lookahead` 用于表示项目的向前看符号。
   - 例如，对于 LR(1) 项目 $A \to \alpha \cdot B\beta, a$，其可以表示为（伪代码）：

     ```
     Item {
       lhs: "A",
       rhs: ["alpha", "B", "beta"],
     ```

```
    dot_pos: 1,
    lookahead: "a",
  }
```

2. `struct ItemSet` 用于表示 LR(1) 项目集。
   - `std::set<Item> items` 用于存储项目集中的项目。
   - 注意：两个除向前看符号外完全相同的项目是分开存储的。这和一般书写时的习惯不同，但是这样做可以方便程序设计。
   - `struct ItemSet` 包含一系列的成员函数，用于对项目集进行操作，如求项目集的闭包、goto 函数等。

2.3.2.3 LR(1) 项目集规范族与 DFA

```
struct Dfa {
  using State = ItemSet;

  std::vector<State> states{};
  std::vector<std::map<Symbol, usize>> transitions{};
};
```

- `struct Dfa` 用于表示 LR(1) 项目集规范族的 DFA。
  - DFA 将项目集规范族中的每个项目集看作一个状态，项目集规范族中的每个项目集之间的转移对应 DFA 中的状态转移。
  - `std::vector<State> states` 用于存储 DFA 的状态集。
  - `std::vector<std::map<Symbol, usize>> transitions` 用于存储 DFA 的状态转移，定义域和值域都用状态在 `states` 中的下标表示。

2.3.3 从字符串中获取文法

`Grammar::from_str` 函数用于从字符串中获取文法。用于表示输入文法的文法 G 定义为 $G = (N, T, P, S)$，其中 $S =$ `<grammar>`，N 和 T 包含下述用 BNF 范式描述的生成式集合 P 中出现的所有元素，非终结符采用 `<nonterminator-name>` 格式命名，终结符采用 `<TerminatorName>` 格式命名。

```
<grammar> ::= <starting-symbol> [[LF]] <lines>

<starting-symbol> ::= <Nonterminator>

<lines> ::= <line> <lines>
          | <line>
```

```
<line> ::= <lhs> " → " <rhs-set> [[LF]]

<rhs-set> ::= <rhs> " | " <rhs-set>
            | <rhs>

<lhs> ::= <Nonterminator>

<rhs> ::= <symbol> " " <rhs>
        | <symbol>

<symbol> ::= <Nonterminator>
           | <Terminator>
```

其中 [[LF]] 表示换行符。用语言描述即是，文法包含起始指定的开始符号和之后的每行一组产生式，不能包含多余空行。每个产生式的左侧是一个非终结符，右侧是一组符号，每组符号内部的符号之间用空格分隔，各组符号之间用“ | ”分隔。为了避免歧义，不能增加或减少任何空格。NonTerminator 和 Terminator 可以包含任意字符，但不能重复；用 “~” 表示空串，规定空串属于 Terminator。

例如，题给文法在这个文法下表示为：

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

2.3.4 算法

2.3.4.1 构造增广文法

为了使 LR(1) 分析表只有一个接受状态，我们需要对文法进行增广。增广文法的定义为 $G' = (N', T', P', S')$，其中 S' 是新的开始符号，$N' = N \cup \{S'\}$，$T' = T$，$P' = P \cup \{S' \to S\}$。函数 `Grammar::self_augment` 用于构造增广文法：

```
void Grammar::self_augment() {
  auto new_start_symbol =
      Symbol(start_symbol.name + '\'', Symbol::NonTerminator);
  while (productions.contains(new_start_symbol))
    new_start_symbol.name.push_back('\'');
  push_production(new_start_symbol, {start_symbol});
  start_symbol = new_start_symbol;
}
```

算法在实现时选择直接在原有的开始符号后面加上 '，如果已经存在，则在后面继续加 '，直到不存在为止。这样做可以保证增广文法的开始符号不会和原有的文法中的符号重复。

2.3.4.2 构造 FIRST 集

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

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

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

函数 `Grammar::build_first_set` 和 `Grammar::first` 用于构造 FIRST 集。前者直接计算并缓存文法中所有符号的 FIRST 集，后者用于动态计算给定符号串的 FIRST 集。具体的步骤如下：

1. 对于每个终结符，将其加入到自身的 FIRST 集中。
2. 遍历文法的每个生成式，每个生成式由一个左侧的非终结符（`lhs`）和一个右侧生成式集合（`rhs_set`）组成。
3. 对于每个生成式 $A \to \alpha$，如果 $\alpha = \varepsilon$，则将 ε 加入 FIRST(A)；否则，如果 $\alpha = a\beta$ 且 $a \in T$，则将 a 加入 FIRST(A)；否则，如果 $\alpha = B\beta$ 且 $B \in N$，则将 FIRST(B) $- \{\varepsilon\}$ 加入 FIRST(A)。对于 $A \to B_1 B_2 ... B_n$（$B_i \in N, 1 < i < n$），若 $\varepsilon \in \bigcap_{i=1}^{k-1} \text{FIRST}(B_i)$，则将 FIRST($B_k$) $- \{\varepsilon\}$ 加入 FIRST(A)；若 $\varepsilon \in \bigcap_{i=1}^{n} \text{FIRST}(B_i)$，则将 ε 加入 FIRST(A)。

实现如下：

```
void Grammar::build_first_set() {
  const auto [terminators, terminators_has_empty] = get_terminators();

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

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

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

```
          continue;
        }

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

    if (!changed)
      break;
  }
  first_set[epr::Grammar::END_SYMBOL].emplace(epr::Grammar::END_SYMBOL);
}
```

```
std::set<Symbol> Grammar::first(const std::vector<Symbol> &str) const {
  std::set<Symbol> ret{};
  for (const auto &symbol : str) {
    const auto &first_set_rhs = first_set.at(symbol);
    ret.insert(first_set_rhs.begin(), first_set_rhs.end());
    if (!first_set_rhs.contains(Symbol::empty_symbol()))
      break;
  }
  return ret;
}
```

2.3.4.3 LR(1) 项目集规范族的 closure 和 go 函数

LR(1) 项目集规范族的 closure 函数的定义如下：

设 I 是文法 G 的一个 LR(1) 项目集合，closure(I) 是从 I 出发,用下面的方法构造的项目集合：

（1）I 中的每一个项目都属于 closure(I)。

（2）若 $[A \to \alpha \cdot B\beta, a] \in \text{closure}(I)$，且 $B \to \eta$ 是 G 的一个产生式，则对任何终结符号 $b \in \text{FIRST}(\beta a)$，若 $[B \to \cdot\, \eta, b] \notin \text{closure}(I)$，则将 $[B \to \cdot\, \eta, b]$ 加入 closure(I)。

（3）重复规则（2），直到 closure(I) 不再增大为止。

实现如下：

```
void ItemSet::self_closure(const Grammar &grammar) {
  while (true) {
    const usize old_size = items.size();

    for (const auto &item : items) {
      if (item.dot_pos ⩾ item.rhs.size()) {
        continue;
      }

      auto get_production_set = [&](const Symbol &symbol
                                    ) → std::set<std::vector<Symbol>> {
        const auto it = grammar.productions.find(symbol);
        if (it == grammar.productions.end())
          return std::set<std::vector<Symbol>>{};
        return it→second;
      };

      const auto &next_symbol = item.rhs.at(item.dot_pos);
      const auto &production_set = get_production_set(next_symbol);
      for (const auto &first = grammar.first(item.back_slice());
           const Symbol &b : first) {
        for (const auto &production : production_set)
          push({next_symbol, production, 0, b});
      }
    }
    if (items.size() == old_size) // unchanged
      break;
  }
}
```

LR(1) 项目集规范族的 go 函数的定义如下：

若 I 是文法 G 的一个 LR(1) 有效项目集，X 是文法符号，定义

$$\mathrm{go}(I, X) = \mathrm{closure}(J)$$

其中，$J = \{[A \to \alpha X \cdot \beta, a] \mid [A \to \alpha \cdot X\beta, a] \in I\}$。

实现如下：

```
ItemSet ItemSet::go(const Grammar &grammar, const Symbol &symbol) const {
  ItemSet buf;
  for (const auto &item : items)
    if (item.dot_pos < item.rhs.size() && item.rhs.at(item.dot_pos) == symbol)
      buf.push(item.advance_dot());
  buf.self_closure(grammar);
  return buf;
}
```

2.3.4.4 构造 LR(1) 项目集规范族

LR(1) 项目集规范族的构造算法如下：

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

（2）对 C 中的每个项目集中的每个项目 $[A \to \alpha \cdot X\beta, a]$，$X \in (T \cup N)$，令 $I = \text{go}(C, X)$，若 $I \notin C$，则将 I 加入 C。

（3）重复上述步骤，直到 C 不再增大为止。

实现如下：

```
Dfa::Dfa(Grammar &grammar) {
  State start_state;
  const auto &start_production_set =
      grammar.productions.at(grammar.start_symbol);
  assert(start_production_set.size() == 1); // ensures that the grammar is
                                            // augmented
  const auto &start_production = *start_production_set.begin();
  start_state.push(
      {grammar.start_symbol, start_production, 0, Grammar::END_SYMBOL}
  );
  start_state.self_closure(grammar);
  states.push_back(start_state);
  transitions.resize(1);

  for (usize idx = 0; idx < states.size(); ++idx)
    for (const auto &symbol : states.at(idx).next_symbols()) {
      const auto &&next_state = states.at(idx).go(grammar, symbol);
      if (next_state.items.empty())
        continue;
      auto it = std::find(states.begin(), states.end(), next_state);
      if (it == states.end()) {
        states.push_back(next_state);
        transitions.resize(states.size());
        it = std::prev(states.end());
      }
      transitions.at(idx).insert({symbol, std::distance(states.begin(), it)});
    }
}
```

2.3.4.5 构造预测分析表

LL(1) 预测分析表实际上是一个（或两个）二维数组，第一维下标是 DFA 中的状态，第二维下标是终结符号 $a \in T$（action 表）或非终结符号 $A \in N$（goto 表）。对于 action 表，其值是一个状态（S_j），表示如果输入符号是 a，则应该进行移进到状态 j 的操作；或者一个生

成式 $A \to \alpha$，表示如果输入符号是 a，则应该使用生成式 $A \to \alpha$ 进行规约；对于 goto 表，其值是一个状态 j，表示在状态 i 时，如果输入符号是 A，则应该跳转到状态 j。

LL(1) 预测分析表的构造方式如同教材中的伪代码。具体步骤：

（1）构造文法 G 的拓广文法 G'。

（2）构造 G' 的 LR(1) 项目集规范族 $C = \{I_0, I_1, ..., I_n\}$。

（3）对于状态 i（代表项目集 I_i，分析动作如下：

- 若 $[A \to \alpha \cdot a\beta, b] \in I_i$，且 $\text{go}(I_i, a) = I_j$，则置 $\text{action}[i, a] = S_j$。
- 若 $[A \to \alpha \cdot, a] \in I_i$，且 $A \neq S'$，则置 $\text{action}[i, a] = R_j$，其中 j 是产生式 $A \to \alpha$ 的编号。
- 若 $[S' \to S \cdot, \$] \in I_i$，则置 $\text{action}[i, \$] = \text{acc}$。

（4）若对非终结符号 A，有 $\text{go}(I_i, A) = I_j$，则置 $\text{goto}[i, A] = j$。

（5）分析表中凡是不能用规则（3）和规则（4）填人信息的空白表项，均置上出错标志 error。

（6）分析程序的初态是包括 $[S' \to \cdot S, \$]$ 的有效项目集所对应的状态。

实现如下：

```
ParsingTable::ParsingTable(const Dfa &dfa, const Grammar &grammar) {
  usize col_idx = 0;
  for (const auto &symbol : grammar.get_terminators().first)
    terminals.insert({symbol, ++col_idx});
  terminals.insert({Grammar::END_SYMBOL, ++col_idx});
  for (const auto &symbol : grammar.get_nonterminators())
    non_terminals.insert({symbol, ++col_idx});

  usize row_idx = 0;
  table.resize(dfa.states.size());
  for (const auto &state : dfa.states) {
    table[row_idx].resize(col_idx + 1);

    // accept and reduce
    for (const auto &item : state.items)
      if (item.dot_pos == item.rhs.size()) {
        if (item.lhs == grammar.start_symbol)
          table[row_idx][terminals.at(Grammar::END_SYMBOL)] = Action{Accept{}};
        else
          table[row_idx][terminals.at(item.lookahead)] =
            Action{Reduce{grammar.production_index.at({item.lhs, item.rhs})}};
      }

    // shift and goto
    for (const auto &[symbol, next_state_idx] : dfa.transitions.at(row_idx))
      if (symbol.type == Symbol::Type::Terminator)
```

```
        table[row_idx][terminals.at(symbol)] = Action{Shift{next_state_idx}};
      else
       table[row_idx][non_terminals.at(symbol)] = Action{Goto{next_state_idx}};

    ++row_idx;
  }
}
```

2.3.4.6 语法分析与错误处理

在 Parser 模块中，我实现了 Parser 类，用于进行语法分析。语法分析器的功能是：

1. 对给定文法调用提供的算法接口构造 LR(1) 项目集规范族和分析表。
2. 调用 Lexer 提供的接口获取 token 流并转换为 symbol 流，并根据 LR(1) 分析表进行语法分析，判断输入的符号串是否符合规则。

根据 LR(1) 分析表对输入串 ω 进行语法分析，采取的算法与教材中伪代码一致：

1. 初始化，将初始状态 S_0 压入状态栈，将 $\omega\$$ 存入输入缓冲区中，并置输入指针指向 $\omega\$$ 的第一个符号。
2. 当栈非空时，重复以下步骤：
   - 令 S 为栈顶状态，a 为输入指针指向的符号。
   - 如果 $\text{action}[S,a] = S_j$，则将 S_j 压入状态栈，输入指针前进。
   - 如果 $\text{action}[S,a] = R_j$，文法 G 中的第 j 个产生式为 $A \to \beta$，则从栈中弹出 $|\beta|$ 个状态和符号。记此时新的栈顶状态为 S'，将 A 压入状态栈，将 $\text{goto}[S',A]$ 压入状态栈。
   - 如果 $\text{action}[S,a] = \text{acc}$，则分析成功，返回。
   - 如果 $\text{action}[S,a]$ 被标记为错误（此处实现采取置空），则遇到语法错误，进行错误处理恢复。

对于语法错误，没有实现复杂的错误处理与恢复策略，采用教材中提到的恢复策略：弹出栈顶的若干个状态（可能为 0 个），直到栈顶状态 S 和输入符号 a 能够匹配 $\text{action}[S,a]$，然后继续分析。

代码如下（不包含输出部分）：

```
void Parser::parse_expr(SymbolStream &&input) {
  input.emplace_back(Grammar::END_SYMBOL);

  std::vector<usize> stack{0};
  std::vector<Symbol> symbols{};
  bool has_error = false;
```

```
for (usize input_idx = 0;;) {
  const auto &cur_state = stack.back();
  const auto &cur_symbol = input.at(input_idx);
  const auto &action = table.get_action(cur_state, cur_symbol);

  if (std::holds_alternative<Accept>(action))
    break;
  bool sync = false;

  std::visit(
      overloaded{
          [&](const Shift &shift) {
            stack.push_back(shift.state);
            symbols.push_back(cur_symbol);
            ++input_idx;
          },

          [&](const Reduce &reduce) {
           const auto &[lhs, rhs] = grammar_.production_list.at(reduce.rule);

            for (usize _ = 0; _ < rhs.size(); ++_) {
              stack.pop_back();
              symbols.pop_back();
            }

            const auto &cur_top_state = stack.back();
            const auto &next_state = table.get_action(cur_top_state, lhs);
            stack.push_back(std::get<Goto>(next_state).state);
            symbols.push_back(lhs);
          },

          [&](const Error &) {
            has_error = true;
            while (true) {
              if (stack.empty() || symbols.empty())
                break;
              if (!std::holds_alternative<Error>(
                      table.get_action(stack.back(), cur_symbol)
                  ))
                break;
              stack.pop_back();
              symbols.pop_back();
              sync = true;
            }
          },
```

```
            [&](const auto &) {
              std::unreachable();
            },
        },
        action
    );

    if (std::holds_alternative<Error>(action) && !sync)
      break;
  }
}
```

3 测试、运行结果与分析说明

3.1 对于给定文法的变换暨 LR(1) 分析表的生成

如前文给出，题给文法表示为

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

我们对其逐个进行如下操作，得到其输出：

构造增广文法，得到的文法；对于每个终结符和非终结符，构造 FIRST 集。得到的增广文法和 FIRST 集如图：

```
═══ Augmented Grammar ═══
StartSymbol: E'
Terminators: {(, ), *, +, -, /, n}
NonTerminators: {E, E', F, T}
Productions: {
  (0) E → E + T
  (1) E → E - T
  (2) E → T
  (3) E' → E
  (4) F → ( E )
  (5) F → n
  (6) T → F
  (7) T → T * F
  (8) T → T / F
}
```

图 1　增广文法

```
═══ FIRST Set ═══
FIRST( $ ) = { $ }
FIRST( ( ) = { ( }
FIRST( ) ) = { ) }
FIRST( * ) = { * }
FIRST( + ) = { + }
FIRST( - ) = { - }
FIRST( / ) = { / }
FIRST( E ) = { (, n }
FIRST( E' ) = { (, n }
FIRST( F ) = { (, n }
FIRST( T ) = { (, n }
FIRST( n ) = { n }
```

图 2　FIRST 集合

构造 LR(1) 项目集规范族，得到的项目集规范族如图：

```
═══ LR(1) Sets of Items ═══
I0:
E → · E + T,     $ + -
E → · E - T,     $ + -
E → · T,         $ + -
E' → · E,        $
F → · ( E ),     $ * + - /
F → · n,         $ * + - /
T → · F,         $ * + - /
T → · T * F,     $ * + - /
T → · T / F,     $ * + - /

I1:
E → · E + T,     ) + -
E → · E - T,     ) + -
E → · T,         ) + -
F → · ( E ),     ) * + - /
F → ( · E ),     $ * + - /
F → · n,         ) * + - /
T → · F,         ) * + - /
T → · T * F,     ) * + - /
T → · T / F,     ) * + - /

I2:
E → E · + T,     $ + -
E → E · - T,     $ + -
E' → E ·,        $

I3:
T → F ·,         $ * + - /

I4:
E → T ·,         $ + -
T → T · * F,     $ * + - /
T → T · / F,     $ * + - /

I5:
F → n ·,         $ * + - /

I6:
E → · E + T,     ) + -
E → · E - T,     ) + -
E → · T,         ) + -
F → · ( E ),     ) * + - /
F → ( · E ),     ) * + - /
F → · n,         ) * + - /
T → · F,         ) * + - /
T → · T * F,     ) * + - /
T → · T / F,     ) * + - /

I7:
E → E · + T,     ) + -
E → E · - T,     ) + -
F → ( E · ),     $ * + - /

I8:
T → F ·,         ) * + - /

I9:
E → T ·,         ) + -
T → T · * F,     ) * + - /
T → T · / F,     ) * + - /

I10:
F → n ·,         ) * + - /

I11:
E → E + · T,     $ + -
F → · ( E ),     $ * + - /
F → · n,         $ * + - /
T → · F,         $ * + - /
T → · T * F,     $ * + - /
T → · T / F,     $ * + - /

I12:
E → E - · T,     $ + -
F → · ( E ),     $ * + - /
F → · n,         $ * + - /
T → · F,         $ * + - /
T → · T * F,     $ * + - /
T → · T / F,     $ * + - /

I13:
F → · ( E ),     $ * + - /
F → · n,         $ * + - /
T → T * · F,     $ * + - /

I14:
F → · ( E ),     $ * + - /
F → · n,         $ * + - /
T → T / · F,     $ * + - /

I15:
E → E · + T,     ) + -
E → E · - T,     ) + -
F → ( E · ),     ) * + - /

I16:
F → ( E ) ·,     $ * + - /

I17:
E → E + · T,     ) + -
F → · ( E ),     ) * + - /
F → · n,         ) * + - /
T → · F,         ) * + - /
T → · T * F,     ) * + - /
T → · T / F,     ) * + - /

I18:
E → E - · T,     ) + -
F → · ( E ),     ) * + - /
F → · n,         ) * + - /
T → · F,         ) * + - /
T → · T * F,     ) * + - /
T → · T / F,     ) * + - /

I19:
F → · ( E ),     ) * + - /
F → · n,         ) * + - /
T → T * · F,     ) * + - /

I20:
F → · ( E ),     ) * + - /
F → · n,         ) * + - /
T → T / · F,     ) * + - /

I21:
E → E + T ·,     $ + -
T → T · * F,     $ * + - /
T → T · / F,     $ * + - /

I22:
E → E - T ·,     $ + -
T → T · * F,     $ * + - /
T → T · / F,     $ * + - /

I23:
T → T * F ·,     $ * + - /

I24:
T → T / F ·,     $ * + - /

I25:
F → ( E ) ·,     ) * + - /

I26:
E → E + T ·,     ) + -
T → T · * F,     ) * + - /
T → T · / F,     ) * + - /

I27:
E → E - T ·,     ) + -
T → T · * F,     ) * + - /
T → T · / F,     ) * + - /

I28:
T → T * F ·,     ) * + - /

I29:
T → T / F ·,     ) * + - /
```

图 3　LR(1) 项目集规范族

构造识别其所有活前缀的 DFA，得到的转移如下图：

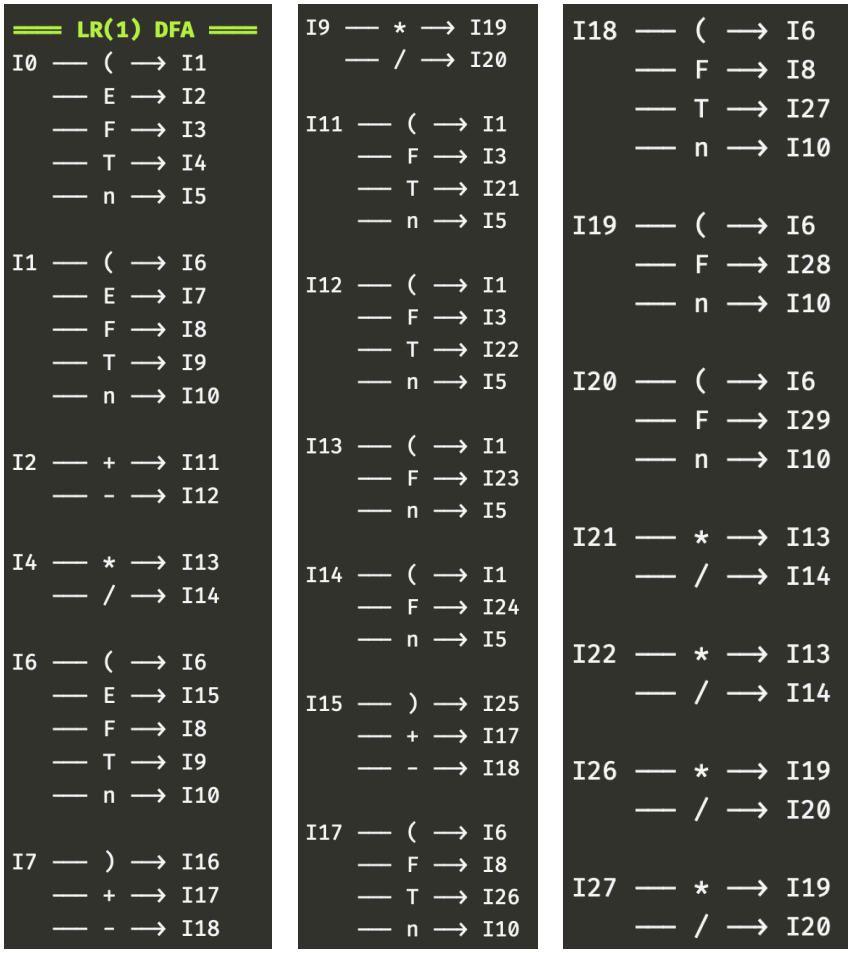


图 4　DFA 的转移

根据得到的转移画出的 DFA 如下图（见下页，原图见附录 B）：

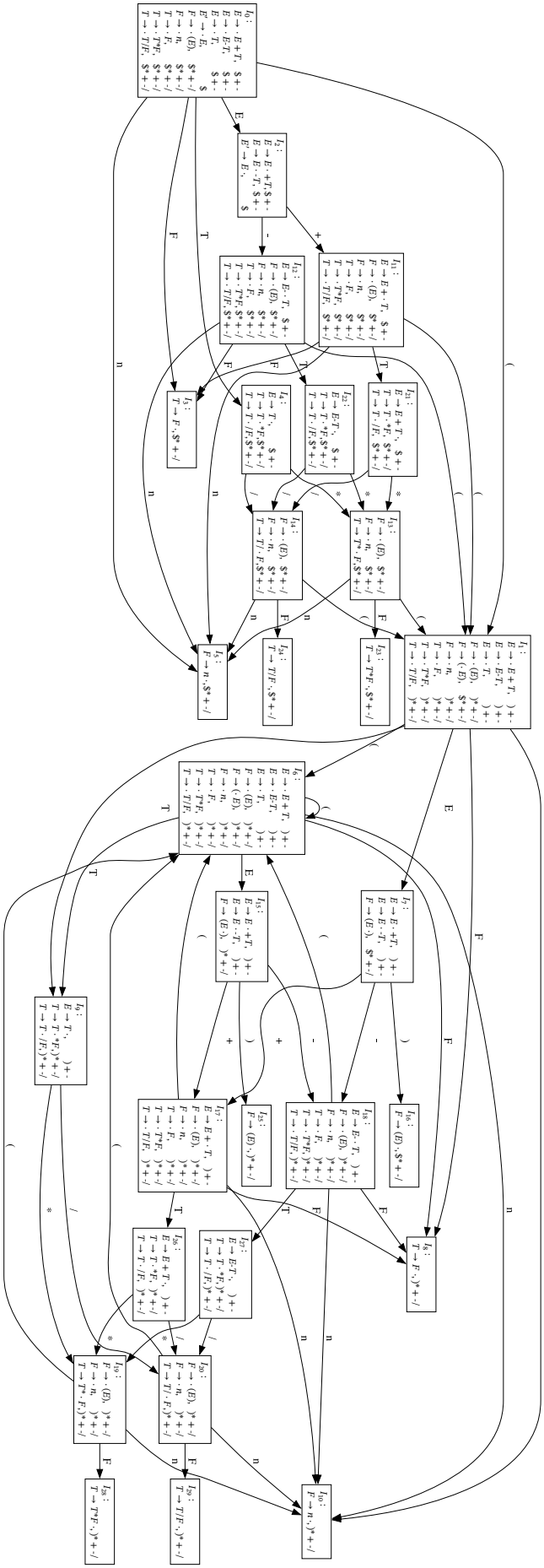


图 5 由给定文法得到的识别所有 LR(1) 活前缀的 DFA

构造 LR(1) 分析表，得到的 LR(1) 分析表如下图：

Parsing Table

	(	)	*	+	-	/	n	$	E	E'	F	T
I0	s1						s5		2		3	4
I1	s6						s10		7		8	9
I2				s11	s12			acc				
I3			r6	r6	r6	r6		r6				
I4			s13	r2	r2	s14		r2				
I5			r5	r5	r5	r5		r5				
I6	s6						s10		15		8	9
I7		s16		s17	s18							
I8		r6	r6	r6	r6	r6						
I9		r2	s19	r2	r2	s20						
I10		r5	r5	r5	r5	r5						
I11	s1						s5				3	21
I12	s1						s5				3	22
I13	s1						s5				23	
I14	s1						s5				24	
I15		s25		s17	s18							
I16			r4	r4	r4	r4		r4				
I17	s6						s10				8	26
I18	s6						s10				8	27
I19	s6						s10				28	
I20	s6						s10				29	
I21			s13	r0	r0	s14		r0				
I22			s13	r1	r1	s14		r1				
I23			r7	r7	r7	r7		r7				
I24			r8	r8	r8	r8		r8				
I25		r4	r4	r4	r4	r4						
I26		r0	s19	r0	r0	s20						
I27		r1	s19	r1	r1	s20						
I28		r7	r7	r7	r7	r7						
I29		r8	r8	r8	r8	r8						

图 6　LR(1) 分析表

3.2 对于任给的算术表达式的语法分析

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

1. [接受] `1 + 1`

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

2. [接受] `(114 + 514) * 1919 / 810`
3. [出错] `1 1`
4. [出错] `1025 - (365) +`
5. [出错] `1 ++ 2 ** 3 // 4 -- 5`
6. [出错] `(9 + ) 8`
7. [出错] `(309 * 23`
8. [出错] `(1 ** ((1 + 4) * (5 - )) - 4 + (1 / (9 - 1))) - 9`（包含 3 个错误）
9. [接受] `((31 * ((41 + 5) * (926 - 5)) - 3 + (4 / (7 - 1)) - 2) + (4 + ((9 - 2) * 3) / 6 - (1 + 8) / (7 - 1)) / ((3 * (4 + 2)) + (15 + (234 * (2 + (9 - 3))) - 2 * (8 - (6 / 3)) + (7 + 1) - (5 / (234 + 1)) + 9 / ((2 * 3) - (8 + 1)) + 4 - (6 * 2)))) / 0`

以下是每个例子的运行结果或部分运行结果。

1. [接受] `1 + 1`

```
1 + 1
═══ Token Stream ═══
n + n

═══ Parsing procedure ═══
```

Stack	Symbols	Input	Action
0		n+n$	Shift 5
0 5	n	+n$	Reduce F → n
0 3	F	+n$	Reduce T → F
0 4	T	+n$	Reduce E → T
0 2	E	+n$	Shift 11
0 2 11	E+	n$	Shift 5
0 2 11 5	E+n	$	Reduce F → n
0 2 11 3	E+F	$	Reduce T → F
0 2 11 21	E+T	$	Reduce E → E + T
0 2	E	$	Accept

图 7　例 1 - 接受

2. [接受] `(114 + 514) * 1919 / 810`

```
(114 + 514) * 1919 / 810
=== Token Stream ===
( n + n ) * n / n

=== Parsing procedure ===
```

Stack	Symbols	Input	Action
0		(n+n)*n/n$	Shift 1
0 1	(	n+n)*n/n$	Shift 10
0 1 10	(n	+n)*n/n$	Reduce F → n
0 1 8	(F	+n)*n/n$	Reduce T → F
0 1 9	(T	+n)*n/n$	Reduce E → T
0 1 7	(E	+n)*n/n$	Shift 17
0 1 7 17	(E+	n)*n/n$	Shift 10
0 1 7 17 10	(E+n	)*n/n$	Reduce F → n
0 1 7 17 8	(E+F	)*n/n$	Reduce T → F
0 1 7 17 26	(E+T	)*n/n$	Reduce E → E + T
0 1 7	(E	)*n/n$	Shift 16
0 1 7 16	(E)	*n/n$	Reduce F → (E)
0 3	F	*n/n$	Reduce T → F
0 4	T	*n/n$	Shift 13
0 4 13	T*	n/n$	Shift 5
0 4 13 5	T*n	/n$	Reduce F → n
0 4 13 23	T*F	/n$	Reduce T → T * F
0 4	T	/n$	Shift 14
0 4 14	T/	n$	Shift 5
0 4 14 5	T/n	$	Reduce F → n
0 4 14 24	T/F	$	Reduce T → T / F
0 4	T	$	Reduce E → T
0 2	E	$	Accept

图 8　例 2 - 接受

3. [出错] `1 1`

```
1 1
=== Token Stream ===
n n

=== Parsing procedure ===
```

Stack	Symbols	Input	Action
0		nn$	Shift 5
0 5	n	n$	Error
0		n$	Shift 5
0 5	n	$	Reduce F → n
0 3	F	$	Reduce T → F
0 4	T	$	Reduce E → T
0 2	E	$	Finish [ERROR OCCURRED]

图 9　例 3 - 出错

4. [出错] `1025 - (365) +`

```
1025 - (365) +
═══ Token Stream ═══
n - ( n ) +

═══ Parsing procedure ═══
```

Stack	Symbols	Input	Action
0		n-(n)+$	Shift 5
0 5	n	-(n)+$	Reduce F → n
0 3	F	-(n)+$	Reduce T → F
0 4	T	-(n)+$	Reduce E → T
0 2	E	-(n)+$	Shift 12
0 2 12	E-	(n)+$	Shift 1
0 2 12 1	E-(	n)+$	Shift 10
0 2 12 1 10	E-(n	)+$	Reduce F → n
0 2 12 1 8	E-(F	)+$	Reduce T → F
0 2 12 1 9	E-(T	)+$	Reduce E → T
0 2 12 1 7	E-(E	)+$	Shift 16
0 2 12 1 7 16	E-(E)	+$	Reduce F → (E)
0 2 12 3	E-F	+$	Reduce T → F
0 2 12 22	E-T	+$	Reduce E → E - T
0 2	E	+$	Shift 11
0 2 11	E+	$	Error
0 2	E	$	Finish [ERROR OCCURRED]

图 10　例 4 - 出错

5. [出错] `1 ++ 2 ** 3 // 4 -- 5`

```
1 ++ 2 ** 3 // 4 -- 5
═══ Token Stream ═══
n + + n * * n / / n - - n

═══ Parsing procedure ═══
```

Stack	Symbols	Input	Action
0		n++n**n//n--n$	Shift 5
0 5	n	++n**n//n--n$	Reduce F → n
0 3	F	++n**n//n--n$	Reduce T → F
0 4	T	++n**n//n--n$	Reduce E → T
0 2	E	++n**n//n--n$	Shift 11
0 2 11	E+	+n**n//n--n$	Error
0 2	E	+n**n//n--n$	Shift 11
0 2 11	E+	n**n//n--n$	Shift 5
0 2 11 5	E+n	**n//n--n$	Reduce F → n
0 2 11 3	E+F	**n//n--n$	Reduce T → F
0 2 11 21	E+T	**n//n--n$	Shift 13
0 2 11 21 13	E+T*	*n//n--n$	Error
0 2 11 21	E+T	*n//n--n$	Shift 13
0 2 11 21 13	E+T*	n//n--n$	Shift 5
0 2 11 21 13 5	E+T*n	//n--n$	Reduce F → n
0 2 11 21 13 23	E+T*F	//n--n$	Reduce T → T * F
0 2 11 21	E+T	//n--n$	Shift 14
0 2 11 21 14	E+T/	/n--n$	Error
0 2 11 21	E+T	/n--n$	Shift 14
0 2 11 21 14	E+T/	n--n$	Shift 5
0 2 11 21 14 5	E+T/n	--n$	Reduce F → n
0 2 11 21 14 24	E+T/F	--n$	Reduce T → T / F
0 2 11 21	E+T	--n$	Reduce E → E + T
0 2	E	--n$	Shift 12
0 2 12	E-	-n$	Error
0 2	E	-n$	Shift 12
0 2 12	E-	n$	Shift 5
0 2 12 5	E-n	$	Reduce F → n
0 2 12 3	E-F	$	Reduce T → F
0 2 12 22	E-T	$	Reduce E → E - T
0 2	E	$	Finish [ERROR OCCURRED]

图 11　例 5 - 出错

6. [出错] `(9 + ) 8`

```
(9 + ) 8
═══ Token Stream ═══
( n + ) n

═══ Parsing procedure ═══
```

Stack	Symbols	Input	Action
0		(n+)n$	Shift 1
0 1	(	n+)n$	Shift 10
0 1 10	(n	+)n$	Reduce F → n
0 1 8	(F	+)n$	Reduce T → F
0 1 9	(T	+)n$	Reduce E → T
0 1 7	(E	+)n$	Shift 17
0 1 7 17	(E+	)n$	Error
0 1 7	(E	)n$	Shift 16
0 1 7 16	(E)	n$	Error
0 1	(	n$	Shift 10
0 1 10	(n	$	Error
0		$	Finish [ERROR OCCURRED]

图 12　例 6 - 出错

7. [出错] `(309 * 23`

```
(309 * 23
═══ Token Stream ═══
( n * n

═══ Parsing procedure ═══
```

Stack	Symbols	Input	Action
0		(n*n$	Shift 1
0 1	(	n*n$	Shift 10
0 1 10	(n	*n$	Reduce F → n
0 1 8	(F	*n$	Reduce T → F
0 1 9	(T	*n$	Shift 19
0 1 9 19	(T*	n$	Shift 10
0 1 9 19 10	(T*n	$	Error
0		$	Finish [ERROR OCCURRED]

图 13　例 7 - 出错

8. [出错] `(1 ** ((1 + 4) * (5 - )) - 4 + (1 / (9 - 1))) - 9)`（包含 3 个错误）

```
(1 ** ((1 + 4) * (5 - )) - 4 + (1 / (9 - 1))) - 9)
=== Token Stream ===
( n * * ( ( n + n ) * ( n - ) ) - n + ( n / ( n - n ) ) ) - n )

=== Parsing procedure ===
```

Stack	Symbols	Input	Action
0		(n**((n+n)*(n-))-n+(n/(n-n)))-n)$	Shift 1
0 1	(	n**((n+n)*(n-))-n+(n/(n-n)))-n)$	Shift 10
0 1 10	(n	**((n+n)*(n-))-n+(n/(n-n)))-n)$	Reduce F → n
0 1 8	(F	**((n+n)*(n-))-n+(n/(n-n)))-n)$	Reduce T → F
0 1 9	(T	**((n+n)*(n-))-n+(n/(n-n)))-n)$	Shift 19
0 1 9 19	(T*	*((n+n)*(n-))-n+(n/(n-n)))-n)$	Error
0 1 9	(T	*((n+n)*(n-))-n+(n/(n-n)))-n)$	Shift 19
0 1 9 19	(T*	((n+n)*(n-))-n+(n/(n-n)))-n)$	Shift 6
0 1 9 19 6	(T*(	(n+n)*(n-))-n+(n/(n-n)))-n)$	Shift 6
0 1 9 19 6 6	(T*((	n+n)*(n-))-n+(n/(n-n)))-n)$	Shift 10
0 1 9 19 6 6 10	(T*((n	+n)*(n-))-n+(n/(n-n)))-n)$	Reduce F → n
0 1 9 19 6 6 8	(T*((F	+n)*(n-))-n+(n/(n-n)))-n)$	Reduce T → F
0 1 9 19 6 6 9	(T*((T	+n)*(n-))-n+(n/(n-n)))-n)$	Reduce E → T
0 1 9 19 6 6 15	(T*((E	+n)*(n-))-n+(n/(n-n)))-n)$	Shift 17
0 1 9 19 6 6 15 17	(T*((E+	n)*(n-))-n+(n/(n-n)))-n)$	Shift 10
0 1 9 19 6 6 15 17 10	(T*((E+n	)*(n-))-n+(n/(n-n)))-n)$	Reduce F → n
0 1 9 19 6 6 15 17 8	(T*((E+F	)*(n-))-n+(n/(n-n)))-n)$	Reduce T → F
0 1 9 19 6 6 15 17 26	(T*((E+T	)*(n-))-n+(n/(n-n)))-n)$	Reduce E → E + T
0 1 9 19 6 6 15	(T*((E	)*(n-))-n+(n/(n-n)))-n)$	Shift 25
0 1 9 19 6 6 15 25	(T*((E)	*(n-))-n+(n/(n-n)))-n)$	Reduce F → (E)
0 1 9 19 6 8	(T*(F	*(n-))-n+(n/(n-n)))-n)$	Reduce T → F
0 1 9 19 6 9	(T*(T	*(n-))-n+(n/(n-n)))-n)$	Shift 19
0 1 9 19 6 9 19	(T*(T*	(n-))-n+(n/(n-n)))-n)$	Shift 6
0 1 9 19 6 9 19 6	(T*(T*(	n-))-n+(n/(n-n)))-n)$	Shift 10
0 1 9 19 6 9 19 6 10	(T*(T*(n	-))-n+(n/(n-n)))-n)$	Reduce F → n
0 1 9 19 6 9 19 6 8	(T*(T*(F	-))-n+(n/(n-n)))-n)$	Reduce T → F
0 1 9 19 6 9 19 6 9	(T*(T*(T	-))-n+(n/(n-n)))-n)$	Reduce E → T
0 1 9 19 6 9 19 6 15	(T*(T*(E	-))-n+(n/(n-n)))-n)$	Shift 18
0 1 9 19 6 9 19 6 15 18	(T*(T*(E-	))-n+(n/(n-n)))-n)$	Error
0 1 9 19 6 9 19 6 15	(T*(T*(E	))-n+(n/(n-n)))-n)$	Shift 25
0 1 9 19 6 9 19 6 15 25	(T*(T*(E)	)-n+(n/(n-n)))-n)$	Reduce F → (E)
0 1 9 19 6 9 19 28	(T*(T*F	)-n+(n/(n-n)))-n)$	Reduce T → T * F
0 1 9 19 6 9	(T*(T	)-n+(n/(n-n)))-n)$	Reduce E → T
0 1 9 19 6 15	(T*(E	)-n+(n/(n-n)))-n)$	Shift 25
0 1 9 19 6 15 25	(T*(E)	-n+(n/(n-n)))-n)$	Reduce F → (E)
0 1 9 19 28	(T*F	-n+(n/(n-n)))-n)$	Reduce T → T * F
0 1 9	(T	-n+(n/(n-n)))-n)$	Reduce E → T
0 1 7	(E	-n+(n/(n-n)))-n)$	Shift 18
0 1 7 18	(E-	n+(n/(n-n)))-n)$	Shift 10
0 1 7 18 10	(E-n	+(n/(n-n)))-n)$	Reduce F → n
0 1 7 18 8	(E-F	+(n/(n-n)))-n)$	Reduce T → F
0 1 7 18 27	(E-T	+(n/(n-n)))-n)$	Reduce E → E - T
0 1 7	(E	+(n/(n-n)))-n)$	Shift 17
0 1 7 17	(E+	(n/(n-n)))-n)$	Shift 6
0 1 7 17 6	(E+(	n/(n-n)))-n)$	Shift 10
0 1 7 17 6 10	(E+(n	/(n-n)))-n)$	Reduce F → n
0 1 7 17 6 8	(E+(F	/(n-n)))-n)$	Reduce T → F
0 1 7 17 6 9	(E+(T	/(n-n)))-n)$	Shift 20
0 1 7 17 6 9 20	(E+(T/	(n-n)))-n)$	Shift 6
0 1 7 17 6 9 20 6	(E+(T/(	n-n)))-n)$	Shift 10
0 1 7 17 6 9 20 6 10	(E+(T/(n	-n)))-n)$	Reduce F → n
0 1 7 17 6 9 20 6 8	(E+(T/(F	-n)))-n)$	Reduce T → F
0 1 7 17 6 9 20 6 9	(E+(T/(T	-n)))-n)$	Reduce E → T
0 1 7 17 6 9 20 6 15	(E+(T/(E	-n)))-n)$	Shift 18
0 1 7 17 6 9 20 6 15 18	(E+(T/(E-	n)))-n)$	Shift 10
0 1 7 17 6 9 20 6 15 18 10	(E+(T/(E-n	))-n)$	Reduce F → n
0 1 7 17 6 9 20 6 15 18 8	(E+(T/(E-F	))-n)$	Reduce T → F
0 1 7 17 6 9 20 6 15 18 27	(E+(T/(E-T	))-n)$	Reduce E → E - T
0 1 7 17 6 9 20 6 15	(E+(T/(E	))-n)$	Shift 25
0 1 7 17 6 9 20 6 15 25	(E+(T/(E)	)-n)$	Reduce F → (E)
0 1 7 17 6 9 20 29	(E+(T/F	)-n)$	Reduce T → T / F
0 1 7 17 6 9	(E+(T	)-n)$	Reduce E → T
0 1 7 17 6 15	(E+(E	)-n)$	Shift 25
0 1 7 17 6 15 25	(E+(E)	-n)$	Reduce F → (E)
0 1 7 17 8	(E+F	-n)$	Reduce T → F
0 1 7 17 26	(E+T	-n)$	Reduce E → E + T
0 1 7	(E	-n)$	Shift 16
0 1 7 16	(E)	-n)$	Reduce F → (E)
0 3	F	-n)$	Reduce T → F
0 4	T	-n)$	Reduce E → T
0 2	E	-n)$	Shift 12
0 2 12	E-	n)$	Shift 5
0 2 12 5	E-n	)$	Error
0		)$	Finish [ERROR OCCURRED]

图 14　例 8 - 出错

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

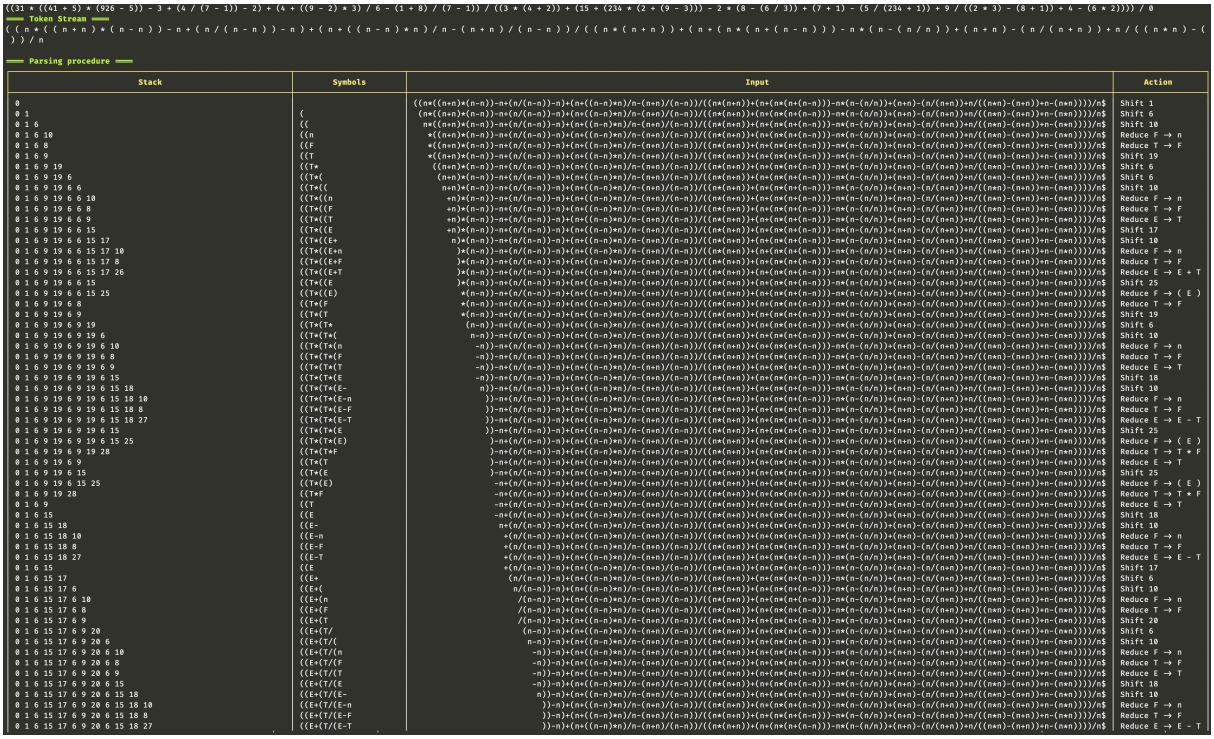


图 15　例 9 - 接受（开头，请放大查看）

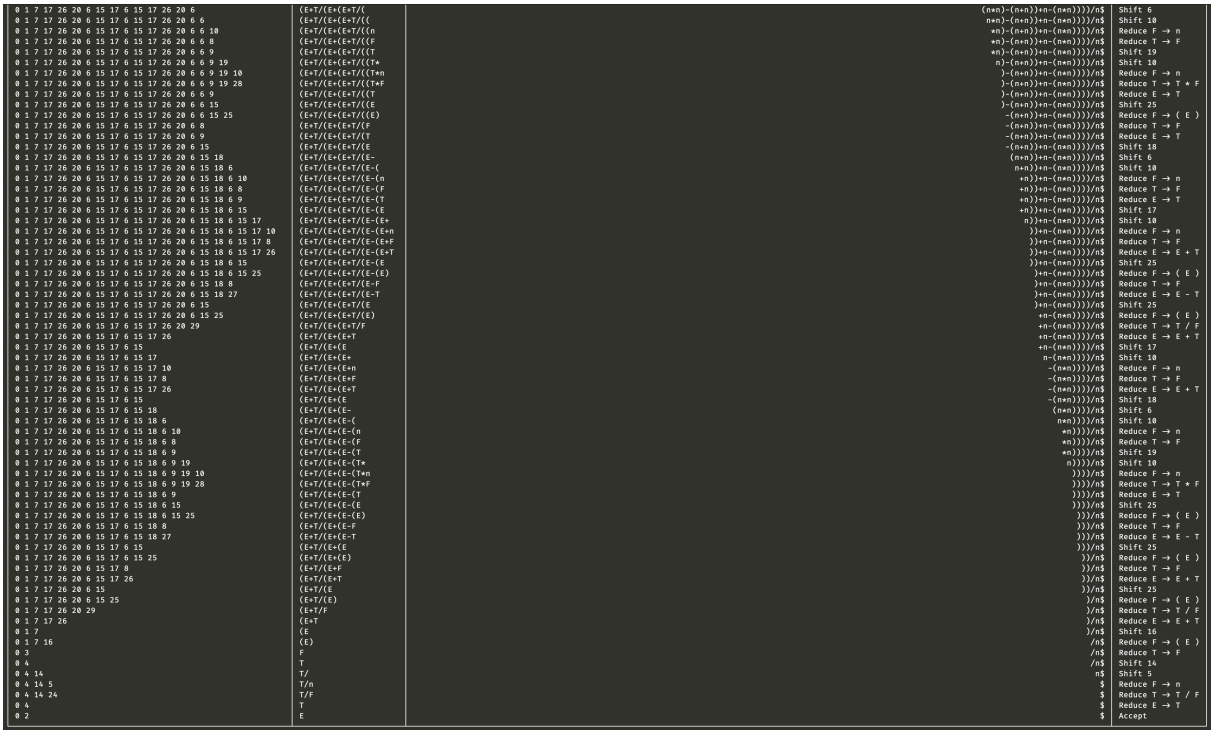


图 16　例 9 - 接受（结尾，请放大查看）

[2]因篇幅较长（足足 357 行），在此只给出运行结果的开头和结尾。

4 总结

在本次实验报告中，我成功实现了一个名为 ExParserR 的自底向上的 LR(1) 算术表达式语法分析器。通过这次实验，我深入理解了 LR(1) 分析法的理论基础和实际应用。实验过程中，我将复杂的理论知识如 LR(1) 分析法应用于实际编程中，实现了构造增广文法、FIRST 集的计算、项目集规范族，以及最终的 LR(1) 分析表的算法。

总的来说，这次实验不仅加深了我对编译原理中 LR(1) 分析法的理解，还提升了我的编程能力和问题解决技巧。通过不断的实践和调整，我们成功实现了一个功能强大的语法分析器。这次经验将为我未来在编译原理和相关领域的学习和研究提供宝贵的基础。

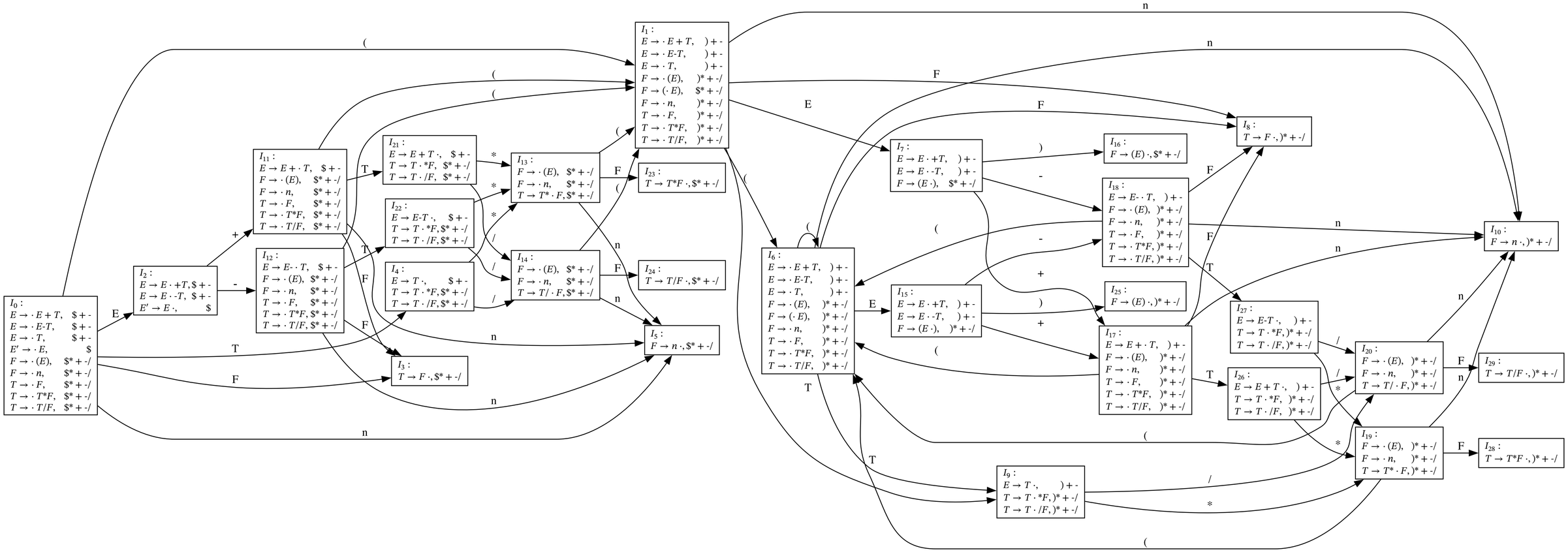
如对本报告有任何疑问或建议，请随时联系我。

感谢阅读。

附录 A – 附件清单

```
ExParserR
├── CMakeLists.txt
└── src
    ├── main.cpp
    ├── parser
    │   ├── dfa.cpp
    │   ├── dfa.h
    │   ├── grammar.cpp
    │   ├── grammar.h
    │   ├── item.cpp
    │   ├── item.h
    │   ├── item_set.cpp
    │   ├── item_set.h
    │   ├── parser.cpp
    │   ├── parser.h
    │   ├── symbol.cpp
    │   └── symbol.h
    ├── simple_lexer
    │   ├── lexer.cpp
    │   ├── lexer.h
    │   └── token.h
    └── util
        ├── all.h
        ├── functional.h
        ├── overloaded.h
        ├── table.h
        └── type.h
```

附录 B – DFA 原尺寸图


I_0 : $E \to \cdot E+T,\ \$+-$; $E \to \cdot E\text{-}T,\ \$+-$; $E \to \cdot T,\ \$+-$; $E' \to \cdot E,\ \$$; $F \to \cdot(E),\ \$*+-/$; $F \to \cdot n,\ \$*+-/$; $T \to \cdot F,\ \$*+-/$; $T \to \cdot T*F,\ \$*+-/$; $T \to \cdot T/F,\ \$*+-/$
I_1 : $E \to \cdot E+T,\ )+-$; $E \to \cdot E\text{-}T,\ )+-$; $E \to \cdot T,\ )+-$; $F \to \cdot(E),\ )*+-/$; $F \to (\cdot E),\ \$*+-/$; $F \to \cdot n,\ )*+-/$; $T \to \cdot F,\ )*+-/$; $T \to \cdot T*F,\ )*+-/$; $T \to \cdot T/F,\ )*+-/$
I_2 : $E \to E \cdot +T,\$+-$; $E \to E \cdot \text{-}T,\ \$+-$; $E' \to E \cdot,\ \$$
I_3 : $T \to F \cdot,\$*+-/$
I_4 : $T \to T \cdot,\ \$+-$; $T \to T \cdot *F,\$*+-/$; $T \to T \cdot /F,\$*+-/$
I_5 : $F \to n \cdot,\$*+-/$
I_6 : $E \to \cdot E+T,\ )+-$; $E \to \cdot E\text{-}T,\ )+-$; $E \to \cdot T,\ )+-$; $F \to \cdot(E),\ )*+-/$; $F \to (\cdot E),\ )*+-/$; $F \to \cdot n,\ )*+-/$; $T \to \cdot F,\ )*+-/$; $T \to \cdot T*F,\ )*+-/$; $T \to \cdot T/F,\ )*+-/$
I_7 : $E \to E \cdot +T,\ )+-$; $E \to E \cdot \text{-}T,\ )+-$; $F \to (E \cdot),\ \$*+-/$
I_8 : $T \to F \cdot,)*+-/$
I_9 : $E \to T \cdot,\ )+-$; $T \to T \cdot *F,)*+-/$; $T \to T \cdot /F,)*+-/$
I_{10} : $F \to n \cdot,)*+-/$
I_{11} : $E \to E+\cdot T,\ \$+-$; $F \to \cdot(E),\ \$*+-/$; $F \to \cdot n,\ \$*+-/$; $T \to \cdot F,\ \$*+-/$; $T \to \cdot T*F,\ \$*+-/$; $T \to \cdot T/F,\ \$*+-/$
I_{12} : $E \to E\text{-}\cdot T,\ \$+-$; $F \to \cdot(E),\ \$*+-/$; $F \to \cdot n,\ \$*+-/$; $T \to \cdot F,\ \$*+-/$; $T \to \cdot T*F,\$*+-/$; $T \to \cdot T/F,\$*+-/$
I_{13} : $F \to \cdot(E),\ \$*+-/$; $F \to \cdot n,\ \$*+-/$; $T \to T*\cdot F,\$*+-/$
I_{14} : $F \to \cdot(E),\ \$*+-/$; $F \to \cdot n,\ \$*+-/$; $T \to T/\cdot F,\$*+-/$
I_{15} : $E \to E \cdot +T,\ )+-$; $E \to E \cdot \text{-}T,\ )+-$; $F \to (E \cdot),\ )*+-/$
I_{16} : $F \to (E) \cdot,\$*+-/$
I_{17} : $E \to E+\cdot T,\ )+-$; $F \to \cdot(E),\ )*+-/$; $F \to \cdot n,\ )*+-/$; $T \to \cdot F,\ )*+-/$; $T \to \cdot T*F,\ )*+-/$; $T \to \cdot T/F,\ )*+-/$
I_{18} : $E \to E\text{-}\cdot T,\ )+-$; $F \to \cdot(E),\ )*+-/$; $F \to \cdot n,\ )*+-/$; $T \to \cdot F,\ )*+-/$; $T \to \cdot T*F,)*+-/$; $T \to \cdot T/F,)*+-/$
I_{19} : $F \to \cdot(E),\ )*+-/$; $F \to \cdot n,\ )*+-/$; $T \to T*\cdot F,)*+-/$
I_{20} : $F \to \cdot(E),\ )*+-/$; $F \to \cdot n,\ )*+-/$; $T \to T/\cdot F,)*+-/$
I_{21} : $E \to E+T \cdot,\ \$+-$; $T \to T \cdot *F,\ \$*+-/$; $T \to T \cdot /F,\ \$*+-/$
I_{22} : $E \to E\text{-}T \cdot,\ \$+-$; $T \to T \cdot *F,\$*+-/$; $T \to T \cdot /F,\$*+-/$
I_{23} : $T \to T*F \cdot,\$*+-/$
I_{24} : $T \to T/F \cdot,\$*+-/$
I_{25} : $F \to (E) \cdot,)*+-/$
I_{26} : $E \to E+T \cdot,\ )+-$; $T \to T \cdot *F,\ )*+-/$; $T \to T \cdot /F,\ )*+-/$
I_{27} : $E \to E\text{-}T \cdot,\ )+-$; $T \to T \cdot *F,)*+-/$; $T \to T \cdot /F,)*+-/$
I_{28} : $T \to T*F \cdot,)*+-/$
I_{29} : $T \to T/F \cdot,)*+-/$