编译原理与技术实验三：语法分析程序的设计与实现
实验报告

毛子恒
2019211397
北京邮电大学 计算机学院

日期：2021 年 11 月 21 日

1 概览

1.1 任务描述

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

$$E \to E+T|E-T|T$$
$$T \to T*F|T/F|F$$
$$F \to (E)|num$$

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. 构造 LR 分析程序。

1.2 开发环境

- macOS Big Sur 11.6.1
- Apple clang version 12.0.5
- cmake version 3.20.3
- Clion 2021.2.1
- Visual Studio Code 1.62.3

2 模块介绍

2.1 模块划分

各模块及其关系如图 1。

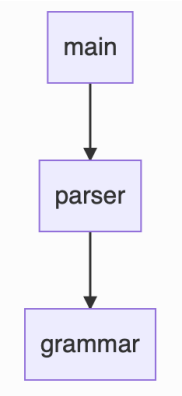

图 1: 模块关系图

其中，grammar 模块定义了文法类，其中实现了文法的输入、拓广文法、构建 LR(1) 项目集规范族及识别所有活前缀 DFA 的算法；parser 模块定义了 LR 预测分析类，实现了从 LR(1) 项目集规范族及识别所有活前缀的 DFA 构造 LR(1) 分析表的算法，以及 LR 分析算法。

主函数的流程如下：

- 从文件中读入文法。
- 构造 LR(1) 项目集规范族及识别所有活前缀的 DFA 并且判断是否是 LR(1) 文法。
- 利用 DFA 构建预测分析表。
- 从文件中读入需要分析的字符串。
- 对字符串进行 LR 分析，输出分析过程。

2.2 文法类

grammar 模块中定义了 LR(1) 文法类：

```
using Symbol = std::string;
using SymbolSet = std::unordered_set<Symbol>;
using ProductionRight = std::deque<std::string>;
using Production = std::pair<Symbol, ProductionRight>;
using Productions = std::unordered_map<Symbol, std::vector<Production>>;
using Item = std::pair<const Production *, ProductionRight::const_iterator>;
extern const std::string EMPTY;
struct ItemSet
{
    std::map<Item, SymbolSet> shift_item;
    std::map<Item, SymbolSet> reduce_item;
    bool operator==(const ItemSet &rhs);
};
struct DFA
{
    std::vector<ItemSet *> states;
    std::vector<std::unordered_map<Symbol, int>> next;
};
class Grammar
{
```

```
21  public:
22      Grammar() = default;
23      void LoadFromFile(std::ifstream &fs);
24      bool ConstructLR1StateMachine();
25      const Symbol &GetStart() const;
26      const SymbolSet &GetTerminal() const;
27      const DFA &GetDfa() const;
28  private:
29      SymbolSet nonterminal;
30      SymbolSet terminal;
31      Productions productions;
32      Symbol start;
33      std::unordered_map<Symbol, SymbolSet> first;
34      std::map<Item, SymbolSet> item_first;
35      DFA dfa;
36      void Closure(ItemSet &item_set);
37      void ConstructFirst(const Symbol &left);
38      void ConstructItemFirst();
39      void ExtendGrammar();
40      void ConstructDFA();
41      bool IsLR1Grammar() const;
42  };
43  template<typename T>
44  bool InSet(const std::unordered_set<T> &a, const T &b);
45  template<typename T1, typename T2>
46  bool InSet(const std::unordered_map<T1, T2> &a, const T1 &b);
47  std::ostream &operator<<(std::ostream &os, const SymbolSet &rhs);
48  std::ostream &operator<<(std::ostream &os, const ProductionRight &rhs);
49  std::ostream &operator<<(std::ostream &os, const Production &rhs);
50  std::ostream &operator<<(std::ostream &os, const Productions &rhs);
51  std::ostream &operator<<(std::ostream &os, const Item &rhs);
52  std::ostream &operator<<(std::ostream &os, const ItemSet &rhs);
53  std::ostream &operator<<(std::ostream &os, const DFA &rhs);
```

`ProductionRight` 类型表示产生式的右部，`Production` 类型表示产生式，`Item` 类型表示一个 LR(0) 项目（即不包含向前看符号集）；`ItemSet` 类型表示一个 LR(1) 项目集，其中分别有移进/待约项目和归约项目；`DFA` 类型表示一个识别所有活前缀的 DFA。

`Grammar` 类中包含有文法的非终结符号集合、终结符号集合、产生式集合、起始符号、非终结符的 FIRST 集合、待约项目的 FIRST 集合。

2.2.1 文法的输入

`LoadFromFile` 方法实现了从一个文件输入流中读取文法，一个描述1.1节中文法的文件示例如下：

```
$ Nonterminal symbols
E T F
$ Terminal symbols
```

```
+ - * / ( ) num
$ Start symbol
E
$ Productions
E -> E + T $ E - T $ T
T -> T * F $ T / F $ F
F -> ( E ) $ num
```

文件依次输入非终结符号集合、终结符号集合、起始符号、文法产生式集合。每个部分的开始都以一行单独的说明字符串标识，各个部分均可包含多行，各个符号之间以空格分隔。产生式中以 `$` 符号代替 `|` 符号。

输入时，程序会对文法的合法性进行基本的判断，包括非终结符号集和终结符号集不重合、起始符号是非终结符号、产生式左部是非终结符号、右部是非终结符号或者终结符号。

2.2.2 构造 LR(1) 项目集规范族及识别所有活前缀的 DFA

`ConstructLR1StateMachine` 方法实现了构造 LR(1) 项目集规范族及识别所有活前缀的 DFA，该方法依次调用 `ExtendGrammar`、`ConstructItemFirst`、`ConstructDFA`、`IsLR1Grammar` 方法，并在这期间输出调试信息。

拓广文法 `ExtendGrammar` 方法实现了拓广文法，即对于文法 $G = (N, T, P, S)$，生成与其等价的文法 $G' = (N \cup \{S'\}, T, P \cup \{S' \to S\}, S')$。

构建非终结符和待约项目的 FIRST 集 `ConstructItemFirst` 方法实现了构建非终结符和移进项目的 FIRST 集。

对于任意产生式 $A \to \alpha$，若 $\alpha \neq \varepsilon$，设该产生式为：

$$A \to Y_1 Y_2 \dots Y_k$$

遍历产生式右部的每一个 Y_i，如果：

- Y_i 是终结符，则 α 的 FIRST 集中增加 Y_i，终止遍历；
- Y_i 是非终结符，如果没有求出它的 FIRST 集，则递归求解。之后，α 的 FIRST 集并上 Y_i 的 FIRST 集。此后检查 Y_i 的 FIRST 集中是否包含 ε（即是否能推导出 ε），若不包含，则终止遍历。

最后，A 的 FIRST 集为各个候选式的 FIRST 集的并。

对于每一个待约项目 $A \to \alpha B \beta$，需要求出 β 的 FIRST 集，方法类似。

该算法的伪代码见**算法 1**。

构造 LR(1) 项目集的闭包 `Closure` 方法构造一个 LR(1) 项目集 I 的闭包 $closure(I)$，构造方法如下：

1. 初始化 $closure(I) \leftarrow I$；

算法 1: 构建非终结符和待约项目的 FIRST 集

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

输出: $\mathbf{FIRST}$

Function `ConstructFirst(`A`)`
  foreach $A \to \alpha \in \mathbf{P}$ **do**
    if $\alpha = \varepsilon$ **then**
      $\mathbf{FIRST}(A) \leftarrow \mathbf{FIRST}(A) \cup \{\varepsilon\}$;
      $\mathbf{FIRST}(\alpha) \leftarrow \{\varepsilon\}$;
    for $i \leftarrow 1$ **to** k **do** // $\alpha = Y_1 Y_2 \dots Y_k$
      if $Y_i \in \mathbf{T}$ **then**
        $\mathbf{FIRST}(\alpha) \leftarrow \mathbf{FIRST}(\alpha) \cup \{Y_i\}$;
        break;
      if $\mathbf{FIRST}(Y_i) = \varnothing$ **then** `ConstructFirst(`Y_i`)`;
      $\mathbf{FIRST}(\alpha) = \mathbf{FIRST}(\alpha) \cup \mathbf{FIRST}(Y_i)$;
      if $\varepsilon \notin \mathbf{FIRST}(Y_i)$ **then** break;
    if $i = k$ **then** $\mathbf{FIRST}(A) \leftarrow \mathbf{FIRST}(A) \cup \{\varepsilon\}$;
    $\mathbf{FIRST}(A) \leftarrow \mathbf{FIRST}(A) \cup \mathbf{FIRST}(\alpha)$;
Function `ConstructItemFirst()`
  foreach $A \to \alpha \in \mathbf{P}$ **do**
    for $i \leftarrow 1$ **to** k **do** // $\alpha = Y_1 Y_2 \dots Y_k$
      if $Y_i \in \mathbf{N}$ **then**
        for $j \leftarrow i + 1$ **to** k **do** // $\beta = Y_{i+1} \dots Y_k$
          if $Y_j \in \mathbf{T}$ **then**
            $\mathbf{FIRST}(\beta) \leftarrow \mathbf{FIRST}(\beta) \cup \{Y_j\}$;
            break;
          if $\mathbf{FIRST}(Y_j) = \varnothing$ **then** `ConstructFirst(`Y_j`)`;
          $\mathbf{FIRST}(\beta) = \mathbf{FIRST}(\beta) \cup \mathbf{FIRST}(Y_i)$;
          if $\varepsilon \notin \mathbf{FIRST}(Y_j)$ **then** break;

2. 对于 $[A \to \alpha \cdot B\beta, a] \in closure(I)$，若 $B \to \eta \in \mathrm{P}$，则 $\forall b \in FIRST(\beta a)$，使 $closure(I) \leftarrow closure(I) \cup [B \to \cdot\eta, b]$；
3. 重复 2 直至 $closure(I)$ 不再增大为止。

该算法的伪代码见**算法 2**。

算法 2: 构造 LR(1) 项目集的闭包

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

输出: $\mathbf{J} = closure(\mathbf{I})$

```
1  Function Closure(I)
2      J ← I;
3      do
4          J′ ← J;
5          foreach [A → α · Bβ, a] ∈ J do
6              foreach B → η ∈ P do
7                  foreach b ∈ (FIRST(β) − ε) do J′ ← [B → ·η, b];
8                  if ε ∈ FIRST(β) then J′ ← [B → ·η, a];
9          if J′ = J then break;
10         J ← J′;
11     while true;
```

构造 LR(1) 项目集规范族 `ConstructDFA` 方法构造 LR(1) 文法的项目集规范族和识别所有活前缀的 DFA。

该算法通过广度优先搜索求出识别所有活前缀的 DFA。该算法的伪代码见**算法 3**。

判断是否是 LR(1) 文法 `IsLR1Grammar` 方法判断文法是否是 LR(1) 文法，即检查每个项目集 $\mathbf{I}$，要求其中任意两个归约项目的向前看符号集合交集为空，任意一个归约项目向前看符号集合和移进项目的移进符号集合交集为空。

2.3 LR 分析

`parser` 模块定义了 LR 分析类：

```
1  class Parser
2  {
3  public:
4      Parser() = default;
5      explicit Parser(const Grammar &grammar);
6      bool ParseString(const ProductionRight &str);
7  private:
```

算法 3: 构造 LR(1) 项目集规范族

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

输出: $\mathbf{M} = (\mathbf{N} \cup \mathbf{T}, \mathbf{Q}, \mathbf{I}_0, \mathbf{F} = \mathbf{Q}, \delta)$

Function `ConstructDFA(I)`

- $\mathbf{I}_0 \leftarrow$ `Closure(`$\{[S' \to \cdot S, \$]\}$`)`;
- $\mathbf{Q} \leftarrow \mathbf{Q} \cup \{\mathbf{I}_0\}$;
- queue.`push`$(\mathbf{I}_0)$;
- total $\leftarrow 0$;
- **while** queue.`empty`$()$ **do**
  - $\mathbf{I}_i \leftarrow$ queue.`front`$()$;
  - queue.`pop`$()$;
  - **foreach** $X \in \mathbf{N} \cup \mathbf{T}$ **do**
    - go$(\mathbf{I}_i, X) \leftarrow$ `Closure(`$\{[A \to \alpha X \cdot \beta, a] | [A \to \alpha \cdot X\beta, a] \in \mathbf{I}_i\}$`)`;
    - **if** go$(\mathbf{I}_i, X) \neq \varnothing$ **then**
      - **if** go$(\mathbf{I}_i, X) \notin \mathbf{Q}$ **then**
        - total $\leftarrow$ total $+ 1$;
        - $\mathbf{I}_{\text{total}} \leftarrow$ go$(\mathbf{I}_i, X)$;
        - $\mathbf{Q} \leftarrow \mathbf{Q} \cup \{\mathbf{I}_{\text{total}}\}$;
        - $\delta(\mathbf{I}_i, X) \leftarrow \mathbf{I}_{\text{total}}$;
        - queue.`push`$(\mathbf{I}_{\text{total}})$;
      - **else** // $\mathbf{I}_j =$ go$(\mathbf{I}_i, X)$
        - $\delta(\mathbf{I}_i, X) \leftarrow \mathbf{I}_j$;

```
8       using Status = std::pair<std::pair<std::vector<int>, std::vector<Symbol>>,
        ↪  ProductionRight>;
9       enum ActionClass;
10      struct ActionItem
11      {
12          ActionClass action_class;
13          union
14          {
15              int index;
16              const Production * production;
17          };
18      };
19      SymbolSet terminal;
20      std::vector<std::unordered_map<Symbol, ActionItem>> action_table;
21      std::vector<std::unordered_map<Symbol, int>> goto_table;
22      int NextStep(Status &status) const;
23      friend std::ostream &operator<<(std::ostream &os, const Parser &rhs);
24      friend std::ostream &operator<<(std::ostream &os, const ActionItem
        ↪  &action_item);
25  };
26  std::ostream &operator<<(std::ostream &os, const std::pair<std::vector<int>,
    ↪  std::vector<Symbol>> &rhs);
```

2.3.1 构建 LR 分析表

`Parser(const Grammar &grammar)` 构造函数通过 LR(1) 项目集规范族构造一个 LR 分析表，构造算法伪代码见**算法 4**。

算法 4: 构建 LR 分析表

输入: $\mathbf{G} = (\mathbf{N}, \mathbf{T}, \mathbf{P}, S), \mathbf{M} = (\mathbf{N} \cup \mathbf{T}, \mathbf{Q}, \mathbf{I}_0, \mathbf{F} = \mathbf{Q}, \delta)$

输出: $\mathbf{action}, \mathbf{goto}$

Function `ConstructParsingTable(G)`

- **foreach** $I_i \in \mathbf{Q}$ **do**
  - **foreach** $\mathbf{I}_j = \delta(\mathbf{I}_i, X)$ **do**
    - **if** $X \in \mathbf{T}$ **then** $\mathbf{action}[i, X] \leftarrow Shift\ j$;
    - **else** $\mathbf{goto}[i, X] \leftarrow j$;
  - **foreach** $[A \to \alpha\cdot, a] \in \mathbf{I}_i \wedge A \neq S'$ **do**
    - $\mathbf{action}[i, a] \leftarrow Reduce\ A \to \alpha$;
  - **if** $[S' \to S, \$] \in \mathbf{I}_i$ **then** $\mathbf{action}[i, \$] \leftarrow ACC$;

2.3.2 LR 分析算法

`ParseString` 方法实现对一个字符串的 LR 分析，该函数首先初始化一个状态，之后依照状

态和当前字符串不返回断调用 `NextStep` 一步步进行 LR 分析，并且同时输出分析过程。该函数的返回值表示分析是否成功。

LR 分析算法的伪代码见**算法 5**。

算法 5: LR 分析算法

输入: 输入符号串 ω，文法 $\mathbf{G}$ 的分析表 $\mathbf{action}, \mathbf{goto}$

输出: 若 $\omega \in L(\mathbf{G})$，则输出 ω 的自底向上分析 answer，否则报告错误

```
Function ParseString(ω)
    statestack.push(0);
    symbolstack.push(NULL);
    buffer ← ω$;
    ip ← 0;
    do
        X ← statestack.top();
        a ← buffer[ip];
        if action[S, a] = Shift S′ then
            statestack.push(S′);
            symbolstack.push(a);
            ip ← ip + 1;
        else if action[S, a] = Reduce A → β then
            for i = 1to |β| do
                statestack.pop();
                symbolstack.pop();
            S′ ← statestack.top();
            statestack.push(goto[S′, A]);
            symbolstack.push(A);
            answer.push_back(A → β);
        else if action[S, a] = ACC then return answer;
        else return error;
    while true;
```

3 用户指南

在项目目录中执行以下命令来编译：

```
mkdir build
cd build
```

```
cmake ..
make
```

编译完成后，运行：

```
./Syntactic ../test/grammar.txt ../test/test1.txt
```

运行截图如图 2所示。

图 2: 运行截图

4 测试结果

4.1 测试集 1

此测试集用于简单地测试程序是否正常运行。

4.1.1 输入

文法如2.2.1节所示。

```
num + num
```

4.1.2 输出

```
[Nonterminal Symbols] F T E
[Terminal Symbols] * - ( num / ) +
[Start Symbol] E
```

```
[Productions]
F -> ( E )
F -> num
T -> T * F
T -> T / F
T -> F
E -> E + T
E -> E - T
E -> T

[Nonterminal Symbols] F T E' E
[Terminal Symbols] * - ( num / ) +
[Start Symbol] E'
[Productions]
F -> ( E )
F -> num
T -> T * F
T -> T / F
T -> F
E' -> E
E -> E + T
E -> E - T
E -> T

[DFA]
I0:
E' -> · E     $
E -> · E + T     + - $
E -> · E - T     + - $
E -> · T     + - $
T -> · T * F     $ - + / *
T -> · T / F     $ - + / *
T -> · F     $ - + / *
F -> · ( E )     + / * - $
F -> · num     + / * - $
--- F --> I3
--- T --> I2
--- num --> I5
--- ( --> I4
--- E --> I1

I1:
E -> E · + T     - $ +
E -> E · - T     - $ +
E' -> E ·     $
--- - --> I7
--- + --> I6

I2:
T -> T · * F     * + / $ -
T -> T · / F     * + / $ -
```

```
E -> T ·      - $ +
--- / --> I9
--- * --> I8

I3:
T -> F ·      * + / $ -

I4:
E ->  · E + T     - + )
E ->  · E - T     - + )
E ->  · T     - + )
T ->  · T * F     ) - + / *
T ->  · T / F     ) - + / *
T ->  · F     ) - + / *
F ->  · ( E )     - + / * )
F -> ( · E )     - $ * + /
F ->  · num     - + / * )
--- F --> I12
--- T --> I11
--- num --> I14
--- ( --> I13
--- E --> I10

I5:
F -> num ·      - $ * + /

I6:
E -> E + · T     $ - +
T ->  · T * F     $ - + / *
T ->  · T / F     $ - + / *
T ->  · F     $ - + / *
F ->  · ( E )     * $ - / +
F ->  · num     * $ - / +
--- num --> I5
--- ( --> I4
--- F --> I3
--- T --> I15

I7:
E -> E - · T     $ - +
T ->  · T * F     $ - + / *
T ->  · T / F     $ - + / *
T ->  · F     $ - + / *
F ->  · ( E )     * $ - / +
F ->  · num     * $ - / +
--- num --> I5
--- ( --> I4
--- F --> I3
--- T --> I16

I8:
```

```
T -> T * · F     * / + - $
F -> · ( E )     * / + - $
F -> · num     * / + - $
--- num --> I5
--- ( --> I4
--- F --> I17

I9:
T -> T / · F     * / + - $
F -> · ( E )     * / + - $
F -> · num     * / + - $
--- num --> I5
--- ( --> I4
--- F --> I18

I10:
E -> E · + T     ) + -
E -> E · - T     ) + -
F -> ( E · )     + / * - $
--- ) --> I21
--- - --> I20
--- + --> I19

I11:
T -> T · * F     * + / - )
T -> T · / F     * + / - )
E -> T ·     ) + -
--- / --> I23
--- * --> I22

I12:
T -> F ·     * + / - )

I13:
E -> · E + T     - + )
E -> · E - T     - + )
E -> · T     - + )
T -> · T * F     ) - + / *
T -> · T / F     ) - + / *
T -> · F     ) - + / *
F -> · ( E )     - + / * )
F -> ( · E )     ) * + / -
F -> · num     - + / * )
--- F --> I12
--- T --> I11
--- num --> I14
--- ( --> I13
--- E --> I24

I14:
F -> num ·     ) * + / -
```

```
I15:
T -> T · * F     * + / $ -
T -> T · / F     * + / $ -
E -> E + T ·     + $ -
--- / --> I9
--- * --> I8

I16:
T -> T · * F     * + / $ -
T -> T · / F     * + / $ -
E -> E - T ·     + $ -
--- / --> I9
--- * --> I8

I17:
T -> T * F ·     - $ / + *

I18:
T -> T / F ·     - $ / + *

I19:
E -> E + · T     + ) -
T -> · T * F     + / ) - *
T -> · T / F     + / ) - *
T -> · F     + / ) - *
F -> · ( E )     * / + ) -
F -> · num     * / + ) -
--- num --> I14
--- ( --> I13
--- F --> I12
--- T --> I25

I20:
E -> E - · T     + ) -
T -> · T * F     + / ) - *
T -> · T / F     + / ) - *
T -> · F     + / ) - *
F -> · ( E )     * / + ) -
F -> · num     * / + ) -
--- num --> I14
--- ( --> I13
--- F --> I12
--- T --> I26

I21:
F -> ( E ) ·     - $ * + /

I22:
T -> T * · F     * / + - )
F -> · ( E )     * / + - )
```

```
F -> · num     * / + - )
--- num --> I14
--- ( --> I13
--- F --> I27

I23:
T -> T / · F     * / + - )
F -> · ( E )     * / + - )
F -> · num     * / + - )
--- num --> I14
--- ( --> I13
--- F --> I28

I24:
E -> E · + T     ) + -
E -> E · - T     ) + -
F -> ( E · )     - + / * )
--- ) --> I29
--- - --> I20
--- + --> I19

I25:
T -> T · * F     * - ) + /
T -> T · / F     * - ) + /
E -> E + T ·     - + )
--- / --> I23
--- * --> I22

I26:
T -> T · * F     * - ) + /
T -> T · / F     * - ) + /
E -> E - T ·     - + )
--- / --> I23
--- * --> I22

I27:
T -> T * F ·     ) - / + *

I28:
T -> T / F ·     ) - / + *

I29:
F -> ( E ) ·     ) * + / -


[Parsing Table]
State 0
    Nonterminal    (
    Action         S 4
    Nonterminal    num
    Action         S 5
```

```
    Terminal       E
    Goto           1
    Terminal       T
    Goto           2
    Terminal       F
    Goto           3
State 1
    Nonterminal    +
    Action         S 6
    Nonterminal    $
    Action         ACC
    Nonterminal    -
    Action         S 7
State 2
    Nonterminal    $
    Action         R E -> T
    Nonterminal    -
    Action         R E -> T
    Nonterminal    *
    Action         S 8
    Nonterminal    +
    Action         R E -> T
    Nonterminal    /
    Action         S 9
State 3
    Nonterminal    -
    Action         R T -> F
    Nonterminal    $
    Action         R T -> F
    Nonterminal    /
    Action         R T -> F
    Nonterminal    +
    Action         R T -> F
    Nonterminal    *
    Action         R T -> F
State 4
    Nonterminal    (
    Action         S 13
    Nonterminal    num
    Action         S 14
    Terminal       E
    Goto           10
    Terminal       T
    Goto           11
    Terminal       F
    Goto           12
State 5
    Nonterminal    /
    Action         R F -> num
    Nonterminal    +
    Action         R F -> num
```

```
    Nonterminal    *
    Action         R F -> num
    Nonterminal    $
    Action         R F -> num
    Nonterminal    -
    Action         R F -> num
State 6
    Nonterminal    (
    Action         S 4
    Nonterminal    num
    Action         S 5
    Terminal       T
    Goto           15
    Terminal       F
    Goto           3
State 7
    Nonterminal    (
    Action         S 4
    Nonterminal    num
    Action         S 5
    Terminal       T
    Goto           16
    Terminal       F
    Goto           3
State 8
    Nonterminal    (
    Action         S 4
    Nonterminal    num
    Action         S 5
    Terminal       F
    Goto           17
State 9
    Nonterminal    (
    Action         S 4
    Nonterminal    num
    Action         S 5
    Terminal       F
    Goto           18
State 10
    Nonterminal    +
    Action         S 19
    Nonterminal    -
    Action         S 20
    Nonterminal    )
    Action         S 21
State 11
    Nonterminal    -
    Action         R E -> T
    Nonterminal    )
    Action         R E -> T
    Nonterminal    *
```

```
    Action          S 22
    Nonterminal     +
    Action          R E -> T
    Nonterminal     /
    Action          S 23
State 12
    Nonterminal     )
    Action          R T -> F
    Nonterminal     -
    Action          R T -> F
    Nonterminal     /
    Action          R T -> F
    Nonterminal     +
    Action          R T -> F
    Nonterminal     *
    Action          R T -> F
State 13
    Nonterminal     (
    Action          S 13
    Nonterminal     num
    Action          S 14
    Terminal        E
    Goto            24
    Terminal        T
    Goto            11
    Terminal        F
    Goto            12
State 14
    Nonterminal     -
    Action          R F -> num
    Nonterminal     /
    Action          R F -> num
    Nonterminal     +
    Action          R F -> num
    Nonterminal     *
    Action          R F -> num
    Nonterminal     )
    Action          R F -> num
State 15
    Nonterminal     -
    Action          R E -> E + T
    Nonterminal     $
    Action          R E -> E + T
    Nonterminal     *
    Action          S 8
    Nonterminal     +
    Action          R E -> E + T
    Nonterminal     /
    Action          S 9
State 16
    Nonterminal     -
```

```
    Action          R E -> E - T
    Nonterminal     $
    Action          R E -> E - T
    Nonterminal     *
    Action          S 8
    Nonterminal     +
    Action          R E -> E - T
    Nonterminal     /
    Action          S 9
State 17
    Nonterminal     *
    Action          R T -> T * F
    Nonterminal     +
    Action          R T -> T * F
    Nonterminal     /
    Action          R T -> T * F
    Nonterminal     $
    Action          R T -> T * F
    Nonterminal     -
    Action          R T -> T * F
State 18
    Nonterminal     *
    Action          R T -> T / F
    Nonterminal     +
    Action          R T -> T / F
    Nonterminal     /
    Action          R T -> T / F
    Nonterminal     $
    Action          R T -> T / F
    Nonterminal     -
    Action          R T -> T / F
State 19
    Nonterminal     (
    Action          S 13
    Nonterminal     num
    Action          S 14
    Terminal        T
    Goto            25
    Terminal        F
    Goto            12
State 20
    Nonterminal     (
    Action          S 13
    Nonterminal     num
    Action          S 14
    Terminal        T
    Goto            26
    Terminal        F
    Goto            12
State 21
    Nonterminal     /
```

```
    Action          R F -> ( E )
    Nonterminal     +
    Action          R F -> ( E )
    Nonterminal     *
    Action          R F -> ( E )
    Nonterminal     $
    Action          R F -> ( E )
    Nonterminal     -
    Action          R F -> ( E )
State 22
    Nonterminal     (
    Action          S 13
    Nonterminal     num
    Action          S 14
    Terminal        F
    Goto            27
State 23
    Nonterminal     (
    Action          S 13
    Nonterminal     num
    Action          S 14
    Terminal        F
    Goto            28
State 24
    Nonterminal     +
    Action          S 19
    Nonterminal     -
    Action          S 20
    Nonterminal     )
    Action          S 29
State 25
    Nonterminal     )
    Action          R E -> E + T
    Nonterminal     -
    Action          R E -> E + T
    Nonterminal     *
    Action          S 22
    Nonterminal     +
    Action          R E -> E + T
    Nonterminal     /
    Action          S 23
State 26
    Nonterminal     )
    Action          R E -> E - T
    Nonterminal     -
    Action          R E -> E - T
    Nonterminal     *
    Action          S 22
    Nonterminal     +
    Action          R E -> E - T
    Nonterminal     /
```

```
    Action          S 23
State 27
    Nonterminal     *
    Action          R T -> T * F
    Nonterminal     +
    Action          R T -> T * F
    Nonterminal     /
    Action          R T -> T * F
    Nonterminal     -
    Action          R T -> T * F
    Nonterminal     )
    Action          R T -> T * F
State 28
    Nonterminal     *
    Action          R T -> T / F
    Nonterminal     +
    Action          R T -> T / F
    Nonterminal     /
    Action          R T -> T / F
    Nonterminal     -
    Action          R T -> T / F
    Nonterminal     )
    Action          R T -> T / F
State 29
    Nonterminal     -
    Action          R F -> ( E )
    Nonterminal     /
    Action          R F -> ( E )
    Nonterminal     +
    Action          R F -> ( E )
    Nonterminal     *
    Action          R F -> ( E )
    Nonterminal     )
    Action          R F -> ( E )

Stack
0
$
Input           num + num $
Output          Shift 5

Stack
0   5
$   num
Input           + num $
Output          Reduce by F -> num

Stack
0   3
$   F
Input           + num $
```

```
Output          Reduce by T -> F

Stack
0   2
$   T
Input           + num $
Output          Reduce by E -> T

Stack
0   1
$   E
Input           + num $
Output          Shift 6

Stack
0   1   6
$   E   +
Input           num $
Output          Shift 5

Stack
0   1   6   5
$   E   +   num
Input           $
Output          Reduce by F -> num

Stack
0   1   6   3
$   E   +   F
Input           $
Output          Reduce by T -> F

Stack
0   1   6   15
$   E   +   T
Input           $
Output          Reduce by E -> E + T

Stack
0   1
$   E
Input           $
Output          Accepted!

Parsing successful!
```

4.1.3 分析

程序首先输出原文法，之后输出拓广文法、识别所有活前缀的 DFA、预测分析表。最后程序给出了分析字符串的过程。

4.2 测试集 2

此测试集测试一个复杂的算术表达式。

4.2.1 输入

文法如2.2.1节所示。

```
( num * ( ( ( ( num + num ) * num ) + num / num ) - num * ( num + num ) ) /
↪  num )
```

4.2.2 输出

由于输出内容过长，不在此展开，请参考输出文件。

4.2.3 分析

正确分析了此字符串。

4.3 测试集 3

此测试集用于测试一个错误的算术表达式。

4.3.1 输入

文法如2.2.1节所示。

```
( num + num ) * / num
```

4.3.2 输出

由于输出内容过长，不在此展开，请参考输出文件。

4.3.3 分析

归约到 `T*` 时，无法再进行归约，分析程序报错并退出。

4.4 测试集 4

此测试集用于测试一个错误的算术表达式。

4.4.1 输入

文法如2.2.1节所示。

```
( ( num + num ) / num
```

4.4.2 输出

由于输出内容过长，不在此展开，请参考输出文件。

4.4.3 分析

归约到 (T/num 时，无法再进行归约，分析程序报错并退出。

4.5 测试集 5

此测试集依照习题 4.10 修改而来。

4.5.1 输入

```
$ Nonterminal symbols
E L
$ Terminal symbols
a , ( )
$ Start symbol
E
$ Productions
E -> ( L ) $ a
L -> L , E $ E
```

```
( ( a ) , a , ( a , a ) )
```

4.5.2 输出

由于输出内容过长，不在此展开，请参考输出文件。

4.6 测试集 6

此测试集依照习题 4.18 修改而来。

4.6.1 输入

```
$ Nonterminal symbols
S A B
$ Terminal symbols
a b
$ Start symbol
S
$ Productions
S -> A
A -> B A $ [empty]
B -> a B $ b
```

```
a b a b
```

4.6.2 输出

由于输出内容过长，不在此展开，请参考输出文件。

5 实验总结

本次实验中我编写了一个 LR(1) 语法分析程序，使我对自底向上语法分析的流程更加清楚，对相关知识点的掌握更加牢固。

此程序的架构比较简单，主要难点在数据结构的设计和算法的实现方面，尤其是对于 LR(1) 项目和项目集，表示方式需要既准确又要便于计算，实现的难度较大。在实现期间，我使用了 C++11 的语法特性，使得算法过程更加清晰，大大减少编程复杂度。

在成功实现这些算法并且通过测试之后，我尝试编写了伪代码，这将会为我未来解题带来很大帮助。

此外，我的语法分析程序仍然存在许多待改进的地方，比如对于对于错误的处理不够全面、在求解 DFA 时的鲁棒性仍然有待提高，由于时间所限，这些情况我无法一一考虑周全。

本次实验除了让我对课内知识有了更多的认识，也使我的 C++ 编程能力得到提高，我从中收获颇丰。