编译原理 实验三 LR(1) 语法分析器

中国人民大学 信息学院 崔冠宇 2018202147

1 实验内容

设计一个 C-- 语言的语法分析器。它的输入输出要求如下：

1. 输入：C-- 语言源文件 `test.cmm`、LR(1) 文法文件 `LR-grammar.txt`。
2. 输出：解析文法得到的 LR(1) 动作表、调用词法分析器的 LR(1) 分析过程中栈的变化以及每一步的动作，输出到 `out.txt`。

2 程序设计原理与方法

如下图所示，**语法分析** (syntax analysis) 是编译过程的第二个主要阶段，这一阶段在词法分析的基础上产生分析树 (parse tree)，为后面的**语义分析** (sementic analysis) 部分提供了支持。本次实验中的自下而上的**语法分析器** (syntax analyzer, parser) 接受词法分析器给出的切分好的单词流，利用上下文无关文法的特例——LR(1) 文法进行语法分析，判定输入是否符合语言的语法。在设计语法分析器之前，我们需要先设计 C-- 语言的语法。

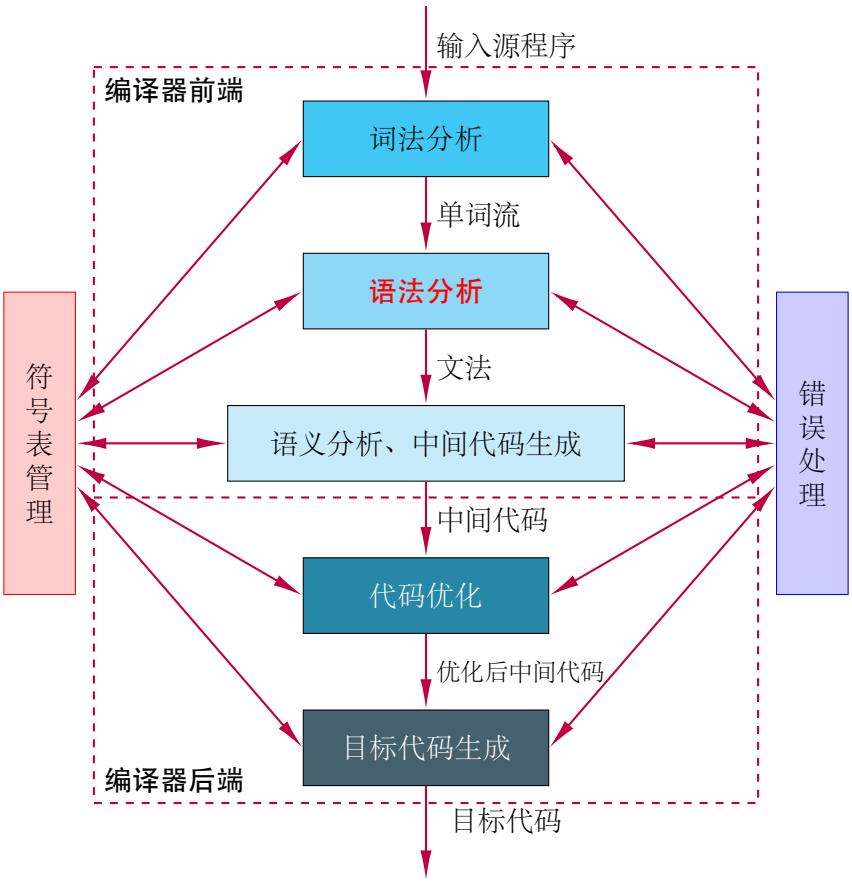

图 1: 编译器结构图

2.1 C-- 语言的语法

C-- 语言的语法以 C 语言的语法为蓝本，参考 C 标准文件 [1] 中 **附录 A.2** 节 **Phrase structure grammar** 修改而成。按语法结构自顶向下的顺序，C-- 语言的语法分为下列几个层次：

1. 外部定义 (External definition)：C-- 语言源文件的顶层结构，由各类声明和定义组成；
2. 语句 (Statements)：C-- 语言保持结构的部分，比如分支、循环等；
3. 声明 (Declarations)：C-- 语言的一类特殊语句，用来定义变量、函数等；
4. 表达式 (Expressions)：C-- 语言的最基本的语法单元，可以求值；

由于语法规则较多，详细的 C-- 语法规则文件放在最后的附录部分。

2.2 LR(1) 分析原理

LR(1) 文法的全称是“自左向右 (left to right)、最右推导的逆过程 (rightmost derivation inversion)、前向观察一符号 (1 token)”的文法，是一种只需要看一个符号进行自下而上分析的确定性的上下文无关文法，满足 LR(1) 文法的语言可以方便地使用栈进行自下而上的语法分析。根据 LR(1) 文法生成语法解析器主要有以下几个步骤：

1. 读入文法，并保存到合适的数据结构；
2. 构造 LR(1) 项目集，计算识别这些项目集的 DFA；
3. 根据 DFA，填写 LR(1) 预测分析表，同时提示用户处理语法中不属于 LR(1) 部分的冲突；
4. 根据分析表，按照 LR(1) 分析方法进行语法分析和错误处理。

3 程序设计流程

3.1 LR(1) 语法分析算法设计

首先给出 LR(1) 分析法核心函数 `LRparse()` 的伪代码：

Algorithm 1 LR(1) 分析的核心算法 `LRparse()`

```
// 先计算动作表
calcLRParseTable()
// 初始化
stateStack.push(0)
symbolStack.push(#)
// 指向第一个词
token = getNextToken()
while True do
    if 该词有词法错误 then
        词法分析器错误处理
    else
        根据状态栈顶和当前 token 查表
        if 没查到 then
            出错处理
        else
            // 分几类情况
            if 是 then 接受项目 ACCEPT
                接受
            else if 是移进项目 SHIFT i then
                stateStack.push(i)
                symbolStack.push(token)
            else if 是归约项目 REDUCE A → B_1 B_2 ··· B_k then
                stateStack 弹栈 k 次
                symbolStack 弹栈 k 次
                // 接下来还要 GOTO
                根据当前栈顶和产生式左部 A 查表
                if 没查到或查到的不是 GOTO then
                    出错处理
                else
                    将查到的新状态压入 stateStack
                    将 A 压入 symbolStack
                end if
            end if
        end if
    end if
end while
```

下面给出各子函数的伪代码：

1. 读入文法，并保存到合适的数据结构的代码略去；

2. 构造预测分析表的函数 `calcLRParseTable()`：

Algorithm 2 计算预测分析表 `calcLRParseTable()`

```
// 先计算 FIRST 和 FOLLOW 表
calcFirstTable()
calcFollowTable()
初始项目 term0 为 [S' -> S, #]
// 计算初始项目集
startNode ← closure(term0)
// 初始节点入队
q.enqueue(startNode)
while 队列不空 do
    node ← q.dequeue()
    for 节点中的每一个项目 do
        if 本项目是规约项目 then
        else
            // 本项目是移进项目
            计算点往后移动的下一个项目
            出边表（根据点后面移进的字符分类，即 (下一个符号, 得到的若干新项目)）outEdge 中增加一条出边
        end if
    end for
    for 出边表中按边上符号分类的每一个项目集 do
        // 新项目集闭包后入队
        q.enqueue(closure(项目集))
        根据边上的符号是终结符（需要移进）还是非终结符（需要归约）填写动作表
    end for
end while
```

3. 构造 FIRST 表的函数 `calcFirstTable()` 以及构造 FOLLOW 表的函数 `calcFollowTable()` 与之前 LL(1) 分析的相同。

4. 计算 LR(1) 项目集闭包的函数 `closure()`：

Algorithm 3 计算 LR(1) 项目集闭包 `closure(terms)`

```
// 首先本身属于闭包
cl = terms
while cl 还能扩大 do
    for terms 中的每一个项目 term do
        if 下一个符号是非终结符 B then
            // 即 term == [ A -> α·Bβ, a ]
            for 所有形如 B -> r 的产生式，以及所有 b ∈ FIRST(βa) do
                将项目 [ B -> · r, b] 加入 cl
            end for
        end if
    end for
end while
return cl
```

4 程序清单

由于代码量很大，为了保持行文思路的连续性，程序清单挪至文末附录处。

5 运行结果

首次运行语法分析器 `parser`，程序以 `grammar.txt` 语法文件作为语法规则计算 LR(1) 分析表。在根据 C-- 语言的语法规则计算 LR(1) 分析表的过程中，遇到了 if-else 的移进-规约冲突，程序将交给用户手动解决，如下图所示：

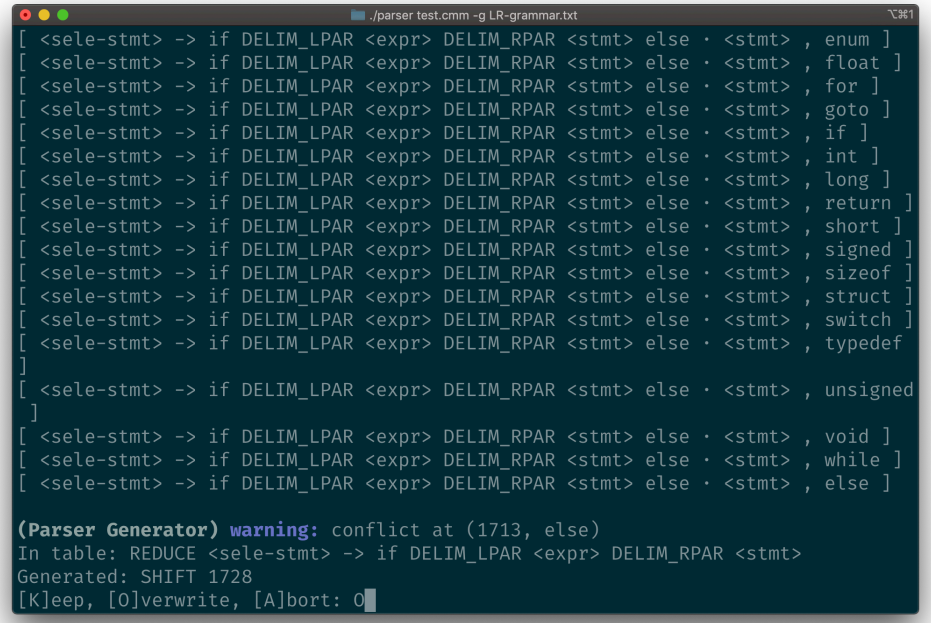

图 2: 冲突的手动解决

根据 C-- 语言的语义规定，else 应该与最近的 if 配对，所以这里应该选择移进，即覆盖表中内容。冲突处理完成后，语法分析器利用产生的分析表 `LR.tbl` 去分析源文件 `test.cmm`，输出结果如下图所示。

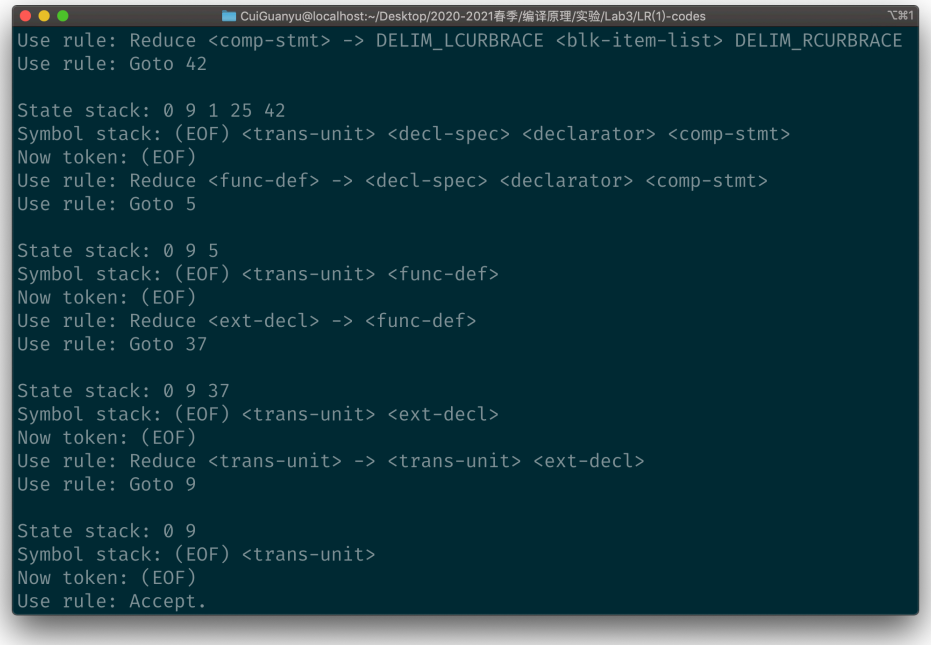

图 3: 输出的分析过程

可见程序运行正确。

6 程序使用说明

语法分析器的使用命令为

```
$ ./parser <filename> [options]
```

其中 `filename` 是文件名，`options` 是附加命令，主要有下列几种：

- `-h, --help` 打印帮助信息；
- `-LL, --LL(1)` 设置分析方式为 LL(1)（默认使用 LR(1) 分析）；
- `-g, --grammar <filename>` 设置输入的语法文件文件名（默认使用内置四则运算语法）；
- `-ti, --table-in <filename>` 设置使用的分析表的文件名；
- `-to, --table-out <filename>` 设置输出的分析表文件名（LL(1) 模式下默认为 `LL.tbl`，LR(1) 模式下默认为 `LR.tbl`）；
- `-o, --output <filename>` 设置输出的分析过程文件名（默认使用 `out.txt`）。

7 总结与完善

7.1 亮点

本程序主要有以下亮点：

1. 支持的语法较为齐全，能够对几乎标准的 C 语言进行语法分析；
2. 支持根据用户输入语法文件进行分析，判断是否为 LR(1) 文法，并且自动生成分析表，减少硬编码的工作；
3. 支持保存分析表供后续分析使用，可以避免每次分析前花费大量时间产生分析表；
4. 对于不是 LR(1) 文法的少部分冲突，可以交给用户手动解决；
5. 支持语法错误提示，提示内容包括错误所在的行列位置，方便用户改正。

7.2 不足之处

语法分析器生成器 (Parser Generator) 无法根据语法文件自动生成错误时的同步信息，导致语法分析无法找到尽可能多的语法错误。

A 语法规则文件

1. C-- 语言的语法规则文件 **grammar.txt**:

```
// -------------------- 外部定义 --------------------
// A.2.4 External definitions
// (6.9)
// translation-unit:
//      external-declaration
//      translation-unit external-declaration
<trans-unit> -> <ext-decl>
<trans-unit> -> <trans-unit> <ext-decl>

// (6.9)
// external-declaration:
//      function-definition
//      declaration
<ext-decl> -> <func-def>
<ext-decl> -> <decl>

// (6.9.1)
// function-definition:
//      declaration-specifiers declarator declaration-list_opt compound-statement
<func-def> -> <decl-spec> <declarator> <decl-list> <comp-stmt>
<func-def> -> <decl-spec> <declarator> <comp-stmt>

// (6.9.1)
// declaration-list:
//      declaration
//      declaration-list declaration
<decl-list> -> <decl>
<decl-list> -> <decl-list> <decl>

// -------------------- 语句 --------------------
// A.2.3 Statements

// (6.8)
// statement:
```

```
35 //      labeled-statement
36 //      compound-statement
37 //      expression-statement
38 //      selection-statement
39 //      iteration-statement
40 //      jump-statement
41 <stmt> -> <labeled-stmt>
42 <stmt> -> <comp-stmt>
43 <stmt> -> <expr-stmt>
44 <stmt> -> <sele-stmt>
45 <stmt> -> <iter-stmt>
46 <stmt> -> <jump-stmt>
47
48 // (6.8.1)
49 // labeled-statement:
50 //      identifier : statement
51 //      case constant-expression : statement
52 //      default : statement
53 <labeled-stmt> -> IDENTIFIER DELIM_COLON <stmt>
54 <labeled-stmt> -> case <const-expr> DELIM_COLON <stmt>
55 <labeled-stmt> -> default DELIM_COLON <stmt>
56
57 // (6.8.2)
58 // compound-statement:
59 //      { block-item-list_opt }
60 <comp-stmt> -> DELIM_LCURBRACE <blk-item-list> DELIM_RCURBRACE
61 <comp-stmt> -> DELIM_LCURBRACE DELIM_RCURBRACE
62
63 // (6.8.2)
64 // block-item-list:
65 //      block-item
66 //      block-item-list block-item
67 <blk-item-list> -> <blk-item>
68 <blk-item-list> -> <blk-item-list> <blk-item>
69
70 // (6.8.2)
71 // block-item:
```

```
72 //      declaration
73 //      statement
74 <blk-item> -> <decl>
75 <blk-item> -> <stmt>
76
77 // (6.8.3)
78 // expression-statement:
79 //      expression_opt ;
80 <expr-stmt> -> <expr> DELIM_SEMICOLON
81 <expr-stmt> -> DELIM_SEMICOLON
82
83 // (6.8.4)
84 // selection-statement:
85 //      if ( expression ) statement
86 //      if ( expression ) statement else statement
87 //      switch ( expression ) statement
88 // 此处有冲突，但可以手工解决
89 <sele-stmt> -> if DELIM_LPAR <expr> DELIM_RPAR <stmt>
90 <sele-stmt> -> if DELIM_LPAR <expr> DELIM_RPAR <stmt> else <stmt>
91 <sele-stmt> -> switch DELIM_LPAR <expr> DELIM_RPAR <stmt>
92
93 // (6.8.5)
94 // iteration-statement:
95 //      while ( expression ) statement
96 //      do statement while ( expression ) ;
97 //      for ( expression_opt ; expression_opt ; expression_opt ) statement
98 //      for ( declaration expression_opt ; expression_opt ) statement
99 <iter-stmt> -> while DELIM_LPAR <expr> DELIM_RPAR <stmt>
100 <iter-stmt> -> do <stmt> while DELIM_LPAR <expr> DELIM_RPAR DELIM_SEMICOLON
101 <iter-stmt> -> for DELIM_LPAR <for-cond> DELIM_RPAR <stmt>
102 <for-cond> -> <expr> DELIM_SEMICOLON <expr> DELIM_SEMICOLON <expr>
103 <for-cond> -> DELIM_SEMICOLON <expr> DELIM_SEMICOLON <expr>
104 <for-cond> -> <expr> DELIM_SEMICOLON DELIM_SEMICOLON <expr>
105 <for-cond> -> <expr> DELIM_SEMICOLON <expr> DELIM_SEMICOLON
106 <for-cond> -> DELIM_SEMICOLON DELIM_SEMICOLON <expr>
107 <for-cond> -> DELIM_SEMICOLON <expr> DELIM_SEMICOLON
108 <for-cond> -> <expr> DELIM_SEMICOLON DELIM_SEMICOLON
```

```
<for-cond> -> DELIM_SEMICOLON DELIM_SEMICOLON
<for-cond> -> <decl> <expr> DELIM_SEMICOLON <expr>
<for-cond> -> <decl> DELIM_SEMICOLON <expr>
<for-cond> -> <decl> <expr> DELIM_SEMICOLON
<for-cond> -> <decl> DELIM_SEMICOLON

// (6.8.6)
// jump-statement:
//      goto identifier ;
//      continue ;
//      break ;
//      return expression_opt ;
<jump-stmt> -> goto DELIM_SEMICOLON
<jump-stmt> -> continue DELIM_SEMICOLON
<jump-stmt> -> break DELIM_SEMICOLON
<jump-stmt> -> return <expr> DELIM_SEMICOLON
<jump-stmt> -> return DELIM_SEMICOLON

// // -------------------- 声明 --------------------
// A.2.2 Declarations

// (6.7)
// declaration:
//      declaration-specifiers init-declarator-list_opt ;
//      static_assert-declaration (不用)
<decl> -> <decl-spec> <init-declarator-list> DELIM_SEMICOLON
<decl> -> <decl-spec> DELIM_SEMICOLON

// (6.7)
// declaration-specifiers:
//      storage-class-specifier declaration-specifiers_opt
//      type-specifier declaration-specifiers_opt
//      type-qualifier declaration-specifiers_opt
//      function-specifier declaration-specifiers_opt (不用)
//      alignment-specifier declaration-specifiers_opt (不用)
<decl-spec> -> <storage-class-spec> <decl-spec>
<decl-spec> -> <storage-class-spec>
```

```
<decl-spec> -> <type-spec> <decl-spec>
<decl-spec> -> <type-spec>
<decl-spec> -> <type-qual> <decl-spec>
<decl-spec> -> <type-qual>

// (6.7)
// init-declarator-list:
//      init-declarator
//      init-declarator-list , init-declarator
<init-declarator-list> -> <init-declarator>
<init-declarator-list> -> <init-declarator-list> DELIM_COMMA <init-declarator>

// (6.7)
// init-declarator:
//      declarator
//      declarator = initializer
<init-declarator> -> <declarator>
<init-declarator> -> <declarator> OP_ASN <initializer>

// (6.7.1)
// storage-class-specifier:
//      typedef
//      extern (不用)
//      static (不用)
//      _Thread_local (不用)
//      auto (不用)
//      register (不用)
<storage-class-spec> -> typedef

// (6.7.2)
// type-specifier:
//      void
//      char
//      short
//      int
//      long
//      float
```

```
183 //      double
184 //      signed
185 //      unsigned
186 //      _Bool (不用)
187 //      _Complex (不用)
188 //      atomic-type-specifier (不用)
189 //      struct-or-union-specifier
190 //      enum-specifier
191 //      typedef-name (冲突, 不用)
192 <type-spec> -> void
193 <type-spec> -> char
194 <type-spec> -> short
195 <type-spec> -> int
196 <type-spec> -> long
197 <type-spec> -> float
198 <type-spec> -> double
199 <type-spec> -> signed
200 <type-spec> -> unsigned
201 <type-spec> -> <struct-union-spec>
202 <type-spec> -> <enum-spec>
203 // <type-spec> -> <typedef-name>
204
205 // (6.7.2.1)
206 // struct-or-union-specifier:
207 //      struct-or-union identifier_opt { struct-declaration-list }
208 //      struct-or-union identifier
209 <struct-union-spec> -> <struct-union> IDENTIFIER DELIM_LCURBRACE <struct-decl-list>
        DELIM_RCURBRACE
210 <struct-union-spec> -> <struct-union> DELIM_LCURBRACE <struct-decl-list>
      DELIM_RCURBRACE
211 <struct-union-spec> -> <struct-union> IDENTIFIER
212
213 // (6.7.2.1)
214 // struct-or-union:
215 //      struct
216 //      union (不用)
217 <struct-union> -> struct
```

```
218
219 // (6.7.2.1)
220 // struct-declaration-list:
221 //      struct-declaration
222 //      struct-declaration-list struct-declaration
223 <struct-decl-list> -> <struct-decl>
224 <struct-decl-list> -> <struct-decl-list> <struct-decl>
225
226 // (6.7.2.1)
227 // struct-declaration:
228 //      specifier-qualifier-list struct-declarator-list_opt ;
229 //      static_assert-declaration (不用)
230 <struct-decl> -> <spec-qual-list> <struct-declarator-list> DELIM_SEMICOLON
231 <struct-decl> -> <spec-qual-list> DELIM_SEMICOLON
232
233 // (6.7.2.1)
234 // specifier-qualifier-list:
235 //      type-specifier specifier-qualifier-list_opt
236 //      type-qualifier specifier-qualifier-list_opt
237 <spec-qual-list> -> <type-spec> <spec-qual-list>
238 <spec-qual-list> -> <type-spec>
239 <spec-qual-list> -> <type-qual> <spec-qual-list>
240 <spec-qual-list> -> <type-qual>
241
242 // (6.7.2.1)
243 // struct-declarator-list:
244 //      struct-declarator
245 //      struct-declarator-list , struct-declarator
246 <struct-declarator-list> -> <struct-declarator>
247 <struct-declarator-list> -> <struct-declarator-list> DELIM_COMMA <struct-declarator
    >
248
249 // (6.7.2.1)
250 // struct-declarator:
251 //      declarator
252 //      declarator_opt : constant-expression (不用)
253 <struct-declarator> -> <declarator>
```

```
254
255 // (6.7.2.2)
256 // enum-specifier:
257 //       enum identifier_opt { enumerator-list }
258 //       enum identifier_opt { enumerator-list , }
259 //       enum identifier
260 <enum-spec> -> enum IDENTIFIER DELIM_LCURBRACE <enumerator-list> DELIM_RCURBRACE
261 <enum-spec> -> enum DELIM_LCURBRACE <enumerator-list> DELIM_RCURBRACE
262 <enum-spec> -> enum IDENTIFIER DELIM_LCURBRACE <enumerator-list> DELIM_COMMA
       DELIM_RCURBRACE
263 <enum-spec> -> enum DELIM_LCURBRACE <enumerator-list> DELIM_COMMA DELIM_RCURBRACE
264 <enum-spec> -> enum IDENTIFIER
265
266 // (6.7.2.2)
267 // enumerator-list:
268 //       enumerator
269 //       enumerator-list , enumerator
270 <enumerator-list> -> <enumerator>
271 <enumerator-list> -> <enumerator-list> DELIM_COMMA <enumerator>
272
273 // (6.7.2.2)
274 // enumerator:
275 //       enumeration-constant
276 //       enumeration-constant = constant-expression
277 <enumerator> -> <enumeration-const>
278 <enumerator> -> <enumeration-const> OP_ASN <const-expr>
279 <enumeration-const> -> IDENTIFIER
280
281
282 // (6.7.2.4) (不用)
283
284 // (6.7.3)
285 // type-qualifier:
286 //       const
287 //       restrict (不用)
288 //       volatile (不用)
289 //       _Atomic (不用)
```

```
<type-qual> -> const

// (6.7.4) - (6.7.5) (不用)

// (6.7.6)
// declarator:
//      pointer_opt direct-declarator
<declarator> -> <pointer> <direct-declarator>
<declarator> -> <direct-declarator>

// (6.7.6)
// direct-declarator:
//      identifier
//      ( declarator )
//      direct-declarator [ type-qualifier-list_opt assignment-expression_opt ]
//      direct-declarator [ static type-qualifier-list_opt assignment-expression ] (不用)
//      direct-declarator [ type-qualifier-list static assignment-expression ] (不用)
//      direct-declarator [ type-qualifier-list_opt * ] (不用)
//      direct-declarator ( parameter-type-list )
//      direct-declarator ( identifier-list_opt )
<direct-declarator> -> IDENTIFIER
<direct-declarator> -> DELIM_LPAR <declarator> DELIM_RPAR
<direct-declarator> -> <direct-declarator> DELIM_LSQBRACKET <type-qual-list> <asn-expr> DELIM_RSQBRACKET
<direct-declarator> -> <direct-declarator> DELIM_LSQBRACKET <type-qual-list> DELIM_RSQBRACKET
<direct-declarator> -> <direct-declarator> DELIM_LSQBRACKET <asn-expr> DELIM_RSQBRACKET
<direct-declarator> -> <direct-declarator> DELIM_LSQBRACKET DELIM_RSQBRACKET
<direct-declarator> -> <direct-declarator> DELIM_LPAR <param-type-list> DELIM_RPAR
<direct-declarator> -> <direct-declarator> DELIM_LPAR <identifier-list> DELIM_RPAR
<direct-declarator> -> <direct-declarator> DELIM_LPAR DELIM_RPAR

// (6.7.6)
// pointer:
```

```
322 //      * type-qualifier-list_opt
323 //      * type-qualifier-list_opt pointer
324 <pointer> -> OP_MUL <type-qual-list>
325 <pointer> -> OP_MUL
326 <pointer> -> OP_MUL <type-qual-list> <pointer>
327 <pointer> -> OP_MUL <pointer>
328
329 // (6.7.6)
330 // type-qualifier-list:
331 //      type-qualifier
332 //      type-qualifier-list type-qualifier
333 <type-qual-list> -> <type-qual>
334 <type-qual-list> -> <type-qual-list> <type-qual>
335
336 // (6.7.6)
337 // parameter-type-list:
338 //      parameter-list
339 //      parameter-list , ... (不用)
340 <param-type-list> -> <param-list>
341
342 // (6.7.6)
343 // parameter-list:
344 //      parameter-declaration
345 //      parameter-list , parameter-declaration
346 <param-list> -> <param-decl>
347 <param-list> -> <param-list> DELIM_COMMA <param-decl>
348
349 // (6.7.6)
350 // parameter-declaration:
351 //      declaration-specifiers declarator
352 //      declaration-specifiers abstract-declarator_opt
353 <param-decl> -> <decl-spec> <declarator>
354 <param-decl> -> <decl-spec> <abstract-declarator>
355 <param-decl> -> <decl-spec>
356
357 // (6.7.6)
358 // identifier-list:
```

```
359 //      identifier
360 //      identifier-list , identifier
361 <identifier-list> -> IDENTIFIER
362 <identifier-list> -> <identifier-list> DELIM_COMMA IDENTIFIER
363
364 // (6.7.7)
365 // type-name:
366 //      specifier-qualifier-list abstract-declarator_opt
367 <type-name> -> <spec-qual-list> <abstract-declarator>
368 <type-name> -> <spec-qual-list>
369
370 // (6.7.7)
371 // abstract-declarator:
372 //      pointer
373 //      pointer_opt direct-abstract-declarator
374 <abstract-declarator> -> <pointer>
375 <abstract-declarator> -> <pointer> <direct-abstract-declarator>
376 <abstract-declarator> -> <direct-abstract-declarator>
377
378 // (6.7.7)
379 // direct-abstract-declarator:
380 //      ( abstract-declarator )
381 //      direct-abstract-declarator_opt [ type-qualifier-list_opt assignment-
    expression_opt ]
382 //      direct-abstract-declarator_opt [ static type-qualifier-list_opt assignment-
    expression ] (不用)
383 //      direct-abstract-declarator_opt [ type-qualifier-list static assignment-
    expression ] (不用)
384 //      direct-abstract-declarator_opt [ * ] direct-abstract-declarator_opt (
    parameter-type-list_opt ) (不用)
385 <direct-abstract-declarator> -> DELIM_LPAR <abstract-declarator> DELIM_RPAR
386 <direct-abstract-declarator> -> DELIM_LSQBRACKET <type-qual-list> <asn-expr>
    DELIM_RSQBRACKET
387 <direct-abstract-declarator> -> DELIM_LSQBRACKET <type-qual-list> DELIM_RSQBRACKET
388 <direct-abstract-declarator> -> DELIM_LSQBRACKET <asn-expr> DELIM_RSQBRACKET
389 <direct-abstract-declarator> -> DELIM_LSQBRACKET DELIM_RSQBRACKET
390
```

```
391 // (6.7.8)
392 // typedef-name: (冲突, 不用)
393 //      identifier
394 // <typedef-name> -> IDENTIFIER
395
396 // (6.7.9)
397 // initializer:
398 //      assignment-expression
399 //      { initializer-list }
400 //      { initializer-list , }
401 <initializer> -> <asn-expr>
402 <initializer> -> DELIM_LCURBRACE <initializer-list> DELIM_RCURBRACE
403 <initializer> -> DELIM_LCURBRACE <initializer-list> DELIM_COMMA DELIM_RCURBRACE
404
405 // (6.7.9)
406 // initializer-list
407 //      designation_opt initializer
408 //      initializer-list , designation_opt initializer
409 <initializer-list> -> <designation> <initializer>
410 <initializer-list> -> <initializer>
411 <initializer-list> -> <initializer-list> DELIM_COMMA <designation> <initializer>
412 <initializer-list> -> <initializer-list> DELIM_COMMA <initializer>
413
414 // (6.7.9)
415 // designation:
416 //      designator-list =
417 <designation> -> <designator-list> OP_ASN
418
419 // (6.7.9)
420 // designator-list:
421 //      designator
422 //      designator-list designator
423 <designator-list> -> <designator>
424 <designator-list> -> <designator-list> <designator>
425
426 // (6.7.9)
427 // designator:
```

```
428 //      [ constant-expression ]
429 //      . identifier
430 <designator> -> DELIM_LSQBRACKET <const-expr> DELIM_RSQBRACKET
431 <designator> -> OP_DOT IDENTIFIER
432
433 // (6.7.10) (不用)
434
435 // -------------------- 表达式 --------------------
436 // A.2.1 Expressions
437 // (6.5.17)
438 // expression:
439 //      assignment-expression
440 //      expression , assignment-expression
441 <expr> -> <asn-expr>
442 <expr> -> <expr> DELIM_COMMA <asn-expr>
443
444 // (6.5.16)
445 // assignment-expression:
446 //      conditional-expression
447 //      unary-expression assignment-operator assignment-expression
448 <asn-expr> -> <cond-expr>
449 <asn-expr> -> <unary-expr> <asn-op> <asn-expr>
450
451 // (6.5.16)
452 // assignment-operator:
453 //      = *= /= %= += -= <<= >>= &= ^= |=
454 <asn-op> -> OP_ASN
455 <asn-op> -> OP_MULASN
456 <asn-op> -> OP_DIVASN
457 <asn-op> -> OP_MODASN
458 <asn-op> -> OP_ADDASN
459 <asn-op> -> OP_SUBASN
460 <asn-op> -> OP_SHLASN
461 <asn-op> -> OP_SHRASN
462 <asn-op> -> OP_ANDASN
463 <asn-op> -> OP_XORASN
464 <asn-op> -> OP_ORASN
```

```
465
466 // (6.6)
467 // constant-expression:
468 //      conditional-expression
469 <const-expr> -> <cond-expr>
470
471 // (6.5.15)
472 // conditional-expression:
473 //      logical-OR-expression
474 //      logical-OR-expression ? expression : conditional-expression
475 <cond-expr> -> <lor-expr>
476 <cond-expr> -> <lor-expr> DELIM_QUESTION <expr> DELIM_COLON <cond-expr>
477
478 // (6.5.14)
479 // logical-OR-expression:
480 //      logical-AND-expression
481 //      logical-OR-expression || logical-AND-expression
482 <lor-expr> -> <land-expr>
483 <lor-expr> -> <lor-expr> OP_LOR <land-expr>
484
485 // (6.5.13)
486 // logical-AND-expression:
487 //      inclusive-OR-expression
488 //      logical-AND-expression && inclusive-OR-expression
489 <land-expr> -> <inc-or-expr>
490 <land-expr> -> <land-expr> OP_LAND <inc-or-expr>
491
492 // (6.5.12)
493 // inclusive-OR-expression:
494 //      exclusive-OR-expression
495 //      inclusive-OR-expression | exclusive-OR-expression
496 <inc-or-expr> -> <exc-or-expr>
497 <inc-or-expr> -> <inc-or-expr> OP_OR <exc-or-expr>
498
499 // (6.5.11)
500 // exclusive-OR-expression:
501 //      AND-expression
```

```
//      exclusive-OR-expression ^ AND-expression
<exc-or-expr> -> <and-expr>
<exc-or-expr> -> <exc-or-expr> OP_XOR <and-expr>

// (6.5.10)
// AND-expression:
//      equality-expression
//      AND-expression & equality-expression
<and-expr> -> <eq-expr>
<and-expr> -> <and-expr> OP_AND <eq-expr>

// (6.5.9)
// equality-expression:
//      relational-expression
//      equality-expression == relational-expression
//      equality-expression != relational-expression
<eq-expr> -> <rel-expr>
<eq-expr> -> <eq-expr> OP_EQ <rel-expr>
<eq-expr> -> <eq-expr> OP_NEQ <rel-expr>

// (6.5.8)
// relational-expression:
//      shift-expression
//      relational-expression < shift-expression
//      relational-expression > shift-expression
//      relational-expression <= shift-expression
//      relational-expression >= shift-expression
<rel-expr> -> <shift-expr>
<rel-expr> -> <rel-expr> OP_LT <shift-expr>
<rel-expr> -> <rel-expr> OP_GT <shift-expr>
<rel-expr> -> <rel-expr> OP_LE <shift-expr>
<rel-expr> -> <rel-expr> OP_GE <shift-expr>

// (6.5.7)
// shift-expression:
//      additive-expression
//      shift-expression << additive-expression
```

```
539 //      shift-expression >> additive-expression
540 <shift-expr> -> <additive-expr>
541 <shift-expr> -> <shift-expr> OP_SHL <additive-expr>
542 <shift-expr> -> <shift-expr> OP_SHR <additive-expr>
543
544 // (6.5.6)
545 // additive-expression:
546 //      multiplicative-expression
547 //      additive-expression + multiplicative-expression
548 //      additive-expression - multiplicative-expression
549 <additive-expr> -> <multiplicative-expr>
550 <additive-expr> -> <additive-expr> OP_ADD <multiplicative-expr>
551 <additive-expr> -> <additive-expr> OP_SUB <multiplicative-expr>
552
553 // (6.5.5)
554 // multiplicative-expression:
555 //      cast-expression
556 //      multiplicative-expression * cast-expression
557 //      multiplicative-expression / cast-expression
558 //      multiplicative-expression % cast-expression
559 <multiplicative-expr> -> <cast-expr>
560 <multiplicative-expr> -> <multiplicative-expr> OP_MUL <cast-expr>
561 <multiplicative-expr> -> <multiplicative-expr> OP_DIV <cast-expr>
562 <multiplicative-expr> -> <multiplicative-expr> OP_MOD <cast-expr>
563
564 // (6.5.4)
565 // cast-expression:
566 //      unary-expression
567 //      ( type-name ) cast-expression
568 <cast-expr> -> <unary-expr>
569 <cast-expr> -> DELIM_LPAR <type-name> DELIM_RPAR <cast-expr>
570
571 // (6.5.3)
572 // unary-expression:
573 //      postfix-expression
574 //      ++ unary-expression
575 //      -- unary-expression
```

```
576 //      unary-operator cast-expression
577 //      sizeof unary-expression
578 //      sizeof ( type-name )
579 //      _Alignof ( type-name ) (不用)
580 <unary-expr> -> <postfix-expr>
581 <unary-expr> -> OP_INC <unary-expr>
582 <unary-expr> -> OP_DEC <unary-expr>
583 <unary-expr> -> <unary-op> <cast-expr>
584 <unary-expr> -> sizeof <unary-expr>
585 <unary-expr> -> sizeof DELIM_LPAR <type-name> DELIM_RPAR
586
587 // (6.5.3)
588 // unary-operator:
589 //      & * + - ~ !
590 <unary-op> -> OP_AND
591 <unary-op> -> OP_MUL
592 <unary-op> -> OP_ADD
593 <unary-op> -> OP_SUB
594 <unary-op> -> OP_NOT
595 <unary-op> -> OP_LNOT
596
597 // (6.5.2)
598 // argument-expression-list:
599 //      assignment-expression
600 //      argument-expression-list , assignment-expression
601 <arg-expr-list> -> <asn-expr>
602 <arg-expr-list> -> <arg-expr-list> DELIM_COMMA <asn-expr>
603
604 // (6.5.2)
605 // postfix-expression:
606 //      primary-expression
607 //      postfix-expression [ expression ]
608 //      postfix-expression ( argument-expression-list_opt )
609 //      postfix-expression . identifier
610 //      postfix-expression -> identifier
611 //      postfix-expression ++
612 //      postfix-expression --
```

```
613 //      ( type-name ) { initializer-list }
614 //      ( type-name ) { initializer-list , }
615 <postfix-expr> -> <prim-expr>
616 <postfix-expr> -> <postfix-expr> DELIM_LSQBRACKET <expr> DELIM_RSQBRACKET
617 <postfix-expr> -> <postfix-expr> DELIM_LPAR <arg-expr-list> DELIM_RPAR
618 <postfix-expr> -> <postfix-expr> DELIM_LPAR DELIM_RPAR
619 <postfix-expr> -> <postfix-expr> OP_DOT IDENTIFIER
620 <postfix-expr> -> <postfix-expr> OP_ARROW IDENTIFIER
621 <postfix-expr> -> <postfix-expr> OP_INC
622 <postfix-expr> -> <postfix-expr> OP_DEC
623 <postfix-expr> -> DELIM_LPAR <type-name> DELIM_RPAR DELIM_LCURBRACE <initializer-
       list> DELIM_RCURBRACE
624 <postfix-expr> -> DELIM_LPAR <type-name> DELIM_RPAR DELIM_LCURBRACE <initializer-
       list> DELIM_COMMA DELIM_RCURBRACE
625
626 // (6.5.1.1) 不用
627
628 // (6.5.1)
629 // primary-expression:
630 //      identifier
631 //      constant
632 //      string-literal
633 //      ( expression )
634 //      generic-selection (不用)
635 <prim-expr> -> IDENTIFIER
636 <prim-expr> -> <constant>
637 <prim-expr> -> DELIM_LPAR <expr> DELIM_RPAR
638 <constant> -> <num-const>
639 <constant> -> CHAR_CONST
640 <constant> -> STR_LITERAL
641 <num-const> -> INT_CONST
642 <num-const> -> FLOAT_CONST
```

B 程序设计清单

工具文件 **util.h**、词法分析器类的定义与实现 **lexer.h / lexer.cpp** 与词法分析器部分相同，仅有略微修改，这里仅给出语法分析器部分新增的文件：

1. **语法分析器类的定义**: **parser.h**:

```
#ifndef PARSER_H
#define PARSER_H

#include "lexer.h"

// 文法结构体
struct Grammar
{
    using SymbolType = std::string;
    // 非终结符
    using NonTerminalType = SymbolType;
    using NonTerminals = std::unordered_set<NonTerminalType>;

    // 终结符
    using TerminalType = SymbolType;
    using Terminals = std::unordered_set<TerminalType>;

    // 起始符号
    using StartSymbolType = NonTerminalType;

    // 产生式(压缩版，键为左端非终结符，值为该非终结符所有产生式的右端)
    using Productions = std::unordered_map<NonTerminalType, std::vector<std::
    vector<SymbolType>>>;

    // 一条产生式(二元组，左端和右端符号)
    using ProductionType = std::pair<NonTerminalType, std::vector<SymbolType>>;

    // 四元组
    // 非终结符
    NonTerminals nonTerminals;
    // 终结符
```

```
31      Terminals terminals;
32      // 开始符号
33      StartSymbolType startSymbol;
34      // 产生式
35      Productions productions;
36
37      // 判断是否非终结符
38      bool isNonTerminal(const NonTerminalType & symbol)
39      {
40          return (nonTerminals.find(symbol) != nonTerminals.end());
41      }
42      // 判断是否终结符
43      bool isTerminal(const TerminalType & symbol)
44      {
45          return (terminals.find(symbol) != terminals.end() || symbol == "");
46      }
47      // 判断是否开始符号
48      bool isStartSymbol(const StartSymbolType & symbol)
49      {
50          return symbol == startSymbol;
51      }
52  };
53
54  class Parser
55  {
56      // ------------------------ 公有成员 ------------------------
57      // ------------------------ LL(1) 分析 ------------------------
58          // LL(1) 预测分析表类型
59          using LL1ParseTableType = std::map<
60              std::pair<Grammar::NonTerminalType, Grammar::TerminalType>,
61              Grammar::ProductionType>;
62          // 两种表类型
63          using FirstTableType = std::map< Grammar::NonTerminalType, std::set<
    Grammar::TerminalType> >;
64          using FollowTableType = std::map< Grammar::NonTerminalType, std::set<
    Grammar::TerminalType> >;
65      // ------------------------ LR 分析 ------------------------
```

```
66          // LR 项目
67          // (产生式, 其它信息)
68          // 产生式 := (左侧符号, 同一左端符号中第几个产生式)
69          // 其它信息 := (点的位置, 展望符号)
70          using LRTerm = std::pair<std::pair<Grammar::NonTerminalType, size_t>,
    std::pair<size_t, Grammar::SymbolType>>;
71
72          // 四类动作
73          enum class ACTION
74          {
75              SHIFT,
76              REDUCE,
77              GOTO,
78              ACCEPT
79          };
80          using ActionTableTerm = std::pair<ACTION, std::any>;
81
82          // LR(0) 分析表
83          // (状态, 符号) -> (ACTION, 附加信息)
84          // 附加信息如下一状态、使用什么产生式归约等
85          using LRParseTableType = std::map<
86              std::pair<size_t, Grammar::SymbolType>, ActionTableTerm
87          >;
88      public:
89
90          // ------------------------ 构造函数 ------------------------
91          // 默认构造函数
92          Parser();
93          // 复制构造(标记删除)
94          Parser(const Parser & other) = delete;
95
96          // ------------------------ 析构函数 ------------------------
97          ~Parser();
98
99          // ------------------------ 成员函数 ------------------------
100         // 关联文件
101         bool openFile(const std::string & srcName);
```

```
102         // 关闭文件
103         void closeFile();
104         // 读取语法
105         bool readGrammar(const std::string & grmName);
106         // ------------------------ LL(1) 语法分析 ------------------------
107         // 计算 LL(1) 预测分析表
108         bool calcLL1ParseTable();
109         // 打印内部表格
110         void printLL1InternalTables(std::ostream & out = std::cout);
111         // LL(1) 解析
112         size_t LL1Parse(std::ostream & out = std::cout);
113         // 保存 LL(1) 分析表
114         bool saveLL1ParseTable(const std::string & fileName);
115         // 读取 LL(1) 分析表
116         bool readLL1ParseTable(const std::string & fileName);
117         // ------------------------ LR(1) 语法分析 ------------------------
118         // 计算 LR 解析表
119         bool calcLRParseTable();
120         // 打印内部表格
121         void printLRInternalTables(std::ostream & out = std::cout);
122         // LR 解析
123         size_t LRParse(std::ostream & out = std::cout);
124         // 保存 LR(1) 分析表
125         bool saveLRParseTable(const std::string & fileName);
126         // 读取 LR(1) 分析表
127         bool readLRParseTable(const std::string & fileName);
128         // 错误处理
129         void errorProcess(const Types::ParserError & error);
130     private:
131         // 打印 FIRST
132         void printFirstTable(std::ostream & out);
133         // 打印 FOLLOW
134         void printFollowTable(std::ostream & out);
135         // 打印预测分析表
136         void printLL1ParseTable(std::ostream & out);
137
138         // 词法分析器
```

```
139         Lexer lexer;
140         // 上下文无关文法
141         Grammar grammar =
142         {
143             // 非终结符
144             {"S", "E", "T", "G", "F", "H"},
145             // 终结符
146             {"OP_ADD", "OP_SUB", "OP_MUL", "OP_DIV", "DELIM_LPAR", "DELIM_RPAR"
    , "IDENTIFIER", "", Shared::endOfFileChar},
147             // 起始符号
148             "S",
149             // 产生式集合
150             {
151                 { "S", {{"E"}} },
152                 { "E", {{"T", "G"}} },
153                 { "G", {{"OP_ADD", "T", "G"}, {"OP_SUB", "T", "G"}, {""}} },
154                 { "T", {{"F", "H"}} },
155                 { "H", {{"OP_MUL", "F", "H"}, {"OP_DIV", "F", "H"}, {""}} },
156                 { "F", {{"DELIM_LPAR", "E", "DELIM_RPAR"}, {"IDENTIFIER"}} }
157             }
158         };
159 
160         // ------------------------ LL(1) 语法分析 ------------------------
161         // 两个表
162         FirstTableType firstTable;
163         FollowTableType followTable;
164         // LL(1) 预测分析表
165         LL1ParseTableType LL1parseTable;
166         // ------------------------ LR(0) 语法分析 ------------------------
167         // LR 动作表
168         LRParseTableType LRparseTable;
169 };
170 
171 #endif
```

2. 语法分析器类的实现: **parser.cpp**:

```
1 #include "parser.h"
```

```
2
3  Parser::Parser(){}
4
5  Parser::~Parser(){}
6
7  // 打开文件
8  bool Parser::openFile(const std::string & srcName)
9  {
10     return lexer.openFile(srcName);
11 }
12
13 // 关闭文件
14 void Parser::closeFile()
15 {
16     lexer.closeFile();
17 }
18
19 // 读取文法
20 bool Parser::readGrammar(const std::string & grmFileName)
21 {
22     // 打开文件
23     std::fstream grmStream;
24     grmStream.open(grmFileName);
25     // 打不开文件
26     if(!grmStream.is_open())
27     {
28         std::cout << "\033[1m(Parser Generator)\033[0m \033[1;35mwarning:\033[0
    m\033[1m Can't open grammar file: "
29             << grmFileName << ", use default grammar instead.\033[0m" << std::
    endl;
30         return true;
31     }
32
33     // 四元组
34     Grammar::NonTerminals nonTerminals;
35     Grammar::Terminals terminals;
36     Grammar::StartSymbolType startSymbol;
```

```
37      Grammar::Productions productions;
38
39      // 循环读取
40      while(!grmStream.eof())
41      {
42          // 一行
43          std::string lineString;
44          // 读进来一行
45          std::getline(grmStream, lineString);
46          if(lineString.empty())
47          {
48              continue;
49          }
50          // 建立字符串流
51          std::stringstream lineStream(lineString);
52          // 当前符号
53          Grammar::SymbolType nowSymbol;
54
55          // 读取左端
56          lineStream >> nowSymbol;
57          if(nowSymbol.size() >= 2 && nowSymbol[0] == '/' && nowSymbol[1] == '/')
58          {
59              continue;
60          }
61          // 第一行确定开始符号
62          if(startSymbol == "")
63          {
64              startSymbol = nowSymbol;
65          }
66          // 确定左端
67          Grammar::NonTerminalType leftPart = nowSymbol;
68          // 左端不能为空
69          if(leftPart == "\"\"" || leftPart == "''")
70          {
71              return false;
72          }
73          // 左端一定是非终结符
```

```
74          nonTerminals.insert(leftPart);
75          terminals.insert(leftPart);
76
77          // 应该是 ->
78          lineStream >> nowSymbol;
79          if(nowSymbol != "->")
80          {
81              return false;
82          }
83
84          // 读取右端
85          std::vector<Grammar::SymbolType> rightPart;
86          while(lineStream >> nowSymbol)
87          {
88              if(nowSymbol.size() >= 2 && nowSymbol[0] == '/' && nowSymbol[1] ==
    '/')
89              {
90                  break;
91              }
92              // 空字符
93              if(nowSymbol == "\"\"" || nowSymbol == "''")
94              {
95                  nowSymbol = "";
96              }
97              rightPart.push_back(nowSymbol);
98              terminals.insert(nowSymbol);
99          }
100         // 加入其中
101         productions[leftPart].push_back(rightPart);
102     }
103
104     // 保留所有符号，然后去掉终结符即为非终结符
105     // 开始去掉所有非终结符
106     for(const auto & nonTerminal : nonTerminals)
107     {
108         terminals.erase(nonTerminal);
109     }
```

```
110     // EOF 也是终结符
111     terminals.insert(Shared::endOfFileChar);
112 
113     // 增广文法
114     if(startSymbol == "<EXTEND-GRAMMAR-START>" && productions[startSymbol].size
        () > 1)
115     {
116         std::cout << startSymbol << "is reserved." << std::endl;
117         return false;
118     }
119     productions["<EXTEND-GRAMMAR-START>"] = {{startSymbol}};
120     startSymbol = "<EXTEND-GRAMMAR-START>";
121     nonTerminals.insert(startSymbol);
122     grammar = {nonTerminals, terminals, startSymbol, productions};
123 
124     return true;
125 }
126 
127 // ------------------------ LL(1) 语法分析 ------------------------
128 // 计算 LL(1) 解析表
129 bool Parser::calcLL1ParseTable()
130 {
131     // 两个表
132     FirstTableType tmpFirstTable;
133     FollowTableType tmpFollowTable;
134 
135     // 最大迭代次数
136     size_t maxIteration = 1e6;
137 
138     // 根据当前 FIRST 表，得到一串符号的 FIRST
139     auto firstOfSymbols = [this, &tmpFirstTable](const std::vector<Grammar::
        SymbolType> & symbols) -> std::set<Grammar::TerminalType>
140     {
141         if(symbols.empty())
142         {
143             return {""};
144         }
```

```
145         // 一开始，空集合
146         std::set<Grammar::TerminalType> s = {};
147         for(size_t i = 0; i < symbols.size(); i++)
148         {
149             std::set<Grammar::TerminalType> firstOfThisSymbol;
150             // 如果是终结符
151             if(this -> grammar.isTerminal(symbols[i]))
152             {
153                 firstOfThisSymbol.insert(symbols[i]);
154             }
155             // 查表得到当前符号的 First
156             else if(grammar.isNonTerminal(symbols[i]))
157             {
158                 // 没有对应产生式
159                 if(tmpFirstTable.find(symbols[i]) == tmpFirstTable.end())
160                 {
161                     std::cout << "Exception: In function firstOfSymbols." <<
    std::endl;
162                     std::cout << symbols[i] << ": No corresponding production."
     << std::endl;
163                     return {};
164                 }
165                 firstOfThisSymbol = tmpFirstTable[symbols[i]];
166             }
167             else
168             {
169                 std::cout << "Exception: In function firstOfSymbols." << std::
    endl;
170                 std::cout << symbols[i] << ": Neither a terminal nor a non-
    terminal." << std::endl;
171                 return {};
172             }
173 
174             // 将除去 空字 的 First 加入
175             s.insert(firstOfThisSymbol.begin(), firstOfThisSymbol.end());
176             s.erase("");
177             // 当前字的 First 不含空，则停止
```

```
178             if(firstOfThisSymbol.find("") == firstOfThisSymbol.end())
179             {
180                 break;
181             }
182             // 最后一个还含有空，则加入空
183             if(i == symbols.size() - 1)
184             {
185                 s.insert("");
186             }
187         }
188         return s;
189     };
190 
191     // 计算 FISRT 表
192     auto calcFirstTable = [this, maxIteration, &tmpFirstTable, firstOfSymbols
    ]() -> bool
193     {
194         size_t iter = 0;
195         // 初始化
196         for(const auto & symbol : grammar.nonTerminals)
197         {
198             // 各符号都是空集合
199             tmpFirstTable[ symbol ] = {};
200         }
201         // 判断是否有更新
202         bool modifiedFlag = true;
203         while(modifiedFlag)
204         {
205             iter++;
206             modifiedFlag = false;
207             // 对每个产生式
208             for(const auto & production : this -> grammar.productions)
209             {
210                 // 左半部
211                 const auto & leftPart = production.first;
212                 // 对于右侧每一个产生式
213                 for(const auto & rightPart : production.second)
```

```
214                 {
215                     // 原集合大小
216                     size_t oldSize = tmpFirstTable[leftPart].size();
217                     // 计算它们的 First
218                     std::set<Grammar::TerminalType> firstOfRightPart =
    firstOfSymbols(rightPart);
219                     // 合并集合
220                     tmpFirstTable[leftPart].insert(firstOfRightPart.begin(),
    firstOfRightPart.end());
221                     // 增加符号后的大小
222                     size_t newSize = tmpFirstTable[leftPart].size();
223                     if(newSize != oldSize)
224                     {
225                         modifiedFlag = true;
226                     }
227                 }
228             }
229             if(iter > maxIteration)
230             {
231                 return false;
232             }
233         }
234         firstTable = tmpFirstTable;
235         return true;
236     };
237 
238     // 计算 FOLLOW 表
239     auto calcFollowTable = [this, maxIteration, &tmpFirstTable, &tmpFollowTable
    , firstOfSymbols]() -> bool
240     {
241         size_t iter = 0;
242         // 初始化
243         for(const auto & symbol : grammar.nonTerminals)
244         {
245             // 各符号都是空集合
246             tmpFollowTable[ symbol ] = {};
247         }
```

```
248         // Follow 表起始字符为结束字符
249         tmpFollowTable[grammar.startSymbol] = {Shared::endOfFileChar};
250         // 判断是否有更新
251         bool modifiedFlag = true;
252         while(modifiedFlag)
253         {
254             iter++;
255             modifiedFlag = false;
256             // 对每一个产生式
257             for(const auto & production : grammar.productions)
258             {
259                 // 计算左部、右部
260                 const auto & leftPart = production.first;
261                 for(const auto & rightPart : production.second)
262                 {
263                     // 考虑每个产生式右边的非终结符
264                     for(size_t i = 0; i < rightPart.size(); i++)
265                     {
266                         // 跳过终结符
267                         if(grammar.isTerminal(rightPart[i]))
268                         {
269                             continue;
270                         }
271                         // 既不是终结符也不是非终结符
272                         if(!grammar.isNonTerminal(rightPart[i]))
273                         {
274                             std::cout << "Exception: In function
    calcFollowTable." << std::endl;
275                             return false;
276                         }
277                         size_t oldSize = tmpFollowTable[rightPart[i]].size();
278                         // 获取右边的符号
279                         std::vector<Grammar::SymbolType> rightSymbols(rightPart
    .begin() + i + 1, rightPart.end());
280                         // 计算右边的符号串的 First
281                         std::set<Grammar::TerminalType> firstOfRightSymbols =
    firstOfSymbols(rightSymbols);
```

```
282                         // 并进去
283                         tmpFollowTable[rightPart[i]].insert(firstOfRightSymbols
    .begin(), firstOfRightSymbols.end());
284                         // 如果有空串，则去掉，并且把产生式左边的符号的 Follow
    加进去
285                         if(tmpFollowTable[rightPart[i]].find("") !=
    tmpFollowTable[rightPart[i]].end())
286                         {
287                             tmpFollowTable[rightPart[i]].erase("");
288                             tmpFollowTable[rightPart[i]].insert(tmpFollowTable[
    leftPart].begin(), tmpFollowTable[leftPart].end());
289                         }
290                         // 新的大小
291                         size_t newSize = tmpFollowTable[rightPart[i]].size();
292                         if(newSize != oldSize)
293                         {
294                             modifiedFlag = true;
295                         }
296                     }
297                 }
298             }
299             if(iter > maxIteration)
300             {
301                 return false;
302             }
303         }
304         followTable = tmpFollowTable;
305         return true;
306     };
307
308     // 计算预测分析表
309     auto calcParseTable = [this, &tmpFirstTable, &tmpFollowTable,
    firstOfSymbols]() -> bool
310     {
311         // 开始填临时分析表
312         LL1ParseTableType tmpParseTable;
313         // 考虑每一条产生式
```

```
314         for(const auto & production : grammar.productions)
315         {
316             // 左侧
317             const auto & leftPart = production.first;
318             // 右侧每一条
319             for(const auto & rightPart : production.second)
320             {
321                 // 产生式
322                 Grammar::ProductionType p(leftPart, rightPart);
323                 // 计算右部符号的 First 集合
324                 const std::set<Grammar::TerminalType> & firstOfRightSymbols =
    firstOfSymbols(rightPart);
325                 // 考虑每一个 First 中的符号 a
326                 for(const auto & firstTerminal : firstOfRightSymbols)
327                 {
328                     // 如果有空串, 则要计算左边的 Follow
329                     if(firstTerminal == "")
330                     {
331                         // 左边符号的 Follow
332                         const auto & followOfLeftSymbol = followTable[leftPart
    ];
333                         //  b in Follow(A), 将 A -> alpha 加入 M[A, b]
334                         for(const auto & followTerminal : followOfLeftSymbol)
335                         {
336                             if(
337                                 // 如果 M[A, b] 没有值
338                                 tmpParseTable.find(
339                                     std::make_pair(leftPart, followTerminal)
340                                 ) == tmpParseTable.end()
341                             )
342                             {
343                                 // 加入
344                                 tmpParseTable[std::make_pair(leftPart,
    followTerminal)] = p;
345                             }
346                             // 如果有值, 但不冲突
347                             else if(tmpParseTable[std::make_pair(leftPart,
```

```
    followTerminal)] == p)
348                             {
349
350                             }
351                             // 冲突，这不是 LL(1) 文法，返回
352                             else
353                             {
354                                 std::cout << "Action conflict: " << leftPart <<
     " " << followTerminal << std::endl;
355                                 return false;
356                             }
357                         }
358                     }
359                     else
360                     {
361                         if(
362                             // 如果 M[A, a] 没有值
363                             tmpParseTable.find(
364                                 std::make_pair(leftPart, firstTerminal)
365                             ) == tmpParseTable.end()
366                         )
367                         {
368                             // 加入
369                             tmpParseTable[std::make_pair(leftPart,
    firstTerminal)] = p;
370                         }
371                         // 如果有值，但不冲突
372                         else if(tmpParseTable[std::make_pair(leftPart,
    firstTerminal)] == p)
373                         {
374
375                         }
376                         // 冲突，这不是 LL(1) 文法，返回
377                         else
378                         {
379                             std::cout << "Action conflict: " << leftPart << " "
     << firstTerminal << std::endl;
```

```
380                             return false;
381                         }
382                     }
383                 }
384             }
385         }
386         this -> LL1parseTable = tmpParseTable;
387         return true;
388     };
389 
390     // 计算 First
391     if(!calcFirstTable())
392     {
393         std::cout << "\033[1m(Parser Generator)\033[0m \033[1;31merror:\033[0m
    \033[1m Build First-Table failed. (max iteration exceeded)\033[0m " << std::
    endl;
394         return false;
395     }
396     // 计算 Follow
397     if(!calcFollowTable())
398     {
399         std::cout << "\033[1m(Parser Generator)\033[0m \033[1;31merror:\033[0m
    \033[1m Build Follow-Table failed. (max iteration exceeded)\033[0m " << std
    ::endl;
400         return false;
401     }
402     // 计算 ParseTable
403     if(!calcParseTable())
404     {
405         std::cout << "\033[1m(Parser Generator)\033[0m \033[1;31merror:\033[0m
    \033[1m Build Parse-Table failed. (not LL(1) grammar)\033[0m" << std::endl;
406         return false;
407     }
408     return true;
409 }
410 
411 // 打印 LL(1) 表格
```

```
412 void Parser::printLL1InternalTables(std::ostream & out)
413 {
414     out << "This is an LL(1) grammar." << std::endl << std::endl;
415     this -> printFirstTable(out);
416     this -> printFollowTable(out);
417     this -> printLL1ParseTable(out);
418 }
419 
420 // LL(1) 语法分析
421 size_t Parser::LL1Parse(std::ostream & out)
422 {
423     // 重置
424     lexer.rewind();
425     // 分析栈
426     std::vector<std::string> parseStack;
427 
428     // 打印分析栈
429     auto printParseStack = [&parseStack, &out]() -> void
430     {
431         out << "Parse stack: ";
432         for(const auto & symbol : parseStack)
433         {
434             out << symbol << " ";
435         }
436         out << std::endl;
437     };
438 
439     // 打印 token
440     auto printToken = [&out](Types::TokenPair token) -> void
441     {
442         out << Shared::typeStrings.at(token.first) << " ";
443         if(token.first >= Types::TokenType::INIT
444             && token.first <= Types::TokenType::ENDOFFILE )
445         {
446             // out;
447         }
448         else if(token.first == Types::TokenType::KEYWORD )
```

```
449         {
450             out << "( " << std::any_cast<std::string>(token.second) << " )";
451         }
452         else if(token.first == Types::TokenType::IDENTIFIER )
453         {
454             out << "( " << Shared::idTable.at(std::any_cast<size_t>(token.
    second)) << " )";
455         }
456         else if(token.first >= Types::TokenType::INT_CONST
457             && token.first <= Types::TokenType::STR_LITERAL )
458         {
459             out << "( " << Shared::constTable.at(std::any_cast<size_t>(token.
    second)) << " )";
460         }
461         else if(token.first >= Types::TokenType::OP_ADD
462             && token.first <= Types::TokenType::OP_SCOPE )
463         {
464             out << "( " << std::any_cast<std::string>(token.second) << " )";
465         }
466         else if(token.first >= Types::TokenType::DELIM_DBQUOTE
467             && token.first <= Types::TokenType::DELIM_QUESTION )
468         {
469             out << "( " << std::any_cast<std::string>(token.second) << " )";
470         }
471     };
472
473     // 初始化入栈
474     parseStack.push_back(Shared::endOfFileChar);
475     parseStack.push_back(grammar.startSymbol);
476
477     // 错误
478     size_t errorCount = 0;
479
480     // 指向第一个词
481     auto token = lexer.getNextToken();
482     while(true)
483     {
```

```
484         // 跳过非实义符号
485         if(token.first == Types::TokenType::INIT)
486         {
487             token = lexer.getNextToken();
488             continue;
489         }
490 
491         // 打印栈
492         printParseStack();
493         out << "Now token: ";
494         printToken(token);
495         out << std::endl;
496 
497         // 如果词法错误，处理
498         if(token.first == Types::TokenType::ERROR)
499         {
500             errorCount++;
501             lexer.errorProcess(std::any_cast<Types::LexerError>(token.second));
502             // 跳过本词
503             token = lexer.getNextToken();
504         }
505         // 栈已经空了
506         else if(parseStack.back() == Shared::endOfFileChar)
507         {
508             // 对上了
509             if(token.first == Types::TokenType::ENDOFFILE)
510             {
511                 out << "Successfully finished." << std::endl;
512                 break;
513             }
514             else
515             {
516                 errorCount++;
517                 this -> errorProcess(Types::ParserError(lexer.getFilePos(), "
    unexpected end of file"));
518                 break;
519             }
```

```
520         }
521         // 非终结符，需要根据预测分析表
522         if(grammar.isNonTerminal(parseStack.back()))
523         {
524             std::string tokenTypeStr = Shared::typeStrings.at(token.first);
525             // 替换掉关键词
526             if(tokenTypeStr == "KEYWORD")
527             {
528                 tokenTypeStr = std::any_cast<std::string>(token.second);
529             }
530             auto tableTermIter = LL1parseTable.find(std::make_pair(parseStack.
    back(), tokenTypeStr));
531             // 找到了
532             if(tableTermIter != LL1parseTable.end())
533             {
534                 // 反向压入产生式
535                 const auto & rightPart = tableTermIter -> second.second;
536                 out << "Use rule: " << parseStack.back() << " -> ";
537                 for(const auto & symbol : rightPart)
538                 {
539                     if(symbol == "")
540                     {
541                         out << "\"\"" << " ";
542                     }
543                     else
544                     {
545                         out << symbol << " ";
546                     }
547                 }
548                 out << std::endl << std::endl;
549                 parseStack.pop_back();
550                 for(auto i = rightPart.rbegin(); i != rightPart.rend(); i++)
551                 {
552                     // 跳过 ""
553                     if(*i != "")
554                     {
555                         parseStack.push_back(*i);
```

```
556                     }
557                 }
558             }
559             else
560             {
561                 errorCount++;
562                 // 错误信息
563                 std::string errorMessage = "unexpected token: " + tokenTypeStr;
564                 errorMessage += ", expected: " + parseStack.back();
565                 this -> errorProcess(Types::ParserError(lexer.getFilePos(),
    errorMessage));
566                 break;
567             }
568         }
569         // 终结符
570         else if(grammar.isTerminal(parseStack.back()))
571         {
572             std::string tokenTypeStr = Shared::typeStrings.at(token.first);
573             // 替换掉关键词
574             if(tokenTypeStr == "KEYWORD")
575             {
576                 tokenTypeStr = std::any_cast<std::string>(token.second);
577             }
578             // 对上了
579             if(parseStack.back() == tokenTypeStr)
580             {
581                 out << "Use rule: Pop stack" << std::endl << std::endl;
582                 parseStack.pop_back();
583                 token = lexer.getNextToken();
584             }
585             else
586             {
587                 errorCount++;
588                 // 错误信息
589                 std::string errorMessage = "unexpected token: " + tokenTypeStr;
590                 errorMessage += ", expected: " + parseStack.back();
591                 this -> errorProcess(Types::ParserError(lexer.getFilePos(),
```

```
    errorMessage));
592                 break;
593             }
594         }
595         else
596         {
597             errorCount++;
598             // 错误信息
599             std::string errorMessage = "unexpected token: " + Shared::
    typeStrings.at(token.first);
600             errorMessage += ", expected: " + parseStack.back();
601             this -> errorProcess(Types::ParserError(lexer.getFilePos(),
    errorMessage));
602             break;
603         }
604     }
605     return errorCount;
606 }
607
608 // 保存 LL(1) 预测分析表
609 bool Parser::saveLL1ParseTable(const std::string & fileName)
610 {
611     std::cout << "Saving parsing table..." << std::endl;
612     // 打开文件
613     std::fstream outStream(fileName, std::ios::out);
614     if(!outStream.is_open())
615     {
616         std::cout << "\033[1m(Parser Generator)\033[0m \033[1;31merror:\033[0m
    \033[1m Can't open output file: "
617             << fileName << "\033[0m" << std::endl;
618         return false;
619     }
620     printLL1ParseTable(outStream);
621     outStream.close();
622     return true;
623 }
624
```

```
625 // 读取 LL(1) 预测分析表
626 bool Parser::readLL1ParseTable(const std::string & fileName)
627 {
628     std::cout << "Reading parsing table..." << std::endl;
629     // 打开文件
630     std::fstream fileStream;
631     fileStream.open(fileName);
632     // 打不开文件
633     if(!fileStream.is_open())
634     {
635         std::cout << "\033[1m(Parser Generator)\033[0m \033[1;31merror:\033[0m
    \033[1m Can't open grammar file: "
636             << fileName << "\033[0m" << std::endl;
637         return false;
638     }
639 
640     // 循环读取
641     while(!fileStream.eof())
642     {
643         // 一行
644         std::string lineString;
645         // 读进来一行
646         std::getline(fileStream, lineString);
647         if(lineString.empty())
648         {
649             continue;
650         }
651         // 建立字符串流
652         std::stringstream lineStream(lineString);
653         // 当前符号
654         std::string nowSymbol;
655 
656         // 非终结符
657         Grammar::NonTerminalType NT;
658         // 终结符
659         Grammar::TerminalType T;
660         // 产生式
```

```
661         Grammar::ProductionType P;
662 
663         // 读取 M [ 部分
664         lineStream >> nowSymbol;
665         if(nowSymbol != "M")
666             return false;
667 
668         lineStream >> nowSymbol;
669         if(nowSymbol != "[")
670             return false;
671 
672         // 读取非终结符
673         lineStream >> NT;
674         if(!grammar.isNonTerminal(NT))
675             return false;
676 
677         // 读取 ,
678         lineStream >> nowSymbol;
679         if(nowSymbol != ",")
680             return false;
681 
682         // 读取终结符
683         lineStream >> T;
684         if(T == "\"\"")
685             T = "";
686         if(!grammar.isTerminal(T))
687             return false;
688 
689         // 读取 ] = 部分
690         lineStream >> nowSymbol;
691         if(nowSymbol != "]")
692             return false;
693 
694         lineStream >> nowSymbol;
695         if(nowSymbol != "=")
696             return false;
697 
```

```
698         // 读取左端
699         Grammar::NonTerminalType leftPart;
700         lineStream >> leftPart;
701         if(!grammar.isNonTerminal(leftPart))
702             return false;
703         // 读取箭头
704         lineStream >> nowSymbol;
705         if(nowSymbol != "->")
706             return false;
707         // 读取右端
708         std::vector<Grammar::SymbolType> rightPart;
709         while(lineStream >> nowSymbol)
710         {
711             // 空字符
712             if(nowSymbol == "\"\"" || nowSymbol == "''")
713             {
714                 nowSymbol = "";
715             }
716             rightPart.push_back(nowSymbol);
717         }
718         P = {leftPart, rightPart};
719         LL1parseTable[std::make_pair(NT, T)] = P;
720     }
721     std::cout << "Done." << std::endl;
722     return true;
723 }
724
725 // ------------------------ LR(1) 语法分析 ------------------------
726 // 计算 LR(1) 解析表
727 bool Parser::calcLRParseTable()
728 {
729     // -------------------- FIRST 表和 FOLLOW 表的填写 --------------------
730     // 两个表
731     FirstTableType tmpFirstTable;
732     FollowTableType tmpFollowTable;
733
734     // 最大迭代次数
```

```
735     size_t maxIteration = 1e6;
736 
737     // 根据当前 FIRST 表，得到一串符号的 FIRST
738     auto firstOfSymbols = [this, &tmpFirstTable](const std::vector<Grammar::
    SymbolType> & symbols) -> std::set<Grammar::TerminalType>
739     {
740         if(symbols.empty())
741         {
742             return {""};
743         }
744         // 一开始，空集合
745         std::set<Grammar::TerminalType> s = {};
746         for(size_t i = 0; i < symbols.size(); i++)
747         {
748             std::set<Grammar::TerminalType> firstOfThisSymbol;
749             // 如果是终结符
750             if(this -> grammar.isTerminal(symbols[i]))
751             {
752                 firstOfThisSymbol.insert(symbols[i]);
753             }
754             // 查表得到当前符号的 First
755             else if(grammar.isNonTerminal(symbols[i]))
756             {
757                 // 没有对应产生式
758                 if(tmpFirstTable.find(symbols[i]) == tmpFirstTable.end())
759                 {
760                     std::cout << "Exception: In function firstOfSymbols." <<
    std::endl;
761                     std::cout << symbols[i] << ": No corresponding production."
     << std::endl;
762                     return {};
763                 }
764                 firstOfThisSymbol = tmpFirstTable[symbols[i]];
765             }
766             else
767             {
768                 std::cout << "Exception: In function firstOfSymbols." << std::
```

```
    endl;
769                 std::cout << symbols[i] << ": Neither a terminal nor a non-
    terminal." << std::endl;
770                 return {};
771             }
772
773             // 将除去 空字 的 First 加入
774             s.insert(firstOfThisSymbol.begin(), firstOfThisSymbol.end());
775             s.erase("");
776             // 当前字的 First 不含空, 则停止
777             if(firstOfThisSymbol.find("") == firstOfThisSymbol.end())
778             {
779                 break;
780             }
781             // 最后一个还含有空, 则加入空
782             if(i == symbols.size() - 1)
783             {
784                 s.insert("");
785             }
786         }
787         return s;
788     };
789
790     // 计算 FISRT 表
791     auto calcFirstTable = [this, maxIteration, &tmpFirstTable, firstOfSymbols
    ]() -> bool
792     {
793         size_t iter = 0;
794         // 初始化
795         for(const auto & symbol : grammar.nonTerminals)
796         {
797             // 各符号都是空集合
798             tmpFirstTable[ symbol ] = {};
799         }
800         // 判断是否有更新
801         bool modifiedFlag = true;
802         while(modifiedFlag)
```

```
803         {
804             iter++;
805             modifiedFlag = false;
806             // 对每个产生式
807             for(const auto & production : this -> grammar.productions)
808             {
809                 // 左半部
810                 const auto & leftPart = production.first;
811                 // 对于右侧每一个产生式
812                 for(const auto & rightPart : production.second)
813                 {
814                     // 原集合大小
815                     size_t oldSize = tmpFirstTable[leftPart].size();
816                     // 计算它们的 First
817                     std::set<Grammar::TerminalType> firstOfRightPart =
    firstOfSymbols(rightPart);
818                     // 合并集合
819                     tmpFirstTable[leftPart].insert(firstOfRightPart.begin(),
    firstOfRightPart.end());
820                     // 增加符号后的大小
821                     size_t newSize = tmpFirstTable[leftPart].size();
822                     if(newSize != oldSize)
823                     {
824                         modifiedFlag = true;
825                     }
826                 }
827             }
828             if(iter > maxIteration)
829             {
830                 return false;
831             }
832         }
833         firstTable = tmpFirstTable;
834         return true;
835     };
836
837     // 计算 FOLLOW 表
```

```
838     auto calcFollowTable = [this, maxIteration, &tmpFirstTable, &tmpFollowTable
    , firstOfSymbols]() -> bool
839     {
840         size_t iter = 0;
841         // 初始化
842         for(const auto & symbol : grammar.nonTerminals)
843         {
844             // 各符号都是空集合
845             tmpFollowTable[ symbol ] = {};
846         }
847         // Follow 表起始字符为结束字符
848         tmpFollowTable[grammar.startSymbol] = {Shared::endOfFileChar};
849         // 判断是否有更新
850         bool modifiedFlag = true;
851         while(modifiedFlag)
852         {
853             iter++;
854             modifiedFlag = false;
855             // 对每一个产生式
856             for(const auto & production : grammar.productions)
857             {
858                 // 计算左部、右部
859                 const auto & leftPart = production.first;
860                 for(const auto & rightPart : production.second)
861                 {
862                     // 考虑每个产生式右边的非终结符
863                     for(size_t i = 0; i < rightPart.size(); i++)
864                     {
865                         // 跳过终结符
866                         if(grammar.isTerminal(rightPart[i]))
867                         {
868                             continue;
869                         }
870                         // 既不是终结符也不是非终结符
871                         if(!grammar.isNonTerminal(rightPart[i]))
872                         {
873                             std::cout << "Exception: In function
```

```
    calcFollowTable." << std::endl;
874                             return false;
875                         }
876                         size_t oldSize = tmpFollowTable[rightPart[i]].size();
877                         // 获取右边的符号
878                         std::vector<Grammar::SymbolType> rightSymbols(rightPart
    .begin() + i + 1, rightPart.end());
879                         // 计算右边的符号串的 First
880                         std::set<Grammar::TerminalType> firstOfRightSymbols =
    firstOfSymbols(rightSymbols);
881                         // 并进去
882                         tmpFollowTable[rightPart[i]].insert(firstOfRightSymbols
    .begin(), firstOfRightSymbols.end());
883                         // 如果有空串，则去掉，并且把产生式左边的符号的 Follow
    加进去
884                         if(tmpFollowTable[rightPart[i]].find("") !=
    tmpFollowTable[rightPart[i]].end())
885                         {
886                             tmpFollowTable[rightPart[i]].erase("");
887                             tmpFollowTable[rightPart[i]].insert(tmpFollowTable[
    leftPart].begin(), tmpFollowTable[leftPart].end());
888                         }
889                         // 新的大小
890                         size_t newSize = tmpFollowTable[rightPart[i]].size();
891                         if(newSize != oldSize)
892                         {
893                             modifiedFlag = true;
894                         }
895                     }
896                 }
897             }
898             if(iter > maxIteration)
899             {
900                 return false;
901             }
902         }
903         followTable = tmpFollowTable;
```

```
904         return true;
905     };
906 
907     // 先计算一下整个文法的 FIRST 和 FOLLOW
908     // 计算 First
909     if(!calcFirstTable())
910     {
911         std::cout << "\033[1m(Parser Generator)\033[0m \033[1;31merror:\033[0m \033[1m Build First-Table failed. (max iteration exceeded)\033[0m " << std::endl;
912         return false;
913     }
914     // 计算 Follow
915     if(!calcFollowTable())
916     {
917         std::cout << "\033[1m(Parser Generator)\033[0m \033[1;31merror:\033[0m \033[1m Build Follow-Table failed. (max iteration exceeded)\033[0m " << std::endl;
918         return false;
919     }
920 
921     // -------------------- LR(1) 开始 --------------------
922     // 打印一个项目
923     auto printTerm = [this](std::ostream & out, const LRTerm & term) -> void
924     {
925         // 打印左端
926         out << "[ " << term.first.first;
927         out << " -> ";
928         const auto & rightPart = this -> grammar.productions[term.first.first][term.first.second];
929         size_t dotPos = term.second.first;
930         for(size_t i = 0; i < dotPos; i++)
931         {
932             out << rightPart[i] << " ";
933         }
934         out << "· ";
935         for(size_t i = dotPos; i < rightPart.size(); i++)
```

```
936         {
937             out << rightPart[i] << " ";
938         }
939         out << ", " << term.second.second << " ]" << std::endl;
940     };
941 
942     // 项目（初始仅一个）
943     LRTerm startTerm = {{grammar.startSymbol, 0}, {0, Shared::endOfFileChar}};
944     std::vector<LRTerm> LRTerms = {startTerm};
945 
946     // 项目到编号
947     std::map<LRTerm, size_t> termToIndex = {{startTerm, 0}};
948 
949     // 每个非终结符开头项目的序号
950     std::map<Grammar::NonTerminalType, std::set<size_t>> NTStartTerms = {{
    grammar.startSymbol, {0}}};
951 
952     // 动态地计算 LR(1) 项目集的闭包
953     //（即如果这个项目还没有遇到过，则需要扩展出来，边产生边计算）
954     // 计算规则：
955     //      1. I 的任何项目都属于 CLOSURE(I)
956     //      2. 若项目 [ A ->   B , a ] 属于 CLOSURE(I),
957     //          B -> r 是文法中的一条规则，b 属于 FIRST( a),
958     //          则 [ B ->  r, b ] 也属于CLOSURE(I)
959     auto closure = [this, firstOfSymbols, &LRTerms, &termToIndex, &NTStartTerms
    ](const std::set<size_t> & _i) -> std::set<size_t>
960     {
961         // 首先本身都属于闭包
962         std::set<size_t> indices = _i;
963         bool modified = true;
964         // 直到当前项目集不再扩大
965         while(modified)
966         {
967             modified = false;
968             // 扫描每个项目，看看后一个字符是不是非终结符，且能够扩展
969             for(const auto & i : indices)
970             {
```

```
971                 // 产生式左部
972                 const auto & leftPart = LRTerms[i].first.first;
973                 // 产生式右部
974                 const auto & rightPart = grammar.productions[leftPart][LRTerms[
    i].first.second];
975                 // 点在最后
976                 if(rightPart.size() <= LRTerms[i].second.first)
977                 {
978                     continue;
979                 }
980                 // 下一个符号是非终结符
981                 const std::string & nextSymbol = rightPart[LRTerms[i].second.
    first];
982                 if(grammar.isNonTerminal(nextSymbol))
983                 {
984                     // 旧 size，判断大小是否改变
985                     size_t oldSize = indices.size();
986                     // 求出下一个非终结符 B 的后面部分
987                     std::vector<Grammar::SymbolType> betaA = \
988                         std::vector<Grammar::SymbolType>
989                         (
990                             rightPart.begin() + LRTerms[i].second.first + 1,
991                             rightPart.end()
992                         );
993                     //   后面并上 a
994                     betaA.push_back(LRTerms[i].second.second);
995                     // 求出后面部分的首符号集 FIRST( a)
996                     auto firstOfBetaA = firstOfSymbols(betaA);
997 
998                     // 先找到所有 B -> r 产生式，然后
999                     // 对于 b 属于 FIRST( a)，将所有 [ B ->  r, b ] 加入CLOSURE
1000                    // 注意更新
1001                    // 对每一条 B -> r 产生式
1002                    for(size_t i = 0; i < grammar.productions.at(nextSymbol).
    size(); i++)
1003                    {
1004                        for(const auto b : firstOfBetaA)
```

```
1005                        {
1006                            // 新生成的项目
1007                            LRTerm newTerm = {{nextSymbol, i}, {0, b}};
1008                            // 没有过，加入
1009                            if(termToIndex.find(newTerm) == termToIndex.end())
1010                            {
1011                                LRTerms.push_back(newTerm);
1012                                termToIndex[newTerm] = LRTerms.size() - 1;
1013                                NTStartTerms[nextSymbol].insert(LRTerms.size()
    - 1);
1014                            }
1015                            indices.insert(termToIndex.at(newTerm));
1016                        }
1017                    }
1018
1019                    if(indices.size() != oldSize)
1020                    {
1021                        modified = true;
1022                    }
1023                }
1024            }
1025
1026        }
1027        return indices;
1028    };
1029
1030    // 计算 LR(0) 的自动机的同时 BFS 产生分析表
1031    auto calcFAAndParseTable = [this, &LRTerms, &termToIndex, &NTStartTerms,
    printTerm, closure]() -> bool
1032    {
1033        LRParseTableType tmpLRparseTable;
1034
1035        // LR 分析 FA 中的节点
1036        // 即为若干产生式的集合
1037        struct LRFANode
1038        {
1039            std::set<size_t> termIndices;
```

```
1040        };
1041
1042        // 各项目集族（FA 的节点）的比较函数
1043        class FANodeCmp
1044        {
1045            public:
1046                bool operator()(const LRFANode & nodeA, const LRFANode & nodeB)
     const
1047                {
1048                    return nodeA.termIndices < nodeB.termIndices;
1049                }
1050        };
1051
1052        // 打印一个节点
1053        auto printNode = [this, &LRTerms, printTerm](std::ostream & out, const
    LRFANode & node) -> void
1054        {
1055            for(const auto & i : node.termIndices)
1056            {
1057                printTerm(out, LRTerms[i]);
1058            }
1059        };
1060
1061        // 打印动作表项
1062        auto printAction = [this](std::ostream & out, const ActionTableTerm &
    term) -> void
1063        {
1064            if(term.first == ACTION::SHIFT)
1065            {
1066                out << "SHIFT " << std::any_cast<size_t>(term.second);
1067            }
1068            else if(term.first == ACTION::REDUCE)
1069            {
1070                out << "REDUCE ";
1071                const auto & production = std::any_cast<std::pair<Grammar::
    NonTerminalType, size_t>>(term.second);
1072                const auto & leftPart = production.first;
```

```
1073                const auto & rightPart = grammar.productions.at(leftPart)[
    production.second];
1074                out << leftPart << " -> ";
1075                for(const auto & symbol : rightPart)
1076                {
1077                    out << (symbol == "" ? "\"\"" : symbol) << " ";
1078                }
1079            }
1080            else if(term.first == ACTION::GOTO)
1081            {
1082                out << "GOTO " << std::any_cast<size_t>(term.second);
1083            }
1084            else if(term.first == ACTION::ACCEPT)
1085            {
1086                out << "ACCEPT";
1087            }
1088            out << std::endl;
1089        };
1090
1091        // 动作表项比较
1092        auto actionEq = [this](const std::pair<ACTION, std::any> & actionA,
    const std::pair<ACTION, std::any> & actionB) -> bool
1093        {
1094            if(actionA.first != actionB.first)
1095                return false;
1096            if(actionA.first == ACTION::ACCEPT)
1097                return true;
1098            else if(actionA.first == ACTION::GOTO || actionA.first == ACTION::
    SHIFT)
1099                return std::any_cast<size_t>(actionA.second) == std::any_cast<
    size_t>(actionB.second);
1100            else if(actionA.first == ACTION::REDUCE)
1101                return std::any_cast<std::pair<Grammar::NonTerminalType, size_t
    >>(actionA.second) ==
1102                    std::any_cast<std::pair<Grammar::NonTerminalType, size_t>>(
    actionB.second);
1103            else return false;
```

```
1104        };
1105
1106        // 冲突处理办法
1107        enum class HANDLINGFLAG
1108        {
1109            KEEP,
1110            OVERWRITE,
1111            ABORT
1112        };
1113        // 冲突处理
1114        auto conflictHandling = [this, printAction](
1115            const ActionTableTerm & inTable, const ActionTableTerm & generated)
    -> HANDLINGFLAG
1116            {
1117                std::cout << "In table: ";
1118                printAction(std::cout, inTable);
1119                std::cout << "Generated: ";
1120                printAction(std::cout, generated);
1121                std::cout << "[K]eep, [O]verwrite, [A]bort: ";
1122
1123                std::string cmd;
1124                std::cin >> cmd;
1125
1126                if(cmd == "k" || cmd == "K")
1127                {
1128                    return HANDLINGFLAG::KEEP;
1129                }
1130                else if(cmd == "o" || cmd == "O")
1131                {
1132                    return HANDLINGFLAG::OVERWRITE;
1133                }
1134                else return HANDLINGFLAG::ABORT;
1135            };
1136
1137        // 节点到序号的映射
1138        std::map<LRFANode, size_t, FANodeCmp> nodeToIndex;
1139        // 节点队列
```

```
1140        std::queue<LRFANode> q;
1141
1142        // 起始节点
1143        LRFANode startNode;
1144        // 扩展成闭包
1145        startNode.termIndices = closure(NTStartTerms.at(grammar.startSymbol));
1146        // 起始节点入队列
1147        q.push(startNode);
1148
1149        // 打印节点 0
1150        std::cout << "Nodes:" << std::endl;
1151        std::cout << "Node 0:" << std::endl;
1152        printNode(std::cout, startNode);
1153        std::cout << std::endl;
1154
1155        // 当队列不空
1156        while(!q.empty())
1157        {
1158            // 节点出队
1159            LRFANode node = q.front();
1160            q.pop();
1161
1162            // 节点加入节点-序号表
1163            if(nodeToIndex.find(node) == nodeToIndex.end())
1164            {
1165                nodeToIndex[node] = nodeToIndex.size();
1166            }
1167            // 获取当前节点的序号
1168            size_t nowNodeIndex = nodeToIndex.at(node);
1169
1170            // 记录出边上标签（移进项目中下一个符号）到项目号集合的映射
1171            std::map<Grammar::SymbolType, std::set<size_t>> outLabelToIndices;
1172            // 考虑每一项
1173            for(const size_t & termIndex : node.termIndices)
1174            {
1175                // 本项目的产生式的左半部分
1176                const auto & leftPart = LRTerms[termIndex].first.first;
```

```
1177                // 产生式的右半部分
1178                const auto & rightPart = grammar.productions.at(leftPart)[
    LRTerms[termIndex].first.second];
1179                // 点的位置
1180                const size_t & dotPos = LRTerms[termIndex].second.first;
1181                // 展望符号
1182                const auto & lookAheadSymbol = LRTerms[termIndex].second.second
    ;
1183                // 如果本项目是归约项目，先判断是否为接受项目，然后填表
1184                if(dotPos >= rightPart.size())
1185                {
1186                    // 接受
1187                    if(grammar.isStartSymbol(leftPart))
1188                    {
1189                        tmpLRparseTable[std::make_pair(nowNodeIndex,
    lookAheadSymbol)] = \
1190                            std::make_pair
1191                                (
1192                                    ACTION::ACCEPT,
1193                                    // 产生式
1194                                    std::any(nullptr)
1195                                );
1196                    }
1197                    else if
1198                    (
1199                        (tmpLRparseTable.find(std::make_pair(nowNodeIndex,
    lookAheadSymbol)) == tmpLRparseTable.end())
1200                        ||
1201                        (
1202                            tmpLRparseTable.find(std::make_pair(nowNodeIndex,
    lookAheadSymbol)) != tmpLRparseTable.end()
1203                            &&
1204                            actionEq
1205                            (
1206                                tmpLRparseTable.at(std::make_pair(nowNodeIndex,
     lookAheadSymbol)),
1207                                std::make_pair
```

```
1208                                    (
1209                                        ACTION::REDUCE,
1210                                        // 产生式
1211                                        std::any(LRTerms[termIndex].first)
1212                                    )
1213                                )
1214                            )
1215                        )
1216                        {
1217                            tmpLRparseTable[std::make_pair(nowNodeIndex,
    lookAheadSymbol)] = \
1218                                std::make_pair
1219                                    (
1220                                        ACTION::REDUCE,
1221                                        // 产生式
1222                                        std::any(LRTerms[termIndex].first)
1223                                    );
1224                        }
1225                        else
1226                        {
1227                            // 手动冲突处理
1228                            std::cout << "\033[1m(Parser Generator)\033[0m
    \033[1;35mwarning:\033[0m conflict at (" << nowNodeIndex << ", " <<
    lookAheadSymbol << ")" << std::endl;
1229                            HANDLINGFLAG flag = conflictHandling
1230                            (
1231                                tmpLRparseTable.at(std::make_pair(nowNodeIndex,
    lookAheadSymbol)),
1232                                std::make_pair
1233                                    (
1234                                        ACTION::REDUCE,
1235                                        // 产生式
1236                                        std::any(LRTerms[termIndex].first)
1237                                    )
1238                            );
1239                            if(flag == HANDLINGFLAG::OVERWRITE)
1240                            {
```

```
1241                            tmpLRparseTable.at(std::make_pair(nowNodeIndex,
    lookAheadSymbol)) =
1242                                std::make_pair
1243                                    (
1244                                        ACTION::REDUCE,
1245                                        // 产生式
1246                                        std::any(LRTerms[termIndex].first)
1247                                    );
1248                        }
1249                        else if(flag == HANDLINGFLAG::ABORT)
1250                        {
1251                            std::cout << "Abort." << std::endl;
1252                            return false;
1253                        }
1254                    }
1255                }
1256                // 本项目是移进项目，则记录下一个符号，填到出边表里面
1257                else
1258                {
1259                    const auto & nextSymbol = rightPart[dotPos];
1260                    // 计算下一项
1261                    auto nextTerm = LRTerms[termIndex];
1262                    // 点往后移动
1263                    nextTerm.second.first++;
1264                    // 新项目要更新
1265                    if(termToIndex.find(nextTerm) == termToIndex.end())
1266                    {
1267                        LRTerms.push_back(nextTerm);
1268                        termToIndex[nextTerm] = LRTerms.size() - 1;
1269                        NTStartTerms[nextSymbol].insert(LRTerms.size() - 1);
1270                    }
1271                    // 找到编号
1272                    size_t nextTermIndex = termToIndex.at(nextTerm);
1273                    if(outLabelToIndices.find(nextSymbol) == outLabelToIndices.
    end())
1274                    {
1275                        outLabelToIndices[nextSymbol] = std::set<size_t>();
```

```
1276                    }
1277                    // 加入编号
1278                    outLabelToIndices[nextSymbol].insert(nextTermIndex);
1279                }
1280            }
1281
1282            // 记录新节点编号、入队新节点，根据出边表填分析表中移进、转移项目
1283            for(const auto & symbolIndices : outLabelToIndices)
1284            {
1285                // 新节点
1286                LRFANode newNode;
1287                newNode.termIndices = closure(symbolIndices.second);
1288
1289                // 如果未访问过，节点加入节点-序号表
1290                if(nodeToIndex.find(newNode) == nodeToIndex.end())
1291                {
1292                    nodeToIndex[newNode] = nodeToIndex.size();
1293                    // 入队
1294                    q.push(newNode);
1295                    // 打印节点
1296                    std::cout << "Node " << nodeToIndex.at(newNode) << " :" <<
    std::endl;
1297                    printNode(std::cout, newNode);
1298                    std::cout << std::endl;
1299                }
1300                // 获取当前节点的序号
1301                size_t newNodeIndex = nodeToIndex.at(newNode);
1302
1303                // 填表
1304                // 如果是终结符 - 移进
1305                if(grammar.isTerminal(symbolIndices.first))
1306                {
1307                    if
1308                    (
1309                        (tmpLRparseTable.find(std::make_pair(nowNodeIndex,
    symbolIndices.first)) == tmpLRparseTable.end())
1310                        ||
```

```
1311                        (
1312                            tmpLRparseTable.find(std::make_pair(nowNodeIndex,
    symbolIndices.first)) != tmpLRparseTable.end()
1313                            &&
1314                            actionEq
1315                            (
1316                                tmpLRparseTable.at(std::make_pair(nowNodeIndex,
     symbolIndices.first)),
1317                                std::make_pair
1318                                (
1319                                    ACTION::SHIFT,
1320                                    // 下一节点
1321                                    std::any(newNodeIndex)
1322                                )
1323                            )
1324                        )
1325                    )
1326                    {
1327                        tmpLRparseTable[std::make_pair(nowNodeIndex,
    symbolIndices.first)] = \
1328                            std::make_pair
1329                                (
1330                                    ACTION::SHIFT,
1331                                    // 下一节点
1332                                    std::any(newNodeIndex)
1333                                );
1334                    }
1335                    else
1336                    {
1337                        // 手动冲突处理
1338                        std::cout << "\033[1m(Parser Generator)\033[0m
    \033[1;35mwarning:\033[0m conflict at (" << nowNodeIndex << ", " <<
    symbolIndices.first << ")" << std::endl;
1339                        HANDLINGFLAG flag = conflictHandling
1340                        (
1341                            tmpLRparseTable.at(std::make_pair(nowNodeIndex,
    symbolIndices.first)),
```

```
1342                                std::make_pair
1343                                    (
1344                                        ACTION::SHIFT,
1345                                        // 下一节点
1346                                        std::any(newNodeIndex)
1347                                    )
1348                            );
1349                            if(flag == HANDLINGFLAG::OVERWRITE)
1350                            {
1351                                tmpLRparseTable.at(std::make_pair(nowNodeIndex,
    symbolIndices.first)) =
1352                                    std::make_pair
1353                                        (
1354                                            ACTION::SHIFT,
1355                                            // 下一节点
1356                                            std::any(newNodeIndex)
1357                                        );
1358                            }
1359                            else if(flag == HANDLINGFLAG::ABORT)
1360                            {
1361                                std::cout << "Abort." << std::endl;
1362                                return false;
1363                            }
1364                        }
1365                    }
1366                    // 是非终结符 - 转移
1367                    else
1368                    // (grammar.isNonTerminal(symbolIndices.first))
1369                    {
1370                        if
1371                        (
1372                            (tmpLRparseTable.find(std::make_pair(nowNodeIndex,
    symbolIndices.first)) == tmpLRparseTable.end())
1373                            ||
1374                            (
1375                                tmpLRparseTable.find(std::make_pair(nowNodeIndex,
    symbolIndices.first)) != tmpLRparseTable.end()
```

```
1376                                &&
1377                                actionEq
1378                                (
1379                                    tmpLRparseTable.at(std::make_pair(nowNodeIndex,
     symbolIndices.first)),
1380                                    std::make_pair
1381                                    (
1382                                        ACTION::GOTO,
1383                                        // 下一节点
1384                                        std::any(newNodeIndex)
1385                                    )
1386                                )
1387                            )
1388                        )
1389                        {
1390                            tmpLRparseTable[std::make_pair(nowNodeIndex,
    symbolIndices.first)] = \
1391                                std::make_pair
1392                                    (
1393                                        ACTION::GOTO,
1394                                        // 下一节点
1395                                        std::any(newNodeIndex)
1396                                    );
1397                        }
1398                        else
1399                        {
1400                            // 手动冲突处理
1401                            std::cout << "\033[1m(Parser Generator)\033[0m
    \033[1;35mwarning:\033[0m conflict at (" << nowNodeIndex << ", " <<
    symbolIndices.first << ")" << std::endl;
1402                            HANDLINGFLAG flag = conflictHandling
1403                            (
1404                                tmpLRparseTable.at(std::make_pair(nowNodeIndex,
    symbolIndices.first)),
1405                                std::make_pair
1406                                    (
1407                                        ACTION::GOTO,
```

```
1408                                    // 下一节点
1409                                    std::any(newNodeIndex)
1410                                )
1411                            );
1412                            if(flag == HANDLINGFLAG::OVERWRITE)
1413                            {
1414                                tmpLRparseTable.at(std::make_pair(nowNodeIndex,
    symbolIndices.first)) =
1415                                    std::make_pair
1416                                    (
1417                                        ACTION::GOTO,
1418                                        // 下一节点
1419                                        std::any(newNodeIndex)
1420                                    );
1421                            }
1422                            else if(flag == HANDLINGFLAG::ABORT)
1423                            {
1424                                std::cout << "Abort." << std::endl;
1425                                return false;
1426                            }
1427                        }
1428                    }
1429                }
1430            }
1431
1432            LRparseTable = tmpLRparseTable;
1433            return true;
1434        };
1435
1436        return calcFAAndParseTable();
1437    }
1438
1439    // 打印 LR 表格
1440    void Parser::printLRInternalTables(std::ostream & out)
1441    {
1442        // out << "This is an LR(0) grammar." << std::endl;
1443        for(const auto & tableTerm : LRparseTable)
```

```
1444     {
1445         out << "M [ " << tableTerm.first.first
1446             << " , " << (tableTerm.first.second == "" ? "\"\"" : tableTerm.
    first.second) << " ] = ";
1447 
1448         if(tableTerm.second.first == ACTION::SHIFT)
1449         {
1450             out << "SHIFT " << std::any_cast<size_t>(tableTerm.second.second);
1451         }
1452         else if(tableTerm.second.first == ACTION::REDUCE)
1453         {
1454             out << "REDUCE ";
1455             const auto & production = std::any_cast<std::pair<Grammar::
    NonTerminalType, size_t>>(tableTerm.second.second);
1456             const auto & leftPart = production.first;
1457             const auto & rightPart = grammar.productions.at(leftPart)[
    production.second];
1458             out << leftPart << " -> ";
1459             for(const auto & symbol : rightPart)
1460             {
1461                 out << (symbol == "" ? "\"\"" : symbol) << " ";
1462             }
1463         }
1464         else if(tableTerm.second.first == ACTION::GOTO)
1465         {
1466             out << "GOTO " << std::any_cast<size_t>(tableTerm.second.second);
1467         }
1468         else if(tableTerm.second.first == ACTION::ACCEPT)
1469         {
1470             out << "ACCEPT";
1471         }
1472         out << std::endl;
1473     }
1474 }
1475 
1476 // LR 分析
1477 size_t Parser::LRParse(std::ostream & out)
```

```
1478 {
1479     // 重置
1480     lexer.rewind();
1481     // 分析栈
1482     // 状态栈
1483     std::vector<size_t> stateStack;
1484     // 符号栈
1485     std::vector<std::string> symbolStack;
1486 
1487     // 打印栈
1488     auto printStack = [&stateStack, &symbolStack , &out]() -> void
1489     {
1490         out << "State stack: ";
1491         for(const auto & symbol : stateStack)
1492         {
1493             out << symbol << " ";
1494         }
1495         out << std::endl;
1496 
1497         out << "Symbol stack: ";
1498         for(const auto & symbol : symbolStack)
1499         {
1500             out << symbol << " ";
1501         }
1502         out << std::endl;
1503     };
1504 
1505     // 打印 token
1506     auto printToken = [&out](Types::TokenPair token) -> void
1507     {
1508         out << Shared::typeStrings.at(token.first) << " ";
1509         if(token.first >= Types::TokenType::INIT
1510             && token.first <= Types::TokenType::ENDOFFILE )
1511         {
1512             // out;
1513         }
1514         else if(token.first == Types::TokenType::KEYWORD )
```

```
1515        {
1516            out << "( " << std::any_cast<std::string>(token.second) << " )";
1517        }
1518        else if(token.first == Types::TokenType::IDENTIFIER )
1519        {
1520            out << "( " << Shared::idTable.at(std::any_cast<size_t>(token.
    second)) << " )";
1521        }
1522        else if(token.first >= Types::TokenType::INT_CONST
1523            && token.first <= Types::TokenType::STR_LITERAL )
1524        {
1525            out << "( " << Shared::constTable.at(std::any_cast<size_t>(token.
    second)) << " )";
1526        }
1527        else if(token.first >= Types::TokenType::OP_ADD
1528            && token.first <= Types::TokenType::OP_SCOPE )
1529        {
1530            out << "( " << std::any_cast<std::string>(token.second) << " )";
1531        }
1532        else if(token.first >= Types::TokenType::DELIM_DBQUOTE
1533            && token.first <= Types::TokenType::DELIM_QUESTION )
1534        {
1535            out << "( " << std::any_cast<std::string>(token.second) << " )";
1536        }
1537    };
1538
1539    // 初始化入栈
1540    stateStack.push_back(0);
1541    symbolStack.push_back(Shared::endOfFileChar);
1542
1543    // 错误
1544    size_t errorCount = 0;
1545
1546    // 指向第一个词
1547    auto token = lexer.getNextToken();
1548    while(true)
1549    {
```

```
1550            // 跳过非实义符号
1551            if(token.first == Types::TokenType::INIT)
1552            {
1553                token = lexer.getNextToken();
1554                continue;
1555            }
1556
1557            // 打印栈
1558            printStack();
1559            out << "Now token: ";
1560            printToken(token);
1561            out << std::endl;
1562
1563            // 如果词法错误，处理
1564            if(token.first == Types::TokenType::ERROR)
1565            {
1566                errorCount++;
1567                lexer.errorProcess(std::any_cast<Types::LexerError>(token.second));
1568                // 跳过本词
1569                token = lexer.getNextToken();
1570            }
1571            else
1572            {
1573                std::string tokenTypeStr = Shared::typeStrings.at(token.first);
1574                // 替换掉关键词
1575                if(tokenTypeStr == "KEYWORD")
1576                {
1577                    tokenTypeStr = std::any_cast<std::string>(token.second);
1578                }
1579                // 查表
1580                if(
1581                    LRparseTable.find(
1582                        std::make_pair(stateStack.back(), tokenTypeStr)
1583                    )
1584                    == LRparseTable.end()
1585                )
1586                {
```

```
1587                // 语法错误.
1588                errorCount++;
1589                // 错误信息
1590                std::string errorMessage = "unexpected token: " + tokenTypeStr;
1591                this -> errorProcess(Types::ParserError(lexer.getFilePos(),
    errorMessage));
1592                break;
1593            }
1594            else
1595            {
1596                auto action = LRparseTable[std::make_pair(stateStack.back(),
    tokenTypeStr)];
1597                // 接受
1598                if(action.first == ACTION::ACCEPT)
1599                {
1600                    out << "Use rule: Accept." << std::endl;
1601                    break;
1602                }
1603                // 移进
1604                else if(action.first == ACTION::SHIFT)
1605                {
1606                    out << "Use rule: Shift " << std::any_cast<size_t>(action.
    second) << std::endl << std::endl;
1607                    // 状态进栈
1608                    stateStack.push_back(std::any_cast<size_t>(action.second));
1609                    // 符号进栈
1610                    symbolStack.push_back(tokenTypeStr);
1611                    token = lexer.getNextToken();
1612                }
1613                // 归约
1614                else if(action.first == ACTION::REDUCE)
1615                {
1616                    const auto & production = std::any_cast<std::pair<Grammar::
    NonTerminalType, size_t>>(action.second);
1617                    const auto & leftPart = production.first;
1618                    const auto & rightPart = grammar.productions[leftPart][
    production.second];
```

```
1619                    out << "Use rule: Reduce " << leftPart << " -> ";
1620                    for(size_t i = 0; i < rightPart.size(); i++)
1621                    {
1622                        out << rightPart[i] << " ";
1623                    }
1624                    out << std::endl;
1625
1626                    // 右部有几个符号
1627                    size_t lenRightPart = rightPart.size();
1628                    // 同时弹栈
1629                    for(size_t i = 0; i < lenRightPart; i++)
1630                    {
1631                        stateStack.pop_back();
1632                        symbolStack.pop_back();
1633                    }
1634                    // 接着要转移
1635                    if(
1636                        (LRparseTable.find(
1637                            std::make_pair(stateStack.back(), leftPart)
1638                        )
1639                        == LRparseTable.end()) ||
1640                        (LRparseTable.find(
1641                            std::make_pair(stateStack.back(), leftPart)
1642                        )
1643                        != LRparseTable.end() &&
1644                            LRparseTable[std::make_pair(stateStack.back(),
    leftPart)].first != ACTION::GOTO)
1645                    )
1646                    {
1647                        // 语法错误.
1648                        errorCount++;
1649                        // 错误信息
1650                        std::string errorMessage = "GOTO error: " + leftPart;
1651                        this -> errorProcess(Types::ParserError(lexer.
    getFilePos(), errorMessage));
1652                        break;
1653                    }
```

```
1654                    else
1655                    {
1656                        action = LRparseTable[std::make_pair(stateStack.back(), leftPart)];
1657                        out << "Use rule: Goto " << std::any_cast<size_t>(action.second) << std::endl << std::endl;
1658                        // 状态进栈
1659                        stateStack.push_back(std::any_cast<size_t>(action.second));
1660                        // 符号进栈
1661                        symbolStack.push_back(leftPart);
1662                    }
1663                }
1664                else
1665                {
1666                    // 语法错误.
1667                    errorCount++;
1668                    // 错误信息
1669                    std::string errorMessage = "Invalid table term! ";
1670                    this -> errorProcess(Types::ParserError(lexer.getFilePos(), errorMessage));
1671                    break;
1672                }
1673            }
1674        }
1675    }
1676    return errorCount;
1677 }
1678
1679 // 保存 LR(1) 预测分析表
1680 bool Parser::saveLRParseTable(const std::string & fileName)
1681 {
1682    std::cout << "Saving parsing table..." << std::endl;
1683    // 打开文件
1684    std::fstream outStream(fileName, std::ios::out);
1685    if(!outStream.is_open())
1686    {
```

```
1687         std::cout << "\033[1m(Parser Generator)\033[0m \033[1;31merror:\033[0m
     \033[1m Can't open output file: "
1688             << fileName << "\033[0m" << std::endl;
1689         return false;
1690     }
1691     printLRInternalTables(outStream);
1692     outStream.close();
1693     std::cout << "Done." << std::endl;
1694     return true;
1695 }
1696 
1697 // 读取 LR(1) 预测分析表
1698 bool Parser::readLRParseTable(const std::string & fileName)
1699 {
1700     std::cout << "Reading parsing table..." << std::endl;
1701     // 打开文件
1702     std::fstream fileStream;
1703     fileStream.open(fileName);
1704     // 打不开文件
1705     if(!fileStream.is_open())
1706     {
1707         std::cout << "\033[1m(Parser Generator)\033[0m \033[1;31merror:\033[0m
     \033[1m Can't open grammar file: "
1708             << fileName << "\033[0m" << std::endl;
1709         return false;
1710     }
1711 
1712     // 循环读取
1713     while(!fileStream.eof())
1714     {
1715         // 一行
1716         std::string lineString;
1717         // 读进来一行
1718         std::getline(fileStream, lineString);
1719         if(lineString.empty())
1720         {
1721             continue;
```

```
1722            }
1723            // 建立字符串流
1724            std::stringstream lineStream(lineString);
1725            // 当前符号
1726            std::string nowSymbol;
1727
1728            // 状态
1729            size_t state = 0;
1730            // 符号
1731            Grammar::SymbolType symbol;
1732            // 动作
1733            std::pair<ACTION, std::any> action;
1734
1735            // 读取 M [ 部分
1736            lineStream >> nowSymbol;
1737            if(nowSymbol != "M")
1738                return false;
1739
1740            lineStream >> nowSymbol;
1741            if(nowSymbol != "[")
1742                return false;
1743
1744            // 读取一个状态
1745            lineStream >> state;
1746
1747            // 读取 ,
1748            lineStream >> nowSymbol;
1749            if(nowSymbol != ",")
1750                return false;
1751
1752            // 读取符号
1753            lineStream >> symbol;
1754            if(symbol == "\"\"")
1755                symbol = "";
1756            if(!grammar.isNonTerminal(symbol) && !grammar.isTerminal(symbol))
1757                return false;
1758
```

```
1759            // 读取 ] = 部分
1760            lineStream >> nowSymbol;
1761            if(nowSymbol != "]")
1762                return false;
1763
1764            lineStream >> nowSymbol;
1765            if(nowSymbol != "=")
1766                return false;
1767
1768            // 读取动作
1769            lineStream >> nowSymbol;
1770
1771            if(nowSymbol == "ACCEPT")
1772            {
1773                action.first = ACTION::ACCEPT;
1774                action.second = std::any(nullptr);
1775                LRparseTable[std::make_pair(state, symbol)] = action;
1776            }
1777            else if(nowSymbol == "SHIFT")
1778            {
1779                // 读取下一状态
1780                size_t shiftState = 0;
1781                lineStream >> shiftState;
1782                action.first = ACTION::SHIFT;
1783                action.second = std::any(shiftState);
1784                LRparseTable[std::make_pair(state, symbol)] = action;
1785            }
1786            else if(nowSymbol == "GOTO")
1787            {
1788                // 读取下一状态
1789                size_t shiftState = 0;
1790                lineStream >> shiftState;
1791                action.first = ACTION::GOTO;
1792                action.second = std::any(shiftState);
1793                LRparseTable[std::make_pair(state, symbol)] = action;
1794            }
1795            else if(nowSymbol == "REDUCE")
```

```
1796        {
1797            // 读取左端
1798            Grammar::NonTerminalType leftPart;
1799            lineStream >> leftPart;
1800            if(!grammar.isNonTerminal(leftPart))
1801                return false;
1802            // 读取箭头
1803            lineStream >> nowSymbol;
1804            if(nowSymbol != "->")
1805                return false;
1806            // 读取右端
1807            std::vector<Grammar::SymbolType> rightPart;
1808            while(lineStream >> nowSymbol)
1809            {
1810                // 空字符
1811                if(nowSymbol == "\"\"" || nowSymbol == "''")
1812                {
1813                    nowSymbol = "";
1814                }
1815                rightPart.push_back(nowSymbol);
1816            }
1817            // 是第几个
1818            size_t index = 0;
1819            size_t total = grammar.productions.at(leftPart).size();
1820            for(index = 0; index < total; index++)
1821            {
1822                if(grammar.productions.at(leftPart)[index] == rightPart)
1823                {
1824                    break;
1825                }
1826            }
1827            if(index == total)
1828                return false;
1829            action.first = ACTION::REDUCE;
1830            action.second = std::any(std::make_pair(leftPart, index));
1831            LRparseTable[std::make_pair(state, symbol)] = action;
1832        }
```

```
1833     }
1834     std::cout << "Done." << std::endl;
1835     return true;
1836 }
1837 
1838 // ------------------------ 其它函数 ------------------------
1839 // 错误处理
1840 void Parser::errorProcess(const Types::ParserError & error)
1841 {
1842     // 行列
1843     size_t row = error.first.first, col = error.first.second - 1;
1844 
1845     std::cout << "\033[1m" << lexer.getSrcName() << ":"
1846         << row << ":"
1847         << col << ": (Parser) \033[31merror: \033[0m\033[1m"
1848         << error.second << "\033[0m" << std::endl;
1849 
1850     std::cout << "    " << lexer.getInBuf() << std::endl;
1851     std::cout << "    ";
1852     for(size_t i = 1; i < col; i++)
1853     {
1854         std::cout << (lexer.getInBuf()[i - 1] == '\t' ? '\t' : ' ');
1855     }
1856     std::cout << "\033[1;2m^\033[0m" << std::endl;
1857 }
1858 
1859 // 打印 FIRST 表
1860 void Parser::printFirstTable(std::ostream & out)
1861 {
1862     out << "FIRST Table:" << std::endl;
1863     for(const auto & i : firstTable)
1864     {
1865         out << "FIRST(" << i.first << "): ";
1866         for(const auto & j : i.second)
1867         {
1868             if(j == "")
1869             {
```

```
1870                 out << "\"\"" << " ";
1871             }
1872             else
1873             {
1874                 out << j << " ";
1875             }
1876         }
1877         out << std::endl;
1878     }
1879     out << std::endl;
1880 };
1881 
1882 // 打印 FOLLOW 表
1883 void Parser::printFollowTable(std::ostream & out)
1884 {
1885     out << "FOLLOW Table:" << std::endl;
1886     for(const auto & i : followTable)
1887     {
1888         out << "FOLLOW(" << i.first << "): ";
1889         for(const auto & j : i.second)
1890         {
1891             out << j << " ";
1892         }
1893         out << std::endl;
1894     }
1895     out << std::endl;
1896 };
1897 
1898 // 打印预测分析表
1899 void Parser::printLL1ParseTable(std::ostream & out)
1900 {
1901     // out << "LL(1) Parse Table:" << std::endl;
1902     for(const auto & tableTerm : this -> LL1parseTable)
1903     {
1904         out << "M [ " << tableTerm.first.first << " , " << tableTerm.first.
    second << " ] = ";
1905         out << tableTerm.second.first << " -> ";
```

```
1906            for(const auto & rightSymbol : tableTerm.second.second)
1907            {
1908                if(rightSymbol == "")
1909                {
1910                    out << "\"\"" << " ";
1911                }
1912                else
1913                {
1914                    out << rightSymbol << " ";
1915                }
1916            }
1917            out << std::endl;
1918        }
1919        out << std::endl;
1920    };
1921
1922    #define INDEPENDENT_PARSER
1923    #ifdef INDEPENDENT_PARSER
1924    int main(int argc, char * argv[])
1925    {
1926        auto printUsage = []() -> void
1927        {
1928            std::cout << "Usage:\n  ./parser <filename> [options]" << std::endl;
1929            std::cout << "Options:\n  -h, --help\t\t\t Print help." << std::endl;
1930            std::cout << "  -LL, --LL(1)\t\t\t Use LL(1) parsing. (Default: Use LR
        (1) parsing.)" << std::endl;
1931            std::cout << "  -g, --grammar\t\t\t Set input grammar file name. (
        Default: Use internal grammar.)" << std::endl;
1932            std::cout << "  -ti, --table-in\t\t Use pre-calculated parsing table."
        << std::endl;
1933            std::cout << "  -to, --table-out\t\t Set output parsing table file name
        . (Default: 'LL(1).tbl' if LL(1) parsing, 'LR(1).tbl' if LR(1) parsing.)" <<
         std::endl;
1934            std::cout << "  -o, --output\t\t\t Set output file name. (Default: 'out
        .txt'.)" << std::endl;
1935        };
1936
```

```
1937     Parser parser;
1938     // 源文件名, 语法文件名, 输入/输出分析表文件名, 输出文件名
1939     std::string srcFileName, grammarFileName, inputTableFileName,
     outputTableFileName, outputFileName = "out.txt";
1940     // 输出文件流
1941     std::fstream outStream;
1942 
1943     enum class FlagIndex
1944     {
1945         // 设置语法文件名
1946         SET_GRAMMARFILE,
1947         // 设置输出文件名
1948         SET_OUTPUTFILE,
1949         // 设置 LL(1) 分析
1950         SET_LLPARSE,
1951         // 设置使用的输入/输出分析表
1952         SET_INPUTTABLEFILE,
1953         SET_OUTPUTTABLEFILE
1954     };
1955 
1956     // 设置相关 Flags
1957     std::bitset<8> setFlags = 0;
1958 
1959     if(argc <= 1)
1960     {
1961         std::cout << "\033[1m(Parser)\033[0m \033[1;31merror:\033[0m Wrong
     usage!" << std::endl;
1962         printUsage();
1963         exit(1);
1964     }
1965 
1966     for(int i = 1; i < argc; i++)
1967     {
1968         std::string cmd = std::string(argv[i]);
1969         if(cmd == "-h" || cmd == "--help")
1970         {
1971             printUsage();
```

```
1972            exit(0);
1973        }
1974        else if(cmd == "-LL" || cmd == "--LL(1)")
1975        {
1976            setFlags.set(size_t(FlagIndex::SET_LLPARSE));
1977        }
1978        else if(cmd == "-g" || cmd == "--grammar")
1979        {
1980            setFlags.set(size_t(FlagIndex::SET_GRAMMARFILE));
1981        }
1982        else if(cmd == "-ti" || cmd == "--table-in")
1983        {
1984            setFlags.set(size_t(FlagIndex::SET_INPUTTABLEFILE));
1985        }
1986        else if(cmd == "-to" || cmd == "--table-out")
1987        {
1988            setFlags.set(size_t(FlagIndex::SET_OUTPUTTABLEFILE));
1989        }
1990        else if(cmd == "-o" || cmd == "--output")
1991        {
1992            setFlags.set(size_t(FlagIndex::SET_OUTPUTFILE));
1993        }
1994        else if
1995            (
1996                (cmd.size() >= 1 && cmd[0] == '-') ||
1997                (cmd.size() >= 2 && cmd[0] == '-' && cmd[1] == '-')
1998            )
1999        {
2000            std::cout << "\033[1m(Parser)\033[0m \033[1;31merror:\033[0m Wrong
    parameter: " << cmd << std::endl;
2001            printUsage();
2002            exit(1);
2003        }
2004        else
2005        {
2006            if(i == 1)
2007            {
```

```
2008                srcFileName = cmd;
2009            }
2010            // 设置 token 文件
2011            if(setFlags.test(size_t(FlagIndex::SET_GRAMMARFILE)))
2012            {
2013                grammarFileName = cmd;
2014                setFlags.set(size_t(FlagIndex::SET_GRAMMARFILE), false);
2015            }
2016            // 设置输出文件
2017            if(setFlags.test(size_t(FlagIndex::SET_OUTPUTFILE)))
2018            {
2019                outputFileName = cmd;
2020                setFlags.set(size_t(FlagIndex::SET_OUTPUTFILE), false);
2021            }
2022            // 设置表格
2023            if(setFlags.test(size_t(FlagIndex::SET_INPUTTABLEFILE)))
2024            {
2025                inputTableFileName = cmd;
2026                setFlags.set(size_t(FlagIndex::SET_INPUTTABLEFILE), false);
2027            }
2028            if(setFlags.test(size_t(FlagIndex::SET_OUTPUTTABLEFILE)))
2029            {
2030                outputTableFileName = cmd;
2031                setFlags.set(size_t(FlagIndex::SET_OUTPUTTABLEFILE), false);
2032            }
2033        }
2034    }
2035
2036    // 打不开文件
2037    if(!parser.openFile(srcFileName))
2038    {
2039        std::cout << "\033[1m(Parser)\033[0m \033[1;31merror:\033[0m\033[1m Can
    't open source file: "
2040            << srcFileName << "\033[0m" << std::endl;
2041        exit(1);
2042    }
2043    outStream.open(outputFileName, std::ios::out);
```

```
2044     if(!outStream.is_open())
2045     {
2046         std::cout << "\033[1m(Parser)\033[0m \033[1;31merror:\033[0m\033[1m Can
    't open output file: "
2047             << outputFileName << "\033[0m" << std::endl;
2048         exit(1);
2049     }
2050 
2051     // 读取语法
2052     if(grammarFileName != "")
2053     {
2054         if(!parser.readGrammar(grammarFileName))
2055         {
2056             std::cout << "\033[1m(Parser Generator)\033[0m \033[1;31merror
    :\033[0m\033[1m Invalid grammar. \033[0m" << std::endl;
2057             exit(1);
2058         }
2059     }
2060     // 计算预测分析表
2061     size_t errorCount = 0;
2062 
2063     // 使用 LL(1) 分析
2064     if(setFlags.test(size_t(FlagIndex::SET_LLPARSE)))
2065     {
2066         // 读取文件
2067         if(!inputTableFileName.empty())
2068         {
2069             if(!parser.readLL1ParseTable(inputTableFileName))
2070             {
2071                 std::cout << "\033[1m(Parser Generator)\033[0m \033[1;31merror
    :\033[0m\033[1m Invalid LL(1) grammar. \033[0m" << std::endl;
2072                 exit(1);
2073             }
2074         }
2075         else if(!parser.calcLL1ParseTable())
2076         {
2077             std::cout << "\033[1m(Parser Generator)\033[0m \033[1;31merror
```

```
:\033[0m\033[1m Invalid LL(1) grammar. \033[0m" << std::endl;
2078            exit(1);
2079        }
2080
2081        // 打印一下
2082        //parser.printLL1InternalTables(std::cout);
2083        //parser.printLL1InternalTables(outStream);
2084        // 保存分析表
2085        if(inputTableFileName.empty() || !outputTableFileName.empty())
2086        {
2087            parser.saveLL1ParseTable(outputTableFileName.empty() ? "LL.tbl" :
    outputTableFileName);
2088        }
2089        std::cout << std::endl;
2090        outStream << std::endl;
2091        std::cout << "This is an LL(1) grammar." << std::endl;
2092        outStream << "This is an LL(1) grammar." << std::endl;
2093        std::cout << "Begin parsing..." << std::endl << std::endl;
2094        outStream << "Begin parsing..." << std::endl << std::endl;
2095        // 分析
2096        errorCount = parser.LL1Parse(std::cout);
2097        if(errorCount > 0)
2098        {
2099            std::cout << errorCount << " error(s) generated." << std::endl;
2100            exit(1);
2101        }
2102        parser.LL1Parse(outStream);
2103    }
2104    else
2105    {
2106        if(!inputTableFileName.empty())
2107        {
2108            if(!parser.readLRParseTable(inputTableFileName))
2109            {
2110                std::cout << "\033[1m(Parser Generator)\033[0m \033[1;31merror
    :\033[0m\033[1m Invalid LR grammar. \033[0m" << std::endl;
2111                exit(1);
```

```
2112            }
2113        }
2114        else if(!parser.calcLRParseTable())
2115        {
2116            std::cout << "\033[1m(Parser Generator)\033[0m \033[1;31merror
    :\033[0m\033[1m Invalid LR grammar. \033[0m" << std::endl;
2117            exit(1);
2118        }
2119
2120        // 打印一下
2121        //parser.printLRInternalTables(std::cout);
2122        //parser.printLRInternalTables(outStream);
2123        // 保存
2124        if(inputTableFileName.empty() || !outputTableFileName.empty())
2125        {
2126            parser.saveLRParseTable(outputTableFileName.empty() ? "LR.tbl" :
    outputTableFileName);
2127        }
2128        std::cout << std::endl;
2129        outStream << std::endl;
2130        std::cout << "This is an LR(1) grammar." << std::endl;
2131        outStream << "This is an LR(1) grammar." << std::endl;
2132        std::cout << "Begin parsing..." << std::endl << std::endl;
2133        outStream << "Begin parsing..." << std::endl << std::endl;
2134        // 分析
2135        errorCount = parser.LRParse(std::cout);
2136        if(errorCount > 0)
2137        {
2138            std::cout << errorCount << " error(s) generated." << std::endl;
2139            exit(1);
2140        }
2141        parser.LRParse(outStream);
2142    }
2143    return 0;
2144 }
2145 #endif
```

参考文献

[1] Programming languages —C (N1570, Committee Draft). http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.