编译原理 实验二 LL(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`、LL(1) 文法文件 `grammar.txt`。
2. 输出：解析文法得到的各符号的 FIRST 集合、各符号的 FOLLOW 集合、LL(1) 预测分析表、调用词法分析器的 LL(1) 分析过程中栈的变化以及每一步所用的产生式，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。**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确定性特例——LL(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。

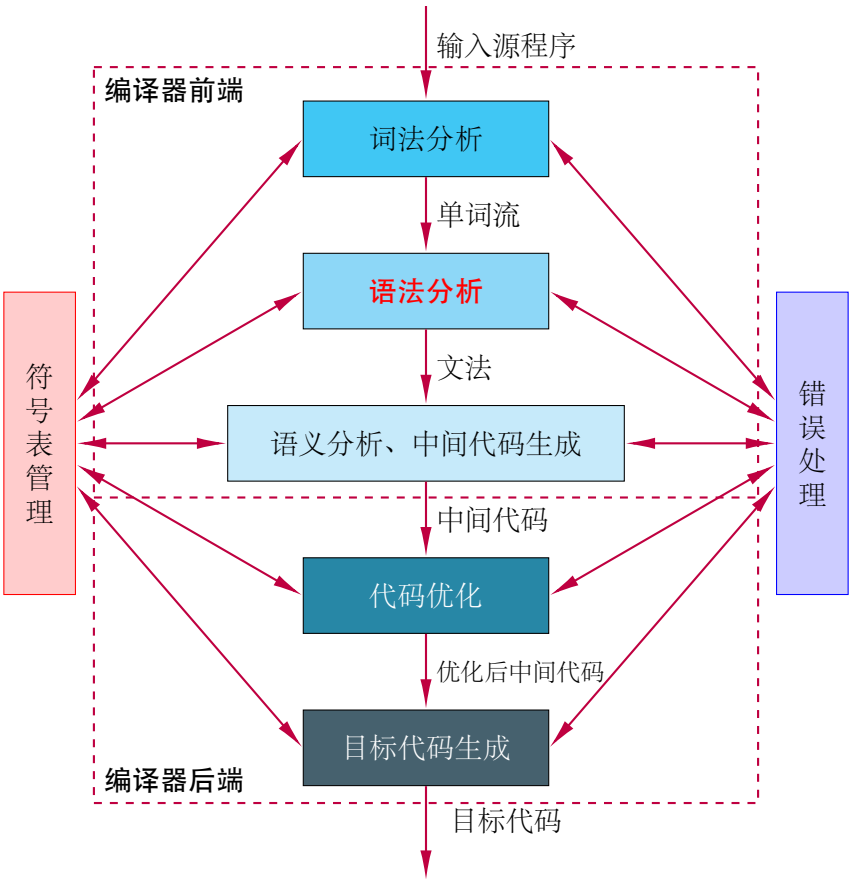

图 1: 编译器结构图

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 LL(1) 分析原理

LL(1) 文法的全称是“自左向右 (left to right)、最左推导 (leftmost derivation)、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无回溯分析的确定性的上下文无关文法，满足 LL(1) 文法的语言可以方便地使用栈进行语法分析。根据 LL(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. 计算每个符号的 FIRST 集合，形成 FIRST 表；
3. 计算每个符号的 FOLLOW 集合，形成 FOLLOW 表；
4. 根据上面两张表格，计算 LL(1) 预测分析表；
5. 根据分析表，按照 LL(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 LL(1) 语法分析算法设计

首先给出 LL(1) 分析法核心函数 `parse` 的伪代码：

Algorithm 1 LL(1) 分析的核心算法 `parse()`

```
// 先计算三个表
calcFirstTable()
calcFollowTable()
calcParseTable()
// 初始化
stack.push($)
stack.push(S)
token = getNextToken()
while True do
    if 该词有词法错误 then
        词法分析器错误处理
    else if stack.top() == '$' then
        // 栈已经空了
        if token == '$' then
            分析结束
            return
        else
            出错处理
        end if
    else if A = stack.top() 是非终结符 then
        // 栈顶为非终结符
        if M[A, token] 有产生式 then
            stack.pop()
            将产生式右侧各符号反序压栈
        else
            出错处理
        end if
    else if stack.top() 是终结符 then
        // 栈顶为终结符
        if stack.top() == token then
            stack.pop()
            token = getNextToken()
        else
            出错处理
        end if
    else
        出错处理
    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 FIRST 表的函数 `calcFirstTable()`：

Algorithm 2 计算 FIRST 表 `calcFirstTable()`

```
// 初始化
for 每一个非终结符 N do
    FIRST[N] = {}
end for
// 用来确定本轮是否有修改
flag = true
while flag == true do
    flag = false
    // 考虑每一个产生式，计算右部的 FIRST
    for 每一个产生式 P → X_1 X_2 ··· X_n do
        firstOfRightPart = {}
        for i = 1 to n do
            firstOfThisSymbol = {}
            // 推到最右端没有 ε 的符号
            if X_i 是终结符 then
                firstOfThisSymbol 中插入 X_i
            else
                firstOfThisSymbol = FIRST[X_i]
            end if
            firstOfRightPart = firstOfRightPart ∪ firstOfThisSymbol
            firstOfRightPart 中删除可能存在的 ε
            if firstOfThisSymbol 不包含 ε then
                break
            end if
            // 最后一个符号仍能推出 ε
            if i == n then
                firstOfRightPart 中加入 ε
            end if
        end for
        // 以上计算 firstOfRightPart 的这个子过程简记作 firstOfSymbols()
        FIRST[P] = FIRST[P] cup firstOfRightPart
        if 上面的集合的大小发生了变化 then
            flag = true
        end if
    end for
end while
```

3. 构造 FOLLOW 表的函数 `calcFollowTable()`：

Algorithm 3 计算 FOLLOW 表 `calcFollowTable()`

```
// 初始化
for 每一个非终结符 N do
    FOLLOW[N] = {}
end for
FOLLOW[StartSymbol] = {#}
// 用来确定本轮是否有修改
flag = true
while flag == true do
    flag = false
    // 考虑每一个产生式
    for 每一个产生式 P → X1X2 ··· Xn do
        // 考虑每一个产生式右端的符号
        for i = 1 to n do
            if Xi 是终结符 then
                continue
            else
                firstOfRightPart = firstOfSymbols(Xi+1...Xn)
                FOLLOW[Xi] = FOLLOW[Xi] ∪ firstOfRightPart
                if FOLLOW[Xi] 有空串 then
                    去掉空串，并将 FOLLOW[P] 并入 FOLLOW[Xi]
                end if
            end if
            if 上面的集合的大小发生了变化 then
                flag = true
            end if
        end for
    end for
end while
```

4. 构造预测分析表的函数 `calcParseTable()`:

Algorithm 4 计算预测分析表 `calcParseTable()`

```
// 考虑每一个产生式
for 每一个产生式 P → α do
    firstOfRightPart = firstOfSymbols(α)
    for 每一个 a 属于 firstOfRightPart do
        if a == ε then
            for 每一个 b 属于 FOLLOW(P) do
                将产生式 P → α 加入 M[P, b]
            end for
        else
            将 P → α 加入 M[P, a]
        end if
    end for
end for
```

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

运行语法分析器程序 `parser`，程序以 `grammar.txt` 语法文件作为语法规则以及 `test.cmm` 源文件作为输入文件，在终端的输出结果以及输出文件如下图所示：

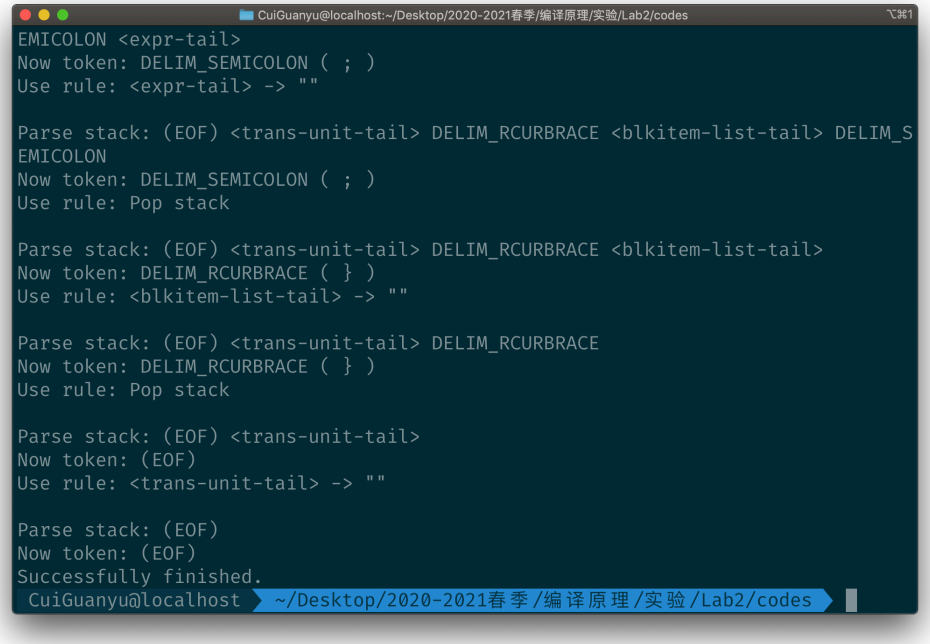

图 2: `parser` 测试

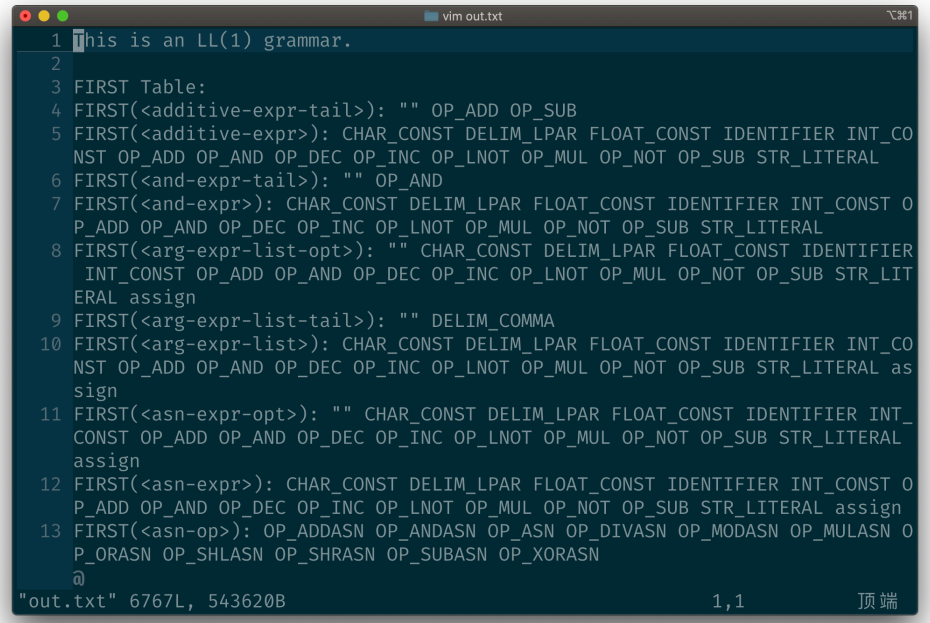

图 3: 输出文件

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；
- `-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）；
- `-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 LL(1) 文法，并且自动生成分析表，减少硬编码的工作；
3. 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
```

```
3
4  // (6.9)
5  // translation-unit:
6  //      external-declaration
7  //      translation-unit external-declaration
8  // 消除左递归
9  <trans-unit> -> <external-decl> <trans-unit-tail>
10 <trans-unit-tail> -> <external-decl> <trans-unit-tail>
11 <trans-unit-tail> -> ""
12
13 // (6.9)
14 // external-declaration:
15 //      function-definition
16 //      declaration
17 // 增加 def 关键字辅助消除冲突
18 <external-decl> -> def <func-def>
19 <external-decl> -> <decl>
20
21 // (6.9.1)
22 // function-definition:
23 //      declaration-specifiers declarator declaration-list_opt compound-statement
24 <func-def> -> <decl-spec> <declarator> <decl-list-opt> <comp-stmt>
25 <decl-list-opt> -> <decl-list>
26 <decl-list-opt> -> ""
27
28 // (6.9.1)
29 // declaration-list:
30 //      declaration
31 //      declaration-list declaration
32 // 消除左递归
33 <decl-list> -> <decl> <decl-list-tail>
34 <decl-list-tail> -> <decl> <decl-list-tail>
35 <decl-list-tail> -> ""
36
37 // -------------------- 语句 --------------------
38 // A.2.3 Statements
39
```

```
// (6.8)
// statement:
//       labeled-statement
//       compound-statement
//       expression-statement
//       selection-statement
//       iteration-statement
//       jump-statement
<stmt> -> <labeled-stmt>
<stmt> -> <comp-stmt>
<stmt> -> <expr-stmt>
<stmt> -> <sele-stmt>
<stmt> -> <iter-stmt>
<stmt> -> <jump-stmt>

// (6.8.1)
// labeled-statement:
//       identifier : statement (有冲突, 不用)
//       case constant-expression : statement
//       default : statement
// 冲突
// <labeled-stmt> -> IDENTIFIER DELIM_COLON <stmt>
<labeled-stmt> -> case <const-expr> DELIM_COLON <stmt>
<labeled-stmt> -> default DELIM_COLON <stmt>

// (6.8.2)
// compound-statement:
//       { block-item-list_opt }
<comp-stmt> -> DELIM_LCURBRACE <blkitem-list-opt> DELIM_RCURBRACE
<blkitem-list-opt> -> <blkitem-list>
<blkitem-list-opt> -> ""

// (6.8.2)
// block-item-list:
//       block-item
//       block-item-list block-item
// 消除左递归
```

```
<blkitem-list> -> <blkitem> <blkitem-list-tail>
<blkitem-list-tail> -> <blkitem> <blkitem-list-tail>
<blkitem-list-tail> -> ""

// (6.8.2)
// block-item:
//      declaration
//      statement
<blkitem> -> <decl>
<blkitem> -> <stmt>

// (6.8.3)
// expression-statement:
//      expression_opt ;
<expr-stmt> -> <expr-opt> DELIM_SEMICOLON
<expr-opt> -> <expr>
<expr-opt> -> ""

// (6.8.4)
// selection-statement:
//      if ( expression ) statement (为消除歧义, 不允许)
//      if ( expression ) statement else statement
//      switch ( expression ) statement
//
<sele-stmt> -> if DELIM_LPAR <expr> DELIM_RPAR <stmt> else <stmt>
<sele-stmt> -> switch DELIM_LPAR <expr> DELIM_RPAR <stmt>

// (6.8.5)
// iteration-statement:
//      while ( expression ) statement
//      do statement while ( expression ) ;
//      for ( expression_opt ; expression_opt ; expression_opt ) statement
//      for ( declaration expression_opt ; expression_opt ) statement
<iter-stmt> -> while DELIM_LPAR <expr> DELIM_RPAR <stmt>
<iter-stmt> -> do <stmt> while DELIM_LPAR <expr> DELIM_RPAR DELIM_SEMICOLON
<iter-stmt> -> for DELIM_LPAR <for-cond> DELIM_RPAR <stmt>
<for-cond> -> <expr-opt> DELIM_SEMICOLON <expr-opt> DELIM_SEMICOLON <expr-opt>
```

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

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


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

// (6.7)
// declaration:
//      declaration-specifiers init-declarator-list_opt ;
//      static_assert-declaration (不用)
// 消除左递归
<decl> -> <decl-spec> <init-declarator-list-opt> DELIM_SEMICOLON
<init-declarator-list-opt> -> <init-declarator-list>
<init-declarator-list-opt> -> ""

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

```
<decl-spec-opt> -> ""

// (6.7)
// init-declarator-list:
//      init-declarator
//      init-declarator-list , init-declarator
// 消除左递归
<init-declarator-list> -> <init-declarator> <init-declarator-list-tail>
<init-declarator-list-tail> -> DELIM_COMMA <init-declarator> <init-declarator-list-
   tail>
<init-declarator-list-tail> -> ""

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

// (6.7.1) (不用)

// (6.7.2)
// type-specifier:
//      void
//      char
//      short
//      int
//      long
//      float
//      double
//      signed
//      unsigned
//      _Bool (不用)
//      _Complex (不用)
//      atomic-type-specifier (不用)
//      struct-or-union-specifier (不用)
```

```
187 //      enum-specifier （不用）
188 //      typedef-name （不用）
189 <type-spec> -> void
190 <type-spec> -> char
191 <type-spec> -> short
192 <type-spec> -> int
193 <type-spec> -> long
194 <type-spec> -> float
195 <type-spec> -> double
196 <type-spec> -> signed
197 <type-spec> -> unsigned
198
199 // (6.7.2.1) - (6.7.5) （不用）
200
201 // (6.7.6)
202 // declarator:
203 //      pointer_opt direct-declarator
204 <declarator> -> <pointer-opt> <direct-declarator>
205 <pointer-opt> -> <pointer>
206 <pointer-opt> -> ""
207
208 // (6.7.6)
209 // direct-declarator:
210 //      identifier
211 //      ( declarator )
212 //      direct-declarator [ type-qualifier-list_opt assignment-expression_opt ]
213 //      direct-declarator [ static type-qualifier-list_opt assignment-expression ]
    （不用）
214 //      direct-declarator [ type-qualifier-list static assignment-expression ] （不
    用）
215 //      direct-declarator [ type-qualifier-list_opt * ] （不用）
216 //      direct-declarator ( parameter-type-list )
217 //      direct-declarator ( identifier-list_opt )
218 // 消除左递归
219 <direct-declarator> -> IDENTIFIER <direct-declarator-tail>
220 <direct-declarator> -> DELIM_LPAR <declarator> DELIM_RPAR <direct-declarator-tail>
221 <direct-declarator-tail> -> DELIM_LSQBRACKET <type-qual-list-opt> <asn-expr-opt>
```

```
   DELIM_RSQBRACKET <direct-declarator-tail>
<direct-declarator-tail> -> DELIM_LPAR <direct-declarator-in-par> DELIM_RPAR <
   direct-declarator-tail>
<direct-declarator-tail> -> ""
<asn-expr-opt> -> <asn-expr>
<asn-expr-opt> -> ""
<direct-declarator-in-par> -> <param-type-list>
<direct-declarator-in-par> -> <identifier-list-opt>
<identifier-list-opt> -> <identifier-list>
<identifier-list-opt> -> ""

// (6.7.6)
// pointer:
//      * type-qualifier-list_opt
//      * type-qualifier-list_opt pointer
// 提取公共左因子
<pointer> -> OP_MUL <type-qual-list-opt> <pointer-opt>
// type-qualifier 不用
// <type-qual-list-opt> -> <type-qual-list>
<type-qual-list-opt> -> ""

// (6.7.6)
// type-qualifier-list: (不用)
//      type-qualifier
//      type-qualifier-list type-qualifier
// <type-qual-list> -> <type-qual> <type-qual-list-tail>
// <type-qual-list-tail> -> <type-qual> <type-qual-list-tail>
// <type-qual-list-tail> -> ""

// (6.7.6)
// parameter-type-list:
//      parameter-list
//      parameter-list , ... (不用)
<param-type-list> -> <param-list>

// (6.7.6)
// parameter-list:
```

```
257 //      parameter-declaration
258 //      parameter-list , parameter-declaration
259 // 消除左递归
260 <param-list> -> <param-decl> <param-list-tail>
261 <param-list-tail> -> DELIM_COMMA <param-decl> <param-list-tail>
262 <param-list-tail> -> ""
263
264 // (6.7.6)
265 // parameter-declaration:
266 //      declaration-specifiers declarator
267 //      declaration-specifiers abstract-declarator_opt (不用)
268 <param-decl> -> <decl-spec> <param-decl-tail>
269 <param-decl-tail> -> <declarator>
270
271 // (6.7.6)
272 // identifier-list:
273 //      identifier
274 //      identifier-list , identifier
275 <identifier-list> -> IDENTIFIER <identifier-list-tail>
276 <identifier-list-tail> -> DELIM_COMMA IDENTIFIER <identifier-list-tail>
277 <identifier-list-tail> -> ""
278
279 // (6.7.7) - (6.7.8) (不用)
280
281 // (6.7.9)
282 // initializer:
283 //      assignment-expression
284 //      { initializer-list }
285 //      { initializer-list , } (不用)
286 <initializer> -> <asn-expr>
287 <initializer> -> DELIM_LCURBRACE <initializer-list> DELIM_RCURBRACE
288
289 // (6.7.9)
290 // initializer-list
291 //      designation_opt initializer
292 //      initializer-list , designation_opt initializer
293 // 消除左递归
```

```
<initializer-list> -> <designation-opt> <initializer> <initializer-list-tail>
<initializer-list-tail> -> DELIM_COMMA <designation-opt> <initializer> <initializer
    -list-tail>
<initializer-list-tail> -> ""
<designation-opt> -> <designation>
<designation-opt> -> ""

// (6.7.9)
// designation:
//      designator-list =
<designation> -> <designator-list> OP_ASN

// (6.7.9)
// designator-list:
//      designator
//      designator-list designator
// 消除左递归
<designator-list> -> <designator> <designator-list-tail>
<designator-list-tail> -> <designator> <designator-list-tail>
<designator-list-tail> -> ""

// (6.7.9)
// designator:
//      [ constant-expression ]
//      . identifier
<designator> -> DELIM_LSQBRACKET <const-expr> DELIM_RSQBRACKET
<designator> -> OP_DOT IDENTIFIER


// -------------------- 表达式 --------------------
// A.2.1 Expressions
// (6.5.17)
// expression:
//      assignment-expression
//      expression , assignment-expression
// 消除左递归
<expr> -> <asn-expr> <expr-tail>
```

```
<expr-tail> -> DELIM_COMMA <asn-expr> <expr-tail>
<expr-tail> -> ""

// (6.5.16)
// assignment-expression:
//      conditional-expression
//      unary-expression assignment-operator assignment-expression
<asn-expr> -> <cond-expr>
// 增加 assign 关键字辅助消除冲突
<asn-expr> -> assign <unary-expr> <asn-op> <asn-expr>

// (6.5.16)
// assignment-operator:
//      = *= /= %= += -= <<= >>= &= ^= |=
<asn-op> -> OP_ASN
<asn-op> -> OP_MULASN
<asn-op> -> OP_DIVASN
<asn-op> -> OP_MODASN
<asn-op> -> OP_ADDASN
<asn-op> -> OP_SUBASN
<asn-op> -> OP_SHLASN
<asn-op> -> OP_SHRASN
<asn-op> -> OP_ANDASN
<asn-op> -> OP_XORASN
<asn-op> -> OP_ORASN

// (6.6)
// constant-expression:
//      conditional-expression
<const-expr> -> <cond-expr>

// (6.5.15)
// conditional-expression:
//      logical-OR-expression
//      logical-OR-expression ? expression : conditional-expression
// 提取公共左因子
<cond-expr> -> <lor-expr> <cond-expr-tail>
```

```
<cond-expr-tail> -> DELIM_QUESTION <expr> DELIM_COLON <cond-expr>
<cond-expr-tail> -> ""

// (6.5.14)
// logical-OR-expression:
//      logical-AND-expression
//      logical-OR-expression || logical-AND-expression
// 消除左递归
<lor-expr> -> <land-expr> <lor-expr-tail>
<lor-expr-tail> -> OP_LOR <land-expr> <lor-expr-tail>
<lor-expr-tail> -> ""

// (6.5.13)
// logical-AND-expression:
//      inclusive-OR-expression
//      logical-AND-expression && inclusive-OR-expression
// 消除左递归
<land-expr> -> <inc-or-expr> <land-expr-tail>
<land-expr-tail> -> OP_LAND <inc-or-expr> <land-expr-tail>
<land-expr-tail> -> ""

// (6.5.12)
// inclusive-OR-expression:
//      exclusive-OR-expression
//      inclusive-OR-expression | exclusive-OR-expression
// 消除左递归
<inc-or-expr> -> <exc-or-expr> <inc-or-expr-tail>
<inc-or-expr-tail> -> OP_OR <exc-or-expr> <inc-or-expr-tail>
<inc-or-expr-tail> -> ""
// (6.5.11)
// exclusive-OR-expression:
//      AND-expression
//      exclusive-OR-expression ^ AND-expression
// 消除左递归
<exc-or-expr> -> <and-expr> <exc-or-expr-tail>
<exc-or-expr-tail> -> OP_XOR <and-expr> <exc-or-expr-tail>
<exc-or-expr-tail> -> ""
```

```
404
405 // (6.5.10)
406 // AND-expression:
407 //      equality-expression
408 //      AND-expression & equality-expression
409 // 消除左递归
410 <and-expr> -> <eq-expr> <and-expr-tail>
411 <and-expr-tail> -> OP_AND <eq-expr> <and-expr-tail>
412 <and-expr-tail> -> ""
413
414 // (6.5.9)
415 // equality-expression:
416 //      relational-expression
417 //      equality-expression == relational-expression
418 //      equality-expression != relational-expression
419 // 消除左递归
420 <eq-expr> -> <rel-expr> <eq-expr-tail>
421 <eq-expr-tail> -> OP_EQ <rel-expr> <eq-expr-tail>
422 <eq-expr-tail> -> OP_NEQ <rel-expr> <eq-expr-tail>
423 <eq-expr-tail> -> ""
424
425 // (6.5.8)
426 // relational-expression:
427 //      shift-expression
428 //      relational-expression < shift-expression
429 //      relational-expression > shift-expression
430 //      relational-expression <= shift-expression
431 //      relational-expression >= shift-expression
432 // 消除左递归
433 <rel-expr> -> <shift-expr> <rel-expr-tail>
434 <rel-expr-tail> -> OP_LT <shift-expr> <rel-expr-tail>
435 <rel-expr-tail> -> OP_GT <shift-expr> <rel-expr-tail>
436 <rel-expr-tail> -> OP_LE <shift-expr> <rel-expr-tail>
437 <rel-expr-tail> -> OP_GE <shift-expr> <rel-expr-tail>
438 <rel-expr-tail> -> ""
439
440 // (6.5.7)
```

```
// shift-expression:
//       additive-expression
//       shift-expression << additive-expression
//       shift-expression >> additive-expression
// 消除左递归
<shift-expr> -> <additive-expr> <shift-expr-tail>
<shift-expr-tail> -> OP_SHL <additive-expr> <shift-expr-tail>
<shift-expr-tail> -> OP_SHR <additive-expr> <shift-expr-tail>
<shift-expr-tail> -> ""

// (6.5.6)
// additive-expression:
//       multiplicative-expression
//       additive-expression + multiplicative-expression
//       additive-expression - multiplicative-expression
// 消除左递归
<additive-expr> -> <multiplicative-expr> <additive-expr-tail>
<additive-expr-tail> -> OP_ADD <multiplicative-expr> <additive-expr-tail>
<additive-expr-tail> -> OP_SUB <multiplicative-expr> <additive-expr-tail>
<additive-expr-tail> -> ""

// (6.5.5)
// multiplicative-expression:
//       cast-expression
//       multiplicative-expression * cast-expression
//       multiplicative-expression / cast-expression
//       multiplicative-expression % cast-expression
// 消除左递归
<multiplicative-expr> -> <cast-expr> <multiplicative-expr-tail>
<multiplicative-expr-tail> -> OP_MUL <cast-expr> <multiplicative-expr-tail>
<multiplicative-expr-tail> -> OP_DIV <cast-expr> <multiplicative-expr-tail>
<multiplicative-expr-tail> -> OP_MOD <cast-expr> <multiplicative-expr-tail>
<multiplicative-expr-tail> -> ""

// (6.5.4)
// cast-expression:
//       unary-expression
```

```
478 //      ( type-name ) cast-expression (冲突, 不用)
479 <cast-expr> -> <unary-expr>
480 // 此句冲突
481 // <cast-expr> -> DELIM_LPAR <type-name> DELIM_RPAR <cast-expr>
482
483 // (6.5.3)
484 // unary-expression:
485 //      postfix-expression
486 //      ++ unary-expression
487 //      -- unary-expression
488 //      unary-operator cast-expression
489 //      sizeof unary-expression (不用)
490 //      sizeof ( type-name ) (不用)
491 //      _Alignof ( type-name ) (不用)
492 <unary-expr> -> <postfix-expr>
493 <unary-expr> -> OP_INC <unary-expr>
494 <unary-expr> -> OP_DEC <unary-expr>
495 <unary-expr> -> <unary-op> <cast-expr>
496 // 提取公共左因子
497 // 略微修改, sizeof ( unary-expression | type-name ) 以消除冲突
498 // <unary-expr> -> sizeof DELIM_LPAR <sizeof-tail> DELIM_RPAR
499 // <sizeof-tail> -> <unary-expr>
500 // <sizeof-tail> -> <type-name>
501
502 // (6.5.3)
503 // unary-operator:
504 //      & * + - ~ !
505 <unary-op> -> OP_AND
506 <unary-op> -> OP_MUL
507 <unary-op> -> OP_ADD
508 <unary-op> -> OP_SUB
509 <unary-op> -> OP_NOT
510 <unary-op> -> OP_LNOT
511
512 // (6.5.2)
513 // argument-expression-list:
514 //      assignment-expression
```

```
515 //      argument-expression-list , assignment-expression
516 <arg-expr-list> -> <asn-expr> <arg-expr-list-tail>
517 <arg-expr-list-tail> -> DELIM_COMMA <asn-expr> <arg-expr-list-tail>
518 <arg-expr-list-tail> -> ""
519
520 // (6.5.2)
521 // postfix-expression:
522 //      primary-expression
523 //      postfix-expression [ expression ]
524 //      postfix-expression ( argument-expression_opt )
525 //      postfix-expression . identifier
526 //      postfix-expression -> identifier
527 //      postfix-expression ++
528 //      postfix-expression --
529 //      ( type-name ) { initializer-list } (冲突, 不用)
530 //      ( type-name ) { initializer-list , } (不用)
531 // 消除左递归
532 <postfix-expr> -> <prim-expr> <postfix-expr-tail>
533 // 此句冲突
534 // <postfix-expr> -> DELIM_LPAR <type-name> DELIM_RPAR DELIM_LCURBRACE <initializer
    -list> DELIM_RCURBRACE <postfix-expr-tail>
535 <postfix-expr-tail> -> DELIM_LSQBRACKET <expr> DELIM_RSQBRACKET <postfix-expr-tail>
536 <postfix-expr-tail> -> DELIM_LPAR <arg-expr-list-opt> DELIM_RPAR <postfix-expr-tail
    >
537 <postfix-expr-tail> -> OP_DOT IDENTIFIER <postfix-expr-tail>
538 <postfix-expr-tail> -> OP_ARROW IDENTIFIER <postfix-expr-tail>
539 <postfix-expr-tail> -> OP_INC <postfix-expr-tail>
540 <postfix-expr-tail> -> OP_DEC <postfix-expr-tail>
541 <postfix-expr-tail> -> ""
542 <arg-expr-list-opt> -> <arg-expr-list>
543 <arg-expr-list-opt> -> ""
544
545 // (6.5.1.1) 不用
546
547 // (6.5.1)
548 // primary-expression:
549 //      identifier
```

```
//      constant
//      string-literal
//      ( expression )
//      generic-selection (不用)
<prim-expr> -> IDENTIFIER
<prim-expr> -> <constant>
<prim-expr> -> DELIM_LPAR <expr> DELIM_RPAR
<constant> -> <num-const>
<constant> -> CHAR_CONST
<constant> -> STR_LITERAL
<num-const> -> INT_CONST
<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>;
```

```
17
18      // 起始符号
19      using StartSymbolType = NonTerminalType;
20
21      // 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)
22      using Productions = std::unordered_map<NonTerminalType, std::vector<std::
        vector<SymbolType>>>;
23
24      // 一条产生式(二元组，左端和右端符号)
25      using ProductionType = std::pair<NonTerminalType, std::vector<SymbolType>>;
26
27      // 四元组
28      // 非终结符
29      NonTerminals nonTerminals;
30      // 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
55 class Parser
56 {
57     // ------------------------- 公有成员 -------------------------
58         // 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     public:
66
67         // ------------------------ 构造函数 ------------------------
68         // 默认构造函数
69         Parser();
70         // 复制构造(标记删除)
71         Parser(const Parser & other) = delete;
72
73         // ------------------------ 析构函数 ------------------------
74         ~Parser();
75
76         // ------------------------ 成员函数 ------------------------
77         // 关联文件
78         bool openFile(const std::string & srcName);
79         // 关闭文件
80         void closeFile();
81         // 读取语法
82         bool readGrammar(const std::string & grmName);
83         // 计算 LL(1) 预测分析表
84         bool calcLL1ParseTable();
85         // 打印内部表格
86         void printInternalTables(std::ostream & out = std::cout);
87         // 解析
```

```
88          size_t parse(std::ostream & out = std::cout);
89          // 错误处理
90          void errorProcess(const Types::ParserError & error);
91      private:
92          // 打印 FIRST
93          void printFirstTable(std::ostream & out);
94          // 打印 FOLLOW
95          void printFollowTable(std::ostream & out);
96          // 打印预测分析表
97          void printParseTable(std::ostream & out);
98
99          // 词法分析器
100         Lexer lexer;
101         // 上下文无关文法
102         Grammar grammar =
103         {
104             {"S", "E", "T", "G", "F", "H"},
105             {"OP_ADD", "OP_SUB", "OP_MUL", "OP_DIV", "DELIM_LPAR", "DELIM_RPAR"
    , "IDENTIFIER", ""},
106             "S",
107             {
108                 { "S", {{"E"}} },
109                 { "E", {{"T", "G"}} },
110                 { "G", {{"OP_ADD", "T", "G"}, {"OP_SUB", "T", "G"}, {""}} },
111                 { "T", {{"F", "H"}} },
112                 { "H", {{"OP_MUL", "F", "H"}, {"OP_DIV", "F", "H"}, {""}} },
113                 { "F", {{"DELIM_LPAR", "E", "DELIM_RPAR"}, {"IDENTIFIER"}} }
114             }
115         };
116         // 两个表
117         FirstTableType firstTable;
118         FollowTableType followTable;
119         // LL(1) 预测分析表
120         LL1ParseTableType parseTable;
121
122 };
123
```

```
124 #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 
111     grammar = {nonTerminals, terminals, startSymbol, productions};
112     return true;
113 }
114 
115 // 计算解析表
116 bool Parser::calcLL1ParseTable()
117 {
118     // 两个表
119     FirstTableType tmpFirstTable;
120     FollowTableType tmpFollowTable;
121 
122     // 最大迭代次数
123     size_t maxIteration = 1e6;
124 
125     // 根据当前 FIRST 表，得到一串符号的 FIRST
126     auto firstOfSymbols = [this, &tmpFirstTable](const std::vector<Grammar::
    SymbolType> & symbols) -> std::set<Grammar::TerminalType>
127     {
128         if(symbols.empty())
129         {
130             return {""};
131         }
132         // 一开始，空集合
133         std::set<Grammar::TerminalType> s = {};
134         for(size_t i = 0; i < symbols.size(); i++)
135         {
136             std::set<Grammar::TerminalType> firstOfThisSymbol;
137             // 如果是终结符
138             if(this -> grammar.isTerminal(symbols[i]))
139             {
140                 firstOfThisSymbol.insert(symbols[i]);
141             }
```

```
142             // 查表得到当前符号的 First
143             else if(grammar.isNonTerminal(symbols[i]))
144             {
145                 // 没有对应产生式
146                 if(tmpFirstTable.find(symbols[i]) == tmpFirstTable.end())
147                 {
148                     std::cout << "Exception: In function firstOfSymbols." <<
    std::endl;
149                     return {};
150                 }
151                 firstOfThisSymbol = tmpFirstTable[symbols[i]];
152             }
153             else
154             {
155                 std::cout << "Exception: In function firstOfSymbols." << std::
    endl;
156                 return {};
157             }
158 
159             // 将除去 空字 的 First 加入
160             s.insert(firstOfThisSymbol.begin(), firstOfThisSymbol.end());
161             s.erase("");
162             // 当前字的 First 不含空, 则停止
163             if(firstOfThisSymbol.find("") == firstOfThisSymbol.end())
164             {
165                 break;
166             }
167             // 最后一个还含有空, 则加入空
168             if(i == symbols.size() - 1)
169             {
170                 s.insert("");
171             }
172         }
173         return s;
174     };
175 
176     // 计算 FISRT 表
```

```
177     auto calcFirstTable = [this, maxIteration, &tmpFirstTable, firstOfSymbols
    ]() -> bool
178     {
179         size_t iter = 0;
180         // 初始化
181         for(const auto & symbol : grammar.nonTerminals)
182         {
183             // 各符号都是空集合
184             tmpFirstTable[ symbol ] = {};
185         }
186         // 判断是否有更新
187         bool modifiedFlag = true;
188         while(modifiedFlag)
189         {
190             iter++;
191             modifiedFlag = false;
192             // 对每个产生式
193             for(const auto & production : this -> grammar.productions)
194             {
195                 // 左半部
196                 const auto & leftPart = production.first;
197                 // 对于右侧每一个产生式
198                 for(const auto & rightPart : production.second)
199                 {
200                     // 原集合大小
201                     size_t oldSize = tmpFirstTable[leftPart].size();
202                     // 计算它们的 First
203                     std::set<Grammar::TerminalType> firstOfRightPart =
    firstOfSymbols(rightPart);
204                     // 合并集合
205                     tmpFirstTable[leftPart].insert(firstOfRightPart.begin(),
    firstOfRightPart.end());
206                     // 增加符号后的大小
207                     size_t newSize = tmpFirstTable[leftPart].size();
208                     if(newSize != oldSize)
209                     {
210                         modifiedFlag = true;
```

```
211                     }
212                 }
213             }
214             if(iter > maxIteration)
215             {
216                 return false;
217             }
218         }
219         firstTable = tmpFirstTable;
220         return true;
221     };
222 
223     // 计算 FOLLOW 表
224     auto calcFollowTable = [this, maxIteration, &tmpFirstTable, &tmpFollowTable
    , firstOfSymbols]() -> bool
225     {
226         size_t iter = 0;
227         // 初始化
228         for(const auto & symbol : grammar.nonTerminals)
229         {
230             // 各符号都是空集合
231             tmpFollowTable[ symbol ] = {};
232         }
233         // Follow 表起始字符为结束字符
234         tmpFollowTable[grammar.startSymbol] = {Shared::endOfFileChar};
235         // 判断是否有更新
236         bool modifiedFlag = true;
237         while(modifiedFlag)
238         {
239             iter++;
240             modifiedFlag = false;
241             // 对每一个产生式
242             for(const auto & production : grammar.productions)
243             {
244                 // 计算左部、右部
245                 const auto & leftPart = production.first;
246                 for(const auto & rightPart : production.second)
```

```
247                 {
248                     // 考虑每个产生式右边的非终结符
249                     for(size_t i = 0; i < rightPart.size(); i++)
250                     {
251                         // 跳过终结符
252                         if(grammar.isTerminal(rightPart[i]))
253                         {
254                             continue;
255                         }
256                         // 既不是终结符也不是非终结符
257                         if(!grammar.isNonTerminal(rightPart[i]))
258                         {
259                             std::cout << "Exception: In function
    calcFollowTable." << std::endl;
260                             return false;
261                         }
262                         size_t oldSize = tmpFollowTable[rightPart[i]].size();
263                         // 获取右边的符号
264                         std::vector<Grammar::SymbolType> rightSymbols(rightPart
    .begin() + i + 1, rightPart.end());
265                         // 计算右边的符号串的 First
266                         std::set<Grammar::TerminalType> firstOfRightSymbols =
    firstOfSymbols(rightSymbols);
267                         // 并进去
268                         tmpFollowTable[rightPart[i]].insert(firstOfRightSymbols
    .begin(), firstOfRightSymbols.end());
269                         // 如果有空串，则去掉，并且把产生式左边的符号的 Follow
    加进去
270                         if(tmpFollowTable[rightPart[i]].find("") !=
    tmpFollowTable[rightPart[i]].end())
271                         {
272                             tmpFollowTable[rightPart[i]].erase("");
273                             tmpFollowTable[rightPart[i]].insert(tmpFollowTable[
    leftPart].begin(), tmpFollowTable[leftPart].end());
274                         }
275                         // 新的大小
276                         size_t newSize = tmpFollowTable[rightPart[i]].size();
```

```
277                         if(newSize != oldSize)
278                         {
279                             modifiedFlag = true;
280                         }
281                     }
282                 }
283             }
284             if(iter > maxIteration)
285             {
286                 return false;
287             }
288         }
289         followTable = tmpFollowTable;
290         return true;
291     };
292 
293     // 计算预测分析表
294     auto calcParseTable = [this, &tmpFirstTable, &tmpFollowTable,
    firstOfSymbols]() -> bool
295     {
296         // 开始填临时分析表
297         LL1ParseTableType tmpParseTable;
298         // 考虑每一条产生式
299         for(const auto & production : grammar.productions)
300         {
301             // 左侧
302             const auto & leftPart = production.first;
303             // 右侧每一条
304             for(const auto & rightPart : production.second)
305             {
306                 // 产生式
307                 Grammar::ProductionType p(leftPart, rightPart);
308                 // 计算右部符号的 First 集合
309                 const std::set<Grammar::TerminalType> & firstOfRightSymbols =
    firstOfSymbols(rightPart);
310                 // 考虑每一个 First 中的符号 a
311                 for(const auto & firstTerminal : firstOfRightSymbols)
```

```
312                 {
313                     // 如果有空串，则要计算左边的 Follow
314                     if(firstTerminal == "")
315                     {
316                         // 左边符号的 Follow
317                         const auto & followOfLeftSymbol = followTable[leftPart
    ];
318                         //  b in Follow(A), 将 A -> alpha 加入 M[A, b]
319                         for(const auto & followTerminal : followOfLeftSymbol)
320                         {
321                             if(
322                                 // 如果 M[A, b] 没有值
323                                 tmpParseTable.find(
324                                     std::make_pair(leftPart, followTerminal)
325                                 ) == tmpParseTable.end()
326                             )
327                             {
328                                 // 加入
329                                 tmpParseTable[std::make_pair(leftPart,
    followTerminal)] = p;
330                             }
331                             // 如果有值，但不冲突
332                             else if(tmpParseTable[std::make_pair(leftPart,
    followTerminal)] == p)
333                             {
334 
335                             }
336                             // 冲突，这不是 LL(1) 文法，返回
337                             else
338                             {
339                                 std::cout << "Collide: " << leftPart << " " <<
    followTerminal << std::endl;
340                                 return false;
341                             }
342                         }
343                     }
344                     else
```

```
345                     {
346                         if(
347                             // 如果 M[A, a] 没有值
348                             tmpParseTable.find(
349                                 std::make_pair(leftPart, firstTerminal)
350                             ) == tmpParseTable.end()
351                         )
352                         {
353                             // 加入
354                             tmpParseTable[std::make_pair(leftPart,
    firstTerminal)] = p;
355                         }
356                         // 如果有值，但不冲突
357                         else if(tmpParseTable[std::make_pair(leftPart,
    firstTerminal)] == p)
358                         {
359 
360                         }
361                         // 冲突，这不是 LL(1) 文法，返回
362                         else
363                         {
364                             std::cout << "Collide: " << leftPart << " " <<
    firstTerminal << std::endl;
365                             return false;
366                         }
367                     }
368                 }
369             }
370         }
371         this -> parseTable = tmpParseTable;
372         return true;
373     };
374 
375     // 计算 First
376     if(!calcFirstTable())
377     {
378         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;
379         return false;
380     }
381     // 计算 Follow
382     if(!calcFollowTable())
383     {
384         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;
385         return false;
386     }
387     // 计算 ParseTable
388     if(!calcParseTable())
389     {
390         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;
391         return false;
392     }
393     return true;
394 }
395
396 // 打印表格
397 void Parser::printInternalTables(std::ostream & out)
398 {
399     out << "This is an LL(1) grammar." << std::endl << std::endl;
400     this -> printFirstTable(out);
401     this -> printFollowTable(out);
402     this -> printParseTable(out);
403 }
404
405 // 语法分析
406 size_t Parser::parse(std::ostream & out)
407 {
408     // 重置
409     lexer.rewind();
410     // 分析栈
```

```
411     std::vector<std::string> parseStack;
412 
413     // 打印分析栈
414     auto printParseStack = [&parseStack, &out]() -> void
415     {
416         out << "Parse stack: ";
417         for(const auto & symbol : parseStack)
418         {
419             out << symbol << " ";
420         }
421         out << std::endl;
422     };
423 
424     // 打印 token
425     auto printToken = [&out](Types::TokenPair token) -> void
426     {
427         out << Shared::typeStrings.at(token.first) << " ";
428         if(token.first >= Types::TokenType::INIT
429             && token.first <= Types::TokenType::ENDOFFILE )
430         {
431             // out;
432         }
433         else if(token.first == Types::TokenType::KEYWORD )
434         {
435             out << "( " << std::any_cast<std::string>(token.second) << " )";
436         }
437         else if(token.first == Types::TokenType::IDENTIFIER )
438         {
439             out << "( " << Shared::idTable.at(std::any_cast<size_t>(token.
    second)) << " )";
440         }
441         else if(token.first >= Types::TokenType::INT_CONST
442             && token.first <= Types::TokenType::STR_LITERAL )
443         {
444             out << "( " << Shared::constTable.at(std::any_cast<size_t>(token.
    second)) << " )";
445         }
```

```
446         else if(token.first >= Types::TokenType::OP_ADD
447             && token.first <= Types::TokenType::OP_SCOPE )
448         {
449             out << "( " << std::any_cast<std::string>(token.second) << " )";
450         }
451         else if(token.first >= Types::TokenType::DELIM_DBQUOTE
452             && token.first <= Types::TokenType::DELIM_QUESTION )
453         {
454             out << "( " << std::any_cast<std::string>(token.second) << " )";
455         }
456     };
457 
458     // 初始化入栈
459     parseStack.push_back(Shared::endOfFileChar);
460     parseStack.push_back(grammar.startSymbol);
461 
462     // 错误
463     size_t errorCount = 0;
464 
465     // 指向第一个词
466     auto token = lexer.getNextToken();
467     while(true)
468     {
469         // 跳过非实义符号
470         if(token.first == Types::TokenType::INIT)
471         {
472             token = lexer.getNextToken();
473             continue;
474         }
475 
476         // 打印栈
477         printParseStack();
478         out << "Now token: ";
479         printToken(token);
480         out << std::endl;
481 
482         // 如果词法错误，处理
```

```
483         if(token.first == Types::TokenType::ERROR)
484         {
485             errorCount++;
486             lexer.errorProcess(std::any_cast<Types::LexerError>(token.second));
487             // 跳过本词
488             token = lexer.getNextToken();
489         }
490         // 栈已经空了
491         else if(parseStack.back() == Shared::endOfFileChar)
492         {
493             // 对上了
494             if(token.first == Types::TokenType::ENDOFFILE)
495             {
496                 out << "Successfully finished." << std::endl;
497                 break;
498             }
499             else
500             {
501                 errorCount++;
502                 this -> errorProcess(Types::ParserError(lexer.getFilePos(), "
    unexpected end of file"));
503                 break;
504             }
505         }
506         // 非终结符，需要根据预测分析表
507         if(grammar.isNonTerminal(parseStack.back()))
508         {
509             std::string tokenTypeStr = Shared::typeStrings.at(token.first);
510             // 替换掉关键词
511             if(tokenTypeStr == "KEYWORD")
512             {
513                 tokenTypeStr = std::any_cast<std::string>(token.second);
514             }
515             auto tableTermIter = parseTable.find(std::make_pair(parseStack.back
    (), tokenTypeStr));
516             // 找到了
517             if(tableTermIter != parseTable.end())
```

```
518             {
519                 // 反向压入产生式
520                 const auto & rightPart = tableTermIter -> second.second;
521                 out << "Use rule: " << parseStack.back() << " -> ";
522                 for(const auto & symbol : rightPart)
523                 {
524                     if(symbol == "")
525                     {
526                         out << "\"\"" << " ";
527                     }
528                     else
529                     {
530                         out << symbol << " ";
531                     }
532                 }
533                 out << std::endl << std::endl;
534                 parseStack.pop_back();
535                 for(auto i = rightPart.rbegin(); i != rightPart.rend(); i++)
536                 {
537                     // 跳过 ""
538                     if(*i != "")
539                     {
540                         parseStack.push_back(*i);
541                     }
542                 }
543             }
544             else
545             {
546                 errorCount++;
547                 // 错误信息
548                 std::string errorMessage = "unexpected token: " + tokenTypeStr;
549                 errorMessage += ", expected: " + parseStack.back();
550                 this -> errorProcess(Types::ParserError(lexer.getFilePos(),
    errorMessage));
551                 break;
552             }
553         }
```

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

```
588         }
589     }
590     return errorCount;
591 }
592
593 // 错误处理
594 void Parser::errorProcess(const Types::ParserError & error)
595 {
596     // 行列
597     size_t row = error.first.first, col = error.first.second - 1;
598
599     std::cout << "\033[1m" << lexer.getSrcName() << ":"
600         << row << ":"
601         << col << ": (Parser) \033[31merror: \033[0m\033[1m"
602         << error.second << "\033[0m" << std::endl;
603
604     std::cout << "    " << lexer.getInBuf() << std::endl;
605     std::cout << "    ";
606     for(size_t i = 1; i < col; i++)
607     {
608         std::cout << (lexer.getInBuf()[i - 1] == '\t' ? '\t' : ' ');
609     }
610     std::cout << "\033[1;2m^\033[0m" << std::endl;
611 }
612
613 // 打印 FIRST 表
614 void Parser::printFirstTable(std::ostream & out)
615 {
616     out << "FIRST Table:" << std::endl;
617     for(const auto & i : firstTable)
618     {
619         out << "FIRST(" << i.first << "): ";
620         for(const auto & j : i.second)
621         {
622             if(j == "")
623             {
624                 out << "\"\"" << " ";
```

```
625             }
626             else
627             {
628                 out << j << " ";
629             }
630         }
631         out << std::endl;
632     }
633     out << std::endl;
634 };
635 
636 // 打印 FOLLOW 表
637 void Parser::printFollowTable(std::ostream & out)
638 {
639     out << "FOLLOW Table:" << std::endl;
640     for(const auto & i : followTable)
641     {
642         out << "FOLLOW(" << i.first << "): ";
643         for(const auto & j : i.second)
644         {
645             out << j << " ";
646         }
647         out << std::endl;
648     }
649     out << std::endl;
650 };
651 
652 // 打印预测分析表
653 void Parser::printParseTable(std::ostream & out)
654 {
655     out << "LL(1) Parse Table:" << std::endl;
656     for(const auto & tableTerm : this -> parseTable)
657     {
658         out << "M[" << tableTerm.first.first << ", " << tableTerm.first.second
    << "]: ";
659         out << tableTerm.second.first << " -> ";
660         for(const auto & rightSymbol : tableTerm.second.second)
```

```
661         {
662             if(rightSymbol == "")
663             {
664                 out << "\"\"" << " ";
665             }
666             else
667             {
668                 out << rightSymbol << " ";
669             }
670         }
671         out << std::endl;
672     }
673     out << std::endl;
674 };
675 
676 #define INDEPENDENT_PARSER
677 #ifdef INDEPENDENT_PARSER
678 int main(int argc, char * argv[])
679 {
680     auto printUsage = []() -> void
681     {
682         std::cout << "Usage:\n  ./parser <filename> [options]" << std::endl;
683         std::cout << "Options:\n  -h, --help\t\t\t Print help." << std::endl;
684         std::cout << "  -g, --grammar\t\t\t Set input grammar file name. (
    Default: Use internal grammar.)" << std::endl;
685         std::cout << "  -o, --output\t\t\t Set output file name. (Default: 'out
    .txt'.)" << std::endl;
686     };
687 
688     Parser parser;
689     // 源文件名, 语法文件名, 输出文件名
690     std::string srcFileName, grammarFileName, outputFileName = "out.txt";
691     // 输出文件流
692     std::fstream outStream;
693 
694     enum class FlagIndex
695     {
```

```
696         SET_GRAMMARFILE,
697         SET_OUTPUTFILE
698     };
699 
700     // 设置相关 Flags
701     std::bitset<2> setFileFlags = 0;
702 
703     if(argc <= 1 || argc % 2 != 0)
704     {
705         std::cout << "\033[1m(Parser)\033[0m \033[1;31merror:\033[0m Wrong
    usage!" << std::endl;
706         printUsage();
707         exit(1);
708     }
709 
710     for(int i = 1; i < argc; i++)
711     {
712         std::string cmd = std::string(argv[i]);
713         if(cmd == "-h" || cmd == "--help")
714         {
715             printUsage();
716             exit(0);
717         }
718         else if(cmd == "-g" || cmd == "--grammar")
719         {
720             setFileFlags.set(size_t(FlagIndex::SET_GRAMMARFILE));
721         }
722         else if(cmd == "-o" || cmd == "--output")
723         {
724             setFileFlags.set(size_t(FlagIndex::SET_OUTPUTFILE));
725         }
726         else
727         {
728             if(i == 1)
729             {
730                 srcFileName = cmd;
731             }
```

```
732             // 设置 token 文件
733             if(setFileFlags.test(size_t(FlagIndex::SET_GRAMMARFILE)))
734             {
735                 grammarFileName = cmd;
736                 setFileFlags.set(size_t(FlagIndex::SET_GRAMMARFILE), false);
737             }
738             // 设置输出文件
739             if(setFileFlags.test(size_t(FlagIndex::SET_OUTPUTFILE)))
740             {
741                 outputFileName = cmd;
742                 setFileFlags.set(size_t(FlagIndex::SET_OUTPUTFILE), false);
743             }
744         }
745     }
746 
747     // 参数不对
748     if(setFileFlags.any())
749     {
750         std::cout << "\033[1m(Parser)\033[0m \033[1;31merror:\033[0m Wrong
    usage!" << std::endl;
751         printUsage();
752         exit(1);
753     }
754     // 打不开文件
755     if(!parser.openFile(srcFileName))
756     {
757         std::cout << "\033[1m(Parser)\033[0m \033[1;31merror:\033[0m\033[1m Can
    't open source file: "
758             << srcFileName << "\033[0m" << std::endl;
759         exit(1);
760     }
761     outStream.open(outputFileName, std::ios::out);
762     if(!outStream.is_open())
763     {
764         std::cout << "\033[1m(Parser)\033[0m \033[1;31merror:\033[0m\033[1m Can
    't open output file: "
765             << outputFileName << "\033[0m" << std::endl;
```

```
766         exit(1);
767     }
768 
769     // 读取语法
770     if(grammarFileName != "")
771     {
772         // 语法检验失败
773         if(!parser.readGrammar(grammarFileName))
774         {
775             std::cout << "\033[1m(Parser Generator)\033[0m \033[1;31merror
    :\033[0m\033[1m Invalid grammar. \033[0m" << std::endl;
776             exit(1);
777         }
778     }
779     // 计算预测分析表
780     size_t errorCount = 0;
781     if(parser.calcLL1ParseTable())
782     {
783         // 打印一下
784         parser.printInternalTables(std::cout);
785         parser.printInternalTables(outStream);
786         // 分析
787         errorCount = parser.parse(std::cout);
788         if(errorCount > 0)
789         {
790             std::cout << errorCount << " error(s) generated." << std::endl;
791             exit(1);
792         }
793         parser.parse(outStream);
794     }
795     else
796     {
797         std::cout << "\033[1m(Parser)\033[0m \033[1;31merror:\033[0m\033[1m
    Invalid grammar. " << std::endl;
798         exit(1);
799     }
800     return 0;
```

```
801 }
802 #endif
```

参考文献

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