

# 编译原理 实验一 词法分析器

中国人民大学 信息学院 崔冠宇 2018202147

## 1 实验内容

设计一个 C++ 语言的词法分析器。它的输入输出要求如下：

1. 输入：C++ 语言源文件 `test.cmm`。
2. 输出：C++ 源文件对应的 `token` 二元组列表文件 `token.out`、变量名表文件 `id.out` 以及常量表文件 `const.out`。

## 2 程序设计原理与方法

如下图所示，词法分析 (lexical analysis) 是编译过程的第一个主要阶段，为后面的语法分析 (syntax analysis) 部分提供了重要的支持。词法分析器 (lexical analyzer, lexer, scanner) 接受输入的源文件，利用正规文法和有限自动机对源文件进行分析，输出对应的符号流。在设计词法分析器之前，我们需要先设计 C++ 语言的词法。

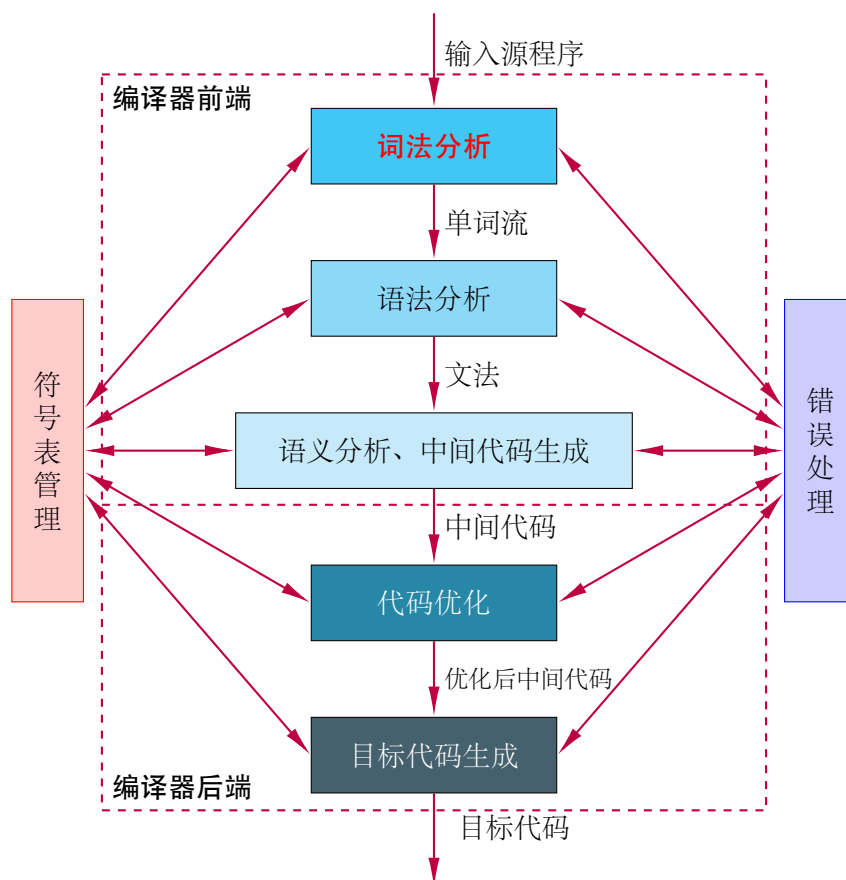


图 1: 编译器结构图

## 2.1 C-- 语言的词法定义

C-- 语言的词法以 C 语言、C++ 语言的词法为蓝本，参考 C 标准文件 [1] 中 5.2.1 节 **Character sets** 以及 6.4 节 **Lexical elements** 以及 C++ 标准文件 [2] 中第 5 章 **Lexical conventions** 修改而成。

### 2.1.1 字符 (character)

字符是构成源文件的基石。C-- 语言的**实义字符**为下列几类之一：

- 大小写拉丁字母 (uppercase & lowercase Latin alphabet, 简称 letter):

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 以及

a b c d e f g h i j k l m n o p q r s t u v w x y z

正规产生式为:  $\text{letter} \rightarrow A|\dots|Z|a|\dots|z$

- 阿拉伯数字 (digit): 0 1 2 3 4 5 6 7 8 9

正规产生式为:  $\text{digit} \rightarrow 0|1|\dots|9$

- 其它可显示 ASCII 字符（主要是运算符和分隔符）：

! " # % & ' ( ) \* + , - . / : ; < = > ? [ \ ] ^ \_ { | } ~

### 2.1.2 单词符号 (token) 及其类型

单词符号 (token) 是编程语言中的最小词法单元。C--语言的**单词符号类别**参考了 C/C++ 语言标准，并且进行了适当的修改与合并。C-- 语言的单词符号类别为下列几类之一：

- 关键字 (keyword), C-- 语言的保留字，为下列各单词之一：

auto bool break case char class const continue default do double else enum false float for  
goto if int long namespace nullptr private protected public return short signed sizeof  
static struct switch this true typedef unsigned using void while

- 标识符 (identifier), 以下划线或大小写拉丁字母开头的下划线、大小写拉丁字母及数字串。正规式产生式如下：

$\text{identifier} \rightarrow (\_|\text{letter})(\_|\text{letter}|\text{digit})^*$

- 常量 (constant), 或称**字面量** (literal), 包含**常数** (numeric-constant, num-const) (为简便起见，只考虑十进制常数)、**字符常量** (character-constant, char-const) 和**字符串字面量** (string-literal, str-literal)。正规式产生式如下：

$\text{constant} \rightarrow \text{num-const}|\text{char-const}|\text{str-literal}$

其中，

- **常数**，是指整数或小数的十进制字面量，且可以具有指数部分。它的正规产生式如下：

$\text{num-const} \rightarrow \text{int-const}|\text{float-const}$

$\text{int-const} \rightarrow \text{integral}(\text{exponential}|\varepsilon)$

$\text{float-const} \rightarrow (\text{integral}).(\text{fractional})(\text{exponential}|\varepsilon)$

$\text{integral} \rightarrow 0|(1|\dots|9)(\text{digit})^*$

$\text{fractional} \rightarrow (\text{digit})^*$

$\text{exponential} \rightarrow (\text{E}|\text{e})(+|-|\varepsilon)(0|(1|\dots|9)(\text{digit})^*)$

- **字符常量**，是指单引号包裹下的单个 ASCII 字符<sup>1</sup>。它的正规产生式如下：

$\text{char-const} \rightarrow '\{\text{ASCII}\}'$

- **字符串字面量**，是指双引号包裹下的任意 ASCII 字符<sup>1</sup> 序列。它的正规产生式如下：

$\text{str-literal} \rightarrow "(\{\text{ASCII}\})^*"$

- **运算符 (operator)**，是 C++ 语言用来运算的符号，包括**简单运算符** (simple-operator, simp-op) 和**复合运算符** (compound-operator, comp-op)。它的正规产生式如下：

$\text{operator} \rightarrow \text{simp-op}|\text{comp-op}$

其中，

- **简单运算符**，是指单字符的运算符，主要有：+ - \* / % > < ! & | ~ ^ . =。它们的正规产生式如下：

$\text{simp-op} \rightarrow \text{add}|\text{sub}|\text{mul}|\text{div}|\text{mod}|\text{gt}|\text{lt}|\text{lnot}|\text{and}|\text{or}|\text{not}|\text{xor}|\text{dot}|\text{asn}$

$\text{add} \rightarrow +$

$\text{sub} \rightarrow -$

$\text{mul} \rightarrow *$

$\text{div} \rightarrow /$

$\text{mod} \rightarrow \%$

$\text{gt} \rightarrow >$

$\text{lt} \rightarrow <$

$\text{lnot} \rightarrow !$

$\text{and} \rightarrow \&$

$\text{or} \rightarrow |$

$\text{not} \rightarrow \sim$

$\text{xor} \rightarrow \wedge$

$\text{dot} \rightarrow .$

$\text{asn} \rightarrow =$

- **复合运算符**，是指多字符的运算符，主要有：-> ++ -- << >> <= >= == != && || += -= \*= /= &= ^= |= <<= >>= ::。它们的正规产生式如下：

$\text{comp-op} \rightarrow \text{arrow}|\text{inc}|\text{dec}|\text{shl}|\text{shr}|\text{le}|\text{ge}|\text{eq}|\text{neq}|\text{land}|\text{lor}|\text{addasn}|\text{subasn}|\text{mulasn}|\text{divasn}|\text{andasn}|\text{xorasn}|\text{orasn}|\text{shlasn}|\text{shrasn}|\text{scope}$

$\text{arrow} \rightarrow ->$

$\text{inc} \rightarrow ++$

$\text{dec} \rightarrow --$

<sup>1</sup>引号和转义字符等特殊情况暂时不考虑。

```
shl → <<
shr → >>
le → <=
ge → >=
eq → ==
neq → !=
land → &&
lor → ||
addasn → +=
subasn → -=
mulasn → *=
divasn → /=
andasn → &=
xorasn → ^=
orasn → |=
shlasn → <<=
shrasn → >>=
scope → ::
```

- 分隔符 (delimiter), 是指 C++ 语言中表示语言结构的一类字符, 主要有: " # ' ( ) , : ; [ ] { } ?。它们的正规产生式如下:

```
delimiter → dbquote|sharp|sgquote|lpar|rpar|comma|colon|semicolon|lsqbracket|rsqbracket|
lcurbrace|rcurbrace|question
dbquote → "
sharp → #
sgquote → '
lpar → (
rpar → )
comma → ,
colon → :
semicolon → ;
lsqbracket → [
rsqbracket → ]
lcurbrace → {
rcurbrace → }
question → ?
```

## 2.2 注释

C++ 语言支持两类注释形式，分别是行内注释和多行注释。

### 2.2.1 行内注释

行内注释的格式为

```
// ... (不可跨行) ... \n
```

词法分析器在分析时，一旦识别到行内注释起始符号 `//`，则**丢弃**注释符号及之后的所有内容，直到遇到**第一个换行符** `\n` 为止。

### 2.2.2 多行注释

多行注释的格式为

```
/* ... (可跨行) ... */
```

词法分析器在分析时，一旦识别到多行注释起始符号 `/*`，则**丢弃**注释符号及之后的所有内容，直到遇到**第一组多行注释结束符** `*/` 为止。

## 3 程序设计流程

### 3.1 单词符号的输出格式

词法分析器的单词符号的输出为二元组 `(type, content)` 流。

为了便于后面的分析，下面规定 `type` 字段的取值范围以及此时 `content` 字段的内容。二元组中 `type` 字段可以为下列各值之一：

- `INIT`，表示初始状态，或没有得到单词符号。此时 `content` 的内容为空；
- `ERROR`，表示错误状态。此时 `content` 内容为错误的位置；
- `KEYWORD`，表示单词符号是一个关键字。此时 `content` 的内容是关键字本身；
- `IDENTIFIER`，表示单词符号是一个标识符。此时 `content` 的内容是标识符表中对应标识符的下标；
- `INT_CONST`，表示单词符号是一个整型常量。此时 `content` 的内容是常量表中对应常量的下标；
- `FLOAT_CONST`，表示单词符号是一个浮点常量。此时 `content` 的内容是常量表中对应常量的下标；

- CHAR\_CONST, 表示单词符号是一个字符常量。此时 content 的内容是常量表中对应常量的下标;
- STR\_LITERAL, 表示单词符号是一个字符串常量。此时 content 的内容是常量表中对应常量的下标;
- OP\_(optype), 运算符单词符号一词一类, 具体比如 OP\_INC 表示 ++。这是所有运算符单词符号的统一形式, 此时 content 的内容是运算符本身;
- DELIM\_(delimtype), 分隔符单词符号一词一类, 具体比如 DEILM\_LPAR 表示 (。这是所有分隔符单词符号的统一形式, 此时 content 的内容是分隔符本身。

## 3.2 词法分析算法设计

---

### Algorithm 1 词法分析核心算法 getNextToken()

---

```

// 初始化
type ← INIT, token ← "", content ← 空
// 跳过空白符等, 保证当前字符为实义字符
while 缓冲区 inBuf 为空, 或字符指针 ch 到达 inBuf 结尾, 且源文件没有结束 do
    读入一行源文件
end while
while *ch 为空白符 do
    推进至下一个字符
end while
if *ch 为 / 且 *(ch + 1) 也为 / then
    // 跳过单行注释
    ch 移到缓冲区末尾
else if *ch 为 / 且 *(ch + 1) 为 * then
    // 跳过多行注释
    移动 ch 以及读取文件, 直到找到注释结束符号或文件结尾
end if

if *ch 为下划线或字母 then
    // 处理标识符和关键字
    return processIDKWD()
else if *ch 为数字 then
    // 处理各类常数
    return processNUMCONST()
else if *ch 为运算符字符之一 then
    // 处理各类运算符
    return processOP()
else if *ch 为分隔符字符之一 then
    // 处理各类分隔符
    return processDELIM()
else
    return (ERROR, 错误信息)
end if

```

---

下面给出各子函数的实现 (其中 processNUMCONST 用状态转换图表示, processOP 穷举过程太长, 由于篇幅原因省略):

**Algorithm 2** 处理标识符和关键字 processIDKWD

---

```

type ← IDENTIFIER
while *ch 为下划线或字母或数字 do
    token ← *ch
    ch 向前移动
end while
if token 是关键字 then
    type ← KEYWORD
    content ← token 在关键字表中的下标
    return (type, content)
end if
if token 不在标识符表中 then
    填写标识符表
end if
content ← token 在符号表中的下标
return (type, content)

```

---

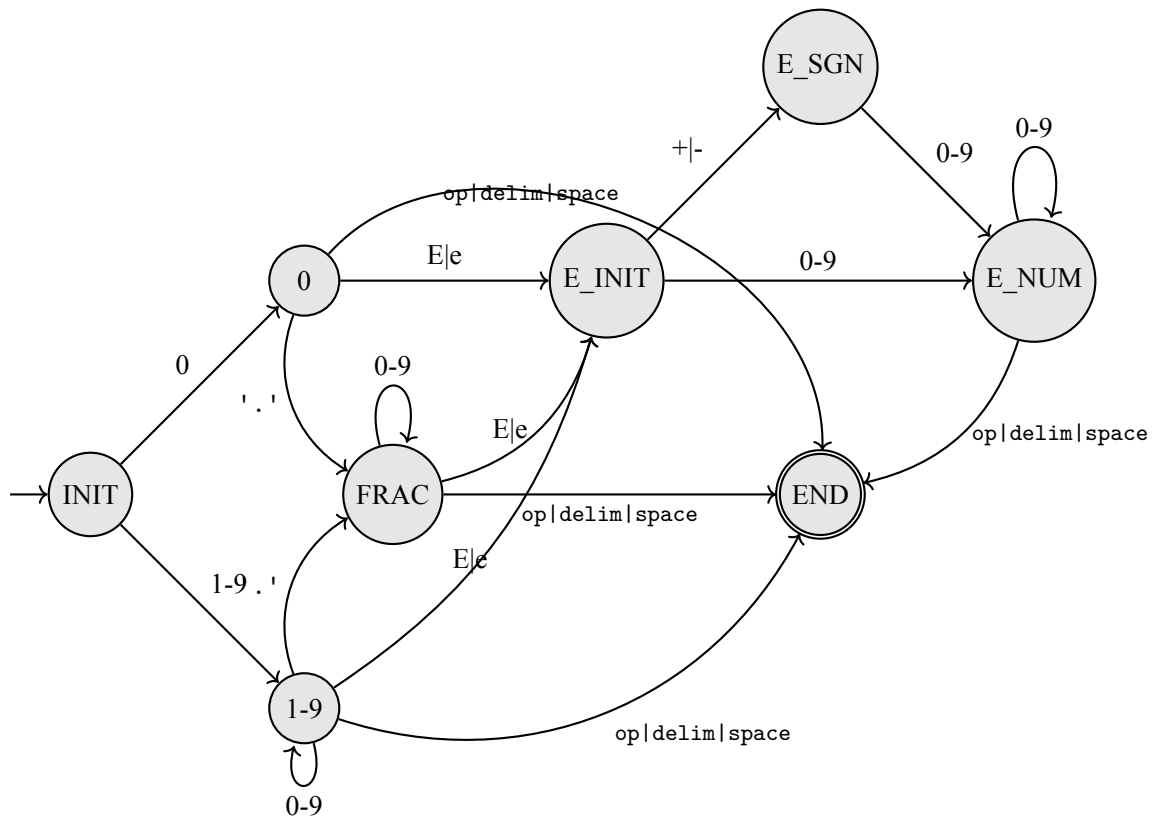


图 2: 处理各类常数 processNUMCONST 的状态图

## 4 程序清单

为了保持思路的连续性，程序清单挪至文末附录处。

**Algorithm 3** 处理各类分隔符 processDELIM

---

```

type ← INIT, token ← ""
if *ch == ':' then
    token ← *ch
    if *(ch + 1) == '.' then
        type ← OP_SCOPE, 将两个字符加入 token
        return (type, token)
    end if
else if *ch 为单引号 then
    return processCHARCONST()
else if *ch == '"' then
    return processSTRLITERAL()
end if
c = *ch, token ← c, ch 前进
return (type(c), token)

```

---

**Algorithm 4** 处理字符常量 processCHARCONST

---

```

token ← '\\', 字符指针前进
if ch 已经到缓冲区末尾或 *ch 为引号 then
    return (ERROR, 错误信息)
end if
token ← *ch, 字符指针前进
if ch 已经到缓冲区末尾或 *ch 不为引号 then
    return (ERROR, 错误信息)
end if
return (CHAR_CONST, token)

```

---

**Algorithm 5** 处理字符串字面量 processSTRLITERAL

---

```

token ← '', 字符指针前进
while ch 没有到缓冲区末尾且没有遇到引号 do
    token ← *ch
end while
if ch 已经到缓冲区末尾 then
    return (ERROR, 错误信息)
end if
token ← '', 字符指针前进
return (STR_LITERAL, token)

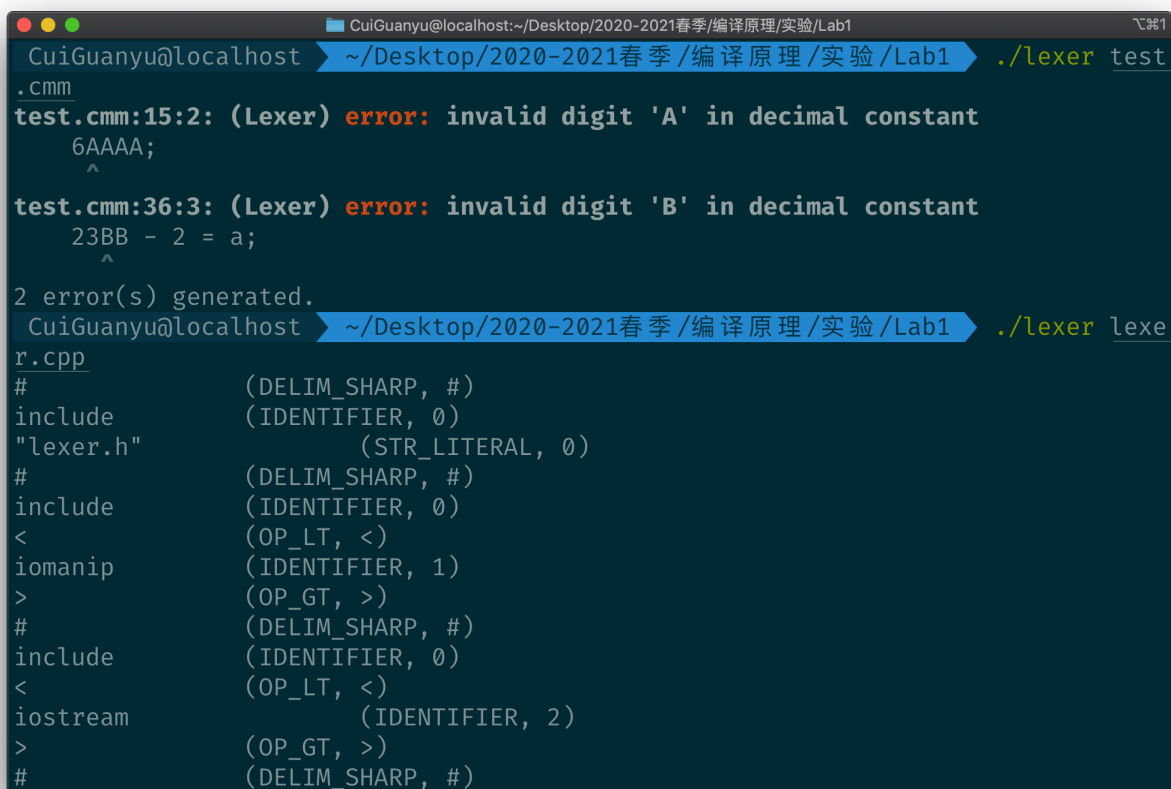
```

---

## 5 运行结果

运行词法分析器程序 `lexer`, 程序以 `test.cmm` 源文件（故意设置错误）以及 `lexer.cpp` 源文件（分析自身）做测试, 在终端的输出结果如下图所示:

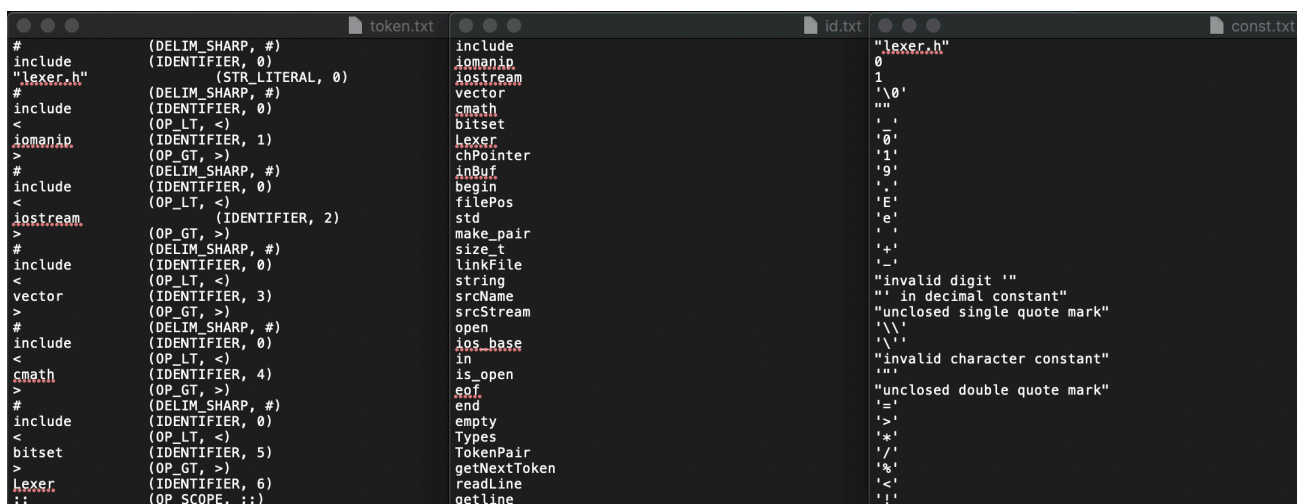




```
CuiGuanyu@localhost: ~/Desktop/2020-2021春季/编译原理/实验/Lab1
CuiGuanyu@localhost ~ ./lexer test
.cmm
test.cmm:15:2: (Lexer) error: invalid digit 'A' in decimal constant
6AAAA;
^
test.cmm:36:3: (Lexer) error: invalid digit 'B' in decimal constant
23BB - 2 = a;
^
2 error(s) generated.
CuiGuanyu@localhost ~ ./lexer lex
r.cpp
# (DELIM_SHARP, #)
include (IDENTIFIER, 0)
"lexer.h" (STR_LITERAL, 0)
# (DELIM_SHARP, #)
include (IDENTIFIER, 0)
< (OP_LT, <)
iomanip (IDENTIFIER, 1)
> (OP_GT, >)
# (DELIM_SHARP, #)
include (IDENTIFIER, 0)
< (OP_LT, <)
iostream (IDENTIFIER, 2)
> (OP_GT, >)
# (DELIM_SHARP, #)
```

图 3: lexer 测试

其中，后者的输出文件 `token.txt`、`id.txt` 以及 `const.txt` 如下图所示：



token.txt	id.txt	const.txt
# (DELIM_SHARP, #) include (IDENTIFIER, 0) "lexer.h" (STR_LITERAL, 0) # (DELIM_SHARP, #) include (IDENTIFIER, 0) < (OP_LT, <) iomanip (IDENTIFIER, 1) > (OP_GT, >) # (DELIM_SHARP, #) include (IDENTIFIER, 0) < (OP_LT, <) iostream (IDENTIFIER, 2) > (OP_GT, >) # (DELIM_SHARP, #) include (IDENTIFIER, 0) < (OP_LT, <) vector (IDENTIFIER, 3) > (OP_GT, >) # (DELIM_SHARP, #) include (IDENTIFIER, 0) < (OP_LT, <) cmath (IDENTIFIER, 4) > (OP_GT, >) # (DELIM_SHARP, #) include (IDENTIFIER, 0) < (OP_LT, <) bitset (IDENTIFIER, 5) > (OP_GT, >) lexer (IDENTIFIER, 6) :: (OP_SCOPE, ::)	include iomanip iostream vector cmath bitset lexer chPointer inBuf begin filePos std make_pair size_t linkFile string srcName srcStream open ios_base in is_open eof end empty Types TokenPair getNextToken readLine getline	"lexer.h" 0 1 '\0' "" ' ' '0' '1' '9' '.' 'E' 'e' ' ' '+' '-' "invalid digit " " ' in decimal constant" "unclosed single quote mark" '\'' "invalid character constant" "" "unclosed double quote mark" '=' '>' '*' '/' '%' '<' '!'

图 4: 输出文件

可见程序运行正确。

## 6 程序使用说明

词法分析器的使用命令为

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

其中 `filename` 是文件名, `options` 是附加命令, 主要有以下几种:

- `-h`, `--help` 打印帮助信息;
- `-t`, `--token <filename>` 设置输出的 `token` 文件名;
- `-i`, `--identifier <filename>` 设置输出的标识符文件名;
- `-c`, `--const <filename>` 设置输出的常量文件名。

## 7 总结与完善

### 7.1 亮点

本程序主要有以下亮点:

1. 支持的关键字、运算符和分隔符等较为齐全, 能够对编译器源文件本身进行词法分析;
2. 含有功能较全的错误提示, 提示内容包括错误所在的行列位置, 方便改正。

### 7.2 不足与可能的改进方法

本程序的不足之处主要是单词类型多, 导致代码过于冗长, 不仅编写时有大量重复劳动, 而且不方便调试, 后期可以使用词法分析器生成器自动生成代码来替代。

## A 程序设计清单

1. 工具文件: `util.h`:

```
1 #ifndef UTIL_H
2 #define UTIL_H
3
4 #include <utility>
```

```
5 #include <any>
6 #include <string>
7 #include <vector>
8 #include <unordered_set>
9 #include <unordered_map>
10
11 #define INDEPENDENT_LEXER
12
13 namespace Types
14 {
15     // Token 类型: 枚举类
16     enum class TokenType
17     {
18         // 初始化状态
19         INIT,
20         // 错误状态
21         ERROR,
22         // ----- 关键字 -----
23         KEYWORD,
24         // ----- 标识符 -----
25         IDENTIFIER,
26         // ----- 常量 -----
27         // 整型常量
28         INT_CONST,
29         // 浮点常量
30         FLOAT_CONST,
31         // 字符常量
32         CHAR_CONST,
33         // 字符串字面量
34         STR_LITERAL,
35         // ----- 运算符 -----
36         // 简单运算符
37         // +
38         OP_ADD,
39         // -
40         OP_SUB,
41         // *
```

```
42      OP_MUL ,
43      // /
44      OP_DIV ,
45      // %
46      OP_MOD ,
47      // >
48      OP_GT ,
49      // <
50      OP_LT ,
51      // !
52      OP_LNOT ,
53      // &
54      OP_AND ,
55      // |
56      OP_OR ,
57      // ~
58      OP_NOT ,
59      // ^
60      OP_XOR ,
61      // .
62      OP_DOT ,
63      // =
64      OP_ASN ,
65      // 复合运算符
66      // ->
67      OP_ARROW ,
68      // ++
69      OP_INC ,
70      // --
71      OP_DEC ,
72      // <<
73      OP_SHL ,
74      // >>
75      OP_SHR ,
76      // <=
77      OP_LE ,
78      // >=
```

```
79      OP_GE ,
80      // ==
81      OP_EQ ,
82      // !=
83      OP_NEQ ,
84      // &&
85      OP_LAND ,
86      // ||
87      OP_LOR ,
88      // +=
89      OP_ADDASN ,
90      // -=
91      OP_SUBASN ,
92      // *=
93      OP_MULASN ,
94      // /=
95      OP_DIVASN ,
96      // &=
97      OP_ANDASN ,
98      // ^=
99      OP_XORASN ,
100     // |=
101     OP_ORASN ,
102     // <<=
103     OP_SHLASN ,
104     // >>=
105     OP_SHRASN ,
106     // ::
107     OP_SCOPE ,
108     // ----- 分隔符 -----
109     // "
110     DELIM_DBQUOTE ,
111     // #
112     DELIM_SHARP ,
113     // '
114     DELIM_SGQUOTE ,
115     // (
```

```
116     DELIM_LPAR ,
117     // )
118     DELIM_RPAR ,
119     // ,
120     DELIM_COMMA ,
121     // :
122     DELIM_COLON ,
123     // ;
124     DELIM_SEMICOLON ,
125     // [
126     DELIM_LSQBACKET ,
127     // ]
128     DELIM_RSQBACKET ,
129     // {
130     DELIM_LCURBRACE ,
131     // }
132     DELIM_RCURBRACE ,
133     // ?
134     DELIM_QUESTION
135 };
136
137 // 文件中位置: <所在行, 所在列> 对
138 using FilePos = std::pair<size_t, size_t>;
139 // 分词结果: <类型-内容> 对
140 using TokenPair = std::pair<Types::TokenType, std::any>;
141 // Lexer 错误: <文件位置, 错误信息> 对
142 using LexerError = std::pair<FilePos, std::string>;
143
144 // 类型枚举-类型字符串
145 using TypeStringTable = std::unordered_map<Types::TokenType, std::string>;
146 // 关键字表
147 using KeywordsTable = std::vector<std::string>;
148 // 运算符字符表
149 using OperatorCharTable = std::unordered_set<char>;
150 // 分隔符字符-分隔符类型名
151 using DelimCharTable = std::unordered_map<char, Types::TokenType>;
152 // 标识符名称表
```

```
153     using IdentifierTable = std::vector<std::string>;
154     // 常数表
155     using ConstTable = std::vector<std::string>;
156     // 符号表
157     using SymbolTable = std::vector<std::string>;
158 }
159
160 namespace Shared
161 {
162     Types::IdentifierTable idTable;
163     Types::ConstTable constTable;
164     Types::SymbolTable symbolTable;
165
166     // 语言的关键字
167     const Types::KeywordsTable keywords = \
168         Types::KeywordsTable({
169             "auto", "bool", "break",
170             "case", "char", "class",
171             "const", "continue", "default",
172             "do", "double", "else",
173             "enum", "false", "float",
174             "for", "goto", "if",
175             "int", "long", "namespace",
176             "nullptr", "private", "protected",
177             "public", "return", "short",
178             "signed", "sizeof", "static",
179             "struct", "switch", "this",
180             "true", "typedef", "unsigned",
181             "using", "void", "while"
182         });
183
184     // 运算符
185     const Types::OperatorCharTable opChars = \
186         Types::OperatorCharTable({
187             '!', '%', '&', '*', '+',
188             '-', '.', '/', '<', '=',
189             '>', '^', '|', '~'
```

```
190     });
191
192     // 分隔符
193     const Types::DelimCharTable delimChars = \
194         Types::DelimCharTable({
195             {'"', Types::TokenType::DELIM_DBQUOTE},
196             {'#', Types::TokenType::DELIM_SHARP},
197             {'\'', Types::TokenType::DELIM_SGQUOTE},
198             {'(', Types::TokenType::DELIM_LPAR},
199             {')', Types::TokenType::DELIM_RPAR},
200             {',', Types::TokenType::DELIM_COMMA},
201             {':', Types::TokenType::DELIM_COLON},
202             {';', Types::TokenType::DELIM_SEMICOLON},
203             {'[', Types::TokenType::DELIM_LSQBRACKET},
204             {']', Types::TokenType::DELIM_RSQBRACKET},
205             {'{', Types::TokenType::DELIM_LCURBRACE},
206             {'}', Types::TokenType::DELIM_RCURBRACE},
207             {'?', Types::TokenType::DELIM_QUESTION} });
208
209     // 枚举类型字符串表
210     const Types::TypeStringTable typeStrings = \
211         Types::TypeStringTable({
212             // 初始化状态
213             {Types::TokenType::INIT, "INIT"},
214             // 错误状态
215             {Types::TokenType::ERROR, "ERROR"},
216             // ----- 关键字 -----
217             {Types::TokenType::KEYWORD, "KEYWORD"},
218             // ----- 标识符 -----
219             {Types::TokenType::IDENTIFIER, "IDENTIFIER"},
220             // ----- 常量 -----
221             // 整型常量
222             {Types::TokenType::INT_CONST, "INT_CONST"},
223             // 浮点常量
224             {Types::TokenType::FLOAT_CONST, "FLOAT_CONST"},
225             // 字符常量
226             {Types::TokenType::CHAR_CONST, "CHAR_CONST"},
```



```
227 // 字符串字面量
228 {Types::TokenType::STR_LITERAL, "STR_LITERAL"},
229 // ----- 运算符 -----
230 // 简单运算符
231 // +
232 {Types::TokenType::OP_ADD, "OP_ADD"},
233 // -
234 {Types::TokenType::OP_SUB, "OP_SUB"},
235 // *
236 {Types::TokenType::OP_MUL, "OP_MUL"},
237 // /
238 {Types::TokenType::OP_DIV, "OP_DIV"},
239 // %
240 {Types::TokenType::OP_MOD, "OP_MOD"},
241 // >
242 {Types::TokenType::OP_GT, "OP_GT"},
243 // <
244 {Types::TokenType::OP_LT, "OP_LT"},
245 // !
246 {Types::TokenType::OP_LNOT, "OP_LNOT"},
247 // &
248 {Types::TokenType::OP_AND, "OP_AND"},
249 // |
250 {Types::TokenType::OP_OR, "OP_OR"},
251 // ~
252 {Types::TokenType::OP_NOT, "OP_NOT"},
253 // ^
254 {Types::TokenType::OP_XOR, "OP_XOR"},
255 // .
256 {Types::TokenType::OP_DOT, "OP_DOT"},
257 // =
258 {Types::TokenType::OP_ASN, "OP_ASN"},
259 // 复合运算符
260 // ->
261 {Types::TokenType::OP_ARROW, "OP_ARROW"},
262 // ++
263 {Types::TokenType::OP_INC, "OP_INC"},
```

```
264      // --
265      {Types::TokenType::OP_DEC, "OP_DEC"},
266      // <<
267      {Types::TokenType::OP_SHL, "OP_SHL"},
268      // >>
269      {Types::TokenType::OP_SHR, "OP_SHR"},
270      // <=
271      {Types::TokenType::OP_LE, "OP_LE"},
272      // >=
273      {Types::TokenType::OP_GE, "OP_GE"},
274      // ==
275      {Types::TokenType::OP_EQ, "OP_EQ"},
276      // !=
277      {Types::TokenType::OP_NEQ, "OP_NEQ"},
278      // &&
279      {Types::TokenType::OP_LAND, "OP_LAND"},
280      // ||
281      {Types::TokenType::OP_LOR, "OP_LOR"},
282      // +=
283      {Types::TokenType::OP_ADDASN, "OP_ADDASN"},
284      // -=
285      {Types::TokenType::OP_SUBASN, "OP_SUBASN"},
286      // *=
287      {Types::TokenType::OP_MULASN, "OP_MULASN"},
288      // /=
289      {Types::TokenType::OP_DIVASN, "OP_DIVASN"},
290      // &=
291      {Types::TokenType::OP_ANDASN, "OP_ANDASN"},
292      // ^=
293      {Types::TokenType::OP_XORASN, "OP_XORASN"},
294      // |=
295      {Types::TokenType::OP_ORASN, "OP_ORASN"},
296      // <<=
297      {Types::TokenType::OP_SHLASN, "OP_SHLASN"},
298      // >>=
299      {Types::TokenType::OP_SHRASN, "OP_SHRASN"},
300      // ::
```

```
301         {Types::TokenType::OP_SCOPE, "OP_SCOPE"},
302         // ----- 分隔符 -----
303         // "
304         {Types::TokenType::DELIM_DBQUOTE, "DELIM_DBQUOTE"},
305         // #
306         {Types::TokenType::DELIM_SHARP, "DELIM_SHARP"},
307         // '
308         {Types::TokenType::DELIM_SGQUOTE, "DELIM_SGQUOTE"},
309         // (
310         {Types::TokenType::DELIM_LPAR, "DELIM_LPAR"},
311         // )
312         {Types::TokenType::DELIM_RPAR, "DELIM_RPAR"},
313         // ,
314         {Types::TokenType::DELIM_COMMA, "DELIM_COMMA"},
315         // :
316         {Types::TokenType::DELIM_COLON, "DELIM_COLON"},
317         // ;
318         {Types::TokenType::DELIM_SEMICOLON, "DELIM_SEMICOLON"},
319         // [
320         {Types::TokenType::DELIM_LSQBACKET, "DELIM_LSQBACKET"},
321         // ]
322         {Types::TokenType::DELIM_RSQBACKET, "DELIM_RSQBACKET"},
323         // {
324         {Types::TokenType::DELIM_LCURBRACE, "DELIM_LCURBRACE"},
325         // }
326         {Types::TokenType::DELIM_RCURBRACE, "DELIM_RCURBRACE"},
327         // ?
328         {Types::TokenType::DELIM_QUESTION, "DELIM_QUESTION"} });
329
330 // 判断是否是关键字
331 std::pair<bool, size_t> isKeyword(const std::string & token)
332 {
333     for(size_t i = 0; i < keywords.size(); i++)
334     {
335         if(keywords[i] == token)
336         {
337             return std::pair<bool, size_t>(true, i);
```

```
338         }
339     }
340     return std::pair<bool, size_t>(false, keywords.size());
341 }
342 // 判断是否是运算符字符
343 bool isOpChar(const char & c)
344 {
345     return opChars.find(c) != opChars.end();
346 }
347 // 判断是否是分隔符字符
348 bool isDelimChar(const char & c)
349 {
350     return delimChars.find(c) != delimChars.end();
351 }
352
353 // 判断符号是否在标识符表出现过
354 std::pair<bool, size_t> inIDTable(const std::string & identifier)
355 {
356     for(size_t i = 0; i < idTable.size(); i++)
357     {
358         if(idTable[i] == identifier)
359         {
360             return std::pair<bool, size_t>(true, i);
361         }
362     }
363     return std::pair<bool, size_t>(false, idTable.size());
364 }
365
366 // 判断符号是否在符号表出现过
367 std::pair<bool, size_t> inSymbolTable(const std::string & symbol)
368 {
369     for(size_t i = 0; i < symbolTable.size(); i++)
370     {
371         if(symbolTable[i] == symbol)
372         {
373             return std::pair<bool, size_t>(true, i);
374         }
375     }
376 }
```

```
375     }
376     return std::pair<bool, size_t>(false, symbolTable.size());
377 }
378 // 判断符号是否在常数表出现过
379 std::pair<bool, size_t> inConstTable(const std::string & constant)
380 {
381     for(size_t i = 0; i < constTable.size(); i++)
382     {
383         if(constTable[i] == constant)
384         {
385             return std::pair<bool, size_t>(true, i);
386         }
387     }
388     return std::pair<bool, size_t>(false, constTable.size());
389 }
390 }
391
392 #endif
```

## 2. 词法分析器类的定义: **lexer.h**:

```
1 #ifndef LEXER_H
2 #define LEXER_H
3
4 #include <fstream>
5 #include "util.h"
6
7 class Lexer
8 {
9     // ----- 公有成员 -----
10 public:
11     // 扫描字符指针类型
12     using ScanPointer = std::string::iterator;
13
14     // ----- 构造函数 -----
15     // 默认构造函数
16     Lexer();
17     // 复制构造(标记删除)
```

```
18     Lexer(const Lexer & other) = delete;
19
20     // ----- 析构函数 -----
21     ~Lexer();
22
23     // ----- 成员函数 -----
24     // 打开源文件
25     bool linkFile(const std::string & srcName);
26     // 是否已经扫描到结尾
27     bool eof();
28     // 解析返回下一个 Token
29     Types::TokenPair getNextToken();
30     // 错误处理
31     void errorProcess(const Types::LexerError & error);
32
33     // ----- 私有成员 -----
34     private:
35         // 源文件名
36         std::string srcName;
37         // 源文件流
38         std::fstream srcStream;
39         // 输入缓冲区
40         std::string inBuf;
41         // 缓冲区字符指针
42         ScanPointer chPointer;
43         // 当前字符在文件中的位置，用于错误处理
44         Types::FilePos filePos;
45 };
46
47 #endif
```

### 3. 词法分析器类的实现: **lexer.cpp**:

```
1 #include "lexer.h"
2
3 #include <iomanip>
4 #include <iostream>
5 #include <vector>
```

```
6 #include <cmath>
7 #include <bitset>
8
9 // 构造函数
10 Lexer::Lexer()
11 : chPointer(inBuf.begin()), filePos(std::make_pair<size_t, size_t>(0, 1)) {}
12
13 // 析构函数
14 Lexer::~Lexer() {}
15
16 // 关联文件
17 bool Lexer::linkFile(const std::string & srcName)
18 {
19     this -> srcName = srcName;
20     // 打开源文件
21     srcStream.open(srcName, std::ios_base::in);
22     return srcStream.is_open();
23 }
24
25 // 判断词法分析是否结束
26 bool Lexer::eof()
27 {
28     // 文件流中没有行没有读取，且输入缓冲区为空或已经分析到结尾
29     return srcStream.eof() && (chPointer == inBuf.end() || inBuf.empty());
30 }
31
32 Types::TokenPair Lexer::getNextToken()
33 {
34     // ----- 工具函数 -----
35     // 从文件流中读一行到缓冲区
36     // 读取成功则返回 true,
37     // 若已经到达文件流结尾，则返回 false
38     auto readLine = [this]() -> bool
39     {
40         if(srcStream.eof())
41         {
42             return false;
```

```
43     }
44     std::getline(srcStream, inBuf);
45     // 增加行数
46     filePos.first++;
47     // 回到行首
48     chPointer = inBuf.begin();
49     filePos.second = 1;
50     return true;
51 };
52
53 // 扫描指针向前推进一个字符
54 // 前进成功则返回 true,
55 // 若已经到达文件流尾部, 则返回 false
56 auto goForward = [this, readLine]() -> bool
57 {
58     // 仍然是保证缓冲区不为空
59     while(inBuf.empty() || chPointer == inBuf.end())
60     {
61         if(!readLine())
62         {
63             return false;
64         }
65     }
66     chPointer++;
67     filePos.second++;
68     return true;
69 };
70
71 // 向前看 k 个字符, 但指针不往前推进
72 auto peekForward = [this](size_t k = 1) -> char
73 {
74     if(chPointer + k > inBuf.end())
75     {
76         return '\\0';
77     }
78     return *(chPointer + k);
79 };
```



```
80
81 // ----- 分析各个类型的函数 -----
82 // 处理标识符和关键字
83 auto processIDKWD = [this, goForward]() -> Types::TokenPair
84 {
85     // 先设置成标识符类型
86     Types::TokenType tokenType = Types::TokenType::IDENTIFIER;
87     std::string token = "";
88     // 如果仍然是下划线、字母或数字
89     while(chPointer != inBuf.end()
90           && (std::isalnum(*chPointer) || *chPointer == '_'))
91     {
92         token.push_back(*chPointer);
93         goForward();
94     }
95     // 标识符还需判定是否为关键字
96     auto keywordResult = Shared::isKeyword(token);
97     // 是关键字，则把关键字的序号作为二元组的内容
98     if(keywordResult.first)
99     {
100         tokenType = Types::TokenType::KEYWORD;
101         return std::make_pair(tokenType, std::any(token));
102     }
103     // 否则就是普通的标识符，需要填写标识符表
104     // 判定是否出现过同名标识符
105     auto idResult = Shared::inIDTable(token);
106     // 没出现过，则插入表中
107     if(!idResult.first)
108     {
109         Shared::idTable.push_back(token);
110         return std::make_pair(tokenType, std::any(Shared::idTable.size() -
1111         1));
112     }
113     return std::make_pair(tokenType, std::any(idResult.second));
114 };
115 // 处理常数
```

```
116     auto processNUMCONST = [this, goForward]() -> Types::TokenPair
117     {
118         // 初始化状态
119         Types::TokenType tokenType = Types::TokenType::INT_CONST;
120         std::string token = "";
121
122         enum class NUM_STATE
123         {
124             INT_INIT, INT_GOT_0, INT_GOT_1TO9,
125             FRAC, EXP_INIT, EXP_GOT_SGN
126         };
127
128         NUM_STATE state = NUM_STATE::INT_INIT;
129         while(chPointer != inBuf.end() && tokenType != Types::TokenType::ERROR)
130         {
131             // 读取第一个字符
132             if(state == NUM_STATE::INT_INIT)
133             {
134                 // 第一个字符为 0 -> 类型为整型, 进入读到零的分析部分
135                 if(*chPointer == '0')
136                 {
137                     token.push_back(*chPointer);
138                     state = NUM_STATE::INT_GOT_0;
139                     goForward();
140                 }
141                 // 第一个字符为 1-9 -> 类型为整型, 进入读到 1-9 的分析部分
142                 else if(*chPointer >= '1' && *chPointer <= '9')
143                 {
144                     token.push_back(*chPointer);
145                     state = NUM_STATE::INT_GOT_1TO9;
146                     goForward();
147                 }
148                 // 应该不会发生
149                 else
150                 {
151                     tokenType = Types::TokenType::ERROR;
152                     break;
```

```
153         }
154     }
155     // 数字部分第一个字符为 0 的情况
156     else if(state == NUM_STATE::INT_GOT_0)
157     {
158         // 接受 . -> 类型为浮点型, 进入小数部分分析
159         if(*chPointer == '.')
160         {
161             token.push_back(*chPointer);
162             tokenType = Types::TokenType::FLOAT_CONST;
163             state = NUM_STATE::FRAC;
164             goForward();
165         }
166         // 运算符或分隔符 -> 类型为整型, 分析结束
167         else if(Shared::isOpChar(*chPointer)
168             || Shared::isDelimChar(*chPointer)
169             || std::isspace(*chPointer))
170         {
171             tokenType = Types::TokenType::INT_CONST;
172             break;
173         }
174         // 读到 E|e -> 类型为整型, 进入指数部分分析
175         else if(*chPointer == 'E' || *chPointer == 'e')
176         {
177             token.push_back(*chPointer);
178             state = NUM_STATE::EXP_INIT;
179             goForward();
180         }
181         // 其它字母数字不接受
182         else
183         {
184             tokenType = Types::TokenType::ERROR;
185             break;
186         }
187     }
188     // 得到的第一个字符为 1-9
189     else if(state == NUM_STATE::INT_GOT_1TO9)
```

```
190     {
191         // 读到 0-9 -> 类型为整型, 保持
192         if(std::isdigit(*chPointer))
193         {
194             //state = NUM_STATE::INT_GOT_1TO9;
195             token.push_back(*chPointer);
196             goForward();
197         }
198         // 接受 . -> 类型为浮点型, 进入小数部分分析
199         else if(*chPointer == '.')
200         {
201             token.push_back(*chPointer);
202             tokenType = Types::TokenType::FLOAT_CONST;
203             state = NUM_STATE::FRAC;
204             goForward();
205         }
206         // 运算符或分隔符 -> 类型为整型, 分析结束
207         else if(Shared::isOpChar(*chPointer)
208             || Shared::isDelimChar(*chPointer)
209             || *chPointer == ' ')
210         {
211             tokenType = Types::TokenType::INT_CONST;
212             break;
213         }
214         // 读到 E|e -> 类型为整型, 进入指数部分分析
215         else if(*chPointer == 'E' || *chPointer == 'e')
216         {
217             token.push_back(*chPointer);
218             state = NUM_STATE::EXP_INIT;
219             goForward();
220         }
221         // 其它字母数字不接受
222         else
223         {
224             tokenType = Types::TokenType::ERROR;
225             break;
226         }
227     }
```

```
227     }
228     // 分析小数部分
229     else if(state == NUM_STATE::FRAC)
230     {
231         // 读取 0-9 -> 类型为浮点数, 继续分析
232         if(std::isdigit(*chPointer))
233         {
234             token.push_back(*chPointer);
235             goForward();
236         }
237         // 运算符或分隔符 -> 类型为浮点型, 分析结束
238         else if(Shared::isOpChar(*chPointer)
239             || Shared::isDelimChar(*chPointer)
240             || *chPointer == ' ')
241         {
242             tokenType = Types::TokenType::FLOAT_CONST;
243             break;
244         }
245         // 读到 E|e -> 类型为浮点型, 进入指数部分分析
246         else if(*chPointer == 'E' || *chPointer == 'e')
247         {
248             token.push_back(*chPointer);
249             state = NUM_STATE::EXP_INIT;
250             goForward();
251         }
252         // 其它字母数字不接受
253         else
254         {
255             tokenType = Types::TokenType::ERROR;
256             break;
257         }
258     }
259     // 分析指数部分
260     else if(state == NUM_STATE::EXP_INIT)
261     {
262         // 如果读到正负号, 加入, 转到分析指数值部分
263         if(*chPointer == '+' || *chPointer == '-')
```

```
264         {
265             token.push_back(*chPointer);
266             state = NUM_STATE::EXP_GOT_SGN;
267             goForward();
268         }
269         // 如果是数字，直接进入分析指数数值部分
270         else if(*chPointer >= '1' && *chPointer <= '9')
271         {
272             state = NUM_STATE::EXP_GOT_SGN;
273         }
274         else
275         {
276             tokenType = Types::TokenType::ERROR;
277             break;
278         }
279     }
280     // 分析指数数值部分
281     else if(state == NUM_STATE::EXP_GOT_SGN)
282     {
283         if(std::isdigit(*chPointer))
284         {
285             token.push_back(*chPointer);
286             state = NUM_STATE::EXP_GOT_SGN;
287             goForward();
288         }
289         // 运算符或分隔符 -> 分析结束
290         else if(Shared::isOpChar(*chPointer)
291             || Shared::isDelimChar(*chPointer)
292             || std::isspace(*chPointer))
293         {
294             break;
295         }
296         // 其它字母数字不接受
297         else
298         {
299             tokenType = Types::TokenType::ERROR;
300             break;
```

```
301         }
302     }
303     else
304     {
305         tokenType = Types::TokenType::ERROR;
306         break;
307     }
308 }
309
310 if(tokenType == Types::TokenType::ERROR)
311 {
312     std::string message = "invalid digit ";
313     message.push_back(*chPointer);
314     message += std::string("' in decimal constant");
315     return std::make_pair(
316         tokenType,
317         std::any( Types::LexerError(filePos, message) )
318     );
319 }
320 // 填常数表
321 auto constResult = Shared::inConstTable(token);
322 // 如果没出现过, 则插入
323 if(!constResult.first)
324 {
325     Shared::constTable.push_back(token);
326     return std::make_pair(tokenType,
327         std::any(Shared::constTable.size() - 1)
328     );
329 }
330 return std::make_pair(tokenType,
331     std::any(constResult.second)
332 );
333 };
334
335 // 处理字符常量
336 auto processCHARCONST = [this, goForward]() -> Types::TokenPair
337 {
```

```
338     Types::TokenType tokenType = Types::TokenType::CHAR_CONST;
339     std::string token = "";
340     // 将单引号送入
341     token.push_back(*chPointer);
342     // 下一个字符
343     goForward();
344     if(chPointer == inBuf.end())
345     {
346         tokenType = Types::TokenType::ERROR;
347         std::string message = "unclosed single quote mark";
348         return std::make_pair(tokenType,
349             std::any(Types::LexerError(filePos, message))
350         );
351     }
352     // 遇到反斜线
353     else if(*chPointer == '\\')
354     {
355         // 送入反斜线
356         token.push_back(*chPointer);
357         goForward();
358
359         if(chPointer == inBuf.end())
360         {
361             tokenType = Types::TokenType::ERROR;
362             std::string message = "unclosed single quote mark";
363             return std::make_pair(tokenType,
364                 std::any(Types::LexerError(filePos, message))
365             );
366         }
367     }
368     token.push_back(*chPointer);
369     goForward();
370     if(chPointer == inBuf.end())
371     {
372         tokenType = Types::TokenType::ERROR;
373         std::string message = "unclosed single quote mark";
374         return std::make_pair(tokenType,
```



```
375         std::any(Types::LexerError(filePos, message))
376     );
377 }
378 // 不是单引号
379 else if(*chPointer != '\\')
380 {
381     tokenType = Types::TokenType::ERROR;
382     std::string message = "invalid character constant";
383     return std::make_pair(tokenType,
384         std::any(Types::LexerError(filePos, message))
385     );
386 }
387 else
388 {
389     token.push_back(*chPointer);
390     goForward();
391 }
392 // 填常数表
393 auto constResult = Shared::inConstTable(token);
394 // 如果没出现过, 则插入
395 if(!constResult.first)
396 {
397     Shared::constTable.push_back(token);
398     return std::make_pair(tokenType,
399         std::any(Shared::constTable.size() - 1)
400     );
401 }
402 return std::make_pair(tokenType,
403     std::any(constResult.second)
404 );
405 };
406
407 // 处理字符串字面量
408 auto processSTRLITERAL = [this, goForward]() -> Types::TokenPair
409 {
410     Types::TokenType tokenType = Types::TokenType::STR_LITERAL;
411     std::string token = "";
```

```
412         // 双引号
413         token.push_back(*chPointer);
414         goForward();
415         while(chPointer != inBuf.end() && !(*chPointer == '"' && *(chPointer -
1) != '\\'))
416         {
417             token.push_back(*chPointer);
418             goForward();
419         }
420         if(chPointer == inBuf.end())
421         {
422             tokenType = Types::TokenType::ERROR;
423             std::string message = "unclosed double quote mark";
424             return std::make_pair(tokenType,
425                 std::any(Types::LexerError(filePos, message))
426             );
427         }
428         // 双引号进入
429         else
430         {
431             token.push_back(*chPointer);
432             goForward();
433         }
434         // 填常数表
435         auto constResult = Shared::inConstTable(token);
436         // 如果没出现过, 则插入
437         if(!constResult.first)
438         {
439             Shared::constTable.push_back(token);
440             return std::make_pair(tokenType,
441                 std::any(Shared::constTable.size() - 1)
442             );
443         }
444         return std::make_pair(tokenType,
445             std::any(constResult.second)
446         );
447     };
```

```
448
449 // 处理各类运算符
450 auto processOP = [this, goForward]() -> Types::TokenPair
451 {
452     // 初始化状态
453     Types::TokenType tokenType = Types::TokenType::INIT;
454     std::string token = "";
455     // ----- 运算符 -----
456     // 简单运算符
457     // + - * / % > < ! & | ~ ^ . =
458     // 复合运算符
459     // -> ++ -- << >> <= >= == != && || += -= *= /= &= ^= |= <<= >>= ::
460
461     // + ++ +=
462     if(*chPointer == '+')
463     {
464         tokenType = Types::TokenType::OP_ADD;
465         token.push_back(*chPointer);
466         goForward();
467         if(*chPointer == '+')
468         {
469             tokenType = Types::TokenType::OP_INC;
470             token.push_back(*chPointer);
471             goForward();
472         }
473         else if(*chPointer == '=')
474         {
475             tokenType = Types::TokenType::OP_ADDASN;
476             token.push_back(*chPointer);
477             goForward();
478         }
479     }
480     // - -> -- -=
481     else if(*chPointer == '-')
482     {
483         tokenType = Types::TokenType::OP_SUB;
484         token.push_back(*chPointer);
```

```
485         goForward();
486         if(*chPointer == '>')
487         {
488             tokenType = Types::TokenType::OP_ARROW;
489             token.push_back(*chPointer);
490             goForward();
491         }
492         else if(*chPointer == '-')
493         {
494             tokenType = Types::TokenType::OP_DEC;
495             token.push_back(*chPointer);
496             goForward();
497         }
498         else if(*chPointer == '=')
499         {
500             tokenType = Types::TokenType::OP_SUBASN;
501             token.push_back(*chPointer);
502             goForward();
503         }
504     }
505     // * *=
506     else if(*chPointer == '*')
507     {
508         tokenType = Types::TokenType::OP_MUL;
509         token.push_back(*chPointer);
510         goForward();
511         if(*chPointer == '=')
512         {
513             tokenType = Types::TokenType::OP_MULASN;
514             token.push_back(*chPointer);
515             goForward();
516         }
517     }
518     // / /=
519     else if(*chPointer == '/')
520     {
521         tokenType = Types::TokenType::OP_DIV;
```

```
522         token.push_back(*chPointer);
523         goForward();
524         if(*chPointer == '=')
525         {
526             tokenType = Types::TokenType::OP_DIVASN;
527             token.push_back(*chPointer);
528             goForward();
529         }
530     }
531     // %
532     else if(*chPointer == '%')
533     {
534         tokenType = Types::TokenType::OP_MOD;
535         token.push_back(*chPointer);
536         goForward();
537     }
538     // > >> >= >>=
539     else if(*chPointer == '>')
540     {
541         tokenType = Types::TokenType::OP_GT;
542         token.push_back(*chPointer);
543         goForward();
544         if(*chPointer == '>')
545         {
546             tokenType = Types::TokenType::OP_SHR;
547             token.push_back(*chPointer);
548             goForward();
549             if(*chPointer == '=')
550             {
551                 tokenType = Types::TokenType::OP_SHRASN;
552                 token.push_back(*chPointer);
553                 goForward();
554             }
555         }
556         else if(*chPointer == '=')
557         {
558             tokenType = Types::TokenType::OP_GE;
```

```
559         token.push_back(*chPointer);
560         goForward();
561     }
562 }
563 // < << <= <<=
564 else if(*chPointer == '<')
565 {
566     tokenType = Types::TokenType::OP_LT;
567     token.push_back(*chPointer);
568     goForward();
569     if(*chPointer == '<')
570     {
571         tokenType = Types::TokenType::OP_SHL;
572         token.push_back(*chPointer);
573         goForward();
574         if(*chPointer == '=')
575         {
576             tokenType = Types::TokenType::OP_SHLASN;
577             token.push_back(*chPointer);
578             goForward();
579         }
580     }
581     else if(*chPointer == '=')
582     {
583         tokenType = Types::TokenType::OP_LE;
584         token.push_back(*chPointer);
585         goForward();
586     }
587 }
588 // ! !=
589 else if(*chPointer == '!=')
590 {
591     tokenType = Types::TokenType::OP_LNOT;
592     token.push_back(*chPointer);
593     goForward();
594     if(*chPointer == '=')
595     {
```

```
596         tokenType = Types::TokenType::OP_NEQ;
597         token.push_back(*chPointer);
598         goForward();
599     }
600 }
601 // & && &=
602 else if(*chPointer == '&')
603 {
604     tokenType = Types::TokenType::OP_AND;
605     token.push_back(*chPointer);
606     goForward();
607     if(*chPointer == '&')
608     {
609         tokenType = Types::TokenType::OP_LAND;
610         token.push_back(*chPointer);
611         goForward();
612     }
613     else if(*chPointer == '=')
614     {
615         tokenType = Types::TokenType::OP_ANDASN;
616         token.push_back(*chPointer);
617         goForward();
618     }
619 }
620 // | || |=
621 else if(*chPointer == '|')
622 {
623     tokenType = Types::TokenType::OP_OR;
624     token.push_back(*chPointer);
625     goForward();
626     if(*chPointer == '|')
627     {
628         tokenType = Types::TokenType::OP_LOR;
629         token.push_back(*chPointer);
630         goForward();
631     }
632     else if(*chPointer == '=')
```

```
633         {
634             tokenType = Types::TokenType::OP_ORASN;
635             token.push_back(*chPointer);
636             goForward();
637         }
638     }
639     // ~
640     else if(*chPointer == '~')
641     {
642         tokenType = Types::TokenType::OP_NOT;
643         token.push_back(*chPointer);
644         goForward();
645     }
646     // ^ ^=
647     else if(*chPointer == '^')
648     {
649         tokenType = Types::TokenType::OP_XOR;
650         token.push_back(*chPointer);
651         goForward();
652         if(*chPointer == '=')
653         {
654             tokenType = Types::TokenType::OP_XORASN;
655             token.push_back(*chPointer);
656             goForward();
657         }
658     }
659     // .
660     else if(*chPointer == '.')
661     {
662         tokenType = Types::TokenType::OP_DOT;
663         token.push_back(*chPointer);
664         goForward();
665     }
666     // = ==
667     else if(*chPointer == '=')
668     {
669         tokenType = Types::TokenType::OP_ASN;
```



```
670         token.push_back(*chPointer);
671         goForward();
672         if(*chPointer == '=')
673         {
674             tokenType = Types::TokenType::OP_EQ;
675             token.push_back(*chPointer);
676             goForward();
677         }
678     }
679     // :: 在分隔符处处理
680     return std::make_pair(tokenType, std::any(token));
681 };
682
683 // 处理分隔符
684 auto processDELIM = [this, goForward, peekForward,
685     processCHARCONST, processSTRLITERAL]() -> Types::TokenPair
686 {
687     // 初始化状态
688     Types::TokenType tokenType = Types::TokenType::INIT;
689     std::string token = "";
690
691     if(*chPointer == ':')
692     {
693         token.push_back(*chPointer);
694         if(peekForward() == ':')
695         {
696             tokenType = Types::TokenType::OP_SCOPE;
697             goForward();
698             token.push_back(*chPointer);
699             goForward();
700             return std::make_pair(tokenType, std::any(token));
701         }
702     }
703     else if(*chPointer == '\\')
704     {
705         return processCHARCONST();
706     }
```

```
707         else if(*chPointer == '"')
708         {
709             return processSTRLITERAL();
710         }
711         char c = *chPointer;
712         token.push_back(c);
713         goForward();
714         return std::make_pair(Shared::delimChars.at(c), std::any(token));
715     };
716
717     // ----- 词法分析开始 -----
718     // ----- 跳过非实义字符 -----
719     // 如果缓冲区为空或前向指针到达结尾
720     while(inBuf.empty() || chPointer == inBuf.end())
721     {
722         // 尝试读取一行文件
723         // 如果最后一行已经读进来了，直接返回
724         if(!readLine())
725         {
726             return std::make_pair(Types::TokenType::INIT, std::any(nullptr));
727         }
728     }
729     // 跳过所有空格，回车等空白符
730     while(std::isspace(*chPointer) || *chPointer == '\\')
731     {
732         if(!goForward())
733         {
734             return std::make_pair(Types::TokenType::INIT, std::any(nullptr));
735         }
736     }
737     // 处理注释等
738     // 单行注释，直接移到行尾
739     if(*chPointer == '/' && peekForward() == '/')
740     {
741         chPointer = inBuf.end();
742         filePos.second = inBuf.size() + 1;
743     }
```

```
744 // 多行注释，需要找到匹配的结束符
745 else if(*chPointer == '/' && peekForward() == '*')
746 {
747     while(!(*chPointer == '*' && peekForward() == '/'))
748     {
749         if(!goForward())
750         {
751             return std::make_pair(Types::TokenType::INIT, std::any(nullptr)
);
752         }
753     }
754     // 跳过结束符
755     if(!(goForward() && goForward()))
756     {
757         return std::make_pair(
758             Types::TokenType::ERROR,
759             std::any(Types::LexerError(filePos, "..."))
760         );
761     }
762 }
763
764 // ----- 处理各类实义字符 -----
765 // 遇到下划线或字母——处理标识符或关键字
766 if(std::isalpha(*chPointer) || *chPointer == '_')
767 {
768     return processIDKWD();
769 }
770 // 遇到数字——处理常数
771 else if(std::isdigit(*chPointer))
772 {
773     return processNUMCONST();
774 }
775 // 是运算符
776 else if(Shared::isOpChar(*chPointer))
777 {
778     return processOP();
779 }
```

```
780 // 是分隔符
781 else if(Shared::isDelimChar(*chPointer))
782 {
783     return processDELIM();
784 }
785 // 是空白符
786 return std::make_pair(Types::TokenType::INIT, std::any(nullptr));
787 }
788
789 void Lexer::errorProcess(const Types::LexerError & error)
790 {
791     std::cout << "\033[1m" << srcName << ":"
792         << error.first.first << ":"
793         << error.first.second << ": (Lexer) \033[31merror: \033[0m\033[1m"
794         << error.second << "\033[0m" << std::endl;
795
796     std::cout << "    " << inBuf << std::endl;
797     std::cout << "    ";
798     for(size_t i = 1; i < filePos.second; i++)
799     {
800         std::cout << (inBuf[i - 1] == '\t' ? '\t' : ' ');
801     }
802     std::cout << "\033[1;2m^\033[0m" << std::endl;
803 }
804
805 #ifdef INDEPENDENT_LEXER
806 int main(int argc, char * argv[])
807 {
808     auto printUsage = []() -> void
809     {
810         std::cout << "Usage:\n  Lexer <filename> [options]" << std::endl;
811         std::cout << "Options:\n  -h, --help\t\t\t Print help." << std::endl;
812         std::cout << "  -t, --token\t\t\t Set output token file name." << std::
813         endl;
814         std::cout << "  -i, --identifier\t\t Set output identifier file name."
815         << std::endl;
816         std::cout << "  -c, --const\t\t\t Set output const file name." << std::
```

```
endl;
815     };
816
817     auto outputToken = [](std::ostream & out, Types::TokenPair & token) -> void
818     {
819         if(token.first == Types::TokenType::KEYWORD)
820         {
821             // 输出 token
822             out << std::any_cast<std::string>(token.second) << "\t\t("
823             // 输出 类型
824                 << Shared::typeStrings.at(token.first) << ", "
825             // 输出 值
826                 << std::any_cast<std::string>(token.second) << ")" << std::endl
827         };
828         // 标识符、常量都是返回的表内下标
829         else if(token.first == Types::TokenType::IDENTIFIER)
830         {
831             out << Shared::idTable.at(std::any_cast<size_t>(token.second)) << "
832             \t\t("
833                 << Shared::typeStrings.at(token.first) << ", "
834                 << std::any_cast<size_t>(token.second) << ")" << std::endl;
835         }
836         else if(token.first >= Types::TokenType::INT_CONST
837             && token.first <= Types::TokenType::STR_LITERAL)
838         {
839             out << Shared::constTable.at(std::any_cast<size_t>(token.second))
840             << "\t\t("
841                 << Shared::typeStrings.at(token.first) << ", "
842                 << std::any_cast<size_t>(token.second) << ")" << std::endl;
843         }
844         // 运算符和分隔符都是返回的内容
845         else if(token.first >= Types::TokenType::OP_ADD
846             && token.first <= Types::TokenType::DELIM_QUESTION)
847         {
848             out << std::any_cast<std::string>(token.second) << "\t\t("
849                 << Shared::typeStrings.at(token.first) << ", "
```

```
848         << std::any_cast<std::string>(token.second) << ")" << std::endl
      ;
849     }
850     else
851     {
852         out << "DEFAULT." << std::endl;
853     }
854 };
855
856 Lexer lexer;
857 std::string srcFileName,
858     tokenFileName = "token.txt",
859     idFileName = "id.txt",
860     constFileName = "const.txt";
861 // 输出文件流
862 std::fstream tokenStream, idStream, constStream;
863
864 enum class FlagIndex
865 {
866     SET_TOKENFILE,
867     SET_IDFILE,
868     SET_CONSTFILE
869 };
870
871 // 设置相关 Flags
872 std::bitset<3> setFileFlags = 0;
873
874 if(argc <= 1 || argc % 2 != 0)
875 {
876     std::cout << "\033[1m(Lexer)\033[0m \033[1;31merror:\033[0m Wrong usage
! " << std::endl;
877     printUsage();
878     exit(1);
879 }
880
881 for(int i = 1; i < argc; i++)
882 {
```

```
883     std::string cmd = std::string(argv[i]);
884     if(cmd == "-h" || cmd == "--help")
885     {
886         printUsage();
887         exit(0);
888     }
889     else if(cmd == "-t" || cmd == "--token")
890     {
891         setFileFlags.set(size_t(FlagIndex::SET_TOKENFILE));
892     }
893     else if(cmd == "-i" || cmd == "--identifier")
894     {
895         setFileFlags.set(size_t(FlagIndex::SET_IDFILE));
896     }
897     else if(cmd == "-c" || cmd == "--const")
898     {
899         setFileFlags.set(size_t(FlagIndex::SET_CONSTFILE));
900     }
901     else
902     {
903         if(i == 1)
904         {
905             srcFileName = cmd;
906         }
907         // 设置 token 文件
908         if(setFileFlags.test(size_t(FlagIndex::SET_TOKENFILE)))
909         {
910             tokenFileName = cmd;
911             setFileFlags.set(size_t(FlagIndex::SET_TOKENFILE), false);
912         }
913         else if(setFileFlags.test(size_t(FlagIndex::SET_IDFILE)))
914         {
915             idFileName = cmd;
916             setFileFlags.set(size_t(FlagIndex::SET_IDFILE), false);
917         }
918         else if(setFileFlags.test(size_t(FlagIndex::SET_CONSTFILE)))
919         {
```

```
920         constFileName = cmd;
921         setFileFlags.set(size_t(FlagIndex::SET_CONSTFILE), false);
922     }
923 }
924 }
925
926 // 参数不对
927 if(setFileFlags.any())
928 {
929     std::cout << "\033[1m(Lexer)\033[0m \033[1;31merror:\033[0m Wrong usage
! " << std::endl;
930     printUsage();
931     exit(1);
932 }
933 // 打不开文件
934 if(!lexer.linkFile(srcFileName))
935 {
936     std::cout << "\033[1m(Lexer)\033[0m \033[1;31merror:\033[0m Can't open
file: " << srcFileName << std::endl;
937     exit(1);
938 }
939
940 std::vector<Types::TokenPair> tokens;
941 size_t errorCount = 0;
942 while(!lexer.eof())
943 {
944     auto result = lexer.getNextToken();
945     if(result.first == Types::TokenType::ERROR)
946     {
947         lexer.errorProcess(std::any_cast<Types::LexerError>(result.second))
;
948         errorCount++;
949     }
950     else if(result.first != Types::TokenType::INIT)
951     {
952         tokens.emplace_back(result);
953     }
```



```
954     }
955     // 错误统计
956     if(errorCount > 0)
957     {
958         std::cout << errorCount << " error(s) generated." << std::endl;
959     }
960
961     // 开始输出文件
962     tokenStream.open(tokenFileName, std::ios_base::out);
963     idStream.open(idFileName, std::ios_base::out);
964     constStream.open(constFileName, std::ios_base::out);
965
966     if(!tokenStream.is_open()
967         || !idStream.is_open()
968         || !constStream.is_open())
969     {
970         std::cout << "\033[1m(Lexer)\033[0m \033[1;31merror:\033[0m Can't open
file: " << argv[1] << std::endl;
971         exit(1);
972     }
973
974     // 输出 token 表
975     if(errorCount == 0)
976     {
977         for(size_t i = 0; i < tokens.size(); i++)
978         {
979             outputToken(std::cout, tokens[i]);
980         }
981     }
982     for(size_t i = 0; i < tokens.size(); i++)
983     {
984         outputToken(tokenStream, tokens[i]);
985     }
986     // 输出标识符表
987     for(auto & i : Shared::idTable)
988     {
989         idStream << i << std::endl;
```

```
990     }
991     // 输出常量表
992     for(auto & i : Shared::constTable)
993     {
994         constStream << i << std::endl;
995     }
996
997     tokenStream.close();
998     idStream.close();
999     constStream.close();
1000
1001     return 0;
1002 }
1003 #endif
```

## 参考文献

- [1] Programming languages —C (N1570, Committee Draft). <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf>.
- [2] Working Draft, Standard for Programming Language C++ (N4885, Committee Draft). <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/n4885.pdf>.