

## 实验二 编制简单词法分析程序

### 1. 介绍

#### 1.1 实验的内容与要求

**实验内容：**通过了解词法分析程序的功能，设计词法分析程序，通过逐个字符的扫描和分解，能够识别出一个一个单词以及单词的分类。

**实验要求：**掌握词法分析程序的功能和构造方法，编制实验报告。

#### 1.2 主体介绍

在一个成熟的编译器中，根据编译原理的体系框架，前半段需要有词法分析，语法分析，语义分析，中间代码生成等等，除此之外，还有一些辅助类型的工作，比如各类非运行时的语法错误报错处理，就分布于编译的各个阶段。其中词法分析是对于一段高级语言程序的第一步处理，在这个阶段，我们需要完成以下任务：

- 能够有效将各类词素进行识别，并且完成细致的分类工作，词素主要分为**保留字**，**标识符**，**常量**，**运算符**，以及**界符**（除去单独的空格，制表符，回车符以外）。
- 能够彻底将两类**注释消除**，以 C/C++ 为例，分为单行注释以及多行注释。
- 词法分析同时完成一部分**语法错误**的报错，比如注释未结尾，非法字符，双引号不匹配，非法运算符，非法常量（主要是数字常量与字符常量）。
- C/C++ 语言中，要将**宏分离**处理，并进行编译前的预处理，比如将类名以及该类的抽象形式（比如是否为泛型）保存在一个数据结构中，再加载一些命名空间的名字（namespace）。
- 对于标识符使用**符号表**来进行管理，并且保留标识符的作用域。

在上一次实验中，我们已经做过简单的词法分析，并且严格根据 DFA 的数学模型来设计了**字符流驱动**的词法分析器。该模型的思路主要是在词法分析器中定义一系列状态，没读入一个字符，就对于词法分析器进行一次状态转换，同时执行函数将分出来的词素存入词法分析的结果中。这样做存在非常巨大的缺点，主要是词法分析无法主动控制字符的流动，无法根据当前字符的前一个字符或者后一个字符联合做出分析决策，严格的数学模型同时也过于复杂了。

在本次词法分析中，我们对于模型进行了较大的改进，仍然是基于字符流，但是分析机制完全改变为**词素首元素驱动型**的 DFA。主要从几个角度，首先是缓冲区（由单个缓冲区改为主从缓冲区配合缓冲指针），其次是文件指针私有化，然后 DFA 周期有所改变（从原来的单个字符改编为一个 tokenAnalysis 周期加上一个 shiftToken 周期），识别机制也进行了细化（处理逻辑分块），最后是增加了报错机制（由于是实验不是项目，仅仅识别一些常用的）。

本文在第二部分进行了词法分析器设计思路的详细阐述，内容包括以下：[1]对比字符流驱动型词法分析器，以及词素首元素驱动型词法分析器的优劣势，以及**改进点**；[2]**整体介绍**词素首元素驱动型词法分析器的处理逻辑，与设计思路；[3]**注释**处理逻辑；[4]**运算符**类词素（一元，二元，三元运算符）处理逻辑；[5]**数字常量**类词素处理逻辑（整数，浮点数，科学计数，二进制，八进制，十六进制等等）；[6]**标识符**类词素处理逻辑；[7]**字符常量**及**字符串常量**类词素处理逻辑；[8]语法**报错**机制；[9]作用域控制的改进方案，增加 tabSpaceRecorder 数据结果，在作用域控制的逻辑考虑**代码缩进**。

在第三部分中，本文对于具体实验的运行结果进行简单的分析，并介绍一些难点突破和实验细节。在第四部分中，本文进行了一个总结，包括本次实验距离实际的 gcc, lcc 等成熟编译器在词法分析部分的差距。在附录中，本文给出了词法分析器的具体实现，并对于函数和分块逻辑有一定的注释与讲解。

### 2. 程序思想与设计

## 2.1 字符流驱动型设计模式以及词素首元素驱动型设计模式的对比

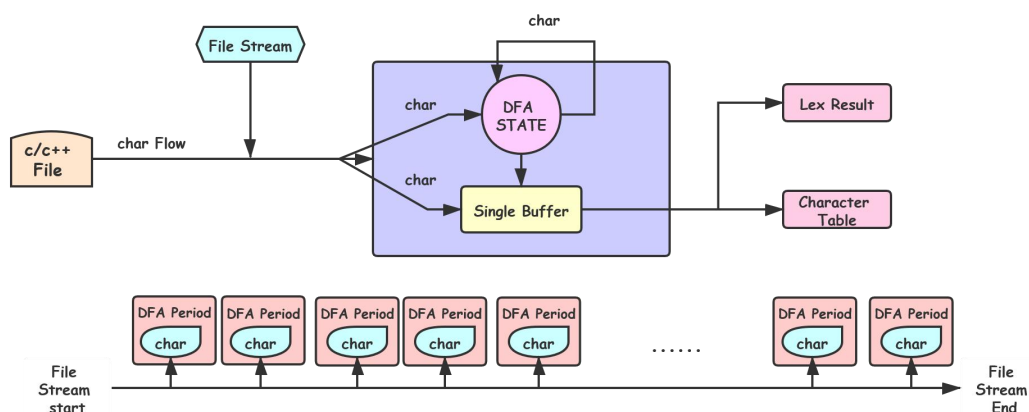


Fig. 1 字符流驱动型词法分析器架构与分析周期

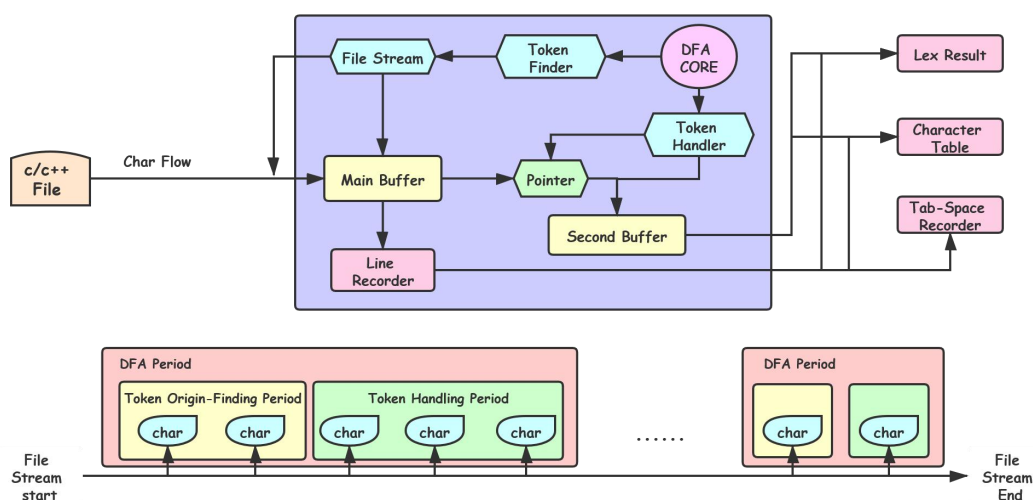


Fig. 2 词素首元素驱动型词法分析器架构与分析周期

上图展示了实验一中的使用的简易字符流驱动型词法分析器的架构模型，以及本次实验所设计的词素首元素驱动型词法分析器的架构模型。

- 从 DFA 周期上看：

- 字符流驱动型每次读入一个字符，都进行一次 DFA 周期，分析器根据当前 DFA 的状态以及字符的类型进行综合决策，采取一定的词法分析策略，策略可能是按兵不动，可能是将字符存入缓冲区，可能是将缓冲区中的字符串存入 LexResult，并清空缓冲区，这样符合数学思维，但是没有考虑到决策是需要参考前后字符关联进行的，这是这个模型最大的败笔；
- 但本模型则从寻找目标词素的首元素开始一个 DFA 周期，周期中首先用 Token Finder 寻找目标词素的首元素，然后根绝首元素的类型启动分块逻辑，分块逻辑中的每一块都有改变缓冲区 Pointer，以及查看 buffer 任意元素的权力，也可以调用 TokenFinder 更新 Main Buffer, 提供了灵活性，更符合程序设计思维。

- 从缓冲区上看：

- 字符流驱动型只有一个缓冲区，由字符流读入，直接输出到 LexResult 中，没有层次的概念，并且 fstream 并非词法分析器所私有，这样做，无法对于字符流的下一个元素进行前瞻，操作不够灵活。
- 本实验中的模型建立了两个缓冲区 Main Buffer 以及 Second Buffer，Main Buffer 负责读取字符流中的一行数据进行缓存，两个 Buffer 之间采用一个 Pointer 来进行关联，Pointer

能够根据分块逻辑，将 Main Buffer 中的内容输入到 Second Buffer 中，Second Buffer 则可以将词素存入到 Lex Result 中。

● 从识别机制来看：

- 字符流驱动型由于其重视数学意义，所以用一个 switch 循环以及一个 for 循环来进行处理，相对比较繁琐。
- 本实验的模型只是根据词素的首元素来进行 switch 分块，其具体实施方案在下面进行介绍，本处不会详细概述。

实际上，举个例子来说，‘/’ 字符可以作为除法的一元运算符，两边可以是标识符或者常量，在词素之间可能还有若干空格，制表符，也可以作为‘/=’的多元运算符，也可以是注释的开始，比如说‘//’以及‘/\*’，处理逻辑与前面完全不同，如果用前一种设计模式来编写，即使是设计之初就考虑到了（耦合性差），DFA 的状态转换的复杂度会增加很多，甚至还需要在底层区控制 fin 的前进和后退。在本实验设计模式中，只需要将‘/’作为词素首元素单独提取，增加私有成员方法即可。

## 2.2 Token-Origin 驱动型 DFA 词法分析器整体处理逻辑

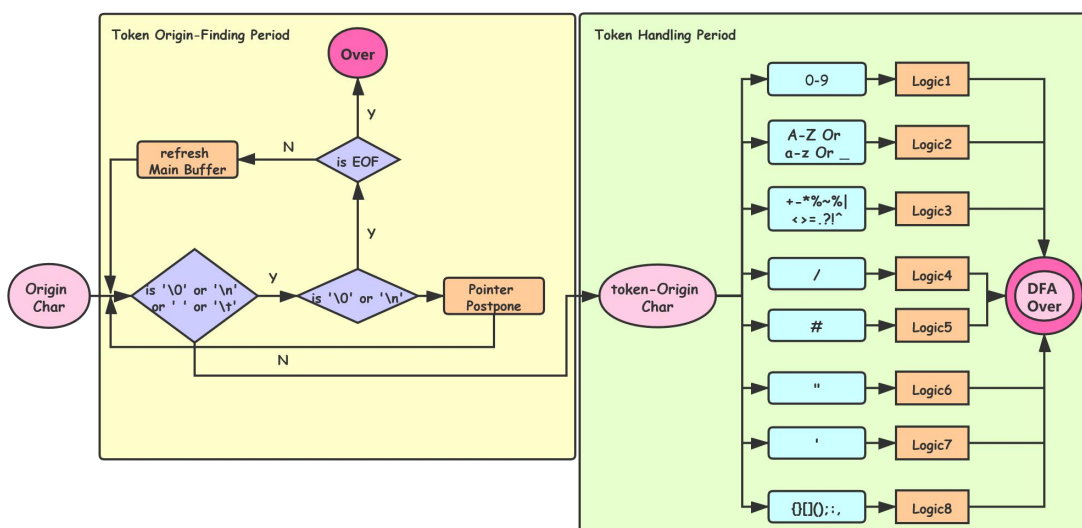


Fig.3 一次 DFA 周期对于一个 token 的处理逻辑

如上图所示，从一个 Origin Char 开始，其可以是任意的字符，进入 token Origin 寻址阶段，寻找下一个目标词素的第一个字符，其大致思想是排除‘\n’、‘\0’、‘\t’、‘\r’以后，每遇到一个‘\n’或者‘\0’（其实扫描到‘\0’可能是 Buffer 初始为空，或者 buffer 被 pointer 扫描到结尾），就更新一次 Main Buffer，将文件中的一行数据更新到 Main Buffer 中。

对于一个 token origin，每一类的处理逻辑是不同的，‘0-9’主要是提取该 token 的数字常量；‘A-Za-z’或者是下划线，是提取该标识符或者保留字；‘+-\*%~%|<>=,?!^’主要是提取运算符，其实<>也有可能是泛型的界符，本实验采取一种简化的思路，将含有泛型(template)的类名存储下来，存在一个数据结构中，实际上这是预编译的过程；‘/’单独拿出来主要是判断是否为注释，若不是，则按照运算符处理。‘#’则主要是提取 C/C++ 中的宏，这也是词法分析非常重要的部分，宏实际情况需要被预编译和预处理；‘以及’分别是字符常量和字符串常量，其逻辑和报错机制也不一样，内部需要考虑转义字符；除去四类空符，其余的‘{}[]() ; ,’是由实际作用的界符，也需要提取出来。

每处理完成一个 token，就是一个 DFA 周期，文件指针以及 buffer 的运用全部是由词法分析器完成的，这二者的耦合减小了程序的复杂程度。

### 2.3 注释处理逻辑

注释在高级语言中不尽相同，在 `python` 中采用 `#` 以及 `'''` 来进行注释，`Matlab` 中使用 `%` 来进行注释，`JavaScript` 中使用 `<!-- -->` 来进行注释，在 `C,C++,Java` 中则使用两种 `/* */` 以及 `//` 来进行多行以及单行的注释。在注释内部，任何字符不会起到作用，直至注释结束；单行注释的边界时 `'\n'` 也就是 `buffer` 内部，而多行注释的边界只有 `*/`，而且这里要进行一类语法错误的报错处理，就是多行注释无结束符匹配。

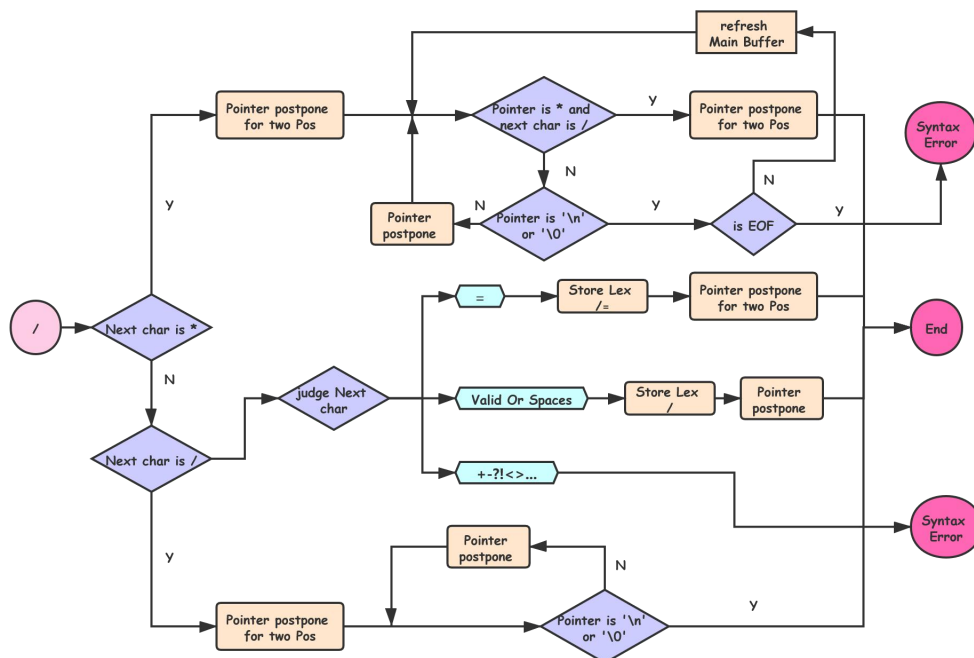


Fig.4 ‘/’处理逻辑 logic4

上图展示了处理逻辑，主要判定其是否为 `/*` 或 `//`，如果是，则全部省略到；若不是注释，就判定运算符是否为 `/=` 或者 `/`，此处还需要判定运算符的合法性，采取一种报错处理。

### 2.4 其他运算符类词素的处理逻辑

运算符在汇编语言中对应不同于 `ALU` 相关的操作，在 `RISC` 指令系统中有着丰富的含义，因此，词法分析需要对其进行单独归类。除了已经处理过的 `/` 以外，存在许多的一元，二元（例如 `==, +=`），以及合法的三元运算符（`>>=`），在词法分析中，我们无需考虑运算符的目数而进行语法树的组合，只要进行分词就足够了。

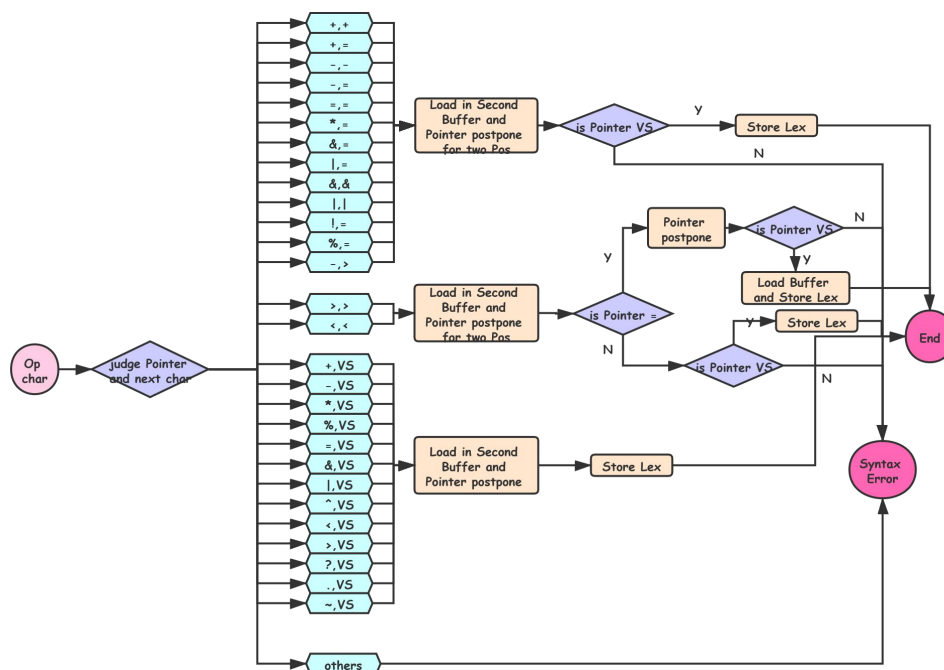


Fig.5 其他运算符处理逻辑 Logic3

上图展示了我们对于其他运算符的处理逻辑，原本的 `[operator]+'`  这样的正则表达式，会匹配到大量的不合法的运算符，我们此处需要进行语法的报错。VS 表示的 **valid and space**，指代字母，数字，下划线，四大空字符，是可以直接跟在运算符后面的合法字符。本词法分析器不承认以下类型的表述：

$$a = b ++ - * \& c;$$

如果要表示这样的语义，建议使用括号这一界符。本逻辑处理了大量的不符合语法规则且于运算符相关的报错。

## 2.5 数字常量类词素的处理逻辑

数字常量主要分整数类型以及浮点类型，其相差一个小数点，除此之外，本文还考虑了科学计数法，二进制数，八进制数，十六进制数这四种类型的数字常量。并且，**long** 类型的整数在结尾由数字 **L**，**float** 类型的浮点数在结尾存在 **F**，我们给出上述类型的数字常量的正则表达式形式：

$$Integer ::= [0-9]^+$$

$$Floating ::= [0-9]^+.[0-9]^+$$

$$ScienceNum ::= [Integer | Floating] \{ E \} \{ [+ -] \} \{ 0, 1 \} Integer$$

$$BinaryNum ::= [0b | 0B] [01]^+$$

$$OctoNum ::= [0o | 0O] [0-7]^+$$

$$HexNum ::= [0X | 0x] [0-9A-Fa-f]^+$$



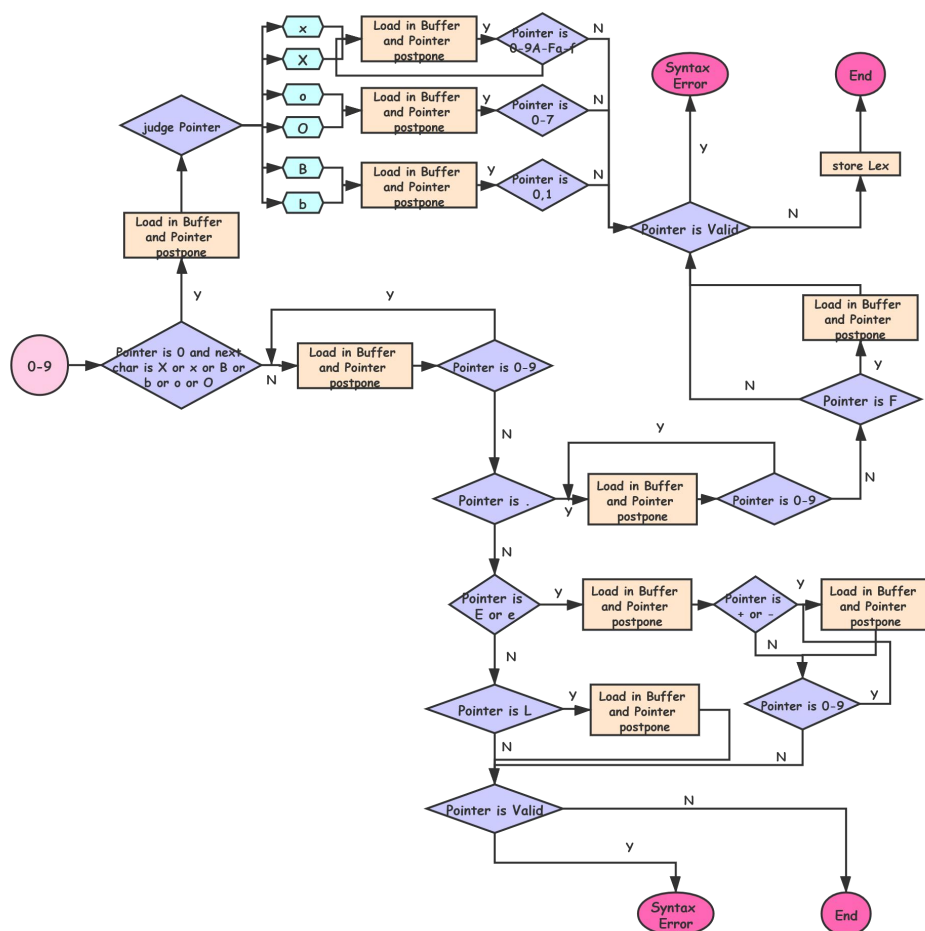


Fig.6 数字类型常数处理逻辑 Logic1

通过以上判别，我们能够分离出数字类型的常数，如果存在数字类型的常数由格式错误时，比如常数后面直接跟非法字母，那么也能够在此部分进行报错处理。

## 2.6 标识符或关键字词素的处理逻辑

标识符与关键字的正规表达式形式上是一样的，所以我们在分词时先将其归为一类，之后，直接遍历关键词表进行匹配即可，表达式如下：

$$Identifier ::= [A-Za-z] + [A-Za-z0-9]^*$$

由于其形式上比较单一，在进行分词提取时逻辑也比较简单，之后，我们会将其填入符号表，标识符实际意义上可以代表变量名，函数名，类名，枚举类型，结构体名，宏替换的常数等等，暂时不进行区分，这里只是注重于分词本身：

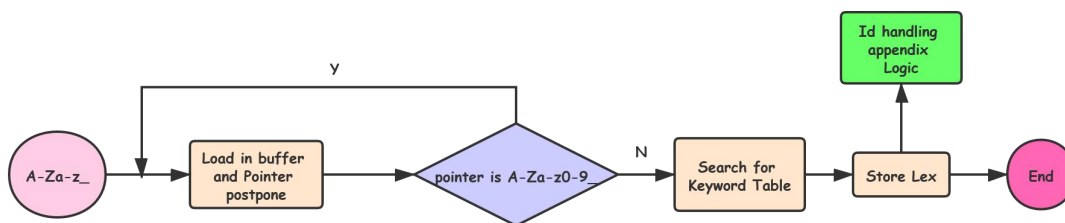


Fig.7 标识符（关键词）处理逻辑 Logic2

上图展示了，对于正则表达式的匹配过程，在结尾处多了两个步骤，一个是对于关键词表进行搜索匹配，决定是属于标识符还是关键字，另一个是在存储 LexResult 以后，还有一个对于符号表的查填过程，其作用域的判定也在整个 appendix logic 内，不是本次实验的重点，所以不进行详述。下面给出实验中使用关键字表，与上一次实验是一样的：

Table.1 关键字表

实验中用于匹配的关键词（C++版本）

if	define	auto	bool	break	case	const_cast	char	class	const
catch	continue	default	delete	do	double	dynamic_cast	else	enum	explicit
extern	false	float	for	friend	goto	include	inline	int	long
mutable	namespace	new	operator	private	protected	public	register	return	short
signed	sizeof	static	struct	switch	template	reinterpret_cast	this	throw	true
try	typedef	typeid	typename	union	unsigned	static_cast	using	virtual	void
volatile	while								

## 2.7 字符或字符串常数词素的处理逻辑

最后，常量除了数字型常量以及宏替换以外，还有字符类型以及字符串类型常量，本文对于字符类型的常量，考虑到两种情况，一种是单个字符，另一种是使用了转义字符。在语法检查的时候，如果单引号之内的字符数量不正确，会报一个语法错误：

$char\_CONST ::= '[\backslash]\{0,1\}[ASCII]\{1\}'$

具体的处理逻辑并不复杂，只是判断单引号是否匹配，是否存在转义字符，字符长度是否为 1：

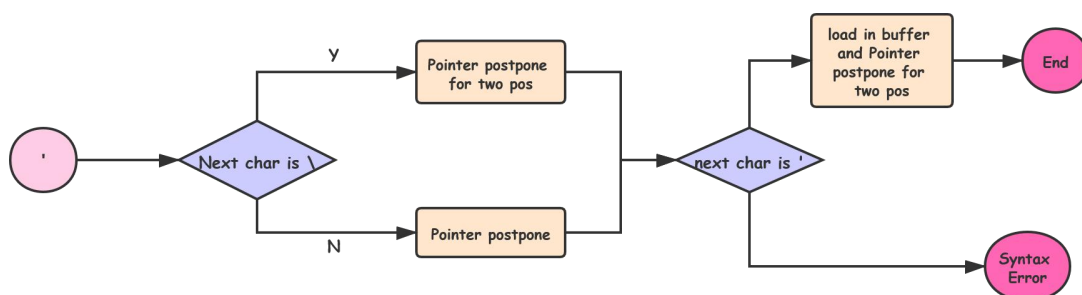


Fig. 8 字符常量给处理逻辑 Logic7

上面的语法错误判定比较简单，如果当出现转义字符时，单引号后面两位不为单引号就报错；当没有出现转义字符时，若单引号后面的以为不为单引号就报错。

字符串在实际情况中也使用非常频繁，其使用时，要注意 C/C++ 语言规范中，是不支持跨行的，我们在进行逻辑判断也非常简单，去寻找同一个 Main Buffer 内的双引号是否匹配来极性报错处理。

$string ::= "[ASCII]^{*}"$

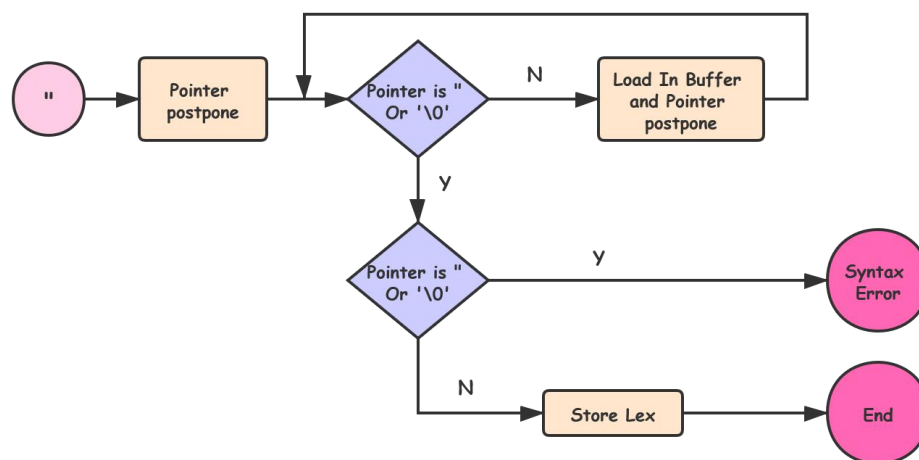


Fig. 9 字符常量处理逻辑 Logic6

除了以上逻辑以外，在处理宏定义时，本实验只是将其进行存储，而没有具体进行，在 gcc 的源代码中，专门设定了宏处理模块，其逻辑也是相当复杂的；在界符处理逻辑中，由于都是单字符，

所以非常简单，但 C++ 中有一类运算符 ':' 代表类的成员变量，类变量，成员方法，类方法等，单个的 ':' 却属于界符的，因此这里需要作为特例来进行处理。

## 2.8 本实验中在词法分析阶段考虑到的语法报错机制

本实验由于不是一个实际的项目，因此其实践意义大于工程意义，工程周期也较短一些，有一类非常重要的问题没有进行涉及，就是编码问题。使用 ASCII 编码的文件，每一个字符都是 1 个字节，严格属于 0-255 之间，其中 0-127 是基本的 ASCII 码，当编码出现问题是，也是编译程序需要进行处理的范畴。

我们本次实验进行的报错处理机制，出于两个角度来考虑，第一，是经常遇到，第二，实在词法分析部分进行处理代价并不是太大，并用单独区书写很复杂的逻辑，而用简单的分支语句就能够完成：

**Table.2 实验中本模块处理的语法错误**

错误编号	错误类型	错误名称	错误具体信息	处理模块
1	Syntax Error	多行注释无结尾	一直到文件结束处，扫描不到*/	LexAnalyser
2	Syntax Error	数字常量格式冗余	数字常量后面紧跟有非法字符，如字母，下划线等等	LexAnalyser
3	Syntax Error	数字常量格式缺省	浮点类型的点，或者科学计数法的 E 后面没有数字	LexAnalyser
4	Syntax Error	非法运算符	合法运算符后面跟有非法字符，主要也是有些运算符号	LexAnalyser
5	Syntax Error	字符串无结尾	字符串在改行以内找不到双引号结尾	LexAnalyser
6	Syntax Error	字符常量格式错误	字符常量后面无单引号，或者单引号内的字符过多	LexAnalyser
7	Syntax Error	非法字符	在读入缓冲区时读到非法字符	LexAnalyser

由于工程时间优先，仅仅在词法分析处理了七类简单的错误，当扫描到错误时，终止 lexanalysis 程序，并且直接在控制台打印出错误的行数，由于并没有完成对于标识符的归类，在词法分析部分暂时无法打印出具体的模块与函数。

## 2.9 作用域控制的改进——tabSpaceRecorder 数据结构

在上一个实验中，使用栈(Stack)来标识符所属的作用域，在扫描到界符 '{' 以及 '}' 的时候，就进行入栈出栈操作，并对于符号表中的标识符记录其所属的作用域，但是忽略了一类情况，很多程序员在进行作用域划分是，喜欢使用缩进来进行，tab 键或者四个空格来表示作用域的进出，代表是 for 循环，while 循环，switch 分支，if 分支，由于本词法分析程序以及进行了行号的记录，于是，设计了 tabSpaceRecorder 来存放每一行之前的缩进数量，写在 loadBuffer 整个函数，也就是更新 Main Buffer 的时候。

在进行作用域的进出时，即使没有界符 {} 的划分，也可以直接将缩进的数量相减，就可以识别出此时需不需要更新作用域了。

## 3. 实验结果与分析

### 3.1 部分源码解析

本次实验设计的 LexAnalyser 的结构与实验一中存在明显的不同，下面进行展示：

```

1.  class LexAnalyser
2.  {
3.  private:
4.      std::ifstream* fin; // 文件指针设计在词法分析器内部
5.      char* buffer; // 主缓冲区
6.      int curLine = 0; // 当前行号
7.      int cur = 0; // LexAnalyser 在 Main Buffer 中的指针 index 形式
8.      void init();
9.      void recordSpace();
10.     bool loadBuffer(std::ifstream* fin);
11.     void shiftToNextToken();
12.     bool searchForNoteEndType_1();

```



```

13.     bool searchForNoteEndType_2();
14.     void extractMacro();
15.     void handleSeperator();
16.     void handleOperator();
17.     void handlerIdentifier();
18.     void handleNumber();
19.     void handleCharConst();
20.     void handleStringConst();
21.     void handleLeftRightAngleBra();
22.     void storeLex(LEX_TYPE type, std::string tokenContent);
23.     void analyseToken();
24. public:
25.     LexAnalyser(std::ifstream& inFile) {
26.         this->fin = &inFile;
27.         init();
28.     }
29.     void lex();
30.     void printLexRes();
31.     void getLexRes(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> &Res);
32. };

```

上述代码是 LexAnalyser 类在头文件中的一部分，其中可以注意到几点：

- 文件流 fstream 的指针保存在词法分析器的内部，这样在进行状态转换时，不会拘束于数学模型，可以自由进行 buffer 的读写，以及指针的移动。
- Main Buffer 这一结构采用成员变量，而 Second Buffer 采用的是临时变量，因为 Main Buffer 的使用需要跨越几个 DFA 周期，但是 Second Buffer 不用。
- Pointer 这一结构以 Buffer[cur] 这一形式给出，而 curLine 当前行号也保存在词法分析器内部。
- loadBuffer 刷新主缓冲区以及 shiftToNextToken，将 Pointer 移动到下一个 Token 的首元素作为私有的方法，是出于安全性能考虑，内部结构对于整体是透明的。
- 词法分析器对外的接口，仅有构造器，lex（词法分析开始分析），printLexRes（打印结果），getLexRes（将结果传入到主模块中）。
- 分块逻辑从 analyseToken 中开始，每执行一次该命令，都是一个 DFA 周期中的一个分析周期，其中根据首元素的类型，会条用 handleSeperator, handleOperator, handlerIdentifier, handleNumber, handleCharConst, handleStringConst, handleLeftRightAngle 进行子逻辑的划分。
- extractMacro 中的宏处理比较简单，后续的语义分析中，如有需要会有所更新。

### 3.2 实战实验结果分析

本次实验所写的词法分析器中主要实现文件 LexAnalyser.cpp 一共是 981 行，我们就是用这个文件来进行词法分析的测试，里面除了比较常见的函数，分支，循环以外，还用到了比较复杂的数组，指针，运算符，泛型模板等等，于是用编译程序来测试编译程序自己：

```

34  /******
35  * 函数名:  recordSpace
36  * 作用:    记录每一行的缩进个数，记录在spaceRecorder中
37  * 参数:    无
38  *****/
39  void LexAnalyser::recordSpace() {
40      int it = 0, count = 0;
41      while (buffer[it] == ' ' || buffer[it] == '\t') {
42          count += (buffer[it++] == '\t') ? TAB_SPACE : 1;
43      }
44      spaceRecord.push_back(pair<int, int>(curLine, count));
45  }
46

```

Fig.11 测试案例源代码

```

Line:29, (SEPARATOR, ;)      Line:30, (SEPARATOR, })      Line:31, (KEYWORD, return)
Line:31, (IDENTIFIER, inFile) Line:31, (SEPARATOR, ;)      Line:32, (SEPARATOR, })
Line:39, (KEYWORD, void)     Line:39, (IDENTIFIER, LexAnalyser) Line:39, (OPERATOR, ::)
Line:39, (IDENTIFIER, recordSpace) Line:39, (SEPARATOR, () Line:39, (SEPARATOR, ))
Line:39, (SEPARATOR, {)      Line:40, (KEYWORD, int)      Line:40, (IDENTIFIER, it)
Line:40, (OPERATOR, =)       Line:40, (CONST, 0)         Line:40, (SEPARATOR, ,)
Line:40, (IDENTIFIER, count) Line:40, (OPERATOR, =)       Line:40, (CONST, 0)
Line:40, (SEPARATOR, ;)      Line:41, (KEYWORD, while)   Line:41, (SEPARATOR, ()
Line:41, (IDENTIFIER, buffer) Line:41, (SEPARATOR, ,)     Line:41, (IDENTIFIER, it)
Line:41, (SEPARATOR, ,)      Line:41, (OPERATOR, ==)     Line:41, (CONST, ,)
Line:41, (OPERATOR, [])      Line:41, (IDENTIFIER, buffer) Line:41, (SEPARATOR, ,)
Line:41, (IDENTIFIER, it)    Line:41, (SEPARATOR, ,)     Line:41, (OPERATOR, ==)
Line:41, (CONST, t)          Line:41, (SEPARATOR, ,)     Line:41, (SEPARATOR, {)
Line:42, (IDENTIFIER, count) Line:42, (OPERATOR, +=)     Line:42, (SEPARATOR, ()
Line:42, (IDENTIFIER, buffer) Line:42, (SEPARATOR, ,)     Line:42, (IDENTIFIER, it)
Line:42, (OPERATOR, ++       Line:42, (SEPARATOR, ,)     Line:42, (OPERATOR, ==)
Line:42, (CONST, t)          Line:42, (SEPARATOR, ,)     Line:42, (OPERATOR, ?)
Line:42, (IDENTIFIER, TAB_SPACE) Line:42, (SEPARATOR, :)    Line:42, (CONST, 1)
Line:42, (SEPARATOR, ;)      Line:43, (SEPARATOR, })      Line:44, (IDENTIFIER, spaceRecord)
Line:44, (OPERATOR, .)        Line:44, (IDENTIFIER, push_back) Line:44, (SEPARATOR, ()
Line:44, (IDENTIFIER, pair)   Line:44, (SEPARATOR, <)     Line:44, (KEYWORD, int)
Line:44, (SEPARATOR, ,)      Line:44, (KEYWORD, int)     Line:44, (SEPARATOR, >)
Line:44, (SEPARATOR, ()      Line:44, (IDENTIFIER, curLine) Line:44, (SEPARATOR, ,)
Line:44, (IDENTIFIER, count) Line:44, (SEPARATOR, ,)     Line:44, (SEPARATOR, ,)
Line:44, (SEPARATOR, ;)      Line:45, (SEPARATOR, })     Line:52, (KEYWORD, bool)
Line:52, (IDENTIFIER, LexAnalyser) Line:52, (OPERATOR, ::)    Line:52, (IDENTIFIER, loadBuffer)
Line:52, (SEPARATOR, ()      Line:52, (IDENTIFIER, ifstream) Line:52, (OPERATOR, *)

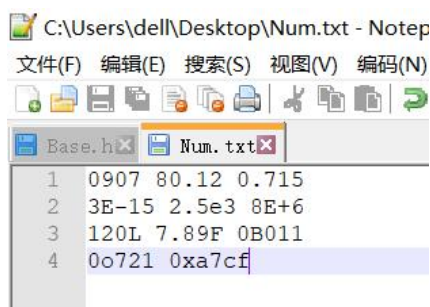
```

Fig.12 词法分析结果

从上述测试结果可以看到，红色矩形框是高级程序所对应的词法分析结果。其中，[1]蓝色框出的两个 Token 是二元运算符，说明其被成功识别，[2]黄色框中的两个 Token 对应了上面的泛型，其此处表达的语义是界符，而不是运算符，也被成功识别出，识别依据实际上是前面的 pair 这个词；[3]绿色框中的三个 token，it 是变量名被识别为标识符，while 作为关键字表中的词也被识别出，pair 虽然是 STL 库中的类名，有模板泛型的形式，但不是保留字，依然被识别为标识符；[4]天蓝色框中的 t 是字符型常量被成功识别；[5] 蓝绿色框中的::也值得注意，其代表成员，是运算符，尽管是界符。

### 3.3 专向测试结果分析

之后我们专门对于一些特殊字符拿出来组成文件，进行专项测试：



词法分析词素:

```

Line:1, (CONST, 0907)
Line:1, (CONST, 80.12)
Line:1, (CONST, 0.715)
Line:2, (CONST, 3E-15)
Line:2, (CONST, 2.5e3)
Line:2, (CONST, 8E+6)
Line:3, (CONST, 120L)
Line:3, (CONST, 7.89F)
Line:3, (CONST, 0B011)
Line:4, (CONST, 0o721)
Line:4, (CONST, 0xa7cf)

```

Fig.13 常数专项测试源文件以及结果

常数专项测试文件中包含了各类合法的整数，浮点数，科学计数法，float 常数，long 常数，二进制，八进制，十六进制的类型，都是本词法分析器能识别的。



词法分析词素:

```

Line:1, (OPERATOR, *)
Line:1, (IDENTIFIER, var1)
Line:1, (OPERATOR, &)
Line:1, (IDENTIFIER, var2)
Line:1, (OPERATOR, %=)
Line:2, (OPERATOR, >>=)
Line:2, (OPERATOR, &=)
Line:2, (OPERATOR, ^)
Line:3, (OPERATOR, !=)
Line:3, (OPERATOR, ~)
Line:4, (OPERATOR, >)
Line:4, (OPERATOR, <<)
Line:4, (IDENTIFIER, vector)
Line:4, (SEPARATOR, <)
Line:4, (KEYWORD, int)
Line:4, (SEPARATOR, ,)
Line:4, (KEYWORD, double)
Line:4, (SEPARATOR, >)
Line:5, (OPERATOR, ::)
Line:5, (SEPARATOR, :)

```

Fig.14 运算符专项测试源文件以及结果

可以看到指针和取地址符被标注为运算符，合法的一元，二元，三元运算符大多被考虑进去。针对<以及>，单独的比较和移位被识别为运算符，但是作为模板使用时，识别为界符。::在作成员时，被标注为运算符，单独作为冒号时，标注为界符。

## 4. 结束语

本次词法分析实验，从字符流驱动型 DFA 改进为词素首元素驱动型 DFA，不拘泥于数学表达式，[1] 在结构上，改进了 buffer 缓冲区，文件指针的从属关系；[2] 在识别逻辑上，将其进行分块，降低工程耦合性；[3] 在功能上，增加了报错机制，缩进识别机制，运算符识别拓展了'::', '>', '^', '>='等符号，增加了泛型机制，数字常量识别拓展了二，八，十六进制，科学计数法等；[4] 在原理上，将一个 token 周期划分为寻址周期和识别周期，一个周期 DFA 完整运转一次。

并且从实验的角度，进行了实战测试，以及专项测试，测试效果良好，能够达到我们预期的分词效果，可以较好为后续的语法分析，语义分析，中间代码分析作牵制工作。

其也有不足之处，主要是宏处理，因为在词法分析之间，实际 gcc 编译器还对于其进行了相关库链接，宏替换等工作，其工程量也非常巨大，但是本实验是在一周之内完成的，不是一个成熟的项目，编码能力于 GNU 团队有较大差距，所以没有专门开辟模块进行处理。

## 5. 附录

### 5.1 工程结构于配置文件说明

在附件中，主要展示的是实验参数以及部分源代码，以及工程的说明，以下是项目的结构：（后续肯定会接着写）

```
|---- Compiler
|   |---- Compiler
|   |   |---- Base.h
|   |   |---- LexAnalyser.h
|   |   |---- LexAnalyser.cpp
|   |   |---- main.cpp
|   |---- data
|   |   |---- testCompiler1.cpp
|   |   |---- testCompiler2.cpp
|   |   |---- Num.txt
|   |   |---- Op.txt
|   |---- result
|   |---- Readme.txt
|   |---- run.bat
```

本实验的文件输入路径用宏定义的方式写在 LexAnalyser.h 头文件中，测试环境使用了 Windows10，VS2019，内存 16G，处理器 i7 第九代，配置中等偏上。

### 5.2 LexAnalyser.h

```
1.  /*****
2.  * 模块名: LexAnalyser
3.  * 文件名: LexAnalyser.h
4.  * 依赖文件: Base.h
5.  * 作用: 对于 C/C++ 语言进行词法分析
6.  * 作者: 陈晓飞
7.  * 时间: 2020.5.25
8.  * 版本: Version 2.1
9.  *****/
10. #ifndef _LEXANALYSER_H_
11. #define _LEXANALYSER_H_
```

```

12. #include "Base.h"
13. #include <iostream>
14. #include <vector>
15. #define MAX_BUFFER_SIZE 1000 // Main Buffer 的最大容量
16. #define inputDir "C://Users/dell/Desktop/testCompiler2.txt"// 文件的输入路径
17.
18. /*****
19.  * 函数名: openFile
20.  * 作用: 打开高级程序文件
21.  * 参数: dir: 高级程序的路径[in]
22.  *****/
23. std::ifstream openFile(std::string dir);
24. /*****
25.  * 函数名: searchKeyword
26.  * 作用: 查找关键词表是否存在该表项
27.  * 参数: id: 待匹配表项[in]
28.  *****/
29. bool searchKeyword(std::string id);
30. /*****
31.  * 函数名: isTemplateClass
32.  * 作用: 查找泛型表是否存在该表项
33.  * 参数: name: 待匹配表项[in]
34.  *****/
35. bool isTemplateClass(std::string name);
36. /*****
37.  * 函数名: isValidID
38.  * 作用: 判定是否为字母, 数字, 下划线
39.  * 参数: id: 待匹配字符[in]
40.  *****/
41. inline bool isValidID(char id);
42. /*****
43.  * 函数名: isOp
44.  * 作用: 判断是否为运算符
45.  * 参数: op: 待判定运算符[in]
46.  *****/
47. inline bool isOp(char op);
48.
49. void raiseSyntaxError(int line, SYNTAX_ERROR errorId);
50.
51. /*****
52.  * 类名: LexAnalyser
53.  * 作用: 进行词法分析
54.  *****/
55. class LexAnalyser
56. {
57. private:
58.     std::ifstream* fin; // 文件指针设计在词法分析器内部
59.     char* buffer; // 主缓冲区
60.     int curLine = 0; // 当前行号
61.     int cur = 0; // LexAnalyser 在 Main Buffer 中的指针 index 形式
62.     /*****
63.     * 函数名: init
64.     * 作用: 初始化 LexAnalyser, 分配相应内存
65.     * 参数: 无
66.     *****/
67.     void init();
68.     /*****
69.     * 函数名: recordSpace
70.     * 作用: 记录每一行的缩进个数, 记录在 spaceRecorder 中
71.     * 参数: 无
72.     *****/
73.     void recordSpace();
74.     /*****
75.     * 函数名: loadBuffer
76.     * 作用: 在 Main Buffer 中更新一行, 并将全局指针指向 Main Buffer 的首地址
77.     * 参数: fin: 高级程序文件输入流指针[in/out]
78.     *****/
79.     bool loadBuffer(std::ifstream* fin);
80.     /*****
81.     * 函数名: shiftToNextToken
82.     * 作用: 从当前位置开始寻找下一个 token 的首位字符, 存储在 buffer[cur]中
83.     * 参数: 无
84.     *****/
85.     void shiftToNextToken();
86.     /*****
87.     * 函数名: searchForNoteEndType_1
88.     * 作用: 对于多行注释进行处理
89.     * 参数: 无
90.     *****/
91.     bool searchForNoteEndType_1();
92.     /*****
93.     * 函数名: searchForNoteEndType_2
94.     * 作用: 对于单行注释进行处理
95.     * 参数: 无
96.     *****/
97.     bool searchForNoteEndType_2();
98.     /*****
99.     * 函数名: extractMacro
100.    * 作用: 提取该宏
101.    * 参数: 无
102.    *****/
103.    void extractMacro();
104.    /*****
105.    * 函数名: handleSeperator
106.    * 作用: 对于 buffer[cur]所在的界符进行处理
107.    * 参数: 无
108.    *****/
109.    void handleSeperator();
110.    /*****
111.    * 函数名: handleOperator
112.    * 作用: 对于 buffer[cur]开始的运算符进行处理
113.    * 参数: 无
114.    *****/

```



```

115.     void handleOperator();
116.     /*****
117.      * 函数名:   handlerIdentifier
118.      * 作用:    对于 buffer[cur]开始的标识符进行处理
119.      * 参数:    无
120.      *****/
121.     void handlerIdentifier();
122.     /*****
123.      * 函数名:   handleNumber
124.      * 作用:    对于 buffer[cur]开始的数字常量进行处理
125.      * 参数:    无
126.      *****/
127.     void handleNumber();
128.     /*****
129.      * 函数名:   handleCharConst
130.      * 作用:    对于 buffer[cur]开始的字符常量进行处理
131.      * 参数:    无
132.      *****/
133.     void handleCharConst();
134.     /*****
135.      * 函数名:   handleStringConst
136.      * 作用:    对于 buffer[cur]开始的字符串常量进行处理
137.      * 参数:    无
138.      *****/
139.     void handleStringConst();
140.     /*****
141.      * 函数名:   handleLeftRightAngleBra
142.      * 作用:    对于 buffer[cur]为<>的情况，区分泛型和运算符
143.      * 参数:    无
144.      *****/
145.     void handleLeftRightAngleBra();
146.     /*****
147.      * 函数名:   storeLex
148.      * 作用:    将词法分析的词素及类型存储到 tokenRes 中
149.      * 参数:    type:      词素的类型      [in]
150.      *          tokenContent : 词素的具体内容 [in]
151.      *****/
152.     void storeLex(LEX_TYPE type, std::string tokenContent);
153.     /*****
154.      * 函数名:   analyseToken
155.      * 作用:    启动分块逻辑，对于 buffer[cur]开始的词素进行分析
156.      * 参数:    无
157.      *****/
158.     void analyseToken();
159. public:
160.     /*****
161.      * 函数名:   LexAnalyser
162.      * 作用:    构造器
163.      * 参数:    inFile: 私有文件指针[in]
164.      *****/
165.     LexAnalyser(std::ifstream& inFile) {
166.         this->fin = &inFile;
167.         init();
168.     }
169.     /*****
170.      * 函数名:   lex
171.      * 作用:    从第一个字符开始，启动 DFA 词法分析周期，制导文件结束
172.      * 参数:    无
173.      *****/
174.     void lex();
175.     /*****
176.      * 函数名:   printLexRes
177.      * 作用:    打印词法分析结果
178.      * 参数:    无
179.      *****/
180.     void printLexRes();
181.     /*****
182.      * 函数名:   getLexRes
183.      * 作用:    将词法分析结果输出给外界
184.      * 参数:    Res      输出句柄[in/out]
185.      *****/
186.     void getLexRes(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> &Res);
187. };
188. #endif

```

### 5.3 LexAnalyser.cpp

```

1.     /*****
2.      * 模块名:   LexAnalyser
3.      * 文件名:   LexAnalyser.cpp
4.      * 依赖文件: LexAnalyser.h
5.      * 作用:    对于 C/C++语言进行词法分析
6.      * 作者:    陈晓飞
7.      * 时间:    2020.5.25
8.      * 版本:    Version 2.1
9.      *****/
10.    #include "LexAnalyser.h"
11.    #include <iostream>
12.    #include <fstream>
13.    #include <istream>
14.    #include <string>
15.    #include <vector>
16.    #include <algorithm>
17.    #include <iomanip>
18.    #include <stack>
19.    using namespace std;
20.
21.    /*****
22.      * 函数名:   openFile
23.      * 作用:    打开高级程序文件
24.      * 参数:    dir: 高级程序的路径[in]
25.      *****/

```



```

26. std::ifstream openFile(std::string dir) {
27.     std::ifstream inFile(dir);
28.     if (!inFile.is_open())
29.     {
30.         std::cout << "未成功打开文件" << std::endl;
31.     }
32.     return inFile;
33. }
34.
35. /*****
36. * 函数名: recordSpace
37. * 作用: 记录每一行的缩进个数, 记录在 spaceRecorder 中
38. * 参数: 无
39. *****/
40. void LexAnalyser::recordSpace() {
41.     int it = 0, count = 0;
42.     while (buffer[it] == ' ' || buffer[it] == '\t') {
43.         count += (buffer[it++] == '\t') ? TAB_SPACE : 1;
44.     }
45.     spaceRecord.push_back(pair<int, int>(curLine, count));
46. }
47.
48. /*****
49. * 函数名: loadBuffer
50. * 作用: 在 Main Buffer 中更新一行, 并将全局指针指向 Main Buffer 的首地址
51. * 参数: fin: 高级程序文件输入流指针[in/out]
52. *****/
53. bool LexAnalyser::loadBuffer(ifstream* fin) {
54.     if ((*fin).is_open() && !(*fin).eof()) {
55.         (*fin).getline(buffer, MAX_BUFFER_SIZE);
56.         curLine++;
57.         recordSpace();
58.         return true;
59.     }
60.     return false;
61. }
62.
63. /*****
64. * 函数名: init
65. * 作用: 初始化 LexAnalyser, 分配相应内存
66. * 参数: 无
67. *****/
68. void LexAnalyser::init() {
69.     buffer = (char*)malloc(MAX_BUFFER_SIZE * sizeof(char));
70.     if (buffer != NULL) {
71.         memset(buffer, '\0', MAX_BUFFER_SIZE * sizeof(char));
72.     }
73. }
74.
75. /*****
76. * 函数名: shiftToNextToken
77. * 作用: 从当前位置开始寻找下一个 token 的首位字符, 存储在 buffer[cur] 中
78. * 参数: 无
79. *****/
80. void LexAnalyser::shiftToNextToken() {
81.     while (buffer[cur] == ' ' || buffer[cur] == '\0' || buffer[cur] == '\n' || buffer[cur] == '\r' || buffer[cur] == '\t') {
82.         if (buffer[cur] == '\0' || buffer[cur] == '\n') { // 如果遇到回车符, 或者程序刚刚开始, 刷新缓冲区, 重置 cur 指针
83.             if ((*this->fin).eof()) { // 高级程序读取结束时, 直接中断
84.                 return;
85.             }
86.             loadBuffer(this->fin);
87.             cur = 0;
88.         }
89.         if (buffer[cur] == ' ' || buffer[cur] == '\t') {
90.             cur++;
91.         }
92.     }
93. }
94.
95. /*****
96. * 函数名: storeLex
97. * 作用: 将词法分析的词素及类型存储到 tokenRes 中
98. * 参数: type: 词素的类型[in]
99. *         tokenContent: 词素的具体内容[in]
100. *****/
101. void LexAnalyser::storeLex(LEX_TYPE type, string tokenContent) {
102.     // cout << "Store:" << tokenContent << endl;
103.     tokenRes.push_back(pair<int, pair<LEX_TYPE, string>>(curLine, pair<LEX_TYPE, string>(type, tokenContent)));
104. }
105.
106. /*****
107. * 函数名: searchForNoteEndType_1
108. * 作用: 对于多行注释进行处理
109. * 参数: 无
110. *****/
111. bool LexAnalyser::searchForNoteEndType_1() {
112.     // cout << "进入注释.Line:" << curLine << endl;
113.     cur += 2;
114.     while (true) {
115.         if (buffer[cur] == '*' && buffer[cur + 1] == '/') { // 注释结束
116.             // cout << "注释结束.Line:" << curLine << endl;
117.             cur += 2;
118.             return true;
119.         }
120.         else if (buffer[cur] == '\0' || buffer[cur] == '\n') && (*this->fin).eof()) { // 程序结束
121.             break;
122.         }
123.         else { // 注释内部
124.             cur++;
125.             shiftToNextToken();
126.         }
127.     }
128.     return false;
129. }
130.

```

```

131.  /*****
132.  * 函数名:    searchForNoteEndType_2
133.  * 作用:      对于单行注释进行处理
134.  * 参数:      无
135.  *****/
136.  bool LexAnalyser::searchForNoteEndType_2() {
137.      // cout << "进入第二类注释" << endl;
138.      while (buffer[cur] != '\n' && buffer[cur] != '\0') {
139.          cur++;
140.      }
141.      if (buffer[cur] == '\0' || (*this->fin).eof()) { // 搜索到行尾, 或者程序结束
142.          shiftToNextToken();
143.          return true;
144.      }
145.      else { // 出现错误-非法字符'\0'
146.          return false;
147.      }
148.  }
149.  /*****
150.  * 函数名:    extractMacro
151.  * 作用:      提取该宏
152.  * 参数:      无
153.  *****/
154.  void LexAnalyser::extractMacro() {
155.      macroBin.push_back(string(buffer).substr((size_t)cur + 1, string(buffer).size() - 1));
156.      loadBuffer(this->fin);
157.  }
158.  /*****
159.  * 函数名:    handleSperator
160.  * 作用:      对于 buffer[cur]所在的界符进行处理
161.  * 参数:      无
162.  *****/
163.  void LexAnalyser::handleSperator() {
164.      if (buffer[cur] == ':' && buffer[cur + 1] == ':') { // 特例,::算作运算符,::算作界符
165.          storeLex(LEX_TYPE::OPERATOR, "::");
166.          cur += 2;
167.      }
168.      else {
169.          storeLex(LEX_TYPE::SEPERATOR, string(1, buffer[cur]));
170.          cur++;
171.      }
172.  }
173.  /*****
174.  * 函数名:    isOp
175.  * 作用:      判断是否为运算符
176.  * 参数:      op: 待判定运算符[in]
177.  *****/
178.  inline bool isOp(char op) {
179.      return op == '+' || op == '-' || op == '/' || op == '%' || op == '=' || op == '|' || op == '<' || op == '>' || op == '!' || op == '.' ||
180.      | op == '~' || op == '?' || op == '^';
181.  }
182.  /*****
183.  * 函数名:    handleOperator
184.  * 作用:      对于 buffer[cur]开始的运算符进行处理
185.  * 参数:      无
186.  *****/
187.  void LexAnalyser::handleOperator() {
188.      char op1 = buffer[cur];
189.      switch (op1)
190.      {
191.      case '<':
192.          if (buffer[cur + 1] == '<') {
193.              if (buffer[cur + 2] == '=') {
194.                  storeLex(LEX_TYPE::OPERATOR, "<=");
195.                  cur += 3;
196.              }
197.              else if (isOp(buffer[cur + 2])) {
198.                  raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
199.              }
200.              else {
201.                  storeLex(LEX_TYPE::OPERATOR, "<");
202.                  cur += 2;
203.              }
204.          }
205.          else if (buffer[cur + 1] == '=') {
206.              if (isOp(buffer[cur + 2])) {
207.                  raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
208.              }
209.              else {
210.                  storeLex(LEX_TYPE::OPERATOR, "=");
211.                  cur += 2;
212.              }
213.          }
214.          else {
215.              if (isOp(buffer[cur + 1])) {
216.                  raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
217.              }
218.              else {
219.                  storeLex(LEX_TYPE::OPERATOR, "<");
220.                  cur++;
221.              }
222.          }
223.          break;
224.      case '>':
225.          if (buffer[cur + 1] == '>') {
226.              if (buffer[cur + 2] == '=') {
227.                  storeLex(LEX_TYPE::OPERATOR, ">=");
228.                  cur += 3;
229.              }
230.              else if (isOp(buffer[cur + 2])) {
231.                  raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
232.              }
233.              else {
234.                  storeLex(LEX_TYPE::OPERATOR, ">");
235.                  cur += 2;
236.              }
237.          }
238.          else {
239.              if (isOp(buffer[cur + 1])) {
240.                  raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
241.              }
242.              else {
243.                  storeLex(LEX_TYPE::OPERATOR, ">");
244.                  cur++;
245.              }
246.          }
247.          break;
248.      }
249.  }

```

```

236.     }
237.     else if (buffer[cur + 1] == '=' ) {
238.         if (isOp(buffer[cur + 2])) {
239.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
240.         }
241.         else {
242.             storeLex(LEX_TYPE::OPERATOR, ">=");
243.             cur += 2;
244.         }
245.     }
246.     else {
247.         if (isOp(buffer[cur + 1])) {
248.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
249.         }
250.         else {
251.             storeLex(LEX_TYPE::OPERATOR, ">");
252.             cur++;
253.         }
254.     }
255.     break;
256. case '+':
257.     if (buffer[cur + 1] == '=' ) {
258.         if (isOp(buffer[cur + 2])) {
259.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
260.         }
261.         else {
262.             storeLex(LEX_TYPE::OPERATOR, "+=");
263.             cur += 2;
264.         }
265.     }
266.     else if (buffer[cur + 1] == '+' ) {
267.         if (isOp(buffer[cur + 2])) {
268.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
269.         }
270.         else {
271.             storeLex(LEX_TYPE::OPERATOR, "++");
272.             cur += 2;
273.         }
274.     }
275.     else {
276.         if (isOp(buffer[cur + 1])) {
277.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
278.         }
279.         else {
280.             storeLex(LEX_TYPE::OPERATOR, "+");
281.             cur++;
282.         }
283.     }
284.     break;
285. case '-':
286.     if (buffer[cur + 1] == '=' ) {
287.         if (isOp(buffer[cur + 2])) {
288.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
289.         }
290.         else {
291.             storeLex(LEX_TYPE::OPERATOR, "-=");
292.             cur += 2;
293.         }
294.     }
295.     else if (buffer[cur + 1] == '-' ) {
296.         if (isOp(buffer[cur + 2])) {
297.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
298.         }
299.         else {
300.             storeLex(LEX_TYPE::OPERATOR, "--");
301.             cur += 2;
302.         }
303.     }
304.     else if (buffer[cur + 1] == '>' ) {
305.         if (isOp(buffer[cur + 2])) {
306.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
307.         }
308.         else {
309.             storeLex(LEX_TYPE::OPERATOR, ">");
310.             cur += 2;
311.         }
312.     }
313.     else {
314.         if (isOp(buffer[cur + 1])) {
315.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
316.         }
317.         else {
318.             storeLex(LEX_TYPE::OPERATOR, "-");
319.             cur++;
320.         }
321.     }
322.     break;
323. case '*':
324.     if (buffer[cur + 1] == '=' ) {
325.         if (isOp(buffer[cur + 2])) {
326.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
327.         }
328.         else {
329.             storeLex(LEX_TYPE::OPERATOR, "*=");
330.             cur += 2;
331.         }
332.     }
333.     else {
334.         if (isOp(buffer[cur + 1])) {
335.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
336.         }
337.         else {
338.             storeLex(LEX_TYPE::OPERATOR, "**");
339.             cur++;
340.         }
341.     }
342.     break;
343. case '/':

```

```

344.     if (buffer[cur + 1] == '=') {
345.         if (isOp(buffer[cur + 2])) {
346.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
347.         }
348.         else {
349.             storeLex(LEX_TYPE::OPERATOR, "/=");
350.             cur += 2;
351.         }
352.     }
353.     else {
354.         if (isOp(buffer[cur + 1])) {
355.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
356.         }
357.         else {
358.             storeLex(LEX_TYPE::OPERATOR, "/");
359.             cur++;
360.         }
361.     }
362.     break;
363. case '%':
364.     if (buffer[cur + 1] == '=') {
365.         if (isOp(buffer[cur + 2])) {
366.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
367.         }
368.         else {
369.             storeLex(LEX_TYPE::OPERATOR, "%=");
370.             cur += 2;
371.         }
372.     }
373.     else {
374.         if (isOp(buffer[cur + 1])) {
375.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
376.         }
377.         else {
378.             storeLex(LEX_TYPE::OPERATOR, "%");
379.             cur++;
380.         }
381.     }
382.     break;
383. case '&':
384.     if (buffer[cur + 1] == '&') {
385.         if (isOp(buffer[cur + 2])) {
386.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
387.         }
388.         else {
389.             storeLex(LEX_TYPE::OPERATOR, "&&");
390.             cur += 2;
391.         }
392.     }
393.     else if (buffer[cur + 1] == '=') {
394.         if (isOp(buffer[cur + 2])) {
395.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
396.         }
397.         else {
398.             storeLex(LEX_TYPE::OPERATOR, "&=");
399.             cur += 2;
400.         }
401.     }
402.     else {
403.         if (isOp(buffer[cur + 1])) {
404.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
405.         }
406.         else {
407.             storeLex(LEX_TYPE::OPERATOR, "&");
408.             cur++;
409.         }
410.     }
411.     break;
412. case '|':
413.     if (buffer[cur + 1] == '|') {
414.         if (isOp(buffer[cur + 2])) {
415.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
416.         }
417.         else {
418.             storeLex(LEX_TYPE::OPERATOR, "||");
419.             cur += 2;
420.         }
421.     }
422.     else if (buffer[cur + 1] == '=') {
423.         if (isOp(buffer[cur + 2])) {
424.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
425.         }
426.         else {
427.             storeLex(LEX_TYPE::OPERATOR, "|=");
428.             cur += 2;
429.         }
430.     }
431.     else {
432.         if (isOp(buffer[cur + 1])) {
433.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
434.         }
435.         else {
436.             storeLex(LEX_TYPE::OPERATOR, "|");
437.             cur++;
438.         }
439.     }
440.     break;
441. case '=':
442.     if (buffer[cur + 1] == '=') {
443.         if (isOp(buffer[cur + 2])) {
444.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
445.         }
446.         else {
447.             storeLex(LEX_TYPE::OPERATOR, "==");
448.             cur += 2;
449.         }
450.     }
451.     else {

```

```

452.         if (isOp(buffer[cur + 1])) {
453.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
454.         }
455.         else {
456.             storeLex(LEX_TYPE::OPERATOR, "=");
457.             cur++;
458.         }
459.     }
460.     break;
461. case '?':
462.     if (isOp(buffer[cur + 1])) {
463.         raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
464.     }
465.     else {
466.         storeLex(LEX_TYPE::OPERATOR, ">");
467.         cur++;
468.     }
469.     break;
470. case '.':
471.     if (isOp(buffer[cur + 1])) {
472.         raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
473.     }
474.     else {
475.         storeLex(LEX_TYPE::OPERATOR, ".");
476.         cur++;
477.     }
478.     break;
479. case '!':
480.     if (buffer[cur + 1] == '=') {
481.         if (isOp(buffer[cur + 2])) {
482.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
483.         }
484.         else {
485.             storeLex(LEX_TYPE::OPERATOR, "!=");
486.             cur += 2;
487.         }
488.     }
489.     else {
490.         if (isOp(buffer[cur + 1])) {
491.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
492.         }
493.         else {
494.             storeLex(LEX_TYPE::OPERATOR, "!");
495.             cur++;
496.         }
497.     }
498.     break;
499. case '^':
500.     if (buffer[cur + 1] == '=') {
501.         if (isOp(buffer[cur + 2])) {
502.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
503.         }
504.         else {
505.             storeLex(LEX_TYPE::OPERATOR, "^=");
506.             cur += 2;
507.         }
508.     }
509.     else {
510.         if (isOp(buffer[cur + 1])) {
511.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
512.         }
513.         else {
514.             storeLex(LEX_TYPE::OPERATOR, "^");
515.             cur++;
516.         }
517.     }
518.     break;
519. case '~':
520.     if (isOp(buffer[cur + 1])) {
521.         raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
522.     }
523.     else {
524.         storeLex(LEX_TYPE::OPERATOR, "~");
525.         cur++;
526.     }
527.     break;
528. default:
529.     break;
530. }
531. }
532. /*****
533. * 函数名: searchKeyword
534. * 作用: 查找关键词表是否存在该表项
535. * 参数: id: 待匹配表项[in]
536. *****/
537. bool searchKeyword(string id) {
538.     for (string kw : keyword) {
539.         if (kw == id) {
540.             return true;
541.         }
542.     }
543.     return false;
544. }
545. /*****
546. * 函数名: isValidID
547. * 作用: 判定是否为字母, 数字, 下划线
548. * 参数: id: 待匹配字符[in]
549. *****/
550. inline bool isValidID(char id) {
551.     return (id >= 'a' && id <= 'z') || (id >= 'A' && id <= 'Z') || id == '_' || (id >= '0' && id <= '9');
552. }
553. /*****
554. * 函数名: handlerIdentifier
555. * 作用: 对于 buffer[cur] 开始的标识符进行处理
556. * 参数: 无
557. *****/
558. void LexAnalyser::handlerIdentifier() {

```



```

560. string secondBuffer = "";
561. secondBuffer += buffer[cur++];
562. while (isValidID(buffer[cur])) {
563.     secondBuffer += buffer[cur++];
564. }
565. if (searchKeyword(secondBuffer)) {
566.     storeLex(LEX_TYPE::KEYWORD, secondBuffer);
567. }
568. else {
569.     storeLex(LEX_TYPE::IDENTIFIER, secondBuffer);
570. }
571.
572. /*****
573. * 函数名:    handleNumber
574. * 作用:      对于 buffer[cur] 开始的数字常量进行处理
575. * 参数:      无
576. *****/
577. void LexAnalyser::handleNumber() {
578.     string secondBuffer = "";
579.     if (buffer[cur] == '0' && (buffer[cur + 1] == 'x' || buffer[cur + 1] == 'X' || buffer[cur + 1] == '0' || buffer[cur + 1] == 'o' || buffer[cur + 1] == 'b' || buffer[cur + 1] == 'B' || buffer[cur + 1] == 'b')) {
580.         switch (buffer[cur + 1])
581.         {
582.             case 'x': case 'X': // 十六进制
583.                 secondBuffer += buffer[cur++];
584.                 secondBuffer += buffer[cur++];
585.                 if ((buffer[cur] >= '0' && buffer[cur] <= '9') || (buffer[cur] >= 'A' && buffer[cur] <= 'F') || (buffer[cur] >= 'a' && buffer[cur] <= 'f')) {
586.                     while ((buffer[cur] >= '0' && buffer[cur] <= '9') || (buffer[cur] >= 'A' && buffer[cur] <= 'F') || (buffer[cur] >= 'a' && buffer[cur] <= 'f')) {
587.                         secondBuffer += buffer[cur++];
588.                     }
589.                     if (isValidID(buffer[cur])) {
590.                         raiseSyntaxError(curLine, SYNTAX_ERROR::NUM_CONST_FORMAT_ERROR);
591.                     }
592.                     else {
593.                         storeLex(LEX_TYPE::CONST, secondBuffer);
594.                     }
595.                 }
596.                 else {
597.                     raiseSyntaxError(curLine, SYNTAX_ERROR::NUM_CONST_UNCOMPELET_ERROR);
598.                 }
599.                 break;
600.             case 'o': case 'O': // 八进制
601.                 secondBuffer += buffer[cur++];
602.                 secondBuffer += buffer[cur++];
603.                 if (buffer[cur] >= '0' && buffer[cur] <= '7') {
604.                     while (buffer[cur] >= '0' && buffer[cur] <= '7') {
605.                         secondBuffer += buffer[cur++];
606.                     }
607.                     if (isValidID(buffer[cur])) {
608.                         raiseSyntaxError(curLine, SYNTAX_ERROR::NUM_CONST_FORMAT_ERROR);
609.                     }
610.                     else {
611.                         storeLex(LEX_TYPE::CONST, secondBuffer);
612.                     }
613.                 }
614.                 else {
615.                     raiseSyntaxError(curLine, SYNTAX_ERROR::NUM_CONST_UNCOMPELET_ERROR);
616.                 }
617.                 break;
618.             case 'b': case 'B':
619.                 secondBuffer += buffer[cur++];
620.                 secondBuffer += buffer[cur++];
621.                 if (buffer[cur] >= '0' && buffer[cur] <= '1') {
622.                     while (buffer[cur] >= '0' && buffer[cur] <= '1') {
623.                         secondBuffer += buffer[cur++];
624.                     }
625.                     if (isValidID(buffer[cur])) {
626.                         raiseSyntaxError(curLine, SYNTAX_ERROR::NUM_CONST_FORMAT_ERROR);
627.                     }
628.                     else {
629.                         storeLex(LEX_TYPE::CONST, secondBuffer);
630.                     }
631.                 }
632.                 else {
633.                     raiseSyntaxError(curLine, SYNTAX_ERROR::NUM_CONST_UNCOMPELET_ERROR);
634.                 }
635.                 break;
636.             default:
637.                 break;
638.         }
639.     }
640.     else {
641.         while (buffer[cur] >= '0' && buffer[cur] <= '9') { // 扫描到第一个不为数字的字符
642.             secondBuffer += buffer[cur++];
643.         }
644.         if (buffer[cur] == '.') { // 浮点
645.             secondBuffer += buffer[cur++];
646.             if (buffer[cur] >= '0' && buffer[cur] <= '9') {
647.                 while (buffer[cur] >= '0' && buffer[cur] <= '9') {
648.                     secondBuffer += buffer[cur++];
649.                 }
650.             }
651.             if (buffer[cur] == 'F') {
652.                 secondBuffer += buffer[cur++];
653.                 if (isValidID(buffer[cur])) {
654.                     raiseSyntaxError(curLine, SYNTAX_ERROR::NUM_CONST_FORMAT_ERROR);
655.                 }
656.                 else {
657.                     storeLex(LEX_TYPE::CONST, secondBuffer);
658.                 }
659.             }
660.             else if (buffer[cur] == 'E' || buffer[cur] == 'e') { // 科学计数法
661.                 secondBuffer += buffer[cur++];
662.                 if (buffer[cur] == '+' || buffer[cur] == '-') {
663.                     secondBuffer += buffer[cur++];
664.                 }
665.                 if (buffer[cur] >= '0' && buffer[cur] <= '9') {
666.                     while (buffer[cur] >= '0' && buffer[cur] <= '9') {
667.                         secondBuffer += buffer[cur++];
668.                     }
669.                     if (isValidID(buffer[cur])) {
670.                         raiseSyntaxError(curLine, SYNTAX_ERROR::NUM_CONST_FORMAT_ERROR);
671.                     }
672.                     else {
673.                         storeLex(LEX_TYPE::CONST, secondBuffer);
674.                     }
675.                 }
676.                 else {
677.                     raiseSyntaxError(curLine, SYNTAX_ERROR::NUM_CONST_UNCOMPELET_ERROR);
678.                 }
679.             }
680.         }
681.     }
682. }

```

```

663.     }
664.     if (buffer[cur] >= '0' && buffer[cur] <= '9') {
665.         while (buffer[cur] >= '0' && buffer[cur] <= '9') {
666.             secondBuffer += buffer[cur++];
667.         }
668.         if (isValidID(buffer[cur]) || buffer[cur] == '.') {
669.             raiseSyntaxError(curLine, SYNTAX_ERROR::NUM_CONST_FORMAT_ERROR);
670.         }
671.         else {
672.             storeLex(LEX_TYPE::CONST, secondBuffer);
673.         }
674.     }
675.     else {
676.         if (isValidID(buffer[cur])) {
677.             raiseSyntaxError(curLine, SYNTAX_ERROR::NUM_CONST_FORMAT_ERROR);
678.         }
679.         else {
680.             raiseSyntaxError(curLine, SYNTAX_ERROR::NUM_CONST_UNCOMPELET_ERROR);
681.         }
682.     }
683. }
684. else if (isValidID(buffer[cur])) {
685.     raiseSyntaxError(curLine, SYNTAX_ERROR::NUM_CONST_FORMAT_ERROR);
686. }
687. else {
688.     storeLex(LEX_TYPE::CONST, secondBuffer);
689. }
690. }
691. else {
692.     if (isValidID(buffer[cur])) {
693.         raiseSyntaxError(curLine, SYNTAX_ERROR::NUM_CONST_FORMAT_ERROR);
694.     }
695.     else {
696.         raiseSyntaxError(curLine, SYNTAX_ERROR::NUM_CONST_UNCOMPELET_ERROR);
697.     }
698. }
699. }
700. else if (buffer[cur] == 'E' || buffer[cur] == 'e') { // 科学计数法
701.     secondBuffer += buffer[cur++];
702.     if (buffer[cur] == '+' || buffer[cur] == '-') {
703.         secondBuffer += buffer[cur++];
704.     }
705.     if (buffer[cur] >= '0' && buffer[cur] <= '9') {
706.         while (buffer[cur] >= '0' && buffer[cur] <= '9') {
707.             secondBuffer += buffer[cur++];
708.         }
709.         if (isValidID(buffer[cur]) || buffer[cur] == '.') {
710.             raiseSyntaxError(curLine, SYNTAX_ERROR::NUM_CONST_FORMAT_ERROR);
711.         }
712.         else {
713.             storeLex(LEX_TYPE::CONST, secondBuffer);
714.         }
715.     }
716.     else {
717.         if (isValidID(buffer[cur])) {
718.             raiseSyntaxError(curLine, SYNTAX_ERROR::NUM_CONST_FORMAT_ERROR);
719.         }
720.         else {
721.             raiseSyntaxError(curLine, SYNTAX_ERROR::NUM_CONST_UNCOMPELET_ERROR);
722.         }
723.     }
724. }
725. else if (buffer[cur] == 'L') { // 为浮点数
726.     secondBuffer += buffer[cur++];
727.     if (isValidID(buffer[cur])) {
728.         raiseSyntaxError(curLine, SYNTAX_ERROR::NUM_CONST_FORMAT_ERROR);
729.     }
730.     else {
731.         storeLex(LEX_TYPE::CONST, secondBuffer);
732.     }
733. }
734. else if (isValidID(buffer[cur])) {
735.     raiseSyntaxError(curLine, SYNTAX_ERROR::NUM_CONST_FORMAT_ERROR);
736. }
737. else { // 判定为十进制整数
738.     storeLex(LEX_TYPE::CONST, secondBuffer);
739. }
740. }
741. }
742.
743. /*****
744. * 函数名:  handleCharConst
745. * 作用:    对于 buffer[cur] 开始的字符常量进行处理
746. * 参数:    无
747. *****/
748. void LexAnalyser::handleCharConst() {
749.     string secondBuffer = "";
750.     if (buffer[++cur] == '\\') { // 含有转义字符
751.         if (buffer[cur + 2] == '\\') {
752.             secondBuffer += buffer[++cur];
753.             storeLex(LEX_TYPE::CONST, secondBuffer);
754.             cur += 2;
755.         }
756.         else {
757.             raiseSyntaxError(curLine, SYNTAX_ERROR::CHAR_SINGLE_QUOTE_UNMATCH_ERROR);
758.         }
759.     }
760.     else { // 不含有转义字符
761.         if (buffer[cur + 1] == '\\') { //
762.             secondBuffer += buffer[cur++];
763.             storeLex(LEX_TYPE::CONST, secondBuffer);
764.             cur++;
765.         }
766.         else {
767.             raiseSyntaxError(curLine, SYNTAX_ERROR::CHAR_SINGLE_QUOTE_UNMATCH_ERROR);
768.         }
769.     }

```

```

770. }
771.
772. /*****
773. * 函数名:   handleStringConst
774. * 作用:     对于 buffer[cur]开始的字符串常量进行处理
775. * 参数:     无
776. *****/
777. void LexAnalyser::handleStringConst() {
778.     string secondBuffer = "";
779.     cur++;
780.     while (buffer[cur] != '\0') {
781.         if (buffer[cur] == '\0') { // 一行结束, C/C++不支持字符串换行
782.             raiseSyntaxError(curLine, SYNTAX_ERROR::STRING_UNCLOSE_ERROR);
783.         }
784.         else {
785.             if (buffer[cur] == '\\') {
786.                 secondBuffer += buffer[cur++];
787.             }
788.             secondBuffer += buffer[cur++];
789.         }
790.     }
791.     cur++;
792.     storeLex(LEX_TYPE::CONST, secondBuffer);
793. }
794.
795. /*****
796. * 函数名:   isTemplateClass
797. * 作用:     查找泛型表是否存在该表项
798. * 参数:     name: 待匹配表项[in]
799. *****/
800. bool isTemplateClass(string name) {
801.     for (string kw : existTemplate) {
802.         if (name==kw) {
803.             return true;
804.         }
805.     }
806.     return false;
807. }
808.
809. /*****
810. * 函数名:   handleLeftRightAngleBra
811. * 作用:     对于 buffer[cur]为<>的情况, 区分泛型和运算符
812. * 参数:     无
813. *****/
814. void LexAnalyser::handleLeftRightAngleBra() {
815.     if (isTemplateClass(tokenRes[tokenRes.size() - 1].second.second) || searchKeyword(tokenRes[tokenRes.size() - 1].second.second)) { // 泛
816.         char bra = buffer[cur]; // 存储是否为<还是>
817.         string secondBuffer = "";
818.         secondBuffer += buffer[cur++];
819.         while (bra==buffer[cur]) {
820.             secondBuffer += buffer[cur++];
821.         }
822.         if (isOp(buffer[cur])) {
823.             raiseSyntaxError(curLine, SYNTAX_ERROR::OP_FORMAT_ERROR);
824.         }
825.         else {
826.             for (unsigned i = 0; i < secondBuffer.size(); i++) {
827.                 storeLex(LEX_TYPE::SEPERATOR, string(1, secondBuffer[i]));
828.             }
829.         }
830.     }
831.     else {
832.         handleOperator();
833.     }
834. }
835.
836. /*****
837. * 函数名:   analyseToken
838. * 作用:     启动分块逻辑, 对于 buffer[cur]开始的词素进行分析
839. * 参数:     无
840. *****/
841. void LexAnalyser::analyseToken() {
842.     if (buffer[cur] == '\0' || buffer[cur] == '\n') { // 表示程序已经分析到末尾
843.         return;
844.     }
845.     char tmp = buffer[cur];
846.     switch (tmp)
847.     {
848.     case '/':
849.         if (buffer[cur + 1] == '*') { // 第一类注释
850.             if (!searchForNoteEndType_1()) {
851.                 raiseSyntaxError(curLine, SYNTAX_ERROR::MULLINE_NOTE_NOT_END_ERROR);
852.             }
853.         }
854.         if (buffer[cur + 1] == '/') {
855.             if (!searchForNoteEndType_2()) {
856.                 raiseSyntaxError(curLine, SYNTAX_ERROR::UNVALID_CHAR_ERROR);
857.             }
858.         }
859.         else {
860.             handleOperator();
861.         }
862.         break;
863.     case '#':
864.         extractMacro(); // 词法分析结果, 将宏所在改行完成提取出来
865.         break;
866.     case '{':case'}':case '[':case']':case '(':case')':case ':':case';':
867.         handleSeperator();
868.         break;
869.     case '<':case'>':
870.         handleLeftRightAngleBra();
871.         break;
872.     case '=':case'?':case'*':case'+':case'-':case'&':case'|':case'%':case'.':case'!':case'~':case'^':
873.         handleOperator();
874.         break;

```

```

875.     case '_': case 'a': case 'b': case 'c': case 'd': case 'e': case 'f': case 'g': case 'h': case 'i':
876.     case 'j': case 'k': case 'l': case 'm': case 'n': case 'o': case 'p': case 'q': case 'r': case 's':
877.     case 't': case 'u': case 'v': case 'w': case 'x': case 'y': case 'z': case 'A': case 'B': case 'C':
878.     case 'D': case 'E': case 'F': case 'G': case 'H': case 'I': case 'J': case 'K': case 'L': case 'M':
879.     case 'N': case 'O': case 'P': case 'Q': case 'R': case 'S': case 'T': case 'U': case 'V': case 'W':
880.     case 'X': case 'Y': case 'Z':
881.         handlerIdentifier();
882.         break;
883.     case '1': case '2': case '3': case '4': case '5': case '6': case '7': case '8': case '9': case '0':
884.         handleNumber();
885.         break;
886.     case '\\':
887.         handleCharConst();
888.         break;
889.     case '\\\"':
890.         handleStringConst();
891.         break;
892.     default:
893.         raiseSyntaxError(curLine, SYNTAX_ERROR::UNVALID_CHAR_ERROR);
894.         break;
895.     }
896. }
897.
898. /*****
899. * 函数名:    lex
900. * 作用:      从第一个字符开始, 启动 DFA 词法分析周期, 制导文件结束
901. * 参数:      无
902. *****/
903. void LexAnalyser::lex() {
904.     while (!(*fin).eof() || buffer[cur] != '\\0') {
905.         shiftToNextToken();
906.         analyseToken();
907.     }
908. }
909.
910. /*****
911. * 函数名:    printLexRes
912. * 作用:      打印词法分析结果
913. * 参数:      无
914. *****/
915. void LexAnalyser::printLexRes() {
916.     cout << "宏定义: " << endl;
917.     for (string it : macroBin) {
918.         cout << it << endl;
919.     }
920.     cout << "词法分析词素: " << endl;
921.     int c = 1;
922.     for (pair<int, pair<LEX_TYPE, string>> it : tokenRes) {
923.         LEX_TYPE type = it.second.first;
924.         string typeStr = (type == LEX_TYPE::SEPERATOR ? "SEPERATOR " : ((type == LEX_TYPE::IDENTIFIER ? "IDENTIFIER " : ((type == LEX_TYPE::
:OPERATOR ? "OPERATOR " : (type == LEX_TYPE::CONST ? "CONST " : "KEYWORD ")));
925.         cout << "Line:" << it.first << " , (" << typeStr << " , " << it.second.second << ")\\n";
926.     }
927.     cout << "每行开始空格数:" << endl;
928.     for (pair<int, int> p : spaceRecord) {
929.         cout << "Line:" << p.first << " , has " << p.second << " Space" << endl;
930.     }
931. }
932.
933. /*****
934. * 函数名:    getLexRes
935. * 作用:      将词法分析结果输出给外界
936. * 参数:      Res 输出句柄[in/out]
937. *****/
938. void LexAnalyser::getLexRes(std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>>& Res)
939. {
940.     Res = tokenRes;
941. }
942.
943. /*****
944. * 函数名:    raiseSyntaxError
945. * 作用:      进行语法错误的报错处理
946. * 参数:      line 行号[in]
947. *           errorId 错误类型[in]
948. *****/
949. void raiseSyntaxError(int line, SYNTAX_ERROR errorId) {
950.     switch (errorId)
951.     {
952.     case SYNTAX_ERROR::NUM_CONST_FORMAT_ERROR:
953.         cout << "Line:" << line << " , 出现非法数字常量格式错误" << endl;
954.         abort();
955.         break;
956.     case SYNTAX_ERROR::UNVALID_CHAR_ERROR:
957.         cout << "Line:" << line << " , 出现非法字符错误" << endl;
958.         abort();
959.         break;
960.     case SYNTAX_ERROR::STRING_UNCLOSE_ERROR:
961.         cout << "Line:" << line << " , 出现字符串常量格式错误" << endl;
962.         abort();
963.         break;
964.     case SYNTAX_ERROR::CHAR_SINGLE_QUOTE_UNMATCH_ERROR:
965.         cout << "Line:" << line << " , 出现非法字符错误" << endl;
966.         abort();
967.         break;
968.     case SYNTAX_ERROR::OP_FORMAT_ERROR:
969.         cout << "Line:" << line << " 出现非法运算符错误( \"w \" )" << endl;
970.         abort();
971.         break;
972.     case SYNTAX_ERROR::MULLINE_NOTE_NOT_END_ERROR:
973.         cout << "Line:" << line << " , 发生注释无结尾错误 TnT" << endl;
974.         abort();
975.         break;
976.     default:
977.         break;
978.     }

```

```
979. }
```

## 5.4 Base.h

```
1.  /*****
2.  * 模块名:      Base
3.  * 文件名:      Base.h
4.  * 依赖文件:    无
5.  * 作用:        编译器中一些基本的数据结构
6.  * 作者:        陈晓飞
7.  * 时间:        2020.5.25
8.  * 版本:        Version 1.1
9.  *****/
10. #ifndef _BASE_H_
11. #define _BASE_H_
12. #include<vector>
13. #include<iostream>
14. #define TAB_SPACE 4 // 一个 tab 键所代表的缩进个数
15. // 词法分析的目标结果类型
16. enum class LEX_TYPE {
17.     IDENTIFIER = 0,
18.     OPERATOR = 1,
19.     CONST = 2,
20.     SEPERATOR = 3,
21.     KEYWORD = 4
22. };
23.
24. enum class SYNTAX_ERROR {
25.     NUM_CONST_FORMAT_ERROR=0,
26.     UNVALID_CHAR_ERROR=1,
27.     STRING_UNCLOSE_ERROR=2,
28.     CHAR_SINGLE_QUOTE_UNMATCH_ERROR=3,
29.     OP_FORMAT_ERROR=4,
30.     MULLINE_NOTE_NOT_END_ERROR=5,
31.     NUM_CONST_UNCOMPELET_ERROR = 6
32. };
33.
34. // 关键词表
35. static std::vector<std::string> keyword = { "include","define","auto","bool","break","case","catch","char","class",
36.     "const","const_cast","continue","default","delete","do","double",
37.     "dynamic_cast","else","enum","explicit","extern","false","float","for",
38.     "friend","goto","if","inline","int","long","mutable","namespace","new",
39.     "operator","private","protected","public","register","reinterpret_cast",
40.     "return","short","signed","sizeof","static","static_cast","struct",
41.     "switch","template","this","throw","true","try","typedef","typeid",
42.     "typename","union","unsigned","using","virtual","void","volatile","while"};
43. static std::vector<std::string> existTemplate = { "vector","pair","stack","string" }; // 泛型类名表
44. static std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> tokenRes; // 存放词法分析的结果
45. static std::vector<std::string> macroBin; // 存放宏
46. static std::vector<std::pair<int, int>> spaceRecord; // 记录每一行的缩进
47. #endif
```

## 5.5 main.cpp

```
1.  #include"LexAnalyser.h"
2.  #include <iostream>
3.  #include <fstream>
4.  #include <istream>
5.  int main()
6.  {
7.      std::vector<std::pair<int, std::pair<LEX_TYPE, std::string>>> Res;
8.      std::ifstream fin = openFile(inputDir);
9.      LexAnalyser analyser(fin);
10.     analyser.lex();
11.     fin.close();
12.     analyser.printLexRes();
13.     analyser.getLexRes(Res);
14.     std::cout << "词法分析器一共解析出" << Res.size() << "个 Token" << std::endl;
15.     return 0;
16. }
```