词法分析程序的设计与实现

2020211435 何家豪

October 5, 2022

1 任务描述

设计并实现 C 语言的词法分析程序，要求实现如下功能：

- 可以识别出用 C 语言编写的源程序中的每个单词符号，并以记号的形式输出每个单词符号。
- 可以识别并跳过源程序中的注释。
- 可以统计源程序中的语句行数、各类单词的个数、以及字符总数，并输出统计结果。
- 检查源程序中出现的词法错误，并报告错误所在位置。
- 对源程序中出现的错误进行适当的恢复，使词法分析可以继续进行，对源程序进行一次扫描，即可检查并报告源程序中出现的所有词法错误。

实验要求 分别采用以下两种方案实现：

1. 采用 C/C++ 作为实现语言，手工编写词法分析程序。
2. 编写 LEX 源程序，利用 LEX 编译程序自动生成词法分析程序。

2 实验环境

- MacBook Pro(13inch, M1, 2020)
- macOS Monterey 12.6
- g++ Apple clang version 14.0.0
- flex 2.6.4 Apple(flex-34)

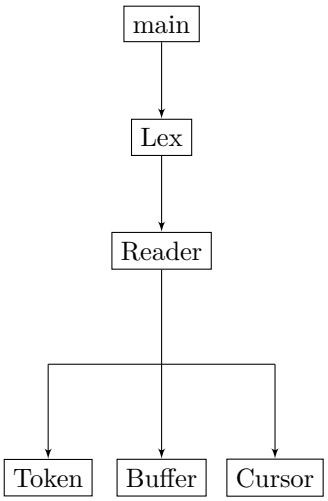
3 手工编写的词法分析程序

3.1 模块划分与功能介绍

3.1.1 模块及模块之间的关系

如图 1 所示，其中 main() 模块是程序的入口，负责初始化一个 Lex 对象，并用它来进行与词法分析相关的工作。Lex 模块用来保存分析出的记号和打印输出结果；Reader 模块负责从 Buffer 中读入字符流并把它们转化为记号；Token 模块定义了一种数据结构，它表示一个记号；Buffer 模块实现了配对缓冲区，将文件内容读入；Cursor 模块定义了一种指针类型，负责记录当前读取到的位置信息，统计行数列数等信息。

图 1: 模块及模块之间的关系



3.1.2 Token 模块

枚举类型 TokenType 在 Token 模块首先定义了一个枚举类型 TokenType，用于描述不同记号的类型，如下所示。

```
enum TokenType
{
    Identifier,  // 标志符
    ConstInt,    // 整数常量
    ConstFloat,  // 浮点常量
    ConstChar,   // 字符常量
    ConstString, // 字符串常量
    KeyWord,     // 关键字
    Operator,    // 运算符
```

```
    Seperator,    // 分隔符
    Error         // 错误标记
};
```

文字常量数组 tokenTypeStr 接下来定义了一个全局变量用于记录记号类型的枚举值与其字面值的对应关系，方便打印输出时使用。

```
const char * tokenTypeStr[] = {
    "Identifier",
    "Int",
    "Float",
    "Char",
    "String",
    "KeyWord",
    "Operator",
    "Seperator",
    "Error"
};
```

Token 类 Token 类中包含两个私有成员变量 _type 和 _attr，分别用于代表一个 token 的类型和属性。Token 类的构造函数便是根据输入参数初始化这两个值。Token 类还包含了两个值的 get 和 set 函数。除此之外，为了方便打印输出，Token 类还用友元的方式重载了 ostream 与 Token 的左移运算符。

Token 类的具体声明如下所示。

```
class Token
{
public:
    Token(TokenType type, string attr);
    Token();
    TokenType type();
    void setType(TokenType);
    string attr();
    void setAttr(string);
    friend ostream &operator<<(ostream &,
        const Token &);
private:
    TokenType _type;
    string _attr;
};
```

3.1.3 Buffer 模块

Buffer 类 Buffer 类定义了一个配对缓冲区。一个 Buffer 对象包含着一对缓冲区，leftBuffer 和 rightBuffer。同时提供了两个函数 fillLeftBuffer() 和 fillRightBuffer() 用于填充这两个缓冲区。值得注意的是，如果在填充缓冲区时，并没有把缓冲区填满，说明已经读到了输入文件流的尽头，此时两个填充函数会在末尾处补上一个文件结束符（EOF）。

在 Buffer 的构造函数中，它要求构造者提供一个文件路径来指定要读取的文件，在构造函数中将该文件打开后保存为自身的文件流 sourceFile；同时指定一个正整数作为单侧缓冲区的大小（如指定 1024 字节，则 leftBuffer 和 rightBuffer 的大小都为 1024 字节）。

同时，为了方便访问 Buffer 内的数据，Buffer 类还重载了索引运算符（“[]”）。在索引中的数字是读取的逻辑位置，并不是在当前 buffer 中的实际位置。所以在读取数据时，会先对这个数字进行取模和整除运算，计算出它对应的实际位置，再去计算得出的那一半 buffer 中寻找对应位置的值进行返回。

在 Buffer 类的析构函数中，它会关闭打开的输入文件流并释放两块 buffer 的空间。

Buffer 类的具体声明如下所示。

```
class Buffer
{
public:
    Buffer(const char *, unsigned int);
    unsigned int fillLeftBuffer();
    unsigned int fillRightBuffer();
    char operator[](int);
    ~Buffer();
private:
    std::ifstream sourceFile;
    unsigned int _bufferSize;
    char *leftBuffer;
    char *rightBuffer;
};
```

3.1.4 Cursor 模块

Cursor 类 Cursor 类定义了一个用于记录位置信息的数据结构。其中包括一个位置的逻辑索引 loc，行数 line，列数 row。

如果不加参数的构造 Cursor 类的对象，将会默认把位置记录在 loc=0，line=1，row=1 的位置。

Cursor 类还定义了一个换行的函数，在每次读到'\n'时调用，可以用于处理换行事件。

与 Token 类类似，Cursor 类也以友元的方式重载了 ostream 与 Token 的左移运算符方便输出。

最后，为了方便直接将 Cursor 类的对象作为 Buffer 的索引使用，Cursor 类还定义了一个类型转换函数，它会直接返回 loc 的值，用于直接当作索引时使用。

Cursor 类的具体声明如下所示。

```
class Cursor
{
public:
    operator unsigned int();
    Cursor();
    Cursor(unsigned int, unsigned int,
        unsigned int);
    void nextLine();
    friend ostream& operator<<(ostream&,
        const Cursor&);
    unsigned int loc, line, row;
};
```

3.1.5 Reader 模块

Reader 模块负责主要的词法分析功能。Reader 模块的私有成员中有一个 Buffer 对象用于输入字符流；有两个 Cursor 对象，一个是前驱指针 begin 用于标记当前读取记号的起始位置，一个是后继指针 end 用于标记当前读取记号的结束位置。

Reader 模块的核心函数有两个，forward() 和 getToken()。

forward() forward() 函数的声明如下所示。

```
char Reader::forward(int check, unsigned int
        step);
```

这个函数的功能是将指针前移。check 参数用于指定迁移的是哪一个指针（0 为前驱指针，1 为后继指针）。之所以要区分的原因是因为：如果迁移的是后继指针，当后继指针前移到缓冲区的边界时，就需要填充另一半缓冲区；而在前移前驱指针时却不用这么做。step 参数用于指定本次要前移多少个逻辑位置。除了有可能会检查是否需要填充另一半缓冲区外，forward() 函数还会在读到换行符的时候处理指针的换行事件。这个函数的返回值是前移指针之后，指针所指向的字符。

getToken() getToken() 函数的声明如下所示。

```
int Reader::getToken(Token& nextToken,
        Cursor& curLoction);
```

这个函数的功能是从 buffer 中读取下一个记号出来。参数中的 nextToken 即为读出来的下一个记号，curLocation 为这个记号出现的位置。getToken() 函数成功的读取了一个记号（不合法的记号也是记号）时返回 1，读取到文件结束时返回 0。

在不考虑注释等特殊情况下，其主要的工作逻辑可以分为以下几步：

1. 读取前驱指针处的字符，根据前驱指针处的字符决定这可能是一个什么样的记号。
2. 依据上一步的判断决定该选择哪个分支去处理这个字符。
3. 进入处理分支后，后继指针不断 forward()，直到程序认为从前驱指针到后继指针中间这一段内容可以作为一个记号时停止。
4. 返回这个记号和它的位置，并将前驱指针前移至下一个有效字符的位置，将后继指针前移至前驱指针的下一个位置。

下面对第 1，2 步的各种情况做详细说明。

处理运算符和分隔符

大部分的运算符和分隔符的处理逻辑类似，在此举一复杂例子说明。若前驱指针所在位置的字符为'>'，那么这个位置出现的有可能是一个大于号（">"），有可能出现的是一个大于等于号（">="），有可能出现的是一个右移运算符（'> >'），有可能出现的是一个右移赋值运算符（">>="）。在这种情况下函数中会让后驱指针向后扫描至多 2 个位置来确定这是一个什么样的符号。如下所示。

```
else if (buffer[begin] == '>') {
    if (buffer[end] == '>') {
        forward(0, 1);
        forward(1, 1);
        if (buffer[end] == '=') {
            forward(0, 2);
            forward(1, 1);
            forward(1, 1);
            nextToken = Token(Operator, "
        Right Shift Assign");
        } else {
            forward(0, 1);
            forward(1, 1);
```

```
            nextToken = Token(Operator, "Right Shift");
        }
    } else if (buffer[end] == '=') {
        forward(0, 2);
        forward(1, 2);
        nextToken = Token(Operator, "Greater Equal");
    } else {
        forward(0, 1);
        forward(1, 1);
        nextToken = Token(Operator, "Greater");
    }
}
```

处理标识符和关键字

当前驱指针指向的是下划线（'_'）或者一个字母时，函数会将它作为一个标识符或者关键字处理。此时将后继指针不断前移，直到遇到一个不是数字，不是字母，不是下划线的位置停止。然后判断从前驱指针到后继指针中间区域的字符是否构成了 32 个 C 语言关键字中的一个，若是，则返回关键字类型；若不是，则返回标识符类型。逻辑比较简单，在此不再赘述代码细节。

处理数字

处理数字是实现该函数的难点。首先，先要分析一下 C 语言中各种合法数字的形式。

- **十进制整数** 最常见的整数类型，以一个非零的数字开头，后面跟着若干个任意数字。如：54。
- **十进制浮点数** 在若干个数字中间出现小数点或者科学计数法（e 或 E）的数字。如：1.1e-4，000.1E+6，00.44。
- **十六进制整数** 由 0x 或 0X 开头，后跟若干个十六进制数字的数。如：0xFF 或 0Xee
- **十六进制制浮点数** 由 0x 开头，中间出现十六进制科学计数法符号 p 或 P 的数字。如：0x5e.5e5eP+5
- **二进制整数** 由 0b 或 0B 开头，后面出现若干个 01 组成的数字。如：0b11，0B001。
- **八进制整数数** 由 0 开头，后跟若干个八进制数字的数。如：07，011。

除此之外，整数和浮点数还有若干标注它们类型的后缀。

对于整数而言 可以使用 u 或 U 来表示这个整数是无符号（unsigned）的，用 l 或 L 来表示这个整数是长（long）的，用 ll 或 LL 来表示这个整数是长长（long long）的。所以下述后缀都是合法的整数后缀：u，l，ul，uL，Ul，UL，lu，lU，Lu，LU，LL，ll，ull，uLL，Ull，ULL，llu，llU，LLu，LLU。

对于浮点数而言 可以使用 f 或 F 来强调这是一个浮点数，用 l 或 L 来表示这个浮点数是双精度（double）的。所以下述后缀都是合法的整数后缀：f，F，l，L。

综上，在考虑到前缀，数字内容本身，以及后缀的各种复杂情况下，函数中采用了正则表达式的方式来判断前驱指针和后继指针中间的内容是否是一个合法的数字。具体流程为：当前驱指针指向的是一个数字时，函数会将它作为一个数字处理。此时将后继指针不断前移，直到遇到一个不是数字，不是字母，不是下划线，不是小数点，也不是加号或减号的位置停止。然后使用正则表达式判断从前驱指针到后继指针中间区域的字符是否构成了上述 6 种合法数字中的一种。若是，返回对应的数字类型；若不是，返回错误类型。

处理字符和字符串

前驱指针指向的是一个单引号（"'"）时，函数会将它作为一个字符处理。此时将后继指针不断后移，直到遇到另一个单引号时停止。然后判断两个单引号中间是否只有一个字符，若有且仅有一个字符，返回字符常量类型；若不是这样，返回错误类型。

前驱指针指向的是一个双引号（'""'）时，函数会将它作为一个字符处理。此时将后继指针不断后移，直到遇到另一个双引号时停止。将两个指针中间区域作为一个字符串处理，返回字符串类型。

3.1.6 Lex 模块

Lex 模块主要负责存储，统计，输出词法分析得到的结果。

TableItem 结构体 这个结构体用于描述得到的记号和记号的位置的对应关系。如下所示。

```
typedef struct TableItem
{
    Token token;
    Cursor loc;
} TableItem;
```

Lex 类 Lex 类内维护着一个记号表 _tokenTable，用于记录目前已经得到的记号。用 tokenNum 统计目前各类记号的个数。用 lines 和 charNum 统计文件的总行数和字符数。同时 Lex 类内还维护着一个 Reader 类的对象用于读记号。

Lex 类的构造函数会接收一个文件路径作为参数，代表要进行词法分析的文件。除此之外 Lex 只有一个 analysis() 函数用于分析记号和 printRes() 函数用于输出结果。逻辑上并无特别之处，不做赘述。

Lex 类的具体声明如下。

```
class Lex
{
public:
    void analysis();
    Lex(const char *);
    void printRes();
private:
    vector<TableItem> _tokenTable;
    unsigned int tokenNum[9];
    Reader _reader;
    unsigned int lines;
    unsigned int charNum;
};
```

3.2 测试程序

3.2.1 编译程序

使用命令行工具 cd 到 LexicalAnalysysis 目录下，执行以下命令：

```
g++ code/*.cpp -o Lex
```

执行成功后，LexicalAnalysis 目录下会生成一个 Lex 可执行文件。

3.2.2 运行测试程序

Lex 程序在执行时的指令格式为：

```
./Lex [filePath]
```

其中 filePath 是一个需要进行词法分析的文件。如：要对 test/test1.c 文件进行词法分析，要执行的指令为：

```
./Lex test/test1.c
```

执行成功后，程序会在标准输出（stdout）输出词法分析的结果。

3.2.3 测试用例

在 test 文件夹下，有 4 个 C 语言源程序文件，它们是为了特定目的编写好的测试用例。res1.txt 到 res4.txt 分别是对这四个 C 语言源程序文件的词法分析结果。其中：test1.c 是一个普通的 C 语言程序；test2.c 是一个拥有一些特殊数字和字符串的 C 语言程序；test3.c 是一个拥有大量特殊的数字，非法数字，非法字符常量的 C 语言程序；test4.c 是一个有大量语法错误的复杂的 C 语言程序，但是值得注意的是，它并没有任何词法错误。

4 利用 LEX 编译程序自动生成词法分析程序

4.1 模块划分与功能介绍

4.1.1 模块划分

一个支持 flex 编译的 lex 源程序的一般划分为以下几个模块。

- **C 语言声明**：用于包含所需的头文件和定义全局变量等，会直接复制进 lex.yy.c 文件。
- **定义正则表达式**：方便后续使用
- **规则段**：每一条规则是一个二元组，二元组的第一项是一个正则表达式，二元组的第二项是若干 C 语言的语句，类似于函数，用于在匹配到对应的正则表达式后执行。
- **自定义过程**：用户自己定义需要用到的函数等，会直接复制到 lex.yy.x 的文件尾部。

在一个 lex 源程序中，以上部分一般以下述格式排列：

```
%{
    // C语言声明
%}
    // 定义正则表达式
%%
    // 规则段
%%
    // 用户自定义过程
```

表 1: 不同记号类型及他们对应的正则表达式（部分）

记号类型	正则表达式
标识符	(_|{letter})(_|{letter}|{digit})*
八进制整数	0[0-7]*{intSuffix}?
二进制整数	0(b|B)(0|1)+{intSuffix}?
十进制整数	[1-9]{digit}*{intSuffix}?
十进制浮点数	{digit}*(({digit}*)|((e|E)[-,+]?{digit}*)|({digit}*)((e|E)[-,+]?{digit}*))(f|F|l|L)?
十六进制整数	0(x|X){xDigit}+{intSuffix}?
十六进制浮点数	0(x|X){xDigit}+({xDigit}+)?(p|P)[-,+]?{digit}+(f|F|l|L)?
字符常量（含错误情况）	'[^']*'
字符串常量（含错误情况）	"(.| |)*"

注：
1 分隔符，运算符和关键字的定义就是它们本身；
2 字符常量和字符串常量包含了错误情况，是因为需要在对应的处理函数里分析错误；
3 *letter*，*digit*，*xDigit*，*intSuffix* 分别代表合法的单个字母，数字，十六进制数字和整数后缀。

4.1.2 C 语言声明模块

与 C++ 实现版本类似，在 C 语言声明模块，首先定义了一个枚举类型 TokenType 用于描述记号的类型。然后定义了与之对应的 tokenTypeStr 数组用于描述记号类型的字面值。

除此之外，line 用来记录当前读取到的行数；tokenNum 数组用来记录各类不同记号的个数；charNum 用来记录源文件的总字符数；col 用来记录当前读取到的列数。

```
%{
enum TokenType
{
    Identifier,  // 标志符
    ConstInt,    // 整数常量
    ConstFloat,  // 浮点常量
    ConstChar,   // 字符常量
    ConstString, // 字符串常量
    KeyWord,     // 关键字
    Operator,    // 运算符
    Seperator,   // 分隔符
    Error        // 错误标记
};
const char * tokenTypeStr[9] = {
    "Identifier",
    "Int",
    "Float",
    "Char",
    "String",
    "KeyWord",
    "Operator",
    "Seperator",
    "Error"
};
unsigned int line = 1;
unsigned int tokenNum[9]={0};
unsigned int charNum = 0;
unsigned int col = 1;
%}
```

4.1.3 定义正则表达式模块

定义正则表达式模块主要对应的是 C++ 实现版本中的 Reader 模块中的分支功能，其中包含着每个类型的记号的正则表达式。因全部定义过于冗长，故选其中重要的几例如表 1 所示。

4.1.4 规则段模块

每当识别出一个记号，便调用其对应的处理函数。其处理函数一般包括以下几个步骤：

1. 检查有无异常；
2. 输出记号及其位置，如有异常，输出异常记号和位置；
3. 更新统计信息：行数，列数，字符总数，记号数。

记号（包括各类记号和异常记号）一般采用如上流程处理。遇到注释需要采用特殊的处理函数处理。

处理注释 在处理注释时采用了类似 C++ 实现版本中的逻辑：

- 若匹配到"//"，则不断调用 input() 函数获取下一个字符，直到遇到换行符。期间注意更新统计信息和处理换行事件。
- 若匹配到"/*"，则不断调用 input() 函数获取下一个

字符，直到遇到"*/。期间注意更新统计信息和处理换行事件。

4.1.5 自定义过程模块

在该模块一共定义了两个函数：main() 函数和 yywrap() 函数。

main() main() 函数主要负责从命令行获取文件路径参数，然后调用 yylex() 进行词法分析，最后计算统计信息并输出。

yywrap() yywrap() 函数主要负责同时处理多个文件时使用，目前本程序只支持同时处理单个文件，所以该函数永远返回 1。

4.2 测试程序

4.2.1 编译程序

使用命令行工具 cd 到 LexicalAnalysysis 目录下，先执行以下命令将 LEX 源文件编译为 C 语言源程序文件：

```
flex -o clex.c flex/clex.l
```

执行成功后，LexicalAnalysis 目录下会生成一个 clex.c 文件。然后编译这个 C 语言源程序文件：

```
gcc clex.c -o CLex
```

执行成功后，LexicalAnalysis 目录下会生成一个 CLex 可执行文件。

4.2.2 运行测试程序

CLex 程序在执行时的指令格式为：

```
./CLex [filePath]
```

其中 filePath 是一个需要进行词法分析的文件。如：要对 test/test1.c 文件进行词法分析，要执行的指令为：

```
./CLex test/test1.c
```

执行成功后，程序会在标准输出（stdout）输出词法分析的结果。

4.2.3 测试用例

为了保证 Lex 版本和 C++ 版本的等价性，Lex 实现版本使用的测试用例与 C++ 实现版本所使用的测试用例一致，依然为 test 文件夹下的 4 个 C 语言源程序文件。lex1.txt 到 lex4.txt 分别是对这四个 C 语言源程序文件的词法分析结果。它们与 C++ 实现版本的区别在于，C++ 版本对于某些 Token 的属性有重命名，比如将 “(” 命名为 “Left Parenthesis”，而在 Lex 版本中，则会直接输出 “(”。除此之外对于非法数字的处理略有区别，但是不会影响合法数字的识别。

5 实验总结

本次实验我使用了 C++ 和 flex 编译程序两种方式实现了 C 语言的词法分析程序。实现的每个细节里都倾注了我的思考。在设计 C++ 版本的总体架构时，我考虑了是否要全局采用自动机的方式实现，后来注意到数字部分的复杂性，故采用了 “总体自动机，局部模式匹配” 的方案，最后在保证了代码不至于太过于繁杂的前提下实现了较全面的数字记号识别。类似这样的思考还有很多，它们锻炼了我对程序的总体架构能力。虽然在之前的形式语言与自动机课程中学习过一部分关于正则表达式的知识，但是在实现 Lex 版本时，还是发现了计算机识别的正则表达式与课本上大有不同。经过一番系统的学习，最终设计出了一套可以识别 C 语言所有记号的法则。这个过程充满了成就感。

本次实验花费了我超出预计很多倍的时间，但同时也是一次完美的实践与理论相结合的经历，我受益颇丰。