北京郵電大學

Beijing University of Posts and Telecommunications

编译原理词法分析实验报告

[C++&fLex]

学院：计算机学院
2020211376 马天成
2022 年 10 月 7 日

北京邮电大学《计算机网络》课程实验报告

<table>
<tr><td>实验
名称</td><td colspan="2">词法分析</td><td>学
院</td><td>计算机</td><td>指导教师</td><td>刘辰</td></tr>
<tr><td>班 级</td><td>班内序
号</td><td colspan="2">学 号</td><td>学生姓名</td><td colspan="2">成绩</td></tr>
<tr><td>2020211305</td><td>2</td><td colspan="2">2020211376</td><td>马天成</td><td colspan="2"></td></tr>
<tr><td>实
验
内
容</td><td colspan="6"></td></tr>
<tr><td>学生
实验
报告</td><td colspan="6">(详见“实验报告和源程序”册)</td></tr>
<tr><td>课
程
设
计
成
绩
评
定</td><td colspan="6">评语:

成绩:

指导教师签名:

年 月 日</td></tr>
</table>

注：评语要体现每个学生的工作情况，可以加页。

目录

1. 实验目的和要求

1.1 实验目的

词法分析程序的设计与实现

1.2 实验要求

设计并实现 C 语言的词法分析程序，要求实现以下功能：
（1）可以识别出 C 语言编写的源程序的每个单词符号，并以记号的形式输出每个符号。
（2）可以识别并跳过源程序中的注释
（3）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
（4）检察院程序中存在的语法错误，并报告错误所在的位置
（5）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所有存在的语法错误。

用 C++和 Lex 实现

2. C++程序

2.1 设计各种词语类型及词法识别自动机

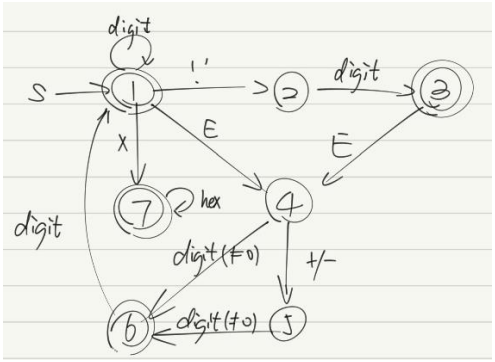
一共有 9 种数据类型，前面八种识别的时候出错自然会归为 ERROR。

```
#ifndef _GLOBAL_H
#define _GLOBAL_H


// 规定数据类型
enum dataType {
    NUMBER,     // 数字常量
    CHAR,       // 字符常量
    STRING,     // 字符串常量
    IDENTIFIER, // 标识符
    KEYWORD,    // 关键字
    PUNCTUATOR, // 运算符
    PRECOMPLIED,// 预编译
    NOTE,       // 注释
    ERROR       // 错误
};


#endif
```

语法自动机：
鉴于有些词语比较简单，所以只有数字和注释做了自动机处理。

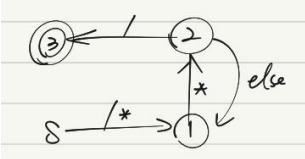
- 数字

```
/* 状态
    1：正常识别整数
    2：前面是小数点
    3：读取小数部分
    4：前面是E
    5：读取E后+/-
    6：读取E数值部分
    7：读取十六进制数值
*/
```



- 多行注释

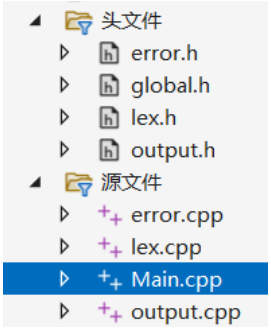
```
/* state
    1: /*
    2: /**
*/
```



2.2 代码结构及编写

2.2.1 模块设计

lex, output, error 有对应.h&.c
1. global.h：规定数据类型
2. Main.c：主函数，运行
3. lex：词法分析模块
4. output：输出正确词法模块
5. error：输出错误词法模块



2.2.2 Main 函数

```cpp
#include <iostream>
#include "lex.h"

using namespace std;

int main(int argc, char const* argv[])
{
    //生成可执行文件的时候，往里面添加
    if (argc != 4) {
        cout << "Usage: Lex_Analysis [infile_name] [outfile_name] [errorfile_name]" << endl;
        exit(1);
    }

    Lex myLex(argv[1], argv[2], argv[3]);
    myLex.parser();     // 执行语法分析
    myLex.print_info(); // 输出结果再terminal和outFile

    return 0;
}
```

Main 函数的主要功能：

```
Microsoft Windows [版本 10.0.19044.2006]
(c) Microsoft Corporation。保留所有权利。

D:\Course\Term5 编译原理\Labs\1\compile_Lex\x64\Release>compile_Lex.exe test.txt out.txt error.txt

Program Analysis Result:

Total lines :   3
Total chars :   754
Total words :   166
numbers :       20
char_consts :   1
strings :       3
identifiers :   52
keywords :      2
punctuators :   77
precompiles :   2
notes : 2
errors :        7
inFile :        test.txt
outFile :       out.txt
errorFile :     error.txt
D:\Course\Term5 编译原理\Labs\1\compile_Lex\x64\Release>
```

1. 获得输入参数，分别为[infile_name] [outfile_name] [errorfile_name]
   若格式不正确则报错 exit(1)。
2. 运行 lex 类分析相应的 inFile。
3. 用 lex 统计输出的函数输出有关的所有统计信息（同时打印在 out 和 terminal 里）。

2.2.3 Lex-识别进入各类词法函数

```
62  // 读取核心函数-------------------------------------------------------------------------
63  void Lex::parser()
64  {
65      // 循环读取词语-----------------------------------------
66      int start_row, start_col;
67      while (!inFile.eof())
68      {
69          // 词法分析外部，进行无意义符号吃掉-----------------
70          while (ch == ' ' || ch == '\t' || ch == '\f' || ch == '\n') { ... }
74          if (ch == EOF)
75              return;
76          // cout << "char " << ch << endl;
77
78          // 获取当前的起始坐标，进入分析，清空缓存区，并get字符到类变量ch中
79          // 文件是一个字符串，但强行拆成行的话，每行结尾必定是换行符
80          // 那么转换一下思路，我们把每一行（除了第一行）的0下标空间放置为上一行的换行符，就可以解决了
81          start_row = this->row_count;
82          start_col = this->col_count-1;
83
84          // ------------------------------------------------------
85          // 进入数字自动机读取
86          if (isdigit(ch)) { ... }
90
91          // 进入字符常量自动机读取
92          else if (ch == '\'') { ... }
96
97          // 进入字符串常量自动机读取
98          else if (ch == '\"') { ... }
98          else if (ch == '\"') { ... }
102
103         // 进入标识符自动机读取
104         else if (isalpha(ch) || ch == '_') {
105             this->type = dataType::IDENTIFIER;
106             parser_identifier(start_row, start_col);
107         }
108
109         // 进入运算子自动机读取
110         else if (ch == '>' || ch == '<'
111             || ch == '=' || ch == '*' || ch == '%' || ch == ':' || ch == '^' || ch == '!'
112             || ch == '+' || ch == '-' || ch == '&' || ch == '|'
113             || ch == '(' || ch == ')' || ch == '[' || ch == ']' || ch == '{' || ch == '}' || ch == '.' || ch == '?' || c
114             ) { ... }
118
119         // 进入预编译自动机读取
120         else if (ch == '#') { ... }
124
125         // '/'有关注释，特殊的
126         else if (ch == '/') { ... }
141
142         // 开头即非法字符，直接输出错误，无类型定义
143         else { ... }
152
153         // 保证到这里，读取了当前词汇并且buffer里有一个边界往后的字符，对应文件指针
154         // 统计计数
155         word_count++;
156         switch (this->type) { ... }
195     }
196 }
```

上图是读取函数核心部分。这里的策略是每次进入判断时，我们已经读取了该词法的第一个字符。然后进行判断，进入相应的此法识别程序。

2.2.4 Lex-各类词法识别函数

```
200     // 数字读取，分为十进制&八进制整数小数----------------------------------------------------------------
201   + void Lex::parser_number(int start_row, int start_col) { ... }
411
412     // 字符常量读取----------------------------------------------------------------------
413   + void Lex::parser_char(int start_row, int start_col) { ... }
443
444     // 字符串读取----------------------------------------------------------------------
445   + void Lex::parser_string(int start_row, int start_col) { ... }
492
493     // 标识符（包括关键字读取）----------------------------------------------------------------------
494   + void Lex::parser_identifier(int start_row, int start_col) { ... }
521
522     // 运算子读取----------------------------------------------------------------------
523   + void Lex::parser_punctuator(int start_row, int start_col) { ... }
578
579     // 预编译信息读取----------------------------------------------------------------------
580   + void Lex::parser_precompiled(int start_row, int start_col) { ... }
609
610     // 注释读取----------------------------------------------------------------------
611   + void Lex::parser_note(int start_row, int start_col) { ... }
```

一共有七个函数。其中标识符包括关键字的读取。

这里只提示重要的操作策略：

- 数字
  - 读取中的自动机策略（如上文所示）
  - 0 开头会有十六进制和八进制两种合法读取方式

```
220         // 读进去十六进制
221         if (ch == 'x')
222             state = 7;
223         // 00开头，报错
224       + else if (ch == '0') { ... }
234         // 读进去八进制（已必定不是0了）
235         else if (isdigit(ch));
```

- 字符常量
  - 字符需转义的符号不能直接读取，需转义

```
425     // 转义符处理
426     else if (ch == '\\') {
427         char next = inFile.peek();
428         if (next == '\n' || next == '\'' || next == '\"' || next == '\\') {
429             buffer.pop_back();
430             getChar();
431         }
432         else {
433             buffer.pop_back();
434             this->type = dataType::ERROR;
435             print_errorFile("' ended with no character", start_row, start_col);
436
437             // 清理buffer并且把当前字符加入buffer
438             buffer.clear();
439             buffer.push_back(ch);
440             return;
441         }
442     }
```

- 字符串常量
  - 字符串里 \+\n 能换行

```
472         // 转义字符，需要看后面一个是谁
473         if (ch == '\\') {
474             getChar();
475             // 输出不闭合错误
476             if (ch == EOF) {
477                 buffer.pop_back();
478                 this->type = dataType::ERROR;
479                 print_errorFile("string with no right \"", start_row, start_col);
480 
481                 // 清理buffer并且把当前字符加入buffer
482                 buffer.clear();
483                 buffer.push_back(ch);
484                 return;
485             }
486             // 换行，需要pop转义符和换行符
487             else if (ch == '\n') {
488                 buffer.pop_back();
489                 buffer.pop_back();
490                 // 吃掉 \t \n
491                 while (inFile.peek() == '\t' || inFile.peek() == '\n') {
492                     getChar();
493                     buffer.pop_back();
494                 }
495             }
496         }
```

- 标识符
  - 标识符结束进行关键字的识别

```
// 是关键字
if (keyword.find(buffer) != keyword.end()) {
    this->type = dataType::KEYWORD;
    print_outFile(start_row, start_col);
}
```

- 运算子
  - 长度为 2 或者 3 的运算符要分**前缀读取**

- 预编译信息
  - 预编译里 **\ + \n** 能换行

```
609         // 转义符编译连接
610         else if (ch == '\\') {
611             // 结束终止
612             if (getChar() == EOF) {
613                 buffer.pop_back();
614                 print_outFile(start_row, start_col);
615                 return;
616             }
617             // 能够转义换行
618             else if (ch == '\n') {
619                 buffer.pop_back();
620                 buffer.pop_back();
621             }
622         }
623     }
```

- 注释
  - 读取中的自动机策略（如上文所示）
  - 分为单行注释和多行注释

2.2.4 Lex-读取单个字符函数

```
41    // 读取单个字符
42    char Lex::getChar()
43    {
44        ch = inFile.get();
45
46        //将内容存入缓存区
47        buffer.push_back(ch);
48        character_count++;
49        col_count++;
50
51        // 换行初始化数值
52        if (ch == '\n') {
53            row_count++;
54            col_count = 1;  // 理解：上一行换行符占据了此行的0下标空间
55        }
56
57        return ch;
58    }
```

我们把换**行符理解为每一行的第 0 号元素**。这样就解决了直观上下标不对应，以及输出词法位置种换行符如何理解的问题。

2.2.5 Lex-输入输出函数

- out 和 error

```
692    // 输出到outFile
693    void Lex::print_outFile(int start_row, int start_col) {
694        outStream->add(this->type, this->buffer, start_row, start_col);
695    }
696
697    // 输出到errorFile
698    void Lex::print_errorFile(string description, int start_row, int start_col) {
699        errorStream->add(this->type, this->buffer, description, start_row, start_col);
700    }
```

调用 output 类输出词法信息到 outFile；
调用 error 类输出错误信息到 errorFile。

- summary

```
702    // 生成结果字符串添加到terminal和outFile
703    void Lex::print_info()
704    {
705        // sumary
706        string ans = "\nProgram Analysis Result:\n";
707        ans += "\nTotal lines :\t" + to_string(this->col_count);
708        ans += "\nTotal chars : \t" + to_string(character_count);
709        ans += "\nTotal words : \t" + to_string(word_count);
710        ans += "\nnumbers :\t" + to_string(number_count);
711        ans += "\nchar_consts :\t" + to_string(char_count);
712        ans += "\nstrings :\t" + to_string(string_count);
713        ans += "\nidentifiers :\t" + to_string(identifier_count);
714        ans += "\nkeywords :\t" + to_string(keyword_count);
715        ans += "\npunctuators :\t" + to_string(punctuator_count);
716        ans += "\nprecompiles :\t" + to_string(precompile_count);
717        ans += "\nnotes :\t" + to_string(note_count);
718        ans += "\nerrors :\t" + to_string(error_count);
719
720        // output
721        outStream->sumary(ans);
722        cout << ans;
723        cout << "\ninFile :\t" << inFile_name;
724        cout << "\noutFile :\t" << outFile_name;
725        cout << "\nerrorFile :\t" << errorFile_name;
726    }
```

生成统计信息输出到 terminal 和 outFile。

2.2.6 output 类 & error 类

主要为类封装，和 outFile 和 errorFile 交互

```
13  class Output
14  {
15  private:
16      ofstream fout;
17
18  public:
19      Output(string outFile_name);
20      ~Output();
21
22      void add(dataType type, string token, int row, int col);
23      void sumary(string str);
24  };
```

```
13  class Error
14  {
15  private:
16      ofstream fout;
17
18  public:
19      Error(string errorFile_name);
20      ~Error();
21
22      void add(dataType type, string token, string description, int row, int col);
23  };
```

2.3 实验结果

- 实验用例：
  test.txt
- 实验结果：
  output.txt error.txt

3. Lex 程序

3.1 Lex 介绍

Lex 是 linux 下的工具，是一个词法分析程序的自动生成工具。

Lex 的基本工作原理为：由正规式生成 NFA，将 NFA 变换成 DFA，DFA 经化简后，模拟生成词法分析器。

其中正规式由开发者使用 Lex 语言编写，其余部分由 Lex 翻译器完成.翻译器将 Lex 源程序翻译成一个名为 lex.yy.c 的 C 语言源文件，此文件含有两部分内容：一部分是根据正规式所构造的 DFA 状态转移表，另一部分是用来驱动该表的总控程序 yylex()。当主程序需要从输入字符流中识别一个记号时，只需要调用一次 yylex()就可以了。为了使用 Lex 所生成的词法分析器，我们需要将 lex.yy.c 程序用 C 编译器进行编译，并将相关支持库函数连入目标代码。Lex 的使用步骤可如下图所示：

[第一部分：定义段]
%%
[第二部分：词法规则段]
%%
[第三部分：辅助函数段]

3.2 Lex 编写结构

3.2.1 定义段

第一部分以符号%{和%}包裹，里面为以 C 语法写的一些定义和声明：例如，文件包含，宏定义，常数定义，全局变量及外部变量定义，函数声明等。这一部分被 Lex 翻译器处理后会全部拷贝到文件 lex.yy.c 中。

```
%{
    int words = 0;
    int character_cnt = 0;
    int lines = 1;
    void charCnt_add(int n);
    void line_add(void);
%}
```

3.2.2 词法规则段

*) 正规表达式规则如下：
（a）正文字符：除元字符以外的其他字符，这些字符在正规式中可以被匹配。
若单个正文字符 c 作为正规式，则可与字符 c 匹配，元字符无法被匹配，如果元字符想要被匹配，则需要通过“转义”的方式，即用” ”包括住元字符，或在元字符前加 \ 。例如 ”+” 和 \+ 都表示加号。
C 语言中的一些转义字符也可以出现在正规式中，例如 \t \n \b 等。
（b）元字符：元字符是 lex 语言中作特殊用途的一些字符，包括：* + ? | { } [ ] ( ). ^ $ “ \ - / < >。
（c）^：表示补集：[^…]表示补集，即匹配除 ^ 之后所列字符以外的任何字符。如 [^0-9]

表示匹配除数字字符 0-9 以外的任意字符。

除 ^ - \ 以外，任何元字符在方括号内失去其特殊含义。

如果要在方括号内表示负号 - ，则要将其至于方括号内的第一个字符位置或者最后一个字符位置，例如[-+0-9][+0-9-]都匹配数字及+ - 号。

（d） . ^$ /：

点运算符 . 匹配除换行之外的任何字符，一般可作为最后一条翻译规则。

（e）^ 匹配行首字符。如：^begin 匹配出现在行首的 begin

（f） $ 匹配行末字符。如：end$ 匹配出现在行末的 end

（g）R1/R2（R1 和 R2 是正规式）表示超前搜索：若要匹配 R1，则必须先看紧跟其后的超前搜索部分是否与 R2 匹配。

```
digit           [0-9]
space           [ \t]
letter          [_A-Za-z]
lineComment     (\/\/([^\n]*(\\+\n)*)*\n)
comment         (\/\*(.*\n*)*\*\/)
string          (\"([^"\n]*(\\\n)*(\\\")*)*\")
id              {letter}({letter}|{digit})*
number          {digit}+(\.{digit}+)?([eE][+-]?{digit}+)?
keyword
   "int"|"long"|"short"|"float"|"double"|"char"|"unsigned"|"si
gned"|"const"|"void"|"volatile"|"enum"|"struct"|"union"|"if"|"
else"|"goto"|"switch"|"case"|"do"|"while"|"for"|"continue"|"br
eak"|"return"|"default"|"typedef"|"auto"|"register"|"extern"|"
static"|"sizeof"
operator
   ">"|">>"|">="|">>="|"<"|"<<"|"<="|"<<="|"!"|"!="|"="|"=="|"
/"|"/="|"*"|"*="|"%"|"%="|"^"|"^="|"|"|"||"|"|="|"?"|"&"|"&&"|
"&="|"+"|"+="|"++"|"-"|"-="|"--"
delimiter       "("|")"|"{"|"}"|";"
char            (\'[^']*\')
illString       (\"([^"\n]*(\\\")*)*\n)
illNum          ({digit}+(\.{digit}+)?([eE][+-]?[^0-
9]))|({digit}+\.[^0-9])
macro           #.*\n
```

*) 正规表达式的书写结束之后，就是对相应的规则进行描述。

Lex 源程序中常用到的变量及函数：

- yyin 和 yyout：这是 Lex 中本身已定义的输入和输出文件指针。这两个变量指明了 lex 生成的词法分析器从哪里获得输入和输出到哪里。默认：键盘输入，屏幕输出。
- yytext 和 yyleng：这也是 lex 中已定义的变量，直接用就可以了。
- yytext：指向当前识别的词法单元（词文）的指针
- yyleng：当前词法单元的长度。
- ECHO：Lex 中预定义的宏，可以出现在动作中，相当于 fprintf(yyout, “%s”,yytext)，即输出当前匹配的词法单元。
- yylex()：词法分析器驱动程序，用 Lex 翻译器生成的 lex.yy.c 内必然含有这个函数。

- yywrap()：词法分析器遇到文件结尾时会调用 yywrap()来决定下一步怎么做：

  若 yywrap()返回 0，则继续扫描

  若返回 1，则返回报告文件结尾的 0 标记。

  由于词法分析器总会调用 yywrap，因此辅助函数中最好提供 yywrap，如果不提供，则在用 C 编译器编译 lex.yy.c 时，需要链接相应的库，库中会给出标准的 yywrap 函数（标准函数返回 1）。

```
{keyword} {
   printf("< %s , - >\n",yytext);
   charCnt_add(yyleng);
   words++ ;
}
{id} {
   printf("< id , %s >\n",yytext);
   charCnt_add(yyleng);
   words++;
}

{number} {
   printf("< num , %s >\n",yytext);
   charCnt_add(yyleng);
}

{operator} {
   printf("< operator , %s >\n",yytext);
   charCnt_add(yyleng);
}

{char} {
   printf("< char , %s >\n",yytext);
   charCnt_add(yyleng);
}

{string} {
   printf("< string , %s >\n",yytext);
   line_add();
       charCnt_add(yyleng);
}

{lineComment} {
   printf("< comment , %s >\n",yytext);
   line_add();
   charCnt_add(yyleng);
}

{comment} {
```

```
    line_add();
    printf("< comment , %s >\n",yytext);
    charCnt_add(yyleng);
}

{delimiter} {
    printf("< %s , - >\n",yytext);
        charCnt_add(yyleng);
}

{illNum} {
    printf("wrong numbers");
        printf("    wrong location: %d line\n",lines);
    line_add();
}

{illString} {
    printf("double quotes not match");
    printf("    wrong location: %d line\n",lines);
    charCnt_add(yyleng);
    line_add();
}

{space} {
     charCnt_add(1);

}

{macro} {
    printf("< macro , %s >\n",yytext);
    lines++;
    charCnt_add(yyleng);

}

. {
    charCnt_add(1);
    printf("unknown character: %s",yytext);
    printf("    wrong location: %d line\n",lines);
}

\n {
    charCnt_add(1);
    lines++;
```

```
}
```

其主要用到了 yytext 以及 yyleng 这两个变量。

yytext 变量存储的是此时根据正则表达式匹配成功的字符串，当其匹配成功后，会进入到相应的分支结构中来，进行相应的统计运算。

3.2.3 辅助函数段

辅助过程主要是对翻译规则的补充，翻译规则部分中某些动作需要调用的过程，如果不是 C 语言的 库函数，则要在此给出具体的定义。

```
int main (void) {
   yylex();
   printf("characters count: %d\n",character_cnt );
   printf("total lines: %d\n",lines);
   printf("word count: %d\n",words);
   return 0;
}
int yywrap() {
   return 1;
}
void charCnt_add(int n) {
   character_cnt+=n;
}
void line_add(void) {
   for(int i=0;i<yyleng;i++)
      if(yytext[i]=='\n')
         lines++;
}
```

通过构建上述的 lex 源程序，以此来进行自动的词法分析程序的生成，通过相应的命令生成 lex.yy.c 程序后，通过 gcc 对该程序进行编译，最终会生成一个.out 文件，这个文件是一个而执行文件，是一个生成的词法分析程序，通过运行该程序，并将自己所写的文件进行导入，然后表明结果的输出文件即可。

3.3 实验结果

- 实验用例：
  文件中的 test.c
- 实验结果：
  文件中的 output.txt

```
< id , InitKeyWord >
< ( , - >
< ) , - >
< ; , - >
< } , - >
characters count: 736
total lines: 45
word count: 54
```

4. 实验总结

本次实验，实践了如何设计自动机，以及各种读取单个字符和匹配之间的冲突和解决。主要有以下几个重点问题：

Q: 读取单个字符和回退引起的行列数值冲突

A: 全局**不采用回退**，直接进行单个字符的读取后直接判断词法匹配。

Q: 数字状态机转化和**十六进制八进制**的转换

A: 设计多状态自动机，并进行状态简化形成上述自动机。

Q: **转义符**问题

A: 预编译信息换行，字符串换行，单个字符转义。

Q: 关键字存储结构

A: 使用 map 存储，方便查找。