词法分析

唐子钰 2020212271

一 环境

- windows10
- visual-studio 2019
- c++11

二 实现内容

我基于c++实现了c语言的词法分析程序，完成了包括但不限于以下内容

- 实现了输入字符的**双buffer缓冲区**，支持每次读入一个字符或读到第一个非分割符，包括 `get_char()、read_buf()、retract()`
- **识别并存储c源程序每个token**，统计时能以记号**<token_name,value> line:number**的形式输出
- 识别并跳过注释，识别双引号字符串或单引号常字符
- 统计源程序语句行数、各类单词总数、字符总数等并在统计时输出统计结果
- 设计了几类**词法错误**并能报告错误信息包括错误所在行
- 支持**完整一次扫描**对源程序进行词法分析（跳过错误），也支持每次**单独识别一个token**（该功能在语法分析才实现）

三 词法分析设计说明

1 symbolic 符号定义

```C++
----------------------------------------------------init_mark-----------------------------------------------------
                                          +           plus_op
                                          -          minus_op
                                          *            mul_op
                                          /            div_op
                                          %            mod_op
                                         +=         plus_eq_op
                                         -=        minus_eq_op
                                         *=          mul_eq_op
                                         /=          div_eq_op
                                         %=          mod_eq_op
                                         ++       self_plus_op
                                         --      self_minus_op
                                         ->   right_pointer_op
                                          =         assign_op
                                          !            not_op
                                          >         greater_op
                                          <            less_op
                                         ==             eq_op
                                         !=         not_eq_op
                                         >=   greater_or_eq_op
                                         <=      less_or_eq_op
                                         <<      shift_left_op
                                         >>     shift_right_op
                                        <<=   shift_left_eq_op
                                        >>=  shift_right_eq_op
                                          &            and_op
                                          |             or_op
                                          ^            xor_op
                                          ~            inv_op
                                         &=         and_eq_op
                                         |=          or_eq_op
                                         ^=         xor_eq_op
                                         &&       and_logi_op
                                         ||        or_logi_op
                                          {        left_brace
                                          }       right_brace
                                          (   left_parentheses
```

```
)  right_parentheses
[       left_bracket
]      right_bracket
;          semicolon
,              comma
#            well_no
```

主要定义一些算术、关系运算符以及一些分隔符

2 关键词定义

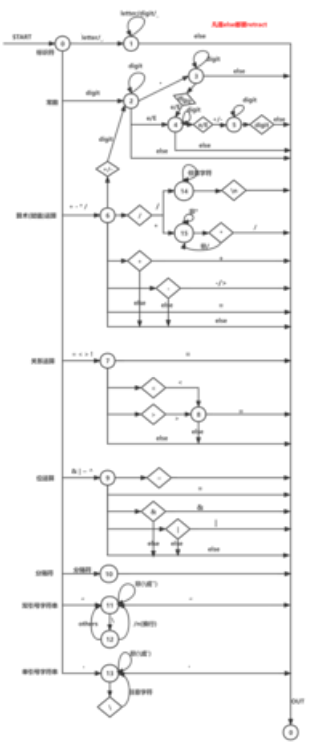
```C++
char keyWords[KEYS_NUM][MAX_TOKEN_LENGTH] = {
  "asm","do","if","return","try","auto","double","inline","short","typedef",
  "bool","dynamic_cast","int","signed","typeid","break","else","long","sizeof","typename",
  "case","enum","mutable","static","union","catch","explicit","namespace","static_cast","unsigned",
  "char","export","new","struct","using","class","extern","operator","switch","virtual",
  "const","FALSE","private","template","void","const_cast","float","protected","this","volatile",
  "continue","for","public","throw","wchar_t","default","friend","register","TRUE","while",
  "delete","goto","reinterpret_cast", };
```

3 token 定义

```C++
<id, ?>              //标识符
<keys, ?>            //关键字
<num, ?>             //常数
<symbolic_name, ->   //符号,如<plus_op, ->(+),<right_pointer_op, ->(->)
<string, "?">        //字符串
<char, '?'>          //字符常量
```

4 状态转换图

> 注意，该转换图中不涉及到错误词法的判断，不过也较为完整，原图见"词法分析状态图.png"



三 算法设计说明

1 封装词法分析类

`Lexical_Analysis.h` 头文件中声明词法分析的整个类，其包括：

- 源文件open以及控制字符数据读入
- 词法分析主程序
- 结果统计与记录

```C++
class Lexical_Analysis {
public:
  ifstream ifs;

  int state;
  int now_rows, id_counts, keys_counts, ArOp_counts, ReOp_counts, BitOp_counts, LogOp_counts, num_counts, Sep_counts, char_counts,string_co
  int tot,err_tot;
```

```
    string rec_marks[MAX_NODES][2];
    string errors[MAX_NODES];
;   char token[MAX_TOKEN_LENGTH];
    char buf[2][MAX_TOKEN_LENGTH]/*两个输入缓冲区*/;
    int nb, pb, lpb;//pb-指向输入缓冲区下一个需要读入的字符    nb-当前pb指针所在的缓冲区0/1    lpb-上一个非空格等字符在输入缓冲区的位置,以便retract
    int token_len;
    bool end, allow_read;

    Lexical_Analysis();//词法分析类构造函数
    bool getFile(char* f);//打开源文件
    bool isSlash(char c) { return c == '/'; }
    bool isDoubleQuotation(char c) { return c == '\"'; }//双引号
    bool isSingleQuotation(char c) { return  c == '\''; }//单引号
    bool isSeparator(char c) { return c == '{' || c == '}' || c == '[' || c == ']' || c == '(' || c == ')' || c == ',' || c == ';' || c == '#
    bool isArOp(char c) { return c == '+' || c == '-' || c == '*' || c == '/' || c == '%'; };//算术运算符
    bool isReOp(char c) { return c == '=' || c == '<' || c == '>' || c == '!'; };//关系运算符
    bool isBitOp(char c) { return c == '&' || c == '|' || c == '^' || c == '~'; };//位运算符
    bool isLetter(char c) { return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'); };//字母
    bool isNum(char c) { return c >= '0' && c <= '9'; };//数字
    bool isKey(char* s);//关键字
    char read();
    char get_char(int flag);//flag=1读一个字符包括' '、'\n'、'\t'、'\r'。flag=0,跳过这四个字符读一个字符。
    void retract();//pb指针回退一步
    void read_buf();//读入输入缓冲区
    void analyze();//词法分析主程序。
    void add_error(string error);//添加报错
    void statistics();//输出词法分析统计结果
};
```

2 数据读入

我是采用的 `double buffer` 的形式进行字符的输入，输入采取文件输入方式，即给定源文件 `src.txt` 进行字符读入。

- 核心方法

  维护两个输入缓冲区 `char buf[2][MAX_TOKEN_LENGTH]`,并维护两个指针： `pb`-指向输入缓冲区下一个需要读入的字符 `nb`-当前pb指针所在的缓冲区0/1

- getFile函数

```C++
/* 打开源代码文件 */
bool Lexical_Analysis::getFile(char* f)
```

- read_buf函数

```C++
/* 将MAX_TOKEN_LENGTH bytes的数据读入输入缓冲区置pb=0,nb^=1 */
void Lexical_Analysis::read_buf()
```

  使用 `ifstream>>noskipws` one by one 读入

- get_char函数

```C++
/* 从输入缓冲区中读入下一个字符,flag=1表示不跳过空格、换行等直接读入下一个字符
同理flag=0表示跳过空格等读下一个字符,如果缓冲区满则调read_buf函数 */
char Lexical_Analysis::get_char(int flag = 0);
```

- retract函数

```C++
/* 字符pb指针回退一步,在某个token词法分析结束时需要retract(多读一个char) */
void Lexical_Analysis::retract()
```

  注：当字符指针pb=0时回退，我们将回退到上一个缓冲区的末尾，因此下一次不用调用 `read_buf` 函数，维护一个 `allow_read` 变量以免重复读入缓冲区

3 词法分析主程序

根据状态转换图可以轻松编程得到词法分析的主程序，初始状态为0，判断第一个输入字符 `c` (非空、换行等)满足什么条件进入对应的状态进行词法分析即可，最后对于多读的字符需要调用 `retract` 函数进行回退，实际上就是 `pb` 指针模减1。

注：token定义为 `char token[MAX_TOKEN_LENGTH]`，即我们默认不会出现超过 `MAX_TOKEN_LENGTH=256` 的记号

```C++
/* 全局进行一次词法分析 */
void Lexical_Analysis::analyze() {
  char c;
```

```
    read_buf();
    while (1) {
      switch (state)
      {
      case 0://识别新词
        memset(token, 0, sizeof(token));
        token_len = 0;
        c = get_char();
        token[token_len++] = c;
        if (c == '\0')return;//终止符
        if (isLetter(c) || c == '_')state = 1;// id
        else if (isNum(c))state = 2; // num
        else if (isArOp(c)) state = 6; // +、-、*、/、%
        else if (isReOp(c)) state = 7;// >、！、<、=
        else if (isBitOp(c))state = 9;// &、|、^、~
        else if (isSeparator(c))state = 10;// 分隔符
        else if (isDoubleQuotation(c)) state = 11;//双引号字符串,注意双引号可以换行而单引号不行
        else if (isSingleQuotation(c))state = 13;//单引号字符串
        else if (isSlash(c)) state = 14;//斜杠即注释。
        break;

      case 1:// 识别标识符
        ...
      case 2-5: // 识别符号数和非符号数
        ...
      case 6:// 算术运算符;注意如果是//或者/*转到注释对应的状态
        ...
      case 7-8:// 关系运算符
        ...

      case 9:// 位运算符
        ...
      case 10:// separator分隔符
        ...
      case 11-12:// 双引号字符串-支持换行(单\)和转义
        ...
      case 13:// 单引号字符串，只允许一个字符且不允许空串且\不hint换行
        ...
      case 14:// 双斜杠注释,\可以hint转义或者换行
        ...
      case 15:// /**/段落注释
        ...
      case 101: {//[Error] unable to find numeric literal operator 'operator""a'
      default:
        break;
      }
    }
  }
```

> 下面仅说明一些细节点的状态转换，其他如标识符识别、分隔符等状态就不再赘述

标识符识别

符合条件正则表达式：$(letter|\_)(letter|digit|\_)^*$

注意分析结束需要判断其是否为 `keys`

常数识别

数的识别包括符号数和非符号数，带符号的数包括两类：

1. $(+|-)num$：即带正负的数，`实际上num等价于2中的数`
2. $num\ e(+|-|\epsilon)\ num$：即带科学技术法的数，`num可以是整数或者小数，且e|E后可接+|-`

注：语法分析不会识别 `num` 前的 $+-$ 号，其必须作为操作符；

具体状态的转换在前面的 `转换图` 已给出，可以大概看下缩水版的代码

C++

```
/* 识别符号数和非符号数 */
    case 2://整数部分
      c = get_char(1);
      token[token_len++] = c;
      if (isNum(c)) state = 2;//维持
      else if (c == '.') state = 3;//转小数识别状态
      else if (c == 'e' || c == 'E') state = 4;//转指数识别状态
      else //识别结束
      break;
    case 3://小数部分
      c = get_char(1);
      token[token_len++] = c;
```

```
      if (isNum(c)) state = 3;//维持
      else if (c == 'e' || c == 'E') state = 4; //转指数识别状态
      else { //识别结束
        retract();
        ...
      }break;
    case 4://指数部分
      c = get_char(1);
      token[token_len++] = c;
      if (...)state = 5;//科学计数法开头可能是e+、e-,转指数纯整数state
      else if (isNum(c)) state = 4;
      else {
        retract();
      }break;
    case 5://指数纯数字部分
      ...
```

支持的常数格式举例如下

C++

```
1
1.
1.1
+1
-1
+1.5
-1.5
1e5
1.5e5
+1.5e5
1e-5
```

算术运算符

算术运算符指如下运算符：

C++

```
/* 算术（赋值）运算 */
"+","plus_op",
"-","minus_op",
"*","mul_op",
"/","div_op",
"%","mod_op",

"+=","plus_eq_op",
"-=","minus_eq_op",
"*=","mul_eq_op",
"/=","div_eq_op",
"%=","mod_eq_op",

"++","self_plus_op",
"--","self_minus_op",

"->","right_pointer_op",
```

在状态6中识别算术运算符，在程序中我们会特别判断`//`或者`/*`的情况以撞到注释状态即14、15的识别。

关系运算符

关系运算符指如下运算符：

C++

```
/* 关系运算 */
"=","assign_op",
"!","not_op",
">","greater_op",
"<","less_op",

"==","eq_op",
"!=","not_eq_op",
">=","greater_or_eq_op",
"<=","less_or_eq_op",

"<<","shift_left_op", //实际上为位运算,为了方便放此处
">>","shift_right_op",

"<<=","shift_left_eq_op",
">>=","shift_right_eq_op",
```

在状态7、8中识别，状态8是特别处理三个字符长度的`<<=`或`>>=`符号，这实际上是位运算符不过为了方便就放这了

位运算符

位运算符指如下运算符：

```C++
    /* 位运算 */
    "&","and_op",
    "|","or_op",
    "^","xor_op",

    "~","inv_op",

    "&=","and_eq_op",
    "|=","or_eq_op",
    "^=","xor_eq_op",

    "&&","and_logi_op", //实际上为逻辑运算，为方便放此处
    "||","or_logi_op",
```

在状态9中识别不再赘述

分隔符

```C++
  /* 分隔符 */
  "{","left_brace",
  "}","right_brace",
  "(","left_parentheses",
  ")","right_parentheses",
  "[","left_bracket",
  "]","right_bracket",
  ";","semicolon",
  ",","comma",
  "#","well_no",
```

在状态10识别

双引号字符串

这是词法分析设计时一个重要的点，需要注意以下地方：

1. 以`"`开始以`"`结束
2. 斜杠`\`包括两层含义

   ①换行，即支持下面这种写法

   ```C++
   char c[10]="asd\
   a\
   a\
   "
   ```

   ②转义

   ```C++
   char c[10]="aa\"aa\n"
   ```

至于错误字符串定义（如无匹配的`"`）见后续的errors模块，其定义了本程序考虑的所有词法错误，这里假设输入字符串是正确的

最终我设置了2个状态11和12来进行分析，状态12特别来处理斜杠`\`

单引号常字符

与双引号字符串不同，单引号字符的`\`只支持转义，不支持换行（即char只能在一行定义），且单引号定义的字符不能定义`空字符`和`multi 字符`，这些在error中会谈到。

```C++
  char c='s';
  char cc='\n';
  char ccc='';// error
  char cccc='s
```

使用一个状态13即可完成分析

双斜杠注释

`//`注释需注意`\`能支持换行，常规情况只能注释一行

```C++
//test\
斜杠换行\
1
```

使用一个状态14即可完成分析

段落注释

`/**/`注释不包括任何转义其它功能，直接注释头`/*`和尾`*/`中的内容即可

```C++
/* test\*\/ */
```

使用一个状态15即可完成分析

四 error 设计

> 所有错误出现后程序都会读入跳过该段并分析后面的源程序，不影响整体的词法分析。这些错误的检测夹杂在词法分析主程序中，在词法分析状态转换图基础下做了适当扩展以实现。词法分析的错误并未涉及到太多，因为能出错的token也不多，大多数要靠语法分析检测，如without initializer。

- [Error] unable to find numeric literal operator operator "?"

  该错误的定义是在分析`num`时，接着输入的字符`c`为 $Letter(not\ e|E),\_$ ，显然一个常数接一个可能是标识符的串肯定会出错，词法分析不允许这种情况。

  ```C++
  int a=123ab;
  >> [Error] unable to find numeric literal operator "ab"
  ```

- [Error] exponent has no digits

  该错误的出现是在定义有符号常数时使用`科学计数法`但后续没有接数字，如下

  ```C++
  int a=1e,b=1e+
  >> [Error] exponent has no digits
  ```

- [Error] missing terminating " character

  该错误的出现是在定义`"`定义的字符串时，其结尾没有与之匹配的`"`，注意由于字符串可以使用`\`来定义换行且可使用`\`转义`\"`，因此这些地方需要做判断，以正确识别错误，具体方法是连续读入一行直到出现`\`、`\n`、`\0`或者`"`，如果是`\n`或`\0`则说明字符串定义结束，报该错；如果是`\`转到状态12分析`\`是转义or换行；如果是`"`则得到正确的字符串。

  ```C++
  "moijiohio\
  kmlkm
  >> [Error] missing terminating " character
  ```

- [Error] missing terminating ' character

  该错误同理是在`'`即单引号定义字符时，其结尾没有与之匹配的`'`，识别方法和前面类似，就不再赘述。需要注意的是我们先假设定义正确，分析完`' '`内内容，如果是`空串`或者`multi串`，需报后续的错。

  ```C++
  char c='a
  >> [Error] missing terminating ' character
  ```

- [Error] empty character constant

  接着上文，`'`定义了空串会该报错

  ```C++
  char c=''
  >> [Error] empty character constant
  ```

- [Error] multi-character character constant

接着上文，`'`定义了复串会报错，注意可以允许`\`hint的转义符号，即`'\n'`、`'\''`可通过。

```C++
char c='aa'
>> [Error] multi-character character constant
```

五 测试

除了必须的词法分析外，我还对各种出现的token数量、字符总数量、行数进行了完整的统计，并规范化输出所有识别的token，且输出所有的词法分析error，统计随着主程序运行一起进行，统计结果和报错由`statistics函数`输出。

且所有识别出的`token`都被我以`string`的形式存于`rec_marks`中，如果需要拓展可以将词法分析主程序改造成**一次识别一个**`token`的形式，只需要主程序每次第二次进入`state=0`返回即可，很容易实现。

1 id.txt

主要测试`id`、`num`的分析

```C++
int main(){
aa
_aab
_aa3b
int
double awdawd _43r23
+2315
1e9 200 1.546 0.561e023 1e+5 1e-5
1e
1e+
int a=123ab;
112adq_qfq
1.e5
}

```

id.txt源文件

```C++
请输入源文件名:
id.txt
<keys,int>              lines:1
<id,main>               lines:1
<left_parentheses, >    lines:1
<right_parentheses, >   lines:1
<left_brace, >          lines:1
<id,aa>                 lines:2
<id,_aab>               lines:3
<id,_aa3b>              lines:4
<keys,int>              lines:5
<keys,double>           lines:6
<id,awdawd>             lines:6
<id,_43r23>             lines:6
<num,+2315>             lines:7
<num,1e9>               lines:8
<num,200>               lines:8
<num,1.546>             lines:8
<num,0.561e023>         lines:8
<num,1e+5>              lines:8
<num,1e-5>              lines:8
<keys,int>              lines:11
<id,a>                  lines:11
<assign_op, >           lines:11
<semicolon, >           lines:11
<num,1.e5>              lines:13
<right_brace, >         lines:14
------------------------------Statistical results-------------------------------
语句行数：15
字符总数：106
标识符总数：7
关键字总数：4
算术运算符总数：0
关系运算符总数：1 //注意赋值运算我当作关系运算符来统计的
位运算符总数：0
逻辑运算符总数：0
常数总数(包括整数、小数和无符号数)：8
分隔符总数：5
字符常数总数：0
字符串总数：0
注释总数：0
```

```
-------------------------------------errors-------------------------------------
lines:9 [Error] exponent has no digits
lines:10 [Error] exponent has no digits
lines:11 [Error] unable to find numeric literal operator "ab"
lines:12 [Error] unable to find numeric literal operator "adq_qfq"
```

2 op.txt

主要测试算术、位、关系等运算符

C++

```
int a=b+c
a+=c
a++
a--
->

=
==
<
<=
<<
<<=

&
&=
&&
||
~a

```

op.txt 源文件

C++

```
请输入源文件名:
op.txt
<keys,int>              lines:1
<id,a>                  lines:1
<assign_op, >           lines:1
<id,b>                  lines:1
<plus_op, >             lines:1
<id,c>                  lines:1
<id,a>                  lines:2
<plus_eq_op, >          lines:2
<id,c>                  lines:2
<id,a>                  lines:3
<self_plus_op, >        lines:3
<id,a>                  lines:4
<self_minus_op, >       lines:4
<right_pointer_op, >    lines:5
<assign_op, >           lines:7
<eq_op, >               lines:8
<less_op, >             lines:9
<less_or_eq_op, >       lines:10
<shift_left_op, >       lines:11
<shift_left_eq_op, >    lines:12
<and_op, >              lines:14
<and_eq_op, >           lines:15
<and_logi_op, >         lines:16
<or_logi_op, >          lines:17
<inv_op, >              lines:18
<id,a>                  lines:18
------------------------------Statistical results-------------------------------
语句行数：19
字符总数：40
标识符总数：8
关键字总数：1
算术运算符总数：4
关系运算符总数：5
位运算符总数：5
逻辑运算符总数：2
常数总数(包括整数、小数和无符号数)：0
分隔符总数：0
字符常数总数：0
字符串总数：0
注释总数：0
-------------------------------------errors-------------------------------------
```

3 str.txt

该文件主要测试字符串、字符以及注释的识别

```C++
"gugigi"
"bhj\n"
"kjkbkjbkjb\
jiohoiho\
jiojoij"
"moijiohio\
kmlkm
/*213*/
's''s\k
//dqwdqwdq\
qweq
''
'aa'

```

str.txt 源文件

```C++
请输入源文件名:
str.txt
<string, "gugigi">lines:1
<string, "bhj\n">lines:2
<string, "kjkbkjbkjbjiohoihojiojoij">lines:5
<char, 's'>lines:9
------------------------------Statistical results------------------------------
语句行数：14
字符总数：95
标识符总数：0
关键字总数：0
算术运算符总数：0
关系运算符总数：0
位运算符总数：0
逻辑运算符总数：0
常数总数(包括整数、小数和无符号数)：0
分隔符总数：0
字符常数总数：3 //注意就算定义了空串或复串我都统计在内
字符串总数：3
注释总数：2
-------------------------------------errors-------------------------------------
lines:7 [Error] missing terminating " character
lines:9 [Error] missing terminating ' character
lines:12 [Error] empty character constant
lines:13 [Error] multi-character character constant
```

六 总结

本次词法分析实验较为完整地完成了实验要求和内容，做到了各类 `token` 的识别、注释的跳过、有符号和无符号数识别状态转换、双引号单引号状态转换逻辑实现以及各类运算符、分隔符的识别，并且列出一系列词法分析错误以供参考，统计所有词法分析结果。且我设计的算法封装性强，各状态部分独立，可拓展性强，并且我绘制了一个完整的状态转换图以供参考，且在命令行中做了较为规范地输出。综合而言我认为自己较好地完成了本次实验，对词法分析理解更深刻，收获颇丰。

补充!

在实现完语法分析后，我又对词法分析的代码进行了较多的重构，因此词法分析的报告是基于一个 previous version，并非提交的最终版本（不过内容也大体一致），重构新补充的内容可以见语法分析的开头有详细说明，或者见验收文档的总结说明。