

编译原理 词法分析 实验报告

姓名：陈俊卉 班级：2020219111 学号：2020212256

编译原理 词法分析 实验报告

- 一、实验内容
- 二、实验环境
- 三、方法一原理分析
 - 1、词法分析程序的功能
 - 2、词法分析程序单词的归类
 - 3、设计思路
 - ① 给出描述该语言各种单词符号的词法以及规则
 - ② 构建状态转换图
- 四、方法一代码展示
- 五、方法一测试
 - ① 测试代码：（完全正确）
验证：
 - ② 测试代码：（28行双引号没有配对）
分析：
- 六、方法二分析
- 七、方法二代码展示
- 八、方法二测试
- 九、实验总结与心得

一、实验内容

设计并实现C语言的词法分析程序，要求实现如下功能：

- (1) 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。
- (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) 检查源程序中存在的词法错误，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回复，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存在的所有词法错误。

实现要求：可以选择一下两种方案中的一种实现。

方法1：采用编程语言，手工编写词法分析程序；

方法2：基于LEX，自动生成词法分析程序。

本次实验采用方法1、2实现，使用的编程语言为C++。

二、实验环境

- windows10系统
- Visual Studio Code
- C++ 17

三、方法一原理分析

1、词法分析程序的功能

词法分析是编译过程的第一步，主要任务是从左到右逐个字符对源程序进行扫描，按照源语言的词法规则识别出单词符号，将识别出来的标识符存入符号表中，并产生用于语法分析的记号序列。其主要过程如下：

- 扫描源程序的字符流；
- 根据目标语言的各种词法规则识别出程序中各种单词、符号的种类，跳过注释和空格；
- 对单词、符号进行检查，若无问题则产生记号序列，否则抛出错误；
- 将识别出的序列放入创建符号表中，给出可能出现的错误的行数。

2、词法分析程序单词的归类

- 变量名(variable)
- 关键字(keyword) C语言共有32个关键字
- 数字常量(number)
 - 整数
 - 小数
 - 科学计数的数字
- 分隔符(界符)、运算符(operator)

keyword与operator具体如下所示。

```
1 // 关键字数组
2 // C语言关键字共有32个
3 char *key_word[] = {
4     // 数据类型关键字
5     "char",
6     "short",
7     "int",
8     "long",
9     "signed",
10    "unsigned",
11    "float",
12    "double",
13    "struct",
14    "union",
15    "enum",
16    "void",
17    // 控制语句关键字
18    "for",
19    "do",
20    "while",
21    "break",
22    "continue",
23    "if",
24    "else",
25    "goto",
26    "switch",
27    "case",
28    "default",
29    "return",
```

```
30 // 存储类型关键字
31 "auto",
32 "extern",
33 "register",
34 "static",
35 "typedef",
36 // 其他关键字
37 "const",
38 "sizeof",
39 "volatile",
40 };
41
42
43 // 运算符和界符
44 char *sign[45] = {
45     "[",
46     "]",
47     "(",
48     ")",
49     ".",
50     "->",
51     "-",
52     "++",
53     "--",
54     "*",
55     "&",
56     "!",
57     "~",
58     "/",
59     "%",
60     "+",
61     "-",
62     "<<",
63     ">>",
64     ">",
65     ">=",
66     "<",
67     "<=",
68     "==",
69     "!=",
70     "^",
71     "|",
72     "&&",
73     "||",
74     "=",
75     "/=",
76     "*=",
77     "%=",
78     "+=",
79     "-=",
80     "<<=",
81     ">>=",
82     "&=",
83     "^=",
84     "!=",
85     ",",
86     ";",
87     "{",
```

```

88     "}",
89     "#",
90 };
91

```

3、设计思路

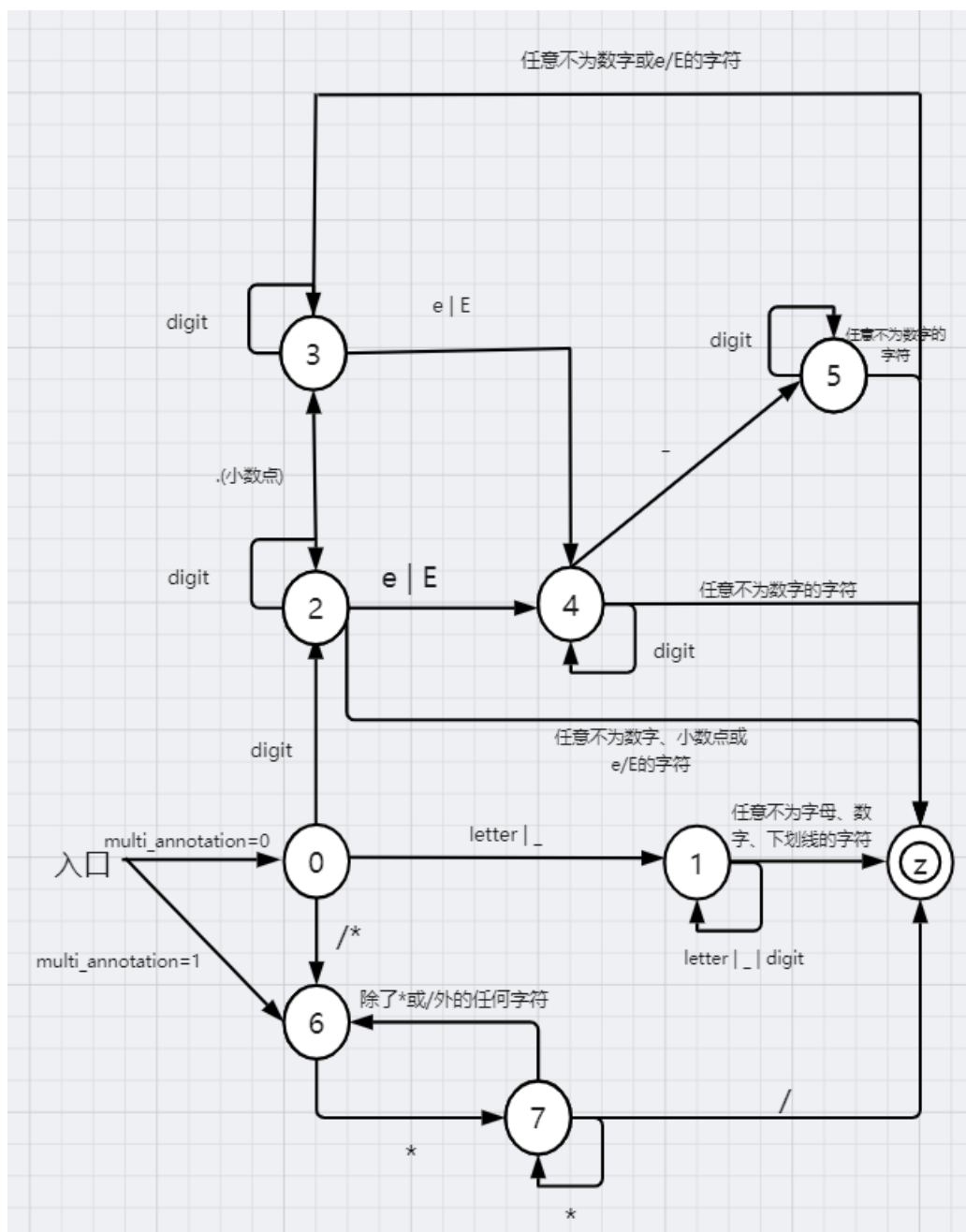
代码模板参考的是课本上的基础代码，并对其加以改进。

输出结构： line x <type, word>

① 给出描述该语言各种单词符号的词法以及规则

- 该点已经在上文给出。

② 构建状态转换图



解释：

- 大部分匹配分界符和运算符，以及单行注释的过程都在state = 0时已经处理完毕，由于种类过于繁多，所以这里并没有在状态转换图中体现出来，将会在下一部分的代码展示中给出。
- 由于该状态图表示的是提取一个词法记号的过程，而本次实验为了标注行数，笔者采用了一行一行输入词法分析函数的方式进行读取。而多行注释是跨行的，所以我们需要一个全局变量 multi-annotation 表示当前是否为多行注释状态（跨行后由于词法分析函数的重新调用，state会被清空）。
- 各个状态的解释：
 - 0：初始状态。处理第一个字符判断后续状态的过程。在此状态，简单的分界符和运算符，与单行注释已经处理完毕。
 - 1：word状态。直到读取到不符合变量名规范的字符才会停止。
 - 2：整数状态。遇到小数点会进入小数状态，遇到e或E进入科学计数法状态，否则结束。
 - 3：小数状态。在C中，即使小数点后没有数字，也是正确的。遇到e或E进入科学计数法状态。
 - 4：科学计数法状态。遇到负号进入负幂次状态（事实上，负号前面如果是数字，也是没有错误的，只不过负号后面不是幂次而是减法了）
 - 5：科学计数法负幂次状态。
 - 6：多行注释中读取到 /* 的状态。
 - 7：多行注释中读取到 /* * 的状态。

四、方法一代码展示

一个需要注意的地方：

- 当前版本的C++中，当想要将string直接当作char *使用时会发生强制类型转换，这会引发编译警告：

```
1 warning: ISO C++ forbids converting a string constant to 'char*' [-Wwrite-strings]
```

可以用c_str()或data()解决该问题。但为了代码的简洁起见，我们仍保留这样的方式。

```
1 #include<iostream>
2 #include<string.h>
3 #include<stdlib.h>
4 #include<fstream>
5 using namespace std;
6
7 #define MAX_BUFFER_SIZE 10000
8 #define MAX_STR_SIZE 10000
9 // 每一行的缓冲区
10 char buffer[MAX_BUFFER_SIZE];
11 // 读取文件中被单引号和双引号引住的字符串 或 一串数字 或 变量
12 char str_in_file[MAX_STR_SIZE];
13
14 // 记录各种数据
15 // 行数
16 int line_num = 0;
17 // 字符总数（包括空格）
18 int sign_num = 0;
19 // 单词总数（关键字+变量名）
```

```
20 int word_num = 0;
21 // 关键字数目
22 int keyword_num = 0;
23 // 变量名数目
24 int variable_num = 0;
25 // 数字常量数目
26 int number_num = 0;
27 // 分界符和操作符数目
28 int operate_num = 0;
29 // 字符串常量数目
30 int str_num = 0;
31 // 单行注释数目
32 int single_ann_num = 0;
33 // 多行注释数目
34 int multi_ann_num = 0;
35
36
37 // 多行注释标记
38 int multi_annotation = 0;
39
40
41 // 关键字数组
42 // C语言关键字共有32个
43 char *key_word[] = {
44     // 数据类型关键字
45     "char",
46     "short",
47     "int",
48     "long",
49     "signed",
50     "unsigned",
51     "float",
52     "double",
53     "struct",
54     "union",
55     "enum",
56     "void",
57     // 控制语句关键字
58     "for",
59     "do",
60     "while",
61     "break",
62     "continue",
63     "if",
64     "else",
65     "goto",
66     "switch",
67     "case",
68     "default",
69     "return",
70     // 存储类型关键字
71     "auto",
72     "extern",
73     "register",
74     "static",
75     "typedef",
76     // 其他关键字
77     "const",
```

```

78     "sizeof",
79     "volatile",
80 };
81
82
83 // 获取下一个字符的函数，并让forward指针向后移动一位
84 char get_char(char *&forward){
85     char this_char = *forward;
86     forward ++;
87     sign_num ++;
88     return this_char;
89 }
90
91 // 回退一格
92 void get_backward(char *&forward){
93     forward --;
94 }
95
96 /*
97     打印结果
98     结构:      行数 <word type, word>
99 */
100 void print(int line_index, char *type, char *str){
101     cout << "line " << line_index << " <" << type << ", " << str << ">" << endl;
102 }
103
104 // 是否为数字
105 bool is_num(char ch){
106     if (ch >= '0' && ch <= '9') return true;
107     else return false;
108 }
109
110 // 是否为字母
111 bool is_letter(char ch){
112     if ((ch >= 'a' && ch <= 'z' ) || (ch >= 'A' && ch <= 'Z')) return true;
113     else return false;
114 }
115
116 // 是否为关键字 一共有32个关键字
117 bool is_keyword(char *str){
118     for (int i = 0; i < 32; i++){
119         if(strcmp(str, key_word[i]) == 0){
120             // 如果是关键字 返回下标 (从1开始)
121             return true;
122         }
123     }
124     // 否则返回false
125     return false;
126 }
127
128
129
130 /*
131     词法分析函数，传入forward指针(当前行)和当前行的下标
132     该函数参考课本中程序的结构
133 */

```

```

134 void lexical_analysis(int line_index, char *forward){
135     // 状态初始化
136     int state = 0;
137
138     // 因为分行 所以只可能到状态6
139     if (multi_annotation == 1){
140         state = 6;
141     }
142
143     // 定义当前字符
144     char now_char = ' ';
145     // 标记字符串位置
146     int pos = 0;
147     // 如果当前还没有到达该行的结束
148     while (now_char != '\0') {
149         switch (state)
150         {
151             case 0:
152             {
153                 // 当前字符
154                 now_char = get_char(forward);
155
156                 // 根据第一个字符进行状态的切换
157                 switch (now_char) {
158                     // 空格跳过
159                     case ' ':
160                     {
161                         pos = 0;
162                         break;
163                     }
164
165                     // 分界符
166                     case '(':
167                     case ')':
168                     case '[':
169                     case ']':
170                     case '{':
171                     case '}':
172                     {
173                         pos = 0;
174
175                         char op[2];
176                         op[0] = now_char;
177
178                         // 分界符和操作符的数量+1
179                         operate_num ++;
180                         // 输出
181                         print(line_index, "delimiter", op);
182                         break;
183                     }
184                     // 单引号和双引号
185                     case '\':
186                     {
187                         state = 0;
188                         // 累计分界符和操作符数目
189                         operate_num ++;
190                         print(line_index, "delimiter", "\\");
191

```



```

192
193 // 读取字符串内的内容
194 now_char = get_char(forward);
195
196 // 记录是否有第二个引号
197 int mark_flag = 1;
198
199 while ((now_char != '\')){
200     // 不为终止符则加入
201     if (now_char != '\0'){
202         str_in_file[pos] = now_char;
203         pos ++;
204         now_char = get_char(forward);
205     }
206     // 如果在第二个单引号出现之前出现了终止符号 抛出一个错
    误
207     else{
208         print(line_index, "error:the str end
without another quotation mark", str_in_file);
209         mark_flag = 0;
210         break;
211     }
212 }
213
214 if (mark_flag == 1) {
215     // 正确生成了字符串 为其加入终结符 避免之前的影响
216     str_in_file[pos] = '\0';
217     // 输出正确内容
218     print(line_index, "string", str_in_file);
219     // 输出第二个引号
220     print(line_index, "delimiter", "\\");
221     // 累计字符串数目
222     str_num ++;
223     // 累计分界符和操作符数目
224     operate_num ++;
225 }
226
227 // reset
228 pos = 0;
229 break;
230 }
231 // 与单引号一样
232 case '"':
233 {
234     state = 0;
235     // 累计分界符和操作符数目
236     operate_num ++;
237     print(line_index, "delimiter", "\\");
238
239     // 读取字符串内的内容
240     now_char = get_char(forward);
241
242     // 记录是否有第二个引号
243     int mark_flag = 1;
244
245     while ((now_char != '"')){
246         // 不为终止符则加入
247         if (now_char != '\0'){

```

```

248         str_in_file[pos] = now_char;
249         pos ++;
250         now_char = get_char(forward);
251     }
252     // 如果在第二个双引号出现之前出现了终止符号 抛出一个错
误
253     else{
254         print(line_index, "error:the str end
without another quotation mark", str_in_file);
255         mark_flag = 0;
256         break;
257     }
258 }
259
260 if (mark_flag == 1) {
261     // 正确生成了字符串 为其加入终结符 避免之前的影响
262     str_in_file[pos] = '\0';
263     // 输出正确内容
264     print(line_index, "string", str_in_file);
265     // 输出第二个引号
266     print(line_index, "delimiter", "\"");
267     // 累计字符串数目
268     str_num ++;
269     // 累计分界符和操作符数目
270     operate_num ++;
271 }
272
273 // reset
274 pos = 0;
275 break;
276 }
277
278 // +
279 // 可能只有一个+ 也可能是++ +=
280 case '+':
281 {
282     state = 0;
283     // 累计分界符和操作符数目
284     operate_num ++;
285     now_char = get_char(forward);
286
287     switch (now_char) {
288         // ++
289         case '+':
290         {
291             print(line_index, "operator", "++");
292             pos = 0;
293             break;
294         }
295         // +=
296         case '=':
297         {
298             print(line_index, "operator", "+=");
299             pos = 0;
300             break;
301         }
302         // 其他情况 只有第一个加号或是错误的操作符
303         // 这里不考虑对形如“+-”的错误操作符

```

```

304         default:
305         {
306             // 因为刚才读了第一个+的下一个字符，这里只输出一个
+，所以指针需要返回

307             get_backward(forward);
308             print(line_index, "operator", "+");
309             pos = 0;
310             break;
311         }
312     }
313     break;
314 }
315
316 // - 与 +类似
317 case '-':
318 {
319     state = 0;
320     // 累计分界符和操作符数目
321     operate_num ++;
322     now_char = get_char(forward);
323
324     switch (now_char) {
325         case '-':
326         {
327             print(line_index, "operator", "--");
328             pos = 0;
329             break;
330         }
331         case '=':
332         {
333             print(line_index, "operator", "-=");
334             pos = 0;
335             break;
336         }
337         // 可能为指针符号
338         case '>':
339         {
340             print(line_index, "operator", "->");
341             pos = 0;
342             break;
343         }
344         default:
345         {
346             get_backward(forward);
347             print(line_index, "operator", "-");
348             pos = 0;
349             break;
350         }
351     }
352     break;
353 }
354
355 // = 与 +类似
356 case '=':
357 {
358     state = 0;
359     // 累计分界符和操作符数目
360     operate_num ++;

```

```

361         now_char = get_char(forward);
362
363         switch (now_char) {
364             case '=':
365             {
366                 print(line_index, "operator", "==");
367                 pos = 0;
368                 break;
369             }
370             default:
371             {
372                 get_backward(forward);
373                 print(line_index, "operator", "=");
374                 pos = 0;
375                 break;
376             }
377         }
378         break;
379     }
380
381     // < 与 +类似
382     case '<':
383     {
384         state = 0;
385         // 累计分界符和操作符数目
386         operate_num ++;
387         now_char = get_char(forward);
388
389         switch (now_char) {
390             case '<':
391             {
392                 // 可能为<=
393                 char now_char = get_char(forward);
394                 if (now_char == '='){
395                     print(line_index, "operator", "<=");
396                     pos = 0;
397                     break;
398                 }
399                 else{
400                     // 回退
401                     get_backward(forward);
402                     print(line_index, "operator", "<");
403                     pos = 0;
404                     break;
405                 }
406             }
407             case '=':
408             {
409                 print(line_index, "operator", "<=");
410                 pos = 0;
411                 break;
412             }
413             default:
414             {
415                 get_backward(forward);
416                 print(line_index, "operator", "<");
417                 pos = 0;
418                 break;

```

```
419         }
420     }
421     break;
422 }
423
424 // > 与 +类似
425 case '>':
426 {
427     state = 0;
428     // 累计分界符和操作符数目
429     operate_num ++;
430     now_char = get_char(forward);
431
432     switch (now_char) {
433         case '>':
434         {
435             // 可能为>>=
436             char now_char = get_char(forward);
437             if (now_char == '='){
438                 print(line_index, "operator", ">>=");
439                 pos = 0;
440                 break;
441             }
442             else{
443                 // 回退
444                 get_backward(forward);
445                 print(line_index, "operator", ">>");
446                 pos = 0;
447                 break;
448             }
449         }
450         case '=':
451         {
452             print(line_index, "operator", ">=");
453             pos = 0;
454             break;
455         }
456         default:
457         {
458             get_backward(forward);
459             print(line_index, "operator", ">");
460             pos = 0;
461             break;
462         }
463     }
464     break;
465 }
466
467 case '*':
468 {
469     state = 0;
470     // 累计分界符和操作符数目
471     operate_num ++;
472     now_char = get_char(forward);
473
474     switch (now_char) {
475         case '=':
476         {
```

```

477         print(line_index, "operator", "*=");
478         pos = 0;
479         break;
480     }
481     default:
482     {
483         get_backward(forward);
484         print(line_index, "operator", "*");
485         pos = 0;
486         break;
487     }
488 }
489 break;
490 }
491
492 case '/':
493 {
494     state = 0;
495     // 累计分界符和操作符数目
496     operate_num ++;
497     now_char = get_char(forward);
498
499     switch (now_char) {
500         case '=':
501         {
502             print(line_index, "operator", "/=");
503             pos = 0;
504             break;
505         }
506
507         // 行内注释
508         case '/':
509         {
510             // reset
511             state = 0;
512             pos = 0;
513             single_ann_num ++;
514             // 该行剩下的均已经不需要遍历 直接到下一行即可 直
515 接return
516             cout << "line " << line_index << " have a
single annotation skipping" << endl;
517             return ;
518         }
519
520         // 多行注释
521         case '*':
522         {
523             // 到多行注释寻找*状态
524             state = 6;
525             pos = 0;
526             multi_ann_num ++;
527             break;
528         }
529
530         default:
531         {
532             get_backward(forward);
533             print(line_index, "operator", "/");

```

```
533         pos = 0;
534         break;
535     }
536 }
537 break;
538 }
539
540 case '&':
541 {
542     state = 0;
543     // 累计分界符和操作符数目
544     operate_num ++;
545     now_char = get_char(forward);
546
547     switch (now_char) {
548         case '&':
549         {
550             print(line_index, "operator", "&");
551             pos = 0;
552             break;
553         }
554         case '=':
555         {
556             print(line_index, "operator", "&=");
557             pos = 0;
558             break;
559         }
560         default:
561         {
562             get_backward(forward);
563             print(line_index, "operator", "&");
564             pos = 0;
565             break;
566         }
567     }
568     break;
569 }
570
571 case '|':
572 {
573     state = 0;
574     // 累计分界符和操作符数目
575     operate_num ++;
576     now_char = get_char(forward);
577
578     switch (now_char) {
579         case '|':
580         {
581             print(line_index, "operator", "||");
582             pos = 0;
583             break;
584         }
585         default:
586         {
587             get_backward(forward);
588             print(line_index, "operator", "|");
589             pos = 0;
590             break;
591         }
592     }
593 }
```

```
591     }
592     }
593     break;
594 }
595
596 case '!':
597 {
598     state = 0;
599     // 累计分界符和操作符数目
600     operate_num ++;
601     now_char = get_char(forward);
602
603     switch (now_char) {
604         case '=':
605         {
606             print(line_index, "operator", "!=");
607             pos = 0;
608             break;
609         }
610         default:
611         {
612             get_backward(forward);
613             print(line_index, "operator", "!");
614             pos = 0;
615             break;
616         }
617     }
618     break;
619 }
620
621 case '%':
622 {
623     state = 0;
624     // 累计分界符和操作符数目
625     operate_num ++;
626     now_char = get_char(forward);
627
628     switch (now_char) {
629         case '=':
630         {
631             print(line_index, "operator", "%=");
632             pos = 0;
633             break;
634         }
635         default:
636         {
637             get_backward(forward);
638             print(line_index, "operator", "%");
639             pos = 0;
640             break;
641         }
642     }
643     break;
644 }
645
646 case '^':
647 {
648     state = 0;
```



```
649 // 累计分界符和操作符数目
650 operate_num ++;
651 now_char = get_char(forward);
652
653 switch (now_char) {
654     case '=':
655     {
656         print(line_index, "operator", "^=");
657         pos = 0;
658         break;
659     }
660     default:
661     {
662         get_backward(forward);
663         print(line_index, "operator", "^");
664         pos = 0;
665         break;
666     }
667 }
668 break;
669 }
670
671 case ',':
672 {
673     // 累计分界符和操作符数目
674     operate_num ++;
675     state = 0;
676     pos = 0;
677     print(line_index, "operator", ",");
678     break;
679 }
680
681 case ';':
682 {
683     // 累计分界符和操作符数目
684     operate_num ++;
685     state = 0;
686     pos = 0;
687     print(line_index, "operator", ";");
688     break;
689 }
690
691 case '#':
692 {
693     // 累计分界符和操作符数目
694     operate_num ++;
695     state = 0;
696     pos = 0;
697     print(line_index, "operator", "#");
698     break;
699 }
700
701 case '.':
702 {
703     // 累计分界符和操作符数目
704     operate_num ++;
705     state = 0;
706     pos = 0;
```

```

707         print(line_index, "operator", ".");
708         break;
709     }
710
711     case '~':
712     {
713         // 累计分界符和操作符数目
714         operate_num ++;
715         state = 0;
716         pos = 0;
717         print(line_index, "operator", "~");
718         break;
719     }
720
721     // 剩下的就是数字或者word（变量名或关键字）
722     default:
723     {
724         // 数字
725         if (is_num(now_char)){
726             str_in_file[pos] = now_char;
727             pos ++;
728             number_num ++;
729             // 将状态转为2，往下查找整个数字
730             state = 2;
731         }
732
733         else{
734             // 变量名或关键字
735             if (is_letter(now_char) || now_char == '_'){
736                 str_in_file[pos] = now_char;
737                 pos ++;
738                 word_num ++;
739                 // 将状态转为1，往下查找整个word
740                 state = 1;
741             }
742         }
743         break;
744     }
745 }
746
747 break;
748 }
749
750 // 处理word
751 case 1:
752 {
753     // 不读完就不停
754     while (1) {
755         now_char = get_char(forward);
756         // 因为第一个已经确定是字母，所以接下来的可以是字母、数字或者下
757         // 划线
758         if (is_num(now_char) || is_letter(now_char) ||
now_char == '_'){
759             str_in_file[pos] = now_char;
760             pos ++;
761         }
762         // 如果出现了空格等别的字符
763         else {

```

```

763 // 给当前要输出的字符串加上结束符
764 str_in_file[pos] = '\0';
765 // 判断该word是不是关键字
766 bool keyword_res = is_keyword(str_in_file);
767 // 如果是关键字
768 if (keyword_res) {
769     print(line_index, "keyword", str_in_file);
770     // word_num 已经加过了 这里只加 keyword_num
771     keyword_num ++;
772 }
773 // 不是关键字
774 else {
775     print(line_index, "variable", str_in_file);
776     // word_num 已经加过了 这里只加 variable_num
777     variable_num ++;
778 }
779
780 // 处理完毕 进行reset
781 // 这个字符不是属于要输出的字符串之内的，回退
782 get_backward(forward);
783 pos = 0;
784 state = 0;
785 break;
786 }
787 }
788 break;
789 }
790
791
792
793 // 处理数字
794 case 2:
795 {
796     // 同样不读完不停
797     while (1) {
798         now_char = get_char(forward);
799
800         // 如果是数字 继续读
801         if (is_num(now_char)){
802             str_in_file[pos] = now_char;
803             pos ++;
804         }
805
806         // 如果下一个不是数字
807         else {
808
809             // 可能是小数
810             if (now_char == '.'){
811                 str_in_file[pos] = now_char;
812                 pos ++;
813                 // 切换到小数状态
814                 state = 3;
815                 // 退出while
816                 break;
817             }
818
819             // 有可能是科学计数法 如1e-3
820             else if (now_char == 'e' || now_char == 'E'){

```

```

821         str_in_file[pos] = now_char;
822         pos ++;
823         // 切换到科学计数法状态
824         state = 4;
825         // 退出while
826         break;
827     }
828
829     // 不是上述的任何符号 则结束数字模式
830     else {
831         // 补终结符
832         str_in_file[pos] = '\0';
833         print(line_index, "integral number",
str_in_file);
834
835         // 该字符没输出 需要回退
836         get_backward(forward);
837
838         // reset
839         pos = 0;
840         state = 0;
841         // 退出while
842         break;
843     }
844 }
845 }
846 break;
847 }
848
849
850 // 小数状态
851 case 3:
852 {
853     while (1) {
854         now_char = get_char(forward);
855         // 是数字
856         if (is_num(now_char)) {
857             str_in_file[pos] = now_char;
858             pos ++;
859         }
860
861         // eg:1.5e-3 转移到科学计数法状态
862         else if (now_char == 'e' || now_char == 'E') {
863             str_in_file[pos] = now_char;
864             pos ++;
865             state = 4;
866             break;
867         }
868
869         // 不是数字或e
870         else {
871             str_in_file[pos] = '\0';
872             print(line_index, "fraction number", str_in_file);
873             // 该字符没输出 需要回退
874             get_backward(forward);
875             // reset
876             pos = 0;
877             state = 0;

```

```

878             // 退出while
879             break;
880         }
881     }
882     break;
883 }
884
885 // 科学计数状态(无负号)
886 case 4:
887 {
888     while (1) {
889
890         now_char = get_char(forward);
891         // 数字
892         if (is_num(now_char)) {
893             str_in_file[pos] = now_char;
894             pos ++;
895         }
896
897         // 负号 科学计数法的负次幂
898         else if (now_char == '-') {
899             str_in_file[pos] = now_char;
900             pos ++;
901             state = 5;
902             break;
903         }
904
905         // 结束幂次正数的科学计数法
906         else {
907             str_in_file[pos] = '\0';
908             print(line_index, "scientific notation number (pos
909 power)", str_in_file);
910             // 该字符没输出 需要回退
911             get_backward(forward);
912             // reset
913             pos = 0;
914             state = 0;
915             // 退出while
916             break;
917         }
918     }
919     break;
920 }
921
922 // 科学技术状态 负幂次
923 case 5:
924 {
925     while (1) {
926         now_char = get_char(forward);
927         // 数字
928         if (is_num(now_char)) {
929             str_in_file[pos] = now_char;
930             pos ++;
931         }
932
933         // 结束科学计数法
934         else {
935             str_in_file[pos] = '\0';

```

```

935         print(line_index, "scientific notation number (neg
power)", str_in_file);
936         // 该字符没输出 需要回退
937         get_backward(forward);
938         // reset
939         pos = 0;
940         state = 0;
941         // 退出while
942         break;
943     }
944 }
945 break;
946 }
947
948 // 多行注释 已经检索到/*      寻找*
949 case 6:
950 {
951     while (1) {
952         now_char = get_char(forward);
953         // 如果该行已经遍历完毕
954         if (now_char == '\0'){
955             // 设置多行注释标记
956             multi_annotation = 1;
957             break;
958         }
959
960         // 找到 *
961         else if (now_char == '*') {
962             // 寻找 /
963             state = 7;
964             pos = 0;
965             break;
966         }
967
968         // 没找到就继续找
969         else {
970             continue;
971         }
972     }
973     break;
974 }
975
976 // 多行注释 已经检索到/* *      寻找/
977 case 7:
978 {
979     now_char = get_char(forward);
980
981     // 已经检索完多行注释
982     if (now_char == '/') {
983         // 直接跳过 回到状态0
984         multi_annotation = 0;
985         cout << "have a multi-annotation skipping" << endl;
986         state = 0;
987         pos = 0;
988         break;
989     }
990
991     // 该行已经遍历完毕（存在这种情况,因为是从case 6 过来的)

```

```

992         // 出去之后该函数会结束 读下一行 所以不能设置state = 6 还是得用标记
993         else if (now_char == '\0') {
994             // 设置多行注释标记
995             multi_annotation = 1;
996             break;
997         }
998
999         // 再接收到*
1000        else if (now_char == '*') {
1001            // 仍然为state = 7
1002            state = 7;
1003            pos = 0;
1004            break;
1005        }
1006
1007        // 该行没结束 且不是/ 则回到状态6
1008        else {
1009            state = 6;
1010            pos = 0;
1011            break;
1012        }
1013        break;
1014    }
1015
1016    // 不在任何情况内 抛出一个错误
1017    default:
1018    {
1019        print(line_index, "error! no state is matching.", "-");
1020        break;
1021    }
1022
1023    }
1024    }
1025 }
1026
1027
1028
1029 int main()
1030 {
1031     // 打开文件
1032     FILE *file = fopen("s.txt", "r");
1033     // 标记当前行数
1034     int line_index = 0;
1035
1036     // 为了标记行,选择一行行输入
1037     while (1){
1038         char *now_line_str = fgets(buffer, 10000, file);
1039
1040         if (now_line_str != NULL){
1041             line_index ++;
1042             lexical_analysis(line_index, buffer);
1043         }
1044         else break;
1045     }
1046
1047     // 如果多行注释始终没有配对成功 则报错
1048     if (multi_annotation == 1) {

```

```

1049     print(line_index, "error: multi-annotation compared
1050         unsuccessfully", "-");
1051     }
1052     cout << endl;
1053     cout << endl;
1054     cout << "行数:" << line_index << endl;
1055     cout << "字符总数:" << sign_num << endl;
1056     cout << "单词总数:" << word_num << endl;
1057     cout << "关键字总数:" << keyword_num << endl;
1058     cout << "变量名数目:" << variable_num << endl;
1059     cout << "数学常量数目:" << number_num << endl;
1060     cout << "单行注释数目:" << single_ann_num << endl;
1061     cout << "多行注释数目:" << multi_ann_num << endl;
1062     cout << "分界符和操作符数目(包括注释):" << operate_num << endl;
1063     cout << "字符串常量数目(包括printf内的输出):" << str_num << endl;
1064 }
1065

```

五、方法一测试

寻找了一段判断素数的C语言代码，并对其增加了一些测试的必要语句，如一些注释等。

① 测试代码：（完全正确）

```

1  /**
2   * use for lex-test.
3   */
4
5  #include <math.h>
6  #include <stdio.h>
7  #define harry 916
8
9  //this is the program
10 int main()
11 {
12     int m, flag;
13     scanf("%d", &m);
14     flag = 1;
15     for(int i = 2; i <= sqrt(m); i++){
16         if(m%i == 0)
17         {
18             flag = 0;
19             break;
20         }
21     }
22     if(flag){
23         //output
24         printf("%d is a primer.\n", m);
25     }
26     else{
27         //output
28         printf("%d is not a primer.\n", m);
29     }
30     return 0;

```


输出:

```

1  have a multi-annotation skipping
2  line 5 <operator, #>
3  line 5 <variable, include>
4  line 5 <operator, <>
5  line 5 <variable, math>
6  line 5 <operator, .>
7  line 5 <variable, h>
8  line 5 <operator, >>
9  line 6 <operator, #>
10 line 6 <variable, include>
11 line 6 <operator, <>
12 line 6 <variable, stdio>
13 line 6 <operator, .>
14 line 6 <variable, h>
15 line 6 <operator, >>
16 line 7 <operator, #>
17 line 7 <variable, define>
18 line 7 <variable, harry>
19 line 7 <integral number, 916>
20 line 9 have a single annotation skipping
21 line 10 <keyword, int>
22 line 10 <variable, main>
23 line 10 <delimiter, (>
24 line 10 <delimiter, )>
25 line 11 <delimiter, {>
26 line 12 <keyword, int>
27 line 12 <variable, m>
28 line 12 <operator, ,>
29 line 12 <variable, flag>
30 line 12 <operator, ;>
31 line 13 <variable, scanf>
32 line 13 <delimiter, (>
33 line 13 <delimiter, ">
34 line 13 <string, %d>
35 line 13 <delimiter, ">
36 line 13 <operator, ,>
37 line 13 <operator, &>
38 line 13 <variable, m>
39 line 13 <delimiter, )>
40 line 13 <operator, ;>
41 line 14 <variable, flag>
42 line 14 <operator, =>
43 line 14 <integral number, 1>
44 line 14 <operator, ;>
45 line 15 <keyword, for>
46 line 15 <delimiter, (>
47 line 15 <keyword, int>
48 line 15 <variable, i>
49 line 15 <operator, =>
50 line 15 <integral number, 2>
51 line 15 <operator, ;>

```

```
52 line 15 <variable, i>
53 line 15 <operator, <=>
54 line 15 <variable, sqrt>
55 line 15 <delimiter, (>
56 line 15 <variable, m>
57 line 15 <delimiter, )>
58 line 15 <operator, ;>
59 line 15 <variable, i>
60 line 15 <operator, ++>
61 line 15 <delimiter, )>
62 line 15 <delimiter, {>
63 line 16 <keyword, if>
64 line 16 <delimiter, (>
65 line 16 <variable, m>
66 line 16 <operator, %>
67 line 16 <variable, i>
68 line 16 <operator, ==>
69 line 16 <integral number, 0>
70 line 16 <delimiter, )>
71 line 17 <delimiter, {>
72 line 18 <variable, flag>
73 line 18 <operator, ==>
74 line 18 <integral number, 0>
75 line 18 <operator, ;>
76 line 19 <keyword, break>
77 line 19 <operator, ;>
78 line 20 <delimiter, }>
79 line 21 <delimiter, }>
80 line 22 <keyword, if>
81 line 22 <delimiter, (>
82 line 22 <variable, flag>
83 line 22 <delimiter, )>
84 line 22 <delimiter, {>
85 line 23 have a single annotation skipping
86 line 24 <variable, printf>
87 line 24 <delimiter, (>
88 line 24 <delimiter, ">
89 line 24 <string, %d is a primer.\n>
90 line 24 <delimiter, ">
91 line 24 <operator, ,>
92 line 24 <variable, m>
93 line 24 <delimiter, )>
94 line 24 <operator, ;>
95 line 25 <delimiter, }>
96 line 26 <keyword, else>
97 line 26 <delimiter, {>
98 line 27 have a single annotation skipping
99 line 28 <variable, printf>
100 line 28 <delimiter, (>
101 line 28 <delimiter, ">
102 line 28 <string, %d is not a primer.\n>
103 line 28 <delimiter, ">
104 line 28 <operator, ,>
105 line 28 <variable, m>
106 line 28 <delimiter, )>
107 line 28 <operator, ;>
108 line 29 <delimiter, }>
109 line 30 <keyword, return>
```

```

110 line 30 <integral number, 0>
111 line 30 <operator, ;>
112 line 31 <delimiter, }>
113
114
115 行数:31
116 字符总数:513
117 单词总数:36
118 关键字总数:9
119 变量名数目:27
120 数学常量数目:6
121 单行注释数目:3
122 多行注释数目:1
123 分界符和操作符数目(包括注释):67
124 字符串常量数目(包括printf内的输出):3

```

验证:

operator有31项。

```

> operator Aa .ab_* 第 1 项, 共 31 项 ↑ ↓ ≡ ×
1 have a multi-annotation skipping
2 line 5 <operator, #>
3 line 5 <variable, include>
4 line 5 <operator, <>
5 line 5 <variable, math>
6 line 5 <operator, .>
7 line 5 <variable, h>
8 line 5 <operator, >>
9 line 6 <operator, #>
10 line 6 <variable, include>
11 line 6 <operator, <>
12 line 6 <variable, stdio>
13 line 6 <operator, .>
14 line 6 <variable, h>
15 line 6 <operator, >>
16 line 7 <operator, #>
17 line 7 <variable, define>
18 line 7 <variable, harry>
19 line 7 <integral number, 916>
20 line 9 have a single annotation skipping
21 line 10 <keyword, int>
22 line 10 <variable, main>
23 line 10 <delimiter, (>
24 line 10 <delimiter, )>
25 line 11 <delimiter, {>
26 line 12 <keyword, int>
27 line 12 <variable, m>
28 line 12 <operator, ,>
29 line 12 <variable, flag>
30 line 12 <operator, ;>
31 line 13 <variable, scanf>
32 line 13 <delimiter, (>
33 line 13 <delimiter, ">
34 line 13 <string, %d>
35 line 13 <delimiter, ">
36 line 13 <operator, ,>
37 line 13 <operator, &>
38 line 13 <variable, m>
39 line 13 <delimiter, )>
40 line 13 <operator, ;>
41 line 14 <variable, flag>

```

delimiter有32项。

```
1 have a multi-annotation skipping
2 line 5 <operator, #>
3 line 5 <variable, include>
4 line 5 <operator, <>
5 line 5 <variable, math>
6 line 5 <operator, .>
7 line 5 <variable, h>
8 line 5 <operator, >>
9 line 6 <operator, #>
10 line 6 <variable, include>
11 line 6 <operator, <>
12 line 6 <variable, stdio>
13 line 6 <operator, .>
14 line 6 <variable, h>
15 line 6 <operator, >>
16 line 7 <operator, #>
17 line 7 <variable, define>
18 line 7 <variable, harry>
19 line 7 <integral number, 916>
20 line 9 have a single annotation skipping
21 line 10 <keyword, int>
22 line 10 <variable, main>
23 line 10 <delimiter, (>
24 line 10 <delimiter, )>
25 line 11 <delimiter, {>
26 line 12 <keyword, int>
27 line 12 <variable, m>
28 line 12 <operator, ,>
29 line 12 <variable, flag>
30 line 12 <operator, ;>
31 line 13 <variable, scanf>
32 line 13 <delimiter, (>
33 line 13 <delimiter, ">
34 line 13 <string, %d>
35 line 13 <delimiter, ">
36 line 13 <operator, ,>
37 line 13 <operator, &>
38 line 13 <variable, m>
39 line 13 <delimiter, )>
40 line 13 <operator, ;>
41 line 14 <variable, flag>
```

注释有4项，所以分界符和操作符数量为67项，符合输出。

关键词有9项，符合输出。

```
1  have a multi-annotation skipping
2  line 5 <operator, #>
3  line 5 <variable, include>
4  line 5 <operator, <>
5  line 5 <variable, math>
6  line 5 <operator, .>
7  line 5 <variable, h>
8  line 5 <operator, >>
9  line 6 <operator, #>
10 line 6 <variable, include>
11 line 6 <operator, <>
12 line 6 <variable, stdio>
13 line 6 <operator, .>
14 line 6 <variable, h>
15 line 6 <operator, >>
16 line 7 <operator, #>
17 line 7 <variable, define>
18 line 7 <variable, harry>
19 line 7 <integral number, 916>
20 line 9 have a single annotation skipping
21 line 10 <keyword, int>
22 line 10 <variable, main>
23 line 10 <delimiter, (>
24 line 10 <delimiter, )>
25 line 11 <delimiter, {>
26 line 12 <keyword, int>
27 line 12 <variable, m>
28 line 12 <operator, ,>
29 line 12 <variable, flag>
30 line 12 <operator, ;>
31 line 13 <variable, scanf>
32 line 13 <delimiter, (>
33 line 13 <delimiter, ">
34 line 13 <string, %d>
35 line 13 <delimiter, ">
36 line 13 <operator, ,>
37 line 13 <operator, &>
38 line 13 <variable, m>
39 line 13 <delimiter, )>
40 line 13 <operator, ;>
41 line 14 <variable, flag>
```

经查证，其他数据也均符合输出。

② 测试代码：（28行双引号没有配对）

```
1  /**
2      use for lex-test.
3  **/
4
5  #include <math.h>
6  #include <stdio.h>
7  #define harry 916
8
9  //this is the program
10 int main()
11 {
12     int m,flag;
13     scanf("%d", &m);
14     flag = 1;
15     for(int i = 2;i <= sqrt(m);i++){
16         if(m%i == 0)
17         {
18             flag = 0;
19             break;
```

```

20     }
21 }
22 if(flag){
23     //output
24     printf("%d is a primer.\n", m);
25 }
26 else{
27     //output
28     printf("%d is not a primer.\n", m);
29 }
30 return 0;
31 }

```

输出：在102行抛出了异常

```

1  have a multi-annotation skipping
2  line 5 <operator, #>
3  line 5 <variable, include>
4  line 5 <operator, <>
5  line 5 <variable, math>
6  line 5 <operator, .>
7  line 5 <variable, h>
8  line 5 <operator, >>
9  line 6 <operator, #>
10 line 6 <variable, include>
11 line 6 <operator, <>
12 line 6 <variable, stdio>
13 line 6 <operator, .>
14 line 6 <variable, h>
15 line 6 <operator, >>
16 line 7 <operator, #>
17 line 7 <variable, define>
18 line 7 <variable, harry>
19 line 7 <integral number, 916>
20 line 9 have a single annotation skipping
21 line 10 <keyword, int>
22 line 10 <variable, main>
23 line 10 <delimiter, (>
24 line 10 <delimiter, )>
25 line 11 <delimiter, {>
26 line 12 <keyword, int>
27 line 12 <variable, m>
28 line 12 <operator, ,>
29 line 12 <variable, flag>
30 line 12 <operator, ;>
31 line 13 <variable, scanf>
32 line 13 <delimiter, (>
33 line 13 <delimiter, ">
34 line 13 <string, %d>
35 line 13 <delimiter, ">
36 line 13 <operator, ,>
37 line 13 <operator, &>
38 line 13 <variable, m>
39 line 13 <delimiter, )>
40 line 13 <operator, ;>

```

```
41 line 14 <variable, flag>
42 line 14 <operator, =>
43 line 14 <integral number, 1>
44 line 14 <operator, ;>
45 line 15 <keyword, for>
46 line 15 <delimiter, (>
47 line 15 <keyword, int>
48 line 15 <variable, i>
49 line 15 <operator, =>
50 line 15 <integral number, 2>
51 line 15 <operator, ;>
52 line 15 <variable, i>
53 line 15 <operator, <=>
54 line 15 <variable, sqrt>
55 line 15 <delimiter, (>
56 line 15 <variable, m>
57 line 15 <delimiter, )>
58 line 15 <operator, ;>
59 line 15 <variable, i>
60 line 15 <operator, ++>
61 line 15 <delimiter, )>
62 line 15 <delimiter, {>
63 line 16 <keyword, if>
64 line 16 <delimiter, (>
65 line 16 <variable, m>
66 line 16 <operator, %>
67 line 16 <variable, i>
68 line 16 <operator, ==>
69 line 16 <integral number, 0>
70 line 16 <delimiter, )>
71 line 17 <delimiter, {>
72 line 18 <variable, flag>
73 line 18 <operator, =>
74 line 18 <integral number, 0>
75 line 18 <operator, ;>
76 line 19 <keyword, break>
77 line 19 <operator, ;>
78 line 20 <delimiter, }>
79 line 21 <delimiter, }>
80 line 22 <keyword, if>
81 line 22 <delimiter, (>
82 line 22 <variable, flag>
83 line 22 <delimiter, )>
84 line 22 <delimiter, {>
85 line 23 have a single annotation skipping
86 line 24 <variable, printf>
87 line 24 <delimiter, (>
88 line 24 <delimiter, ">
89 line 24 <string, %d is a primer.\n>
90 line 24 <delimiter, ">
91 line 24 <operator, ,>
92 line 24 <variable, m>
93 line 24 <delimiter, )>
94 line 24 <operator, ;>
95 line 25 <delimiter, }>
96 line 26 <keyword, else>
97 line 26 <delimiter, {>
98 line 27 have a single annotation skipping
```

```

99 | line 28 <variable, printf>
100 | line 28 <delimiter, (>
101 | line 28 <delimiter, ">
102 | line 28 <error:the str end without another quotation mark, %d is not a
    | primer.\n, m);
103 | >
104 | line 29 <delimiter, }>
105 | line 30 <keyword, return>
106 | line 30 <integral number, 0>
107 | line 30 <operator, ;>
108 | line 31 <delimiter, }>
109 |
110 |
111 | 行数:31
112 | 字符总数:511
113 | 单词总数:35
114 | 关键字总数:9
115 | 变量名数目:26
116 | 数学常量数目:6
117 | 单行注释数目:3
118 | 多行注释数目:1
119 | 分界符和操作符数目(包括注释):63
120 | 字符串常量数目(包括printf内的输出):2

```

分析:

- 该词法分析程序能够识别用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；
- 可以识别并且跳过源程序的注释；
- 可以统计源程序的语句行数，各类单词的个数，字符总数，输出统计结果；
- 可以检查源程序中存在的词法错误，并报告错误所在的位置；
- 可以对出现的错误进行适当回复，继续进行词法分析。

上述要求都能按要求完成。

六、方法二分析

lex实际上就是根据所输入的正则表达式对相应类型的词语进行辨识，识别到相应的词语后根据输入的对应c语句进行操作或输出。

lex首先通过下列语句进行编译，

```
1 | flex lex.i
```

生成一个名为lex.yy.c的文件。再对该文件进行编译：

```
1 | gcc lex.yy.c -lflex
```

生成.out文件。执行该文件可以输出结果。

正则表达式由书中第三章推导而来。

七、方法二代码展示

```
1  %{
2  #include<stdlib.h>
3  #include<stdio.h>
4  %}
5
6  digit      [0-9]
7  letter     [a-zA-Z]
8  id         ({letter}|_){letter}|{digit}|_)*
9  integer    {digit}+
10 realnum    {digit}+"."{digit}+
11 operator   "+"|-|"*"|"/"|<|=|>|>=>|>>|<<|&|&&|"%"
12 delim      [\t\n\r\0\v\040]
13 empty      {delim}+
14 string     \"[^\"]*\"
15
16 %%
17 {id}        {printf("identifier: %s\n",yytext);}
18 {integer}   {printf("integer: %s\n",yytext);}
19 {realnum}   {printf("realnum: %s\n",yytext);}
20 {operator}  {printf("operator: %s\n",yytext);}
21 "("        {printf("left bracket: %s\n",yytext);}
22 ")"        {printf("right bracket: %s\n",yytext);}
23 "{"        {printf("left bracket: %s\n",yytext);}
24 "}"        {printf("right bracket: %s\n",yytext);}
25 ","        {printf("comma: %s\n",yytext);}
26 ":"        {printf("colon: %s\n",yytext);}
27 ";"        {printf("semicolon: %s\n",yytext);}
28 "#".*      {printf("head file: %s\n",yytext);}
29 {empty}     { }
30 {string}    {printf("string: %s\n",yytext);}
31 .          {printf("can't not identify: %s\n",yytext);}
32
33 %%
34 int main(int argc,char **argv)
35 {
36     yyin = fopen("./test.c", "r");
37     yylex();
38     return 0;
39 }
40 int yywrap()
41 {
42     return 1;
43 }
```

八、方法二测试

输入test.c进行测试:

```
1  #include <math.h>
2  #include <stdio.h>
3  int main()
4  {
5      int m,flag;
6      scanf("%d", &m);
```

```

7     flag = 1;
8     for(int i = 2; i <= sqrt(m); i++){
9         if(m%i == 0)
10            {
11                flag = 0;
12                break;
13            }
14    }
15    if(flag){
16        printf("%d is a primer.\n", m);
17    }
18    else{
19        printf("%d is not a primer.\n", m);
20    }
21    return 0;
22 }

```

测试结果:

```

1  head file: #include <math.h>
2  head file: #include <stdio.h>
3  identifier: int
4  identifier: main
5  left bracket: (
6  right bracket: )
7  left bracket: {
8  identifier: int
9  identifier: m
10 comma: ,
11 identifier: flag
12 semicolon: ;
13 identifier: scanf
14 left bracket: (
15 string: "%d"
16 comma: ,
17 operator: &
18 identifier: m
19 right bracket: )
20 semicolon: ;
21 identifier: flag
22 operator: =
23 integer: 1
24 semicolon: ;
25 identifier: for
26 left bracket: (
27 identifier: int
28 identifier: i
29 operator: =
30 integer: 2
31 semicolon: ;
32 identifier: i
33 operator: <=
34 identifier: sqrt
35 left bracket: (
36 identifier: m

```

```
37 right bracket: )
38 semicolon: ;
39 identifier: i
40 operator: +
41 operator: +
42 right bracket: )
43 left bracket: {
44 identifier: if
45 left bracket: (
46 identifier: m
47 operator: %
48 identifier: i
49 operator: ==
50 integer: 0
51 right bracket: )
52 left bracket: {
53 identifier: flag
54 operator: =
55 integer: 0
56 semicolon: ;
57 identifier: break
58 semicolon: ;
59 right bracket: }
60 right bracket: }
61 identifier: if
62 left bracket: (
63 identifier: flag
64 right bracket: )
65 left bracket: {
66 identifier: printf
67 left bracket: (
68 string: "%d is a primer.\n"
69 comma: ,
70 identifier: m
71 right bracket: )
72 semicolon: ;
73 right bracket: }
74 identifier: else
75 left bracket: {
76 identifier: printf
77 left bracket: (
78 string: "%d is not a primer.\n"
79 comma: ,
80 identifier: m
81 right bracket: )
82 semicolon: ;
83 right bracket: }
84 identifier: return
85 integer: 0
86 semicolon: ;
87 right bracket: }
88
```

九、实验总结与心得

通过本次实验，我深刻地认识到了词法分析对于程序编译的重要性。而对词法分析来说，状态的确定与相互的转换时十分重要的。没有前置的思考与统筹，是很难系统地写出一个结构化的程序结构的。

将词法与语法分析安排在同一遍中，词法分析作为语法分析的子程序时，避免了中间文件，省去了取送符号的工作，有利于编译程序的效率。而词法分析与语法分析以生产者和消费者的关系同步运行时，也能够提高效率。

将词法分析独立出来，让我们至少能够简化词法、语法分析中的一项任务。例如，如果一个语法分析器必须把空白字符和注释当做语法单元进行处理，那么它就会比那些假设空白字符和注释已经被词法分析器过滤掉的处理器复杂得多。如果我们正在设计一个新语言，将词法和语法分开考虑有助于我们得到一个更加清晰地的语言设计方案。同时可以提高编译器的效率：把词法分析器独立出来使我们能够使用专用于词法分析任务、不进行语法分析的技术。此外，我们可以使用专门的用于读取输入字符的缓冲技术来显著提高编译器的速度。还可以增强编译器的可移植性。输入设备相关的特殊性可以被限制在词法分析器中。通过实验，我深刻了解了上述益处。

我也尝试了使用lex进行词法分析。lex的完备性与便捷性令我惊讶。这更加吸引我对词法分析进行更加深入的学习。