

作业题目

C语言的语法分析器

基本信息

姓名	班级	学号	专业	指导老师
杨元睿	2019211307	2019211447	计算机科学与技术	张玉洁

运行环境

编译器	语言	标准	平台
visual studio 2019	c++	c++11或者以上	win10

实验要求

- 可以识别出用C语言编写的源程序中的每个单词符号。并且以记号形式输出每个单词符号。

- 可以跳过注释。
- 可以统计程序中的语句行数，各类单词的个数，以及字符总数。
- 检查源程序中的词法错误，并且可以报告错误的位置。
- 对错误适当的恢复。可以让词法分析继续进行。
- 一次扫描即可。

我的理解(假设)

关于C语言的说明

1. 标识符。标识符是以 `_` 或者字母开头的一串字符串。其实是有长度限制的。但是标准已经将长度大大加长，并且不同的编译器也不一样。所以这里我就不提出警告了。
2. 数字。数字是以 **整数部分+小数部分+指数部分** 构成的。后面会详细说明。
3. 关键字。我这里默认C语言中有32个关键字。
4. 界符。也可以叫做标点符号。主要有 `() [] ; : { } ' "` 这几种。
5. 注释。 `/* */` 和 `/**`。其中 `/**` 要配合 `\n` 来判断。是不是结束了。
6. 分隔符。 `' '`、`'\t'`、`'\n'`。一定要注意不是只有空格是分隔符。遇到其他的也要做处理。
7. 运算符。
 1. 算数运算符。 `+` `-` `*` `/` `%` `++` `--`
 2. 逻辑运算符。 `&&` `||` `!`
 3. 位操作运算符。 `&` `|` `~` `^` `>>` `<<`
 4. 赋值运算符。 `=` `+=` `-=` `*=` `/=` `%=` `&=` `|=` `~=` `^=` `>>=` `<<=`
 5. 关系运算符。 `>` `<` `==` `<=` `>=` `!=`
 6. 一些要考虑的特殊的。 `->` `.` 这两个是结构体结构体指针用的。

关于错误处理的说明

- 本程序是词法分析。所以能够处理的错误是：单词拼写错误比如 `int a = 0xGG`。非法字符比如程序里有 `@` 这些非法字符。
- 其他错误不在我的处理范围。一开始我也想错了。比如语句之后有没有 `;`。这个首先要识别是不是是个句子，已经超出了语法分析的范畴了。

对于预编译命令

- 这里我认为是属于预处理器的阶段，并不由词法分析来管。
- 我的处理是一旦读到 `#` 就进入处理，然后读到 `"\n"` 就结束。

TOKEN的形式

- < 种别码, 属性值 >
- 很多词是我是一词一码的。比如 `while` `[` 等等。输出形式：< `while` , `-` >
- 赋值号和比较符号我是一类一码。比如 `+=` `<` 等等。输出形式：< `assign` , `=` > < `relop` , `LT` >

关于识别数字

数字有三种形式：

1. 10进制的数。这个不仅仅包含整数。也包含小数和指数。比如 `123.123e10`。注意：经过我直接在编译器的里的测试 `.123` `123.e10` 其实都是可以编译通过的。也就是说 `.` 前面可以没有字符，点后面也没有字符。
2. 8进制数。以0开头。比如 `0123`
3. 16进制数。以0x开头。比如 `0xffff`
4. 引入了8进制之后就会有这种情况。 `00123.123e123`。那么这个到底是算作错误的8进制数，还是算成有前导0的十进制数呢？我在编译器里面测试结果如下。

```
test.c x
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     float a = 00000022;
7     printf("%.1f\n",a);
8     float b = 00000022.2;
9     printf("%.1f\n",b);
10    return 0;
11 }
```

可以看到如果没有小数点那么就是8进制。如果有就是十进制

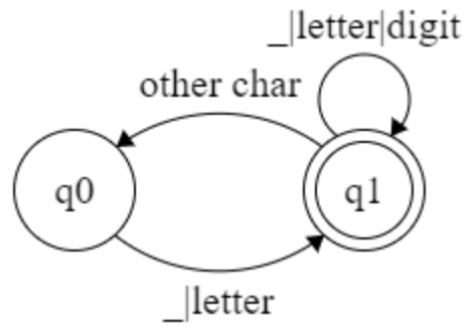
```
D:\Study\0_北邮课程\北邮大三上\北邮大三上编译原理\作业\c语言词法分析器\kit/
18.0
22.2
```

错误

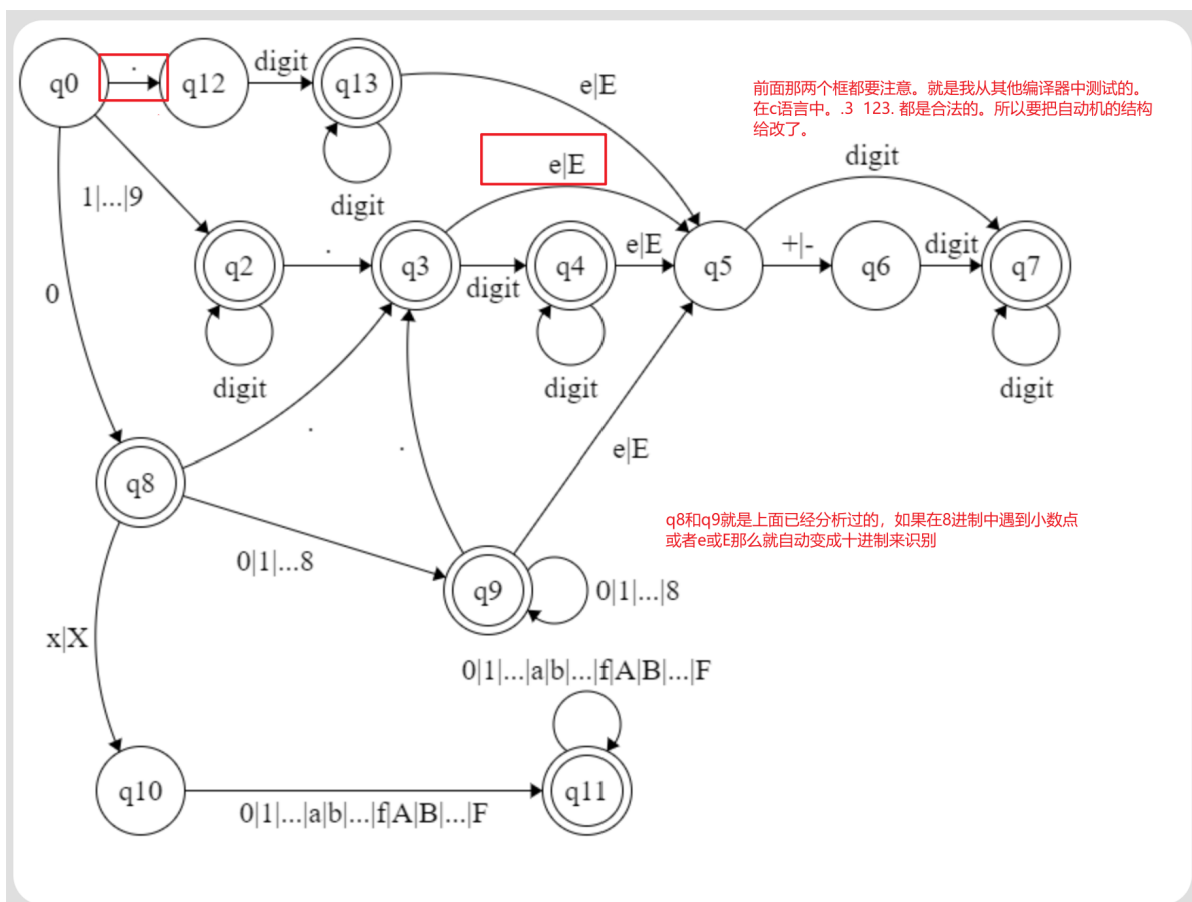
0. 标识符不能以数字开头
1. 有不合法的字符
2. 数字形态不正确
3. 注释没有闭合
4. 字符没有闭合
5. 字符长度太长
6. 字符串没有闭合
7. 字符串里有非法字符
8. 字符为空 `''` 这种就是不合法的（一开始没有想到，但是要注意字符串可以为空 `""`）

我的设计与实现

识别标识符的自动机



识别数字的自动机



识别数字是最困难的一步。费了很多心思。**PPT**上关于识别数字的过程并不完整。尤其要考虑**C**语言有**8**进制和**16**进制的区别。

识别预处理 字符 注释 字符串的自动机

对应的程如下图，只截图了部分。就不一一列举了。

```
case '*' : case '%' : case '=' : case '!' : case '~' :  
case '^' :  
{ ... }  
// + ++ +=  
case '+' :  
{ ... }  
// - -> -- -=  
case '-' :  
{ ... }  
// & && &=  
case '&' :  
{ ... }  
// | | | |=  
case '|' :
```

关于自动机的补充

如果某个状态读入了一个字符。这个字符不在接下来任何的转译路径上。那么此时就可以进行一些操作，最终转换到 `state0`。

比如 `123abc` 他会在读到 `a` 的时候发现是非法的。然后 **指针回退**。先把 `123` 是个数字识别出来。然后进入 `state0` 之后接着识别 `abc`。会再把 `abc` 识别出来。但显然 `123abc` 是一个不合法的格式。所以还会进入错误处理程序报错。

很显然我们还有一种做法。那就是把 `123abc` 直接抛弃。然后报错。进入 `state0`。

我认为这两种做法都是可以的。因为无论怎么样最终都报错了。所以其实进入一些错误情况的时候，会有不同的操作的。我选择的是第一种方法。

处理错误的一些转移我没有在上面自动机的图片里标出。因为基本都是转移到 `state0`。

符号表的设计

下面的表格中。第一个是指哈希表。并不是表里的属性。只是做一个索引用。

例如 `symbol["yyr"]`。后面的才是符号表的内容。

hash	name(名字)	property(属性)	class	type	value
	yyr	id	-	-	-

我们可以发现。后三个实际上在词法分析阶段都是不能填写的。`class` 其实是看它的作用域在哪里。`type` 是指出它的类型 `int` `ptr` 还是什么。`value` 是指这个的值。这个其实都要通过对于句子的识别来了。比如 `int yyr = 3`。这个显然不是我们词法分析要做的。

所以在词法分析里，为了方便其实最终就只有两项就好了。但是如果为了后面和其他的对接，应该要保留后面的所有项目。但好像老师说过是独立的。所以就先两项。

关于缓冲区

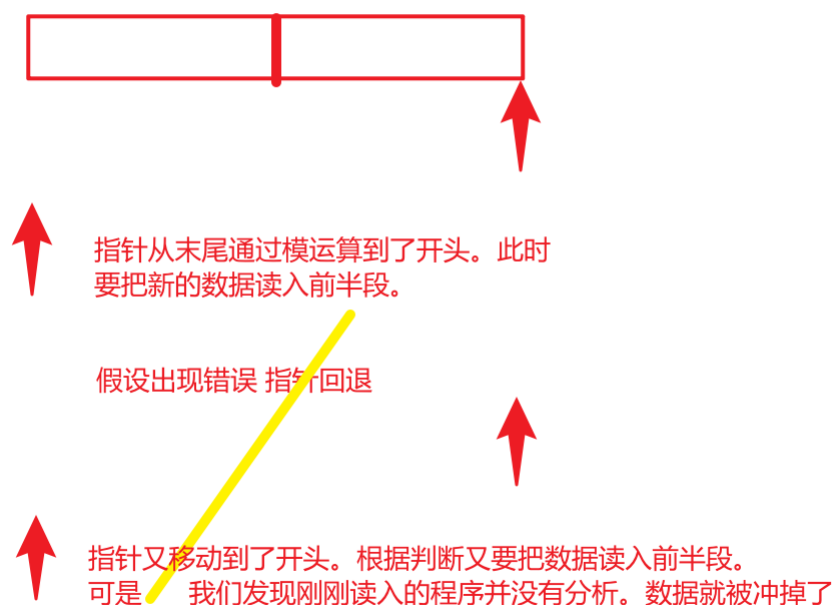
虽然现在缓冲区都够用了。但是为了预防万一我还是实现一下。

其实分析的时候直接用 `getline` 读取每一行就好了。但是如果真的有人恶意写的程序一直都是。一直打 `aaaaa`。打个几个GB的。这样就会崩溃。

我采用的方法和书上相同。就是把一个缓冲区拆成两块。但是我采用了更简便的方法。就是设置了 `flag` 的标识位。

```
1  if (this->pointer2 == 0 && flag==0)
2      {
3          cnt = fread(this->buffer, sizeof(char), hBLength,
4          this->cProgram);
5          if (cnt < hBLength) buffer[cnt] = -1; //添加结束符号 因为
6          fread不会把-1读入
7          flag = 1;
8      }
9
10     if (this->pointer2 == hBLength && flag == 1)
11     {
12         cnt = fread(this->buffer + hBLength, sizeof(char),
13         hBLength, this->cProgram);
14         if (cnt < hBLength) buffer[cnt+hBLength] = -1; //添加结
15         束符号 因为fread不会把-1读入
16         flag = 0;
17     }
```


这样做是为了防止指针回退的时候重复判断。多读入一次。把还未分析的数据冲掉。



设置 `flag` 就可以让读入的时候比如一次前 一次后 一次前 一次后 交互执行。不允许出现两次读入到前半段或者后半段的时候。

关于指针回退(一开始我不知道为什么要回退，现在明白了)

之前一直提到指针回退的概念。一开始并不知道指针回退是干嘛的。

比如 `a+b`。当我们识别到 `+` 之后现在组成的字符是 `a+`。这显然不符合条件。但是此时 `+` 已经被读入了。如果继续识别。那么直接识别的是 `b`。很显然我们会把 `+` 吞了。所以当识别到 `a+` 的时候。我们要让指针回退。然后状态回到 `state0`。接着从 `state0` 再次读取字符。那么读取的就是 `+` 号了。可以识别出 `+` 号了。

这种处理方法很简单也很高效。

部分实现代码展示与说明

类里面的函数和数据结构

```
1  class analyzer
2  {
3  public:
4      analyzer();//构造函数
5      ~analyzer();//析构函数
6      bool getTheCProgram();
7      bool iniKeyWords();//初始化哈希表
8      bool inierrorType();//初始化错误类型
9      bool readFileToBuffer();//读入到缓冲区 这个函数后来没用 因为直接
集成到 手写的 getChar()里
10     void changeState();//改变state的状态
11     bool canMakeId(char c);//可不可以是标识符
12     bool canMake8Base(char c);//可不可以生成8进制
13     bool canMake16Base(char c);//可不可以生成16进制
14     bool isDigit(char c);//是不是数字
15     bool isLetter(char c);//是不是数字
16     bool isUnderline(char c);//是不是下划线
17     void dealError(string info="");//根据错误表打印错误
18     char getChar();//自己手写的配合缓冲区切换的getchar
19     void fallBackPoint();//回退指针 利用 mod运算
20     void printResult(string info="");//打印每个识别到的单词 根据
info传类别不同
21     void printSum();//打印汇总信息 统计多少行 多少单词 符号
22 public:
23     FILE* cProgram; //输入C程序文件指针
24     FILE* resultFile;//输出记号流的文件指针
25     FILE* sumFile;//输出汇总信息的文件指针
26     FILE* errorFile;//输出错误信息的文件指针
27     FILE* errorType;//只读文件 用于读取错误的种类
28     char buffer[bLength];//指针
29     int pointer1;//第一个指针
30     int pointer2;//第二个指针
31     bool willEnd;//提示读取的文件是不是将要结束了 后来没用到
32     int lineNum;//实际行号
33     int pLineNum;//预测行号
34     int wordNum;//单词的数量
35     unordered_map<string, pair<string,string>> keyWords;//关键
字的哈希表
36     unordered_map<string, int> numOfWords;//所有字符的哈希表
```

```

37     unordered_map<string, pair<string, string>>
symbolTable; //id的符号表 删减版 详细见实验报告
38     vector<string> errorVector; //错误表
39     int state; //状态
40     string token; //已经扫过的一个单词
41     bool flag = 0; //用于切换缓冲区的服务
42 };

```

最主要的函数

整体结构

最主要的函数是 `void changeState()` 函数。也是代码的关键所在。

这个函数就是一直读入字符然后来不停地状态转换和分析。

采取的框架仍然是利用 `switch case` 来实现。以下是部分代码。

```

1     char ch = 0;
2     while (1)
3     {
4         if (ch == -1) break;
5         switch (this->state)
6         {
7             case 0: //初始状态
8             {
9                 //不断地读取字符 但是遇到空格和tab要忽略
10                token.clear();
11                while (1)
12                {
13                    ch = this->getChar();
14                    this->lineNum = this->pLineNum;
15                    if (ch != ' ' && ch != '\t') break; //如果不是
空格和tab就可以进入下一步 如果是就一直重复读取
16                }
17                //识别字符
18                switch (ch)
19                {

```

整体是一个无限循环。只有碰到 **EOF** 也就是 **-1**。才会停止。里面就是先针对状态的 **switch**。接着就是状态里面根据不同字符的判断 用 **if** 或者 **switch**。

细节之错误处理和正常识别

```
1      else
2      {
3          state = 0; //转换到初始状态
4          this->fallBackPoint(); //回退指针
5          if (this->keyWords.count(this->token)) //判断是
           不是在关键字表里
6      {
7          this->printResult();
8      }
9      else
10     {
11         this->printResult("id");
12     }
13 }
```

以上是一段正常识别的例子。这里我们可以看到。正常识别也要回退指针。比如 **abc+def**。如果一个单词没有边界。肯定是无法正常识别的。也就是说碰壁了。就可以识别了。识别 **abc** 的时候遇到了 **+** 号。表明 **abc** 是一个标识符。结束了。但是 **+** 仍然要被识别。所以要回退指针。状态回到0。

```
1      else if(isLetter(ch) || isUnderline(ch))
2      {
3          state = 0;
4          fallBackPoint();
5          dealError("0");
6          //加入
7          printResult("dec");
8      }
```

这是一段错误的例子。这里我们可以看到。相比正常识别。它会多了一步。**dealError**。比如 **0123abc**。当 **0123** 识别到 **a** 的时候就结束了。虽然 **abc** 可以继续被识别。但是这种数字开头的方式是不会被允许的。所以要报错。报不报错是由当前状态和判断所决定的。

输出

我是以文件的形式来输出的。

```
1     else if (info == "id")
2     {
3         fprintf(resultFile, "line : %d < %s , %s > \n", this->lineNum, "id", token.c_str());
4         this->numOfWords["id"]++;
5         if (this->symbolTable.count(token) == 0)
6         {
7             this->symbolTable[token] = { "id", token };
8         }
9     }
```

上述这段代码可以看到。我输出到了文件中。并且输出之后把它存到了符号表里。注意：由于我使用了哈希表来优化。所以标识符的输出和存储和一般不同。

a	b	c	d
---	---	---	---

如果是这种方式那么输出应该是。 `<id,0> <id,1> <id,2>` 等等。因为虽然是a。但是它其实是符号表的第0项。由于是线性表。所以必须要后面有个数是入口指针。否则从头到尾便利实在太慢。

但是我采用的是。

`unordered_map<string,>>` 直接就把一个字符串映射到了某个地址。那么此时就可以输出 `<id,a> <id,b>` 等等了。字符串就是它的符号表的索引。

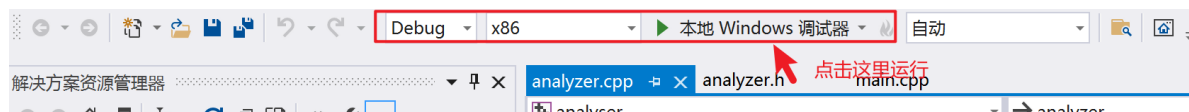
程序使用方法以及样例测试

程序使用方法

.vs	2021/9/24 15:58	文件夹
analyser	2021/10/9 13:32	文件夹
Debug	2021/10/8 18:44	文件夹
Release	2021/10/8 18:28	文件夹
x64	2021/10/8 23:02	文件夹
analyser.sln	2021/9/24 15:58	Visual Studio Soluti...

点击这个项目文件

接下来会进入程序的主界面



本次词法分析开始！
请输入测试样例的文件名o(^▽^)o

本次词法分析开始！
请输入测试样例的文件名o(^▽^)o
test4.c
本次词法分析用时 1.136000 秒
本次词法分析结束了！ \ (≧◇≦) /

运行之后会弹出这个对话框代表结束了。接下来我们查看结果。

.vs	2021/9/24 15:58	文件夹
analyser	2021/10/9 13:32	文件夹
Debug	2021/10/8 18:44	文件夹
Release	2021/10/8 18:28	文件夹
x64	2021/10/8 23:02	文件夹
analyser.sln	2021/9/24 15:58	Visual Studio Soluti...

名称	修改日期	类型	大小
Debug	2021/10/10 20:10	文件夹	
Release	2021/10/8 18:28	文件夹	
x64	2021/10/8 23:02	文件夹	
analyser.vcxproj	2021/10/8 23:02	VC++ Project	8 KB
analyser.vcxproj.filters	2021/10/6 15:28	VC++ Project Filter...	2 KB
analyser.vcxproj.user	2021/10/2 16:34	Per-User Project O...	1 KB
analyzer.cpp	2021/10/10 20:13	C++ Source File	21 KB
analyzer.h	2021/10/10 20:13	C Header File	2 KB
error.txt 这个关于错误的输出结果	2021/10/10 20:13	文本文档	0 KB
errorType.txt 这个是错误表 作为文件读入 不要修改	2021/10/9 13:29	文本文档	1 KB
keyWords.txt 这个是关键表 作为文件读入 不要修改	2021/10/3 17:02	文本文档	1 KB
main.cpp	2021/10/10 20:10	C++ Source File	1 KB
result.txt 这个是每行统计的输出结果	2021/10/10 20:13	文本文档	6,129 KB
sum.txt 这个是汇总的输出结果	2021/10/10 20:13	文本文档	99 KB
test1.c 这个是测试样例1	2021/10/10 19:45	C Source File	1 KB
test2.c 这个是测试样例2	2021/10/10 19:46	C Source File	1 KB
test3.c 这个是测试样例3	2021/10/10 19:46	C Source File	1 KB
test4.c 这个是测试样例4	2021/10/10 19:57	C Source File	1,998 KB
test5.c 这个是测试样例5	2021/10/10 19:48	C Source File	1,998 KB

样例测试

1. 普通小程序测试+混乱排版

test1.c

```
1  # include <stdio.h>
2  # include          <stdlib.h>
3
4  int main(void)
5  {
6      int a;
7          int b;
8      a = 100;
9      a = .1e3;
10     int c = 3;
11     printf("%d",a+b);
12         if(a)
13         {
14             while(a--)
15             {
16                 ++b;
17                 ++c;
18             }
19             if(b <= c)
20             {
21                 printf("%s", "Cowboy Bebop");
22             }
23         }
24     }
25
26     switch(a)
27     {
28         case 3 :
29         case 7 :
30         default :
31             break;
32     }
33
34     return 0;
35 }
36
```

error.txt

无结果 因为无词法错误

result.txt

```
1 line : 4 < int , - >
2 line : 4 < id , main >
3 line : 4 < ( , - >
4 line : 4 < void , - >
5 line : 4 < ) , - >
6 line : 5 < { , - >
7 line : 6 < int , - >
8 line : 6 < id , a >
9 line : 6 < ; , - >
10 line : 7 < int , - >
11 line : 7 < id , b >
12 line : 7 < ; , - >
13 line : 8 < id , a >
14 line : 8 < assign , = >
15 line : 8 < dec , 100 >
16 line : 8 < ; , - >
17 line : 9 < id , a >
18 line : 9 < assign , = >
19 line : 9 < dec , .1e3 >
20 line : 9 < ; , - >
21 line : 10 < int , - >
22 line : 10 < id , c >
23 line : 10 < assign , = >
24 line : 10 < dec , 3 >
25 line : 10 < ; , - >
26 line : 11 < id , printf >
27 line : 11 < ( , - >
28 line : 11 < str , %d >
29 line : 11 < , >
30 line : 11 < id , a >
31 line : 11 < + , - >
32 line : 11 < + , - >
33 line : 11 < id , b >
```



```
34 line : 11 < ) , - >
35 line : 11 < ; , - >
36 line : 12 < if , - >
37 line : 12 < ( , - >
38 line : 12 < id , a >
39 line : 12 < ) , - >
40 line : 13 < { , - >
41 line : 14 < while , - >
42 line : 14 < ( , - >
43 line : 14 < id , a >
44 line : 14 < , >
45 line : 14 < ) , - >
46 line : 15 < { , - >
47 line : 16 < , >
48 line : 16 < id , b >
49 line : 16 < ; , - >
50 line : 17 < , >
51 line : 17 < id , c >
52 line : 17 < ; , - >
53 line : 18 < } , - >
54 line : 19 < if , - >
55 line : 19 < ( , - >
56 line : 19 < id , b >
57 line : 19 < relop , LE >
58 line : 19 < id , c >
59 line : 19 < ) , - >
60 line : 20 < { , - >
61 line : 21 < id , printf >
62 line : 21 < ( , - >
63 line : 21 < str , %s >
64 line : 21 < , >
65 line : 21 < str , Cowboy Bebop >
66 line : 21 < ) , - >
67 line : 21 < ; , - >
68 line : 22 < } , - >
69 line : 24 < } , - >
70 line : 26 < switch , - >
71 line : 26 < ( , - >
72 line : 26 < id , a >
73 line : 26 < ) , - >
74 line : 27 < { , - >
75 line : 28 < case , - >
76 line : 28 < dec , 3 >
```

```
77 line : 28 < : , - >
78 line : 29 < case , - >
79 line : 29 < dec , 7 >
80 line : 29 < : , - >
81 line : 30 < default , - >
82 line : 30 < : , - >
83 line : 31 < break , - >
84 line : 31 < ; , - >
85 line : 32 < } , - >
86 line : 34 < return , - >
87 line : 34 < oct , 0 >
88 line : 34 < ; , - >
89 line : 35 < } , - >
```

可以看到结果与上述小程序符合

sum.txt

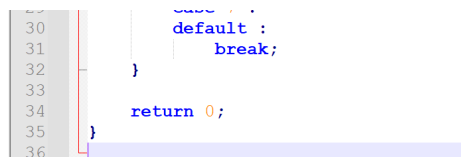
```
1 此程序一共有36行
2 此程序一共有89个单词
3
4 default有1个
5 int有4个
6 }有5个
7 =有3个
8 id有17个
9 void有1个
10 (有7个
11 )有7个
12 {有5个
13 ;有11个
14 num有6个
15 str有3个
16 ,有2个
17 +有2个
18 if有2个
19 while有1个
20 --有1个
21 ++有2个
22 <=有1个
```

```

23 case有2个
24 switch有1个
25 :有3个
26 break有1个
27 return有1个
28
29 下面是本程序的标识符
30 < id, main>
31 < id, a>
32 < id, b>
33 < id, printf>
34 < id, c>

```

可以看到统计结果和源程序相符合。尤其是行数的统计。



```

30
31
32
33
34
35
36

```

虽然最后没有字符了是空行。但是因为你在写程序的时候打了。所以也要统计进去。

2.各种错误的测试小程序+混乱排版

test2.c

```

1  # include <stdio.h>
2  # include          <stdlib.h>
3  # include <cstring>
4
5  @ int m;
6
7  char c = ' ';
8
9  void test()
10 {
11     int dist[100];
12     memset(dist,-1,sizeof dist);
13 }
14
15 int c = 0xppppp;

```

```

16
17 char* 8989___cowboy;
18 char cowboy = '123123123123';
19         int main(void)
20         {
21             /*
22             */
23

```

error.txt

```

1 5行 错误是：有不合法的字符
2 7行 错误是：字符为空
3 15行 错误是：数字形态不正确
4 17行 错误是：标识符不能以数字开头
5 18行 错误是：字符长度太长
6 21行 错误是：注释没有闭合

```

result.txt

```

1 line : 5 < int , - >
2 line : 5 < id , m >
3 line : 5 < ; , - >
4 line : 7 < char , - >
5 line : 7 < id , c >
6 line : 7 < assign , = >
7 line : 7 < ; , - >
8 line : 9 < void , - >
9 line : 9 < id , test >
10 line : 9 < ( , - >
11 line : 9 < ) , - >
12 line : 10 < { , - >
13 line : 11 < int , - >
14 line : 11 < id , dist >
15 line : 11 < [ , - >
16 line : 11 < dec , 100 >
17 line : 11 < ] , - >

```

```
18 line : 11 < ; , - >
19 line : 12 < id , memset >
20 line : 12 < ( , - >
21 line : 12 < id , dist >
22 line : 12 < , >
23 line : 12 < - , - >
24 line : 12 < - , - >
25 line : 12 < dec , 1 >
26 line : 12 < , >
27 line : 12 < sizeof , - >
28 line : 12 < id , dist >
29 line : 12 < ) , - >
30 line : 12 < ; , - >
31 line : 13 < } , - >
32 line : 15 < int , - >
33 line : 15 < id , c >
34 line : 15 < assign , = >
35 line : 15 < id , ppppp >
36 line : 15 < ; , - >
37 line : 17 < char , - >
38 line : 17 < * , - >
39 line : 17 < dec , 8989 >
40 line : 17 < id , ___cowboy >
41 line : 17 < ; , - >
42 line : 18 < char , - >
43 line : 18 < id , cowboy >
44 line : 18 < assign , = >
45 line : 18 < ; , - >
46 line : 19 < int , - >
47 line : 19 < id , main >
48 line : 19 < ( , - >
49 line : 19 < void , - >
50 line : 19 < ) , - >
51 line : 20 < { , - >
52
```

sum.txt

- 1 此程序一共有23行
- 2 此程序一共有51个单词

```
3
4  int有4个
5  }有1个
6  =有3个
7  id有12个
8  {有2个
9  ;有7个
10 char有3个
11 (有3个
12 void有2个
13 )有3个
14 [有1个
15 num有3个
16 ]有1个
17 ,有2个
18 -有2个
19 *有1个
20 sizeof有1个
21
22 下面是本程序的标识符
23  < id, m>
24  < id, c>
25  < id, test>
26  < id, dist>
27  < id, memset>
28  < id, cowboy>
29  < id, ___cowboy>
30  < id, ppppp>
31  < id, main>
32
```

3.一些特殊情况的小程序

test3.c

```

1 char str[100] = "cowboy \";
2 float b = 0123.123;
3
4 struct p{
5     int a;
6     int b;
7 };
8
9 struct p* x;
10 x->a;
11 x->b;
12

```

- 这个程序里面有几处一开始让人忽略的特殊情况。比如字符串里的 `\` 那么此时他就真的代表字符串里有元素是 `"`。此时不能判断成结尾。因为判断成结尾了之后。很显然后面的引号就会没有匹配。就要报错。这个和事实情况不符合。
- `0123.123` 不能被判断成8进制。我在正规编译器上做过测试。会把它当成10进制。
- `->` 运算符很容易被忽略。容易把它识别成一个 `-` 一个 `>` 号

error.txt

无

result.txt

```

1 line : 1 < char , - >
2 line : 1 < id , str >
3 line : 1 < [ , - >
4 line : 1 < dec , 100 >
5 line : 1 < ] , - >
6 line : 1 < assign , = >
7 line : 1 < str , cowboy \" >
8 line : 1 < ; , - >
9 line : 2 < float , - >
10 line : 2 < id , b >
11 line : 2 < assign , = >
12 line : 2 < dec , 0123.123 >

```

```
13 line : 2 < ; , - >
14 line : 4 < struct , - >
15 line : 4 < id , p >
16 line : 4 < { , - >
17 line : 5 < int , - >
18 line : 5 < id , a >
19 line : 5 < ; , - >
20 line : 6 < int , - >
21 line : 6 < id , b >
22 line : 6 < ; , - >
23 line : 7 < } , - >
24 line : 7 < ; , - >
25 line : 9 < struct , - >
26 line : 9 < id , p >
27 line : 9 < * , - >
28 line : 9 < id , x >
29 line : 9 < ; , - >
30 line : 10 < id , x >
31 line : 10 < -> , - >
32 line : 10 < id , a >
33 line : 10 < ; , - >
34 line : 11 < id , x >
35 line : 11 < -> , - >
36 line : 11 < id , b >
37 line : 11 < ; , - >
38
```








- 看第7行。它把字符串识别成了一个完整的。
- 看第12行。它分辨出了这个是10进制不是8进制。
- 看第10和35行。它分辨出了这个是 -> 运算符。

sum.txt

```
1 此程序一共有12行
2 此程序一共有37个单词
3
4 char有1个
5 str有1个
6 }有1个
7 =有2个
```


本次词法分析开始!
请输入测试样例的文件名o(∇^o
test4.c
本次词法分析用时 1.136000 秒
本次词法分析结束了! ㄟ(≧◇≦)ㄏ

分析它花了不少时间。我们来看一下结果。尤其是result文件很大。

 error.txt	2021/10/9 15:00	文本文档	0 KB
 errorTpye.txt	2021/10/9 13:29	文本文档	1 KB
 keyWords.txt	2021/10/3 17:02	文本文档	1 KB
 main.cpp	2021/10/8 23:01	C++ Source File	1 KB
 result.txt	2021/10/9 15:00	文本文档	6,129 KB
 sum.txt	2021/10/9 15:00	文本文档	99 KB
 test.c	2021/10/9 14:56	C Source File	1,998 KB

error.txt

无

result.txt

我们随便找几行来对比

test.c error.txt sum.txt result.txt

```

41748 case -2:
41749     if (8 * sizeof(Py_ssize_t) > 2 * PyLong_SHIFT) {
41750         return -(Py_ssize_t) (((((size_t)digits[1]) << PyLong_SHIFT) | (size_t)1) < 0);
41751     }
41752     break;
41753 case 3:
41754     if (8 * sizeof(Py_ssize_t) > 3 * PyLong_SHIFT) {
41755         return (Py_ssize_t) ((((((size_t)digits[2]) << PyLong_SHIFT) | (size_t)1) < 0);
41756     }
41757     break;
41758 case -3:
41759     if (8 * sizeof(Py_ssize_t) > 3 * PyLong_SHIFT) {
41760         return -(Py_ssize_t) ((((((size_t)digits[2]) << PyLong_SHIFT) | (size_t)1) < 0);
41761     }
41762     break;
41763 case 4:

```

可以看到分析的是正确的

```

212140 line: 41750 (<), ->
212141 line: 41750 (<), ->
212142 line: 41750 (<), ->
212143 line: 41751 (<), ->
212144 line: 41752 < break, ->
212145 line: 41752 (<), ->
212146 line: 41753 < case, ->
212147 line: 41753 < dec, 3 >
212148 line: 41753 (<), ->
212149 line: 41754 < if, ->
212150 line: 41754 (<), ->
212151 line: 41754 < dec, 8 >
212152 line: 41754 < *, ->
212153 line: 41754 < sizeof, ->
212154 line: 41754 (<), ->
212155 line: 41754 < id, Py_ssize_t >

```

可以发现基本正确。也就是说本程序对缓冲区的测试是成功的。


sum.txt

The screenshot shows a C program in `test.c` that counts tokens in a file named `sum.txt`. The program uses `fopen` to open the file, `fscanf` to read tokens, and `fclose` to close it. The output of the program is shown in the terminal window, displaying the number 41793.

```
test.c error.txt result.txt sum.txt
41793
```

```
1 此程序一共有41793行
2 此程序一共有212520个单词
3
4 )有3452个
5 =有8665个
6 struct有540个
7 id有64737个
8 enum有4个
9 ;有20397个
10 {有3448个
```

可以看到统计是正确的。



The screenshot shows the GDB interface with the following content:

test.c | **error.txt** | **sum.txt**

```
2060 ic PyObject * pyx_n_s_constant_to_str;
2061 ic PyObject * pyx_n_s_constructed_tid_to_last_frame;
2062 ic PyObject * pyx_n_s_current_frames;
2063 ic PyObject * pyx_n_s_debug;
```

sum.txt

```
68
69 下面是本程序的标识符
70 < id, pyx_n_s_constructed_tid_to_last_frame>
71 < id, METH_COEXIST>
```

test.c | **error.txt** | **sum.txt**

```
2060 ic PyObject * pyx_n_s_constant_to_str;
2061 ic PyObject * pyx_n_s_constructed_tid_to_last_frame;
```

sum.txt

```
1529 < id, pyx_n_s_SUPPORT_GEVENT>
1530 < id, pyx_n_s_constant_to_str>
```

可以到符号表里任意的标识符都统计到了都标识了。由于我用的是哈希表所以顺序可能会不同

5.超长的程序+错误进行压力测试

test5.c

我在两万行放了一些错误

```
19998      goto __pyx_L181;
19999      }
20000      char c = ' ' char c1 = 'cc' int b = 0xGGG;
20001      @@@
20002      /* "_pydevd_bundle/pydevd_cython.pyx":1129
```

我又修改了程序让注释没有闭合

```
41568      }
41569      @
41570      /*
41571      static CYTHON_INLINE PyObject* __Pyx_PyUnicode_FromString(const char* c_str) {
41794      #endif /* Py_PYTHON_H
41795
```

```
本次词法分析开始!
请输入测试样例的文件名o(▽^)^o
test5.c
本次词法分析用时 1.123000 秒
本次词法分析结束了! ㄟ(≧◇≦)ㄟ
```

error.txt

```
1 20000行 错误是: 字符为空
2 20000行 错误是: 字符长度太长
3 20000行 错误是: 数字形态不正确
4 20001行 错误是: 有不合法的字符
5 20001行 错误是: 有不合法的字符
6 20001行 错误是: 有不合法的字符
7 41569行 错误是: 有不合法的字符
8 41570行 错误是: 注释没有闭合
9
```

符合预期

result.txt

```
test.c error.txt result.txt sum.txt
211150 line : 41563 < dec , 1 >
211151 line : 41563 < ) , - >
211152 line : 41564 < return , - >
211153 line : 41564 < - , - >
211154 line : 41564 < - , - >
211155 line : 41564 < dec , 1 >
211156 line : 41564 < ; , - >
211157 line : 41565 < , >
211158 line : 41565 < id , t >
211159 line : 41565 < ; , - >
211160 line : 41566 < } , - >
211161 line : 41567 < return , - >
211162 line : 41567 < oct , 0 >
211163 line : 41567 < ; , - >
211164 line : 41568 < } , - >
211165
```

可以看到到最后因为全部注释掉了。所以后面的东西不记入词法分析了

sum.txt

```
test.c error.txt result.txt sum.txt
1 此程序一共有41795行
2 此程序一共有211164个单词
3
4 }有3410个
5 =有8642个
```

统计也是符合的

实验总结

程序的优点

- 较为完整的，基于自己给出的假设，完成了词法分析。并且各种情况基本考虑完善，尤其是数字，还有各种符号。
- 使用了哈希表 `unordered_map` 进行了优化。使得查表都是根据字符串索引 $O(1)$ 的复杂度内实现。
- 改进了缓冲区的各种判断。设标志位。让它一直交替读入。

程序还可以完善的地方

- 对于转移字符的支持不够。因为在c语言中允许 `\123` `\xa1` 这种转移字符。我并没有支持这种情况。我只支持普通的 `\n` 这些转义字符。
- 一些细小的语法没有支持。比如 `123L` 这种语法表示的是123是一个 `long` 的类型。这种类型我没有支持。
- 自动机的实现比较臃肿。其实有一种方法叫做 表驱动法 。可以让代码简洁很多。

实验的收获

- "基于什么样的假设"。这个是上课的时候老师提醒的和强调的。我一开始不知道是什么意思，我也不知道有什么用。直到写程序的时候才知道。c语言的标准，规则都太多太多了。如果我不提前说我的假设的话，就会让看我程序的人摸不着头脑，就不会清楚我的词法分析到底实现到了什么样的程度！而且每个人的想法都是不一样的。这些东西都是很灵活的。有可能一个东西我认为该报错，你认为不报错。我认为有个东西的格式分类是这样的，你认为是那样的。如果我不提前说我的假设，那么别人阅读我的报告和程序的时候完全可以基于自己的假设去读。这样就乱套了。
- 本次实验也让我对编译的过程以及计算机更加了解了。如何以计算机的方式来思考问题，让它能够识别单词是个很重要的步骤。在思考这个过程中，我的编程水平也大有长进。