# 跟vczh看实例学编译原理——二：实现Tinymoe的词法分析

文章中引用的代码均来自<https://github.com/vczh/tinymoe>。

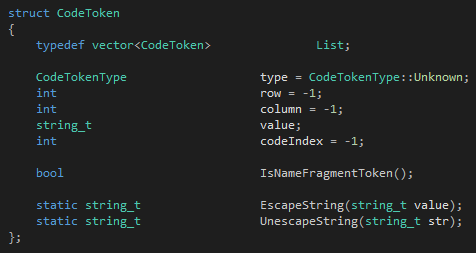
实现Tinymoe的第一步自然是一个词法分析器。词法分析其所作的事情很简单，就是把一份代码分割成若干个token，记录下他们所在文件的位置，以及丢掉不必要的信息。但是Tinymoe是一个按行分割的语言，自然token列表也就是二维的，第一维是行，第二维是每一行的token。在继续讲词法分析器之前，先看看Tinymoe包含多少token：

* 符号：(、)、,、:、&、+、-、\*、/、\、%、<、>、<=、>=、=、<>
* 关键字：module、using、phrase、sentence、block、symbol、type、cps、category、expression、argument、assignable、list、end、and、or、not
* 数字：123、456.789
* 字符串：”abc\r\ndef”
* 标识符：tinymoe
* 注释：-- this is a comment

Tinymoe对于token有一个特殊的规定，就是字符串和注释都是单行的。因此如果一个字符串在没有结束之前就遇到了换行，那么这种写法定义为你遇到了一个没有右双引号的字符串，需要报个错，然后下一行就不是这个字符串的内容了。

一个词法分析器所需要做的事情，就是把一个字符串分解成描述此法的数据结构。既然上面已经说到Tinymoe的token列表是二维的，因此数据结构肯定会体现这个观点。Tinymoe的词法分析器代码可以在这里找到：<https://github.com/vczh/tinymoe/blob/master/Development/Source/Compiler/TinymoeLexicalAnalyzer.h>。

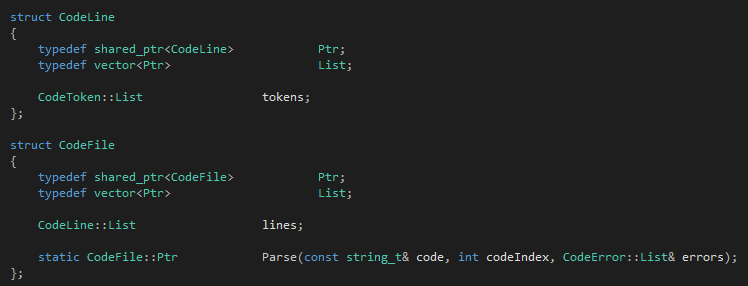
首先是token：



CodeTokenType是一个枚举类型，标记一个token的类型。这个类型比较细化，每一个关键字有自己的类型，每一个符号也有自己的类型，剩下的按种类来分。我们可以看到token需要记录的最关键的东西只有三个：类型、内容和代码位置。在token记录代码位置是十分重要的，正确地记录代码位置可以让你能够报告带位置的错误、从语法树的节点还原成代码位置、甚至在调试的时候可以把指令也换成位置。

这里需要提到的是，string\_t是一个typedef，具体的声明可以在这里看到：<https://github.com/vczh/tinymoe/blob/master/Development/Source/TinymoeSTL.h>。Tinymoe是完全由标准的C++11和STL写成的，但是为了适应不同情况的需要，Tinymoe分为依赖code page的版本和Unicode版本。如果编译Tinymoe代码的时候声明了全局的宏UNICODE\_TINYMOE的话，那Tinymoe所有的字符处理将使用wchar\_t，否则使用char。char的类型和Tinymoe编译器在运行的时候操作系统当前的code page是绑定的。所以这里会有类似string\_t啊、ifstream\_t啊、char\_t等类型，会在不同的编译选项的影响下指向不同的STL类型或者原生的C++类型。github上的VC++2013工程使用的是wchar\_t的版本，所以string\_t就是std::wstring。

Tinymoe的词法分析器除了token的类型以外，肯定还需要定义整个文件结构在词法分析后的结果：



这个数据结构体现了“Tinymoe的token列表是二维的”的这个观点。一个文件会被词法分析器处理成一个shared\_ptr<CodeFIle>对象，CodeFile::lines记录了所有非空的行，CodeLine::tokens记录了该行的所有token。

现在让我们来看词法分析的具体过程。关于如何从正则表达式构造词法分析器可以在这里（<http://www.cppblog.com/vczh/archive/2008/05/22/50763.html>）看到，不过我们今天要讲一讲如何人肉构造词法分析器。方法其实是一样的，首先人肉构造状态机，然后把用状态机分析输入的字符串的代码抄过来就可以了。但是很少有人会解耦得那么开，因为这样写很容易看不懂，其次有可能会遇到一些极端情况是你无法用纯粹的正则表达式来分词的，譬如说C++的raw string literal：R”tinymoe(这里的字符串没有转义)tinymoe”。一个用【R”<一些字符>(】开始的字符串只能由【)<同样的字符>”】来结束，要顺利分析这种情况，只能通过在状态机里面做hack才能解决。这就是为什么我们人肉构造词法分析器的时候，会把状态和动作都混在一起写，因为这样便于处理特殊情况。

不过幸好的是，Tinymoe并没有这种情况发生。所以我们可以直接从状态机入手。为了简单起见，我在下面的状态机中去掉所有不是+和-的符号。首先，我们需要一个起始状态和一个结束状态：

首先我们添加整数和标识符进去：

0-9

0-9

a-zA-Z\_

a-zA-Z\_

其次是加减和浮点：

0-9

0-9

a-zA-Z\_

a-zA-Z\_

+-

0-9

.

最后把字符串和注释补全：

0-9

0-9

a-zA-Z\_

a-zA-Z\_

+

0-9

.

-

-

^\n

“

”

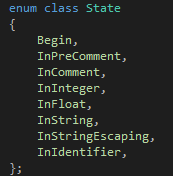
^”\\\n

\\

^\n

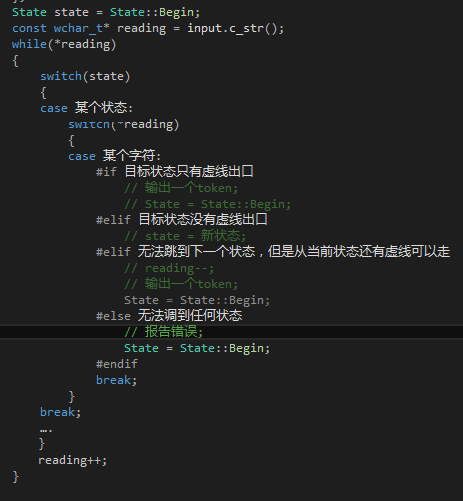
这样状态机就已经完成了。读过编译原理的可能会问，为什么终结状态都是由虚线而不是带有输入的实现指向的？因为虚线在这里有特殊的意义，也就是说它不能移动输入的字符串的指针，而且他还要负责添加一个token。当状态跳到End之后，那他就会变成Start，所以**实际上Start和End是同一个状态**。这个状态机也不是输入什么字符都能跳转到下一个状态的。所以**当你发现输入的字符让你无路可走的时候，你就是遇到了一个词法错误**。

这样我们的设计就算完成了，接下来就是如何用C++来实现它了。为了让代码更容易阅读，我们应该给Start和1-9这么多状态起名字，做法如下：



在这里类似状态3这样的状态被我省略掉了，因为这个状态唯一的出路就是虚线，所以跳到这个状态意味着你要立刻执行虚线，也就是说你不需要做“跳到这个状态”这个动作。因此它不需要有一个名字。

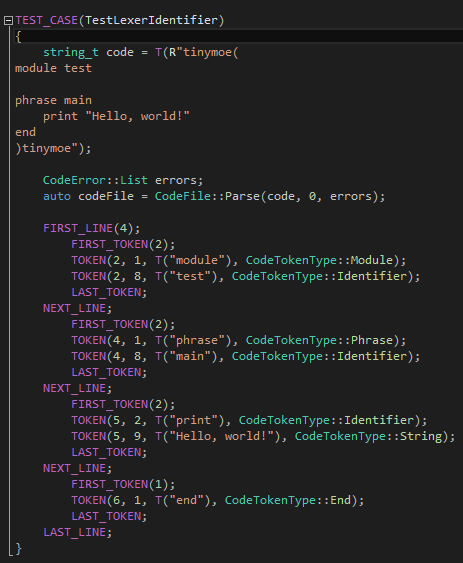
然后你只要按照下面的做法翻译这个状态机就可以了：



只要写到这里，那么我们就初步完成了词法分析器了。其实任何系统的主要功能都是相对容易实现的，往往是次要的功能才需要花费大量的精力来完成，而且还很容易出错。在这里“次要的功能”就是——记录token的行列号，还有维护CodeFile::lines避免空行的出现！

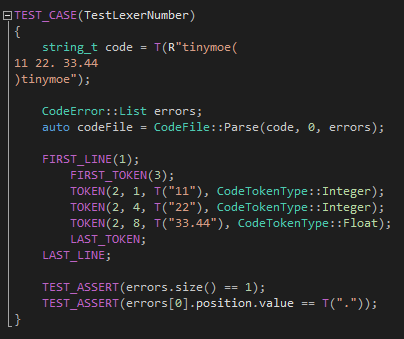
尽管我已经做过了那么多次词法分析器，但是我仍然无法一气呵成写对，仍然会出一些bug。面对编译器这种纯计算程序，debug的最好方法就是写单元测试。不过对于不熟悉单元测试的人来讲可能很难一下子想到要做什么测试，在这里我可以把我给Tinymoe谢的一部分单元测试在这里贴一下。

第一个，当然是传说中的“Hello, world!”测试了：

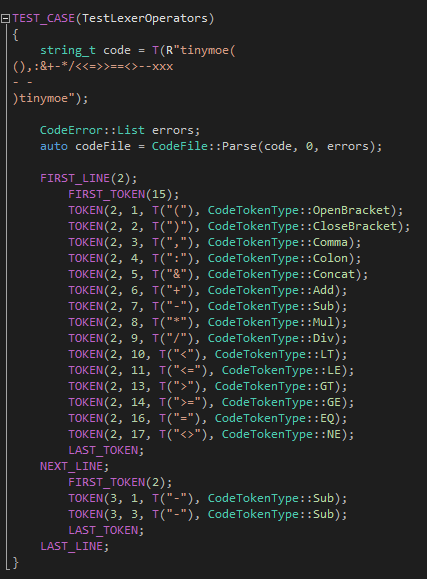


TEST\_CASE和TEST\_ASSERT（这里暂时没有直接用到TEST\_ASSERT）是我为了开发Tinymoe随手撸的一个宏，可以在Tinymoe的代码里看到。为了检查我们有没有粗心大意，我们有必要检查词法分析器的任何一个输出的数据，譬如每一行有多少token，譬如每一个token的行号列好内容和类型。我为了让这些枯燥的测试用例容易看懂，在这个文件（<https://github.com/vczh/tinymoe/blob/master/Development/TinymoeUnitTest/TestLexicalAnalyzer.cpp>）的开头可以看到FIRST\_LINE、FIRST\_TOKEN、TOKEN、LAST\_TOKEN、NEXT\_LINE、LAST\_LINE是怎么定义的。其实吧这些宏展开，就是一个普通的遍历CodeFile::lines和CodeLine::tokens的程序，然后TEST\_ASSERT一下CodeToken的每一个成员是否我们所需要的值。我相信看到这里很多人肯定把重点放在宏和炫技上，而不是如何设计测试用例上，这是有前途的程序员和没前途的程序员面对一份资料的反应的重要区别之一。没前途的程序员总是会把注意力放在一些莫名其妙的地方，其中一个例子就是“过早优化”。

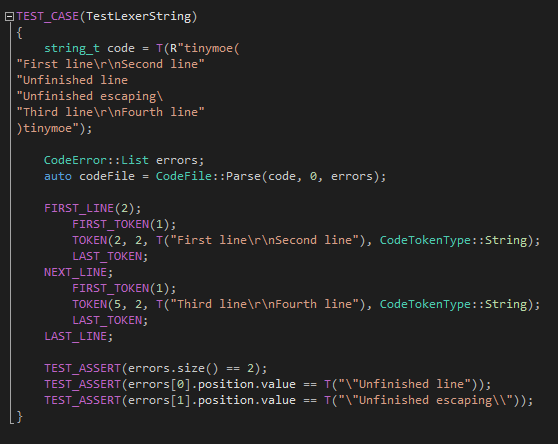
第二个测试用例针对的是整数和浮点的输出和报错上，重点在于检查每一个token的列号是不是正确的计算了出来：



第三个测试用例的重点主要是-符号和—注释的分析：



第四个测试用例则是测试字符串的escaping和在面对换行的时候是否正确的处理（之前提到字符串不能换行，遇到一个突然的换行将会被看成缺少双引号）：



鉴于词法分析本来内容也不多，所以这篇文章也不会太长。相信有前途的程序员也会在这里得到一些编译原理以外的知识。下一篇文章将会描述Tinymoe的函数头的语法分析部分，将会描述一个编译器的不带歧义的语法分析是如何人肉出来的。敬请期待。