# 跟vczh看实例学编译原理——三：Tinymoe与无歧义语法分析

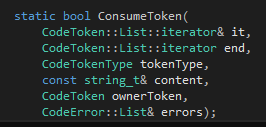
文章中引用的代码均来自<https://github.com/vczh/tinymoe>。

看了前面的三篇文章，大家应该基本对Tinymoe的代码有一个初步的感觉了。在正确分析“print sum from 1 to 100”之前，我们首先得分析“phrase sum from (lower bound) to (upper bound)”这样的声明。Tinymoe的函数声明又很多关于block和sentence的配置，不过这里并不打算将所有细节，我会将重点放在如何写一个针对无歧义语法的递归下降语法分析器上。所以我们这里不会涉及sentence和block的什么category和cps的配置。

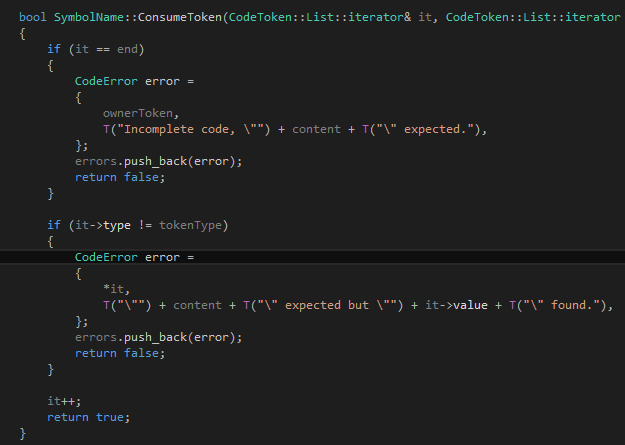
虽然“print sum from 1 to 100”无法用无歧义的语法分析的方法做出来，但是我们可以借用对“phrase sum from (lower bound) to (upper bound)”的语法分析的结果，动态构造能够分析“print sum from 1 to 100”的语法分析器。这种说法看起来好像很高大上，但是其实并没有什么特别难的技巧。关于“构造”的问题我将在下一篇文章《跟vczh看实例学编译原理——三：Tinymoe与有歧义语法分析》详细介绍。

在我之前的博客里我曾经写过《[如何手写语法分析器](http://www.cppblog.com/vczh/archive/2008/06/15/53373.html)》，这篇文章讲了一些简单的写递归下降语法分析器的规则，尽管很多人来信说这篇文章帮他们解决了很多问题，但实际上细节还不够丰富，用来对编程语言做语法分析的话，还是会觉得复杂性太高。这篇文章也同时作为《[如何手写语法分析器](http://www.cppblog.com/vczh/archive/2008/06/15/53373.html)》的补充。好了，我们开始进入无歧义语法分析的主题吧。

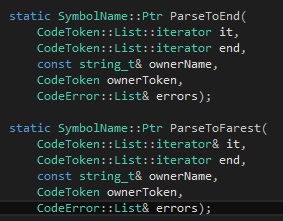
我们需要的第一个函数是用来读token并判断其内容是不是我们希望看到的东西。这个函数比较特别，所以单独拿出来讲。在词法分析里面我们已经把文件分行，每一行一个CodeToken的列表。但是由于一个函数声明独占一行，因此在这里我们只需要对每一行进行分析。我们判断这一行是否以cps、category、symbol、type、phrase、sentence或block开头，如果是那Tinymoe就认为这一定是一个声明，否则就是普通的代码。所以这里假设我们找到了一行代码以上面的这些token作为开头，于是我们就要进入语法分析的环节。作为整个分析器的基础，我们需要一个ConsumeToken的函数：



作为一个纯粹的C++11的项目，我们应该使用STL的迭代器。其实在写语法分析器的时候，基于迭代器的代码也比基于“token在数组里的下表”的代码要简单得多。这个函数所做的内容是这样的，它查看it指向的那个token，如果token的类型跟tokenType描述的一样，他就it++然后返回true；否则就是用content和ownerToken来产生一个错误信息加入errors列表里，然后返回false。当然，如果传进去的参数it本身就等于end，那自然要产生一个错误。自然，函数体也十分简单：



那对于标识符和数字怎么办呢？明眼人肯定一眼就看出来，这是给检查符号用的，譬如左括号、右括号、冒号和关键字等。在声明里面我们是不需要太复杂的东西的，因此我们还需要两外一个函数来输入标识符。Tinymoe事实上有两个针对标识符的语法分析函数，第一个是读入标识符，第二个不仅要读入标识符还要判断是否到了行末否则报错：



在这里我需要强调一个重点，在写语法分析器的时候，函数的各式一定要整齐划一。Tinymoe的语法分析函数有两个格式，分别是针对parse一行的一个部分，和parse一个文件的一些行的。ParseToEnd和ParseToFarest就属于parse一行的一个部分的函数。这种函数的格式如下：

1. 返回值一定是语法树的一个节点。在这里我们以share\_ptr<SymbolName>作为标识符的节点。一个标识符可以含有多个标识符token，譬如说the Chinese people、the real Tinymoe programmer什么的。因此我们可以简单地推测SymbolName里面有一个vector<CodeToken>。这个定义可以在TinymoeLexicalAnalyzer.h的最后一部分找到。
2. 前两个参数分别是iterator&和指向末尾的iterator。末尾通常指“从这里开始我们不希望这个parser函数来读”，当然这通常就意味着行末。我们把“末尾”这个概念抽取出来，在某些情况下可以得到极大的好处。
3. 最后一个token一定是vector<CodeError>&，用来存放过程中遇到的错误。

除了函数格式以外，我们还需要所有的函数都遵循某些前置条件和后置条件。在语法分析里，如果你试图分析一个结构但是不幸出现了错误，这个时候，你有可能可以返回一个语法树的节点，你也有可能什么都返回不了。于是这里就有两种情况：

1. 你可以返回，那就证明虽然输入的代码有错误，但是你成功的进行了错误恢复——实际上就是说，你觉得你自己可以猜测这份代码的作者实际是要表达什么意思——于是你要移动第一个iterator，让他指向你第一个还没读到的token，以便后续的语法分析过程继续进行下去。
2. 你不能返回，那就证明你恢复不了，因此第一个iterator不能动。因为这个函数有可能只是为了测试一下当前的输入是否满足一个什么结构。既然他不是，而且你恢复不出来——也就是你猜测作者本来也不打算在这里写某个结构——因此iterator不能动，表示你什么都没有读。

当你根据这样的格式写了很多语法分析函数之后，你会发现你可以很容易用简单结构的语法分析函数，拼凑出一个复杂的语法分析函数。