# 跟vczh看实例学编译原理——三：Tinymoe与无歧义语法分析

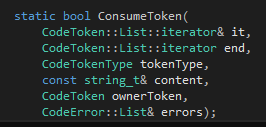
文章中引用的代码均来自<https://github.com/vczh/tinymoe>。

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。

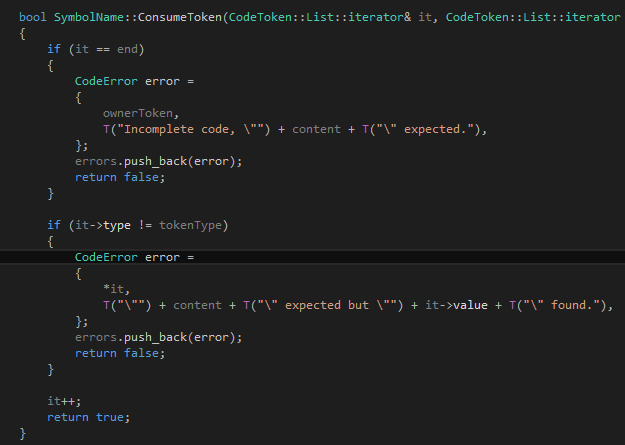
虽然“print sum from 1 to 100”无法用无歧义的语法分析的方法做出来，但是我们可以借用对“phrase sum from (lower bound) to (upper bound)”的语法分析的结果，动态构造能够分析“print sum from 1 to 100”的语法分析器。这种说法看起来好像很高大上，但是其实并没有什么特别难的技巧。关于“构造”的问题我将在下一篇文章《跟vczh看实例学编译原理——三：Tinymoe与有歧义语法分析》详细介绍。

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。

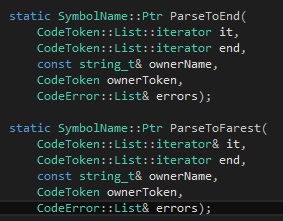
我们需要的第一个函数是用来读token并判断其内容是不是我们希望看到的东西。这个函数比较特别，所以单独拿出来讲。在词法分析里面我们已经把文件分行，每一行一个CodeToken的列表。但是由于一个函数声明独占一行，因此在这里我们只需要对每一行进行分析。我们判断这一行是否以cps、category、symbol、type、phrase、sentence或block开头，如果是那Tinymoe就认为这一定是一个声明，否则就是普通的代码。所以这里假设我们找到了一行代码以上面的这些token作为开头，于是我们就要进入语法分析的环节。作为整个分析器的基础，我们需要一个ConsumeToken的函数：



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：

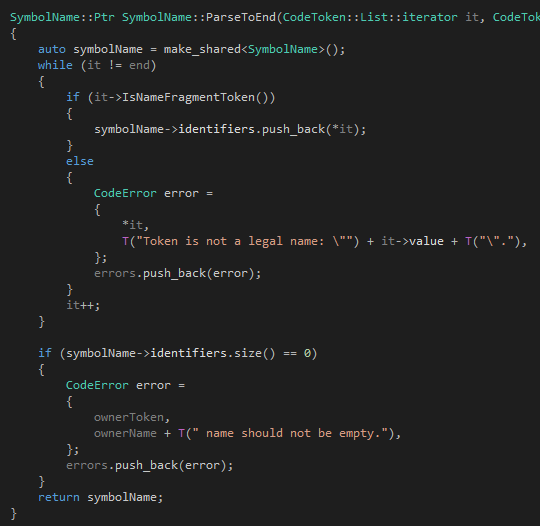
1. 返回值一定是语法树的一个节点。在这里我们以share\_ptr<SymbolName>作为标识符的节点。一个标识符可以含有多个标识符token，譬如说the Chinese people、the real Tinymoe programmer什么的。因此我们可以简单地推测SymbolName里面有一个vector<CodeToken>。这个定义可以在TinymoeLexicalAnalyzer.h的最后一部分找到。
2. 前两个参数分别是iterator&和指向末尾的iterator。末尾通常指“从这里开始我们不希望这个parser函数来读”，当然这通常就意味着行末。我们把“末尾”这个概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

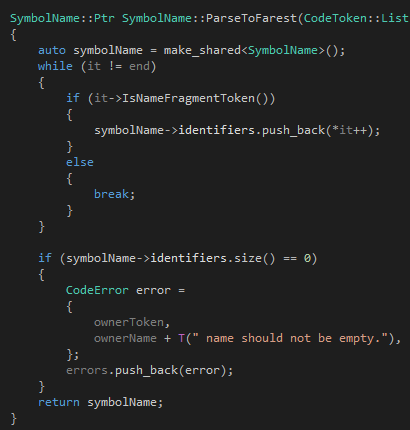
当你根据这样的格式写了很多语法分析函数之后，你会发现你可以很容易用简单结构的语法分析函数，拼凑出一个复杂的语法分析函数。但是由于Tinymoe的声明并没有一个编程语言那么复杂，所以这种嵌套结构出现的次数并不多，于是我们这里先跳过关于嵌套的讨论，等到后面具体分析“函数指针类型的参数”的时候自然会涉及到。

说了这么多，我觉得也应该上ParseToEnd和ParseToFarest的代码了。首先是ParseToEnd：



我们很快就可以发现，其实语法分析器里面绝大多数篇幅的代码都是关于错误处理的，真正处理正确代码的部分其实很少。ParseToEnd做的事情不多，他就是从it开始一直读到end的位置，把所有不是标识符的token都扔掉，然后把所有遇到的标识符token都连起来作为一个完整的标识符。也就是说，ParseToEnd遇到类似“the real 100 Tinymoe programmer”的时候，他会返回“the real Tinymoe programmer”，然后在“100”的地方报一个错误。

ParseToFarest的逻辑差不多：



只是当这个函数遇到“the real 100 Tinymoe programmer”的时候，会返回“the real”，然后把光标移动到“100”，但是没有报错。

看了这几个基本的函数之后，我们可以进入正题了。做语法分析器，当然还是从文法开始。跟上一篇文章一样，我们来尝试直接构造一下文法。但是语法分析器跟词法分析器的文法的区别是，词法分析其的文法可以 “定义函数”和“调用函数”。

首先，我们来看symbol的文法：

**SymbolName** ::= <identifier> { <identifier> }

**Symbol** ::= “symbol” **SymbolName**

其次，就是type的声明。type是多行的，不过我们这里只关心开头的一样：

**Type** ::= “type” **SymbolName** [ “:” **SymbolName** ]

在这里，中括号代表可有可无，大括号代表重复0次或以上。现在让我们来看函数的声明。函数的生命略为复杂：

Function ::= (“phrase” | “sentence” | “block”) { **SymbolName** | “(“ **Argument** “)” } [ “:” SymbolName ]

**Argument** ::= [“list” | “expression” | “argument” | “assignable”] **SymbolName**

**Argument** ::= **SymbolName**

**Argument** ::= **Function**

**Declaration** ::= **Symbol** | **Type** | **Function**

在这里我们看到Function递归了自己，这是因为函数的参数可以是另一个函数。为了让这个参数调用起来更加漂亮一点，你可以把参数写成函数的形式，譬如说：

pharse (the number) is odd : **odd numbers**

return the number % 2 == 1

end

print all (**phrase (the number) is wanted**) in (numbers)

repeat with **the number** in all numbers

if **the number is wanted**

print the number

end

end

end

print main

print all **odd numbers** in array of (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

end

我们给“(the number) is odd”这个判断一个数字是不是奇数的函数，起了一个别名叫“odd numbers”，这个别名不能被调用，但是他等价于一个只读的变量保存着奇数函数的函数指针。于是我们就可以把它传递给“print all (…) in (…)”这个函数的第一个参数了。第一个参数声明成函数，所以我们可以在print函数内直接调用这个参数指向的odd numbers函数。

事实上Tinymoe的SymbolName是可以包含关键字的，但是我为了不让它写的太长，于是我就简单的写成了上面的那条式子。那Argument是否可以包含关键字呢？答案当然是可以的，只是当它以list、expression、argument、assignable、phrase、sentence、block开始的时候，我们强行认为他有额外的意义。

现在一个Tinymoe的声明的第一行都由Declaration来定义。当我们识别出一个正确的Declaration之后，我们就可以根据分析的结果来对后面的行进行分析。譬如说symbol后面没有东西，于是就这么完了。type后面都是成员函数，所以我们一直找到“end”为止。函数的函数体就更复杂了，所以我们会直接跳到下一个看起来像Declaration的东西——也就是以symbol、type、phrase、sentence、block、cps、category开始的行。这些步骤都很简单，所以问题的重点就是，**如何根据Declaration的文法来处理输入的字符串**。

为了让文法可以真正的运行，我们需要把它做成状态机。根据之前的描述，这个状态及仍然需要有“定义函数”和“执行函数”的能力。我们可以先假装他们是正则表达式，然后把整个状态机画出来。这个时候，“函数”本身我们把它看成是一个跟标识符无关的输入，然后就可以得到下面的状态机：

<identifier>

<identifier>

**SymbolName** ::=

**Symbol** ::=

**SymbolName**

“symbol”

**Type** ::=

**SymbolName**

“type”

**Function** ::=

“phrase”, “block”, “sentence”

**Argument**

**Argument**

“:”

**SymbolName**

**Argument** ::=

“list”, “expression”, “argument”, “assignable”

**SymbolName**

**SymbolName**

**Function**

这样我们的状态机就暂时完成了。但是现在还不能直接把它转换成代码，因为当我们遇到一个输入，而我们可以选择调用函数，而且可以用的函数还不止一个的时候，那应该怎么办呢？答案就是要检查我们的文法是不是有歧义。

文法的歧义是一个很有意思的问题。在我们真的实践一个编译器的时候，我们会遇到三种歧义：

1. 文法本身就是有歧义的，譬如说C++著名的A\* B;的问题。当A是一个变量的时候，这是一个乘法表达式。当A是一个类型的时候，这是一个变量声明语句。如果我们在语法分析的时候不知道A到底指向的是什么东西，那我们根本无从得知这一句话到底要取什么意思，于是要么返回错误，要么两个结果一起返回。这就是问法本身固有的歧义。
2. 文法本身没有歧义，但是在分析的过程中却无法在走每一步的时候都能够算出唯一的“下一个状态”。譬如说C#著名的A<B>C;问题。当A是一个变量的时候，这个语句是不成立的，因为C#的一个语句的根节点不能是一个操作符（这里是“>”）。当A是一个类型的时候，这是一个变量声明语句。从结果来看，这并没有歧义，但是当我们读完A<B>的时候仍然不知道这个语句的结构到底是取哪一种。实际上B作为类型参数，他也可以有B<C>这样的结构，因此这个B可以是任意长的。也就是说我们只有在“>”结束之后再多读几个字符才能得到正确的判断。譬如说C是“(1)”，那我们知道A**应该**是一个模板函数。如果C是一个名字，A多半**应该**是一个类型。因此我们在做语法分析的时候，只能两种情况一起考虑，并行处理。最后如果哪一个情况分析不下去了，就简单的扔掉，剩下的就是我们所需要的。
3. 文法本身没有歧义，而且分析的过程中只要你每一步都往后多看最多N个token，酒可以算出唯一的“下一个状态”到底是什么。这个想必大家都很熟悉，因为这个N就是LookAhead。所谓的LL(1)、SLR(1)、LR(1)、LALR(1)什么的，这个1其实就是N=1的情况。N当然不一定是1，他也可以是一个很大的数字，譬如说8。一个文法的LookAhead是多少，这是文法本身固有的属性。一个LR(2)的状态机你非要在语法分析的时候只LookAhead一个token，那也会遇到第二种歧义的情况。如果C语言没有typedef的话，那他就是一个带有LookAhead的没有歧义的语言了。

看一眼我们刚才写出来的文法，明显就是LookAhead=0的情况，而且连左递归都没有，写起来肯定很容易。那接下来我们要做的就是给“函数”算first set。一个函数的first set，顾名思义就是，他的第一个token都可以是什么。SymbolName、Symbol、Type、Function都不用看了，因为他们的文法第一个输入都是token，那就是他们的first set。最后就剩下Argument。Argument的第一个token除了list、expression、argument和assignable以外，还有Function。因此Argument的first set就是这些token加上Function的first set。如果文法有左递归的话，也可以用类似的方法做，只要我们在函数A->B->C->…->A的时候，知道A正在计算于是返回空集就可以了。当然，只有左递归才会遇到这种情况。

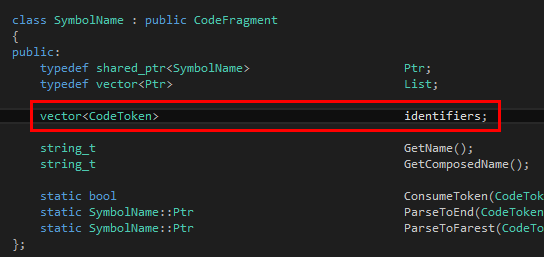
然后我们检查一下每一个状态，可以发现，任何一个状态出去的所有边，他接受的token或者函数的first set都是没有交集的。譬如Argument的0状态，第一条边接受的token、第二条边接受的SymbolName的first set，和第三条边接受的Function的first set，是没有交集的，所以我们就可以断定，这个文法一定没有歧义。按照上次状态机到代码的写法，我们可以机械的写出代码了。写代码的时候，我们把每一个文法的函数，都写成一个C++的函数。每到一个状态的时候，我们看一下当前的token是什么，然后再决定走哪条边。如果选中的边是token边，那我们就跳过一个token。如果选中的边是函数边，那我们不跳过token，转而调用那个函数，让函数自己去跳token。《[如何手写语法分析器](http://www.cppblog.com/vczh/archive/2008/06/15/53373.html)》用的也是一样的方法，如果对这个过程不清楚的，可以再看一遍这个文章。

于是我们到了定义语法树的时候了。幸运的是，我们可以直接从文法上看到语法树的形状，然后稍微做一点微调就可以了。我们把每一个函数都看成一个类，然后使用下面的规则：

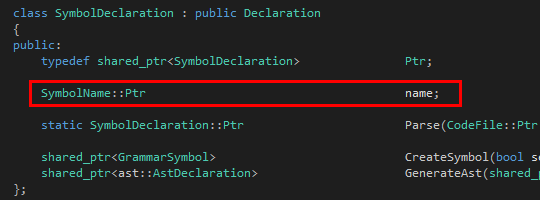
1. 对于A、A B、A B C等的情况，我们转换成class里面有若干个成员变量。
2. 对于A | B的情况，我们把它做成继承，A的语法树和B的语法树继承自一个基类，然后这个基类的指针就放在class里面做成员变量。
3. 对于{ A }，或者A { A }的情况，那个成员变量就是一个vector。
4. 对于[ A ]的情况，我们就当A看，区别只是，这个成员变量可以填nullptr。

对于每一个函数，要不要用shared\_ptr来装则见仁见智。于是我们可以直接通过上面的文法得到我们所需要的语法树：

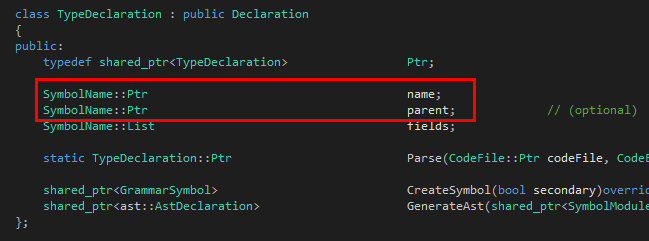
首先是SymbolName：



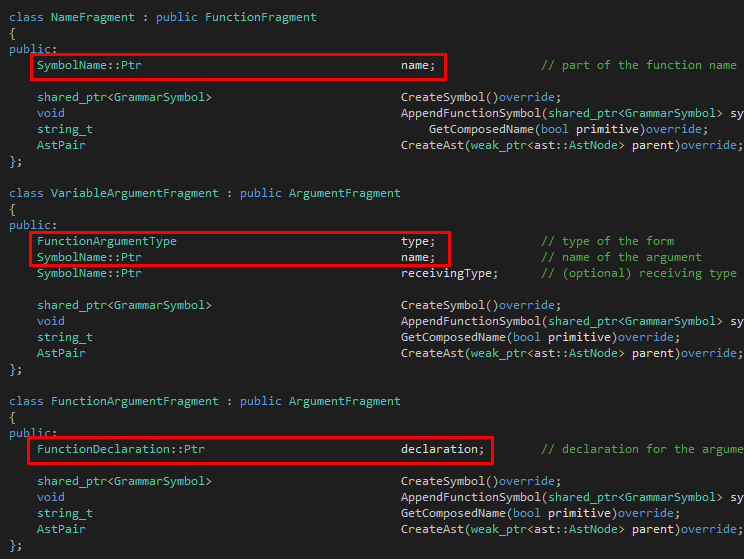
其次是Symbol：



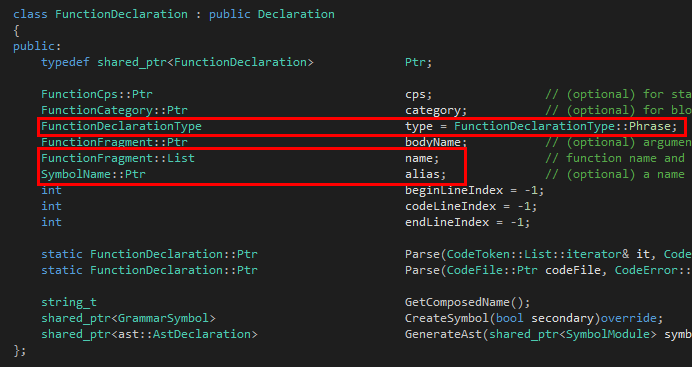
然后是Type：



接下来是Argument：



最后是Function：



大家可以看到，在Argument那里，同时出去的三条边就组成了三个子类，都继承自FunctionFragment。图中红色的部分就是Tinymoe源代码里在上述的文法里出现的那部分。至于为什么还有多出来的部分，其实是因为这里的文法是为了叙述方便简化过的。至于Tinymoe关于函数声明的所有语法可以分别看下面的四个github的wiki page：

<https://github.com/vczh/tinymoe/wiki/Phrases,-Sentences-and-Blocks>

<https://github.com/vczh/tinymoe/wiki/Manipulating-Functions>

<https://github.com/vczh/tinymoe/wiki/Category>

<https://github.com/vczh/tinymoe/wiki/State-and-Continuation>