# 跟vczh看实例学编译原理——一：Tinymoe的设计哲学

自从《序》胡扯了快一个月之后，终于迎来了正片。之所以系列文章叫《看实例学编译原理》，是因为整个系列会通过带大家一步一步实现Tinymoe的过程，来介绍编译原理的一些知识点。

但是第一个系列还没到开始处理Tinymoe源代码的时候，首先的跟大家讲一讲我设计Tinymoe的故事。为什么这种东西要等到现在才讲呢，因为之前没有文档，将了也是白讲啊。Tinymoe在github的wiki分为两部分，一部分是介绍语法的，另一部分是介绍一个最小的标准库是如何实现出来的，地址在 <https://github.com/vczh/tinymoe/wiki> 不带问号的那些都是写完了的。

## 系列文章的目标

在介绍Tinymoe之前，先说一下这个系列文章的目标。Ideally，只要一个人看完了这个系列，他就可以在下面这些地方得到**入门**：

1. 词法分析
2. 歧义与不歧义的语法分析
3. 语义分析
4. 符号表
5. 全文CPS变换
6. 编译生成高效的其他语言的代码
7. 编译生成自己的指令集
8. 带GC的虚拟机
9. 类型推导（intersection type，union type，concept mapping）
10. 跨过程分析（inter-procedural analyzing）

当然，这并不能让你成为一个大牛，但是至少自己做做实验搞一点高大上的东西骗师妹们是没有问题了。

## Tinymoe设计的目标

虽然想法很多年前就已经有了，但是这次我想把它实现出来，是为了完成《如何设计一门语言》的后续。光讲大道理是没有意义的，至少得有一个例子，让大家知道这些事情到底是什么样子的。因此Tinymoe有一点教学的意义，不管是使用它还是实现它。

首先，**处理Tinymoe需要的知识点多，用于编译原理教学**。既然是为了展示编译原理的基础知识，因此语言本身不可能是那种烂大街的C系列的东西。当然除了知识点以外，还会让大家深刻的理解到，**难实现跟难用，是完全没有关系的**！Tinymoe用起来可爽了，啊哈哈哈哈哈。

其次，**Tinymoe容易嵌入其他语言的程序，作为DSL使用，可以调用宿主程序提供的功能**。这严格的来讲不算语言本身的功能，而是实现本身的功能。就算是C++也可以设计为嵌入式，lua也可以被设计为编译成exe的。一个语言本身的设计并不会对如何使用它有多大的限制。为了让大家看了这个系列之后，写出来的东西不总是玩具，因此这也是设计的目标之一。

第三，**Tinymoe语法优化于描述复杂的逻辑**，而不是优化与复杂的数据结构和算法（虽然也可以）。Tinymoe本身是不存在任何细粒度控制内存的能力的，而且虽然可以实现复杂的数据结构和算法，但是本身描述这些东西最多也就跟javascript一样容易——其实就是不容易。但是Tinymoe设计的时候，是为了让大家把Tinymoe当成是一门可以设计DSL的语言，因此对复杂逻辑的描述能力特别强。唯一的前提就是，你懂得如何给Tinymoe写库。很好的使用和很好地实现一个东西是相辅相成的。我在设计Tinymoe之初，很多pattern我也不知道，只是因为设计Tinymoe遵循了科学的方法，因此最后我发现Tinymoe竟然具有如此强大的描述能力。当然对于读者们本身，也会在阅读系列文章的有类似的感觉。

最后，**Tinymoe是一个动态类型语言**。这纯粹是我的个人爱好了。对一门动态类型语言做静态分析那该多有趣啊，啊哈哈哈哈哈哈。

## Tinymoe的设计哲学

当然我并不会为了写文章就无线提高Tinymoe的实现难度的。为了把他控制在一个正常水平，因此设计Tinymoe的第一条就是：

**小规模的语言核心+大规模的标准库**

其实这跟C++差不多。但是C++由于想做的事情实在是太多了，譬如说视图包涵所有范式等等，因此就算这么做，仍然让C++本身包含的东西过于巨大（其实我还是觉得C++不难怎么办）。

语言核心小，实现起来当然容易。**但是你并不能为了让语言核心小就牺牲什么功能**。因此精心设计一个核心是必须的，因为所有你想要但是不想加入语言的功能，从此就可以用库来实现了。

譬如说，Tinymoe通过有条件地暴露continuation，要求编译器在编译Tinymoe的时候做一次全文CPS变换。这个东西说容易也不是那么容易，但是至少比你做分支循环异常处理什么的全部加起来要简单多了吧。所以我只提供continuation，剩下的控制流全部用库来做。这样有三个好处：

1. **语言简单，实现难度降低**。
2. **为了让库可以发挥应有的作用，语言的功能的选择十分的正交化**。不过这仍然在一定的程度上提高了学习的难度。但是并不是所有人都需要写库对吧，很多人只需要会用库就够了。通过一点点的牺牲，正交化可以充分发挥程序员的想象能力。这对于以DSL为目的的语言来说是不可或缺的。
3. **标准库本身可以作为编译器的测试用例**。你只需要准备足够多的测试用例来运行标准库，那么你只要用C++（假设你用C++来实现Tinymoe）来跑他们，那所有的标准库都会得到运行。运行结果如果对，那你对编译器的实现也就有信心了。为什么呢，因为标准库大量的使用了语言的各种功能，而且是无节操的使用。如果这样都能过，那普通的程序就更能过了。

**扩展后的东西跟原生的东西外观一致**

这是很重要的。如果扩展出来的东西跟原生的东西长得不一样，用起来就觉得很傻逼。Java的string不能用==来判断内容就是这样的一个例子。虽然他们有的是理由证明==的反直觉设计是对的——但是反直觉就是反直觉，就是一个大坑。

这种例子还有很多，譬如说go的数组和表的类型啦，go本身如果不要数组和表的话，是写不出长得跟原生数组和表一样的数组和表的。其实这也不是一个大问题，问题是go给数组和表的样子搞特殊化，还有那个反直觉的slice的赋值问题（会合法溢出！），类似的东西实在是太多了。一个东西特例太多，坑就无法避免。所以其实在我看来，go还不如给C语言加上erlang的actor功能了事。

反而C++在这件事情上就做得很好。如果你对C++不熟悉的话，有时候根本分不清什么是编译器干的，什么是标准库干的。譬如说static\_cast和dynamic\_cast长得像一个模板函数，因此boost就可以用类似的手法加入lexical\_cast和针对shared\_ptr的static\_pointer\_cast和dynamic\_pointer\_cast，整个标准库和语言本身浑然一体。这样子做的好处是，当你在培养对语言本身的直觉的时候，你也在培养对标准库的直觉，培养直觉这件事情你不用做两次。你对一个东西的直觉越准，学习新东西的速度就越快。所以C++的设计刚好可以让你在熬过第一个阶段的学习之后，后面都觉得无比的轻松。

## Tinymoe的实现难点

首先，**语法分析需要对Tinymoe程序处理三遍**。Tinymoe对于语句设计使得对一个Tinymoe程序做语法分析不是那么直接（虽然比C++什么的还是容易多了）。举个例子：

module hello world

phrase sum from (lower bound) to (upper bound)

…

end

sentence print (message)

…

end

phrase main

print sum from 1 to 100

end

第一遍分析是词法分析，这个时候得把每一个token的行号记住。第二遍分析是不带歧义的语法分析，目标是把所有的函数头抽取出来，然后组成一个全局符号表。第三遍分析就是对函数体里面的语句做带歧义的语法分析了。因为Tinymoe允许你定义变量，所以符号表肯定是一边分析一边修改的。于是对于“print sum from 1 to 100”这一句，如果你没有发现“phrase sum from (lower bound) to (upper bound)”和“sentence print (message)”，那根本无从下手。

还有另一个例子：

module exception handling

…

phrase main

try

do something bad

catch

print “bad thing happened”

end

end

当语法分析做到“try”的时候，因为发现存在**try函数的定义**，所以Tinymoe知道接下来的“do something bad”属于调用try这个块函数所需提供的代码块里面的代码。接下来是“catch”，Tinymoe怎么知道catch是接在try后面，而不是放在try里面的呢？这仍然是由于**catch函数的定义**告诉我们的。关于这方面的语法知识可以[点击这里查看](https://github.com/vczh/tinymoe/wiki/Category)。

正因为如此，我们需要首先知道函数的定义，然后才能分析函数体里面的代码。虽然这在一定程度上造成了Tinymoe的语法分析复杂度的提升，但是其复杂度本身并不高。比C++简单就不说了，就算是C、C#和Java，由于其语法元素太多，导致**不需要多次分析**所降低的复杂度被完全的抵消，结果跟实现Tinymoe的语法分析器的难度不相上下。

其次，**CPS变换后的代码需要特殊处理，否则直接执行容易导致call stack积累的没用的东西过多**。因为Tinymoe可以自定义操作符，所以操作符跟C++一样在编译的时候被转换成了函数调用。每一个函数调用都是会被CPS变换的。尽管每一行的函数调用次数不多，但是如果你的程序油循环，**循环是通过递归来描述**（而不是实现，由于CPS变换后Tinymoe做了优化，所以不存在实际上的递归）的，如果直接执行CPS变换后的代码，算一个1加到1000都会导致stack overflow。可见其call stack里面堆积的closure数量之巨大。

我在做Tinymoe代码生成的实验的时候，为了简单我在单元测试里面直接产生了对应的C#代码。一开始没有处理CPS而直接调用，程序不仅慢，而且容易stack overflow。但是我们知道（其实你们以后才会知道），CPS变换后的代码里面几乎所有的call stack项都是浪费的，因此我把整个在生成C#代码的时候修改成，如果需要调用continuation，就返回**调用continuation的语句组成的lambda表达式**，在最外层用一个循环去驱动他直到返回null为止。这样做了之后，就算Tinymoe的代码有递归，call stack里面也不会因为递归而积累call stack item了。于是生成的C#代码执行飞快，而且无论你怎么递归也永远不会造成stack overflow了。这个美妙的特性几乎所有语言都做不到，啊哈哈哈哈哈。

当然这也是有代价的，因为本质上我只是把保存在stack上的context转移到heap上。不过多亏了.net 4.0的强大的background GC，这样做丝毫没有多余的性能上的损耗。当然这也意味着，一个高性能的Tinymoe虚拟机，需要一个牛逼的垃圾收集器作为靠山。context产生的closure在函数体真的被执行完之后就会被很快地收集，所以CPS加上这种做法并不会对GC产生额外的压力，所有的压力仍然来源于你自己所创建的数据结构。

第三，**Tinymoe需要动态类型语言的类型推导**。当然你不这么做而把Tinymoe的程序当javascript那样的程序处理也没有问题。但是我们知道，正是因为V8对javascript的代码进行了类型推导，才产生了那么优异的性能。因此这算是一个优化上的措施。

最后，**Tinymoe还需要跨过程分析和对程序的控制流的化简**（譬如continuation转状态机等）。目前具体怎么做我还在学习当中。不过我们想，既然repeat函数是通过递归来描述的，那我们能不能通过对所有代码进行inter-procedural analyzing，而发现诸如

repeat for 3 times

do something good

end

就是一个循环，从而生成用真正的循环指令（譬如说goto）呢？这个问题是个很有意思的问题，我觉得我如果可以通过学习静态分析从而解决它，不进我的能力会得到提升，我对你们的科普也会做得更好。

## 后记