程序语言与它们的工具

程序语言与它们的工具

谈论了这么多程序语言的事情，说得好像语言的好坏就是选择它们的决定性因素。然而我一直没有提到的一个问题是，“程序语言”和“程序语言工具”的设计，其实完全是两码事。一个优秀的程序语言，有可能由于设计者的忽视或者时间短缺，没有提供良好的辅助工具。而一个不怎么好的程序语言，由于用的人多了，往往就会有人花大力气给它设计工具，结果大大的提高了易用性和程序员的生产力。我曾经提到，程序语言其实不是工具，它们是像木头，钉子，胶水一样的材料。如果有公司做出非常好的胶水，粘性极强，但它的包装不好，一打开就到处乱跑，弄得一团糟。你是愿意买这样的胶水还是稍微差一点但粘性足够，包装设计合理，容易涂抹，容易存储的呢？我想大部分人会选择后者，除非后者的粘性实在太弱，那样的话包装再好都白搭。

这就是为什么虽然我这么欣赏 Scheme，却没有用 Scheme 或者 Racket 来构造 PySonar 和 RubySonar，甚至没有选择 Scala 和 Clojure，而是“臭名昭著”的 Java。这不只是因为 PySonar 最初的代码由于项目原因是用 Java 写的，而且因为 Java 正好有足够的表达能力，可以实现这样的系统，但是最重要的其实是，Java 的工具非常成熟和迅捷。很难想象如果缺少了 Eclipse 我还能在三个月内做出像 PySonar 那样的东西。而现在我只用了一个月就做出了 RubySonar，其中很大的功劳在于 IntelliJ。这些 IDE 的跳转功能，让我可以在代码中自由穿梭。而它们的 refactor 功能，让我不必再为变量的命名而烦恼，因为只要临时起个不重复的名字就行，以后改起来小菜一碟。另外我还经常使用这些 IDE 里面的 debugger，利用它们我可以很方便的找到 bug 的起因。PySonar2 在有一段时间变得很慢，看不出是哪里出了问题。最后我下载了一个 JProfiler 试用版，很快就发现了问题的所在。如果这问题出现在 Scheme 代码里面，恐怕就要费很多功夫才能找到，因为 Scheme 没有像 JProfiler 那样的工具。

但这并不等于说学习 Scheme 是没有用处的。恰恰相反，Scheme 的知识在任何时候都是非常有用的。一个只学过 Java 的程序员基本上是不可能写出我那样的 Java 代码的。虽然那看起来是 Java，但是其实 Scheme 的灵魂已经融入到其中了。我从 Scheme 学到的知识不但让我知道 Java 可以怎么用，而且让我知道 Java 本身是如何被造出来的。我知道 Java 哪些地方是好的，哪些地方是不好的，从而能够择其善而避其不善。我的代码没有用任何的“Java 设计模式”，也没有转弯抹角的重载。

其实我有空的时候在设计和实现自己的语言（由于缺乏想象力，暂命名为 Yin），它的实现语言也在最近换成了 Java。Yin 的语法接近于 Scheme，好像理所当然应该用 Scheme 或者 Racket 来实现。有些人可能已经看到了我 GitHub 上面的第一个 prototype 实现（项目已经进入私密状态）用的是 Typed Racket。Racket 在很大程度上是比 Java 好的语言，然而它却有一个让我非常恼火的问题，以至于最后我怀疑自己能否用它顺利实现自己的语言。

这个问题就是，当运行出现错误的时候，Racket 不告诉我出错代码的具体行号，甚至出错的原因都不说清楚。我经常看到这样一些出错信息：

```
“函数调用参数个数错误”
“变量 a 没有定义，位于 loop 处”
```

只说是函数调用，函数叫什么名字不说。只说是 loop，文件里那么多 loop，到底是哪一个不知道。出错信息里面往往有很多别的垃圾信息，把你指向 Racket 系统里面的某一个文件。有时候把代码拷贝进 DrRacket 才能找到位置，可是很多时候甚至 DrRacket 都不行。每当遇到这些就让我思路被打断很长时间，导致代码质量的下降。

其它的 Scheme 实现也有类似的问题，像 Petite Chez 这样的就更加严重，只有商业版的 Chez Scheme 会好一些，所以这里不只是小小的批评一下。这种对工具设计的不在意心理，在 Lisp 和 Scheme 等函数式语言的社区里非常普遍。每当有人抱怨它们出错信息混乱，没有 debugger，没有基本的静态检查，铁杆 Schemer 们就会鄙视你说：“Aziz 说得好，我从来不 debug，因为我从来不写 bug。”“函数式语言编程跟普通语言不一样。你要先把小块的代码调试好了，问题都找到了，再组合起来。”“当程序有问题却找不到在哪里的时候，说明我思路混乱，我就把它重写一遍……”我很无语，天才就是这样被传说出来的 :)

除了由于高傲，Scheme 不提供出错位置的另外一个重要原因，其实是因为它的宏系统。由于 Scheme 的核心非常小，被设计为可以扩展成各种不同的语言，所以绝大部分的代码其实是由宏展开而成的。而由于 Scheme 的宏可以包含非常复杂的代码变换（比 C 语言的宏要强大许多），如果被展开的代码出了问题，是很难回溯找到程序员自己写的那块代码的。即使找到了也很难说清楚那块代码本来是什么东西，因为编译器看到的只是经过宏展开后的代码。如果实现者为了图简单没有把原来的位置信息存起来，那就完全没有办法找到了。这问题有点像有些 C++ 编译器给模板代码的出错信息。

所以出现这样的问题，不仅仅是语言设计者的心态问题，而且是语言自己的设计问题。我觉得 Lisp 的宏系统其实是一

个多余的东西，带来的麻烦多于好处。一个语言应该是拿来用的，而不是拿来扩展的。如果连最基本的报错信息都因此不能准确定位，扩展能力再强又有什么意义呢？所以强调一个语言可以扩展甚至变成另外一种语言，其实是过度抽象。一个设计良好的语言应该基本上不需要宏系统，所以 Yin 语言的语法虽然像 Lisp，但我不会提供任何宏的能力。而且由于以上的经历，Yin 语言从一开头就为方便工具的设计做出了努力。