Lisp 已死，Lisp 万岁！

有一句古话，叫做“国王已死，国王万岁！”它的意思是，老国王已经死去，国王的儿子现在继位。这句话的幽默，就在于这两个“国王”其实指的不是同一个人，而你咋一看还以为它自相矛盾。今天我的话题仿效了这句话，叫做“Lisp 已死，Lisp 万岁！”希望到最后你会明白这是什么意思。

首先，我想总结一下 Lisp 的优点。你也许已经知道，Lisp 身上最重要的一些优点，其实已经“遗传”到了几乎每种流行的语言身上（Java，C#，JavaScript，Python, Ruby，Haskell，……）。由于我已经在其他博文里详细的叙述过其中一些，所以现在只把这些 Lisp 的优点简单列出来（关键部分加了链接）：

- Lisp 的语法是世界上最精炼，最美观，也是语法分析起来最高效的语法。这是 Lisp 独一无二的，其他语言都没有的优点。有些人喜欢设计看起来很炫的语法，其实都是自找麻烦。为什么这么说呢，请参考这篇《谈语法》。
- Lisp 是第一个可以在程序的任何位置定义函数，并且可以把函数作为值传递的语言。这样的设计使得它的表达能力非常强大。这种理念被 Python，JavaScript，Ruby 等语言所借鉴。
- Lisp 有世界上最强大的宏系统（macro system）。这种宏系统的表达力几乎达到了理论所允许的极限。如果你只见过 C 语言的“宏”，那我可以告诉你它是完全没法跟 Lisp 的宏系统相提并论的。
- Lisp 是世界上第一个使用垃圾回收（garbage collection）的语言。这种超前的理念，后来被 Java，C# 等语言借鉴。

想不到吧，现代语言的很多优点，其实都是来自于 Lisp — 世界上第二古老的程序语言。所以有人才会说，每一种现代语言都在朝着 Lisp 的方向“进化”。如果你相信了这话，也许就会疑惑，为什么 Lisp 今天没有成为主流，为什么 Lisp Machine 会被 Unix 打败。其实除了商业原因之外，还有技术上的问题。

早期的 Lisp 其实普遍存在一个非常严重的问题：它使用 dynamic scoping。所谓 dynamic scoping 就是说，如果你的函数定义里面有“自由变量”，那么这个自由变量的值，会随着函数的“调用位置”的不同而发生变化。

比如下面我定义一个函数 f，它接受一个参数 y，然后返回 x 和 y 的积。

```
(setq f
      (let ((x 1))
        (lambda (y) (* x y))))
```

这里 x 对于函数 (lambda (y) (* x y)) 来说是个“自由变量”（free variable），因为它不是它的参数。

看着这段代码，你会很自然的认为，因为 x 的值是 1，那么 f 被调用的时候，结果应该等于 (* 1 y)，也就是说应该等于 y 的值。可是这在 dynamic scoping 的语言里结果如何呢？我们来看看吧。

（你可以在 emacs 里面试验以下的结果，因为 Emacs Lisp 使用的就是 dynamic scoping。）

如果我们在函数调用的外层定义一个 x，值为 2：

```
(let ((x 2))
  (funcall f 2))
```

因为这个 x 跟 f 定义处的 x 的作用域不同，所以它们不应该互相干扰。所以我们应该得到 2。可是，这段代码返回的结果却为 4。

再来。我们另外定义一个 x，值为 3：

```
(let ((x 3))
  (funcall f 2))
```

我们的期望值还是 2，可是结果却是 6。

再来。如果我们直接调用：

```
(funcall f 2)
```

你想这次总该得到 2 了吧？结果，出错了：

```
Debugger entered--Lisp error: (void-variable x)
  (* x y)
  (lambda (y) (* x y))(2)
  funcall((lambda (y) (* x y)) 2)
  eval_r((funcall f 2) nil)
  eval-last-sexp-1(nil)
```

```
  eval-last-sexp(nil)
  call-interactively(eval-last-sexp nil nil)
```

看到问题了吗？f 的行为，随着调用位置的一个“名叫 x”的变量的值而发生变化。而这个 x，跟 f 定义处的 x 其实根本就不是同一个变量，它们只不过名字相同而已。这会导致非常难以发现的错误，也就是早期的 Lisp 最令人头痛的地方。好在现在的大部分语言其实已经吸取了这个教训，所以你不再会遇到这种让人发疯的痛苦。不管是 Scheme, Common Lisp, Haskell, OCaml, Python, JavaScript…… 都不使用 dynamic scoping。

那现在也许你了解了，什么是让人深恶痛绝的 dynamic scoping。如果我告诉你，Lisp Machine 所使用的语言 Lisp Machine Lisp 使用的也是 dynamic scoping，你也许就明白了为什么 Lisp Machine 会失败。因为它跟现在的 Common Lisp 和 Scheme，真的是天壤之别。我宁愿写 C++，Java 或者 Python，也不愿意写 Lisp Machine Lisp 或者 Emacs Lisp。

话说回来，为什么早期的 Lisp 会使用 dynamic scoping 呢？其实这根本就不是一个有意的“设计”，而是一个无意的“巧合”。你几乎什么都不用做，它就成那个样子了。这不是开玩笑，如果你在 emacs 里面显示 f 的值，它会打印出：

```
'(lambda (y) (* x y))
```

这说明 f 的值其实是一个 S 表达式，而不是像 Scheme 一样的“闭包”（closure）。原来，Emacs Lisp 直接把函数定义处的 S 表达式 ‘(lambda (y) (* x y)) 作为了函数的“值”，这是一种很幼稚的做法。如果你是第一次实现函数式语言的新手，很有可能就会这样做。Lisp 的设计者当年也是这样的情况。

简单倒是简单，麻烦事接着就来了。调用 f 的时候，比如 (funcall f 2)，y 的值当然来自参数 2，可是 x 的值是多少呢？答案是：不知道！不知道怎么办？到“外层环境”去找呗，看到哪个就用哪个，看不到就报错。所以你就看到了之前出现的现象，函数的行为随着一个完全无关的变量而变化。如果你单独调用 (funcall f 2) 就会因为找不到 x 的值而出错。

那么正确的实现函数的做法是什么呢？是制造“闭包”(closure)。这也就是 Scheme，Common Lisp 以及 Python，C# 的做法。在函数定义被解释或者编译的时候，当时的自由变量（比如 x）的值，会跟函数的代码绑在一起，被放进一种叫做“闭包”的结构里。比如上面的函数，就可以表示成这个样子：(Closure ‘(lambda (y) (* x y)) ‘((x . 1)))。

在这里我用 (Closure ...) 表示一个“结构”（就像 C 语言的 struct）。它的第一个部分，是这个函数的定义。第二个部分是 ‘((x . 1))，它是一个“环境”，其实就是一个从变量到值的映射（map）。利用这个映射，我们记住函数定义处的那个 x 的值，而不是在调用的时候才去瞎找。

我不想在这里深入细节。如果你对实现语言感兴趣的话，可以参考我的另一篇博文《怎样写一个解释器》。它教你如何实现一个正确的，没有以上毛病的解释器。

与 dynamic scoping 相对的就是“lexical scoping”。我刚才告诉你的闭包，就是 lexical scoping 的实现方法。第一个实现 lexical scoping 的语言，其实不是 Lisp 家族的，而是 Algol 60。“Algol”之所以叫这名字，是因为它的设计初衷是用来实现算法（algorithm）。其实 Algol 比起 Lisp 有很多不足，但在 lexical scoping 这一点上它却做对了。Scheme 从 Algol 60 身上学到了 lexical scoping，成为了第一个使用 lexical scoping 的“Lisp 方言”。9 年之后，Lisp 家族的“集大成者” Common Lisp 诞生了，它也采用了 lexical scoping。看来英雄所见略同。

你也许发现了，Lisp 其实不是一种语言，而是很多种语言。这些被人叫做“Lisp 家族”的语言，其实共同点只是它们的“语法”：它们都是基于 S 表达式。如果你因此对它们同样赞美的话，那么你赞美的其实只是 S 表达式，而不是这些语言本身。因为一个语言的本质应该是由它的语义决定的，而跟语法没有很大关系。你甚至可以给同一种语言设计多种不同的语法，而不改变这语言的本质。比如，我曾经给 TeX 设计了 Lisp 的语法，我把它叫做 SchTeX（Scheme + TeX）。SchTeX 的文件看起来是这个样子：

```
(documentclass article (11pt))
(document
  (abstract (...))
  (section (First Section)
      ... )
  (section (Second Section)
      ... )
)
```

很明显，虽然这看起来像是 Scheme，本质却仍然是 TeX。

所以，因为 Scheme 的语法使用 S 表达式，就把 Scheme 叫做 Lisp 的“方言”，其实是不大准确的做法。Scheme 和 Emacs Lisp，Common Lisp 其实是三种不同的语言。Racket 曾经叫做 PLT Scheme，但是它跟 Scheme 的区别日益增加，以至于现在 PLT 把它改名叫 Racket。这是有他们的道理的。

所以，你也许明白了为什么这篇文章的标题叫做“Lisp 已死，Lisp 万岁！” 因为这句话里面的两个 “Lisp”其实是完全不同的语言。“Lisp 已死”，其实是说 Lisp Machine Lisp 这样的 Lisp，由于严重的设计问题，已经死去。而“Lisp 万岁”，是说像 Scheme，Common Lisp 这样的 Lisp，还会继续存在。它们先进于其它语言的地方，也会更多的被

借鉴，被发扬广大。

（其实老 Lisp 的死去还有另外一个重要的原因，那就是因为早期的 Lisp 编译器生成的代码效率非常低下。这个问题我留到下一篇博文再讲。）