序言

约翰麦卡锡和他的学生于1958年开始Lisp的首个实现工作. Lisp是Fortran之后仍然在使用的最古老的语言.[1] 更值得一提的是,它仍然处在编程语言技术的最前沿. 懂Lisp的程序员会告诉你,这种语言有某种东西使得它与众不同.

部分使Lisp与众不同的原因是它被设计成能够自己演变. 你能用Lisp定义新的Lisp操作符. 随着新的抽象概念越来越流行(比如面向对象编程),我们总是发现在Lisp中实现它们很容易. 就象DNA, 这样的语言不会过时.

0.1 新的工具

为什么要学Lisp? 因为它能让你做其它语言中不能做的事情. 如果你仅想写一个函数返回小于n的非负整数的总和, 那么用Lisp和C是差不多的:

```
; Lisp                    /* C */
(defun sum (n)            int sum(int n){
  (let ((s 0))              int i, s = 0;
    (dotimes (i n s)        for(i = 0; i < n; i++)
      (incf s i))))           s += i;
                             return(s);
                           }
```

如果你只想做这样简单的事,那用哪种语言是无关紧要的. 假设你想写一个函数接受数n, 返回一个*函数*: 它把n增加到它的自变量上去:

```
; Lisp
(defun addn (n)
  #'(lambda (x)
      (+ x n)))
```

在C语言中addn会是怎样的? 你根本写不出来.

你可能会想,谁会想要做这样的事情? 编程语言教你不要试图去做它们不提供的事. 你用某种语言写程序的时候,你得用那种语言思考问题,而且想要你不能描述的东西是很因难的. 我刚开始写程序—用Basic—的时候,我并不想要递归,因为我根本不知道有这么回事. 我用Basic思考. 我只有递推算法的概念,为什么我想要递归呢?

[1] McCarthy, John. Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I. CACM, 3:4 (April 1960),pp. 184-195.

如果你不渴望逻辑*封装*(上面的例子中就使用了它),请相信, 目前Lisp程序员一直在使用它. 很难找到任意长度的Lisp代码不利用封装的. 到112页你会自己使用它.

封装只是我们不能在其它语言中找到的抽象概念之一. 另外一个更有价值的Lisp 的独特特征是Lisp程序是用Lisp的数据结构来表示的. 这意味着你可以编写能写程序的程序. 人们真的需要这个吗? 是的—它们叫做宏, 有经验的程序员也一直在使用它. 到173页你就可以写自己的宏了.

有了宏,封装,和运行时类型,Lisp超过了面向对象编程. 如果你理解上面这句话, 也许你不需要读这本书了. 你得相当理解Lisp才能看到为什么这不是假话. 但这不仅仅是说说而已. 这是一个重要的论点,它的证明在第17章用代码明确地写出来了.

第2章到第13章会逐渐地介绍所有你需要理解第17章的代码的概念. 你的努力会有所回报,打个比方: 你在C++中编程会感觉到窒息,就象有经验的C++程序员用Basic编程会感到窒息一样. 更令人鼓舞的可能是,如果我们思考一下这种感觉从何而来. Basic令习惯C++的人窒息是因为有经验的C++程序员知道有无法在Basic 中表达的技术. 同样地, 学习Lisp不仅是让你学会了一门新的语言—它会教你新的更强大的考虑程序的方法.

0.2 新的技术

就象上一节提到的,Lisp提供其它语言不能提供的工具. 但远远不止这些. 单独地讲,伴随Lisp的新东西—自动内存管理,显式的类型,封装,等等—每一项都使编程更加容易. 它们合在一起组成了一个临界值使得产生一种新的编程方法成为可能.

Lisp被设计成可扩展的: 它让你定义自己的操作符. 因为Lisp是由和你的程序一样的函数和宏构成的. 所以扩展Lisp和写一段Lisp程序一样容易. 事实上它是这么容易(和有用)以至于扩展这种语言成了标准的实践. 当你写程序向下逼近语言的时候, 你也在构造语言向上逼近你的程序. 你不但至上而下也至下而上工作.

几乎所有的程序都可以从让语言裁剪得以适应自己的需要中获得好处, 但程序越复杂, 自下而上设计法就显得越有价值. 一个自下而上的程序可以写成一系列的层, 每一层担当上一层的编程语言. TEX是最早用这种方法写的程序之一. 你可以用任何语言自下而上设计程序,但对这种风格来说,Lisp是最最自然的运载工具.

自下而上编程法自然而然地产生可扩展的软件. 如果你始终坚持自下而上编程原则直到你程序的最高层, 那么这一层就成为用户的可编程语言. 可扩展性的思想深深地扎根于Lisp, 使得它成为编写可扩展软件的理想语言. 1980年代最成功的程序中的三个提供了Lisp作为扩展语言: Gnu Emacs, Autocad, 和Interleaf.

自下而上也是获得可重复使用软件的最好方法. 编写可复用软件的本质是把共性从个性中分离出来. 而自下而上法天然就创造了这种分离. 不是把你所有的精力花在编写一个巨大的应用上, 而是用部分精力去构造一种语言, 用部分精力(比例相对小些)在这种语言上面写你的应用. 和应用有关的特性都集中在最上层, 以下的层可以组成一种语言, 用来编写和这个应用类似的应用—还有什么会比编程语言更具可复用性的呢?

Lisp让你不仅写出更复杂的程序, 而且写得更快. Lisp程序往往很短—这种语

言给了你更大的概念, 所以你不需要用多少. 就象Frederick Brooks指出的,写程序所花的时间通常取决于它的长度. 因此这个单独的事实意味着编写Lisp程序花的时间较少. 这种效应被Lisp的动态特性放大: 在Lisp中, 编辑–编译–测试循环是如此之短, 以致编程是实时的.

更大的抽象和交互式的环境能改变各种组织开发软件的方式. 术语"快速原型" 描述了一种首先在Lisp中使用的编程法: 在Lisp中, 你可以用比写一个规格说明更少的时间写出一个原型来, 而且这种原型如此抽象, 可以作为一个比用英语写的更好的规格说明. 还有Lisp让你做出从原型到产品软件的平*的转变. 当编写Lisp程序时考虑到速度,经过现代编译器的编译,它们和任何用其它高级语言写的程序跑得一样快.

除非你相当熟悉Lisp,本序言象是一堆夸夸其谈和没有意义的声明. Lisp胜过面向对象编程? 你构造语言逼近你的程序? Lisp编程是实时的? 这些话是什么意思? 现在这些说法就象一些空的湖泊. 随着你学到更多实际的Lisp特点,看到更多的有指导意义的程序范例, 它们就会被实际经验装满, 而有了明确的形状.

0.3 新的方法

本书的目标之一是不仅仅解释Lisp语言,而是一种由Lisp造成的新方法. 这是一种你在将来会见得更多的方法. 随着编程环境变得更强大, 编程语言变得更抽象, Lisp编程风格正逐渐取代旧的计划–然后–实现的模式.

在旧的模式中, 错误(bug)永远不该出现. 事先辛苦作出的周密规格说明被期望来保证程序完美地运行. 理论上听起来不错. 不幸的是, 规格说明都是人写的, 也是由人来实现的. 实际的结果是, 计划–然后–实现方法工作得不太好.

作为OS/360工程的经理, Frederick Brooks非常熟悉这种传统的方法. 他也非常熟悉它的后果:

> 任何OS/360的用户很快意识到它还应该做得多好. . . 而且产品已经晚了, 它用了比原计划多的内存,成本是估计的好几倍,而且运行一直不太好,直到第一个版本之后的好几个版本.[2]

这就是对那个时代最成功系统之一的描述.

旧模式的问题是它忽略人的局限性. 在旧模式中,你打赌规格说明不会有严重的缺陷,实现它们是一桩把它们翻译成代码的简单事情. 经验证明这实在是一个糟糕的打赌. 如果赌规格说明会搞错, 代码充满错误会更保险些.

这其实就是新编程模式所假定的. 它设法尽量降低错误的成本,而不是希望人们不犯错误. 错误的成本是改正它所花的时间. 使用强大的语言和优秀的编程环境, 这种成本会极大地降低. 编程设计可以较少地依赖于计划,更多地依赖于探索.

计划是一种必要的恶. 它是对风险的反应: 工作越是危险,预先做好计划也就越重要. 强大的工具降低了风险, 也降低了计划的需要. 程序的设计可以从实现它的经验中受益—这可能是最有用的信息来源.

---

[2]Brooks, Frederick P. The Mythical Man-Month. Addison-Wesley, Reading (MA), 1975, p. 16.

快速原型法不仅是一种更快或更好地写程序的方法, 有些程序可能不用此法就根本写不出来.

即使是最有雄心的人也会在巨大的工作面前退缩. 如果能使自己相信某件事不用费很大劲儿(哪怕从表面上看),从它做起就会容易些. 这就是为什么这么多大事是从小事开始的. 快速原型法让我们从小事做起.

Lisp风格自1960年代以来一直朝这个方向演化. 你在Lisp中可以如此快速地写出原型,以至于你能经历好几个设计和实现的循环, 而在旧的模式中,你可能刚写完规格说明. 你不必过于担心设计缺陷,因为你会更早地发现它们. 你也不必过于担心错误(bugs). 当你用函数风格编程,错误只有局部的影响. 当你使用一种很抽象的语言,某些错误(例如孤悬指针)就永远不会出现了,而剩下的错误很容易找, 因为你的程序更短了. 当你有交互式开发环境时,你能立即修正错误,不必经历编辑,编译,测试的漫长过程.

Lisp风格这么发展是因为它结出了成果. 听起来有点奇怪,少些计划意味着更好的设计. 技术史上相似的例子不胜枚举. 一个类似的事件发生在十五世纪的绘画界里. 在油画颜料流行之前,画家用一种叫蛋彩的物质作画,它不能混合或覆盖. 出错的高昂代价使画家变得保守. 后来随着油画颜料的出现,作画风格有了极大的改变. 油画颜料"允许第二次思考."[3]它为因难主题的处理,例如人物画等提供了决定性的有利条件.

新介质不仅使画家作画更容易了. 它使新的更有抱负的作画方式成为可能. Janson写到:

> 如果没有油画颜料,佛兰德大师们的征服可见的现实的口号就会大打折扣. 于是从技术的角度来看,他们也当之无愧地称得上"现代绘画之父",因为油画颜料从此成为画家的基本介质.[4]

作为一种材料,蛋彩并不比油画颜料逊色. 但油画颜料的弹性给了想象力更大的空间—这是决定性的因素.

程序设计正经历着相似的变革. 新的介质是"面向对象的动态语言"—即Lisp. 不是说我们所有的软件在几年内都要用Lisp来写. 从蛋彩到油彩的过渡也不是一夜之间完成的; 油彩开始只在一些领风气之先的艺术中心流行,而且经常结合着蛋彩使用. 我们现在似乎正处在这个阶段. Lisp在大学,实验室和一些处于前沿的公司中使用. 同时从Lisp那儿借用的思想越来越多地出现在主流中: 交互式开发环境,无用单元收集,运行时类型仅是其中几例.

强大的工具正在减轻探索的风险. 这对程序员来说是个好消息,因为它意味着我们能进行更有抱负的项目. 油彩的应用的确有这样的效果. 它被采用后的时期正是绘画的黄金年代. 类似的迹象已经发生在程序设计领域.

---

[3] Murray, Peter and Linda. The Art of the Renaissance. Thames and Hudson, London, 1963, p. 85.

[4] Janson, W. J. History of Art, 3rd Edition. Abrams, New York, 1986, p. 374.

这个类比只适用于作在画布上的画. 画在墙上的画仍旧用壁画颜料. 我也不是说绘画风格是由技术变迁驱动的; 相反的说法可能更接近事实.