On Lisp: Common Lisp 高级编程技术[1]

[美] Paul Graham 著　　Chun Tian (binghe),
Kov Chai 译

May 30, 2008

[1]原书站点: http://www.paulgraham.com/onlisp.html

译者序

《*On Lisp*》不是一本 Lisp 的入门教材，它更适合读过《*ANSI Common Lisp*》或者《*Practical Common Lisp*》的 Lisp 学习者。它对 Lisp 宏本身及其使用做了非常全面的说明，同时自底向上的编程思想贯穿全书，这也是《*On Lisp*》得名的原因，即，基于 Lisp，扩展 Lisp。

原作者 Paul Graham 同时也是 *ANSI Common Lisp* 一书的作者。

我是一个 Lisp 程序员，Linux 系统管理员。将现存的几部 Common Lisp 重要著作翻译成中文，介绍给国内的广大计算机专业学生和编程爱好者，是我多年来的心愿。希望这本《*On Lisp*》中文版能对国内的读者有所帮助。《*On Lisp*》的成书年代早在 1994 年 ANSI Common Lisp 标准发布以前，该书中使用了许多古老的 Lisp 操作符，原书中的一些代码已经无法在最新的 Common Lisp 平台上执行了。而我翻译本书的原则是希望读者在阅读此书同时直接能够测试书中的代码，以达到最快的学习目的。对于 Lisp 初学者来说，Lisp 语言庞大复杂的发展历史固然重要但对这些知识的学习可以不必操之过急。所以，在中文翻译版里，所有源代码都被改成符合现行 Common Lisp 标准的形式，这在本书的后半部分——涉及更高级技术的部分——可以多次看到，凡是译者修改原文的地方都回给出明确的脚注和相关解释。欢迎熟悉 Common Lisp 的读者通过邮件就这些修改的地方和我讨论。

本书的 LaTeX 源文件可以在下面的 Subversion 资源库找到：

```
https://cl-net-snmp.svn.sourceforge.net/svnroot/cl-net-snmp
/onlisp-cn/trunk
```

欢迎提交各类补丁。

任何关于本书的译文或者其他跟 Lisp 有关的问题，欢迎在水木社区函数型编程语言 (FuncProgram) 板上讨论。

我要特别感谢来自 AMD/ATI 的 Kov Chai[1] 同学，他对本书进行了细致的校对，并独立翻译了第 5、6、25 和 25.7 章，Kov Chai 还主导了本书的 LaTeX排版工作。

留学日本的 Lisp 程序员 Jianshi Huang[2] 同学是我最初翻译本书时的合作校对人，他对本书前三章进行了仔细的校对并给出了大量专业的建议和关键文字的翻译。

来自浙江大学微软技术俱乐部 (MSTC) 的 Emily Lee 同学，她以一个 Lisp 初学者的角度仔细阅读了本书的前四章，并提出了大量文法层面的改进意见。

[1] E-mail: tchaikov@gmail.com
[2] E-mail: jianshi.huang@gmail.com

感谢水木社区和浙大飘渺水云间这两个 BBS 上的热心网友。他们是本书的第一批读者。感谢水木的 Dieken, texlive 等网友给我发来一些翻译的修改意见。

感谢可爱的钱琳 (PreTTyHeLL) 同学，2007 年里对她的短暂追求造就了我翻译此书——这项伟大的毅力考验——的最初不良动机。幸运的是，最后我认识到“一见钟情”对我来说是不可能的事情，放弃这个念头，从此全身心投入到工作和学习之中。

感谢我已经离异的父母，是他们在我的早年无意中培养了我冷酷无情的性格和强烈的逆反心理。后者对我来说尤为重要，在不良社会风气盛行的今天，对社会主流的逆反心态反而铸就了我神一般的行事风格。缺少温暖的童年时代，使我在长大以后更愿意关心和帮助他人，也因此结交了许多朋友。

感谢我的母校浙江大学。我在紫金港校区图书馆里度过了大部分周末时光。本书的翻译工作有很多就是在图书馆的工具书阅览室里完成的，那里有很多英汉辞典可供翻译时参考。

最后，我要感谢网易公司，尤其是网易杭州研究院系统管理组的同事们。在我工作了两年之后，我可以有更多时间学习感兴趣的计算机领域，自由安排自己的时间。翻译本书的大部分时间来自于此。

Chun TIAN (binghe)[3]
NetEase.com, Inc.

[3]E-mail: binghe.lisp@gmail.com

λ

- 2006 年 8 月 1 日，LispWorks 5.0 发布。
- 2008 年 3 月 28 日，LispWorks 5.1 发布。

前言

本书适用于那些想成为更好的 Lisp 程序员的人. 本书假设读者已经熟悉 Lisp, 但不要求有广泛的编程经验. 最初几章里会包含一些知识回顾. 我希望这些章节也会让有经验的 Lisp 程序员感兴趣, 因为它们以新的视角展示了熟知的主题.

通常很难用一句话来说清楚一门编程语言的本质, 但 John Foderato 的这句描述已经很到位了:

> Lisp 是一门可编程的编程语言. (**Lisp is a programmable programming language.**)

当然 Lisp 的特性绝不止这些, 但这种随心所欲使用 Lisp 的能力, 在很大程度上正是 Lisp 专家和新手的区别之处. 和其他程序员自上而下写程序不同, 资深 Lisp 程序员从语言自底向上构造他们的程序[4].

本书教授如何使用自底向上的编程风格, 因为这是 Lisp 的强项.

自底向上的设计 (Bottom-up Design)

随着软件复杂度的增长, 自底向上设计的重要性也正在日益提高. 今天的程序可能不得不面对极其复杂甚至开放式的需求. 在这种情况下,传统的自上而下方法有时会失效. 一种新的编程风格由此诞生, 它和当前大部分计算机科学课程的思路截然不同: 一个自底向上的程序由一系列的层写成, 每一层都作为更上一层的编程语言. X Window 和 TEX 就是这种程序设计风格的范例.

本书同时有两重主题: 首先, 就是说 Lisp 对于以自底向上的编程风格来说是一种自然的语言, 同时, 自底向上的编程风格也是编写 Lisp 程序的一种自然的方式. *On Lisp* 因此将吸引两类读者. 对于那些有兴趣编写可扩展程序的人, 本书将告诉你如果有了正确的语言你能做什么. 对于 Lisp 程序员来说, 本书提供一套关于怎样使用 Lisp 才能发挥其最大优势的实践性说明.

本书选用现在的这个书名是为了强调自底向上编程对于 Lisp 的重要性. 你不再仅仅是用 Lisp 编写程序, *在 Lisp 之上 (On Lisp)*, 你可以构造自己的语言, 然后再用这个语言来写程序.

理论上用任何语言都可以写出的自底向上风格的程序, 但 Lisp 对于这种编程风格来说是最自然的载体. 在 Lisp 里, 自底向上设计不是一种专用于少见的

[4]译者注: 原文是"As well as writing their programs down toward the language, experienced Lisp programmers build the language up toward their programs."

大型或困难的程序的特别技术. 任何程度的程序都可以部分地用这种方式编写. Lisp 从一开始就是种可扩展的语言. 语言本身基本上就是一个 Lisp 函数的集合, 这些函数和你自己定义的没有本质区别. 更进一步的是, Lisp 函数可以表达成列表, 这也是 Lisp 的数据结构. 这就意味着你可以写能生成 Lisp 代码的 Lisp 函数.

一个好的 Lisp 程序员必须懂得如何利用上述的这种可能性. 通常的途径是定义一种称为*宏*的操作符. 驾驭宏是从编写正确的 Lisp 程序到编写漂亮的程序过程中最重要的一步. 入门级 Lisp 书籍给宏留下的篇幅仅限于一个宏的简短的概述: 一个关于宏是什么的解释, 连带少许示例暗示你能用它实现一些奇妙的东西. 不过在本书里这些奇妙的东西将得到特别的重视. 本书的一个目标就是收集所有那些人们至今都很难学到的使用宏的经验.

一般入门级 Lisp 书籍都不太强调 Lisp 和其他语言的区别, 当然这是可以理解的. 它们不得不把信息传递给那些被调教成只会用 Pascal 术语来思考编程的学生们. 如果非要细究这些区别的话, 只会把问题复杂化: 例如 `defun` 虽然*看起来*像一个过程定义, 但实际上是一个程序在写另一个程序然后生成一段代码, 这段代码生成了一个函数对象然后用函数定义时给出的第一个参数作为它的索引.

本书的目的之一就是解释究竟是什么使 Lisp 不同于其他语言. 当我刚开始学 Lisp 的时候, 同等条件下, 我知道我更倾向于用 Lisp 而不是 C、Pascal 或 Fortran 来写程序. 我也知道这不只是个人好恶的问题. 但我真的决定要解释 Lisp 在某些方面是更好的语言的时候, 我就得好好准备一下了.

当某些人问 Louis Armstrong 什么是爵士乐时, 他回答说 "如果你问爵士乐是什么, 那你永远不会知道." 但他确实以一种方式回答了这个问题: 他向人们*展示*了什么是爵士乐. 同样也只有一种方式来解释 Lisp 的威力，就是演示那些对于其他语言来说极其困难甚至不可能实现的技术. 多数关于编程的书籍，包括 Lisp 编程书籍，采用的都是那些你可以用任何其它语言编写的程序. *On Lisp* 打交道的会是那类你只能用 Lisp 来写的程序. 可扩展性, 自底向上程序设计, 交互式开发, 源代码转换, 嵌入式语言——这些都是 Lisp 展示其高级特性的场合.

当然从理论上讲, 任何图灵等价的编程语言能做的事，其它任何语言都可以做到. 但这种能力和编程语言的能力却完全是两码事. 理论上任何你能用编程语言做到的事也可以用图灵机来做, 但实际上在图灵机上编程得不偿失.

所以当我说这本书是关于如何做那些其他语言不可能做到事情的时候, 我并非指数学意义上的 "不可能", 而是从编程语言的角度出发的. 这就是说, 如果你不得不用 C 来写本书中的一些程序, 你可能需要先用 C 写一个 Lisp 编译器. 举个例子, C 语言中的嵌入 Prolog—你能想象这需要多少工作量吗? 第 24 章将说明如何用 180 行 Lisp 做到这点.

尽管我希望能比单单演示 Lisp 的强大之处做得更多些。我也想解释*为何* Lisp 与众不同. 这是一个更微妙的问题，这个问题是那么难回答，它无法使

用诸如 “符号计算” 这样的术语来搪塞. 我将尽我所学，尽可能清晰地解释这些问题.

本书计划

由于函数是 Lisp 编程的基础, 本书从关于函数的章节开始. 第 2 章解释 Lisp 函数究竟是什么以及他们所提供的编程方式的可能性. 第 3 章讨论函数型编程的优点, 这是 Lisp 程序最主要的风格. 第 4 章展示如何用函数来扩展 Lisp. 第 5 章建议了一种新的抽象方式, 返回其他函数的函数. 最后, 第 6 章显示了怎样使用函数来代替传统的数据结构.

本书剩下的篇幅则更加关注宏. 因为宏本身就有更多内容, 部分是因为它们至今还没有适当的出版物. 第 7–10 章形成一个完整的宏的指导教程. 完成后你将了解一个有经验的 Lisp 程序员所知的关于宏的大多数内容: 它们如何工作; 怎样定义, 测试, 以及调试它们; 何时应该使用以及何时不应该使用宏; 宏的主要类型; 怎样写生成宏展开代码的程序; 宏风格一般如何区别于 Lisp 风格; 以及怎样检测和修复每一种影响宏的独特的问题.

随后, 第 11–18 章展示了一些可以用宏来构造的强有力的抽象. 第 11 章展示如何写典型的宏—那些创造上下文, 或者实现循环或条件判断的宏. 第 12 章解释宏在操作普通变量中的角色. 第 13 章展示宏如何通过将计算转移到编译期来使程序运行得更快. 第 14 章介绍了 anaphoric(首语重复) 宏, 可以允许你在程序里使用代词. 第 15 章展示了宏如何为第 5 章里定义的函数生成器提供一个更便利的接口. 第 16 章展示了如何使用定义宏的宏来让 Lisp 为你写程序. 第 17 章讨论读取宏 (read-macro), 以及第 18 章, 解构宏.

第 19 章开始了本书的第四部分, 转向嵌入式语言. 第 19 章通过展示同一个程序, 一个回答数据库查询的程序, 先是用解释器, 然后用真正的嵌入式语言, 来介绍这一主题. 第 20 章展示了如何将续延 (continuation) 概念引入 Common Lisp 程序, 这是一种描述延续性计算的对象. 续延是一个强有力的工具, 可以用来实现多处理和非确定性选择. 将这些控制结构嵌入到 Lisp 中的讨论分别在第 21 和 22 章. 非确定性允许你写出有先见之明的程序, 听起来就像一种不寻常力量的抽象. 第 23 和 24 章展示了两种嵌入式语言, 展示非确定性的存在意义: 一个完整的 ATN 解析器, 以及一个嵌入式 Prolog, 总共才 200 行代码.

这些程序的长短对它们本身来说并无意义. 如果你倾向于编写无法理解的代码, 无人能告诉你 200 行代码能做干什么. 关键在于, 这些程序并不是因为依赖于编程技巧才变得短小, 而是由于它们是以 Lisp 固有的, 自然的方式写成. 第 23 和 24 章的关键之处不是如何用一页代码实现 ATN 解析器或者用两页实现 Prolog, 而是想说明这些程序, 当给出它们最自然的 Lisp 实现的时候是如此的简短. 后面这两个章节的嵌入式语言用实例证明了我开始时的双重观点: Lisp 对于以自底向上的编程风格来说是一种自然的语言, 同时自底向上的编程风格

也是编写 Lisp 程序的一种自然的方式.

本书以关于面向对象编程的讨论做结, 特别讨论了 CLOS, Common Lisp 对象系统. 通过将这一主题留到最后, 我们可以更加清楚地看到, 面向对象的编程方式是一种扩展, 这种扩展植根于一些早已存在于 Lisp 的思想之上. 它是多种可以建立*在 Lisp 上*的抽象之一.

自成一章的注释开始于第 253 页. 这些注释里包括参考文献, 附加或者替换的代码, 或者是有关 Lisp 的但跟主题无关的一些描述. 注释是用页面留白上的小圆圈标注出来的, 就像这样. 另外还有一个关于包 (packages) 的附录, 在第 247 页.

正如一次纽约的观光旅游可能是一次世界上大多数文化的观光那样, 一次对 Lisp 作为可扩展编程语言的学习也能勾画出大部分 Lisp 技术. 这里描述的大多数技术通常都被 Lisp 社区所了解, 但很多内容至今也没有在任何地方有记载. 而一些问题, 例如宏的适当角色或者变量捕捉的本质, 甚至对于很有经验的 Lisp 程序员来说也只有些模糊的理解.

示例

Lisp 是个语言家族. 由于 Common Lisp 仍然是在广泛使用的方言, 本书的大部分示例都是用 Common Lisp 写的. 这一语言最初于 1984 年 Guy Steele 写的的一本出版物 *Common Lisp: the Language* (CLTL1) 里被定义. 这一定义在 1990 年该书第二版 (CLTL2) 出版以后被其取而代之了, CLTL2 可能会成为未来的 ANSI 标准.

本书包含几百个示例, 小到简单的表达式, 大到可运行的 Prolog 实现. 本书中的代码的编写, 尽量照顾到了各个细节, 使得它们可以在任何版本的 Common Lisp 上运行[5]. 有极少数示例需要用到 CLTL1 规范之外的特性, 这些示例将会正文中被明确标记出来. 最后几个章节里包括一些 Scheme 的示例代码, 这些代码也会有清楚的标记.

所有代码可以通过匿名 FTP 从 `endor.harvard.edu` 下载, 在 `pub/onlisp` 目录里. 问题和评论可以发到 `onlisp@das.harvard.edu`. [6]

致谢

写此书时, 我要特别感谢 Robert Morris 的帮助. 我经常去向他寻求建议并且每次都满载而归. 本书的一些示例代码就来自他, 包括某页的 `for` 版本, 某页的 `aand` 版本, 某页的 `match`, 某页的广度优先 `true-choose`, 以及第 ?? 节

[5] 译者注: 译文重新编写了部分代码, 以确保所有代码都可以在当前的 Common Lisp 标准下运行,并且在 SBCL, LispWorks 等主要平台下进行了测试, 所有对代码的修改会通过脚注明确标示出来.

[6] 译者注: 中文读者可以发帖到水木社区 BBS 的 `FuncProgram` 板.

的 Prolog 解释器. 事实上, 整本书都反映 (有时基本是抄录) 了过去七年来我跟 Robert 之间的对话. (谢谢你, rtm!)

我也要向 David Moon 致以特别的感谢, 他仔细地阅读了大部分手稿并且给出许多非常有用的评论. 第 12 章是按照他的建议完全重写了的, 某页关于变量捕捉的示例代码也是他提供的.

我很幸运地拥有 David Touretzky 和 Skoma Brittain 这两位技术审稿人. 某些章节就是在他们的建议下追加或者重写的. ?? 页给出的一个替代的非确定性选择操作符 (alternative true nondeterministic choice operator) 就是基于了 David Toureztky 的一个建议.

其他一些人欣然阅读了部分或全部的手稿, 包括 Tom Cheatham, Richard Draves (他在1985 年也帮助重写了 `alambda` 和 `propmacro`), John Foderaro, David Hendler, George Luger, Robert Muller, Mark Nitzberg, 以及 Guy Steele.

我感谢 Cheatham 教授以及整个哈佛, 他们提供了让我撰写此书的条件. 也感谢 Aiken 实验室的全体成员, 包括 Tony Hartman, Janusz Juda, Harry Bochner, 以及 Joanne Klys.

Prentice Hall 的工作人员干得非常出色. 我为与 Alan Apt 这位优秀的编辑和好伙伴一起共事感到幸运. 同时也感谢 Mona Pompili、Shirley Michaels，以及 Shirley McGuire 的组织工作和他们的幽默.

剑桥 Bow and Arrow 出版社的无与伦比的 Gino Lee 制作了封面. 封面上的那颗树暗示了某页上的观点.

本书使用 LaTeX 排版, 这是一种由 Leslie Lamport 在 Donald Knuth 的 TeX 基础上设计的语言, 另外使用了来自 L. A. Carr, Van Jacobson 和 Guy Steele 的宏. 插图由 John Vlissides 和 Scott Stanton 设计的 Idraw 完成. 整本书用 L. Peter Deutsch 的 Ghostscript 生成之后, 在 Tim Theisen 的 Ghostview 里预览. Chiron Inc. 公司的 Gary Bisbee 制作了能用来进行照相制版的拷贝.

我要感谢其他许多人, 包括 Paul Becker, Phil Chapnick, Alice Hartley, Glenn Holloway, Meichun Hsu, Krzysztof Lenk, Arman Maghbouleh, Howard Mullings, Nancy Parmet, Robert Penny, Gary Sabot, Patrick Slaney, Steve Strassman, Dave Watkins, Weickers 一家, 还有 Bill Woods.

最后, 我要感谢我的父母, 谢谢他们为我树立的榜样和对我的鼓励; 还有 Jackie, 他教给我那些要是我听进去他的话，就可以学到的知识.

我希望阅读此书是件乐事. 在所有我知道的语言中, Lisp 是我的最爱, 只因它是最优美的. 本书正是关于最 Lisp 化的 Lisp. 写作这本书的过程充满了乐趣, 希望你在阅读此书时能感同身受.

Paul Graham

目　录

第 1 章

可扩展语言

不久以前，如果你问 Lisp 是干什么，很多人会回答说“用在人工智能上”。事实上，Lisp 和人工智能之间的联系只是历史的偶然。Lisp 由 John McCarthy 发明，他也是第一个提出 “人工智能” 这一名词的人。那时他的学生和同事用 Lisp 写程序，于是它就被称作一种 AI 语言。这个典故在 1980 年代 AI 短暂升温时又被多次提起，到现在这已经差不多成了习惯。

幸运的是，“AI 并不是 Lisp 的全部” 的观点已经开始为人们所了解。近年来软硬件的长足发展已经让 Lisp 走出了象牙塔: 它目前用于 GNU Emacs，Unix 下最好的文本编辑器; AutoCAD，工业标准的桌面 CAD 程序; 还有 Interleaf，领先的高端出版系统。Lisp 在这些程序里的应用跟 AI 已经没有任何关系。

如果 Lisp 不是一种 AI 语言，那它是什么? 与其根据那些使用它的公司来判断 Lisp，我们不如直接看看语言本身。什么是你可以用 Lisp 做到而其他语言没法做到的呢? Lisp 的一个最显著的优点是可以对其量身定制，让它与用它写的程序相配合。Lisp 本身就是一个 Lisp 程序，Lisp 程序可以表达成列表，那也是 Lisp 的数据结构。总之，这两个原则意味着任何用户都可以为 Lisp 增加新的操作符，而这些新成员和那些内置的操作符是没有区别的。

1.1 渐进式设计

由于 Lisp 给了你自定义操作符的自由，你就可以随意地将它变成你所需要的语言。如果你在写一个文本编辑器，那么可以把 Lisp 转换成专用于写文本编辑器的语言。如果你在编写 CAD 程序，那么可以把 Lisp 转换成专用于写 CAD 程序的语言。并且如果你还不太清楚你要写哪种程序，那么用 Lisp 来写会比较安全。无论你想写哪种程序，在你写的时候，Lisp 都可以演变成用于写那种程序的语言。

如果你还不太确定你要写哪种程序? 对有些人来说，这句话已经是老一套了。这与特定的做事模式有巨大差异，这种模式有两步: (1) 仔细计划你打算做的事情，接下来(2) 去执行它。在这种模式眼中，如果 Lisp 鼓励你在决定程序应该如何工作之前就开始写程序，它只不过是怂恿你匆忙上马，草率决定而已。

好吧，故事不是这样的。先策划再实现的方法可能是建造水坝或者发动袭击

的方式，但经验并未表明这种方法也适用于写程序。为什么？也许是因为计算机的要求太苛刻了。也许是因为程序之中包含比水坝或者袭击更多的变数。或许老方法不再奏效，是因为旧式的冗余观念不适用于软件开发: 如果一个大坝浇铸了额外的 30% 的混凝土，那是为了以后错误操作留下的裕量，但如果一个程序多做了额外 30% 的工作，那就是一个错误。

很难说清楚为什么旧的方法失效了，但所有人都心知肚明老办法不再行之有效。究竟什么时候软件按时交付过？有经验的程序员知道无论你多小心地计划一个程序，当你着手写它的时候，之前制定的计划在某些地方就会变得不够完美。有时计划甚至会错得无可救药。却很少有“先策划再实施”这一方法的受害者站出来质疑它的有效性。相反他们把这都归咎于人为的过失: 只要计划做的更有前瞻性，所有的麻烦就都可以避免。由于即使最杰出的程序员在他们进行具体实现的时候也难免陷入麻烦，因此要人们必须具备那种程度的前瞻性可能过于苛求了。也许这种先策划再实施的方法可以被另外一种更适合我们自身限制的方法取代。

如果有合适的工具，我们完全可以换一种角度看待编程。为什么我们要在具体实现之前计划好一切呢？盲目启动一个项目的最大危险是我们可能不小心就使自己陷入困境。但如果我们有一种更加灵活的语言，是否能减轻一些这种担心呢？我们可以，而且确实是这样。Lisp 的灵活性带来了全新的编程方式。在 Lisp 中，可以边写程序边做计划。

为什么要等事后诸葛亮呢？正如 Montaigne[1] 所发现的那样，如果要理清自己的思路，试着把它们写下来会是最好的办法。一旦你可以把自己从陷入困境的危险中解脱出来，那你就可以完全地驾驭这种可能性。边设计边施工有两个重要的后果: 程序可以花更少的时间去写，因为当你把计划和实际动手写放在一起的时候，你总可以集中精力在一个实际的程序上; 然后让它变得越来越好，因为最终的设计必定是进化的成果。只要在把握你程序的命运时坚持一个原则: 只要你每明确一个错误的部分，就立即重写它，那么最终的产品将会比事先你花几个星期的时间精心设计的结果更加优雅。

Lisp 的适应能力使这种编程思想成为可能。确实，Lisp 的最大危险是它可能会把你宠坏了。你使用 Lisp 一段时间后，你会开始对语言和应用程序之间的结合变得敏感，当你回过头去使用另一种语言时，你总会有这样的感觉: 它无法提供你所需要的灵活性。

1.2 自底向上程序设计

一个长期存在的编程原则是:作为程序的功能性单元不应该过于臃肿。如果程序里某些组件的规模增长超过了它可读的程度，它就会成为一团乱麻，藏匿

[1] 译者注：Montaigne，即 Michel Ryquem de Montaigne。国内一般译作“蒙田”。他是法国文艺复兴后期重要的人文主义学者，他曾说过“我本人就是作品的内容”。

其中的错误就好像巨型城市里的逃犯那样难以捉摸。这样的软件将难以阅读，难以测试，调试起来也会痛苦不堪。

按照这一原则，一个大型程序必须细分成小块，并且越大规模的程序就应该分得越细。但你怎样划分一个程序呢？传统的观点被称为*自顶向下的设计*: 你说 “这个程序的目的是完成这七件事，那么我就把它分成七个主要的子例程。第一个子例程要做这四件事，所以它将进一步细分成它自己的四个子例程”，如此这般。这一过程持续到整个程序被细分到合适的粒度——每一部分都足够大可以做一些实际的事情，但也足够小到可以作为一个基本单元来理解。

有经验的 Lisp 程序员用另一种不同的方式来细化他们的程序。类似自顶向下的设计那样，他们根据一种叫做*自底向上的设计*原则来处理——通过改变语言来适应程序。在 Lisp 中，你不仅是根据语言向下写程序，也可以根据程序向上构造语言。在你编程的时候你可能会想 “Lisp 要是有这样或者那样的操作符就好了。” 那你就可以直接去实现它。后来你意识到使用新的操作符可以简化程序中另一部分的设计，诸如此类。语言和程序一同演进。就像交战两国的边界一样，语言和程序的界限不断地变化，直到最终它们稳定在山川和河流的边缘，这也就是你要解决的问题本身的自然边界。最后你的程序看起来就好像语言就是为解决它而设计的。并且当语言和程序都非常适应彼此时，你得到的将是清晰，短小和高效的代码。

值得强调的是，自底向上的设计并不意味着只是换个次序写程序。当你以自底向上的方式工作时，你通常得到的是一个完全不同的程序。你将得到的是一个带有更多抽象操作符的更大的语言，和一个用它写的更精练的程序，而不是单个的整块的程序。你将得到一个拱而不是梁。

在典型的程序中，一旦把那些仅仅是做非逻辑工作的部分抽象掉，剩下的代码就短小多了; 你构造的语言越高阶，程序从上层逻辑到下层语言的距离就越短。这带来了几点好处:

1. 通过让语言担当更多的工作，自底向上设计产生的程序会更加短小轻快。一个更短小的程序就不必划分成那么多的组件了，并且更少的组件意味着程序会更易于阅读和修改。更少的组件也使得着组件之间的连接会更少，因而错误发生的机会也会相应减少。一个机械设计师往往努力去减少机器上运动部件的数量，同样有经验的 Lisp 程序员使用自底向上的设计方法来减小他们程序的规模和复杂度。

2. 自底向上的设计促进了代码重用。当你写两个或更多程序时，许多你为第一个程序所写的工具也会对后面的程序有帮助。一旦你积累下来了雄厚的工具基础，写一个新程序所耗费的精力和从原始 (raw) Lisp 环境白手起家相比，前者可能只是后者的几分之一。

3. 自底向上的设计提高了程序的可读性。一个这种类型的抽象要求读者去理解一个通用操作符，而一个具体的函数抽象则要求读者去理解一个专

用的子例程。[2]

4. 由于自底向上的设计导致你总是去关注代码中的模式，这种工作方式有助于澄清设计程序时的思路。如果一个程序中两个相距遥远的组件在形式上很相似，你可能就会注意到这种相似性然后可能会以更简单的方式重新设计程序。

对于其他非 Lisp 的语言来说，自底向上的设计在某种程度上也是可能的。大家熟悉的库函数就是自底向上设计的一种体现。尽管如此，Lisp 还能提供比其他语言更强大的威力，而且在使用 Lisp 风格来做这样的设计时，很大一部分工作是在扩展这个语言本身，以致于 Lisp 不仅是另一门编程语言，还是一种完全不同的编程方式。

可以认为这种开发风格更加适合那类基于小组开发的程序。无论如何，与此同时，它还让一个小组所能做更多的事情。在《人月神话》一书中，Frederick Brooks提出了一组程序员的生产力并不随人员数量线性增长的命题。随着组内人数的提高，个体程序员的生产力将有所下降。Lisp 编程经验以一种更加令人振奋的方式表明如下定律: 随着组内人数的减少，个体程序员的生产力将会提高。一个小组的成功，相对而言，仅仅是因为它规模小。当一个小组开始运用 Lisp 所带来的技术优势时，它毫无疑问地会走向成功。

1.3 可扩展软件

随着软件复杂度的提高，编程的 Lisp 风格也变得愈加重要。专业用户现在对软件的要求如此之多以致于我们几乎无法预见到他们的所有需求。就算用户自己也没办法预测到他们所有的需求。但如果我们不能给他们一个现成的软件，让它能完成用户想要的每一个功能，我们也可以给他们一个可扩展的软件。我们把自己的软件从单单一个程序变成了一门编程语言，然后高级用户就可以在此基础上构造他们所需的额外特性。

自底向上的设计很自然地产生了可扩展的程序。最简单的自底向上程序包括两层: 语言和程序。复杂程序可以被写成多个层次，每一层作为其上层的编程语言。如果这一哲学被一直沿用到最上面的那层，那最上面的这一层对于用户来说就变成了一门编程语言。这样一个可扩展性体现在每一层次的程序，与那些先按照传统黑盒方法写成，事后才加上可扩展性的那些系统相比，更有可能成为一门好得多的编程语言。

X–Window 和 TEX 是遵循这一设计原则编写而成的早期实例。在 1980 年代，更强大的硬件使得新一代的程序能使用 Lisp 作为它们的扩展语言。首先是 GNU Emacs，流行的 Unix 文本编辑器。紧接着是 AutoCAD，第一个提

[2] “但是没人能读懂你的程序，除非理解了所有新的实用函数。”要知道为什么这种认识是一种误解，请参考第 4.8 节。

供 Lisp 作为其扩展语言的大型商业软件。1991 年 Interleaf 发布了他们软件的新版本，它不仅采用 Lisp 作为扩展语言，甚至该软件大部分就是用 Lisp 实现的。

Lisp 是一个用来编写可扩展程序的特别合适的语言，主要是因为它本身就是一个可扩展的程序。如果你用 Lisp 写你的程序以便将这种可扩展性转移到用户那里，你事实上已经毫不费力地得到了一个可扩展语言。并且用 Lisp 扩展一个 Lisp 程序，和用一个传统语言做同样的事情相比，它们的区别就好比面对面交谈和使用书信联系的区别。在一个以简单地提供访问外部程序的方式来达到可扩展性的程序里，我们最乐观的估计也无非是两个黑箱之间彼此通过预先定义好的渠道进行通信。在 Lisp 里，这些扩展有权限直接访问整个底层程序。这并不是说你必须授予用户访问你程序中每一个部分的权限——只是说你现在有机会决定是否赋给它们这样的权限。

当权限的取舍和交互式环境结合在一起，你就拥有了处于最佳状态的可扩展性。任何软件，如果你想自己把它作为基础在其上进行扩展，在你心里就好比有了一个非常大，可能过于巨大的完整的蓝图。要是其中的有些东西不敢确定，该怎么办？如果原始程序是用 Lisp开发的，那就可以交互式地试探它：你可以检查它的数据结构；你可以调用它的函数；你甚至可能去看它最初的源代码。这种反馈信息让你能信心百倍地写程序——去写更加雄心勃勃的扩展，并且会写得更快。交互式环境总是可以使编程更加简单，但它对写扩展的人来说无疑更具价值。

可扩展的程序是一把双刃剑，但近来的经验表明，和一把钝剑相比，用户更喜欢双刃剑。可扩展的程序看起来正在流行，不论它们是否暗藏危险。

1.4 扩展 Lisp

有两种方式可以给 Lisp 增加新操作符：函数和宏。在 Lisp 里，你定义的函数和那些内置函数具有相同的地位。如果你想要一个新的改版的 `mapcar`，那你就可以先自己定义，然后就像使用 `mapcar` 那样来使用它。例如，如果有一个函数，你想把从 1 到 10 之间的所有整数分别传给它，然后把函数的返回值组成的列表留下，你可以创建一个新列表然后把它传给 `mapcar`：

```
(mapcar fn
        (do* ((x 1 (1+ x))
              (result (list x) (push x result)))
             ((= x 10) (nreverse result))))
```

但这种手法既不美观又缺乏效率。[3] 换种办法，你也可以定义一个新的映射函数 `map1-n` (见 48 页)，然后像下面那样调用它：

[3] 你也可以使用 Common Lisp 的 series 宏把代码写得更简洁，但那也只能证明同样的观点，因为这些宏就是 Lisp 本身的扩展

```
(map1-n fn 10)
```

定义函数相对来说比较直截了当。宏提供了一种更通用，但不太容易理解的定义新操作符的手段。宏是用来写程序的程序。这句话意味深长，深入地探究这个问题正是本书的主要目的。

考虑周到地使用宏，可以使程序令人惊叹地清晰简洁。这些好处绝非唾手可得。尽管到最后，宏将被视为世界上最自然的东西，但最初理解它们的时候却非常艰难。部分原因是它们比函数更加一般化，所以书写它们的时候要考虑的事情更多。但宏难于理解的最主要原因是它们属于外来事物。没有任何一门其它语言有像 Lisp 宏那样的东西。所以学习宏可能必须要从头脑中清除从其他语言那里偶然学到的先入为主的观念。其中最重要的就是程序这一概念要被完全颠覆。凭什么数据结构是变化的，并且其中的数据可被修改，而程序却不能呢? 在 Lisp 中，程序就是数据，但这一事实的深层含义需要些时日才能体会到。

如果你需要花一些时间才能习惯宏，那么这些时间绝对是值得的。即使像迭代这样平淡无奇的用法中，宏也可以使程序明显变得更短小精悍。假设一个程序需要在某个程序体上从 a 到 b 来迭代 x。Lisp 内置的 `do` 用于更加一般的场合。对于简单的迭代来说它并不能产生可读性最好的代码:

```
(do ((x a (+ 1 x)))
    ((> x b))
  (print x))
```

另一方面，假如我们可以只写成这样:

```
(for (x a b)
  (print x))
```

宏使这成为可能。使用六行代码 (见某页) 我们就可以把 `for` 加入到语言中来，就好像原装的一样。并且正如后面的章节所显示的那样，写一个 `for` 对于我们可以用宏来做什么来说还只是个开始。

你也并不限于每次只给 Lisp 扩展一个函数或者宏。如果你需要，就可以在 Lisp 之上构造一个完整的语言，然后用它来写你的程序。Lisp 对于写编译器和解释器来说是极为优秀的语言，但它提供了定义新语言的另外一种方式，通常更加简洁并且当然也只需要更少的工作: 定义一种语言作为 Lisp 的变形。然后 Lisp 中不用改变的部分可以在新语言里 (例如运算或者 I/O) 继续像原来那样使用，你只需要实现有变化的那部分 (例如控制结构)。以这种方式实现的语言称为嵌入式语言。

嵌入式语言是自底向上程序设计的自然产物。Common Lisp 里已经有了好几种。其中最著名的 CLOS 将在最后一章里讨论。但你也可以定义自己的嵌入式语言。你可以得到一个完全为你程序度身定制的语言，甚至它们最后看起来跟 Lisp 已经非常不同。

1.5 为什么 (或者说何时) 用 Lisp

这些新的可能性并非来自某一个神奇的源头。在这种观点下，Lisp 就像一个拱顶。究竟哪一块楔形石头 (拱石) 托起了整个拱呢? 这个问题本身就是错误的; 每一块都是。如同拱一样，Lisp 是一组相互契合的特性。我们可以列出这些特性中的一部分：动态存储分配和垃圾收集，运行时类型系统，函数对象，生成列表的内置解析器，一个接受列表形式的程序的编译器，交互式环境等等，但 Lisp 的威力不能单单归功于它们中的任何一个。是上述这些特性一同造就了 Lisp 编程现在的模样。

在过去的二十年间，人们的编程方式发生了变化。其中许多变化——交互式环境、动态链接，甚至面向对象的程序设计——就是一次又一次的尝试，它们把 Lisp 的一些灵活性带给其它编程语言。关于拱顶的那个比喻说明了这些尝试是怎样的成功。

众所周知，Lisp 和 Fortran 是目前仍在使用中的两门最古老的编程语言。可能最关键的问题在于它们在语言设计的哲学上体现出来了截然相反的两个极端。Fortran 被发明出来代替汇编语言。Lisp 被发明出来表述算法。如此截然不同的意图产生了迥异的两门语言，Fortran 使编译器作者的生活更轻松; 而 Lisp 则使程序员的生活更轻松。自从那时起，大多数编程语言都落在了两极之间。Fortran 和 Lisp 它们自己也逐渐在向中间地带靠拢。Fortran 现在看起来更像 Algol 了，而 Lisp 也改掉了它年幼时的一些很低效的语言习惯。

最初的 Fortran 和 Lisp 在某种程度上定义了一个战场。战场的一边的口号是 “效率！(并且，还有几乎不可能实现。)” 在战场的另一边，口号是 “抽象！(并且不管怎么说，这不是产品级软件。)” 就好像诸神在冥冥之中决定古希腊战争的结果那样，编程语言这场战争的结局取决于硬件。每一年都在往 Lisp 更有利的方向发展。现在对 Lisp 的争议听起来已经有点儿像 1970 年代早期汇编语言程序员对于高级语言的论点。问题不再是*为什么用 Lisp?*，而是*何时用 Lisp?*

第 2 章

函数

函数是 Lisp 程序的基石. 它们同时也是 Lisp 语言的基石. 在多数语言里 + (加) 操作符都和用户自定义的函数多少有些不一样. 但 Lisp 采用的是统一模型, 即函数应用, 来描述一个程序所能完成的所有计算. 在 Lisp 里, + 是一个函数, 就好像你自己定义的那些函数一样.

事实上, 除了很少数称为*特殊形式 (special form)* 的操作符之外, 整个 Lisp 的核心就是函数的集合. 有什么可以阻止你给这个集合添加新函数呢? 答案是没有: 如果你觉得某件事 Lisp 应该能做, 那你完全可以把它写出来, 然后你的新函数可以得到和内置函数同等的待遇.

对于程序员来说，这一事实产生的后果非同一般. 它意味着任何新函数都可以被认为是对 Lisp 语言的扩充, 也可以被看成是特定应用的一部分. 典型情况是, 一个有经验的 Lisp 程序员两边都写一些, 然后不断调整语言和应用之间的界限直到它们彼此完美地配合在一起. 本书正是关于如何在语言和应用之间达到最佳的结合点. 由于我们向这一最终目标迈进的每一步都依赖于函数, 所以自然应该先从函数开始.

2.1 作为数据的函数

两件事使 Lisp 函数与众不同. 一个是前面提到的, Lisp 本身就是函数的集合. 这意味着我们可以自己给 Lisp 增加新的操作符. 关于函数的另一件重要的事情, 是要了解函数也是 Lisp 的对象.

Lisp 提供了其他语言里能找到的大多数数据类型. 我们有整数和浮点数、字符串、数组、结构体等等. 但 Lisp 还支持一种初看起来令人感到惊奇的数据类型: 函数. 几乎所有编程语言都提供某种形式的函数或过程. 说“Lisp 把函数作为一种数据类型提供出来”又是什么意思呢？这意味着在 Lisp 里我们可以像对待其他熟悉的数据类型那样来对待函数, 就像整数那样: 在运行期创建一个新函数, 把它们存在变量和结构体里面, 把它们作为参数传给其他函数, 以及让它们作为函数的返回值.

这种在运行期创建和返回函数的能力特别有用. 这个优点可能初看起来还让人心存疑虑, 就好像那些可以在某些计算机上运行的可以修改自身的机器语言一样. 但对于 Lisp 来说, 在运行期创建新函数的技术简直就是家常便饭.

2.2 定义函数

多数人会从用 `defun` 来创建函数开始学习. 下面的表达式定义了一个叫 `double` 的函数, 它的返回值是被传入参数的两倍.

```
> (defun double (x) (* x 2))
DOUBLE
```

如果把上述定义送入 Lisp, 我们就可以从其他函数调用 `double`, 或者从最顶层 (toplevel)调用:

```
> (double 1)
2
```

一个 Lisp 代码文件通常主要由类似这样的函数定义组成, 这有点像 C 或者 Pascal 这类语言中的过程定义. 但接下来就有很多不同了. 这些 `defun` 并不只是过程定义, 它们也是 Lisp 调用. 后面我们考察在 `defun` 的背后发生了什么时, 这种区别就会变得更明显.

同时, 函数本身也是对象. `defun` 实际所做的就是构造一个这样的对象, 然后把它存放在第一个参数所给出的名字下. 所以当我们调用 `double` 时, 我们得到的是这个名字所对应的那个函数对象. 得到这个对象的通常做法是使用 `#'` (井号–单引号) 操作符. 这个操作符的作用可以被理解成：它能将名字映射到实际的函数对象. 通过把它放到 `double` 这个名字前面:

```
> #'double
#<Interpreted Function DOUBLE 5811A899>
```

我们得到了通过上面的定义创建的实际对象. 尽管它的打印形式在不同的 Lisp 实现中各不相同[1], Common Lisp 函数是第一类 (first-class) 对象, 它和整数和字符串这些更为我们所熟悉的对象享有完全相同的权利. 所以, 我们可以把这个函数作为参数来传递, 返回这个函数, 把它存在数据结构里, 诸如此类:

```
> (eq #'double (car (list #'double)))
T
```

我们甚至可以不用 `defun` 来定义函数. 如同大多数 Lisp 对象那样, 我们可以通过其文字表达的形式来直接指定它. 就像当我们想要表达一个整数时, 只要使用这个整数本身. 要表达一个字符串, 我们使用括在两个双引号之间的一系列字符. 当我们要想表达一个函数, 我们可以使用一种称为 *λ–表达式 (lambda-expression)* 的东西. 一个 λ–表达式是一个由三部分组成的列表: `lambda` 符号、参数列表，以及包含零个以上表达式的主体. 下面这个 λ–表达式相当于一个和 `double` 等价的函数:

[1] 译者注: 函数对象在交互式环境下的输出格式是 Lisp 厂商自行定义的. 本书中使用的是 CMUCL (CMU Common Lisp) 环境, 但提示符部分仍然保留了 >, 输出的样子看起来和原书有些不同, 但这无关紧要. 而且 CMUCL 的输出格式包含的信息更丰富一些。

```
(lambda (x) (* x 2))
```

它描述了一个函数, 带有一个参数 x, 并且返回 $2x$.

一个 λ–表达式也可以被看作是一个函数的名字. 如果说 `double` 是一个正规的名字, 就像“米开朗琪罗”, 那么 `(lambda (x) (* x 2))` 就相当于一个确切的描述, 就像“完成西斯庭大教堂穹顶壁画的那个人”. 通过把一个井号–引号放在 λ–表达式的前面, 我们就得到了相应的函数:

```
> #'(lambda (x) (* x 2))
#<Interpreted Function (LAMBDA (X) (* X 2)) 5812FFB1>
```

这个函数和 `double` 的表现相同, 但它们是两个不同的对象.

在函数调用中, 函数名出现在前面, 接下来是参数:

```
> (double 3)
6
```

由于 λ–表达式同时也是函数的名字, 因而它们也可以出现在函数调用的最前面:

```
> ((lambda (x) (* x 2)) 3)
6
```

在 Common Lisp 里, 我们可以同时拥有名为 `double` 的函数和变量.

```
> (setq double 2)
2
> (double double)
4
```

当一个名字出现在函数调用的第一位置, 或者前置一个 `#'` 的时候, 它被用来指一个函数. 其他场合下它则被当作一个变量名.

因此我们说 Common Lisp 拥有分离的函数和变量*命名空间* *(name-space)*. 我们可以同时有一个叫 `foo` 变量以及一个叫 `foo` 的函数, 且它们不必相同. 这种情形可能令人困惑, 并且可能导致代码一定程度上的丑陋, 但这是 Common Lisp 程序员必须面对的东西.[2]

如果需要的话, Common Lisp 还提供了两个函数用于将符号映射到它所代表的函数或者变量. `symbol-value` 函数以一个符号为参数, 返回其对应变量的值:

```
> (symbol-value 'double)
2
```

而 `symbol-function` 则用来得到一个全局定义的函数:

```
> (symbol-function 'double)
#<Interpreted Function DOUBLE 5811A899>
```

[2] 译者注: 拥有分开的变量和函数命名空间的 Lisp 称为 Lisp–2, 在另一类 Lisp–1 下, 变量和函数定义在同一个命名空间里, 最著名的这种 Lisp 方言是 Scheme. 关于 Lisp–1 vs. Lisp–2 的讨论在网上有很多, 一般观点认为 Lisp–1 对于编译器来说更难实现.

注意到, 由于函数也是普通的对象, 所以变量也可以把函数作为它的值:

```
> (setq x #'append)
#<Function APPEND 10064D71>
> (eq (symbol-value 'x) (symbol-function 'append))
T
```

深入分析的话, defun 实际上是把它第一个参数的 symbol-function 设置成了用它其余部分构造的函数. 下面两个表达式基本上做了同一件事:

```
(defun double (x) (* x 2))

(setf (symbol-function 'double)
      #'(lambda (x) (* x 2)))
```

所以 defun 和其他语言的过程定义具有相同的效果——把一个名字和一段代码关联起来. 但是底层手法完全不同. 我们不需要用 defun 来创建函数, 函数也不是一定要保存在一些符号的值里. 如同其他语言里的过程定义的 defun, 它的潜在特征其实是个更一般的手法: 构造一个函数和把它跟一个确定的名字关联起来, 其实是两个分开的操作. 当我们不需要用到 Lisp 中所谓函数的所有含义的时候，用 defun来生成函数就和在其他限制更多的语言里一样的简单.

2.3 函数型参数

函数, 作为数据对象, 就意味着我们可以像对待其他对象那样, 把它传递给其他函数. 这种性质对于 Lisp 这种自底向上程序设计至关重要.

如果一门语言把函数作为数据对象，那么它就必然也会提供某种方式让我们能调用它们. 在 Lisp 里, 这个函数就是 apply. 一般而言, 我们用两个参数来调用 apply: 一个函数和它的参数列表. 下列四个表达式具有相同的效果:

```
(+ 1 2)

(apply #'+ '(1 2))

(apply (symbol-function '+) '(1 2))

(apply #'(lambda (x y) (+ x y)) '(1 2))
```

在 Common Lisp 里, apply 可以带有任意数量的参数, 最前面给出的函数, 将被应用到一个列表，该列表由其余参数 cons 到最后一个参数产生，最后一个参数也是一个列表. 所以表达式

```
(apply #'+ 1 '(2))
```

等价于前面四个表达式. 如果不方便以列表的形式提供参数, 我们可以使用 funcall, 它和 apply 唯一的区别也就在这里. 表达式

```
(funcall #'+ 1 2)
```

和上面的那些效果相同.

很多内置的 Common Lisp 函数可以带函数型参数. 其中用得最广泛的是映射类的函数. 例如 `mapcar` 带有两个以上参数——一个函数加上一个以上的列表 (每个列表都分别是函数的参数), 然后它可以将参数里的函数依次作用在每个列表的元素上:

```
> (mapcar #'(lambda (x) (+ x 10))
          '(1 2 3))
(11 12 13)
> (mapcar #'+
          '(1 2 3)
          '(10 100 1000))
(11 102 1003)
```

Lisp 程序经常需要对一个列表中的每一项都做一些操作然后再把结果同样以列表的形式返回. 上述的第一个例子介绍了完成这一功能的常用办法: 生成一个你所需功能的函数, 然后用 mapcar 把它映射到列表上.

我们已经了解到，能把函数当作数据来使用给编程带来了极大的便利. 在许多语言里, 即便我们可以像 `mapcar` 那样把函数作为参数传递进去, 那也只能是在先前的代码文件中定义好了的函数. 如果只有一小段代码需要把列表中的每一项都加上 10, 我们就不得不定义一个函数, 叫做 `plus_ten` 或者类似的名字, 而这个函数仅仅在这一小段代码中被使用. 有了 λ–表达式, 我们就可以直接表达函数.

Common Lisp 和它之前的方言之间的一个最大的区别就是它包含有大量使用函数型参数的内置函数. 其中, 除了随处可见的 `mapcar`, 还有两个最常用的函数就是 `sort` 和 `remove-if`. 前者是通用的排序函数. 它接受一个列表和一个谓词, 然后将原列表中的元素两两使用该谓词进行比较, 并将得到的排好了序的列表返回.

```
> (sort '(1 4 2 5 6 7 3) #'<)
(1 2 3 4 5 6 7)
```

记住 `sort` 函数工作方式的一种方法是, 如果你用 < 排序一个没有重复元素的列表, 那么当你把 < 应用到结果列表的时候, 它将会返回真.[3]

如果 `remove-if` 函数不包含在 Common Lisp 中的话, 那它就应该是你写的第一个工具. 它接受一个函数和列表, 并且返回这个列表中所有调用那个函数返回假的元素.

```
> (remove-if #'evenp '(1 2 3 4 5 6 7))
(1 3 5 7)
```

[3]译者注：即`(apply #'< '(1 2 3 4 5 6 7)) => T`。

作为一个接受函数作为参数的函数示例, 这里给出一个 remove-if 的受限版本:

```
(defun our-remove-if (fn lst)
  (if (null lst)
      nil
      (if (funcall fn (car lst))
          (our-remove-if fn (cdr lst))
          (cons (car lst) (our-remove-if fn (cdr lst))))))
```

注意到在这个定义里 fn 并没有前缀 #'. 因为函数就是数据对象, 变量可以将一个函数作为它的正规值. 这就是事情的原委. #' 仅用来引用那些以符号命名的函数——通常是用 defun 全局定义的.

正如第 4 章将要展示的那样, 编写那种接受函数作为参数的新工具是自下而上程序设计的重要环节. Common Lisp 自带了非常多的工具函数, 很多你想要的可能已经有了. 但无论是使用内置的工具, 比如 sort, 还是编写你的实用工具, 基本原则是一样的：与其把功能写死，不如传进去一个函数参数。

2.4 作为属性的函数

函数作为 Lisp 对象这一事实也为我们创造条件，让我们能够编写出那种可以随时扩展以满足新需求的程序. 假设我们需要写一个以动物类型作为参数并产生相应行为的函数. 在大多数语言中, 会使用 case 语句达到这个目的, 同样我们也可以在 Lisp 里用一样的办法:

```
(defun behave (animal)
  (case animal
    (dog (wag-tail)
         (bark))
    (rat (scurry)
         (squeak))
    (cat (rub-legs)
         (scratch-carpet))))
```

如果我们要增加一种新动物该怎么办呢? 如果我们计划增加新的动物, 那么把 behave 定义成下面的样子可能更好一些:

```
(defun behave (animal)
  (funcall (get animal 'behavior)))
```

然后把每种个体动物的行为以单独的函数形式保存, 例如存放在以它们名字命名的属性列表里:

```
(setf (get 'dog 'behavior)
      #'(lambda ()
          (wag-tail)
          (bark)))
```

以这种方式, 要增加一种新动物, 你需要做的全部事情就是定义一个新的属性. 不需要重写任何函数.

上述第二种方法尽管更灵活, 但是看起来更慢一些. 实际上也是如此. 如果速度很关键, 我们可以使用结构体来代替属性表, 而且尤其要用编译过的函数代替解释性的函数. (第 2.9 节解释了怎样做到这些.) 使用了结构体和编译函数, 上面的代码就更富灵活性，其速度可以达到甚至超过那些使用 `case` 语句的实现.

函数的这种用法相当于面向对象编程中的*方法*概念. 通常来讲, 方法是作为对象一种属性的函数, 也就是我们已有的那些. 如果你把继承引入这个模型, 你就得到了面向对象编程的全部要素. 第 25 章将用少得惊人的代码来说明这一点.

面向对象编程的一大亮点是它可以使程序可扩展. 这种前景在 Lisp 界并未激起涟漪, 因为在这里可扩展性从来就是被默认的（一个重要特性）. 如果我们要的可扩展性对于继承的需求不是很多, 那可能纯 Lisp 就足够对付了.

2.5 作用域

Common Lisp 是一种词法作用域 (lexically scope) 的 Lisp. Scheme 是最早的有词法作用域的方言; 在 Scheme 以前, 动态作用域 (dynamic scope) 被看作 Lisp 的决定性属性之一.

词法作用域和动态作用域的区别在于语言处理自由变量的方式不一样. 当一个符号被用来表达变量时，我们称这个符号在表达式里是*被绑定的* (bound) 的, 这里的变量可以是参数, 也可以来自像 `let` 和 `do` 这样的变量绑定操作符. 如果符号不受到约束, 就认为它是*自由*的. 下面的例子具体说明了作用域:

```
(let ((y 7))
  (defun scope-test (x)
    (list x y)))
```

在函数表达式里, x 是受约束的, 而 y 是自由的. 自由变量的有趣之处在于它们应有的值并不是很明显. 一个约束变量的值是毫无疑问的——当 `scope-test` 被调用时, x 的值就是通过参数传给它的值. 但 y 的值应该是什么呢? 这个问题要看具体方言的作用域规则.

在动态作用域的 Lisp 里, 要想找出当 `scope-test` 执行时自由变量的值, 我们要往回逐个检查函数的调用链. 当我们发现 y 被绑定时，这个被绑定的值即被用在 `scope-test` 中. 如果我们没有发现, 那我们就取 y 的全局值. 这样, 在一个动态作用域的 Lisp 里, 在调用的时候 y 将会产生这样的值:

```
> (let ((y 5))
    (scope-test 3))
(3 5)
```

在动态作用域里, 当 scope-test 被定义时 y 被绑定到 7 就没有任何意义了. 当 scope-test 被调用时 y 只有一个值, 就是 5.

在词法作用域的 Lisp 里, 和往回逐个检查函数的调用链不一样的是, 我们检查当这个函数被定义时所处的环境. 在一个词法作用域 Lisp 里, 我们的示例将捕捉到当 scope-test 被定义时变量 y 的绑定. 所以下面的事情将会发生在 Common Lisp 里:

```
> (let ((y 5))
    (scope-test 3))
(3 7)
```

这里将 y 绑定到 5 在调用时对返回值没有任何效果.

尽管你仍然可以通过将变量声明为 *special* 来得到动态作用域, 词法作用域是 Common Lisp 的默认行为. 整体来看, Lisp 社区对动态作用域的过时几乎没什么留恋. 因为它经常会导致痛苦而又难以捉摸的 bug. 而词法作用域不仅只是一种避免错误的方式. 在下一章我们会看到, 它同时也带来了一些崭新的编程技术.

2.6 闭包

由于 Common Lisp 是词法作用域的, 当我们定义一个包含自由变量的函数时, 系统必须在函数定义的时候保存那些变量的绑定. 这种函数和一组变量绑定的组合称为闭包. 闭包被广泛用于各种应用场合.

闭包在 Common Lisp 程序中如此无所不在以至于你可能已经用了却不知情. 每当你给 mapcar 一个包含自由变量的前缀 #’ 的 λ–表达式时, 你就在使用闭包. 例如, 假设我们想写一个函数, 它接受一个数列并且给每个数增加确定的数量. 这个 list+ 函数

```
(defun list+ (lst n)
  (mapcar #'(lambda (x) (+ x n))
          lst))
```

将做到我们想要的:

```
> (list+ '(1 2 3) 10)
(11 12 13)
```

如果我们仔细观察 list+ 里传给 mapcar 的那个函数, 它实际上是个闭包. 那个 n 是自由的, 它的绑定来自周围的环境. 在词法作用域下, 映射函数的每一次这样的使用都将导致创建一个闭包. [4]

[4]在动态作用域（作为默认作用域）的情况下，这种表达方式也会一样工作，虽然其原理不同——只要 mapcar 的参数里没有一个以“n”作为名字。

闭包在 Abelson 和 Sussman 的经典教材 *Structure and Interpretation of Computer Programs* 一书中扮演了更加重要的角色. 闭包是带有局部状态的函数. 使用这种状态最简单的方式是如下的情况:

```
(let ((counter 0))
  (defun new-id ()   (incf counter))
  (defun reset-id () (setq counter 0)))
```

这两个函数共享一个计数器变量. 前者返回计数器的下一个值, 后者把计数器重置到 0. 这种方式避免了对计数器变量非预期的引用, 尽管同样的功能也可以用一个全局的计数器变量完成.

能返回一个带有本地状态的函数也是很有用的. 例如这个 `make-adder` 函数

```
(defun make-adder (n)
  #'(lambda (x) (+ x n)))
```

接受一个数值参数, 然后返回一个闭包, 当后者被调用时, 能够把之前那个数加到它的参数上. 我们可以根据需要生成任意数量的这种加法器:

```
> (setq add2 (make-adder 2)
        add10 (make-adder 10))
#<Interpreted Function "LAMBDA (N)" 58121711>
> (funcall add2 5)
7
> (funcall add10 3)
13
```

在 `make-adder` 返回的那些闭包里, 内部状态都是固定的, 但其实也有可能生成那种可以要求改变他们状态的闭包.

```
(defun make-adderb (n)
  #'(lambda (x &optional change)
      (if change
          (setq n x)
          (+ x n))))
```

这个新版本的 `make-adder` 函数返回一个闭包, 当以一个参数被调用时, 其行为就跟旧版本的一样.

```
> (setq addx (make-adderb 1))
#<Interpreted Function "LAMBDA (N)" 5812A2F9>
> (funcall addx 3)
4
```

尽管如此, 当这个新型的加法器用非空的第二个参数调用时, 它内部的 n 的拷贝将被重置成由第一个参数指定的值:

```
> (funcall addx 100 t)
100
> (funcall addx 3)
103
```

```
(defun make-dbms (db)
  (list
    #'(lambda (key)
        (cdr (assoc key db)))
    #'(lambda (key val)
        (push (cons key val) db)
        key)
    #'(lambda (key)
        (setf db (delete key db :key #'car))
        key)))
```

图 2.1: 一个列表里的三个闭包

甚至有可能返回共享同一数据对象的一组闭包. 图 2.1 包含有一个创建原始数据库的函数. 它接受一个关联表 (db), 并且相应地返回一个由查询、追加和删除这三个闭包所组成的列表.

对 make-dbms 的每次调用创建一个新数据库——封闭在共享的关联表之上的一组新函数.

```
> (setq cities (make-dbms '((boston . us) (paris . france))))
(#<Interpreted Function "LAMBDA (DB)" 581345C9>
 #<Interpreted Function "LAMBDA (DB)" 58134619>
 #<Interpreted Function "LAMBDA (DB)" 58134669>)
```

数据库里实际的关联表对外界是不可见的, 我们甚至不知道它是个关联表——但是它可以通过组成 cities 的那些函数访问到:

```
> (funcall (car cities) 'boston)
US
> (funcall (second cities) 'london 'england)
LONDON
> (funcall (car cities) 'london)
ENGLAND
```

调用一个列表的 car 多少有些难看. 实际的程序中, 函数访问的入口可能隐藏在结构体里. 当然也可以设法更清晰地使用它们——数据库可以间接地通过类似这样的函数访问:

```
(defun lookup (key db)
  (funcall (car db) key))
```

尽管如此, 闭包的行为不会受到如此包装的影响.

实际程序中的闭包和数据结构往往比我们在 make-adder 和 make-dbms 里看到的更为精巧. 这里用到的单个共享变量也可以发展成任意数量的变量，每个都可以约束到任意的数据结构上。

闭包是 Lisp 的众多独特的和看得见模得着的优势之一. 某些 Lisp 程序, 如果努努力的话, 还有可能翻译到不那么强大的其它语言上, 但只要试着去翻译上

面那些使用了闭包的程序, 就会明白这种抽象帮我们省去了多少工作. 后续章节将继续探讨闭包的更多细节. 第 5 章展示了如何用它们构造复合函数, 然后在第 6 章里会继续介绍它们如何被用来替代传统的数据结构.

2.7 局部函数

当我们用 λ–表达式来定义函数时, 我们就会面对一个使用 `defun` 时所没有的限制: 一个用 λ–表达式定义的函数由于没有名字, 因此也就没有办法引用其自身. 这意味着在 Common Lisp 里我们不能使用 `lambda` 来定义递归函数.

如果我们想要应用某些函数到一个列表的所有元素上, 我们可以使用最熟悉的 Lisp 语句:

```
> (mapcar #'(lambda (x) (+ 2 x))
          '(2 5 7 3))
(4 7 9 5)
```

如果我们想要把一个递归函数作为第一个参数送给 `mapcar` 呢? 如果函数已经用 `defun` 定义了, 我们就可以通过名字简单地引用它:

```
> (mapcar #'copy-tree '((a b) (c d e)))
((A B) (C D E))
```

但现在假设这个函数必须是一个闭包, 它从 `mapcar` 所处的环境获得绑定. 在我们的 `list+` 例子里,

```
(defun list+ (lst n)
  (mapcar #'(lambda (x) (+ x n))
          lst))
```

`mapcar` 的第一个参数, `#'(lambda (x) (+ x n))`, 必须定义在 `list+` 里因为它需要捕捉 `n` 的绑定. 到目前为止都还一切正常, 但如果我们要给 `mapcar` 传递一个函数, 而这个函数在需要本地绑定同时也是递归的呢? 我们不能使用一个在其他地方通过 `defun` 定义的函数, 因为我们需要本地环境的绑定. 并且我们也不能使用 `lambda` 来定义一个递归函数, 因为这个函数将无法引用其自身.

Common Lisp 提供了 `labels` 以跳出这一两难的困境. 除了在一个重要方面有所保留外, `labels` 基本可以看作是 `let` 的函数版本. `labels` 表达式里的每一个绑定规范必须符合如下形式:

(⟨name⟩ ⟨parameters⟩ . ⟨body⟩)

在 `labels` 表达式里, ⟨name⟩ 将指向与下面表达式等价的函数:

#'(lambda ⟨parameters⟩ . ⟨body⟩)

例如,

```
> (labels ((inc (x) (1+ x)))
    (inc 3))
> 4
```

尽管如此, 在 `let` 与 `labels` 之间有一个重要的区别. 在 `let` 表达式里, 一个变量的值不能依赖于同一个 `let` 里生成的另一个变量——就是说, 你不能说

```
(let ((x 10) (y x))
  y)
```

然后期待这个新的 y 能反映出那个新 x 的值来. 相反, 在 `labels` 里定义的函数 f 的函数体里就可以引用那里定义的其他函数, 包括 f 本身, 这就使定义递归函数成为可能.

使用 `labels` 我们就可以写出类似 `list+` 这样的函数了，但这里 `mapcar` 的第一个参数是递归函数：

```
(defun count-instances (obj lsts)
  (labels ((instances-in (lst)
             (if (consp lst)
                 (+ (if (eq (car lst) obj) 1 0)
                    (instances-in (cdr lst)))
                 0)))
    (mapcar #'instances-in lsts)))
```

该函数接受一个对象和一个列表, 然后返回该对象在列表的每个元素 (作为列表) 中出现的次数, 所组成的列表:

```
> (count-instances 'a '((a b c) (d a r p a) (d a r) (a a)))
(1 2 1 2)
```

2.8 尾递归

递归函数调用它们自身. 如果调用以后没其他事可做, 这种调用就称为*尾递归 (tail-recursive)*. 下面这个函数不是尾递归的

```
(defun our-length (lst)
  (if (null lst)
      0
      (1+ (out-length (cdr lst)))))
```

因为在从递归调用返回之后我们又把结果传给了 `1+`. 而下面这个函数就是尾递归的[5]

```
(defun our-find-if (fn lst)
  (if (funcall fn (car lst))
      (car lst)
      (our-find-if fn (cdr lst))))
```

[5]原书勘误: 如果没有元素被找到, `our-find-if` 函数将无限递归下去.

因为通过递归调用得到的值被立即返回了.

尾递归是一种令人青睐的特性, 因为许多 Common Lisp 编译器都可以把尾递归转化成循环. 若使用这种编译器, 你就可以在源代码里书写优雅的递归而不必担心函数调用在运行期产生的系统开销.

一个不是尾递归的函数经常可以通过嵌入一个使用累积器的本地函数, 被转换成尾递归的形式. 在这里, 聚集器指的是一个参数，它代表着到目前为止计算得到的值. 例如 `our-length` 可以转换成

```
(defun our-length (lst)
  (labels ((rec (lst acc)
             (if (null lst)
                 acc
                 (rec (cdr lst) (1+ acc)))))
    (rec lst 0)))
```

上面定义的函数里, 截止到现在，所有见到的列表元素的总数被包含在另一个参数 `acc` 里面了. 当递归运行到达列表的结尾, `acc` 的值就是总的长度, 只要直接返回它就可以了. 通过在调用树从上往下走的过程中累计这个值, 而不是从下往上地在返回的时候再计算它, 我们就可以将 `rec` 尾递归化.

许多 Common Lisp 编译器可以做尾递归优化, 但这并非对所有编译器都是默认的行为. 所以在你编写尾递归函数时, 你也许需要将

```
(proclaim '(optimize speed))
```

写在文件的最前面, 确保编译器可以利用你的苦心进行优化.[6]

如果提供尾递归和类型声明, 现有的 Common Lisp 编译器就能生成速度与 C 程序相媲美，甚至赶超它的代码. Richard Gabriel 给出下列函数作为示例, 它可以从 `1` 累加到 `n`:

```
(defun triangle (n)
  (labels ((tri (c n)
             (declare (type fixnum n c))
             (if (zerop n)
                 c
                 (tri (the fixnum (+ n c))
                      (the fixnum (- n 1))))))
    (tri 0 n)))
```

这就是快速的 Common Lisp 代码的榜样. 开始的时候，用这种方式写程序可能会觉得不太自然. 通常的办法是首先以尽可能自然的方式来写一个函数, 然后如果必要的话, 把它转化成一个尾递归的等价形式.

[6] `(optimize speed)` 的声明应该是 `(optimize (speed 3))` 的简写. 但是有一种 Common Lisp 实现, 若使用前一种声明，则会进行尾递归优化，而后一种声明则不会产生这种优化.

2.9 编译

Lisp 函数可以单独编译或者按文件编译. 如果你只是在 toplevel 下输入一个 defun 表达式,

```
> (defun foo (x) (1+ x))
FOO
```

多数实现会创建一个解释函数. 你可以使用 compiled-function-p 来检查一个函数是否已被编译:

```
> (compiled-function-p #'foo)
NIL
```

我们可以把函数名传给 compile, 用这种办法它来编译 foo

```
> (compile 'foo)
FOO
```

这就可以编译 foo 的定义并且把之前的解释版本替换成编译版本.

```
> (compiled-function-p #'foo)
T
```

编译和解释函数都是 Lisp 对象, 它们的行为表现是相同的, 只是对 compiled-function-p 的反应不一样. 直接给出的函数也可以编译: compile 希望它的第一个参数是一个名字, 但如果你给它一个 nil, 它就会编译第二个参数给出的 λ–表达式.

```
> (compile nil '(lambda (x) (+ x 2)))
#<Function "LAMBDA (X)" 58128F31>
```

如果你同时给出名字和函数参数, compile 的效果就相当于编译一个 defun 了:

```
> (progn (compile 'bar '(lambda (x) (* x 3)))
         (compiled-function-p #'bar))
T
```

把 compile 集成进语言里意味着一个程序可以随时构造和编译新函数. 不过，显式调用 compile 和调用 eval 一样，都属于一种非常规的手段, 也同样值得怀疑. [7] 当第 2.1 节里说在运行时创建新函数属常用编程技术时, 它指的是从类似 make-adder 那样的函数中生成的闭包, 并非指从原始列表里调用 compile 得到的新函数. 调用 compile 并非常用编程技术——这是极其罕见的. 所以要注意不要在不必要情况下使用它。除非你在 Lisp 之上实现另一种语言 , 但即使是这种情况, 你用宏也有可能完成你想做的事情。

有两类函数，你不能把它们作为参数送给 compile. 根据 CLTL2 (第 667 页), 你不能编译一个 “解释性地定义在一个非空词法环境中的” 函数. 那就是说, 在 toplevel 下你定义一个带有 let 的 foo

[7] 某页有关于为何说显式调用 eval 有害的一个解释.

```
> (let ((y 2))
    (defun foo (x) (+ x y)))
```

然后 (compile 'foo) 并不一定能正常工作.[8] 你也不能在一个已经编译了的函数上调用 compile, CLTL2 隐晦地暗示 "结果...是不确定的".

通常编译 Lisp 代码的方法，不是用 compile 对函数逐个地编译, 而是用 compile-file 编译整个文件. 该函数接受一个文件名，然后创建源代码的一个编译版本——一般情况下，编译出的文件和源文件有相同的基本文件名，但扩展名不一样. 当编译过的文件被加载后, compiled-function-p 对文件里定义的所有函数，返回值都是真.

以后的章节还依赖于编译带来的另一种效果: 如果当一个函数出现在另一个函数中, 并且外面的函数已经编译的话, 那么里面的函数也将会被编译. CLTL2 里并没有明确这一行为, 但所有正规的实现都支持它.

对于那些返回函数的函数来说, 内层函数的编译行为是很清楚的. 当 make-adder (第 17 页) 被编译时, 它将返回一个编译过的函数:

```
> (compile 'make-adder)
MAKE-ADDER
> (compiled-function-p (make-adder 2))
T
```

后面的章节里将说明, 这一事实对于实现嵌入式语言来说尤为重要. 如果一种新语言是通过转换来实现的, 并且转换出来的代码是编译了的, 那它也会产生编译了的输出——这个转换过程也就成为了新语言事实上的编译器. (在第 73 页有一个简单的例子。)

如果我们有一个特别小的函数, 我们可能要求它被编译成内联的. 否则调用这个函数的开销可能比这个函数本身的开销都大. 如果我们定义了一个函数:

```
(defun 50th (lst) (nth 49 lst))
```

并且做了声明:

```
(proclaim '(inline 50th))
```

然后一个编译的函数里针对 50th 的引用就不需要一个实际的函数调用了. 如果我们定义并且编译一个调用了 50th 的函数,

```
(defun foo (lst)
  (+ (50th lst) 1))
```

那么当 foo 被编译时, 50th 的代码应该被编译进它里面. 就好像我们一开始写的就是

```
(defun foo (lst)
  (+ (nth 49 lst) 1))
```

[8] 把这段代码写在文件里然后再编译是没问题的. 这一限制是由于具体实现的原因, 被强加在了解释型函数上, 而绝不是因为在清楚明白的词法环境中定义函数有什么不对.

那样. 缺点是，如果我们改动了 `50th` 的定义的话, 那么就必须重新编译 `foo`, 否则它用的还是原来的定义. 内联函数的限制基本上和宏差不多 (见第 7.9 节).

2.10 来自列表的函数

在一些早期的 Lisp 方言中, 函数被表示成一个列表. 这给了程序员们不同寻常的编写和执行他们自己的 Lisp 程序的能力. 在 Common Lisp 中, 函数不再由列表组成——好的实现把它们编译成本地机器代码. 但你仍然可以写出那种编写程序的程序, 因为列表是编译器的输入.

再怎么强调 “Lisp 程序可以编写 Lisp 程序” 都不为过, 尤其是当这一事实经常被忽视时. 即使有经验的 Lisp 程序员也很少意识到他们因这种语言特性而得到的种种好处. 比如, 正是这个特性使得 Lisp 宏如此强大. 本书里描述的大部分技术都依赖于这个能力，即编写处理 Lisp 表达式的程序的能力.

第 3 章

函数式编程

前一章解释了 Lisp 和 Lisp 程序两者是如何由单一的原材料——函数, 建造起来的. 与任何建筑材料一样, 它的品质既影响了我们所建造的事物的种类, 也影响着我们建造它们的方式.

本章描述了在 Lisp 的天地里较常用的那种编程方法. 这些方法的高度精妙让我们能尝试编写更加难缠的程序. 下一章将介绍一种尤其重要的编程方法, 是 Lisp 让我们得以运用这种方法: 即程序的开发形式，可以是逐渐进化的，而不用遵循先计划再实现的老办法.

3.1 函数式设计

一件物品的特征会受到其原材料的影响. 例如，一个木结构建筑和石结构建筑看起来就会感觉不一样. 甚至当你离得很远, 分不清原材料究竟是木头还是石头, 你也可以大体说出它是用什么造的. 与之相似, Lisp 函数的特征也影响着 Lisp 程序的结构.

函数式编程 意味着利用返回值而不是副作用来写程序. 副作用 包括破坏性修改对象 (例如通过 `rplaca`) 以及变量赋值 (例如通过 `setq`). 如果副作用很少并且本地化, 程序就会容易阅读, 测试和调试. Lisp 并不总是写成这种风格的, 但随着时间的推移, Lisp 和函数式编程之间的关系变得越来越密不可分.

有个例子，它可以说明函数式编程和你在其他语言的行事方法究竟有何不同. 假设由于某种原因我们想把列表里的元素逆序一下. 和写一个函数逆序一个列表相反, 我们写一个函数, 它接受一个列表, 然后返回一个带有相同元素但以相反顺序排列的列表.

图 3.1 包含一个对列表求逆的函数. 它把列表看作数组, 按位置取反; 它的返回值是不合理的:

```
> (setq lst '(a b c))
(A B C)
> (bad-reverse lst)
NIL
> lst
(C B A)
```

函数如其名, `bad-reverse` 与好的 Lisp 风格相去甚远. 更糟糕的是, 它还有其它

```
(defun bad-reverse (lst)
  (let* ((len (length lst))
         (ilimit (truncate (/ len 2))))
    (do ((i 0 (1+ i))
         (j (1- len) (1- j)))
        ((>= i ilimit))
      (rotatef (nth i lst) (nth j lst)))))
```

图 3.1: 一个对列表求逆的函数

```
(defun good-reverse (lst)
  (labels ((rev (lst acc)
             (if (null lst)
                 acc
                 (rev (cdr lst) (cons (car lst) acc)))))
    (rev lst nil)))
```

图 3.2: 一个返回逆序表的函数

丑陋之处: 因为它的正常工作依赖于副作用, 它把调用者完全带离了函数式编程的思路.

尽管是典型的反面教材, `bad-reverse` 还是有地方可圈可点: 它展示了交换两个值的 Common Lisp 习惯用法. `rotatef` 宏可以轮转任何普通变量的值——所谓普通变量是指那些可以作为 `setf` 第一个参数的变量. 当它只应用于两个参数时, 效果就是把它们交换.

与之相对, 图 3.2 显示了一个能返回逆序表的函数. 通过使用 `good-reverse`, 我们得到的返回值是逆序后的列表，而原始列表原封不动:

```
> (setq lst '(a b c)
(A B C)
> (good-reverse lst)
(C B A)
> lst
(A B C)
```

过去常认为可以通过外表来判断一个人的性格. 不管这个说法对于人来说是否灵验, 但是对于 Lisp 来说，这常常是可行的. 函数式程序有着和命令式程序不同的外形. 函数式程序的结构完全是由表达式里参数的组合表现出来的, 并且由于参数是缩进的, 函数式代码看起来在缩进方面显得更为灵动. 函数式代码看起来就像页面上的流体[1]; 命令式代码则看起来坚实、愚钝, 就像 Basic 语言那样.

即使远远的看上去, 从 `bad-` 和 `good-reverse` 两个函数的形状也能分清孰优孰劣. 另外, `good-reverse` 不仅短些, 也更加高效: $O(n)$ 而不是 $O(n^2)$.

[1] 某页有一个很典型的例子

我们不用自己动手写一个 `reverse` 函数, 因为 Common Lisp 已经有内置的了. 有必要简单了解一下该函数, 因为它经常能让一些函数式编程上的错误观念显现出来. 像 `good-reverse` 那样, 内置的 `reverse` 通过返回值工作, 并不修改它的参数. 但学习 Lisp 的人们可能会错认为它像 `bad-reverse` 那样依赖于副作用. 如果这些学习者想在程序里的某个地方逆序一个列表, 他们可能会写

```
(reverse lst)
```

结果还很奇怪为什么函数调用没有效果. 事实上, 如果我们希望利用那样一个函数提供的*效果*, 就必须在调用代码里自己处理. 就是说, 需要把程序改成这样

```
(setq lst (reverse lst))
```

调用 `reverse` 这样的操作符，其本意就是取其返回值, 而不是为了它们的副作用. 你自己的程序也值得以这种风格写——不仅因为它固有的好处, 而是因为, 如果你不这样写, 就等于在跟语言过不去.

在比较 `bad-` 和 `good-reverse` 时我们还忽略了一点, 那就是 `bad-reverse` 里没有点对 (cons). 它在原始列表上面操作而不构造新列表. 这样是比较危险的，因为可能在程序的其他地方还需要原始列表，但为了效率有时可能必须这样做. 为满足这种需要, Common Lisp 还提供了一个 $O(n)$ 的称为 `nreverse` 的求逆函数的破坏性版本.

所谓破坏性函数, 是指那类能改变传给它的参数的函数. 即便如此, 破坏性函数也通常通过取返回值的方式工作: 你必须假定 `nreverse` 将会回收利用你作为参数传给它的列表, 但不能假设它帮你求逆了那个列表. 和以前一样, 逆序的列表只能通过返回值拿到. 你仍然不能把

```
(nreverse lst)
```

写在一个函数的中间，然后假定从那以后 `lst` 就是逆序的了. 下面的情况发生在大多数实现里:

```
> (setq lst '(a b c))
(A B C)
> (nreverse lst)
(C B A)
> lst
(A)
```

要想真正求逆一个列表, 你就不得不把 `lst` 赋值成返回值, 这和使用原来的 `reverse` 是一样的.

如果有函数被声明具有破坏性, 这并不是说：调用它就是为了副作用. 危险之处在于，某些破坏性的函数给人留下了破坏性的印象. 例如,

```
(nconc x y)
```

几乎总是和

```
(setq x (nconc x y))
```

具有相同的效果. 如果你写的代码依赖于前一个用法, 有的时候它可以正常工作. 然而当 x 是 nil 的时候，它的行为就会出乎你的意料.

只有少数 Lisp 操作符，其本意就是为了副作用. 一般而言, 内置操作符本来是为了调用后取返回值的. 不要被 sort、remove 或者 substitute 这样的名字所误导. 如果你需要副作用, 那就对返回值使用 setq.

这个规则主张某些副作用其实不可避免. 具有函数式的编程思想并不是说要杜绝副作用. 它只是说除非必要最好不要有.

养成这个习惯可能是需要花些时间的. 一个好的开始是尽可能少地使用下列操作符:

```
set setq setf psetf psetq incf decf push pop pushnew rplaca
rplacd rotatef shiftf remf remprop remhash
```

还包括 let*, 命令式程序经常藏匿于其中. 在这里要求限量使用这些操作符的目的只是希望倡导良好的 Lisp 风格, 而不是想制定清规戒律. 然而, 仅此一项就可让你受益匪浅了.

在其他语言里, 导致副作用的最普遍原因就是让一个函数返回多个值的需求. 如果函数只能返回一个值, 那它们就不得不通过改变参数来“返回” 其余的值. 幸运的是，在 Common Lisp 里不必这样做, 因为任何函数都可以返回多值.

举例来说，内置函数 truncate 返回两个值, 被截断的整数, 以及原来数字的小数部分. 在典型的实现中, 在最外层调用这个函数时两个值都会返回:

```
> (truncate 26.21875)
26
0.21875
```

当调用方只需要一个值时, 被使用的就是第一个值:

```
> (= (truncate 26.21875) 26)
T
```

通过使用 multiple-value-bind，调用方代码可以两个值都捕捉到. 该操作符接受一个变量列表, 一个调用, 以及一段程序体. 变量将被绑定到函数调用的对应返回值，而这段程序体会依照绑定后的变量求值:

```
> (multiple-value-bind (int frac) (truncate 26.21875)
    (list int frac))
(26 0.21875)
```

最后, 为了返回多值, 我们使用 values 操作符:

```
> (defun powers (x)
    (values x (sqrt x) (expt x 2)))
POWERS
```

```
> (multiple-value-bind (base root square) (powers 4)
    (list base root square))
(4 2.0 16)
```

一般来说，函数式编程是个好主意. 对于 Lisp 来说尤其如此, 因为 Lisp 在演化过程中已经支持了这种编程方式. 诸如 `reverse` 和 `nreverse` 这样的内置操作符的本意就是以这种方式被使用的. 其他操作符, 例如 `values` 和 `multiple-value-bind`, 是为了便于进行函数式编程而专门提供的.

3.2 内外颠倒的命令式

函数式程序代码的用意和那些更常见的方法, 即命令式程序相比可能显得更加明确一些. 函数式程序告诉你它想要什么; 命令式程序告诉你它要做什么. 函数式程序说 “返回一个由 `a` 和 `x` 的第一个元素的平方所组成的列表:”

```
(defun fun (x)
  (list 'a (expt (car x) 2)))
```

而命令式程序则会说 “取得 `x` 的第一个元素, 把它平方, 然后返回由 `a` 和那个平方所组成的列表”:

```
(defun imp (x)
  (let (y sqr)
    (setq y (car x))
    (setq sqr (expt y 2))
    (list 'a sqr)))
```

Lisp 程序员有幸可以同时用这两种方式来写程序. 某些语言只适合于命令式编程——尤其是 Basic, 以及大多数机器语言. 事实上, `imp` 的定义和多数 Lisp 编译器从 `fun` 生成的机器语言代码在形式上很相似.

既然编译器能为你做，为什么还要自己写这样的代码呢？对于许多程序员来说，这甚至不是个问题。语言给我们的思想打上烙印：一些过去使用命令式编程的人已经开始习惯于命令式地构建程序，并且会觉得写命令式程序比写函数式程序容易。这一思维定势是值得克服的，如果有一种语言可以帮助你做到的话。

对于其他语言的同行来说, 刚开始使用 Lisp 可能像第一次踏入溜冰场那样. 事实上在冰上比在干地面上更容易行走——如果使用溜冰鞋的话. 然后你对这项运动的看法就会彻底改观.

溜冰鞋对于冰的意义, 和函数式编程对 Lisp 的意义是一样的. 这两样东西在一起允许你更优雅地移动, 事半功倍. 但如果你已经习惯于另一种行走模式, 开始的时候你就不能感觉到这一点. 把 Lisp 作为第二语言学习的一个障碍就是学会如何用函数式的风格来编程.

幸运地是, 有一种把命令式程序转换成函数式程序的诀窍. 开始时你可以把这一诀窍用到已完成的代码里. 不久以后你就能够预想到这个过程, 在写代码的

同时做转换了. 在这之后一段时间, 你就可以从一开始就用函数式的思想构思你的程序了.

这个诀窍就是认识到命令式程序其实是一个从里到外翻过来的函数式程序. 要想翻出命令式程序中蕴含的函数式的那个, 也只需从外到里翻一下. 让我们在 imp 上实验一下这个技术.

我们首先注意到的是初始 let 里 y 和 sqr 的创建. 这就标志着以后不会有好事了. 就像运行期的 eval, 需要未初始化变量的情况很罕见, 它们因而被看作程序出现问题的征兆. 这些变量就像插在程序上, 用来固定的图钉, 它们被用来防止程序自己卷回到原形.

不过，我们暂时先不考虑它们，直接看函数的结尾。一个命令式程序里最后发生的事情，也就是函数式程序最外层发生的事情。所以我们的第一步是抓取最终对 list 的调用，并且把程序的其余部分塞到里面去——就好像把一件衬衫从里到外翻过来。我们继续重复做相同的转换，就好像我们先翻衬衫的袖子，然后再翻袖口那样。

从结尾处开始, 我们将 sqr 替换成 (expt y 2), 得到:

```
(list 'a (expt y 2))
```

然后我们将 y 替换成 (car x):

```
(list 'a (expt (car x) 2))
```

现在我们可以把其余代码扔掉了, 已经把所有内容都填到最后一个表达式里了. 在这个过程中我们消除了对变量 y 和 sqr 的需要, 也可以把 let 扔掉了.

最终的结果比开始的时候要短小, 而且更加好懂. 在原先的代码里, 我们面对最终的表达式 (list 'a sqr), 无法立即弄清 sqr 的值从何而来. 现在，返回值的来历则像交通指示图一样一览无余.

本章的这个例子很短, 但这里提到的技术是可以推而广之的. 的确，它对于大型函数来说更有价值. 即使那些有副作用的函数, 也可以把其中没有副作用的那部分清理得干净一些.

3.3 函数式接口

某些副作用比其他的更糟糕. 例如, 尽管这个函数调用了 nconc

```
(defun qualify (expr)
  (nconc (copy-list expr) (list 'maybe)))
```

但它没有破坏引用透明.[2] 如果你用一个给定参数调用它, 它总是返回相同 (equal) 的值. 从调用者的角度来看, qualify 就和纯函数型代码一样. 但我们不能对 bad-reverse (第 26 页) 也说同样的话, 那个函数确实修改了它的参数.

[2] 关于引用透明的定义见某页

如果不把所有副作用的有害程度都划上等号, 而是有方法能区别对待这些情况, 那样将会对我们很有帮助. 我们可以非正式地说, 如果一个函数修改某些其他函数都不拥有的东西, 那么它就是无害的. 例如, `qualify` 里的 `nconc` 就是无害的, 因为作为第一个参数的列表是新生成的. 没其他函数拥有它.

通常情况下, 我们在谈论拥有者关系时, 并不说一个变量被函数所拥有, 而是必须说被函数的调用所拥有. 尽管这里并没有其他函数拥有变量 x,

```
(let ((x 0))
  (defun total (y)
    (incf x y)))
```

但一次调用的效果会在接下来的调用中看到. 所以规则应当是: 一个给定的调用 (invocation) 可以安全地修改它唯一拥有的东西.

究竟谁是参数和返回值的拥有者? 依照 Lisp 的习惯，是函数的调用拥有那些作为返回值得到的对象, 但它并不拥有那些作为参数传给它的对象. 凡是修改它们参数的函数都应该打上 “破坏性的” 的标签，以示区别, 但对于那些修改了返回给它们的对象的那些函数就没有特别的分类了.

接下来的这个函数支持了上面说法, 例如:

```
(defun ok (x)
  (nconc (list 'a x) (list 'c)))
```

它调用了 `nconc`, 由于被 `nconc` 拼接的列表总是新建的而非传给 `ok` 作为参数的那个列表, 所以 `ok` 本身也是好的.

如果稍微写得不同一点儿, 例如:

```
(defun not-ok (x)
  (nconc (list 'a) x (list 'c)))
```

那么对 `nconc` 的调用就会修改传给 `not-ok` 的参数了.

许多 Lisp 程序没有遵守这一惯例, 至少在局部上是这样. 尽管如此, 正如我们从`ok` 那里看到的那样, 局部的违背不会让主调函数变质. 而且那些与前述情况相符的函数仍然会保留很多纯函数式代码的优点.

要想写出真正意义上的函数式代码, 我们还要再增加一个条件. 函数不能和其他不遵守这些规则的代码共享对象. 例如, 尽管这个函数没有副作用,

```
(defun anything (x)
  (+ x *anything*))
```

但它的返回值依赖于全局变量 `*anything*`. 这样如果任何其他函数可以改变这个变量的值, `anything` 就可以返回任何东西.

要是把代码写成让每次调用都只修改它自己拥有的东西的话, 那这样的代码就基本上和纯的函数式代码一样好了. 一个满足前述所有条件的函数至少对外界看来具有一个函数式接口: 如果你用同一个参数调用它两次, 你应当会得到同

样的结果. 并且这也是, 正如下一章所展示的那样, 自底向上程序设计最重要的组成部分.

破坏性操作符的一个问题是, 就像全局变量那样, 它们将破坏程序的局部性. 当你写函数式代码时, 你可以集中注意力: 你只需考虑这个函数调用的那些函数, 被哪些函数调用, 以及你正在写的这个函数. 当你想要破坏性地修改某些东西时这个好处就消失了. 它可能被任何地方用到.

上面的条件不能保证你能得到和纯粹的函数式代码一样的局部性, 尽管它们确实在某种程度上有所改进. 例如, 假设 `f` 调用了 `g`, 如下:

```
(defun f (x)
  (let ((val (g x)))
    ; safe to modify val here?
    ))
```

在 `f` 里把某些东西 nconc 到 `val` 上面安全吗? 如果 `g` 是 `identity` 的话就不安全: 这样我们就修改了某些原本作为参数传给`f` 本身的东西.

所以, 就算我们要修改那些按照这个规定写就的程序, 还是不得不看看 `f` 之外的东西. 虽然要多操心一些, 但也用不着看得太多: 现在我们不用复查整个程序的代码, 只消考虑从 `f` 开始的那棵子树就行了.

一个推论是函数不应当返回任何不能安全修改的东西. 这样的话, 就应当避免写那些返回包含引用对象的函数. 如果我们定义 `exclaim` 使它的返回值包含一个引用列表,

```
(defun exclaim (expression)
  (append expression '(oh my)))
```

那么任何后续的对返回值的破坏性修改

```
> (exclaim '(lions and tigers and bears))
(LIONS AND TIGERS AND BEARS OH MY)
> (nconc * '(goodness))
(LIONS AND TIGERS AND BEARS OH MY GOODNESS)
```

将替换函数里的列表:

```
> (exclaim '(fixnums and bignums and floats))
(FIXNUMS AND BIGNUMS AND FLOATS OH MY GOODNESS)
```

为了避免 `exclaim` 的这个问题, 它应该被写成:

```
(defun exclaim (expression)
  (append expression (list 'oh 'my)))
```

对于“函数不应返回引用列表”这一规则有一个主要的例外: 那些生成宏展开的函数. 宏展开器可以安全地在它们的展开式里包含引用的列表, 如果这些展开式是直接送到编译器那里的.

其他方面, 对于引用的列表应当总是持怀疑态度. 它们的许多其他用法像是某些原本就应当用诸如 `in` (某页) 这样的宏来完成的.

3.4 交互式编程

前一章说明了函数式的编程风格是一种组织程序的好办法. 但它的好处还不止于此. Lisp 程序员并非完全从美感出发才采纳函数式风格的. 他们使用这种风格是因为它使他们的工作更轻松. 在 Lisp 的动态环境里, 函数式程序能以非同寻常的速度写就, 与此同时, 写出的程序也非同寻常的可靠.

在 Lisp 里调试程序相对简单. 很多信息在运行期是可见的, 可以帮助追查错误的根源. 但更重要的是你可以轻易地测试程序. 你不需要编译一个程序然后一次性测试所有东西. 你可以在 toplevel 循环里通过逐个地调用每个函数来测试它们.

增量测试非常有用, 为了更好地利用它, Lisp 风格也有了相应的改进. 用函数式风格写出的程序可以每次一个函数地去理解它, 从读者的观点来看这是它的主要优点. 而且, 函数式风格也及其适合增量测试: 以这种风格写出的程序可以每次一个函数地进行测试. 当一个函数既不检查也不改变外部状态时, 任何 bug 都会立即现形. 这样, 一个函数影响外面世界的唯一渠道是通过其返回值, 在你能想到的范围内, 你可以完全相信那些产生它们的代码.

事实上有经验的 Lisp 程序员会尽可能地让他们的程序便于测试:

1. 他们试图把副作用分离到少量函数里, 以便程序中更多的部分可以写成纯函数式风格.
2. 如果一个函数必须产生副作用, 他们至少试图给它一个函数式的接口.
3. 他们给每个函数赋予一个单一的, 定义良好的功能.

当一个函数按照这种办法写成, 程序员们就可以用一组有代表性的情况进行测试, 测试好了然后就使用另一组情况测试. 如果每一块砖都各司其职, 那么墙就会屹立不倒.

在 Lisp 里, 墙可以得到更好的设计. 设想那种跟某人的距离远到有一分钟传输延迟的对话. 现在想象跟隔壁房间里某人说话. 你将不只是得到一个同样但是更快的对话, 而是将得到一个完全不同类型的对话. 在 Lisp 中, 开发软件就像是面对面的交谈. 你可以边写代码边做测试. 和对话相似，即时的回应对于开发来说一样有戏剧化的效果. 你不只是把同一个程序写得更快, 而是写出不同类型的程序了.

怎么会这样? 当测试更便捷时你就可以更频繁地进行测试. 对于 Lisp, 就像其他语言那样,开发是一个编码和测试的循环往复的周期过程. 但在 Lisp 里这一周期更短: 单个函数, 甚至函数的一部分都可以成为一个开发周期. 并且如果你边写代码边测试的话, 当错误发生时你就知道该检查哪里: 应该看看最后写的那部分. 正如听起来那样简单, 这一原则极大地增强了自底向上编程的可行性. 它带来了额外的信赖感, 使得 Lisp 程序员至少在一定程度上从旧式的计划–实现的软件开发风格中解脱了出来.

第 1.1 节强调了自底向上的设计是一个进化的过程。在这个过程中，你在写程序的同时也就是在构造一门语言。这一方法只有当你信赖底层代码时才可行。如果你真的想把这一层作为语言来使用，你就必须假设, 如同使用其他语言时那样，任何你遇到的 bug 都是你程序里的 bug，而不是语言本身的。

这样你的新抽象就必须承担这一重任，同时你还要能随时根据新的需求演变它们吗？就是这样；在 Lisp 里你可以同时做到这两件事。当你用函数式风格写程序并且增量测试它们的时候，你可以得到随心所欲的灵活性，加上通过只有仔细计划才能得到的可靠性。

第 4 章

实用函数

Common Lisp 操作符分为三种类型: 可以自定义的函数和宏, 以及不能自定义的特殊形式 (special form). 本章将讲述用函数来扩展 Lisp 的技术. 但这里的 "技术" 和通常的意思不太一样. 关于这些函数, 重要的不是要知道怎样写, 而是要知道它们从何而来. 编写 Lisp 扩展所使用的技术和你编写其他任何 Lisp 函数所使用的技术大同小异. 编写 Lisp 扩展的难点并不在于代码怎么写，而是要决定写什么.

4.1 实用工具的诞生

在最简单的情形, 自底向上程序设计意味着猜测究竟是谁设计了你的 Lisp. 在你写程序的同时, 你也在为 Lisp 增加那些可以让你程序更容易编写的新操作符. 这些新操作符被称为*实用工具*.

"实用工具" 这一术语并无精确的定义. 有一段代码, 如果把它看成一个单独的程序的话, 感觉太小, 要是把它作为特定程序的一个组成部分的话, 这段代码又太通用了, 这时就可以称之为实用工具. 举例来说, 一个数据库不能称为实用工具, 但是一个在列表上做单一操作的函数就可以. 大多数实用工具和 Lisp 已有的函数和宏很相似. 事实上, 许多 Common Lisp 内置的操作符就源自实用工具. 用于收集列表中所有满足条件元素的 `remove-if-not` 函数, 在它成为 Common Lisp 的一部分以前, 就被程序员们各自私下里定义了多年.

学习编写实用工具与其说是学习编写的技术, 不如说是学习养成编写实用工具的习惯. 自底向上程序设计意味着同时写程序和编程语言. 为了做好这一点, 你必须培养出一种能看出程序中缺少何种操作符的洞察力. 你必须能够在看到一个程序时说, "啊, 其实你真正的意思是*这个*."

举个例子, 假设 `nicknames` 是这样一个函数, 它接受一个名字，然后构造出一个列表，列表由这个名字的所有昵称组成. 有了这个函数, 我们怎样收集一个名字列表所对应的所有昵称呢? 某个正在学习 Lisp 的人可能写出类似这样的函数:

```
(defun all-nicknames (names)
  (if (null names)
      nil
```

```
      (nconc (nicknames (car names))
             (all-nicknames (cdr names)))))
```

而一个更有经验的 Lisp 程序员可能一看到这样的函数就会说“啊, 其实你真正想要的是 mapcan.” 然后, 不再被迫定义并调用一个新函数来找出一组人的所有昵称, 现在只消用一个表达式就够了:

```
(mapcan #'nicknames people)
```

all-nicknames 的定义完全是在重复地发明轮子. 它的问题还不只于此: 它同时也葬送了一个机会: 本可以用通用操作符来直接完成某件事, 却使用了一个专用函数来实现它.

在这种情况下操作符 mapcan 已经存在了. 任何知道 mapcan 的人在看到 all-nicknames 时都会觉得有点不太舒服. 要想在自底向上程序设计方面做得好就要在缺少的操作符尚未被写出时同样觉得不舒服. 你必须能够在说出“你真正想要的是 x”的同时, 知道 x 应该是什么.

和其他技术不同, Lisp 编程伴随着新实用工具的按需产生. 本章的目的就是展示这些工具是怎样产生的. 假设 towns 是一个附近城镇的列表, 按从近到远排序, bookshops 函数返回一个城市中所有书店的列表. 如果想要查找最近的一个有书店的城市以及该城市里的书店, 我们可能一开始会这样做:

```
(let ((town (find-if #'bookshops towns)))
  (values town (bookshops town)))
```

但是这样有个不妥之处: 当 find-if 找到了一个 bookshops 返回非空值的元素时, 这个值被直接丢掉了, 然后马上又要重新计算一次. 如果 bookshops 是一个耗时的函数调用, 那么这个用法将是既丑陋又低效的. 为了避免不必要的工作, 我们用下列函数来代替:

```
(defun find-books (towns)
  (if (null towns)
      nil
      (let ((shops (bookshops (car towns))))
        (if shops
            (values (car towns) shops)
            (find-books (cdr towns))))))
```

这样, 调用 (find-books towns) 至少能得到我们想要的结果, 并且避免了不必要的计算. 但是别急, 我们在将来的某一天会不会再做一次类似的搜索呢? 这里我们真正想要的是一个实用工具, 它集成了 find-if 和 some 的功能, 并且能返回符合要求的元素和判断函数的返回值. 这样的一个实用工具可能被定义成:

```
(defun find2 (fn lst)
  (if (null lst)
      nil
      (let ((val (funcall fn (car lst))))
```

```
    (if val
        (values (car lst) val)
        (find2 fn (cdr lst))))))
```

注意到 `find-books` 和 `find2` 之间的相似程度. 的确, 后者可以看作是前者提炼后的结果. 现在, 借助这个新的实用工具, 我们就可以用单个表达式达到我们最初的目标了:

```
(find2 #'bookshops towns)
```

Lisp 编程有一个独一无二的特征, 就是函数在作为参数时扮演了一个重要的角色[1]. 这也是 Lisp 被广泛采纳用于自底向上程序设计的部分原因. 当你能把一个函数的筋肉作为函数型参数传进函数时, 你就可以更轻易地从这个函数中抽象出它的神髓.

程序设计的入门课程一开始就教授如何通过这种抽象来减少重复的努力. 前几课的内容之一就是: 切忌把程序的行为写死在代码里面[2]. 与其定义两个函数, 它们几乎做相同的事但其中只有一两个常量不同, 不如定义成一个函数然后把那些常量以参数的形式传给它. 在 Lisp 里我们可以进一步发展这个观点, 因为我们可以把整个函数都作为参数传递. 在过去两个例子里, 我们都从一个专用的函数走向了带有函数型参数的更为通用的函数. 虽然在第一个例子里我们用了预定义的 `mapcan`, 第二个例子里则写了一个新的实用工具, `find2`, 但都遵循了同样的一般原则: 与其将通用的和专用的混在一起, 不如定义一个通用的然后把专用的部分作为参数.

如果慎重使用这个原则, 就会得到明显更为优雅的程序. 它不是驱动自底向上程序设计的唯一方法, 但却是主要的一个. 本章定义的 32 个实用工具里, 有 18 个带有函数型参数.

4.2 在抽象上投资

如果说简洁是智慧的灵魂, 那么它和效率也同是优秀软件的本质特征. 编写和维护一个程序的开销与其长度成正比.同等条件下, 程序越短越好.

从这一角度来看, 编写实用工具可以被视为一种投资. 通过把 `find-books` 替换成实用工具, `find2`, 最后我们得到的仍是那么多行程序. 但从某种角度来看我们确实让程序变得更短了, 因为实用工具的长度不必算在当前这个程序的帐上.

把对 Lisp 的扩展看作资本支出并不只是会计上的手段. 实用工具可以放在单独的文件里; 它们既不会在我们编写程序时分散我们的精力, 也不会在事后因为某些原因我们去修改以往写的程序时被卷入.

[1] 译者注: 随着诸如 Haskell 这样的纯函数型语言的出现, Lisp 的这一特征可能不再是独一无二的了.

[2] 译者注: 原文为“don't wire in behavior”.

尽管如此, 作为一项投资, 实用工具还是需要得到额外关注. 尤其重要的一点是它们必须写得好. 由于它们要被重复使用, 所以任何不正确或者低效率之处都将会成倍地偿还. 除此之外, 还要注意它们的设计: 一个新的实用工具必须为通用场合而作, 而不是仅仅着眼于手头的问题. 最后, 和任何其他资本支出一样, 我们不能急于求成. 如果你考虑创造一些新操作符作为程序开发的副产品, 但又不敢确定以后在其他场合还能用到它们, 那就先做出来, 但只把它和使用到它的个别程序放在一起. 等以后如果你在其他程序里用到这些操作符了, 就可以把它们从子程序提升到实用工具的层面然后将它们通用化.

`find2` 这个实用工具看来是一笔好的投资. 通过 7 行代码的本金, 我们立即得到了 7 行收益. 这一实用工具在首次使用时就已收回成本了. Guy Steele 写道, 一门编程语言应该"顺应我们追求简洁的自然倾向:"

> ... 我们倾向于相信一种编程构造产生的开销与它所导致的编程者的不适程度成正比 (我这里所说的"相信"指的是下意识的倾向而非有意的好恶). 确实, 对于语言设计者来说, 的确应该把这个心理学原则熟记于心.我们认为加法的成本较低, 部分的原因是由于我们可以用单一的字符 "+" 来表示它. 即使一种编程构造开销较大, 如果我们写代码的时候能比其他更便宜的方法省一半力气的话, 也会更喜欢用它.

在任何语言里, 这一"对简洁代码的倾向性"将招致麻烦, 除非允许用新的实用工具来表达自身. 最简短的表达方式很少是最高效的. 如果我们想知道一个列表是否比另一个列表更长, 原始的 Lisp 将诱使我们写出

```
(> (length x) (length y))
```

如果我们想将一个函数映射到几个列表上, 我们可能同样会有先将这些列表连接在一起的想法:

```
(mapcar fn (append x y z))
```

这些例子说明编写实用工具对于某些情形尤为重要: 一不小心就会误入低效率的歧途. 一个装备了合适的实用工具的语言将会使我们写出更为抽象的程序. 如果这些实用工具被合理的定义了, 它们也会促使我们进一步写出更强的实用工具来.

一组实用工具集将无疑会使整个编程工作更为简单. 但它们还有更重要的作用: 它们会让你写出更好的程序. 厨师看到对味的食材会忍不住动手烹饪, 文人骚客们也一样, 他们有了合适的题材就会文思如泉涌. 这就是为何艺术家们喜欢在他们的工作室里放很多工具和材料. 他们知道如果手头已经有他们需要的东西, 就会更想开始一项新的创作. 同样的现象也出现在自底向上编写的程序中. 一旦你已写好一个新的实用工具, 你可能发现对它的使用往往超乎预想.

```
(proclaim '(inline last1 single append1 conc1 mklist))

(defun last1 (lst)
  (car (last lst)))

(defun single (lst)
  (and (consp lst) (not (cdr lst))))

(defun append1 (lst obj)
  (append lst (list obj)))

(defun conc1 (lst obj)
  (nconc lst (list obj)))

(defun mklist (obj)
  (if (listp obj) obj (list obj)))
```

图 4.1: 操作列表的一些小函数

接下来的章节描述几类实用函数. 它们绝不代表所有你可能追加到 Lisp 里的函数的类型. 尽管如此, 这里作为示例给出的所有实用工具都已经在实践中被充分证明了存在价值.

4.3 列表上的操作

列表最初曾是 Lisp 的主要数据结构. 事实上, "Lisp" 这个名字就来自"LISt Processing (列表处理)". 不过请不要被这一历史事实所蒙蔽, 尽管如此, Lisp 跟列表处理之间的关系并不比马球衫 (Polo shirts) 和马球之间的关系好多少. 一个高度优化的 Common Lisp 程序里可能根本看不到列表.

尽管如此, 至少在编译期它们还是列表. 最专业的程序, 在运行期很少使用列表, 相反可能会在编译期生成宏展开式时大量使用列表. 所以尽管列表的角色在现代 Lisp 方言里被削弱了, 列表上的各种操作在 Lisp 程序里仍然是重要部分.

图 4.1 和 4.2 里包括了一些构造和检查列表的函数. 那些在图 4.1 中给出的都是些值得定义的最小实用工具. 为了满足效率的需要, 它们应该全部被声明为 inline. (见 23 页)

第一个函数, `last1`, 返回一个列表的最后一个元素. 内置的 `last` 函数其实返回的是列表的最后一个点对, 而不是最后一个元素. 多数时候人们都是通过 `(car (last ...))` 的方式来得到其最后一个元素的. 它是否值得我们为这种情况写一个新的实用工具吗? 是的, 如果它可以有效地替代一个内置操作符, 那么就值得.

注意到 `last1` 没有任何错误检查. 一般而言, 本书中定义的代码都将不做任何错误检查. 部分只是为了使这些示例代码更加清晰. 但是在相对短小的实用工

```
(defun longer (x y)
  (labels ((compare (x y)
             (and (consp x)
                  (or (null y)
                      (compare (cdr x) (cdr y))))))
    (if (and (listp x) (listp y))
        (compare x y)
        (> (length x) (length y)))))

(defun filter (fn lst)
  (let ((acc nil))
    (dolist (x lst)
      (let ((val (funcall fn x)))
        (if val (push val acc))))
    (nreverse acc)))

(defun group (source n)
  (if (zerop n) (error "zero length"))
  (labels ((rec (source acc)
             (let ((rest (nthcdr n source)))
               (if (consp rest)
                   (rec rest (cons (subseq source 0 n) acc))
                   (nreverse (cons source acc))))))
    (if source (rec source nil) nil)))
```

图 4.2: 操作列表的一些较大函数

具里不做任何错误检查也是有原因的. 如果我们试一下这个:

```
> (last1 "blub")
>>Error: "blub" is not a list.
Broken at LAST...
```

这一错误将被 last 本身捕捉到. 当实用工具规模很小时, 它们从开始传递的位置开始形成的抽象层很薄. 正如可以看透的薄冰那样, 人们可以一眼看清像 last1 这种实用工具, 从而理解从它们底层抛出的错误.

single 函数判断某个东西是否为单元素列表. Lisp 程序经常需要做这种测试. 在一开始实现的时候可能会错误地采用来自英语的自然翻译:

```
(= (length lst) 1)
```

如果写成这个样子, 测试将会极其低效. 其实只要我们看完列表的第一个元素就知道所有我们想知道的事情了.

接下来是 append1 和 nconc1. 两个都是在列表结尾处追加一个新元素, 只不过后者是破坏性的. 这些函数很小, 但它们用得如此频繁，所以还是值得定义的. 而且 append1 在过去的 Lisp 方言里确实是被预定义了的.

然后是 mklist, 它 (至少) 在 Interlisp 里是被预定义了的. 它的目的是确保某个东西是列表. 很多 Lisp 函数被写成要么返回一个单一的值，要么返回一个由多个值组成的列表. 假设 lookup 是这样一个函数, 然后我们希望把该函数作用在称为 data 的列表里每一个元素上得到的结果收集在一起. 我们可以写成这样:

```
(mapcan #'(lambda (d) (mklist (lookup d)))
        data)
```

图 4.2 包括了一些更大的列表实用工具示例. 第一个, longer, 从效率而非抽象的观点上来看是有用的. 它比较两个列表, 然后只有当前一个列表更长的时候才返回真. 当比较两个列表的长度时, 很可能只写成这样:

```
(> (length x) (length y))
```

这样用之所以低效是因为它让程序从头到尾遍历两个列表. 如果一个列表的长度远远超过另一个, 那么在超出较短列表长度上的进行的所有遍历操作将都是徒劳. 像 longer 那样做并且并行地遍历两个列表会快一些.

嵌入在 longer 里面的是一个用来比较两个列表长度的递归函数. 不过由于 longer 只是用来比较长度的, 它应该可以用在任何可以作为参数传给 length 的对象的长度比较上. 但是并行比较长度只可能在列表上发生, 所以这个内部函数只有当两个参数都是列表时才被调用.

下一个函数, filter, 它对于 some 来说相当于 remove-if-not 对 find-if. 内置的 remove-if-not 返回所有的这些值: 当你用 find-if 带着相同的函数依次在一个列表的 cdr 上调用时返回的值. 类似地, filter 返回那些 some 依次作用在列表 cdr 上时的返回值:

```
> (filter #'(lambda (x) (if (numberp x) (1+ x)))
          '(a 1 2 b 3 c d 4))
(2 3 4 5)
```

你给 filter 一个函数和一个列表, 然后得到一个当把函数作用在列表的每个元素上返回的非空值所组成的列表.

注意到 filter 使用了一个累加器, 它的工作方式和第 2.8 章描述的尾递归函数一样. 实际上, 编写尾递归函数的目的就是让编译器能够生成形如 filter 那样的代码. 对于 filter 来说, 这种直接的迭代定义比尾递归的形式来得简单. 对于列表聚积操作来说, filter 定义中的 push 和 nreverse 组合是标准的 Lisp 句法.

图 4.2 中的最后一个函数用来将列表分组成子列表. 你给 group 一个列表 l 和一个数字 n, 那它将返回一个新列表, 由列表 l 的元素按长度为 n 的子列表组成. 最后剩余的元素放在最后一个子列表里. 这样如果我们给出 2 作为第二个参数, 我们就得到一个关联表 (assoc–list):

```
> (group '(a b c d e f g) 2)
((A B) (C D) (E F) (G))
```

为了把 group 写成尾递归的 (见第 2.8 节)，这个函数编得有些拐弯抹角。快速原型开发的基本道理可以用在整个程序的开发上, 但它对于单个函数的编写也一样适用. 当写一个像 flatten 这样的函数时, 从最简单的可能实现方式开始可能是个好主意. 然后, 一旦这个最简版本可用了, 如果有必要的话，你就可以用更有效率的迭代或者尾递归版本来代替它. 如果最早的版本足够短小, 可以把它以注释的形式留下来用于表述它的复杂替代者的行为. (group 和图 4.1, 4.3 中其他函数的简化版本包含在书后位于某页的 note 里.)

group 定义的与众不同之处在于它至少检查了一种错误: 如果第二个参数为 0, 那么这个函数就会陷入一个无限的递归之中.

从某种意义上讲, 本书的示例也遵循了通常的 Lisp 实践经验: 使章节之间彼此不相互依赖, 示例代码尽可能用原始 Lisp 来写. 但考虑到在定义宏的时候，group 函数会非常有用, 因而它会是个例外, 这个函数将再次出现在后续章节的某些地方.

图 4.2 中的所有函数都是作用在列表的最上层 (top–level) 结构上. 图 4.3 给出了两个下降到嵌套列表里的函数示例. 前一个, flatten, 也是 Interlisp 预定义的. 它返回由一个列表中的所有原子 (atom), 或者说是元素的元素所组成的列表, 也就是:

```
> (flatten '(a (b c) ((d e) f)))
(A B C D E F)
```

图 4.3 中的另一个函数, prune, 它对于 remove-if 的意义就相当于copy-tree 对于 copy-list 那样. 也就是说, 它会向下递归到子列表里:

```
(defun flatten (x)
  (labels ((rec (x acc)
             (cond ((null x) acc)
                   ((atom x) (cons x acc))
                   (t (rec (car x) (rec (cdr x) acc))))))
    (rec x nil)))
(defun prune (test tree)
  (labels ((rec (tree acc)
             (cond ((null tree) (nreverse acc))
                   ((consp (car tree))
                    (rec (cdr tree)
                         (cons (rec (car tree) nil) acc)))
                   (t (rec (cdr tree)
                           (if (funcall test (car tree))
                               acc
                               (cons (car tree) acc)))))))
    (rec tree nil)))
```

图 4.3: 使用双递归的列表实用工具

```
> (prune #'evenp '(1 2 (3 (4 5) 6) 7 8 (9)))
(1 (3 (5)) 7 (9))
```

每一个函数返回真的叶子都被删掉了.

4.4 搜索

本节给出一些用于搜索列表的函数示例. 虽然 Common Lisp 已经提供了丰富的内置函数来完成同样的功能, 但对于某些任务来说光靠这些函数仍然有些捉襟见肘–或者说它们至少无法高效地完成功能. 我们在第 35 页里描述的那个假想案例就说明了这一点.图 4.4 中的第一个实用工具, `find2`, 就是我们回应此需求的一个定义.

下一个实用工具, `before`, 也是为了相似的意图而写的. 它告诉你在一个列表中的对象是否在另一个对象的前面:

```
> (before 'b 'd '(a b c d))
(B C D)
```

在原始 Lisp 里这个需求可以足够容易地草草写就:

```
(< (position 'b '(a b c d)) (position 'd '(a b c d)))
```

但是后面这句话既低效又易错: 低效率是因为我们不需要两个对象都找, 只需找到前一个对象即可; 而容易出错是因为如果两个对象中的任何一个不在列表里, 那么 `nil` 将被返回作为 `<` 的参数. 使用 `before` 可以同时修补这两个问题.

```
(defun find2 (fn lst)
  (if (null lst)
      nil
      (let ((val (funcall fn (car lst))))
        (if val
            (values (car lst) val)
            (find2 fn (cdr lst))))))

(defun before (x y lst &key (test #'eql))
  (and lst
       (let ((first (car lst)))
         (cond ((funcall test y first) nil)
               ((funcall test x first) lst)
               (t (before x y (cdr lst) :test test))))))

(defun after (x y lst &key (test #'eql))
  (let ((rest (before y x lst :test test)))
    (and rest (member x rest :test test))))

(defun duplicate (obj lst &key (test #'eql))
  (member obj (cdr (member obj lst :test test))
          :test test))

(defun split-if (fn lst)
  (let ((acc nil))
    (do ((src lst (cdr src)))
        ((or (null src) (funcall fn (car src)))
         (values (nreverse acc) src))
      (push (car src) acc))))
```

图 4.4: 搜索列表的函数.

由于感觉上 before 和测试成员关系非常相似, 它被定义成跟内置的 member 函数很相似. 就像 member 那样带有一个可选的 test 参数, 默认设置为 eql. 同时, 它不再简单地返回一个 t, 而是试图返回潜在的有用信息: 以作为第一个参数给出的对象为首的 cdr.

注意到 before 函数如果在第二个参数之间遇到第一个参数就会立即返回. 这样如果第二个参数在列表中不存在它将返回真:

```
> (before 'a 'b '(a))
(A)
```

通过调用 after 我们可以做更为细致的测试, 要求两个参数都出现在列表里:

```
> (after 'a 'b '(b a d))
(A D)
> (after 'a 'b '(a))
NIL
```

如果 (member *o l*) 在列表 *l* 里找到了 *o*, 它会同时返回列表 *l* 中以 *o* 开头的那个 cdr. 这一返回值可以被用来, 例如, 找出列表中的重复元素. 如果 *o* 在列表 *l* 中重复出现, 那么它可以用 member 在返回的列表 cdr 中找到. 这一句法被包含在下一个实用工具中, duplicate:

```
> (duplicate 'a '(a b c a d))
(A D)
```

其他用于检测重复性的实用工具也可以采用相同的原则被写出来.

很多挑剔的语言设计者为 Common Lisp 使用 nil 同时代表逻辑假和空列表感到惊异. 某些时候这确实会带来麻烦 (见 182 页), 但对于像 duplicate 这样的函数来说则非常便利. 至于判断元素是否属于一个序列 (sequence) 的那些函数, 用空序列来表示否定的结果还是比较符合常理的.

图 4.4 的最后一个函数也是 member 的某种泛化. member 返回传入列表的 cdr，从找到的元素开始, 而 split-if 把原列表的两个部分都返回了. 该实用工具主要用于以某种方式排序的列表:

```
> (split-if #'(lambda (x) (> x 4))
            '(1 2 3 4 5 6 7 8 9 10))
(1 2 3 4)
(5 6 7 8 9 10)
```

图 4.5 包含有另一种类型的搜索函数: 它们在列表元素之间进行比较. 第一个函数 most, 每次查看一个元素. 它接受一个列表和一个用来打分的函数, 然后返回分数最高的元素. 分数相等的时候排在前面的元素优先.

```
> (most #'length '((a b) (a b c) (a) (e f g)))
(A B C)
3
```

```
(defun most (fn lst)
  (if (null lst)
      (values nil nil)
      (let* ((wins (car lst))
             (max (funcall fn wins)))
        (dolist (obj (cdr lst))
          (let ((score (funcall fn obj)))
            (when (> score max)
              (setq wins obj
                    max score))))
        (values wins max))))

(defun best (fn lst)
  (if (null lst)
      nil
      (let ((wins (car lst)))
        (dolist (obj (cdr lst))
          (if (funcall fn obj wins)
              (setq wins obj)))
        wins)))

(defun mostn (fn lst)
  (if (null lst)
      (values nil nil)
      (let ((result (list (car lst)))
            (max (funcall fn (car lst))))
        (dolist (obj (cdr lst))
          (let ((score (funcall fn obj)))
            (cond ((> score max)
                   (setq max score
                         result (list obj)))
                  ((= score max)
                   (push obj result)))))
        (values (nreverse result) max))))
```

图 4.5: 带有元素比较的搜索函数.

出于便利, most 也返回了获胜元素的分数.

一个更加通用的搜索类型由 best 提供. 该实用工具接受一个函数和一个列表, 但这里的函数必须是个两参数谓词. 它返回的元素在该谓词下胜过所有其他元素.

```
> (best #'> '(1 2 3 4 5))
5
```

我们可以认为 best 等价于 sort 的 car, 但前者更有效率. 函数的调用者有责任提供一个能在列表所有元素上定义全序的谓词. 否则列表中元素的顺序将影响结果; 和之前一样, 在平手的情况下，先出场的元素获胜.

最后, mostn 接受一个函数和一个列表然后返回一个由最高分的所有元素组成的列表 (以及这个最高分本身):

```
> (mostn #'length '((a b) (a b c) (a) (e f g)))
((A B C) (E F G))
3
```

4.5 映射

还有一类广泛使用的 Lisp 函数是映射函数, 它们将一个函数应用到一个参数的序列上. 图 4.6 展示了一些新的映射函数示例. 开始的三个用来将一个函数应用到一个数值区间上而无需构造包含它们的列表. 前两个, map0-n 和 map1-n, 工作在正整数区间上:

```
> (map0-n #'1+ 5)
(1 2 3 4 5 6)
```

它们两个都用更通用的支持任意数值区间的 mapa-b 来写成:

```
> (mapa-b #'1+ -2 0 .5)
(-1 -0.5 0.0 0.5 1.0)
```

mapa-b 下面还有更为通用的 map->, 可以在任意类型的对象序列上工作. 序列开始于用第二个参数给出的对象, 序列的结束条件用第三个参数给出的函数来定义, 序列的后继元素则由第四个参数给出的函数来生成. 通过 map-> 有可能操纵任何数据结构, 就和数字序列的方法一样. 我们可以用 map-> 定义 mapa-b, 像这样:

```
(defun mapa-b (fn a b &optional (step 1))
  (map-> fn
         a
         #'(lambda (x) (> x b))
         #'(lambda (x) (+ x step))))
```

出于效率考虑, 内置的 mapcan 是破坏性的, 它也可用下列代码表达:

```
(defun map0-n (fn n)
  (mapa-b fn 0 n))

(defun map1-n (fn n)
  (mapa-b fn 1 n))

(defun mapa-b (fn a b &optional (step 1))
  (do ((i a (+ i step))
       (result nil))
      ((> i b) (nreverse result))
    (push (funcall fn i) result)))

(defun map-> (fn start test-fn succ-fn)
  (do ((i start (funcall succ-fn i))
       (result nil))
      ((funcall test-fn i) (nreverse result))
    (push (funcall fn i) result)))

(defun mappend (fn &rest lsts)
  (apply #'append (apply #'mapcar fn lsts)))

(defun mapcars (fn &rest lsts)
  (let ((result nil))
    (dolist (lst lsts)
      (dolist (obj lst)
        (push (funcall fn obj) result)))
    (nreverse result)))

(defun rmapcar (fn &rest args)
  (if (some #'atom args)
      (apply fn args)
      (apply #'mapcar
             #'(lambda (&rest args)
                 (apply #'rmapcar fn args))
             args)))
```

图 4.6: 映射函数.

```
(defun our-mapcan (fn &rest lsts)
  (apply #'nconc (apply #'mapcar fn lsts)))
```

由于 mapcan 用 nconc 把列表拼接在一起, 第一个参数返回的列表最好是新创建的, 否则等下次看的时候它可能就变了. 这也是为什么 nicknames (第 35 页) 被定义成一个根据昵称"生成列表" 的函数. 如果它简单地返回一个存放在其他地方的列表, 那么使用 mapcan 会很不安全. 替代方案是我们只能用 append 把返回的列表拼接在一起. 对于这类情况, mappend 提供了一个 mapcan 的非破坏性版本.

下一个实用工具, mapcars, 用于那些你想在一组列表上 mapcar 某个函数的场合. 如果我们有两个数列，并且希望得到一个它们的平方根列表, 用原始 Lisp 我们这样写:

```
(mapcar #'sqrt (append list1 list2))
```

但这里的 cons 没有必要. 我们将 list1 和 list2 串在一起之后立即又把结果丢弃了. 借助 mapcars 我们用下面的方法就可以得到相同结果:

```
(mapcars #'sqrt list1 list2)
```

而且还避免了多余的 cons.

图 4.6 中最后一个函数是 mapcar, 用于树的版本. 它的名字, rmapcar, 是"递归 mapcar" 的缩写, 并且所有 mapcar 在扁平列表上能完成的功能, 它都可以在树上做到:

```
> (rmapcar #'princ '(1 2 (3 4 (5) 6) 7 (8 9)))
123456789
(1 2 (3 4 (5) 6) 7 (8 9))
```

和 mapcar 一样, 它可以接受多于一个列表作为参数:

```
> (rmapcar #'+ '(1 (2 (3) 4)) '(10 (20 (30) 40)))
(11 (22 (33) 44))
```

后面出现的某些函数应该会调用 rmapcar, 包括某页上的 rep-on.

长远来看, 传统的列表映射函数可能会被 CLTL 中介绍的新的串行宏 (series macro) 所取代. 例如,

```
(mapa-b #'fn a b c)
```

可以被改写成[3]:

```
(collect (map-fn t #'fn (scan-range :from a :upto b :by c)))
```

[3]译者注: 原书的写法是 (collect (#Mfn (scan-range :from a :upto b :by c))), 两种写法是等价的. CLTL says: The # macro character syntax #M makes it easy to specify uses of map-fn where type is t and the function is a named function. The notation (#Mfunction ...) is an abbreviation for (map-fn t #'function ...). 由于目前串行宏的标准实现 cl-series 包在加载以后的缺省情况下并不定义 #M 这个宏, 所以这里采用了通俗写法.

```
(defun readlist (&rest args)
  (values (read-from-string
           (concatenate 'string "("
                                (apply #'read-line args)
                                ")"))))
(defun prompt (&rest args)
  (apply #'format *query-io* args)
  (read *query-io*))

(defun break-loop (fn quit &rest args)
  (format *query-io* "Entering break-loop.~%")
  (loop
    (let ((in (apply #'prompt args)))
      (if (funcall quit in)
          (return)
          (format *query-io* "~A~%" (funcall fn in))))))
```

图 4.7: I/O 函数.

尽管如此, 映射函数仍然是有需求的. 映射函数在某些场合更加清晰优雅. 一些我们可以轻易用 `map->` 表达的东西可能用 series 来表达会有困难. 最后, 映射函数, 和其他函数一样, 可以作为参数来传递.

4.6 I/O

图 4.7 给出了三个 I/O 实用工具的例子. 不同程序对这类实用工具的需要有所不同. 图 4.7 的那些只是些示例. 第一个用于你希望用户在输入表达式省略括号的时候; 它读入一行然后以列表形式返回:

```
> (readlist)
Call me "Ed"
(CALL ME "Ed")
```

函数定义中调用 `values` 是为了只得到一个返回值 (`read-from-string` 本身会返回第二个值，但这个值在这种情况下没有意义).

函数 `prompt` 将输出一个问题和读取答案结合起来. 它带有跟 `format` 函数类似的参数表, 除了一开始的流参数.

```
> (prompt "Enter a number between ~A and ~A.~%>> " 1 10)
Enter a number between 1 and 10.
>> 3
3
```

最后, 如果你希望模拟 Lisp 的顶层环境，那么 `break-loop` 可以帮上忙。它接受两个函数和一个 `&rest` 参数, 后者一次又一次地作为参数传给 `prompt`. 当输

```
(defun mkstr (&rest args)
  (with-output-to-string (s)
    (dolist (a args) (princ a s))))

(defun symb (&rest args)
  (values (intern (apply #'mkstr args))))

(defun reread (&rest args)
  (values (read-from-string (apply #'mkstr args))))

(defun explode (sym)
  (map 'list #'(lambda (c)
                 (intern (make-string 1
                                      :initial-element c)))
            (symbol-name sym)))
```

图 4.8: 操作符号和字符串的函数.

入使得第二个函数返回逻辑假的时候, 那第一个参数将会应用在这个输入上. 所以我们可以像这样来模仿真实的 Lisp 顶层环境:

```
> (break-loop #'eval #'(lambda (x) (eq x :q)) ">> ")
Enter break-loop.
>> (+ 2 3)
5
>> :q
:Q
```

随便说一下, 这也是 Common Lisp 厂商主张对运行期进行授权的原因. 如果你可以在运行期调用 `eval`, 那么任何 Lisp 程序都可以包含 Lisp.

4.7 符号和字符串

符号和字符串是紧密相关的. 通过打印和读取函数, 我们可以这两种表示之间来回转换. 图 4.8 包含了做这种边界操作的实用工具示例. 其中第一个是 `mkstr`, 它接受任意数量的参数，并将它们的打印形式连接起来，形成一个字符串:

```
> (mkstr pi " pieces of " 'pi)
"3.141592653589793 pieces of PI"
```

在 `mkstr` 的基础之上我们编写了 `symb`, 它大多数时候被用来构造符号. 它接受一个或更多的参数然后返回一个符号 (如需要的话，则会新建一个)，使其打印名称等于所有参数连接在一起的字符串. 它可以接受任何支持可打印表示的对象作为参数: 符号、字符串、数字，甚至列表.

```
> (symb 'ar "Madi" #L #L 0)
|ARMadiLL0|
```

symb 首先调用 mkstr 将所有参数连接成单个字符串, 然后把这个字符串发给 intern. 这个函数是 Lisp 传统上的符号构造器: 它接受一个字符串，然后，如果无法找到一个打印输出和该字符串相同的符号, 就创建一个满足此条件的新符号.

任何字符串都可以作为符号的打印名称, 甚至是一个包含有小写字母或者诸如括号这样的宏字符的字符串也可以. 当符号名称里包含这些奇怪的字符时, 它将被原样打印在两条竖线中间. 在源代码中, 这样的符号应该被放在两条竖线之间, 否则就必须用反斜线转义:

```
> (let ((s (symb '(a b))))
    (and (eq s '|(A B)|) (eq s 'A B)))
T
```

下一个函数 reread, 是 symb 的通用化版本. 它接受一系列对象, 然后打印并重读它们. 它可以像 symb 那样返回符号, 但也可以返回其他任何 read 能返回的东西. 其中读取宏将会被调用而不是被看作函数的一部分, 这样 a:b 将被读成包 (package) a 中的符号 b, 而不是当前包中的符号 |a:b|.[4] 这个更通用的函数同时也更加挑剔: 如果 reread 的参数不合 Lisp 语法, 它将生成一个错误.

图 4.8 中的最后一个函数在几种早期方言是预定义的: explode 接受一个符号，然后返回一个由该符号名称里的字符所组成的列表.

```
> (explode 'bomb)
(B O M B)
```

毫无疑问该函数不会被包含在 Common Lisp 中. 如果你发现自己需要处理符号本身, 那你很可能在做某件低效率的事情. 尽管如此, 在原型开发, 而非产品软件中, 这类实用工具还是有其用武之地的.

4.8 紧凑性

如果你的代码里使用了大量实用工具, 某些读者可能会抱怨它们难以理解. 那些还没能自如地使用 Lisp 的人只能习惯阅读原始的 Lisp. 事实上, 他们可能一直对可扩展语言的思想有隔阂. 当读到一个严重依赖于实用工具的程序时, 在他们来看作者可能是纯粹出于怪癖而决定用某种私人语言来写程序.

会有人提出, 所有这些新操作符使程序更难读了. 一个人必须首先理解所有的这些新操作符才能读懂程序. 要想知道为什么这类说法是错误的, 不妨考虑第 36 页的那个例子, 在那里我们想要找到最近的书店. 如果我们使用 find2 来写程序, 某人可能会抱怨说, 在他能够读懂这个程序之前不得不先理解该实用工

[4] 有关包的介绍, 可以参见某页开始的附录.

具的定义. 好吧, 假设你没有用 `find2`. 那么现在不需要先理解 `find2` 了, 读者将不得不去理解 `find-books` 的定义, 该函数相当于把 `find2` 和查找书店的特定任务混在了一起. 理解 `find2` 并不比理解 `find-books` 更难. 另一方面, 在这里我们只用了一次这个新的实用工具. 实用工具意味着重复使用. 在实际的程序里, 它意味着在下列两种情况中做出选择, 理解 `find2`, 或者不得不去理解三到四种特定的搜索例程. 显然前者更容易些.

所以, 阅读一个自底向上程序确实需要一个人能够理解所有作者定义的新操作符. 但它的工作量几乎总是比理解在没有这些操作符的情况下的所有代码要少很多.

如果人们抱怨说使用实用工具使得你的代码难于阅读了, 他们很可能根本没有意识到如果你不使用这些实用工具的话代码看起来将是什么样子. 自底向上程序设计让本来规模很大的程序看起来短小简单. 给人的感觉就是, 这程序并没有做很多事, 所以应该很好懂. 当缺乏经验的读者们更仔细地阅读程序，结果发现事情并没有想象的那么简单, 他们就会灰心丧气.

我们在其他领域发现了相同现象: 一个合理设计的机器可能只有少数几个部件, 然而看起来会更复杂, 因为这些部件被挤在了更小的空间里. 自底向上的程序有种观念上的紧密性. 阅读这种程序可能需要花一些力气, 但如果不是这样写的话，你会需要花更多的精力来读懂它们。

有一种情况下，你应该有意地避免使用实用工具，即: 如果你需要写一个小程序，它将独立于其余部分的代码发布. 一个实用工具通常至少要被使用两到三次才值得引入, 但在小程序里, 如果一个实用工具用得太少的话，可能就没有必要包含它了。

第 5 章

函数作为返回值

上一章展示了“将函数作为参数传递” 是如何为抽象带来更多可能的. 我们对函数能做的事情越多, 我们从这些可能性获得的好处也就越多. 如果能定义一种函数, 让它产生并返回新的函数, 那么我们就能成倍放大那些以函数作为参数的实用工具的功能.

这一章要介绍的实用工具就被用来操作函数. 要是把它们中的多数写成宏, 让这些宏来操纵表达式会显得更自然一些, 至少在 Common Lisp 里是这样的. 在第 15 章会把一层宏加到这些操作符之上. 不管怎样, 就算我们仅仅是通过宏来调用函数, “了解哪些工作能由函数来完成” 这一点也至关重要.

5.1 Common Lisp 的演化

Common Lisp 最初提供了几组互补的函数. `remove-if` 和 `remove-if-not` 就是这样的一组函数. 倘若 `pred` 是一个参数的谓词, 那么

```
(remove-if-not #'pred lst)
```

和下面语句等价

```
(remove-if #'(lambda (x) (not (pred x))) lst)
```

通过更换其中一个语句中传进去的函数参数, 我们可以得到同另一个语句完全相同的效果. 既然是这样, 为什么要同时保留两个语句呢? CLTL2 里包含了一个新的函数, 这个函数就是为了解决上述问题而设计的: `complement` 需要一个谓词 p 作为参数, 它返回一个函数, 这个函数的返回值总是和谓词得到的返回值相反. 当 p 返回真的时候, 它的补 (complement) 就返回假, 反之亦然. 现在我们可以把

```
(remove-if-note #'pred lst)
```

换成与之等价的

```
(remove-if (complement #'pred) lst)
```

有了 `complement`, 就没有多少理由再用那些 `-if-not` 函数了. [1] 事实上, CLTL2 (某页) 说那些函数现在已经淘汰了. 如果它们还在 Common Lisp 里面, 那只是为了兼容性考虑.

[1] `remove-if-not` 可能是个例外, 它比 `remove-if` 更常用一些.

新的 complement 操作符仅是冰山一角: 返回函数的函数. 这在很早就是 Scheme 的习惯用法中重要的一部分了. Scheme 是 Lisp 家族中第一个能把函数作为词法闭包 (lexical closures) 的语言, 而且正是这一点让 “函数作为返回值” 变得有趣起来.

这并不意味着我们不能在动态作用域的 Lisp 里返回函数. 下面的函数能同时在动态作用域或词法作用域下工作:

```
(defun joiner (obj)
  (typecase obj
    (cons   #'append)
    (number #'+)))
```

上面的函数以一个对象作为参数, 按照参数的类型, 返回相应的函数把这种对象累加起来. 通过它, 我们可以定义一个多态 (polymorphic) 的 join 函数, 这个函数可以用于一组数字或者列表.

```
(defun join (&rest args)
  (apply (joiner (car args)) args))
```

然而, “只能返回一个常量函数” 是给我们在动态作用域下的一个限制. 在这个限制下, 我们所无法做到 (或者说无法做得好) 的是在运行期构造函数: 尽管 joiner 可以返回两个函数之一, 但是这两个选择是事先给定的, 无法变更.

在第 17 页, 我们见到了另一个用来返回函数的函数, 它就依赖于词法作用域:

```
(defun make-adder (n)
  #'(lambda (x) (+ x n)))
```

对 make-adder 的调用后, 会得到一个闭包, 闭包的行为视当初传入函数的参数值而定:

```
> (setq add3 (make-adder 3))
#<Interpreted Function "LAMBDA (N)" 58121479>
> (funcall add3 2)
5
```

在词法作用域下, 我们不再仅仅是从一组预先确定的函数中选一个, 而是在运行时创造新的闭包. 但要是动态作用域的话, 这个技术就不再可行了.[2] 如果我们考虑一下 complement 是怎么写的, 可以想到它必定也会返回一个闭包:

```
(defun complement (fn)
  #'(lambda (&rest args) (not (apply fn args))))
```

complement 返回的函数使用了之前 complement 在被调用时传入的参数值 fn. 因此, complement 没有只是从几个常量函数里挑一个返回, 而能够定制一个函数, 让它返回任何函数的反:

[2] 在动态作用域中, 我们或许可以写出类似 make-adder 的代码, 但是很难说那会正常工作. n 的绑定将取决于函数最终被调用时所处的那个环境, 因此我们很难对这个过程能有什么控制.

```
(defvar *!equivs* (make-hash-table))

(defun ! (fn)
  (or (gethash fn *!equivs*) fn))

(defun def! (fn fn!)
  (setf (gethash fn *!equivs*) fn!))
```

图 5.1: 返回破坏性的等价物。

```
> (remove-if (complement #'oddp) '(1 2 3 4 5 6))
(1 3 5)
```

把函数作为参数这个能力是进行抽象时的一个强有力的工具. 而能够编写返回函数的函数, 使得我们可以把这个能力发挥到极致. 接下来的几个小节将会展示几个实用工具的例子, 它们都是返回函数的函数.

5.2 正交性

一门正交的语言能让我们只要运用多种不同的方式对数量很有限的操作符加以组合, 就得到很强的表达能力. 玩具积木是非常正交的, 套装塑料模型就很难说它是正交的. `complement` 主要的功劳就是它让一门语言更为正交了. 在 `complement` 出现之前, Common Lisp 曾有成对的函数, 如 `remove-if` 和 `remove-if-not`、`subst-if` 和 `subst-if-not`, 等等. 自从有了 `complement` 我们可以只用一半数量的函数就完成了全部的功能.

同样, `setf` 宏也增强了 Lisp 的正交性. Lisp 的早期方言常会用成对的函数分别实现读数据和写数据的功能. 举例来说, 对于属性列表 (property-list), 就会有一个函数用来加入属性, 另一个函数则被用来查询它们. 在 Common Lisp 里面, 我们只有后者, 即 `get`. 为了加入一个属性, 我们把 `get` 和 `setf` 一同使用:

```
(setf (get 'ball 'color) 'red)
```

我们或许无法让 Common Lisp 变得更精简, 但是我们可以作些努力达到差不多的效果，即: 使用这门语言的一个较小的子集. 可以定义一些新的操作符, 就像 `complement` 和 `setf` 那样, 让它们帮助我们更接近这个目标吗? 至少另外还有一种方式让函数成对出现. 许多函数都有其破坏性 (destructive) 的版本: 像 `remove-if` 和 `delete-if`、`reverse` 和 `nreverse`、`append` 和 `nconc`. 定义一个操作符, 让它返回一个函数的破坏性版本, 这样就可以不直接使用那些破坏性的函数了.

图 5.1 中的代码实现了破坏性版本的标记. 全局的哈希表 `*!equivs*` 把函数映射到其对应的破坏性版本; `!` 返回函数的破坏性版本; 最后 `def!` 更新和设置

```
(defun memoize (fn)
  (let ((cache (make-hash-table :test #'equal))))
    #'(lambda (&rest args)
        (multiple-value-bind (val win) (gethash args cache)
          (if win
              val
              (setf (gethash args cache)
                    (apply fn args))))))
```

图 5.2: 记忆性的实用工具

它们. 之所以用 ! (惊叹号) 的原因, 是缘于 Scheme 的一个命名习惯, Scheme 中具有副作用的函数名后面都会被加上 !. 现在, 我们一旦定义了

```
(def! #'remove-if #'delete-if)
```

就可以把

```
(delete-if #'oddp lst)
```

取而代之, 换成

```
(funcall (! #'remove-if) #'oddp lst)
```

上面的代码中, Common Lisp 有些许尴尬, 它模糊了这个思路的良好初衷, 要是用 Scheme 就明了多了:

```
((! remove-if) oddp lst)
```

除了更强的正交性, ! 操作符还带来了一系列其它的好处. 它让程序更清晰明了, 因为我们可以一眼就看出 `(! #'foo)` 是与 `foo` 对应的破坏性版本. 另外, 它还让破坏性操作在源代码里总以一个明确可辨的形式出现. 这样的好处在于当我们在找 bug 的时候, 这些地方应该得到更多的重视.

由于函数及其对应的破坏性版本的取舍经常在运行期之前就能确定下来, 因此把 ! 定义成一个宏会是最高效的选择, 或者也可以为它提供一个读取宏 (read macro). [3]

5.3 记住过去

如果某些函数的计算量非常大, 而且我们有时会对它们执行相同的调用, 这时“记住过去” 就有用了: 就是让函数把所有以往调用的返回值都缓存下来, 之后每次被调用时, 都首先在缓存里查找一下, 看看返回值是不是以前算过.

图 5.2 中展示了一个通用化了的记忆性实用工具. 我们传给 `memoize` 一个函数, 它就能返回一个对应的有记忆的版本——即一个闭包, 该闭包含有存储以往调用结果的 hash-table.

[3] 译者注: 本书将在第 17 章介绍读取宏的相关知识.

```
(defun compose (&rest fns)
  (if fns3
      (let ((fn1 (car (last fns)))
            (fns (butlast fns)))
        #'(lambda (&rest args)
            (reduce #'funcall fns
                    :from-end t
                    :initial-value (apply fn1 args))))
      #'identity))
```

图 5.3: 组合函数的操作符

```
> (setq slowid (memoize #'(lambda (x) (sleep 5) x)))
#<Interpreted Function "LAMBDA (FN)" 58140A29>
> (time (funcall slowid 1))
Elapsed Time = 5.15 seconds
1
> (time (funcall slowid 1))
Elapsed Time = 0.00 seconds
1
```

有了具有记忆的函数, 重复的调用就变成 hash-table 的查找操作. 当然, 这也带来了每次调用开始时进行查找导致额外开销, 但是既然我们只会把那些计算开销足够大的那些函数进行记忆化的处理, 那么就可以认为付出这个代价是合算的.

尽管对绝大多数情况来说这个 `memoize` 实现已经够好了, 但是它还是有些局限性. 它认为只有参数列表 `equal` 的调用才是等同的, 这个要求可能会对那些有关键字参数的函数过于严格了. 而且这个函数仅适用于那些返回单值的函数, 因而无法保存多值, 更不用说返回了.

5.4 函数的组合

函数 f 的补被记为 $\sim f$. 第 5.1 节展示了使用闭包可以将 $\sim$ 定义为一个 Lisp 函数的可能性. 另一个常见的函数操作是组合, 它被记作 $\circ$. 如果 f 和 g 是两个函数, 那么 $f \circ g$ 也是函数, 并且 $f \circ g(x) = f(g(x))$. 同样的, 通过使用闭包的方式, 也可以把 $\circ$ 定义为一个 Lisp 函数.

图 5.3 定义了一个名为 `compose` 的函数, 它接受任意数量的函数, 并返回它们的组合. 举个例子

```
(compose #'list #'1+)
```

会返回一个函数, 该函数等价于

```
#'(lambda (x) (list (1+ x)))
```

所有传给 compose 作为参数的函数都必须是只接受一个参数的函数, 不过最后一个函数参数例外.对于它则没有这样的限制, 不管这个函数接受什么样的参数, 都会返回组合后的函数:

```
> (funcall (compose #'1+ #'find-if) #'oddp '(2 3 4))
4
```

由于 not 是一个 Lisp 函数, 所以 complement 是一个 compose 的特例. 它可以这样定义:

```
(defun complement (pred)
  (compose #'not pred))
```

把函数结合在一起使用的方法并不止组合一种. 举例来说, 我们经常会看到下面这样的表达式

```
(mapcar #'(lambda (x)
            (if (slave x)
                (owner x)
                (employer x)))
        people)
```

我们也可以定义操作符，借助它来自动地构造这种函数. 用图 5.4 中定义的 fif, 我们用下面的语句可以达到同样的效果:

```
(mapcar (fif #'slave #'owner #'employer)
        people)
```

图 5.3 中提供了几种构造函数, 它们被用来生成常见的函数类型. 其中第二个构造函数是为下面的情形预备的:

```
(find-if #'(lambda (x)
             (and (signed x) (sealed x) (delivered x)))
         docs)
```

作为第二个参数传给 find-if 的谓词函数定义了一个由三个谓词确定的交集, 这三个谓词将会在这个谓词函数里被调用. fint 的名字取意“function intersection”, 借助它, 我们可以把代码写成这样:

```
(find-if (fint #'signed #'sealed #'delivered) docs)
```

另外, 我们还可以定义一个类似的操作符, 让它返回一组谓词操作的并集. fun 与 fint 类似, 不过前者用的是 or 而非 and.

5.5 在 cdr 上递归

由于递归函数对于 Lisp 程序非常之重要, 因此设计一些实用工具来构造它们是值得的. 本节和下一节将会介绍一些函数, 它们能构造两种最常用的递归函

```
(defun fif (if then &optional else)
  #'(lambda (x)
      (if (funcall if x)
          (funcall then x)
          (if else (funcall else x)))))

(defun fint (fn &rest fns)
  (if (null fns)
      fn
      (let ((chain (apply #'fint fns)))
        #'(lambda (x)
            (and (funcall fn x) (funcall chain x))))))

(defun fun (fn &rest fns)
  (if (null fns)
      fn
      (let ((chain (apply #'fun fns)))
        #'(lambda (x)
            (or (funcall fn x) (funcall chain x))))))
```

图 5.4: 更多的函数构造操作符

数. 在 Common Lisp 里使用这些函数会有些不自然. 一旦我们接触到宏的内容, 就可以了解如何把这个机制包装得更优雅一些. 第 15.2 节和 15.3 节将会介绍那些用来生成递归函数的宏.

如果同一个模式频频出现在程序里, 这就是一个标志, 它意味着这个程序应该用更高层次的抽象改写. 在 Lisp 程序里, 有什么模式比下面这个函数更常见的呢:

```
(defun our-length (lst)
  (if (null lst)
      0
      (1+ (our-length (cdr lst)))))
```

或者比这个函数更眼熟:

```
(defun our-every (fn lst)
  (if (null lst)
      t
      (and (funcall fn (car lst))
           (our-every fn (cdr lst)))))
```

这两个函数在结构上有颇多共同点. 它们两个都递归地在一个列表的 cdr 上依次操作, 每一步对同一个表达式求值, 不过初始条件下除外, 初始条件下两个函数都会返回一个特定的值. 这种模式如此频繁地在 Lisp 程序中出现, 使得有经验的程序员能够不假思索地阅读或写出这样的代码. 事实上, 我们可以迅速从这

```
(defun lrec (rec &optional base)
  (labels ((self (lst)
             (if (null lst)
                 (if (functionp base)
                     (funcall base)
                     base)
               (funcall rec (car lst)
                            #'(lambda ()
                                (self (cdr lst)))))))
    #'self))
```

图 5.5: 用来定义线性列表的递归函数的函数.

个例子中吸取教训, 即为什么把一个模式封装成新的抽象层的的需求迟迟没有出现, 其原因就在于此.

不管怎么样, 它仍然还是一个模式. 我们不应再直接手写这些函数, 而该转而设计一个的新函数, 由它代劳生成函数的工作. 图 5.5 中的函数构造器名叫 `lrec` ("list recurser"), 它可以满足绝大多数的那些在列表上对其 cdr 进行递归操作的需要.

`lrec` 的第一个参数必须是一个接受两个参数的函数, 一个参数是列表的当前 car, 另一个参数是个函数, 通过调用这个函数, 递归得以进行. 有了 `lrec`, 我们可以把 `our-length` 写成:

```
(lrec #'(lambda (x f) (1+ (funcall f))) 0)
```

为了得到列表的长度, 我们不需要关心列表上的元素到底是什么, 也不会中途停止, 因此对象 `x` 的值总是被忽略不用, 而函数 `f` 却总是被调用. 不过我们需要利用起来这些可能性, 重写 `our-every`. 举例来说, 可以用 `oddp`:[4]

```
(lrec #'(lambda (x f) (and (oddp x) (funcall f))) t)
```

在 `lrec` 的定义里使用了 `labels` 来生成一个局部的递归函数, 函数名叫 `self`. 如果要执行递归, 会传两个参数给 `rec` 函数, 两个参数分别是当前列表的 car, 和一个含有递归调用的函数. 以 `our-every` 为例, 是否继续递归由一个 `and` 决定. 如果 `and` 的第一个参数返回的是假, 那么我们就此中止. 换句话说, 传到递归结构里面的不能是个值, 而只能是函数. 这样才能获得返回值(如果我们想要的话).

图 5.6 展示了一些用 `lrec` 定义的 Common Lisp 的内建函数.[5] 使用 `lrec` 来定义函数并不一定会得到最高效的实现. 事实上, 用 `lrec` 和其它本章将要定义的其它递归函数生成器的方法来实现函数的办法, 是与尾递归的思想背道而

[4] 在一个使用较广的 Common Lisp 实现中, `functionp` 在碰到 `t` 和 `nil` 时都会返回真. 在这个实现下, 不管把这两个值中哪一个作为第二个参数传给 `lrec` 都无法使程序正常工作.

[5] 在一些实现里, 如果要显示这些函数, 你必须事先把 `*print-circle*` 设置成 `t`.

```
; copy-list
(lrec #'(lambda (x f) (cons x (funcall f))))

; remove-duplicates
(lrec #'(lambda (x f) (adjoin x (funcall f))))

; find-if, for some function fn
(lrec #'(lambda (x f) (if (fn x) x (funcall f))))

; some, for some function fn
(lrec #'(lambda (x f) (or (fn x) (funcall f))))
```

图 5.6: 用 lrec 生成的函数.

图 5.7: 用列表表示的树。

驰的. 鉴于这个原因, 这些生成器最适合在程序的最初版本里使用, 或者用在那些速度不太关键的地方.

5.6 在子树上递归

在 Lisp 程序里还有另外一种常用的递归形式: 即在子树上进行递归. 当你开始使用嵌套列表, 而且希望递归地访问列表的 car 和 cdr 的时候, 这种递归形式就会出现了.

Lisp 的列表是一种全能型的数据结构. 列表能表示从序列, 集合, 映射, 数组, 以至于树的各类数据结构. 目前有几种不同的方法来用列表表示一颗树. 最常见的一种是把列表看作二叉树, 二叉树的左子树是 car, 右子树则是 cdr. (实际上, 这往往就是列表的内部实现.) 图 5.7 中有三个例子, 分别展示了列表以及它们所表示的树. 其中, 树上的每个内节点都对应着相应列表的点对表示中的一个点, 因而把列表看成下面的形式能更容易理解这种的树型结构:

```
(a b c)     = (a . (b . (c . nil)))
(a b (c d)) = (a . (b . ((c . (d . nil)) . nil)))
```

任意列表都可以被看成一颗二叉树. 同样的, Common Lisp 里类似 copy-list 和 copy-tree 的函数对中, 两个函数的区别也在于此. 前者把列表当作一个序列来处理, 即如果碰到列表中含有子列表的情况, 那么子列表作为序列里的元素, 是不会被复制的:

```
> (setq x     '(a b)
        listx (list x 1))
((A B) 1)
> (eq x (car (copy-list listx)))
T
```

与之相对, copy-tree 会把列表当成树来拷贝, 即把子列表当成子树来看, 所以子列表也一样会被复制:

```
> (eq x (car (copy-tree listx)))
NIL
```

我们可以自己定义一个 copy-tree, 见下面的代码:

```
(defun our-copy-tree (tree)
  (if (atom tree)
      tree
      (cons (our-copy-tree (car tree))
            (if (cdr tree) (our-copy-tree (cdr tree))))))
```

可以看出, 上面的定义是一种常用模式的实例之一. (接下来, 有些函数的写法会显得有些不自然, 这是为了让这个模式变得更明显一些.) 不妨看看下面的例子, 它能够统计出一颗树里叶子节点的数量:

```
(defun count-leaves (tree)
  (if (atom tree)
      1
      (1+ (count-leaves (car tree))
          (or (if (cdr tree) (count-leaves (cdr tree)))
              1))))
```

一颗树上的叶子数会小于当它被表示成列表形式时, 列表中原子的数量.

```
> (count-leaves '((a b (c d)) (e) f))
10
```

而树用 点对 形式来表示时, 你可以注意到树上每个叶子都对应一个原子. 在点对表示中, ((a b (c d)) (e) f) 中有四个 nil 是在列表表示中看不到的 (每对括弧都有一个), 所以 count-leaves 的返回值是 10.

在上一章中, 我们定义了几个用来操作树的实用工具. 比如说, flatten (第 43 页) 接受一颗树, 并返回一个含有树上所有原子的列表. 换句话说, 如果你传给 flatten 一个嵌套列表, 你所得到的返回列表和前面的列表形式相同, 不过除了最外面那对之外, 其它的括弧都不见了:

```
> (flatten '((a b (c d)) (e) f ()))
(A B C D E F)
```

这个函数也可以像下面那样定义 (尽管效率有点低):

```
(defun flatten (tree)
  (if (atom tree)
      (mklist tree)
      (nconc (flatten (car tree))
             (if (cdr tree) (flatten (cdr tree))))))
```

最后, 看一下 rfind-if, 它是 find-if 的递归版本, rfind-if 不仅能用在线性的列表上, 而且对树也一样适用:

```
(defun rfind-if (fn tree)
  (if (atom tree)
      (and (funcall fn tree) tree)
      (or (rfind-if fn (car tree))
          (if (cdr tree) (rfind-if fn (cdr tree))))))
```

为了让 find-if 的应用范围更广, 使之能用于树结构, 我们必须在两者中择一: 让它仅仅搜索叶子节点, 或是搜索整个子树. 我们的 rfind-if 选择了前者, 因而调用方就可以认为作为第一个参数传入的函数只用会在原子上:

```
> (rfind-if (fint #'numberp #'oddp) '(2 (3 4) 5))
3
```

copy-tree, count-leaves, flatten 和 rfind-if, 这四个函数的形式竟然如此的相似. 实际上, 它们都是一个原型函数的衍生品或实例, 这个原型函数被用来进行子树上的递归操作. 和我们对待 cdr 上递归的态度一样, 我们不希望让这个原型默默无闻地埋没在代码当中, 相反, 我们要写一个函数来产生这种原型函数的实例.

要得到原型本身, 让我们先研究一下这些函数, 找出那些部分是不属于模式的. 从根本上来说, our-copy-tree 有两种情形:

1. 基本的情况下, 函数直接把参数作为返回值返回
2. 在递归的时候, 函数对左子树 (car) 的递归结果和右子树 (cdr) 的递归结果使用 cons

因此, 我们肯定可以通过调用一个有两个参数的构造函数, 来表示 our-copy-tree:

```
(ttrav #'cons #'identity)
```

图 5.8 中展示了 ttrav ("tree traverser") 的一个实现. 在递归情况下, 我们不是传入一个值, 而是传入两个, 一个对应左子树, 一个对应右子树. 如果 base 参数是个函数, 那么将把当前叶子节点作为参数传给它. 在对线性列表进行递归操作时, 基本情况的返回值总是 nil, 不过在树结构的递归操作中, 基本情况的值有可能是个更有意思的值, 而且我们也许会需要用到它.

在 ttrav 的帮助下, 我们可以重新定义除 rfind-if 之外前面提到的所有函数. (这些函数在图 5.9 中可以找到.) 要定义 rfind-if, 我们需要一个更为通用的树结构递归操作函数的生成器, 这种函数生成器让我们能控制递归调用发生的时机, 以及是否继续递归. 我们把一个函数作为传给 ttrav 的第一个参数, 这个函数的参数将是递归调用的返回值. 对于通常的情形, 我们会取而代之用一个函数, 该函数接受两个闭包, 闭包分别自己表示调用. 这样, 我们就可以编写那些能自主控制递归过程的递归函数了.

用 ttrav 实现的函数通常会遍历整颗树. 这样做对于像 count-leaves 或者 flatten 这样的函数是没有问题的, 它们不管如何都会遍历全树. 然而, 我

```
(defun ttrav (rec &optional (base #'identity))
  (labels ((self (tree)
                 (if (atom tree)
                     (if (functionp base)
                         (funcall base tree)
                         base)
                     (funcall rec (self (car tree))
                                  (if (cdr tree)
                                      (self (cdr tree)))))))
    #'self))
```

图 5.8: 为在树上进行递归操作而设计的函数.

```
; our-copy-tree
(ttrav #'cons)

; count-leaves
(ttrav #'(lanbda (l r) (+ l (or r 1))) 1)

; flatten
(ttrav #'nconc #'mklist)
```

图 5.9: 用 `ttrav` 表示的函数.

```
(defun trec (rec &optional (base #'identiy))
  (labels
      ((self (tree)
             (if (atom tree)
                 (if (functionp base)
                     (funcall base tree)
                     base)
                 (funcall rec tree
                              #'(lambda ()
                                  (self (car tree)))
                              #'(lambda ()
                                  (if (cdr tree)
                                      (self (cdr tree))))))))
    #'self))
```

图 5.10: 为在树上进行递归操作而设计的函数.

们需要 rfind-if 一发现它所要找的元素就停止遍历. 这个函数必须交给更通用的 trec 来生成, 见图 5.10. trec 的第二个参数应当是一个具有三个参数的函数, 三个参数分别是: 当前的对象, 以及两个递归调用. 后两个参数将是用来表示对左子树和右子树进行递归的两个闭包. 使用 trec 我们可以这样定义 flatten:

```
(trec #'(lambda (o l r) (nconc (funcall l) (funcall r)))
```

现在, 我们同样可以把 rfind-if 写成这样 (下面的例子用了 oddp):

```
(trec #'(lambda (o l r) (or (funcall l) (funcall r)))
      #'(lambda (tree) (and (oddp tree) tree)))
```

5.7 何时构造一个函数

很不幸地，如果用构造函数，而不是用 sharp-quoted 的 lambda 表达式来表示函数会在运行时让程序做一些不必要的工作。虽然 sharp-quoted 的 lambda 表达式是一个常量, 但是对构造函数的调用将会在运行时估值。如果你真的必须在运行时执行这个调用，可能使用构造函数并非上策。不过，至少有的时候我们可以在事前就调用这个构造函数。通过使用 #.，即 sharp-dot 读取宏，我们可以让函数在读取时 (read-time) 就被构造出来。假设 compose 和它的参数在下面的表达式被读取时已经被定义了，那么我们可以这样写，举例如下：

```
(find-if #.(compose #'oddp #'truncate) lst)
```

这样做的话，对 compose 的调用就会被 reader 估值，估值得到的函数则被作为常量安插在我们的代码之中。由于 oddp 和 truncate 两者都是内置函数，因此在读取时对 compose 进行估值可以被认为是安全可行的，当然，前提是那个时候 compose 自己已经加载了。

一般而言，由宏来完成函数组合或者合并的工作会更事半功倍，同时程序的效率也会提高。这一点对函数拥有具有单独名字空间的 Common Lisp 来说尤其如此。在介绍了宏的相关知识后，我们会在第 15 章故地重游，再次回到这一章中曾走到过的大多数山山水水，所不同的是，到那时候你会骑上更纯种的宝马，配上更奢华的鞍具。

第 6 章

函数作为表达方式

通常说来，数据结构被用来描述事物。数组可以被用来描述一次坐标变换，树结构可以用来表示命令的层次结构，图则可以用来表示一张铁路网。在 Lisp 里，我们常会使用闭包来作为一种表现形式。在闭包里，变量绑定可以保存信息，也能扮演在复杂数据结构中指针 所担当的角色。如果让一组闭包之间共享绑定，或者让它们之间能相互引用，那么我们就可以创建混合型的对象类型，它同时继承了数据结构和程序逻辑两者的优点。

其实在表象之下，共享绑定就是指针。闭包只是让我们能在更高的抽象层面上操作指针。通过使用闭包来表示我们以往用静态数据结构来表示的对象，我们就往往可能得到更为优雅，效率更好的程序。

6.1 网络

闭包有三个有用的特性：它们是动态的，它们拥有局部状态，而且我们可以创建闭包的多个实例。那么带有局部状态的动态对象的多个拷贝能在什么地方一展身手呢？答案是：和网络有关的程序。许多情形下，我们可以把网络中的节点表示成闭包。闭包在拥有其局部状态的同时，它还能引用其它闭包。因而，一个表示网络中节点的闭包是能够知道作为它发送数据目的地的其他几个节点 (闭包) 的。换句话说，我们有能力把网络结构直接翻译成代码。

在本节和下一节里，我们会分别了解两种遍历网络的方法。首先我们会顺着传统的办法，即把节点定义成结构体，并用与之分离的代码来遍历网络。然后在下一节，我们将会用一个单一的抽象模型来构造通用功能的程序。

我们将选一个最简单的应用作为例子：一个能运行 twenty questions [1] 的程序。我们的网络将会是一棵二叉树。每一个非叶子节点都会含有一个是非题，并且根据对这个问题的回答，遍历过程会在左子树或者右子树上继续往下进行。叶子节点将会含有所有的返回值。当遍历过程遇到叶子节点时，叶子节点的值会被作为遍历过程的返回值返回。如图 6.1 所示，是程序运行一轮 twenty questions 的样子。

用习惯的办法着手，可能就是先定义某种数据结构来表示节点。节点应

[1] 译者注：twenty questions 曾是一个国外很流行的电视智力节目，同时也是人工智能的一个重要题材。

```
> (run-node 'people)
Is the person a man?
>> yes
Is he living?
>> no
Was he American?
>> yes
Is he on a coin?
>> yes
Is the coin a penny?
>> yes
LINCOLN
```

图 6.1: 一轮 twenty questions 游戏

```
(defstruct node  contents yes no)

(defvar *nodes* (make-hash-table))

(defun defnode (name conts &optional yes no)
  (setf (gethash name *nodes*)
        (make-node :contents conts
                   :yes      yes
                   :no       no)))
```

图 6.2: 节点的表示方法及其定义。

```
(defnode 'people "Is the person a man?" 'male 'female)

(defnode 'male "Is he living?" 'liveman 'deadman)

(defnode 'deadman "Was he American?" 'us 'them)

(defnode 'us "Is he on a coin?" 'coin 'cidence)

(defnode 'coin "Is the coin a penny?" 'penny 'coins)

(defnode 'penny 'lincoln)
```

图 6.3: 作为样例的网络。

```
(defun run-node (name)
  (let ((n (gethash name *nodes*)))
    (cond ((node-yes n)
           (format t "~A~%>> " (node-contents n))
           (case (read)
             (yes (run-node (node-yes n)))
             (t   (run-node (node-no n)))))
          (t (node-contents n)))))
```

图 6.4: 用来遍历网络的函数。

该包含这几样信息：它是否是叶子节点；如果是的话，那返回值是什么，倘若不是，要问什么问题；还有与答案对应的下个节点是什么。图 6.2 里定义了一个信息足够详尽的数据结构。它的设计目标是让数据所占的空间最小化。`contents` 字段中要么是问题要么是返回值。如果该节点不是叶子节点，那么 `yes` 和 `no` 字段会告诉我们与问题的答案对应的去向；如果节点是个叶子节点，我们自然会知道这一点，因为这些字段会是空的。全局的 `*nodes*` 是一个哈希表，在表中，我们会用节点的名字来索引节点。最后，`defnode` 会新建一个节点 两种节点都可以，并把它保存到 `*nodes*` 中。用这些原材料，我们就可以定义树的第一个节点了：

```
(defnode 'people "Is the person a man?"
         'male 'female)
```

图 6.3 中所示的网络正好足够我们运行 6.1 中所示的一轮游戏。

现在，我们要做的就是写一个能遍历这个网络的函数了，这个函数应该打印出问题，顺着答案所指示的路径走下去。这个函数，即 `run-mode` 如图 6.4 所示。给定一个名字，我们就根据名字找到对应的节点。如果该节点不是叶子节点，就把 `contents` 作为问题打印出来，按照答案不同，我们继续顺着两条可能的途径之一继续遍历。如果该节点是一个叶子节点，`run-node` 会径直返

```
(defun *nodes* (make-hash-table))

(defun defnode (name conts &optional yes no)
  (setf (gethash name *name*)
        (if yes
            #'(lambda ()
                (format t "~A~%>> " conts)
                (case (read)
                  (yes (funcall (gethash yes *nodes*)))
                  (t   (funcall (gethash no  *nodes*)))))
            #'(lambda () conts))))
```

图 6.5: 编译成闭包形式的网络。

回 contents。使用图 6.3 中定义的网络，这个函数就能生成图 6.1 中的输出信息。

6.2 编译后的网络

在上一节，我们编写了一个使用了网络的程序，也许使用其它任何一种语言都能写出来这样的程序。的确，这个程序太简单了，看上去似乎很难把它写成另外的模样。但是事实上，我们可以把程序打理得更简洁一些。

图 6.5 就是明证。这就是让我们的网络运行起来所需要的所有代码。在这里，不再把节点定义成一个结构，也没有用一个单独的函数来遍历这些节点，而是把节点表示成闭包。原来保存在数据结构里的数据现在被放在了闭包里的变量绑定之中。没有必要运行 run-node 了，它已经隐含在了节点自身里面。要启动遍历过程，我们仅需 funcall 一下我们起始的那个节点就可以了：

```
(funcall (gethash 'people *nodes*))
Is the person a man
>>
```

从这一点开始，接下来的人机对话就和上个版本的实现一样了。

借助把节点都表示成闭包的方式，我们得以将 twenty questions 网络完全转化成代码 (而非数据)。正如我们所看到的，程序代码必须在运行时按照名字来查找节点函数。然而，如果我们确信网络在运行的时候不会被重新定义，那就可以更进一步：让节点函数直接调用它们的下一站目标函数，而不必再动用哈希表了。

如图 6.6 所示，是新版的程序代码。现在 *node* 从哈希表改成了一个列表。像以前一样，所有的节点还是用 defnode 来定义的，不过定义之时还没有闭包被生成。在所有的节点都被定义之后，我们就调用 compile-net 来一次性地编译整个网络。这个函数递归地进行处理，一直往下，直至树的叶子节

```
(defun *nodes* nil)

(defun defnode (&rest args)
  (push args *nodes*)
  args)

(defun compile-net (root)
  (let ((node (assoc root *nodes*)))
    (if (null node)
        nil
      (let ((conts (second node))
            (yes (third node))
            (no (fourth node)))
        (if yes
            (let ((yes-fn (compile-net yes))
                  (no-fn  (compile-net no)))
              #'(lambda ()
                  (format t "~A~%>> " conts)
                  (funcall (if (eq (read) 'yes)
                               yes-fn
                               no-fn))))
            #'(lambda () conts))))))
```

图 6.6: 使用静态引用的编译过程。

点，在递归过程层层返回时，每一步都返回了两个目标函数对应的节点 (或称函数)，而不仅仅是给出它们的名字。[2] 当最外面的 compile-net 调用返回时，它给出的函数将表示一个我们所需的那部分网络。

```
> (setq n (compile-net 'people))
#<Compiled-Function BF3C06>
> (funcall n)
Is the person a man?
>>
```

注意到，compile-net 在两个层面的含义上进行了编译。按照通常编译的含义，它把网络的抽象表示翻译成了代码。更进一层，如果 compile-net 自身被编译的话，那它就会返回编译后的函数。(见第 22 页)

在编译完成网络后，我们就不再需要由 defnode 建成的列表了。切断列表与程序的联系 (例如将 *nodes* 设为 nil)，然后垃圾收集器就会把它回收了。

6.3 展望

有许多涉及网络的程序都能通过把节点编译成闭包的形式来实现。闭包作为

[2]这个版本的程序假定程序中的网络是一种树结构，这个假设对这个应用来说肯定是成立的。

数据对象，和各种数据结构一样能被用来表现事物的属性。这样做需要一些和习惯相左的思考方式，但是作为回报的是更为迅速，更为优雅的程序。

宏在相当程度上将有助于我们把闭包作为一种表达方式来使用。“用闭包来表示” 是 “编译” 的另外一种说法。而且由于宏是在编译时完成它们的工作的，因而它们理所应当地就是这种技术的最佳载体。在介绍了宏技术之后，第 23 章和第 24 章里会呈上更大规模的程序，这些程序将会使用这里曾用过的方法写成。

第 7 章

宏

Lisp 的宏特性让你能通过一些变换来实现操作符。宏定义本质上是一个能生成 Lisp 代码的函数——一个能写程序的程序。这一小小开端引发了巨大的可能性，同时也伴随了难以预料的风险。第 7–10 章将带你走入宏的世界。本章会解释宏如何工作，介绍编写和测试它们的技术，然后分析一些宏风格中存在的问题。

7.1 宏是如何工作的

由于我们可以调用宏并得到它的返回值，因此宏往往被人们跟函数联系在一起。宏定义有时跟函数定义相似，而且不严谨地说，被人们称为 “内置函数” 的 `do` 其实就是一个宏。但如果把两者过于混为一谈，就会造成很多困惑和混淆。宏和常规函数的工作方式截然不同，并且只有知道宏为何不同，以及怎样不同才是用好它们的关键。一个函数只产生结果，而宏却产生*表达式*——当它被求值时，才会产生结果。

要入门，最好的办法就是直接看个例子。假设我们想要写一个宏 `nil!`，它能把一个参数[1] 设置为 `nil`。我们希望 `(nil! x)` 和 `(setq x nil)` 具有相同的效果。我们通过将 `nil!` 定义成一个宏来把前一种形式 (form) 的实例变成后一种形式的实例。

```
> (defmacro nil! (var)
    (list 'setq var nil))
NIL!
```

用英语转述的话，这个定义相当于告诉 Lisp: “无论何时你看到一个形如 `(nil!` *var*`)` 的表达式，在对它求值之前先把它转化成 `(setq` *var* `nil)` 的形式。”

宏产生的表达式将在宏被调用的位置被求值。一个宏调用是个列表，其第一个元素是一个宏的名称。当我们把宏调用 `(nil! x)` 输入到 toplevel 的时候发生了什么？Lisp 会注意到 `nil!` 是一个宏的名称，并且

[1] 译者注：本章大量出现两个单词，“argument” 和 “parameter”，它们单独出现时一般都译成 “参数”，但其实两者是有区别的，前者相当于函数调用中的实参，后者相当于形参。本章目前的翻译是，在两者同时出现时，为避免混淆，“argument” 译成 “变元”，而 “parameter” 译成 “参数”。

1. 按照上述定义指定的方法构造表达式，然后

2. 在宏被调用的地方求值该表达式。

构造新表达式的那一步被称为*宏展开 (macroexpansion)*。Lisp 查找 `nil!` 的定义，其定义展示了如何为宏调用构建一个替代品。`nil!` 的定义像函数那样应用到宏调用中作为参数给出的*表达式*上。它返回一个含有三个元素的列表，这三个元素分别是: `setq`、作为参数传递给宏的那个表达式，以及 `nil`。在本例中，`nil!` 的参数是 `x`，宏展开式是 `(setq x nil)`。

宏展开完成之后是第二步：*求值 (evaluation)*。Lisp 求值宏展开式 `(setq x nil)` 时就好像是你原本就写在那儿的一样。求值并不总是立即发生在展开之后，不过在 toplevel 下确是如此。一个发生在函数定义里的宏调用将在函数编译时被展开, 但展开式——或者说它产生的对象代码——要等到函数被调用时才会被求值。

你遇到的和宏有关的困难中，有很多或许都可以通过分清宏展开和求值来避免。当编写宏的时候，要清楚哪些操作是在宏展开期进行，哪些操作是在求值期进行的，通常，这两步操作的分别是两类截然不同的对象。宏展开步骤处理的是表达式，而求值步骤处理的则是它们的值。

有时宏展开会比 `nil!` 里的情况更复杂。`nil!` 的展开式只是对一个内置特殊形式 (special form) 的调用，但有时一个宏的展开式将会是另一个宏调用，就好像是一层套一层的俄罗斯套娃。在这种情况下，宏展开就会继续抽丝剥茧直到获得一个不含有宏的表达式。这一步骤中可以包含任意多次的展开操作，直到最终停下来。

许多语言提供了某种形式的宏，但 Lisp 宏却非凡的强大。当一个 Lisp 文件被编译时，一个解析器读取源代码然后将其输出发给编译器。这里有一个天才手笔：解析器的输出由 *Lisp 对象的列表*组成。通过使用宏，我们可以操作这种处于解析器和编译器之间的中间状态的程序。如果必要的话，这些操作可以是非常广泛的。一个用于生成展开式的宏拥有 Lisp 的全部威力可以任其驱驰。事实上，宏是货真价实的 Lisp 函数——那种能返回表达式的函数。虽然 `nil!` 的定义中只包含一个简单的对 `list` 的调用，但其他宏里可能会调用整个的子程序来生成其展开式。

能改变编译器所看到的东西差不多相当于能够对代码进行重写。所以我们就可以为语言增加任何构造然后通过变换把它定义到已存在的构造上。

7.2 反引用 (backquote)

反引用 (backquote) 是引用 (quote) 的特别版本，它可用来创建 Lisp 表达式模板。反引用最普遍的用途之一是被用在在宏的定义里。

反引用字符，‘，得名的原因是：它和通常的引号“’”相似，只不过方向相反。当反引用独自作为一个表达式前缀的时候，它的行为和引号是相同的：

‘(a b c) 等价于 ’(a b c).

反引用只有当它和逗号“,”，以及逗号–at 符号“,@”一同出现时才变得有用。如果说反引用创建了一个模板，那么逗号就在反引用中创建了一个槽位 (slot)。一个反引用列表等价于将其元素引用起来调用一次 `list`。也就是，

‘(a b c) 等价于 (list ’a ’b ’c).

在反引用的作用域里，一个逗号告诉 Lisp: “把引用关掉。” 当逗号出现在一个列表元素之前时，它的效果就相当于取消引用，把该元素原样放在那里。所以

‘(a ,b c ,d) 等价于 (list ’a b ’c d).

插入到结果列表里的不再是符号 `b`，取而代之的是它的值。无论逗号出现在嵌套列表里的层次有多深，它都仍然有效，

```
> (setq a 1 b 2 c 3)
3
> ‘(a ,b c)
(A 2 C)
> ‘(a (,b c))
(A (2 C))
```

而且它们也可以出现在引用的列表里，或者引用的子列表里：

```
> ‘(a b ,c (’,(+ a b c)) (+ a b) ’c ’((,a ’b)))
(A B 3 (’6) (+ A B) ’C ’((1 2)))
```

一个逗号能抵消一个反引用的效果，所以逗号在数量上必须和反引用匹配。如果某个操作符出现在逗号的外层，或者出现在包含逗号的那个表达式的外层，那么我们说这个逗号被这个操作符所包围。例如在 ‘(,a ,(b ‘,c)) 中，最后一个逗号就被前一个逗号和两个反引号所包围。通行的规则是：一个被 n 个逗号包围的逗号必须被至少 $n+1$ 个反引号所包围。很明显，由此可以知道：逗号不能出现在一个反引用的表达式的外面。只要遵守上述规则，反引用和逗号可以被嵌套使用。下面的任何一个表达式如果输入到 toplevel 下都将生成一个错误：

```
,x    ‘(a ,,b c)    ‘(a ,(b ,c) d)    ‘(,,‘a)
```

嵌套的反引用可能只有在宏定义的宏里才需要。这两个主题将在第 16 章里讨论。

反引用通常被用来创建列表。[2]任何用反引用生成的列表也都可以用 `list` 和正常的引用来实现。使用反引用的好处只是在于它使表达式更加易于阅

[2]反引用也可以用于创建向量 (vector)，不过这个用法很少在宏定义里出现。

使用反引用：

```
(defmacro nif (expr pos zero neg)
  `(case (truncate (signum ,expr))
     (1 ,pos)
     (0 ,zero)
     (-1 ,neg)))
```

不使用反引用：

```
(defmacro nif (expr pos zero neg)
  (list 'case
     (list 'truncate (list 'signum expr))
     (list 1 pos)
     (list 0 zero)
     (list -1 neg)))
```

图 7.1: 一个使用和不使用反引用的宏定义。

读，因为一个反引用的表达式和它将要生成的表达式很相似。在前一章里我们把 `nil!` 定义成：

```
(defmacro nil! (var)
  (list 'setq var nil))
```

借助反引用，这个同样的宏可以定义成：

```
(defmacro nil! (var)
  `(setq ,var nil))
```

在本例中，是否使用反引用的差别还不算太大。不过，随着宏定义长度的增加，使用反引用的重要性也会相应提高。图 7.1 包含了两个 `nif` 可能的定义，这个宏实现了三路数值条件选择。[3]

首先，第一个参数会被求值成一个数字。然后会根据这个数字的正负、是否为零，来决定第二、第三和第四个参数中哪一个将被求值：

```
> (mapcar #'(lambda (x)
              (nif x 'p 'z 'n))
          '(0 2.5 -8))
(Z P N)
```

图 7.1 中的两个定义分别定义了同一个宏，但是前者使用的是反引用，而后者则通过显式调用 `list` 来构造它的展开式。以 `(nif x 'p 'z 'n)` 为例，从第一个定义中容易看出这个表达式可以展开成

[3]这个宏的定义稍微有些不自然，这是为了避免使用 gensyms。第 138 页上给出了一个更好的定义。

```
(case (truncate (signum x))
  (1 'p)
  (0 'z)
  (-1 'n))
```

因为这个宏定义体看起来就像它生成的宏展开式。要想理解不使用反引用的第二个版本，你将不得不在头脑中重演一遍展开式的构造过程。

逗号–at 符号，`,@`，是逗号的变种，它的行为和逗号相似，但有一点不同：它不只是像逗号那样将表达式的值插入到其所在的位置，逗号–at 拼接到当前位置。拼接这个操作可以被理解为：在插入的同时，去掉被插入对象最外层的括号：

```
> (setq b '(1 2 3))
(1 2 3)
> `(a ,b c)
(A (1 2 3) C)
> `(a ,@b c)
(A 1 2 3 C)
```

逗号导致列表 `(1 2 3)` 被插入到 `b` 所在的位置，而逗号–at 使得列表中的元素被插入到那里。对于逗号–at 的使用，还有一些额外的限制：

1. 为了确保其参数可以被拼接，逗号–at 必须发生在一个序列 (sequence) [4]之中。诸如 `` `,@b `` 一类的说法是错误的，因为没有什么地方可供 `b` 的值进行拼接。

2. 要进行拼接的对象必须是一个列表，除非它出现在列表的最后。[5] 表达式 `` `(a ,@1) `` 将被求值成 `(a . 1)`，但如果尝试将一个原子[6] (atom) 拼接到一个列表的中间位置，例如 `` `(a ,@1 b) ``，将导致一个错误。

逗号–at 一般用在接受不确定数量参数的宏里，以及将这些参数传给同样接受不确定数量参数的函数和宏里。这一情况通常广泛用于实现隐式的块 (block)。Common Lisp 提供几种将代码分组到块的操作符，包括 `block`、`tagbody`, 以及 `progn`。这些操作符很少直接出现在源代码里；它们经常隐式使用——也就是隐藏在宏的背后。

隐式块出现在任何一个带有表达式体的内置宏里。例如 `let` 和 `cond` 里都有隐式的 `progn` 存在。做这种事情的内建宏里，最简单的一个可能要算 `when` 了：

[4]译者注：序列 (sequence) 是 Common Lisp 标准定义的数据类型，其两个子类型分别是列表 (list) 和向量 (vector)。一些内置函数可以作用在序列上，无论其究竟是列表还是向量。

[5]译者注：这种情况下，必须严格把逗号–at 的出现场合限定在列表的最后，向量也是不允许的。

[6]译者注：原子 (atom) 也是 Common Lisp 标准定义的数据类型，所有不是列表的 Lisp 对象都是原子，包括向量 (vector) 在内。

```
(when (eligible obj)
  (do-this)
  (do-that)
  obj)
```

如果 (eligible obj) 返回真，那么其余的表达式将会被求值，并且整个 when 表达式会返回其中最后一个表达式的值。作为一个使用逗号–at的示例，下面是 when 的一种可能实现：

```
(defmacro our-when (test &body body)
  `(if ,test
       (progn
         ,@body)))
```

这一定义使用了一个 &body 参数 (它和 &rest 功能相同，只有美观输出的时候不太一样) 来接受可变数量的参数，然后一个逗号–at 将它们拼接到一个 progn 表达式里。在上述调用的宏展开式里，宏调用体里面的三个表达式将出现在单个 progn 中：

```
(if (eligible obj)
    (progn (do-this)
           (do-that)
           obj))
```

多数需要迭代处理其参数的宏都采用类似方式拼接它们。

逗号–at 的效果也可以不用反引用来实现。例如，表达式 `(a ,@b c) 等价 (equal) 于 (cons 'a (append b (list 'c)))。逗号–at 的存在只是为了使这种表达式生成的表达式的可读性更好。

宏定义 (通常) 生成列表。尽管宏展开式可以用函数 list 来生成，但反引用的列表模板可以令这一任务更为简单。一个用 defmacro 和反引用定义的宏，在形式上和一个用 defun 定义的函数非常相似。只是不要被这种相似性所迷惑，反引用使宏定义既容易书写也容易阅读。

由于反引用经常出现在宏定义里，以致于人们有时误以为反引用是 defmacro 的一部分。关于反引用的最后一件要记住的事情是它有自己存在的意义，这跟它在宏定义中的角色无关。你可以在任何需要构造序列的场合使用反引用：

```
(defun greet (name)
  `(hello ,name))
```

7.3 定义简单的宏

在编程领域，最快的学习方式通常是尽快地开始实践。完全理论上的理解可以稍后再说。因此本章介绍一种可以立即开始编写宏的方法。虽然该方法的适用范围很窄，但在这个范围内却可以高度机械化地实现。(如果你以前写过宏，可以跳过本章。)

调用：(memq x choices)
展开：(member x choices :test #'eq)

图 7.2: 用于写 memq 的图解

作为一个例子，我们考虑一下如何写出一个 Common Lisp 内置函数 member 的变种。member 缺省使用 eql 来测试等价性。如果你想要用 eq 来测试等价性，你就不得不显式地写成这样：

```
(member x choices :test #'eq)
```

如果经常这样做，那我们可能会想要写一个 member 的变型，让它总是使用 eq。有些早期的 Lisp 方言有这样一个函数，叫做 memq：

```
(memq x choices)
```

通常应该将 memq 定义为内联 (inline) 函数，但出于示例的目的我们会让它以宏的面目出现。

该方法：从你想要定义的这个宏的一次典型调用开始。把它写在一张纸上，然后下面写上它应该展开成的表达式。图 7.2 给出了两个这样的表达式。通过宏调用，构造出你这个宏的参数列表，同时给每个参数命名。这个例子中有两个实参，所以我们将会有两个形参，把它们叫做 obj 和 lst：

```
(defmacro memq (obj lst)
```

现在回到之前写下的两个表达式。对于宏调用中的每个参数，画一条线把它和它在展开式里出现的位置连起来。在图 7.2 里有两条并行的线。为了写出宏的实体，把你的注意力转移到展开式。让主体以一个反引用开头。现在，开始逐个表达式地阅读展开式。每当你发现一个括号，如果它不是宏调用中实参的一部分，就把它放在宏定义里。所以紧接着反引用会有一个左括号。对于展开式里的每个表达式

1. 如果没有线将它和宏调用连在一起，那么就把表达式本身写下来。
2. 如果存在一条跟宏调用中某个参数的连接，把出现在宏参数列表的对应位置的那个符号写下来，前置一个逗号。

第一个元素 member 上没有连接，所以我们照原样使用 member：

```
(defmacro memq (obj lst)
  `(member
```

尽管如此，x 上有一条线指向源表达式中的第一个变元，所以我们在宏的主体中使用第一个参数，带一个逗号：

```
(defmacro memq (obj lst)
  `(member ,obj
```

```
(while hungry
  (stare-intently)
  (meow)
  (rub-against-legs))

(do ()
    ((not hungry))
  (stare-intently)
  (meow)
  (rub-against-legs))
```

图 7.3: 用于写 while 的图解

以这种方式继续进行，最后完成的宏定义是：

```
(defmacro memq (obj lst)
  `(member ,obj ,lst :test #'eq))
```

到目前为止，我们还只能写出带有固定数量参数的宏。现在假设我们打算写一个 while 宏，它接受一个条件表达式和一个代码体，然后循环执行代码直到条件表达式返回真。图 7.3 含有一个描述猫的行为的 while 循环示例。

要想写出这样一个宏，我们需要对一下我们的技术稍作修改。和之前一样，从写一个简单的宏调用开始。接下来，构造宏的参数列表，但是在你想要接受任意多个参数地方，以一个 &rest 或 &body 参数结束：

```
(defmacro while (test &body body)
```

现在在宏调用的下面写出目标展开式，并且和之前一样画线连接宏调用的变元和它们在展开式中的位置。尽管如此，当你遇到一个变元的序列，它们相当于一个单独的 &rest 或 &body 参数时，把它们看作一个组，对整个序列单独画一条线。图 7.3 给出了结果的图解。

为了写出宏定义的主体，在表达式上按之前的过程处理。在前面给出的两条规则之外，我们还要加上一条：

3. 如果存在一条从一系列展开式里的表达式到宏调用里的一系列变元的连线，那么把对应的 &rest 或 &body 参数写下来，前置一个逗号–at。

于是宏定义的结果将是：

```
(defmacro while (test &body body)
  `(do ()
       ((not ,test))
     ,@body))
```

要想构造一个带有表达式体的宏，就必须有参数充当打包装箱的角色。这里宏调用中的多个参数被串起来放到 body 里，然后当 body 被拼接进展开式时再被拆散开。

```
> (defmacro while (test &body body)
    `(do ()
         ((not ,test))
       ,@body))
WHILE

> (pprint (macroexpand '(while (able) (laugh))))

(BLOCK NIL
  (LET NIL
    (TAGBODY
      #:G61
      (IF (NOT (ABLE)) (RETURN NIL))
      (LAUGH)
      (GO #:G61))))
T
> (pprint (macroexpand-1 '(while (able) (laugh))))

(DO NIL
    ((NOT (ABLE)))
  (LAUGH))
T
```

图 7.4: 一个宏和它的两级展开。

用本章所述的这个方法，我们能写出最简单的宏——那类仅在参数位置上做手脚的。宏可以比这做得多得多。第 7.7 章将展示一个无法用简单的反引用列表来表达的例子，并且为了生成展开式，宏成为了真正意义上的程序。

7.4 测试宏展开

写好了一个宏，我们如何测试它呢？像 memq 这样的宏，它结构较为简单，只消看看它的代码就能弄清其行为方式。而当编写结构更复杂的宏时，我们必须有办法检查它们展开之后是否正确。

图 7.4 给出了一个宏定义，和用来查看其展开式的两个办法。内置函数 macroexpand 接受一个表达式，并返回其宏展开式。把一个宏调用传给 macroexpand 能看到宏调用在求值之前最终展开的样子，但是一个完成的展开式在你测试一个宏的时候并不总是你想要的。当一个测试中的宏依赖于其他宏的时候，它们也会一起被展开，所以一个完整的宏展开有时是难于阅读的。

从图 7.4 中给出的第一个表达式中，很难说 while 是否如愿展开了，因为内置的宏 do 也被展开了，而且它里面的 prog 宏也被展开了。我们要的是一种能看到仅仅一次宏展开操作结果的方法。这也就是内置函数 macroexpand-1 的目的，在第二个例子里给出了；macroexpand-1 只做一次宏展开后就结束，即

```
(defmacro mac (expr)
  `(pprint (macroexpand-1 ',expr)))
```

图 7.5: 一个用于测试宏展开的宏。

使得到展开式仍是一个宏调用。

如果每当我们想查看一个宏调用的展开式时，都得输入像下面那样的表达式，这会很让人头痛：

```
(pprint (macroexpand-1 '(or x y)))
```

图 7.5 定义了一个新的宏，它让我们有一个简单的替代方法：

```
(mac (or x y))
```

调试函数的典型方法是调用它们，同样道理对于宏来说就是展开它们。不过由于宏调用涉及到两层计算，所以它也就有两个地方可能会出问题。如果一个宏行为不正常，大多数时候你只通过看它的展开式子就能找出有错的地方。不过也有一些时候，展开式看起来是对的，所以你想对它进行求值以便找出问题所在。如果展开式里含有自由变量，你可能需要先设置一些变量。在某些系统里，你可以复制展开式并且把它粘贴到 toplevel 环境里，或者选择它然后在菜单里选 `eval`。在最坏的情况下你也可以将 `macroexpand-1` 返回的列表设置在一个变量里，然后对它调用 `eval`：

```
> (setq exp (macroexpand-1 '(memq 'a '(a b c))))
(MEMBER (QUOTE A) (QUOTE (A B C)) :TEST (FUNCTION EQ))
> (eval exp)
(A B C)
```

最后，宏展开不只是调试的辅助手段，它也是一种学习如何编写宏的方式。Common Lisp 带有超过一百个内置宏，其中一些还颇为复杂。通过查看这些宏的展开过程你经常能看到它们是怎样写出来的。

7.5 参数列表的解构

解构 (destructuring) 是用在处理函数调用中的一种赋值操作[7]的通用形式。如果你定义一个带有多个变元的函数

```
(defun foo (x y z)
  (+ x y z))
```

然后当该函数被调用时

[7] 解构通常用在创建变量绑定，而非 `do` 那样的操作符里。尽管如此，概念上来讲解构也是一种赋值的方式，如果你把列表解构到已有的变量而非新变量上是完全可行的。就是说，没有什么可以阻止你用解构的方法来做类似 `setq` 这样的事情。

```
(foo 1 2 3)
```

函数的参数按位置被赋值成函数调用中的变元：x 到 1，y 到 2，以及 z 到 3。*解构 (destructuring)* 描述了按位赋值对于任意列表结构的情形，跟扁平列表 (x y z) 的情形相似。

Common Lisp 的 destructuring-bind 宏 (CLTL2 新增) 接受一个模式，一个求值到列表的变元，以及一个表达式体，然后在求值表达式时将模式中的参数绑定到列表的对应元素上：

```
> (destructuring-bind (x (y) . z) '(a (b) c d)
    (list x y z))
(A B (C D))
```

这一新操作符和其它类似的操作符构成了第 18 章的主题。

解构在宏参数列表里也是可能的。Common Lisp 的 defmacro 宏允许任意列表结构作为参数列表。当宏调用被展开时，宏调用中的各部分将会以类似 destructuring-bind 的方式被赋值到宏的参数上面。内置的 dolist 宏就利用了这种参数列表的解构技术。在一个像这样的调用里：

```
(dolist (x '(a b c))
  (print x))
```

展开函数必须把 x 和 '(a b c) 从作为第一个参数给出的列表里抽取出来。这个任务可以通过给 dolist 适当的参数列表隐式地完成：[8]

```
(defmacro our-dolist ((var list &optional result) &body body)
  `(progn
     (mapc #'(lambda (,var) ,@body)
           ,list)
     (let ((,var nil))
       ,result)))
```

在 Common Lisp 中，诸如 dolist 这样的宏通常封装一个变元列表，它不属于宏体的一部分。由于 dolist 接受一个可选的 result 参数，它无论如何必须将其参数的第一部分封装到单独的列表里。但就算这个多余的列表结构是没有必要的，它也可以让对 dolist 的调用更易于阅读。假设我们想要定义一个宏 when-bind，它的功能和 when 差不多，除此之外它还能绑定一些变量到测试表达式返回的值上。这个宏最好的实现办法可能要用到一个嵌套的参数表：

```
(defmacro when-bind ((var expr) &body body)
  `(let ((,var ,expr))
     (when ,var
       ,@body)))
```

然后这样调用：

[8] 该版本用一种奇怪的方式来写以避免使用 gensym，这个操作符以后会详细介绍。

```
(defmacro our-expander (name) `(get ,name 'expander))

(defmacro our-defmacro (name parms &body body)
  (let ((g (gensym)))
    `(progn
       (setf (our-expander ',name)
             #'(lambda (,g)
                 (block ,name
                   (destructuring-bind ,parms (cdr ,g)
                     ,@body))))
       ',name)))

(defun our-macroexpand-1 (expr)
  (if (and (consp expr) (our-expander (car expr)))
      (funcall (our-expander (car expr)) expr)
      expr))
```

图 7.6: 一个 defmacro 的草稿

```
(when-bind (input (get-user-input))
  (process input))
```

而不是原本这样调用：

```
(let ((input (get-user-input)))
  (when input
    (process input)))
```

审慎地使用它，参数列表解构技术可以带来更加清晰的代码。最起码，它可以用在诸如 when-bind 和 dolist 这样的宏里，它们接受两个或更多的实参，和一个表达式体。

7.6 一个宏的模型

关于“宏究竟做了什么”的形式化描述将是既冗长又令人困惑的。即使有经验的程序员也无法将这样一个复杂的描述完全记在脑子里。想象一下 defmacro 会被怎样定义，通过这种方式来记忆它的行为会更方便些。

在 Lisp 里这样的解释方法历史悠久。*Lisp 1.5 Programmer's Manual*，于 1962 年首次出版，该书里给出了一个用 Lisp 写的 eval 函数的定义作为参考。由于 defmacro 自身也是一个宏，所以我们可以用同样方法来对待它，正如图 7.6 那样。这一定义里使用了几种我们尚未提及的技术，所以某些读者可能需要稍后再回过头来读懂它。

图 7.6 中的定义相当准确地再现了宏的行为，但就像任何草稿一样它是不完整的。它不能正确地处理 &whole 关键字。而且 defmacro 实际上作为其第一个

参数的 macro-function 存储的是一个带有两个变元的函数：宏调用本身，以及其发生时的词法环境。尽管如此，这些缺失的特性只有在最刁钻的宏里才会用到。就算假设宏就是像图 7.6 那样定义的，你在实际对宏的使用中也很难遇到和上述假设不一致的场合。例如，本书定义的每一个宏，都符合上述定义。

图 7.6 的定义里产生的一个展开函数是被带井号引用了的 λ-表达式。那将使它成为一个闭包：宏定义中的任何自由符号应该指向 defmacro 发生时所在环境里的变量。所以下列代码是可能的：

```
(let ((op 'setq))
  (defmacro our-setq (var val)
    (list op var val)))
```

上述代码对于 CLTL2 来说是没问题的。但在 CLTL1 里，宏展开器是被定义在空词法环境里的[9]，所以在一些老的 Common Lisp 实现里这个 our-setq 的定义将不会正常工作。

7.7 作为程序的宏

宏定义并不一定只是一个反引用列表。一个宏本质上是一个对表达式进行变换的函数。这个函数可以调用 list 来生成结果，但是同样也可以调用一整个长达数百行代码的子程序达到这个目的。

第 7.3 节给出了一个编写宏的简单方式。借助这一技术我们可以写出那种展开式中含有跟宏调用中相同的子表达式的宏。不幸的是，只有最简单的宏能满足这一条件。作为一个更复杂的例子，考虑内置的宏 do。要把 do 实现成一个只是把参数移来移去的宏是不可能的了。展开过程中必须构造出一些在宏调用中没有出现的复杂表达式。

关于编写宏的一个更通用的方法是，考虑你具体调用宏的时候使用的那些表达式，你希望它们被展开成什么样子，就写个能把前者变换成后者的程序。试着手工展开一个示例，然后看看在表达式从一种形式变换到另一种形式的过程中究竟发生了什么。通过在示例上的这些工作，你就可以大致明白在你将要写的宏里将需要些什么。

图 7.7 显示了 do 的一个实例，以及它应该展开成的表达式。手工做展开有助于理清你对于宏如何工作的理解。例如，在试着写展开式时你就不得不使用 psetq 来更新局部变量，如果不手写展开式说不定就会忽视这一点。

内置的宏 psetq (因 “并行 setq” 而得名) 在行为上和 setq 相似，除了它所有的 (第偶数个) 参数将在做任何赋值之前被求值。如果一个普通的 setq 带有两个以上参数，那么第一个参数的新值在第四个参数的求值过程中是可见的。

```
> (let ((a 1))
    (setq a 2 b a)
```

[9]关于这一区别实际起作用的例子，请参见某页的注释。

```
(do ((w 3)
     (x 1 (1+ x))
     (y 2 (1+ y))
     (z))
    ((> x 10) (princ z) y)
  (princ x)
  (princ y))
```

应该被展开成类似这样：

```
(prog ((w 3) (x 1) (y 2) (z nil))
   foo
    (if (> x 10)
        (return (progn (princ z) y)))
    (princ x)
    (princ y)
    (psetq x (1+ x) y (1+ y))
   (go foo))
```

图 7.7: do 的预期展开过程

```
    (list a b))
(2 2)
```

这里，因为 a 首先被设置，于是 b 得到了它的新值，2。一个 psetq 在行为上就像它的参数被并行地赋了值：

```
> (let ((a 1))
    (psetq a 2 b a)
    (list a b))
(2 1)
```

所以这里的 b 得到了 a 的旧值。这个 psetq 宏是特别为支持像 do 这样的宏而提供的，也就是需要并行地对它们的一些参数进行求值。(如果这里使用 setq 而非 psetq，那么最后定义出来的就不是 do 而是 do* 了。)

通过观察展开式，另一件很明显的事情是我们不能真的使用 foo 来做循环标签。设想如果 foo 也被用作 do 宏的表达主体里的循环标签会怎样呢？第 9 章将会具体解决这个问题；对于目前的情况，只要说宏展开必须使用一个由 gensym 返回的特殊匿名符号来代替 foo 就够了。

为了写出 do，我们接下来考虑一下，需要做些什么才能把图 7.7 中的第一个表达式变换成第二个。为了完成这样的变换，仅仅把宏的参数放在某个反引用列表的正确位置上已经不够了，我们需要做更多的工作。初始的 prog 需要紧跟着一个由符号和它们的初始绑定所组成的列表，而这些信息需要从传给 do 的第二个参数里解出。图 7.8 中的函数 make-initforms 将返回这样的一个列表。我们还需要为 psetq 构造一个参数的列表，但它的情况更为复杂因为并非

```
(defmacro our-do (bindforms (test &rest result) &body body)
  (let ((label (gensym)))
    `(prog ,(make-initforms bindforms)
       ,label
       (if ,test
           (return (progn ,@result)))
       ,@body
       (psetq ,@(make-stepforms bindforms))
       (go ,label))))

(defun make-initforms (bindforms)
  (mapcar #'(lambda (b)
              (if (consp b)
                  (list (car b) (cadr b))
                  (list b nil)))
          bindforms))

(defun make-stepforms (bindforms)
  (mapcan #'(lambda (b)
              (if (and (consp b) (third b))
                  (list (car b) (third b))
                  nil))
          bindforms))
```

图 7.8: 实现 do。

所有的符号都需要被更新。在图 7.8 中，make-stepforms 返回 psetq 所需的参数。有了这两个函数，定义的其余部分就相当直接了。

图 7.8 中的代码并不完全是 do 在真正的实现里的写法。为了强调在宏展开过程中完成的计算，make-initforms 和 make-stepforms 被切成分离的函数。未来，这样的代码通常会被留在 defmacro 表达式中。

通过这个宏的定义，我们开始领教到宏的能耐了。一个宏在构造表达式时对 Lisp 有完全的访问权限。用于生成展开式的代码可以是完全独立的一个程序。

7.8 宏风格

良好风格的含义对于宏来说有些不同。风格既体现在代码被人阅读的时候也体现在被 Lisp 求值的时候。宏的介入，使上述这些活动在略微不同寻常的场合下发生了。

有两类不同的代码跟一个宏定义相关联：*展开器代码*，被宏用来生成其展开式，以及*展开式代码*，出现在展开式本身的代码中。风格的原则对于每一类代码来说是不同的。通常对于程序来说，好的编码风格体现在清晰和高效上。这些原则在宏代码的两个类型上以相反的方向各有侧重：展开器代码偏爱清晰胜过效率，而展开式代码偏爱效率胜过清晰。

只有在编译了的代码里效率才是最重要的，而在编译了的代码里宏调用已经被展开了。如果展开器代码很高效，它只会使得代码的编译过程稍微快一些，但对程序如何运行来说没有任何差别。由于宏调用的展开只是编译器工作的很小一部分，那些可以高效展开的宏通常甚至不会在编译速度上产生明显的差异。所以大多数时候你都可以用编写一个程序的快速的第一版的那种方式，安全地编写宏展开器代码。如果展开器代码做了一些不必要的工作或者做了很多 cons，那又能怎样呢？你的时间最好花在改进程序的其他部分上面。如果在展开器代码里有一个清晰度和速度之间的选择，清晰度当然应该胜出。宏定义通常比函数定义更加难于阅读，因为宏定义里含有在两个不同时间求值的表达式的混合体。如果这些困惑可以通过不计展开器代码的效率开销而有所减轻，那这笔交易划得来。

举个例子，假设我们想要把一个版本的 and 定义为宏。由于 (and a b c) 等价于 (if a (if b c))，我们可以像图 7.9 中的第一个定义那样用 if 来实现 and。根据我们评判普通代码的标准，our-and 写得不好。展开器代码是递归的，而且在每次递归里都要需要查找同一个列表的每个后继 cdr 的长度。如果该代码打算在运行期求值，最好将该宏定义成像 our-andb 里的那样，它没有任何浪费就生成了同样的展开式。尽管如此，作为一个宏定义来说 our-and 至少是不错的，如果不是有个更好的。在每次递归中调用 length 可能是低效的，但它的代码组织方式可以更加清晰说明其展开式跟 and 的连接词数量之间

```
(defmacro our-and (&rest args)
  (case (length args)
    (0 t)
    (1 (car args))
    (t `(if ,(car args)
            (our-and ,@(cdr args))))))

(defmacro our-andb (&rest args)
  (if (null args)
      t
      (labels ((expander (rest)
                 (if (cdr rest)
                     `(if ,(car rest)
                          ,(expander (cdr rest)))
                     (car rest))))
        (expander args))))
```

图 7.9: 两个等价于 and 的宏。

的依赖关系。

任何事情都有例外。在 Lisp 里，对编译期和运行期的区分是人为的，所以任何依赖于此的规则也同样是人为的。在某些程序里，编译期也就是运行期。如果你在编写一个程序，它的主要目的就是进行代码变换，并且它使用宏来实现这个功能，那么一切就都变了：展开器代码成为了你的程序，而展开式是程序的输出。很明显，在这种情况下展开器代码应写得尽可能高效。尽管如此，还是可以说大多数展开器代码 (a) 只影响编译速度，而且 (b) 也不会影响太多——意味着代码的清晰度应该几乎总是作为首要考虑。

对于展开式代码来说，正好相反。对宏展开式来说清晰与否不太重要是因为它们很少被查看，尤其是被其他人查看。平时禁用的 goto 在展开式里被有限地解禁，被贬低的 setq 也不是那么被瞧不起了[10]。

结构化编程的拥护者不喜欢源代码里的 goto。这并不是说他们认为机器语言的跳转指令是有害的——它们通过更抽象的结构被隐藏在源代码里。在 Lisp 里 goto 被谴责其实是因为很容易把它们隐藏起来：你可以替代使用 do，并且如果你没有 do 可用，还可以自己写一个。很明显，如果你打算在 goto 的基础上构建新抽象，goto 一定会存在于某些地方。这样在一个新宏的定义中使用 go 也未必是不好的风格，如果它不能用一些已存在的宏来写的话。

类似地，setq 不被推荐的理由是它令我们难以看清一个给定的变量从哪里获得其值。尽管如此，考虑到宏展开式也不会被很多人阅读，所在在宏展开式里使用 setq 进行变量创建就是相对无害的了。如果你查看一些内置宏的展开式，你将看到许多的 setq。

[10]译者注：setq 被贬低的原因是因为它用来给变量赋值，通常认为函数型编程里应禁止使用赋值。

在某些场合下展开式代码的清晰性更加重要一些。如果你在编写一个复杂的宏，你可能最后还是要阅读它的展开式，至少在你调试它的时候。同样，在简单的宏里，只有一个反引用用来将展开器代码跟展开式代码分开，所以如果这样的宏生成了丑陋的展开式，这一丑陋在你的源代码里也能全部看到。尽管如此，甚至当展开式代码的清晰性成为需求时，效率应该仍然处于支配地位。对于大多数代码来说效率都是重要的。有两件事使得效率对于宏展开来说尤为重要：它们的普遍性和不可见性。

宏通常用于实现通用性的实用工具，这些工具会被用在程序的每个地方。如此常用的一些东西是无法忍受低效率的。那些看起来几乎无害的宏，在对它的所有调用都展开以后，可能会占据你程序中的相当篇幅。这样的宏得到的重视应当比因为它们的长度所获得的重视更多才是。尤其是要避免 cons。一个做了不必要的 cons 的实用工具可能会毁掉一个本来高效的程序的性能。

关注展开式代码效率的另一个原因就是它们严重的不可见性。如果一个函数被实现得不好，那么它会在你每次查看其定义时表明这一事实。宏就不是这样了。展开式代码的低效率在宏的定义里可能并不明显，这也就是需要更加关注它的全部原因。

7.9 宏的依赖关系

如果你重定义了一个函数，其他调用它的函数们将会自动得到新版本。[11] 同样的情况对宏来说就不一定了。当函数被编译时，函数定义中的宏调用就会被替换成它的展开式。如果我们在主调函数编译以后，重定义那个宏会发生什么呢？由于对最初的宏调用的无迹可寻，所以函数里的展开式不能被更新。该函数的行为将继续反映出宏的原来的定义：

```
> (defmacro mac (x) `(1+ ,x))
MAC
> (setq fn (compile nil '(lambda (y) (mac y))))
#<Function "LAMBDA (Y)" 58133F31>
> (defmacro mac (x) `(+ ,x 100))
MAC
> (funcall fn 1)
2
```

如果在定义宏之前，调用这个宏的代码就已经被编译了的话，类似的问题就会发生。CLTL2 说“一个宏定义必须在其首次使用之前被编译器看到。”各种实现对违反这个规则的后果反应各有不同。幸运的是能很容易地避免这两类问题。如果坚持下述两点原则，你就从来不会为过时或者不存在的宏定义所困扰：

1. 在宏被函数 (或其他宏) 调用之前定义它们。

[11] 编译时内联 (inline) 的函数除外，它们和宏的重定义受到相同的约束。

2. 当一个宏被重定义时，同时也重新编译所有调用它的函数 (或宏)——可以直接或通过其他宏来做。

有些人建议将一个程序中所有的宏放在一个单独的文件里，便于保证宏定义被首先编译。这样有点过头了。我们建议把诸如 `while` 此类通用性的宏放在单独的文件，但通用性实用工具应该无论如何都要和程序的其余部分分开，不论这些实用工具到底是函数还是宏。

某些宏只是为了用在程序的某个特定部分而写的，自然，这种宏应该跟使用它们的代码放在一起。只要保证每一个宏的定义都出现在任何对它们的调用之前，你的程序就可以很好地编译。单单因为它们都是宏，所以就把所有的宏集中写在一起，这样做不会带来任何好处，只会让你的代码更难以阅读。

7.10 来自函数的宏

本节描述如何将函数转化为宏。将函数转化为宏的第一步是问问你自己是否真的需要这么做。难道你就不能只是把函数声明为 `inline` (p.23) 吗？

然而，确实有一些正当的理由让我们探讨“如何将函数转化为宏”这个问题。当你刚开始写宏的时候，假设自己写的是函数一般会有些帮助——这样做产生的宏一般多少会有些问题，但这至少能让你开始起步。关注宏与函数之间关系的另一个原因是看看它们究竟有何*不同*。最后，Lisp 程序员们有时确实想要把函数改造成宏。

函数转化为宏的难度取决于该函数的一些特性。最容易转化的一类函数具有下列特性：

1. 其函数体只有单个表达式。
2. 其参数列表只由参数名组成。
3. 不创建任何新变量 (参数除外)。
4. 不是递归的 (也不属于任何相互递归组)。
5. 每个参数在函数体里只出现一次。
6. 没有一个参数，它的值会在其参数列表之前的另一个参数出现之前被用到。
7. 不含有自由变量。

其中一个满足这些规定的内置 Common Lisp 函数是 `second`，它返回列表的第二个元素。它可以被定义成：

```
(defun second (x) (cadr x))
```

一个满足上述所有条件的函数，你可以轻易地将其转化成等价的宏定义。简单地把一个反引用放在函数体的前面再把逗号放在每一个出现在参数列表里的符号前面就好了：

```
(defmacro second (x) `(cadr ,x))
```

当然，该宏也不是在所有相同条件下都可以使用。它不能作为 apply 或者 funcall 的第一个参数给出，and it should not be called in environments where the functions it calls have new local bindings. 不过对于普通的内联调用，second 宏应该和 second 函数做同样的事。

当函数体有不止一个表达式时，这种技术要稍微变通一下，因为一个宏必须展开成一个单独的表达式。所以条件 1 不能满足，你就必须追加一个 progn。函数 noisy-second：

```
(defun noisy-second (x)
  (princ "Someone is taking a cadr!")
  (cadr x))
```

可以用下列宏来重现：

```
(defmacro noisy-second (x)
  `(progn
     (princ "Someone is taking a cadr!")
     (cadr x)))
```

当函数因为有 &rest 或者 &body 参数而不满足条件 2 时，规则是一样的，除了参数的处理，代替简单地把一个逗号放在前面，参数必须被拼接到一个 list 调用里。这样

```
(defun sum (&rest args)
  (apply #'+ args))
```

变成

```
(defmacro sum (&rest args)
  `(apply #'+ (list ,@args)))
```

当条件 3 无法满足——函数体里有新变量被创建——关于插入逗号的规则必须被修改。代替在所有参数列表里的符号前面放逗号，我们只把逗号放在那些指向参数的符号前面。例如，在：

```
(defun foo (x y z)
  (list x (let ((x y))
            (list x z))))
```

最后两个 x 的实例都没有指向参数 x。第二个实例是不求值的，而第三个实例指向了由 let 建立的新变量。所以只有第一个实例会带上逗号：

```
(defmacro foo (x y z)
  `(list ,x (let ((x ,y))
              (list x ,z))))
```

有时不满足条件 4、5 和 6 的函数也能转化为宏。尽管如此，这些主题会在以后的章节里分别处理。宏里面的递归问题在第 10.4 节里处理，而多重和乱序求值的危险将分别在第 10.1 和 10.2 节里介绍。

至于条件 7，是有可能用一种技术让宏来模拟闭包的，这种技术类似 32 页中提到的一个错误。不过介于这个办法有点取巧，和本书中名门正派的作风不大协调，因此我们就此点到为止。

7.11 符号宏 (symbol–macro)

CLTL2 为 Common Lisp 引入了一种新型的宏，符号宏 (symbol–macro)。如果说一个正常的宏调用看起来像一个函数调用，那么一个符号宏“调用”看起来就像一个符号。

符号宏只能用于局部定义。`symbol-macrolet` 的 special form 可以在其体内，令一个孤立的符号其行为像一个表达式：

```
> (symbol-macrolet ((hi (progn (print "Howdy")
                               1)))
    (+ hi 2))
"Howdy"
3
```

`symbol-macrolet` 主体中的表达式将会像每一个参数位置的 `hi` 都被替换成 `(progn (print "Howdy") 1)` 那样进行求值。

看上去，符号宏就像不带任何参数的宏。在没有参数的时候，宏就变成了简单的缩写。尽管如此，这并不是说符号宏一点用也没有。他们在第 15 章 (第 187 页) 和第 18 章 (第 203 页) 都有用到，而且在以后给出的实例里是也不可或缺的。

第 8 章

何时使用宏

我们如何知道一个给定的函数是否真的应该是函数，而不是宏呢？多数时候，在需要和不需要用到宏的两种情况之间会有一个明确的界限。缺省情况下，我们应该用函数，因为如果函数能解决问题，而偏要用上宏的话，会让程序变得不优雅。我们应当只有在宏能带来特别的好处时才使用它们。

什么情况下，宏能给我们带来好处呢？这就是本章的主题。通常这不是好处的问题，而是一种需要。大多数我们用宏可以做到的事情，函数都无法完成。第 8.1 节列出了只能用宏来实现的几种操作符。尽管如此，也有一小类 (但很有意思) 的非典型案例，在这里一个操作符要谨慎地决定到底是写成函数还是宏。对于这些情形，第 8.2 节给出了关于宏的正反两方面的观点。最后，在充分考察了宏的能力之后，我们在第 8.3 节里转向一个相关的问题：人们用宏做哪种事情？

8.1 当别无他法时

优秀设计的一个通用原则就是：当你发现在程序的几个地方上出现了相似的代码时，就应该写一个子例程然后将那些相似的代码序列替换成对同一个子例程的调用。当我们将此原则应用到 Lisp 程序时，就必须决定这个“子例程” 应该是函数还是宏。

在某些场合很容易决定应当写一个宏而不是函数，因为只有宏才能满足需求。一个像 `1+` 这样的函数或许既可以写成函数也可以写成宏：

```
(defun 1+ (x) (+ 1 x))

(defmacro 1+ (x) `(+ 1 ,x))
```

但是来自第 7.3 节的 `while`，则只能被定义成一个宏：

```
(defmacro while (test &body body)
  `(do ()
       ((not ,test))
     ,@body))
```

没有办法用一个函数来重现这个宏的行为。`while` 的定义里拼接了一个作为 `body` 传入 `do` 的主体里的表达式，它只有当 `test` 表达式返回 `nil` 时才会被

求值。没有函数可以做到这一点；在一个函数调用里，所有参数在函数调用开始之前就会被求值。

当你需要一个宏的时候，你想从它那里得到什么呢？宏有两件事是函数做不到的：宏可以控制 (或阻止) 对其参数的求值，以及它可以展开进入到主调方的上下文中。任何要用宏的应用，最后归根到底都需要上述两个属性中的至少一个。

“宏不求值其参数” 这一非正式的解释稍微有些问题。更准确的说法是宏控制宏调用中参数的求值。取决于参数出现在宏展开式中的位置，它们可以被求值一次，多次，或者根本不求值。宏的这一控制体现在四种主要方式上：

1. 变换 *(Transformation)*. Common Lisp 的 `setf` 宏就是一类在求值前拆散其参数的宏之一。一个内置的访问函数 (access function) 通常都有一个对应的逆操作，其目的是对那个本来会被访问函数获取到的对象赋值。`car` 的逆操作是 `rplaca`，对于 `cdr` 来说是 `rplacd`，诸如此类。在 `setf` 中，我们将这种对访问函数的调用当成是要被赋值的变量使用，就像在 `(setf (car x) 'a)` 中那样，然后该表达式会被宏展开成 `(progn (rplaca x 'a) 'a)`.

   为了达到这样的效果，`setf` 必须查看其第一个参数的内部。想要知道上述情况下需要 `rplaca`，`setf` 就必须能够知道它的第一个参数是个以 `car` 开始的表达式。这样的话，`setf` 以及其他对参数进行变换的操作符，就必须被写成一个宏。

2. 绑定 *(Binding)*. 词法变量必须直接出现在源代码中。例如，由于 `setq` 的第一个参数是不求值的，所以任何构建在 `setq` 之上的东西都必须是一个展开到 `setq` 的宏，而不能是一个调用它的函数。对于像 `let` 这样的操作符也是同样的道理，其实参必须作为 `lambda` 表达式的形参出现，以及像 `do` 这样展开到 `let` 的宏，等等。任何的新操作符，如果它希望替换其参数的词法绑定，那么它就必须写成宏。

3. 条件求值 *(Conditional evaluation)*. 函数的所有参数都会被求值。在像 `when` 这样的结构里，我们希望一些参数仅在特定条件下才被求值。这种灵活性只有通过宏才可能达到。

4. 多重求值 *(Multiple evaluation)*. 不但函数的所有参数都会被求值，而且都明确地只被求值一次。我们需要宏来定义像 `do` 这样的结构，其中特定的参数可以被重复求值。

也有几种方式可以利用宏产生的内联表达式带来的优势。这里要重点强调一下，宏展开后生成的展开式将会出现在宏调用所在的词法环境中，因为下列三种用法中的两种都依赖于这一事实。它们是：

5. *利用调用方环境 (Using the calling environment).* 一个宏可以生成含有其绑定来自宏调用上下文环境的变量的展开式。接下来这个宏：

```
(defmacro foo (x)
  `(+ ,x y))
```

其行为依赖于当 `foo` 被调用时 `y` 的绑定。

这种词法交流通常更多地被视为传染源而非乐趣的来源。一般来说，写这样的宏不是什么好习惯。函数式编程的思想对于宏来说也是适用的：与一个宏交流的最佳方式是通过其参数。事实上，使用调用方环境的情况非常罕见，因而，如果出现这样的用法，那十有八九就是什么地方出了问题。(见第 9 章。) 纵观本书中的所有宏，只有续延传递 (continuation–passing) 宏 (第 20 章) 和 ATN 编译器 (第 23 章) 的某些部分以这种方式利用了调用方环境。

6. *包装新环境 (Wrapping a new environment).* 一个宏也可以使其参数在一个新的词法环境下被求值。经典例子就是 `let`，它可以用 `lambda` 实现成宏的形式 (见 ?? 页)。在一个像 `(let ((y 2)) (+ x y))` 这样的表达式体中，`y` 将指向新变量。

7. *减少函数调用 (Saving function calls).* 宏展开的内联插入的第三个结果是宏调用在编译后的代码中没有额外开销。在运行期，宏调用已经被替换成了它的展开式。(同样的原理对于声明为 `inline` 的函数来说也是正确的。)

很明显的一点，情形 5 和 6，如果不是有意为之，将产生变量捕捉上的问题，这可能是宏的编写者担心的所有事情里面，最头疼的一件。变量捕捉将在第 9 章里进行讨论。

与其说有七种使用宏的方式，不如说有六个半。在理想的世界里，所有 Common Lisp 编译器都会遵守 `inline` 声明，所以减少函数调用将是内联函数的任务，而不是宏的。这个建立理想世界的重任就作为练习留给读者吧。

8.2 宏还是函数?

前一节处理的是简单情况。任何需要在参数求值前访问它们的操作符都应该被写成宏，因为没有其它选择。对于那些同时支持两种写法的操作符呢？比如说，操作符 `avg`，它返回其参数的平均值。它可以被定义成一个函数

```
(defun avg (&rest args)
  (/ (apply #'+ args) (length args)))
```

但把它定义成宏也比较合适：

```
(defmacro avg (&rest args)
  `(/ (+ ,@args) ,(length args)))
```

因为函数版本在每次 `avg` 被调用时，都对 `length` 做了一次不必要的的调用。在编译期我们可能不知道这些参数的值，但我们却清楚参数的个数，所以在那时进行 `length` 的调用正合适。当我们面对这样的选择时可以考虑下列几点：

利

1. *编译期计算 (Computation at compile-time).* 一个宏调用在两个时期卷入计算：当宏被展开时，以及当展开式被求值时。Lisp 程序中的所有宏展开在程序被编译时就完成了，而在编译期可以完成的每一位计算就等于在运行期不会将程序拖慢。如果一个操作符可以被写成在宏展开阶段做一些工作，那么把它做成宏将会使程序更加高效，因为无论什么工作只要一个聪明的编译器自身都无法做到，一个函数就不得不在运行期做。第 13 章描述了类似 `avg` 的能在宏展开时期做掉一些它们的工作的宏。

2. *跟 Lisp 集成 (Integration with Lisp).* 有时，使用宏代替函数将使程序更紧密地跟 Lisp 集成。与其写一个程序来解决一个特定的问题，你可以用宏将此问题变换成另一个 Lisp 已经知道如何解决的问题。这种观念，当可能时，通常可以使程序更小也更高效：更小是因为 Lisp 帮你做了一部分工作，而更高效是因为产品级 Lisp 系统通常比用户程序做了更多的优化。这一优势多数出现在嵌入式语言里，我们从第 19 章开始描述。

3. *节省函数调用 (Saving function calls).* 一个宏调用在其出现的地方直接展开成代码。所以如果你把经常使用的代码片段写成宏，你就可以在每次使用它的时候节省一次函数调用。在早期的 Lisp 方言里，程序员们借助于宏的这一属性在运行期节省函数调用。在 Common Lisp 中，这一工作被声明为 `inline` 类型的函数接手了。

   通过将函数声明为 `inline`，你请求将它编译进调用的代码里，就像宏一样。尽管如此，理论和实践之间还是有些距离；CLTL2 (229 页) 说 “一个编译器可以随意地忽略该声明”，而且某些 Common Lisp 编译器确实是这样做的。

在某些情况下，效率因素和跟 Lisp 之间紧密集成的组合优势可以充分证实使用宏的必要性。在第 19 章的查询编译器里，可以转移到编译期的计算量相当可观，这使我们有理由把整个程序变成一个单独的巨大的宏。尽管效率是初衷，这一转移同时也使程序更加接近 Lisp：在新的版本中使用 Lisp 表达式变得更容易了，比如说可以在查询的时候用 Lisp 的算术表达式。

弊

4. *函数即是数据 (Functions are data)*, 而宏对于编译器来说更像是一些指令。函数可以作为参数被传递 (例如用 `apply`)，被函数返回，或者保存在数据结构里。对于宏来说这些事情都不可能。

   在某些情况下，你可以通过将宏调用封装在一个 lambda–表达式里来得到你想要的。这可以工作，例如，如果你想用 `apply` 或者 `funcall` 来调用特定的宏：

   ```
   > (funcall #'(lambda (x y) (avg x y)) 1 3)
   2
   ```

   尽管如此，这样做并不便利。它也并不总能正常工作：甚至如果，像 `avg`，该宏带有一个 `&rest` 类型参数，没有办法给它传递可变数量的变元。

5. *源代码清晰 (Clarity of source code)*. 宏定义和等价的函数定义相比更难阅读。所以如果将某些东西写成宏只能使程序稍微好一些，那么最好还是改为使用函数。

6. *运行期清晰 (Clarity at runtime)*. 宏有时比函数更加难以调试。如果你在含有许多宏的代码里得到一个运行期错误，你在 backtrace 里看到的代码将包含所有这些宏调用的展开式，而它们和你最初写的代码看起来可能很不一样。

7. *递归 (Recursion)*. 在宏里使用递归不像在函数里那么简单。尽管展开一个宏里的展开函数可能是递归的，但展开式本身可能不是。第 10.4 节将处理跟宏里的递归有关的主题。

在决定何时使用宏的时候需要权衡利弊，综合考虑所有这些因素。只有靠经验才能知道哪一个因素在起主导作用。尽管如此，出现在后续章节里的宏的示例涵盖了大多数对宏有利的情形。如果一个潜在的宏符合这里给出的条件，那么把它写成这样可能就是合适的。

最后，应该注意运行期清晰 (观点 6) 很少是问题。调试那种用很多宏写成的代码并不像你想象的那样困难。如果一个宏的定义长达数百行，在运行期调试它的展开式的确是件苦差事。但至少实用工具往往出现在小的，可靠的程序层次中。通常它们的定义长度少于 15 行。所以就算你最终只得仔细检查一系列的 backtrace，这种宏也不会让你云遮雾绕，摸不着头脑。

8.3 宏的应用

在考察了宏能用来做什么之后，下一个问题是：我们可以将宏用在哪一类程序里？关于宏的用途，最接近正式表述的说法可能会说：它们主要用于句法转

换 (syntactic transformations)。这并不是要严格限制宏的使用范围。由于 Lisp 程序从列表中生成[1]，而列表是 Lisp 数据结构，“句法转换” 的确有很大的发挥空间。第 19–24 章展示的整个程序，其目的就可以被描述成 “句法转换”，而且从效果上来看，所有宏都是这样。

宏的种种应用一起构成了一条缎带，这些应用从像 `while` 这样的小型通用目标的宏，直到后面章节里定义的大型、特殊用途的宏都有。缎带的一端是*实用工具*，它们和每个 Lisp 都内置的那些宏是一样的。它们通常短小、通用，而且相互独立。尽管如此，你也可以为一些特别类型的程序编写实用工具，然后当你有一组宏用于，比如说，图形程序的时候，它们看起来就像是一种专门用于图形编程的语言。在缎带的远端，宏允许你用一种跟 Lisp 截然不同的语言来编写整个程序。以这种方式使用宏的做法被称为实现*嵌入式语言*。

实用工具是自底向上风格的首批成果。甚至当一个程序规模很小而不必分层构建时，它也仍然能够从最底层，Lisp 本身的扩充里受益。实用工具 `nil!`，将其参数设置为 `nil`，只能被定义成一个宏：

```
(defmacro nil! (x)
  `(setf ,x nil))
```

观察这个 `nil!`，可能有人会说它什么都做不了，最多是可以让我们少输入几个字罢了。是的，但是充其量，宏所能做的也就是让你少打些字而已。如果有人非要这样想的话，那么其实编译器的工作也不过是让人们用机器语言编程的时候可以少些。实用工具的价值不应该被低估，因为它们的效果会积少成多：几层简单的宏拉开了一个优雅的程序和一个晦涩的程序之间的差距。

多数实用工具是含有模式的。当你注意到代码中存在一个模式，那么可以考虑将其转换成实用工具。模式是计算机最擅长的。为什么有程序可以帮你做，你还要自己做呢？假设在写某个程序的时候，你发现你自己以同样的通用形式在很多地方做循环操作：

```
(do ()
    ((not ⟨condition⟩))
  . ⟨body of code⟩)
```

当你在你的代码里发现一个重复的模式时，这个模式经常会有一个名字。这里，模式的名字是 *while*。如果我们想把作为一个实用工具提供出来，我们将用宏的形式，因为我们需要带有条件判断的、以及重复的求值。如果我们使用来自第 82 页的这个定义来定义 `while`：

```
(defmacro while (test &body body)
  `(do ()
       ((not ,test))
     ,@body))
```

[1] 从列表中生成，是指列表作为编译器的输入。函数不再是从列表中生成，虽然它们曾经在一些早期方言里确实是这样。

那么就可以将该模式的所有实例替换成：

```
(while ⟨condition⟩
  . ⟨body of code⟩)
```

这样做就令代码更短，而且同时也会更清晰地表明代码的意图。

这种变换其参数的能力使得宏在编写接口时特别有用。适当的宏可以在本应需要输入冗长复杂表达式的地方只输入简短的表达式。尽管图形界面减少了为最终用户编写这类宏的需要，程序员却一直使用这种类型的宏。最普通的例子是 `defun`，它使得函数绑定，在表面上，类似于一个用 Pascal 或 C 这样的语言定义的函数。第 2 章提到下面两个表达式差不多具有相同的效果：

```
(defun foo (x) (* x 2))

(setf (symbol-function 'foo)
      #'(lambda (x) (* x 2)))
```

这样 `defun` 就可以实现成一个将前者转换成后者的宏。我们可以想象它会这样写：

```
(defmacro our-defun (name parms &body body)
  `(progn
     (setf (symbol-function ',name)
           #'(lambda ,parms (block ,name ,@body)))
     ',name))
```

像 `while` 和 `nil!` 这样的宏可以被视为通用性实用工具。任何 Lisp 程序都可以使用它们。但是特定领域也完全可以有它们自己的实用工具。没理由假设基本 Lisp 是你用来扩展编程语言的唯一层次。如果你正在编写一个 CAD 程序，举个例子，有时最佳结果可能会是将它写在两层里：一门专用于 CAD 程序的语言 (或者如果你偏爱更现代的术语，一个工具箱 (toolkit))，以及在这层之上的，你的特定应用。

Lisp 模糊了许多对其他语言来说理所当然的差异。在其他语言里，在编译期和运行期，程序和数据，以及语言和程序之间具有真正概念上的差异。而在 Lisp 里，这些差异就仅仅存在于口头约定了。例如在语言和程序之间就没有明确的界限。你可以根据手头程序的情况自行界定。所以是否将一个低层代码称为工具箱或者语言真的只不过是个术语上的问题了。将其看作语言的一个优势是它建议你可以扩展这个语言，对 Lisp 来说，通过实用工具。

设想我们正在编写一个交互式的 2D 绘图程序。出于简化考虑，我们将假定程序处理的对象只有线段，表示成一个起点 $\langle x, y\rangle$ 和一个向量 $\langle dx, dy\rangle$。这样一个程序要做的事情之一就是平移一组对象。这就是图 8.1 中函数 `move-objs` 的目标。出于效率考虑，我们不想在每个操作结束后重绘整个屏幕——只画那些改变了的部分。因此两次调用了函数 `bounds`，它返回表示一组对象的矩形边界的四个坐标 (最小 x，最小 y，最大 x，最大 y)。`move-objs` 的操作部分被夹

```
(defun move-objs (objs dx dy)
  (multiple-value-bind (x0 y0 x1 y1) (bounds objs)
    (dolist (o objs)
      (incf (obj-x o) dx)
      (incf (obj-y o) dy))
    (multiple-value-bind (xa ya xb yb) (bounds objs)
      (redraw (min x0 xa) (min y0 ya)
              (max x1 xb) (max y1 yb)))))

(defun scale-objs (objs factor)
  (multiple-value-bind (x0 y0 x1 y1) (bounds objs)
    (dolist (o objs)
      (setf (obj-dx o) (* (obj-dx o) factor)
            (obj-dy o) (* (obj-dy o) factor)))
    (multiple-value-bind (xa ya xb yb) (bounds objs)
      (redraw (min x0 xa) (min y0 ya)
              (max x1 xb) (max y1 yb)))))
```

图 8.1: 最初的移动和缩放。

在了两次对 bounds 调用的中间，它们分别找到移动前后的矩形边界，然后重绘整个区域。

函数 scale-objs 被用来改变一组对象的大小。由于区域边界可能随缩放因子的不同而放大或者缩小，这个函数也必须在两次 bounds 调用之间发生作用。随着我们绘图程序开发进度的不断推进，这个模式一次又一次地出现在我们眼前：在旋转，翻转，转置等函数里。

通过一个宏，我们可以抽象出这些函数共有的通用代码。图 8.2 中的宏 with-redraw 提供了一个图 8.1 中函数共享的骨架。[2] 结果，它们现在可以每个函数用四行代码即可定义了，正如图 8.2 结尾那样。通过这两个函数，这个新宏在简洁性方面已经证明了它是值得的。并且这两个函数在屏幕重绘的细节部分被抽象掉以后变得清晰多了。

对 with-redraw 来说，有一种看法是将其作为一个用于编写交互性绘图程序的语言里的结构。随着我们开发出更多这样的宏，它们不管从名义上，还是在实际上都会构成一门专用的编程语言，并且我们的程序也将开始表现出其不俗之处，这正是我们用特制的语言撰写程序所期望得到的效果。

宏的另一个主要用途就是实现嵌入式语言。Lisp 在编写编程语言方面是一种特别优秀的语言，因为 Lisp 程序可以表达成列表，而且 Lisp 还有内置的解析器 (read) 和编译器 (compile) 用于以这种方式表达的程序。多数时候你甚至不需要调用 compile；你可以让你的嵌入式语言隐式地编译，通过编译那些用来做转换的代码 (第 22 页)。

[2] 这个宏的定义里使用了下一章才会出现的 gensym。它的作用接下来就会说明。

```
(defmacro with-redraw ((var objs) &body body)
  (let ((gob (gensym))
        (x0 (gensym)) (y0 (gensym))
        (x1 (gensym)) (y1 (gensym)))
    `(let ((,gob ,objs))
       (multiple-value-bind (,x0 ,y0 ,x1 ,y1) (bounds ,gob)
         (dolist (,var ,gob) ,@body)
         (multiple-value-bind (xa ya xb yb) (bounds ,gob)
           (redraw (min ,x0 xa) (min ,y0 ya)
                   (max ,x1 xb) (max ,y1 yb)))))))

(defun move-objs (objs dx dy)
  (with-redraw (o objs)
    (incf (obj-x o) dx)
    (incf (obj-y o) dy)))

(defun scale-objs (objs factor)
  (with-redraw (o objs)
    (setf (obj-dx o) (* (obj-dx o) factor)
          (obj-dy o) (* (obj-dy o) factor))))
```

图 8.2: 骨肉分离后的移动和缩放。

嵌入式语言并非完全是构建于 Lisp 之上的语言，它更多的是和 Lisp 融合在一起，这使得其语法成为了一个 Lisp 和新语言中特有结构的混合体。实现嵌入式语言的初级方式是用 Lisp 给它写一个解释器。有可能的话，一个更好的方法是通过语法转换来实现这种语言：将每个表达式转换成 Lisp 代码然后让解释器可以通过求值的方式来运行它。这就是宏大展身手的时候了。宏的工作恰好是将一种类型的表达式转换成另一种类型，所以在编写嵌入式语言时，它们是不二之选。

一般而言，嵌入式语言可以通过转换来实现的部分越多越好。主要是可以节省工作量。举个例子，如果新语言里用到算术，你就不需要面对表示和处理数值量的所有复杂性。如果 Lisp 的算术能力可以满足你的需要，那么你可以简单地将你的算术表达式转换成等价的 Lisp 表达式，然后将其余的留给 Lisp 处理。

代码转换通常都会使你的嵌入式语言更快。而解释器在速度方面一直处于劣势。当代码里出现循环时，解释器通常都必须在每次迭代中都去解释代码而编译器却只需做一次编译。一个拥有解释器的嵌入式语言将因此会很慢，就算解释器本身是编译的。但如果新语言里的表达式被转换成 Lisp 了，那么这些转换出来的代码就会被 Lisp 编译器所编译。一个如此实现的语言不需要在运行期承受解释的开销。在没有为你的语言写一个真正编译器的情况下，借助宏将得到最好的性能。事实上，转换新语言的宏可以看作该语言的编译器——只不过它

依赖已有的 Lisp 编译器来做大部分工作。

这里我们将不会考虑任何嵌入式语言的例子，第 19–25 章都是关于该主题的。第 19 章特别处理了解释与转换嵌入式语言之间的区别，并且用同时用这两种方法实现了同样的语言。

有一本关于 Common Lisp 的书断言宏的作用域是有限的，并且引用了一个事实作为依据，在所有 CLTL1 里定义的操作符中，只有少于 10% 的操作符是宏。这就好比是说因为我们的房子是用砖砌成的，我们的家具也必须得是。宏在一个 Common Lisp 程序中所占的比例多少完全要看这个程序想干什么。有的程序里可能根本没有宏，而有的程序可能全是宏。

第 9 章

变量捕捉

宏对于一类称为变量捕捉的问题有些无能为力。变量捕捉发生在宏展开导致名字冲突的时候：某些符号出乎意料地最终引用了来自另一个上下文中的变量。不经意中的变量捕捉可能造成极其难以察觉的 bug。本章将介绍如何预见和避免它们的办法。尽管如此，有意的变量捕捉却也是一种有用的编程技术，而且第 14 章里全是靠这种技术实现的宏。

9.1 宏参数捕捉

一个宏，如果它对无意识的变量捕捉毫无防备，那么它就是一个有 bug 的宏。为了避免写出这样的宏，我们必须确切地知道何时捕捉会发生。变量捕捉，按照实际的情形可以被归到下面两类情况：宏参数捕捉和自由符号捕捉。所谓参数捕捉，就是一个在宏调用中作为参数传递的符号偶然地引用到了宏展开式本身建立的变量上。考虑下面这个 `for` 宏的定义，它像一个 Pascal `for` 循环迭代在一组表达式体上：

```
(defmacro for ((var start stop) &body body)
  `(do ((,var ,start (1+ ,var))
        (limit ,stop))
       ((> ,var limit))
     ,@body))
```

这个宏乍看之下是没问题的。它甚至似乎也可以正常工作：

```
> (for (x 1 5)
    (princ x))
12345
NIL
```

确实，这个错误是如此的隐蔽，使得我们可能使用这个版本的宏几百次，每次它都能顺利完成任务。但如果我们以这种方式调用它，就会出问题了：

```
(for (limit 1 5)
  (princ limit))
```

我们可能期待这个表达式和之前的那个有同样效果。但它并没有任何输出：它产生了一个错误。为了找到原因，我们查看它的展开式：

```
(do ((limit 1 (1+ limit))
     (limit 5))
    ((> limit limit))
  (print limit))
```

现在错误的地方就很明显了。在宏展开式本身的符号和作为参数传递给宏的符号之间出现了名字冲突。宏展开*捕捉*了 limit。这导致它在同一个 do 里出现了两次，这是非法的。

由变量捕捉导致的错误比较罕见，但频率越低其性质就越恶劣。上个捕捉还是相对比较温和的——至少，这次我们得到了一个错误。更普遍的情况是，一个捕捉了变量的宏只是产生错误的结果，却没有给出任何迹象显示问题出在什么地方。在下面的例子中，

```
> (let ((limit 5))
    (for (i 1 10)
      (when (> i limit)
        (princ i))))
NIL
```

产生的代码静悄悄地什么也不做。

9.2 自由符号捕捉

偶尔会出现一种情况，宏定义本身含有的一个符号，在宏被展开时无意中引用到其所在环境中的一个绑定。假设有一个程序，希望将程序运行中产生的警告信息保存在一个列表里以供事后检查，而不是在其发生时直接打印输出给用户。某人写了一个宏 gripe，它接受一个警告信息，然后将其加入到一个全局列表 w 中：

```
(defvar w nil)

(defmacro gripe (warning)                                    ; wrong
  `(progn (setq w (nconc w (list ,warning)))
          nil))
```

之后，另一个人希望写一个函数 sample-ratio，来返回两个列表的长度比。如果任何一个列表的元素少于两个，函数就改为返回 nil，同时产生一个警告说明这个函数在处理的是一个统计学上没有意义的样本。(实际的警告本可以带有更多的信息，但它们的内容与本例无关。)

```
(defun sample-ratio (v w)
  (let ((vn (length v)) (wn (length w)))
    (if (or (< vn 2) (< wn 2))
        (gripe "sample < 2")
        (/ vn wn))))
```

如果用 w = (b) 来调用 sample-ratio，那么它将会警告说它的一个参数只有一个元素，所得出的结果从统计上来讲没有意义。但是当对 gripe 的调用被展开时，sample-ratio 就好像被定义成：

```
(defun sample-ratio (v w)
  (let ((vn (length v)) (wn (length w)))
    (if (or (< vn 2) (< wn 2))
        (progn (setq w (nconc w (list "sample < 2")))
               nil)
        (/ vn wn))))
```

这里的问题是 gripe 被用在一个 w 存在局部绑定的上下文中。产生的警告没有保存到全局警告列表里，而是被 nconc 连接到了 sample-ratio 的一个参数的结尾。不但警告丢失了，而且列表 (b) 也被追加了一个额外的字符串，它可能还作为数据用在了程序的其他位置：

```
> (let ((lst '(b)))
    (sample-ratio nil lst)
    lst)
(B "sample < 2")
> w
NIL
```

9.3 何时捕捉发生

许多宏的编写者都希望能通过查看宏的定义，就可以预见到所有可能来自上述两种捕捉类型的问题。变量捕捉有些难以捉摸，需要一些经验才能预料到那些被捕捉的变量在程序中所有捣乱的伎俩。幸运的是，可以在你的宏定义中检测并排除可能被捕捉的符号而无需考虑它们的捕捉是如何搞砸你的程序的。本节介绍一套直接了当的检测规则，用它就可以找出可捕捉的符号。本章的其余部分则解释了避免出现变量捕捉的相关技术。

我们接下来提出的规则可以被用来定义可捕捉变量，但是它基于几个从属的概念，所以在继续之前必须首先给这些概念下个定义：

自由 *(free)*：表达式中出现的一个符号 *s* 称为自由的，当且仅当它被用作表达式中的变量，但表达式却没有为它创建一个绑定。

在下列表达式里，

```
(let ((x y) (z 10))
  (list w x z))
```

w, x 和 z 都是自由地出现在 list 表达式，该表达式没有创建任何绑定。尽管如此，外围的 let 表达式为 x 和 z 创建了绑定，所以在作为整体的 let 里面，只有 y 和 w 是自由地出现。注意到在

```
(let ((x x))
  x)
```

里 x 的第二个实例是自由的——它并不在为 x 创建的新绑定的范围内。

框架 *(skeleton)*：宏展开式的框架是整个展开式，除去任何在宏调用中作为实参的部分。

如果 foo 被定义为：

```
(defmacro foo (x y)
  `(/ (+ ,x 1) ,y))
```

并且被这样调用：

```
(foo (- 5 2) 6)
```

然后得到展开式：

```
(/ (+ (- 5 2) 1) 6)
```

这一展开式的框架就是上面这个表达式在形参 x 和 y 要插入的地方留下空洞后的样子：

```
(/ (+         1)  )
```

定义了这两个概念，关于如何判断可捕捉的符号就可以表述为一个简洁的原则：

可捕捉 *(capturable)*：一个符号在某些宏展开里被认为是可捕捉的，如果它满足下面条件之一：(a) 它作为自由符号出现在宏展开式的框架里，或者 (b) 它被绑定到框架的一部分，而该框架中含有传递给宏的参数，这些参数或被绑定或被求值。

一些例子可以显示这一规则的影响。在最简单的情形下：

```
(defmacro cap1 ()
  '(+ x 1))
```

x 可被捕捉是因为它作为自由符号出现在框架里。这就是导致 gripe 中 bug 的原因。在这个宏里：

```
(defmacro cap2 (var)
  `(let ((x ...)
         (,var ...))
     ...))
```

x 可被捕捉是因为它被绑定在一个表达式里，而同时也有一个宏调用的参数被绑定了。(这就是 for 中出现的错误.) 同样对于下面两个宏：

```
(defmacro cap3 (var)
  `(let ((x ...))
     (let ((,var ...))
       ...)))

(defmacro cap4 (var)
  `(let ((,var ...))
     (let ((x ...))
       ...)))
```

x 在两个宏里都是可捕捉的。尽管如此，如果 x 的绑定和作为参数传递的变量没有一个上下文，在这个上下文中，两者是同时可见的，就像在这个宏里：

```
(defmacro safe1 (var)
  `(progn (let ((x 1))
            (print x))
          (let ((,var 1))
            (print ,var))))
```

那么 x 将不会被捕捉到。并非所有绑定在框架里的变量都是有风险的。尽管如此，如果宏调用的参数在一个由框架建立的绑定里被求值，

```
(defmacro cap5 (&body body)
  `(let ((x ...))
     ,@body))
```

那么如此绑定的变量就有被捕捉的风险：在 cap4 中，x 是可捕捉的。不过对于下面这种情况，

```
(defmacro safe2 (expr)
  `(let ((x ,expr))
     (cons x 1)))
```

x 是不可捕捉的，因为当传给 expr 的参数被求值时，x 的新绑定将不是可见的。同时，请注意我们只需关心那些框架变量的绑定。在这个宏里

```
(defmacro safe3 (var &body body)
  `(let ((,var ...))
     ,@body))
```

没有符号处于意料之外的被捕捉风险之中 (假设用户期望第一个参数将被绑定)。

现在让我们来看看 for 最初的定义，通过新的规则来发现可捕捉的符号：

```
(defmacro for ((var start stop) &body body)             ; wrong
  `(do ((,var ,start (1+ ,var))
        (limit ,stop))
       ((> ,var limit))
     ,@body))
```

现在可以看出 for 的这一定义可能遭受两种方式的捕捉：limit 可能会作为 for 的第一个参数被传递，就像在最初的例子里那样：

```
(for (limit 1 5)
  (princ limit))
```

但就算 limit 出现在循环体里也是危险的：

```
(let ((limit 0))
  (for (x 1 10)
    (incf limit x))
  limit)
```

以这种方式使用 for 的某人可能期望他自己的 limit 绑定就是在循环里增值的那个，然后整个表达式返回 55；事实上，只有那个由展开式框架生成的 limit 绑定将被增值：

```
(do ((x 1 (1+ x))
     (limit 10))
    ((> x limit))
  (incf limit x))
```

并且由于这个变量控制迭代过程，整个循环甚至将不会停止。

本节中介绍的这些规则是作为一个参考提供的，在实际编程中仅仅具有指导意义。它们甚至不是形式化定义的，更不能完全保证其正确性。捕捉是一个不能明确定义的问题，它依赖于你期望的行为。例如，在这样的表达式里，

```
(let ((x 1)) (list x))
```

我们不会将 x 在(list x) 被求值时指向了新变量视为一个错误。这正是 let 要做的事。检测捕捉的规则也是不精确的。你可以写出一个通过了这些测试的宏，却仍然可以遭受意料之外的捕捉。例如，

```
(defmacro pathological (&body body)                          ; wrong
  (let* ((syms (remove-if (complement #'symbolp)
                          (flatten body)))
         (var (nth (random (length syms))
                   syms)))
    `(let ((,var 99))
       ,@body)))
```

当这个宏被调用时，宏主体中的表达式就像是在一个 progn 中被求值——但是主体中有一个随机选出的变量将带有一个不同的值。这很明显是一个捕捉，但它通过了我们的测试，因为这个变量并没有出现在框架里。尽管如此，实践表明该规则绝大多数时候都是正确的：很少有人 (如果真有的话) 会想写出一个类似上面那个例子的宏。

易于被捕捉的：

```
(defmacro before (x y seq)
  `(let ((seq ,seq))
     (< (position ,x seq)
        (position ,y seq))))
```

一个正确的版本：

```
(defmacro before (x y seq)
  `(let ((xval ,x) (yval ,y) (seq ,seq))
     (< (position xval seq)
        (position yval seq))))
```

图 9.1: 用 `let` 避免捕捉。

9.4 通过更好的命名避免捕捉

前面两节将变量捕捉区分为两种类型：参数捕捉，在这里一个用在参数中的符号被宏框架建立的绑定捕捉到，以及自由符号捕捉，在这里一个宏展开式中的自由符号被宏展开所在地的一个绑定捕捉到。后一种情况经常可以通过给全局变量设置可区分的名字来处理。在 Common Lisp 中，传统上给全局变量一个以星号开始和结束的名字。例如，定义当前包的变量叫做 `*package*`。(这样的名字可以发音为 “star–package–star” 来强调它不是个普通变量。)

所以 `gripe` 的作者的的确确有责任把那些警告保存在一个名字类似 `*warnings*` 而非 `w` 的变量中。如果 `sample-ratio` 的作者非要用 `*warnings*` 来做函数参数，那他碰到的每个 bug 都是自找的，但如果他觉得用 `w` 作为参数的名字应该比较保险，就不应该再怪他了。

9.5 通过预先求值避免捕捉

有时参数捕捉可以通过在任何宏展开创建的绑定之外求值可能有危险的参数来轻易地修正。最简单的情形可以这样处理：让宏以一个 `let` 表达式开头。图 9.1 包含宏 `before` 的两个版本，该宏接受两个对象和一个序列，然后当且仅当第一个对象在序列中出现于第二个对象之前时返回真。[1] 第一个定义是不正确的。它开始的 `let` 确保了作为 `seq` 传递的形式 (form) 只被求值一次，但是它不能有效地避免下面这个问题：

```
> (before (progn (setq seq '(b a)) 'a)
          'b
```

[1] 该宏仅用在这个例子里，它既不该被实现成一个宏，也使用了低效率的算法。对于一个合适的定义，见 44 页。

```
          '(a b))
NIL
```

这相当于问“是否在 (a b) 中 a 在 b 的前面？”如果 before 是正确的，它将返回真。宏展开式显示了实际发生的事：对 < 的第一个参数的求值重新排列了那个将在第二个参数里被搜索的列表。

```
(let ((seq '(a b)))
  (< (position (progn (setq seq '(b a)) 'a)
               seq)
     (position 'b seq)))
```

要想避免这个问题，只需在一个巨大的 let 里求值所有参数就够了。这样图 9.1 中的第二个定义对于捕捉就是安全的了。

不幸的是，这种 let 技术只能在很有限的一类情况下才可行：

1. 所有具有捕捉风险的参数都只被求值一次，并且
2. 没有参数需要在宏框架建立的绑定范围内被求值。

这个规则将非常多的宏排除在外了。我们比较赞成的 for 宏就同时违反了这两个限制。尽管如此，我们可以使用一个该手法的变种，令诸如 for 这样的宏达到捕捉安全：将其主体形式 (body forms) 包装在一个位于任何局部创建的绑定之外的 λ–表达式里。

某些宏，包括那些用于迭代的，其产生的展开式在表达式出现在宏调用的位置将在一个新建的绑定中被求值。例如在 for 的定义中，循环体必须在一个由宏创建的 do 中进行求值。出现在循环体中的变量因此就会被 do 创建的变量绑定所捕捉。我们可以通过将主体包装在一个闭包里来保护主体中的变量不被捕捉，同时，在循环里，代替了原来直接插入这些表达式自身，只是简单地函数调用这个闭包。

图 9.2 给出了 for 的使用该技术的一个版本。由于闭包是 for 展开式做的第一件事，出现在宏体中的自由符号将全部指向宏调用环境中的变量。现在 do 通过闭包的参数跟宏体通信。闭包需要从 do 知道的全部就是当前迭代的数字，所以它只有一个参数，也就是宏调用中作为索引指定的那个符号。

这种将表达式包装进 lambda 的方法也不能包治百病。你可以用它来保护一个代码体，但闭包有时也起不到任何作用，例如，当存在同一个变量在同一个 let 或 do 里被绑定两次的风险时 (就像开始的那个有缺陷的 for 那样)。幸运的是，在这种情况下，通过重写 for 将其主体包装在一个闭包里，我们同时页消除了 do 为 var 参数建立绑定的需要。原先那个 for 中的 var 参数变成了闭包的参数并且在 do 里面可以被一个实际的符号 count 替换掉。所以这个 for 的新定义对于捕捉是完全免疫的，就像 9.3 节里的测试所显示的那样。

使用必要的缺点是它们可能会是低效的。我们可能正在引入又一次函数调用。潜在的更坏情况是，如果编译器没有给闭包动态作用域 (dynamic

捕捉脆弱的：

```
(defmacro for ((var start stop) &body body)
  `(do ((,var ,start (1+ ,var))
        (limit ,stop))
       ((> ,var limit))
     ,@body))
```

一个正确版本：

```
(defmacro for ((var start stop) &body body)
  `(do ((b #'(lambda (,var) ,@body))
        (count ,start (1+ count))
        (limit ,stop))
       ((> count limit))
     (funcall b count)))
```

图 9.2: 用闭包避免捕捉。

extent)，闭包所需的空间将不得不在运行期分配在堆里。[2]

9.6 通过生成符号 (gensym) 避免捕捉

有一种切实可行的方法来避免宏参数捕捉：将可捕捉的符号用生成符号 (gensym) 替换。在最初版本的 `for` 中，问题出现在当两个符号出乎意料地重名时。如果我们想要避免这种情形：宏框架里含有的符号也同时出现在了调用方代码里，我们也许会给宏定义里的符号取怪异的名字，寄希望以此来摆脱参数捕捉的魔爪：

```
(defmacro for ((var start stop) &body body)                    ; wrong
  `(do ((,var ,start (1+ ,var))
        (xsf2jsh ,stop))
       ((> ,var xsf2jsh))
     ,@body))
```

但是这并不解决问题。它并没有消除 bug，只是让出现的机会更小些。并且还有一个不那么小的问题——可以想象在同一个宏的嵌套实例中仍然会出现名字冲突。

我们需要某种方式来确保一个符号是唯一的。Common Lisp 函数 `gensym` 的存在目的就在于此。它返回一个符号，称为一个*生成符号*，可以保证不会和任何明确输入的或者由程序生成的符号 `eq` 相等。

Lisp 是如何确保这一点的呢？在 Common Lisp 里，每一个包都保持一个该包知道的所有符号的列表。(关于包 (package) 的介绍，见 247 页。) 一个出现

[2]译者注：dynamic extent 是一种 Lisp 编译器优化技术，详情请见 Common Lisp HyperSpec 的有关内容。

在此列表里的符号我们说它被约束 *(intern)* 在这个包里。对 gensym 的每一次调用将返回唯一的，未约束的符号。而由于每一个被 read 见到的符号都会被约束，所以没人可以输入任何跟一个生成符号相同的东西。这就是说，如果你开始一个表达式

```
(eq (gensym) ...
```

没有办法完成它并且让它返回真。

让 gensym 为你生成一个符号，这个办法类似于“选用怪名字”的方法，而且更进一步——gensym 将给你一个甚至不会出现在电话薄里的名字。当 Lisp 不得不显示一个生成符号时[3]，

```
> (gensym)
#:G47
```

它打印出来的东西基本上就相当于 “张三” 的 Lisp 等价物，一个为那种名字无关紧要的东西编造出来的毫无意义的名字。并且为了确保我们不会对此产生任何错觉，生成符号在显示上前缀一个星号– 冒号，这是一个特殊的读取宏 (read–macro)，其存在是为了让我们在试图第二次读取该生成符号时产生一个错误。

在 CLTL2 Common Lisp 里，生成符号的打印形式中的数字来自 *gensym-counter*，一个总是绑定到某个整数的全局变量。通过重设该计数器我们可以使两个生成符号打印输出相同

```
> (setq x (gensym))
#:G48
> (setq *gensym-counter* 48 y (gensym))
#:G48
> (eq x y)
NIL
```

但它们将不会是同一个东西。

图 9.3 含有一个使用生成符号的 for 的正确定义。现在就不存在 limit 跟传进宏的 form 里的符号产生碰撞了。它已经被替换成一个在现场生成的符号。在宏的每次展开时，limit 所在的位置都会被一个在展开期创建的唯一符号所取代。

for 的正确定义对于首次尝试编写来说还是很复杂的。完成后的代码，如同一个证明出来的定理，经常会有许多麻烦之处和错误。所以不要担心你可能会对一个宏写好几个版本。开始写类似 for 这样的宏时，你可以在先不考虑变量捕捉问题的情况下写出第一个版本，然后回过头来为那些可能卷入捕捉的符号制作生成符号。

[3]译者注：生成符号的实际符号命名规则是特定实现相关的，没有标准。我们也几乎不需要关心它。

易于被捕捉的：

```
(defmacro for ((var start stop) &body body)
  `(do ((,var ,start (1+ ,var))
        (limit ,stop))
       ((> ,var limit))
     ,@body))
```

一个正确的版本：

```
(defmacro for ((var start stop) &body body)
  (let ((gstop (gensym)))
    `(do ((,var ,start (1+ ,var))
          (,gstop ,stop))
         ((> ,var ,gstop))
       ,@body)))
```

图 9.3: 用生成符号避免捕捉。

9.7 通过包避免捕捉

从某种程度来讲，有可能通过将宏定义在它们自己的包里来避免捕捉。如果你创建一个 macros 包并且在那里定义 for，那么你甚至可以使用最开始给出的定义

```
(defmacro for ((var start stop) &body body)
  `(do ((,var ,start (1+ ,var))
        (limit ,stop))
       ((> ,var limit))
     ,@body))
```

然后安全地从其他任何包调用它。如果你从另一个包，比如说 mycode 里调用 for，那么甚至你用 limit 作为第一个参数，它将是 mycode::limit——跟 macros::limit 不同的符号，后者才是出现在宏框架中的那个。

尽管如此，包没有提供一个针对捕捉问题的非常通用的解决方案。首要问题是，宏是某些程序不可分割的组成部分，将它们从自己的包里分离出来会很不方便。另一方面，这种方法对于 macros 包里的其他代码不会提供任何捕捉保护。

9.8 其他名字空间里的捕捉

上述各节都将捕捉说成是一种仅影响变量的问题。尽管多数捕捉都是变量捕捉，问题同样也可以出现在 Common Lisp 的其他名字空间里。

函数也可能被局部绑定，并且函数绑定也要对无意中的捕捉负同等责任。例如，

```
> (defun fn (x) (+ x 1))
FN
> (defmacro mac (x) `(fn ,x))
MAC
> (mac 10)
11
> (labels ((fn (y) (- y 1)))
    (mac 10))
9
```

正如捕捉规则所预计的那样，自由出现在 mac 框架中的 fn 存在捕捉风险。当 fn 被局部重新绑定时，mac 返回了一个和正常时候不同的值。

对于这种情况需要做什么呢？当处于捕捉风险中的符号是一个内置函数或者宏的名字时，那么我们有理由什么也不做。CLTL2 (260 页) 说如果任何内置的名字被用作局部函数或宏绑定，“后果是未定义的。” 所以你的宏无论做了什么都没关系——任何重绑定了内置函数的人将会遇到比你这个宏更多的其他问题。

另一方面，你可以使用和保护变量名一样的方式来保护函数名免于被宏参数捕捉：通过使用生成符号作为宏框架局部定义的任何函数的名字。避免自由符号捕捉，就像上面那种情况的，更复杂一些。针对变量的自由符号捕捉保护方法是给它们可区别的全局名称：例如用 *warnings* 代替 w 。这个解决方案对函数是不切实际的，因为不存在区分全局函数名的惯例——大多数函数都是全局的。如果你担心一个宏可能会在一个局部重定义了它所需函数的环境下被调用，最佳的解决方案可能就是将你的代码放在一个不同的包里。

代码块名字 (block-name) 同样可以被捕捉，比如说那些被 go 和 throw 使用的标签 (tag)。当你的宏需要这些符号时，你应该使用生成符号，就像 89 页上 our-do 的定义里那样。

还需要注意的是像 do 这样的操作符隐式封装在一个名为 nil 的块里。这样在 do 里面的一个 return 或 return-from nil 将从 do 本身而非包含这个 do 的表达式里返回：

```
> (block nil
    (list 'a
          (do ((x 1 (1+ x)))
              (nil)
            (if (> x 5)
                (return-from nil x)
                (princ x)))))
12345
(A 6)
```

如果 do 没有创建一个名为 nil 的块，这个例子将只返回 6，而不是 (A 6)。

do 里面的隐式块不是问题，因为 do 的这种工作方式广为人知。尽管如此，如果你写一个展开到 do 的宏，它将捕捉 nil 这个块名称。在一个类似 for 的

宏里，return 或 return-from nil 将从 for 表达式而非封装这个 for 表达式的块中返回。

9.9 为何要庸人自扰?

前面举的例子中有些非常做作。看着它们，有人可能会说 “变量捕捉既然这么少见——为什么还要操心它呢？” 回答这个问题有两个方法。一个是用另一个问题反问道：要是你写得出没有 bug 的程序，为什么还要写有小 bug 的程序呢？

一个更长的答案是指出在现实应用程序中，对你代码的使用方式做任何假设都是危险的。任何 Lisp 程序都具备现在被称之为 “开放式架构” 的特征。如果你正在写的代码将被其他人使用，他们可能以一种你预想不到的方式来使用它。而且你要担心的不光是人。程序也写程序。可能没人会写这样的代码

```
(before (progn (setq seq '(b a)) 'a)
        'b
        '(a b))
```

但是程序生成的代码经常看起来就像这样。甚至就算单个的宏能够生成简单合理的展开式，一旦你开始把宏嵌套着调用，展开式就可能变成巨大的，而且看上去没人能写得出来的程序。在这个前提下，就有必要去预防那些可能使你的宏不正确地展开的情况，就算这种情况像是有意设计出来的。

最后，避免变量捕捉不管怎么说并不是非常困难。它很快会成为你的第二直觉。Common Lisp 中经典的 defmacro 好比厨子手中的菜刀：美妙的想法看上去会有些危险，但是这件利器一到了专家那里，就如入庖丁之手，游刃有余。

第 10 章

其他的宏陷阱

编写宏需要额外小心。一个函数被隔离在它自己的词法世界中，但是一个宏，因为它要被展开进调用方的代码，除非仔细编写否则将会给用户带来不愉快的惊喜。第 9 章解释了变量捕捉，这类惊喜中最大的一种。本章将讨论在编写宏时需要避免的另外四个问题。

10.1 求值的数量

前一章里出现了一些不正确版本的 `for`。图 10.1 给出了另外两个，顺带一个正确的版本用于比较。

尽管第二个 `for` 并不是那么容易发生变量捕捉，但是它还是有个 bug。它将生成一个展开式其中作为 `stop` 传递的形式 (form) 在每次迭代时都会被求值。在最好的情况下，这类宏只是低效率，重复做一些它本来可以只做一次的事。如果 `stop` 有副作用，那么宏事实上就会产生不正确的结果。例如，这个循环将不会停止，因为目标在每次迭代时都会倒退：

```
> (let ((x 2))
    (for (i 1 (incf x))
      (princ i)))
```

在编写类似 `for` 的宏时，必须记住一个宏的参数是形式 (form)，不是值。取决于它们出现在表达式的位置，它们可能会被求值不止一次。在这种情况下，解决方案是绑定一个变量到到 `stop` 形式返回的值上，然后在循环过程中引用这个变量。

除非进行迭代是有意为之，否则宏应该确保表达式被求值的次数刚好等于它们出现在宏调用里的次数。很明显，在有的情况下，这一规则并不适用：如果参数总是被求值的话，Common Lisp 的 `or` 就不那么有用了 (那就会沦落为一个 `Pascal` 的 `or` 了)。但是在这种情况下用户知道他们期望有多少次求值。对于第二个版本的 `for` 来说就不是这样了：用户没理由假设那个 `stop` 形式会被求值不止一次，而且在事实上也没有理由说明应该这样做。一个写成第二个版本的 `for` 那样的宏十有八九是因为写错了。

对基于 `setf` 之上的宏来说，无意识的多重求值尤其难以处理。Common Lisp 提供了几个实用工具使编写这样的宏更容易些。具体的问题，以及解决方

一个正确版本：

```
(defmacro for ((var start stop) &body body)
  (let ((gstop (gensym)))
    `(do ((,var ,start (1+ ,var))
          (,gstop ,stop))
         ((> ,var ,gstop))
       ,@body)))
```

导致多重求值：

```
(defmacro for ((var start stop) &body body)
  `(do ((,var ,start (1+ ,var)))
       ((> ,var ,stop))
     ,@body))
```

错误的求值顺序：

```
(defmacro for ((var start stop) &body body)
  (let ((gstop (gensym)))
    `(do ((,gstop ,stop)
          (,var ,start (1+ ,var)))
         ((> ,var ,gstop))
       ,@body)))
```

图 10.1: 控制参数求值。

案，将在第 12 章里进行讨论。

10.2 求值的顺序

表达式求值的顺序，虽然不像它们的求值次数那样重要，但有时也会成为问题。在 Common Lisp 函数调用中，参数是从左到右求值的：

```
> (setq x 10)
10
> (+ (setq x 3) x)
6
```

对于宏来说最好也是一样。宏通常应该确保表达式按它们出现在宏调用中的顺序进行求值。

在图 10.1 中，第三个版本的 `for` 里同样含有一个细微的 bug。参数 `stop` 将会在 `start` 前被求值，即使它们以相反的顺序出现在宏调用里：

```
> (let ((x 1))
    (for (i x (setq x 13))
      (princ i)))
13
NIL
```

这个宏给人一种不愉快的时光倒流的感觉。[1] 对 `stop` 形式的求值影响了 `start` 形式的返回值，即使 `start` 形式在字面上先出现。

`for` 的正确版本会确保其参数以它们出现的顺序被求值：

```
> (let ((x 1))
    (for (i x (setq x 13))
      (princ i)))
12345678910111213
NIL
```

现在在 `stop` 形式里设置 `x` 的值对前一个参数的返回值没有影响。

尽管前面这个例子是杜撰出来的，这类问题在某些情况下还是会真实发生的，而且这样的一个 bug 将极难被发现。或许很少有人会写出这种宏里面一个参数的求值会影响到另一个参数返回值的代码，但人们可能在无意之中偶然写出了这样的代码。有意这样使用时必须正确工作，同时一个实用工具也一定不能掩盖 bug。如果有人写出类似前面示例的代码，它很可能是误写成的，但 `for` 的正确版本将使错误更容易检测出来。

[1] 原文是：This macro gives a disconcerting impression of going back in time.

10.3 非函数型展开器

Lisp 期望它所生成的宏展开式代码都是纯函数型的，就像第 3 章里描述的那样。展开器代码除了作为参数传给它的形式(form) 之外不应该有其他依赖，并且它影响外界的唯一渠道只能是通过其返回值。

正如 CLTL2 (685 页) 所述，在编译代码中的宏调用可以保证不会在运行期重新展开。另一方面，Common Lisp 并不保证一个宏调用将在何时被展开，或者被展开多少次。如果一个宏的展开式随上述两种情况而变的话，那么应该将其视为错误。例如，假设我们想要统计某些宏被使用的次数。我们不能简单地在源代码文件里做一次搜索，因为宏可能会在由程序生成的代码里被调用。我们可能因此想把宏定义成下面这样：

```
(defmacro nil! (x)                                          ; wrong
  (incf *nil!s*)
  `(setf ,x nil))
```

通过这一定义，全局的 `*nils*` 的值将在每次对 `nil` 的调用被展开时增长一次。尽管如此，如果我们期望这个变量的值可以告诉我们 `nil` 被调用了多少次的话就错了。一个宏可以，并且经常会被展开不只一次。例如，一个对你代码进行变换的预处理器在它决定是否变换代码之前可能不得不展开表达式中的宏调用。

作为一项通用规则，展开器代码除了其参数之外不应依赖于其他任何东西。所以任何宏，比如说通过字符串来构造展开式的那种，应当小心不要对宏展开时所在的包做任何假设。下面这个简明但相当有代表性的例子，

```
(defmacro string-call (opstring &rest args)                 ; wrong
  `(,(intern opstring) ,@args))
```

定义了一个宏，它接受一个操作符的打印名称，然后将其展开成一个对该操作符的调用：

```
> (defun our+ (x y) (+ x y))
OUR+
> (string-call "OUR+" 2 3)
5
```

对 `intern` 的调用接受一个字符串然后返回对应的符号。尽管如此，如果我们省略了可选的包参数，它将在当前包里寻找符号。该展开式将因此依赖于展开式生成时所在的包，并且除非 `our+` 在那个包里可见，否则展开式将是一个对未知符号的调用。

Miller 和 Benson 的 *Lisp Style and Design* 一书中提到了一个展开式代码中的副作用带来的问题的尤其丑陋的例子。CLTL2 (78 页) 提到，在 Common Lisp 中，绑定在 `&rest` 形参上的列表并不保证是新生成的。它们可能共享了程

序其他地方的列表结构。这样带来的结果是，你不应该破坏性地修改 `&rest` 形参，因为你不知道你将会改掉其他什么东西。

这种可能性对于函数和宏都有影响。对于函数来说，问题出现在使用 `apply` 时。在一个 Common Lisp 的有效实现中接下来的事情将发生。假设我们定义一个函数 `et-al`，它返回一个其参数结尾追加了 `et al` 的列表：

```
(defun et-al (&rest args)
  (nconc args (list 'et 'al)))
```

如果我们正常调用这个函数，它看起来工作正常：

```
> (et-al 'smith 'jones)
(SMITH JONES ET AL)
```

尽管如此，如果我们通过 `apply` 调用它，就会替换到已有的数据结构：

```
> (setq greats '(leonardo michelangelo))
(LEONARDO MICHELANGELO)
> (apply #'et-al greats)
(LEONARDO MICHELANGELO ET AL)
> greats
(LEONARDO MICHELANGELO ET AL)
```

至少，一个 Common Lisp 的有效实现可能会这样做，尽管到目前为止没有一个看起来是这样的。

对于宏来说，就更加危险了。一个替换了 `&rest` 形参的宏可以因此替换掉整个宏调用。这就是说，最终你可能写出一个难以察觉的自我重写的程序。这种危险也更加真实——它实际发生在当前的实现里。如果我们定义一个宏，它将某些东西 nconc 到它的 `&rest` 参数里[2]

```
(defmacro echo (&rest args)
  `',(nconc args (list 'amen)))
```

然后定义一个函数来调用它：

```
(defun foo () (echo x))
```

在一种广泛使用的 Common Lisp 里会发生下面的事[3]：

```
> (foo)
(X AMEN AMEN)
> (foo)
(X AMEN AMEN AMEN)
```

不只是 `foo` 返回了错误的结果，它甚至每次返回不同的结果，因为每一次宏展开都替换了 `foo` 的定义。

[2] `` `',(foo) `` 等价于 `` `(quote ,(foo)) ``.

[3] 译者注：实际测试了一下，不同的实现行为很不相同，而且上述定义在同一种实现里解释和编译时的行为也可能不同。

这个例子同时也阐述了之前提到的一个给定宏可能被多次展开的观点。在这种特殊的实现里，对 foo 的第一次调用返回了含有两个 amen 的列表。出于某种原因该实现在 foo 被定义时就做了一次宏展开，然后接下来每次调用时再展开一次。

将 foo 定义成这样会更安全一些：

```
(defmacro echo (&rest args)
  `'(,@args amen))
```

因为一个逗号-at等价于一个 append 而非 nconc。重定义了这个宏之后，foo 也需要重新定义一下，就算它没有编译也是一样，因为 echo 的前一个版本导致它把自己重写了。

在宏里，受这种危险影响的不只是 &rest 参数。任何宏参数只要是列表就应该单独对待。如果我们定义了一个会修改其参数的宏，以及一个调用该宏的函数，

```
(defmacro crazy (expr) (nconc expr (list t)))

(defun foo () (crazy (list)))
```

然后主调函数的源代码就会被修改了，在一种实现里我们首次调用时得到这个：

```
> (foo)
(T T)
```

这件事在编译的代码里也和解释代码一样会发生。

结论是，不要试图通过破坏性修改参数列表结构来避免构造 consing。这样得到的程序就算可以工作也将是不可移植的。如果你真想在接受变长参数的函数中避免 consing，一种解决方案是使用宏，由此将 consing 切换到编译期。对于宏的这种应用，见第 13 章。

应该避免在宏展开器返回的表达式上做破坏性操作，如果这些表达式里包括了引用的列表。就其本身而言这不只是一个对于宏的限制，而是第 3.3 节中概述的原则的一个实例。

10.4 递归

有时会很自然地将一个函数定义为递归的。有些函数天生就是递归的，就像这个：

```
(defun our-length (x)
  (if (null x)
      0
      (1+ (our-length (cdr x)))))
```

这个可以工作：

```
(defun ntha (n lst)
  (if (= n 0)
      (car lst)
      (ntha (- n 1) (cdr lst))))
```

这个不能编译：

```
(defmacro nthb (n lst)
  `(if (= ,n 0)
       (car ,lst)
       (nthb (- ,n 1) (cdr ,lst))))
```

图 10.2: 对递归函数的错误类比

这一定义从某种程度来讲比它等价的迭代形式看起来更自然一些 (尽管可能也更慢一些)：

```
(defun our-length (x)
  (do ((len 0 (1+ len))
       (y x (cdr y)))
      ((null y) len)))
```

一个既不递归，也不属于某个多重递归函数集合的函数，可以通过第 7.10 节描述的简单技术被转换为一个宏。尽管如此，仅是插入反引用和逗号对递归函数是没有用的。让我们以内置的 `nth` 为例。(为简单起见，我们这个版本的 `nth` 将不做错误检查。) 图 10.2 给出了一个将 `nth` 定义成宏的错误尝试。表面上看 `nthb` 似乎和 `ntha` 等价，但是一个包含对 `nthb` 调用的程序将不能编译，因为对该调用的展开过程无法终止。

一般而言，宏里含有对另一个宏的引用是可以的，只要展开过程可以在某个地方终止。`nthb` 的麻烦在于每次展开都含有一个对其本身的引用。函数版本，`ntha`，之所以会终止因为它在 `n` 的值上递归，这个值在每次递归中减小。但是宏展开式只能访问到形式 (form)，而不是它们的值。当编译器试图宏展开，比如说，`(nthb x y)` 时，第一次展开将得到

```
(if (= x 0)
    (car y)
    (nthb (- x 1) (cdr y)))
```

然后又会被展开成

```
(if (= x 0)
    (car y)
    (if (= (- x 1) 0)
        (car (cdr y))
        (nthb (- (- x 1) 1) (cdr (cdr y)))))
```

```
(defmacro nthd (n lst)
  `(nth-fn ,n ,lst))

(defun nth-fn (n lst)
  (if (= n 0)
      (car lst)
      (nth-fn (- n 1) (cdr lst))))

(defmacro nthe (n lst)
  `(labels ((nth-fn (n lst)
              (if (= n 0)
                  (car lst)
                  (nth-fn (- n 1) (cdr lst)))))
     (nth-fn ,n ,lst)))
```

图 10.3: 修复问题的两种方式

如此这般地进入无限循环。一个宏展开成对自身的调用是可以的，但不是这么用的。

像 nthb 这样的递归宏其真正危险之处在于它们通常在解释器里工作正常。然后当你最终将程序跑起来，然后试图编译它的时候，它甚至不能编译通过。不仅如此，通常也不会有提示，指出问题来自一个递归的宏；编译器只会陷入无限循环，让你来找出究竟哪里搞错了。

在这个案例中，ntha 是尾递归的。一个尾递归函数可以轻易转换成一个与之等价的迭代形式，然后用作宏的模型。一个像 nthb 的宏可以写成

```
(defmacro nthc (n lst)
  `(do ((n2 ,n (1- n2))
        (lst2 ,lst (cdr lst2)))
       ((= n2 0) (car lst2))))
```

所以从理论上讲，把一个递归函数改造成宏也并非不可能。但是，要转换更复杂的递归函数可能会比较困难，甚至不可能。

这取决于你要宏做什么，有时候你可能会发现改成用宏和函数的组合就够了。图 10.3 给出了两种方式来生成一个看起来递归的宏。第一种策略就是括在 nthd 里面简单地让宏展开成一个对递归函数的调用。举个例子，如果你使用宏的目的，仅仅是希望帮助用户避免引用参数的麻烦，那么这种方法就可以胜任了。

如果你需要一个宏是因为你想要将其展开式嵌入到宏调用的词法环境中，那么你可能更喜欢采用 nthe 这个示例中的方案。内置的 labels 特殊形式 (见 2.7 节) 创建了一个局部函数定义。相比 nthd[4] 会调用全局定义的函数 nth-fn，nthe 的每次展开将在它里面拥有一个它自己的该函数的版本。

[4]译者注：这里改掉一个原书错误，nthc 应该是 nthe。

```
(defmacro ora (&rest args)
  (or-expand args))

(defun or-expand (args)
  (if (null args)
      nil
      (let ((sym (gensym)))
        `(let ((,sym ,(car args)))
           (if ,sym
               ,sym
               ,(or-expand (cdr args)))))))

(defmacro orb (&rest args)
  (if (null args)
      nil
      (let ((sym (gensym)))
        `(let ((,sym ,(car args)))
           (if ,sym
               ,sym
               (orb ,@(cdr args)))))))
```

图 10.4: 递归的展开函数

尽管你不能将递归函数直接转化成一个宏，你却可以写出一个宏，让它的展开式是递归生成的。一个宏的展开函数就是正常的 Lisp 函数，并且当然是可以递归的。例如，如果我们想自己定义内置 `or`，我们将会用到一个递归展开的函数。

图 10.4 给出了为 `or` 定义递归展开函数的两种方式。宏 `ora` 调用递归函数 `or-expand` 来生成展开式。这个宏将可以工作，并且因此将等价于 `orb`。尽管 `orb` 是递归的，但它是在宏的参数个数上做递归 (这可以在宏展开期得到)，而不依赖于它们的值 (这在宏展开期是得不到的)。可能看起来这个展开式包含有到 `orb` 自身的引用，但其实由一次宏展开步骤生成的对 `orb` 的调用将在下一次展开式替换成一个 `let`，最后表达式里得到的只是一个 `let` 的嵌套栈；`(orb x y)` 展开成的代码等价于：

```
(let ((g2 x))
  (if g2
      g2
      (let ((g3 y))
        (if g3 g3 nil))))
```

事实上，`ora` 和 `orb` 是等价的，具体使用哪种风格完全看个人喜好。

第 11 章

经典宏

本章介绍如何定义最常用的几种宏。它们可以大致归为三类——带有一定重叠。第一组宏创建上下文 (context)。任何令其参数在一个新的上下文环境里求值的操作符都必须被定义成宏。本章的前两节描述两种基本类型的上下文，并且展示如何定义它们。

接下来的三节描述带有条件和重复求值的宏。一个其参数被求值少于一次或者多于一次的操作符，也同样必须被定义成宏。在做条件求值和重复求值的操作符之间没有明显区别：本章某些示例两件事都做 (as well as binding)。最后一节解释了条件求值和重复求值之间的另一种相似性：在某些场合，它们都可以用函数来完成。

11.1 创建上下文

这里的上下文有两层意思。一类上下文是词法环境。特殊形式 `let` 创建一个新的词法环境；`let` 主体中的表达式将在一个可能包含新变量的环境中被求值。如果 `x` 在 toplevel 下被设置为 `a`，那么

```
(let ((x 'b)) (list x))
```

将必定返回 `(b)`，因为对 `list` 的调用被放在一个新环境里，它包含一个新的 `x`，其值为 `b`。

一个带有表达式体的操作符通常必须被定义成一个宏。除了类似 `prog1` 和 `progn` 这种情况以外，这类操作符的目地通常都将是导致其主体在某种新上下文环境中被求值。一个宏将被用来在主体周围包装上下文创建的代码，即使这个上下文环境里不包含新的词法变量。

图 11.1 显示了如何通过 `lambda` 将 `let` 定义为一个宏。一个 `our-let` 展开到一个函数应用——

```
(our-let ((x 1) (y 2))
  (+ x y))
```

展开成

```
((lambda (x y) (+ x y)) 1 2)
```

```
(defmacro our-let (binds &body body)
  `((lambda ,(mapcar #'(lambda (x)
                         (if (consp x) (car x) x))
                     binds)
      ,@body)
    ,@(mapcar #'(lambda (x)
                  (if (consp x) (cadr x) nil))
              binds)))
```

图 11.1: let 的宏实现

```
(defmacro when-bind ((var expr) &body body)
  `(let ((,var ,expr))
     (when ,var
       ,@body)))

(defmacro when-bind* (binds &body body)
  (if (null binds)
      `(progn ,@body)
      `(let (,(car binds))
         (if ,(caar binds)
             (when-bind* ,(cdr binds) ,@body)))))

(defmacro with-gensyms (syms &body body)
  `(let ,(mapcar #'(lambda (s)
                     `(,s (gensym)))
                 syms)
     ,@body))
```

图 11.2: 绑定变量的宏

图 11.2 包含三个新的创建词法环境的宏。第 7.5 节使用了 when-bind 作为参数列表解构的示例，所以这个宏已经在第 85 页描述过了。更一般的 when-bind* 接受一个由成对的 (*symbol expression*) 形式所组成的列表——就和 let 的第一个参数的形式相同。如果任何 *expression* 返回 nil，那么整个 when-bind* 表达式就返回 nil。同样，它的主体在每个符号像在 let* 里那样被绑定的情况下求值：

```
> (when-bind* ((x (find-if #'consp '(a (1 2) b)))
               (y (find-if #'oddp x)))
    (+ y 10))
11
```

最后，宏 with-gensyms 本身就是用来编写宏的。许多宏定义以符号生成开始，有时需要生成相当数量。宏 with-redraw (第 104 页) 不得不创建五个：

```
(defmacro with-redraw ((var objs) &body body)
  (let ((gob (gensym))
        (x0 (gensym)) (y0 (gensym))
        (x1 (gensym)) (y1 (gensym)))
    ...))
```

这样的定义可以通过使用 with-gensyms 得以简化，后者将整个变量列表绑定到符号生成上。借助新宏我们只需写成：

```
(defmacro with-redraw ((var objs) &body body)
  (with-gensyms (gob x0 y0 x1 y1)
    ...))
```

这个新宏将被广泛用于后续章节。

如果我们需要绑定某些变量，然后取决于某些条件，求值一族表达式中的一个，我们只需在 let 里使用一个条件判断：

```
(let ((sun-place 'park) (rain-place 'library))
  (if (sunny)
      (visit sun-place)
      (visit rain-place)))
```

不幸的是，对于相反的情形没有简便的写法，就是说我们总是想要求值相同的代码，但在绑定的那里必须随某些条件而变。

图 11.3 包含一个处理类似情况的宏。就像其名字所代表的，condlet 是 cond 和 let 的杂交后代。它接受一个绑定语句的列表，接着是一个代码主体。每个绑定语句都以一个测试表达式开始；代码主体将在第一个测试表达式为真的绑定语句所构造的绑定环境中被求值。出现在某些语句却没出现另一些语句里的变量将被绑定到 nil，如果最后被选中的语句里没有绑定它们的话：

```
> (condlet (((= 1 2) (x (princ 'a)) (y (princ 'b)))
            ((= 1 1) (y (princ 'c)) (x (princ 'd)))
```

```
(defmacro condlet (clauses &body)
  (let ((bodfn (gensym))
        (vars (mapcar #'(lambda (v) (cons v (gensym)))
                      (remove-duplicates
                        (mapcar #'car
                                (mappend #'cdr clauses))))))
    `(labels ((,bodfn ,(mapcar #'car vars)
                ,@body))
       (cond ,@(mapcar #'(lambda (cl)
                           (condlet-clause vars cl bodfn))
                       clauses)))))

(defun condlet-clause (vars cl bodfn)
  `(,(car cl) (let ,(mapcar #'cdr vars)
                (let ,(condlet-binds vars cl)
                  (,bodfn ,@(mapcar #'cdr vars))))))

(defun condlet-binds (vars cl)
  (mapcar #'(lambda (bindform)
              (if (consp bindform)
                  (cons (cdr (assoc (car bindform) vars))
                        (cdr bindform))))
          (cdr cl)))
```

图 11.3: `cond` 与 `let` 的组合

```
            (t       (x (princ 'e)) (z (princ 'f))))
    (list x y z))
CD
(D C NIL)
```

condlet 的定义可以被理解成是 our-let 定义的一般化。后者将其主体做成一个函数，然后被应用到初值 (initial value) 形式的求值结果上。condlet 展开后的代码用 labels 定义了一个本地函数，然后一个 cond 语句来决定哪一组初值将被求值并传给该函数。

注意到展开器使用 mappend 代替 mapcan 来从绑定语句中解出变量名。这是因为 mapcan 是破坏性的，根据第 10.3 节里的警告，它比较危险，会修改参数列表结构。

11.2 with- 宏

除了词法环境以外还有另一种上下文。广义上来讲，上下文是世界的状态，包括特殊变量的值，数据结构的内容，以及 Lisp 之外事物的状态。构造这种类型上下文的操作符也必须被定义成宏，除非它们的代码主体要被打包进闭包里。

上下文构造宏的名称经常以 with- 开始。这类宏里最常用的一个恐怕就是 with-open-file 了。它的主体和一个根据用户给定变量新打开的文件一起求值：

```
(with-open-file (s "dump" :direction :output)
  (princ 99 s))
```

该表达式求值结束以后文件 "dump" 将自动被关闭，它的内容将是两个字符 "99"。

这个操作符很明显要被定义为一个宏，因为它绑定了 s。尽管如此，只要是能让形式在一个新的上下文中进行求值的操作符都必须被定义为一个宏。ignore-errors 宏，CLTL2 新增的，使其参数就像在一个 progn 里求值一样。如果一个错误在任何点上发生了，整个 ignore-errors 形式将简单地返回 nil。(这可能是有用的，例如，当读取用户输入时。) 尽管 ignore-errors 没有创建任何变量，它仍然必须被定义为一个宏，因为它的参数在一个新的上下文里被求值了。

一般而言，创建上下文的宏将被展开成一个代码块；附加的表达式可能被放在主体之前，之后，或者前后都有。如果是出现在主体之后，其目的可能是为了保持系统的一致状态——去清理某些东西。例如，with-open-file 必须关闭它打开的文件。在这种情形里，典型的方法是将上下文创建的宏展开进一个 unwind-protect 里。

unwind-protect 的目的是确保特定表达式被求值，甚至当执行被中断时。它接受一个或更多参数，这些参数按顺序执行。如果一切正常的话它将返回第一个参数的值，就像 prog1。区别在于，其余的参数即使当错误或者抛出的异常中断了第一个参数的执行时也会被执行。

```
> (setq x 'a)
A
> (unwind-protect
    (progn (princ "What error?")
           (error "This error."))
    (setq x 'b))
What error?
>>Error: This error.
```

该 unwind-protect 形式整体上产生了一个错误。尽管如此，在返回到 toplevel 之后，我们注意到它的第二个参数仍然被求值了：

```
> x
B
```

因为 with-open-file 展开到一个 unwind-protect 中，它打开的文件通常都将会被关闭，即使在求值其表达式体有错误发生。

上下文创建宏多数是为特定应用而写。作为示例，假设我们在写一个处理多重、远程数据库的程序。程序每次跟一个数据库通信，这个数据库由全局变量 *db* 指定。在使用一个数据库之前，我们必须对其锁定，以确保没有其他程序能同时使用它。当我们完成操作后需要对其解锁。如果我们想要对数据库 db 查询 q 的值，我们可能会像这样说：

```
(let ((temp *db*))
  (setq *db* db)
  (lock *db*)
  (prog1 (eval-query q)
         (release *db*)
         (setq *db* temp)))
```

通过宏我们可以将所有这些例行手续隐藏起来。图 11.4 定义了一个允许我们在一个更高的抽象层面上管理数据库的宏。使用 with-db，我们只需说：

```
(with-db db
  (eval-query q))
```

调用 with-db 也会更安全，因为它展开到一个 unwind-protect 而不是一个简单的 prog1。

图 11.4 中的两个定义阐述了编写此类宏的两种可能方式。第一种是个纯粹的宏，第二种是函数与宏的组合。当 with- 宏变得愈发复杂时，第二种方法更具实践意义。

完全使用宏：

```
(defmacro with-db (db &body body)
  (let ((temp (gensym)))
    `(let ((,temp *db*))
       (unwind-protect
         (progn
           (setq *db* ,db)
           (lock *db*)
           ,@body)
         (progn
           (release *db*)
           (setq *db* ,temp))))))
```

宏与函数的组合：

```
(defmacro with-db (db &body body)
  (let ((gbod (gensym)))
    `(let ((,gbod #'(lambda () ,@body)))
       (declare (dynamic-extent ,gbod))
       (with-db-fn *db* ,db ,gbod))))

(defun with-db-fn (old-db new-db body)
  (unwind-protect
    (progn
      (setq *db* new-db)
      (lock *db*)
      (funcall body))
    (progn
      (release *db*)
      (setq *db* old-db))))
```

图 11.4: 一个典型的 `with-` 宏

```
(defmacro if3 (test t-case nil-case ?-case)
  `(case ,test
     ((nil) ,nil-case)
     (?     ,?-case)
     (t     ,t-case)))

(defmacro nif (expr pos zero neg)
  (let ((g (gensym)))
    `(let ((,g ,expr))
       (cond ((plusp ,g) ,pos)
             ((zerop ,g) ,zero)
             (t ,neg)))))
```

图 11.5: 做条件求值的宏

在 CLTL2 Common Lisp 中，`dynamic-extent` 声明允许含有主体的闭包被更有效率地分配空间 (在 CLTL1 实现中，它将被忽略)。我们只有在 `with-db-fn` 调用期间才需要这个闭包，于是该声明说，允许编译器从栈上为其分配空间。这些空间将在 `let` 表达式退出后自动回收，而不是在后面被垃圾收集器回收。

11.3 条件求值

有时我们需要让宏调用中的某个参数仅在特定条件下才被求值。这超出了函数的能力，因为函数总是会求值其全部参数。诸如 `if`，`and` 和 `cond` 这样的内置操作符能够在保护其某些参数免于被求值，除非其他参数返回特定值。例如，在这个表达式中

```
(if t
    'phew
    (/ x 0))
```

第三个参数如果被求值的话将导致一个除零错误。但由于只有前两个参数将被求值，`if` 从整体上将总是安全地返回 `phew`。

我们可以通过编写宏，将调用展开到已有的操作符上来创造这类新操作符。图 11.5 中的两个宏是许多可能的 `if` 变种中的两个。`if3` 的定义显示了我们应如何定义一个三值逻辑的条件选择。代替了将 `nil` 视为假然后将其他所有东西都视为真，这个宏考虑到了三种真值类型：真，假，以及*不确定*，表示成 `?`。它可能用于下面这个关于五岁小孩的描述：

```
(while (not sick)
  (if3 (cake-permitted)
       (eat-cake)
       (throw 'tantrum nil)
       (plead-insistently)))
```

```
(defmacro in (obj &rest choices)
  (let ((insym (gensym)))
    `(let ((,insym ,obj))
       (or ,@(mapcar #'(lambda (c) `(eql ,insym ,c))
                     choices)))))

(defmacro inq (obj &rest args)
  `(in ,obj ,@(mapcar #'(lambda (a)
                          `',a)
                      args)))

(defmacro in-if (fn &rest choices)
  (let ((fnsym (gensym)))
    `(let ((,fnsym ,fn))
       (or ,@(mapcar #'(lambda (c)
                         `(funcall ,fnsym ,c))
                     choices)))))

(defmacro >case (expr &rest clauses)
  (let ((g (gensym)))
    `(let ((,g ,expr))
       (cond ,@(mapcar #'(lambda (cl) (>casex g cl))
                       clauses)))))

(defmacro >casex (g cl)
  (let ((key (car cl)) (rest (cdr cl)))
    (cond ((consp key) `((in ,g ,@key) ,@rest))
          ((inq key t otherwise) `(t ,@rest))
          (t (error "bad >case clause")))))
```

图 11.6: 使用条件求值的宏

这个新的条件选择展开成一个 case。(那个 nil 键不得不封装在一个列表里是因为一个单独的 nil 键会有歧义。) 最后三个参数里仅有一个将被求值，取决于第一个参数的值。

nif 的名字代表 “数值型 if”。该宏的另一种实现出现在 78 页上。它接受数值表达式作为第一个参数，然后根据它的符号来求值接下来三个参数中的一个。

```
> (mapcar #'(lambda (x)
              (nif x 'p 'z 'n))
          '(0 1 -1))
(Z P N)
```

图 11.6 包含了更多的几个使用了条件求值的宏。宏 in 用来高效地测试集合的成员关系。当你想要测试一个对象是否为某备选对象集合之一时，你可以将这个查询表达式一个析取：

```
(let ((x (foo)))
  (or (eql x (bar)) (eql x (baz))))
```

或者你也可以用集合的成员关系来表达：

```
(member (foo) (list (bar) (baz)))
```

后者更抽象但也更低效。该 member 表达式在从两个地方引起了毫无必要的开销。它需要构造点对，因为它必须将所有备选对象连结成一个列表以便 member 进行查找。并且为了把备选项做成列表形式它们全都要被求值，尽管某些值可能根本不需要。如果 (foo) 的值等于 (bar) 的值，那么就不需要求值 (baz) 了。不管它在建模上多么抽象，使用 member 都不是个好方法。我们可以通过宏来得到更有效率的抽象：in 把 member 的抽象与 or 的效率结合在了一起。等价的 in 表达式

```
(in (foo) (bar) (baz))
```

跟 member 表达式的形态相同，但却可以展开成

```
(let ((#:g25 (foo)))
  (or (eql #:g25 (bar))
      (eql #:g25 (baz))))
```

情况经常是这样，当我们需要在简洁和高效两种习惯用法之间两者择一时，我们遵循中庸之道，方法是编写宏将前者变换成为后者。

发音为 “in queue” 的 inq 是 in 的引用变种，类似 setq 之于 set。表达式

```
(inq operator + - *)
```

展开成

```
(in operator '+ '- '*)
```

和 member 的缺省行为一样， in 和 inq 使用 eql 来测试等价性。当你想要使用一些其他的测试——或者任何其他的一元函数——那么可以使用更一般的 in-if。in-if 对于 same 好比是 in 对于 member。表达式

```
(member x (list a b) :test #'equal)
```

也可以写作

```
(in-if #'(lambda (y) (equal x y)) a b)
```

而

```
(some #'oddp (list a b))
```

就变成

```
(in-if #'oddp a b)
```

将 cond 和 in 组合使用的话，我们还能定义出 case 的一个有用的变种。Common Lisp 的 case 宏假定它的键都是常量。有时我们可能想要一个 case 的行为，但其中的键可以被求值。对于这类情形我们定义了 >case，就跟 case 的行为一样，除了它每个子句里的键在比较之前先被求值。(名字中的 > 旨在暗示通常用来展示求值过程的那个箭头符号。) 因为 >case 使用了 in，只有它需要的那个键才会被求值。

由于键可以是 Lisp 表达式，没有办法指出 (x y) 到底是一个函数调用还是两个键组成的列表。为了避免这种二义性，键 (除了 t 和 otherwise) 必须总是放在列表里给出，哪怕是只有一个。在 case 表达式里，由于会产生歧义nil 不能作为子句的 car 出现。在一个 >case 表达式里，nil 作为子句的 car 就不再有歧义了，但它的含义是该子句的其余部分将不会被求值。

为清晰起见，生成每一个 >case 子句展开式的代码被定义在一个单独的函数 >casex 里。注意到 >casex 本身还用到了 inq。

11.4 迭代

有时函数的麻烦之处并非在于它们的参数总是被求值，而在于它们只能被求值一次。因为一个函数的每个参数都将被求值刚好一次，如果我们想要定义一个操作符来接受某些表达式体并且迭代于其上，我们将不得不把它定义成一个宏。

最简单的例子就是一个能够按顺序永远求值其参数的宏：

```
(defmacro forever (&body body)
  `(do ()
       (nil)
     ,@body))
```

这不过是当你不给它任何循环关键字时 loop 宏该做的事[1]。你可能认为无限循环没有什么用处(或者说没什么大的用处)。但当它跟 block 和 return-from 组合起来使用时, 这类宏就变成了表达循环总是在紧急情况下终止的最自然的方式。

某些最简单的迭代宏在图 11.7 中给出。我们已经见过 while (82 页)，其主体将在一个测试表达式返回真时求值。跟它相反的是 till，做同样的事只不过是当测试表达式返回假时。最后 for，同样在前面看到过 (116 页)，在一个数字范围上做迭代。

通过定义这些宏展开到 do 上面，我们就允许在它们的主体里使用 go 和 return。正如 do 从 block 和 tagbody 处继承了这些权力，while，till

[1]译者注：loop 宏可能是 Common Lisp 里最复杂的一个宏，它定义了非常多的循环关键字来精确设计一个循环行为，相当于一个 Fortran 风格的小型嵌入式语言。多数开源 Common Lisp 实现直接采用了来自 MIT 的 loop 参考实现。但在不使用任何循环关键字时，loop 宏所做的就是无条件地循环求值其表达式体。

```
(defmacro while (test &body body)
  `(do ()
       ((not ,test))
     ,@body))

(defmacro till (test &body body)
  `(do ()
       (,test)
     ,@body))

(defmacro for ((var start stop) &body body)
  (let ((gstop (gensym)))
    `(do ((,var ,start (1+ ,var))
          (,gstop ,stop))
         ((> ,var ,gstop))
       ,@body)))
```

图 11.7: 简单的迭代宏

和 for 也从 do 处继承了它们。就像 118 页上所解释的那样，do 内部隐含 block 里的 nil 标签将被图 11.7 中的宏所捕捉。虽然与其说这是个 bug，不如说它是个特性，但它至少应该被清楚地提到。

当你需要定义更强大的迭代结构时，宏是必不可少的。图 11.8 里包括了两个 dolist 的一般化；两者都在求值主体时绑定一组变量到一个列表中相继的子序列上。例如，给定两个参数，do-tuples/o 将成对迭代：

```
> (do-tuples/o (x y) '(a b c d)
    (princ (list x y)))
(A B)(B C)(C D)
NIL
```

给定相同的参数，do-tuples 将会做同样的事，然后折回到列表的开头：

```
> (do-tuples/o (x y) '(a b c d)
    (princ (list x y)))
(A B)(B C)(C D)(D A)
NIL
```

两个宏都返回 nil，除非一个显式的 return 出现在主体中。

在需要处理某种路径表示的程序里，这类迭代会经常用到。后缀 /o 和 /c 用来说明这两个版本分别在开放和封闭的路径上操作。举个例子，如果 points 是一个点的列表而 (drawline x y) 在 x 和 y 之间画线，那么画一条从起点到终点的路径我们写成

```
(do-tuples/o (x y) points (drawline x y))
```

假如 points 是一个多边形的节点列表，为了画出它的边界，我们写成

```
(defmacro do-tuples/o (parms source &body body)
  (if parms
      (let ((src (gensym)))
        `(prog ((,src ,source))
           (mapc #'(lambda ,parms ,@body)
                 ,@(map0-n #'(lambda (n)
                               `(nthcdr ,n ,src))
                           (1- (length parms))))))))

(defmacro do-tuples/c (parms source &body body)
  (if parms
      (with-gensyms (src rest bodfn)
        (let ((len (length parms)))
          `(let ((,src ,source))
             (when (nthcdr ,(1- len) ,src)
               (labels ((,bodfn ,parms ,@body))
                 (do ((,rest ,src (cdr ,rest)))
                     ((not (nthcdr ,(1- len) ,rest))
                      ,@(mapcar #'(lambda (args)
                                    `(,bodfn ,@args))
                                (dt-args len rest src))
                      nil)
                   (,bodfn ,@(map1-n #'(lambda (n)
                                         `(nth ,(1- n)
                                               ,rest))
                                     len)))))))))

(defun dt-args (len rest src)
  (map0-n #'(lambda (m)
              (map1-n #'(lambda (n)
                          (let ((x (+ m n)))
                            (if (>= x len)
                                `(nth ,(- x len) ,src)
                                `(nth ,(1- x) ,rest))))
                      len))
          (- len 2)))
```

图 11.8: 迭代子序列的宏

```
(do-tuples/c (x y) points (drawline x y))
```

作为第一个实参给出的形参列表可以是任意长度，相应的迭代就会按照那个长度的组合进行。如果只给一个参数，两者都会降级成 dolist：

```
> (do-tuples/o (x) '(a b c) (princ x))
ABC
NIL
> (do-tuples/c (x) '(a b c) (princ x))
ABC
NIL
```

do-tuples/c 的定义比 do-tuples/o 更复杂一些，因为它要在搜索到列表结尾时折返回来。如果有 n 个参数，do-tuples/c 必须在返回之前多做 $n-1$ 次迭代：

```
> (do-tuples/c (x y z) '(a b c d)
    (princ (list x y z)))
(A B C)(B C D)(C D A)(D A B)
NIL
> (do-tuples/c (w x y z) '(a b c d)
    (princ (list w x y z)))
(A B C D)(B C D A)(C D A B)(D A B C)
NIL
```

前一个对 do-tuples/c 调用的展开式显示在图 11.9 中。生成过程的困难之处是那些展示折返到列表开头的调用序列。这些调用 (在本例中，有两个) 由 dt-args 生成。

11.5 多值迭代

内置的 do 宏在多重返回值出现之前就已经存在了。幸运的是 do 可以进化成支持这种新情况，因为 Lisp 的进化掌握在程序员的手中。图 11.10 包含一个支持多值的 do* 版本。在 mvdo* 里，每一个初值语句可以绑定多于一个变量：

```
> (mvdo* ((x 1 (1+ x))
          ((y z) (values 0 0) (values z x)))
         ((> x 5) (list x y z))
    (princ (list x y z)))
(1 0 0)(2 0 2)(3 2 3)(4 3 4)(5 4 5)
(6 5 6)
```

这类迭代是有用的，例如，在交互式图形程序里，经常需要处理诸如坐标和区域这样的多元量。

假设我们想要写一个简单的交互游戏，其中被追踪的对象之间要避免挤在一起。如果两个追踪者同时命中你，那么你输了；如果它们自己撞到一起，那么你赢了。图 11.11 显示了该游戏的主循环是如何用 mvdo* 来写成的。

也有可能写出一个 mvdo，并行绑定其局部变量：

```
(do-tuples/c (x y z) '(a b c d)
  (princ (list x y z)))
```

展开成：

```
(let ((#:g2 '(a b c d)))
  (when (nthcdr 2 #:g2)
    (labels ((#:g4 (x y z)
               (princ (list x y z))))
      (do ((#:g3 #:g2 (cdr #:g3)))
          ((not (nthcdr 2 #:g3))
           (#:g4 (nth 0 #:g3)
                 (nth 1 #:g3)
                 (nth 0 #:g2))
           (#:g4 (nth 1 #:g3)
                 (nth 0 #:g2)
                 (nth 1 #:g2))
           nil)
        (#:g4 (nth 0 #:g3)
              (nth 1 #:g3)
              (nth 2 #:g3))))))
```

图 11.9: 一个 do-tuples/c 调用的展开

```
> (mvdo ((x 1 (1+ x))
         ((y z) (values 0 0) (values z x)))
        ((> x 5) (list x y z))
    (princ (list x y z)))
(1 0 0)(2 0 1)(3 1 2)(4 2 3)(5 3 4)
(6 4 5)
```

do 的定义中对 psetq 的需要在第 87 页里描述过了。为了定义 mvdo，我们需要一个多值版本的 psetq。由于 Common Lisp 并未提供，我们不得不自己写一个，如图 11.12 所示。新的宏像这样工作：

```
> (let ((w 0) (x 1) (y 2) (z 3))
    (mvpsetq (w x) (values 'a 'b) (y z) (values w x))
    (list w x y z))
(A B 0 1)
```

mvpsetq 的定义依赖于三个工具函数：mklist (41 页)，group (42 页)，以及 shuffle，在这里定义，用来交错两个列表：

```
> (shuffle '(a b c) '(1 2 3 4))
(A 1 B 2 C 3 4)
```

借助 mvpsetq，我们就可以定义 mvdo 了，如图 11.13 所示。和 condlet 一样，这个宏使用了 mappend 来代替 mapcar 以避免修改最初的宏调用。这种 mappend-mklist 写法可以使一棵树扁平一层：

```
(defmacro mvdo* (parm-cl test-cl &body body)
  (mvdo-gen parm-cl parm-cl test-cl body))

(defun mvdo-gen (binds rebinds test body)
  (if (null binds)
      (let ((label (gensym)))
        `(prog nil
           ,label
           (if ,(car test)
               (return (progn ,@(cdr test))))
           ,@body
           ,@(mvdo-rebind-gen rebinds)
           (go ,label)))
      (let ((rec (mvdo-gen (cdr binds) rebinds test body)))
        (let ((var/s (caar binds)) (expr (cadar binds)))
          (if (atom var/s)
              `(let ((,var/s ,expr)) ,rec)
              `(multiple-value-bind ,var/s ,expr ,rec))))))

(defun mvdo-rebind-gen (rebinds)
  (cond ((null rebinds) nil)
        ((< (length (car rebinds)) 3)
         (mvdo-rebind-gen (cdr rebinds)))
        (t
         (cons (list (if (atom (caar rebinds))
                         'setq
                         'multiple-value-setq)
                     (caar rebinds)
                     (third (car rebinds)))
               (mvdo-rebind-gen (cdr rebinds))))))
```

图 11.10: `do*` 的多值绑定版本。

```
(mvdo* (((px py) (pos player)   (move player mx my))
        ((x1 y1) (pos obj1)     (move obj1 (- px x1)
                                           (- py y1)))
        ((x2 y2) (pos obj2)     (move obj2 (- px x2)
                                           (- py y2)))
        ((mx my) (mouse-vector) (mouse-vector))
        (win     nil            (touch obj1 obj2))
        (lose    nil            (and (touch obj1 player)
                                     (touch obj2 player))))
       ((or win lose) (if win 'win 'lose))
  (clear)
  (draw obj1)
  (draw obj2)
  (draw player))
```

(pos *obj*) 返回代表 *obj* 位置的两个值 x，y。开始的时候三个对象的位置是随机的。

(move *obj dx dy*) 根据类型和向量 $\langle dx, dy \rangle$ 来移动对象 *obj*。返回的两个值 x，y 代表其新位置。

(mouse-vector) 返回代表当前鼠标移动位置的两个值 mx，my。

(touch *obj1 obj2*) 返回真如果 *obj1* 和 *obj2* 发生接触。

(clear) 清空游戏区域。

(draw *obj*) 在当前位置绘制 *obj*。

图 11.11: 一个碰撞游戏。

```
(defmacro mvpsetq (&rest args)
  (let* ((pairs (group args 2))
         (syms  (mapcar #'(lambda (p)
                            (mapcar #'(lambda (x) (gensym))
                                    (mklist (car p))))
                        pairs)))
    (labels ((rec (ps ss)
               (if (null ps)
                   `(setq
                     ,@(mapcan #'(lambda (p s)
                                   (shuffle (mklist (car p))
                                            s))
                               pairs syms))
                   (let ((body (rec (cdr ps) (cdr ss))))
                     (let ((var/s (caar ps))
                           (expr (cadar ps)))
                       (if (consp var/s)
                           `(multiple-value-bind ,(car ss)
                                                 ,expr
                              ,body)
                           `(let ((,@(car ss) ,expr))
                              ,body)))))))
      (rec pairs syms))))

(defun shuffle (x y)
  (cond ((null x) y)
        ((null y) x)
        (t (list* (car x) (car y)
                  (shuffle (cdr x) (cdr y))))))
```

图 11.12: psetq 的多值版本。

```
(defmacro mvdo (binds (test &rest result) &body body)
  (let ((label (gensym))
        (temps (mapcar #'(lambda (b)
                           (if (listp (car b))
                               (mapcar #'(lambda (x)
                                           (gensym))
                                       (car b))
                               (gensym)))
                       binds)))
    `(let ,(mappend #'mklist temps)
       (mvpsetq ,@(mapcan #'(lambda (b var)
                              (list var (cadr b)))
                          binds
                          temps))
       (prog ,(mapcar #'(lambda (b var) (list b var))
                      (mappend #'mklist (mapcar #'car binds))
                      (mappend #'mklist temps))
         ,label
         (if ,test
             (return (progn ,@result)))
         ,@body
         (mvpsetq ,@(mapcan #'(lambda (b)
                                (if (third b)
                                    (list (car b)
                                          (third b))))
                            binds))
         (go ,label)))))
```

图 11.13: do 的多值绑定版本

```
(mvdo ((x 1 (1+ x))
       ((y z) (values 0 0) (values z x)))
      ((> x 5) (list x y z))
  (princ (list x y z)))
```

展开成：

```
(let (#:g2 #:g3 #:g4)
  (mvpsetq #:g2 1
           (#:g3 #:g4) (values 0 0))
  (prog ((x #:g2) (y #:g3) (z #:g4))
     #:g1
     (if (> x 5)
         (return (progn (list x y z))))
     (princ (list x y z))
     (mvpsetq x (1+ x)
              (y z) (values z x))
     (go #:g1)))
```

图 11.14: 一个 mvdo 调用的展开。

```
> (mappend #'mklist '((a b c) d (e (f g) h) ((i)) j))
(A B C D E (F G) H (I) J)
```

为了帮助理解这个相当大的宏，图 11.14 中含有一个展开示例。

11.6 对宏的需要

宏并不是保护参数免于求值的唯一方式。另一种方法是将它们封装在闭包里。条件和重复求值的相似之处是这两个问题本质上都不需要宏。例如，我们可以将某个版本的 if 写成函数：

```
(defun fnif (test then &optional else)
  (if test
      (funcall then)
      (if else (funcall else))))
```

我们可以通过将 then 和 else 参数表达成闭包来保护它们，所以代替

```
(if (rich) (go-sailing) (rob-bank))
```

我们可以说

```
(fnif (rich)
      #'(lambda () (go-sailing))
      #'(lambda () (rob-bank)))
```

如果所有我们想要的就是条件求值，那么宏也不是非要不可的。它们只是使程序更清晰罢了。尽管如此，当我们需要介入参数形式，或者绑定作为参数传递的变量时，宏就是必需的了。

同样的情况也适用于那些用于迭代的宏。尽管只有宏才提供这种定义一个可以带有表达式体的迭代结构的方式，其实用函数来做迭代也是可能的，只要循环体被包装在那个函数里。[2]例如内置函数 `mapc` 就是 `dolist` 的对应函数型版本。表达式

```
(dolist (b bananas)
  (peel b)
  (eat b))
```

和

```
(mapc #'(lambda (b)
          (peel b)
          (eat b))
      bananas)
```

具有相同的副作用。(尽管前者返回 `nil` 而后者返回 `bananas` 列表)。我们同样可以将 `forever` 实现成一个函数，

```
(defun forever (fn)
  (do ()
      (nil)
    (funcall fn)))
```

如果我们愿意传给它一个闭包来代替一个表达式体的话。

尽管如此，迭代结构通常想要做比像 `forever` 这种单纯的迭代更多的事：它们通常想要做一个绑定和迭代的组合。使用函数，能绑定的东西有限。如果你想把变量绑定到列表的后继元素上，可以使用一个映射函数。但如果需求比这个更复杂，你就得写一个宏了。

[2] 写一个不需要其参数封装在函数里的迭代函数也并非不可能。我们可以写一个函数在作为其参数传递的表达式上调用 `eval`。对于为何调用 `eval` 通常是不好的行为的解释，见 **??** 页。

第 12 章

广义变量

第 8 章曾提到宏的一个优点是它们变换其参数的能力。这类宏中的一个是 setf。本章关注 setf 的内涵，然后给出一些宏作为例子，它们将建立在 setf 的基础之上。

要在 setf 上编写正确无误的宏是一件难事，其难度令人咋舌。为介绍这一主题，第一节将给出一个有点小问题的简单例子。接下来一节将解释该宏的错误之处，然后展示如何修复它。第三和第四节会介绍一些基于 setf 的实用工具的例子，而最后一节会说明如何定义你自己的 setf 逆。

12.1 概念

内置宏 setf 是 setq 的广义形式。setf 的第一个参数可以是一个函数调用而不仅仅是个变量：

```
> (setq lst '(a b c))
(A B C)
> (setf (car lst) 480)
480
> lst
(480 B C)
```

一般而言 (setf x y) 可以理解成 "see to it that x evaluates to y." 作为一个宏，setf 可以深入到其参数内部来看看需要做些什么来使这样一个语句为真。如果第一个参数 (在宏展开以后) 是一个符号，那么 setf 就只会展开成一个 setq。但如果第一个参数是一个查询，则 setf 展开到对应的断言上。由于第二个参数是一个常量，前面的例子可以展开成：

```
(progn (rplaca lst 480) 480)
```

这种从查询到断言的变换称为*逆*。所有最常用的 Common Lisp 访问函数都有预定义的逆，包括 car、cdr、nth、aref[1]、get、gethash，以及那些由 defstruct 创建的访问函数。(完整的列表在 CLTL2 的第 125 页。)

[1]译者注：偶然发现的一段古老的来自 Lisp Machine 时代的源代码文档更加精辟地讨论了 aref，摘录如下：

> Common Lisp 的不完备之处在于某些 Common Lisp 提供的原语没有明确定义。例如，SETF 宏可以使 (SETF (AREF *array index*) *new-value*) 改变一个数组的内

一个可以作为 setf 第一个参数的表达式称为广义变量。广义变量已经成为一种强有力的抽象。宏调用和广义变量的相似之处在于：任何能展开成可逆引用的宏调用，其本身就会是可逆的[2]。

当我们也在 setf 之上编写我们自己的宏时，这种组合可以产生显然更为清爽的程序。其中一个我们可以定义在 setf 上面的宏是 toggle[3]，

```
(defmacro toggle (obj)
  `(setf ,obj (not ,obj)))
```

它可以开关一个广义变量的值：

```
> (let ((lst '(a b c)))
    (toggle (car lst))
    lst)
(NIL B C)
```

现在考虑下面这个简单的应用。假设某人——一个肥皂剧作者、活跃的爱管闲事的人，或者党政官员——想要维护一个小镇上所有居民之间关系的数据库。在所需的表里有一个记录朋友的表：

```
(defvar *friends* (make-hash-table))
```

这个哈希表的项本身也是哈希表，在其中潜在的朋友被映射到 t 或者 nil：

```
(setf (gethash 'mary *friends*) (make-hash-table))
```

为了使 John 成为 Mary 的朋友，我们可以说：

```
(setf (gethash 'john (gethash 'mary *friends*)) t)
```

这个镇被分为两个派系。正如派系该做的那样，每个人都说 “任何人不是朋友就是敌人”，所以镇上的每个人都被迫要加入一方或者另一方。这样当某人转变立场时，他的所有朋友变成敌人而所有敌人则变成朋友。

只用内置操作符来切换是否 x 是 y 的朋友，我们必须说成：

```
(setf (gethash x (gethash y *friends*))
      (not (gethash x (gethash y *friends*))))
```

容。因为 setf 展开成修改数组的 Lisp 代码，所以一定存在一个形式上改变数组的原语。Common Lisp 并没有给出一个能够修改数组的原语。

而当时 Lisp Machine 上有一种称为 ZetaLisp 的方言，作为 Common Lisp 的完备超集解决了所有的类似问题。原文出处：

http://jrm-code-project.googlecode.com/svn/trunk/kmachine/k-lisp.txt

[2]原文：A macro call resembles a generalized variable in that any macro call which expands into an invertible reference will itself be invertible. 译者注：这意味着可逆的宏或者广义变量都是可以嵌套使用的，有些 “正则序” 的意味。

[3]这个定义是不正确的，下一节将给出解释。

这是个相当复杂的表达式，尽管去掉 setf 之后要简单许多。如果我们在数据库上定义了一个像下面这样的访问宏：

```
(defmacro friend-of (p q)
  `(gethash ,p (gethash ,q *friends*)))
```

那么在这个宏和 toggle 之间，我们就有了更好的装备来处理数据库的改变。前面那个更新可以简单地表达成：

```
(toggle (friend-of x y))
```

广义变量就像是美味的健康食品。它们能让你的程序良好地模块化，同时变得更为优雅。如果你为你的数据结构提供了通过宏或者可逆函数的访问能力，其他模块就可以使用 setf 来修改你的数据结构而无需知道它们的内部细节。

12.2 多重求值问题

前一节警告过我们的 toggle 初始定义是不正确的：

```
(defmacro toggle (obj)                                    ; wrong
  `(setf ,obj (not ,obj)))
```

它会遇到第 10.1 节里描述过的问题，多重求值。麻烦出现在当其参数带有副作用时。例如，如果 lst 是一个对象列表，并且我们这样写：

```
(toggle (nth (incf i) lst))
```

然后期待它能开关第 (i+1) 个元素。尽管如此，在 toggle 的当前定义下这个调用将展开成：

```
(setf (nth (incf i) lst)
      (not (nth (incf i) lst)))
```

这会使 i 增长两次，并且将第 (i+1) 个元素设置成第 (i+2) 个元素的相反值。所以在这个例子里

```
> (let ((lst '(t nil t))
        (i -1))
    (toggle (nth (incf i) lst))
    lst)
(T NIL T)
```

对 toggle 的调用看起来没有效果。

仅仅是把作为 toggle 参数给出的表达式插入到 setf 的第一个参数的位置上是不够的。我们必须深入到表达式内部看看它到底做了什么：如果它含有子形式 (subform)，我们就得把它们分离开并且在它们有副作用时单独进行求值。一般而言，这是个复杂的事务。

```
(defmacro allf (val &rest args)
  (with-gensyms (gval)
    `(let ((,gval ,val))
       (setf ,@(mapcan #'(lambda (a) (list a gval))
                       args)))))

(defmacro nilf (&rest args) `(allf nil ,@args))

(defmacro tf (&rest args) `(allf t ,@args))

(defmacro toggle (&rest args)
  `(progn
     ,@(mapcar #'(lambda (a) `(toggle2 ,a))
               args)))

(define-modify-macro toggle2 () not)
```

图 12.1: 操作在广义变量上的宏。

处于简化目的，Common Lisp 提供了一个宏来自动定义有限的一类 `setf` 上的宏。这个宏叫做 `define-modify-macro`，接受三个参数：宏名，它的附加参数 (出现在广义变量之后)，以及那个为广义变量产生新值的函数[4]的名字。

使用 `define-modify-macro`，我们可以像下面这样定义 `toggle`：

```
(define-modify-macro toggle () not)
```

换句话说，这个定义是说 “为求值一个形如 `(toggle` *place*`)` 的表达式，先找到由 *place* 制定的位置，并且如果保存在那里的值是 *val*，将其替换成 `(not` *val*`)` 的值”。这是在同样的例子里使用的新宏：

```
> (let ((lst '(t nil t))
        (i -1))
    (toggle (nth (incf i) lst))
    lst)
(NIL NIL T)
```

这个版本给出了正确的结果，但它可以做得更通用。由于 `setf` 和 `setq` 都可以带有可变数量的参数，`toggle` 也应如此。我们可以通过在修改宏 (modify-macro) 的基础上定义另一个宏来增加这种能力，如图 12.1 所示。

[4] 一般意义上的函数名：`1+` 或者 `(lambda (x) (+ x 1))` 都可以。

译者注：现行 Common Lisp 标准 (CLHS) 事实上要求 `define-modify-macro` 和 `define-compiler-macro` 的第三个参数的类型必须是符号。虽然绝大多数 Common Lisp 厂商出于历史原因仍然支持在这个参数上使用 λ–表达式，但至少有一个厂商 LispWorks 已经在其 5.1 版以后修正了实现，只允许使用符号了。

```
(define-modify-macro concf (obj) nconc)

(defun conc1f/function (place obj)
  (nconc place (list obj)))

(define-modify-macro conc1f (obj) conc1f/function)

(defun concnew/function (place obj &rest args)
  (unless (apply #'member obj place args)
    (nconc place (list obj))))

(define-modify-macro concnew (obj &rest args)
  concnew/function)
```

图 12.2: 广义变量上的列表操作。

12.3 新的实用工具

本节给出一些操作在广义变量上的新实用工具的例子。它们必须是宏，以便将参数完整地传给 `setf`。

图 12.1 显示了构建在 `setf` 上的四个新宏。首先，`allf`，用于将一定数量的广义变量设置为同一个值。构建于其上的是 `nilf` 和 `tf`，相应地将它们的参数设置为 `nil` 和 `t`。这些宏很简单，但它们改变某些东西。

像 `setq` 那样，`setf` 也可以接受多个参数——交替出现的变量和值：

```
(setf x 1 y 2)
```

这些新的实用工具也可以，但你可以少给出一半参数。如果你想要把一定数量的变量初始化到 `nil`，代替

```
(setf x nil y nil z nil)
```

你只需说

```
(nilf x y z)
```

就够了。最后一个宏，`toggle`，前一节描述过了：它和 `nilf` 相似，但可以给其每个参数设置相反的真值。

这四个宏阐释了关于赋值操作符的一个重要观点。即使我们只想在普通变量上使用一个操作符，也值得将其展开到一个 `setf` 而非 `setq` 上。如果第一个参数是个符号，`setf` 将直接展开到一个 `setq`。由于我们无需任何额外开销就可以拥有 `setf` 的一般性，所以很少值得在展开式里使用 `setq`。

图 12.2[5] 包含三个破坏性修改列表结尾的宏。第 3.1 节提到依靠

[5] 译者注：源代码根据现行 Common Lisp 标准做了改动，我们额外定义了两个辅助函数以便确保 `define-modify-macro` 的第三个参数只能是符号。

```
(nconc x y)
```

的副作用是不安全的，而必须代替写成[6]

```
(setq x (nconc x y))
```

这一习惯用法被嵌入在 concf 中了。更特殊的 conc1f 和 concnew 就像是用于列表另一端的 push 和 pushnew：conc1f 在列表结尾追加一个元素，而 concnew 做同样的事，但只有当这个元素不是列表成员时才做。

第 2.2 节提到一个函数的名字即可以是符号也可以是一个 λ–表达式。这样将整个 λ–表达式作为第三个参数传给 define-modify-macro 也可以，就像 conc1f 定义里那样。[7] 使用 39 页上的 conc1 ，这个宏也可以写成：

```
(define-modify-macro conc1f (obj) conc1)
```

图 12.2 应该在一种情况下保留使用。如果你正计划通过在结尾处追加元素的方式来构造一个列表，那么最好是使用 push，然后再 nreverse 这个列表。在一个列表的开头做一些事比在结尾做要便宜一些，因为要想在结尾处做事你首先得到达那里。Common Lisp 有许多用于前者的操作符而用于后者的很少，这很可能也鼓励了高效率的编程。

12.4 更复杂的实用工具

并非所有 setf 上的宏都可以通过 define-modify-macro 来定义。举个例子，假设我们想要定义一个宏 _f 破坏性应用一个函数到一个广义变量上。内置宏 incf 就是一个 + 的 setf 简化形式、代替

```
(setf x (+ x y))
```

我们只需说

```
(incf x y)
```

新的 _f 是广义形式的下列想法：incf 能展开成一个对 + 的调用，而 _f 将展开到以第一个参数给定的操作符的调用上。例如，在第 104 页上 scale-objs 的定义里，我们必须写成

```
(setf (obj-dx o) (* (obj-dx o) factor))
```

用 _f 的话将变成

[6] 译者注：当作为 nconc 第一个参数的变量为空列表，也就是 nil 时，该变量在 nconc 执行之后将仍是 nil，而不是整个 nconc 表达式的那个相当于其第二个参数的值。

[7] 译者注：正如前面两个脚注里提到的那样，Common Lisp 标准并没有定义 define-modify-macro 的第三个参数可以是符号之外的其他东西，尽管 λ–表达式出现在一个函数调用形式的函数位置上确实是合法的。原书作者试图通过类比来说明 λ–表达式用在 define-modify-macro 中的合法性，这是不恰当的，请读者注意。

```
(_f * (obj-dx o) factor)
```

编写 _f 的不正确方式可能是：

```
(defmacro _f (op place &rest args)                    ; wrong
  `(setf ,place (,op ,place ,@args)))
```

不幸的是，我们使用 define-modify-macro 来定义一个正确的 _f，因为应用到广义变量上的操作符是通过参数给定的。

像这类更复杂的宏不得不手写。为了使这类宏容易编写，Common Lisp 提供了函数 get-setf-expansion[8]，接受一个广义变量并返回所有用于获取和设置其值的必要信息。我们将看到如果使用这个信息，通过为下面这个形式手工生成一个展开式：

```
(incf (aref a (incf i)))
```

当我们在广义变量上调用 get-setf-expansion 时，我们可以得到五个值作为用于宏展开式中的原材料：

```
> (get-setf-expansion '(aref a (incf i)))
(#:G4 #:G5)
(A (INCF I))
(#:G6)
(SYSTEM:SET-AREF #:G6 #:G4 #:G5)
(AREF #:G4 #:G5)
```

最开始两个值是临时变量列表，以及应该给它们赋的值。所以我们可以这样开始展开式：

```
(let* ((#:g4 a)
       (#:g5 (incf i)))
  ...)
```

这些绑定应该被创建在 let* 里因为一般来说这些值形式可能引用到更早出现的变量。第三[9]和第五个值是另一个临时变量并且该形式将返回广义变量的初值。由于我们想要给这个值加 1，所以我们将后者包在一个对 1+ 的调用里：

```
(let* ((#:g4 a)
       (#:g5 (incf i))
       (#:g6 (1+ (aref #:g4 #:g5))))
  ...)
```

[8]译者注：原书中给出的函数实际上是 get-setf-method，但这个函数已经不在现行 Common Lisp 标准中了，参见 X3J13 Issue 308: SETF-METHOD-VS-SETF-METHOD。经查目前常见的 Common Lisp 实现里只有较年轻的 SBCL 没有提供这个过时的函数。替代的 get-setf-expansion 接受两个参数，一个 *place* 以及可选的 *environment* 环境参数。本书后面对于所有采用 get-setf-method 的地方一律直接改用 get-setf-expansion，不再另行说明。

[9]第三个值当前总是一个单元素列表。它被返回成一个列表来提供 (目前为止还不可能) 在广义变量中保存多值的可能性。

最后， get-setf-expansion 返回的第四个值是一个必须在新绑定环境下进行的赋值：

```
(let* ((#:g4 a)
       (#:g5 (incf i))
       (#:g6 (1+ (aref #:g4 #:g5))))
  (system:set-aref #:g6 #:g4 #:g5))
```

经常发生的是，该形式将引用到不在 Common Lisp 定义之中的内部函数。通常 setf 掩盖了这些函数的存在，但它们必须存在于某处。因为关于它们的所有东西都是具体实现相关的，所以可移植的代码应该使用由 get-setf-expansion 返回的形式，而不是直接引用诸如 system:set-aref 这样的函数。

现在要实现 _f 我们只需写几乎跟我们刚才手工展开 incf 做的事情差不多的一个宏。唯一的区别就是，不再将 let* 里的最后一个形式包装在一个 1+ 调用里，我们将它包装在来自 _f 参数的一个表达式里。_f 的定义在图 12.3 中给出。

这是个很有用的实用工具。现在有了它，举个例子，我们就可以很容易地将任何命名函数替换成一个记忆化的 (第 5.3 节) 的等价函数。[10] 要记忆化 foo 我们可以用：

```
(_f memoize (symbol-function 'foo))
```

有了 _f 也可以使定义其他 setf 上的宏变得简单。例如，我们现在可以将 conc1f (图 12.2) 定义成：

```
(defmacro conc1f (lst obj)
  `(_f nconc ,lst (list ,obj)))
```

图 12.3 包含了 setf 上其他一些有用的宏。下一个，pull，是内置的 pushnew 的相反操作。这一对像是 push 和 pop 的更有眼光的版本；pushnew 将一个新元素推到一个列表的上面如果它还不是其成员，而 pull 则是破坏性地从一个列表里移除选定的元素。pull 定义中的 &rest 参数使 pull 可以接受和 delete 相同的关键字参数：

```
> (setq x '(1 2 (a b) 3))
(1 2 (A B) 3)
> (pull 2 x)
(! (A B) 3)
> (pull '(a b) x :test #'equal)
(1 3)
> x
(1 3)
```

你几乎可以认为这个宏是想这样定义的：

[10] 尽管如此，内置函数不应该以这种方式被记忆化。Common Lisp 禁止重定义内置函数。

```
(defmacro _f (op place &rest args)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-expansion place)
    `(let* (,@(mapcar #'list vars forms)
            (,(car var) (,op ,access ,@args)))
       ,set)))

(defmethod pull (obj place &rest args)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-expansion place)
    (let ((g (gensym)))
      `(let* ((,g ,obj)
              ,@(mapcar #'list vars forms)
              (,(car var) (delete ,g ,access ,@args)))
         ,set))))

(defmacro pull-if (test place &rest args)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-expansion place)
    (let ((g (gensym)))
      `(let* ((,g ,test)
              ,@(mapcar #'list vars forms)
              (,(car var) (delete-if ,g ,access ,@args)))
         ,set))))

(defmacro popn (n place)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-expansion place)
    (with-gensyms (gn glst)
      `(let* ((,gn ,n)
              ,@(mapcar #'list vars forms)
              (,glst ,access)
              (,(car var) (nthcdr ,gn ,glst)))
         (prog1 (subseq ,glst 0 ,gn)
                ,set)))))
```

图 12.3: `setf` 上更复杂的宏。

```
(defmacro pull (obj seq &rest args)                    ; wrong
  `(setf ,seq (delete ,obj ,seq ,@args)))
```

尽管如果它真的这样定义，它将同时遇到求值顺序和求值数量方面的问题。我们可以将一个版本的 pull 定义成简单的修改宏：

```
(define-modify-macro pull (obj &rest args)
  (lambda (seq obj &rest args)
    (apply #'delete obj seq args)))
```

但由于修改宏必须将广义变量作为其第一个参数，这样我们就不得不以相反的次序给出前两个参数，这样就不太直观了。

更一般的 pull-if 接受一个初始的函数参数，并且代替 delete 展开成一个 delete-if：

```
> (let ((lst '(1 2 3 4 5 6)))
    (pull-if #'oddp lst)
    lst)
(2 4 6)
```

这两个宏阐释了另一个一般性观点。如果下层函数接受可选参数，建立在其上的宏也应该这样做。pull 和 pull-if 都把可选参数传给它们的 delete 们了。

图 12.3 里的最后一个宏，popn，是 pop 的一般形式。代替只从列表里弹出一个元素，它能弹出并返回任意长度的一个子序列：

```
> (setq x '(a b c d e f))
(A B C D E F)
> (popn 3 x)
(A B C)
> x
(D E F)
```

图 12.4 含有一个排序其参数的宏。如果 x 和 y 是变量并且我们想要确保 x 的值不是两个值中较小的那个，我们可以写成：

```
(if (> y x) (rotatef x y))
```

但如果我们想在三个或者更多变量上做这件事，所需的代码量就会迅速增长。代替手写这样的代码，我们可以让 sortf 来为我们写。该宏接受一个比较操作符加上任意数量的广义变量，然后交换它们的值直到它们处于由操作符决定的顺序位置。在最简单的情形，参数可以是通常的变量：

```
> (setq x 1 y 2 z 3)
3
> (sortf > x y z)
3
> (list x y z)
(3 2 1)
```

```
(defmacro sortf (op &rest places)
  (let* ((meths (mapcar #'(lambda (p)
                            (multiple-value-list
                              (get-setf-expansion p)))
                        places))
         (temps (apply #'append (mapcar #'third meths))))
    `(let* ,(mapcar #'list
                    (mapcan #'(lambda (m)
                                (append (first m)
                                        (third m)))
                            meths)
                    (mapcan #'(lambda (m)
                                (append (second m)
                                        (list (fifth m))))
                            meths))
       ,@(mapcon #'(lambda (rest)
                     (mapcar
                       #'(lambda (arg)
                           `(unless (,op ,(car rest) ,arg)
                              (rotated ,(car rest) ,arg)))
                       (cdr rest)))
                 temps)
       ,@(mapcar #'fourth meths))))
```

图 12.4: 一个排序其参数的宏

```
(sortf > x (aref ar (incf i)) (car lst))
```

展开 (在一种可能的实现里) 成：

```
(let* ((#:g1 x)
       (#:g4 ar)
       (#:g3 (incf i))
       (#:g2 (aref #:g4 #:g3))
       (#:g6 lst)
       (#:g5 (car #:g6)))
  (unless (> #:g1 #:g2)
    (rotatef #:g1 #:g2))
  (unless (> #:g1 #:g5)
    (rotatef #:g1 #:g5))
  (unless (> #:g2 #:g5)
    (rotatef #:g2 #:g5))
  (setq x #:g1)
  (system:set-aref #:g2 #:g4 #:g3)
  (system:set-car #:g6 #:g5))
```

图 12.5: 一个 sortf 调用的展开式

一般情况下，它们可以是任何可逆的表达式。假设 cake 是一个返回某人的一块蛋糕的可逆函数，而 bigger 是一个定义在蛋糕上的比较函数。如果我们想要强制规定 moe 的 cake 不得小于 larry 的 cake，后者也不得小于 curly 的，我们写成：

```
(sortf bigger (cake 'moe) (cake 'larry) (cake 'curly))
```

sortf 的定义在轮廓上与 _f 相似。它开始于一个 let*，其中由 get-setf-expansion 返回的临时变量被绑定到广义变量的初始值上。sortf 的核心是中央的 mapcon 表达式，用于生成排序这些临时变量的代码。宏的这部分生成的代码会随着参数个数指数增长。在排序之后，广义变量会被用那些由 get-setf-expansion 返回的形式重新赋值。使用的算法是 $O(n^2)$ 冒泡排序，但这个宏并非用于带有巨大数量的参数的调用。

图 12.5 给出了一个 sortf 的展开式。在初始的 let* 中，参数和它们的子形式按照从左到右的顺序小心地求值。然后出现三个表达式比较并且可能交换临时变量的值：第一个和第二个比较，然后第一个和第三个，然后第二个和第三个。最后广义变量从左到右被重新赋值。尽管很少出现问题，宏参数应该通常按从左到右的顺序进行赋值，和它们被求值的顺序保持一致。

诸如 _f 和 sortf 这类操作符跟接受函数型参数的函数有一定相似之处。应该认识到它们是完全不同的东西。一个类似 find-if 的函数接受一个函数并调用它；而一个类似 _f 的宏接受一个*名字*，然后让它称为一个表达式的 car。_f 和 sortf 都可以被写成接受函数型参数。例如，_f 可以被写成：

```
(defmacro _f (op place &rest args)
  (let ((g (gensym)))
    (multiple-value-bind (vars forms var set access)
                         (get-setf-expansion place)
      `(let* ((,g ,op)
              ,@(mapcar #'list vars forms)
              (,(car var) (funcall ,g ,access ,@args)))
         ,set))))
```

然后调用 `(_f #'+ x 1)`。但是 `_f` 的原始版本不但可以做到这个版本能做到的所有功能，而且由于它处理的是名字，它还可以接受一个宏或者特殊形式的名字。就像 `+` 那样，你还可以调用，举个例子，`nif` (138 页)：

```
> (let ((x 2))
    (_f nif x 'p 'z 'n)
    x)
P
```

12.5 定义逆

12.1 节解释了任何展开成可逆引用的宏调用其本身也将是可逆的。尽管如此，你却不必只为使其可逆而将一个操作符定义成宏。通过使用 `defsetf` 你可以告诉 Lisp 如何对任意函数或宏调用求逆。

这个宏可以两种方式来使用。在最简单的情形里，它的参数是两个符号：

```
(defsetf symbol-value set)
```

在更复杂的形式下，一个对 `defsetf` 的调用类似 `defmacro` 调用，带有一个附加的参数用于更新值形式。例如，下面这个可以为 `car` 定义一个可能的逆：

```
(defsetf car (lst) (new-car)
  `(progn (rplaca ,lst ,new-car)
          ,new-car))
```

在 `defmacro` 和 `defsetf` 之间有一个重要区别：后者自动为其参数创建生成符号 (gensym)。通过上面给出的定义，`(setf (car x) y)` 将展开成：

```
(let* ((#:g2 x)
       (#:g1 y))
  (progn (rplaca #:g2 #:g1)
         #:g1))
```

这样我们写 `defsetf` 展开器时就无需担心诸如变量捕捉，或者求值的次数和顺序之类的问题了。

在 CLTL2 Common Lisp 中，也可以直接通过 `defun` 来定义 `setf` 的逆，这样前面的示例也可以写成：

```
(defvar *cache* (make-hash-table))

(defun retrieve (key)
  (multiple-value-bind (x y) (gethash key *cache*)
    (if y
        (values x y)
        (cdr (assoc key *world*)))))

(defsetf retrieve (key) (val)
  `(setf (gethash ,key *cache*) ,val))
```

图 12.6: 一个非对称的逆

```
(defun (setf car) (new-car lst)
  (rplaca lst new-car)
  new-car)
```

更新的值应该作为这样一个函数的第一个参数。将其作为函数的返回值也很方便。

目前为止的示例都建议广义变量被用于指向一个数据结构中的某个位置。歹徒将他的人质带进地牢，然后救援人员再把她带上来；他们都沿着同样的路径，但是不同的方向。人们觉得 `setf` 必须以这种方式工作并不奇怪，因为所有预定义的逆看起来都是这种形式的；确实，`place` 也是对一个需要求逆的参数的传统命名。

原则上，`setf` 可以更一般：一个访问形式和它的逆甚至不需要操作在同一个数据结构上。假设在某些应用里我们想要缓存数据库更新。这可能是必要的，举个例子，如果即时做实际的更新缺乏效率，或者如果所有更新必须在提交之前验证一致性时。

假设 `*world*` 是实际的数据库。为简单起见，我们将它做成一个元素为 `(key . val)` 形式的关联表 (assoc–list)。图 12.6 显示了一个称为 `retrieve` 的查询函数。如果 `*world*` 是

```
((a . 2) (b . 16) (c . 50) (d . 20) (f . 12))
```

那么

```
> (retrieve 'c)
50
```

不同于一个 `car` 调用，一个 `retrieve` 调用并不指向一个数据结构中的特定位置。返回值可能来自两个位置里的一个。而 `retrieve` 的逆，同样定义在图 12.6 中，仅指向它们中的一个：

```
> (setf (retrieve 'n) 77)
77
```

```
> (retrieve 'n)
77
T
```

该查询返回第二个值 t，以表明在缓存中找到了答案。

就像宏本身那样，广义变量是一种具有非凡威力的抽象。这里可能还有更多有待探索的东西。当然，有的用户很可能已经发现了一些使用广义变量的方法，通过它们能得到更为优雅和强大的程序。但也有可能以全新的方式来使用 setf 逆，或者找到其他类似的有用的变换技术。

第 13 章

编译期计算

前面的章节描述了几类必须用宏来实现的操作符。本章描述可以用函数来解决，但用宏可以更加高效的一类问题。第 8.2 节列出了在给定情形下使用宏的利弊。有利的一面包括 “在编译期做计算”。通过定义一个操作符为宏，有时你可在其展开时完成它的某些工作。本章会关注那些充分利用这种可能性的宏。

13.1 新的实用工具

第 8.2 节里提出了使用宏将计算转移到编译期的可能性。这里我们有一个宏 avg 的例子，它返回其参数的平均值：

```
> (avg pi 4 5)
4.047...
```

图 13.1 显示了 avg 首先定义成函数然后又定义成宏。当 avg 定义成宏时，对 length 的调用可以在编译器做。在宏版本里我们也避免了在运行期处理 &rest 参数的开销。在许多实现里，写成宏的 avg 将会更快。

这类源自在展开期就知道参数数量的节约可以和我们从 in (138 页) 中得到的另一类组合起来，后者甚至可以避免求值一些参数。图 13.2 含有两个版本的 most-of，它在其多数参数为真时返回真：

```
> (most-of t t t nil)
T
```

宏版本展开成的代码，就像 in，只求值它需要数量的参数。例如，(most-of (a) (b) (c)) 展开的等价代码：

```
(defun avg (&rest args)
  (/ (apply #'+ args) (length args)))

(defmacro avg (&rest args)
  `(/ (+ ,@args) ,(length args)))
```

图 13.1: 求平均值时转移计算。

```
(defun most-of (&rest args)
  (let ((all 0)
        (hits 0))
    (dolist (a args)
      (incf all)
      (if a (incf hits)))
    (> hits (/ all 2))))

(defmacro most-of (&rest args)
  (let ((need (floor (/ (length args) 2)))
        (hits (gensym)))
    `(let ((,hits 0))
       (or ,@(mapcar #'(lambda (a)
                         `(and ,a (> (incf ,hits) ,need)))
                     args)))))
```

图 13.2: 转移和避开计算。

```
(let ((count 0))
  (or (and (a) (> (incf count) 1))
      (and (b) (> (incf count) 1))
      (and (c) (> (incf count) 1))))
```

在最佳情形里，刚好过半的参数将被求值。

如果知道一个宏的部分参数的值也可以转移计算到编译期。图 13.3 给出了这样的一个宏。函数 nthmost 接受一个数 n 以及一个数列，然后返回数列中第 n 大的那个数；和其他序列函数一样，它是从零开始索引的：

```
> (nthmost 2 '(2 6 1 5 3 4))
4
```

函数版本写得非常简单。它对列表排序然后在结果上调用 nth。由于 sort 是破坏性的，nthmost 在排序之前先复制列表。这样写的话，nthmost 会从两个方面影响效率：它构造新的点对，而且排序整个参数列表，尽管我们只关心前 n 个。

如果我们在编译期知道 n 的值，就可以另眼看待这个问题了。图 13.3 中的其余代码定义了一个宏版本的 nthmost。这个宏做的第一件事是去检查它的第一个参数。如果第一个参数字面上不是一个数，它就被展开成和我们上面看到的相同的代码。如果第一个参数是一个数的话，我们可以采用另一个办法。比方说，如果你想要找到一个盘子里第三大的那块饼干，那么你可以依次查看每一块饼干同时保持手里总是拿着已知最大的三块，用这个办法达到目的。当你检查完所有的饼干之后，你手里最小的那块饼干就是你要找的了。如果 n 是一个小常数，并且这个数字远小于饼干的总数，那么和“先对它们的全部进行排序” 的方法相比，这种技术可以让你更方便地得到想找的那块饼干。

```
(defun nthmost (n lst)
  (nth n (sort (copy-list lst) #'>)))

(defmacro nthmost (n lst)
  (if (and (integerp n) (< n 20))
      (with-gensyms (glst gi)
        (let ((syms (map0-n #'(lambda (x) (gensym)) n)))
          `(let ((,glst ,lst))
             (unless (< (length ,glst) ,(1+ n))
               ,@(gen-start glst syms)
               (dolist (,gi ,glst)
                 ,(nthmost-gen gi syms t))
               ,(car (last syms))))))
      `(nth ,n (sort (copy-list ,lst) #'>))))

(defun gen-start (glst syms)
  (reverse
    (maplist #'(lambda (syms)
                 (let ((var (gensym)))
                   `(let ((,var (pop ,glst)))
                      ,(nthmost-gen var (reverse syms)))))
             (reverse syms))))

(defun nthmost-gen (var vars &optional long?)
  (if (null vars)
      nil
      (let ((else (nthmost-gen var (cdr vars) long?)))
        (if (and (not long?) (null else))
            `(setq ,(car vars) ,var)
            `(if (> ,var ,(car vars))
                 (setq ,@(mapcan #'list
                                 (reverse vars)
                                 (cdr (reverse vars)))
                       ,(car vars) ,var)
                 ,else)))))
```

图 13.3: 使用编译期知道的参数。

```
(nthmost 2 nums)
```

展开成：

```
(let ((#:g7 nums))
  (unless (< (length #:g7) 3)
    (let ((#:g6 (pop #:g7)))
      (setq #:g1 #:g6))
    (let ((#:g5 (pop #:g7)))
      (if (> #:g5 #:g1)
          (setq #:g2 #:g1 #:g1 #:g5)
          (setq #:g2 #:g5)))
    (let ((#:g4 (pop #:g7)))
      (if (> #:g4 #:g1)
          (setq #:g3 #:g2 #:g2 #:g1 #:g1 #:g4)
          (if (> #:g4 #:g2)
              (setq #:g3 #:g2 #:g2 #:g4)
              (setq #:g3 #:g4))))
    (dolist (#:g8 #:g7)
      (if (> #:g8 #:g1)
          (setq #:g3 #:g2 #:g2 #:g1 #:g1 #:g8)
          (if (> #:g8 #:g2)
              (setq #:g3 #:g2 #:g2 #:g8)
              (if (> #:g8 #:g3)
                  (setq #:g3 #:g8)
                  nil))))
    #:g3))
```

图 13.4: `nthmost` 的展开式。

这是一种当 n 在编译期已知时采取的策略。在它的展开式里，宏创建了 n 个变量，然后调用 `nthmost-gen` 来生成那些求值成查看每一块饼干的代码。图 13.4 给出了一个示例的宏展开。宏 `nthmost` 在行为上跟原来的函数一样，除了不能作为 `apply` 的参数传递以外。这里使用宏的理由完全是为了效率：宏版本不在运行期构造新点对，并且如果 n 是一个小的常数，那么比较的次数可以更少。

难道为了写出高效的程序，就必须兴师动众，编这么大的一个宏么？对于本例来说，可能不是。这里之所以给出两个版本的 `nthmost`，主要的原因是想举一个例子，它揭示了一个普遍的原则：当某些参数在编译期已知时，你可以用宏来生成更高效的代码。是否利用这种可能性取决于你想获得多少好处，以及你可以付出多少额外的努力来编写一个高效的宏版本。由于 `nthmost` 的宏版本既长又复杂，它可能只有在极端场合才值得去写。尽管如此，编译期已知的信息总是一个值得考虑的因素，就算你选择不去利用它。

13.2 举例：贝塞尔曲线

就像 `with-` 宏 (第 11.2 节) 那样，用于编译期计算的宏更像是为特定应用而写的，而非通用目的的实用工具。一个通用目的的实用工具在编译期能知道多少东西？它的参数个数，可能还有某些参数的值。如果我们想要利用其他限定条件，它们可能就只能是程序自己才懂得和使用的信息了。

本节将作为一个实例，展示宏是怎样加速贝塞尔曲线的生成的。如果对曲线的操作是交互式的话，那么它们的生成速度必须得非常快才行。可以看出如果曲线的分段数是事先知道的，大多数计算就可以在编译期完成。把我们的曲线生成器写成一个宏，我们就可以将预先计算好的值嵌入到代码里。这应该比把它们保存在数组里，这种更常规的优化方式甚至还要快。

一条贝塞尔曲线是由四个点确定的——两个端点和两个控制点。当工作在两维时，这些点定义了曲线上所有点的 x 和 y 坐标的参数方程。如果两个端点是 (x_0, y_0) 和 (x_3, y_3)，以及两个控制点 (x_1, y_1) 和 (x_2, y_2)，那么曲线上点的方程就是：

$$x = (x_3 - 3x_2 + 3x_1 - x_0)u^3 + (3x_2 - 6x_1 + 3x_0)u^2 + (3x_1 - 3x_0)u + x_0$$
$$y = (y_3 - 3y_2 + 3y_1 - y_0)u^3 + (3y_2 - 6y_1 + 3y_0)u^2 + (3y_1 - 3y_0)u + y_0$$

如果我们用 u 在 0 和 1 之间的 n 个值来求值这个方程，我们就得到曲线上的 n 个点。举个例子，如果我们想把曲线画成 20 个分段，那么我们将用 $u = .05, .1, \ldots, .95$ 来求值方程。对于 u 在 0 或 1 上的求值是不需要的，因为如果 $u = 0$ 它们将生成第一个端点 (x_0, y_0)，而当 $u = 1$ 时它们将生成第二个端点 (x_3, y_3)。

一个明显的优化是令 n 为定值，提前计算 n 的指数，然后将它们存在一个 $(n-1)\cdot 3$ 的数组里。通过将曲线生成器定义成一个宏，我们甚至可以做得更好。如果 n 在展开时已知，程序可以简单地展开成 n 条画线指令。那些预先计算好的 n 的指数，可以直接作为字面上的值插入到宏展开式里而不必再保存在数组里了。

图 13.5 里含有一个实现了这一策略的曲线生成宏。代替了立即画线，它将生成的点 dump 到一个数组里。当一个曲线被交互式移动时，每一个实例必须画两次：一次显示它，以及在画下一个之前删除它。在两次画线之间，这些点必须被存在某个地方。

当 $n = 20$ 时，`genbez` 展开成 21 个 `setf`。由于 u 的指数直接出现在代码里，我们省下了在运行期查找它们的开销，以及在启动时计算它们的开销。和 u 的指数一样，数组的索引以常量的形式出现在展开式中，所以对那些 `(setf aref)` 的边界检查也可以在编译期完成。

13.3 应用

后面的章节将会提到其它一些宏，它们也用到了编译期已知信息。其中一个很好的例子是 `if-match` (?? 页)。在这里例子里面，模式匹配器会比较两个序列，序列中可能含有变量，在比较的过程中，模式匹配器会分析是否存在某种给这些变量赋值的方式，可以让两个序列相等。`if-match` 的设计显示：如果序列中的一个在编译期已知，并且只有这个序列里含有变量，那么匹配可以做得更高效。一个办法是在运行期比较两个序列并构造列表来保存这个过程中建立的变量绑定，不过我们可以改成用一个宏根据已知序列生成进行精确的比较的代码，并且可以在真正的 Lisp 变量里保存绑定。

第 19–24 章里描述的嵌入式语言，也在很大程度上利用了这些可在编译期获得的信息。由于嵌入式语言就是编译器，利用这些信息是其唯一自然的工作方式。这是一个普遍规律：越是精心设计的宏，它对其参数的约束也就越多，并且你利用这些约束来产生高效的代码的机会也就越好。

```
(defconstant *segs* 20)
(defconstant *du* (/ 1.0 *segs*))
(defconstant *pts* (make-array (list (1+ *segs*) 2)))

(defmacro genbez (x0 y0 x1 y1 x2 y2 x3 y3)
  (with-gensyms (gx0 gx1 gy0 gy1 gx3 gy3)
    `(let ((,gx0 ,x0) (,gy0 ,y0)
           (,gx1 ,x1) (,gy1 ,y1)
           (,gx3 ,x3) (,gy3 ,y3))
       (let ((cx (* (- ,gx1 ,gx0) 3))
             (cy (* (- ,gy1 ,gy0) 3))
             (px (* (- ,x2 ,gx1) 3))
             (py (* (- ,y2 ,gy1) 3)))
         (let ((bx (- px cx))
               (by (- py cy))
               (ax (- ,gx3 px ,gx0))
               (ay (- ,gy3 py ,gy0)))
           (setf (aref *pts* 0 0) ,gx0
                 (aref *pts* 0 1) ,gy0)
           ,@(map1-n #'(lambda (n)
                         (let* ((u (* n *du*))
                                (u^2 (* u u))
                                (u^3 (expt u 3)))
                           `(setf (aref *pts* ,n 0)
                                  (+ (* ax ,u^3)
                                     (* bx ,u^2)
                                     (* cx ,u)
                                     ,gx0)
                                  (aref *pts* ,n 1)
                                  (+ (* ay ,u^3)
                                     (* by ,u^2)
                                     (* cy ,u)
                                     ,gy0))))
                     (1- *segs*))
           (setf (aref *pts* *segs* 0) ,gx3
                 (aref *pts* *segs* 1) ,gy3))))))
```

图 13.5: 生成贝塞尔曲线的宏。

第 14 章

Anaphoric 宏

第 9 章将变量捕捉仅仅视为一种问题——把它当成某种意料之外，并且只会让程序变糟的负面因素。本章将显示变量捕捉也可以有建设性地被使用。如果没有这个特性，一些有用的宏就无法写出来。

Lisp 程序里想要测试一个表达式是否返回非空值，并且如果是的话对这个值做某些事的情形并不少见，如果表达式有一定的求值开销，那么一个人通常必须这样做：

```
(let ((result (big-long-calculation)))
  (if result
      (foo result)))
```

难道就不能简单点儿让我们只需像英语里的说法：

```
(if (big-long-calculation)
    (foo it))
```

通过利用变量捕捉，我们可以写出一个以这种方式工作的 `if` 版本。

14.1 Anaphoric 变种

在自然语言里，*anaphor* 是一个指向会话中过去某件事的表达式。英语中最常用的 anaphor 可能是 “it”，就像在 “Get the wrench and put it on the table (拿个扳手并且把它放在桌子上)” 里那样。anaphor 是日常语言中极大的便利——试着想象没有它们会发生什么——但它们在编程语言里却很少出现。这在大多数情况下还是好的。anaphor 表达式经常会产生歧义，而当今的编程语言设计上无法处理语法歧义。

尽管如此，在 Lisp 程序中引入一种非常有限形式的 anaphor 还是有可能的。一个 anaphor，实际上是一个可捕捉的符号。我们可以通过指定某些作为代词的符号来使用 anaphor，然后编写宏有意地捕捉这些符号。

在 `if` 的新版本里，符号 `it` 就是那个我们想要捕捉的东西。anaphoric if，简称为 `aif`，被定义成下面这个样子：

```
(defmacro aif (test-form then-form &optional else-form)
  `(let ((it ,test-form))
     (if it ,then-form ,else-form)))
```

```
(defmacro aif (test-form then-form &optional else-form)
  `(let ((it ,test-form))
     (if it ,then-form ,else-form)))

(defmacro awhen (test-form &body body)
  `(aif ,test-form
        (progn ,@body)))

(defmacro awhile (expr &body body)
  `(do ((it ,expr ,expr))
       ((not it))
     ,@body))

(defmacro aand (&rest args)
  (cond ((null args) t)
        ((null (cdr args)) (car args))
        (t `(aif ,(car args) (aand ,@(cdr args))))))

(defmacro acond (&rest clauses)
  (if (null clauses)
      nil
      (let ((cl1 (car clauses))
            (sym (gensym)))
        `(let ((,sym ,(car cl1)))
           (if ,sym
               (let ((it ,sym)) ,@(cdr cl1))
               (acond ,@(cdr clauses)))))))
```

图 14.1: Common Lisp 操作符的 anaphoric 变种。

然后像前面例子里那样使用：

```
(aif (big-long-calculation)
     (foo it))
```

当你使用一个 `aif` 时，符号 `if` 会被绑定到测试表达式返回的结果上。在宏调用中，`it` 看起来是自由的，但事实上表达式 `(foo it)` 通过 `aif` 的展开将被插入到一个符号 `it` 被绑定了的上下文中：

```
(let ((it (big-long-calculation)))
  (if it (foo it) nil))
```

这样一个在源代码中看起来自由的符号就被宏展开所绑定了。本章里所有的 anaphoric 宏都使用了同样技术的某些变种。

图 14.1 包含了一些 Common Lisp 操作符的 anaphoric 变种。`aif` 之后就是 `awhen`，`when` 的明显 anaphoric 变种：

```
(awhen (big-long-calculation)
```

```
  (foo it)
  (bar it))
```

aif 和 awhen 都是经常会有用的，但 awhile 可能是这些 anaphoric 宏中唯一一个比它的正常表兄弟，while (定义在 82 页)，更经常被需要用到。诸如 while 和 awhile 这样的宏典型地被用在一个程序需要从某些外部数据源中拉数据的情形。并且当你正在拉一个数据源时，除非你正在简单地等待其改变状态，否则你将通常想要对你从里面找到的东西做某些事：

```
(awhile (poll *fridge*)
  (eat it))
```

aand 的定义与前面的几个相比更复杂一些。它提供了一个 and 的 anaphoric 版本；当求值它的每一个实参时，it 将被绑定前一个参数所返回的值上。[1] 在实践中，aand 倾向于在那些做条件查询的程序中使用，例如这里：

```
(aand (owner x) (address it) (town it))
```

它返回 x 的拥有者 (如果有的话) 的地址 (如果有的话) 中的城镇 (如果有的话)。如果不使用 aand，该表达式将不得不写成

```
(let ((own (owner x)))
  (if own
      (let ((adr (address own)))
        (if adr (town adr)))))
```

从 aand 的定义可以看出，它的展开式将随宏调用中的实参的数量而变。如果没有实参，那么 aand，就像正常的 and 那样，应该简单地返回 t。否则展开式将递归地生成，每一步都会在嵌套的 aif 链中产生一层：

```
(aif ⟨first arument⟩
     ⟨expansion for rest of arguments⟩)
```

一个 aand 的展开必须在只剩下一个实参时终止，而不是像大多数递归函数那样继续展开直到 nil 为止。如果递归过程一直进行直到没有任何合取式留下来，展开式将总是下列形式：

```
(aif ⟨ c_1 ⟩
     ⋮
     (aif ⟨ c_n⟩
          t)...)
```

这样的一个表达式将总是返回 t 或者 nil，上面的示例将不会正确工作。

[1] 尽管人们倾向于把 and 和 or 放在一起考虑，但写一个 anaphoric 版本的 or 没有任何意义。一个 or 表达式中的实参只有当它前面的实参求值到 nil 才会被求值，所以一个 aor 中的 anaphor 将没有任何用处。

```
(defmacro alambda (parms &body body)
  `(labels ((self ,parms ,@body))
     #'self))

(defmacro ablock (tag &rest args)
  `(block ,tag
     ,(funcall (alambda (args)
                 (case (length args)
                   (0 nil)
                   (1 (car args))
                   (t `(let ((it ,(car args)))
                         ,(self (cdr args))))))
               args)))
```

图 14.2: 更多的 anaphoric 变种。

第 10.4 章曾警告过：如果一个宏总是产生一个包含对其自身调用的展开式，那么展开过程将不会终止。尽管是递归的，但 `aand` 是安全的，因为在基本情形里它的展开式没有引用到 `aand`。

最后一个例子，`acond`，用于 `cond` 子句的其余部分想使用测试表达式返回值的场合。(这种情形如此普遍以至于某些 Scheme 实现提供了一种方式来使用 `cond` 子句中测试表达式的返回值。)

在一个 `acond` 子句的展开式里，测试表达式的结果开始时将被保存在一个符号生成的变量里，然后符号 `it` 可以只被绑定到子句的其余部分里。当宏创建绑定时，它们应该总是采取尽可能窄的作用域。这里，如果我们省去这个生成符号而是立即将 `it` 绑定到测试表达式的结果上，就像这样：

```
(defmacro acond (&rest clauses)                          ; wrong
  (if (null clauses)
      nil
      (let ((cl1 (car clauses)))
        `(let ((it ,(car cl1)))
           (if it
               (progn ,@(cdr cl1))
               (acond ,@(cdr clauses)))))))
```

那么 `it` 绑定的作用域也将涵盖剩下的的测试表达式。

图 14.2 包括一些更复杂的 anaphoric 变种。宏 `alambda` 用于字面引用到一个递归函数。什么时候一个人想要字面引用到一个递归函数呢？我们可以通过一个带 `#'`的 λ–表达式来字面应用到一个函数：

```
#'(lambda (x) (* x 2))
```

但正如第 2 章里解释的，你不能用一个简单的 λ–表达式来表达递归函数。代替的方法是你不得不通过 `labels` 定义一个局部函数。下面这个函数 (来自 20 页)

```
(defun count-instances (obj lsts)
  (labels ((instances-in (lst)
             (if (consp lst)
                 (+ (if (eq (car lst) obj) 1 0)
                    (instances-in (cdr lst)))
                 0)))
    (mapcar #'instances-in lsts)))
```

接受一个对象和列表，然后返回一个由列表中每个元素里含有的对象个数所组成的数列：

```
> (count-instances 'a '((a b c) (d a r p a) (d a r) (a a)))
(1 2 1 2)
```

通过 anaphora 我们可以将这些东西变成字面递归函数。`alambda` 宏使用 `labels` 来创建一个函数，这样就可以用来表达，例如，阶乘函数：

```
(alambda (x) (if (= x 0) 1 (* x (self (1- x)))))
```

使用 `alambda` 我们可以定义一个等价版本的 `count-instances`，像这样：

```
(defun count-instances (obj lists)
  (mapcar (alambda (list)
            (if list
                (+ (if (eq (car list) obj) 1 0)
                   (self (cdr list)))
                0))
          lists))
```

不同于图 14.1 和 14.2 里的其他宏，它们都捕捉 `it`，`alambda` 捕捉 `self`。一个 `alambda` 实例展开进一个 `labels` 表达式，其中 `self` 被绑定到被定义的函数上。`alambda` 表达式不但更小一些，而且看起来很像与之类似的 `lambda` 表达式，这使得代码使用起来便于阅读。

这个新宏被用在 `ablock` 的定义里，内置的 `block` special form 的一个anaphoric 版本。在一个 `block` 中，参数按照从左到右求值。在 `ablock` 里也是一样，只是在这里每一个变量 `it` 将被绑定到前一个表达式的值上。

这个宏应谨慎使用。Though convenient at times, `ablock` would tend to beat what could be nice functional programs into imperative form. The following is, unfortunately, a characteristically ugly example:

```
> (ablock north-pole
    (princ "ho ")
    (princ it)
    (princ it)
    (return-from north-pole))
ho ho ho
NIL
```

无论何时一个有意使用变量捕捉的宏被导出到另一个包里，有必要同时导出那些被捕捉的符号。例如，无论 aif 被导出到哪里，it 也应同样被导出。否则出现在宏定义里的 it 和宏调用里使用的 it 将会是不同的符号。

14.2 失败

在 Common Lisp 中符号 nil 有至少三个不同的工作。它首先是一个空列表，也就是

```
> (cdr '(a))
NIL
```

除了空列表以外，nil 被用来表示逻辑假，例如这里

```
> (= 1 0)
NIL
```

以及最后，函数返回 nil 来表明失败。例如，内置 find-if 的工作是返回一个列表中满足某些测试条件的第一个元素。如果没有发现这样的元素，find-if 将返回 nil：

```
> (find-if #'oddp '(2 4 6))
NIL
```

不幸的是，我们无法判断这种情形，find-if 成功了，却是成功地发现了 nil：

```
> (find-if #'null '(2 nil 6))
NIL
```

在实践中，用 nil 来同时表示假和空列表并没有导致太多的麻烦。事实上，这样可能相当便利。尽管如此，用 nil 来表示失败却是一个痛处，因为它意味着一个像 find-if 这样的函数返回的结果可能是有歧义的。

区分失败和一个 nil 返回值的问题出现在任何做查找操作的函数里。对于这个问题 Common Lisp 提供了不少于三种解决方案。最常用的手法，在多重返回值出现之前，是故意返回一个列表结构。例如，区分 assoc 的失败就没有任何麻烦；当执行成功时它返回成对的问题和答案：

```
> (setq synonyms '((yes . t) (no . nil)))
((YES . T) (NO))
> (assoc 'no synonyms)
(NO)
```

按照这个思路，如果我们担心 find-if 带来的歧义，我们可以用 member-if，它不是只返回满足测试的元素，而是返回以该元素开始的整个 cdr：

```
> (member-if #'null '(2 nil 6))
(NIL 6)
```

自从多重返回值诞生以后，这个问题就有了另一个解决方案：用一个值代表数据而第二个值指示成功或失败。内置的 gethash 以这种方式工作。它总是返回两个值，第二个值代表是否有任何东西被找到了：

```
> (setf edible                        (make-hash-table)
        (gethash 'olive-oil edible) t
        (gethash 'motor-oil edible) nil)
NIL
> (gethash 'motor-oil edible)
NIL
T
```

这样如果你想要检测所有三种可能情况，你可以使用类似下面的句法：

```
(defun edible? (x)
  (multiple-value-bind (val found?) (gethash x edible)
    (if found?
        (if val 'yes 'no)
        'maybe)))
```

这样就可以将失败和逻辑假区分开了：

```
> (mapcar #'edible? '(motor-oil olive-oil iguana))
(NO YES MAYBE)
```

Common Lisp 还支持第三种指示失败的方法：让访问函数接受一个特殊对象作为参数，假设是一个生成符号，然后在失败的时候返回这个对象。这种方法被用于 get，它接受一个可选参数来表示当特定属性没有找到时返回的东西：

```
> (get 'life 'meaning (gensym))
#:G618
```

在可能使用多重返回值的地方，gethash 所采用的那种方法是最清晰的。我们不希望必须给每一个访问函数传递附加参数，就像对 get 所做的那样。并且和其他两种替代方法相比，使用多重返回值更加通用；find-if 可以被写成返回两个值，而 gethash 却不可能在不做 consing 的情况下被重写成返回无歧义的列表。这样在编写新的用于查询的函数，或者对于其他可能失败的任务时，通常采用 gethash 的模型会更好一些。

在 edible? 里出现的写法只是一个被宏很好地隐藏起来的固定写法。对于像 gethash 这样的访问函数我们将希望有一个新版本的 aif，它不再绑定和测试同一个值，而是绑定第一个值但同时测试第二个值。这个新版本的 aif，称为 aif2，在图 14.3 中给出。使用它我们可以将 edible? 写成：

```
(defun edible? (x)
  (aif2 (gethash x edible)
        (if it 'yes 'no)
        'maybe))
```

```
(defmacro aif2 (test &optional then else)
  (let ((win (gensym)))
    `(multiple-value-bind (it ,win) ,test
       (if (or it ,win) ,then ,else))))

(defmacro awhen2 (test &body body)
  `(aif2 ,test
         (progn ,@body)))

(defmacro awhile2 (test &body body)
  (let ((flag (gensym)))
    `(let ((,flag t))
       (while ,flag
         (aif2 ,test
               (progn ,@body)
               (setq ,flag nil))))))

(defmacro acond2 (&rest clauses)
  (if (null clauses)
      nil
      (let ((cl1 (car clauses))
            (val (gensym))
            (win (gensym)))
        `(multiple-value-bind (,val ,win) ,(car cl1)
           (if (or ,val ,win)
               (let ((it ,val)) ,@(cdr cl1))
               (acond2 ,@(cdr clauses)))))))
```

图 14.3: 多值 anaphoric 宏。

```
(let ((g (gensym)))
  (defun read2 (&optional (str *standard-input*))
    (let ((val (read str nil g)))
      (unless (equal val g) (values val t)))))

(defmacro do-file (filename &body body)
  (let ((str (gensym)))
    `(with-open-file (,str ,filename)
       (awhile2 (read2 ,str)
         ,@body))))
```

图 14.4: 文件实用工具。

图 14.3 还包含有 awhen，awhile，和 acond 的类似替代版本。作为一个使用 acond2 的例子，见 204 页上 match 的定义。通过使用这个宏我们可以用一个 cond 的形式来表达，否则函数将变得更长并且缺少对称性。

内置的 read 指示错误的方式跟 get 一样。它接受一个可选参数来说明是否在遇到文件结尾时生成一个错误，并且如果不生成的话，将返回什么值。图 14.4 包括了一个用第二个返回值指示失败的替代版 read：read2 返回两个值，输入的表达式和一个用 nil 代表文件结束的标志。它调用 read 时带上了一个生成符号用于文件结束时返回，但却省去了每次调用 read2 时构造生成符号的麻烦，这个函数被定义成一个带有编译期生成符号的私有拷贝的闭包。

图 14.4 也包括了一个可以方便地在一个文件里的所有表达式上做迭代的宏，用 awhile2 和 read2 写成的。使用 do-file 我们可以，例如，这样写出一个 load 版本来：

```
(defun our-load (filename)
  (do-file filename (eval it)))
```

14.3 引用透明 (Referential Transparency)

Anaphoric 宏有时被认为是违反了引用透明，这是 Gelernter 和 Jagannathan 的定义：

> 一个语言是*引用透明*的如果 (a) 每一个子表达式都可以被替换成任何其他与之具有相等值的子表达式并且 (b) 一个表达式在给定上下文中的所有出现都带有相同的值。

注意到这个标准是应用于语言，而不是程序的。没有一个带有赋值的语言是引用透明的。在下面这个表达式中

```
(list x
      (setq x (not x))
      x)
```

第一个和最后一个 x 带有不同的值，因为被一个 setq 干预了。必须承认，这是丑陋的代码。这一事实甚至可能意味着 Lisp 不是引用透明的。

Norvig 提到将 if 重定义成下面这个样子将会很方便：

```
(defmacro if (test then &optional else)
  `(let ((that ,test))
     (if that ,then ,else)))
```

但拒绝这个宏的理由也是它违背了引用透明。

尽管如此，这里的问题是出在重定义了内置操作符，而不是因为使用了 anaphora。上面定义中的子句 (b) 要求一个表达式 “在给定上下文中” 必须总是返回相同的值。如果是在这个 let 表达式中就没有问题，

```
(let ((that 'which))
  ...)
```

符号 that 表示一个新变量，因为 let 就是被用于创建一个新的上下文。

上面那个宏的麻烦在于它重定义了 if，而 if 并非用于创建一个新的上下文。如果我们给 anaphoric 宏不同的名字就解决问题了。(根据 CLTL2，重定义 if 总是非法的。) 由于 aif 定义的一部分就是建立一个新的上下文并且其中 it 是一个新变量，所以这样一个宏并不违背引用透明。

现在，aif 确实违背了另一个原则，这和引用透明没有任何关系：that newly established variables somehow be indicated in the source code. 前面的那个 let 表达式清楚地表明 that 将指向一个新变量。可能会有争议说一个 aif 里面的 it 绑定就没有那么清楚。尽管如此，这里有一个不算很强的争辩：aif 只创建了一个变量，并且这个变量的创建是我们使用 aif 的唯一理由。

Common Lisp 自身并不认为这一原则是不可违反的。CLOS 函数 call-next-method 的绑定依赖于上下文的方式和 aif 函数体中符号 it 的绑定方式是一样的。(关于 call-next-method 应如何实现的一个建议方案，见 225 页上的 defmeth 宏。) 在任何情况下，这类原则最终只有一个目的：使程序更容易阅读。并且 anaphora 确实是程序更容易阅读，就像它们使英语更容易阅读那样。

第 15 章

返回函数的宏

第 5 章显示了如何编写返回其他函数的函数。宏使得组合操作符的任务变得更加简单了。本章将显示如何使用宏来构建等价于第 5 章里定义的那些抽象，但更加清晰也更高效。

15.1 函数的构造

如果 f 和 g 都是函数，那么 $f \circ g(x) = f(g(x))$。第 5.4 节显示了怎样将 $\circ$ 实现成一个叫做 `compose` 的函数：

```
> (funcall (compose #'list #'1+) 2)
(3)
```

本节里，我们考虑用宏来定义更好的函数构造器的方式。图 15.1 中包含一个称为 `fn` 的通用函数构造器，可以根据描述来构造复合函数。它的参数应该是一个 `(`*operator* `.` *arguments*`)` 形式的表达式。*operator* 可以是一个函数或宏的名字——或者特殊对待的 `compose`。*arguments* 可以是一个参数的函数或宏的名字，或者可以作为 `fn` 参数的表达式。例如，

```
(fn (and integerp oddp))
```

产生一个等价于

```
#'(lambda (x) (and (integerp x) (oddp x)))
```

的函数。

如果我们使用 `compose` 作为操作符 (operator)，我们就得到一个表达所有参数复合的函数，但不需要像 `compose` 被定义为函数时那样的显式 `funcall` 调用。例如，

```
(fn (compose list 1+ truncate))
```

展开成：

```
#'(lambda (#:g1) (list (1+ (truncate #:g1))))
```

后者允许对 `list` 和 `1+` 这种简单函数进行内联编译。`fn` 宏接受一般意义上的操作符名称；λ–表达式也是允许的，就像

```
(defmacro fn (expr) `#',(rbuild expr))

(defun rbuild (expr)
  (if (or (atom expr) (eq (car expr) 'lambda))
      expr
      (if (eq (car expr) 'compose)
          (build-compose (cdr expr))
          (build-call (car expr) (cdr expr)))))

(defun build-call (op fns)
  (let ((g (gensym)))
    `(lambda (,g)
       (,op ,@(mapcar #'(lambda (f)
                          `(,(rbuild f) ,g))
                      fns)))))

(defun build-compose (fns)
  (let ((g (gensym)))
    `(lambda (,g)
       ,(labels ((rec (fns)
                   (if fns
                       `(,(rbuild (car fns))
                         ,(rec (cdr fns)))
                       g)))
          (rec fns)))))
```

图 15.1: 通用函数构造宏。

```
(fn (compose (lambda (x) (+ x 3)) truncate))
```

可以展开成

```
#'(lambda (#:g2) ((lambda (x) (+ x 3)) (truncate #:g2)))
```

Here the function expressed as a lambda-expression will certainly be compiled inline, whereas a sharp-quoted lambda-expression given as an argument to the function `compose` would have to be funcalled.

第 5.4 节显示了如何再定义三个函数构造器：`fif`，`fint`，以及 `fun`。这些函数现在被统一到通用的 `fn` 宏了。使用 `and` 操作符将产生一个参数操作符的交集：

```
> (mapcar (fn (and integerp oddp))
          '(c 3 p 0))
(NIL T NIL NIL)
```

而 `or` 操作符则产生并集：

```
> (mapcar (fn (or integerp symbolp))
          '(c 3 p 0.2))
(T T T NIL)
```

并且 `if` 产生一个函数其函数体是条件执行的：

```
> (map1-n (fn (if oddp 1+ identity)) 6)
(2 2 4 4 6 6)
```

尽管如此，除此三个以外我们还可以使用其他函数：

```
> (mapcar (fn (list 1- identity 1+))
          '(1 2 3))
((0 1 2) (1 2 3) (2 3 4))
```

并且 `fn` 表达式里的参数本身也可以是表达式：

```
> (remove-if (fn (or (and integerp oddp)
                     (and consp cdr)))
             '(1 (a b) c (d) 2 3.4 (e f g)))
(C (D) 2 3.4)
```

使 `fn` 将 `compose` 作为特殊情况考虑并没有使它强大多少。如果你把嵌套的参数传给 `fn`，那就可以得到函数的复合。例如，

```
(fn (list (1+ truncate)))
```

展开成：

```
#'(lambda (#:g1)
    (list ((lambda (#:g2) (1+ (truncate #:g2))) #:g1)))
```

这相当于

```
(compose #'list #'1+ #'truncate)
```

`fn` 宏将 `compose` 特殊对待只是为了使类似调用容易阅读。

15.2 在 cdr 上做递归

第 5.5 和 5.6 节显示了如何编写构造递归函数的函数。接下来两节将显示 anaphoric 宏如何能给我们在那里定义的函数提供一个更清晰的接口。

第 5.5 节显示了如何定义一个称为 `lrec` 的扁平列表递归器。通过 `lrec` 我们可以将下面这个函数：

```
(defun our-every (fn lst)
  (if (null lst)
      t
      (and (funcall fn (car lst))
           (our-every fn (cdr lst)))))
```

对于例如 `oddp` 的调用表达成：

```
(lrec #'(lambda (x f) (and (oddp x) (funcall f)))
      t)
```

宏这里可以使事情更容易些。为了表达一个递归函数我们至少需要说明那些事情呢？如果我们可以首语重复地引用当前列表的 cdr (作为 `it`) 以及递归调用 (作为 `rec`)，那么我们应该可以表达成类似这样：

```
(alrec (and (oddp it) rec) t)
```

图 **??** 包含有可以允许我们这样表达的相关宏定义。

```
> (funcall (alrec (and (oddp it) rec) t)
           '(1 3 5))
T
```

这个新宏的工作方式是将作为第二个参数给出的表达式转化成一个传递给 `lrec` 的函数。由于第二个参数可能首语重复地引用到 `it` 或 `rec`，在宏展开式里函数的主体必须出现在为这些符号建立的绑定作用域里。

图 15.2 事实上含有两个不同版本的 `alrec`。前面例子里使用的版本需要用到符号宏 (symbol macro，见第 7.11 节)。由于只有较新的 Common Lisp 版本才支持符号宏[1]，所以图 15.2 里也包含了一个稍微不那么方便的 `alrec` 版本其中 `rec` 被定义成一个局部函数。代价是，作为一个函数，`rec` 将不得不被包在括号里：

```
(alrec (and (oddp it) (rec)) t)
```

在提供了 `symbol-macrolet` 的 Common Lisp 实现里推荐使用最初那个版本。

Common Lisp 带有分开的函数命名空间，这使得通过这些递归构造器定义命名函数略有不便：

[1]译者注：这些问题现在已经完全不存在了，几乎所有的现行 Common Lisp 实现 (除了 GCL, GNU Common Lisp) 都支持 ANSI Common Lisp 标准——跟 CLTL2 几乎没有多少区别。

```
(defmacro alrec (rec &optional base)
  "cltl2 version"
  (let ((gfn (gensym)))
    `(lrec #'(lambda (it ,gfn)
               (symbol-macrolet ((rec (funcall ,gfn)))
                 ,rec))
           ,base)))

(defmacro alrec (rec &optional base)
  "cltl1 version"
  (let ((gfn (gensym)))
    `(lrec #'(lambda (it ,gfn)
               (labels ((rec () (funcall ,gfn)))
                 ,rec))
           ,base)))

(defmacro on-cdrs (rec base &rest lsts)
  `(funcall (alrec ,rec #'(lambda () ,base)) ,@lsts))
```

图 15.2: 递归列表的宏。

```
(setf (symbol-function 'our-length)
      (alrec (1+ rec) 0))
```

图 15.2 里的最后一个宏打算抽象这一过程。使用 `on-cdrs` 我们可以只需这样写：

```
(defun our-length (lst)
  (on-cdrs (1+ rec) 0 lst))

(defun our-every (fn lst)
  (on-cdrs (and (funcall fn it) rec) t lst))
```

图 15.3 显示了用这个新宏定义的一些现存的 Common Lisp 函数。在 `on-cdrs` 的表达下，这些函数被削减到它们最基本的形式，并且使我们注意到它们之间在其他表达方式下可能看不到的相似之处。

图 15.4 含有一些可以很容易地用 `on-cdrs` 来定义的新实用工具。开始的三个，`unions`，`intersections`，和 `differences` 分别实现了集合的并，交，及其逆操作。Common Lisp 有这些操作的内置函数，但它们每次只能用于两个列表。这样如果我们想要找到三个列表的并集就不得不写成：

```
> (union '(a b) (union '(b c) '(c d)))
(A B C D)
```

新的 `unions` 跟 `union` 行为相似，但可以接受任意数量的参数，这样我们只需说：

```
> (unions '(a b) '(b c) '(c d))
(D C A B)
```

```
(defun our-copy-list (lst)
  (on-cdrs (cons it rec) nil lst))

(defun our-remove-duplicates (lst)
  (on-cdrs (adjoin it rec) nil lst))

(defun our-find-if (fn lst)
  (on-cdrs (if (funcall fn it) it rec) nil lst))

(defun our-some (fn lst)
  (on-cdrs (or (funcall fn it) rec) nil lst))
```

图 15.3: 用 on-cdrs 定义的 Common Lisp 函数。

```
(defun unions (&rest sets)
  (on-cdrs (union it rec) (car sets) (cdr sets)))

(defun intersections (&rest sets)
  (unless (some #'null sets)
    (on-cdrs (intersection it rec) (car sets) (cdr sets))))

(defun differences (set &rest outs)
  (on-cdrs (set-difference rec it) set outs))

(defun maxmin (args)
  (when args
    (on-cdrs (multiple-value-bind (mx mn) rec
               (values (max mx it) (min mn it)))
             (values (car args) (car args))
             (cdr args))))
```

图 15.4: 用 on-cdrs 定义的新实用工具。

和 union 一样，unions 并不保持初始列表中的元素顺序。

同样的关系也适用于 Common Lisp 的 intersection 和更一般的 intersections 之间。在这个函数的定义里，对于宏参数的初始测试出于效率原因被加上；如果集合中有空集它将短路掉整个计算过程。

Common Lisp 也有一个称为 set-difference 的函数，其接受两个列表然后返回属于第一个但不属于第二个的元素：

```
> (set-difference '(a b c d) '(a c))
(D B)
```

我们的新版本像 - 那样来处理多重参数。例如 (differences x y z) 等价于 (set-difference x (unions y z))，尽管不像后者那样需要做 cons。

```
> (differences '(a b c d e) '(a f) '(d))
(B C E)
```

这些集合操作符仅仅是用来举例。对于它们没有实际需要，因为它们所表现的列表递归的退化情形已经被内置的 reduce 处理了。例如，代替

```
(unions ...)
```

你也可以只需说

```
((lambda (&rest args) (reduce #'union args)) ...)
```

尽管如此，在一般情况下 on-cdrs 比 reduce 更强有力。

因为 rec 指向一个调用而非一个值，我们可以使用 on-cdrs 来创建返回多值的函数。图 15.4 中最后一个函数，maxmin，利用这种可能性在一次列表遍历中同时找出最大和最小的元素：

```
> (maxmin '(3 4 2 8 5 1 6 7))
8
1
```

在后续章节中出现的代码里也可能用到 on-cdrs。例如，compile-cmds (第 214 页)

```
(defun compile-cmds (cmds)
  (if (null cmds)
      'regs
      `(,@(car cmds) ,(compile-cmds (cdr cmds))))))
```

可以简单地定义成：

```
(defun compile-cmds (cmds)
  (on-cdrs `(,@it ,rec) 'regs cmds))
```

15.3 在子树上递归

宏在列表上递归做的事，它们也能在子树上递归做。本节里，我们用宏来给 5.6 节里定义的树递归器定义更加清晰的接口。

在 5.6 节里我们定义了两个树递归构造器，总是遍历整棵树的 ttrav，以及更为复杂，但允许你控制何时递归停止的 trec。使用这些函数，我们可以将 our-copy-tree

```
(defun our-copy-tree (tree)
  (if (atom tree)
      tree
      (cons (our-copy-tree (car tree))
            (if (cdr tree) (our-copy-tree (cdr tree))))))
```

表达成

```
(ttrav #'cons)
```

而一个对 rfind-if

```
(defun rfind-if (fn tree)
  (if (atom tree)
      (and (funcall fn tree) tree)
      (or (rfind-if fn (car tree))
          (and (cdr tree) (rfind-if fn (cdr tree))))))
```

的调用，例如 oddp，可以表达成：

```
(trec #'(lambda (o l r) (or (funcall l) (funcall r)))
      #'(lambda (tree) (and (oddp tree) tree)))
```

Anaphoric 宏可以给 trec 做出一个更好的接口，就像前一节它们对 lrec 所做的那样。一个满足一般需求的宏将必须能够首语重复引用到三个东西：当前所在树，我们称之为 it，递归下降左子树，我们称之为 left，以及递归下降右子树，我们称之为 right。建立好这些约定以后，我们就应该可以像下面这样用新宏来表达前面的函数：

```
(atrec (cons left right))

(atrec (or left right) (and (oddp it) it))
```

图 15.5 包含有这个宏的定义。

在没有 symbol-macrolet 的 Lisp 版本中，我们可以使用图 15.5 中的第二个定义来定义 atrec。这个版本将 left 和 right 定义成局部函数，所以 our-copy-tree 就必须写成：

```
(atrec (cons (left) (right)))
```

```
(defmacro atrec (rec &optional (base 'it))
  "cltl2 version"
  (let ((lfn (gensym)) (rfn (gensym)))
    `(trec #'(lambda (it ,lfn ,rfn)
               (symbol-macrolet ((left (funcall ,lfn))
                                 (right (funcall ,rfn)))
                 ,rec))
           #'(lambda (it) ,base))))

(defmacro atrec (rec &optional (base 'it))
  "cltl1 version"
  (let ((lfn (gensym)) (rfn (gensym)))
    `(trec #'(lambda (it ,lfn ,rfn)
               (labels ((left () (funcall ,lfn))
                        (right () (funcall ,rfn)))
                 ,rec))
           #'(lambda (it) ,base))))

(defmacro on-trees (rec base &rest trees)
  `(funcall (atrec ,rec ,base) ,@trees))
```

图 15.5: 在树上做递归的宏。

```
(defun our-copy-tree (tree)
  (on-trees (cons left right) it tree))

(defun count-leaves (tree)
  (on-trees (+ left (or right 1)) 1 tree))

(defun flatten (tree)
  (on-trees (nconc left right) (mklist it) tree))

(defun rfind-if (fn tree)
  (on-trees (or left right)
            (and (funcall fn it) it)
            tree))
```

图 15.6: 用 on-trees 定义的函数。

```
(defconstant unforced (gensym))

(defstruct delay forced closure)

(defmacro delay (expr)
  (let ((self (gensym)))
    `(let ((,self (make-delay :forced unforced)))
       (setf (delay-closure ,self)
             #'(lambda ()
                 (setf (delay-forced ,self) ,expr)))
       ,self)))

(defun force (x)
  (if (delay-p x)
      (if (eq (delay-forced x) unforced)
          (funcall (delay-closure x))
          (delay-forced x))
      x))
```

图 15.7: `force` 和 `delay` 的实现。

出于便利，我们也定义了一个 `on-trees` 宏，跟前一节里的 `on-cdrs` 相似。图 15.6 显示了用 `on-trees` 定义的四个在 5.6 节里定义的函数。

正如第 5 章里注释的那样，由那一章里的递归生成器构造的函数将不是尾递归的。使用 `on-cdrs` 或 `on-trees` 来定义一个函数将不一定得到最高效的实现。和底层的 `trec` 和 `lrec` 一样，这些宏主要用于原型设计以及效率不是关键的程序部分里。尽管如此，本章和第 5 章的底层观点是一个人可以编写函数生成器然后把一个干净的宏接口放在它们之上。同样的技术也可以等价地用于构造那些能够产生特别高效代码的函数生成器上。

15.4 延迟求值

延迟求值 意味着只有当你需要一个表达式的值时才去求值它。一种使用延迟求值的方式是构造一个称为 *delay* 的对象。delay 是某个表达式的值的占位符。它代表一个承诺，如果在后面某个时候需要的话可以传递表达式的值。同时，由于这个承诺本身是个 Lisp 对象，它可以像它所代表的值那样被使用。然后当表达式的值被需要时，delay 可以把它返回。

Scheme 对 delay 有内置的支持。Scheme 操作符 `force` 和 `delay` 可以像图 15.7 里给出的那样在 Common Lisp 中实现。delay 被表示成一个带有两部分的结构体。第一个字段代表是否 delay 已经被求值了，并且如果是的话就含有这个值。第二个字段含有一个闭包，它可以被调用而得到该 delay 所代表的值。宏 `delay` 接受一个表达式，然后返回一个代表其值的 delay：

```
> (let ((x 2))
    (setq d (delay (1+ x))))
#S(DELAY ...)
```

要想调用一个 delay 里的闭包就要 `force` 这个 delay。函数 `force` 接受任意对象：对于普通对象它就是一个 `identity` 函数，但对于 delay 它是一个对 delay 所代表的值的要求。

```
> (force 'a)
A
> (force d)
3
```

无论何时我们需要处理可能是 delay 的对象时就用 `force`。例如，如果我们正在排序一个可能含有 delay 的列表，我们将用：

```
(sort lst #'(lambda (x y) (> (force x) (force y))))
```

在这种嵌套的 form 里使用 delay 稍微有些不方便。在实际应用中，它们可能被暗藏在另一层抽象里。

第 16 章

宏定义宏

第 17 章

读取宏 (read macros)

第 18 章

解构 (destructuring)

```
(defun match (x y &optional binds)
  (acond2
    ((or (eql x y) (eql x '_) (eql y '_)) (values binds t))
    ((binding x binds) (match it y binds))
    ((binding y binds) (match x it binds))
    ((varsym? x) (values (cons (cons x y) binds) t))
    ((varsym? y) (values (cons (cons y x) binds) t))
    ((and (consp x) (consp y) (match (car x) (car y) binds))
     (match (cdr x) (cdr y) it))
    (t (values nil nil))))

(defun varsym? (x)
  (and (symbolp x) (eq (char (symbol-name x) 0) #      ?)))

(defun binding (x binds)
  (labels ((recbind (x binds)
             (aif (assoc x binds)
                  (or (recbind (cdr it) binds)
                      it))))
    (let ((b (recbind x binds)))
      (values (cdr b) b))))
```

图 18.1: 匹配函数

第 19 章

一个查询编译器

第 20 章

续延 (continuation)

第 21 章

多处理

第 22 章

非確定性

第 23 章

Parsing with ATNs

```
(defmacro defnode (name &rest arcs)
  `(=defun ,name (pos regs) (choose ,@arcs)))

(defmacro down (sub next &rest cmds)
  `(=bind (* pos regs) (,sub pos (cons nil regs))
          (,next pos ,(compile-cmds cmds))))

(defmacro cat (cat next &rest cmds)
  `(if (= (length *sent*) pos)
       (fail)
       (let ((* (nth pos *sent*)))
         (if (member ',cat (types *))
             (,next (1+ pos) ,(compile-cmds cmds))
             (fail)))))

(defmacro jump (next &rest cmds)
  `(,next pos ,(compile-cmds cmds)))

(defun compile-cmds (cmds)
  (if (null cmds)
      'regs
      `(,@(car cmds) ,(compile-cmds (cdr cmds)))))

(defmacro up (expr)
  `(let ((* (nth pos *sent*)))
     (=values ,expr pos (cdr regs))))

(defmacro getr (key &optional (regs 'regs))
  `(let ((result (cdr (assoc ',key (car ,regs)))))
     (if (cdr result) result (car result))))

(defmacro set-register (key val regs)
  `(cons (cons (cons ,key ,val) (car ,regs))
         (cdr ,regs)))

(defmacro setr (key val regs)
  `(set-register ',key (list ,val) ,regs))

(defmacro pushr (key val regs)
  `(set-register ',key
                 (cons ,val (cdr (assoc ',key (car ,regs))))
                 ,regs))
```

图 23.1: 节点和弧的编译。

第 24 章

Prolog

第 25 章

面向对象的 Lisp

本章讨论了 Lisp 中的面向对象编程。Common Lisp 提供了一组操作符可供编写面向对象的程序时使用。这些操作符和起来，并称为 Common Lisp Object System，或者叫 CLOS。在这里我们不把 CLOS 仅仅看作一种编写面向对象程序的手段，而把它本身就当成一个 Lisp 程序。从这个角度来看待 CLOS 是理解 Lisp 和面向对象编程之间关系的关键。

25.1 万变不离其宗[1]

面向对象的编程意味着程序组织方式的一次变革。历史上的另一个变化与这个变革有几分类似，即发生在处理器计算能力分配方式上的变化。在 1970 年代，多用户计算机系统指的就是联接到大量哑终端[2]的一、两个大型机。时至今日，这个词更有可能说的是大量用网络互相联接的工作站。现在，系统的处理能力分配在单个的用户上，而非集中在一台大型计算机上。

这与面向对象编程有很大程度上的相似，后者把传统的程序结构拆分开来：它不再是让单一的程序逻辑去操纵那些被动的数据，而是让数据自己知道该做些什么，程序逻辑就隐含于这些新的数据“对象”之间的交互过程中。

举例来说，假设我们要算出一个二维图形的面积。解决这个问题的一个办法就是写一个单独的函数，让它检查参数的类型，然后分情况处理：

```
(defun area (x)
  (cond ((rectangle-p x) (* (height x) (width x)))
        ((circle-p x) (* pi (expt (radius x) 2)))))
```

面向对象的方法则是让每种对象自己就能够计算出自身的面积。`area` 这个函数就被拆开，同时每条语句都被分到对象的对应类型中去，比如 `rectangle` 类可能就会看起来像这样

```
#'(lambda (x) (* (height x) (width x)))
```

至于 `circle` 则会是这样

[1] 译者注：在原文中，本节的标题是“Plus ça Change”。它源自法国谚语“plus ça change, plus c’est la même chose”，字面意思是：变化得越多，越是原来的物事。平时使用中常常略作前半句。

[2] 译者注：指和常用的工作站相比，功能较有限的计算机终端。

```
#'(lambda (x) (* pi (expt (radius x) 2)))
```

在这种模式下，我们向对象询问该对象的面积，然后对象则根据所属类型所提供的方法来作出回应。

CLOS 的到来似乎意味着 Lisp 正在改变自己，以拥抱面向对象的编程方式。与其这样说，不如改成： Lisp 还在墨守成规，用老样子来拥抱面向对象编程，这样还确切一些。不过 Lisp 里面的那些基本原理没有一个名字，面向对象编程却有，所以时下有种趋势要把 Lisp 算成面向对象的语言。另一种说法：Lisp 是一门可扩展的语言，在这种语言里，面向对象编程的机制和结构可以轻松实现，这种说法恐怕更接近真相。

由于 CLOS 是原来就有的，所以把 Lisp 说成面向对象的编程语言并没有误导。然而，如果就这样看待 Lisp 未免太小觑它了。诚然，Lisp 是一种面向对象的编程语言，但是原因并不是它采纳了面向对象的编程模式。事实在于，这种编程模式只是 Lisp 的抽象系统提供的又一种可能性而已。为了证明这种可能性，我们有了 CLOS ——一个 Lisp 程序，它让 Lisp 成为了一门面向对象的语言。

本章的主旨在于：通过把 CLOS 作为一个嵌入式语言的实例来研究，进而揭示 Lisp 和面向对象编程之间的联系。这同时也是了解 CLOS 本身的一个很好的手段，要学习一个编程语言的特性，没什么方法能比了解这个特性的实现更有效的了。在第 7.6，那些宏就是用这种方式来讲解的。下一节将会有一个类似的对面向对象抽象是如何建立在 Lisp 之上的一个粗略的介绍。其中提到的程序将被第 **??** 节到第 25.5 节作为一个基准实现来参考。

25.2 阳春版 Lisp 中的对象

我们可以用 Lisp 来模拟各种各样不同种类的语言。有一种特别直接的办法可以把面向对象编程的理念对应到 Lisp 的基本抽象机制上。不过，CLOS 的庞大规模让我们难以认清这个事实。因此，在我们开始了解 CLOS 能让我们做什么之前，不妨先看看我们用最原始的 Lisp 都能干些什么。

我们在面向对象编程中想要的大多数特性，其实在 Lisp 里面已经有了。我们可以用少得出奇的代码来得到剩下的那部分。在本节中，我们将会用两页纸的代码实现一个对象系统，这个系统对于相当多真实的应用已经够用了。面向对象编程，简而言之，就是：

1. 具有属性的对象

2. 它能对各种消息作出反应，

3. 而且对象能从它的父对象继承相应的属性和方法。

在 Lisp 里面已经有好几种存放成组属性的方法。其中一种就是把对象实现成哈希表，把对象的属性作为哈希表里的表项。这样我们就可以用 gethash 来访问指定的属性：

```
(gethash 'color obj)
```

由于函数是数据对象，我们同样可以把它们当作属性保存起来。这就是说，我们的对象系统也可以有方法了，要调用对象的特定方法就 funcall 一下哈希表里的同名属性：

```
(funcall (gethash 'move obj) obj 10)
```

据此，我们可以定义一种 Smalltalk 风格的消息传递语法：

```
(defun tell (obj message &rest args)
  (apply (gethash message obj) obj args))
```

这样的话，要告诉 (tell) obj 移动 10 个单位，就可以说

```
(tell obj 'move 10)
```

事实上，阳春版 Lisp 唯一缺少的要素就是继承机制，不过我们可以用六行代码来实现一个初步的版本，这个版本用一个递归版的 gethash 来完成这个功能：

```
(defun rget (obj prop)
  (multiple-value-bind (val win) (gethash prop obj)
    (if win
        (values val win)
        (let ((par (gethash 'parent obj)))
          (and par (rget par prop))))))
```

如果我们在原本用 gethash 的地方换成 rget，我们就会得到继承而来的属性和方法。如此这般，我们可以指定对象的父类：

```
(setf (gethash 'parent obj) obj2)
```

到现在为止，我们只是有了单继承——即一个对象只能有一个父类。不过我们可以把 parent 属性改成一个列表，这样就能有多继承了，如图 25.1 中定义的 rget。

在单继承体系里面，当我们需要得到对象的某个属性时，只需要递归地在对象的祖先中向上搜索。如果在对象本身里面没有我们想要的属性信息时，就检查它的父类，如此这般直到找到。在多继承体系里，我们一样会需要做这样的搜索，但是这次的搜索会有点复杂，因为对象的多个祖先会构成一个图，而不再只是个简单列表了。我们不能用深度优先来搜索这个图。如果允许有多个父类，我们有如图 25.2 中所示的继承树：a 继承自 b 和 c，其中 c 继承于 d。深度优先 (或叫高度优先) 的遍历会依次走过 a、b、c 和 d。倘若想要的属性同时

```
(defun rget (obj prop)
  (some2 #'(lambda (a) (gethash prop a))
         (get-ancestors obj)))

(defun get-ancestors (obj)
  (labels ((getall (x)
             (append (list x)
                     (mapcan #'getall
                             (gethash 'parent x)))))
    (stable-sort (delete-duplicates (getall obj))
                 #'(lambda (x y)
                     (member y (gethash 'parents x))))))

(defun some2 (fn lst)
  (if (atom lst)
      nil
      (multiple-value-bind (val win) (funcall fn (car lst))
        (if (or val win)
            (values val win)
            (some2 fn (cdr lst))))))
```

图 25.1: 多继承

图 25.2: 到同一基类的多条路径

存在于在 d 和 c 里，那么我们将会得到 d 中的属性，而非 c 中的那个。这种情况会违反一个原则：即子类应当会覆盖基类中提供的缺省值。

如果需要实现继承系统的基本理念，我们就绝不能在检查一个对象的子类之前，提前检查该对象。在本例中，正确的搜索顺序应该是 a、b、c、d。那怎么样才能保证搜索的顺序是先尝试子孙再祖先呢？最简单的办法是构造一个列表，列表由原始对象的所有祖先构成，然后对列表排序，让列表中没有一个对象出现在它的子孙之前，最后再依次查看每个元素。

get-ancestors 采用了这种策略，它会返回一个按照上面规则排序的列表，列表中的元素是对象和它的祖先们。为了避免在排序时把同一层次的祖先顺序打乱，get-ancestors 使用的是 stable-sort 而非 sort。一旦排序完毕，rget 只要找到第一个具有期望属性的对象就可以了。(实用工具 some2 是 some 的一个修改版，它能适用于 gethash 这类用第二个返回值表示成功或失败的函数。)

对象的祖先列表中元素的顺序是先从最具体的开始，最后到到最一般的类型。如果 orange 是 citrus 的子类型，后者又是 fruit 的子类型，那么列表的顺序就会像这样：(orange citrus fruit)。

如果有对象它具有多个父类，那么这些前辈的座次会是从左到右排列的。也

```
(defun obj (&rest parents)
  (let ((obj (make-hash-table)))
    (setf (gethash 'parents obj) parents)
    (ancestors obj)
    obj))

(defun ancestors (obj)
  (or (gethash 'ancestors obj)
      (setf (gethash 'ancestors obj) (get-ancestors obj))))

(defun rget (obj prop)
  (some2 #'(lambda (a) (gethash prop a))
         (ancestors obj)))
```

图 25.3: 用来新建对象的函数

就是，如果我们说

```
(setf (gethash 'parents x) (list y z))
```

那么当我们在搜索一个继承得来的属性时，y 就会优先于 z 被考虑。举个例子，我们可以说爱国的无赖首先是一个无赖，然后才是爱国者：

```
> (setq scoundrel (make-hash-table)
        patriot (make-hash-table)
        patriotic-scoundrel (make-hash-table))
#<Hash-Table C4219E>
> (setf (gethash 'serves scoundrel) 'self
        (gethash 'serves patriot)   'country
        (gethash 'parents patriotic-scoundrel)
                 (list scoundrel patriot))
(#<Hash-Table C41C7E> #<Hash-Table C41F0E>)
> (rget patriotic-scoundrel 'serves)
SELF
T
```

现在让我们对这个简陋的系统加以改进。我们可以从对象创建函数着手。这个函数将会在新建对象时，构造一个该对象祖先的列表。虽然当前的版本是在进行查询的时候构造这种表的，但是我们没有理由不提前一些做这件事情。图 25.3 中定义了一个名为 obj 的函数，这个函数被用来生成新的对象，在对象的祖先列表被保存在对象本身里。为了用上保存的祖先列表，我们同时重新定义了 rget。

另一个可以改进的地方是消息调用的语法。tell 本身是多余的东西，并且由于它的原因，动词被排到了第二位。这意味着我们的程序读起来不再像是熟悉的 Lisp 前缀表达式了：

```
(tell (tell obj 'find-owner) 'find-owner)
```

```
(defmacro defprop (name &optional meth?)
  `(progn
     (defun ,name (obj &rest args)
       ,(if meth?
            `(run-methods obj ',name args)
            `(rget obj ',name)))
     (defsetf ,name (obj) (val)
       `(setf (gethash ',',name ,obj) ,val))))

(defun run-methods (obj name args)
  (let ((meth (rget obj name)))
    (if meth
        (apply meth obj args)
      (error "No ~A method for ~A." name obj))))
```

图 25.4: 函数式的语法。

我们可以通过把每个属性定义成函数来去掉 `tell` 这种语法，如图 25.4 所示。可选参数 `meth?` 的值如果是真的话，那表示这个属性应该被当作方法来处理，否则它应该被当成一个 slot，并径直返回 `rget` 所取到的值。一旦我们把这两种属性中任一种，像这样定义好了：

```
(defprop find-owner t)
```

我们就可以用函数调用的方式来引用它，同时代码读起来又有 Lisp 的样子了：

```
(find-owner (find-owner obj))
```

现在，原先的例子也变得更有可读性了：

```
> (progn
    (setq scoundrel (obj))
    (setq patriot (obj))
    (setq patriotic-scoundrel (obj scoundrel patriot))
    (defprop serves)
    (setf (serves scoundrel) 'self)
    (setf (serves patriot) 'country)
    (serves patriotic-scoundrel))
SELF
T
```

在当前的实现里，一个对象中一个名字至多对应一个方法。这个方法要么是对象自己的，要么是通过继承得来的。要是能在这个问题上拥有更多的灵活性，让我们能把本地的方法和继承来的方法组合起来，那肯定会带来许多方便。比如说，我们会希望某个对象的 `move` 方法沿用其父类的 `move` 方法，但是除此之外还要在调用之前或者之后运行一些其它的代码。

为了让这个设想变成现实，我们将修改程序，加上 before、after 和 around 方法。before 方法让我们能吩咐程序，“先别急，把这事做完再说”。这些方法

```
(defstruct meth around before primary after)

(defmacro meth- (field obj)
  (let ((gobj (gensym)))
    '(let ((,gobj ,obj))
       (and (meth-p ,gobj)
            (,(symb 'meth- field) ,gobj)))))

(defun run-methods (obj name args)
  (let ((pri (rget obj name :primary)))
    (if pri
        (let ((ar (rget obj name :around)))
          (if ar
              (apply ar obj args)
              (run-core-methods obj name args pri)))
        (error "No primary ~A method for ~A." name obj))))

(defun run-core-methods (obj name args &optional pri)
  (multiple-value-prog1
   (progn (run-befores obj name args)
          (apply (or pri (rget obj name :primary))
                 obj args))
   (run-afters obj name args)))

(defun rget (obj prop &optional meth (skip 0))
  (some2 #'(lambda (a)
             (multiple-value-bind (val win) (gethash prop a)
               (if win
                   (case meth (:around (meth- around val))
                              (:primary (meth- primary val))
                              (t (values val win))))))
         (nthcdr skip (ancestors obj))))
```

图 25.5: 辅助的方法。

会在该方法中其余部分运行前，作为前奏，被先行调用，after 方法让我们可以要求程序说，“又及，把这事也给办了”。而这些方法会在最后调用，成为了收场。在两者之间，我们会执行曾经作为完整的方法的函数，现在被称为*主方法* (primary method)。它的返回值将被作为整个方法的返回值，即使 after 方法在其后调用。

before 和 after 方法让我们能用新的行为把主方法包起来。around 方法提供了一种更奇妙的方法实现这个功能。如果有 around 方法存在，那么被调用的就不再是主方法，*而是* around 方法。并且，around 方法有办法调用主方法 (用 `call-next`，该函数在图 25.7 中提供)，至于调不调则是它的自由。

如图 25.5 和图 25.6 所示，为了让这些辅助的方法生效，我们对 `run-method`

```
(defun run-befores (obj prop args)
  (dolist (a (ancestors obj))
    (let ((bm (meth- before (gethash prop a))))
      (if bm (apply bm obj args)))))

(defun run-afters (obj prop args)
  (labels ((rec (lst)
                (when lst
                  (rec (cdr lst))
                  (let ((am (meth- after
                                   (gethash prop (car lst)))))
                    (if am (apply am (car lst) args))))))
    (rec (ancestors obj))))
```

图 25.6: 辅助的方法 (续)。

和 `rget` 加以了改进。在之前的版本里，当我们调用对象的某个方法时，运行的仅是一个函数：即最匹配的那个主函数。我们将会运行搜索祖先列表时找到的第一个方法。加上辅助方法的支持，调用的顺序将变成这样：

1. 倘若有的话，先是最匹配的 around 方法
2. 否则的话，依次是：
   (a) 所有的 before 方法，从最匹配的到最不匹配的。
   (b) 最匹配的主方法 (这是我们以前会调用的)。
   (c) 所有的 after 方法，从最不匹配的到最匹配的。

同时也注意到，方法不再作为单个的函数出现了，而成了一个有四个成员的结构。现在要定义一个 (主) 方法，不能再像这样说了：

```
(setf (gethash 'move obj) #'(lambda ...))
```

我们改口说：

```
(setf (meth-primary (gethash 'move obj)) #'(lambda ...))
```

基于上面的还有其它的一些原因，我们的下一步将会定义一个宏，让它帮我们定义方法。

图 25.7 里就定义了这样的一个宏。代码中有很大篇幅被用来实现两个函数，这两个函数让方法能引用其它的方法。around 和主方法可以使用 `call-next` 来调用*下一个方法*，所谓下一个方法指的是倘若当前方法不存在，就会被调用的方法。举个例子，如果当前运行的方法是唯一的一个 around 方法，那么下一个方法就是常见的由 before 方法、最匹配的主方法和 after 方法三者合体而成的夹心饼干。在最匹配的主方法里，下一个方法则会是第二匹

```
(defmacro defmeth ((name &optional (type :primary))
                   obj parms &body body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (defprop ,name t)
       (unless (meth-p (gethash ',name ,gobj))
         (setf (gethash ',name ,gobj) (make-meth)))
       (setf (,(symb 'meth- type) (gethash ',name ,gobj))
             ,(build-meth name type gobj parms body)))))

(defun build-meth (name type gobj parms body)
  (let ((gargs (gensym)))
    `#'(lambda (&rest ,gargs)
         (labels
             ((call-next ()
                         ,(if (or (eq type :primary)
                                  (eq type :around))
                              `(cnm ,gobj ',name (cdr ,gargs) ,type)
                            '(error "Illegal call-next.")))
              (next-p ()
                      ,(case type
                         (:around
                          `(or (rget ,gobj ',name :around 1)
                               (rget ,gobj ',name :primary)))
                         (:primary
                          `(rget ,gobj ',name :primary 1))
                         (t nil))))
           (apply #'(lambda ,parms ,@body) ,gargs)))))

(defun cnm (obj name args type)
  (case type
    (:around (let ((ar (rget obj name :around 1)))
               (if ar
                   (apply ar obj args)
                   (run-core-methods obj name args))))
    (:primary (let ((pri (rget obj name :primary 1)))
                (if pri
                    (apply pri obj args)
                    (error "No next method."))))))
```

图 25.7: 定义方法。

配的主方法。由于 call-next 的行为取决于它被调用的地方，因此 call-next 绝对不会用一个 defun 来在全局定义，不过它可以在每个由 defmeth 定义的方法里局部定义。

around 方法或者主方法可以用 next-p 来获知下一个方法是否存在。如果当前的方法是个主方法，而且主方法所属的对象是没有父类的，那么就不会有下一个方法。由于当没有下个方法时， call-next 会报错，因此应该经常调用 next-p 试试深浅。像 call-next，next-p 也是在方法里面单独地局部定义的。

下面将介绍新宏 defmeth 的使用方法。如果我们只是希望定义 rectangle 对象的 area 方法，我们会说

```
(setq rectangle (obj))
(defprop height)
(defprop width)
(defmeth (area) rectangle (r)
  (* (height r) (width r)))
```

现在，一个 rectangle 实例的面积就会由类型中对应方法计算得出：

```
> (let ((myrec (obj rectangle)))
    (setf (height myrec) 2
          (width myrec)  3)
    (area myrec))
6
```

这里有个复杂一些的例子，假设我们为 filesystem 对象定义了一个 backup 方法：

```
(setq filesystem (obj))
(defmeth (backup :before) filesystem (fs)
  (format t "Remember to mount the tape.~%"))
(defmeth (backup) filesystem (fs)
  (format t "Oops, deleted all your files.~%")
  'done)
(defmeth (backup :after) filesystem (fs)
  (format t "Well, that was easy.~%"))
```

正常的调用次序如下：

```
> (backup (obj filesystem))
Remember to mount the tape.
Oops, deleted all your files.
Well, that was easy.
DONE
```

接下来，我们想要知道备份一次会花费多少时间，所以可以定义下面的 around 方法：

```
(defmeth (backup :around) filesystem (fs)
  (time (call-next)))
```

```
(defmacro undefmeth ((name &optional (type :primary)) obj)
  '(setf (,(symb 'meth- type) (gethash ',name ,obj))
         nil))
```

图 25.8: 去掉方法

现在只要调用 filesystem 子类的 backup (除非有更匹配的 around 方法介入)，那么我们的 around 方法就会被执行。它会运行平常时候在 backup 里运行的那些代码，不同之处是把它们放到了一个 time 的调用里执行。time 的返回值则会被作为 backup 方法调用的值返回。

```
> (backup (obj filesystem))
Remember to mount the tape.
Oops, deleted all your files.
Well, that was easy.
Elapsed Time = .01 seconds
DONE
```

一旦我们知道了备份操作的需要花费多少时间，我们就希望去掉这个 around 方法。调用 undermeth 可达到这个目的 (如图 25.8)，它的参数和 defmeth 的前两个参数相同：

```
(undefmeth (backup :around) filesystem)
```

另外一个我们可能需要修改的是对象的父类列表。但是进行了这种修改之后，我们还应该相应地更新该对象以及其所有子类的的祖先列表。到目前为止，还没有办法从对象那里获知它的子类信息，所以我们必须另加一个 children 属性。

图 25.9 中的代码被用来操作对象的父类和子类。这里不再用 gethash 来获得父类和子类信息，而是分别改用操作符 parents 和 children。其中后者是个宏，因而它对于 setf 是透明的。前者是一个函数，它的逆操作被 defsetf 定义为 set-parents，这个函数包揽了所有的相关工作，让新的双向链接系统能保持其一致性。

为了更新一颗子树里所有对象的祖先，set-parents 调用了 maphier，这个函数的作用相当于继承树里的 mapc。mapc 对列表里每个元素运行一个函数，同样的，maphier 也会对对象和它所有的后代应用指定的函数。除非这些节点构成没有公共子节点的树，否则有的对象会被传入这个函数一次以上。在这里，这不会导致问题，因为调用多次 get-ancestors 和调用一次，效果相同。

现在，要修改继承层次结构的话，我们只要在对象的 parents 上调用 setf 就可以了：

```
> (progn (pop (parents patriotic-scoundrel))
         (serves patriotic-scoundrel))
```

```
(defmacro children (obj)
  `(gethash 'children ,obj))

(defun parents (obj)
  (gethash 'parents obj))

(defun set-parents (obj pars)
  (dolist (p (parents obj))
    (setf (children p)
          (delete obj (children p))))
  (setf (gethash 'parents obj) pars)
  (dolist (p pars)
    (pushnew obj (children p)))
  (maphier #'(lambda (obj)
               (setf (gethash 'ancestors obj)
                     (get-ancestors obj)))
           obj)
  pars)

(defsetf parents set-parents)

(defun maphier (fn obj)
  (funcall fn obj)
  (dolist (c (children obj))
    (maphier fn c)))

(defun obj (&rest parents)
  (let ((obj (make-hash-table)))
    (setf (parents obj) parents)
    obj))
```

图 25.9: 维护父类和子类的联系。

```
COUNTRY
T
```

当这个层次结构被修改的时候，受到影响的子孙列表和祖先列表会同时自动地更新。(children 本不是让人直接修改的，但是这也不是不可以。只要我们定义一个和 set-parents 对应的 set-children 就可以了。) 图 25.9 中的最后一个函数就是被重新定义了的 obj，重写之后的它就能借助新的代码实现功能。

这是最后一项对我们这个系统的改进，这次我们要开发新的手段来组合方法。现在，会被调用的唯一主方法将是最匹配的那个 (虽然它可以用 call-next 来调用其它的主方法)。要是我们希望能把对象所有祖先的主方法的结果组合起来呢？比如说，假设 my-orange 是 orange 的子类，而 orange 又是 citrus 的子类。如果 props 方法用在 citrus 上的返回值是 (round acidic)，相应的，orange 的返回值是 (orange sweet)，my-orange 的结果是 (dented)。要是能让 (props my-orange) 能返回这些值的并集就好办多了：(dented orange sweet round acidic)。

假如我们能让方法对所有主方法的返回值应用某个函数，而不是仅仅返回最匹配的那个主函数的返回值，那么就能解决这个问题了。图 25.10 中定义有一个宏，这个宏让我们能指定方法的组合手段，图中还定义了新版本的 run-core-methods，它能够让我们把方法组合在一起使用。我们用 defcomb 定义方法的组合形式，它把方法名作为第一个参数，第二个参数描述了期望的组合方式。通常，这第二个参数应该是一个函数。不过，它也可以是：:progn、:and、:or 和 :standard中的一个。如果使用前三个，系统就会用相应的操作符来组合主方法，用 :standard 的话，就表示我们想用以前的办法来执行方法。

图 25.10 中的核心函数是新的 run-core-methods。如果被调用的方法没有名为 mcombine 的属性，那么一切如常。否则，mcombine 应该是个函数 (比如 +)，或者是个关键字 (比如 :or)。前面一种情况，所有主方法返回值构成的列表会被送进这个函数。[3]如果是后者的情况，我们会用和这个关键字对应的函数对主方法一一迭代地操作。

如图 25.11 所示，and 和 or 这两个操作符必须要特殊处理。它们被特殊对待的原因不是因为它们是 special form，而是因为它们的短路 (short-circuit) 估值方式:

```
> (or 1 (princ "wahoo"))
1
```

这里，什么都不会被打印出来，因为 or 一看到非 nil 的参数就会立即返回。与之类似，如果有一个更匹配的方法返回真的话，那么剩下的

用 or 组合的主方法将不会被调用。为了实现 and 和 or 的这种短路求值，我们用了两个专门的函数：comb-and 和 comb-or。

[3]如果代码写得更讲究一些，可以考虑用 reduce，这样可以避免手动 cons。

```
(defmacro defcomb (name op)
  `(progn
     (defprop ,name t)
     (setf (get ',name 'mcombine)
           ,(case op
              (:standard nil)
              (:progn '#'(lambda (&rest args)
                           (car (last args))))
              (t op)))))

(defun run-core-methods (obj name args &optional pri)
  (let ((comb (get name 'mcombine)))
    (if comb
        (if (symbolp comb)
            (funcall (case comb (:and #'comb-and)
                           (:or #'comb-or))
                     obj name args (ancestors obj))
          (comb-normal comb obj name args))
      (multiple-value-prog1
       (progn (run-befores obj name args)
              (apply (or pri (rget obj name :primary))
                     obj args))
       (run-afters obj name args)))))

(defun comb-normal (comb obj name args)
  (apply comb
         (mapcan #'(lambda (a)
                     (let* ((pm (meth- primary
                                       (gethash name a)))
                            (val (if pm
                                     (apply pm obj args))))
                       (if val (list val))))
                 (ancestors obj))))
```

图 25.10: 方法的组合。

```
(defun comb-and (obj name args ancs &optional (last t))
  (if (null ancs)
      last
    (let ((pm (meth- primary (gethash name (car ancs)))))
      (if pm
          (let ((new (apply pm obj args)))
            (and new
                 (comb-and obj name args (cdr ancs) new)))
        (comb-and obj name args (cdr ancs) last)))))

(defun comb-or (obj name args ancs)
  (and ancs
       (let ((pm (meth- primary (gethash name (car ancs)))))
         (or (and pm (apply pm obj args))
             (comb-or obj name args (cdr ancs))))))
```

图 25.11: 方法的组合 (续)。

为了实现我们之前的例子，可以这样写：

```
(setq citrus (obj))
(setq orange (obj citrus))
(setq my-orange (obj orange))

(defmeth (props) citrus (c) '(round acidic))
(defmeth (props) orange (c) '(orange sweet))
(defmeth (props) my-orange (m) '(dented))

(defcomb props #'(lambda (&rest args) (reduce #'union args)))
```

这样定义之后，`props` 就能返回所有主方法返回值的并集了：[4]

```
> (props my-orange)
(DENTED ORANGE SWEET ROUND ACIDIC)
```

这个例子恰巧显示了一个只有在 Lisp 用面向对象编程才会面临的选择：是把信息保存在 slot 里，还是保存在方法里。

以后，如果想要 `props` 方法恢复到缺省的行为，只要把方法的组合方式改回标准模式 (standard) 即可：

```
> (defcomb props :standard)
NIL
> (props my-orange)
(DENTED)
```

要注意，before 和 after 方法只是在标准的组合模式下才会有效。而 around 方法会像以前那样工作。

[4] 由于 `props` 里用的组合函数是 `union`，因此列表里的元素不一定会按照原来的顺序排列。

本节中展示的程序只是作为一个演示模型，而不是想以它为基础，进行面向对象编程。写这个模型的着眼点是简洁而非效率。不管如何，这至少是一个可以工作的模型，因此也可以被用在试验性质的开发和原型开发中。如果你有意这样用它的话，有一个小改动可以让它的效率有相当的改进：如果对象只有一个父类的话，就不要计算或者保存它的祖先列表。

25.3 类和实例

上一节中写了一个尽可能短小的程序来重新实现 CLOS。理解它为我们进而理解 CLOS 铺平了道路。在下面几节中，我们会仔细了解 CLOS 本身。

在我们的这个简单实现里，没有把类和实例作语法上的区分，也没有把属性 (slot) 和方法分开。在 CLOS 里，我们用 `defclass` 定义类，同时把属性组成列表一同声明：

```
(defclass circle ()
  (radius center))
```

这个表达式的意思是，`circle` 类没有父类，但是有两个 slot：`radius` 和 `center`。我们用下面的语句可以新建一个 circle 类的实例：

```
(make-instance 'circle)
```

不幸的是，我们还没有定义读取 `circle` 中 slot 的方式，因此我们创建的任何实例都只是个摆设。为了访问特定 slot，我们需要为它定义一个对应的访问 (accessor) 函数：

```
(defclass circle ()
  ((radius :accessor circle-radius)
   (center :accessor circle-center)))
```

现在，如果我们建立了一个 circle 的实例，就可以用 `setf` 和与之对应的访问函数来设置它的 `radius` 和 `center` slot：

```
> (setf (circle-radius (make-instance 'circle)) 2)
2
```

如果像下面那样定义 slot，那么我们也可以在 `make-instance` 里直接完成这种初始化的工作：

```
(defclass circle ()
  ((radius :accessor circle-radius :initarg :radius)
   (center :accessor circle-center :initarg :center)))
```

在 slot 定义中出现的 `:initarg` 关键字表示：接下来的实参将要在 `make-instance` 中成为一个关键字参数。这个关键字实参的值将会被作为该 slot 的初始值：

```
> (circle-radius (make-instance 'circle
                   :radius 2
                   :center '(0 . 0)))
2
```

使用:initform，我们也可以定义一些 slot，让它们能初始化自己。shape 类中的 visible：

```
(defclass shape ()
  ((color   :accessor shape-color   :initarg :color)
   (visible :accessor shape-visible :initarg :visible
            :initform t)))
```

会缺省地被设置成 t：

```
> (shape-visible (make-instance 'shape))
T
```

如果一个 slot 同时具有 initarg 和 initform，那么当 initarg 被指定的时候，它享有优先权：

```
> (shape-visible (make-instance 'shape :visible nil))
NIL
```

slot 会被实例和子类继承下来。如果一个类有多个父类，那么它会继承得到这些父类 slot 的并集。因此，如果我们把 screen-circle 类同时定义成 circle 和 shape 两个类的子类，

```
(defclass screen-circle (circle shape)
  nil)
```

那么 screen-circle 会具有四个 slot，每个父类继承两个 slot。注意到，一个类并不一定要自己新建一些新的 slot，screen-circle 的意义就在于提供了一个可以实例化的类型，它同时继承自 circle 和 shape。

以前可以用在 circle 和 shape 实例的那些访问函数和 initarg 会对 screen-circle 类型的实例继续生效：

```
> (shape-color (make-instance 'screen-circle
                              :color 'red :radius 3))
RED
```

如果在 defclass 里给 color 指定一个 initform，我们就可以让所有的 screen-circle 的对应 slot 都有个缺省值：

```
(defclass screen-circle (circle shape)
  ((color :initform 'purple)))
```

这样，screen-circle 类型的实例在缺省情况下就会是紫色的了：

```
> (shape-color (make-instance 'screen-circle))
PURPLE
```

不过我们还是显式地指定一个 :color initarg，把这个 slot 初始化成其他的颜色。

在我们之前实现的简装版面向对象编程框架里，实例的值可以直接从父类的 slot 继承得到。在 CLOS 中，实例包含 slot 的方式却和类不一样。我们通过在父类里定义 initform 来为实例定义可被继承的缺省值。在某种程度上，这样处理更有灵活性。因为 initform 不仅可以是一个常量，它还可以是一个每次都返回不同值的表达式：

```
(defclass random-dot ()
  ((x :accessor dot-x :initform (random 100))
   (y :accessor dot-y :initform (random 100))))
```

每创建一个 random-dot 实例，它在 x 和 y 轴上的坐标都会是从 0 到 99 之间的一个随机整数：

```
> (mapcar #'(lambda (name)
              (let ((rd (make-instance 'random-dot)))
                (list name (dot-x rd) (dot-y rd))))
          '(first second third))
((FIRST 25 8) (SECOND 26 15) (THIRD 75 59))
```

在我们的简装版实现里，我们对两种 slot 不加区别：一种是实例自己具有的 slot，这种 slot 实例和实例之间可以不同；另一种 slot 应该是在整个类里面都相同的。在 CLOS 中，我们可以指定某些 slot 是*共享的*，换句话说，就是让这些 slot 的值在每个实例里都是相同的。达到这个效果，我们可以把 slot 声明成 :allcation :class 的。(另一个选项是 :allcation :instance。不过由于这是缺省的设置，因此就没有必要再显式地指定了。) 比如说，如果所有的猫头鹰都是夜间生活的动物，那么我们可以让 nocturnal 这个 slot 作为 owl 类的共享 slot，同时让它的初始值为 t：

```
(defclass owl ()
  ((nocturnal :accessor owl-nocturnal
              :initform t
              :allocation :class)))
```

现在，所有的 owl 实例都会继承这个 slot 了：

```
> (owl-nocturnal (make-instance 'owl))
T
```

如果我们改动了这个 slot 的“局部”值，那么我们实际上修改的是保存在这个类里面的值：

```
> (setf (owl-nocturnal (make-instance 'owl)) 'maybe)
MAYBE
> (owl-nocturnal (make-instance 'owl))
MAYBE
```

这种机制或许会造成一些困扰，所以我们可能会希望让这个 slot 成为只读的。在我们为一个 slot 定义访问函数的同时，也是在为这个 slot 的值定义一个读和写的方法。如果我们需要让这个值可读，但是不可写，那么我们可以给这个 slot 仅仅设置一个 reader 函数，而不是全功能的访问函数：

```
(defclass owl ()
  ((nocturnal :accessor owl-nocturnal
              :initform t
              :allocation :class)))
```

现在如果尝试修改 `owl` 实例的 `nocturnal` slot 的话，就会产生一个错误：

```
> (setf (owl-nocturnal (make-instance 'owl)) nil)
>> Error: The function (SETF OWL-NOCTURNAL) is undefined.
```

25.4 方法

在我们的简装版实现中，强调了一个具有词法作用域的语言里，其 slot 和方法之间的相似性。实现的程序中，保存和继承主方法的方式和对 slot 值的处理方式没有什么不同。slot 和方法区别只在于：把一个名字定义成 slot，是通过

```
(defprop area)
```

把 `area` 作为一个函数实现的，这个函数得到并返回一个值。而把这个名字定义成一个方法，则是通过

```
(defprop area t)
```

把 `area` 实现成一个函数，这个函数在得到值之后，会 funcall 这个值，同时把函数的参数传给它。

在 CLOS 中，实现这个功能的单元仍然被称为“方法”，同时也可以定义这些方法，让它们看上去就像类的属性一样。这里，我们为 `circle` 类定义一个名为 `area` 的方法：

```
(defmethod area ((c circle))
  (* pi (expt (circle-radius c) 2)))
```

这个方法的参数列表表示，这是个接受一个参数的函数，参数应该是 `circle` 类型的实例。

和简单实现里一样，我们像调用一个函数那样调用这个方法：

```
> (area (make-instance 'circle :radius 1))
3.14...
```

我们同样可以让方法接受更多的参数：

```
(defmethod move ((c circle) dx dy)
  (incf (car (circle-center c)) dx)
  (incf (cdr (circle-center c)) dy)
  (circle-center c))
```

如果我们对一个 circle 的实例调用这个方法，circle 实例的中心会移动 ⟨dx,dy⟩：

```
> (move (make-instance 'circle :center '(1 . 1)) 2 3)
(3 . 4)
```

方法的返回值表明了圆形的新位置。

和我们的简装版实现一样，如果一个实例对应的类及其父类有个方法，那么调用这个方法会使最匹配的方法被调用。因此，如果 unit-circle 是 circle 的子类，同时具有如下所示的 area 方法：

```
(defmethod area ((c unit-circle)) pi)
```

那么当我们对一个 unit-circle 的实例调用 area 方法的时候，将被调用的不是更一般的那个方法，而是在上面定义 area。

当一个类有多个父类时，它们的优先级从左到右依次降低。patriotic-scoundrel 类的定义如下：

```
(defclass scoundrel nil nil)
(defclass patriot nil nil)
(defclass patriot-scoundrel (scoundrel patriot) nil nil)
```

我们认为爱国的无赖，他首先是一个无赖，然后才是一个爱国者。当对两个父类都有一个合用的方法时：

```
(defmethod self-or-country? ((s scoundrel))
  'self)

(defmethod self-or-country? ((p patriot))
  'country)
```

scoundrel 类的方法会这样被执行：

```
> (self-or-country? (make-instance 'patriotic-scoundrel))
SELF
```

到现在为止，几个例子都让人觉得，CLOS 中的方法是针对一个类的方法。实际上，CLOS 中的方法是更为通用的一个概念。在 move 方法的参数列表中，(c circle) 这一组被叫做特化 (specialized) 的参数，它表示，如果 move 的第一个参数是 circle 类的一个实例的话，就适用这个方法。对于 CLOS 方法，不止一个参数可以被特化。下面的方法就有两个特化参数和一个可选的非特化参数：

```
(defmethod combine ((ic ice-cream) (top topping)
                    &optional (where :here))
  (append (list (name ic) 'ice-cream)
          (list 'with (name top) 'topping)
          (list 'in 'a
                (case where
                  (:here 'glass)
                  (:to-go 'styrofoam))
                'dish)))
```

如果 combine 的前两个参数分别是 ice-cream 和 topping 的实例的话，上面定义的方法就会被调用。如果我们定义几个最简单类以便构造实例

```
(defclass stuff () ((name :accessor name :initarg :name)))
(defclass ice-cream (stuff) nil)
(defclass topping (stuff) nil)
```

那么我们就能定义并运行这个方法了：

```
> (combine (make-instance 'ice-cream :name 'fig)
           (make-instance 'topping :name 'olive)
           :here)
(FIG ICE-CREAM WITH OLIVE TOPPING IN A GLASS DISH)
```

倘若方法特化了一个以上的参数，这时就没有办法再把方法当成类的属性了。我们的 combine 方法是属于 ice-cream 类还是属于 topping 类呢？在 CLOS 里，所谓“对象响应消息”的模型不复存在。如果我们像下面那样调用函数，这种模型似乎还是顺理成章的：

```
(tell obj 'move 2 3)
```

可以很清楚地看到，在这里我们是在调用 obj 的 move 方法。但是一旦我们废弃这种语法，而改用函数风格的等价操作：

```
(move obj 2 3)
```

我们就需要定义 move，让它能根据它的第一个参数 *dispatch* 操作，即按照第一个参数的类型来调用适合的方法。

走出了这一步，于是有个问题浮出了水面：为什么只能根据第一个参数来进行 dispatch 呢？CLOS 的回答是：为什么不呢？在 CLOS 中，方法能够指定任意个数的参数进行特化，而且这并不限于用户自定义的类，Common Lisp 类型[5]也一样可以，甚至能针对单个的特定对象特化。下面是一个名为 combine的方法，它被用于字符串：

```
(defmethod combine ((s1 string) (s2 string) &optional int?)
  (let ((str (concatenate 'string s1 s2)))
    (if int? (intern str) str)))
```

[5]或者更准确地说，是 CLOS 定义的一系列形似类型的类，这些类的定义和 Common Lisp 的内建类型体系是平行对应的。

这不仅意味着方法不再是类的属性，而且还表明，我们可以根本不用定义类就能使用方法了。

```
> (combine "I am not a " "cook.")
"I am not a cook."
```

下面，第二个参数将对符号 palindrome 进行特化：

```
(defmethod combine ((s1 sequence) (x (eql 'palindrome))
                    &optional (length :odd))
  (concatenate (type-of s1)
               s1
               (subseq (reverse s1)
                       (case length (:odd 1) (:even 0)))))
```

上面的这个方法能生成任意元素序列的回文：[6]

```
> (combine '(able was i ere) 'palindrome)
(ABLE WAS I ERE I WAS ABLE)
```

到现在，我们讲述的内容已经超出了面向对象的范畴，这具有更普遍的意义。CLOS 在设计的时候就已经认识到，在对象方法的背后，更深层次的思想是分派 (dispatch) 的概念，即选择合适方法的依据可以不仅仅是单独的一个参数，而且可以基于多于一个参数的类型进行。当我们基于这种更通用的表示手段来构造方法时，方法就可以脱离特定的类而存在了。方法不再在逻辑上从属于类，它现在和其它的同名方法成为了一体。CLOS 把这样的一组方法称作 *generic 函数*。所有的 combine 方法隐式地定义了名为 combine 的 generic 函数。

我们可以显式地用 defgeneric 宏定义 generic 函数。虽然没有必要专门调用 defgeneric 来定义一个 generic 函数，但是这个定义却是一个安置文档，或者为一些错误加入保护措施的好地方。我们在下面的定义中两样都用上了：

```
(defgeneric combine (x y &optional z)
  (:method (x y &optional z)
           "I can't combine these arguments.")
  (:documentation "Combines things."))
```

由于这里为 combine 定义的方法没有特化任何参数，所以如果没有其它方法适用的话，这个方法就会被调用。

```
> (combine #'expt "chocolate")
"I can't combine these arguments."
```

倘若上面的方法没有被定义，这个调用就会产生一个错误。

generic 函数也加入了一个我们把方法当成对象属性时没有的限制：当所有的同名方法加盟一个 generic 方法时，这些同名方法的参数列表必须一致。这

[6] 在一个 Common Lisp 实现中 (否则这个实现就完美了)，concatenate 不会接受 cons 作为它的第一个参数，因此这个方法调用在这种情况下将无法正常工作。

就是为什么我们所有的 combine 方法都另有一个可选参数的原因。如果让第一个定义的 combine 方法接受三个参数，那么当我们试着去定义另一个只有两个参数的方法时，就会出错。

CLOS 要求所有同名方法的参数列表必须是一致的。两个参数列表取得一致的前提是：它们必须具有相同数量的必选参数，相同数量的可选参数，并且 &rest 和 &key 的使用也要相互兼容。不同方法最后用的关键字参数 (keyword parameter) 可以不一样，不过 defgeneric 会坚持要求让它的所有方法接受一个特定的最小集。下面每对参数列表，两两之间是相互一致的：

```
(x)             (a)
(x &optional y) (a &optional b)
(x y &rest z)   (a b &rest c)
(x y &rest z)   (a b &key c d)
```

而下列的每组都不一致：

```
(x)             (a b)
(x &optional y) (a &optional b c)
(x &optional y) (a &rest b)
(x &key x y)    (a)
```

重新定义方法就像重定义函数一样。由于只有必选参数才能被特化，每个方法都唯一地对应着它的 generic function 及其必选参数的类型。如果我们定义另一个有着相同特化参数的方法，那么新的方法就会覆盖原来的方法。因而，如果我们这样写道：

```
(defmethod combine ((x string) (y string)
                    &optional ignore)
  (concatenate 'string x " + " y))
```

那么就会重新定义当 combine 的头两个参数都是 string 时，这个方法的行为。

不幸的是，如果我们不希望重新定义方法，而是想删除它，CLOS 中并没有一个内建的 defmethod 的逆操作。万幸的是，这是 Lisp，所以我们可以自己写一个。如何手工删除一个方法的具体细节被记录在了图 25.12 中的 undefmethod 的实现里。就像调用 defmethod 时一样，我们在使用这个宏的时候，把参数传入这个宏，不过不同之处在于，这次我们并没有把整个的参数列表作为第二个或者第三个参数传进去，我们只是把必选参数的类名送入这个宏。所以，如果要删除两个 string 的 combine 方法，我们可以这样写：

```
(undefmethod combine (string string))
```

没有特化的参数被缺省指定为类 t，所以，如果我们之前定义了一个方法，而且这个方法有必选参数，但是这些参数没有特化的话：

```
(defmethod combine ((fn function) &optional y)
  (funcall fn x y))
```

```
(defmacro undefmethod (name &rest args)
  (if (consp (car args))
      (udm name nil (car args))
    (udm name (list (car args)) (cadr args))))

(defun udm (name qual specs)
  (let ((classes (mapcar #'(lambda (s)
                             `(find-class ',s))
                         specs)))
    `(remove-method (symbol-function ',name)
                    (find-method (symbol-function ',name)
                                 ',qual
                                 (list ,@classes)))))
```

图 25.12: 用于删除方法的宏。

我们可以用下面的语句把它去掉

```
(undefmethod combine (function t))
```

如果希望删除整个的 generic function，那么我们可以用和删除任意函数相同的方法来达到这个目的，即调用 `fmakunbound`：

```
(fmakunbound 'combine)
```

25.5 辅助方法和组合

在 CLOS 里，辅助函数还是和我们的精简版实现一样的运作。到现在，我们只看到了主方法，但是我们一样可以用 before、after 和 around 方法。可以通过在方法的名字后面加上限定关键字 (qualifying keyword)，来定义这些辅助函数。假如我们为 `speaker` 类定义一个主方法 `speak` 如下：

```
(defclass speaker nil nil)

(defmethod speak ((s speak) string)
  (format t "~A" string))
```

那么，对一个 `speaker` 的实例调用 `speak` 方法，就会把方法的第二个参数打印出来：

```
> (speak (make-instance 'speaker)
         "life is not what it used to be")
life is not what it used to be
NIL
```

现在定义一个名为 `intellectual` 的子类，让它把主方法 `speak` 用 before 和 after 方法包装起来，

```
(defclass intellectual (speaker) nil)

(defmethod speak :before ((i intellectual) string)
  (princ "Perhaps "))

(defmethod speak :after ((i intellectual) string)
  (princ " in some sense"))
```

然后，我们就能新建一个 speaker 的子类，让这个子类总是会自己加上最后一个 (以及第一个) 词：

```
> (speak (make-instance 'intellectual)
         "life is not what it used to be")
Perhaps life is not what it used to be in some sense
NIL
```

在标准的方法组合方式中，方法调用的顺序和我们精简版实现中规定的顺序是一样的：所有的 before 方法是从最匹配的开始，然后是最匹配的主方法，接着是 after 方法，after 方法是最匹配的最后才调用。因此，如果我们像下面这样为 `speaker` 超类定义 before 或者 after 方法，

```
(defmethod speak :before ((s speaker) string)
  (princ "I think "))
```

这些方法会在夹心饼干的中间被调用：

```
> (speak (make-instance 'intellectual)
         "life is not what it used to be")
Perhaps I think life is not what it used to be in some sense
NIL
```

无论被调用的是什么 before 或 after 方法，generic 函数的返回值总是最匹配的主方法的值，在本例中，返回的值就是 `format` 返回的 `nil`。

如果有 around 方法的话，这个论断就要稍加改动。倘若一个对象的继承树中有一个类具有 around 方法，或者更准确地说，如果有 around 方法特化了 generic 函数的某些参数，那么这个 around 方法会被首先调用，然后其余的这些方法是否会被运行将取决于这个 around 方法。在我们的精简版实现中，一个 around 方法或者主方法能够通过运行一个函数，调用下一个方法：我们以前定义的名为 `call-next` 在 CLOS 被叫做 `call-next-method`。与我们的 `next-p` 相对应，在 CLOS 中同样也有一个叫 `next-method-p` 的函数。有了 around 方法，我们可以定义 `speaker` 的另一个子类，这个子类说话会更慎重一些：

```
(defclass courtier (speaker) nil)

(defmethod speak :around ((c courtier) string)
  (format t "Does the King believe that ~A? " string)
```

```
  (if (eq (read) 'yes)
      (if (next-method-p) (call-next-method))
    (format t "Indeed, it is a preposterous idea.~%"))
  'bow)
```

当 speak 的第一个参数是个 courtier 实例时，这个 around 方法会帮弄臣把话说得更四平八稳：

```
> (speak (make-instance 'courtier) "kings will last")
Does the King believe that kings will last? yes
I think kings will last
BOW
> (speak (make-instance 'courtier) "the world is round")
Does the King believe that the world is round? no
Indeed, it is a preposterous idea.
BOW
```

可以注意到，和 before 和 after 方法，around 方法的返回值被作为 generic 函数的返回值返回了。

一般来说，方法调用的顺序如下所列，这些内容是从第 25.2 节里摘抄下来的：

1. 倘若有的话，先是最匹配的 around 方法
2. 否则的话，依次是：
   (a) 所有的 before 方法，从最匹配的到最不匹配的。
   (b) 最匹配的主方法 (这是我们以前会调用的)。
   (c) 所有的 after 方法，从最不匹配的到最匹配的。

这种组合方法的方式被称为标准的方法组合。和我们之前的简装版一样，这里一样有办法以其它的方式组合方法。比如说，让一个 generic 函数返回所有可用的主方法返回值之和。

在我们的程序里，我们通过调用 defcomb 来指定组合方法的方式。缺省情况下，方法是以上面列出的规则调用的，不过如果我们像这样写的话：

```
(defcomb price #'+)
```

就能让 price 这个函数返回所有适用主方法的和。

在 CLOS 中这被称为操作符方法组合。在我们的程序里，这个方法组合的效果就好像对这样一个 Lisp 表达式求值：该表达式中的第一个元素是某个操作符，传给操作符的参数是对所有适用主方法的调用，而调用的顺序是按照匹配程度从高到低的。如果我们定义 price 的 generic 函数，让它使用 + 来组合返回值，同时假设 price 没有适用的 around 方法，那么调用 price 的效果就如同它是用下面的语句定义的：

```
(defun price (&rest args)
  (+ (apply ⟨most specific primary method⟩ args)
     ⋮
     (apply ⟨most specific primary method⟩ args)))
```

如果有适用的 around 方法的话，它们有更高的优先级，这和标准方法组合是一样的。在操作符方法组合里，around 方法仍然可以通过 `call-next-method` 来调用下一个方法。不过在这里主方法就不能调用 `call-next-method` 了。(这一点是和精简版的不同之处，在精简版里，我们是允许主方法调用 `call-next` 的。)

在 CLOS 里，我们可以对一个 generic 函数指定它所使用的方法组合类型，传给 `defgeneric` 的缺省参数 `:method-combination` 就是用来实现这一功能的。如下所示：

```
(defgeneric price (x)
  (:method-combination +))
```

现在这个 `price` 方法就会用 `+` 这种方法组合了。如果我们定义几种有价格的类，

```
(defclass jacket nil nil)
(defclass trousers nil nil)
(defclass suit (jacet trousers) nil)

(defmethod price + ((jk jacket)) 350)
(defmethod price + ((tr trousers)) 200)
```

那么当我们要知道一个 `suit` 实例的价格时，就会得到各个适用的 `price` 方法之和：

```
> (price (make-instance 'suit))
550
```

下面所列的符号可以被用作 `defmethod` 的第二个参数，同时它们也可以用在 `defgeneric` 的 `:method-combination` 选项上：

```
+   and   append   list   max   min   nconc   or   progn
```

用 `define-method-combination`，你可以自己定义其它的方法组合方式：参见 CLTL2，第 830 页。

你一旦定义了一个 generic 函数要使用的方法组合方式，那么所有这个函数对应的方法就必须使用和你所指定的方式相同类型的方法组合。如果我们试图把其它操作符 (或 `:before` 和 `:after`) 用作 `price`的 `defmethod` 方法里的第二个参数，就会导致错误。倘若我们一定要改变 `price` 的方法组合方式的话，我们只能通过 `fmakunbound` 来删除整个 `price` 的 generic 函数.

25.6 CLOS 与 Lisp

CLOS 为嵌入式语言树立了一个好榜样。这种编程方式有两大好处：

1. 嵌入式语言在概念上可以很好地与它们所处的领域很好融合在一起，因此在嵌入式语言中，我们得以继续以原来的术语来思考程序代码。

2. 嵌入式语言可以是非常强大的，因为它们可以利用被作为基础的那门语言已有的全部本事。

CLOS 把这两点都占全了。它和 Lisp 集成得天衣无缝，同时，它灵活地运用了 Lisp 中已有的抽象机制。事实上，我们可以透过 CLOS 可以看出 Lisp 的神韵，就像物件上虽然蒙着薄布，我们还是能看出来它的形状一样。

我们与 CLOS 沟通交互的渠道是一层宏，这并不是个巧合。宏是用来转换程序的，而从本质上说，CLOS 就是一个程序，它把用面向对象的抽象形式编写而成的程序翻译转换成为用 Lisp 的抽象形式构造而成的程序。

正如本章前两节所展示的，由于面向对象编程的抽象形式能被如此清晰简洁地实现成基于 Lisp 的抽象形式，我们几乎可以把前者说成后者的一个特殊形式了。我们能毫不费力地把面向对象编程里的对象实现成 Lisp 对象，把对象的方法实现为词法闭包。利用这种同构性，我们得以用区区几行代码实现了一个面向对象编程的初步框架，用寥寥几页篇幅就容下了一个 CLOS 的简单实现。

虽然 CLOS 和我们的简单实现相比，其规模要大很多，功能也强了很多，但是它还没有大到能把其根基伪装成一门嵌入式语言。以 `defmethod` 为例。虽然 CLTL2 没有明确地提出，但是 CLOS 的方法具有词法闭包的所有能力。如果我们在某个变量的作用域内定义几个方法：

```
(let ((transactions 0))
  (defmethod withdraw ((a account) amt)
    (incf transactions)
    (decf (balance a) amt))
  (defmethod deposit ((a account) amt)
    (incf transactions)
    (incf (balance a) amt))
  (defun transactions ()
    transactions))
```

那么在运行时，它们就会像闭包一样，共享这个变量。这些方法之所以会这样是因为，在语法带来的表象之下，它们就是闭包。如果观察一下 `defmethod` 的展开式，可以发现它的程序体被原封不动地保存在了井号–引号里的 lambda 表达式中。

第 7.6 节中曾提到，思忖宏的运行方式比考虑它们是什么意思要容易些。与之相似，理解 CLOS 的法门在于弄清 CLOS 是如何映射到 Lisp 基本的抽象形式中的。

25.7 何时用对象

面向对象的风格有几个明显的好处。不同的程序希望在不同程度上从中受益。这些情况有两种趋势。一种情况，有的程序，比如说一些模拟程序，如果用面向对象编程的抽象形式来表达它们是最为自然的。而另外一种程序之所以选用面向对象的风格来编写，主要的原因是希望让程序的可扩展性更好些。

可扩展性的确是面向对象编程带来的巨大好处之一。程序不再被写成囫囵的一团，而是分成小块，每个部分都以自己的功用命名。所以如果事后有其他人需要修改这个程序的话，他就能很方便地找到需要改动的那部分代码。倘若我们希望 ob 类型的对象显示在屏幕上的样子有所改变的话，我们可以修改 ob 类的 display 方法。要是我们希望创建一个类，让这个类的实例与 ob 的实例大体一样，只在某些方面有些差异，那么我们可以从 ob 派生一个子类，在这个子类里面，我们仅仅修改我们想要的那些属性，其它所有的东西都会从 ob 类缺省地继承得到。如果我们只是想让某一个 ob 对象的行为和其它 ob 对象有些不一样，可以就新建一个 ob 对象，然后直接修改这个对象的属性。倘若要修改的程序原来写得很认真，那么我们就可以在完成上述各种修改的同时，甚至不用看程序中其它的代码一眼。从这个角度上来说，以面向对象的思想写出的程序就像被组织成表格一样：只要找到对应的单元格，我们就可以迅速安全地修改程序。

对于扩展性来说，它从面向对象风格得到的东西是最少的。实际上，为了要实现可扩展性，基本上不需要什么外部的支持，所以，一个可扩展的程序完全可以不写成面向对象的。如果要说前面的几章说明了什么道理的话，那就是 Lisp 程序是可以不用写为囫囵一团的。Lisp 给出了全系列的实现扩展性的方案。比如说，你可以把程序实现成一张表格：即一个由保存在数组里的闭包构成的程序。

假如你想要的就是可扩展性，那么你大可不必在“面向对象”编程和“传统”形式的编程中两者取其一。你常常可以不依赖面向对象的技术，就能赋予一个 Lisp 程序它所需要的可扩展性，不多也不少。属于类的 slot 是一种全局变量。在本可以用使用参数的地方，却要用全局变量，我们知道这样做有些不合适。和这种情形有几分相似，如果本来可以用原始的 Lisp 就轻松完成的程序，偏要写成一堆类和实例，这样做或许也不是很妥当。有了 CLOS，Common Lisp 已经成为了被广泛使用的最强大的面向对象语言。有讽刺意味的是，Common Lisp 也是一门对它来说，面向对象编程最为无足轻重的语言。

附录: 包 (packages)

包 (packages)，是 Common Lisp 把代码组织成模块的方式。早期的 Lisp 方言有一张符号表，即 *oblist*。在这张表里列出了系统中所有已经读取到的符号。借助 oblist 里的符号表项，系统得以存取数据，诸如对象的值，以及属性列表等。被保存在 oblist 里的符号被称为 *interned*。

晚近的 Lisp 方言把 oblist 的概念放到了一个个包里面。现在，符号不仅仅是被 intern 了，而是被 intern 在某个包里。包之所以支持模块化是因为在一个包里的 intern 的符号只有在其被显式申明为能被其它包访问的时候，它才能为外部访问 (除非用一些歪门邪道的招数)。

包是一种 Lisp 对象。当前包常常被保存在一个名为 `*package*` 的全局变量里面。当 Common Lisp 启动时，当前包就是用户包：或者叫 `user` (CLTL1 实现)，或者叫 `common-lisp-user` (CLTL2 实现)。

包一般使用它们的名字相互区别，而这些名字采用的是字符串的形式。要知道当前包的包名，可以试试：

```
> (package-name *package*)
"COMMON-LISP-USER"
```

通常，当一个符号被读入的时候，它就被 intern 到当前的包里了。要弄清给定符号所 intern 的是哪个包，我们可以用 `symbol-package`：

```
> (symbol-package 'foo)
#<Package "COMMON-LISP-USER" 4CD15E>
```

这个返回值是实际的包对象。为了下面要使用，我们给 `foo` 赋一个值：

```
> (setq foo 99)
99
```

使用 `in-package`，我们就可以切换到另一个新的包，如果需要的话这个包会被创建出来[7]：

```
> (in-package 'mine :use 'common-lisp)
#<Package "MINE\ 63390E>
```

此时此刻应该会响起诡异的背景音乐，因为我们已经身处另一个世界：在这里 `foo` 已经不似从前了：

```
MINE> foo
>>Error: FOO has no global value.
```

[7]在较早期的 Common Lisp 实现下，请省略掉 `:use` 参数

为什么会这样？因为之前被我们设置成 99 的那个 foo 和现在 mine 里面的这个 foo 是两码事。[8]要从用户包之外引用原来的这个 foo，我们必须把包名和两个冒号作为它的前缀：

```
MINE> common-lisp-user::foo
99
```

因此，具有相同打印名称的不同符号得以在不同包中共存。这样，就可以在名为 common-lisp-user 的包里有一个 foo，同时在 mine 包里也有一个 foo，并且它们两个是不一样符号。实际上，这就是 package 的一部分用意所在，即：你在为你的函数和变量取名字的同时，就不用担心别人会把一样的名字用在其它东西上。现在，就算有重名的情况，重名的符号之间也是互不相干的。

与此同时，包也提供了一种信息隐藏的手段。对程序来说，它必须使用名字来引用不同的函数和变量。如果你不让一个名字在你的包之外可见的话，那么另一个包中的代码就无法使用或者修改这个名字所引用的对象。

在写程序的时候，把包的名字带上两个冒号做为前缀用不是一个好习惯。你要是这样做的话，就违背了模块化设计的初衷，而这正是包本来想要提供的。如果你不得不使用双冒号来引用一个符号，这应该就是有人根本就不希望你引用它。

一般来说，如果你只应该引用那些被 *expert* 了的那些符号。把符号从它所属的包 export 出来，我们就能让这个符号对其它包变得可见。要引出一个符号，我们可以调用 (你肯定已经猜到了) export：

```
MINE> (in-package 'common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (export 'bar)
T
> (setq bar 5)
5
```

如果把 bar *import* 进 mine 我们就能更进一步，让 mine 能和 user package 共享 bar 这个符号：

```
MINE> (import 'common-lisp-user:bar)
T
MINE> bar
5
```

在导入了 bar 之后，我们可以完全不用加上任何包的限定符就能引用它了。现在，这两个包共享了同一个符号——再没有一个独立的 mine:bar 了。

万一已经有了一个会怎么样呢？在这种情况下，import 调用会导致一个错误，就像下面我们试着 import foo 时造成的错误一样：

[8]有的 Common Lisp 实现会在 toplevel 提示符的前面显示包的名字。这个特性不是必须的，但的确是比较贴心的设计。

```
MINE> (import 'common-lisp-user::foo)
>>Error: FOO is already present in MINE.
```

之前，我们在 mine 里对 foo 进行了一次不成功的求值，这次求值顺带着使得一个名为 foo 的符号被加入了 mine。由于这个符号在全局范围内还没有值，因此产生了一个错误，但是输入符号名字的直接后果就是使它被 intern 进了这个包。所以，当我们现在想把 foo 引进 mine 的时候，mine 里面已经有一个相同名字的符号了。

通过让一个包使用 (use) 另一个包，我们也能批量的引入符号：

```
MINE> (use-package 'common-lisp-user)
T
```

这样，所有 user package 引出的符号就会自动地被引进到 mine 里面去了。(要是 user package 已经引出了 foo 的话，这个函数调用也会出一个错。)

如 CLTL1 所言，包含内建操作符和变量名字的包被称为 common-lisp 而不是 lisp，因此新一些的包在缺省情况下已不再使用 lisp 包了。由于我们通过调用 in-package 创建了 mine，而在这次调用中也 use 了这个包，所以所有 Common Lisp 的名字在 mine 中都是可见的：

```
MINE> #'cons
#<Compiled-Function CONS 462A3E>
```

在实际的编程中，你不得不让所有新编写的包使用 common-lisp (或者其他某个含 Lisp 操作符的包)。否则你甚至会没办法跳出这个新的包。[9]

一般来说，在编译后的代码中，不会像刚才这样在顶层进行包的操作。更多的时候，这些关于包的函数调用会被包含在源文件中。通常，只要把 in-package 和 defpackage 放在源文件的开头就可以了。(defpackage 宏是 CLTL2 里新引进的，但是有些较老的实现也提供了它。) 如果你要编写一个独立的包，下面列出了你可能会放在对应的源文件最开始地方的代码：

```
(in-package 'my-application :use 'common-lisp)
(defpackage my-application
            (:use common-lisp my-utilities)
            (:nicknames app)
            (:export win lose draw))
```

这会使得该文件里所有的代码，或者更准确地说，文件里所有的名字，都纳入了 my-application 这个包。my-application 同时使用了 common-lisp 和 my-utilities，因此，不用加任何包名作为前缀，所有被引出的符号都可以直接使用。

my-application 本身仅仅引出了三个符号，它们分别是：win、lose 和 draw。由于在调用 in-package 的时候，我们给 my-application 取了

[9]译者注：即你不仅没有办法使用 cons，更糟糕的是，你也不能用 in-package 切换到其它包。

一个绰号 app，在其它包里面的代码可以用类似 app:win 的名字来引用这些符号。

像这样的用包来提供的模块化的确有点不自然。我们的包里面不是对象，而是一堆名字。每个使用 common-lisp 的包都引入了 cons 这个名字，原因在于 common-lisp 包含了一个叫这个名字的函数。但是，这样会导致一个名字叫 cons 的变量也在每个使用 common-lisp 可见。这样的事情同样也会在 Common Lisp 的其他名字空间重演。如果包(package) 这个机制让你头痛，那么这就是一个最主要的原因——包不是基于对象的，它们基于名字。

和包相关的操作会发生在读取时(read-time)，而非运行时。这可能会造成一些困扰。我们输入的第二个表达式：

```
(symbol-package 'foo)
```

之所以会返回它返回的那个值是因为：读取这个查询语句的同时，答案就被生成了。为了求值这个表达式，Lisp 必须先读入它，这意味着要 intern foo。

再来个例子，看看下面把两个表达式交换顺序的结果，这两个表达式上面曾出现过：

```
MINE> (in-package 'common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (export 'bar)
```

通常来说，在顶层输入两个表达式的效果等价于把这两个表达式放在一个 progn里面。不过这次有些不同。如果我们这样说

```
MINE> (progn (in-package 'common-lisp-user)
                (export 'bar))
>>Error: MINE::BAR is not accessible in COMMON-LISP-USER.
```

我们则会得到个错误提示。错误的原因在于 progn 表达式在求值之前就已经被 read 处理过了。当调用 read 时，当前包还是 mine，因而 bar 被认为是 mine:bar。运行这个表达式的效果就好像我们想要 export mine:bar，而不是从user 包里 common-lisp-user:bar。

package 被如此定义，使得编写那些把符号当作数据的程序成为一桩麻烦事。举个例子，要是我们像下面那样定义 noise：

```
(in-package 'other :use 'common-lisp)
(defpackage other
               (:use common-lisp)
               (:export noise))
(defun noise (animal)
  (case animal
     (dog 'woof)
     (cat 'meow)
     (pig 'oink)))
```

这样的话，如果我们从另外一个包调用 noise，同时传进去的参数是不恰当的符号，noise 会走到 case 语句的末尾，并且返回 nil：

```
OTHER> (in-package 'common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (other:noise 'pig)
NIL
```

这是因为我们传进去的参数是 common-lisp-user:pig (这不是想冒犯你)，然而 case 接受 key 是 other:pig。为了让 noise 如所想的那样工作，我们必须把里面用到的所有六个符号都引出来，再在我们调用 noise 的包里面引入它们。

在这个例子里面，我们也可以通过使用关键字而不是常规的符号，来绕过这个问题。倘若 noise 像下面这样定义

```
(defun noise (animal)
 (case animal
   (:dog :woof)
   (:cat :meow)
   (:pig :oink)))
```

这样我们就能从任意一个包调用安全地调用这个函数了：

```
OTHER> (in-package 'common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (other:noise :pig)
:OINK
```

关键字就像金子：普适而且自身就能表明其价值。它们不论在哪里都是可见的，而且从来就没有要加上引用才能使用它们的必要。符号驱动的函数几乎总是应当使用关键字编写而成，比如 defamaph (第 ?? 页)。

包里面有很多地方让人困惑不解。这里对这一主题的介绍仅仅是冰山一角。要知道所有的细节，请参考 CLTL2，第 11 章。

Notes

索　引