ANSI Common Lisp 中文翻譯版

習題參考解答

P.Graham “ANSI Common LISP” Answer for Practice [http://www.shido.info/lisp/pacl2_e.html] ── by SHIDO, Takafumi (takafumi@shido.info)

譯文詞彙

本書詞彙部分參考計算機科學詞彙表 [http://github.com/JuanitoFatas/Computer-Science-

Glossary]。

相關閱讀

Chris Riesbeck 關於本書的剖析 [http://www.cs.northwestern.edu/academics/courses/325/readings/graham/graham-notes.html]

Google Common Lisp 風格指南 [http://juanitofatas.com/Google-Common-Lisp-Style-Guide/GoogleCLSG-zhCN.xml]。

下載離線版本

注意，因為文檔總是在不斷地更新和修正當中，請定期下載最新的離線文檔，確保獲得最好的閱讀體驗。

HTML 格式下載 [https://media.readthedocs.org/htmlzip/ansi-common-lisp/latest/ansi-common-lisp.zip]。

樣式

使用 huangz1990 [http://huangz.me] 所開發的 der [https://github.com/huangz1990/der] 樣式。

貢獻

請至 GitHub Repo [https://github.com/acl-translation/acl-chinese]。

正體中文

前言

本書的目的是快速及全面的教你 Common Lisp 的有關知識。它實際上包含兩本書。前半部分用大量的例子來解釋 Common Lisp 裡面重要的概念。後半部分是一個最新 Common Lisp 辭典，涵蓋了所有 ANSI Common Lisp 的運算子。

這本書的目標讀者

ANSI Common Lisp 這本書適合學生或者是專業的程式設計師去讀。本書假設讀者閱讀前沒有 Lisp 的相關知識。有別的程式語言的編程經驗也許對讀本書有幫助，但也不是必須的。本書從解釋 Lisp 中最基本的概念開始，並對於 Lisp 最容易迷惑初學者的地方進行特別的強調。

本書也可以作爲教授 Lisp 編程的課本，也可以作爲人工智能課程和其他編程語言課程中，有關 Lisp 部分的參考書。想要學習 Lisp 的專業程式設計師肯定會很喜歡本書所採用的直截了當、注重實踐的方法。那些已經在使用 Lisp 編程的人士將會在本書中發現許多有用的實體，此外，本書也是一本方便的 ANSI Common Lisp 參考書。

如何使用這本書

學習 Lisp 最好的辦法就是拿它來編程。況且在學習的同時用你學到的技術進行編程，也是非常有趣的一件事。編寫本書的目的就是讓讀者儘快的入門，在對 Lisp 進行簡短的介紹之後， 第 2 章開始用 21 頁的內容，介紹了著手編寫 Lisp 程式時可能會用到的所有知識。 3-9 章講解了 Lisp 裡面一些重要的知識點。這些章節特彆強調了一些重要的概念，比如指標在 Lisp 中扮演的角色，如何使用遞迴來解決問題，以及第一級函數的重要性等等。

針對那些想要更深入了解 Lisp 的讀者： 10-14 章包含了宏、CLOS、列表操作、程式優化，以及一些更高級的課題，比如包和讀取宏。

15-17 章通過 3 個 Common Lisp 的實際應用，總結了之前章節所講解的知識：一個是進行邏輯推理的程式，另一個是 HTML 生成器，最後一個是針對物件導向程式設計的嵌入式語言。

本書的最後一部分包含了 4 個附錄，這些附錄應該對所有的讀者都有用： 附錄 A-D 包括了一個如何除錯程式的指南， 58 個 Common Lisp 運算子的源程式，一個關於 ANSI Common Lisp 和以前的 Lisp 語言區別的總結，以及一個包括所有 ANSI Common Lisp 的參考手冊。

本書還包括一節備註。這些備註包括一些說明，一些參考條目，一些額外的程式，以及一些對偶然出現的不正確表述的糾正。備註在文中用一個小圓圈來表示，像這樣：○

Tip: 譯註: 由於小圈圈 ○ 實在太不明顯了，譯文中使用 λ 符號來表示備註。

λ [http://ansi-common-lisp.readthedocs.org/en/latest/zhCN/notes-cn.html#viii-notes-viii]

原始碼

雖然本書介紹的是 ANSI Common Lisp ，但是本書中的程式可以在任何版本的 Common Lisp 中運行。那些依賴 Lisp 語言新特性的例子的旁邊，會有註釋告訴你如何把它們運行於舊版本的 Lisp 中。

本書中所有的程式都可以在互聯網上下載到。你可以在網路上找到這些程式，它們還附帶著一個免費軟體的連結，一些過去的論文，以及 Lisp 的 FAQ 。還有很多有關 Lisp 的資源可以在此找到：http://www.eecs.harvard.edu/onlisp/ 原始碼可以在此 FTP 服務器上下載： ftp://ftp.eecs.harvard.edu:/pub/onlisp/ 讀者的問題和意見可以發送到 pg@eecs.harvard.edu 。

Note: 譯註：下載的連結都壞掉了，本書的原始碼可以到此下載：https://raw.github.com/acl-translation/acl-chinese/master/code/acl2.lisp

On Lisp

在整本 On Lisp 書中，我一直試著指出一些 Lisp 獨一無二的特性，這些特性使得 Lisp 更像 “Lisp” 。並展示一些 Lisp 能讓你完成的新事情。比如說宏： Lisp 程式設計師能夠並且經常編寫一些能夠寫程式的程式。對於程式生成程式這種特性，因爲 Lisp 是主流語言中唯一一個提供了相關抽象使得你能夠方便地實現這種特性的編程語言，所以 Lisp 是主流語言中唯一一個廣泛運用這個特性的語言。我非常樂意邀請那些想要更進一步了解宏和其他高級 Lisp 技術的讀者，讀一下本書的姐妹篇：On Lisp [http://www.paulgraham.com/onlisp.html]。

Note: On Lisp 已經由知名 Lisp 黑客 —— 田春 —— 翻譯完成，可以在網路上找到。 —— 田春（知名 Lisp 黑客、Practical Common Lisp 譯者）

鳴謝

在所有幫助我完成這本的朋友當中，我想特別的感謝一下 Robert Morris 。他的重要影

響反應在整本書中。他的良好影響使這本書更加優秀。本書中好一些實體程式都源自他手。這些程式包括 138 頁的 Henley 和 249 頁的模式匹配器。

我很高興能有一個高水平的技術審稿小組：Skona Brittain, John Foderaro, Nick Levine, Peter Norvig 和 Dave Touretzky。本書中幾乎所有部分都得益於它們的意見。 John Foderaro 甚至重寫了本書 5.7 節中一些程式。

另外一些人通篇閱讀了本書的手稿，它們是：Ken Anderson, Tom Cheatham, Richard Fateman, Steve Hain, Barry Margolin, Waldo Pacheco, Wheeler Ruml 和 Stuart Russell。特別要提一下，Ken Anderson 和 Wheeler Ruml 給予了很多有用的意見。

我非常感謝 Cheatham 教授，更廣泛的說，哈佛，提供我編寫這本書的一些必要條件。另外也要感謝 Aiken 實驗室的人員：Tony Hartman, Dave Mazieres, Janusz Juda, Harry Bochner 和 Joanne Klys。

我非常高興能再一次有機會和 Alan Apt 合作。還有這些在 Prentice Hall 工作的人士：Alan, Mona, Pompili Shirley McGuire 和 Shirley Michaels, 能與你們共事我很高興。

本書用 Leslie Lamport 寫的 LaTeX 進行排版。LaTeX 是在 Donald Knuth 編寫的 TeX 的基礎上，又加了 L.A.Carr, Van Jacobson 和 Guy Steele 所編寫的宏完成。書中的圖表是由 John Vlissides 和 Scott Stanton 編寫的 Idraw 完成的。整本書的預覽是由 Tim Theisen 寫的 Ghostview 完成的。 Ghostview 是根據 L. Peter Deutsch 的 Ghostscript 創建的。

我還需要感謝其他的許多人，包括：Henry Baker, Kim Barrett, Ingrid Bassett, Trevor Blackwell, Paul Becker, Gary Bisbee, Frank Deutschmann, Frances Dickey, Rich 和 Scott Draves, Bill Dubuque, Dan Friedman, Jenny Graham, Alice Hartley, David Hendler, Mike Hewett, Glenn Holloway, Brad Karp, Sonya Keene, Ross Knights, Mutsumi Komuro, Steffi Kutzia, David Kuznick, Madi Lord, Julie Mallozzi, Paul McNamee, Dave Moon, Howard Mullings, Mark Nitzberg, Nancy Parmet 和其家人, Robert Penny, Mike Plusch, Cheryl Sacks, Hazem Sayed, Shannon Spires, Lou Steinberg, Paul Stoddard, John Stone, Guy Steele, Steve Strassmann, Jim Veitch, Dave Watkins, Idelle and Julian Weber, the Weickers, Dave Yost 和 Alan Yuille。

另外，著重感謝我的父母和 Jackie。

高德納 [http://zh.wikipedia.org/zh-cn/%E9%AB%98%E5%BE%B7%E7%BA%B3]給他的經典叢書起名爲《計算機程式設計藝術》。在他的圖靈獎獲獎感言中，他解釋說這本書的書名源自於內心深處的潛意識 —— 潛意識告訴他，編程其實就是追求編寫最優美的程式。

就像建築設計一樣，編程既是一門工程技藝也是一門藝術。一個程式要遵循數學原理也要符合物理定律。但是建築師的目的不僅僅是建一個不會倒塌的建築。更重要的是，他

們要建一個優美的建築。

像高德納一樣，很多程式設計師認爲編程的真正目的，不僅僅是編寫出正確的程式，更重要的是寫出優美的程式。幾乎所有的 Lisp 黑客也是這麼想的。 Lisp 黑客精神可以用兩句話來概括：編程應該是有趣的。程式應該是優美的。這就是我在這本書中想要傳達的精神。

保羅•葛拉漢姆 (Paul Graham) [http://paulgraham.com/]

第一章：簡介

約翰麥卡錫 [http://zh.wikipedia.org/zh-cn/%E7%BA%A6%E7%BF%B0%C2%B7%E9%BA%A6%E5%8D%A1%E9%94%A1]和他的學生於 1958 年展開 Lisp 的初次實現工作。 Lisp 是繼 FORTRAN 之後，仍在使用的最古老的程式語言。 λ [http://acl.readthedocs.org/en/latest/zhTW/notes.html#notes-1] 更值得注意的是，它仍走在程式語言技術的最前面。懂 Lisp 的程式設計師會告訴你，有某種東西使 Lisp 與衆不同。

Lisp 與衆不同的部分原因是，它被設計成能夠自己進化。你能用 Lisp 定義新的 Lisp 運算子。當新的抽象概念風行時（如物件導向程式設計），我們總是發現這些新概念在 Lisp 是最容易來實現的。Lisp 就像生物的 DNA 一樣，這樣的語言永遠不會過時。

1.1 新的工具（New Tools）

爲什麼要學 Lisp？因爲它讓你能做一些其它語言做不到的事情。如果你只想寫一個函數來返回小於 n 的數字總和，那麼用 Lisp 和 C 是差不多的：

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

如果你只想做這種簡單的事情，那用什麼語言都不重要。假設你想寫一個函數，輸入一個數 n ，返回把 n 與傳入參數 (argument)相加的函數。

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

在 C 語言中 addn 怎麼實現？你根本寫不出來。

你可能會想，誰會想做這樣的事情？程式語言教你不要做它們沒有提供的事情。你得針對每個程式語言，用其特定的思維來寫程式，而且想得到你所不能描述的東西是很困難的。當我剛開始編程時 —— 用 Baisc —— 我不知道什麼是遞迴，因爲我根本不知道有這個東西。我是用 Basic 在思考。我只能用迭代的概念表達算法，所以我怎麼會知道遞迴呢？

如果你沒聽過詞法閉包 「Lexical Closure」 [http://zh.wikipedia.org/zh-cn/%E9%97%AD%E5%8C%85_(%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E
(上述 addn 的範例)，相信我， Lisp 程式設計師一直在使用它。很難找到任何長度的 Common Lisp 程式，沒有用到閉包的好處。在 112 頁前，你自己會持續使用它。

閉包僅是其中一個我們在別的語言找不到的抽象概念之一。另一個更有價值的 Lisp 特點是， Lisp 程式是用 Lisp 的資料結構來表示。這表示你可以寫出會寫程式的程式。人們真的需要這個嗎？沒錯 —— 它們叫做宏，有經驗的程式設計師也一直在使用它。學到 173 頁你就可以自己寫出自己的宏了。

有了宏、閉包以及運行期型別，Lisp 凌駕在物件導向程式設計之上。如果你了解上面那句話，也許你不應該閱讀此書。你得充分了解 Lisp 才能明白爲什麼此言不虛。但這不是空泛之言。這是一個重要的論點，並且在 17 章用程式相當明確的證明了這點。

第二章到第十三章會循序漸進地介紹所有你需要理解第 17 章程式的概念。你的努力會有所回報：你會感到在 C++ 編程是窒礙難行的，就像有經驗的 C++ 程式設計師用 Basic 編程會感到窒息一樣。更加鼓舞人心的是，如果我們思考爲什麼會有這種感覺。 編寫 Basic 對於平常用 C++ 編程是令人感到窒息的，是因爲有經驗的 C++ 程式設計師知道一些用 Basic 不可能表達出來的技術。同樣地，學習 Lisp 不僅教你學會一門新的語言 —— 它教你嶄新的並且更強大的程式思考方法。

1.2 新的技術（New Techniques）

如上一節所提到的， Lisp 賦予你別的語言所沒有的工具。不僅僅如此，就 Lisp 帶來的新特性來說 —— 自動記憶體管理 (automatic memory management)，顯式型別 (manifest typing)，閉包 (closures)等 —— 每一項都使得編程變得如此簡單。結合起來，它們組成了一個關鍵的部分，使得一種新的編程方式是有可能的。

Lisp 被設計成可擴展的：讓你定義自己的運算子。這是可能的，因爲 Lisp 是由和你程式一樣的函數與宏所構成的。所以擴展 Lisp 就和寫一個 Lisp 程式一樣簡單。事實上，它是如此的容易（和有用），以至於擴展語言自身成了標準實踐。當你在用 Lisp 語言編程時，你也在創造一個適合你的程式的語言。你由下而上地，也由上而下地工作。

幾乎所有的程式，都可以從訂作適合自己所需的語言中受益。然而越複雜的程式，由下而上的程式設計就顯得越有價值。一個由下而上所設計出來的程式，可寫成一系列的層，每層擔任上一層的程式語言。 TeX [http://en.wikipedia.org/wiki/TeX] 是最早使用這種方法所寫的程式之一。你可以用任何語言由下而上地設計程式，但 Lisp 是本質上最適合這種方法的工具。

由下而上的編程方法，自然發展出可擴展的軟體。如果你把由下而上的程式設計的原則，想成你程式的最上層，那這層就成爲使用者的程式語言。正因可擴展的思想深植於

Lisp 當中，使得 Lisp 成爲實現可擴展軟體的理想語言。三個 1980 年代最成功的程式提供 Lisp 作爲擴展自身的語言: GNU Emacs [http://www.gnu.org/software/emacs/]， Autocad [http://www.autodesk.com.tw/adsk/servlet/pc/index?siteID=1170616&id=14977606] ，和 Interleaf [http://en.wikipedia.org/wiki/Interleaf] 。

由下而上的編程方法，也是得到可重用軟體的最好方法。寫可重用軟體的本質是把共同的地方從細節中分離出來，而由下而上的編程方法本質地創造這種分離。與其努力撰寫一個龐大的應用，不如努力創造一個語言，用相對小的努力在這語言上撰寫你的應用。和應用相關的特性集中在最上層，以下的層可以組成一個適合這種應用的語言 —— 還有什麼比程式語言更具可重用性的呢？

Lisp 讓你不僅編寫出更複雜的程式，而且寫的更快。 Lisp 程式通常很簡短 —— Lisp 給了你更高的抽象化，所以你不用寫太多程式碼。就像 Frederick Brooks [http://en.wikipedia.org/wiki/Fred_Brooks] 所指出的，編程所花的時間主要取決於程式的長度。因此僅僅根據這個單獨的事實，就可以推斷出用 Lisp 編程所花的時間較少。這種效果被 Lisp 的動態特點放大了：在 Lisp 中，編輯-編譯-測試迴圈短到使編程像是即時的。

更高的抽象化與互動的環境，能改變各個機構開發軟體的方式。術語快速建型描述了一種始於 Lisp 的編程方法：在 Lisp 裡，你可以用比寫規格說明更短的時間，寫一個原型出來，而這種原型是高度抽象化的，可作爲一個比用英語所寫的更好的規格說明。而且 Lisp 讓你可以輕易的從原型轉成產品軟體。當寫一個考慮到速度的 Common Lisp 程式時，通過現代編譯器的編譯，Lisp 與其他的高階語言所寫的程式運行得一樣快。

除非你相當熟悉 Lisp ，這個簡介像是無意義的言論和冠冕堂皇的宣告。*Lisp* 凌駕物件導向程式設計？ 你創造適合你程式的語言？ *Lisp* 編程是即時的？ 這些說法是什麼意思？現在這些說法就像是枯竭的湖泊。隨著你學到更多實際的 Lisp 特色，見過更多可運行的程式，這些說法就會被實際經驗之水所充滿，而有了明確的形狀。

1.3 新的方法（New Approach）

本書的目標之一是不僅是教授 Lisp 語言，而是教授一種新的編程方法，這種方法因爲有了 Lisp 而有可能實現。這是一種你在未來會見得更多的方法。隨著開發環境變得更強大，程式語言變得更抽象， Lisp 的編程風格正逐漸取代舊的規劃-然後-實現 (*plan-and-implement*)的模式。

在舊的模式中，錯誤永遠不應該出現。事前辛苦訂出縝密的規格說明，確保程式完美的運行。理論上聽起來不錯。不幸地，規格說明是人寫的，也是人來實現的。實際上結果是， 規劃-然後-實現 模型不太有效。

身爲 OS/360 的項目經理， Frederick Brooks [http://en.wikipedia.org/wiki/Fred_Brooks] 非

常熟悉這種傳統的模式。他也非常熟悉它的後果：

> 任何 OS/360 的用戶很快的意識到它應該做得更好...再者，產品推遲，用了更多的記憶體，成本是估計的好幾倍，效能一直不好，直到第一版後的好幾個版本更新，效能才算還可以。

而這卻描述了那個時代最成功系統之一。

舊模式的問題是它忽略了人的侷限性。在舊模式中，你打賭規格說明不會有嚴重的缺失，實現它們不過是把規格轉成程式碼的簡單事情。經驗顯示這實在是非常壞的賭注。打賭規格說明是誤導的，程式到處都是臭蟲 (bug) 會更保險一點。

這其實就是新的編程模式所假設的。設法儘量降低錯誤的成本，而不是希望人們不犯錯。錯誤的成本是修補它所花費的時間。使用強大的語言跟好的開發環境，這種成本會大幅地降低。編程風格可以更多地依靠探索，較少地依靠事前規劃。

規劃是一種必要之惡。它是評估風險的指標：越是危險，預先規劃就顯得更重要。強大的工具降低了風險，也降低了規劃的需求。程式的設計可以從最有用的資訊來源中受益：過去實作程式的經驗。

Lisp 風格從 1960 年代一直朝著這個方向演進。你在 Lisp 中可以如此快速地寫出原型，以致於你已歷經好幾個設計和實現的迴圈，而在舊的模式當中，你可能才剛寫完規格說明。你不必擔心設計的缺失，因爲你將更快地發現它們。你也不用擔心有那麼多臭蟲。當你用函數式風格來編程，你的臭蟲只有區域的影響。當你使用一種很抽象的語言，某些臭蟲(如迷途指標 [http://zh.wikipedia.org/zh-cn/%E8%BF%B7%E9%80%94%E6%8C%87%E9%92%88])不再可能發生，而剩下的臭蟲很容易找出，因爲你的程式更短了。當你有一個互動的開發環境，你可以即時修補臭蟲，不必經歷 編輯，編譯，測試的漫長過程。

Lisp 風格會這麼演進是因爲它產生的結果。聽起來很奇怪，少的規劃意味著更好的設計。技術史上相似的例子不勝列舉。一個相似的變革發生在十五世紀的繪畫圈裡。在油畫流行前，畫家使用一種叫做蛋彩 [http://zh.wikipedia.org/zh-cn/%E8%9B%8B%E5%BD%A9%E7%95%AB]的材料來作畫。蛋彩不能被混和或塗掉。犯錯的代價非常高，也使得畫家變得保守。後來隨著油畫顏料的出現，作畫風格有了大幅地改變。油畫“允許你再來一次”這對困難主題的處理，像是畫人體，提供了決定性的有利條件。

新的材料不僅使畫家更容易作畫了。它使新的更大膽的作畫方式成爲可能。 Janson 寫道：

> 如果沒有油畫顏料，弗拉芒大師們的征服可見的現實的口號就會大打折扣。於是，從技術的角度來說，也是如此，但他們當之無愧地稱得上是“現代繪畫之父”，油畫

顏料從此以後成爲畫家的基本顏料。

做爲一種介質，蛋彩與油畫顏料一樣美麗。但油畫顏料的彈性給想像力更大的發揮空間 —— 這是決定性的因素。

程式設計正經歷著相同的改變。新的介質像是“動態的物件導向語言” —— 即 Lisp 。這不是說我們所有的軟體在幾年內都要用 Lisp 來寫。從蛋彩到油畫的轉變也不是一夜完成的；油彩一開始只在領先的藝術中心流行，而且經常混合著蛋彩來使用。我們現在似乎正處於這個階段。 Lisp 被大學，研究室和某些頂尖的公司所使用。同時，從 Lisp 借鑑的思想越來越多地出現在主流語言中：交互式編程環境 (interactive programming environment)、垃圾回收(garbage collection) [http://zh.wikipedia.org/zh-cn/%E5%9E%83%E5%9C%BE%E5%9B%9E%E6%94%B6_(%E8%A8%88%E7%AE%97%E6
運行期型別 (run-time typing)，僅舉其中幾個。

強大的工具正降低探索的風險。這對程式設計師來說是好消息，因爲意味者我們可以從事更有野心的項目。油畫的確有這個效果。採用油畫後的時期正是繪畫的黃金時期。類似的跡象正在程式設計的領域中發生。

第二章：歡迎來到 Lisp

本章的目的是讓你儘快開始編程。本章結束時，你會掌握足夠多的 Common Lisp 知識來開始寫程式。

2.1 形式 (Form)

人可以通過實踐來學習一件事，這對於 Lisp 來說特別有效，因爲 Lisp 是一門交互式的語言。任何 Lisp 系統都含有一個交互式的前端，叫做頂層(toplevel)。你在頂層輸入 Lisp 表達式，而系統會顯示它們的值。

Lisp 通常會打印一個提示符告訴你，它正在等待你的輸入。許多 Common Lisp 的實現用 `>` 作爲頂層提示符。本書也沿用這個符號。

一個最簡單的 Lisp 表達式是整數。如果我們在提示符後面輸入 `1`，

```
> 1
1
>
```

系統會打印出它的值，接著印出另一個提示符，告訴你它在等待更多的輸入。

在這個情況裡，打印的值與輸入的值相同。數字 `1` 稱之爲對自身求值。當我們輸入需要做某些計算來求值的表達式時，生活變得更加有趣了。舉例來說，如果我們想把兩個數相加，我們輸入像是：

```
> (+ 2 3)
5
```

在表達式 `(+ 2 3)` 裡，`+` 稱爲運算子，而數字 `2` 跟 `3` 稱爲實參。

在日常生活中，我們會把表達式寫作 `2 + 3`，但在 Lisp 裡，我們把 `+` 運算子寫在前面，接著寫實參，再把整個表達式用一對括號包起來：`(+ 2 3)`。這稱爲前序表達式。一開始可能覺得這樣寫表達式有點怪，但事實上這種表示法是 Lisp 最美妙的東西之一。

舉例來說，我們想把三個數加起來，用日常生活的表示法，要寫兩次 `+` 號，

```
2 + 3 + 4
```

而在 Lisp 裡，只需要增加一個實參：

```
(+ 2 3 4)
```

日常生活中用 + 時，它必須有兩個實參，一個在左，一個在右。前序表示法的靈活性代表著，在 Lisp 裡， + 可以接受任意數量的實參，包含了沒有實參：

```
> (+)
0
> (+ 2)
2
> (+ 2 3)
5
> (+ 2 3 4)
9
> (+ 2 3 4 5)
14
```

由於運算子可接受不定數量的實參，我們需要用括號來標明表達式的開始與結束。

表達式可以巢狀。即表達式裡的實參，可以是另一個複雜的表達式：

```
> (/ (- 7 1) (- 4 2))
3
```

上面的表達式用中文來說是， (七減一) 除以 (四減二) 。

Lisp 表示法另一個美麗的地方是：它就是如此簡單。所有的 Lisp 表達式，要麼是 1 這樣的數原子，要麼是包在括號裡，由零個或多個表達式所構成的列表。以下是合法的 Lisp 表達式：

```
2 (+ 2 3) (+ 2 3 4) (/ (- 7 1) (- 4 2))
```

稍後我們將理解到，所有的 Lisp 程式都採用這種形式。而像是 C 這種語言，有著更複雜的語法：算術表達式採用中序表示法；函數呼叫採用某種前序表示法，實參用逗號隔開；表達式用分號隔開；而一段程式用大括號隔開。

在 Lisp 裡，我們用單一的表示法，來表達所有的概念。

2.2 求值 (Evaluation)

上一小節中，我們在頂層輸入表達式，然後 Lisp 顯示它們的值。在這節裡我們深入理解一下表達式是如何被求值的。

在 Lisp 裡， `+` 是函數，然而如 `(+ 2 3)` 的表達式，是函數呼叫。

當 Lisp 對函數呼叫求值時，它做下列兩個步驟：

1. 首先從左至右對實參求值。在這個例子當中，實參對自身求值，所以實參的值分別是 `2` 跟 `3` 。
2. 實參的值傳入以運算子命名的函數。在這個例子當中，將 `2` 跟 `3` 傳給 `+` 函數，返回 `5` 。

如果實參本身是函數呼叫的話，上述規則同樣適用。以下是當 `(/ (- 7 1) (- 4 2))` 表達式被求值時的情形：

1. Lisp 對 `(- 7 1)` 求值: `7` 求值爲 `7` ， `1` 求值爲 `1` ，它們被傳給函數 `-` ，返回 `6` 。
2. Lisp 對 `(- 4 2)` 求值: `4` 求值爲 `4` ， `2` 求值爲 `2` ，它們被傳給函數 `-` ，返回 `2` 。
3. 數值 `6` 與 `2` 被傳入函數 `/` ，返回 `3` 。

但不是所有的 Common Lisp 運算子都是函數，不過大部分是。函數呼叫都是這麼求值。由左至右對實參求值，將它們的數值傳入函數，來返回整個表達式的值。這稱爲 Common Lisp 的求值規則。

Note: 逃離麻煩

如果你試著輸入 Lisp 不能理解的東西，它會打印一個錯誤訊息，接著帶你到一種叫做中斷迴圈（break loop）的頂層。 中斷迴圈給予有經驗的程式設計師一個機會，來找出錯誤的原因，不過最初你只會想知道如何從中斷迴圈中跳出。 如何返回頂層取決於你所使用的 Common Lisp 實現。在這個假定的實現環境中，輸入 `:abort` 跳出：

```
> (/ 1 0)
Error: Division by zero
      Options: :abort, :backtrace
>> :abort
>
```

附錄 A 示範了如何除錯 Lisp 程式，並給出一些常見的錯誤例子。

一個不遵守 Common Lisp 求值規則的運算子是 `quote` 。 `quote` 是一個特殊的運算子，意味著它自己有一套特別的求值規則。這個規則就是：什麼也不做。 `quote` 運算子接受一個實參，並完封不動地返回它。

```
> (quote (+ 3 5))
```

```
(+ 3 5)
```

爲了方便起見，Common Lisp 定義 ' 作爲 `quote` 的縮寫。你可以在任何的表達式前，貼上一個 '，與呼叫 `quote` 是同樣的效果：

```
> '(+ 3 5)
(+ 3 5)
```

使用縮寫 ' 比使用整個 `quote` 表達式更常見。

Lisp 提供 `quote` 作爲一種保護表達式不被求值的方式。下一節將解釋爲什麼這種保護很有用。

2.3 資料 (Data)

Lisp 提供了所有在其他語言找的到的，以及其他語言所找不到的資料型別。一個我們已經使用過的型別是整數（integer），整數用一系列的數字來表示，比如： `256` 。另一個 Common Lisp 與多數語言有關，並很常見的資料型別是字串（string），字串用一系列被雙引號包住的字元串表示，比如： `"ora et labora"` [3] 。整數與字串一樣，都是對自身求值的。

[3] “ora et labora” 是拉丁文，意思是禱告與工作。

有兩個通常在別的語言所找不到的 Lisp 資料型別是符號（symbol）與列表（lists），符號是英語的單詞 (words)。無論你怎麼輸入，通常會被轉換爲大寫：

```
> 'Artichoke
ARTICHOKE
```

符號（通常）不對自身求值，所以要是想引用符號，應該像上例那樣用 ' 引用它。

列表是由被括號包住的零個或多個元素來表示。元素可以是任何型別，包含列表本身。使用列表必須要引用，不然 Lisp 會以爲這是個函數呼叫：

```
> '(my 3 "Sons")
(MY 3 "Sons")
> '(the list (a b c) has 3 elements)
(THE LIST (A B C) HAS 3 ELEMENTS)
```

注意引號保護了整個表達式（包含內部的子表達式）被求值。

你可以呼叫 `list` 來創建列表。由於 `list` 是函數，所以它的實參會被求值。這裡我們看一個在函數 `list` 呼叫裡面，呼叫 `+` 函數的例子：

```
> (list 'my (+ 2 1) "Sons")
(MY 3 "Sons")
```

我們現在來到領悟 Lisp 最卓越特性的地方之一。*Lisp*的程式是用列表來表示的。如果實參的優雅與彈性不能說服你 Lisp 表示法是無價的工具，這裡應該能使你信服。這代表著 Lisp 程式可以寫出 Lisp 程式。 Lisp 程式設計師可以（並且經常）寫出能爲自己寫程式的程式。

不過得到第 10 章，我們才來考慮這種程式，但現在了解到列表和表達式的關係是非常重要的，而不是被它們搞混。這也就是爲什麼我們需要 `quote` 。如果一個列表被引用了，則求值規則對列表自身來求值；如果沒有被引用，則列表被視爲是程式碼，依求值規則對列表求值後，返回它的值。

```
> (list '(+ 2 1) (+ 2 1))
((+ 2 1) 3)
```

這裡第一個實參被引用了，所以產生一個列表。第二個實參沒有被引用，視爲函數呼叫，經求值後得到一個數字。

在 Common Lisp 裡有兩種方法來表示空列表。你可以用一對不包括任何東西的括號來表示，或用符號 `nil` 來表示空表。你用哪種表示法來表示空表都沒關係，但它們都會被顯示爲 `nil` :

```
> ()
NIL
> nil
NIL
```

你不需要引用 `nil` （但引用也無妨），因爲 `nil` 是對自身求值的。

2.4 列表操作 (List Operations)

用函數 `cons` 來構造列表。如果傳入的第二個實參是列表，則返回由兩個實參所構成的新列表，新列表為第一個實參加上第二個實參：

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

可以通過把新元素建立在空表之上，來構造一個新列表。上一節所看到的函數 `list` ，不過就是一個把幾個元素加到 `nil` 上的快捷方式：

```
> (cons 'a (cons 'b nil))
(A B)
```

```
> (list 'a 'b)
(A B)
```

取出列表元素的基本函數是 car 和 cdr 。對列表取 car 返回第一個元素，而對列表取 cdr 返回第一個元素之後的所有元素：

```
> (car '(a b c))
A
> (cdr '(a b c))
(B C)
```

你可以把 car 與 cdr 混合使用來取得列表中的任何元素。如果我們想要取得第三個元素，我們可以：

```
> (car (cdr (cdr '(a b c d))))
C
```

不過，你可以用更簡單的 third 來做到同樣的事情：

```
> (third '(a b c d))
C
```

2.5 真與假 (Truth)

在 Common Lisp 裡，符號 t 是表示邏輯 真 的預設值。與 nil 相同， t 也是對自身求值的。如果實參是一個列表，則函數 listp 返回 真 ：

```
> (listp '(a b c))
T
```

函數的返回值將會被解釋成邏輯 真 或邏輯 假 時，則稱此函數爲謂詞（*predicate*）。在 Common Lisp 裡，謂詞的名字通常以 p 結尾。

邏輯 假 在 Common Lisp 裡，用 nil ，即空表來表示。如果我們傳給 listp 的實參不是列表，則返回 nil 。

```
> (listp 27)
NIL
```

由於 nil 在 Common Lisp 裡扮演兩個角色，如果實參是一個空表，則函數 null 返回 真 。

```
> (null nil)
```

```
T
```

而如果實參是邏輯 假，則函數 not 返回 真：

```
> (not nil)
T
```

null 與 not 做的是一樣的事情。

在 Common Lisp 裡，最簡單的條件式是 if 。通常接受三個實參：一個 *test* 表達式，一個 *then* 表達式和一個 *else* 表達式。若 test 表達式求值爲邏輯 真，則對 then 表達式求值，並返回這個值。若 test 表達式求值爲邏輯 假，則對 else 表達式求值，並返回這個值：

```
> (if (listp '(a b c))
      (+ 1 2)
      (+ 5 6))
3
> (if (listp 27)
      (+ 1 2)
      (+ 5 6))
11
```

與 quote 相同， if 是特殊的運算子。不能用函數來實現，因爲實參在函數呼叫時永遠會被求值，而 if 的特點是，只有最後兩個實參的其中一個會被求值。 if 的最後一個實參是選擇性的。如果忽略它的話，預設值是 nil：

```
> (if (listp 27)
    (+ 1 2))
NIL
```

雖然 t 是邏輯 真 的預設表示法，任何非 nil 的東西，在邏輯的上下文裡通通被視爲 真。

```
> (if 27 1 2)
1
```

邏輯運算子 and 和 or 與條件式類似。兩者都接受任意數量的實參，但僅對能影響返回值的幾個實參求值。如果所有的實參都爲 真 （即非 nil ），那麼 and 會返回最後一個實參的值：

```
> (and t (+ 1 2))
3
```

如果其中一個實參爲 假 ，那之後的所有實參都不會被求值。 or 也是如此，只要碰到一個爲 真 的實參，就停止對之後所有的實參求值。

以上這兩個運算子稱爲宏。宏和特殊的運算子一樣，可以繞過一般的求值規則。第十章解釋了如何編寫你自己的宏。

2.6 函數 (Functions)

你可以用 defun 來定義新函數。通常接受三個以上的實參：一個名字，一組用列表表示的實參，以及一個或多個組成函數體的表達式。我們可能會這樣定義 third：

```
> (defun our-third (x)
    (car (cdr (cdr x))))
OUR-THIRD
```

第一個實參說明此函數的名稱將是 our-third 。第二個實參，一個列表 (x) ，說明這個函數會接受一個形參： x 。這樣使用的佔位符符號叫做變數。當變數代表了傳入函數的實參時，如這裡的 x ，又被叫做形參。

定義的剩餘部分， (car (cdr (cdr x))) ，即所謂的函數主體。它告訴 Lisp 該怎麼計算此函數的返回值。所以呼叫一個 our-third 函數，對於我們作爲實參傳入的任何 x ，會返回 (car (cdr (cdr x)))：

```
> (our-third '(a b c d))
C
```

既然我們已經討論過了變數，理解符號是什麼就更簡單了。符號是變數的名字，符號本身就是以物件的方式存在。這也是爲什麼符號，必須像列表一樣被引用。列表必須被引用，不然會被視爲程式碼。符號必須要被引用，不然會被當作變數。

你可以把函數定義想成廣義版的 Lisp 表達式。下面的表達式測試 1 和 4 的和是否大於 3：

```
> (> (+ 1 4) 3)
T
```

通過將這些數字替換爲變數，我們可以寫個函數，測試任兩數之和是否大於第三個數：

```
> (defun sum-greater (x y z)
    (> (+ x y) z))
SUM-GREATER
> (sum-greater 1 4 3)
T
```

Lisp 不對程式、過程以及函數作區別。函數做了所有的事情（事實上，函數是語言的主要部分）。如果你想要把你的函數之一作爲主函數（*main* function），可以這麼做，但平常你就能在頂層中呼叫任何函數。這表示當你編程時，你可以把程式拆分成一小塊一小塊地來做除錯。

2.7 遞迴 (Recursion)

上一節我們所定義的函數，呼叫了別的函數來幫它們做事。比如 `sum-greater` 呼叫了 `+` 和 `>` 。函數可以呼叫任何函數，包括自己。自己呼叫自己的函數是遞迴的。 Common Lisp 函數 `member` ，測試某個東西是否爲列表的成員。下面是定義成遞迴函數的簡化版：

```
> (defun our-member (obj lst)
    (if (null lst)
        nil
    (if (eql (car lst) obj)
        lst
        (our-member obj (cdr lst)))))
OUR-MEMBER
```

謂詞 `eql` 測試它的兩個實參是否相等；此外，這個定義的所有東西我們之前都學過了。下面是運行的情形：

```
> (our-member 'b '(a b c))
(B C)
> (our-member 'z '(a b c))
NIL
```

下面是 `our-member` 的定義對應到英語的描述。爲了知道一個物件 `obj` 是否爲列表 `lst` 的成員，我們

1. 首先檢查 `lst` 列表是否爲空列表。如果是空列表，那 `obj` 一定不是它的成員，結束。
2. 否則，若 `obj` 是列表的第一個元素時，則它是列表的成員。
3. 不然只有當 `obj` 是列表其餘部分的元素時，它才是列表的成員。

當你想要了解遞迴函數是怎麼工作時，把它翻成這樣的敘述有助於你理解。

起初，許多人覺得遞迴函數很難理解。大部分的理解難處，來自於對函數使用了錯誤的比喻。人們傾向於把函數理解爲某種機器。原物料像實參一樣抵達；某些工作委派給其它函數；最後組裝起來的成品，被作爲返回值運送出去。如果我們用這種比喻來理解函數，那遞迴就自相矛盾了。機器怎可以把工作委派給自己？它已經在忙碌中了。

較好的比喻是，把函數想成一個處理的過程。在過程裡，遞迴是在自然不過的事情了。日常生活中我們經常看到遞迴的過程。舉例來說，假設一個歷史學家，對歐洲歷史上的人口變化感興趣。研究文獻的過程很可能是：

1. 取得一個文獻的複本
2. 尋找關於人口變化的資訊
3. 如果這份文獻提到其它可能有用的文獻，研究它們。

過程是很容易理解的，而且它是遞迴的，因爲第三個步驟可能帶出一個或多個同樣的過程。

所以，別把 `our-member` 想成是一種測試某個東西是否爲列表成員的機器。而是把它想成是，決定某個東西是否爲列表成員的規則。如果我們從這個角度來考慮函數，那麼遞迴的矛盾就不復存在了。

2.8 閱讀 Lisp (Reading Lisp)

上一節我們所定義的 `our-member` 以五個括號結尾。更複雜的函數定義更可能以七、八個括號結尾。剛學 Lisp 的人看到這麼多括號會感到氣餒。這叫人怎麼讀這樣的程式，更不用說編了？怎麼知道哪個括號該跟哪個匹配？

答案是，你不需要這麼做。 Lisp 程式設計師用縮排來閱讀及編寫程式，而不是括號。當他們在寫程式時，他們讓文字編輯器顯示哪個括號該與哪個匹配。任何好的文字編輯器，特別是 Lisp 系統自帶的，都應該能做到括號匹配（paren-matching）。在這種編輯器中，當你輸入一個括號時，編輯器指出與其匹配的那一個。如果你的編輯器不能匹配括號，別用了，想想如何讓它做到，因爲沒有這個功能，你根本不可能編 Lisp 程式 [1] 。

有了好的編輯器之後，括號匹配不再會是問題。而且由於 Lisp 縮排有通用的慣例，閱讀程式也不是個問題。因爲所有人都使用一樣的習慣，你可以忽略那些括號，通過縮排來閱讀程式。

任何有經驗的 Lisp 黑客，會發現如果是這樣的 `our-member` 的定義很難閱讀：

```
(defun our-member (obj lst) (if (null lst) nil (if
(eql (car lst) obj) lst (our-member obj (cdr lst)))))
```

但如果程式適當地縮排時，他就沒有問題了。可以忽略大部分的括號而仍能讀懂它：

```
defun our-member (obj lst)
 if null lst
    nil
```

```
    if eql (car lst) obj
       lst
       our-member obj (cdr lst)
```

事實上，這是你在紙上寫 Lisp 程式的實用方法。等輸入程式至計算機的時候，可以利用編輯器匹配括號的功能。

2.9 輸入輸出 (Input and Output)

到目前爲止，我們已經利用頂層偷偷使用了 I/O 。對實際的交互程式來說，這似乎還是不太夠。在這一節，我們來看幾個輸入輸出的函數。

最普遍的 Common Lisp 輸出函數是 `format` 。接受兩個或兩個以上的實參，第一個實參決定輸出要打印到哪裡，第二個實參是字串模版，而剩餘的實參，通常是要插入到字串模版，用打印表示法（printed representation）所表示的物件。下面是一個典型的例子：

```
> (format t "~A plus ~A equals ~A. ~%" 2 3 (+ 2 3))
2 plus 3 equals 5.
NIL
```

注意到有兩個東西被打印出來。第一行是 `format` 印出來的。第二行是呼叫 `format` 函數的返回值，就像平常頂層會打印出來的一樣。通常像 `format` 這種函數不會直接在頂層呼叫，而是在程式內部裡使用，所以返回值不會被看到。

`format` 的第一個實參 `t` ，表示輸出被送到預設的地方去。通常是頂層。第二個實參是一個用作輸出模版的字串。在這字串裡，每一個 `~A` 表示了被填入的位置，而 `~%` 表示一個換行。這些被填入的位置依序由後面的實參填入。

標準的輸入函數是 `read` 。當沒有實參時，會讀取預設的位置，通常是頂層。下面這個函數，提示使用者輸入，並返回任何輸入的東西：

```
(defun askem (string)
 (format t "~A" string)
 (read))
```

它的行爲如下：

```
> (askem "How old are you?")
How old are you?29

29
```

記住 `read` 會一直永遠等在這裡，直到你輸入了某些東西，並且（通常要）按下回車。

因此，不印出明確的提示資訊是很不明智的，程式會給人已經死機的印象，但其實它是在等待輸入。

第二件關於 `read` 所需要知道的事是，它很強大： `read` 是一個完整的 Lisp 解析器（parser）。不僅是可以讀入字元，然後當作字串返回它們。它解析它所讀入的東西，並返回產生出來的 Lisp 物件。在上述的例子，它返回一個數字。

`askem` 的定義雖然很短，但體現出一些我們在之前的函數沒看過的東西。函數主體可以有不只一個表達式。函數主體可以有任意數量的表達式。當函數被呼叫時，會依序求值，函數會返回最後一個的值。

在之前的每一節中，我們堅持所謂"純粹的" Lisp —— 即沒有副作用的 Lisp 。副作用是指，表達式被求值後，對外部世界的狀態做了某些改變。當我們對一個如 `(+ 1 2)` 這樣純粹的 Lisp 表達式求值時，沒有產生副作用。它只返回一個值。但當我們呼叫 `format` 時，它不僅返回值，還印出了某些東西。這就是一種副作用。

當我們想要寫沒有副作用的程式時，則定義多個表達式的函數主體就沒有意義了。最後一個表達式的值，會被當成函數的返回值，而之前表達式的值都被捨棄了。如果這些表達式沒有副作用，你沒有任何理由告訴 Lisp，爲什麼要去對它們求值。

2.10 變數 (Variables)

`let` 是一個最常用的 Common Lisp 的運算子之一，它讓你引入新的區域變數（local variable）：

```
> (let ((x 1) (y 2))
      (+ x y))
3
```

一個 `let` 表達式有兩個部分。第一個部分是一組創建新變數的指令，指令的形式爲 *(variable expression)* 。每一個變數會被賦予相對應表達式的值。上述的例子中，我們創造了兩個變數， `x` 和 `y` ，分別被賦予初始值 `1` 和 `2` 。這些變數只在 `let` 的函數體內有效。

一組變數與數值之後，是一個有表達式的函數體，表達式依序被求值。但這個例子裡，只有一個表達式，呼叫 `+` 函數。最後一個表達式的求值結果作爲 `let` 的返回值。以下是一個用 `let` 所寫的，更有選擇性的 `askem` 函數：

```
(defun ask-number ()
 (format t "Please enter a number. ")
 (let ((val (read)))
   (if (numberp val)
```

```
        val
        (ask-number))))
```

這個函數創建了變數 `val` 來儲存 `read` 所返回的物件。因爲它知道該如何處理這個物件，函數可以先觀察你的輸入，再決定是否返回它。你可能猜到了， `numberp` 是一個謂詞，測試它的實參是否爲數字。

如果使用者不是輸入一個數字， `ask-number` 會持續呼叫自己。最後得到一個只接受數字的函數：

```
> (ask-number)
Please enter a number. a
Please enter a number. (ho hum)
Please enter a number. 52
52
```

我們已經看過的這些變數都叫做區域變數。它們只在特定的上下文裡有效。另外還有一種變數叫做全局變數（global variable），是在任何地方都是可視的。 [2]

你可以給 `defparameter` 傳入符號和值，來創建一個全局變數：

```
> (defparameter *glob* 99)
*GLOB*
```

全局變數在任何地方都可以存取，除了在定義了相同名字的區域變數的表達式裡。爲了避免這種情形發生，通常我們在給全局變數命名時，以星號作開始與結束。剛才我們創造的變數可以唸作 “星-glob-星” (star-glob-star)。

你也可以用 `defconstant` 來定義一個全局的常數：

```
(defconstant limit (+ *glob* 1))
```

我們不需要給常數一個獨一無二的名字，因爲如果有相同名字存在，就會有錯誤產生(error)。如果你想要檢查某些符號，是否爲一個全局變數或常數，使用 `boundp` 函數：

```
> (boundp '*glob*)
T
```

2.11 賦值 (Assignment)

在 Common Lisp 裡，最普遍的賦值運算子（assignment operator）是 `setf` 。可以用來給全局或區域變數賦值：

```
> (setf *glob* 98)
98
> (let ((n 10))
    (setf n 2)
    n)
2
```

如果 setf 的第一個實參是符號（symbol），且符號不是某個區域變數的名字，則 setf 把這個符號設爲全局變數：

```
> (setf x (list 'a 'b 'c))
(A B C)
```

也就是說，通過賦值，你可以隱式地創建全局變數。 不過，一般來說，還是使用 defparameter 明確地創建全局變數比較好。

你不僅可以給變數賦值。傳入 setf 的第一個實參，還可以是表達式或變數名。在這種情況下，第二個實參的值被插入至第一個實參所引用的位置：

```
> (setf (car x) 'n)
N
> x
(N B C)
```

setf 的第一個實參幾乎可以是任何引用到特定位置的表達式。所有這樣的運算子在附錄 D 中被標註爲 “可設置的”（“settable”）。你可以給 setf 傳入（偶數）個實參。一個這樣的表達式

```
(setf a 'b
      c 'd
      e 'f)
```

等同於依序呼叫三個單獨的 setf 函數：

```
(setf a 'b)
(setf c 'd)
(setf e 'f)
```

2.12 函數式編程 (Functional Programming)

函數式編程意味著撰寫利用返回值而工作的程式，而不是修改東西。它是 Lisp 的主流範式。大部分 Lisp 的內建函數被呼叫是爲了取得返回值，而不是副作用。

舉例來說，函數 remove 接受一個物件和一個列表，返回不含這個物件的新列表：

```
> (setf lst '(c a r a t))
(C A R A T)
> (remove 'a lst)
(C R T)
```

爲什麼不乾脆說 `remove` 從列表裡移除一個物件？因爲它不是這麼做的。原來的表沒有被改變：

```
> lst
(C A R A T)
```

若你真的想從列表裡移除某些東西怎麼辦？在 Lisp 通常你這麼做，把這個列表當作實參，傳入某個函數，並使用 `setf` 來處理返回值。要移除所有在列表 `x` 的 `a`，我們可以說：

```
(setf x (remove 'a x))
```

函數式編程本質上意味著避免使用如 `setf` 的函數。起初可能覺得這根本不可能，更遑論去做了。怎麼可以只憑返回值來建立程式？

完全不用到副作用是很不方便的。然而，隨著你進一步閱讀，會驚訝地發現需要用到副作用的地方很少。副作用用得越少，你就更上一層樓。

函數式編程最重要的優點之一是，它允許交互式測試（interactive testing）。在純函數式的程式裡，你可以測試每個你寫的函數。如果它返回你預期的值，你可以有信心它是對的。這額外的信心，集結起來，會產生巨大的差別。當你改動了程式裡的任何一個地方，會得到即時的改變。而這種即時的改變，使我們有一種新的編程風格。類比於電話與信件，讓我們有一種新的通訊方式。

2.13 迭代 (Iteration)

當我們想重複做一些事情時，迭代比遞迴來得更自然。典型的例子是用迭代來產生某種表格。這個函數

```
(defun show-squares (start end)
  (do ((i start (+ i 1)))
      ((> i end) 'done)
    (format t "~A ~A~%" i (* i i))))
```

列印從 `start` 到 `end` 之間的整數的平方：

```
> (show-squares 2 5)
2 4
```

```
3 9
4 16
5 25
DONE
```

do 宏是 Common Lisp 裡最基本的迭代運算子。和 let 類似， do 可以創建變數，而第一個實參是一組變數的規格說明列表。每個元素可以是以下的形式

```
(variable initial update)
```

其中 *variable* 是一個符號， *initial* 和 *update* 是表達式。最初每個變數會被賦予 *initial* 表達式的值；每一次迭代時，會被賦予 *update* 表達式的值。在 show-squares 函數裡， do 只創建了一個變數 i 。第一次迭代時， i 被賦與 start 的值，在接下來的迭代裡， i 的值每次增加 1 。

第二個傳給 do 的實參可包含一個或多個表達式。第一個表達式用來測試迭代是否結束。在上面的例子中，測試表達式是 (> i end) 。接下來在列表中的表達式會依序被求值，直到迭代結束。而最後一個值會被當作 do 的返回值來返回。所以 show-squares 總是返回 done 。

do 的剩餘參陣列成了迴圈的函數體。在每次迭代時，函數體會依序被求值。在每次迭代過程裡，變數被更新，檢查終止測試條件，接著（若測試失敗）求值函數體。

作爲對比，以下是遞迴版本的 show-squares ：

```
(defun show-squares (i end)
   (if (> i end)
     'done
     (progn
       (format t "~A ~A~%" i (* i i))
       (show-squares (+ i 1) end))))
```

唯一的新東西是 progn 。 progn 接受任意數量的表達式，依序求值，並返回最後一個表達式的值。

爲了處理某些特殊情況， Common Lisp 有更簡單的迭代運算子。舉例來說，要遍歷列表的元素，你可能會使用 dolist 。以下函數返回列表的長度：

```
(defun our-length (lst)
  (let ((len 0))
    (dolist (obj lst)
      (setf len (+ len 1)))
    len))
```

這裡 dolist 接受這樣形式的實參*(variable expression)*，跟著一個具有表達式的函數主體。函數主體會被求值，而變數相繼與表達式所返回的列表元素綁定。因此上面的迴圈說，對於列表 lst 裡的每一個 obj ，遞增 len 。很顯然這個函數的遞迴版本是：

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

也就是說，如果列表是空表，則長度爲 0 ；否則長度就是對列表取 cdr 的長度加一。遞迴版本的 our-length 比較易懂，但由於它不是尾遞迴（tail-recursive）的形式 (見 13.2 節)，效率不是那麼高。

2.14 函數作爲物件 (Functions as Objects)

函數在 Lisp 裡，和符號、字串或列表一樣，是稀鬆平常的物件。如果我們把函數的名字傳給 function ，它會返回相關聯的物件。和 quote 類似， function 是一個特殊運算子，所以我們無需引用（quote）它的實參：

```
> (function +)
#<Compiled-Function + 17BA4E>
```

這看起來很奇怪的返回值，是在典型的 Common Lisp 實現裡，函數可能的打印表示法。

到目前爲止，我們僅討論過，不管是 Lisp 打印它們，還是我們輸入它們，看起來都是一樣的物件。但這個慣例對函數不適用。一個像是 + 的內建函數 ，在內部可能是一段機器語言程式碼（machine language code）。每個 Common Lisp 實現，可以選擇任何它喜歡的外部表示法（external representation）。

如同我們可以用 ' 作爲 quote 的縮寫，也可以用 #' 作爲 function 的縮寫：

```
> #'+
#<Compiled-Function + 17BA4E>
```

這個縮寫稱之爲升引號（sharp-quote）。

和別種物件類似，可以把函數當作實參傳入。有個接受函數作爲實參的函數是 apply 。apply 接受一個函數和實參列表，並返回把傳入函數應用在實參列表的結果：

```
> (apply #'+ '(1 2 3))
6
> (+ 1 2 3)
6
```

apply 可以接受任意數量的實參，只要最後一個實參是列表即可：

```
> (apply #'+ 1 2 '(3 4 5))
15
```

函數 funcall 做的是一樣的事情，但不需要把實參包裝成列表。

```
> (funcall #'+ 1 2 3)
6
```

Note: 什麼是 lambda ？

lambda 表達式裡的 lambda 不是一個運算子。而只是個符號。 在早期的 Lisp 方言裡，lambda 存在的原因是：由於函數在內部是用列表來表示， 因此辨別列表與函數的方法，就是檢查第一個元素是否爲 lambda 。

在 Common Lisp 裡，你可以用列表來表達函數， 函數在內部會被表示成獨特的函數物件。因此不再需要 *lambda* 了。 如果需要把函數記爲

```
((x) (+ x 100))
```

而不是

```
(lambda (x) (+ x 100))
```

也是可以的。

但 Lisp 程式設計師習慣用符號 lambda ， 來撰寫函數， 因此 Common Lisp 爲了傳統，而保留了 lambda 。

defun 宏，創建一個函數並給函數命名。但函數不需要有名字，而且我們不需要 defun 來定義他們。和大多數的 Lisp 物件一樣，我們可以直接引用函數。

要直接引用整數，我們使用一系列的數字；要直接引用一個函數，我們使用所謂的 *lambda* 表達式。一個 lambda 表達式是一個列表，列表包含符號 lambda ， 接著是形參列表，以及由零個或多個表達式所組成的函數體。

下面的 lambda 表達式，表示一個接受兩個數字並返回兩者之和的函數：

```
(lambda (x y)
 (+ x y))
```

列表 `(x y)` 是形參列表，跟在它後面的是函數主體。

一個 `lambda` 表達式可以作爲函數名。和普通的函數名稱一樣， lambda 表達式也可以是函數呼叫的第一個元素，

```
> ((lambda (x) (+ x 100)) 1)
101
```

而通過在 `lambda` 表達式前面貼上 `#'` ，我們得到對應的函數，

```
> (funcall #'(lambda (x) (+ x 100))
           1)
```

`lambda` 表示法除上述用途以外，還允許我們使用匿名函數。

2.15 型別 (Types)

Lisp 處理型別的方法非常靈活。在很多語言裡，變數是有型別的，得宣告變數的型別才能使用它。在 Common Lisp 裡，數值才有型別，而變數沒有。你可以想像每個物件，都貼有一個標明其型別的標籤。這種方法叫做顯式型別（*manifest typing*）。你不需要宣告變數的型別，因爲變數可以存放任何型別的物件。

雖然從來不需要宣告型別，但出於效率的考量，你可能會想要宣告變數的型別。型別宣告在第 13.3 節時討論。

Common Lisp 的內建型別，組成了一個類別的層級。物件總是不止屬於一個型別。舉例來說，數字 27 的型別，依普遍性的增加排序，依序是 `fixnum` 、 `integer` 、 `rational` 、 `real` 、 `number` 、 `atom` 和 `t` 型別。（數值型別將在第 9 章討論。）型別 `t` 是所有型別的基類（supertype）。所以每個物件都屬於 `t` 型別。

函數 `typep` 接受一個物件和一個型別，然後判定物件是否爲該型別，是的話就返回真：

```
> (typep 27 'integer)
T
```

我們會在遇到各式內建型別時來討論它們。

2.16 展望 (Looking Forward)

本章僅談到 Lisp 的表面。然而，一種非比尋常的語言形象開始出現了。首先，這個語言用單一的語法，來表達所有的程式結構。語法基於列表，列表是一種 Lisp 物件。函

數本身也是 Lisp 物件，函數能用列表來表示。而 Lisp 本身就是 Lisp 程式。幾乎所有你定義的函數，與內建的 Lisp 函數沒有任何區別。

如果你對這些概念還不太了解，不用擔心。 Lisp 介紹了這麼多新穎的概念，在你能駕馭它們之前，得花時間去熟悉它們。不過至少要了解一件事：在這些概念當中，有著優雅到令人吃驚的概念。

Richard Gabriel [http://en.wikipedia.org/wiki/Richard_P._Gabriel] 曾經半開玩笑的說， C 是拿來寫 Unix 的語言。我們也可以說， Lisp 是拿來寫 Lisp 的語言。但這是兩種不同的論述。一個可以用自己編寫的語言和一種適合編寫某些特定型別應用的語言，是有著本質上的不同。這開創了新的編程方法：你不但在語言之中編程，還把語言改善成適合程式的語言。如果你想了解 Lisp 編程的本質，理解這個概念是個好的開始。

Chapter 2 總結 (Summary)

1. Lisp 是一種交互式語言。如果你在頂層輸入一個表達式， Lisp 會顯示它的值。
2. Lisp 程式由表達式組成。表達式可以是原子，或一個由運算子跟著零個或多個實參的列表。前序表示法代表運算子可以有任意數量的實參。
3. Common Lisp 函數呼叫的求值規則： 依序對實參從左至右求值，接著把它們的值傳入由運算子表示的函數。 `quote` 運算子有自己的求值規則，它完封不動地返回實參。
4. 除了一般的資料型別， Lisp 還有符號跟列表。由於 Lisp 程式是用列表來表示的，很輕鬆就能寫出能編程的程式。
5. 三個基本的列表函數是 `cons` ，它創建一個列表； `car` ，它返回列表的第一個元素；以及 `cdr` ，它返回第一個元素之後的所有東西。
6. 在 Common Lisp 裡， `t` 表示邏輯 `真` ，而 `nil` 表示邏輯 `假` 。在邏輯的上下文裡，任何非 `nil` 的東西都視爲 `真` 。基本的條件式是 `if` 。 `and` 與 `or` 是相似的條件式。
7. Lisp 主要由函數所組成。可以用 `defun` 來定義新的函數。
8. 自己呼叫自己的函數是遞迴的。一個遞迴函數應該要被想成是過程，而不是機器。
9. 括號不是問題，因爲程式設計師通過縮排來閱讀與編寫 Lisp 程式。
10. 基本的 I/O 函數是 `read` ，它包含了一個完整的 Lisp 語法分析器，以及 `format` ，它通過字串模板來產生輸出。
11. 你可以用 `let` 來創造新的區域變數，用 `defparameter` 來創造全局變數。
12. 賦值運算子是 `setf` 。它的第一個實參可以是一個表達式。
13. 函數式編程代表避免產生副作用，也是 Lisp 的主導思維。
14. 基本的迭代運算子是 `do` 。
15. 函數是 Lisp 的物件。可以被當成實參傳入，並且可以用 lambda 表達式來表示。
16. 在 Lisp 裡，是數值才有型別，而不是變數。

Chapter 2 習題 (Exercises)

1. 描述下列表達式求值之後的結果：

```
(a) (+ (- 5 1) (+ 3 7))

(b) (list 1 (+ 2 3))

(c) (if (listp 1) (+ 1 2) (+ 3 4))

(d) (list (and (listp 3) t) (+ 1 2))
```

2. 給出 3 種不同表示 `(a b c)` 的 `cons` 表達式。
3. 使用 `car` 與 `cdr` 來定義一個函數，返回一個列表的第四個元素。
4. 定義一個函數，接受兩個實參，返回兩者當中較大的那個。
5. 這些函數做了什麼？

```
(a) (defun enigma (x)
      (and (not (null x))
           (or (null (car x))
               (enigma (cdr x)))))

(b) (defun mystery (x y)
      (if (null y)
          nil
          (if (eql (car y) x)
              0
              (let ((z (mystery x (cdr y))))
                (and z (+ z 1))))))
```

6. 下列表達式， `x` 該是什麼，才會得到相同的結果？

```
(a) > (car (x (cdr '(a (b c) d))))
    B
(b) > (x 13 (/ 1 0))
    13
(c) > (x #'list 1 nil)
    (1)
```

7. 只使用本章所介紹的運算子，定義一個函數，它接受一個列表作爲實參，如果有一個元素是列表時，就返回真。
8. 給出函數的迭代與遞迴版本：

a. 接受一個正整數，並打印出數字數量的點。
b. 接受一個列表，並返回 `a` 在列表裡所出現的次數。

9. 一位朋友想寫一個函數，返回列表裡所有非 `nil` 元素的和。他寫了此函數的兩個版本，但兩個都不能工作。請解釋每一個的錯誤在哪裡，並給出正確的版本。

```
(a) (defun summit (lst)
      (remove nil lst)
      (apply #'+ lst))

(b) (defun summit (lst)
      (let ((x (car lst)))
        (if (null x)
            (summit (cdr lst))
            (+ x (summit (cdr lst))))))
```

腳註

[1] 在 vi，你可以用 :set sm 來啓用括號匹配。在 Emacs，M-x lisp-mode 是一個啓用的好方法。

[2] 真正的區別是詞法變數（lexical）與特殊變數（special variable），但到第六章才會討論這個主題。

第三章：列表

列表是 Lisp 的基本資料結構之一。在最早的 Lisp 方言裡，列表是唯一的資料結構：“Lisp” 這個名字起初是 “LISt Processor” 的縮寫。但 Lisp 已經超越這個縮寫很久了。Common Lisp 是一個有著各式各樣資料結構的通用性程式語言。

Lisp 程式開發通常呼應著開發 Lisp 語言自身。在最初版本的 Lisp 程式，你可能使用很多列表。然而之後的版本，你可能換到快速、特定的資料結構。本章描述了你可以用列表所做的很多事情，以及使用它們來示範一些普遍的 Lisp 概念。

3.1 構造 (Conses)

在 2.4 節我們介紹了 `cons` , `car` , 以及 `cdr` ，基本的 List 操作函數。 `cons` 真正所做的事情是，把兩個物件結合成一個有兩部分的物件，稱之爲 *Cons* 物件。概念上來說，一個 *Cons* 是一對指標；第一個是 `car` ，第二個是 `cdr` 。

Cons 物件提供了一個方便的表示法，來表示任何型別的物件。一個 *Cons* 物件裡的一對指標，可以指向任何型別的物件，包括 *Cons* 物件本身。它利用到我們之後可以用 `cons` 來構造列表的可能性。

我們往往不會把列表想成是成對的，但它們可以這樣被定義。任何非空的列表，都可以被視爲一對由列表第一個元素及列表其餘元素所組成的列表。 Lisp 列表體現了這個概念。我們使用 *Cons* 的一半來指向列表的第一個元素，然後用另一半指向列表其餘的元素(可能是別的 *Cons* 或 `nil`)。 Lisp 的慣例是使用 `car` 代表列表的第一個元素，而用 `cdr` 代表列表的其餘的元素。所以現在 `car` 是列表的第一個元素的同義詞，而 `cdr` 是列表的其餘的元素的同義詞。列表不是不同的物件，而是像 *Cons* 這樣的方式連結起來。

當我們想在 `nil` 上面建立東西時，

```
> (setf x (cons 'a nil))
(A)
```

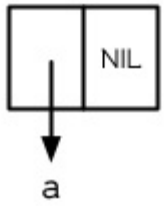

圖 3.1 一個元素的列表

產生的列表由一個 *Cons* 所組成，見圖 3.1。這種表達 *Cons* 的方式叫做箱子表示法 (box notation)，因爲每一個 Cons 是用一個箱子表示，內含一個 `car` 和 `cdr` 的指標。當我們呼叫 `car` 與 `cdr` 時，我們得到指標指向的地方：

```
> (car x)
A
> (cdr x)
NIL
```

當我們構造一個多元素的列表時，我們得到一串 *Cons* (a chain of conses):

```
> (setf y (list 'a 'b 'c))
(A B C)
```

產生的結構見圖 3.2。現在當我們想得到列表的 `cdr` 時，它是一個兩個元素的列表。

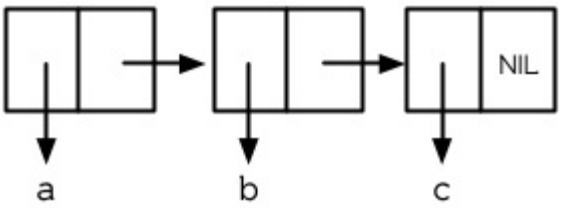

圖 3.2 三個元素的列表

```
> (cdr y)
(B C)
```

在一個有多個元素的列表中， `car` 指標讓你取得元素，而 `cdr` 讓你取得列表內其餘的東西。

一個列表可以有任何型別的物件作爲元素，包括另一個列表：

```
> (setf z (list 'a (list 'b 'c) 'd))
(A (B C) D)
```

當這種情況發生時，它的結構如圖 3.3 所示；第二個 *Cons* 的 `car` 指標也指向一個列表：

```
> (car (cdr z))
(B C)
```

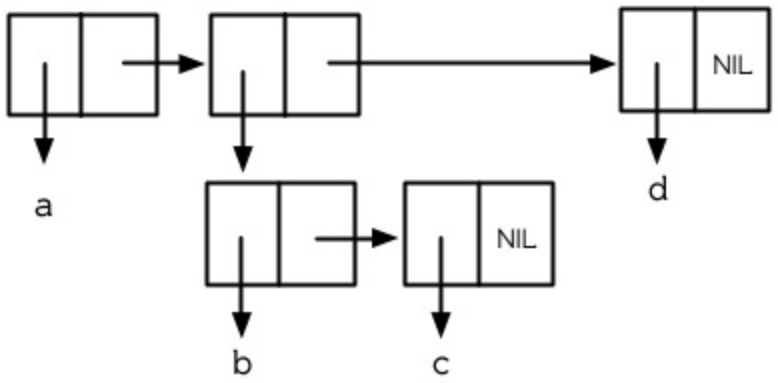

圖 3.3 巢狀列表

前兩個我們構造的列表都有三個元素；只不過 `z` 列表的第二個元素也剛好是一個列表。像這樣的列表稱爲巢狀列表，而像 `y` 這樣的列表稱之爲平坦列表 (*flat*list)。

如果參數是一個 *Cons* 物件，函數 `consp` 返回真。所以我們可以這樣定義 `listp`：

```
(defun our-listp (x)
  (or (null x) (consp x)))
```

因爲所有不是 *Cons* 物件的東西，就是一個原子 (atom)，判斷式 `atom` 可以這樣定義：

```
(defun our-atom (x) (not (consp x)))
```

注意， `nil` 既是一個原子，也是一個列表。

3.2 等式 (Equality)

每一次你呼叫 `cons` 時， Lisp 會配置一塊新的記憶體給兩個指標。所以如果我們用同樣的參數呼叫 `cons` 兩次，我們得到兩個數值看起來一樣，但實際上是兩個不同的物件：

```
> (eql (cons 'a nil) (cons 'a nil))
NIL
```

如果我們也可以詢問兩個列表是否有相同元素，那就很方便了。 Common Lisp 提供了這種目的另一個判斷式： `equal` 。而另一方面 `eql` 只有在它的參數是相同物件時才返回真，

```
> (setf x (cons 'a nil))
(A)
> (eql x x)
T
```

本質上 `equal` 若它的參數打印出的值相同時，返回真：

```
> (equal x (cons 'a nil))
T
```

這個判斷式對非列表結構的別種物件也有效，但一種僅對列表有效的版本可以這樣定義：

```
> (defun our-equal (x y)
    (or (eql x y)
        (and (consp x)
             (consp y)
             (our-equal (car x) (car y))
             (our-equal (cdr x) (cdr y)))))
```

這個定義意味著，如果某個 `x` 和 `y` 相等(`eql`)，那麼他們也相等(`equal`)。

勘誤：這個版本的 `our-equal` 可以用在符號的列表 (list of symbols)，而不是列表 (list)。

3.3 爲什麼 Lisp 沒有指標 (Why Lisp Has No Pointers)

一個理解 Lisp 的祕密之一是意識到變數是有值的，就像列表有元素一樣。如同 *Cons* 物件有指標指向他們的元素，變數有指標指向他們的值。

你可能在別的語言中使用過顯式指標 (explicitly pointer)。在 Lisp，你永遠不用這麼做，因爲語言幫你處理好指標了。我們已經在列表看過這是怎麼實現的。同樣的事情發生在變數身上。舉例來說，假設我們想要把兩個變數設成同樣的列表：

```
 > (setf x '(a b c))
(A B C)
> (setf y x)
(A B C)
```

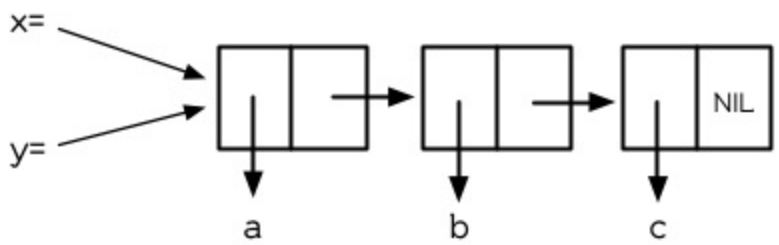

圖 3.4 兩個變數設爲相同的列表

當我們把 `x` 的值賦給 `y` 時，究竟發生什麼事呢？記憶體中與 `x` 有關的位置並沒有包含這

個列表，而是一個指標指向它。當我們給 y 賦一個相同的值時， Lisp 複製的是指標，而不是列表。（圖 3.4 顯式賦值 x 給 y 後的結果）無論何時，你將某個變量的值賦給一個變數時，兩個變數的值將會是 eql 的：

```
> (eql x y)
T
```

Lisp 沒有指標的原因是因爲每一個值，其實概念上來說都是一個指標。當你賦一個值給變數或將這個值存在資料結構中，其實被儲存的是指向這個值的指標。當你要取得變數的值，或是存在資料結構中的內容時， Lisp 返回指向這個值的指標。但這都在檯面下發生。你可以不加思索地把值放在結構裡，或放“在”變數裡。

爲了效率的原因， Lisp 有時會選擇一個折衷的表示法，而不是指標。舉例來說，因爲一個小整數所需的記憶體空間，少於一個指標所需的空間，一個 Lisp 實現可能會直接處理這個小整數，而不是用指標來處理。但基本要點是，程式設計師預設可以把任何東西放在任何地方。除非你宣告你不願這麼做，不然你能夠在任何的資料結構，存放任何型別的物件，包括結構本身。

3.4 建立列表 (Building Lists)

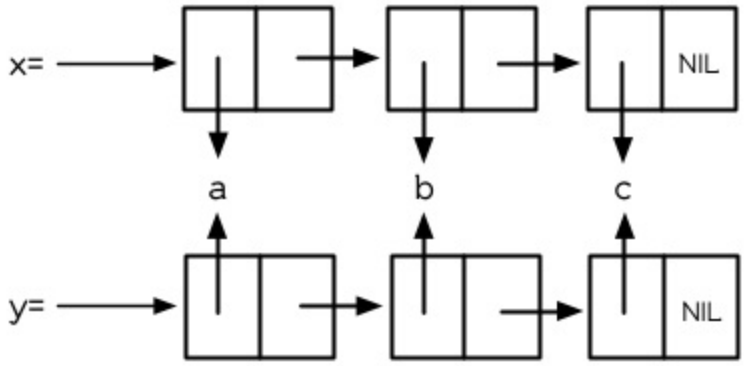

圖 3.5 複製的結果

函數 copy-list 接受一個列表，然後返回此列表的複本。新的列表會有同樣的元素，但是裝在新的 *Cons* 物件裡：

```
> (setf x '(a b c)
        y (copy-list x))
(A B C)
```

圖 3.5 展示出結果的結構; 返回值像是有著相同乘客的新公交。我們可以把 copy-list 想成是這麼定義的：

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

這個定義暗示著 x 與 (copy-list x) 會永遠 equal，並永遠不 eql，除非 x 是 NIL。

最後，函數 append 返回任何數目的列表串接 (concatenation)：

```
> (append '(a b) '(c d) 'e)
(A B C D . E)
```

通過這麼做，它複製所有的參數，除了最後一個

3.5 範例：壓縮 (Example: Compression)

作爲一個例子，這節將示範如何實現簡單形式的列表壓縮。這個算法有一個令人印象深刻的名字，遊程編碼(run-length encoding)。

```
(defun compress (x)
  (if (consp x)
      (compr (car x) 1 (cdr x))
      x))

(defun compr (elt n lst)
  (if (null lst)
      (list (n-elts elt n))
      (let ((next (car lst)))
        (if (eql next elt)
            (compr elt (+ n 1) (cdr lst))
            (cons (n-elts elt n)
                  (compr next 1 (cdr lst)))))))

(defun n-elts (elt n)
  (if (> n 1)
      (list n elt)
      elt))
```

圖 3.6 遊程編碼 (Run-length encoding)：壓縮

在餐廳的情境下，這個算法的工作方式如下。一個女服務生走向有四個客人的桌子。“你們要什麼？” 她問。“我要特餐，” 第一個客人說。 “我也是，” 第二個客人說。“聽起來不錯，” 第三個客人說。每個人看著第四個客人。 “我要一個 cilantro soufflé，” 他小聲地說。 (譯註：蛋奶酥上面灑香菜跟醬料）

瞬息之間，女服務生就轉身踩著高跟鞋走回櫃檯去了。 “三個特餐，” 她大聲對廚師

說，“還有一個香菜蛋奶酥。”

圖 3.6 示範了如何實現這個壓縮列表演算法。函數 compress 接受一個由原子組成的列表，然後返回一個壓縮的列表：

```
> (compress '(1 1 1 0 1 0 0 0 0 1))
((3 1) 0 1 (4 0) 1)
```

當相同的元素連續出現好幾次，這個連續出現的序列 (sequence)被一個列表取代，列表指明出現的次數及出現的元素。

主要的工作是由遞迴函數 compr 所完成。這個函數接受三個參數： elt ， 上一個我們看過的元素； n ，連續出現的次數；以及 lst ，我們還沒檢視過的部分列表。如果沒有東西需要檢視了，我們呼叫 n-elts 來取得 n elts 的表示法。如果 lst 的第一個元素還是 elt ，我們增加出現的次數 n 並繼續下去。否則我們得到，到目前爲止的一個壓縮的列表，然後 cons 這個列表在 compr 處理完剩下的列表所返回的東西之上。

要復原一個壓縮的列表，我們呼叫 uncompress (圖 3.7)

```
> (uncompress '((3 1) 0 1 (4 0) 1))
(1 1 1 0 1 0 0 0 0 1)
```

```
(defun uncompress (lst)
  (if (null lst)
      nil
      (let ((elt (car lst))
            (rest (uncompress (cdr lst))))
        (if (consp elt)
            (append (apply #'list-of elt)
                    rest)
            (cons elt rest)))))

(defun list-of (n elt)
  (if (zerop n)
      nil
      (cons elt (list-of (- n 1) elt))))
```

圖 3.7 遊程編碼 (Run-length encoding)：解壓縮

這個函數遞迴地遍歷這個壓縮列表，逐字複製原子並呼叫 list-of ，展開成列表。

```
> (list-of 3 'ho)
(HO HO HO)
```

我們其實不需要自己寫 list-of 。內建的 make-list 可以辦到一樣的事情 —— 但它使用

了我們還沒介紹到的關鍵字參數 (keyword argument)。

圖 3.6 跟 3.7 這種寫法不是一個有經驗的Lisp 程式設計師用的寫法。它的效率很差，它沒有儘可能的壓縮，而且它只對由原子組成的列表有效。在幾個章節內，我們會學到解決這些問題的技巧。

```
載入程式

在這節的程式是我們第一個實質的程式。
當我們想要寫超過數行的函數時，
通常我們會把程式寫在一個檔案，
然後使用 load 讓 Lisp 讀取函數的定義。
如果我們把圖 3.6 跟 3.7 的程式，
存在一個檔案叫做，"compress.lisp"然後輸入

(load "compress.lisp")

到頂層，或多或少的，
我們會像在直接輸入頂層一樣得到同樣的效果。

注意：在某些實現中，Lisp 檔案的擴展名會是".lsp"而不是".lisp"。
```

3.6 存取 (Access)

Common Lisp 有額外的存取函數，它們是用 `car` 跟 `cdr` 所定義的。要找到列表特定位置的元素，我們可以呼叫 `nth`，

```
> (nth 0 '(a b c))
A
```

而要找到第 `n` 個 `cdr`，我們呼叫 `nthcdr`：

```
> (nthcdr 2 '(a b c))
(C)
```

`nth` 與 `nthcdr` 都是零索引的 (zero-indexed); 即元素從 `0` 開始編號，而不是從 `1` 開始。在 Common Lisp 裡，無論何時你使用一個數字來參照一個資料結構中的元素時，都是從 `0` 開始編號的。

兩個函數幾乎做一樣的事; `nth` 等同於取 `nthcdr` 的 `car`。沒有檢查錯誤的情況下，`nthcdr` 可以這麼定義：

```
(defun our-nthcdr (n lst)
  (if (zerop n)
      lst
```

```
      (our-nthcdr (- n 1) (cdr lst))))
```

函數 `zerop` 僅在參數爲零時，才返回真。

函數 `last` 返回列表的最後一個 *Cons* 物件：

```
> (last '(a b c))
(C)
```

這跟取得最後一個元素不一樣。要取得列表的最後一個元素，你要取得 `last` 的 `car` 。

Common Lisp 定義了函數 `first` 直到 `tenth` 可以取得列表對應的元素。這些函數不是 零索引的 (zero-indexed)：

`(second x)` 等同於 `(nth 1 x)` 。

此外， Common Lisp 定義了像是 `caddr` 這樣的函數，它是 `cdr` 的 `cdr` 的 `car` 的縮寫 (`car` of `cdr` of `cdr`)。所有這樣形式的函數 `cxr` ，其中 x 是一個字串，最多四個 `a` 或 `d` ，在 Common Lisp 裡都被定義好了。使用 `cadr` 可能會有異常 (exception)產生，在所有人都可能會讀的程式裡，使用這樣的函數，可能不是個好主意。

3.7 映射函數 (Mapping Functions)

Common Lisp 提供了數個函數來對一個列表的元素做函數呼叫。最常使用的是 `mapcar` ，接受一個函數以及一個或多個列表，並返回把函數應用至每個列表的元素的結果，直到有的列表沒有元素爲止：

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

> (mapcar #'list
          '(a b c)
          '(1 2 3 4))
((A 1) (B 2) (C 3))
```

相關的 `maplist` 接受同樣的參數，將列表的漸進的下一個 `cdr` 傳入函數。

```
> (maplist #'(lambda (x) x)
           '(a b c))
((A B C) (B C) (C))
```

其它的映射函數，包括 `mapc` 我們在 89 頁討論（譯註：5.4 節最後），以及 `mapcan` 在 202 頁（譯註：12.4 節最後）討論。

3.8 樹 (Trees)

Cons 物件可以想成是二元樹， `car` 代表左子樹，而 `cdr` 代表右子樹。舉例來說，列表 `(a (b c) d)` 也是一棵由圖 3.8 所代表的樹。 （如果你逆時針旋轉 45 度，你會發現跟圖 3.3 一模一樣）

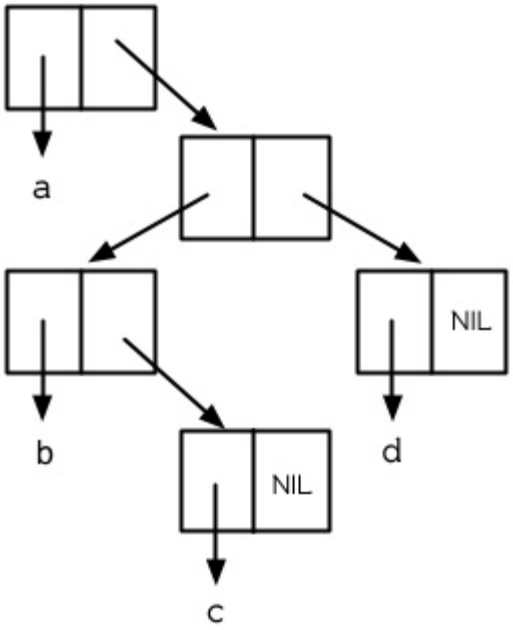

圖 3.8 二元樹 (Binary Tree)

Common Lisp 有幾個內建的操作樹的函數。舉例來說， `copy-tree` 接受一個樹，並返回一份副本。它可以這麼定義：

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

把這跟 36 頁的 `copy-list` 比較一下； `copy-tree` 複製每一個 *Cons* 物件的 `car` 與 `cdr` ，而 `copy-list` 僅複製 `cdr` 。

沒有內部節點的二元樹沒有太大的用處。 Common Lisp 包含了操作樹的函數，不只是因爲我們需要樹這個結構，而是因爲我們需要一種方法，來操作列表及所有內部的列表。舉例來說，假設我們有一個這樣的列表：

```
(and (integerp x) (zerop (mod x 2)))
```

而我們想要把各處的 x 都換成 y 。呼叫 substitute 是不行的，它只能替換序列(sequence)中的元素：

```
> (substitute 'y 'x '(and (integerp x) (zerop (mod x 2))))
(AND (INTEGERP X) (ZEROP (MOD X 2)))
```

這個呼叫是無效的，因爲列表有三個元素，沒有一個元素是 x 。我們在這所需要的是 subst ，它替換樹之中的元素。

```
> (subst 'y 'x '(and (integerp x) (zerop (mod x 2))))
(AND (INTEGERP Y) (ZEROP (MOD Y 2)))
```

如果我們定義一個 subst 的版本，它看起來跟 copy-tree 很相似：

```
> (defun our-subst (new old tree)
    (if (eql tree old)
        new
        (if (atom tree)
            tree
            (cons (our-subst new old (car tree))
                  (our-subst new old (cdr tree))))))
```

操作樹的函數通常有這種形式， car 與 cdr 同時做遞迴。這種函數被稱之爲是 雙重遞迴 (doubly recursive)。

3.9 理解遞迴 (Understanding Recursion)

學生在學習遞迴時，有時候是被鼓勵在紙上追蹤 (trace)遞迴程式呼叫 (invocation)的過程。 (288頁「譯註：附錄 A 追蹤與回溯 [http://acl.readthedocs.org/en/latest/zhCN/appendix-A-cn.html]」可以看到一個遞迴函數的追蹤過程。)但這種練習可能會誤導你：一個程式設計師在定義一個遞迴函數時，通常不會特別地去想函數的呼叫順序所導致的結果。

如果一個人總是需要這樣子思考程式，遞迴會是艱難的、沒有幫助的。遞迴的優點是它精確地讓我們更抽象地來檢視算法。你不需要考慮真正函數時所有的呼叫過程，就可以判斷一個遞迴函數是否是正確的。

要知道一個遞迴函數是否做它該做的事，你只需要問，它包含了所有的情況嗎？舉例來說，下面是一個尋找列表長度的遞迴函數：

```
> (defun len (lst)
    (if (null lst)
        0
        (+ (len (cdr lst)) 1)))
```

我們可以藉由檢查兩件事情，來確信這個函數是正確的：

1. 對長度爲 0 的列表是有效的。
2. 給定它對於長度爲 n 的列表是有效的，它對長度是 n+1 的列表也是有效的。

如果這兩點是成立的，我們知道這個函數對於所有可能的列表都是正確的。

我們的定義顯然地滿足第一點：如果列表(lst) 是空的(nil)，函數直接返回 0 。現在假定我們的函數對長度爲 n 的列表是有效的。我們給它一個 n+1 長度的列表。這個定義說明了，函數會返回列表的 cdr 的長度再加上 1 。 cdr 是一個長度爲 n 的列表。我們經由假定可知它的長度是 n 。所以整個列表的長度是 n+1 。

我們需要知道的就是這些。理解遞迴的祕密就像是處理括號一樣。你怎麼知道哪個括號對上哪個？你不需要這麼做。你怎麼想像那些呼叫過程？你不需要這麼做。

更複雜的遞迴函數，可能會有更多的情況需要討論，但是流程是一樣的。舉例來說，41 頁的 our-copy-tree ，我們需要討論三個情況： 原子，單一的 *Cons* 物件， n+1 的 *Cons* 樹。

第一個情況（長度零的列表）稱之爲基本用例(*base case*)。當一個遞迴函數不像你想的那樣工作時，通常是因爲基本用例是錯的。下面這個不正確的 member 定義，是一個常見的錯誤，整個忽略了基本用例：

```
(defun our-member (obj lst)
  (if (eql (car lst) obj)
      lst
      (our-member obj (cdr lst))))
```

我們需要初始一個 null 測試，確保在到達列表底部時，沒有找到目標時要停止遞迴。如果我們要找的物件沒有在列表裡，這個版本的 member 會陷入無窮迴圈。附錄 A 更詳細地檢視了這種問題。

能夠判斷一個遞迴函數是否正確只不過是理解遞迴的上半場，下半場是能夠寫出一個做你想做的事情的遞迴函數。 6.9 節討論了這個問題。

3.10 集合 (Sets)

列表是表示小集合的好方法。列表中的每個元素都代表了一個集合的成員：

```
> (member 'b '(a b c))
(B C)
```

當 member 要返回“真”時，與其僅僅返回 t ，它返回由尋找物件所開始的那部分。邏輯上來說，一個 *Cons* 扮演的角色和 t 一樣，而經由這麼做，函數返回了更多資訊。

一般情況下， member 使用 eql 來比較物件。你可以使用一種叫做關鍵字參數的東西來重寫預設的比較方法。多數的 Common Lisp 函數接受一個或多個關鍵字參數。這些關鍵字參數不同的地方是，他們不是把對應的參數放在特定的位置作匹配，而是在函數呼叫中用特殊標籤，稱爲關鍵字，來作匹配。一個關鍵字是一個前面有冒號的符號。

一個 member 函數所接受的關鍵字參數是 :test 參數。

如果你在呼叫 member 時，傳入某個函數作爲 :test 參數，那麼那個函數就會被用來比較是否相等，而不是用 eql 。所以如果我們想找到一個給定的物件與列表中的成員是否相等(equal)，我們可以：

```
> (member '(a) '((a) (z)) :test #'equal)
((A) (Z))
```

關鍵字參數總是選擇性添加的。如果你在一個呼叫中包含了任何的關鍵字參數，他們要擺在最後; 如果使用了超過一個的關鍵字參數，擺放的順序無關緊要。

另一個 member 接受的關鍵字參數是 :key 參數。藉由提供這個參數，你可以在作比較之前，指定一個函數運用在每一個元素：

```
> (member 'a '((a b) (c d)) :key #'car)
((A B) (C D))
```

在這個例子裡，我們詢問是否有一個元素的 car 是 a 。

如果我們想要使用兩個關鍵字參數，我們可以使用其中一個順序。下面這兩個呼叫是等價的：

```
> (member 2 '((1) (2)) :key #'car :test #'equal)
((2))
> (member 2 '((1) (2)) :test #'equal :key #'car)
((2))
```

兩者都詢問是否有一個元素的 car 等於(equal) 2。

如果我們想要找到一個元素滿足任意的判斷式像是—— oddp ，奇數返回真——我們可以使用相關的 member-if ：

```
> (member-if #'oddp '(2 3 4))
(3 4)
```

我們可以想像一個限制性的版本 member-if 是這樣寫成的：

```
(defun our-member-if (fn lst)
  (and (consp lst)
       (if (funcall fn (car lst))
           lst
           (our-member-if fn (cdr lst)))))
```

函數 adjoin 像是條件式的 cons 。它接受一個物件及一個列表，如果物件還不是列表的成員，才構造物件至列表上。

```
> (adjoin 'b '(a b c))
(A B C)
> (adjoin 'z '(a b c))
(Z A B C)
```

通常的情況下它接受與 member 函數同樣的關鍵字參數。

集合論中的並集 (union)、交集 (intersection)以及補集 (complement)的實現，是由函數 union 、 intersection 以及 set-difference 。

這些函數期望兩個（正好 2 個）列表（一樣接受與 member 函數同樣的關鍵字參數）。

```
> (union '(a b c) '(c b s))
(A C B S)
> (intersection '(a b c) '(b b c))
(B C)
> (set-difference '(a b c d e) '(b e))
(A C D)
```

因爲集閤中沒有順序的概念，這些函數不需要保留原本元素在列表被找到的順序。舉例來說，呼叫 set-difference 也有可能返回 (d c a) 。

3.11 序列 (Sequences)

另一種考慮一個列表的方式是想成一系列有特定順序的物件。在 Common Lisp 裡，序列(*sequences*)包括了列表與向量 (vectors)。本節介紹了一些可以運用在列表上的序列函數。更深入的序列操作在 4.4 節討論。

函數 length 返回序列中元素的數目。

```
> (length '(a b c))
3
```

我們在 24 頁 (譯註：2.13節 `our-length`)寫過這種函數的一個版本（僅可用於列表）。

要複製序列的一部分，我們使用 `subseq` 。第二個（需要的）參數是第一個開始引用進來的元素位置，第三個（選擇性）參數是第一個不引用進來的元素位置。

```
> (subseq '(a b c d) 1 2)
(B)
>(subseq '(a b c d) 1)
(B C D)
```

如果省略了第三個參數，子序列會從第二個參數給定的位置引用到序列尾端。

函數 `reverse` 返回與其參數相同元素的一個序列，但順序顛倒。

```
> (reverse '(a b c))
(C B A)
```

一個迴文 (palindrome) 是一個正讀反讀都一樣的序列 —— 舉例來說， `(abba)` 。如果一個迴文有偶數個元素，那麼後半段會是前半段的鏡射 (mirror)。使用 `length` 、 `subseq` 以及 `reverse` ，我們可以定義一個函數

```
(defun mirror? (s)
  (let ((len (length s)))
    (and (evenp len)
         (let ((mid (/ len 2)))
           (equal (subseq s 0 mid)
                  (reverse (subseq s mid)))))))
```

來檢測是否是迴文：

```
> (mirror? '(a b b a))
T
```

Common Lisp 有一個內建的排序函數叫做 `sort` 。它接受一個序列及一個比較兩個參數的函數，返回一個有同樣元素的序列，根據比較函數來排序：

```
> (sort '(0 2 1 3 8) #'>)
(8 3 2 1 0)
```

你要小心使用 `sort` ，因爲它是破壞性的(*destructive*)。考慮到效率的因素， `sort` 被允許修改傳入的序列。所以如果你不想你本來的序列被改動，傳入一個副本。

使用 `sort` 及 `nth` ，我們可以寫一個函數，接受一個整數 `n` ，返回列表中第 `n` 大的元素：

```
(defun nthmost (n lst)
  (nth (- n 1)
       (sort (copy-list lst) #'>)))
```

我們把整數減一因爲 nth 是零索引的，但如果 nthmost 是這樣的話，會變得很不直觀。

```
(nthmost 2 '(0 2 1 3 8))
```

多努力一點，我們可以寫出這個函數的一個更有效率的版本。

函數 every 和 some 接受一個判斷式及一個或多個序列。當我們僅輸入一個序列時，它們測試序列元素是否滿足判斷式：

```
> (every #'oddp '(1 3 5))
T
> (some #'evenp '(1 2 3))
T
```

如果它們輸入多於一個序列時，判斷式必須接受與序列一樣多的元素作爲參數，而參數從所有序列中一次提取一個：

```
> (every #'> '(1 3 5) '(0 2 4))
T
```

如果序列有不同的長度，最短的那個序列，決定需要測試的次數。

3.12 棧 (Stacks)

用 *Cons* 物件來表示的列表，很自然地我們可以拿來實現下推棧 (pushdown stack)。這太常見了，以致於 Common Lisp 提供了兩個宏給堆疊使用： (push x y) 把 x 放入列表 y 的前端。而 (pop x) 則是將列表 x 的第一個元素移除，並返回這個元素。

兩個函數都是由 setf 定義的。如果參數是常數或變數，很簡單就可以翻譯出對應的函數呼叫。

表達式

```
(push obj lst)
```

等同於

```
(setf lst (cons obj lst))
```

而表達式

```
(pop lst)
```

等同於

```
(let ((x (car lst)))
  (setf lst (cdr lst))
  x)
```

所以，舉例來說：

```
> (setf x '(b))
(B)
> (push 'a x)
(A B)
> x
(A B)
> (setf y x)
(A B)
> (pop x)
(A)
> x
(B)
> y
(A B)
```

以上，全都遵循上述由 `setf` 所給出的相等式。圖 3.9 展示了這些表達式被求值後的結構。

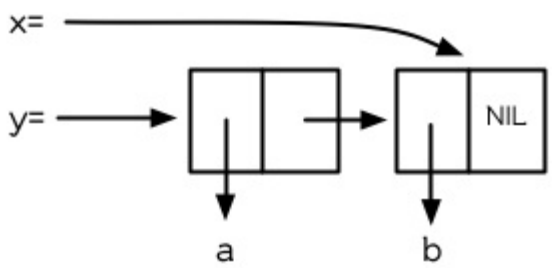

圖 3.9 push 及 pop 的效果

你可以使用 `push` 來定義一個給列表使用的互動版 `reverse` 。

```
(defun our-reverse (lst)
  (let ((acc nil))
    (dolist (elt lst)
      (push elt acc))
    acc))
```

在這個版本，我們從一個空列表開始，然後把 lst 的每一個元素放入空表裡。等我們完成時，lst 最後一個元素會在最前端。

pushnew 宏是 push 的變種，使用了 adjoin 而不是 cons：

```
 > (let ((x '(a b)))
     (pushnew 'c x)
     (pushnew 'a x)
     x)
(C A B)
```

在這裡， c 被放入列表，但是 a 沒有，因爲它已經是列表的一個成員了。

3.13 點狀列表 (Dotted Lists)

呼叫 list 所構造的列表，這種列表精確地說稱之爲正規列表(*proper*list)。一個正規列表可以是 NIL 或是 cdr 是正規列表的 *Cons* 物件。也就是說，我們可以定義一個只對正規列表返回真的判斷式： [3]

```
(defun proper-list? (x)
  (or (null x)
      (and (consp x)
           (proper-list? (cdr x)))))
```

至目前爲止，我們構造的列表都是正規列表。

然而， cons 不僅是構造列表。無論何時你需要一個具有兩個欄位 (field)的列表，你可以使用一個 *Cons* 物件。你能夠使用 car 來參照第一個欄位，用 cdr 來參照第二個欄位。

```
> (setf pair (cons 'a 'b))
(A . B)
```

因爲這個 *Cons* 物件不是一個正規列表，它用點狀表示法來顯示。在點狀表示法，每個 *Cons* 物件的 car 與 cdr 由一個句點隔開來表示。這個 *Cons* 物件的結構展示在圖 3.10 。

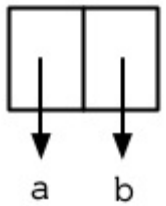

圖3.10 一個成對的 *Cons* 物件 (A cons used as a pair)

一個非正規列表的 *Cons* 物件稱之爲點狀列表 (dotted list)。這不是個好名字，因爲非正規列表的 Cons 物件通常不是用來表示列表：`(a . b)` 只是一個有兩部分的資料結構。

你也可以用點狀表示法表示正規列表，但當 Lisp 顯示一個正規列表時，它會使用普通的列表表示法：

```
> '(a . (b . (c . nil)))
(A B C)
```

順道一提，注意列表由點狀表示法與圖 3.2 箱子表示法的關聯性。

還有一個過渡形式 (intermediate form)的表示法，介於列表表示法及純點狀表示法之間，對於 `cdr` 是點狀列表的 *Cons* 物件：

```
> (cons 'a (cons 'b (cons 'c 'd)))
(A B C . D)
```

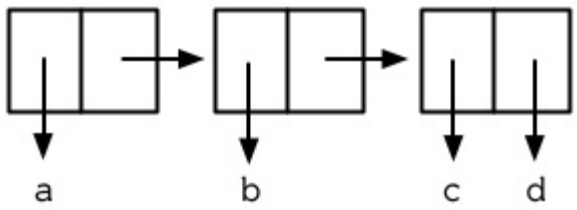

圖 3.11 一個點狀列表 (A dotted list)

這樣的 *Cons* 物件看起來像正規列表，除了最後一個 cdr 前面有一個句點。這個列表的結構展示在圖 3.11 ; 注意它跟圖3.2 是多麼的相似。

所以實際上你可以這麼表示列表 `(a b)`，

```
(a . (b . nil))
(a . (b))
(a b . nil)
(a b)
```

雖然 Lisp 總是使用後面的形式，來顯示這個列表。

3.14 關聯列表 (Assoc-lists)

用 *Cons* 物件來表示映射 (mapping)也是很自然的。一個由 *Cons* 物件組成的列表稱之爲

關聯列表(*assoc-list*or *alist*)。這樣的列表可以表示一個翻譯的集合，舉例來說：

```
> (setf trans '((+ . "add") (- . "subtract")))
((+ . "add") (- . "subtract"))
```

關聯列表很慢，但是在初期的程式中很方便。 Common Lisp 有一個內建的函數 `assoc`，用來取出在關聯列表中，與給定的鍵值有關聯的 *Cons* 對：

```
> (assoc '+ trans)
(+ . "add")
> (assoc '* trans)
NIL
```

如果 `assoc` 沒有找到要找的東西時，返回 `nil` 。

我們可以定義一個受限版本的 `assoc`：

```
(defun our-assoc (key alist)
  (and (consp alist)
       (let ((pair (car alist)))
         (if (eql key (car pair))
             pair
             (our-assoc key (cdr alist))))))
```

和 `member` 一樣，實際上的 `assoc` 接受關鍵字參數，包括 `:test` 和 `:key` 。 Common Lisp 也定義了一個 `assoc-if` 之於 `assoc`，如同 `member-if` 之於 `member` 一樣。

3.15 範例：最短路徑 (Example: Shortest Path)

圖 3.12 包含一個搜索網路中最短路徑的程式。函數 `shortest-path` 接受一個起始節點，目的節點以及一個網路，並返回最短路徑，如果有的話。

在這個範例中，節點用符號表示，而網路用含以下元素形式的關聯列表來表示：

(node . neighbors)

所以由圖 3.13 展示的最小網路 (minimal network)可以這樣來表示：

```
(setf min '((a b c) (b c) (c d)))
```

```
(defun shortest-path (start end net)
  (bfs end (list (list start)) net))

(defun bfs (end queue net)
  (if (null queue)
```

```
      nil
      (let ((path (car queue)))
        (let ((node (car path)))
          (if (eql node end)
              (reverse path)
              (bfs end
                   (append (cdr queue)
                           (new-paths path node net))
                   net))))))

(defun new-paths (path node net)
  (mapcar #'(lambda (n)
              (cons n path))
          (cdr (assoc node net))))
```

圖 3.12 廣度優先搜索(breadth-first search)

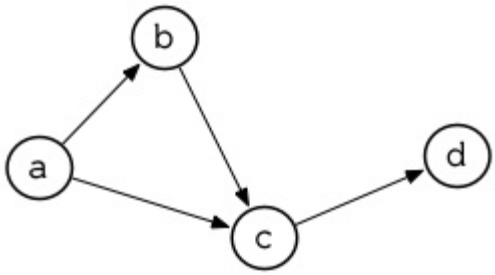

圖 3.13 最小網路

要找到從節點 a 可以到達的節點，我們可以：

```
> (cdr (assoc 'a min))
(B C)
```

圖 3.12 程式使用廣度優先的方式搜索網路。要使用廣度優先搜索，你需要維護一個含有未探索節點的佇列。每一次你到達一個節點，檢查這個節點是否是你要的。如果不是，你把這個節點的子節點加入佇列的尾端，並從佇列起始選一個節點，從這繼續搜索。藉由總是把較深的節點放在佇列尾端，我們確保網路一次被搜索一層。

圖 3.12 中的程式較不複雜地表示這個概念。我們不僅想要找到節點，還想保有我們怎麼到那的紀錄。所以與其維護一個具有節點的佇列 (queue)，我們維護一個已知路徑的佇列，每個已知路徑都是一列節點。當我們從佇列取出一個元素繼續搜索時，它是一個含有佇列前端節點的列表，而不只是一個節點而已。

函數 `bfs` 負責搜索。起初佇列只有一個元素，一個表示從起點開始的路徑。所以 `shortest-path` 呼叫 `bfs`，並傳入 `(list (list start))` 作爲初始佇列。

bfs 函數第一件要考慮的事是，是否還有節點需要探索。如果佇列爲空， bfs 返回 nil 指出沒有找到路徑。如果還有節點需要搜索， bfs 檢視佇列前端的節點。如果節點的 car 部分是我們要找的節點，我們返回這個找到的路徑，並且爲了可讀性的原因我們反轉它。如果我們沒有找到我們要找的節點，它有可能在現在節點之後，所以我們把它的子節點（或是每一個子路徑）加入佇列尾端。然後我們遞迴地呼叫 bfs 來繼續搜尋剩下的佇列。

因爲 bfs 廣度優先地搜索，第一個找到的路徑會是最短的，或是最短之一：

```
> (shortest-path 'a 'd min)
(A C D)
```

這是佇列在我們連續呼叫 bfs 看起來的樣子：

```
((A))
((B A) (C A))
((C A) (C B A))
((C B A) (D C A))
((D C A) (D C B A))
```

在佇列中的第二個元素變成下一個佇列的第一個元素。佇列的第一個元素變成下一個佇列尾端元素的 cdr 部分。

圖 3.12 的程式，不是搜索一個網路最快的方法，但它給出了列表具有多功能的概念。在這個簡單的程式中，我們用三種不同的方式使用了列表：我們使用一個符號的列表來表示路徑，一個路徑的列表來表示在廣度優先搜索中的佇列 [4] ，以及一個關聯列表來表示網路本身。

3.16 垃圾 (Garbages)

有很多原因可以使列表變慢。列表提供了循序存取而不是隨機存取，所以列表取出一個指定的元素比陣列慢，同樣的原因，錄音帶取出某些東西比在光盤上慢。電腦內部裡，*Cons* 物件傾向於用指標表示，所以走訪一個列表意味著走訪一系列的指標，而不是簡單地像陣列一樣增加索引值。但這兩個所花的代價與配置及回收 *Cons* 核 (cons cells)比起來小多了。

自動記憶體管理(*Automatic memory management*)是 Lisp 最有價值的特色之一。 Lisp 系統維護著一段記憶體稱之爲堆疊(*Heap*)。系統持續追蹤堆疊當中沒有使用的記憶體，把這些記憶體發放給新產生的物件。舉例來說，函數 cons ，返回一個新配置的 *Cons* 物件。從堆疊中配置記憶體有時候通稱爲 *consing* 。

如果記憶體永遠沒有釋放， Lisp 會因爲創建新物件把記憶體用完，而必須要關閉。所

以系統必須週期性地通過搜索堆疊 (heap)，尋找不需要再使用的記憶體。不需要再使用的記憶體稱之爲垃圾 (*garbage*)，而清除垃圾的動作稱爲垃圾回收 (*garbage collection*或GC)。

垃圾是從哪來的？讓我們來創造一些垃圾：

```
> (setf lst (list 'a 'b 'c))
(A B C)
> (setf lst nil)
NIL
```

一開始我們呼叫 `list`， `list` 呼叫 `cons`，在堆疊上配置了一個新的 *Cons* 物件。在這個情況我們創出三個 *Cons* 物件。之後當我們把 `lst` 設爲 `nil`，我們沒有任何方法可以再存取 `lst`，列表 `(a b c)`。 [5]

因爲我們沒有任何方法再存取列表，它也有可能是不存在的。我們不再有任何方式可以存取的物件叫做垃圾。系統可以安全地重新使用這三個 *Cons* 核。

這種管理記憶體的方法，給程式設計師帶來極大的便利性。你不用顯式地配置 (allocate)或釋放 (dellocate)記憶體。這也表示了你不需要處理因爲這麼做而可能產生的臭蟲。記憶體泄漏 (Memory leaks)以及迷途指標 (dangling pointer)在 Lisp 中根本不可能發生。

但是像任何的科技進步，如果你不小心的話，自動記憶體管理也有可能對你不利。使用及回收堆疊所帶來的代價有時可以看做 `cons` 的代價。這是有理的，除非一個程式從來不丟棄任何東西，不然所有的 *Cons* 物件終究要變成垃圾。 Consing 的問題是，配置空間與清除記憶體，與程式的常規運作比起來花費昂貴。近期的研究提出了大幅改善記憶體回收的演算法，但是 consing 總是需要代價的，在某些現有的 Lisp 系統中，代價是昂貴的。

除非你很小心，不然很容易寫出過度顯式創建 cons 物件的程式。舉例來說， `remove` 需要複製所有的 `cons` 核，直到最後一個元素從列表中移除。你可以藉由使用破壞性的函數避免某些 consing，它試著去重用列表的結構作爲參數傳給它們。破壞性函數會在 12.4 節討論。

當寫出 `cons` 很多的程式是如此簡單時，我們還是可以寫出不使用 `cons` 的程式。典型的方法是寫出一個純函數風格，使用很多列表的第一版程式。當程式進化時，你可以在程式的關鍵部分使用破壞性函數以及/或別種資料結構。但這很難給出通用的建議，因爲有些 Lisp 實現，記憶體管理處理得相當好，以致於使用 `cons` 有時比不使用 `cons` 還快。這整個議題在 13.4 做更進一步的細部討論。

無論如何 consing 在原型跟實驗時是好的。而且如果你利用了列表給你帶來的靈活性，你有較高的可能寫出後期可存活下來的程式。

Chapter 3 總結 (Summary)

1. 一個 *Cons* 是一個含兩部分的資料結構。列表用鏈結在一起的 *Cons* 組成。
2. 判斷式 `equal` 比 `eql` 來得不嚴謹。基本上，如果傳入參數印出來的值一樣時，返回真。
3. 所有 Lisp 物件表現得像指標。你永遠不需要顯式操作指標。
4. 你可以使用 `copy-list` 複製列表，並使用 `append` 來連接它們的元素。
5. 遊程編碼是一個餐廳中使用的簡單壓縮演算法。
6. Common Lisp 有由 `car` 與 `cdr` 定義的多種存取函數。
7. 映射函數將函數應用至逐項的元素，或逐項的列表尾端。
8. 巢狀列表的操作有時被考慮爲樹的操作。
9. 要判斷一個遞迴函數是否正確，你只需要考慮是否包含了所有情況。
10. 列表可以用來表示集合。數個內建函數把列表當作集合。
11. 關鍵字參數是選擇性的，並不是由位置所識別，是用符號前面的特殊標籤來識別。
12. 列表是序列的子型別。 Common Lisp 有大量的序列函數。
13. 一個不是正規列表的 *Cons* 稱之爲點狀列表。
14. 用 cons 物件作爲元素的列表，可以拿來表示對應關係。這樣的列表稱爲關聯列表 (assoc-lists)。
15. 自動記憶體管理拯救你處理記憶體配置的煩惱，但製造過多的垃圾會使程式變慢。

Chapter 3 習題 (Exercises)

1. 用箱子表示法表示以下列表：

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

2. 寫一個保留原本列表中元素順序的 `union` 版本：

```
> (new-union '(a b c) '(b a d))
(A B C D)
```

3. 定義一個函數，接受一個列表並返回一個列表，指出相等元素出現的次數，並由最常見至最少見的排序：

```
> (occurrences '(a b a d a c d c a))
((A . 4) (C . 2) (D . 2) (B . 1))
```

4. 爲什麼 `(member '(a) '((a) (b)))` 返回 nil?

5. 假設函數 `pos+` 接受一個列表並返回把每個元素加上自己的位置的列表：

```
> (pos+ '(7 5 1 4))
(7 6 3 7)
```

使用 (a) 遞迴 (b) 迭代 (c) `mapcar` 來定義這個函數。

6. 經過好幾年的審議，政府委員會決定列表應該由 `cdr` 指向第一個元素，而 `car` 指向剩下的列表。定義符合政府版本的以下函數：

```
(a) cons
(b) list
(c) length (for lists)
(d) member (for lists; no keywords)
```

勘誤：要解決 3.6 (b)，你需要使用到 6.3 節的參數 `&rest` 。

7. 修改圖 3.6 的程式，使它使用更少 cons 核。 （提示：使用點狀列表）
8. 定義一個函數，接受一個列表並用點狀表示法印出：

```
> (showdots '(a b c))
(A . (B . (C . NIL)))
NIL
```

9. 寫一個程式來找到 3.15 節裡表示的網路中，最長有限的路徑 (不重複)。網路可能包含迴圈。

腳註

[3] 這個敘述有點誤導，因爲只要是對任何東西都不返回 nil 的函數，都不是正規列表。如果給定一個環狀 cdr 列表(cdr-circular list)，它會無法終止。環狀列表在 12.7 節 討論。

[4] 12.3 小節會展示更有效率的佇列實現方式。

[5] 事實上，我們有一種方式來存取列表。全局變數 `*` , `**` , 以及 `***` 總是設定爲最後三個頂層所返回的值。這些變數在除錯的時候很有用。

第四章：特殊資料結構

在之前的章節裡，我們討論了列表，Lisp 最多功能的資料結構。本章將示範如何使用 Lisp 其它的資料結構：陣列（包含向量與字串），結構以及雜湊表。它們或許不像列表這麼靈活，但存取速度更快並使用了更少空間。

Common Lisp 還有另一種資料結構：實體（instance）。實體將在 11 章討論，講述 CLOS。

4.1 陣列 (Array)

在 Common Lisp 裡，你可以呼叫 `make-array` 來構造一個陣列，第一個實參爲一個指定陣列維度的列表。要構造一個 `2 x 3` 的陣列，我們可以：

```
> (setf arr (make-array '(2 3) :initial-element nil))
#<Simple-Array T (2 3) BFC4FE>
```

Common Lisp 的陣列至少可以有七個維度，每個維度至多可以有 1023 個元素。

`:initial-element` 實參是選擇性的。如果有提供這個實參，整個陣列會用這個值作爲初始值。若試著取出未初始化的陣列內的元素，其結果爲未定義（undefined）。

用 `aref` 取出陣列內的元素。與 Common Lisp 的存取函數一樣， `aref` 是零索引的（zero-indexed）：

```
> (aref arr 0 0)
NIL
```

要替換陣列的某個元素，我們使用 `setf` 與 `aref`：

```
> (setf (aref arr 0 0) 'b)
B
> (aref arr 0 0)
B
```

要表示字面常數的陣列（literal array），使用 `#na` 語法，其中 `n` 是陣列的維度。舉例來說，我們可以這樣表示 `arr` 這個陣列：

```
#2a((b nil nil) (nil nil nil))
```

如果全局變數 *print-array* 爲真，則陣列會用以下形式來顯示：

```
> (setf *print-array* t)
T
> arr
#2A((B NIL NIL) (NIL NIL NIL))
```

如果我們只想要一維的陣列，你可以給 make-array 第一個實參傳一個整數，而不是一個列表：

```
> (setf vec (make-array 4 :initial-element nil))
#(NIL NIL NIL NIL)
```

一維陣列又稱爲向量（*vector*）。你可以通過呼叫 vector 來一步驟構造及填滿向量，向量的元素可以是任何型別：

```
> (vector "a" 'b 3)
#("a" b 3)
```

字面常數的陣列可以表示成 #na ，字面常數的向量也可以用這種語法表達。

可以用 aref 來存取向量，但有一個更快的函數叫做 svref ，專門用來存取向量。

```
> (svref vec 0)
NIL
```

在 svref 內的 “sv” 代表“簡單向量”（“simple vector”），所有的向量預設是簡單向量。[1]

4.2 範例：二元搜索 (Example: Binary Search)

作爲一個範例，這小節示範如何寫一個在排序好的向量裡搜索物件的函數。如果我們知道一個向量是排序好的，我們可以比（65頁） find 做的更好， find 必須依序檢視每一個元素。我們可以直接跳到向量中間開始找。如果中間的元素是我們要找的物件，搜索完畢。要不然我們持續往左半部或往右半部搜索，取決於物件是小於或大於中間的元素。

圖 4.1 包含了一個這麼工作的函數。其實這兩個函數： bin-search 設置初始範圍及發送控制信號給 finder ， finder 尋找向量 vec 內 obj 是否介於 start 及 end 之間。

```
(defun bin-search (obj vec)
  (let ((len (length vec)))
    (and (not (zerop len))
```

```
          (finder obj vec 0 (- len 1)))))

(defun finder (obj vec start end)
  (let ((range (- end start)))
    (if (zerop range)
        (if (eql obj (aref vec start))
            obj
            nil)
        (let ((mid (+ start (round (/ range 2)))))
          (let ((obj2 (aref vec mid)))
            (if (< obj obj2)
                (finder obj vec start (- mid 1))
                (if (> obj obj2)
                    (finder obj vec (+ mid 1) end)
                    obj)))))))
```

圖 4.1: 搜索一個排序好的向量

如果要找的 range 縮小至一個元素，而如果這個元素是 obj 的話，則 finder 直接返回這個元素，反之返回 nil 。如果 range 大於 1 ，我們設置 middle (round 返回離實參最近的整數) 為 obj2 。如果 obj 小於 obj2 ，則遞迴地往向量的左半部尋找。如果 obj 大於 obj2 ，則遞迴地往向量的右半部尋找。剩下的一個選擇是 obj=obj2 ，在這個情況我們找到要找的元素，直接返回這個元素。

如果我們插入下面這行至 finder 的起始處：

```
(format t "~A~%" (subseq vec start (+ end 1)))
```

我們可以觀察被搜索的元素的數量，是每一步往左減半的：

```
> (bin-search 3 #(0 1 2 3 4 5 6 7 8 9))
#(0 1 2 3 4 5 6 7 8 9)
#(0 1 2 3)
#(3)
3
```

4.3 字元與字串 (Strings and Characters)

字串是字元組成的向量。我們用一系列由雙引號包住的字元，來表示一個字串常數，而字元 c 用 #\c 表示。

每個字元都有一個相關的整數 —— 通常是 ASCII 碼，但不一定是。在多數的 Lisp 實現裡，函數 char-code 返回與字元相關的數字，而 code-char 返回與數字相關的字元。

字元比較函數 char< （小於）， char<= （小於等於)， char= （等於)， char>= （大於

等於)， char> （大於)，以及 char/= （不同)。他們的工作方式和 146 頁（譯註 9.3 節)比較數字用的運算子一樣。

```
> (sort "elbow" #'char<)
"below"
```

由於字串是字元向量，序列與陣列的函數都可以用在字串。你可以用 aref 來取出元素，舉例來說，

```
> (aref "abc" 1)
#\b
```

但針對字串可以使用更快的 char 函數：

```
> (char "abc" 1)
#\b
```

可以使用 setf 搭配 char （或 aref ）來替換字串的元素：

```
> (let ((str (copy-seq "Merlin")))
   (setf (char str 3) #\k)
   str)
```

如果你想要比較兩個字串，你可以使用通用的 equal 函數，但還有一個比較函數，是忽略字母大小寫的 string-equal ：

```
> (equal "fred" "fred")
T
> (equal "fred" "Fred")
NIL
>(string-equal "fred" "Fred")
T
```

Common Lisp 提供大量的操控、比較字串的函數。收錄在附錄 D，從 364 頁開始。

有許多方式可以創建字串。最普遍的方式是使用 format 。將第一個參數設爲 nil 來呼叫 format ，使它返回一個原本會印出來的字串：

```
> (format nil "~A or ~A" "truth" "dare")
"truth or dare"
```

但若你只想把數個字串連結起來，你可以使用 concatenate ，它接受一個特定型別的符號，加上一個或多個序列：

```
> (concatenate 'string "not " "to worry")
"not to worry"
```

4.4 序列 (Sequences)

在 Common Lisp 裡，序列型別包含了列表與向量（因此也包含了字串）。有些用在列表的函數，實際上是序列函數，包括 `remove` 、 `length` 、 `subseq` 、 `reverse` 、 `sort` 、 `every` 以及 `some` 。所以 46 頁（譯註 3.11 小節的 `mirror?` 函數）我們所寫的函數，也可以用在別種序列上：

```
> (mirror? "abba")
T
```

我們已經看過四種用來取出序列元素的函數： 給列表使用的 `nth` ， 給向量使用的 `aref` 及 `svref` ，以及給字串使用的 `char` 。 Common Lisp 也提供了通用的 `elt` ，對任何種類的序列都有效：

```
> (elt '(a b c) 1)
B
```

針對特定型別的序列，特定的存取函數會比較快，所以使用 `elt` 是沒有意義的，除非在程式當中，有需要支援通用序列的地方。

使用 `elt` ，我們可以寫一個針對向量來說更有效率的 `mirror?` 版本：

```
(defun mirror? (s)
  (let ((len (length s)))
    (and (evenp len)
         (do ((forward 0 (+ forward 1))
              (back (- len 1) (- back 1)))
             ((or (> forward back)
                  (not (eql (elt s forward)
                            (elt s back))))
              (> forward back))))))
```

這個版本也可用在列表，但這個實現更適合給向量使用。頻繁的對列表呼叫 `elt` 的代價是昂貴的，因爲列表僅允許循序存取。而向量允許隨機存取，從任何元素來存取每一個元素都是廉價的。

許多序列函數接受一個或多個，由下表所列的標準關鍵字參數：

參數	用途	預設值
:key	應用至每個元素的函數	identity

:test	作來比較的函數	eql
:from-end	若爲真，反向工作。	nil
:start	起始位置	0
:end	若有給定，結束位置。	nil

一個接受所有關鍵字參數的函數是 `position` ，返回序列中一個元素的位置，未找到元素時則返回 `nil` 。我們使用 `position` 來示範關鍵字參數所扮演的角色。

```
> (position #\a "fantasia")
1
> (position #\a "fantasia" :start 3 :end 5)
4
```

第二個例子我們要找在第四個與第六個字元間，第一個 `a` 所出現的位置。 `:start` 關鍵字參數是第一個被考慮的元素位置，預設是序列的第一個元素。 `:end` 關鍵字參數，如果有給的話，是第一個不被考慮的元素位置。

如果我們給入 `:from-end` 關鍵字參數，

```
> (position #\a "fantasia" :from-end t)
7
```

我們得到最靠近結尾的 `a` 的位置。但位置是像平常那樣計算；而不是從尾端算回來的距離。

`:key` 關鍵字參數是序列中每個元素在被考慮之前，應用至元素上的函數。如果我們說，

```
> (position 'a '((c d) (a b)) :key #'car)
1
```

那麼我們要找的是，元素的 `car` 部分是符號 `a` 的第一個元素。

`:test` 關鍵字參數接受需要兩個實參的函數，並定義了怎樣是一個成功的匹配。預設函數爲 `eql` 。如果你想要匹配一個列表，你也許想使用 `equal` 來取代：

```
> (position '(a b) '((a b) (c d)))
NIL
> (position '(a b) '((a b) (c d)) :test #'equal)
0
```

`:test` 關鍵字參數可以是任何接受兩個實參的函數。舉例來說，給定 `<` ，我們可以詢問第一個使第一個參數比它小的元素位置：

```
> (position 3 '(1 0 7 5) :test #'<)
2
```

使用 subseq 與 position，我們可以寫出分開序列的函數。舉例來說，這個函數

```
(defun second-word (str)
  (let ((p1 (+ (position #\  str) 1)))
    (subseq str p1 (position #\  str :start p1))))
```

返回字串中第一個單字空格後的第二個單字：

```
> (second-word "Form follows function")
"follows"
```

要找到滿足謂詞的元素，其中謂詞接受一個實參，我們使用 position-if 。它接受一個函數與序列，並返回第一個滿足此函數的元素：

```
> (position-if #'oddp '(2 3 4 5))
1
```

position-if 接受除了 :test 之外的所有關鍵字參數。

有許多相似的函數，如給序列使用的 member 與 member-if 。分別是， find （接受全部關鍵字參數）與 find-if （接受除了 :test 之外的所有關鍵字參數）：

```
> (find #\a "cat")
#\a

> (find-if #'characterp "ham")
#\h
```

不同於 member 與 member-if ，它們僅返回要尋找的物件。

通常一個 find-if 的呼叫，如果解讀爲 find 搭配一個 :key 關鍵字參數的話，會顯得更清楚。舉例來說，表達式

```
(find-if #'(lambda (x)
             (eql (car x) 'complete))
         lst)
```

可以更好的解讀爲

```
(find 'complete lst :key #'car)
```

函數 remove （22 頁）以及 remove-if 通常都可以用在序列。它們跟 find 與 find-if

是一樣的關係。另一個相關的函數是 `remove-duplicates`，僅保留序列中每個元素的最後一次出現。

```
> (remove-duplicates "abracadabra")
"cdbra"
```

這個函數接受前表所列的所有關鍵字參數。

函數 `reduce` 用來把序列壓縮成一個值。它至少接受兩個參數，一個函數與序列。函數必須是接受兩個實參的函數。在最簡單的情況下，一開始函數用序列前兩個元素作爲實參來呼叫，之後接續的元素作爲下次呼叫的第二個實參，而上次返回的值作爲下次呼叫的第一個實參。最後呼叫最終返回的值作爲 `reduce` 整個函數的返回值。也就是說像是這樣的表達式：

```
(reduce #'fn '(a b c d))
```

等同於

```
(fn (fn (fn 'a 'b) 'c) 'd)
```

我們可以使用 `reduce` 來擴充只接受兩個參數的函數。舉例來說，要得到三個或多個列表的交集(intersection)，我們可以：

```
> (reduce #'intersection '((b r a d 's) (b a d) (c a t)))
(A)
```

4.5 範例：解析日期 (Example: Parsing Dates)

作爲序列操作的範例，本節示範了如何寫程式來解析日期。我們將編寫一個程式，可以接受像是 “16 Aug 1980” 的字串，然後返回一個表示日、月、年的整數列表。

```
(defun tokens (str test start)
  (let ((p1 (position-if test str :start start)))
    (if p1
        (let ((p2 (position-if #'(lambda (c)
                                   (not (funcall test c)))
                               str :start p1)))
          (cons (subseq str p1 p2)
                (if p2
                    (tokens str test p2)
                    nil)))
        nil)))

(defun constituent (c)
```

```
  (and (graphic-char-p c)
       (not (char= c #\ ))))
```

圖 4.2 辨別符號 (token)

圖 4.2 裡包含了某些在這個應用裡所需的通用解析函數。第一個函數 `tokens`，用來從字串中取出語元（token）。給定一個字串及測試函數，滿足測試函數的字元組成子字串，子字串再組成列表返回。舉例來說，如果測試函數是對字母返回真的 `alpha-char-p` 函數，我們得到：

```
> (tokens "ab12 3cde.f" #'alpha-char-p 0)
("ab" "cde" "f")
```

所有不滿足此函數的字元被視爲空白 —— 他們是語元的分隔符，但永遠不是語元的一部分。

函數 `constituent` 被定義成用來作爲 `tokens` 的實參。

在 Common Lisp 裡，圖形字元是我們可見的字元，加上空白字元。所以如果我們用 `constituent` 作爲測試函數時，

```
> (tokens "ab12 3cde.f gh" #'constituent 0)
("ab12" "3cde.f" "gh")
```

則語元將會由空白區分出來。

圖 4.3 包含了特別爲解析日期打造的函數。函數 `parse-date` 接受一個特別形式組成的日期，並返回代表這個日期的整數列表：

```
> (parse-date "16 Aug 1980")
(16 8 1980)
```

```
(defun parse-date (str)
  (let ((toks (tokens str #'constituent 0)))
    (list (parse-integer (first toks))
          (parse-month (second toks))
          (parse-integer (third toks)))))

(defconstant month-names
  #("jan" "feb" "mar" "apr" "may" "jun"
    "jul" "aug" "sep" "oct" "nov" "dec"))

(defun parse-month (str)
  (let ((p (position str month-names
                     :test #'string-equal)))
    (if p
```

```
        (+ p 1)
        nil)))
```

圖 4.3 解析日期的函數

`parse-date` 使用 `tokens` 來解析日期字串，接著呼叫 `parse-month` 及 `parse-integer` 來轉譯年、月、日。要找到月份，呼叫 `parse-month` ，由於使用的是 `string-equal` 來匹配月份的名字，所以輸入可以不分大小寫。要找到年和日，呼叫內建的 `parse-integer` ， `parse-integer` 接受一個字串並返回對應的整數。

如果需要自己寫程式來解析整數，也許可以這麼寫：

```
(defun read-integer (str)
  (if (every #'digit-char-p str)
      (let ((accum 0))
        (dotimes (pos (length str))
          (setf accum (+ (* accum 10)
                         (digit-char-p (char str pos)))))
        accum)
    nil))
```

這個定義示範了在 Common Lisp 中，字元是如何轉成數字的 —— 函數 `digit-char-p` 不僅測試字元是否爲數字，同時返回了對應的整數。

4.6 結構 (Structures)

結構可以想成是豪華版的向量。假設你要寫一個程式來追蹤長方體。你可能會想用三個向量元素來表示長方體：高度、寬度及深度。與其使用原本的 `svref` ，不如定義像是下面這樣的抽象，程式會變得更容易閱讀，

```
(defun block-height (b) (svref b 0))
```

而結構可以想成是，這些函數通通都替你定義好了的向量。

要想定義結構，使用 `defstruct` 。在最簡單的情況下，只要給出結構及欄位的名字便可以了：

```
(defstruct point
  x
  y)
```

這裡定義了一個 `point` 結構，具有兩個欄位 `x` 與 `y` 。同時隱式地定義了 `make-point` 、 `point-p` 、 `copy-point` 、 `point-x` 及 `point-y` 函數。

2.3 節提過， Lisp 程式可以寫出 Lisp 程式。這是目前所見的明顯例子之一。當你呼叫 defstruct 時，它自動生成了其它幾個函數的定義。有了宏以後，你將可以自己來辦到同樣的事情（如果需要的話，你甚至可以自己寫出 defstruct ）。

每一個 make-point 的呼叫，會返回一個新的 point 。可以通過給予對應的關鍵字參數，來指定單一欄位的值：

```
(setf p (make-point :x 0 :y 0))
#S(POINT X 0 Y 0)
```

存取 point 欄位的函數不僅被定義成可取出數值，也可以搭配 setf 一起使用。

```
> (point-x p)
0
> (setf (point-y p) 2)
2
> p
#S(POINT X 0 Y 2)
```

定義結構也定義了以結構爲名的型別。每個點的型別層級會是，型別 point ，接著是型別 structure ，再來是型別 atom ，最後是 t 型別。所以使用 point-p 來測試某個東西是不是一個點時，也可以使用通用性的函數，像是 typep 來測試。

```
> (point-p p)
T
> (typep p 'point)
T
```

我們可以在本來的定義中，附上一個列表，含有欄位名及預設表達式，來指定結構欄位的預設值。

```
(defstruct polemic
  (type (progn
          (format t "What kind of polemic was it? ")
          (read)))
  (effect nil))
```

如果 make-polemic 呼叫沒有給欄位指定初始值，則欄位會被設成預設表達式的值：

```
> (make-polemic)
What kind of polemic was it? scathing
#S(POLEMIC :TYPE SCATHING :EFFECT NIL)
```

結構顯示的方式也可以控制，以及結構自動產生的存取函數的字首。以下是做了前述兩件事的 point 定義：

```
(defstruct (point (:conc-name p)
                  (:print-function print-point))
  (x 0)
  (y 0))

(defun print-point (p stream depth)
  (format stream "#<~A, ~A>" (px p) (py p)))
```

`:conc-name` 關鍵字參數指定了要放在欄位前面的名字，並用這個名字來生成存取函數。預設是 `point-` ；現在變成只有 `p` 。不使用預設的方式使程式的可讀性些微降低了，只有在需要常常用到這些存取函數時，你才會想取個短點的名字。

`:print-function` 是在需要顯示結構出來看時，指定用來打印結構的函數 —— 需要顯示的情況比如，要在頂層顯示時。這個函數需要接受三個實參：要被印出的結構，在哪裡被印出，第三個參數通常可以忽略。 [2] 我們會在 7.1 節討論流（stream）。現在來說，只要知道流可以作爲參數傳給 `format` 就好了。

函數 `print-point` 會用縮寫的形式來顯示點：

```
> (make-point)
#<0,0>
```

4.7 範例：二元搜索樹 (Example: Binary Search Tree)

由於 `sort` 本身系統就有了，極少需要在 Common Lisp 裡編寫排序程式。本節將示範如何解決一個與此相關的問題，這個問題尚未有現成的解決方案：維護一個已排序的物件集合。本節的程式會把物件存在二元搜索樹裡（ *binary search tree* ）或稱作 BST。當二元搜索樹平衡時，允許我們可以在與時間成 `log n` 比例的時間內，來尋找、添加或是刪除元素，其中 `n` 是集合的大小。

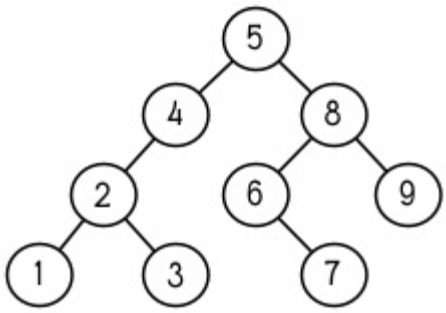

圖 4.4: 二元搜索樹

二元搜索樹是一種二元樹，給定某個排序函數，比如 `<` ，每個元素的左子樹都 `<` 該元素，而該元素 `<` 其右子樹。圖 4.4 展示了根據 `<` 排序的二元樹。

圖 4.5 包含了二元搜索樹的插入與尋找的函數。基本的資料結構會是 node （節點），節點有三個部分：一個欄位表示存在該節點的物件，以及各一個欄位表示節點的左子樹及右子樹。可以把節點想成是有一個 car 和兩個 cdr 的一個 cons 核（cons cell）。

```
(defstruct (node (:print-function
                  (lambda (n s d)
                    (format s "#<~A>" (node-elt n)))))
  elt (l nil) (r nil))

(defun bst-insert (obj bst <)
  (if (null bst)
      (make-node :elt obj)
      (let ((elt (node-elt bst)))
        (if (eql obj elt)
            bst
            (if (funcall < obj elt)
                (make-node
                   :elt elt
                   :l (bst-insert obj (node-l bst) <)
                   :r (node-r bst))
                (make-node
                   :elt elt
                   :r (bst-insert obj (node-r bst) <)
                   :l (node-l bst)))))))

(defun bst-find (obj bst <)
  (if (null bst)
      nil
      (let ((elt (node-elt bst)))
        (if (eql obj elt)
            bst
            (if (funcall < obj elt)
                (bst-find obj (node-l bst) <)
                (bst-find obj (node-r bst) <))))))

(defun bst-min (bst)
  (and bst
       (or (bst-min (node-l bst)) bst)))

(defun bst-max (bst)
  (and bst
       (or (bst-max (node-r bst)) bst)))
```

圖 4.5 二元搜索樹：查詢與插入

一棵二元搜索樹可以是 nil 或是一個左子、右子樹都是二元搜索樹的節點。如同列表可由連續呼叫 cons 來構造，二元搜索樹將可以通過連續呼叫 bst-insert 來構造。這個函數接受一個物件，一棵二元搜索樹及一個排序函數，並返回將物件插入的二元搜索樹。和 cons 函數一樣， bst-insert 不改動做爲第二個實參所傳入的二元搜索樹。以下是如何使用這個函數來構造一棵叉搜索樹：

```
> (setf nums nil)
NIL
> (dolist (x '(5 8 4 2 1 9 6 7 3))
    (setf nums (bst-insert x nums #'<)))
NIL
```

圖 4.4 顯示了此時 nums 的結構所對應的樹。

我們可以使用 bst-find 來找到二元搜索樹中的物件，它與 bst-insert 接受同樣的參數。先前敘述所提到的 node 結構，它像是一個具有兩個 cdr 的 cons 核。如果我們把 16 頁的 our-member 拿來與 bst-find 比較的話，這樣的類比更加明確。

與 member 相同， bst-find 不僅返回要尋找的元素，也返回了用尋找元素做爲根節點的子樹：

```
> (bst-find 12 nums #'<)
NIL
> (bst-find 4 nums #'<)
#<4>
```

這使我們可以區分出無法找到某物，以及成功找到 nil 的情況。

要找到二元搜索樹的最小及最大的元素是很簡單的。要找到最小的，我們沿著左子樹的路徑走，如同 bst-min 所做的。要找到最大的，沿著右子樹的路徑走，如同 bst-max 所做的：

```
> (bst-min nums)
#<1>
> (bst-max nums)
#<9>
```

要從二元搜索樹裡移除元素一樣很快，但需要更多程式碼。圖 4.6 示範了如何從二元搜索樹裡移除元素。

```
(defun bst-remove (obj bst <)
  (if (null bst)
      nil
      (let ((elt (node-elt bst)))
        (if (eql obj elt)
            (percolate bst)
            (if (funcall < obj elt)
                (make-node
                   :elt elt
                   :l (bst-remove obj (node-l bst) <)
                   :r (node-r bst))
                (make-node
                   :elt elt
```

```
                    :r (bst-remove obj (node-r bst) <)
                    :l (node-l bst)))))))

(defun percolate (bst)
  (cond ((null (node-l bst))
         (if (null (node-r bst))
             nil
             (rperc bst)))
        ((null (node-r bst)) (lperc bst))
        (t (if (zerop (random 2))
               (lperc bst)
               (rperc bst)))))

(defun rperc (bst)
  (make-node :elt (node-elt (node-r bst))
             :l (node-l bst)
             :r (percolate (node-r bst))))
```

圖 4.6 二元搜索樹：移除

勘誤：此版 `bst-remove` 的定義已被回報是壞掉的，請參考 這裡 [https://gist.github.com/2868263] 獲得修復版。

函數 `bst-remove` 接受一個物件，一棵二元搜索樹以及排序函數，並返回一棵與本來的二元搜索樹相同的樹，但不包含那個要移除的物件。和 `remove` 一樣，它不改動做爲第二個實參所傳入的二元搜索樹：

```
> (setf nums (bst-remove 2 nums #'<))
#<5>
> (bst-find 2 nums #'<)
NIL
```

此時 `nums` 的結構應該如圖 4.7 所示。（另一個可能性是 `1` 取代了 `2` 的位置。）

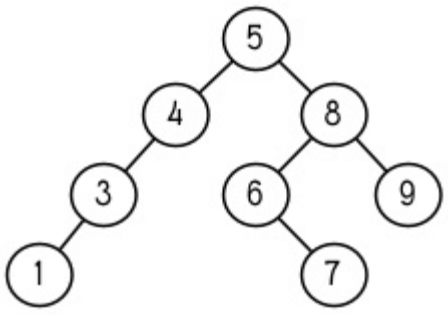

圖 4.7: 二元搜索樹

移除需要做更多工作，因爲從內部節點移除一個物件時，會留下一個空缺，需要由其中一個孩子來填補。這是 `percolate` 函數的用途。當它替換一個二元搜索樹的樹根

（topmost element）時，會找其中一個孩子來替換，並用此孩子的孩子來填補，如此這般一直遞迴下去。

爲了要保持樹的平衡，如果有兩個孩子時， `perlocate` 隨機擇一替換。表達式 `(random 2)` 會返回 `0` 或 `1` ，所以 `(zerop (random 2))` 會返回真或假。

```
(defun bst-traverse (fn bst)
  (when bst
    (bst-traverse fn (node-l bst))
    (funcall fn (node-elt bst))
    (bst-traverse fn (node-r bst))))
```

圖 4.8 二元搜索樹：遍歷

一旦我們把一個物件集合插入至二元搜索樹時，中序遍歷會將它們由小至大排序。這是圖 4.8 中， `bst-traverse` 函數的用途：

```
> (bst-traverse #'princ nums)
13456789
NIL
```

（函數 `princ` 僅顯示單一物件）

本節所給出的程式，提供了一個二元搜索樹實現的腳手架。你可能想根據應用需求，來充實這個腳手架。舉例來說，這裡所給出的程式每個節點只有一個 `elt` 欄位；在許多應用裡，有兩個欄位會更有意義， `key` 與 `value` 。本章的這個版本把二元搜索樹視爲集合看待，從這個角度看，重複的插入是被忽略的。但是程式可以很簡單地改動，來處理重複的元素。

二元搜索樹不僅是維護一個已排序物件的集合的方法。他們是否是最好的方法，取決於你的應用。一般來說，二元搜索樹最適合用在插入與刪除是均勻分佈的情況。有一件二元搜索樹不擅長的事，就是用來維護優先佇列（priority queues）。在一個優先佇列裡，插入也許是均勻分佈的，但移除總是在一個另一端。這會導致一個二元搜索樹變得不平衡，而我們期望的複雜度是 `O(log(n))` 插入與移除操作，將會變成 `O(n)` 。如果用二元搜索樹來表示一個優先佇列，也可以使用一般的列表，因爲二元搜索樹最終會作用的像是個列表。

4.8 雜湊表 (Hash Table)

第三章示範過列表可以用來表示集合（sets）與映射（mappings）。但當列表的長度大幅上升時（或是 10 個元素），使用雜湊表的速度比較快。你通過呼叫 `make-hash-table` 來構造一個雜湊表，它不需要傳入參數：

```
> (setf ht (make-hash-table))
#<Hash-Table BF0A96>
```

和函數一樣，雜湊表總是用 `#<...>` 的形式來顯示。

一個雜湊表，與一個關聯列表類似，是一種表達對應關係的方式。要取出與給定鍵值有關的數值，我們呼叫 `gethash` 並傳入一個鍵值與雜湊表。預設情況下，如果沒有與這個鍵值相關的數值， `gethash` 會返回 `nil` 。

```
> (gethash 'color ht)
NIL
NIL
```

在這裡我們首次看到 Common Lisp 最突出的特色之一：一個表達式竟然可以返回多個數值。函數 `gethash` 返回兩個數值。第一個值是與鍵值有關的數值，第二個值說明了雜湊表是否含有任何用此鍵值來儲存的數值。由於第二個值是 `nil` ，我們知道第一個 `nil` 是預設的返回值，而不是因爲 `nil` 是與 `color` 有關的數值。

大部分的實現會在頂層顯示一個函數呼叫的所有返回值，但僅期待一個返回值的程式，只會收到第一個返回值。 5.5 節會說明，程式如何接收多個返回值。

要把數值與鍵值作關聯，使用 `gethash` 搭配 `setf` ：

```
> (setf (gethash 'color ht) 'red)
RED
```

現在如果我們再次呼叫 `gethash` ，我們會得到我們剛插入的值：

```
> (gethash 'color ht)
RED
T
```

第二個返回值證明，我們取得了一個真正儲存的物件，而不是預設值。

存在雜湊表的物件或鍵值可以是任何型別。舉例來說，如果我們要保留函數的某種訊息，我們可以使用雜湊表，用函數作爲鍵值，字串作爲詞條（entry）：

```
> (setf bugs (make-hash-table))
#<Hash-Table BF4C36>
> (push "Doesn't take keyword arguments."
        (gethash #'our-member bugs))
("Doesn't take keyword arguments.")
```

由於 `gethash` 預設返回 `nil` ，而 `push` 是 `setf` 的縮寫，可以輕鬆的給雜湊表新添一個詞

條。（有困擾的 our-member 定義在 16 頁。）

可以用雜湊表來取代用列表表示集合。當集合變大時，雜湊表的查詢與移除會來得比較快。要新增一個成員到用雜湊表所表示的集合，把 gethash 用 setf 設成 t：

```
> (setf fruit (make-hash-table))
#<Hash-Table BFDE76>
> (setf (gethash 'apricot fruit) t)
T
```

然後要測試是否爲成員，你只要呼叫：

```
> (gethash 'apricot fruit)
T
T
```

由於 gethash 預設返回真，一個新創的雜湊表，會很方便地是一個空集合。

要從集閤中移除一個物件，你可以呼叫 remhash，它從一個雜湊表中移除一個詞條：

```
> (remhash 'apricot fruit)
T
```

返回值說明了是否有詞條被移除；在這個情況裡，有。

雜湊表有一個迭代函數： maphash，它接受兩個實參，接受兩個參數的函以及雜湊表。該函數會被每個鍵值對呼叫，沒有特定的順序：

```
> (setf (gethash 'shape ht) 'spherical
        (gethash 'size ht) 'giant)
GIANT

> (maphash #'(lambda (k v)
               (format t "~A = ~A~%" k v))
           ht)
SHAPE = SPHERICAL
SIZE = GIANT
COLOR = RED
NIL
```

maphash 總是返回 nil，但你可以通過傳入一個會累積數值的函數，把雜湊表的詞條存在列表裡。

雜湊表可以容納任何數量的元素，但當雜湊表空間用完時，它們會被擴張。如果你想要確保一個雜湊表，從特定數量的元素空間大小開始時，可以給 make-hash-table 一個選擇性的 :size 關鍵字參數。做這件事情有兩個理由：因爲你知道雜湊表會變得很大，你

想要避免擴張它；或是因爲你知道雜湊表會是很小，你不想要浪費記憶體。 `:size` 參數不僅指定了雜湊表的空間，也指定了元素的數量。平均來說，在被擴張前所能夠容納的數量。所以

```
(make-hash-table :size 5)
```

會返回一個預期存放五個元素的雜湊表。

和任何牽涉到查詢的結構一樣，雜湊表一定有某種比較鍵值的概念。預設是使用 `eql`，但你可以提供一個額外的關鍵字參數 `:test` 來告訴雜湊表要使用 `eq`， `equal`，還是 `equalp`：

```
> (setf writers (make-hash-table :test #'equal))
#<Hash-Table C005E6>
> (setf (gethash '(ralph waldo emerson) writers) t)
T
```

這是一個讓雜湊表變得有效率的取捨之一。有了列表，我們可以指定 `member` 爲判斷相等性的謂詞。有了雜湊表，我們可以預先決定，並在雜湊表構造時指定它。

大多數 Lisp 編程的取捨（或是生活，就此而論）都有這種特質。起初你想要事情進行得流暢，甚至賠上效率的代價。之後當程式變得沉重時，你犧牲了彈性來換取速度。

Chapter 4 總結 (Summary)

1. Common Lisp 支援至少 7 個維度的陣列。一維陣列稱爲向量。
2. 字串是字元的向量。字元本身就是物件。
3. 序列包括了向量與列表。許多序列函數都接受標準的關鍵字參數。
4. 處理字串的函數非常多，所以用 Lisp 來解析字串是小菜一碟。
5. 呼叫 `defstruct` 定義了一個帶有命名欄位的結構。它是一個程式能寫出程式的好例子。
6. 二元搜索樹見長於維護一個已排序的物件集合。
7. 雜湊表提供了一個更有效率的方式來表示集合與映射 (mappings)。

Chapter 4 習題 (Exercises)

1. 定義一個函數，接受一個平方陣列（square array，一個相同維度的陣列 `(n n)`），並將它順時針轉 90 度。

```
> (quarter-turn #2A((a b) (c d)))
#2A((C A) (D B))
```

你會需要用到 361 頁的 `array-dimensions` 。

2. 閱讀 368 頁的 `reduce` 說明，然後用它來定義：

```
(a) copy-list
(b) reverse（針對列表）
```

3. 定義一個結構來表示一棵樹，其中每個節點包含某些資料及三個小孩。定義：

```
(a) 一個函數來複製這樣的樹（複製完的節點與本來的節點是不相等（ `eql` ）的）
(b) 一個函數，接受一個物件與這樣的樹，如果物件與樹中各節點的其中一個欄位相等時，返回真。
```

4. 定義一個函數，接受一棵二元搜索樹，並返回由此樹元素所組成的，一個由大至小排序的列表。
5. 定義 `bst-adjoin` 。這個函數應與 `bst-insert` 接受相同的參數，但應該只在物件不等於任何樹中物件時將其插入。

勘誤：`bst-adjoin` 的功能與 `bst-insert` 一模一樣。

6. 任何雜湊表的內容可以由關聯列表（assoc-list）來描述，其中列表的元素是 `(k . v)` 的形式，對應到雜湊表中的每一個鍵值對。定義一個函數：

```
(a) 接受一個關聯列表，並返回一個對應的雜湊表。
(b) 接受一個雜湊表，並返回一個對應的關聯列表。
```

腳註

[1] 一個簡單陣列大小是不可調整、元素也不可替換的，並不含有填充指標（fill-pointer）。陣列預設是簡單的。簡單向量是個一維的簡單陣列，可以含有任何型別的元素。

[2] 在 ANSI Common Lisp 裡，你可以給一個 `:print-object` 的關鍵字參數來取代，它只需要兩個實參。也有一個宏叫做 `print-unreadable-object` ，能用則用，可以用 `#<...>` 的語法來顯示物件。

第五章：控制流

2.2 節介紹過 Common Lisp 的求值規則，現在你應該很熟悉了。本章的運算子都有一個共同點，就是它們都違反了求值規則。這些運算子讓你決定在程式當中何時要求值。如果普通的函數呼叫是 Lisp 程式的樹葉的話，那這些運算子就是連結樹葉的樹枝。

5.1 區塊(Blocks)

Common Lisp 有三個構造區塊（block）的基本運算子： `progn` 、 `block` 以及 `tagbody` 。我們已經看過 `progn` 了。在 `progn` 主體中的表達式會依序求值，並返回最後一個表達式的值：

```
> (progn
    (format t "a")
    (format t "b")
    (+ 11 12))
ab
23
```

由於只返回最後一個表達式的值，代表著使用 `progn` （或任何區塊）涵蓋了副作用。

一個 `block` 像是帶有名字及緊急出口的 `progn` 。第一個實參應爲符號。這成爲了區塊的名字。在主體中的任何地方，可以停止求值，並通過使用 `return-from` 指定區塊的名字，來立即返回數值：

```
> (block head
    (format t "Here we go.")
    (return-from head 'idea)
    (format t "We'll never see this."))
Here we go.
IDEA
```

呼叫 `return-from` 允許你的程式，從程式的任何地方，突然但優雅地退出。第二個傳給 `return-from` 的實參，用來作爲以第一個實參爲名的區塊的返回值。在 `return-from` 之後的表達式不會被求值。

也有一個 `return` 宏，它把傳入的參數當做封閉區塊 `nil` 的返回值：

```
> (block nil
    (return 27))
27
```

許多接受一個表達式主體的 Common Lisp 運算子，皆隱含在一個叫做 nil 的區塊裡。比如，所有由 do 構造的迭代函數：

```
> (dolist (x '(a b c d e))
    (format t "~A " x)
    (if (eql x 'c)
        (return 'done)))
A B C
DONE
```

使用 defun 定義的函數主體，都隱含在一個與函數同名的區塊，所以你可以：

```
(defun foo ()
  (return-from foo 27))
```

在一個顯式或隱式的 block 外，不論是 return-from 或 return 都不會工作。

使用 return-from，我們可以寫出一個更好的 read-integer 版本：

```
(defun read-integer (str)
  (let ((accum 0))
    (dotimes (pos (length str))
      (let ((i (digit-char-p (char str pos))))
        (if i
            (setf accum (+ (* accum 10) i))
            (return-from read-integer nil))))
    accum))
```

68 頁的版本在構造整數之前，需檢查所有的字元。現在兩個步驟可以結合，因爲如果遇到非數字的字元時，我們可以捨棄計算結果。出現在主體的原子（atom）被解讀爲標籤（labels)；把這樣的標籤傳給 go ，會把控制權交給標籤後的表達式。以下是一個非常醜的程式片段，用來印出一至十的數字：

```
> (tagbody
    (setf x 0)
    top
      (setf x (+ x 1))
      (format t "~A " x)
      (if (< x 10) (go top)))
1 2 3 4 5 6 7 8 9 10
NIL
```

這個運算子主要用來實現其它的運算子，不是一般會用到的運算子。大多數迭代運算子都隱含在一個 tagbody，所以是可能可以在主體裡（雖然很少想要）使用標籤及 go 。

如何決定要使用哪一種區塊建構子呢（block construct）？幾乎任何時候，你會使用

progn 。如果你想要突然退出的話，使用 block 來取代。多數程式設計師永遠不會顯式地使用 tagbody 。

5.2 語境（Context）

另一個我們用來區分表達式的運算子是 let 。它接受一個程式碼主體，但允許我們在主體內設置新變數：

```
> (let ((x 7)
        (y 2))
    (format t "Number")
    (+ x y))
Number
9
```

一個像是 let 的運算子，創造出一個新的詞法語境（lexical context）。在這個語境裡有兩個新變數，然而在外部語境的變數也因此變得不可視了。

概念上說，一個 let 表達式等同於函數呼叫。在 2.14 節證明過，函數可以用名字來引用，也可以通過使用一個 lambda 表達式從字面上來引用。由於 lambda 表達式是函數的名字，我們可以像使用函數名那樣，把 lambda 表達式作爲函數呼叫的第一個實參：

```
> ((lambda (x) (+ x 1)) 3)
4
```

前述的 let 表達式，實際上等同於：

```
((lambda (x y)
   (format t "Number")
   (+ x y))
 7
 2)
```

如果有關於 let 的任何問題，應該是如何把責任交給 lambda ，因爲進入一個 let 等同於執行一個函數呼叫。

這個模型清楚的告訴我們，由 let 創造的變數的值，不能依賴其它由同一個 let 所創造的變數。舉例來說，如果我們試著：

```
(let ((x 2)
      (y (+ x 1)))
  (+ x y))
```

在 (+ x 1) 中的 x 不是前一行所設置的值，因爲整個表達式等同於：

```
((lambda (x y) (+ x y)) 2
                        (+ x 1))
```

這裡明顯看到 `(+ x 1)` 作爲實參傳給函數，不能引用函數內的形參 `x` 。

所以如果你真的想要新變數的值，依賴同一個表達式所設立的另一個變數？在這個情況下，使用一個變形版本 `let*` ：

```
> (let* ((x 1)
         (y (+ x 1)))
    (+ x y))
3
```

一個 `let*` 功能上等同於一系列巢狀的 `let` 。這個特別的例子等同於：

```
(let ((x 1))
  (let ((y (+ x 1)))
    (+ x y)))
```

`let` 與 `let*` 將變數初始值都設爲 `nil` 。 `nil` 爲初始值的變數，不需要依附在列表內:

```
> (let (x y)
    (list x y))
(NIL NIL)
```

`destructuring-bind` 宏是通用化的 `let` 。與其接受單一變數，一個模式 (pattern) —— 一個或多個變數所構成的樹 —— 並將它們與某個實際的樹所對應的部份做綁定。舉例來說：

```
> (destructuring-bind (w (x y) . z) '(a (b c) d e)
    (list w x y z))
(A B C (D E))
```

若給定的樹（第二個實參）沒有與模式匹配（第一個參數）時，會產生錯誤。

5.3 條件 (Conditionals)

最簡單的條件式是 `if` ；其餘的條件式都是基於 `if` 所構造的。第二簡單的條件式是 `when` ，它接受一個測試表達式（test expression）與一個程式碼主體。若測試表達式求值返回真時，則對主體求值。所以

```
(when (oddp that)
  (format t "Hmm, that's odd.")
  (+ that 1))
```

等同於

```
(if (oddp that)
    (progn
      (format t "Hmm, that's odd.")
      (+ that 1)))
```

when 的相反是 unless ；它接受相同的實參，但僅在測試表達式返回假時，才對主體求值。

所有條件式的母體 (從正反兩面看) 是 cond ， cond 有兩個新的優點：允許多重條件判斷，與每個條件相關的程式碼隱含在 progn 裡。 cond 預期在我們需要使用巢狀 if 的情況下使用。 舉例來說，這個僞 member 函數

```
(defun our-member (obj lst)
  (if (atom lst)
      nil
      (if (eql (car lst) obj)
          lst
          (our-member obj (cdr lst)))))
```

也可以定義成：

```
(defun our-member (obj lst)
  (cond ((atom lst) nil)
        ((eql (car lst) obj) lst)
        (t (our-member obj (cdr lst)))))
```

事實上，Common Lisp 實現大概會把 cond 翻譯成 if 的形式。

總得來說呢， cond 接受零個或多個實參。每一個實參必須是一個具有條件式，伴隨著零個或多個表達式的列表。當 cond 表達式被求值時，測試條件式依序求值，直到某個測試條件式返回真才停止。當返回真時，與其相關聯的表達式會被依序求值，而最後一個返回的數值，會作爲 cond 的返回值。如果符合的條件式之後沒有表達式的話：

```
> (cond (99))
99
```

則會返回條件式的值。

由於 cond 子句的 t 條件永遠成立，通常我們把它放在最後，作爲預設的條件式。如果沒有子句符合時，則 cond 返回 nil ，但利用 nil 作爲返回值是一種很差的風格 (這種問題可能發生的例子，請看 292 頁)。譯註: **Appendix A, unexpected nil** 小節。

當你想要把一個數值與一系列的常數比較時，有 case 可以用。我們可以使用 case 來定義一個函數，返回每個月份中的天數：

```
(defun month-length (mon)
  (case mon
    ((jan mar may jul aug oct dec) 31)
    ((apr jun sept nov) 30)
    (feb (if (leap-year) 29 28))
    (otherwise "unknown month")))
```

一個 case 表達式由一個實參開始，此實參會被拿來與每個子句的鍵值做比較。接著是零個或多個子句，每個子句由一個或一串鍵值開始，跟隨著零個或多個表達式。鍵值被視爲常數；它們不會被求值。第一個參數的值被拿來與子句中的鍵值做比較 (使用 eql)。如果匹配時，子句剩餘的表達式會被求值，並將最後一個求值作爲 case 的返回值。

預設子句的鍵值可以是 t 或 otherwise 。如果沒有子句符合時，或是子句只包含鍵值時，

```
> (case 99 (99))
NIL
```

則 case 返回 nil 。

typecase 宏與 case 相似，除了每個子句中的鍵值應爲型別修飾符 (type specifiers)，以及第一個實參與鍵值比較的函數使用 typep 而不是 eql (一個 typecase 的例子在 107 頁)。 譯註: **6.5** 小節。

5.4 迭代 (Iteration)

最基本的迭代運算子是 do ，在 2.13 小節介紹過。由於 do 包含了隱式的 block 及 tagbody ，我們現在知道是可以在 do 主體內使用 return 、 return-from 以及 go 。

2.13 節提到 do 的第一個參數必須是說明變數規格的列表，列表可以是如下形式：

```
(variable  initial  update)
```

initial 與 update 形式是選擇性的。若 update 形式忽略時，每次迭代時不會更新變數。若 initial 形式也忽略時，變數會使用 nil 來初始化。

在 23 頁的例子中（譯註: 2.13 節），

```
(defun show-squares (start end)
  (do ((i start (+ i 1)))
```

```
      ((> i end) 'done)
    (format t "~A ~A~%" i (* i i))))
```

update 形式引用到由 do 所創造的變數。一般都是這麼用。如果一個 do 的 update 形式，沒有至少引用到一個 do 創建的變數時，反而很奇怪。

當同時更新超過一個變數時，問題來了，如果一個 update 形式，引用到一個擁有自己的 update 形式的變數時，它會被更新呢？或是獲得前一次迭代的值？使用 do 的話，它獲得後者的值：

```
> (let ((x 'a))
    (do ((x 1 (+ x 1))
         (y x x))
        ((> x 5))
      (format t "(~A ~A)  " x y)))
(1 A)  (2 1)  (3 2)  (4 3)  (5 4)
NIL
```

每一次迭代時， x 獲得先前的值，加上一； y 也獲得 x 的前一次數值。

但也有一個 do* ，它有著和 let 與 let* 一樣的關係。任何 initial 或 update 形式可以參照到前一個子句的變數，並會獲得當下的值：

```
> (do* ((x 1 (+ x 1))
        (y x x))
       ((> x 5))
  (format t "(~A ~A) " x y))
(1 1) (2 2) (3 3) (4 4) (5 5)
NIL
```

除了 do 與 do* 之外，也有幾個特別用途的迭代運算子。要迭代一個列表的元素，我們可以使用 dolist :

```
> (dolist (x '(a b c d) 'done)
    (format t "~A " x))
A B C D
DONE
```

當迭代結束時，初始列表內的第三個表達式 (譯註: done) ， 會被求值並作爲 dolist 的返回值。預設是 nil 。

有著同樣的精神的是 dotimes ， 給定某個 n ， 將會從整數 0 ， 迭代至 n-1 :

```
(dotimes (x 5 x)
  (format t "~A " x))
0 1 2 3 4
```

```
5
```

dolist 與 dotimes 初始列表的第三個表達式皆可省略，省略時為 nil 。注意該表達式可引用到迭代過程中的變數。

（譯註：第三個表達式即上例之 x ，可以省略，省略時 dotimes 表達式的回傳值為 nil ）

Note: do 的重點 (THE POINT OF do)

在 “The Evolution of Lisp” 裡，Steele 與 Garbriel 陳述了 do 的重點， 表達的實在太好了，值得整個在這裡引用過來：

撇開爭論語法不談，有件事要說明的是，在任何一個編程語言中，一個迴圈若一次只能更新一個變數是毫無用處的。 幾乎在任何情況下，會有一個變數用來產生下個值，而另一個變數用來累積結果。如果迴圈語法只能產生變數， 那麼累積結果就得藉由賦值語句來“手動”實現...或有其他的副作用。具有多變數的 do 迴圈，體現了產生與累積的本質對稱性，允許可以無副作用地表達迭代過程：

```
(defun factorial (n)
  (do ((j n (- j 1))
       (f 1 (* j f)))
    ((= j 0) f)))
```

當然在 step 形式裡實現所有的實際工作，一個沒有主體的 do 迴圈形式是較不尋常的。

函數 mapc 和 mapcar 很像，但不會 cons 一個新列表作爲返回值，所以使用的唯一理由是爲了副作用。它們比 dolist 來得靈活，因爲可以同時遍歷多個列表：

```
> (mapc #'(lambda (x y)
            (format t "~A ~A  " x y))
      '(hip flip slip)
      '(hop flop slop))
HIP HOP  FLIP FLOP  SLIP SLOP
(HIP FLIP SLIP)
```

總是回傳 mapc 的第二個參數。

5.5 多值 (Multiple Values)

曾有人這麼說，爲了要強調函數式編程的重要性，每個 Lisp 表達式都返回一個值。現在事情不是這麼簡單了；在 Common Lisp 裡，一個表達式可以返回零個或多個數值。最

多可以返回幾個值取決於各家實現，但至少可以返回 19 個值。

多值允許一個函數返回多件事情的計算結果，而不用構造一個特定的結構。舉例來說，內建的 `get-decoded-time` 返回 9 個數值來表示現在的時間：秒，分，時，日期，月，年，天，以及另外兩個數值。

多值也使得查詢函數可以分辨出 `nil` 與查詢失敗的情況。這也是爲什麼 `gethash` 返回兩個值。因爲它使用第二個數值來指出成功還是失敗，我們可以在雜湊表裡儲存 `nil` ，就像我們可以儲存別的數值那樣。

`values` 函數返回多個數值。它一個不少地返回你作爲數值所傳入的實參：

```
> (values 'a nil (+ 2 4))
A
NIL
6
```

如果一個 `values` 表達式，是函數主體最後求值的表達式，它所返回的數值變成函數的返回值。多值可以原封不地通過任何數量的返回來傳遞：

```
> ((lambda () ((lambda () (values 1 2)))))
1
2
```

然而若只預期一個返回值時，第一個之外的值會被捨棄：

```
> (let ((x (values 1 2)))
    x)
1
```

通過不帶實參使用 `values` ，是可能不返回值的。在這個情況下，預期一個返回值的話，會獲得 `nil` ：

```
> (values)
> (let ((x (values)))
    x)
NIL
```

要接收多個數值，我們使用 `multiple-value-bind` ：

```
> (multiple-value-bind (x y z) (values 1 2 3)
    (list x y z))
(1 2 3)

> (multiple-value-bind (x y z) (values 1 2)
```

```
    (list x y z))
(1 2 NIL)
```

如果變數的數量大於數值的數量，剩餘的變數會是 `nil` 。如果數值的數量大於變數的數量，多餘的值會被捨棄。所以只想印出時間我們可以這麼寫:

```
> (multiple-value-bind (s m h) (get-decoded-time)
    (format t "~A:~A:~A" h m s))
"4:32:13"
```

你可以藉由 `multiple-value-call` 將多值作爲實參傳給第二個函數:

```
> (multiple-value-call #'+ (values 1 2 3))
6
```

還有一個函數是 `multiple-value-list`:

```
> (multiple-value-list (values 'a 'b 'c))
(A B C)
```

看起來像是使用 `#'list` 作爲第一個參數的來呼叫 `multiple-value-call` 。

5.6 中止 (Aborts)

你可以使用 `return` 在任何時候離開一個 `block` 。有時候我們想要做更極端的事，在數個函數呼叫裡將控制權轉移回來。要達成這件事，我們使用 `catch` 與 `throw` 。一個 `catch` 表達式接受一個標籤（tag），標籤可以是任何型別的物件，伴隨著一個表達式主體:

```
(defun super ()
  (catch 'abort
    (sub)
    (format t "We'll never see this.")))

(defun sub ()
  (throw 'abort 99))
```

表達式依序求值，就像它們是在 `progn` 裡一樣。在這段程式裡的任何地方，一個帶有特定標籤的 `throw` 會導致 `catch` 表達式直接返回:

```
> (super)
99
```

一個帶有給定標籤的 `throw` ， 爲了要到達匹配標籤的 `catch` ， 會將控制權轉移 (因此殺

掉進程)給任何有標籤的 `catch` 。如果沒有一個 `catch` 符合欲匹配的標籤時， `throw` 會產生一個錯誤。

呼叫 `error` 同時中斷了執行，本來會將控制權轉移到呼叫樹（calling tree）的更高點，取而代之的是，它將控制權轉移給 Lisp 錯誤處理器（error handler）。通常會導致呼叫一個中斷迴圈（break loop）。以下是一個假定的 Common Lisp 實現可能會發生的事情：

```
> (progn
    (error "Oops!")
    (format t "After the error."))
Error: Oops!
       Options: :abort, :backtrace
>>
```

譯註：2 個 >> 顯示進入中斷迴圈了。

關於錯誤與狀態的更多訊息，參見 14.6 小節以及附錄 A。

有時候你想要防止程式被 `throw` 與 `error` 打斷。藉由使用 `unwind-protect` ，可以確保像是前述的中斷，不會讓你的程式停在不一致的狀態。一個 `unwind-protect` 接受任何數量的實參，並返回第一個實參的值。然而即便是第一個實參的求值被打斷時，剩下的表達式仍會被求值：

```
> (setf x 1)
1
> (catch 'abort
    (unwind-protect
      (throw 'abort 99)
      (setf x 2)))
99
> x
2
```

在這裡，即便 `throw` 將控制權交回監測的 `catch` ， `unwind-protect` 確保控制權移交時，第二個表達式有被求值。無論何時，一個確切的動作要伴隨著某種清理或重置時，`unwind-protect` 可能會派上用場。在 121 頁提到了一個例子。

5.7 範例： 日期運算 (Example: Date Arithmetic)

在某些應用裡，能夠做日期的加減是很有用的 —— 舉例來說，能夠算出從 1997 年 12 月 17 日，六十天之後是 1998 年 2 月 15 日。在這個小節裡，我們會編寫一個實用的工具來做日期運算。我們會將日期轉成整數，起始點設置在 2000 年 1 月 1 日。我們會使用內建的 + 與 - 函數來處理這些數字，而當我們轉換完畢時，再將結果轉回日期。

要將日期轉成數字，我們需要從日期的單位中，算出總天數有多少。舉例來說，2004 年 11 月 13 日的天數總和，是從起始點至 2004 年有多少天，加上從 2004 年到 2004 年 11 月有多少天，再加上 13 天。

有一個我們會需要的東西是，一張列出非潤年每月份有多少天的表格。我們可以使用 Lisp 來推敲出這個表格的內容。我們從列出每月份的長度開始：

```
> (setf mon '(31 28 31 30 31 30 31 31 30 31 30 31))
(31 28 31 30 31 30 31 31 30 31 30 31)
```

我們可以通過應用 + 函數至這個列表來測試總長度：

```
> (apply #'+ mon)
365
```

現在如果我們反轉這個列表並使用 `maplist` 來應用 + 函數至每下一個 `cdr` 上，我們可以獲得從每個月份開始所累積的天數：

```
> (setf nom (reverse mon))
(31 30 31 30 31 31 30 31 30 31 28 31)
> (setf sums (maplist #'(lambda (x)
                          (apply #'+ x))
                      nom))
(365 334 304 273 243 212 181 151 120 90 59 31)
```

這些數字體現了從二月一號開始已經過了 31 天，從三月一號開始已經過了 59 天……等等。

我們剛剛建立的這個列表，可以轉換成一個向量，見圖 5.1，轉換日期至整數的程式。

```
(defconstant month
  #(0 31 59 90 120 151 181 212 243 273 304 334 365))

(defconstant yzero 2000)

(defun leap? (y)
  (and (zerop (mod y 4))
       (or (zerop (mod y 400))
           (not (zerop (mod y 100))))))

(defun date->num (d m y)
  (+ (- d 1) (month-num m y) (year-num y)))

(defun month-num (m y)
  (+ (svref month (- m 1))
     (if (and (> m 2) (leap? y)) 1 0)))

(defun year-num (y)
```

```
  (let ((d 0))
    (if (>= y yzero)
        (dotimes (i (- y yzero) d)
          (incf d (year-days (+ yzero i))))
        (dotimes (i (- yzero y) (- d))
          (incf d (year-days (+ y i))))))))

(defun year-days (y) (if (leap? y) 366 365))
```

圖 **5.1** 日期運算：轉換日期至數字

典型 Lisp 程式的生命週期有四個階段：先寫好，然後讀入，接著編譯，最後執行。有件 Lisp 非常獨特的事情之一是，在這四個階段時， Lisp 一直都在那裡。可以在你的程式編譯 (參見 10.2 小節)或讀入時 (參見 14.3 小節) 來呼叫 Lisp。我們推導出 `month` 的過程示範了，如何在撰寫一個程式時使用 Lisp。

效率通常只跟第四個階段有關係，運行期（run-time）。在前三個階段，你可以隨意的使用列表擁有的威力與靈活性，而不需要擔心效率。

若你使用圖 5.1 的程式來造一個時光機器（time machine），當你抵達時，人們大概會不同意你的日期。即使是相對近的現在，歐洲的日期也曾有過偏移，因爲人們會獲得更精準的每年有多長的概念。在說英語的國家，最後一次的不連續性出現在 1752 年，日期從 9 月 2 日跳到 9 月 14 日。

每年有幾天取決於該年是否是潤年。如果該年可以被四整除，我們說該年是潤年，除非該年可以被 100 整除，則該年非潤年 —— 而要是它可以被 400 整除，則又是潤年。所以 1904 年是潤年，1900 年不是，而 1600 年是。

要決定某個數是否可以被另個數整除，我們使用函數 `mod` ，返回相除後的餘數：

```
> (mod 23 5)
3
> (mod 25 5)
0
```

如果第一個實參除以第二個實參的餘數爲 0，則第一個實參是可以被第二個實參整除的。函數 `leap?` 使用了這個方法，來決定它的實參是否是一個潤年：

```
> (mapcar #'leap? '(1904 1900 1600))
(T NIL T)
```

我們用來轉換日期至整數的函數是 `date->num` 。它返回日期中每個單位的天數總和。要找到從某月份開始的天數和，我們呼叫 `month-num` ，它在 `month` 中查詢天數，如果是在潤年的二月之後，則加一。

要找到從某年開始的天數和， date->num 呼叫 year-num ，它返回某年一月一日相對於起始點（2000.01.01）所代表的天數。這個函數的工作方式是從傳入的實參 y 年開始，朝著起始年（2000）往上或往下數。

```
(defun num->date (n)
  (multiple-value-bind (y left) (num-year n)
    (multiple-value-bind (m d) (num-month left y)
      (values d m y))))

(defun num-year (n)
  (if (< n 0)
      (do* ((y (- yzero 1) (- y 1))
            (d (- (year-days y)) (- d (year-days y))))
           ((<= d n) (values y (- n d))))
      (do* ((y yzero (+ y 1))
            (prev 0 d)
            (d (year-days y) (+ d (year-days y))))
           ((> d n) (values y (- n prev))))))

(defun num-month (n y)
  (if (leap? y)
      (cond ((= n 59) (values 2 29))
            ((> n 59) (nmon (- n 1)))
            (t        (nmon n)))
      (nmon n)))

(defun nmon (n)
  (let ((m (position n month :test #'<)))
    (values m (+ 1 (- n (svref month (- m 1)))))))

(defun date+ (d m y n)
  (num->date (+ (date->num d m y) n)))
```

圖 **5.2** 日期運算：轉換數字至日期

圖 5.2 展示了程式的下半部份。函數 num->date 將整數轉換回日期。它呼叫了 num-year 函數，以日期的格式返回年，以及剩餘的天數。再將剩餘的天數傳給 num-month ，分解出月與日。

和 year-num 相同， num-year 從起始年往上或下數，一次數一年。並持續累積天數，直到它獲得一個絕對值大於或等於 n 的數。如果它往下數，那麼它可以返回當前迭代中的數值。不然它會超過年份，然後必須返回前次迭代的數值。這也是爲什麼要使用 prev ， prev 在每次迭代時會存入 days 前次迭代的數值。

函數 num-month 以及它的子程式（subroutine） nmon 的行爲像是相反地 month-num 。他們從常數向量 month 的數值到位置，然而 month-num 從位置到數值。

圖 5.2 的前兩個函數可以合而爲一。與其返回數值給另一個函數， num-year 可以直接

呼叫 num-month 。現在分成兩部分的程式，比較容易做交互測試，但是現在它可以工作了，下一步或許是把它合而爲一。

有了 date->num 與 num->date ，日期運算是很簡單的。我們在 date+ 裡使用它們，可以從特定的日期做加減。如果我們想透過 date+ 來知道 1997 年 12 月 17 日六十天之後的日期:

```
> (multiple-value-list (date+ 17 12 1997 60))
(15 2 1998)
```

我們得到，1998 年 2 月 15 日。

Chapter 5 總結 (Summary)

1. Common Lisp 有三個基本的區塊建構子： progn ； 允許返回的 block ； 以及允許 goto 的 tagbody 。很多內建的運算子隱含在區塊裡。
2. 進入一個新的詞法語境，概念上等同於函數呼叫。
3. Common Lisp 提供了適合不同情況的條件式。每個都可以使用 if 來定義。
4. 有數個相似迭代運算子的變種。
5. 表達式可以返回多個數值。
6. 計算過程可以被中斷以及保護，保護可使其免於中斷所造成的後果。

Chapter 5 練習 (Exercises)

1. 將下列表達式翻譯成沒有使用 let 與 let* ，並使同樣的表達式不被求值 2 次。

```
(a) (let ((x (car y)))
      (cons x x))
(b) (let* ((w (car x))
           (y (+ w z)))
      (cons w y))
```

2. 使用 cond 重寫 29 頁的 mystery 函數。（譯註: 第二章的練習第 5 題的 (b) 部分)
3. 定義一個返回其實參平方的函數，而當實參是一個正整數且小於等於 5 時，不要計算其平方。
4. 使用 case 與 svref 重寫 month-num (圖 5.1)。
5. 定義一個迭代與遞迴版本的函數，接受一個物件 x 與向量 v ，並返回一個列表，包含了向量 v 當中，所有直接在 x 之前的物件：

```
> (precedes #\a "abracadabra")
(#\c #\d #\r)
```

6. 定義一個迭代與遞迴版本的函數，接受一個物件與列表，並返回一個新的列表，在原本列表的物件之間加上傳入的物件：

```
> (intersperse '- '(a b c d))
(A - B - C - D)
```

7. 定義一個接受一系列數字的函數，並在若且唯若每一對（pair）數字的差爲一時，返回真，使用

```
(a) 遞迴
(b) do
(c) mapc 與 return
```

8. 定義一個單遞迴函數，返回兩個值，分別是向量的最大與最小值。
9. 圖 3.12 的程式在找到一個完整的路徑時，仍持續遍歷佇列。在搜索範圍大時，這可能會產生問題。

```
(a) 使用 catch 與 throw 來變更程式，使其找到第一個完整路徑時，直接返回它。
(b) 重寫一個做到同樣事情的程式，但不使用 catch 與 throw。
```

第六章：函數

理解函數是理解 Lisp 的關鍵之一。概念上來說，函數是 Lisp 的核心所在。實際上呢，函數是你手邊最有用的工具之一。

6.1 全局函數 (Global Functions)

謂詞 `fboundp` 告訴我們，是否有個函數的名字與給定的符號綁定。如果一個符號是函數的名字，則 `symbol-name` 會返回它：

```
> (fboundp '+)
T
> (symbol-function '+)
#<Compiled-function + 17BA4E>
```

可通過 `symbol-function` 給函數配置某個名字：

```
(setf (symbol-function 'add2)
  #'(lambda (x) (+ x 2)))
```

新的全局函數可以這樣定義，用起來和 `defun` 所定義的函數一樣：

```
> (add2 1)
3
```

實際上 `defun` 做了稍微多的工作，將某些像是

```
(defun add2 (x) (+ x 2))
```

翻譯成上述的 `setf` 表達式。使用 `defun` 讓程式看起來更美觀，並或多或少幫助了編譯器，但嚴格來說，沒有 `defun` 也能寫程式。

通過把 `defun` 的第一個實參變成這種形式的列表 `(setf f)`，你定義了當 `setf` 第一個實參是 `f` 的函數呼叫時，所會發生的事情。下面這對函數把 `primo` 定義成 `car` 的同義詞：

```
(defun primo (lst) (car lst))

(defun (setf primo) (val lst)
  (setf (car lst) val))
```

在函數名是這種形式 `(setf f)` 的函數定義中，第一個實參代表新的數值，而剩餘的實

參代表了傳給 f 的參數。

現在任何 primo 的 setf，會是上面後者的函數呼叫：

```
> (let ((x (list 'a 'b 'c)))
    (setf (primo x) 480)
    x)
(480 b c)
```

不需要爲了定義 (setf primo) 而定義 primo，但這樣的定義通常是成對的。

由於字串是 Lisp 表達式，沒有理由它們不能出現在程式碼的主體。字串本身是沒有副作用的，除非它是最後一個表達式，否則不會造成任何差別。如果讓字串成爲 defun 定義的函數主體的第一個表達式，

```
(defun foo (x)
  "Implements an enhanced paradigm of diversity"
  x)
```

那麼這個字串會變成函數的文檔字串（documentation string）。要取得函數的文檔字串，可以通過呼叫 documentation 來取得：

```
> (documentation 'foo 'function)
"Implements an enhanced paradigm of diversity"
```

6.2 區域函數 (Local Functions)

通過 defun 或 symbol-function 搭配 setf 定義的函數是全局函數。你可以像存取全局變數那樣，在任何地方存取它們。定義區域函數也是有可能的，區域函數和區域變數一樣，只在某些上下文內可以存取。

區域函數可以使用 labels 來定義，它是一種像是給函數使用的 let 。它的第一個實參是一個新區域函數的定義列表，而不是一個變數規格說明的列表。列表中的元素爲如下形式：

```
(name parameters . body)
```

而 labels 表達式剩餘的部份，呼叫 name 就等於呼叫 (lambda parameters . body) 。

```
> (labels ((add10 (x) (+ x 10))
           (consa  (x) (cons 'a x)))
    (consa (add10 3)))
(A . 13)
```

label 與 let 的類比在一個方面上被打破了。由 labels 表達式所定義的區域函數，可以被其他任何在此定義的函數引用，包括自己。所以這樣定義一個遞迴的區域函數是可能的：

```
> (labels ((len (lst)
             (if (null lst)
                 0
                 (+ (len (cdr lst)) 1))))
    (len '(a b c)))
3
```

5.2 節展示了 let 表達式如何被理解成函數呼叫。 do 表達式同樣可以被解釋成呼叫遞迴函數。這樣形式的 do：

```
(do ((x a (b x))
     (y c (d y)))
    ((test x y) (z x y))
  (f x y))
```

等同於

```
(labels ((rec (x y)
           (cond ((test x y)
                  (z x y))
                 (t
                  (f x y)
                  (rec (b x) (d y))))))
  (rec a c))
```

這個模型可以用來解決，任何你仍然對於 do 行爲仍有疑惑的問題。

6.3 參數列表 (Parameter Lists)

2.1 節我們示範過，有了前序表達式， + 可以接受任何數量的參數。從那時開始，我們看過許多接受不定數量參數的函數。要寫出這樣的函數，我們需要使用一個叫做剩餘（*rest*）參數的東西。

如果我們在函數的形參列表裡的最後一個變數前，插入 &rest 符號，那麼當這個函數被呼叫時，這個變數會被設成一個帶有剩餘參數的列表。現在我們可以明白 funcall 是如何根據 apply 寫成的。它或許可以定義成：

```
(defun our-funcall (fn &rest args)
  (apply fn args))
```

我們也看過運算子中，有的參數可以被忽略，並可以預設設成特定的值。這樣的參數稱爲選擇性參數（optional parameters）。（相比之下，普通的參數有時稱爲必要參數「required parameters」）如果符號 `&optional` 出現在一個函數的形參列表時，

```
(defun pilosoph (thing &optional property)
  (list thing 'is property))
```

那麼在 `&optional` 之後的參數都是選擇性的，預設爲 `nil`：

```
> (philosoph 'death)
(DEATH IS NIL)
```

我們可以明確指定預設值，通過將預設值附在列表裡給入。這版的 `philosoph`

```
(defun philosoph (thing &optional (property 'fun))
  (list thing 'is property))
```

有著更鼓舞人心的預設值：

```
> (philosoph 'death)
(DEATH IS FUN)
```

選擇性參數的預設值可以不是常數。可以是任何的 Lisp 表達式。若這個表達式不是常數，它會在每次需要用到預設值時被重新求值。

一個關鍵字參數（keyword parameter）是一種更靈活的選擇性參數。如果你把符號 `&key` 放在一個形參列表，那在 `&key` 之後的形參都是選擇性的。此外，當函數被呼叫時，這些參數會被識別出來，參數的位置在哪不重要，而是用符號標籤（譯註: : ）識別出來：

```
> (defun keylist (a &key x y z)
    (list a x y z))
KEYLIST

> (keylist 1 :y 2)
(1 NIL 2 NIL)

> (keylist 1 :y 3 :x 2)
(1 2 3 NIL)
```

和普通的選擇性參數一樣，關鍵字參數預設值爲 `nil`，但可以在形參列表中明確地指定預設值。

關鍵字與其相關的參數可以被剩餘參數收集起來，並傳遞給其他期望收到這些參數的函

數。舉例來說，我們可以這樣定義 adjoin：

```
(defun our-adjoin (obj lst &rest args)
  (if (apply #'member obj lst args)
      lst
      (cons obj lst)))
```

由於 adjoin 與 member 接受一樣的關鍵字，我們可以用剩餘參數收集它們，再傳給 member 函數。

5.2 節介紹過 destructuring-bind 宏。在通常情況下，每個模式（pattern）中作爲第一個參數的子樹，可以與函數的參數列表一樣複雜：

```
(destructuring-bind ((&key w x) &rest y) '((:w 3) a)
  (list w x y))
(3 NIL (A))
```

6.4 範例：實用函數 (Example: Utilities)

2.6 節提到過，Lisp 大部分是由 Lisp 函陣列成，這些函數與你可以自己定義的函數一樣。這是程式語言中一個有用的特色：你不需要改變你的想法來配合語言，因爲你可以改變語言來配合你的想法。如果你想要 Common Lisp 有某個特定的函數，自己寫一個，而這個函數會成爲語言的一部分，就跟內建的 + 或 eql 一樣。

有經驗的 Lisp 程式設計師，由上而下（top-down）也由下而上 (bottom-up)地工作。當他們朝著語言撰寫程式的同時，也打造了一個更適合他們程式的語言。通過這種方式，語言與程式結合的更好，也更好用。

寫來擴展 Lisp 的運算子稱爲實用函數（utilities）。當你寫了更多 Lisp 程式時，會發現你開發了一系列的程式，而在一個項目寫過許多的實用函數，下個項目裡也會派上用場。

專業的程式設計師常發現，手邊正在寫的程式，與過去所寫的程式有很大的關聯。這就是軟體重用讓人聽起來很吸引人的原因。但重用已經被聯想成物件導向程式設計。但軟體不需要是面向物件的才能重用 —— 這是很明顯的，我們看看程式語言（換言之，編譯器），是重用性最高的軟體。

要獲得可重用軟體的方法是，由下而上地寫程式，而程式不需要是面向物件的才能夠由下而上地寫出。實際上，函數式風格相比之下，更適合寫出重用軟體。想想看 sort 。在 Common Lisp 你幾乎不需要自己寫排序程式； sort 是如此的快與普遍，以致於它不值得我們煩惱。這才是可重用軟體。

```
(defun single? (lst)
  (and (consp lst) (null (cdr lst))))

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

(defun map-int (fn n)
  (let ((acc nil))
    (dotimes (i n)
      (push (funcall fn i) acc))
    (nreverse acc)))

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

(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)
              (setf wins obj
                    max  score))))
        (values wins max))))
```

圖 **6.1** 實用函數

你可以通過撰寫實用函數，在程式裡做到同樣的事情。圖 6.1 挑選了一組實用的函數。前兩個 `single?` 與 `append1` 函數，放在這的原因是要示範，即便是小程式也很有用。前一個函數 `single?`，當實參是只有一個元素的列表時，返回真。

```
> (single? '(a))
T
```

而後一個函數 `append1` 和 `cons` 很像，但在列表後面新增一個元素，而不是在前面:

```
> (append1 '(a b c) 'd)
(A B C D)
```

下個實用函數是 `map-int`，接受一個函數與整數 `n`，並返回將函數應用至整數 `0` 到 `n-1` 的結果的列表。

這在測試的時候非常好用（一個 Lisp 的優點之一是，互動環境讓你可以輕鬆地寫出測

試）。如果我們只想要一個 0 到 9 的列表，我們可以：

```
> (map-int #'identity 10)
(0 1 2 3 4 5 6 7 8 9)
```

然而要是我們想要一個具有 10 個隨機數的列表，每個數介於 0 至 99 之間（包含 99），我們可以忽略參數並只要:

```
> (map-int #'(lambda (x) (random 100))
           10)
(85 50 73 64 28 21 40 67 5 32)
```

map-int 的定義說明了 Lisp 構造列表的標準做法（idiom）之一。我們創建一個累積器 acc，初始化是 nil，並將之後的物件累積起來。當累積完畢時，反轉累積器。 [1]

我們在 filter 中看到同樣的做法。 filter 接受一個函數與一個列表，將函數應用至列表元素上時，返回所有非 nil 元素:

```
> (filter #'(lambda (x)
              (and (evenp x) (+ x 10)))
          '(1 2 3 4 5 6 7))
(12 14 16)
```

另一種思考 filter 的方式是用通用版本的 remove-if 。

圖 6.1 的最後一個函數， most ，根據某個評分函數（scoring function），返回列表中最高分的元素。它返回兩個值，獲勝的元素以及它的分數:

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

如果平手的話，返回先馳得點的元素。

注意圖 6.1 的最後三個函數，它們全接受函數作爲參數。 Lisp 使得將函數作爲參數傳遞變得便捷，而這也是爲什麼，Lisp 適合由下而上程式設計的原因之一。成功的實用函數必須是通用的，當你可以將細節作爲函數參數傳遞時，要將通用的部份抽象起來就變得容易許多。

本節給出的函數是通用的實用函數。可以用在任何種類的程式。但也可以替特定種類的程式撰寫實用函數。確實，當我們談到宏時，你可以凌駕於 Lisp 之上，寫出自己的特定語言，如果你想這麼做的話。如果你想要寫可重用軟體，看起來這是最靠譜的方式。

6.5 閉包 (Closures)

函數可以如表達式的值，或是其它物件那樣被返回。以下是接受一個實參，並依其型別返回特定的結合函數：

```
(defun combiner (x)
  (typecase x
    (number #'+)
    (list #'append)
    (t #'list)))
```

在這之上，我們可以創建一個通用的結合函數:

```
(defun combine (&rest args)
  (apply (combiner (car args))
         args))
```

它接受任何型別的參數，並以適合它們型別的方式結合。（爲了簡化這個例子，我們假定所有的實參，都有著一樣的型別。）

```
> (combine 2 3)
5
> (combine '(a b) '(c d))
(A B C D)
```

2.10 小節提過詞法變數（lexical variables）只在被定義的上下文內有效。伴隨這個限制而來的是，只要那個上下文還有在使用，它們就保證會是有效的。

如果函數在詞法變數的作用域裡被定義時，函數仍可引用到那個變數，即便函數被作爲一個值返回了，返回至詞法變數被創建的上下文之外。下面我們創建了一個把實參加上 3 的函數：

```
> (setf fn (let ((i 3))
             #'(lambda (x) (+ x i))))
#<Interpreted-Function C0A51E>
> (funcall fn 2)
5
```

當函數引用到外部定義的變數時，這外部定義的變數稱爲自由變數（free variable）。函數引用到自由的詞法變數時，稱之爲閉包（closure）。 [2] 只要函數還存在，變數就必須一起存在。

閉包結合了函數與環境（environment）；無論何時，當一個函數引用到周圍詞法環境的某個東西時，閉包就被隱式地創建出來了。這悄悄地發生在像是下面這個函數，是一樣

的概念:

```
(defun add-to-list (num lst)
  (mapcar #'(lambda (x)
              (+ x num))
          lst))
```

這函數接受一個數字及列表，並返回一個列表，列表元素是元素與傳入數字的和。在 lambda 表達式裡的變數 `num` 是自由的，所以像是這樣的情況，我們傳遞了一個閉包給 `mapcar` 。

一個更顯著的例子會是函數在被呼叫時，每次都返回不同的閉包。下面這個函數返回一個加法器（adder）:

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

它接受一個數字，並返回一個將該數字與其參數相加的閉包（函數）。

```
> (setf add3 (make-adder 3))
#<Interpreted-Function C0EBF6>
> (funcall add3 2)
5
> (setf add27 (make-adder 27))
#<Interpreted-Function C0EE4E>
> (funcall add27 2)
29
```

我們可以產生共享變數的數個閉包。下面我們定義共享一個計數器的兩個函數:

```
(let ((counter 0))
  (defun reset ()
    (setf counter 0))
  (defun stamp ()
    (setf counter (+ counter 1))))
```

這樣的一對函數或許可以用來創建時間戳章（time-stamps）。每次我們呼叫 `stamp` 時，我們獲得一個比之前高的數字，而呼叫 `reset` 我們可以將計數器歸零:

```
> (list (stamp) (stamp) (reset) (stamp))
(1 2 0 1)
```

你可以使用全局計數器來做到同樣的事情，但這樣子使用計數器，可以保護計數器被非預期的引用。

Common Lisp 有一個內建的函數 `complement` 函數，接受一個謂詞，並返回謂詞的補數（complement）。比如：

```
> (mapcar (complement #'oddp)
          '(1 2 3 4 5 6))
(NIL T NIL T NIL T)
```

有了閉包以後，很容易就可以寫出這樣的函數：

```
(defun our-complement (f)
  #'(lambda (&rest args)
      (not (apply f args))))
```

如果你停下來好好想想，會發現這是個非凡的小例子；而這僅是冰山一角。閉包是 Lisp 特有的美妙事物之一。閉包開創了一種在別的語言當中，像是不可思議的程式設計方法。

6.6 範例：函數構造器 (Example: Function Builders)

Dylan 是 Common Lisp 與 Scheme 的混合物，有著 Pascal 一般的語法。它有著大量返回函數的函數：除了上一節我們所看過的 *complement* ，Dylan 包含: `compose` 、 `disjoin` 、 `conjoin` 、 `curry` 、 `rcurry` 以及 `always` 。圖 6.2 有這些函數的 Common Lisp 實現，而圖 6.3 示範了一些從定義延伸出的等價函數。

```
(defun compose (&rest fns)
  (destructuring-bind (fn1 . rest) (reverse fns)
    #'(lambda (&rest args)
        (reduce #'(lambda (v f) (funcall f v))
                rest
                :initial-value (apply fn1 args)))))

(defun disjoin (fn &rest fns)
  (if (null fns)
      fn
      (let ((disj (apply #'disjoin fns)))
        #'(lambda (&rest args)
            (or (apply fn args) (apply disj args))))))

(defun conjoin (fn &rest fns)
  (if (null fns)
      fn
      (let ((conj (apply #'conjoin fns)))
        #'(lambda (&rest args)
            (and (apply fn args) (apply conj args))))))

(defun curry (fn &rest args)
  #'(lambda (&rest args2)
```

```
      (apply fn (append args args2))))

(defun rcurry (fn &rest args)
  #'(lambda (&rest args2)
      (apply fn (append args2 args))))

(defun always (x) #'(lambda (&rest args) x))
```

圖 **6.2 Dylan** 函數建構器

首先， compose 接受一個或多個函數，並返回一個依序將其參數應用的新函數，即，

```
(compose #'a #'b #'c)
```

返回一個函數等同於

```
#'(lambda (&rest args) (a (b (apply #'c args))))
```

這代表著 compose 的最後一個實參，可以是任意長度，但其它函數只能接受一個實參。

下面我們建構了一個函數，先給取參數的平方根，取整後再放回列表裡，接著返回:

```
> (mapcar (compose #'list #'round #'sqrt)
          '(4 9 16 25))
((2) (3) (4) (5))
```

接下來的兩個函數， disjoin 及 conjoin 同接受一個或多個謂詞作爲參數： disjoin 當任一謂詞返回真時，返回真，而 conjoin 當所有謂詞返回真時，返回真。

```
> (mapcar (disjoin #'integerp #'symbolp)
          '(a "a" 2 3))
(T NIL T T)
```

```
> (mapcar (conjoin #'integerp #'symbolp)
          '(a "a" 2 3))
(NIL NIL NIL T)
```

若考慮將謂詞定義成集合， disjoin 返回傳入參數的聯集（union），而 conjoin 則是返回傳入參數的交集（intersection）。

```
      cddr = (compose #'cdr #'cdr)
      nth  = (compose #'car #'nthcdr)
      atom = (compose #'not #'consp)
           = (rcurry #'typep 'atom)
        <= = (disjoin #'< #'=)
     listp = (disjoin #'< #'=)
```

```
           = (rcurry #'typep 'list)
        1+ = (curry #'+ 1)
           = (rcurry #'+ 1)
        1- = (rcurry #'- 1)
    mapcan = (compose (curry #'apply #'nconc) #'mapcar
complement = (curry #'compose #'not)
```

圖 **6.3** 某些等價函數

函數 `curry` 與 `rcurry` （“right curry”）精神上與前一小節的 `make-adder` 相同。兩者皆接受一個函數及某些參數，並返回一個期望剩餘參數的新函數。下列任一個函數等同於 `(make-adder 3)`：

```
(curry #'+ 3)
(rcurry #'+ 3)
```

當函數的參數順序重要時，很明顯可以看出 `curry` 與 `rcurry` 的差別。如果我們 `curry #'-`，我們得到一個用其參數減去某特定數的函數，

```
(funcall (curry #'- 3) 2)
1
```

而當我們 `rcurry #'-` 時，我們得到一個用某特定數減去其參數的函數:

```
(funcall (rcurry #'- 3) 2)
-1
```

最後， `always` 函數是 Common Lisp 函數 `constantly` 。接受一個參數並原封不動返回此參數的函數。和 `identity` 一樣，在很多需要傳入函數參數的情況下很有用。

6.7 動態作用域 (Dynamic Scope)

2.11 小節解釋過區域與全局變數的差別。實際的差別是詞法作用域（lexical scope）的詞法變數（lexical variable），與動態作用域（dynamic scope）的特別變數（special variable）的區別。但這倆幾乎是沒有區別，因爲區域變數幾乎總是是詞法變數，而全局變數總是是特別變數。

在詞法作用域下，一個符號引用到上下文中符號名字出現的地方。區域變數預設有著詞法作用域。所以如果我們在一個環境裡定義一個函數，其中有一個變數叫做 `x`，

```
(let ((x 10))
  (defun foo ()
    x))
```

則無論 foo 被呼叫時有存在其它的 x，主體內的 x 都會引用到那個變數:

```
> (let ((x 20)) (foo))
10
```

而動態作用域，我們在環境中函數被呼叫的地方尋找變數。要使一個變數是動態作用域的，我們需要在任何它出現的上下文中宣告它是 special。如果我們這樣定義 foo：

```
(let ((x 10))
  (defun foo ()
    (declare (special x))
    x))
```

則函數內的 x 就不再引用到函數定義裡的那個詞法變數，但會引用到函數被呼叫時，當下所存在的任何特別變數 x：

```
> (let ((x 20))
    (declare (special x))
    (foo))
20
```

新的變數被創建出來之後， 一個 declare 呼叫可以在程式的任何地方出現。 special 宣告是獨一無二的，因爲它可以改變程式的行爲。 13 章將討論其它種類的宣告。所有其它的宣告，只是給編譯器的建議；或許可以使程式運行的更快，但不會改變程式的行爲。

通過在頂層呼叫 setf 來配置全局變數，是隱式地將變數宣告爲特殊變數:

```
> (setf x 30)
30
> (foo)
30
```

在一個檔案裡的程式碼，如果你不想依賴隱式的特殊宣告，可以使用 defparameter 取代，讓程式看起來更簡潔。

動態作用域什麼時候會派上用場呢？通常用來暫時給某個全局變數賦新值。舉例來說，有 11 個變數來控制物件印出的方式，包括了 *print-base* ，預設是 10 。如果你想要用 16 進制顯示數字，你可以重新綁定 *print-base*：

```
> (let ((*print-base* 16))
    (princ 32))
20
32
```

這裡顯示了兩件事情，由 princ 產生的輸出，以及它所返回的值。他們代表著同樣的數字，第一次在被印出時，用 16 進制顯示，而第二次，因爲在 let 表達式外部，所以是用十進制顯示，因爲 *print-base* 回到之前的數值， 10 。

6.8 編譯 (Compilation)

Common Lisp 函數可以獨立被編譯或挨個檔案編譯。如果你只是在頂層輸入一個 defun 表達式：

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

許多實現會創建一個直譯的函數（interpreted function）。你可以將函數傳給 compiled-function-p 來檢查一個函數是否有被編譯:

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

若你將 foo 函數名傳給 compile :

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

則這個函數會被編譯，而直譯的定義會被編譯出來的取代。編譯與直譯函數的行爲一樣，只不過對 compiled-function-p 來說不一樣。

你可以把列表作爲參數傳給 compile 。這種 compile 的用法在 161 頁 (譯註: 10.1 小節)。

有一種函數你不能作爲參數傳給 compile : 一個像是 stamp 或是 reset 這種，在頂層明確使用詞法上下文輸入的函數 (即 let) [3] 在一個檔案裡面定義這些函數，接著編譯然後載入檔案是可以的。直譯的程式會有這樣的限制是實作的原因，而不是因爲在詞法上下文裡明確定義函數有什麼問題。

通常要編譯 Lisp 程式不是挨個函數編譯，而是使用 compile-file 編譯整個檔案。這個函數接受一個檔案名，並創建一個原始碼的編譯版本 —— 通常會有同樣的名稱，但不同的擴展名。當編譯過的檔案被載入時， compiled-function-p 應給所有定義在檔案內的函數返回真。

當一個函數包含在另一個函數內時，包含它的函數會被編譯，而且內部的函數也會被編譯。所以 make-adder (108 頁)被編譯時，它會返回編譯的函數:

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

6.9 使用遞迴 (Using Recursion)

比起多數別的語言，遞迴在 Lisp 中扮演了一個重要的角色。這主要有三個原因：

1. 函數式程式設計。遞迴演算法有副作用的可能性較低。
2. 遞迴資料結構。 Lisp 隱式地使用了指標，使得遞迴地定義資料結構變簡單了。最常見的是用在列表：一個列表的遞迴定義，列表爲空表，或是一個 `cons` ，其中 `cdr` 也是個列表。
3. 優雅性。Lisp 程式設計師非常關心它們的程式是否美麗，而遞迴演算法通常比迭代演算法來得優雅。

學生們起初會覺得遞迴很難理解。但 3.9 節指出了，如果你想要知道是否正確，不需要去想遞迴函數所有的呼叫過程。

同樣的如果你想寫一個遞迴函數。如果你可以描述問題是怎麼遞迴解決的，通常很容易將解法轉成程式。要使用遞迴來解決一個問題，你需要做兩件事：

1. 你必須要示範如何解決問題的一般情況，通過將問題切分成有限小並更小的子問題。
2. 你必須要示範如何通過 —— 有限的步驟，來解決最小的問題 —— 基本用例。

如果這兩件事完成了，那問題就解決了。因爲遞迴每次都將問題變得更小，而一個有限的問題終究會被解決的，而最小的問題僅需幾個有限的步驟就能解決。

舉例來說，下面這個找到一個正規列表（proper list）長度的遞迴算法，我們每次遞迴時，都可以找到更小列表的長度：

1. 在一般情況下，一個正規列表的長度是它的 `cdr` 加一。
2. 基本用例，空列表長度爲 `0` 。

當這個描述翻譯成程式時，先處理基本用例；但公式化遞迴演算法時，我們通常從一般情況下手。

前述的演算法，明確地描述了一種找到正規列表長度的方法。當你定義一個遞迴函數時，你必須要確定你在分解問題時，問題實際上越變越小。取得一個正規列表的 `cdr` 會給出 `length` 更小的子問題，但取得環狀列表（circular list）的 `cdr` 不會。

這裡有兩個遞迴算法的範例。假定參數是有限的。注意第二個範例，我們每次遞迴時，將問題分成兩個更小的問題：

第一個例子， `member` 函數，我們說某物是列表的成員，需滿足：如果它是第一個元素的成員或是 `member` 的 `cdr` 的成員。但空列表沒有任何成員。

第二個例子， `copy-tree` 一個 `cons` 的 `copy-tree` ，是一個由 `cons` 的 `car` 的 `copy-tree` 與 `cdr` 的 `copy-tree` 所組成的。一個原子的 `copy-tree` 是它自己。

一旦你可以這樣描述算法，要寫出遞迴函數只差一步之遙。

某些算法通常是這樣表達最自然，而某些算法不是。你可能需要翻回前面，試試不使用遞迴來定義 `our-copy-tree` (41 頁，譯註: 3.8 小節)。另一方面來說，23 頁 (譯註: 2.13 節) 迭代版本的 `show-squares` 可能更容易比 24 頁的遞迴版本要容易理解。某些時候是很難看出哪個形式比較自然，直到你試著去寫出程式來。

如果你關心效率，有兩個你需要考慮的議題。第一，尾遞迴（tail-recursive），會在 13.2 節討論。一個好的編譯器，使用迴圈或是尾遞迴的速度，應該是沒有或是區別很小的。然而如果你需要使函數變成尾遞迴的形式時，或許直接用迭代會更好。

另一個需要銘記在心的議題是，最顯而易見的遞迴算法，不一定是最有效的。經典的例子是費氏函數。它是這樣遞迴地被定義的，

1. Fib(0) = Fib(1) = 1
2. Fib(n) = Fib(n-1)+Fib(n-2)

直接翻譯這個定義，

```
(defun fib (n)
  (if (<= n 1)
      1
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

這樣是效率極差的。一次又一次的重複計算。如果你要找 `(fib 10)` ，這個函數計算 `(fib 9)` 與 `(fib 8)` 。但要計算出 `(fib 9)` ，它需要再次計算 `(fib 8)` ，等等。

下面是一個算出同樣結果的迭代版本:

```
(defun fib (n)
  (do ((i n (- i 1))
       (f1 1 (+ f1 f2))
       (f2 1 f1))
      ((<= i 1) f1)))
```

迭代的版本不如遞迴版本來得直觀，但是效率遠遠高出許多。這樣的事情在實踐中常發生嗎？非常少 —— 這也是爲什麼所有的教科書都使用一樣的例子 —— 但這是需要注意的事。

Chapter 6 總結 (Summary)

1. 命名函數是一個存在符號的 `symbol-function` 部分的函數。 `defun` 宏隱藏了這樣的細節。它也允許你定義文檔字串（documentation string），並指定 `setf` 要怎麼處理函數呼叫。
2. 定義區域函數是有可能的，與定義區域變數有相似的精神。
3. 函數可以有選擇性參數（optional）、剩餘（rest）以及關鍵字（keyword）參數。
4. 實用函數是 Lisp 的擴展。他們是由下而上編程的小規模範例。
5. 只要有某物引用到詞法變數時，它們會一直存在。閉包是引用到自由變數的函數。你可以寫出返回閉包的函數。
6. Dylan 提供了構造函數的函數。很簡單就可以使用閉包，然後在 Common Lisp 中實現它們。
7. 特別變數（special variable）有動態作用域 (dynamic scope)。
8. Lisp 函數可以單獨編譯，或（更常見）編譯整個檔案。
9. 一個遞迴演算法通過將問題細分成更小、更小的子問題來解決問題。

Chapter 6 練習 (Exercises)

1. 定義一個 `tokens` 版本 (67 頁)，接受 `:test` 與 `:start` 參數，預設分別是 `#'constituent` 與 `0` 。(譯註: 67 頁在 4.5 小節)
2. 定義一個 `bin-search` (60 頁)的版本，接受 `:key` , `:test` , `start` 與 `end` 參數，有著一般的意義與預設值。(譯註: 60 頁在 4.1 小節)
3. 定義一個函數，接受任何數目的參數，並返回傳入的參數。
4. 修改 `most` 函數 (105 頁)，使其返回 2 個數值，一個列表中最高分的兩個元素。(譯註: 105 頁在 6.4 小節)
5. 用 `filter` (105 頁) 來定義 `remove-if` （沒有關鍵字）。(譯註: 105 頁在 6.4 小節)
6. 定義一個函數，接受一個參數、一個數字，並返回目前傳入參數中最大的那個。
7. 定義一個函數，接受一個參數、一個數字，若傳入參數比上個參數大時，返回真。函數第一次呼叫時應返回 `nil` 。
8. 假設 `expensive` 是一個接受一個參數的函數，一個介於 0 至 100 的整數（包含 100)，返回一個耗時的計算結果。定義一個函數 `frugal` 來返回同樣的答案，但僅在沒見過傳入參數時呼叫 `expensive` 。
9. 定義一個像是 `apply` 的函數，但在任何數字印出前，預設用 8 進制印出。

腳註

[1] 在這個情況下， `nreverse` (在 222 頁描述)和 `reverse` 做一樣的事情，但更有效率。

[2] “閉包”這個名字是早期的 Lisp 方言流傳而來。它是從閉包需要在動態作用域裡實現的方式衍生而來。

[3] 以前的 ANSI Common Lisp， `compile` 的第一個參數也不能是一個已經編譯好的函數。

第七章：輸入與輸出

Common Lisp 有著威力強大的 I/O 工具。針對輸入以及一些普遍讀取字元的函數，我們有 `read`，包含了一個完整的解析器 (parser)。針對輸出以及一些普遍寫出字元的函數，我們有 `format`，它自己幾乎就是一個語言。本章介紹了所有基本的概念。

Common Lisp 有兩種流 (streams)，字元流與二進制流。本章描述了字元流的操作；二進制流的操作涵蓋在 14.2 節。

7.1 流 (Streams)

流是用來表示字元來源或終點的 Lisp 物件。要從檔案讀取或寫入，你將檔案作爲流打開。但流與檔案是不一樣的。當你在頂層讀入或印出時，你也可以使用流。你甚至可以創建可以讀取或寫入字串的流。

輸入預設是從 `*standard-input*` 流讀取。輸出預設是在 `*standard-output*` 流。最初它們大概會在相同的地方：一個表示頂層的流。

我們已經看過 `read` 與 `format` 是如何在頂層讀取與印出。前者接受一個應是流的選擇性參數，預設是 `*standard-input*`。`format` 的第一個參數也可以是一個流，但當它是 `t` 時，輸出被送到 `*standard-output*`。所以我們目前爲止都只用到預設的流而已。我們可以在任何流上面做同樣的 I/O 操作。

路徑名（pathname）是一種指定一個檔案的可移植方式。路徑名包含了六個部分：host、device、directory、name、type 及 version。你可以通過呼叫 `make-pathname` 搭配一個或多個對應的關鍵字參數來產生一個路徑。在最簡單的情況下，你可以只指明名字，讓其他的部分留爲預設：

```
> (setf path (make-pathname :name "myfile"))
#P"myfile"
```

開啓一個檔案的基本函數是 `open`。它接受一個路徑名 [1] 以及大量的選擇性關鍵字參數，而若是開啓成功時，返回一個指向檔案的流。

你可以在創建流時，指定你想要怎麼使用它。 無論你是要寫入流、從流讀取或是兩者皆是， `direction` 參數都會偵測到。三個對應的數值是 `:input`, `:output`, `:io`。如果是用來輸出的流， `if-exists` 參數說明了如果檔案已經存在時該怎麼做；通常它應該是 `:supersede` (譯註: 取代)。所以要創建一個可以寫至 `"myfile"` 檔案的流，你可以：

```
> (setf str (open path :direction :output
                       :if-exists :supersede))
#<Stream C017E6>
```

流的列印表示法因實現而異 (implementation-dependent)。

現在我們可以把這個流作爲第一個參數傳給 `format`，它會在流印出，而不是頂層：

```
> (format str "Something~%")
NIL
```

如果我們在此時檢視這個檔案，輸出也許會、也許不會在那裡。某些實現會將輸出儲存成一塊 (chunks)再寫出。它也許不會出現，直到我們將流關閉：

```
> (close str)
NIL
```

當你使用完時，永遠記得關閉檔案；在你還沒關閉之前，內容是不保證會出現的。現在如果我們檢視檔案 “myfile”，應該有一行：

```
Something
```

如果我們只想從一個檔案讀取，我們可以開啓一個具有 `:direction :input` 的流：

```
> (setf str (open path :direction :input))
#<Stream C01C86>
```

我們可以對一個檔案使用任何輸入函數。7.2 節會更詳細的描述輸入。這裡作爲一個範例，我們將使用 `read-line` 從檔案來讀取一行文字：

```
> (read-line str)
"Something"
> (close str)
NIL
```

當你讀取完畢時，記得關閉檔案。

大部分時間我們不使用 `open` 與 `close` 來操作檔案的 I/O。`with-open-file` 宏通常更方便。它的第一個參數應該是一個列表，包含了變數名、伴隨著你想傳給 `open` 的參數。在這之後，它接受一個程式碼主體，它會被綁定至流的變數一起被求值，其中流是通過將剩餘的參數傳給 `open` 來創建的。之後這個流會被自動關閉。所以整個檔案寫入動作可以表示爲：

```
(with-open-file (str path :direction :output
```

```
                              :if-exists :supersede)
  (format str "Something~%"))
```

with-open-file 宏將 close 放在 unwind-protect 裡 (參見 92 頁，譯註: 5.6 節)，即使一個錯誤打斷了主體的求值，檔案是保證會被關閉的。

7.2 輸入 (Input)

兩個最受歡迎的輸入函數是 read-line 及 read 。前者讀入換行符 (newline)之前的所有字元，並用字串返回它們。它接受一個選擇性流參數 (optional stream argument)；若流忽略時，預設爲 *standard-input* ：

```
> (progn
    (format t "Please enter your name: ")
    (read-line))
Please enter your name: Rodrigo de Bivar
"Rodrigo de Bivar"
NIL
```

譯註：Rodrigo de Bivar 人稱熙德 (El cid)，十一世紀的西班牙民族英雄。

如果你想要原封不動的輸出，這是你該用的函數。(第二個返回值只在 read-line 在遇到換行符之前，用盡輸入時返回真。)

在一般情況下， read-line 接受四個選擇性參數: 一個流；一個參數用來決定遇到 end-of-file 時，是否產生錯誤；若前一個參數爲 nil 時，該返回什麼；第四個參數 (在 235 頁討論)通常可以省略。

所以要在頂層顯示一個檔案的內容，我們可以使用下面這個函數：

```
(defun pseudo-cat (file)
  (with-open-file (str file :direction :input)
    (do ((line (read-line str nil 'eof)
               (read-line str nil 'eof)))
        ((eql line 'eof))
      (format t "~A~%" line))))
```

如果我們想要把輸入解析爲 Lisp 物件，使用 read 。這個函數恰好讀取一個表達式，在表達式結束時停止讀取。所以可以讀取多於或少於一行。而當然它所讀取的內容必須是合法的 Lisp 語法。

如果我們在頂層使用 read ，它會讓我們在表達式裡面，想用幾個換行符就用幾個：

```
> (read)
```

```
(a
b
c)
(A B C)
```

換句話說，如果我們在一行裡面輸入許多表達式， `read` 會在第一個表達式之後，停止處理字元，留下剩餘的字元給之後讀取這個流的函數處理。所以如果我們在一行輸入多個表達式，來回應 `ask-number` (20 頁。譯註：2.10 小節)所印出提示符，會發生如下情形:

```
> (ask-number)
Please enter a number. a b
Please enter a number. Please enter a number. 43
43
```

兩個連續的提示符 (successive prompts)在第二行被印出。第一個 `read` 呼叫會返回 `a` ，而它不是一個數字，所以函數再次要求一個數字。但第一個 `read` 只讀取到 `a` 的結尾。所以下一個 `read` 呼叫返回 `b` ，導致了下一個提示符。

你或許想要避免使用 `read` 來直接處理使用者的輸入。前述的函數若使用 `read-line` 來獲得使用者輸入會比較好，然後對結果字串呼叫 `read-from-string` 。這個函數接受一個字串，並返回第一個讀取的表達式:

```
> (read-from-string "a b c")
A
2
```

它同時返回第二個值，一個指出停止讀取字串時的位置的數字。

在一般情況下， `read-from-string` 可以接受兩個選擇性參數與三個關鍵字參數。兩個選擇性參數是 `read` 的第三、第四個參數: 一個 end-of-file (這個情況是字串) 決定是否報錯，若不報錯該返回什麼。關鍵字參數 `:start` 及 `:end` 可以用來劃分從字串的哪裡開始讀。

所有的這些輸入函數是由基本函數 (primitive) `read-char` 所定義的，它讀取一個字元。它接受四個與 `read` 及 `read-line` 一樣的選擇性參數。Common Lisp 也定義一個函數叫做 `peek-char` ，跟 `read-char` 類似，但不會將字元從流中移除。

7.3 輸出 (Output)

三個最簡單的輸出函數是 `prin1` , `princ` 以及 `terpri` 。這三個函數的最後一個參數皆爲選擇性的流參數，預設是 `*standard-output*` 。

prin1 與 princ 的差別大致在於 prin1 給程式產生輸出，而 princ 給人類產生輸出。所以舉例來說， prin1 會印出字串左右的雙引號，而 princ 不會:

```
> (prin1 "Hello")
"Hello"
"Hello"
> (princ "Hello")
Hello
"Hello"
```

兩者皆返回它們的第一個參數 (譯註: 第二個值是返回值) —— 順道一提，是用 prin1 印出。 terpri 僅印出一新行。

有這些函數的背景知識在解釋更爲通用的 format 是很有用的。這個函數幾乎可以用在所有的輸出。他接受一個流 (或 t 或 nil)、一個格式化字串 (format string)以及零個或多個額外的參數。格式化字串可以包含特定的格式化指令 (format directives)，這些指令前面有波浪號 ~ 。某些格式化指令作爲字串的佔位符 (placeholder)使用。這些位置會被格式化字串之後，所給入參數的表示法所取代。

如果我們把 t 作爲第一個參數，輸出會被送至 *standard-output* 。如果我們給 nil ， format 會返回一個它會如何印出的字串。爲了保持簡短，我們會在所有的範例裡示範怎麼做。

由於每人的觀點不同， format 可以是令人驚訝的強大或是極爲可怕的複雜。有大量的格式化指令可用，而只有少部分會被大多數程式設計師使用。兩個最常用的格式化指令是 ~A 以及 ~% 。(你使用 ~a 或 ~A 都沒關係，但後者較常見，因爲它讓格式化指令看起來一目了然。) 一個 ~A 是一個值的佔位符，它會像是用 princ 印出一般。一個 ~% 代表著一個換行符 (newline)。

```
> (format nil "Dear ~A, ~% Our records indicate..."
                                        "Mr. Malatesta")
"Dear Mr. Malatesta,
   Our records indicate..."
```

這裡 format 返回了一個值，由一個含有換行符的字串組成。

~S 格式化指令像是 ~A ，但它使用 prin1 印出物件，而不是 princ 印出:

```
> (format t "~S  ~A" "z" "z")
"z" z
NIL
```

格式化指令可以接受參數。 ~F 用來印出向右對齊 (right-justified)的浮點數，可接受五個參數:

1. 要印出字元的總數。預設是數字的長度。
2. 小數之後要印幾位數。預設是全部。
3. 小數點要往右移幾位 (即等同於將數字乘 10)。預設是沒有。
4. 若數字太長無法滿足第一個參數時，所要印出的字元。如果沒有指定字元，一個過長的數字會儘可能使用它所需的空間被印出。
5. 數字開始印之前左邊的字元。預設是空白。

下面是一個有五個參數的罕見例子:

```
? (format nil "~10,2,0,'*,' F" 26.21875)
"      26.22"
```

這是原本的數字取至小數點第二位、(小數點向左移 0 位)、在 10 個字元的空間裡向右對齊，左邊補滿空白。注意作爲參數給入是寫成 `'*` 而不是 `#\*` 。由於數字塞得下 10 個字元，不需要使用第四個參數。

所有的這些參數都是選擇性的。要使用預設值你可以直接忽略對應的參數。如果我們想要做的是，印出一個小數點取至第二位的數字，我們可以說:

```
> (format nil "~,2,,,F" 26.21875)
"26.22"
```

你也可以忽略一系列的尾隨逗號 (trailing commas)，前面指令更常見的寫法會是:

```
> (format nil "~,2F" 26.21875)
"26.22"
```

警告：當 `format` 取整數時，它不保證會向上進位或向下舍入。就是說 `(format nil "~,1F" 1.25)` 可能會是 `"1.2"` 或 `"1.3"` 。所以如果你使用 `format` 來顯示資訊時，而使用者期望看到某種特定取整數方式的數字 (如: 金額數量)，你應該在印出之前先顯式地取好整數。

7.4 範例：字串代換 (Example: String Substitution)

作爲一個 I/O 的範例，本節示範如何寫一個簡單的程式來對文字檔案做字串替換。我們即將寫一個可以將一個檔案中，舊的字串 `old` 換成某個新的字串 `new` 的函數。最簡單的實現方式是將輸入檔案裡的每一個字元與 `old` 的第一個字元比較。如果沒有匹配，我們可以直接印出該字元至輸出。如果匹配了，我們可以將輸入的下一個字元與 `old` 的第二個字元比較，等等。如果輸入字元與 `old` 完全相等時，我們有一個成功的匹配，則我們印出 `new` 至檔案。

而要是 `old` 在匹配途中失敗了，會發生什麼事呢？舉例來說，假設我們要找的模式(pattern)是 `"abac"` ，而輸入檔案包含的是 `"ababac"` 。輸入會一直到第四個字元才發現不匹配，也就是在模式中的 `c` 以及輸入的 `b` 才發現。在此時我們可以將原本的 `a` 寫至輸出檔案，因爲我們已經知道這裡沒有匹配。但有些我們從輸入讀入的字元還是需要留著: 舉例來說，第三個 `a` ，確實是成功匹配的開始。所以在我們要實現這個算法之前，我們需要一個地方來儲存，我們已經從輸入讀入的字元，但之後仍然需要的字元。

一個暫時儲存輸入的佇列 (queue)稱作緩衝區 (buffer)。在這個情況裡，因爲我們知道我們不需要儲存超過一個預定的字元量，我們可以使用一個叫做環狀緩衝區 `ring buffer` 的資料結構。一個環狀緩衝區實際上是一個向量。是使用的方式使其成爲環狀: 我們將之後的元素所輸入進來的值儲存起來，而當我們到達向量結尾時，我們重頭開始。如果我們不需要儲存超過 `n` 個值，則我們只需要一個長度爲 `n` 或是大於 `n` 的向量，這樣我們就不需要覆寫正在用的值。

在圖 7.1 的程式裡，實現了環狀緩衝區的操作。 `buf` 有五個欄位 (field): 一個包含存入緩衝區的向量，四個其它欄位用來放指向向量的索引 (indices)。兩個索引是 `start` 與 `end` ，任何環狀緩衝區的使用都會需要這兩個索引: `start` 指向緩衝區的第一個值，當我們取出一個值時， `start` 會遞增 (incremented)； `end` 指向緩衝區的最後一個值，當我們插入一個新值時， `end` 會遞增。

另外兩個索引， `used` 以及 `new` ，是我們需要給這個應用的基本環狀緩衝區所加入的東西。它們會介於 `start` 與 `end` 之間。實際上，它總是符合

```
start ≤ used ≤ new ≤ end
```

你可以把 `used` 與 `new` 想成是當前匹配 (current match) 的 `start` 與 `end` 。當我們開始一輪匹配時， `used` 會等於 `start` 而 `new` 會等於 `end` 。當下一個字元 (successive character)匹配時，我們需要遞增 `used` 。當 `used` 與 `new` 相等時，我們將開始匹配時，所有存在緩衝區的字元讀入。我們不想要使用超過從匹配時所存在緩衝區的字元，或是重複使用同樣的字元。因此這個 `new` 索引，開始等於 `end` ，但它不會在一輪匹配我們插入新字元至緩衝區一起遞增。

函數 `bref` 接受一個緩衝區與一個索引，並返回索引所在位置的元素。藉由使用 `index` 對向量的長度取 `mod` ，我們可以假裝我們有一個任意長的緩衝區。呼叫 `(new-buf n)` 會產生一個新的緩衝區，能夠容納 `n` 個物件。

要插入一個新值至緩衝區，我們將使用 `buf-insert` 。它將 `end` 遞增，並把新的值放在那個位置 (譯註: 遞增完的位置)。相反的 `buf-pop` 返回一個緩衝區的第一個數值，接著將 `start` 遞增。任何環狀緩衝區都會有這兩個函數。

```
(defstruct buf
  vec (start -1) (used -1) (new -1) (end -1))

(defun bref (buf n)
  (svref (buf-vec buf)
         (mod n (length (buf-vec buf)))))

(defun (setf bref) (val buf n)
  (setf (svref (buf-vec buf)
               (mod n (length (buf-vec buf))))
        val))

(defun new-buf (len)
  (make-buf :vec (make-array len)))

(defun buf-insert (x b)
  (setf (bref b (incf (buf-end b))) x))

(defun buf-pop (b)
  (prog1
    (bref b (incf (buf-start b)))
    (setf (buf-used b) (buf-start b)
          (buf-new  b) (buf-end   b))))

(defun buf-next (b)
  (when (< (buf-used b) (buf-new b))
    (bref b (incf (buf-used b)))))

(defun buf-reset (b)
  (setf (buf-used b) (buf-start b)
        (buf-new  b) (buf-end   b)))

(defun buf-clear (b)
  (setf (buf-start b) -1 (buf-used  b) -1
        (buf-new   b) -1 (buf-end   b) -1))

(defun buf-flush (b str)
  (do ((i (1+ (buf-used b)) (1+ i)))
      ((> i (buf-end b)))
    (princ (bref b i) str)))
```

圖 **7.1** 環狀緩衝區的操作

接下來我們需要兩個特別爲這個應用所寫的函數: `buf-next` 從緩衝區讀取一個值而不取出，而 `buf-reset` 重置 `used` 與 `new` 到初始值，分別是 `start` 與 `end` 。如果我們已經把至 `new` 的值全部讀取完畢時， `buf-next` 返回 `nil` 。區別這個值與實際的值不會產生問題，因爲我們只把值存在緩衝區。

最後 `buf-flush` 透過將所有作用的元素，寫至由第二個參數所給入的流，而 `buf-clear` 通過重置所有的索引至 `-1` 將緩衝區清空。

在圖 7.1 定義的函數被圖 7.2 所使用，包含了字串替換的程式。函數 `file-subst` 接受四個參數；一個查詢字串，一個替換字串，一個輸入檔案以及一個輸出檔案。它創建了代表每個檔案的流，然後呼叫 `stream-subst` 來完成實際的工作。

第二個函數 `stream-subst` 使用本節開始所勾勒的算法。它一次從輸入流讀一個字元。直到輸入字元匹配要尋找的字串時，直接寫至輸出流 (1)。當一個匹配開始時，有關字元在緩衝區 `buf` 排隊等候 (2)。

變數 `pos` 指向我們想要匹配的字元在尋找字串的所在位置。如果 `pos` 等於這個字串的長度，我們有一個完整的匹配，則我們將替換字串寫至輸出流，並清空緩衝區 (3)。如果在這之前匹配失敗，我們可以將緩衝區的第一個元素取出，並寫至輸出流，之後我們重置緩衝區，並從 `pos` 等於 0 重新開始 (4)。

```
(defun file-subst (old new file1 file2)
  (with-open-file (in file1 :direction :input)
     (with-open-file (out file2 :direction :output
                                :if-exists :supersede)
       (stream-subst old new in out))))

(defun stream-subst (old new in out)
  (let* ((pos 0)
         (len (length old))
         (buf (new-buf len))
         (from-buf nil))
    (do ((c (read-char in nil :eof)
            (or (setf from-buf (buf-next buf))
                (read-char in nil :eof))))
        ((eql c :eof))
      (cond ((char= c (char old pos))
             (incf pos)
             (cond ((= pos len)             ; 3
                    (princ new out)
                    (setf pos 0)
                    (buf-clear buf))
                   ((not from-buf)          ; 2
                    (buf-insert c buf))))
            ((zerop pos)                    ; 1
             (princ c out)
             (when from-buf
               (buf-pop buf)
               (buf-reset buf)))
            (t                              ; 4
             (unless from-buf
               (buf-insert c buf))
             (princ (buf-pop buf) out)
             (buf-reset buf)
             (setf pos 0))))
    (buf-flush buf out)))
```

圖 **7.2** 字串替換

下列表格展示了當我們將檔案中的 `"baro"` 替換成 `"baric"` 所發生的事，其中檔案只有一個單字 `"barbarous"`：

CHARACTER	SOURCE	MATCH	CASE	OUTPUT	BUFFER
b	file	b	2		b
a	file	a	2		b a
r	file	r	2		b a r
b	file	o	4	b	b.a r b.
a	buffer	b	1	a	a.r b.
r	buffer	b	1	r	r.b.
b	buffer	b	1		r b:
a	file	a	2		r b:a
r	file	r	2		r b:a
o	file	o	3	baric	r b:a r
u	file	b	1	u	
a	file	b	1	s	

第一欄是當前字元 —— `c` 的值；第二欄顯示是從緩衝區或是直接從輸入流讀取；第三欄顯示需要匹配的字元 —— `old` 的第 **posth** 字元；第四欄顯示那一個條件式 (case)被求值作爲結果；第五欄顯示被寫至輸出流的字元；而最後一欄顯示緩衝區之後的內容。在最後一欄裡， `used` 與 `new` 的位置一樣，由一個冒號 (: colon)表示。

在檔案 `"test1"` 裡有如下文字：

```
The struggle between Liberty and Authority is the most conspicuous feature
in the portions of history with which we are earliest familiar, particularly
in that of Greece, Rome, and England.
```

在我們對 `(file-subst " th" " z" "test1" "test2")` 求值之後，讀取檔案 `"test2"` 爲:

```
The struggle between Liberty and Authority is ze most conspicuous feature
in ze portions of history with which we are earliest familiar, particularly
in zat of Greece, Rome, and England.
```

爲了使這個例子儘可能的簡單，圖 7.2 的程式只將一個字串換成另一個字串。很容易擴展爲搜索一個模式而不是一個字面字串。你只需要做的是，將 `char=` 呼叫換成一個你想要的更通用的匹配函數呼叫。

7.5 宏字元 (Macro Characters)

一個宏字元 (macro character)是獲得 `read` 特別待遇的字元。比如小寫的 `a` ，通常與小寫 `b` 一樣處理，但一個左括號就不同了: 它告訴 Lisp 開始讀入一個列表。

一個宏字元或宏字元組合也稱作 `read-macro` (讀取宏) 。許多 Common Lisp 預定義的讀取宏是縮寫。比如說引用 (Quote): 讀入一個像是 `'a` 的表達式時，它被讀取器展開成 `(quote a)` 。當你輸入引用的表達式 (quoted expression)至頂層時，它們在讀入之時就會被求值，所以一般來說你看不到這樣的轉換。你可以透過顯式呼叫 `read` 使其現形:

```
> (car (read-from-string "'a"))
QUOTE
```

引用對於讀取宏來說是不尋常的，因爲它用單一字元表示。有了一個有限的字元集，你可以在 Common Lisp 裡有許多單一字元的讀取宏，來表示一個或更多字元。

這樣的讀取宏叫做派發 (dispatching)讀取宏，而第一個字元叫做派發字元 (dispatching character)。所有預定義的派發讀取宏使用井號 (`#`)作爲派發字元。我們已經見過好幾個。舉例來說， `#'` 是 `(function ...)` 的縮寫，同樣的 `'` 是 `(quote ...)` 的縮寫。

其它我們見過的派發讀取宏包括 `#(...)` ，產生一個向量； `#nA(...)` 產生陣列； `#\` 產生一個字元； `#S(n ...)` 產生一個結構。當這些型別的每個物件被 `prin1` 顯示時 (或是 `format` 搭配 `~S`)，它們使用對應的讀取宏 [2] 。這表示著你可以寫出或讀回這樣的物件:

```
> (let ((*print-array* t))
    (vectorp (read-from-string (format nil "~S"
                                       (vector 1 2)))))
T
```

當然我們拿回來的不是同一個向量，而是具有同樣元素的新向量。

不是所有物件被顯示時都有著清楚 (distinct)、可讀的形式。舉例來說，函數與雜湊表，傾向於這樣 `#<...>` 被顯示。實際上 `#<...>` 也是一個讀取宏，但是特別用來產生當遇到 `read` 的錯誤。函數與雜湊表不能被寫出與讀回來，而這個讀取宏確保使用者不會有這樣的幻覺。 [3]

當你定義你自己的事物表示法時 (舉例來說，結構的印出函數)，你要將此準則記住。要不使用一個可以被讀回來的表示法，或是使用 `#<...>` 。

Chapter 7 總結 (Summary)

1. 流是輸入的來源或終點。在字元流裡，輸入輸出是由字元組成。
2. 預設的流指向頂層。新的流可以由開啓檔案產生。
3. 你可以解析物件、字元組成的字串、或是單獨的字元。
4. `format` 函數提供了完整的輸出控制。
5. 爲了要替換文字檔案中的字串，你需要將字元讀入緩衝區。
6. 當 `read` 遇到一個宏字元像是 `'`，它呼叫相關的函數。

Chapter 7 練習 (Exercises)

1. 定義一個函數，接受一個檔案名並返回一個由字串組成的列表，來表示檔案裡的每一行。
2. 定義一個函數，接受一個檔案名並返回一個由表達式組成的列表，來表示檔案裡的每一行。
3. 假設有某種格式的檔案檔案，註解是由 `%` 字元表示。從這個字元開始直到行尾都會被忽略。定義一個函數，接受兩個檔案名稱，並拷貝第一個檔案的內容去掉註解，寫至第二個檔案。
4. 定義一個函數，接受一個二維浮點陣列，將其用簡潔的欄位顯示。每個元素應印至小數點二位，一欄十個字元寬。（假設所有的字元可以容納）。你會需要 `array-dimensions` (參見 361 頁，譯註: Appendix D)。
5. 修改 `stream-subst` 來允許萬用字元 (wildcard) 可以在模式中使用。若字元 `+` 出現在 `old` 裡，它應該匹配任何輸入字元。
6. 修改 `stream-subst` 來允許模式可以包含一個用來匹配任何數字的元素，以及一個可以匹配任何英文字元的元素或是一個可以匹配任何字元的元素。模式必須可以匹配任何特定的輸入字元。(提示: `old` 可以不是一個字串。)

腳註

[1] 你可以給一個字串取代路徑名，但這樣就不可攜了 (portable)。
[2] 要讓向量與陣列這樣被顯示，將 `*print-array*` 設爲真。
[3] Lisp 不能只用 `#'` 來表示函數，因爲 `#'` 本身無法提供表示閉包的方式。

第八章：符號

我們一直在使用符號。在符號看似簡單的表面之下，又好像沒有那麼簡單。起初最好不要糾結於背後的實現機制。可以把符號當成資料物件與名字那樣使用，而不需要理解兩者是如何關聯起來的。但到了某個時間點，停下來思考背後是究竟是如何工作會是很有用的。本章解釋了背後實現的細節。

8.1 符號名 (Symbol Names)

第二章描述過，符號是變數的名字，符號本身以物件所存在。但 Lisp 符號的可能性，要比在多數語言僅允許作爲變數名來得廣泛許多。實際上，符號可以用任何字串當作名字。可以通過呼叫 `symbol-name` 來獲得符號的名字：

```
> (symbol-name 'abc)
"ABC"
```

注意到這個符號的名字，打印出來都是大寫字母。預設情況下， Common Lisp 在讀入時，會把符號名字所有的英文字母都轉成大寫。代表 Common Lisp 預設是不分大小寫的：

```
> (eql 'abc 'Abc)
T
> (CaR '(a b c))
A
```

一個名字包含空白，或其它可能被讀取器認爲是重要的字元的符號，要用特殊的語法來引用。任何存在垂直槓 (vertical bar)之間的字元序列將被視爲符號。可以如下這般在符號的名字中，放入任何字元：

```
> (list '|Lisp 1.5| '|| '|abc| '|ABC|)
(|Lisp 1.5| || |abc| ABC)
```

當這種符號被讀入時，不會有大小寫轉換，而宏字元與其他的字元被視爲一般字元。

那什麼樣的符號不需要使用垂直槓來參照呢？基本上任何不是數字，或不包含讀取器視爲重要的字元的符號。一個快速找出你是否可以不用垂直槓來引用符號的方法，是看看 Lisp 如何印出它的。如果 Lisp 沒有用垂直槓表示一個符號，如上述列表的最後一個，那麼你也可以不用垂直槓。

記得，垂直槓是一種表示符號的特殊語法。它們不是符號的名字之一：

```
> (symbol-name '|a b c|)
"a b c"
```

(如果想要在符號名稱內使用垂直槓，可以放一個反斜線在垂直槓的前面。)

譯註: 反斜線是 \ (backslash)。

8.2 屬性列表 (Property Lists)

在 Common Lisp 裡，每個符號都有一個屬性列表（property-list）或稱爲 `plist` 。函數 `get` 接受符號及任何型別的鍵值，然後返回在符號的屬性列表中，與鍵值相關的數值：

```
> (get 'alizarin 'color)
NIL
```

它使用 `eql` 來比較各個鍵。若某個特定的屬性沒有找到時， `get` 返回 `nil` 。

要將值與鍵關聯起來時，你可以使用 `setf` 及 `get` :

```
> (setf (get 'alizarin 'color) 'red)
RED
> (get 'alizarin 'color)
RED
```

現在符號 `alizarin` 的 `color` 屬性是 `red` 。

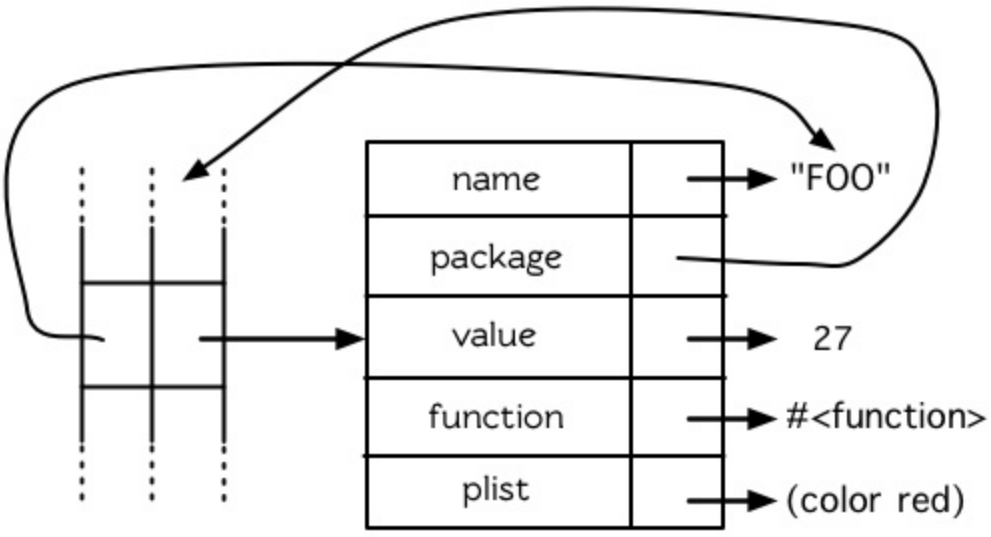

圖 **8.1** 符號的結構

```
> (setf (get 'alizarin 'transparency) 'high)
HIGH
```

```
> (symbol-plist 'alizarin)
(TRANSPARENCY HIGH COLOR RED)
```

注意，屬性列表不以關聯列表（assoc-lists）的形式表示，雖然用起來感覺是一樣的。

在 Common Lisp 裡，屬性列表用得不多。他們大部分被雜湊表取代了（4.8 小節）。

8.3 符號很不簡單 (Symbols Are Big)

當我們輸入名字時，符號就被悄悄地創建出來了，而當它們被顯示時，我們只看的到符號的名字。某些情況下，把符號想成是表面所見的東西就好，別想太多。但有時候符號不像看起來那麼簡單。

從我們如何使用以及檢視符號，符號看起來像是整數那樣的小物件。而符號實際上確實是一個物件，差不多像是由 `defstruct` 定義的那種結構。符號可以有名字、主包（home package）、作爲變數的值、作爲函數的值以及帶有一個屬性列表。圖 8.1 示範了符號在內部是如何表示的。

很少有程式會使用很多符號，以致於值得用其它的東西來代替符號以節省空間。但需要記住的是，符號是實際的物件，不僅是名字而已。當兩個變數設成相同的符號時，與兩個變數設成相同列表一樣：兩個變數的指標都指向同樣的物件。

8.4 創建符號 (Creating Symbols)

8.1 節示範了如何取得符號的名字。另一方面，用字串生成符號也是有可能的。但比較複雜一點，因爲我們需要先介紹包（package）。

概念上來說，包是將名字映射到符號的符號表（symbol-tables）。每個普通的符號都屬於一個特定的包。符號屬於某個包，我們稱爲符號被包扣押（intern）了。函數與變數用符號作爲名稱。包藉由限制哪個符號可以存取來實現模組性（modularity），也是因爲這樣，我們才可以引用到函數與變數。

大多數的符號在讀取時就被扣押了。在第一次輸入一個新符號的名字時，Lisp 會產生一個新的符號物件，並將它扣押到當下的包裡（預設是 `common-lisp-user` 包)。但也可以通過給入字串與選擇性包參數給 `intern` 函數，來扣押一個名稱爲字串名的符號:

```
> (intern "RANDOM-SYMBOL")
RANDOM-SYMBOL
NIL
```

選擇性包參數預設是當前的包，所以前述的表達式，返回當前包裡的一個符號，此符號

的名字是“RANDOM-SYMBOL”，若此符號尚未存在時，會創建一個這樣的符號出來。第二個返回值告訴我們符號是否存在；在這個情況，它不存在。

不是所有的符號都會被扣押。有時候有一個自由的（uninterned）符號是有用的，這和公用電話本是一樣的原因。自由的符號叫做 *gensyms* 。我們將會在第 10 章討論宏（Macro）時，理解 `gensym` 的作用。

8.5 多重包 (Multiple Packages)

大的程式通常切分爲多個包。如果程式的每個部分都是一個包，那麼開發程式另一個部分的某個人，將可以使用符號來作爲函數名或變數名，而不必擔心名字在別的地方已經被用過了。

在沒有提供定義多個命名空間的語言裡，工作於大項目的程式設計師，通常需要想出某些規範（convention），來確保他們不會使用同樣的名稱。舉例來說，程式設計師寫顯示相關的程式（display code）可能用 `disp_` 開頭的名字，而寫數學相關的程式（math code）的程式設計師僅使用由 `math_` 開始的程式。所以若是數學相關的程式裡，包含一個做快速傅立葉轉換的函數時，可能會叫做 `math_fft` 。

包不過是提供了一種方便的方式來自動辦到此事。如果你將函數定義在單獨的包裡，可以隨意使用你喜歡的名字。只有你明確導出（ `export` ）的符號會被別的包看到，而通常前面會有包的名字(或修飾符)。

舉例來說，假設一個程式分爲兩個包， `math` 與 `disp` 。如果符號 `fft` 被 `math` 包導出，則 `disp` 包裡可以用 `math:fft` 來參照它。在 `math` 包裡，可以只用 `fft` 來參照。

下面是你可能會放在檔案最上方，包含獨立包的程式：

```
(defpackage "MY-APPLICATION"
            (:use "COMMON-LISP" "MY-UTILITIES")
            (:nicknames "APP")
            (:export "WIN" "LOSE" "DRAW"))

(in-package my-application)
```

`defpackage` 定義一個新的包叫做 `my-application` [1] 它使用了其他兩個包， `common-lisp` 與 `my-utilities` ，這代表著可以不需要用包修飾符（package qualifiers）來存取這些包所導出的符號。許多包都使用了 `common-lisp` 包 —— 因爲你不會想給 Lisp 自帶的運算子與變數再加上修飾符。

`my-application` 包本身只輸出三個符號: `WIN` 、 `LOSE` 以及 `DRAW` 。由於呼叫 `defpackage` 給了 `my-application` 一個匿稱 `app` ，則別的包可以這樣引用到這些符號，比如

app:win 。

defpackage 伴隨著一個 in-package ，確保當前包是 my-application 。所有其它未修飾的符號會被扣押至 my-application —— 除非之後有別的 in-package 出現。當一個檔案被載入時，當前的包總是被重置成載入之前的值。

8.6 關鍵字 (Keywords)

在 keyword 包的符號 (稱爲關鍵字)有兩個獨特的性質：它們總是對自己求值，以及可以在任何地方引用它們，如 :x 而不是 keyword:x 。我們首次在 44 頁 (譯註: 3.10 小節）介紹關鍵字參數時， (member '(a) '((a) (z)) test: #'equal) 比 (member '(a) '((a) (z)) :test #'equal) 讀起來更自然。現在我們知道爲什麼第二個較彆扭的形式才是對的。 test 前的冒號字首，是關鍵字的識別符。

爲什麼使用關鍵字而不用一般的符號？因爲關鍵字在哪都可以存取。一個函數接受符號作爲實參，應該要寫成預期關鍵字的函數。舉例來說，這個函數可以安全地在任何包裡呼叫:

```
(defun noise (animal)
  (case animal
    (:dog :woof)
    (:cat :meow)
    (:pig :oink)))
```

但如果是用一般符號寫成的話，它只在被定義的包內正常工作，除非關鍵字也被導出了。

8.7 符號與變數 (Symbols and Variables)

Lisp 有一件可能會使你困惑的事情是，符號與變數的從兩個非常不同的層面互相關聯。當符號是特別變數（special variable）的名字時，變數的值存在符號的 value 欄位（圖 8.1）。 symbol-value 函數引用到那個欄位，所以在符號與特殊變數的值之間，有直接的連接關係。

而對於詞法變數（lexical variables）來說，事情就完全不一樣了。一個作爲詞法變數的符號只不過是個佔位符（placeholder）。編譯器會將其轉爲一個寄存器（register）或記憶體位置的引用位址。在最後編譯出來的

在程式裡，我們無法追蹤這個符號 (除非它被保存在除錯器「debugger」的某個地方)。因此符號與詞法變數的值之間是沒有連接的；只要一有值，符號就消失了。

8.8 範例：隨機文字 (Example: Random Text)

如果你要寫一個處理單詞的程式，通常使用符號會比字串來得好，因爲符號概念上是原子性的（atomic）。符號可以用 `eql` 一步比較完成，而字串需要使用 `string=` 或 `string-equal` 逐一字元做比較。作爲一個範例，本節將示範如何寫一個程式來產生隨機文字。程式的第一部分會讀入一個範例檔案（越大越好），用來累積之後所給入的相關單詞的可能性（likeilhood）的資訊。第二部分在每一個單詞都根據原本的範例，產生一個隨機的權重（weight）之後，隨機走訪根據第一部分所產生的網路。

產生的文字將會是部分可信的（locally plausible），因爲任兩個出現的單詞也是輸入檔案裡，兩個同時出現的單詞。令人驚訝的是，獲得看起來是 —— 有意義的整句 —— 甚至整個段落是的頻率相當高。

圖 8.2 包含了程式的上半部，用來讀取範例檔案的程式。

```
(defparameter *words* (make-hash-table :size 10000))

(defconstant maxword 100)

(defun read-text (pathname)
  (with-open-file (s pathname :direction :input)
    (let ((buffer (make-string maxword))
          (pos 0))
      (do ((c (read-char s nil :eof)
              (read-char s nil :eof)))
          ((eql c :eof))
        (if (or (alpha-char-p c) (char= c #\'))
            (progn
              (setf (aref buffer pos) c)
              (incf pos))
            (progn
              (unless (zerop pos)
                (see (intern (string-downcase
                              (subseq buffer 0 pos))))
                (setf pos 0))
              (let ((p (punc c)))
                (if p (see p)))))))))

(defun punc (c)
  (case c
    (#\. '|.|) (#\, '|,|) (#\; '|;|)
    (#\! '|!|) (#\? '|?|) ))

(let ((prev `|.|))
  (defun see (symb)
    (let ((pair (assoc symb (gethash prev *words*))))
      (if (null pair)
          (push (cons symb 1) (gethash prev *words*))
```

```
            (incf (cdr pair))))
      (setf prev symb)))
```

圖 **8.2** 讀取範例檔案

從圖 8.2 所導出的資料，會被存在雜湊表 `*words*` 裡。這個雜湊表的鍵是代表單詞的符號，而值會像是下列的關聯列表（assoc-lists）：

```
((|sin| . 1) (|wide| . 2) (|sights| . 1))
```

使用彌爾頓的失樂園 [http://zh.wikipedia.org/wiki/%E5%A4%B1%E6%A8%82%E5%9C%92]作爲範例檔案時，這是與鍵 `|discover|` 有關的值。它指出了 “discover” 這個單詞，在詩裡面用了四次，與 “wide” 用了兩次，而 “sin” 與 ”sights” 各一次。(譯註: 詩可以在這裡找到 http://www.paradiselost.org/)

函數 `read-text` 累積了這個資訊。這個函數接受一個路徑名（pathname），然後替每一個出現在檔案中的單詞，生成一個上面所展示的關聯列表。它的工作方式是，逐字讀取檔案的每個字元，將累積的單詞存在字串 `buffer` 。 `maxword` 設成 `100` ，程式可以讀取至多 100 個單詞，對英語來說足夠了。

只要下個字元是一個字（由 `alpha-char-p` 決定）或是一撇(apostrophe)，就持續累積字元。任何使單詞停止累積的字元會送給 `see` 。數種標點符號（punctuation）也被視爲是單詞；函數 `punc` 返回標點字元的僞單詞（pseudo-word）。

函數 `see` 註冊每一個我們看過的單詞。它需要知道前一個單詞，以及我們剛確認過的單詞 —— 這也是爲什麼要有變數 `prev` 存在。起初這個變數設爲僞單詞裡的句點；在 `see` 函數被呼叫後， `prev` 變數包含了我們最後見過的單詞。

在 `read-text` 返回之後， `*words*` 會包含輸入檔案的每一個單詞的條目（entry）。通過呼叫 `hash-table-count` 你可以了解有多少個不同的單詞存在。鮮少有英文檔案會超過 10000 個單詞。

現在來到了有趣的部份。圖 8.3 包含了從圖 8.2 所累積的資料來產生文字的程式。 `generate-text` 函數導出整個過程。它接受一個要產生幾個單詞的數字，以及選擇性傳入前一個單詞。使用預設值，會讓產生出來的檔案從句子的開頭開始。

```
(defun generate-text (n &optional (prev '|.|))
  (if (zerop n)
      (terpri)
      (let ((next (random-next prev)))
        (format t "~A " next)
        (generate-text (1- n) next))))
```

```
(defun random-next (prev)
  (let* ((choices (gethash prev *words*))
         (i (random (reduce #'+ choices
                            :key #'cdr))))
    (dolist (pair choices)
      (if (minusp (decf i (cdr pair)))
          (return (car pair))))))
```

圖 **8.3** 產生文字

要取得一個新的單詞， `generate-text` 使用前一個單詞，接著呼叫 `random-next` 。 `random-next` 函數根據每個單詞出現的機率加上權重，隨機選擇伴隨輸入文字中 `prev` 之後的單詞。

現在會是測試運行下程式的好時機。但其實你早看過一個它所產生的範例： 就是本書開頭的那首詩，是使用彌爾頓的失樂園作爲輸入檔案所產生的。

(譯註: 詩可在這裡看，或是瀏覽書的第 vi 頁)

Half lost on my firmness gains more glad heart,

Or violent and from forage drives

A glimmering of all sun new begun

Both harp thy discourse they match’d,

Forth my early, is not without delay;

For their soft with whirlwind; and balm.

Undoubtedly he scornful turn’d round ninefold,

Though doubled now what redounds,

And chains these a lower world devote, yet inflicted?

Till body or rare, and best things else enjoy’d in heav’n

To stand divided light at ev’n and poise their eyes,

Or nourish, lik’ning spiritual, I have thou appear.

— Henley

Chapter 8 總結 (Summary)

1. 符號的名字可以是任何字串，但由 `read` 創建的符號預設會被轉成大寫。
2. 符號帶有相關聯的屬性列表，雖然他們不需要是相同的形式，但行爲像是 assoc-lists 。
3. 符號是實質的物件，比較像結構，而不是名字。
4. 包將字串映射至符號。要在包裡給符號創造一個條目的方法是扣留它。符號不需要被扣留。
5. 包通過限制可以引用的名稱增加模組性。預設的包會是 user 包，但爲了提高模組性，大的程式通常分成數個包。
6. 可以讓符號在別的包被存取。關鍵字是自身求值並在所有的包裡都可以存取。
7. 當一個程式用來操作單詞時，用符號來表示單詞是很方便的。

Chapter 8 練習 (Exercises)

1. 可能有兩個同名符號，但卻不 `eql` 嗎？
2. 估計一下用字串表示 “FOO” 與符號表示 foo 所使用記憶體空間的差異。
3. 只使用字串作爲實參 來呼叫 137 頁的 `defpackage` 。應該使用符號比較好。爲什麼使用字串可能比較危險呢？
4. 加入需要的程式碼，使圖 7.1 的程式可以放在一個叫做 `"RING"` 的包裡，而圖 7.2 的程式放在一個叫做 `"FILE"` 包裡。不需要更動現有的程式。
5. 寫一個確認引用的句子是否是由 Henley 生成的程式 (8.8 節)。
6. 寫一版 Henley，接受一個單詞，並產生一個句子，該單詞在句子的中間。

腳註

[1] 呼叫 `defpackage` 裡的名字全部大寫的緣故在 8.1 節提到過，符號的名字預設被轉成大寫。

第九章：數字

處理數字是 Common Lisp 的強項之一。Common Lisp 有著豐富的數值型別，而 Common Lisp 操作數字的特性與其他語言比起來更受人喜愛。

9.1 型別 (Types)

Common Lisp 提供了四種不同型別的數字：整數、浮點數、比值與複數。本章所講述的函數適用於所有型別的數字。有幾個不能用在複數的函數會特別說明。

整數寫成一串數字：如 `2001` 。浮點數是可以寫成一串包含小數點的數字，如 `253.72` ，或是用科學表示法，如 `2.5372e2` 。比值是寫成由整陣列成的分數：如 `2/3` 。而複數 `a+bi` 寫成 `#c(a b)` ，其中 `a` 與 `b` 是任兩個型別相同的實數。

謂詞 `integerp` 、 `floatp` 以及 `complexp` 針對相應的數字型別返回真。圖 9.1 展示了數值型別的層級。

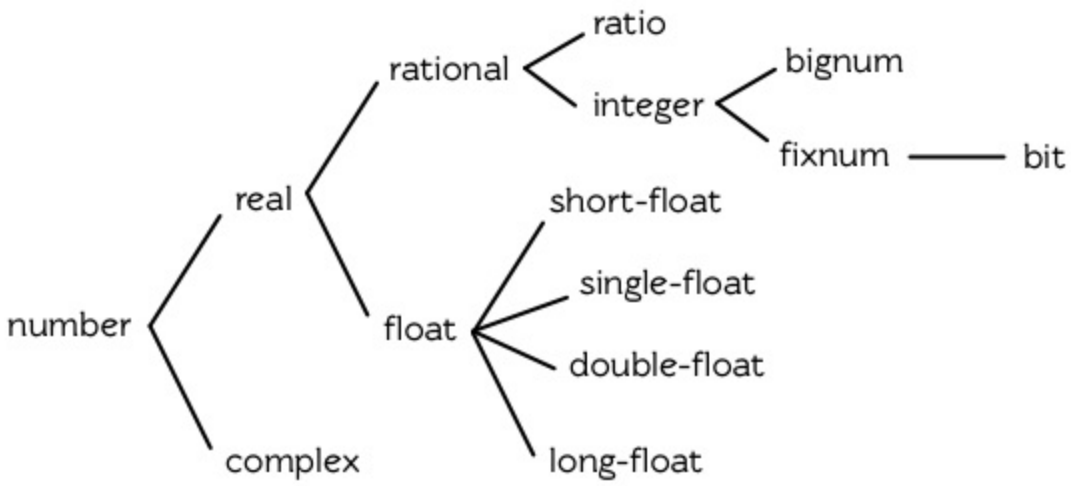

圖 **9.1:** 數值型別

要決定計算過程會返回何種數字，以下是某些通用的經驗法則：

1. 如果數值函數接受一個或多個浮點數作爲參數，則返回值會是浮點數 (或是由浮點陣列成的複數)。所以 `(+ 1.0 2)` 求值爲 `3.0` ，而 `(+ #c(0 1.0) 2)` 求值爲 `#c(2.0 1.0)` 。
2. 可約分的比值會被轉換成最簡分數。所以 `(/ 10 2)` 會返回 `5` 。
3. 若計算過程中複數的虛部變成 `0` 時，則複數會被轉成實數 。所以 `(+ #c(1 -1) #c(2 1))` 求值成 `3` 。

第二、第三個規則可以在讀入參數時直接應用，所以：

```
> (list (ratiop 2/2) (complexp #c(1 0)))
(NIL NIL)
```

9.2 轉換及取出 (Conversion and Extraction)

Lisp 提供四種不同型別的數字的轉換及取出位數的函數。函數 `float` 將任何實數轉換成浮點數:

```
> (mapcar #'float '(1 2/3 .5))
(1.0 0.6666667 0.5)
```

將數字轉成整數未必需要轉換，因爲它可能牽涉到某些資訊的喪失。函數 `truncate` 返回任何實數的整數部分:

```
> (truncate 1.3)
1
0.29999995
```

第二個返回值 `0.29999995` 是傳入的參數減去第一個返回值。(會有 0.00000005 的誤差是因爲浮點數的計算本身就不精確。)

函數 `floor` 與 `ceiling` 以及 `round` 也從它們的參數中導出整數。使用 `floor` 返回小於等於其參數的最大整數，而 `ceiling` 返回大於或等於其參數的最小整數，我們可以將 `mirror?` (46 頁，譯註: 3.11 節)改成可以找出所有迴文（palindromes）的版本:

```
(defun palindrome? (x)
  (let ((mid (/ (length x) 2)))
    (equal (subseq x 0 (floor mid))
           (reverse (subseq x (ceiling mid))))))
```

和 `truncate` 一樣， `floor` 與 `ceiling` 也返回傳入參數與第一個返回值的差，作爲第二個返回值。

```
> (floor 1.5)
1
0.5
```

實際上，我們可以把 `truncate` 想成是這樣定義的:

```
(defun our-truncate (n)
    (if (> n 0)
        (floor n)
```

```
        (ceiling n)))
```

函數 round 返回最接近其參數的整數。當參數與兩個整數的距離相等時， Common Lisp 和很多程式語言一樣，不會往上取（round up）整數。而是取最近的偶數：

```
> (mapcar #'round '(-2.5 -1.5 1.5 2.5))
(-2 -2 2 2)
```

在某些數值應用中這是好事，因爲舍入誤差（rounding error）通常會互相抵消。但要是用戶期望你的程式將某些值取整數時，你必須自己提供這個功能。 [1] 與其他的函數一樣， round 返回傳入參數與第一個返回值的差，作爲第二個返回值。

函數 mod 僅返回 floor 返回的第二個返回值；而 rem 返回 truncate 返回的第二個返回值。我們在 94 頁（譯註： 5.7 節）曾使用 mod 來決定一個數是否可被另一個整除，以及 127 頁（譯註： 7.4 節）用來找出環狀緩衝區（ring buffer）中，元素實際的位置。

關於實數，函數 signum 返回 1 、 0 或 -1 ，取決於它的參數是正數、零或負數。函數 abs 返回其參數的絕對值。因此 (* (abs x) (signum x)) 等於 x 。

```
> (mapcar #'signum '(-2 -0.0 0.0 0 .5 3))
(-1 -0.0 0.0 0 1.0 1)
```

在某些應用裡， -0.0 可能自成一格（in its own right），如上所示。實際上功能上幾乎沒有差別，因爲數值 -0.0 與 0.0 有著一樣的行爲。

比值與複數概念上是兩部分的結構。(譯註：像 **Cons** 這樣的兩部分結構) 函數 numerator 與 denominator 返回比值或整數的分子與分母。（如果數字是整數，前者返回該數，而後者返回 1 。）函數 realpart 與 imgpart 返回任何數字的實數與虛數部分。（如果數字不是複數，前者返回該數字，後者返回 0 。）

函數 random 接受一個整數或浮點數。這樣形式的表達式 (random n) ，會返回一個大於等於 0 並小於 n 的數字，並有著與 n 相同的型別。

9.3 比較 (Comparison)

謂詞 = 比較其參數，當數值上相等時 —— 即兩者的差爲零時，返回真。

```
> (= 1 1.0)
T
> (eql 1 1.0)
NIL
```

= 比起 eql 來得寬鬆，但參數的型別需一致。

用來比較數字的謂詞爲 < （小於）、 <= （小於等於）、 = （等於）、 >= （大於等於）、 > (大於) 以及 /= （不相等）。以上所有皆接受一個或多個參數。只有一個參數時，它們全返回真。

```
(<= w x y z)
```

等同於二元運算子的結合（conjunction），應用至每一對參數上:

```
(and (<= w x) (<= x y) (<= y z))
```

由於 /= 若它的兩個參數不等於時會返回真，表達式

```
(/= w x y z)
```

等同於

```
(and (/= w x) (/= w y) (/= w z)
     (/= x y) (/= y z) (/= y z))
```

特殊的謂詞 zerop 、 plusp 與 minusp 接受一個參數，分別於參數 = 、 > 、 < 零時，返回真。雖然 -0.0 （如果實現有使用它）前面有個負號，但它 = 零，

```
> (list (minusp -0.0) (zerop -0.0))
(NIL T)
```

因此對 -0.0 使用 zerop ，而不是 minusp 。

謂詞 oddp 與 evenp 只能用在整數。前者只對奇數返回真，後者只對偶數返回真。

本節定義的謂詞中，只有 = 、 /= 與 zerop 可以用在複數。

函數 max 與 min 分別返回其參數的最大值與最小值。兩者至少需要給一個參數:

```
> (list (max 1 2 3 4 5) (min 1 2 3 4 5))
(5 1)
```

如果參數含有浮點數的話，結果的型別取決於各家實現。

9.4 算術 (Arithematic)

用來做加減的函數是 + 與 - 。兩者皆接受任何數量的參數，包括沒有參數，在沒有參數的情況下返回 0 。（譯註: - 在沒有參數的情況下會報錯，至少要一個參數）一個這樣形式的表達式 (- n) 返回 -n 。一個這樣形式的表達式

```
(- x y z)
```

等同於

```
(- (- x y) z)
```

有兩個函數 1+ 與 1- ，分別將參數加 1 與減 1 後返回。 1- 有一點誤導，因爲 (1- x) 返回 x-1 而不是 1-x 。

宏 incf 及 decf 分別遞增與遞減數字。這樣形式的表達式 (incf x n) 類似於 (setf x (+ x n)) 的效果，而 (decf x n) 類似於 (setf x (- x n)) 的效果。這兩個形式裡，第二個參數皆是選擇性給入的，預設值爲 1 。

用來做乘法的函數是 * 。接受任何數量的參數。沒有參數時返回 1 。否則返回參數的乘積。

除法函數 / 至少要給一個參數。這樣形式的呼叫 (/ n) 等同於 (/ 1 n) ，

```
> (/ 3)
1/3
```

而這樣形式的呼叫

```
(/ x y z)
```

等同於

```
(/ (/ x y) z)
```

注意 - 與 / 兩者在這方面的相似性。

當給定兩個整數時， / 若第一個不是第二個的倍數時，會返回一個比值:

```
> (/ 365 12)
365/12
```

舉例來說，如果你試著找出平均每一個月有多長，可能會有頂層在逗你玩的感覺。在這個情況下，你需要的是，對比值呼叫 float ，而不是對兩個整數做 / 。

```
> (float 365/12)
30.416666
```

9.5 指數 (Exponentiation)

要找到 \(x^n\) 呼叫 `(expt x n)`，

```
> (expt 2 5)
32
```

而要找到 \(log_nx\) 呼叫 `(log x n)`：

```
> (log 32 2)
5.0
```

通常返回一個浮點數。

要找到 \(e^x\) 有一個特別的函數 `exp`，

```
> (exp 2)
7.389056
```

而要找到自然對數，你可以使用 `log` 就好，因爲第二個參數預設爲 `e`：

```
> (log 7.389056)
2.0
```

要找到立方根，你可以呼叫 `expt` 用一個比值作爲第二個參數，

```
> (expt 27 1/3)
3.0
```

但要找到平方根，函數 `sqrt` 會比較快:

```
> (sqrt 4)
2.0
```

9.6 三角函數 (Trigometric Functions)

常數 `pi` 是 `π` 的浮點表示法。它的精度取決於各家實現。函數 `sin` 、 `cos` 及 `tan` 分別可以找到正弦、餘弦及正交函數，其中角度以徑度表示：

```
> (let ((x (/ pi 4)))
    (list (sin x) (cos x) (tan x)))
(0.7071067811865475d0 0.7071067811865476d0 1.0d0)
;;; 譯註: CCL 1.8  SBCL 1.0.55 下的結果是
;;; (0.7071067811865475D0 0.7071067811865476D0 0.9999999999999999D0)
```

這些函數都接受負數及複數參數。

函數 `asin` 、 `acos` 及 `atan` 實現了正弦、餘弦及正交的反函數。參數介於 `-1` 與 `1` 之間（包含）時， `asin` 與 `acos` 返回實數。

雙曲正弦、雙曲餘弦及雙曲正交分別由 `sinh` 、 `cosh` 及 `tanh` 實現。它們的反函數同樣爲 `asinh` 、 `acosh` 以及 `atanh` 。

9.7 表示法 (Representations)

Common Lisp 沒有限制整數的大小。可以塞進一個字（word）記憶體的小整數稱爲定長數(fixnums)。在計算過程中，整數無法塞入一個字時，Lisp 切換至使用多個字的表示法（一個大數 「bignum」）。所以整數的大小限製取決於實體記憶體，而不是語言。

常數 `most-positive-fixnum` 與 `most-negative-fixnum` 表示一個實現不使用大數所可表示的最大與最小的數字大小。在很多實現裡，它們爲：

```
> (values most-positive-fixnum most-negative-fixnum)
536870911
-536870912
;;; 譯註: CCL 1.8 的結果爲
1152921504606846975
-1152921504606846976
;;; SBCL 1.0.55 的結果爲
4611686018427387903
-4611686018427387904
```

謂詞 `typep` 接受一個參數及一個型別名稱，並返回指定型別的參數。所以，

```
> (typep 1 'fixnum)
T
> (type (1+ most-positive-fixnum) 'bignum)
T
```

浮點數的數值限制是取決於各家實現的。 Common Lisp 提供了至多四種型別的浮點數：短浮點 `short-float` 、 單浮點 `single-float` 、雙浮點 `double-float` 以及長浮點 `long-float` 。Common Lisp 的實現是不需要用不同的格式來表示這四種型別（很少有實現這麼幹）。

一般來說，短浮點應可塞入一個字，單浮點與雙浮點提供普遍的單精度與雙精度浮點數的概念，而長浮點，如果想要的話，可以是很大的數。但實現可以不對這四種型別做區別，也是完全沒有問題的。

你可以指定你想要何種格式的浮點數，當數字是用科學表示法時，可以通過將 `e` 替換爲 `s f d l` 來得到不同的浮點數。（你也可以使用大寫，這對長浮點來說是個好主意，因爲 `l` 看起來太像 `1` 了。）所以要表示最大的 `1.0` 你可以寫 `1L0` 。

（譯註: `s` 爲短浮點、 `f` 爲單浮點、 `d` 爲雙浮點、 `l` 爲長浮點。）

在給定的實現裡，用十六個全局常數標明了每個格式的限制。它們的名字是這種形式: `m-s-f` ，其中 `m` 是 `most` 或 `least` ， `s` 是 `positive` 或 `negative` ，而 `f` 是四種浮點數之一。 λ [http://acl.readthedocs.org/en/latest/zhTW/notes.html#notes-150]

浮點數下溢（underflow）與溢出（overflow），都會被 Common Lisp 視爲錯誤 :

```
> (* most-positive-long-float 10)
Error: floating-point-overflow
```

9.8 範例：追蹤光線 (Example: Ray-Tracing)

作爲一個數值應用的範例，本節示範了如何撰寫一個光線追蹤器 (ray-tracer)。光線追蹤是一個高級的 (deluxe)渲染算法: 它產生出逼真的圖像，但需要花點時間。

要產生一個 3D 的圖像，我們至少需要定義四件事: 一個觀測點 (eye)、一個或多個光源、一個由一個或多個平面所組成的模擬世界 (simulated world)，以及一個作爲通往這個世界的窗戶的平面 (圖像平面「image plane」)。我們產生出的是模擬世界投影在圖像平面區域的圖像。

光線追蹤獨特的地方在於，我們如何找到這個投影: 我們一個一個像素地沿著圖像平面走，追蹤回到模擬世界裡的光線。這個方法帶來三個主要的優勢: 它讓我們容易得到現實世界的光學效應 (optical effect)，如透明度 (transparency)、反射光 (reflected light)以及產生陰影 (cast shadows)；它讓我們可以直接用任何我們想要的幾何的物體，來定義出模擬的世界，而不需要用多邊形 (polygons)來建構它們；以及它很簡單實現。

```
(defun sq (x) (* x x))

(defun mag (x y z)
  (sqrt (+ (sq x) (sq y) (sq z))))

(defun unit-vector (x y z)
  (let ((d (mag x y z)))
    (values (/ x d) (/ y d) (/ z d))))
```

```
(defstruct (point (:conc-name nil))
  x y z)

(defun distance (p1 p2)
  (mag (- (x p1) (x p2))
       (- (y p1) (y p2))
       (- (z p1) (z p2))))

(defun minroot (a b c)
  (if (zerop a)
      (/ (- c) b)
      (let ((disc (- (sq b) (* 4 a c))))
        (unless (minusp disc)
          (let ((discrt (sqrt disc)))
            (min (/ (+ (- b) discrt) (* 2 a))
                 (/ (- (- b) discrt) (* 2 a))))))))
```

圖 **9.2** 實用數學函數

圖 9.2 包含了我們在光線追蹤器裡會需要用到的一些實用數學函數。第一個 `sq` ，返回其參數的平方。下一個 `mag` ，返回一個給定 `x y z` 所組成向量的大小 (magnitude)。這個函數被接下來兩個函數用到。我們在 `unit-vector` 用到了，此函數返回三個數值，來表示與單位向量有著同樣方向的向量，其中向量是由 `x y z` 所組成的:

```
> (multiple-value-call #'mag (unit-vector 23 12 47))
1.0
```

我們在 `distance` 也用到了 `mag` ，它返回三維空間中，兩點的距離。（给 `point` 结构定义一个 `conc-name` （值为 `nil` ），代表访问字段的函数名会跟字段名相同：举例来说，`x` 而不是 `point-x` 。）

最後 `minroot` 接受三個實數， `a` , `b` 與 `c` ，並返回滿足等式 \(ax^2+bx+c=0\) 的最小實數 `x` 。當 `a` 不爲 \(0\) 時，這個等式的根由下面這個熟悉的式子給出:

\[x = \dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}\]

圖 9.3 包含了定義一個最小光線追蹤器的程式。 它產生通過單一光源照射的黑白圖像，與觀測點 (eye)處於同個位置。 (結果看起來像是閃光攝影術 (flash photography)拍出來的)

`surface` 結構用來表示模擬世界中的物體。更精確的說，它會被 `included` 至定義具體型別物體的結構裡，像是球體 (spheres)。 `surface` 結構本身只包含一個欄位: 一個 `color` 範圍從 0 (黑色) 至 1 (白色)。

```
(defstruct surface color)
```

```
(defparameter *world* nil)
(defconstant eye (make-point :x 0 :y 0 :z 200))

(defun tracer (pathname &optional (res 1))
  (with-open-file (p pathname :direction :output)
    (format p "P2 ~A ~A 255" (* res 100) (* res 100))
    (let ((inc (/ res)))
      (do ((y -50 (+ y inc)))
          ((< (- 50 y) inc))
        (do ((x -50 (+ x inc)))
            ((< (- 50 x) inc))
          (print (color-at x y) p))))))

(defun color-at (x y)
  (multiple-value-bind (xr yr zr)
                       (unit-vector (- x (x eye))
                                    (- y (y eye))
                                    (- 0 (z eye)))
    (round (* (sendray eye xr yr zr) 255))))

(defun sendray (pt xr yr zr)
  (multiple-value-bind (s int) (first-hit pt xr yr zr)
    (if s
        (* (lambert s int xr yr zr) (surface-color s))
        0)))

(defun first-hit (pt xr yr zr)
  (let (surface hit dist)
    (dolist (s *world*)
      (let ((h (intersect s pt xr yr zr)))
        (when h
          (let ((d (distance h pt)))
            (when (or (null dist) (< d dist))
              (setf surface s hit h dist d))))))
    (values surface hit)))

(defun lambert (s int xr yr zr)
  (multiple-value-bind (xn yn zn) (normal s int)
    (max 0 (+ (* xr xn) (* yr yn) (* zr zn)))))
```

圖 **9.3** 光線追蹤。

圖像平面會是由 x 軸與 y 軸所定義的平面。觀測者 (eye) 會在 z 軸，距離原點 200 個單位。所以要在圖像平面可以被看到，插入至 `*worlds*` 的表面 (一開始爲 `nil`)會有著負的 z 座標。圖 9.4 說明了一個光線穿過圖像平面上的一點，並擊中一個球體。

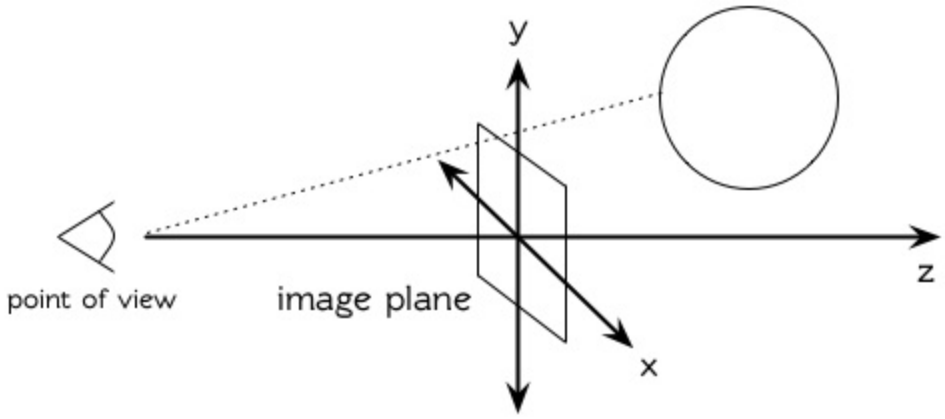

圖 **9.4:** 追蹤光線。

函數 `tracer` 接受一個路徑名稱，並寫入一張圖片至對應的檔案。圖片檔案會用一種簡單的 ASCII 稱作 PGM 的格式寫入。默認情況下，圖像會是 100x100 。我們 PGM 檔案的標頭 (headers) 會由標籤 `P2` 組成，伴隨著指定圖片寬度 (breadth)與高度 (height)的整數，初始爲 100，單位爲 pixel，以及可能的最大值 (255)。檔案剩餘的部份會由 10000 個介於 0 (黑)與 1 (白)整陣列成，代表著 100 條 100 像素的水平線。

圖片的解析度可以通過給入明確的 `res` 來調整。舉例來說，如果 `res` 是 `2` ，則同樣的圖像會被渲染成 200x200 。

圖片是一個在圖像平面 100x100 的正方形。每一個像素代表著穿過圖像平面抵達觀測點的光的數量。要找到每個像素光的數量， `tracer` 呼叫 `color-at` 。這個函數找到從觀測點至該點的向量，並呼叫 `sendray` 來追蹤這個向量回到模擬世界的軌跡； `sandray` 會返回一個數值介於 0 與 1 之間的亮度 (intensity)，之後會縮放成一個 0 至 255 的整數來顯示。

要決定一個光線的亮度， `sendray` 需要找到光是從哪個物體所反射的。要辦到這件事，我們呼叫 `first-hit` ，此函數研究在 `*world*` 裡的所有平面，並返回光線最先抵達的平面（如果有的話）。如果光沒有擊中任何東西， `sendray` 僅返回背景顏色，按慣例是 `0` (黑色)。如果光線有擊中某物的話，我們需要找出在光擊中時，有多少數量的光照在該平面。

朗伯定律 [http://zh.wikipedia.org/zh-tw/%E6%AF%94%E5%B0%94%EF%BC%8D%E6%9C%97%E4%BC%AF%E5%AE%9A%E5 告訴我們，由平面上一點所反射的光的強度，正比於該點的單位法向量 (unit normal vector) *N* (這裡是與平面垂直且長度爲一的向量)與該點至光源的單位向量 *L* 的點積 (dot-product):

\[i = N·L\]

如果光剛好照到這點， N 與 L 會重合 (coincident)，則點積會是最大值， `1` 。如果將在這時候將平面朝光轉 90 度，則 N 與 L 會垂直，則兩者點積會是 `0` 。如果光在平面後面，則點積會是負數。

在我們的程式裡，我們假設光源在觀測點 (eye)，所以 `lambert` 使用了這個規則來找到平面上某點的亮度 (illumination)，返回我們追蹤的光的單位向量與法向量的點積。

在 `sendray` 這個值會乘上平面的顏色 (即便是有好的照明，一個暗的平面還是暗的)來決定該點之後總體亮度。

爲了簡單起見，我們在模擬世界裡會只有一種物體，球體。圖 9.5 包含了與球體有關的程式碼。球體結構包含了 `surface` ，所以一個球體會有一種顏色以及 `center` 和 `radius` 。呼叫 `defsphere` 添加一個新球體至世界裡。

```
(defstruct (sphere (:include surface))
  radius center)

(defun defsphere (x y z r c)
  (let ((s (make-sphere
             :radius r
             :center (make-point :x x :y y :z z)
             :color  c)))
    (push s *world*)
    s))

(defun intersect (s pt xr yr zr)
  (funcall (typecase s (sphere #'sphere-intersect))
           s pt xr yr zr))

(defun sphere-intersect (s pt xr yr zr)
  (let* ((c (sphere-center s))
         (n (minroot (+ (sq xr) (sq yr) (sq zr))
                     (* 2 (+ (* (- (x pt) (x c)) xr)
                             (* (- (y pt) (y c)) yr)
                             (* (- (z pt) (z c)) zr)))
                     (+ (sq (- (x pt) (x c)))
                        (sq (- (y pt) (y c)))
                        (sq (- (z pt) (z c)))
                        (- (sq (sphere-radius s)))))))
    (if n
        (make-point :x  (+ (x pt) (* n xr))
                    :y  (+ (y pt) (* n yr))
                    :z  (+ (z pt) (* n zr))))))

(defun normal (s pt)
  (funcall (typecase s (sphere #'sphere-normal))
           s pt))

(defun sphere-normal (s pt)
  (let ((c (sphere-center s)))
```

```
    (unit-vector (- (x c) (x pt))
                 (- (y c) (y pt))
                 (- (z c) (z pt)))))
```

圖 **9.5** 球體。

函數 `intersect` 判斷與何種平面有關，並呼叫對應的函數。在此時只有一種， `sphere-intersect` ，但 `intersect` 是寫成可以容易擴展處理別種物體。

我們要怎麼找到一束光與一個球體的交點 (intersection)呢？光線是表示成點 \(p =〈x_0,y_0,x_0〉\) 以及單位向量 \(v =〈x_r,y_r,x_r〉\) 。每個在光上的點可以表示爲 \(p+nv\)，對於某個 *n* —— 即 \(〈x_0+nx_r,y_0+ny_r,z_0+nz_r〉\) 。光擊中球體的點的距離至中心 \(〈x_c,y_c,z_c〉\) 會等於球體的半徑 *r* 。所以在下列這個交點的方程式會成立:

\[r = \sqrt{ (x_0 + nx_r - x_c)^2 + (y_0 + ny_r - y_c)^2 + (z_0 + nz_r - z_c)^2 }\]

這會給出

\[an^2 + bn + c = 0\]

其中

\[\begin{split}a = x_r^2 + y_r^2 + z_r^2\\b = 2((x_0-x_c)x_r + (y_0-y_c)y_r + (z_0-z_c)z_r)\\c = (x_0-x_c)^2 + (y_0-y_c)^2 + (z_0-z_c)^2 - r^2\end{split}\]

要找到交點我們只需要找到這個二次方程式的根。它可能是零、一個或兩個實數根。沒有根代表光沒有擊中球體；一個根代表光與球體交於一點 (擦過 「grazing hit」)；兩個根代表光與球體交於兩點 (一點交於進入時、一點交於離開時)。在最後一個情況裡，我們想要兩個根之中較小的那個； *n* 與光離開觀測點的距離成正比，所以先擊中的會是較小的 *n* 。所以我們呼叫 `minroot` 。如果有一個根， `sphere-intersect` 返回代表該點的 \(〈x_0+nx_r,y_0+ny_r,z_0+nz_r〉\) 。

圖 9.5 的另外兩個函數， `normal` 與 `sphere-normal` 類比於 `intersect` 與 `sphere-intersect` 。要找到垂直於球體很簡單 —— 不過是從該點至球體中心的向量而已。

圖 9.6 示範了我們如何產生圖片； `ray-test` 定義了 38 個球體（不全都看的見）然後產生一張圖片，叫做 “sphere.pgm” 。

(譯註：PGM 可移植灰度圖格式，更多資訊參見 wiki [http://en.wikipedia.org/wiki/Portable_graymap])

```
(defun ray-test (&optional (res 1))
  (setf *world* nil)
  (defsphere 0 -300 -1200 200 .8)
```

```
(defsphere -80 -150 -1200 200 .7)
(defsphere 70 -100 -1200 200 .9)
(do ((x -2 (1+ x)))
    ((> x 2))
  (do ((z 2 (1+ z)))
      ((> z 7))
    (defsphere (* x 200) 300 (* z -400) 40 .75)))
(tracer (make-pathname :name "spheres.pgm") res))
```

圖 **9.6** 使用光線追蹤器

圖 9.7 是產生出來的圖片，其中 `res` 參數爲 10。

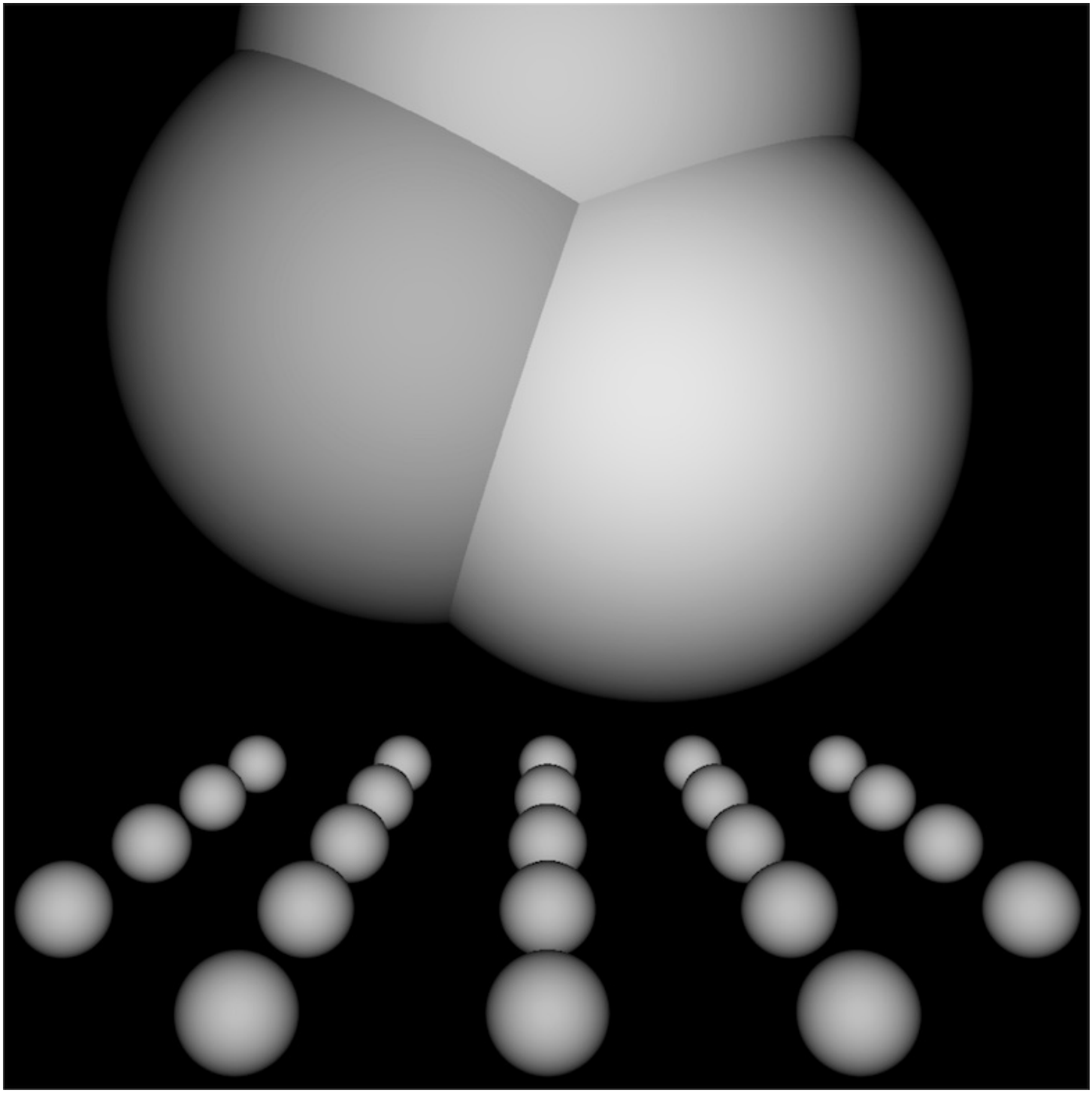

圖 **9.7:** 追蹤光線的圖

一個實際的光線追蹤器可以產生更複雜的圖片，因爲它會考慮更多，我們只考慮了單一光源至平面某一點。可能會有多個光源，每一個有不同的強度。它們通常不會在觀測點，在這個情況程式需要檢查至光源的向量是否與其他平面相交，這會在第一個相交的平面上產生陰影。將光源放置於觀測點讓我們不需要考慮這麼複雜的情況，因爲我們看不見在陰影中的任何點。

一個實際的光線追蹤器不僅追蹤光第一個擊中的平面，也會加入其它平面的反射光。一個實際的光線追蹤器會是有顏色的，並可以模型化出透明或是閃耀的平面。但基本的算法會與圖 9.3 所示範的差不多，而許多改進只需要遞迴的使用同樣的成分。

一個實際的光線追蹤器可以是高度優化的。這裡給出的程式爲了精簡寫成，甚至沒有如 Lisp 程式設計師會最佳化的那樣，就僅是一個光線追蹤器而已。僅加入型別與行內宣告 (13.3 節)就可以讓它變得兩倍以上快。

Chapter 9 總結 (Summary)

1. Common Lisp 提供整數 (integers)、比值 (ratios)、浮點數 (floating-point numbers)以及複數 (complex numbers)。
2. 數字可以被約分或轉換 (converted)，而它們的位數 (components)可以被取出。
3. 用來比較數字的謂詞可以接受任意數量的參數，以及比較下一數對 (successive pairs) ── /= 函數除外，它是用來比較所有的數對 (pairs)。
4. Common Lisp 幾乎提供你在低階科學計算機可以看到的數值函數。同樣的函數普遍可應用在多種型別的數字上。
5. Fixnum 是小至可以塞入一個字 (word)的整數。它們在必要時會悄悄但花費昂貴地轉成大數 (bignum)。Common Lisp 提供最多四種浮點數。每一個浮點表示法的限制是實現相關的 (implementation-dependent)常數。
6. 一個光線追蹤器 (ray-tracer)通過追蹤光線來產生圖像，使得每一像素回到模擬的世界。

Chapter 9 練習 (Exercises)

1. 定義一個函數，接受一個實數列表，若且唯若 (iff)它們是非遞減 (nondecreasing)順序時返回真。
2. 定義一個函數，接受一個整數 `cents` 並返回四個值，將數字用 `25-` , `10-` , `5-` , `1-` 來顯示，使用最少數量的硬幣。(譯註: `25-` 是 25 美分，以此類推)
3. 一個遙遠的星球住著兩種生物， wigglies 與 wobblies 。 Wigglies 與 wobblies 唱歌一樣厲害。每年都有一個比賽來選出十大最佳歌手。下面是過去十年的結果:

YEAR	1	2	3	4	5	6	7	8	9	10
WIGGLIES	6	5	6	4	5	5	4	5	6	5
WOBBLIES	4	5	4	6	5	5	6	5	4	5

寫一個程式來模擬這樣的比賽。你的結果實際上有建議委員會每年選出 10 個最佳歌手嗎？

4. 定義一個函數，接受 8 個表示二維空間中兩個線段端點的實數，若線段沒有相交，則返回假，或返回兩個值表示相交點的 `x` 座標與 `y` 座標。
5. 假設 `f` 是一個接受一個 (實數) 參數的函數，而 `min` 與 `max` 是有著不同正負號的非零實數，使得 `f` 對於參數 `i` 有一個根 (返回零)並滿足 `min < i < max`。定義一個函數，接受四個參數， `f` , `min` , `max` 以及 `epsilon` ，並返回一個 `i` 的近似值，準確至正負 `epsilon` 之內。
6. *Honer's method* 是一個有效率求出多項式的技巧。要找到 \(ax^3+bx^2+cx+d\) 你對 `x(x(ax+b)+c)+d` 求值。定義一個函數，接受一個或多個參數 —— x 的值伴隨著 *n* 個實數，用來表示 `(n-1)` 次方的多項式的係數 —— 並用 *Honer's method* 計算出多項式的值。

譯註: Honer's method on wiki [http://en.wikipedia.org/wiki/Horner's_method]

7. 你的 Common Lisp 實現使用了幾個位元來表示定長數？
8. 你的 Common Lisp 實現提供幾種不同的浮點數？

腳註

[1] 當 `format` 取整顯示時，它不保證會取成偶數或奇數。見 125 頁 (譯註: 7.4 節)。

第十章：宏

Lisp 程式碼是用 Lisp 物件的列表來表示。2.3 節宣稱這讓 Lisp 可以寫出可自己寫程式的程式。本章將示範如何跨越表達式與程式碼的界線。

10.1 求值 (Eval)

如何產生表達式是很直觀的：呼叫 `list` 即可。我們沒有考慮到的是，如何使 Lisp 將列表視爲程式碼。這之間缺少的一環是函數 `eval` ，它接受一個表達式，將其求值，然後返回它的值:

```
> (eval '(+ 1 2 3))
6
> (eval '(format t "Hello"))
Hello
NIL
```

如果這看起很熟悉的話，這是應該的。這就是我們一直交談的那個 `eval` 。下面這個函數實現了與頂層非常相似的東西:

```
(defun our-toplevel ()
  (do ()
      (nil)
    (format t "~%> ")
    (print (eval (read)))))
```

也是因爲這個原因，頂層也稱爲讀取—求值—打印迴圈 (read-eval-print loop, REPL)。

呼叫 `eval` 是跨越程式碼與列表界限的一種方法。但它不是一個好方法:

1. 它的效率低下： `eval` 處理的是原始列表 (raw list)，要不當下編譯它，或是用直譯器求值。兩種方法都比執行編譯過的程式來得慢許多。
2. 表達式在沒有詞法語境 (lexical context)的情況下被求值。舉例來說，如果你在一個 `let` 裡呼叫 `eval` ，傳給 `eval` 的表達式將無法參照由 `let` 所設置的變數。

有許多更好的方法 (下一節敘述)來利用產生程式碼的這個可能性。當然 `eval` 也是有用的，唯一合法的用途像是在頂層迴圈使用它。

對於程式設計師來說， `eval` 的主要價值大概是作爲 Lisp 的概念模型。我們可以想像 Lisp 是由一個長的 `cond` 表達式定義而成:

```
(defun eval (expr env)
  (cond ...
        ((eql (car expr) 'quote) (cdr expr))
        ...
        (t (apply (symbol-function (car expr))
                  (mapcar #'(lambda (x)
                              (eval x env))
                          (cdr expr))))))
```

許多表達式由預設子句 (default clause)來處理，預設子句獲得 `car` 所參照的函數，將 `cdr` 所有的參數求值，並返回將前者應用至後者的結果。 [1]

但是像 `(quote x)` 那樣的句子就不能用這樣的方式來處理，因爲 `quote` 就是爲了防止它的參數被求值而存在的。所以我們需要給 `quote` 寫一個特別的子句。這也是爲什麼本質上將其稱爲特殊運算子 (special operator): 一個需要被實現爲 `eval` 的一個特殊情況的運算子。

函數 `coerce` 與 `compile` 提供了一個類似的橋樑，讓你把列表轉成程式碼。你可以 `coerce` 一個 lambda 表達式，使其成爲函數，

```
> (coerce '(lambda (x) x) 'function)
#<Interpreted-Function BF9D96>
```

而如果你將 `nil` 作爲第一個參數傳給 `compile`，它會編譯作爲第二個參數傳入的 lambda 表達式。

```
> (compile nil '(lambda (x) (+ x 2)))
#<Compiled-Function BF55BE>
NIL
NIL
```

由於 `coerce` 與 `compile` 可接受列表作爲參數，一個程式可以在動態執行時構造新函數。但與呼叫 `eval` 比起來，這不是一個從根本解決的辦法，並且需抱有同樣的疑慮來檢視這兩個函數。

函數 `eval` , `coerce` 與 `compile` 的麻煩不是它們跨越了程式碼與列表之間的界限，而是它們在執行期做這件事。跨越界線的代價昂貴。大多數情況下，在編譯期做這件事是沒問題的，當你的程式執行時，幾乎不用成本。下一節會示範如何辦到這件事。

10.2 宏 (Macros)

寫出能寫程式的程式的最普遍方法是通過定義宏。宏是通過轉換 (transformation)而實現的運算子。你通過說明你一個呼叫應該要翻譯成什麼，來定義一個宏。這個翻譯稱爲宏

展開(macro-expansion)，宏展開由編譯器自動完成。所以宏所產生的程式碼，會變成程式的一個部分，就像你自己輸入的程式一樣。

宏通常透過呼叫 `defmacro` 來定義。一個 `defmacro` 看起來很像 `defun` 。但是與其定義一個函數呼叫應該產生的值，它定義了該怎麼翻譯出一個函數呼叫。舉例來說，一個將其參數設爲 `nil` 的宏可以定義成如下:

```
(defmacro nil! (x)
  (list 'setf x nil))
```

這定義了一個新的運算子，稱爲 `nil!` ，它接受一個參數。一個這樣形式 `(nil! a)` 的呼叫，會在求值或編譯前，被翻譯成 `(setf a nil)` 。所以如果我們輸入 `(nil! x)` 至頂層，

```
> (nil! x)
NIL
> x
NIL
```

完全等同於輸入表達式 `(setf x nil)` 。

要測試一個函數，我們呼叫它，但要測試一個宏，我們看它的展開式 (expansion)。

函數 `macroexpand-1` 接受一個宏呼叫，並產生它的展開式:

```
> (macroexpand-1 '(nil! x))
(SETF X NIL)
T
```

一個宏呼叫可以展開成另一個宏呼叫。當編譯器（或頂層）遇到一個宏呼叫時，它持續展開它，直到不可展開爲止。

理解宏的祕密是理解它們是如何被實現的。在檯面底下，它們只是轉換成表達式的函數。舉例來說，如果你傳入這個形式 `(nil! a)` 的表達式給這個函數

```
(lambda (expr)
  (apply #'(lambda (x) (list 'setf x nil))
         (cdr expr)))
```

它會返回 `(setf a nil)` 。當你使用 `defmacro` ，你定義一個類似這樣的函數。`macroexpand-1` 全部所做的事情是，當它看到一個表達式的 `car` 是宏時，將表達式傳給對應的函數。

10.3 反引號 (Backquote)

反引號讀取宏 (read-macro)使得從模版 (templates)建構列表變得有可能。反引號廣泛使用在宏定義中。一個平常的引用是鍵盤上的右引號 (apostrophe)，然而一個反引號是一個左引號。(譯註: open quote 左引號，closed quote 右引號)。它稱作“反引號”是因爲它看起來像是反過來的引號 (titled backwards)。

(譯註: 反引號是鍵盤左上方數字 1 左邊那個: `，而引號是 enter 左邊那個 ')

一個反引號單獨使用時，等於普通的引號:

```
> `(a b c)
(A B C)
```

和普通引號一樣，單一個反引號保護其參數被求值。

反引號的優點是，在一個反引號表達式裡，你可以使用 , （逗號）與 ,@ （comma-at）來重啓求值。如果你在反引號表達式裡，在某個東西前面加逗號，則它會被求值。所以我們可以使用反引號與逗號來建構列表模版:

```
> (setf a 1 b 2)
2
> `(a is ,a and b is ,b)
(A IS 1 AND B IS 2)
```

通過使用反引號取代呼叫 `list`，我們可以寫出會產生出展開式的宏。舉例來說 `nil!` 可以定義爲:

```
(defmacro nil! (x)
  `(setf ,x nil))
```

,@ 與逗號相似，但將（本來應該是列表的）參數扒開。將列表的元素插入模版來取代列表。

```
> (setf lst '(a b c))
(A B C)
> `(lst is ,lst)
(LST IS (A B C))
> `(its elements are ,@lst)
(ITS ELEMENTS ARE A B C)
```

,@ 在宏裡很有用，舉例來說，在用剩餘參數表示程式碼主體的宏。假設我們想要一個 `while` 宏，只要初始測試表達式爲真，對其主體求值:

```
> (let ((x 0))
    (while (< x 10)
       (princ x)
       (incf x)))
0123456789
NIL
```

我們可以通過使用一個剩餘參數，蒐集主體的表達式列表，來定義一個這樣的宏，接著使用 comma-at 來扒開這個列表放至展開式裡:

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

10.4 範例：快速排序法(Example: Quicksort)

圖 10.1 包含了重度依賴宏的一個範例函數 —— 一個使用快速排序演算法 λ [http://acl.readthedocs.org/en/latest/zhTW/notes.html#notes-164] 來排序向量的函數。這個函數的工作方式如下:

```
(defun quicksort (vec l r)
  (let ((i l)
        (j r)
        (p (svref vec (round (+ l r) 2))))     ; 1
    (while (<= i j)                            ; 2
      (while (< (svref vec i) p) (incf i))
      (while (> (svref vec j) p) (decf j))
      (when (<= i j)
        (rotatef (svref vec i) (svref vec j))
        (incf i)
        (decf j)))
    (if (>= (- j l) 1) (quicksort vec l j))     ; 3
    (if (>= (- r i) 1) (quicksort vec i r)))
  vec)
```

圖 **10.1** 快速排序。

1. 開始你通過選擇某個元素作爲主鍵（ *pivot* ）。許多實現選擇要被排序的序列中間元素。
2. 接著你分割（partition）向量，持續交換元素，直到所有主鍵左邊的元素小於主鍵，右邊的元素大於主鍵。
3. 最後，如果左右分割之一有兩個或更多元素時，你遞迴地應用這個算法至向量的那些分割上。

每一次遞迴時，分割越變越小，直到向量完整排序爲止。

在圖 10.1 的實現裡，接受一個向量以及標記欲排序範圍的兩個整數。這個範圍當下的中間元素被選爲主鍵 (p)。接著從左右兩端開始產生分割，並將左邊太大或右邊太小的元素交換過來。(將兩個參數傳給 rotatef 函數，交換它們的值。)最後，如果一個分割含有多個元素時，用同樣的流程來排序它們。

除了我們前一節定義的 while 宏之外，圖 10.1 也用了內建的 when , incf , decf 以及 rotatef 宏。使用這些宏使程式看起來更加簡潔與清晰。

10.5 設計宏 (Macro Design)

撰寫宏是一種獨特的程式設計，它有著獨一無二的目標與問題。能夠改變編譯器所看到的東西，就像是能夠重寫它一樣。所以當你開始撰寫宏時，你需要像語言設計者一樣思考。

本節快速給出宏所牽涉問題的概要，以及解決它們的技巧。作爲一個例子，我們會定義一個稱爲 ntimes 的宏，它接受一個數字 *n* 並對其主體求值 *n* 次。

```
> (ntimes 10
    (princ "."))
..........
NIL
```

下面是一個不正確的 ntimes 定義，說明了宏設計中的某些議題:

```
(defmacro ntimes (n &rest body)
  `(do ((x 0 (+ x 1)))
       ((>= x ,n))
     ,@body))
```

這個定義第一眼看起來可能沒問題。在上面這個情況，它會如預期的工作。但實際上它在兩個方面壞掉了。

一個宏設計者需要考慮的問題之一是，無意的變數捕捉 (variable capture)。這發生在當一個在宏展開式裡用到的變數，恰巧與展開式即將插入的語境裡，有使用同樣名字作爲變數的情況。不正確的 ntimes 定義創造了一個變數 x 。所以如果這個宏在已經有 x 作爲名字的地方被呼叫時，它可能無法做到我們所預期的:

```
> (let ((x 10))
    (ntimes 5
       (setf x (+ x 1)))
    x)
10
```

如果 ntimes 如我們預期般的執行，這個表達式應該會對 x 遞增五次，最後返回 15 。但因爲宏展開剛好使用 x 作爲迭代變數， setf 表達式遞增那個 x ，而不是我們要遞增的那個。一旦宏呼叫被展開，前述的展開式變成:

```
> (let ((x 10))
    (do ((x 0 (+ x 1)))
        ((>= x 5))
      (setf x (+ x 1)))
    x)
```

最普遍的解法是不要使用任何可能會被捕捉的一般符號。取而代之的我們使用 gensym (8.4 小節)。因爲 read 函數 intern 每個它見到的符號，所以在一個程式裡，沒有可能會有任何符號會 eql gensym。如果我們使用 gensym 而不是 x 來重寫 ntimes 的定義，至少對於變數捕捉來說，它是安全的:

```
(defmacro ntimes (n &rest body)
  (let ((g (gensym)))
    `(do ((,g 0 (+ ,g 1)))
         ((>= ,g ,n))
       ,@body)))
```

但這個宏在另一問題上仍有疑慮: 多重求值 (multiple evaluation)。因爲第一個參數被直接插入 do 表達式，它會在每次迭代時被求值。當第一個參數是有副作用的表達式，這個錯誤非常清楚地表現出來:

```
> (let ((v 10))
    (ntimes (setf v (- v 1))
      (princ ".")))
.....
NIL
```

由於 v 一開始是 10 ， 而 setf 返回其第二個參數的值，應該印出九個句點。實際上它只印出五個。

如果我們看看宏呼叫所展開的表達式，就可以知道爲什麼:

```
> (let ((v 10))
    (do ((#:g1 0 (+ #:g1 1)))
        ((>= #:g1 (setf v (- v 1))))
      (princ ".")))
```

每次迭代我們不是把迭代變數 (gensym 通常印出前面有 #: 的符號)與 9 比較，而是與每次求值時會遞減的表達式比較。這如同每次我們查看地平線時，地平線都越來越近。

避免非預期的多重求值的方法是設置一個變數，在任何迭代前將其設爲有疑惑的那個表

達式。這通常牽扯到另一個 gensym:

```
(defmacro ntimes (n &rest body)
  (let ((g (gensym))
        (h (gensym)))
    `(let ((,h ,n))
       (do ((,g 0 (+ ,g 1)))
           ((>= ,g ,h))
         ,@body))))
```

終於，這是一個 ntimes 的正確定義。

非預期的變數捕捉與多重求值是折磨宏的主要問題，但不只有這些問題而已。有經驗後，要避免這樣的錯誤與避免更熟悉的錯誤一樣簡單，比如除以零的錯誤。

你的 Common Lisp 實現是一個學習更多有關宏的好地方。藉由呼叫展開至內建宏，你可以理解它們是怎麼寫的。下面是大多數實現對於一個 cond 表達式會產生的展開式:

```
> (pprint (macroexpand-1 '(cond (a b)
                                (c d e)
                                (t f))))
(IF A
    B
    (IF C
        (PROGN D E)
        F))
```

函數 pprint 印出像程式碼一樣縮排的表達式，這在檢視宏展開式時特別有用。

10.6 通用化參照 (Generalized Reference)

由於一個宏呼叫可以直接在它出現的地方展開成程式碼，任何展開爲 setf 表達式的宏呼叫都可以作爲 setf 表達式的第一個參數。 舉例來說，如果我們定義一個 car 的同義詞，

```
(defmacro cah (lst) `(car ,lst))
```

然後因爲一個 car 呼叫可以是 setf 的第一個參數，而 cah 一樣可以:

```
> (let ((x (list 'a 'b 'c)))
    (setf (cah x) 44)
    x)
(44 B C)
```

撰寫一個展開成一個 setf 表達式的宏是另一個問題，是一個比原先看起來更爲困難的

問題。看起來也許你可以這樣實現 incf，只要

```
(defmacro incf (x &optional (y 1)) ; wrong
  `(setf ,x (+ ,x ,y)))
```

但這是行不通的。這兩個表達式不相等:

```
(setf (car (push 1 lst)) (1+ (car (push 1 lst))))

(incf (car (push 1 lst)))
```

如果 lst 是 nil 的話，第二個表達式會設成 (2)，但第一個表達式會設成 (1 2)。

Common Lisp 提供了 define-modify-macro 作爲寫出對於 setf 限制類別的宏的一種方法 它接受三個參數: 宏的名字，額外的參數 (隱含第一個參數 place)，以及產生出 place 新數值的函數名。所以我們可以將 incf 定義爲

```
(define-modify-macro our-incf (&optional (y 1)) +)
```

另一版將元素推至列表尾端的 push 可寫成:

```
(define-modify-macro append1f (val)
  (lambda (lst val) (append lst (list val))))
```

後者會如下工作:

```
> (let ((lst '(a b c)))
    (append1f lst 'd)
    lst)
(A B C D)
```

順道一提， push 與 pop 都不能定義爲 modify-macros，前者因爲 place 不是其第一個參數，而後者因爲其返回值不是更改後的物件。

10.7 範例：實用的宏函數 (Example: Macro Utilities)

6.4 節介紹了實用函數 (utility)的概念，一種像是構造 Lisp 的通用運算子。我們可以使用宏來定義不能寫作函數的實用函數。我們已經見過幾個例子: nil! , ntimes 以及 while ，全部都需要寫成宏，因爲它們全都需要某種控制參數求值的方法。本節給出更多你可以使用宏寫出的多種實用函數。圖 10.2 挑選了幾個實踐中證實值得寫的實用函數。

```
(defmacro for (var start stop &body body)
  (let ((gstop (gensym)))
```

```
    `(do ((,var ,start (1+ ,var))
          (,gstop ,stop))
         ((> ,var ,gstop))
       ,@body)))

(defmacro in (obj &rest choices)
  (let ((insym (gensym)))
    `(let ((,insym ,obj))
       (or ,@(mapcar #'(lambda (c) `(eql ,insym ,c))
                     choices)))))

(defmacro random-choice (&rest exprs)
  `(case (random ,(length exprs))
     ,@(let ((key -1))
         (mapcar #'(lambda (expr)
                     `(,(incf key) ,expr))
                 exprs))))

(defmacro avg (&rest args)
  `(/ (+ ,@args) ,(length args)))

(defmacro with-gensyms (syms &body body)
  `(let ,(mapcar #'(lambda (s)
                     `(,s (gensym)))
                 syms)
     ,@body))

(defmacro aif (test then &optional else)
  `(let ((it ,test))
     (if it ,then ,else)))
```

圖 **10.2:** 實用宏函數

第一個 `for`，設計上與 `while` 相似 (164 頁，譯註: 10.3 節)。它是給需要使用一個綁定至一個值的範圍的新變數來對主體求值的迴圈:

```
> (for x 1 8
          (princ x))
12345678
NIL
```

這比寫出等效的 `do` 來得省事，

```
(do ((x 1 (+ x 1)))
    ((> x 8))
  (princ x))
```

這非常接近實際的展開式:

```
(do ((x 1 (1+ x))
```

```
      (#:g1 8))
    ((> x #:g1))
  (princ x))
```

宏需要引入一個額外的變數來持有標記範圍 (range)結束的值。 上面在例子裡的 `8` 也可是個函數呼叫，這樣我們就不需要求值好幾次。額外的變數需要是一個 gensym ，爲了避免非預期的變數捕捉。

圖 10.2 的第二個宏 `in` ，若其第一個參數 `eql` 任何自己其他的參數時，返回真。表達式我們可以寫成:

```
(in (car expr) '+ '- '*)
```

我們可以改寫成:

```
(let ((op (car expr)))
  (or (eql op '+)
      (eql op '-)
      (eql op '*)))
```

確實，第一個表達式展開後像是第二個，除了變數 `op` 被一個 gensym 取代了。

下一個例子 `random-choice` ，隨機選取一個參數求值。在 74 頁 (譯註: 第 4 章的圖 4.6) 我們需要隨機在兩者之間選擇。 `random-choice` 宏實現了通用的解法。一個像是這樣的呼叫:

```
(random-choice (turn-left) (turn-right))
```

會被展開爲:

```
(case (random 2)
  (0 (turn-left))
  (1 (turn-right)))
```

下一個宏 `with-gensyms` 主要預期用在宏主體裡。它不尋常，特別是在特定應用中的宏，需要 gensym 幾個變數。有了這個宏，與其

```
(let ((x (gensym)) (y (gensym)) (z (gensym)))
       ...)
```

我們可以寫成

```
(with-gensyms (x y z)
       ...)
```

到目前爲止，圖 10.2 定義的宏，沒有一個可以定義成函數。作爲一個規則，寫成宏是因爲你不能將它寫成函數。但這個規則有幾個例外。有時候你或許想要定義一個運算子來作爲宏，好讓它在編譯期完成它的工作。宏 avg 返回其參數的平均值，

```
> (avg 2 4 8)
14/3
```

是一個這種例子的宏。我們可以將 avg 寫成函數，

```
(defun avg (&rest args)
  (/ (apply #'+ args) (length args)))
```

但它會需要在執行期找出參數的數量。只要我們願意放棄應用 avg ，爲什麼不在編譯期呼叫 length 呢？

圖 10.2 的最後一個宏是 aif ，它在此作爲一個故意變數捕捉的例子。它讓我們可以使用變數 it 來參照到一個條件式裡的測試參數所返回的值。也就是說，與其寫成

```
(let ((val (calculate-something)))
  (if val
      (1+ val)
      0))
```

我們可以寫成

```
(aif (calculate-something)
     (1+ it)
     0)
```

小心使用 (*Use judiciously*)，預期的變數捕捉可以是一個無價的技巧。Common Lisp 本身在多處使用它: 舉例來說 next-method-p 與 call-next-method 皆依賴於變數捕捉。

像這些宏明確示範了爲何要撰寫替你寫程式的程式。一旦你定義了 for ，你就不需要寫整個 do 表達式。值得寫一個宏只爲了節省打字嗎？非常值得。節省打字是程式設計的全部；一個編譯器的目的便是替你省下使用機械語言輸入程式的時間。而宏允許你將同樣的優點帶到特定的應用裡，就像高階語言帶給程式語言一般。通過審慎的使用宏，你也許可以使你的程式比起原來大幅度地精簡，並使程式更顯著地容易閱讀、撰寫及維護。

如果仍對此懷疑，考慮看看如果你沒有使用任何內建宏時，程式看起來會是怎麼樣。所有宏產生的展開式，你會需要用手產生。你也可以將這個問題用在另一方面。當你在撰寫一個程式時，捫心自問，我需要撰寫宏展開式嗎？如果是的話，宏所產生的展開式就是你需要寫的東西。

10.8 源自 Lisp (On Lisp)

現在宏已經介紹過了，我們看過更多的 Lisp 是由超乎我們想像的 Lisp 寫成。許多不是函數的 Common Lisp 運算子是宏，而他們全部用 Lisp 寫成的。只有二十五個 Common Lisp 內建的運算子是特殊運算子。

John Foderaro [http://www.franz.com/about/bios/jkf.lhtml] 將 Lisp 稱爲“可程式的程式語言。” λ [http://acl.readthedocs.org/en/latest/zhTW/notes.html#notes-173] 通過撰寫你自己的函數與宏，你將 Lisp 變成任何你想要的語言。 (我們會在 17 章看到這個可能性的圖形化示範)無論你的程式適合何種形式，你確信你可以將 Lisp 塑造成適合它的語言。

宏是這個靈活性的主要成分之一。它們允許你將 Lisp 變得完全認不出來，但仍然用一種有原則且高效的方法來實作。在 Lisp 社區裡，宏是個越來越感興趣的主題。可以使用宏辦到驚人之事是很清楚的，但更確信的是宏背後還有更多需要被探索。如果你想的話，可以通過你來發現。Lisp 永遠將進化放在程式設計師手裡。這是它爲什麼存活的原因。

Chapter 10 總結 (Summary)

1. 呼叫 `eval` 是讓 Lisp 將列表視爲程式碼的一種方法，但這是不必要而且效率低落的。
2. 你通過敘說一個呼叫會展開成什麼來定義一個宏。檯面底下，宏只是返回表達式的函數。
3. 一個使用反引號定義的主體看起來像它會產生出的展開式 (expansion)。
4. 宏設計者需要注意變數捕捉及多重求值。宏可以通過漂亮印出 (pretty-printing)來測試它們的展開式。
5. 多重求值是大多數展開成 `setf` 表達式的問題。
6. 宏比函數來得靈活，可以用來定義許多實用函數。你甚至可以使用變數捕捉來獲得好處。
7. Lisp 存活的原因是它將進化交給程式設計師的雙手。宏是使其可能的部分原因之一。

Chapter 10 練習 (Exercises)

1. 如果 `x` 是 `a`， `y` 是 `b` 以及 `z` 是 `(c d)`，寫出反引用表達式僅包含產生下列結果之一的變數:

```
(a) ((C D) A Z)

(b) (X B C D)
```

```
(c) ((C D A) Z)
```

2. 使用 `cond` 來定義 `if` 。
3. 定義一個宏，接受一個數字 *n* ，伴隨著一個或多個表達式，並返回第 *n* 個表達式的值:

```
> (let ((n 2))
    (nth-expr n (/ 1 0) (+ 1 2) (/ 1 0)))
3
```

4. 定義 `ntimes` (167 頁，譯註: 10.5 節)使其展開成一個 (區域)遞迴函數，而不是一個 `do` 表達式。
5. 定義一個宏 `n-of` ，接受一個數字 *n* 與一個表達式，返回一個 *n* 個漸進值:

```
> (let ((i 0) (n 4))
    (n-of n (incf i)))
(1 2 3 4)
```

6. 定義一個宏，接受一變數列表以及一個程式碼主體，並確保變數在程式碼主體被求值後恢復 (revert)到原本的數值。
7. 下面這個 `push` 的定義哪裡錯誤?

```
(defmacro push (obj lst)
  `(setf ,lst (cons ,obj ,lst)))
```

舉出一個不會與實際 `push` 做一樣事情的函數呼叫例子。

8. 定義一個將其參數翻倍的宏:

```
> (let ((x 1))
    (double x)
    x)
2
```

腳註

[1] 要真的複製一個 Lisp 的話， `eval` 會需要接受第二個參數 (這裡的 `env`) 來表示詞法環境 (lexical enviroment)。這個模型的 `eval` 是不正確的，因爲它在對參數求值前就取出函數，然而 Common Lisp 故意沒有特別指出這兩個操作的順序。

第十一章：Common Lisp 物件系統

Common Lisp 物件系統，或稱 CLOS，是一組用來實作物件導向程式設計的運算集。由於它們有著相同的歷史，通常將這些運算子視爲一個群組。 λ [http://acl.readthedocs.org/en/latest/zhTW/notes.html#notes-176] 技術上來說，它們與其他部分的 Common Lisp 沒什麼大不同： `defmethod` 和 `defun` 一樣，都是整合在語言中的一部分。

11.1 物件導向程式設計 Object-Oriented Programming

物件導向程式設計代表程式組織方式的改變。這個改變跟已經發生過的處理器運算處理能力分佈的變化雷同。在 1970 年代，一個多用戶的計算機系統，有大量的啞終端 [http://zh.wikipedia.org/wiki/%E5%93%91%E7%BB%88%E7%AB%AF](dumb terminal)連接到一個或兩個大型機。現在更可能是用大量相互通過網路連接的工作站來表示。系統的運算處理能力，現在分佈至個體用戶上，而不是集中在一臺大型的計算機上。

物件導向程式設計所帶來的變革與上例非常類似，前者打破了傳統程式的組織方式。不再讓單一的程式去操作那些資料，而是告訴資料自己該做什麼，程式隱含在這些新的資料“物件”的交互過程之中。

舉例來說，假設我們要算出一個二維圖形的面積。一個辦法是寫一個單獨的函數，讓它檢查其參數的型別，然後視型別做處理，如圖 11.1 所示。

```
(defstruct rectangle
  height width)

(defstruct circle
  radius)

(defun area (x)
  (cond ((rectangle-p x)
         (* (rectangle-height x) (rectangle-width x)))
        ((circle-p x)
         (* pi (expt (circle-radius x) 2)))))

> (let ((r (make-rectangle)))
    (setf (rectangle-height r) 2
          (rectangle-width r) 3)
    (area r))
6
```

圖 **11.1:** 使用結構及函數來計算面積

使用 CLOS 我們可以寫出一個等效的程式，如圖 11.2 所示。在物件導向模型裡，我們的程式被拆成數個獨一無二的方法，每個方法爲某些特定型別的參數而生。圖 11.2 中的兩個方法，隱性地定義了一個與圖 11.1 相似作用的 `area` 函數，當我們呼叫 `area` 時，Lisp 檢查參數的型別，並呼叫相對應的方法。

```
(defclass rectangle ()
  (height width))

(defclass circle ()
  (radius))

(defmethod area ((x rectangle))
  (* (slot-value x 'height) (slot-value x 'width)))

(defmethod area ((x circle))
  (* pi (expt (slot-value x 'radius) 2)))

> (let ((r (make-instance 'rectangle)))
    (setf (slot-value r 'height) 2
          (slot-value r 'width) 3)
    (area r))
6
```

圖 **11.2:** 使用型別與方法來計算面積

通過這種方式，我們將函數拆成獨一無二的方法，面向物件暗指繼承 (*inheritance*) ——槽（slot）與方法（method）皆有繼承。在圖 11.2 中，作爲第二個參數傳給 `defclass` 的空列表列出了所有基類。假設我們要定義一個新類，上色的圓形 (colored-circle)，則上色的圓形有兩個基類， `colored` 與 `circle`：

```
(defclass colored ()
  (color))

(defclass colored-circle (circle colored)
  ())
```

當我們創造 `colored-circle` 類的實體 (instance)時，我們會看到兩個繼承：

1. `colored-circle` 的實體會有兩個槽：從 `circle` 類繼承而來的 `radius` 以及從 `colored` 類繼承而來的 `color`。
2. 由於沒有特別爲 `colored-circle` 定義的 `area` 方法存在，若我們對 `colored-circle` 實體呼叫 `area`，我們會獲得替 `circle` 類所定義的 `area` 方法。

從實踐層面來看，物件導向程式設計代表著以方法、類、實體以及繼承來組織程式。爲什麼你會想這麼組織程式？面向物件方法的主張之一說這樣使得程式更容易改動。如果我們想要改變 `ob` 類物件所顯示的方式，我們只需要改動 `ob` 類的 `display` 方法。如果我

們希望創建一個新的類，大致上與 ob 相同，只有某些方面不同，我們可以創建一個 ob 類的子類。在這個子類裡，我們僅改動我們想要的屬性，其他所有的屬性會從 ob 類默認繼承得到。要是我們只是想讓某個 ob 物件和其他的 ob 物件不一樣，我們可以新建一個 ob 物件，直接修改這個物件的屬性即可。若是當時的程式寫的很講究，我們甚至不需要看程式中其他的程式碼一眼，就可以完成種種的改動。 λ [http://acl.readthedocs.org/en/latest/zhTW/notes.html#notes-178]

11.2 類與實體 (Class and Instances)

在 4.6 節時，我們看過了創建結構的兩個步驟：我們呼叫 defstruct 來設計一個結構的形式，接著通過一個像是 make-point 這樣特定的函數來創建結構。創建實體 (instances) 同樣需要兩個類似的步驟。首先我們使用 defclass 來定義一個類別 (Class):

```
(defclass circle ()
  (radius center))
```

這個定義說明了 circle 類別的實體會有兩個槽 (*slot*)，分別名爲 radius 與 center （槽類比於結構裡的欄位 「field」）。

要創建這個類的實體，我們呼叫通用的 make-instance 函數，而不是呼叫一個特定的函數，傳入的第一個參數爲類別名稱：

```
> (setf c (make-instance 'circle))
#<CIRCLE #XC27496>
```

要給這個實體的槽賦值，我們可以使用 setf 搭配 slot-value：

```
> (setf (slot-value c 'radius) 1)
1
```

與結構的欄位類似，未初始化的槽的值是未定義的 (undefined)。

11.3 槽的屬性 (Slot Properties)

傳給 defclass 的第三個參數必須是一個槽定義的列表。如上例所示，最簡單的槽定義是一個表示其名稱的符號。在一般情況下，一個槽定義可以是一個列表，第一個是槽的名稱，伴隨著一個或多個屬性 (property)。屬性像關鍵字參數那樣指定。

通過替一個槽定義一個存取器 (accessor)，我們隱式地定義了一個可以引用到槽的函數，使我們不需要再呼叫 slot-value 函數。如果我們如下更新我們的 circle 類定義，

```
(defclass circle ()
  ((radius :accessor circle-radius)
   (center :accessor circle-center)))
```

那我們能夠分別通過 circle-radius 及 circle-center 來引用槽：

```
> (setf c (make-instance 'circle))
#<CIRCLE #XC5C726>

> (setf (circle-radius c) 1)
1

> (circle-radius c)
1
```

通過指定一個 :writer 或是一個 :reader ， 而不是 :accessor ， 我們可以獲得存取器的寫入或讀取行爲。

要指定一個槽的預設值，我們可以給入一個 :initform 參數。若我們想要在 make-instance 呼叫期間就將槽初始化，我們可以用 :initarg 定義一個參數名。 [1] 加入剛剛所說的兩件事，現在我們的類定義變成：

```
(defclass circle ()
  ((radius :accessor circle-radius
           :initarg :radius
           :initform 1)
   (center :accessor circle-center
           :initarg :center
           :initform (cons 0 0))))
```

現在當我們創建一個 circle 類的實體時，我們可以使用關鍵字參數 :initarg 給槽賦值，或是將槽的值設爲 :initform 所指定的預設值。

```
> (setf c (make-instance 'circle :radius 3))
#<CIRCLE #XC2DE0E>
> (circle-radius c)
3
> (circle-center c)
(0 . 0)
```

注意 initarg 的優先序比 initform 要高。

我們可以指定某些槽是共享的 —— 也就是每個產生出來的實體，共享槽的值都會是一樣的。我們通過宣告槽擁有 :allocation :class 來辦到此事。（另一個辦法是讓一個槽有 :allocation :instance ，但由於這是預設設置，不需要特別再宣告一次。）當我們在一個實體中，改變了共享槽的值，則其它實體共享槽也會獲得相同的值。所以我們會

想要使用共享槽來保存所有實體都有的相同屬性。

舉例來說，假設我們想要模擬一羣成人小報 (a flock of tabloids)的行爲。（譯註：可以看看什麼是 tabloids [http://tinyurl.com/9n4dckk]。）在我們的模擬中，我們想要能夠表示一個事實，也就是當一家小報採用一個頭條時，其它小報也會跟進的這個行爲。我們可以通過讓所有的實體共享一個槽來實現。若 `tabloid` 類別像下面這樣定義，

```
(defclass tabloid ()
  ((top-story :accessor tabloid-story
              :allocation :class)))
```

那麼如果我們創立兩家小報，無論一家的頭條是什麼，另一家的頭條也會是一樣的：

```
> (setf daily-blab (make-instance 'tabloid)
        unsolicited-mail (make-instance 'tabloid))
#<TABLOID #x302000EFE5BD>
> (setf (tabloid-story daily-blab) 'adultery-of-senator)
ADULTERY-OF-SENATOR
> (tabloid-story unsolicited-mail)
ADULTERY-OF-SENATOR
```

譯註： ADULTERY-OF-SENATOR 參議員的性醜聞。

若有給入 `:documentation` 屬性的話，用來作爲 `slot` 的文檔字串。通過指定一個 `:type`，你保證一個槽裡只會有這種型別的元素。型別宣告會在 13.3 節講解。

11.4 基類 (Superclasses)

`defclass` 接受的第二個參數是一個列出其基類的列表。一個類別繼承了所有基類槽的聯集。所以要是我們將 `screen-circle` 定義成 `circle` 與 `graphic` 的子類，

```
(defclass graphic ()
  ((color :accessor graphic-color :initarg :color)
   (visible :accessor graphic-visible :initarg :visible
            :initform t)))

(defclass screen-circle (circle graphic) ())
```

則 `screen-circle` 的實體會有四個槽，分別從兩個基類繼承而來。一個類別不需要自己創建任何新槽； `screen-circle` 的存在，只是爲了提供一個可創建同時從 `circle` 及 `graphic` 繼承的實體。

存取器及 `:initargs` 參數可以用在 `screen-circle` 的實體，就如同它們也可以用在 `circle` 或 `graphic` 類別那般：

```
> (graphic-color (make-instance 'screen-circle
                                :color 'red :radius 3))
RED
```

我們可以使每一個 screen-circle 有某種預設的顏色，通過在 defclass 裡這個槽指定一個 :initform：

```
(defclass screen-circle (circle graphic)
  ((color :initform 'purple)))
```

現在 screen-circle 的實體預設會是紫色的：

```
> (graphic-color (make-instance 'screen-circle))
PURPLE
```

11.5 優先序 (Precedence)

我們已經看過類別是怎樣能有多個基類了。當一個實體的方法同時屬於這個實體所屬的幾個類時，Lisp 需要某種方式來決定要使用哪個方法。優先序的重點在於確保這一切是以一種直觀的方式發生的。

每一個類別，都有一個優先序列表：一個將自身及自身的基類從最具體到最不具體所排序的列表。在目前看過的例子中，優先序還不是需要討論的議題，但在更大的程式裡，它會是一個需要考慮的議題。

以下是一個更複雜的類別層級：

```
(defclass sculpture () (height width depth))

(defclass statue (sclpture) (subject))

(defclass metalwork () (metal-type))

(defclass casting (metalwork) ())

(defclass cast-statue (statue casting) ())
```

圖 11.3 包含了一個表示 cast-statue 類別及其基類的網路。

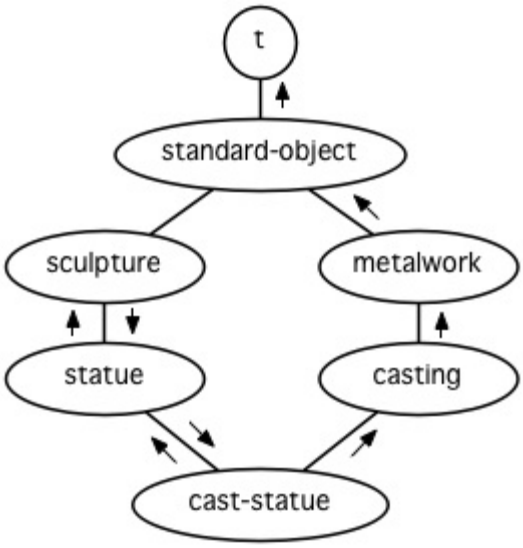

圖 **11.3:** 類別層級

要替一個類別建構一個這樣的網路，從最底層用一個節點表示該類別開始。接著替類別最近的基類畫上節點，其順序根據 `defclass` 呼叫裡的順序由左至右畫，再來給每個節點重複這個過程，直到你抵達一個類別，這個類別最近的基類是 `standard-object` ——即傳給 `defclass` 的第二個參數爲 `()` 的類別。最後從這些類別往上建立連結，到表示 `standard-object` 節點爲止，接著往上加一個表示類別 `t` 的節點與一個連結。結果會是一個網路，最頂與最下層各爲一個點，如圖 11.3 所示。

一個類別的優先序列表可以通過如下步驟，遍歷對應的網路計算出來：

1. 從網路的底部開始。
2. 往上走，遇到未探索的分支永遠選最左邊。
3. 如果你將進入一個節點，你發現此節點右邊也有一條路同樣進入該節點時，則從該節點退後，重走剛剛的老路，直到回到一個節點，這個節點上有尚未探索的路徑。接著返回步驟 2。
4. 當你抵達表示 `t` 的節點時，遍歷就結束了。你第一次進入每個節點的順序就決定了節點在優先序列表的順序。

這個定義的結果之一（實際上講的是規則 3）在優先序列表裡，類別不會在其子類別出現前出現。

圖 11.3 的箭頭示範了一個網路是如何遍歷的。由這個圖所決定出的優先序列表爲：`cast-statue` , `statue` , `sculpture` , `casting` , `metalwork` , `standard-object` , `t` 。有時候會用 *specific* 這個詞，作爲在一個給定的優先序列表中來引用類別的位置的速記法。優先序列表從最高優先序排序至最低優先序。

優先序的主要目的是，當一個通用函數 (generic function)被呼叫時，決定要用哪個方

法。這個過程在下一節講述。另一個優先序重要的地方是，當一個槽從多個基類繼承時。408 頁的備註解釋了當這情況發生時的應用規則。 λ
[http://acl.readthedocs.org/en/latest/zhTW/notes.html#notes-183]

11.6 通用函數 (Generic Functions)

一個通用函數 (generic function) 是由一個或多個方法組成的一個函數。方法可用 `defmethod` 來定義，與 `defun` 的定義形式類似：

```
(defmethod combine (x y)
  (list x y))
```

現在 `combine` 有一個方法。若我們在此時呼叫 `combine` ，我們會獲得由傳入的兩個參數所組成的一個列表：

```
> (combine 'a 'b)
(A B)
```

到現在我們還沒有做任何一般函數做不到的事情。一個通用函數不尋常的地方是，我們可以繼續替它加入新的方法。

首先，我們定義一些可以讓新的方法引用的類別：

```
(defclass stuff () ((name :accessor name :initarg :name)))
(defclass ice-cream (stuff) ())
(defclass topping (stuff) ())
```

這裡定義了三個類別： `stuff` ，只是一個有名字的東西，而 `ice-cream` 與 `topping` 是 `stuff` 的子類。

現在下面是替 `combine` 定義的第二個方法：

```
(defmethod combine ((ic ice-cream) (top topping))
  (format nil "~A ice-cream with ~A topping."
          (name ic)
          (name top)))
```

在這次 `defmethod` 的呼叫中，參數被特化了 (*specialized*)：每個出現在列表裡的參數都有一個類別的名字。一個方法的特化指出它是應用至何種類別的參數。我們剛定義的方法僅能在傳給 `combine` 的參數分別是 `ice-cream` 與 `topping` 的實體時。

而當一個通用函數被呼叫時， Lisp 是怎麼決定要用哪個方法的？ Lisp 會使用參數的類別與參數的特化匹配且優先序最高的方法。這表示若我們用 `ice-cream` 實體與 `topping`

實體去呼叫 combine 方法，我們會得到我們剛剛定義的方法：

```
> (combine (make-instance 'ice-cream :name 'fig)
           (make-instance 'topping :name 'treacle))
"FIG ice-cream with TREACLE topping"
```

但使用其他參數時，我們會得到我們第一次定義的方法：

```
> (combine 23 'skiddoo)
(23 SKIDDOO)
```

因爲第一個方法的兩個參數皆沒有特化，它永遠只有最低優先權，並永遠是最後一個呼叫的方法。一個未特化的方法是一個安全手段，就像 case 表達式中的 otherwise 子句。

一個方法中，任何參數的組合都可以特化。在這個方法裡，只有第一個參數被特化了：

```
(defmethod combine ((ic ice-cream) x)
  (format nil "~A ice-cream with ~A."
          (name ic)
          x))
```

若我們用一個 ice-cream 的實體以及一個 topping 的實體來呼叫 combine ，我們仍然得到特化兩個參數的方法，因爲它是最具體的那個：

```
> (combine (make-instance 'ice-cream :name 'grape)
           (make-instance 'topping :name 'marshmallow))
"GRAPE ice-cream with MARSHMALLOW topping"
```

然而若第一個參數是 ice-cream 而第二個參數不是 topping 的實體的話，我們會得到剛剛上面所定義的那個方法：

```
> (combine (make-instance 'ice-cream :name 'clam)
           'reluctance)
"CLAM ice-cream with RELUCTANCE"
```

當一個通用函數被呼叫時，參數決定了一個或多個可用的方法 (*applicable* methods)。如果在呼叫中的參數在參數的特化約定內，我們說一個方法是可用的。

如果沒有可用的方法，我們會得到一個錯誤。如果只有一個，它會被呼叫。如果多於一個，最具體的會被呼叫。最具體可用的方法是由呼叫傳入參數所屬類別的優先序所決定的。由左往右審視參數。如果有一個可用方法的第一個參數，此參數特化給某個類，其類的優先序高於其它可用方法的第一個參數，則此方法就是最具體的可用方法。平手時比較第二個參數，以此類推。 [2]

在前面的例子裡，很容易看出哪個是最具體的可用方法，因爲所有的物件都是單繼承的。一個 ice-cream 的實體是，按順序來， ice-cream ， stuff ， standard-object ，以及 t 類別的成員。

方法不需要在由 defclass 定義的類別層級來做特化。他們也可以替型別做特化（更精準的說，可以反映出型別的類別）。以下是一個給 combine 用的方法，對數字做了特化：

```
(defmethod combine ((x number) (y number))
  (+ x y))
```

方法甚至可以對單一的物件做特化，用 eql 來決定：

```
(defmethod combine ((x (eql 'powder)) (y (eql 'spark)))
  'boom)
```

單一物件特化的優先序比類別特化來得高。

方法可以像一般 Common Lisp 函數一樣有複雜的參數列表，但所有組成通用函數方法的參數列表必須是一致的 (*congruent*)。參數的數量必須一致，同樣數量的選擇性參數（如果有的話），要嘛一起使用 &rest 或是 &key ，會都不要用。下面的參數列表對是全部一致的，

```
(x)              (a)
(x &optional y)  (a &optional b)
(x y &rest z)    (a b &key c)
(x y &key z)     (a b &key c d)
```

而下列的參數列表對不是一致的：

```
(x)              (a b)
(x &optional y)  (a &optional b c)
(x &optional y)  (a &rest b)
(x &key x y)     (a)
```

只有必要參數可以被特化。所以每個方法都可以通過名字及必要參數的特化獨一無二地識別出來。如果我們定義另一個方法，有著同樣的修飾符及特化，它會覆寫掉原先的。所以通過說明

```
(defmethod combine ((x (eql 'powder)) (y (eql 'spark)))
  'kaboom)
```

我們重定義了當 combine 方法的參數是 powder 與 spark 時， combine 方法幹了什麼事

兒。

11.7 輔助方法 (Auxiliary Methods)

方法可以透過如 `:before`， `:after` 以及 `:around` 等輔助方法來增強。 `:before` 方法允許我們說：“嘿首先，先做這個。” 最具體的 `:before` 方法優先被呼叫，作爲其它方法呼叫的序幕 (prelude)。 `:after` 方法允許我們說 “P.S. 也做這個。” 最具體的 `:after` 方法最後被呼叫，作爲其它方法呼叫的閉幕 (epilogue)。在這之間，我們運行的是在這之前僅視爲方法的方法，而準確地說應該叫做主方法 (*primary method*)。這個主方法呼叫所返回的值爲方法的返回值，甚至 `:after` 方法在之後被呼叫也不例外。

`:before` 與 `:after` 方法允許我們將新的行爲包在呼叫主方法的周圍。 `:around` 方法提供了一個更戲劇的方式來辦到這件事。如果 `:around` 方法存在的話，會呼叫的是 `:around` 方法而不是主方法。則根據它自己的判斷， `:around` 方法自己可能會呼叫主方法（通過函數 `call-next-method`，這也是這個函數存在的目的）。

這稱爲標準方法組合機制 (*standard method combination*)。在標準方法組合機制裡，呼叫一個通用函數會呼叫

1. 最具體的 `:around` 方法，如果有的話。
2. 否則，依序，
    a. 所有的 `:before` 方法，從最具體到最不具體。
    b. 最具體的主方法
    c. 所有的 `:after` 方法，從最不具體到最具體

返回值爲 `:around` 方法的返回值（情況 1）或是最具體的主方法的返回值（情況 2）。

輔助方法通過在 `defmethod` 呼叫中，在方法名後加上一個修飾關鍵字 (qualifying keyword)來定義。如果我們替 `speaker` 類別定義一個主要的 `speak` 方法如下：

```
(defclass speaker () ())

(defmethod speak ((s speaker) string)
  (format t "~A" string))
```

則使用 `speaker` 實體來呼叫 `speak` 僅印出第二個參數：

```
> (speak (make-instance 'speaker)
         "I'm hungry")
I'm hungry
NIL
```

通過定義一個 intellectual 子類，將主要的 speak 方法用 :before 與 :after 方法包起來，

```
(defclass intellectual (speaker) ())

(defmethod speak :before ((i intellectual) string)
  (princ "Perhaps "))

(defmethod speak :after ((i intellectual) string)
  (princ " in some sense"))
```

我們可以創建一個說話前後帶有慣用語的演講者：

```
> (speak (make-instance 'intellectual)
         "I am hungry")
Perhaps I am hungry in some sense
NIL
```

如同先前標準方法組合機制所述，所有的 :before 及 :after 方法都被呼叫了。所以如果我們替 speaker 基類定義 :before 或 :after 方法，

```
(defmethod speak :before ((s speaker) string)
  (princ "I think "))
```

無論是哪個 :before 或 :after 方法被呼叫，整個通用函數所返回的值，是最具體主方法的返回值 —— 在這個情況下，爲 format 函數所返回的 nil 。

而在有 :around 方法時，情況就不一樣了。如果有一個替傳入通用函數特別定義的 :around 方法，則優先呼叫 :around 方法，而其它的方法要看 :around 方法讓不讓它們被運行。一個 :around 或主方法，可以通過呼叫 call-next-method 來呼叫下一個方法。在呼叫下一個方法前，它使用 next-method-p 來檢查是否有下個方法可呼叫。

有了 :around 方法，我們可以定義另一個，更謹慎的， speaker 的子類別：

```
(defclass courtier (speaker) ())

(defmethod speak :around ((c courtier) string)
  (format t "Does the King believe that ~A?" string)
  (if (eql (read) 'yes)
      (if (next-method-p) (call-next-method))
      (format t "Indeed, it is a preposterous idea. ~%"))
  'bow)
```

當傳給 speak 的第一個參數是 courtier 類的實體時，朝臣 (courtier)的舌頭有了

:around 方法保護，就不會被割掉了：

```
> (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) "kings will last")
Does the King believe that kings will last? no
Indeed, it is a preposterous idea.
BOW
```

記得由 :around 方法所返回的值即通用函數的返回值，這與 :before 與 :after 方法的返回值不一樣。

11.8 方法組合機制 (Method Combination)

在標準方法組合中，只有最具體的主方法會被呼叫（雖然它可以通過 call-next-method 來呼叫其它方法）。但我們可能會想要把所有可用的主方法的結果彙總起來。

用其它組合手段來定義方法也是有可能的 —— 舉例來說，一個返回所有可用主方法的和的通用函數。運算子 (*Operator*)方法組合可以這麼理解，想像它是 Lisp 表達式的求值後的結果，其中 Lisp 表達式的第一個元素是某個運算子，而參數是按照具體性呼叫可用主方法的結果。如果我們定義 price 使用 + 來組合數值的通用函數，並且沒有可用的 :around 方法，它會如它所定義的方式動作：

```
(defun price (&rest args)
  (+ (apply 〈most specific primary method〉 args)
     .
     .
     .
     (apply 〈least specific primary method〉 args)))
```

如果有可用的 :around 方法的話，它們根據優先序決定，就像是標準方法組合那樣。在運算子方法組合裡，一個 around 方法仍可以通過 call-next-method 呼叫下個方法。然而主方法就不可以使用 call-next-method 了。

我們可以指定一個通用函數的方法組合所要使用的型別，藉由在 defgeneric 呼叫里加入一個 method-combination 子句：

```
(defgeneric price (x)
  (:method-combination +))
```

現在 price 方法會使用 + 方法組合；任何替 price 定義的 defmethod 必須有 + 來作爲第二個參數。如果我們使用 price 來定義某些型別，

```
(defclass jacket () ())
(defclass trousers () ())
(defclass suit (jacket trousers) ())

(defmethod price + ((jk jacket)) 350)
(defmethod price + ((tr trousers)) 200)
```

則可以獲得一件西裝的價格，也就是所有可用方法的總和：

```
> (price (make-instance 'suit))
550
```

下列符號可以用來作爲 defmethod 的第二個參數或是作爲 defgeneric 呼叫中，method-combination 的選項：

```
+    and    append    list    max    min    nconc    or    progn
```

你也可以使用 standard，yields 標準方法組合。

一旦你指定了通用函數要用何種方法組合，所有替該函數定義的方法必須用同樣的機制。而現在如果我們試著使用另個運算子（ :before 或 after ）作為 defmethod 給 price 的第二個參數，則會拋出一個錯誤。如果我們想要改變 price 的方法組合機制，我們需要通過呼叫 fmakunbound 來移除整個通用函數。

而現在如果我們試著使用另個運算子（ :before 或 after) 作為 defmethod 給 price 的第二個參數，則會拋出一個錯誤。

11.9 封裝 (Encapsulation)

面向物件的語言通常會提供某些手段，來區別物件的表示法以及它們給外在世界存取的介面。隱藏實現細節帶來兩個優點：你可以改變實現方式，而不影響物件對外的樣子，而你可以保護物件在可能的危險方面被改動。隱藏細節有時候被稱爲封裝 (*encapsulated*)。

雖然封裝通常與物件導向程式設計相關聯，但這兩個概念其實是沒相乾的。你可以只擁有其一，而不需要另一個。我們已經在 108 頁 (譯註： 6.5 小節。)看過一個小規模的封裝例子。函數 stamp 及 reset 通過共享一個計數器工作，但呼叫時我們不需要知道這個計數器，也保護我們不可直接修改它。

在 Common Lisp 裡，包是標準的手段來區分公開及私有的資訊。要限制某個東西的存取，我們將它放在另一個包裡，並且針對外部介面，僅輸出需要用的名字。

我們可以通過輸出可被改動的名字，來封裝一個槽，但不是槽的名字。舉例來說，我們可以定義一個 counter 類別，以及相關的 increment 及 clear 方法如下：

```
(defpackage "CTR"
  (:use "COMMON-LISP")
  (:export "COUNTER" "INCREMENT" "CLEAR"))

(in-package ctr)

(defclass counter () ((state :initform 0)))

(defmethod increment ((c counter))
  (incf (slot-value c 'state)))

(defmethod clear ((c counter))
  (setf (slot-value c 'state) 0))
```

在這個定義下，在包外部的程式只能夠創造 counter 的實體，並呼叫 increment 及 clear 方法，但不能夠存取 state 。

如果你想要更進一步區別類的內部及外部介面，並使其不可能存取一個槽所存的值，你也可以這麼做。只要在你將所有需要引用它的程式碼定義完，將槽的名字 unintern：

```
(unintern 'state)
```

則沒有任何合法的、其它的辦法，從任何包來引用到這個槽。 λ [http://acl.readthedocs.org/en/latest/zhTW/notes.html#notes-191]

11.10 兩種模型 (Two Models)

物件導向程式設計是一個令人疑惑的話題，部分的原因是因爲有兩種實現方式：訊息傳遞模型 (message-passing model)與通用函數模型 (generic function model)。一開始先有的訊息傳遞。通用函數是廣義的訊息傳遞。

在訊息傳遞模型裡，方法屬於物件，且方法的繼承與槽的繼承概念一樣。要找到一個物體的面積，我們傳給它一個 area 消息：

```
tell obj area
```

而這呼叫了任何物件 obj 所擁有或繼承來的 area 方法。

有時候我們需要傳入額外的參數。舉例來說，一個 move 方法接受一個說明要移動多遠的參數。如我我們想要告訴 obj 移動 10 個單位，我們可以傳下面的消息：

```
(move obj 10)
```

訊息傳遞模型的侷限性變得清晰。在訊息傳遞模型裡，我們僅特化 (specialize) 第一個參數。 牽扯到多物件時，沒有規則告訴方法該如何處理 —— 而物件回應消息的這個模型使得這更加難處理了。

在訊息傳遞模型裡，方法是物件所有的，而在通用函數模型裡，方法是特別爲物件打造的 (specialized)。 如果我們僅特化第一個參數，那麼通用函數模型和訊息傳遞模型就是一樣的。但在通用函數模型裡，我們可以更進一步，要特化幾個參數就幾個。這也表示了，功能上來說，訊息傳遞模型是通用函數模型的子集。如果你有通用函數模型，你可以僅特化第一個參數來模擬出訊息傳遞模型。

Chapter 11 總結 (Summary)

1. 在物件導向程式設計中，函數 `f` 通過定義擁有 `f` 方法的物件來隱式地定義。物件從它們的父母繼承方法。
2. 定義一個類別就像是定義一個結構，但更加囉嗦。一個共享的槽屬於一整個類別。
3. 一個類別從基類中繼承槽。
4. 一個類別的祖先被排序成一個優先序列表。理解優先序算法最好的方式就是通過視覺。
5. 一個通用函數由一個給定名稱的所有方法所組成。一個方法通過名稱及特化參數來識別。參數的優先序決定了當呼叫一個通用函數時會使用哪個方法。
6. 方法可以通過輔助方法來增強。標準方法組合機制意味著如果有 `:around` 方法的話就呼叫它；否則依序呼叫 `:before` ，最具體的主方法以及 `:after` 方法。
7. 在運算子方法組合機制中，所有的主方法都被視爲某個運算子的參數。
8. 封裝可以通過包來實現。

10. 物件導向程式設計有兩個模型。通用函數模型是廣義的訊息傳遞模型。

Chapter 11 練習 (Exercises)

1. 替圖 11.2 所定義的類定義存取器、 initforms 以及 initargs 。重寫相關的程式，使其再也不用呼叫 `slot-value` 。
2. 重寫圖 9.5 的程式碼，使得球體與點爲類別，而 `intersect` 及 `normal` 爲通用函數。
3. 假設有若干類別定義如下：

```
(defclass a (c d)   ...)  (defclass e ()   ...)
(defclass b (d c)   ...)  (defclass f (h) ...)
(defclass c ()      ...)  (defclass g (h) ...)
```

```
(defclass d (e f g) ...)  (defclass h ()  ...)
```

a. 畫出表示類別 a 祖先的網路以及列出 a 的實體歸屬的類別，從最相關至最不相關排列。
b. 替類別 b 也做 (a) 小題的要求。

4. 假定你已經有了下列函數：

`precedence` ：接受一個物件並返回其優先序列表，列表由最具體至最不具體的類組成。

`methods`：接受一個通用函數並返回一個列出所有方法的列表。

`specializations` ：接受一個方法並返回一個列出所有特化參數的列表。返回列表中的每個元素是類別或是這種形式的列表 `(eql x)`，或是 `t`（表示該參數沒有被特化）。

使用這些函數（不要使用 `compute-applicable-methods` 及 `find-method` ），定義一個函數 `most-spec-app-meth` ，該函數接受一個通用函數及一個列出此函數被呼叫過的參數，如果有最相關可用的方法的話，返回它。

5. 不要改變通用函數 `area` 的行爲（圖 11.2），
6. 舉一個只有通用函數的第一個參數被特化會很難解決的問題的例子。

腳註

[1] Initarg 的名稱通常是關鍵字，但不需要是。
[2] 我們不可能比較完所有的參數而仍有平手情形存在，因爲這樣我們會有兩個有著同樣特化的方法。這是不可能的，因爲第二個的定義會覆寫掉第一個。

第十二章：結構

3.3 節中介紹了 Lisp 如何使用指標允許我們將任何值放到任何地方。這種說法是完全有可能的，但這並不一定都是好事。

例如，一個物件可以是它自己的一個元素。這是好事還是壞事，取決於程式設計師是不是有意這樣設計的。

12.1 共享結構 (Shared Structure)

多個列表可以共享 `cons` 。在最簡單的情況下，一個列表可以是另一個列表的一部分。

```
> (setf part (list 'b 'c))
(B C)
> (setf whole (cons 'a part))
(A B C)
```

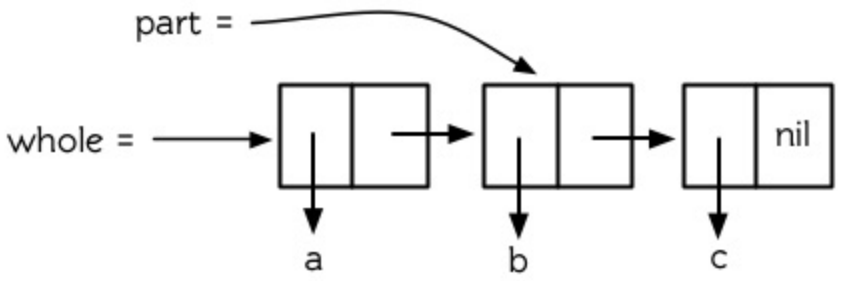

圖 **12.1** 共享結構

執行上述操作後，第一個 `cons` 是第二個 `cons` 的一部分 (事實上，是第二個 `cons` 的 `cdr`)。在這樣的情況下，我們說，這兩個列表是共享結構 (Share Structure)。這兩個列表的基本結構如圖 12.1 所示。

其中，第一個 `cons` 是第二個 `cons` 的一部分 (事實上，是第二個 `cons` 的 `cdr`)。在這樣的情況下，我們稱這兩個列表爲共享結構 (Share Structure)。這兩個列表的基本結構如圖 12.1 所示。

使用 `tailp` 判斷式來檢測一下。將兩個列表作爲它的輸入參數，如果第一個列表是第二個列表的一部分時，則返回 `T` ：

```
> (tailp part whole)
T
```

我們可以把它想像成：

```
(defun our-tailp (x y)
  (or (eql x y)
      (and (consp y)
                          (our-tailp x (cdr y)))))
```

如定義所表明的，每個列表都是它自己的尾端， `nil` 是每一個正規列表的尾端。

在更複雜的情況下，兩個列表可以是共享結構，但彼此都不是對方的尾端。在這種情況下，他們都有一個共同的尾端，如圖 12.2 所示。我們像這樣構建這種情況：

```
(setf part (list 'b 'c)
                        whole1 (cons 1 part)
                        whole2 (cons 2 part))
```

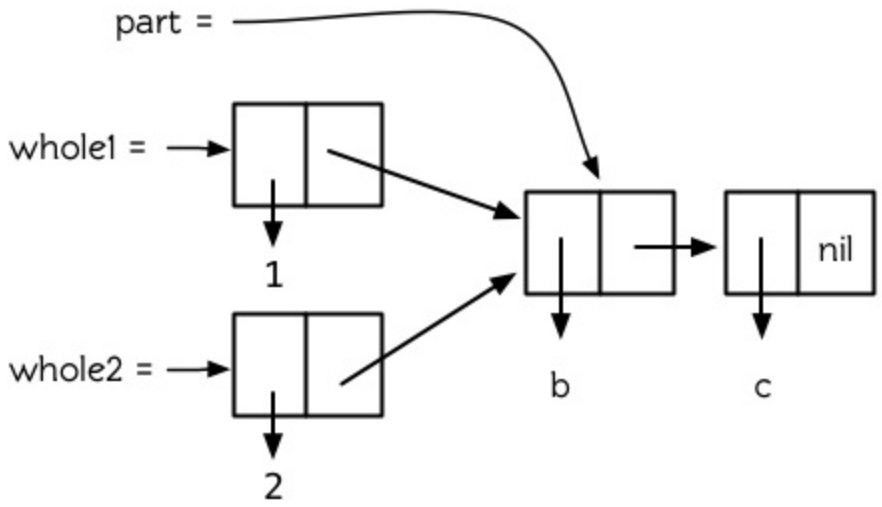

圖 **12.2** 被共享的尾端

現在 `whole1` 和 `whole2` 共享結構，但是它們彼此都不是對方的一部分。

當存在巢狀列表時，重要的是要區分是列表共享了結構，還是列表的元素共享了結構。頂層列表結構指的是，直接構成列表的那些 `cons` ，而不包含那些用於構造列表元素的 `cons` 。圖 12.3 是一個巢狀列表的頂層列表結構 (譯者注：圖 12.3 中上面那三個有黑色陰影的 `cons` 即構成頂層列表結構的 `cons`)。

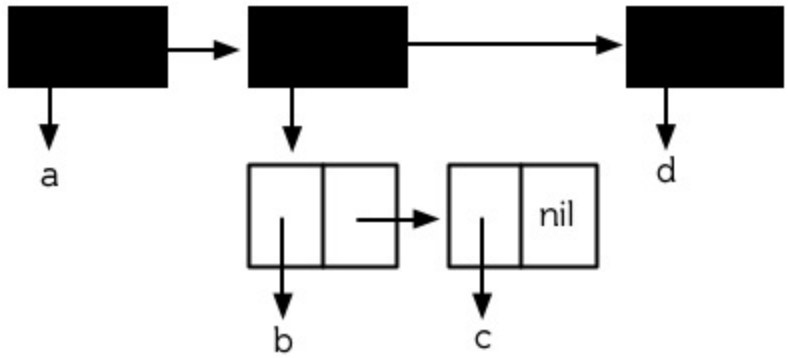

圖 **12.3** 頂層列表結構

兩個 `cons` 是否共享結構，取決於我們把它們看作是列表還是樹[http://zh.wikipedia.org/wiki/%E6%A0%91_(%E6%95%B0%E6%8D%AE%E7%BB%93%E6%可能存在兩個巢狀列表，當把它們看作樹時，它們共享結構，而看作列表時，它們不共享結構。圖 12.4 構建了這種情況，兩個列表以一個元素的形式包含了同一個列表，程式碼如下：

```
(setf element (list 'a 'b)
      holds1 (list 1 element 2)
      holds2 (list element 3))
```

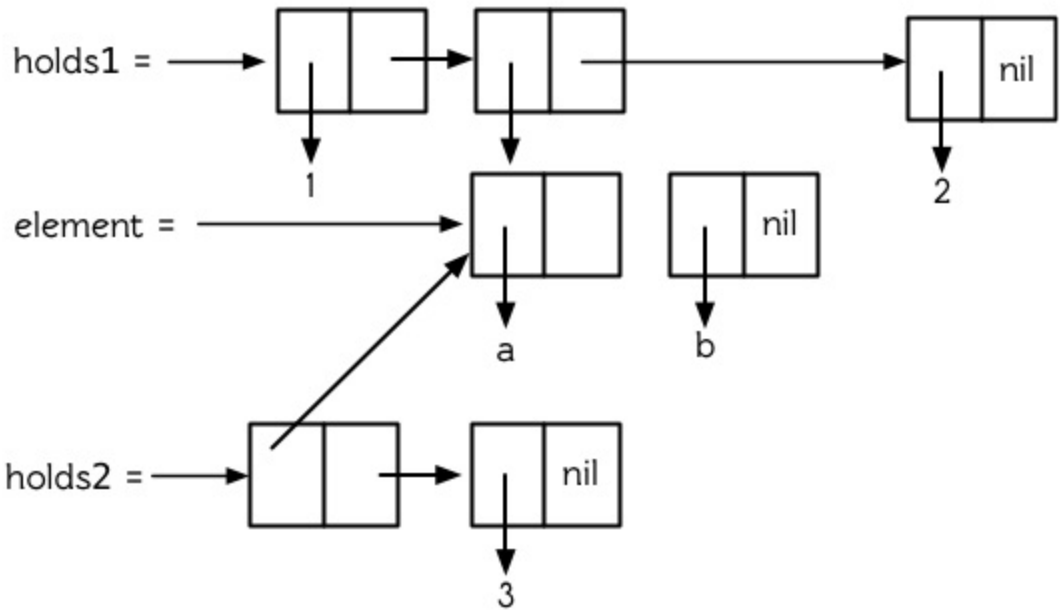

圖 **12.4** 共享子樹

雖然 `holds1` 的第二個元素和 `holds2` 的第一個元素共享結構 (其實是相同的)，但如果把 `holds1` 和 `holds2` 看成是列表時，它們不共享結構。僅當兩個列表共享頂層列表結構時，才能說這兩個列表共享結構，而 `holds1` 和 `holds2` 沒有共享頂層列表結構。

如果我們想避免共享結構，可以使用複製。函數 copy-list 可以這樣定義：

```
(defun our-copy-list (lst)
   (if (null lst)
       nil
       (cons (car lst) (our-copy-list (cdr lst)))))
```

它返回一個不與原始列表共享頂層列表結構的新列表。函數 copy-tree 可以這樣定義：

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

它返回一個連原始列表的樹型結構也不共享的新列表。圖 12.5 顯示了對一個巢狀列表使用 copy-list 和 copy-tree 的區別。

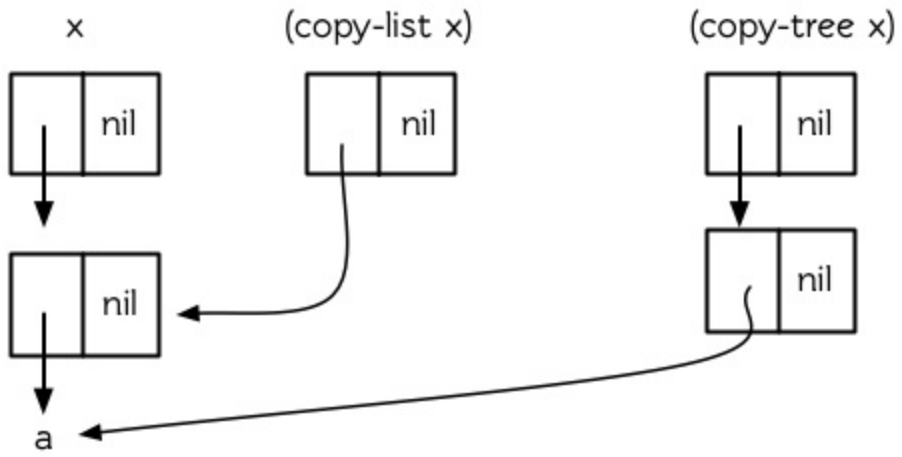

圖 **12.5** 兩種複製

12.2 修改 (Modification)

爲什麼要避免共享結構呢？之前討論的共享結構問題僅僅是個智力練習，到目前爲止，並沒使我們在實際寫程式的時候有什麼不同。當修改一個被共享的結構時，問題出現了。如果兩個列表共享結構，當我們修改了其中一個，另外一個也會無意中被修改。

上一節中，我們介紹了怎樣構建一個是其它列表的尾端的列表：

```
(setf whole (list 'a 'b 'c)
      tail (cdr whole))
```

因爲 whole 的 cdr 與 tail 是相等的，無論是修改 tail 還是 whole 的 cdr ，我們修改的

都是同一個 cons：

```
> (setf (second tail ) 'e)
E
> tail
(B E)
> whole
(A B E)
```

同樣的，如果兩個列表共享同一個尾端，這種情況也會發生。

一次修改兩個物件並不總是錯誤的。有時候這可能正是你想要的。但如果無意的修改了共享結構，將會引入一些非常微妙的 bug。Lisp 程式設計師要培養對共享結構的意識，並且在這類錯誤發生時能夠立刻反應過來。當一個列表神祕的改變了的時候，很有可能是因爲改變了其它與之共享結構的物件。

真正危險的不是共享結構，而是改變被共享的結構。爲了安全起見，乾脆避免對結構使用 setf (以及相關的運算，比如： pop ， rplaca 等)，這樣就不會遇到問題了。如果某些時候不得不修改列表結構時，要搞清楚要修改的列表的來源，確保它不要和其它不需要改變的物件共享結構。如果它和其它不需要改變的物件共享了結構，或者不能預測它的來源，那麼複製一個副本來進行改變。

當你呼叫別人寫的函數的時候要加倍小心。除非你知道它內部的操作，否則，你傳入的參數時要考慮到以下的情況：

1.它對你傳入的參數可能會有破壞性的操作

2.你傳入的參數可能被保存起來，如果你呼叫了一個函數，然後又修改了之前作爲參數傳入該函數的物件，那麼你也就改變了函數已保存起來作爲它用的物件[1]。

在這兩種情況下，解決的方法是傳入一個拷貝。

在 Common Lisp 中，一個函數呼叫在遍歷列表結構 (比如， mapcar 或 remove-if 的參數)的過程中不允許修改被遍歷的結構。關於評估這種程式的重要性並沒有明確的規定。

12.3 範例：佇列 (Example: Queues)

共享結構並不是一個總讓人擔心的特性。我們也可以對其加以利用的。這一節展示了怎樣用共享結構來表示佇列 [http://zh.wikipedia.org/wiki/%E9%98%9F%E5%88%97]。佇列物件是我們可以按照資料的插入順序逐個檢出資料的倉庫，這個規則叫做先進先出 (FIFO, first in, first out) [http://zh.wikipedia.org/zh-

cn/%E5%85%88%E9%80%B2%E5%85%88%E5%87%BA]。

用列表表示栈 (stack) [http://zh.wikipedia.org/wiki/%E6%A0%88]比較容易，因爲栈是從同一端插入和檢出。而表示佇列要困難些，因爲佇列的插入和檢出是在不同端。爲了有效的實現佇列，我們需要找到一種辦法來指向列表的兩個端。

圖 12.6 給出了一種可行的策略。它展示怎樣表示一個含有 a，b，c 三個元素的佇列。一個佇列就是一對列表，最後那個 `cons` 在相同的列表中。這個列表對由被稱作頭端 (front)和尾端 (back)的兩部分組成。如果要從佇列中檢出一個元素，只需在其頭端 `pop`，要插入一個元素，則創建一個新的 `cons`，把尾端的 `cdr` 設置成指向這個 `cons`，然後將尾端指向這個新的 `cons`。

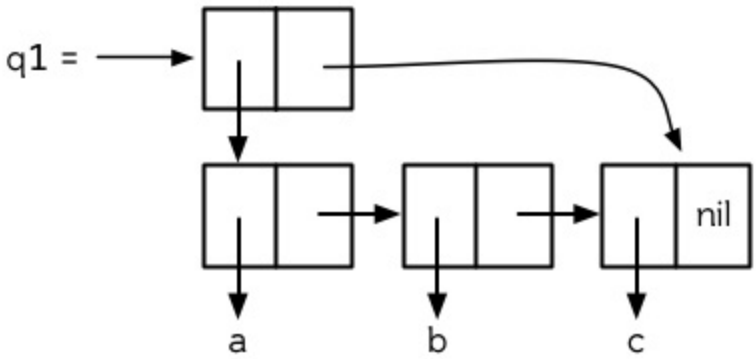

圖 **12.6** 一個佇列的結構

```
(defun make-queue () (cons nil nil))

(defun enqueue (obj q)
  (if (null (car q))
      (setf (cdr q) (setf (car q) (list obj)))
      (setf (cdr (cdr q)) (list obj)
            (cdr q) (cdr (cdr q)))))
  (car q))

(defun dequeue (q)
  (pop (car q)))
```

圖 **12.7** 佇列實現

圖 12.7 中的程式實現了這一策略。其用法如下：

```
> (setf q1 (make-queue))
(NIL)
> (progn (enqueue 'a q1)
         (enqueue 'b q1)
         (enqueue 'c q1))
(A B C)
```

現在， q1 的結構就如圖 12.6 那樣：

```
> q1
((A B C) C)
```

從佇列中檢出一些元素：

```
> (dequeue q1)
A
> (dequeue q1)
B
> (enqueue 'd q1)
(C D)
```

12.4 破壞性函數 (Destructive Functions)

Common Lisp 包含一些允許修改列表結構的函數。爲了提高效率，這些函數是具有破壞性的。雖然它們可以回收利用作爲參數傳給它們的 cons ，但並不是因爲想要它們的副作用而呼叫它們 (譯者注：因爲這些函數的副作用並沒有任何保證，下面的例子將說明問題)。

比如， delete 是 remove 的一個具有破壞性的版本。雖然它可以破壞作爲參數傳給它的列表，但它並不保證什麼。在大多數的 Common Lisp 的實現中，會出現下面的情況：

```
> (setf lst '(a r a b i a) )
(A R A B I A)
> (delete 'a lst )
(R B I)
> lst
(A R B I)
```

正如 remove 一樣，如果你想要副作用，應該對返回值使用 setf ：

```
(setf lst (delete 'a lst))
```

破壞性函數是怎樣回收利用傳給它們的列表的呢？比如，可以考慮 nconc —— append 的破壞性版本。[2]下面是兩個參數版本的實現，其清楚地展示了兩個已知列表是怎樣被縫在一起的：

```
(defun nconc2 ( x y)
    (if (consp x)
        (progn
           (setf (cdr (last x)) y)
            x)
```

```
        y))
```

我們找到第一個列表的最後一個 *Cons* 核 (cons cells)，把它的 `cdr` 設置成指向第二個列表。一個正規的多參數的 `nconc` 可以被定義成像附錄 B 中的那樣。

函數 `mapcan` 類似 `mapcar` ，但它是用 `nconc` 把函數的返回值 (必須是列表) 拼接在一起的：

```
> (mapcan #'list
          '(a b c)
          '(1 2 3 4))
( A 1 B 2 C 3)
```

這個函數可以定義如下：

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

使用 `mapcan` 時要謹慎，因爲它具有破壞性。它用 `nconc` 拼接返回的列表，所以這些列表最好不要再在其它地方使用。

這類函數在處理某些問題的時候特別有用，比如，收集樹在某層上的所有子結點。如果 `children` 函數返回一個節點的孩子節點的列表，那麼我們可以定義一個函數返回某節點的孫子節點的列表如下：

```
(defun grandchildren (x)
   (mapcan #'(lambda (c)
               (copy-list (children c)))
           (children x)))
```

這個函數呼叫 `copy-list` 時存在一個假設 —— `chlidren` 函數返回的是一個已經保存在某個地方的列表，而不是構建了一個新的列表。

一個 `mapcan` 的無損變體可以這樣定義：

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

如果使用 `mappend` 函數，那麼 `grandchildren` 的定義就可以省去 `copy-list` ：

```
(defun grandchildren (x)
   (mappend #'children (children x)))
```

12.5 範例：二元搜索樹 (Example: Binary Search Trees)

在某些情況下，使用破壞性操作比使用非破壞性的顯得更自然。第 4.7 節中展示了如何維護一個具有二分搜索格式的有序物件集 (或者說維護一個二元搜索樹 (BST) [http://zh.wikipedia.org/zh-cn/%E4%BA%8C%E5%85%83%E6%90%9C%E5%B0%8B%E6%A8%B9])。第 4.7 節中給出的函數都是非破壞性的，但在我們真正使用BST的時候，這是一個不必要的保護措施。本節將展示如何定義更符合實際應用的具有破壞性的插入函數和刪除函數。

圖 12.8 示範了如何定義一個具有破壞性的 `bst-insert` (第 72 頁「譯者注：第 4.7 節」)。相同的輸入參數，能夠得到相同返回值。唯一的區別是，它將修改作爲第二個參數輸入的 BST。 在第 2.12 節中說過，具有破壞性並不意味著一個函數呼叫具有副作用。的確如此，如果你想使用 `bst-insert!` 構造一個 BST，你必須像呼叫 `bst-insert` 那樣呼叫它：

```
> (setf *bst* nil)
NIL
> (dolist (x '(7 2 9 8 4 1 5 12))
(setf *bst* (bst-insert! x *bst* #'<)))
NIL
```

```
(defun bst-insert! (obj bst <)
  (if (null bst)
      (make-node :elt obj)
      (progn (bsti obj bst <)
             bst)))

(defun bsti (obj bst <)
  (let ((elt (node-elt bst)))
    (if (eql obj elt)
        bst
        (if (funcall < obj elt)
            (let ((l (node-l bst)))
              (if l
                  (bsti obj l <)
                  (setf (node-l bst)
                        (make-node :elt obj))))
            (let ((r (node-r bst)))
              (if r
                  (bsti obj r <)
                  (setf (node-r bst)
                        (make-node :elt obj)))))))))
```

圖 **12.8:** 二元搜索樹：破壞性插入

你也可以爲 BST 定義一個類似 push 的功能，但這超出了本書的範圍。(好奇的話，可以

參考第 409 頁 「譯者注：即備註 204 」 的宏定義。)

與 bst-remove (第 74 頁「譯者注：第 4.7 節」) 對應，圖 12.9 展示了一個破壞性版本的 bst-delete 。同 delete 一樣，我們呼叫它並不是因爲它的副作用。你應該像呼叫 bst-remove 那樣呼叫 bst-delete :

```
> (setf *bst* (bst-delete 2 *bst* #'<) )
#<7>
> (bst-find 2 *bst* #'<)
NIL
```

```
(defun bst-delete (obj bst <)
  (if bst (bstd obj bst nil nil <))
  bst)

(defun bstd (obj bst prev dir <)
  (let ((elt (node-elt bst)))
    (if (eql elt obj)
        (let ((rest (percolate! bst)))
          (case dir
            (:l (setf (node-l prev) rest))
            (:r (setf (node-r prev) rest))))
      (if (funcall < obj elt)
          (if (node-l bst)
              (bstd obj (node-l bst) bst :l <))
          (if (node-r bst)
              (bstd obj (node-r bst) bst :r <))))))

(defun percolate! (bst)
  (cond ((null (node-l bst))
         (if (null (node-r bst))
             nil
             (rperc! bst)))
        ((null (node-r bst)) (lperc! bst))
        (t (if (zerop (random 2))
               (lperc! bst)
               (rperc! bst)))))

(defun lperc! (bst)
  (setf (node-elt bst) (node-elt (node-l bst)))
  (percolate! (node-l bst)))

(defun rperc! (bst)
  (setf (node-elt bst) (node-elt (node-r bst)))
  (percolate! (node-r bst)))
```

圖 **12.9:** 二元搜索樹：破壞性刪除

譯註： 此範例已被回報爲錯誤的，一個修復的版本請造訪這裡 [https://gist.github.com/2868339]。

12.6 範例：雙向鏈表 (Example: Doubly-Linked Lists)

普通的 Lisp 列表是單向鏈表，這意味著其指標指向一個方向：我們可以獲取下一個元素，但不能獲取前一個。在雙向鏈表[http://zh.wikipedia.org/wiki/%E5%8F%8C%E5%90%91%E9%93%BE%E8%A1%A8]中，指標指向兩個方向，我們獲取前一個元素和下一個元素都很容易。這一節將介紹如何創建和操作雙向鏈表。

圖 12.10 示範了如何用結構來實現雙向鏈表。將 `cons` 看成一種結構，它有兩個欄位：指向資料的 `car` 和指向下一個元素的 `cdr` 。要實現一個雙向鏈表，我們需要第三個欄位，用來指向前一個元素。圖 12.10 中的 `defstruct` 定義了一個含有三個欄位的物件 `dl` (用於“雙向連結”)，我們將用它來構造雙向鏈表。`dl` 的 `data` 欄位對應一個 `cons` 的 `car`，`next` 欄位對應 `cdr` 。 `prev` 欄位就類似一個 `cdr` ，指向另外一個方向。(圖 12.11 是一個含有三個元素的雙向鏈表。) 空的雙向鏈表爲 `nil` ，就像空的列表一樣。

```
(defstruct (dl (:print-function print-dl))
  prev data next)

(defun print-dl (dl stream depth)
  (declare (ignore depth))
  (format stream "#<DL ~A>" (dl->list dl)))

(defun dl->list (lst)
  (if (dl-p lst)
      (cons (dl-data lst) (dl->list (dl-next lst)))
      lst))

(defun dl-insert (x lst)
  (let ((elt (make-dl :data x :next lst)))
    (when (dl-p lst)
      (if (dl-prev lst)
          (setf (dl-next (dl-prev lst)) elt
                (dl-prev elt) (dl-prev lst)))
      (setf (dl-prev lst) elt))
    elt))

(defun dl-list (&rest args)
  (reduce #'dl-insert args
          :from-end t :initial-value nil))

(defun dl-remove (lst)
  (if (dl-prev lst)
      (setf (dl-next (dl-prev lst)) (dl-next lst)))
  (if (dl-next lst)
      (setf (dl-prev (dl-next lst)) (dl-prev lst)))
  (dl-next lst))
```

圖 **12.10:** 構造雙向鏈表

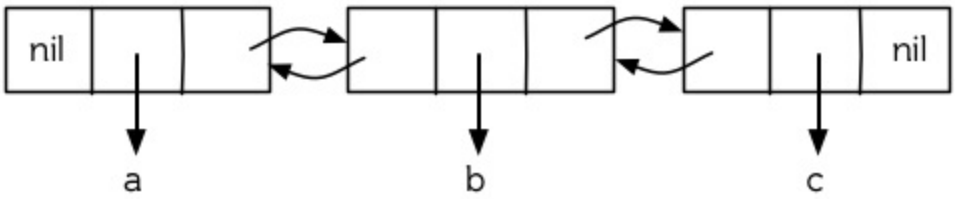

圖 **12.11:** 一個雙向鏈表。

爲了便於操作，我們爲雙向鏈表定義了一些實現類似 `car` ， `cdr` ， `consp` 功能的函數：`dl-data` ， `dl-next` 和 `dl-p` 。 `dl->list` 是 `dl` 的打印函數(`print-function`)，其返回一個包含 `dl` 所有元素的普通列表。

函數 `dl-insert` 就像針對雙向鏈表的 `cons` 操作。至少，它就像 `cons` 一樣，是一個基本構建函數。與 `cons` 不同的是，它實際上要修改作爲第二個參數傳遞給它的雙向鏈表。在這種情況下，這是自然而然的。我們 `cons` 內容到普通列表前面，不需要對普通列表的 `rest` (譯者注： `rest` 即 `cdr` 的另一種表示方法，這裡的 `rest` 是對通過 `cons` 構建後列表來說的，即修改之前的列表) 做任何修改。但是要在雙向鏈表的前面插入元素，我們不得不修改列表的 `rest` (這裡的 `rest` 即指沒修改之前的雙向鏈表) 的 `prev` 欄位來指向這個新元素。

幾個普通列表可以共享同一個尾端。因爲雙向鏈表的尾端不得不指向它的前一個元素，所以不可能存在兩個雙向鏈表共享同一個尾端。如果 `dl-insert` 不具有破壞性，那麼它不得不複製其第二個參數。

單向鏈表(普通列表)和雙向鏈表另一個有趣的區別是，如何持有它們。我們使用普通列表的首端，來表示單向鏈表，如果將列表賦值給一個變數，變數可以通過保存指向列表第一個 `cons` 的指標來持有列表。但是雙向鏈表是雙向指向的，我們可以用任何一個點來持有雙向鏈表。 `dl-insert` 另一個不同於 `cons` 的地方在於 `dl-insert` 可以在雙向鏈表的任何位置插入新元素，而 `cons` 只能在列表的首端插入。

函數 `dl-list` 是對於 `dl` 的類似 `list` 的功能。它接受任意多個參數，它會返回一個包含以這些參數作爲元素的 `dl` ：

```
> (dl-list 'a 'b 'c)
#<DL (A B C)>
```

它使用了 `reduce` 函數 (並設置其 `from-end` 參數爲 `true`， `initial-value` 爲 `nil`)，其功能等價於

```
(dl-insert 'a (dl-insert 'b (dl-insert 'c nil)) )
```

如果將 dl-list 定義中的 #'dl-insert 換成 #'cons，它就相當於 list 函數了。下面是 dl-list 的一些常見用法：

```
> (setf dl (dl-list 'a 'b))
#<DL (A B)>
> (setf dl (dl-insert 'c dl))
#<DL (C A B)>
> (dl-insert 'r (dl-next dl))
#<DL (R A B)>
> dl
#<DL (C R A B)>
```

最後，dl-remove 的作用是從雙向鏈表中移除一個元素。同 dl-insert 一樣，它也是具有破壞性的。

12.7 環狀結構 (Circular Structure)

將列表結構稍微修改一下，就可以得到一個環形列表。存在兩種環形列表。最常用的一種是其頂層列表結構是一個環的，我們把它叫做 cdr-circular ，因爲環是由一個 cons 的 cdr 構成的。

構造一個單元素的 cdr-circular 列表，可以將一個列表的 cdr 設置成列表自身：

```
> (setf x (list 'a))
(A)
> (progn (setf (cdr x) x) nil)
NIL
```

這樣 x 就是一個環形列表，其結構如圖 12.12 (左) 所示。

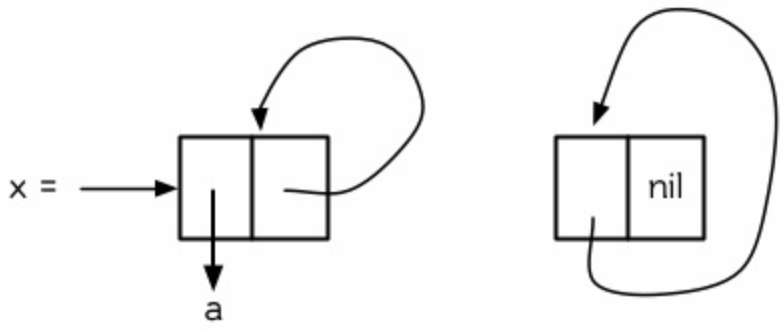

圖 **12.12** 環狀列表。

如果 Lisp 試著打印我們剛剛構造的結構，將會顯示 (a a a a a ..... —— 無限個 a)。但如果設置全局變數 *print-circle* 爲 t 的話，Lisp 就會採用一種方式打印出一個能代表環形結構的物件：

```
> (setf *print-circle* t )
T
> x
#1=(A . #1#)
```

如果你需要，你也可以使用 #n= 和 #n# 這兩個讀取宏，來自己表示共享結構。

cdr-cicular 列表十分有用，比如，可以用來表示緩衝區、池。下面這個函數，可以將一個普通的非空列表，轉換成一個對應的 cdr-cicular 列表：

```
(defun circular (lst)
        (setf (cdr (last lst)) lst))
```

另外一種環狀列表叫做 car-circular 列表。car-circular 列表是一個樹，並將其自身當作自己的子樹的結構。因爲環是通過一個 cons 的 car 形成的，所以叫做 car-circular。這裡構造了一個 car-circular，它的第二個元素是它自身：

```
> (let ((y (list 'a )))
(setf (car y) y)
      y)
#i=(#i#)
```

圖 12.12 (右) 展示了其結構。這個 car-circular 是一個正規列表。 cdr-circular 列表都不是正規列表，除開一些特殊情況 car-circular 列表是正規列表。

一個列表也可以既是 car-circular，又是 cdr-circular。一個 cons 的 car 和 cdr 均是其自身：

```
> (let ((c (cons 11)) )
      (setf (car c) c
            (cdr c) c)
      c)
#1=(#1# . #1#)
```

很難想像這樣的一個列表有什麼用。實際上，了解環形列表的主要目的就是爲了避免因爲偶然因素構造出了環形列表，因爲，將一個環形列表傳給一個函數，如果該函數遍歷這個環形列表，它將進入死迴圈。

環形結構的這種問題在列表以外的其他物件中也存在。比如，一個陣列可以將陣列自身當作其元素：

```
> (setf *print-array* t )
T
> (let ((a (make-array 1)) )
          (setf (aref a 0) a)
          a)
#1=#(#1#)
```

實際上，任何可以包含元素的物件都可能包含其自身作爲元素。

用 defstruct 構造出環形結構是相當常見的。比如，一個結構 c 是一顆樹的元素，它的 parent 欄位所指向的結構 p 的 child 欄位也恰好指向 c 。

```
> (progn (defstruct elt
          (parent nil ) (child nil) )
     (let ((c (make-elt) )
                (p (make-elt)) )
          (setf (elt-parent c) p
                     (elt-child p) c)
          c) )
#1=#S(ELT PARENT #S(ELT PARENT NIL CHILD #1#) CHILD NIL)
```

要實現像這樣一個結構的打印函數 (print-function)，我們需要將全局變數 *print-circle* 綁定爲 t ，或者避免打印可能構成環的欄位。

12.8 常數結構 (Constant Structure)

因爲常數實際上是程式碼的一部分，所以我們也不應該修改常數，或者你可能不經意地寫了自重寫的程式碼。一個通過 quote 引用的列表是一個常數，所以一定要小心，不要修改被引用的列表的任何 cons。比如，如果我們用下面的程式，來測試一個符號是不是算術運算符：

```
(defun arith-op (x)
(member x '(+ - * /)))
```

如果被測試的符號是算術運算符，它的返回值將至少一個被引用列表的一部分。如果我們修改了其返回值，

```
> (nconc (arith-op '*) '(as i t were))
(* / AS IT WERE)
```

那麼我就會修改 arith-op 函數中的一個列表，從而改變了這個函數的功能：

```
> (arith-op 'as )
(AS IT WERE)
```

寫一個返回常數結構的函數，並不一定是錯誤的。但當你考慮使用一個破壞性的操作是否安全的時候，你必須考慮到這一點。

有幾個其它方法來實現 `arith-op`，使其不返回被引用列表的部分。一般地，我們可以通過將其中的所有引用(`quote`) 替換成 `list` 來確保安全，這使得它每次被呼叫都將返回一個新的列表：

```
(defun arith-op (x)
        (member x (list '+ '- '* '/)))
```

這裡，使用 `list` 是一種低效的解決方案，我們應該使用 `find` 來替代 `member`：

```
(defun arith-op (x)
        (find x '(+ - * /)))
```

這一節討論的問題似乎只與列表有關，但實際上，這個問題存在於任何複雜的物件中：陣列，字元串，結構，實體等。你不應該逐字地去修改程式的程式碼段。

即使你想寫自修改程式，通過修改常數來實現並不是個好辦法。編譯器將常數編譯成了程式碼，破壞性的操作可能修改它們的參數，但這些都是沒有任何保證的事情。如果你想寫自修改程式，正確的方法是使用閉包 (見 6.5 節)。

Chapter 12 總結 (Summary)

1. 兩個列表可以共享一個尾端。多個列表可以以樹的形式共享結構，而不是共享頂層列表結構。可通過拷貝方式來避免共用結構。
2. 共享結構通常可以被忽略，但如果你要修改列表，則需要特別注意。因爲修改一個含共享結構的列表可能修改所有共享該結構的列表。
3. 佇列可以被表示成一個 `cons` ，其的 `car` 指向佇列的第一個元素， `cdr` 指向佇列的最後一個元素。
4. 爲了提高效率，破壞性函數允許修改其輸入參數。
5. 在某些應用中，破壞性的實現更適用。
6. 列表可以是 `car-circular` 或 `cdr-circular` 。 Lisp 可以表示圓形結構和共享結構。
7. 不應該去修改的程式碼段中的常數形式。

Chapter 12 練習 (Exercises)

1. 畫三個不同的樹，能夠被打印成 `((A) (A) (A))` 。寫一個表達式來生成它們。
2. 假設 `make-queue` ， `enqueue` 和 `dequeue` 是按照圖 12.7 中的定義，用箱子表式法畫

出下面每一步所得到的佇列的結構圖：

```
> (setf q (make-queue))
(NIL)
> (enqueue 'a q)
(A)
> (enqueue 'b q)
(A B)
> (dequeue q)
A
```

3. 定義一個函數 `copy-queue`，可以返回一個 queue 的拷貝。
4. 定義一個函數，接受兩個輸入參數 `object` 和 `queue`，能將 `object` 插入到 `queue` 的首端。
5. 定義一個函數，接受兩個輸入參數 `object` 和 `queue`，能具有破壞性地將 `object` 的第一個實體 (`eql` 等價地) 移到 `queue` 的首端。
6. 定義一個函數，接受兩個輸入參數 `object` 和 `lst` (`lst` 可能是 `cdr-circular` 列表)，如果 `object` 是 `lst` 的成員時返回真。
7. 定義一個函數，如果它的參數是一個 `cdr-circular` 則返回真。
8. 定義一個函數，如果它的參數是一個 `car-circular` 則返回真。

腳註

[1] 比如，在 Common Lisp 中，修改一個被用作符號名的字元串被認爲是一種錯誤，因爲內部的定義並沒宣告它是從參數複製來的，所以必須假定修改傳入內部的任何參數中的字元串來創建新的符號是錯誤的。

[2] 函數名稱中 n 的含義是 “non-consing”。一些具有破壞性的函數以 n 開頭。

第十三章：速度

Lisp 實際上是兩種語言：一種能寫出快速執行的程式，一種則能讓你快速的寫出程式。在程式開發的早期階段，你可以爲了開發上的便捷捨棄程式的執行速度。一旦程式的結構開始固化，你就可以精煉其中的關鍵部分以使得它們執行的更快。

由於各個 Common Lisp 實現間的差異，很難針對優化給出通用的建議。在一個實現上使程式變快的修改也許在另一個實現上會使得程式變慢。這是難免的事兒。越強大的語言，離機器底層就越遠，離機器底層越遠，語言的不同實現沿著不同路徑趨向它的可能性就越大。因此，即便有一些技巧幾乎一定能夠讓程式運行的更快，本章的目的也只是建議而不是規定。

13.1 瓶頸規則 (The Bottleneck Rule)

不管是什麼實現，關於優化都可以整理出三點規則：它應該關注瓶頸，它不應該開始的太早，它應該始於算法。

也許關於優化最重要的事情就是要意識到，程式中的大部分執行時間都是被少數瓶頸所消耗掉的。 正如高德納 [http://en.wikipedia.org/wiki/Donald_Knuth]所說，“在一個與 I/O 無關 (Non-I/O bound) 的程式中，大部分的運行時間集中在大概 3% 的原始碼中。” λ [http://acl.readthedocs.org/en/latest/zhTW/notes.html#notes-213] 優化程式的這一部分將會使得它的運行速度明顯的提升；相反，優化程式的其他部分則是在浪費時間。

因此，優化程式時關鍵的第一步就是找到瓶頸。許多 Lisp 實現都提供性能分析器 (profiler) 來監視程式的運行並報告每一部分所花費的時間量。 爲了寫出最爲高效的程式，性能分析器非常重要，甚至是必不可少的。 如果你所使用的 Lisp 實現帶有性能分析器，那麼請在進行優化時使用它。另一方面，如果實現沒有提供性能分析器的話，那麼你就不得不通過猜測來尋找瓶頸，而且這種猜測往往都是錯的！

瓶頸規則的一個推論是，不應該在程式的初期花費太多的精力在優化上。高德納 [http://en.wikipedia.org/wiki/Donald_Knuth]對此深信不疑：“過早的優化是一切 (至少是大多數) 問題的源頭。” λ [http://acl.readthedocs.org/en/latest/zhTW/notes.html#notes-214] 在剛開始寫程式的時候，通常很難看清真正的瓶頸在哪，如果這個時候進行優化，你很可能是在浪費時間。優化也會使程式的修改變得更加困難，邊寫程式邊優化就像是在用風乾非常快的顏料來畫畫一樣。

在適當的時候做適當的事情，可以讓你寫出更優秀的程式。Lisp 的一個優點就是能讓你用兩種不同的工作方式來進行開發：快速地寫出執行較慢的程式，或者，放慢寫程式的

速度，精雕細琢，從而得出執行得較快的程式。

在程式開發的初期階段，工作通常在第一種模式下進行，只有當性能成爲問題的時候，才切換到第二種模式。 對於非常底層的語言，比如彙編，你必須優化程式的每一行。但這麼做會浪費你大部分的精力，因爲瓶頸僅僅是其中很小的那部分程式。一個更加抽象的語言能夠讓你把主要精力集中在瓶頸上， 達到事半功倍的效果。

當真正開始優化的時候，還必須從最頂端入手。 在使用各種低層次的編碼技巧 (low-level coding tricks) 之前，請先確保你已經使用了最爲高效的算法。 這麼做的潛在好處相當大 —— 甚至可能大到你都不再需要玩那些奇淫技巧。 當然本規則還是要和前一個規則保持平衡。 有些時候，關於算法的決策必須儘早進行。

13.2 編譯 (Compilation)

有五個參數可以控制程式的編譯方式： *speed* (速度)代表編譯器產生程式碼的速度；*compilation-speed* (編譯速度)代表程式被編譯的速度； *safety* (安全) 代表要對目標程式碼進行錯誤檢查的數量； *space* (空間)代表目標程式碼的大小和記憶體需求量；最後，*debug* (除錯)代表爲了除錯而保留的資訊量。

> **Note:** 互動與直譯 (INTERACTIVE VS. INTERPRETED)
>
> Lisp 是一種互動式語言 (Interactive Language)，但是互動式的語言不必都是直譯型的。早期的 Lisp 都通過直譯器實現，因此認爲 Lisp 的特質都依賴於它是被直譯的想法就這麼產生了。但這種想法是錯誤的：Common Lisp 既是編譯型語言，又是直譯型語言。
>
> 至少有兩種 Common Lisp 實現甚至都不包含直譯器。在這些實現中，輸入到頂層的表達式在求值前會被編譯。因此，把頂層叫做直譯器的這種說法，不僅是落伍的，甚至還是錯誤的。

編譯參數不是真正的變數。它們在宣告中被分配從 `0` (最不重要) 到 `3` (最重要) 的權值。如果一個主要的瓶頸發生在某個函數的內層迴圈中，我們或許可以添加如下的宣告：

```
(defun bottleneck (...)
  (do (...)
      (...)
    (do (...)
        (...)
      (declare (optimize (speed 3) (safety 0)))
      ...)))
```

一般情況下，應該在程式寫完並且經過完善測試之後，才考慮加上那麼一句宣告。

要讓程式在任何情況下都儘可能地快，可以使用如下宣告：

```
(declaim (optimize (speed 3)
                   (compilation-speed 0)
                   (safety 0)
                   (debug 0)))
```

考慮到前面提到的瓶頸規則 [1]，這種苛刻的做法可能並沒有什麼必要。

另一種特別重要的優化就是由 Lisp 編譯器完成的尾遞迴優化。當 *speed* (速度)的權值最大時，所有支持尾遞迴優化的編譯器都將保證對程式碼進行這種優化。

如果在一個呼叫返回時呼叫者中沒有殘餘的計算，該呼叫就被稱爲尾遞迴。下面的程式返回列表的長度：

```
(defun length/r (lst)
  (if (null lst)
      0
      (1+ (length/r (cdr lst)))))
```

這個遞迴呼叫不是尾遞迴，因爲當它返回以後，它的值必須傳給 `1+` 。相反，這是一個尾遞迴的版本，

```
(defun length/rt (lst)
  (labels ((len (lst acc)
             (if (null lst)
                 acc
                 (len (cdr lst) (1+ acc)))))
    (len lst 0)))
```

更準確地說，區域函數 `len` 是尾遞迴呼叫，因爲當它返回時，呼叫函數已經沒什麼事情可做了。 和 `length/r` 不同的是，它不是在遞迴回溯的時候構建返回值，而是在遞迴呼叫的過程中積累返回值。 在函數的最後一次遞迴呼叫結束之後， `acc` 參數就可以作爲函數的結果值被返回。

出色的編譯器能夠將一個尾遞迴編譯成一個跳轉 (goto)，因此也能將一個尾遞迴函數編譯成一個迴圈。在典型的機器語言程式碼中，當第一次執行到表示 `len` 的指令片段時，棧上會有資訊指示在返回時要做些什麼。由於在遞迴呼叫後沒有殘餘的計算，該資訊對第二層呼叫仍然有效：第二層呼叫返回後我們要做的僅僅就是從第一層呼叫返回。因此，當進行第二層呼叫時，我們只需給參數設置新的值，然後跳轉到函數的起始處繼續執行就可以了，沒有必要進行真正的函數呼叫。

另一個利用函數呼叫抽象，卻又沒有開銷的方法是使函數內聯編譯。對於那些呼叫開銷比函數體的執行代價還高的小型函數來說，這種技術非常有價值。例如，以下程式用來

判斷列表是否僅有一個元素：

```
(declaim (inline single?))

(defun single? (lst)
  (and (consp lst) (null (cdr lst))))
```

因爲這個函數是在全局被宣告爲內聯的，引用了 `single?` 的函數在編譯後將不需要真正的函數呼叫。 [2] 如果我們定義一個呼叫它的函數，

```
(defun foo (x)
  (single? (bar x)))
```

當 `foo` 被編譯後， `single?` 函數體中的程式碼將會被編譯進 `foo` 的函數體，就好像我們直接寫了以下的程式一樣：

```
(defun foo (x)
  (let ((lst (bar x)))
    (and (consp lst) (null (cdr lst)))))
```

內聯編譯有兩個限制。首先，遞迴函數不能內聯。其次，如果一個內聯函數被重新定義，我們就必須重新編譯呼叫它的任何函數，否則呼叫仍然使用原來的定義。

在一些早期的 Lisp 方言中，有時候會使用宏（ 10.2 節）來避免函數呼叫。這種做法在 Common Lisp 中通常是沒有必要的。

不同 Lisp 編譯器的優化方式千差萬別。如果你想了解你的編譯器爲某個函數生成的程式碼，試著呼叫 `disassemble` 函數：它接受一個函數或者函數名，並顯示該函數編譯後的形式。即便你看到的東西是完全無法理解的，你仍然可以使用 `disassemble` 來判斷宣告是否起效果：編譯函數的兩個版本，一個使用優化宣告，另一個不使用優化宣告，然後觀察由 `disassemble` 顯示的兩組程式之間是否有差異。同樣的技巧也可以用於檢驗函數是否被內聯編譯。不論情況如何，都請優先考慮使用編譯參數，而不是手動調優的方式來優化程式。

13.3 型別宣告 (Type Declarations)

如果 Lisp 不是你所學的第一門編程語言，那麼你也許會感到困惑，爲什麼這本書還沒說到型別宣告這件事來？畢竟，在很多流行的編程語言中，型別宣告是必須要做的。

在多數編程語言裡，你必須爲每個變數宣告型別，並且變數也只可以持有與該型別相一致的值。這種語言被稱爲強型別(*strongly typed*) 語言。除了給程式設計師們徒增了許多負擔外，這種方式還限制了你能做的事情。使用這種語言，很難寫出那些需要多種型別

的參數一起工作的函數，也很難定義出可以包含不同種類元素的資料結構。當然，這種方式也有它的優勢，比如無論何時當編譯器碰到一個加法運算，它都能夠事先知道這是一個什麼型別的加法運算。如果兩個參數都是整數型別，編譯器可以直接在目標程式碼中生成一個固定 (hard-wire) 的整數加法運算。

正如 2.15 節所講，Common Lisp 使用一種更加靈活的方式：顯式型別 (manifest typing) [3] 。有型別的是值而不是變數。變數可以用於任何型別的物件。

當然，這種靈活性需要付出一定的速度作爲代價。由於 `+` 可以接受好幾種不同型別的數，它不得不在運行時查看每個參數的型別來決定採用哪種加法運算。

在某些時候，如果我們要執行的全都是整數的加法，那麼每次查看參數型別的這種做法就說不上高效了。Common Lisp 處理這種問題的方法是：讓程式設計師儘可能地提示編譯器。比如說，如果我們提前就能知道某個加法運算的兩個參數是定長數 (fixnums) ，那麼就可以對此進行宣告，這樣編譯器就會像 C 語言的那樣爲我們生成一個固定的整數加法運算。

因爲顯式型別也可以通過宣告型別來生成高效的程式碼，所以強型別和顯式型別兩種方式之間的差別並不在於運行速度。真正的區別是，在強型別語言中，型別宣告是強制性的，而顯式型別則不強加這樣的要求。 在 Common Lisp 中，型別宣告完全是可選的。它們可以讓程式運行的更快，但(除非錯誤)不會改變程式的行爲。

全局宣告以 `declaim` 伴隨一個或多個宣告的形式來實現。 一個型別宣告是一個列表，包含了符號 `type` ，後跟一個型別名，以及一個或多個變數組成。舉個例子，要爲一個全局變數宣告型別，可以這麼寫：

```
(declaim (type fixnum *count*))
```

在 ANSI Common Lisp 中，可以省略 `type` 符號，將宣告簡寫爲：

```
(declaim (fixnum *count*))
```

區域宣告通過 `declare` 完成，它接受的參數和 `declaim` 的一樣。宣告可以放在那些創建變數的 程式碼體之前：如 `defun` 、 `lambda` 、 `let` 、 `do` ，諸如此類。比如說，要把一個函數的參數宣告爲定長數，可以這麼寫：

```
(defun poly (a b x)
  (declare (fixnum a b x))
  (+ (* a (expt x 2)) (* b x)))
```

在型別宣告中的變數名指的就是該宣告所在的上下文中的那個變數 —— 那個通過賦值可以改變它的值的變數。

你也可以通過 `the` 爲某個表達式的值宣告型別。如果我們提前就知道 `a` 、 `b` 和 `x` 是足夠小的定長數，並且它們的和也是定長數的話，那麼可以進行以下宣告：

```
(defun poly (a b x)
  (declare (fixnum a b x))
  (the fixnum (+ (the fixnum (* a (the fixnum (expt x 2))))
                 (the fixnum (* b x)))))
```

看起來是不是很笨拙啊？幸運的是有兩個原因讓你很少會這樣使用 `the` 把你的數值運算程式碼變得散亂不堪。其一是很容易通過宏，來幫你插入這些宣告。其二是某些實現使用了特殊的技巧，即便沒有型別宣告的定長數運算也能足夠快。

Common Lisp 中有相當多的型別 —— 恐怕有無數種型別那麼多，如果考慮到你可以自己定義新的型別的話。　型別宣告只在少數情況下至關重要，可以遵照以下兩條規則來進行：

1. 當函數可以接受若干不同型別的參數(但不是所有型別)時，可以對參數的型別進行宣告。如果你知道一個對 `+` 的呼叫總是接受定長數型別的參數，或者一個對 `aref` 的呼叫第一個參數總是某種特定種類的陣列，那麼進行型別宣告是值得的。
2. 通常來說，只有對型別層級中接近底層的型別進行宣告，才是值得的：將某個東西的型別宣告爲 `fixnum` 或者 `simple-array` 也許有用，但將某個東西的型別宣告爲 `integer` 或者 `sequence` 或許就沒用了。

型別宣告對內容複雜的物件特別重要，這包括陣列、結構和物件實體。這些宣告可以在兩個方面提升效率：除了可以讓編譯器來決定函數參數的型別以外，它們也使得這些物件可以在記憶體中更高效地表示。

如果對陣列元素的型別一無所知的話，這些元素在記憶體中就不得不用一塊指標來表示。但假如預先就知道陣列包含的元素僅僅是 —— 比方說 —— 雙精度浮點數 (double-floats)，那麼這個陣列就可以用一組實際的雙精度浮點數來表示。這樣陣列將佔用更少的空間，因爲我們不再需要額外的指標指向每一個雙精度浮點數；同時，對陣列元素的存取也將更快，因爲我們不必沿著指標去讀取和寫元素。

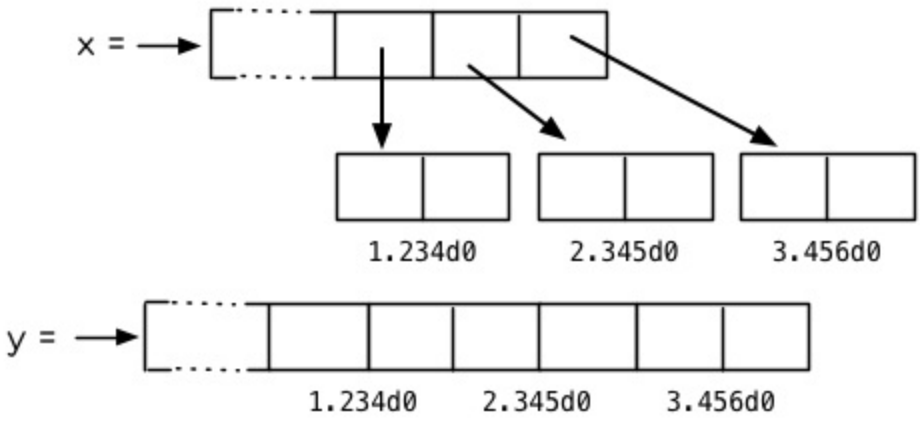

圖 **13.1**：指定元素型別的效果

你可以通過 `make-array` 的 `:element-type` 參數指定陣列包含值的種類。這樣的陣列被稱爲特化陣列(specialized array)。圖 13.1 爲我們示範了如下程式碼在多數實現上求值後發生的事情：

```
(setf x (vector 1.234d0 2.345d0 3.456d0)
      y (make-array 3 :element-type 'double-float)
      (aref y 0) 1.234d0
      (aref y 1) 2.345d0
      (aref y 2) 3.456d0))
```

圖 13.1 中的每一個矩形方格代表記憶體中的一個字 (a word of memory)。這兩個陣列都由未特別指明長度的頭部 (header) 以及後續三個元素的某種表示構成。對於 `x` 來說，每個元素都由一個指標表示。此時每個指標碰巧都指向雙精度浮點數，但實際上我們可以存儲任何型別的物件到這個向量中。對 `y` 來說，每個元素實際上都是雙精度浮點數。 `y` 更快而且佔用更少空間，但意味著它的元素只能是雙精度浮點數。

注意我們使用 `aref` 來引用 `y` 的元素。一個特化的向量不再是一個簡單向量，因此我們不再能夠通過 `svref` 來引用它的元素。

除了在創建陣列時指定元素的型別，你還應該在使用陣列的程式碼中，宣告陣列的維度以及它的元素型別。一個完整的向量宣告如下：

```
(declare (type (vector fixnum 20) v))
```

以上程式碼宣告了一個僅含有定長數，並且長度固定爲 `20` 的向量。

```
(setf a (make-array '(1000 1000)
                    :element-type 'single-float
                    :initial-element 1.0s0))

(defun sum-elts (a)
  (declare (type (simple-array single-float (1000 1000))
                 a))
  (let ((sum 0.0s0))
    (declare (type single-float sum))
    (dotimes (r 1000)
      (dotimes (c 1000)
        (incf sum (aref a r c))))
    sum))
```

圖 **13.2** 對陣列元素求和

最爲通用的陣列宣告形式由陣列型別以及緊接其後的元素型別和一個維度列表構成：

```
(declare (type (simple-array fixnum (4 4)) ar))
```

圖 13.2 示範了如何創建一個 1000×1000 的單精度浮點數陣列，以及如何編寫一個將該陣列元素相加的函數。陣列以列主序 (row-major order)存儲，遍歷時也應儘可能以此序進行。

我們將用 `time` 來比較 `sum-elts` 在有宣告和無宣告兩種情況下的性能。 `time` 宏顯示表達式求值所花費時間的某種度量(取決於實現)。對被編譯的函數求取時間才是有意義的。在某個實現中，如果我們以獲取最快速

程式碼的編譯參數編譯 `sum-elts` ，它將在不到半秒的時間內返回：

```
> (time (sum-elts a))
User Run Time = 0.43 seconds
1000000.0
```

如果我們把 *sum-elts* 中的型別宣告去掉並重新編譯它，同樣的計算將花費超過5秒的時間：

```
> (time (sum-elts a))
User Run Time = 5.17 seconds
1000000.0
```

型別宣告的重要性 —— 特別是對陣列和數來說 —— 怎麼強調都不過分。上面的例子中，僅僅兩行程式碼，就可以讓 `sum-elts` 變快 12 倍。

13.4 避免垃圾 (Garbage Avoidance)

Lisp 除了可以讓你推遲考慮變數的型別以外，它還允許你推遲對記憶體分配的考慮。在程式的早期階段，暫時忽略記憶體分配和臭蟲等問題，將有助於解放你的想象力。等到程式基本固定下來以後，就可以開始考慮怎麼減少動態分配，從而讓程式運行得更快。

但是，並不是構造（consing）用得少的程式就一定快。多數 Lisp 實現一直使用著差勁的垃圾回收器，在這些實現中，過多的記憶體分配容易讓程式運行變得緩慢。因此，『高效的程式應該儘可能地減少 `cons` 的使用』這種觀點，逐漸成爲了一種傳統。最近這種傳統開始有所改變，因爲一些實現已經用上了相當先進（sophisticated）的垃圾回收器，它們實行一種更爲高效的策略：創建新的物件，用完之後拋棄而不是進行回收。

本節介紹了幾種方法，用於減少程式中的構造。但構造數量的減少是否有利於加快程式的運行，這一點最終還是取決於實現。最好的辦法就是自己去試一試。

減少構造的辦法有很多種。有些辦法對程式的修改非常少。 例如，最簡單的方法就是

使用破壞性函數。下表羅列了一些常用的函數，以及這些函數對應的破壞性版本。

安全	破壞性
append	nconc
reverse	nreverse
remove	delete
remove-if	delete-if
remove-duplicates	delete-duplicates
subst	nsubst
subst-if	nsubst-if
union	nunion
intersection	nintersection
set-difference	nset-difference

當確認修改列表是安全的時候，可以使用 `delete` 替換 `remove` ，用 `nreverse` 替換 `reverse` ，諸如此類。

即便你想完全擺脫構造，你也不必放棄在執行時創建物件的可能性。你需要做的是避免在運行中爲它們分配空間和通過垃圾回收收回空間。通用方案是你自己預先分配記憶體塊 (block of memory)，以及明確回收用過的塊。預先可能意味著在編譯期或者某些初始化例程中。具體情況還應具體分析。

例如，當情況允許我們利用一個有限大小的堆棧時，我們可以讓堆棧在一個已經分配了空間的向量中增長或縮減，而不是構造它。Common Lisp 內建支持把向量作爲堆棧使用。如果我們傳給 `make-array` 可選的 `fill-pointer` 參數，我們將得到一個看起來可擴展的向量。 `make-array` 的第一個參數指定了分配給向量的存儲量，而 `fill-pointer` 指定了初始有效長度：

```
> (setf *print-array* t)
T
> (setf vec (make-array 10 :fill-pointer 2
                           :initial-element nil))
#(NIL NIL)
```

我們剛剛創建的向量對於操作序列的函數來說，仍好像只含有兩個元素，

```
> (length vec)
2
```

但它能夠增長直到十個元素。因爲 `vec` 有一個填充指標，我們可以使用 `vector-push` 和 `vector-pop` 函數推入和彈出元素，就像它是一個列表一樣：

```
> (vector-push 'a vec)
2
> vec
#(NIL NIL A)
> (vector-pop vec)
A
> vec
#(NIL NIL)
```

當我們呼叫 `vector-push` 時，它增加填充指標並返回它過去的值。只要填充指標小於 `make-array` 的第一個參數，我們就可以向這個向量中推入新元素；當空間用盡時，`vector-push` 返回 `nil`。目前我們還可以向 `vec` 中推入八個元素。

使用帶有填充指標的向量有一個缺點，就是它們不再是簡單向量了。我們不得不使用 `aref` 來代替 `svref` 引用元素。代價需要和潛在的收益保持平衡。

```
(defconstant dict (make-array 25000 :fill-pointer 0))

(defun read-words (from)
  (setf (fill-pointer dict) 0)
  (with-open-file (in from :direction :input)
    (do ((w (read-line in nil :eof)
            (read-line in nil :eof)))
        ((eql w :eof))
      (vector-push w dict))))

(defun xform (fn seq) (map-into seq fn seq))

(defun write-words (to)
  (with-open-file (out to :direction :output
                          :if-exists :supersede)
    (map nil #'(lambda (x)
                 (fresh-line out)
                 (princ x out))
             (xform #'nreverse
                    (sort (xform #'nreverse dict)
                          #'string<)))))
```

圖 **13.3** 生成同韻字辭典

當應用涉及很長的序列時，你可以用 `map-into` 代替 `map`。`map-into` 的第一個參數不是一個序列型別，而是用來存儲結果的，實際的序列。這個序列可以是該函數接受的其他序列參數中的任何一個。所以，打個比方，如果你想爲一個向量的每個元素加 1，你可以這麼寫：

```
(setf v (map-into v #'1+ v))
```

圖 13.3 展示了一個使用大向量應用的例子：一個生成簡單的同韻字辭典 (或者更確切的

說，一個不完全韻辭典)的程式。函數 `read-line` 從一個每行僅含有一個單詞的檔案中讀取單詞，而函數 `write-words` 將它們按照字母的逆序打印出來。比如，輸出的起始可能是

```
a amoeba alba samba marimba...
```

結束是

```
...megahertz gigahertz jazz buzz fuzz
```

利用填充指標和 `map-into`，我們可以把程式寫的既簡單又高效。

在數值應用中要當心大數 (bignums)。大數運算需要構造，因此也就會比較慢。即使程式的最後結果爲大數，但是，通過調整計算，將中間結果保存在定長數中，這種優化也是有可能的。

另一個避免垃圾回收的方法是，鼓勵編譯器在棧上分配物件而不是在堆上。如果你知道只是臨時需要某個東西，你可以通過將它宣告爲 `dynamic extent` 來避免在堆上分配空間。

通過一個動態範圍 (dynamic extent)變數宣告，你告訴編譯器，變數的值應該和變數保持相同的生命期。什麼時候值的生命期比變數長呢？這裡有個例子：

```
(defun our-reverse (lst)
  (let ((rev nil))
    (dolist (x lst)
      (push x rev))
    rev))
```

在 `our-reverse` 中，作爲參數傳入的列表以逆序被收集到 `rev` 中。當函數返回時，變數 `rev` 將不復存在。然而，它的值 —— 一個逆序的列表 —— 將繼續存活：它被送回呼叫函數，一個知道它的命運何去何從的地方。

相比之下，考慮如下 `adjoin` 實現：

```
(defun our-adjoin (obj lst &rest args)
  (if (apply #'member obj lst args)
      lst
      (cons obj lst)))
```

在這個例子裡，我們可以從函數的定義看出，`args` 參數中的值 (列表) 哪兒也沒去。它不必比存儲它的變數活的更久。在這種情形下把它宣告爲動態範圍的就比較有意義。如果我們加上這樣的宣告：

```
(defun our-adjoin (obj lst &rest args)
  (declare (dynamic-extent args))
  (if (apply #'member obj lst args)
      lst
      (cons obj lst)))
```

那麼編譯器就可以 (但不是必須)在棧上爲 `args` 分配空間，在 `our-adjoin` 返回後，它將自動被釋放。

13.5 範例: 存儲池 (Example: Pools)

對於涉及資料結構的應用，你可以通過在一個存儲池 (pool)中預先分配一定數量的結構來避免動態分配。當你需要一個結構時，你從池中取得一份，當你用完後，再把它送回池中。爲了示範存儲池的使用，我們將快速的編寫一段記錄港口中船舶數量的程式原型 (prototype of a program)，然後用存儲池的方式重寫它。

```
(defparameter *harbor* nil)

(defstruct ship
  name flag tons)

(defun enter (n f d)
  (push (make-ship :name n :flag f :tons d)
        *harbor*))

(defun find-ship (n)
  (find n *harbor* :key #'ship-name))

(defun leave (n)
  (setf *harbor*
        (delete (find-ship n) *harbor*)))
```

圖 **13.4** 港口

圖 13.4 中展示的是第一個版本。 全局變數 `harbor` 是一個船隻的列表， 每一艘船隻由一個 `ship` 結構表示。 函數 `enter` 在船只進入港口時被呼叫； `find-ship` 根據給定名字 (如果有的話) 來尋找對應的船隻；最後， `leave` 在船隻離開港口時被呼叫。

一個程式的初始版本這麼寫簡直是棒呆了，但它會產生許多的垃圾。當這個程式運行時，它會在兩個方面構造：當船只進入港口時，新的結構將會被分配；而 `harbor` 的每一次增大都需要使用構造。

我們可以通過在編譯期分配空間來消除這兩種構造的源頭 (sources of consing)。圖 13.5 展示了程式的第二個版本，它根本不會構造。

```
(defconstant pool (make-array 1000 :fill-pointer t))

(dotimes (i 1000)
  (setf (aref pool i) (make-ship)))

(defconstant harbor (make-hash-table :size 1100
                                     :test #'eq))

(defun enter (n f d)
  (let ((s (if (plusp (length pool))
               (vector-pop pool)
               (make-ship))))
    (setf (ship-name s)        n
          (ship-flag s)        f
          (ship-tons s)        d
          (gethash n harbor) s)))

(defun find-ship (n) (gethash n harbor))

(defun leave (n)
  (let ((s (gethash n harbor)))
    (remhash n harbor)
    (vector-push s pool)))
```

圖 **13.5** 港口（第二版）

嚴格說來，新的版本仍然會構造，只是不在運行期。在第二個版本中， `harbor` 從列表變成了雜湊表，所以它所有的空間都在編譯期分配了。一千個 `ship` 結構體也會在編譯期被創建出來，並被保存在向量池(vector pool) 中。(如果 `:fill-pointer` 參數爲 `t` ，填充指標將指向向量的末尾。) 此時，當 `enter` 需要一個新的結構時，它只需從池中取來一個便是，無須再呼叫 `make-ship` 。 而且當 `leave` 從 `harbor` 中移除一艘 `ship` 時，它把它送回池中，而不是拋棄它。

我們使用存儲池的行爲實際上是肩負起記憶體管理的工作。這是否會讓我們的程式更快仍取決於我們的 Lisp 實現怎樣管理記憶體。總的說來，只有在那些仍使用著原始垃圾回收器的實現中，或者在那些對 GC 的不可預見性比較敏感的實時應用中才值得一試。

13.6 快速運算子 (Fast Operators)

本章一開始就宣稱 Lisp 是兩種不同的語言。就某種意義來講這確實是正確的。如果你仔細看過 Common Lisp 的設計，你會發現某些特性主要是爲了速度，而另外一些主要爲了便捷性。

例如，你可以通過三個不同的函數取得向量給定位置上的元素： `elt` 、 `aref` 、 `svref` 。如此的多樣性允許你把一個程式的性能提升到極致。 所以如果你可以使用 `svref` ，完事兒！ 相反，如果對某段程式來說速度很重要的話，或許不應該呼叫 `elt` ，它既可

以用於陣列也可以用於列表。

對於列表來說，你應該呼叫 `nth` ，而不是 `elt` 。然而只有單一的一個函數 —— `length` —— 用於計算任何一個序列的長度。爲什麼 Common Lisp 不單獨爲列表提供一個特定的版本呢？因爲如果你的程式正在計算一個列表的長度，它在速度上已經輸了。在這個例子中，就像許多其他的例子一樣，語言的設計暗示了哪些會是快速的而哪些不是。

另一對相似的函數是 `eql` 和 `eq` 。前者是驗證同一性 (identity) 的默認判斷式，但如果你知道參數不會是字元或者數字時，使用後者其實更快。兩個物件 *eq* 只有當它們處在相同的記憶體位置上時才成立。數字和字元可能不會與任何特定的記憶體位置相關，因此 `eq` 不適用於它們 (即便多數實現中它仍然能用於定長數)。對於其他任何種類的參數，`eq` 和 `eql` 將返回相同的值。

使用 `eq` 來比較物件總是最快的，因爲 Lisp 所需要比較的僅僅是指向物件的指標。因此 `eq` 雜湊表 (如圖 13.5 所示) 應該會提供最快的存取。 在一個 `eq` 雜湊表中， `gethash` 可以只根據指標查找，甚至不需要查看它們指向的是什麼。然而，存取不是唯一要考慮的因素； *eq* 和 *eql* 雜湊表在拷貝型垃圾回收算法 (copying garbage collection algorithm)中會引起額外的開銷，因爲垃圾回收後需要對一些哈希值重新進行計算 (rehashing)。如果這變成了一個問題，最好的解決方案是使用一個把定長數作爲鍵值的 `eql` 雜湊表。

當被調函數有一個剩餘參數時，呼叫 `reduce` 可能是比 `apply` 更高效的一種方式。例如，相比

```
(apply #'+ '(1 2 3))
```

寫成如下可以更高效：

```
(reduce #'+ '(1 2 3))
```

它不僅有助於呼叫正確的函數，還有助於按照正確的方式呼叫它們。餘留、可選和關鍵字參數 是昂貴的。只使用普通參數，函數呼叫中的參量會被呼叫者簡單的留在被調者能夠找到的地方。但其他種類的參數涉及運行時的處理。關鍵字參數是最差的。針對內建函數，優秀的編譯器採用特殊的辦法把使用關鍵字參量的呼叫編譯成快速程式碼 (fast code)。但對於你自己編寫的函數，避免在程式中對速度敏感的部分使用它們只有好處沒有壞處。另外，不把大量的參量都放到餘留參數中也是明智的舉措，如果這可以避免的話。

不同的編譯器有時也會有一些它們獨門優化。例如，有些編譯器可以針對鍵值是一個狹小範圍中的整數的 `case` 語句進行優化。查看你的用戶手冊來了解那些實現特有的優化的建議吧。

13.7 二階段開發 (Two-Phase Development)

在以速度至上的應用中，你也許想要使用諸如 C 或者彙編這樣的低級語言來重寫一個 Lisp 程式的某部分。你可以對用任何語言編寫的程式使用這一技巧 —— C 程式的關鍵部分經常用彙編重寫 —— 但語言越抽象，用兩階段（two phases）開發程式的好處就越明顯。

Common Lisp 沒有規定如何集成其他語言所編寫的程式碼。這部分留給了實現決定，而幾乎所有的實現都提供了某種方式來實現它。

使用一種語言編寫程式然後用另一種語言重寫它其中部分看起來可能是一種浪費。事實上，經驗顯示這是一種好的開發軟體的方式。先針對功能、然後是速度比試著同時達成兩者來的簡單。

如果編程完全是一個機械的過程 —— 簡單的把規格說明翻譯爲程式碼 —— 在一步中把所有的事情都搞定也許是合理的。但編程永遠不是如此。不論規格說明多麼精確， 編程總是涉及一定量的探索 —— 通常比任何人能預期到的還多的多。

一份好的規格說明，也許會讓編程看起來像是簡單的把它們翻譯成程式碼的過程。這是一個普遍的誤區。編程必定涉及探索，因爲規格說明必定含糊不清。如果它們不含糊的話，它們就都算不上規格說明。

在其他領域，儘可能精準的規格說明也許是可取的。如果你要求一塊金屬被切割成某種形狀，最好準確的說出你想要的。但這個規則不適用於軟體，因爲程式和規格說明由相同的東西構成：文字。你不可能編寫出完全合意的規格說明。如果規格說明有那麼精確的話，它們就變成程式了。 λ [http://acl.readthedocs.org/en/latest/zhTW/notes.html#notes-229]

對於存在著可觀數量的探索的應用（再一次，比任何人承認的還要多，將實現分成兩個階段是值得的。而且在第一階段中你所使用的手段 (medium）不必就是最後的那個。例如，製作銅像的標準方法是先從粘土開始。你先用粘土做一個塑像出來，然後用它做一個模子，在這個模子中鑄造銅像。在最後的塑像中是沒有丁點粘土的，但你可以從銅像的形狀中認識到它發揮的作用。試想下從一開始就只用一塊兒銅和一個鑿子來製造這麼個一模一樣的塑像要多難啊！出於相同的原因，首先用 Lisp 來編寫程式，然後用 C 改寫它，要比從頭開始就用 C 編寫這個程式要好。

Chapter 13 總結 (Summary)

1. 不應過早開始優化，應該關注瓶頸，而且應該從算法開始。
2. 有五個不同的參數控制編譯。它們可以在本地宣告也可以在全局宣告。

3. 優秀的編譯器能夠優化尾遞迴，將一個尾遞迴的函數轉換爲一個迴圈。內聯編譯是另一種避免函數呼叫的方法。
4. 型別宣告並不是必須的，但它們可以讓一個程式更高效。型別宣告對於處理數值和陣列的程式碼特別重要。
5. 少的構造可以讓程式更快，特別是在使用著原始的垃圾回收器的實現中。解決方案是使用破壞性函數、預先分配空間塊、以及在棧上分配。
6. 某些情況下，從預先分配的存儲池中提取物件可能是有價值的。
7. Common Lisp 的某些部分是爲了速度而設計的，另一些則爲了靈活性。
8. 編程必定存在探索的過程。探索和優化應該被分開 —— 有時甚至需要使用不同的語言。

Chapter 13 練習 (Exercises)

1. 檢驗你的編譯器是否支持 (observe)內斂宣告。
2. 將下述函數重寫爲尾遞迴形式。它被編譯後能快多少？

```
(defun foo (x)
  (if (zerop x)
      0
      (1+ (foo (1- x)))))
```

注意：你需要增加額外的參數。

3. 爲下述程式增加宣告。你能讓它們變快多少？

```
(a) 在 5.7 節中的日期運算程式碼。
(b) 在 9.8 節中的光線跟蹤器 (ray-tracer)。
```

4. 重寫 3.15 節中的廣度優先搜索的程式，讓它儘可能減少使用構造。
5. 使用存儲池修改 4.7 節中的二元搜索的程式碼。

腳註

[1] 較早的實現或許不提供 `declaim`；需要使用 `proclaim` 並且引用這些參量 (quote the argument)。

[2] 爲了讓內聯宣告 (inline declaration) 有效，你同時必須設置編譯參數，告訴它你想獲得最快的程式碼。

[3] 有兩種方法可以描述 Lisp 宣告型別 (typing) 的方式：從型別資訊被存放的位置或者從它被使用的時間。顯示型別 (manifest typing) 的意思是型別資訊與資料物件 (data objects) 綁定，而運行時型別(run-time typing) 的意思是型別資訊在運行時被使用。實際上，兩者是一回事兒。

第十四章：進階議題

本章是選擇性閱讀的。本章描述了 Common Lisp 裡一些更深奧的特性。Common Lisp 像是一個冰山：大部分的功能對於那些永遠不需要他們的多數用戶是看不見的。你或許永遠不需要自己定義包 (Package)或讀取宏 (read-macros)，但當你需要時，有些例子可以讓你參考是很有用的。

14.1 型別標識符 (Type Specifiers)

型別在 Common Lisp 裡不是物件。舉例來說，沒有物件對應到 `integer` 這個型別。我們像是從 `type-of` 函數裡所獲得的，以及作爲傳給像是 `typep` 函數的參數，不是一個型別，而是一個型別標識符 (type specifier)。

一個型別標識符是一個型別的名稱。最簡單的型別標識符是像是 `integer` 的符號。這些符號形成了 Common Lisp 裡的型別層級。在層級的最頂端是型別 `t` —— 所有的物件皆爲型別 `t` 。而型別層級不是一棵樹。從 `nil` 至頂端有兩條路，舉例來說：一條從 `atom` ，另一條從 `list` 與 `sequence` 。

一個型別實際上只是一個物件集合。這意味著有多少型別就有多少個物件的集合：一個無窮大的數目。我們可以用原子的型別標識符 (atomic type specifiers)來表示某些集合：比如 `integer` 表示所有整數集合。但我們也可以建構一個複合型別標識符 (compound type specifiers)來參照到任何物件的集合。

舉例來說，如果 `a` 與 `b` 是兩個型別標識符，則 `(or a b)` 表示分別由 `a` 與 `b` 型別所表示的聯集 (union)。也就是說，一個型別 `(or a b)` 的物件是型別 `a` 或 型別 `b` 。

如果 `circular?` 是一個對於 `cdr` 爲環狀的列表返回真的函數，則你可以使用適當的序列集合來表示： [1]

```
(or vector (and list (not (satisfies circular?))))
```

某些原子的型別標識符也可以出現在複合型別標識符。要表示介於 1 至 100 的整數（包含），我們可以用：

```
(integer 1 100)
```

這樣的型別標識符用來表示一個有限的型別 (finite type)。

在一個複合型別標識符裡，你可以通過在一個參數的位置使用 `*` 來留下某些未指定的資

訊。所以

```
(simple-array fixnum (* *))
```

描述了指定給 `fixnum` 使用的二維簡單陣列 (simple array)集合，而

```
(simple-array fixnum *)
```

描述了指定給 `finxnum` 使用的簡單陣列集合 (前者的超型別 「supertype」)。尾隨的星號可以省略，所以上個例子可以寫爲：

```
(simple-array fixnum)
```

若一個複合型別標識符沒有傳入參數，你可以使用一個原子。所以 `simple-array` 描述了所有簡單陣列的集合。

如果有某些複合型別標識符你想重複使用，你可以使用 `deftype` 定義一個縮寫。這個宏與 `defmacro` 相似，但會展開成一個型別標識符，而不是一個表達式。通過表達

```
(deftype proseq ()
         '(or vector (and list (not (satisfies circular?)))))
```

我們定義了 `proseq` 作爲一個新的原子型別標識符：

```
> (typep #(1 2) 'proseq)
T
```

如果你定義一個接受參數的型別標識符，參數會被視爲 Lisp 形式（即沒有被求值），與 `defmacro` 一樣。所以

```
(deftype multiple-of (n)
  `(and integer (satisfies (lambda (x)
                             (zerop (mod x ,n))))))
```

(譯註: 注意上面

程式碼是使用反引號 `)

定義了 *(multiple-of n)* 當成所有 `n` 的倍數的標識符：

```
> (type 12 '(multiple-of 4))
T
```

型別標識符會被直譯 (interpreted)，因此很慢，所以通常你最好定義一個函數來處理這類的測試。

14.2 二進制流 (Binary Streams)

第 7 章曾提及的流有二進制流 (binary streams)以及字元流 (character streams)。一個二進制流是一個整數的來源及/或終點，而不是字元。你通過指定一個整數的子型別來創建一個二進制流 —— 當你打開流時，通常是用 `unsigned-byte` —— 來作爲 `:element-type` 的參數。

關於二進制流的 I/O 函數僅有兩個， `read-byte` 以及 `write-byte` 。所以下面是如何定義複製一個檔案的函數：

```
(defun copy-file (from to)
  (with-open-file (in from :direction :input
                           :element-type 'unsigned-byte)
    (with-open-file (out to :direction :output
                            :element-type 'unsigned-byte)
      (do ((i (read-byte in nil -1)
              (read-byte in nil -1)))
          ((minusp i))
        (declare (fixnum i))
        (write-byte i out)))))
```

僅通過指定 `unsigned-byte` 給 `:element-type` ，你讓操作系統選擇一個字節 (byte)的長度。舉例來說，如果你明確地想要讀寫 7 比特的整數，你可以使用：

```
(unsigned-byte 7)
```

來傳給 `:element-type` 。

14.3 讀取宏 (Read-Macros)

7.5 節介紹過宏字元 (macro character)的概念，一個對於 `read` 有特別意義的字元。每一個這樣的字元，都有一個相關聯的函數，這函數告訴 `read` 當遇到這個字元時該怎麼處理。你可以變更某個已存在宏字元所相關聯的函數，或是自己定義新的宏字元。

函數 `set-macro-character` 提供了一種方式來定義讀取宏 (read-macros)。它接受一個字元及一個函數，因此當 `read` 碰到該字元時，它返回呼叫傳入函數後的結果。

Lisp 中最古老的讀取宏之一是 `'` ，即 `quote` 。我們可以定義成：

```
(set-macro-character #\'
```

```
        #'(lambda (stream char)
                  (list (quote quote) (read stream t nil t))))
```

當 read 在一個普通的語境下遇到 ' 時，它會返回在當前流和字元上呼叫這個函數的結果。(這個函數忽略了第二個參數，第二個參數永遠是引用字元。)所以當 read 看到 'a 時，會返回 (quote a) 。

譯註: read 函數接受的參數 (read &optional stream eof-error eof-value recursive)

現在我們明白了 read 最後一個參數的用途。它表示無論 read 呼叫是否在另一個 read 裡。傳給 read 的參數在幾乎所有的讀取宏裡皆相同：傳入參數有流 (stream)；接著是第二個參數， t ，說明了 read 若讀入的東西是 end-of-file 時，應不應該報錯；第三個參數說明了不報錯時要返回什麼，因此在這裡也就不重要了；而第四個參數 t 說明了這個 read 呼叫是遞迴的。

(譯註：困惑的話可以看看 read 的定義 [https://gist.github.com/3467235])

你可以（通過使用 make-dispatch-macro-character ）來定義你自己的派發宏字元（dispatching macro character），但由於 # 已經是一個宏字元，所以你也可以直接使用。六個 # 打頭的組合特別保留給你使用： #! 、 #? 、 ##[、 ##] 、 #{ 、 #} 。

你可以通過呼叫 set-dispatch-macro-character 定義新的派發宏字元組合，與 set-macro-character 類似，除了它接受兩個字元參數外。下面的

程式碼定義了 #? 作爲返回一個整數列表的讀取宏。

```
(set-dispatch-macro-character #\# #\?
  #'(lambda (stream char1 char2)
      (list 'quote
            (let ((lst nil))
              (dotimes (i (+ (read stream t nil t) 1))
                (push i lst))
              (nreverse lst)))))
```

現在 #?n 會被讀取成一個含有整數 0 至 n 的列表。舉例來說：

```
> #?7
(1 2 3 4 5 6 7)
```

除了簡單的宏字元，最常定義的宏字元是列表分隔符 (list delimiters)。另一個保留給用戶的字元組是 #{ 。以下我們定義了一種更複雜的左括號：

```
(set-macro-character #\} (get-macro-character #\)))

(set-dispatch-macro-character #\# #\{
  #'(lambda (stream char1 char2)
      (let ((accum nil)
            (pair (read-delimited-list #\} stream t)))
        (do ((i (car pair) (+ i 1)))
            ((> i (cadr pair))
             (list 'quote (nreverse accum)))
          (push i accum)))))
```

這定義了一個這樣形式 `#{x y}` 的表達式，使得這樣的表達式被讀取爲所有介於 `x` 與 `y` 之間的整數列表，包含 `x` 與 `y`：

```
> #{2 7}
(2 3 4 4 5 6 7)
```

函數 `read-delimited-list` 正是爲了這樣的讀取宏而生的。它的第一個參數是被視爲列表結束的字元。爲了使 `}` 被識別爲分隔符，必須先給它這個角色，所以程式在開始的地方呼叫了 `set-macro-character`。

如果你想要在定義一個讀取宏的檔案裡使用該讀取宏，則讀取宏的定義應要包在一個 `eval-when` 表達式裡，來確保它在編譯期會被求值。不然它的定義會被編譯，但不會被求值，直到編譯檔案被載入時才會被求值。

14.4 包 (Packages)

一個包是一個將名字映對到符號的 Lisp 物件。當前的包總是存在全局變數 `*package*` 裡。當 Common Lisp 啓動時，當前的包會是 `*common-lisp-user*`，通常稱爲用戶包 (user package)。函數 `package-name` 返回包的名字，而 `find-package` 返回一個給定名稱的包:

```
> (package-name *package*)
"COMMON-LISP-USER"
> (find-package "COMMON-LISP-USER")
#<Package "COMMON-LISP-USER" 4CD15E>
```

通常一個符號在讀入時就被 interned 至當前的包裡面了。函數 `symbol-package` 接受一個符號並返回該符號被 interned 的包。

```
(symbol-package 'sym)
#<Package "COMMON-LISP-USER" 4CD15E>
```

有趣的是，這個表達式返回它該返回的值，因爲表達式在可以被求值前必須先被讀入，

而讀取這個表達式導致 sym 被 interned。爲了之後的用途，讓我們給 sym 一個值:

```
> (setf sym 99)
99
```

現在我們可以創建及切換至一個新的包：

```
> (setf *package* (make-package 'mine
                                :use '(common-lisp)))
#<Package "MINE" 63390E>
```

現在應該會聽到詭異的背景音樂，因爲我們來到一個不一樣的世界了： 在這裡 sym 不再是本來的 sym 了。

```
MINE> sym
Error: SYM has no value
```

爲什麼會這樣？因爲上面我們設爲 99 的 sym 與 mine 裡的 sym 是兩個不同的符號。 [2] 要在用戶包之外參照到原來的 sym，我們必須把包的名字加上兩個冒號作爲前綴：

```
MINE> common-lisp-user::sym
99
```

所以有著相同打印名稱的不同符號能夠在不同的包內共存。可以有一個 sym 在 common-lisp-user 包，而另一個 sym 在 mine 包，而他們會是不一樣的符號。這就是包存在的意義。如果你在分開的包內寫你的程式，你大可放心選擇函數與變數的名字，而不用擔心某人使用了同樣的名字。即便是他們使用了同樣的名字，也不會是相同的符號。

包也提供了資訊隱藏的手段。程式應通過函數與變數的名字來參照它們。如果你不讓一個名字在你的包之外可見的話，那麼另一個包中的

程式碼就無法使用或者修改這個名字所參照的物件。

通常使用兩個冒號作爲包的前綴也是很差的風格。這麼做你就違反了包本應提供的模組性。如果你不得不使用一個雙冒號來參照到一個符號，這是因爲某人根本不想讓你用。

通常我們應該只參照被輸出 (*exported*)的符號。如果我們回到用戶包裡，並輸出一個被 interned 的符號，

```
MINE> (in-package common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (export 'bar)
T
> (setf bar 5)
```

```
5
```

我們使這個符號對於其它的包是可視的。現在當我們回到 mine ，我們可以僅使用單冒號來參照到 bar ，因爲他是一個公開可用的名字：

```
> (in-package mine)
#<Package "MINE" 63390E>
MINE> common-lisp-user:bar
5
```

通過把 bar 輸入 (import)至 mine 包，我們就能進一步讓 mine 和 user 包可以共享 bar 這個符號：

```
MINE> (import 'common-lisp-user:bar)
T
MINE> bar
5
```

在輸入 bar 之後，我們根本不需要用任何包的限定符 (package qualifier)，就能參照它了。這兩個包現在共享了同樣的符號；不可能會有一個獨立的 mine:bar 了。

要是已經有一個了怎麼辦？在這種情況下， import 呼叫會產生一個錯誤，如下面我們試著輸入 sym 時便知：

```
MINE> (import 'common-lisp-user::sym)
Error: SYM is already present in MINE.
```

在此之前，當我們試著在 mine 包裡對 sym 進行了一次不成功的求值，我們使 sym 被 interned 至 mine 包裡。而因爲它沒有值，所以產生了一個錯誤，但輸入符號名的後果就是使這個符號被 intern 進這個包。所以現在當我們試著輸入 sym 至 mine 包裡，已經有一個相同名稱的符號了。

另一個方法來獲得別的包內符號的存取權是使用(use)它：

```
MINE> (use-package 'common-lisp-user)
T
```

現在所有由用戶包 (譯註: common-lisp-user 包）所輸出的符號，可以不需要使用任何限定符在 mine 包裡使用。(如果 sym 已經被用戶包輸出了，這個呼叫也會產生一個錯誤。)

含有自帶運算子及變數名字的包叫做 common-lisp 。由於我們將這個包的名字在創建 mine 包時作爲 make-package 的 :use 參數，所有的 Common Lisp 自帶的名字在 mine 裡

都是可視的:

```
MINE> #'cons
#<Compiled-Function CONS 462A3E>
```

在編譯後的

程式碼中, 通常不會像這樣在頂層進行包的操作。更常見的是包的呼叫會包含在源檔案裡。通常，只要把 `in-package` 和 `defpackage` 放在源檔案的開頭就可以了，正如 137 頁所示。

這種由包所提供的模組性實際上有點奇怪。我們不是物件的模組 (modules)，而是名字的模組。

每一個使用了 `common-lisp` 的包，都可以存取 `cons` ，因爲 `common-lisp` 包裡有一個叫這個名字的函數。但這會導致一個名字爲 `cons` 的變數也會在每個使用了 `common-lisp` 包裡是可視的。如果包使你困惑，這就是主要的原因；因爲包不是基於物件而是基於名字。

14.5 Loop 宏 (The Loop Facility)

`loop` 宏最初是設計來幫助無經驗的 Lisp 用戶來寫出迭代的

程式碼。與其撰寫 Lisp 程式碼，你用一種更接近英語的形式來表達你的程式，然後這個形式被翻譯成 Lisp。不幸的是， `loop` 比原先設計者預期的更接近英語：你可以在簡單的情況下使用它，而不需了解它是如何工作的，但想在抽象層面上理解它幾乎是不可能的。

如果你是曾經計劃某天要理解 `loop` 怎麼工作的許多 Lisp 程式設計師之一，有一些好消息與壞消息。好消息是你並不孤單：幾乎沒有人理解它。壞消息是你永遠不會理解它，因爲 ANSI 標準實際上並沒有給出它行爲的正式規範。

這個宏唯一的實際定義是它的實現方式，而唯一可以理解它（如果有人可以理解的話）的方法是通過實體。ANSI 標準討論 `loop` 的章節大部分由例子組成，而我們將會使用同樣的方式來介紹相關的基礎概念。

第一個關於 `loop` 宏我們要注意到的是語法 (*syntax*)。一個 `loop` 表達式不是包含子表達式而是子句 (*clauses*)。這些子句不是由括號分隔出來；而是每種都有一個不同的語法。在這個方面上， `loop` 與傳統的 Algol-like 語言相似。但其它 `loop` 獨特的特性，使得它與 Algol 不同，也就是在 `loop` 宏裡調換子句的順序與會發生的事情沒有太大的關聯。

一個 `loop` 表達式的求值分爲三個階段，而一個給定的子句可以替多於一個的階段貢獻

程式碼。這些階段如下：

1. 序幕 (*Prologue*)。 被求值一次來做爲迭代過程的序幕。包括了將變數設至它們的初始值。
2. 主體 (*Body*) 每一次迭代時都會被求值。
3. 閉幕 (*Epilogue*) 當迭代結束時被求值。決定了 `loop` 表達式的返回值（可能返回多個值）。

我們會看幾個 `loop` 子句的例子，並考慮何種

程式碼會貢獻至何個階段。

舉例來說，最簡單的 `loop` 表達式，我們可能會看到像是下列的

程式碼：

```
> (loop for x from 0 to 9
        do (princ x))
0123456789
NIL
```

這個 `loop` 表達式印出從 `0` 至 `9` 的整數，並返回 `nil` 。第一個子句，

```
for x from 0 to 9
```

貢獻

程式碼至前兩個階段，導致 `x` 在序幕中被設爲 `0` ，在主體開頭與 `9` 來做比較，在主體結尾被遞增。第二個子句，

```
do (princ x)
```

貢獻

程式碼給主體。

一個更通用的 `for` 子句說明了起始與更新的形式 (initial and update form)。停止迭代可以被像是 `while` 或 `until` 子句來控制。

```
> (loop for x = 8 then (/ x 2)
        until (< x 1)
        do (princ x))
8421
```

```
NIL
```

你可以使用 and 來創建複合的 for 子句，同時初始及更新兩個變數：

```
> (loop for x from 1 to 4
        and y from 1 to 4
        do (princ (list x y)))
(1 1)(2 2)(3 3)(4 4)
NIL
```

要不然有多重 for 子句時，變數會被循序更新。

另一件在迭代

程式碼通常會做的事是累積某種值。舉例來說：

```
> (loop for x in '(1 2 3 4)
        collect (1+ x))
(2 3 4 5)
```

在 for 子句使用 in 而不是 from ，導致變數被設爲一個列表的後續元素，而不是連續的整數。

在這個情況裡， collect 子句貢獻

程式碼至三個階段。在序幕，一個匿名累加器 (anonymous accumulator)設為 nil ；在主體裡， (1+ x) 被累加至這個累加器，而在閉幕時返回累加器的值。

這是返回一個特定值的第一個例子。有用來明確指定返回值的子句，但沒有這些子句時，一個 collect 子句決定了返回值。所以我們在這裡所做的其實是重複了 mapcar 。

loop 最常見的用途大概是蒐集呼叫一個函數數次的結果：

```
> (loop for x from 1 to 5
        collect (random 10))
(3 8 6 5 0)
```

這裡我們獲得了一個含五個隨機數的列表。這跟我們定義過的 map-int 情況類似 (105 頁「譯註: 6.4 小節。」)。如果我們有了 loop ，爲什麼還需要 map-int ？另一個人也可以說，如果我們有了 map-int ，爲什麼還需要 loop ？

一個 collect 子句也可以累積值到一個有名字的變數上。下面的函數接受一個數字的列表並返回偶數與奇數列表：

```
(defun even/odd (ns)
  (loop for n in ns
        if (evenp n)
           collect n into evens
           else collect n into odds
        finally (return (values evens odds))))
```

一個 `finally` 子句貢獻

程式碼至閉幕。在這個情況它指定了返回值。

一個 `sum` 子句和一個 `collect` 子句類似，但 `sum` 子句累積一個數字，而不是一個列表。要獲得 `1` 至 `n` 的和，我們可以寫：

```
(defun sum (n)
  (loop for x from 1 to n
        sum x))
```

`loop` 更進一步的細節在附錄 D 討論，從 325 頁開始。舉個例子，圖 14.1 包含了先前章節的兩個迭代函數，而圖 14.2 示範了將同樣的函數翻譯成 `loop` 。

```
(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)
              (setf wins obj
                    max  score))))
        (values wins max))))

(defun num-year (n)
  (if (< n 0)
      (do* ((y (- yzero 1) (- y 1))
            (d (- (year-days y)) (- d (year-days y))))
           ((<= d n) (values y (- n d))))
      (do* ((y yzero (+ y 1))
            (prev 0 d)
            (d (year-days y) (+ d (year-days y))))
           ((> d n) (values y (- n prev))))))
```

圖 **14.1** 不使用 **loop** 的迭代函數

```
(defun most (fn lst)
  (if (null lst)
      (values nil nil)
      (loop with wins = (car lst)
```

```
            with max = (funcall fn wins)
            for obj in (cdr lst)
            for score = (funcall fn obj)
            when (> score max)
                 (do (setf wins obj
                           max score)
            finally (return (values wins max))))))

(defun num-year (n)
  (if (< n 0)
      (loop for y downfrom (- yzero 1)
            until (<= d n)
            sum (- (year-days y)) into d
            finally (return (values (+ y 1) (- n d))))
      (loop with prev = 0
            for y from yzero
            until (> d n)
            do (setf prev d)
            sum (year-days y) into d
            finally (return (values (- y 1)
                                    (- n prev))))))
```

圖 **14.2** 使用 **loop** 的迭代函數

一個 `loop` 的子句可以參照到由另一個子句所設置的變數。舉例來說，在 `even/odd` 的定義裡面， `finally` 子句參照到由兩個 `collect` 子句所創建的變數。這些變數之間的關係，是 `loop` 定義最含糊不清的地方。考慮下列兩個表達式：

```
(loop for y = 0 then z
      for x from 1 to 5
      sum 1 into z
      finally (return y z))

(loop for x from 1 to 5
      for y = 0 then z
      sum 1 into z
      finally (return y z))
```

它們看起來夠簡單 —— 每一個有四個子句。但它們返回同樣的值嗎？它們返回的值多少？你若試著在標準中想找答案將徒勞無功。每一個 `loop` 子句本身是夠簡單的。但它們組合起來的方式是極爲複雜的 —— 而最終，甚至標準裡也沒有明確定義。

由於這類原因，使用 `loop` 是不推薦的。推薦 `loop` 的理由，你最多可以說，在像是圖 14.2 這般經典的例子中， `loop` 讓

程式碼看起來更容易理解。

14.6 狀況 (Conditions)

在 Common Lisp 裡，狀況 (condition)包括了錯誤以及其它可能在執行期發生的情況。當一個狀況被捕捉時 (signalled)，相應的處理程式 (handler)會被呼叫。處理錯誤狀況的預設處理程式通常會呼叫一個中斷迴圈 (break-loop)。但 Common Lisp 提供了多樣的運算子來捕捉及處理錯誤。要覆寫預設的處理程式，甚至是自己寫一個新的處理程式也是有可能的。

多數的程式設計師不會直接處理狀況。然而有許多更抽象的運算子使用了狀況，而要了解這些運算子，知道背後的原理是很有用的。

Common lisp 有數個運算子用來捕捉錯誤。最基本的是 `error` 。一個呼叫它的方法是給入你會給 `format` 的相同參數：

```
> (error "Your report uses ~A as a verb." 'status)
Error: Your report uses STATUS as a verb
                        Options: :abort, :backtrace
>>
```

如上所示，除非這樣的狀況被處理好了，不然執行就會被打斷。

用來捕捉錯誤的更抽象運算子包括了 `ecase` 、 `check-type` 以及 `assert` 。前者與 `case` 相似，要是沒有鍵值匹配時會捕捉一個錯誤：

```
> (ecase 1 (2 3) (4 5))
Error: No applicable clause
                        Options: :abort, :backtrace
>>
```

普通的 `case` 在沒有鍵值匹配時會返回 `nil` ，但由於利用這個返回值是很差的編碼風格，你或許會在當你沒有 `otherwise` 子句時使用 `ecase` 。

`check-type` 宏接受一個位置，一個型別名以及一個選擇性字串，並在該位置的值不是預期的型別時，捕捉一個可修正的錯誤 (correctable error)。一個可修正錯誤的處理程式會給我們一個機會來提供一個新的值：

```
> (let ((x '(a b c)))
                (check-type (car x) integer "an integer")
                x)
Error: The value of (CAR X), A, should be an integer.
Options: :abort, :backtrace, :continue
>> :continue
New value of (CAR X)? 99
(99 B C)
>
```

在這個例子裡， `(car x)` 被設爲我們提供的新值，並重新執行，返回了要是 `(car x)` 本

來就包含我們所提供的值所會返回的結果。

這個宏是用更通用的 assert 所定義的， assert 接受一個測試表達式以及一個有著一個或多個位置的列表，伴隨著你可能傳給 error 的參數：

```
> (let ((sandwich '(ham on rye)))
    (assert (eql (car sandwich) 'chicken)
            ((car sandwich))
            "I wanted a ~A sandwich." 'chicken)
    sandwich)
Error: I wanted a CHICKEN sandwich.
Options: :abort, :backtrace, :continue
>> :continue
New value of (CAR SANDWICH)? 'chicken
(CHICKEN ON RYE)
```

要建立新的處理程式也是可能的，但大多數程式設計師只會間接的利用這個可能性，通過使用像是 ignore-errors 的宏。如果它的參數沒產生錯誤時像在 progn 裡求值一樣，但要是在求值過程中，不管什麼參數報錯，執行是不會被打斷的。取而代之的是，ignore-errors 表達式會直接返回兩個值： nil 以及捕捉到的狀況。

舉例來說，如果在某個時候，你想要用戶能夠輸入一個表達式，但你不想要在輸入是語法上不合時中斷執行，你可以這樣寫：

```
(defun user-input (prompt)
  (format t prompt)
  (let ((str (read-line)))
    (or (ignore-errors (read-from-string str))
        nil)))
```

若輸入包含語法錯誤時，這個函數僅返回 nil :

```
> (user-input "Please type an expression")
Please type an expression> #%@#+!!
NIL
```

腳註

[1] 雖然標準沒有提到這件事，你可以假定 and 以及 or 型別標示符僅考慮它們所要考慮的參數，與 or 及 and 宏類似。

[2] 某些 Common Lisp 實現，當我們不在用戶包下時，會在頂層提示符前打印包的名字。

第十五章：範例：推論

接下來三章提供了大量的 Lisp 程式例子。選擇這些例子來說明那些較長的程式所採取的形式，和 Lisp 所擅長解決的問題型別。

在這一章中我們將要寫一個基於一組 `if-then` 規則的推論程式。這是一個經典的例子——不僅在於其經常出現在教科書上，還因爲它反映了 Lisp 作爲一個“符號計算”語言的本意。這個例子散發著很多早期 Lisp 程式的氣息。

15.1 目標 (The Aim)

在這個程式中，我們將用一種熟悉的形式來表示資訊：包含單個判斷式，以及跟在之後的零個或多個參數所組成的列表。要表示 Donald 是 Nancy 的家長，我們可以這樣寫：

```
(parent donald nancy)
```

事實上，我們的程式是要表示一些從已有的事實作出推斷的規則。我們可以這樣來表示規則：

```
(<- head body)
```

其中，`head` 是 那麼...部分 (then-part)，`body` 是 如果...部分 (if-part)。在 `head` 和 `body` 中我們使用以問號爲前綴的符號來表示變數。所以下面這個規則：

```
(<- (child ?x ?y) (parent ?y ?x))
```

表示：如果 y 是 x 的家長，那麼 x 是 y 的孩子；更恰當地說，我們可以通過證明 `(parent y x)` 來證明 `(child x y)` 的所表示的事實。

可以把規則中的 *body* 部分(if-part) 寫成一個複雜的表達式，其中包含 `and` , `or` 和 `not` 等邏輯操作。所以當我們想要表達 “如果 x 是 y 的家長，並且 x 是男性，那麼 x 是 y 的父親” 這樣的規則，我們可以寫：

```
(<- (father ?x ?y) (and (parent ?x ?y) (male ?x)))
```

一些規則可能依賴另一些規則所產生的事實。比如，我們寫的第一個規則是爲了證明 `(child x y)` 的事實。如果我們定義如下規則：

```
(<- (daughter ?x ?y) (and (child ?x ?y) (female ?x)))
```

然後使用它來證明 `(daughter x y)` 可能導致程式使用第一個規則去證明 `(child x y)`。

表達式的證明可以回溯任意數量的規則，只要它最終結束於給出的已知事實。這個過程有時候被稱爲反向連結 (backward-chaining)。之所以說 反向 (backward) 是因爲這一類推論先考慮 *head* 部分，這是爲了在繼續證明 *body* 部分之前檢查規則是否有效。連結 (chaining) 來源於規則之間的依賴關係，從我們想要證明的內容到我們的已知條件組成一個連結 (儘管事實上它更像一棵樹)。 λ [http://acl.readthedocs.org/en/latest/zhTW/notes.html#notes-248]

15.2 匹配 (Matching)

我們需要有一個函數來做模式匹配以完成我們的反向連結 (back-chaining) 程式，這個函數能夠比較兩個包含變數的列表，它會檢查在給變數賦值後是否可以使兩個列表相等。舉例，如果 `?x` 和 `?y` 是變數，那麼下面兩個列表：

```
(p ?x ?y c ?x)
(p  a  b c  a)
```

當 `?x = a` 且 `?y = b` 時匹配，而下面兩個列表：

```
(p ?x b ?y a)
(p ?y b  c a)
```

當 `?x = ?y = c` 時匹配。

我們有一個 `match` 函數，它接受兩棵樹，如果這兩棵樹能匹配，則返回一個關聯列表（assoc-list）來顯示他們是如何匹配的：

```
(defun match (x y &optional binds)
  (cond
   ((eql x y) (values binds t))
   ((assoc x binds) (match (binding x binds) y binds))
   ((assoc y binds) (match x (binding y binds) binds))
   ((var? x) (values (cons (cons x y) binds) t))
   ((var? y) (values (cons (cons y x) binds) t))
   (t
    (when (and (consp x) (consp y))
      (multiple-value-bind (b2 yes)
                           (match (car x) (car y) binds)
        (and yes (match (cdr x) (cdr y) b2)))))))

(defun var? (x)
  (and (symbolp x)
       (eql (char (symbol-name x) 0) #\?)))
```

```
(defun binding (x binds)
  (let ((b (assoc x binds)))
    (if b
        (or (binding (cdr b) binds)
            (cdr b)))))
```

圖 **15.1:** 匹配函數。

```
> (match '(p a b c a) '(p ?x ?y c ?x))
((?Y . B) (?X . A))
T
> (match '(p ?x b ?y a) '(p ?y b c a))
((?Y . C) (?X . ?Y))
T
> (match '(a b c) '(a a a))
NIL
```

當 match 函數逐個元素地比較它的參數時候，它把 binds 參數中的值分配給變數，這被稱爲綁定 (bindings)。如果成功匹配， match 函數返回生成的綁定；否則，返回 nil 。當然並不是所有成功的匹配都會產生綁定，我們的 match 函數就像 gethash 函數那樣返回第二個值來表明匹配成功：

```
> (match '(p ?x) '(p ?x))
NIL
T
```

如果 match 函數像上面那樣返回 nil 和 t ，表明這是一個沒有產生綁定的成功匹配。下面用中文來描述 match 算法是如何工作的：

1. 如果 x 和 y 在 eql 上相等那麼它們匹配；否則，
2. 如果 x 是一個已綁定的變數，並且綁定匹配 y ，那麼它們匹配；否則，
3. 如果 y 是一個已綁定的變數，並且綁定匹配 x ，那麼它們匹配；否則，
4. 如果 x 是一個未綁定的變數，那麼它們匹配，並且爲 x 建立一個綁定；否則，
5. 如果 y 是一個未綁定的變數，那麼它們匹配，並且爲 y 建立一個綁定；否則，
6. 如果 x 和 y 都是 cons ，並且它們的 car 匹配，由此產生的綁定又讓 cdr 匹配，那麼它們匹配。

下面是一個例子，按順序來說明以上六種情況：

```
> (match '(p ?v  b ?x  d (?z ?z))
         '(p  a ?w  c ?y ( e  e))
         '((?v . a) (?w . b)))
((?Z . E) (?Y . D) (?X . C) (?V . A) (?W . B))
T
```

match 函數通過呼叫 binding 函數在一個綁定列表中尋找變數（如果有的話）所關聯的值。這個函數必須是遞迴的，因爲有這樣的情況 “匹配建立一個綁定列表，而列表中變數只是間接關聯到它的值： ?x 可能被綁定到一個包含 (?x . ?y) 和 (?y . a) 的列表”：

```
> (match '(?x a) '(?y ?y))
((?Y . A) (?X . ?Y))
T
```

先匹配 ?x 和 ?y，然後匹配 ?y 和 a，我們間接確定 ?x 是 a。

15.3 回答查詢 (Answering Queries)

在介紹了綁定的概念之後，我們可以更準確的說一下我們的程式將要做什麼：它得到一個可能包含變數的表達式，根據我們給定的事實和規則返回使它正確的所有綁定。比如，我們只有下面這個事實：

```
(parent donald nancy)
```

然後我們想讓程式證明：

```
(parent ?x ?y)
```

它會返回像下面這樣的表達：

```
(((?x . donald) (?y . nancy)))
```

它告訴我們只有一個可以讓這個表達式爲真的方法： ?x 是 donald 並且 ?y 是 nancy。

在通往目標的路上，我們已經有了一個的重要部分：一個匹配函數。 下面是用來定義規則的一段

程式碼：

```
(defvar *rules* (make-hash-table))

(defmacro <- (con &optional ant)
  `(length (push (cons (cdr ',con) ',ant)
                 (gethash (car ',con) *rules*))))
```

圖 **15.2** 定義規則

規則將被包含於一個叫做 *rules* 的雜湊表，通過頭部 (head) 的判斷式構建這個哈系

表。這樣做加強了我們無法使用判斷式中的變數的限制。雖然我們可以通過把所有這樣的規則放在分離的列表中來消除限制，但是如果這樣做，當我們需要證明某件事的時侯不得不和每一個列表進行匹配。

我們將要使用同一個宏 `<-` 去定義事實 (facts)和規則 (rules)。一個事實將被表示成一個沒有 *body* 部分的規則。這和我們對規則的定義保持一致。一個規則告訴我們你可以通過證明 *body* 部分來證明 *head* 部分，所以沒有 *body* 部分的規則意味著你不需要通過證明任何東西來證明 *head* 部分。這裡有兩個對應的例子：

```
> (<- (parent donald nancy))
1
> (<- (child ?x ?y) (parent ?y ?x))
1
```

呼叫 `<-` 返回的是給定判斷式下存儲的規則數量；用 `length` 函數來包裝 `push` 能使我們免於看到頂層中的一大堆返回值。

下面是我們的推論程式所需的大多數

程式碼：

```
(defun prove (expr &optional binds)
  (case (car expr)
    (and (prove-and (reverse (cdr expr)) binds))
    (or  (prove-or (cdr expr) binds))
    (not (prove-not (cadr expr) binds))
    (t   (prove-simple (car expr) (cdr expr) binds))))

(defun prove-simple (pred args binds)
  (mapcan #'(lambda (r)
              (multiple-value-bind (b2 yes)
                                   (match args (car r)
                                          binds)
                (when yes
                  (if (cdr r)
                      (prove (cdr r) b2)
                      (list b2)))))
          (mapcar #'change-vars
                  (gethash pred *rules*))))

(defun change-vars (r)
  (sublis (mapcar #'(lambda (v) (cons v (gensym "?")))
                  (vars-in r))
          r))

(defun vars-in (expr)
  (if (atom expr)
      (if (var? expr) (list expr))
    (union (vars-in (car expr))
```

```
            (vars-in (cdr expr)))))
```

圖 **15.3:** 推論。

上面

程式碼中的 `prove` 函數是推論進行的樞紐。它接受一個表達式和一個可選的綁定列表作爲參數。如果表達式不包含邏輯操作，它呼叫 `prove-simple` 函數，前面所說的連結(chaining)正是在這個函數裡產生的。這個函數查看所有擁有正確判斷式的規則，並嘗試對每一個規則的 *head* 部分和它想要證明的事實做匹配。對於每一個匹配的 *head*，使用匹配所產生的新的綁定在 *body* 上呼叫 `prove`。對 `prove` 的呼叫所產生的綁定列表被 `mapcan` 收集並返回：

```
> (prove-simple 'parent '(donald nancy) nil)
(NIL)
> (prove-simple 'child '(?x ?y) nil)
(((#:?6 . NANCY) (#:?5 . DONALD) (?Y . #:?5) (?X . #:?6)))
```

以上兩個返回值指出有一種方法可以證明我們的問題。（一個失敗的證明將返回 nil。）第一個例子產生了一組空的綁定，第二個例子產生了這樣的綁定：`?x` 和 `?y` 被（間接）綁定到 `nancy` 和 `donald`。

順便說一句，這是一個很好的例子來實踐 2.13 節提出的觀點。因爲我們用函數式的風格來寫這個程式，所以可以交互式地測試每一個函數。

第二個例子返回的值裡那些 *gensyms* 是怎麼回事？如果我們打算使用含有變數的規則，我們需要避免兩個規則恰好包含相同的變數。如果我們定義如下兩條規則：

```
(<- (child ?x ?y) (parent ?y ?x))

(<- (daughter ?y ?x) (and (child ?y ?x) (female ?y)))
```

第一條規則要表達的意思是：對於任何的 `x` 和 `y`， 如果 `y` 是 `x` 的家長，則 `x` 是 `y` 的孩子。第二條則是：對於任何的 `x` 和 `y`， 如果 `y` 是 `x` 的孩子並且 `y` 是女性，則 `y` 是 `x` 的女兒。在每一條規則內部，變數之間的關係是顯著的，但是兩條規則使用了相同的變數並非我們刻意爲之。

如果我們使用上面所寫的規則，它們將不會按預期的方式工作。如果我們嘗試證明“ a 是 b 的女兒”，匹配到第二條規則的 *head* 部分時會將 `a` 綁定到 `?y`，將 `b` 綁定到 ?x。我們無法用這樣的綁定匹配第一條規則的 *head* 部分：

```
> (match '(child ?y ?x)
         '(child ?x ?y)
```

```
          '((?y . a) (?x . b)))
NIL
```

爲了保證一條規則中的變數只表示規則中各參數之間的關係，我們用 *gensyms* 來代替規則中的所有變數。這就是 `change-vars` 函數的目的。一個 *gensym* 不可能在另一個規則中作爲變數出現。但是因爲規則可以是遞迴的，我們必須防止出現一個規則和自身衝突的可能性，所以在定義和使用一個規則時都要呼叫 `chabge-vars` 函數。

現在只剩下定義用以證明複雜表達式的函數了。下面就是需要的函數：

```
(defun prove-and (clauses binds)
  (if (null clauses)
      (list binds)
      (mapcan #'(lambda (b)
                  (prove (car clauses) b))
              (prove-and (cdr clauses) binds))))

(defun prove-or (clauses binds)
  (mapcan #'(lambda (c) (prove c binds))
          clauses))

(defun prove-not (clause binds)
  (unless (prove clause binds)
    (list binds)))
```

圖 **15.4** 邏輯運算子 **(Logical operators)**

操作一個 `or` 或者 `not` 表達式是非常簡單的。操作 `or` 時，我們提取在 `or` 之間的每一個表達式返回的綁定。操作 `not` 時，當且僅當在 `not` 裡的表達式產生 `none` 時，返回當前的綁定。

`prove-and` 函數稍微複雜一點。它像一個過濾器，它用之後的表達式所建立的每一個綁定來證明第一個表達式。這將導致 `and` 裡的表達式以相反的順序被求值。除非呼叫 `prove` 中的 `prove-and` 函數則會先逆轉它們。

現在我們有了一個可以工作的程式，但它不是很友好。必須要解析 `prove-and` 返回的綁定列表是令人厭煩的，它們會變得更長隨著規則變得更加複雜。下面有一個宏來幫助我們更愉快地使用這個程式：

```
(defmacro with-answer (query &body body)
  (let ((binds (gensym)))
    `(dolist (,binds (prove ',query))
       (let ,(mapcar #'(lambda (v)
                         `(,v (binding ',v ,binds)))
                     (vars-in query))
         ,@body))))
```

圖 **15.5** 介面宏 **(Interface macro)**

它接受一個 `query` （不被求值）和若干表達式構成的 `body` 作爲參數，把 `query` 所生成的每一組綁定的值賦給 `query` 中對應的模式變數，並計算 `body` 。

```
> (with-answer (parent ?x ?y)
    (format t "~A is the parent of ~A.~%" ?x ?y))
DONALD is the parent of NANCY.
NIL
```

這個宏幫我們做了解析綁定的工作，同時爲我們在程式中使用 `prove` 提供了一個便捷的方法。下面是這個宏展開的情況：

```
(with-answer (p ?x ?y)
  (f ?x ?y))

;;將被展開成下面的
```

程式碼

```
(dolist (#:g1 (prove '(p ?x ?y)))
    (let ((?x (binding '?x #:g1))
          (?y (binding '?y #:g1)))

        (f ?x ?y)))
```

圖 **15.6: with-answer** 呼叫的展開式

下面是使用它的一個例子：

```
(<- (parent donald nancy))
(<- (parent donald debbie))
(<- (male donald))
(<- (father ?x ?y) (and (parent ?x ?y) (male ?x)))
(<- (= ?x ?y))
(<- (sibling ?x ?y) (and (parent ?z ?x)
                         (parent ?z ?y)
                         (not (= ?x ?y)))))

;;我們可以像下面這樣做出推論

> (with-answer (father ?x ?y)
    (format t "~A is the father of ~A.~%" ?x ?y))
DONALD is the father of DEBBIE.
DONALD is the father of NANCY.
NIL
> (with-answer (sibling ?x ?y))
```

```
    (format t "~A is the sibling of ~A.~%" ?x ?y))
DEBBLE is the sibling of NANCY.
NANCY is the  sibling of DEBBIE.
NIL
```

圖 **15.7:** 使用中的程式

15.4 分析 (Analysis)

看上去，我們在這一章中寫的

程式碼，是用簡單自然的方式去實現這樣一個程式。事實上，它的效率非常差。我們在這裡是其實是做了一個解釋器。我們能夠把這個程式做得像一個編譯器。

這裡做一個簡單的描述。基本的思想是把整個程式打包到兩個宏 `<-` 和 `with-answer` ，把已有程式中在運行期做的多數工作搬到宏展開期（在 10.7 節的 `avg` 可以看到這種構思的雛形) 用函數取代列表來表示規則，我們不在運行時用 `prove` 和 `prove-and` 這樣的函數來解釋表達式，而是用相應的函數把表達式轉化成

程式碼。當一個規則被定義的時候就有表達式可用。爲什麼要等到使用的時候才去分析它呢？這同樣適用於和 `<-` 呼叫了相同的函數來進行宏展開的 `with-answer` 。

聽上去好像比我們已經寫的這個程式複雜很多，但其實可能只是長了兩三倍。想要學習這種技術的讀者可以看 *On Lisp* 或者 *Paradigms of Artificial Intelligence Programming* ，這兩本書有一些使用這種風格寫的範例程式。

第十六章：範例：產生 HTML

本章的目標是完成一個簡單的 HTML 產生器 —— 這個程式可以自動生成一系列包含超文字連結的網頁。除了介紹特定 Lisp 技術之外，本章還是一個典型的自底向上編程（bottom-up programming）的例子。 我們以一些通用 HTML 實用函數作爲開始，繼而將這些例程看作是一門編程語言，從而更好地編寫這個產生器。

16.1 超文字標記語言 (HTML)

HTML （HyperText Markup Language，超文字標記語言）用於構建網頁，是一種簡單、易學的語言。本節就對這種語言作概括性介紹。

當你使用網頁瀏覽器閱覽網頁時，瀏覽器從遠程服務器獲取 HTML 檔案，並將它們顯示在你的屏幕上。每個 HTML 檔案都包含任意多個標籤（tag），這些標籤相當於發送給瀏覽器的指令。

```
<center>
<h2>Your Fortune</h2>
</center>
<br><br>
Welcome to the home page of the Fortune Cookie
Institute.  FCI is a non-profit institution
dedicated to the development of more realistic
fortunes.  Here are some examples of fortunes
that fall within our guidelines:
<ol>
<li>Your nostril hairs will grow longer.
<li>You will never learn how to dress properly.
<li>Your car will be stolen.
<li>You will gain weight.
</ol>
Click <a href="research.html">here</a> to learn
more about our ongoing research projects.
```

圖 **16.1** 一個 **HTML** 檔案

圖 16.1 給出了一個簡單的 HTML 檔案，圖 16.2 展示了這個 HTML 檔案在瀏覽器裡顯示時大概是什麼樣子。

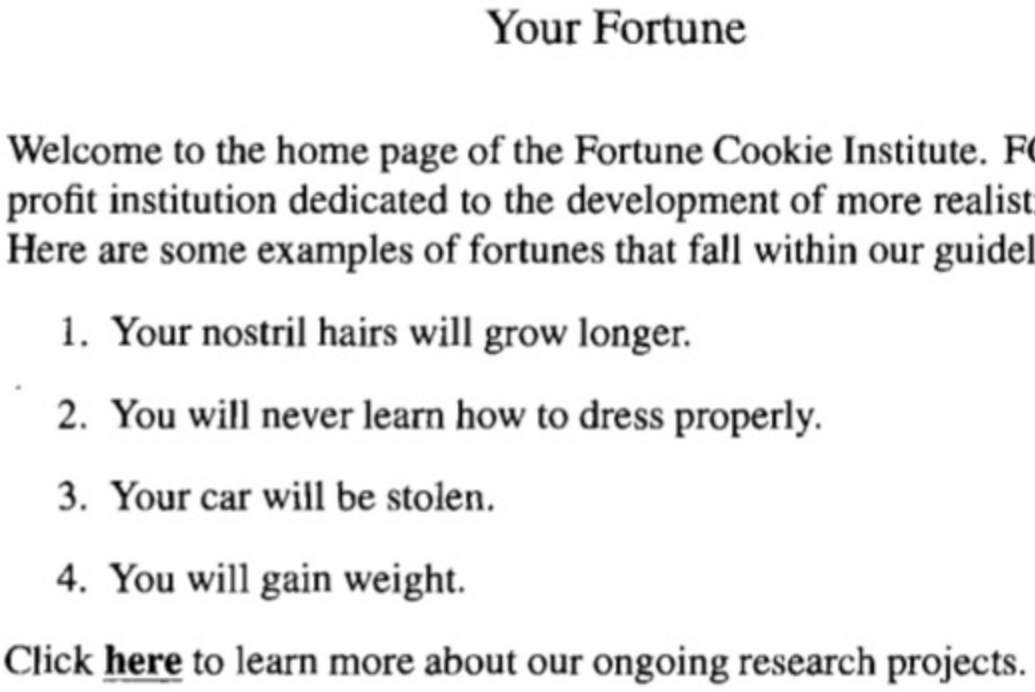

圖 **16.2** 一個網頁

注意在三角符號之間的文字並沒有被顯示出來，這些用三角符號包圍的文字就是標籤。HTML 的標籤分爲兩種，一種是成雙成對地出現的：

```
<tag>...</tag>
```

第一個標籤標誌著某種情景（environment）的開始，而第二個標籤標誌著這種情景的結束。 這種標籤的一個例子是 `<h2>`：所有被 `<h2>` 和 `</h2>` 包圍的文字，都會使用比平常字體尺寸稍大的字體來顯示。

另外一些成雙成對出現的標籤包括：創建帶編號列表的 `<ol>` 標籤（ol 代表 ordered list，有序表），令文字居中的 `<center>` 標籤，以及創建連結的 `<a>` 標籤（a 代表 anchor，錨點）。

被 `<a>` 和 `</a>` 包圍的文字就是超文字（hypertext）。 在大多數瀏覽器上，超文字都會以一種與衆不同的方式被凸顯出來 —— 它們通常會帶有下劃線 —— 並且點擊這些文字會讓瀏覽器跳轉到另一個頁面。 在標籤 `a` 之後的部分，指示了連結被點擊時，瀏覽器應該跳轉到的位置。

一個像

```
<a href="foo.html">
```

這樣的標籤，就標識了一個指向另一個 HTML 檔案的連結，其中這個 HTML 檔案和當

前網頁的檔案夾相同。 當點擊這個連結時，瀏覽器就會獲取並顯示 `foo.html` 這個檔案。

當然，連結並不一定都要指向相同檔案夾下的 HTML 檔案，實際上，一個連結可以指向互聯網的任何一個檔案。

和成雙成對出現的標籤相反，另一種標籤沒有結束標記。 在圖 16.1 裡有一些這樣的標籤，包括：創建一個新文字行的 `<br>` 標籤（br 代表 break ，斷行），以及在列表情景中，創建一個新列表項的 `<li>` 標籤（li 代表 list item ，列表項）。

HTML 還有不少其他的標籤，但是本章要用到的標籤，基本都包含在圖 16.1 裡了。

16.2 HTML 實用函數 (HTML Utilities)

```
(defmacro as (tag content)
  `(format t "<~(~A~)>~A</~(~A~)>"
           ',tag ,content ',tag))

(defmacro with (tag &rest body)
  `(progn
     (format t "~&<~(~A~)>~%" ',tag)
     ,@body
     (format t "~&</~(~A~)>~%" ',tag)))

(defmacro brs (&optional (n 1))
  (fresh-line)
  (dotimes (i n)
    (princ "<br>"))
  (terpri))
```

圖 **16.3** 標籤生成例程

本節會定義一些生成 HTML 的例程。 圖 16.3 包含了三個基本的、生成標籤的例程。 所有例程都將它們的輸出發送到 `*standard-output*` ； 可以通過重新綁定這個變數，將輸出重定向到一個檔案。

宏 `as` 和 `with` 都用於在標籤之間生成表達式。其中 `as` 接受一個字元串，並將它打印在兩個標籤之間：

```
> (as center "The Missing Lambda")
<center>The Missing Lambda</center>
NIL
```

`with` 則接受一個程式碼主體，並將它放置在兩個標籤之間：

```
> (with center
    (princ "The Unbalanced Parenthesis"))
<center>
The Unbalanced Parenthesis
</center>
NIL
```

兩個宏都使用了 `~(...~)` 來進行格式化，從而創建包含小寫字母的標籤。 HTML 並不介意標籤是大寫還是小寫，但是在包含許許多多標籤的 HTML 檔案中，小寫字母的標籤可讀性更好一些。

除此之外， `as` 傾向於將所有輸出都放在同一行，而 `with` 則將標籤和內容都放在不同的行裡。 （使用 `~&` 來進行格式化，以確保輸出從一個新行中開始。） 以上這些工作都只是爲了讓 HTML 更具可讀性，實際上，標籤之外的空白並不影響頁面的顯示方式。

圖 16.3 中的最後一個例程 `brs` 用於創建多個文字行。 在很多瀏覽器中，這個例程都可以用於控制垂直間距。

```
(defun html-file (base)
  (format nil "~(~A~).html" base))

(defmacro page (name title &rest body)
  (let ((ti (gensym)))
    `(with-open-file (*standard-output*
                      (html-file ,name)
                      :direction :output
                      :if-exists :supersede)
       (let ((,ti ,title))
         (as title ,ti)
         (with center
           (as h2 (string-upcase ,ti)))
         (brs 3)
         ,@body))))
```

圖 **16.4 HTML** 檔案生成例程

圖 16.4 包含用於生成 HTML 檔案的例程。 第一個函數根據給定的符號（symbol）返回一個檔案名。 在一個實際應用中，這個函數可能會返回指向某個特定檔案夾的路徑（path）。 目前來說，這個函數只是簡單地將 `.html` 後綴追加到給定符號名的後邊。

宏 `page` 負責生成整個頁面，它的實現和 `with-open-file` 很相似： `body` 中的表達式會被求值，求值的結果通過 `*standard-output*` 所綁定的流，最終被寫入到相應的 HTML 檔案中。

6.7 小節示範了如何臨時性地綁定一個特殊變數。 在 113 頁的例子中，我們在 `let` 的體內將 `*print-base*` 綁定爲 `16` 。 這一次，通過將 `*standard-output*` 和一個指向 HTML

檔案的流綁定，只要我們在 page 的函數體內呼叫 as 或者 princ ，輸出就會被傳送到 HTML 檔案裡。

page 宏的輸出先在頂部打印 title ，接著打印 body 部分的輸出。

如果我們呼叫

```
(page 'paren "The Unbalanced Parenthesis"
  (princ "Something in his expression told her..."))
```

這會產生一個名爲 paren.html 的檔案（檔案名由 html-file 函數生成），檔案中的內容爲：

```
<title>The Unbalanced Parenthesis</title>
<center>
<h2>THE UNBALANCED PARENTHESIS</h2>
</center>
<br><br><br>
Something in his expression told her...
```

除了 title 標籤以外，以上輸出的所有 HTML 標簽在前面已經見到過了。 被 <title> 標籤包圍的文字並不顯示在網頁之內，它們會顯示在瀏覽器窗口，用作頁面的標題。

```
(defmacro with-link (dest &rest body)
  `(progn
     (format t "<a href=\"~A\">" (html-file ,dest))
     ,@body
     (princ "</a>")))

(defun link-item (dest text)
  (princ "<li>")
  (with-link dest
    (princ text)))

(defun button (dest text)
  (princ "[ ")
  (with-link dest
    (princ text))
  (format t " ]~%"))
```

圖 **16.5** 生成連結的例程

圖片 16.5 給出了用於生成連結的例程。 with-link 和 with 很相似：它根據給定的網址 dest ，創建一個指向 HTML 檔案的連結。 而連結內部的文字，則通過求值 body 參數中的程式碼段得出：

```
> (with-link 'capture
```

```
    (princ "The Captured Variable"))
<a href="capture.html">The Captured Variable</a>
"</a>"
```

with-link 也被用在 link-item 當中，這個函數接受一個字元串，並創建一個帶連結的列表項：

```
> (link-item 'bq "Backquote!")
<li><a href="bq.html">Backquote!</a>
"</a>"
```

最後， button 也使用了 with-link ，從而創建一個被方括號包圍的連結：

```
> (button 'help "Help")
[ <a href="help.html">Help</a> ]
NIL
```

16.3 迭代式實用函數 (An Iteration Utility)

在這一節，我們先暫停一下編寫 HTML 產生器的工作，轉到編寫迭代式例程的工作上來。

你可能會問，怎樣才能知道，什麼時候應該編寫主程式，什麼時候又應該編寫子例程？

實際上，這個問題，沒有答案。

通常情況下，你總是先開始寫一個程式，然後發現需要寫一個新的例程，於是你轉而去編寫新例程，完成它，接著再回過頭去編寫原來的程式。 時間關係，要在這裡示範這個開始-完成-又再開始的過程是不太可能的，這裡只展示這個迭代式例程的最終形態，需要注意的是，這個程式的編寫並不如想象中的那麼簡單。 程式通常需要經歷多次重寫，才會變得簡單。

```
(defun map3 (fn lst)
  (labels ((rec (curr prev next left)
             (funcall fn curr prev next)
             (when left
               (rec (car left)
                    curr
                    (cadr left)
                    (cdr left)))))
    (when lst
      (rec (car lst) nil (cadr lst) (cdr lst)))))
```

圖 **16.6** 對樹進行迭代

圖 16.6 裡定義的新例程是 mapc 的一個變種。它接受一個函數和一個列表作爲參數，對於傳入列表中的每個元素，它都會用三個參數來呼叫傳入函數，分別是元素本身，前一個元素，以及後一個元素。（當沒有前一個元素或者後一個元素時，使用 nil 代替。）

```
> (map3 #'(lambda (&rest args) (princ args))
        '(a b c d))
(A NIL B) (B A C) (C B D) (D C NIL)
NIL
```

和 mapc 一樣， map3 總是返回 nil 作爲函數的返回值。需要這類例程的情況非常多。在下一個小節就會看到，這個例程是如何讓每個頁面都實現“前進一頁”和“後退一頁”功能的。

map3 的一個常見功能是，在列表的兩個相鄰元素之間進行某些處理：

```
> (map3 #'(lambda (c p n)
            (princ c)
            (if n (princ " | ")))
        '(a b c d))
A | B | C | D
NIL
```

程式設計師經常會遇到上面的這類問題，但只要花些功夫，定義一些例程來處理它們，就能爲後續工作節省不少時間。

16.4 生成頁面 (Generating Pages)

一本書可以有任意數量的大章，每個大章又有任意數量的小節，而每個小節又有任意數量的分節，整本書的結構呈現出一棵樹的形狀。

儘管網頁使用的術語和書本不同，但多個網頁同樣可以被組織成樹狀。

本節要構建的是這樣一個程式，它生成多個網頁，這些網頁帶有以下結構： 第一頁是一個目錄，目錄中的連結指向各個節點（section）頁面。 每個節點包含一些指向項（item）的連結。 而一個項就是一個包含純文字的頁面。

除了頁面本身的連結以外，根據頁面在樹狀結構中的位置，每個頁面都會帶有前進、後退和向上的連結。 其中，前進和後退連結用於在同級（sibling）頁面中進行導航。 舉個例子，點擊一個項頁面中的前進連結時，如果這個項的同一個節點下還有下一個項，那麼就跳到這個新項的頁面裡。 另一方面，向上連結將頁面跳轉到樹形結構的上一層——如果當前頁面是項頁面，那麼返回到節點頁面；如果當前頁面是節點頁面，那麼返回到目錄頁面。 最後，還會有索引頁面：這個頁面包含一系列連結，按字母順序排列所有項。

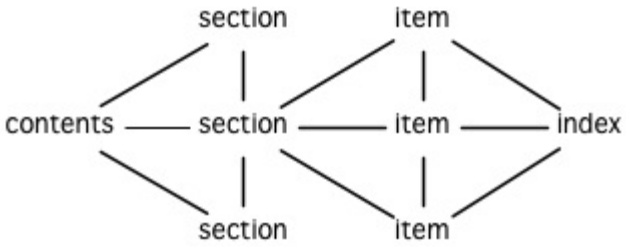

圖 **16.7** 網站的結構

圖 16.7 展示了生成程式創建的頁面所形成的連結結構。

```
(defparameter *sections* nil)

(defstruct item
  id title text)

(defstruct section
  id title items)

(defmacro defitem (id title text)
  `(setf ,id
         (make-item :id     ',id
                    :title  ,title
                    :text   ,text)))

(defmacro defsection (id title &rest items)
  `(setf ,id
         (make-section :id    ',id
                       :title ,title
                       :items (list ,@items))))

(defun defsite (&rest sections)
  (setf *sections* sections))
```

圖 **16.8** 定義一個網站

圖 16.8 包含定義頁面所需的資料結構。程式需要處理兩類物件：項和節點。這兩類物件的結構很相似，不過節點包含的是項的列表，而項包含的是文字塊。

節點和項兩類物件都帶有 `id` 域。 標識符（id）被用作符號（symbol），並達到以下兩個目的：在 `defitem` 和 `defsection` 的定義中， 標識符會被設置到被創建的項或者節點當中，作爲我們引用它們的一種手段；另一方面，標識符還會作爲相應檔案的前綴名（base name），比如說，如果項的標識符爲 `foo` ，那麼項就會被寫到 `foo.html` 檔案當中。

節點和項也同時帶有 `title` 域。這個域的值應該爲字元串，並且被用作相應頁面的標

題。

在節點裡，項的排列順序由傳給 `defsection` 的參數決定。 與此類似，在目錄裡，節點的排列順序由傳給 `defsite` 的參數決定。

```
(defconstant contents "contents")
(defconstant index    "index")

(defun gen-contents (&optional (sections *sections*))
  (page contents contents
    (with ol
      (dolist (s sections)
        (link-item (section-id s) (section-title s))
        (brs 2))
      (link-item index (string-capitalize index)))))

(defun gen-index (&optional (sections *sections*))
  (page index index
    (with ol
      (dolist (i (all-items sections))
        (link-item (item-id i) (item-title i))
        (brs 2)))))

(defun all-items (sections)
  (let ((is nil))
    (dolist (s sections)
      (dolist (i (section-items s))
        (setf is (merge 'list (list i) is #'title<))))
    is))

(defun title< (x y)
  (string-lessp (item-title x) (item-title y)))
```

圖 **16.9** 生成索引和目錄

圖 16.9 包含的函數用於生成索引和目錄。 常數 `contents` 和 `index` 都是字元串，它們分別用作 `contents` 頁面的標題和 `index` 頁面的標題；另一方面，如果有其他頁面包含了目錄和索引這兩個頁面，那麼這兩個常數也會作爲這些頁面檔案的前綴名。

函數 `gen-contents` 和 `gen-index` 非常相似。 它們都打開一個 HTML 檔案，生成標題和連結列表。 不同的地方是，索引頁面的項必須是有序的。 有序列表通過 `all-items` 函數生成，它遍歷各個項並將它加入到保存已知項的列表當中，並使用 `title<` 函數作爲排序函數。 注意，因爲 `title<` 函數對大小寫敏感，所以在對比標題前，輸入必須先經過 `string-lessp` 處理，從而忽略大小寫區別。

實際程式中的對比操作通常更複雜一些。舉個例子，它們需要忽略無意義的句首詞彙，比如 `"a"` 和 `"the"` 。

```
(defun gen-site ()
  (map3 #'gen-section *sections*)
  (gen-contents)
  (gen-index))

(defun gen-section (sect <sect sect>)
  (page (section-id sect) (section-title sect)
    (with ol
      (map3 #'(lambda (item <item item>)
                (link-item (item-id item)
                           (item-title item))
                (brs 2)
                (gen-item sect item <item item>))
            (section-items sect)))
    (brs 3)
    (gen-move-buttons (if <sect (section-id <sect))
                      contents
                      (if sect> (section-id sect>)))))

(defun gen-item (sect item <item item>)
  (page (item-id item) (item-title item)
    (princ (item-text item))
    (brs 3)
    (gen-move-buttons (if <item (item-id <item))
                      (section-id sect)
                      (if item> (item-id item>)))))

(defun gen-move-buttons (back up forward)
  (if back (button back "Back"))
  (if up (button up "Up"))
  (if forward (button forward "Forward")))
```

圖 **16.10** 生成網站、節點和項

圖 16.10 包含其餘的程式： `gen-site` 生成整個頁面集合，並呼叫相應的函數，生成節點和項。

所有頁面的集合包括目錄、索引、各個節點以及各個項的頁面。 目錄和索引的生成由圖 16.9 中的程式完成。 節點和項由分別由生成節點頁面的 `gen-section` 和生成項頁面的 `gen-item` 完成。

這兩個函數的開頭和結尾非常相似。 它們都接受一個物件、物件的左兄弟、物件的右兄弟作爲參數；它們都從物件的 `title` 域中提取標題內容；它們都以呼叫 `gen-move-buttons` 作爲結束，其中 `gen-move-buttons` 創建指向左兄弟的後退按鈕、指向右兄弟的前進按鈕和指向雙親（parent）物件的向上按鈕。 它們的不同在於函數體的中間部分： `gen-section` 創建有序列表，列表中的連結指向節點包含的項，而 `gen-item` 創建的項則連結到相應的文字頁面。

項所包含的內容完全由用戶決定。 比如說，將 HTML 標籤作爲內容也是完全沒問題的。 項的文字當然也可以由其他程式來生成。

圖 16.11 示範了如何手工地定義一個微型網頁。 在這個例子中，列出的項都是 Fortune 餅乾公司新推出的產品。

```
(defitem des "Fortune Cookies: Dessert or Fraud?" "...")

(defitem case "The Case for Pessimism" "...")

(defsection position "Position Papers" des case)

(defitem luck "Distribution of Bad Luck" "...")

(defitem haz "Health Hazards of Optimism" "...")

(defsection abstract "Research Abstracts" luck haz)

(defsite position abstract)
```

圖 **16.11** 一個微型網站

第十七章：範例：物件

在本章裡，我們將使用 Lisp 來自己實現物件導向語言。這樣子的程式稱爲嵌入式語言 (*embedded language*)。嵌入一個物件導向語言到 Lisp 裡是一個絕佳的例子。同時作爲一個 Lisp 的典型用途，並示範了面向物件的抽象是如何多自然地在 Lisp 基本的抽象上構建出來。

17.1 繼承 (Inheritance)

11.10 小節解釋過通用函數與訊息傳遞的差別。

在訊息傳遞模型裡，

1. 物件有屬性，
2. 並回應消息，
3. 並從其父類繼承屬性與方法。

當然了，我們知道 CLOS 使用的是通用函數模型。但本章我們只對於寫一個迷你的物件系統 (minimal object system)感興趣，而不是一個可與 CLOS 匹敵的系統，所以我們將使用訊息傳遞模型。

我們已經在 Lisp 裡看過許多保存屬性集合的方法。一種可能的方法是使用雜湊表來代表物件，並將屬性作爲雜湊表的條目保存。接著可以通過 `gethash` 來存取每個屬性：

```
(gethash 'color obj)
```

由於函數是資料物件，我們也可以將函數作爲屬性保存起來。這表示我們也可以有方法；要呼叫一個物件特定的方法，可以通過 `funcall` 一下雜湊表裡的同名屬性：

```
(funcall (gethash 'move obj) obj 10)
```

我們可以在這個概念上，定義一個 Smalltalk 風格的訊息傳遞語法，

```
(defun tell (obj message &rest args)
  (apply (gethash message obj) obj args))
```

所以想要一個物件 `obj` 移動 10 單位，我們可以說：

```
(tell obj 'move 10)
```

事實上，純 Lisp 唯一缺少的原料是繼承。我們可以通過定義一個遞迴版本的 gethash 來實現一個簡單版，如圖 17.1 。現在僅用共 8 行程式碼，便實現了物件導向程式設計的 3 個基本元素。

```
(defun rget (prop obj)
  (multiple-value-bind (val in) (gethash prop obj)
    (if in
        (values val in)
        (let ((par (gethash :parent obj)))
          (and par (rget prop par))))))

(defun tell (obj message &rest args)
  (apply (rget message obj) obj args))
```

圖 **17.1**：繼承

讓我們用這段程式，來試試本來的例子。我們創建兩個物件，其中一個物件是另一個的子類：

```
> (setf circle-class (make-hash-table)
        our-circle   (make-hash-table)
        (gethash :parent our-circle) circle-class
        (gethash 'radius our-circle) 2)
2
```

circle-class 物件會持有給所有圓形使用的 area 方法。它是接受一個參數的函數，該參數爲傳來原始消息的物件：

```
> (setf (gethash 'area circle-class)
        #'(lambda (x)
            (* pi (expt (rget 'radius x) 2))))
#<Interpreted-Function BF1EF6>
```

現在當我們詢問 our-circle 的面積時，會根據此類所定義的方法來計算。我們使用 rget 來讀取一個屬性，用 tell 來呼叫一個方法：

```
> (rget 'radius our-circle)
2
T
> (tell our-circle 'area)
12.566370614359173
```

在開始改善這個程式之前，值得停下來想想我們到底做了什麼。僅使用 8 行程式碼，我們使純的、舊的、無 CLOS 的 Lisp ，轉變成一個物件導向語言。我們是怎麼完成這項壯舉的？應該用了某種祕訣，才會僅用了 8 行程式碼，就實現了物件導向程式設計。

的確有一個祕訣存在，但不是編程的奇技淫巧。這個祕訣是，Lisp 本來就是一個面向物件的語言了，甚至說，是種更通用的語言。我們需要做的事情，不過就是把本來就存在的抽象，再重新包裝一下。

17.2 多重繼承 (Multiple Inheritance)

到目前爲止我們只有單繼承 —— 一個物件只可以有一個父類。但可以通過使 `parent` 屬性變成一個列表來獲得多重繼承，並重新定義 `rget` ，如圖 17.2 所示。

在只有單繼承的情況下，當我們想要從物件取出某些屬性，只需要遞迴地延著祖先的方嚮往上找。如果物件本身沒有我們想要屬性的有關資訊，可以檢視其父類，以此類推。有了多重繼承後，我們仍想要執行同樣的搜索，但這件簡單的事，卻被物件的祖先可形成一個圖，而不再是簡單的樹給複雜化了。不能只使用深度優先來搜索這個圖。有多個父類時，可以有如圖 17.3 所示的層級存在： `a` 起源於 `b` 及 `c` ，而他們都是 `d` 的子孫。一個深度優先（或說高度優先）的遍歷結果會是 `a` , `b` , `d`, `c` , `d` 。而如果我們想要的屬性在 `d` 與 `c` 都有的話，我們會獲得存在 `d` 的值，而不是存在 `c` 的值。這違反了子類可覆寫父類提供預設值的原則。

如果我們想要實現普遍的繼承概念，就不應該在檢查其子孫前，先檢查該物件。在這個情況下，適當的搜索順序會是 `a` , `b` , `c` , `d` 。那如何保證搜索總是先搜子孫呢？最簡單的方法是用一個物件，以及按正確優先順序排序的，由祖先所構成的列表。通過呼叫 `traverse` 開始，建構一個列表，表示深度優先遍歷所遇到的物件。如果任一個物件有共享的父類，則列表中會有重復元素。如果僅保存最後出現的複本，會獲得一般由 CLOS 定義的優先序列表。（刪除所有除了最後一個之外的複本，根據 183 頁所描述的算法，規則三。）Common Lisp 函數 `delete-duplicates` 定義成如此作用的，所以我們只要在深度優先的基礎上呼叫它，我們就會得到正確的優先序列表。一旦優先序列表創建完成， `rget` 根據需要的屬性搜索第一個符合的物件。

我們可以通過利用優先序列表的優點，舉例來說，一個愛國的無賴先是一個無賴，然後才是愛國者：

```
> (setf scoundrel            (make-hash-table)
        patriot              (make-hash-table)
        patriotic-scoundrel (make-hash-table)
        (gethash 'serves scoundrel) 'self
        (gethash 'serves patriot) 'country
        (gethash :parents patriotic-scoundrel)
                 (list scoundrel patriot))
(#<Hash-Table C41C7E> #<Hash-Table C41F0E>)
> (rget 'serves patriotic-scoundrel)
SELF
T
```

到目前爲止，我們有一個強大的程式，但極其醜陋且低效。在一個 Lisp 程式生命週期的第二階段，我們將這個初步框架提煉成有用的東西。

17.3 定義物件 (Defining Objects)

第一個我們需要改善的是，寫一個用來創建物件的函數。我們程式表示物件以及其父類的方式，不需要給用戶知道。如果我們定義一個函數來創建物件，用戶將能夠一個步驟就創建出一個物件，並指定其父類。我們可以在創建一個物件的同時，順道構造優先序列表，而不是在每次當我們需要找一個屬性或方法時，才花費龐大代價來重新構造。

如果我們要維護優先序列表，而不是在要用的時候再構造它們，我們需要處理列表會過時的可能性。我們的策略會是用一個列表來保存所有存在的物件，而無論何時當某些父類被改動時，重新給所有受影響的物件生成優先序列表。這代價是相當昂貴的，但由於查詢比重定義父類的可能性來得高許多，我們會省下許多時間。這個改變對我們的程式的靈活性沒有任何影響；我們只是將花費從頻繁的操作轉到不頻繁的操作。

圖 17.4 包含了新的程式。 λ [http://acl.readthedocs.org/en/latest/zhTW/notes.html#notes-273] 全局的 `*objs*` 會是一個包含所有當前物件的列表。函數 `parents` 取出一個物件的父類；相反的 `(setf parents)` 不僅配置一個物件的父類，也呼叫 `make-precedence` 來重新構造任何需要變動的優先序列表。這些列表與之前一樣，由 `precedence` 來構造。

用戶現在不用呼叫 `make-hash-table` 來創建物件，呼叫 `obj` 來取代， `obj` 一步完成創建一個新物件及定義其父類。我們也重定義了 `rget` 來利用保存優先序列表的好處。

```
(defvar *objs* nil)

(defun parents (obj) (gethash :parents obj))

(defun (setf parents) (val obj)
  (prog1 (setf (gethash :parents obj) val)
         (make-precedence obj)))

(defun make-precedence (obj)
  (setf (gethash :preclist obj) (precedence obj))
  (dolist (x *objs*)
    (if (member obj (gethash :preclist x))
        (setf (gethash :preclist x) (precedence x)))))

(defun obj (&rest parents)
  (let ((obj (make-hash-table)))
    (push obj *objs*)
    (setf (parents obj) parents)
    obj))

(defun rget (prop obj)
  (dolist (c (gethash :preclist obj))
```

```
    (multiple-value-bind (val in) (gethash prop c)
      (if in (return (values val in))))))
```

圖 **17.4:** 創建物件

17.4 函數式語法 (Functional Syntax)

另一個可以改善的空間是消息呼叫的語法。 `tell` 本身是無謂的雜亂不堪，這也使得動詞在第三順位才出現，同時代表著我們的程式不再可以像一般 Lisp 前序表達式那樣閱讀:

```
(tell (tell obj 'find-owner) 'find-owner)
```

我們可以使用圖 17.5 所定義的 `defprop` 宏，通過定義作爲函數的屬性名稱來擺脫這種 `tell` 語法。若選擇性參數 `meth?` 爲真的話，會將此屬性視爲方法。不然會將屬性視爲槽，而由 `rget` 所取回的值會直接返回。一旦我們定義了屬性作爲槽或方法的名字，

```
(defmacro defprop (name &optional meth?)
  `(progn
     (defun ,name (obj &rest args)
       ,(if meth?
          `(run-methods obj ',name args)
          `(rget ',name obj)))
     (defun (setf ,name) (val obj)
       (setf (gethash ',name obj) val))))

(defun run-methods (obj name args)
  (let ((meth (rget name obj)))
    (if meth
        (apply meth obj args)
        (error "No ~A method for ~A." name obj))))
```

圖 **17.5:** 函數式語法

```
(defprop find-owner t)
```

我們就可以在函數呼叫裡引用它，則我們的程式讀起來將會再次回到 Lisp 本來那樣：

```
(find-owner (find-owner obj))
```

我們的前一個例子在某種程度上可讀性變得更高了：

```
> (progn
    (setf scoundrel           (obj)
          patriot             (obj)
```

```
          patriotic-scoundrel (obj scoundrel patriot))
    (defprop serves)
    (setf (serves scoundrel) 'self
          (serves patriot) 'country)
    (serves patriotic-scoundrel))
SELF
T
```

17.5 定義方法 (Defining Methods)

到目前爲止，我們藉由敘述如下的東西來定義一個方法：

```
(defprop area t)

(setf circle-class (obj))

(setf (area circle-class)
      #'(lambda (c) (* pi (expt (radius c) 2))))
```

```
(defmacro defmeth (name obj parms &rest body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (setf (gethash ',name ,gobj)
             (labels ((next () (get-next ,gobj ',name)))
               #'(lambda ,parms ,@body))))))

(defun get-next (obj name)
  (some #'(lambda (x) (gethash name x))
        (cdr (gethash :preclist obj))))
```

圖 **17.6** 定義方法。

在一個方法裡，我們可以通過給物件的 `:preclist` 的 `cdr` 獲得如內建 `call-next-method` 方法的效果。所以舉例來說，若我們想要定義一個特殊的圓形，這個圓形在返回面積的過程中印出某個東西，我們可以說：

```
(setf grumpt-circle (obj circle-class))

(setf (area grumpt-circle)
      #'(lambda (c)
          (format t "How dare you stereotype me!~%")
          (funcall (some #'(lambda (x) (gethash 'area x))
                         (cdr (gethash :preclist c)))
                   c)))
```

這裡 `funcall` 等同於一個 `call-next-method` 呼叫，但他..

圖 17.6 的 defmeth 宏提供了一個便捷方式來定義方法，並使得呼叫下個方法變得簡單。一個 defmeth 的呼叫會展開成一個 setf 表達式，但 setf 在一個 labels 表達式裡定義了 next 作爲取出下個方法的函數。這個函數與 next-method-p 類似（第 188 頁「譯註: 11.7 節」），但返回的是我們可以呼叫的東西，同時作為 call-next-method 。λ [http://acl.readthedocs.org/en/latest/zhTW/notes.html#notes-273] 前述兩個方法可以被定義成：

```
(defmeth area circle-class (c)
  (* pi (expt (radius c) 2)))

(defmeth area grumpy-circle (c)
  (format t "How dare you stereotype me!~%")
  (funcall (next) c))
```

順道一提，注意 defmeth 的定義也利用到了符號捕捉。方法的主體被插入至函數 next 是區域定義的一個上下文裡。

17.6 實體 (Instances)

到目前爲止，我們還沒有將類別與實體做區別。我們使用了一個術語來表示兩者，物件 (*object*)。將所有的物件視爲一體是優雅且靈活的，但這非常沒效率。在許多面向物件應用裡，繼承圖的底部會是複雜的。舉例來說，模擬一個交通情況，我們可能有少於十個物件來表示車子的種類，但會有上百個物件來表示特定的車子。由於後者會全部共享少數的優先序列表，創建它們是浪費時間的，並且浪費空間來保存它們。

圖 17.7 定義一個宏 inst ，用來創建實體。實體就像其他物件一樣（現在也可稱爲類別），有區別的是只有一個父類且不需維護優先序列表。它們也沒有包含在列表 *objs** 裡。在前述例子裡，我們可以說：

```
(setf grumpy-circle (inst circle-class))
```

由於某些物件不再有優先序列表，函數 rget 以及 get-next 現在被重新定義，檢查這些物件的父類來取代。獲得的效率不用拿靈活性交換。我們可以對一個實體做任何我們可以給其它種物件做的事，包括創建一個實體以及重定義其父類。在後面的情況裡，(setf parents) 會有效地將物件轉換成一個“類別”。

17.7 新的實現 (New Implementation)

我們到目前爲止所做的改善都是犧牲靈活性交換而來。在這個系統的開發後期，一個 Lisp 程式通常可以犧牲些許靈活性來獲得好處，這裡也不例外。目前爲止我們使用雜湊表來表示所有的物件。這給我們帶來了超乎我們所需的靈活性，以及超乎我們所想的花

費。在這個小節裡，我們會重寫我們的程式，用簡單向量來表示物件。

```
(defun inst (parent)
  (let ((obj (make-hash-table)))
    (setf (gethash :parents obj) parent)
    obj))

(defun rget (prop obj)
  (let ((prec (gethash :preclist obj)))
    (if prec
        (dolist (c prec)
          (multiple-value-bind (val in) (gethash prop c)
            (if in (return (values val in)))))
      (multiple-value-bind (val in) (gethash prop obj)
        (if in
            (values val in)
            (rget prop (gethash :parents obj)))))))

(defun get-next (obj name)
  (let ((prec (gethash :preclist obj)))
    (if prec
        (some #'(lambda (x) (gethash name x))
              (cdr prec))
      (get-next (gethash obj :parents) name))))
```

圖 **17.7:** 定義實體

這個改變意味著放棄動態定義新屬性的可能性。目前我們可通過引用任何物件，給它定義一個屬性。現在當一個類別被創建時，我們會需要給出一個列表，列出該類有的新屬性，而當實體被創建時，他們會恰好有他們所繼承的屬性。

在先前的實現裡，類別與實體沒有實際區別。一個實體只是一個恰好有一個父類的類別。如果我們改動一個實體的父類，它就變成了一個類別。在新的實現裡，類別與實體有實際區別；它使得將實體轉成類別不再可能。

在圖 17.8-17.10 的程式是一個完整的新實現。圖片 17.8 給創建類別與實體定義了新的運算子。類別與實體用向量來表示。表示類別與實體的向量的前三個元素包含程式自身要用到的資訊，而圖 17.8 的前三個宏是用來引用這些元素的：

```
(defmacro parents (v) `(svref ,v 0))
(defmacro layout (v) `(the simple-vector (svref ,v 1)))
(defmacro preclist (v) `(svref ,v 2))

(defmacro class (&optional parents &rest props)
  `(class-fn (list ,@parents) ',props))

(defun class-fn (parents props)
  (let* ((all (union (inherit-props parents) props))
         (obj (make-array (+ (length all) 3)
```

```
                          :initial-element :nil)))
    (setf (parents obj)  parents
          (layout obj)   (coerce all 'simple-vector)
          (preclist obj) (precedence obj))
    obj))

(defun inherit-props (classes)
  (delete-duplicates
    (mapcan #'(lambda (c)
                (nconc (coerce (layout c) 'list)
                       (inherit-props (parents c))))
            classes)))

(defun precedence (obj)
  (labels ((traverse (x)
             (cons x
                   (mapcan #'traverse (parents x)))))
    (delete-duplicates (traverse obj))))

(defun inst (parent)
  (let ((obj (copy-seq parent)))
    (setf (parents obj)  parent
          (preclist obj) nil)
    (fill obj :nil :start 3)
    obj))
```

圖 **17.8:** 向量實現：創建

1. `parents` 欄位取代舊實現中，雜湊表條目裡 `:parents` 的位置。在一個類別裡，`parents` 會是一個列出父類的列表。在一個實體裡，`parents` 會是一個單一的父類。
2. `layout` 欄位是一個包含屬性名字的向量，指出類別或實體的從第四個元素開始的設計 (layout)。
3. `preclist` 欄位取代舊實現中，雜湊表條目裡 `:preclist` 的位置。它會是一個類別的優先序列表，實體的話就是一個空表。

因爲這些運算子是宏，他們全都可以被 `setf` 的第一個參數使用（參考 10.6 節）。

`class` 宏用來創建類別。它接受一個含有其基類的選擇性列表，伴隨著零個或多個屬性名稱。它返回一個代表類別的物件。新的類別會同時有自己本身的屬性名，以及從所有基類繼承而來的屬性。

```
> (setf *print-array* nil
        gemo-class (class nil area)
        circle-class (class (geom-class) radius))
#<Simple-Vector T 5 C6205E>
```

這裡我們創建了兩個類別：`geom-class` 沒有基類，且只有一個屬性，`area`；`circle-`

class 是 gemo-class 的子類，並添加了一個屬性， radius 。 [1] circle-class 類的設計

```
> (coerce (layout circle-class) 'list)
(AREA RADIUS)
```

顯示了五個欄位裡，最後兩個的名稱。 [2]

class 宏只是一個 class-fn 的介面，而 class-fn 做了實際的工作。它呼叫 inherit-props 來彙整所有新物件的父類，彙整成一個列表，創建一個正確長度的向量，並適當地配置前三個欄位。（ preclist 由 precedence 創建，本質上 precedence 沒什麼改變。）類別餘下的欄位設置爲 :nil 來指出它們尚未初始化。要檢視 circle-class 的 area 屬性，我們可以：

```
> (svref circle-class
         (+ (position 'area (layout circle-class)) 3))
:NIL
```

稍後我們會定義存取函數來自動辦到這件事。

最後，函數 inst 用來創建實體。它不需要是一個宏，因爲它僅接受一個參數：

```
> (setf our-circle (inst circle-class))
#<Simple-Vector T 5 C6464E>
```

比較 inst 與 class-fn 是有益學習的，它們做了差不多的事。因爲實體僅有一個父類，不需要決定它繼承什麼屬性。實體可以僅拷貝其父類的設計。它也不需要構造一個優先序列表，因爲實體沒有優先序列表。創建實體因此與創建類別比起來來得快許多，因爲創建實體在多數應用裡比創建類別更常見。

```
(declaim (inline lookup (setf lookup)))

(defun rget (prop obj next?)
  (let ((prec (preclist obj)))
    (if prec
        (dolist (c (if next? (cdr prec) prec) :nil)
          (let ((val (lookup prop c)))
            (unless (eq val :nil) (return val))))
        (let ((val (lookup prop obj)))
          (if (eq val :nil)
              (rget prop (parents obj) nil)
              val)))))

(defun lookup (prop obj)
  (let ((off (position prop (layout obj) :test #'eq)))
    (if off (svref obj (+ off 3)) :nil)))
```

```
(defun (setf lookup) (val prop obj)
  (let ((off (position prop (layout obj) :test #'eq)))
    (if off
        (setf (svref obj (+ off 3)) val)
        (error "Can't set ~A of ~A." val obj))))
```

圖 **17.9:** 向量實現：存取

現在我們可以創建所需的類別層級及實體，以及需要的函數來讀寫它們的屬性。圖 17.9 的第一個函數是 `rget` 的新定義。它的形狀與圖 17.7 的 `rget` 相似。條件式的兩個分支，分別處理類別與實體。

1. 若物件是一個類別，我們遍歷其優先序列表，直到我們找到一個物件，其中欲找的屬性不是 `:nil` 。如果沒有找到，返回 `:nil` 。
2. 若物件是一個實體，我們直接查找屬性，並在沒找到時遞迴地呼叫 `rget` 。

`rget` 與 `next?` 新的第三個參數稍後解釋。現在只要了解如果是 `nil` ， `rget` 會像平常那樣工作。

函數 `lookup` 及其反相扮演著先前 `rget` 函數裡 `gethash` 的角色。它們使用一個物件的 `layout` ，來取出或設置一個給定名稱的屬性。這條查詢是先前的一個複本：

```
> (lookup 'area circle-class)
:NIL
```

由於 `lookup` 的 `setf` 也定義了，我們可以給 `circle-class` 定義一個 `area` 方法，通過：

```
(setf (lookup 'area circle-class)
      #'(lambda (c)
          (* pi (expt (rget 'radius c nil) 2))))
```

在這個程式裡，和先前的版本一樣，沒有特別區別出方法與槽。一個“方法”只是一個欄位，裡面有著一個函數。這將很快會被一個更方便的前端所隱藏起來。

```
(declaim (inline run-methods))

(defmacro defprop (name &optional meth?)
  `(progn
     (defun ,name (obj &rest args)
       ,(if meth?
            `(run-methods obj ',name args)
            `(rget ',name obj nil)))
     (defun (setf ,name) (val obj)
       (setf (lookup ',name obj) val))))
```

```
(defun run-methods (obj name args)
  (let ((meth (rget name obj nil)))
    (if (not (eq meth :nil))
        (apply meth obj args)
        (error "No ~A method for ~A." name obj))))

(defmacro defmeth (name obj parms &rest body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (defprop ,name t)
       (setf (lookup ',name ,gobj)
             (labels ((next () (rget ,gobj ',name t)))
               #'(lambda ,parms ,@body))))))
```

圖 **17.10:** 向量實現：宏介面

圖 17.10 包含了新的實現的最後部分。這段程式碼沒有給程式加入任何威力，但使程式更容易使用。宏 defprop 本質上沒有改變；現在僅呼叫 lookup 而不是 gethash 。與先前相同，它允許我們用函數式的語法來引用屬性：

```
> (defprop radius)
(SETF RADIUS)
> (radius our-circle)
:NIL
> (setf (radius our-circle) 2)
2
```

如果 defprop 的第二個選擇性參數爲真的話，它展開成一個 run-methods 呼叫，基本上也沒什麼改變。

最後，函數 defmeth 提供了一個便捷方式來定義方法。這個版本有三件新的事情：它隱含了 defprop ，它呼叫 lookup 而不是 gethash ，且它呼叫 regt 而不是 278 頁的 get-next (譯註: 圖 17.7 的 get-next)來獲得下個方法。現在我們理解給 rget 添加額外參數的理由。它與 get-next 非常相似，我們同樣通過添加一個額外參數，在一個函數裡實現。若這額外參數爲真時， rget 取代 get-next 的位置。

現在我們可以達到先前方法定義所有的效果，但更加清晰：

```
(defmeth area circle-class (c)
  (* pi (expt (radius c) 2)))
```

注意我們可以直接呼叫 radius 而無須呼叫 rget ，因爲我們使用 defprop 將它定義成一個函數。因爲隱含的 defprop 由 defmeth 實現，我們也可以呼叫 area 來獲得 our-circle 的面積：

```
> (area our-circle)
```

```
12.566370614359173
```

17.8 分析 (Analysis)

我們現在有了一個適合撰寫實際面向物件程式的嵌入式語言。它很簡單，但就大小來說相當強大。而在典型應用裡，它也會是快速的。在一個典型的應用裡，操作實體應比操作類別更常見。我們重新設計的重點在於如何使得操作實體的花費降低。

在我們的程式裡，創建類別既慢且產生了許多垃圾。如果類別不是在速度爲關鍵考量時創建，這還是可以接受的。會需要速度的是存取函數以及創建實體。這個程式裡的沒有做編譯優化的存取函數大約與我們預期的一樣快。 λ [http://acl.readthedocs.org/en/latest/zhTW/notes.html#notes-284] 而創建實體也是如此。且兩個操作都沒有用到構造 (consing)。除了用來表達實體的向量例外。會自然的以爲這應該是動態地配置才對。但我們甚至可以避免動態配置實體，如果我們使用像是 13.4 節所提出的策略。

我們的嵌入式語言是 Lisp 編程的一個典型例子。只不過是一個嵌入式語言就可以是一個例子了。但 Lisp 的特性是它如何從一個小的、受限版本的程式，進化成一個強大但低效的版本，最終演化成快速但稍微受限的版本。

Lisp 惡名昭彰的緩慢不是 Lisp 本身導致（Lisp 編譯器早在 1980 年代就可以產生出與 C 編譯器一樣快的程式碼），而是由於許多程式設計師在第二個階段就放棄的事實。如同 Richard Gabriel 所寫的，

> 要在 Lisp 撰寫出性能極差的程式相當簡單；而在 C 這幾乎是不可能的。 λ [http://acl.readthedocs.org/en/latest/zhTW/notes.html#notes-284-2]

這完全是一個真的論述，但也可以解讀爲讚揚或貶低 Lisp 的論點：

1. 通過犧牲靈活性換取速度，你可以在 Lisp 裡輕鬆地寫出程式；在 C 語言裡，你沒有這個選擇。
2. 除非你優化你的 Lisp 程式，不然要寫出緩慢的軟體根本易如反掌。

你的程式屬於哪一種解讀完全取決於你。但至少在開發初期，Lisp 使你有犧牲執行速度來換取時間的選擇。

有一件我們範例程式沒有做的很好的事是，它不是一個稱職的 CLOS 模型（除了可能沒有說明難以理解的 `call-next-method` 如何工作是件好事例外）。如大象般龐大的 CLOS 與這個如蚊子般微小的 70 行程式之間，存在多少的相似性呢？當然，這兩者的差別是出自於教育性，而不是探討有多相似。首先，這使我們理解到“面向物件”的廣度。我們的程式比任何被稱爲是面向物件的都來得強大，而這只不過是 CLOS 的一小部

分威力。

我們程式與 CLOS 不同的地方是，方法是屬於某個物件的。這個方法的概念使它們與對第一個參數做派發的函數相同。而當我們使用函數式語法來呼叫方法時，這看起來就跟 Lisp 的函數一樣。相反地，一個 CLOS 的通用函數，可以派發它的任何參數。一個通用函數的組件稱爲方法，而若你將它們定義成只對第一個參數特化，你可以製造出它們是某個類或實體的方法的錯覺。但用物件導向程式設計的訊息傳遞模型來思考 CLOS 最終只會使你困惑，因爲 CLOS 凌駕在物件導向程式設計之上。

CLOS 的缺點之一是它太龐大了，並且 CLOS 費煞苦心的隱藏了物件導向程式設計，其實只不過是改寫 Lisp 的這個事實。本章的例子至少闡明了這一點。如果我們滿足於舊的訊息傳遞模型，我們可以用一頁多一點的程式碼來實現。物件導向程式設計不過是 Lisp 可以做的小事之一而已。更發人深省的問題是，Lisp 除此之外還能做些什麼？

腳註

[1] 當類別被顯示時， `*print-array*` 應當是 `nil` 。 任何類別的 `preclist` 的第一個元素都是類別本身，所以試圖顯示類別的內部結構會導致一個無限迴圈。

[2] 這個向量被 coerced 成一個列表，只是爲了看看裡面有什麼。有了 `*print-array*` 被設成 `nil` ，一個向量的內容應該不會顯示出來。

附錄 A：除錯

這個附錄示範了如何除錯 Lisp 程式，並給出你可能會遇到的常見錯誤。

中斷迴圈 (Breakloop)

如果你要求 Lisp 做些它不能做的事，求值過程會被一個錯誤訊息中斷，而你會發現你位於一個稱爲中斷迴圈的地方。中斷迴圈工作的方式取決於不同的實現，但通常它至少會顯示三件事：一個錯誤資訊，一組選項，以及一個特別的提示符。

在中斷迴圈裡，你也可以像在頂層那樣給表達式求值。在中斷迴圈裡，你或許能夠找出錯誤的起因，甚至是修正它，並繼續你程式的求值過程。然而，在一個中斷迴圈裡，你想做的最常見的事是跳出去。多數的錯誤起因於打錯字或是小疏忽，所以通常你只會想終止程式並返回頂層。在下面這個假定的實現裡，我們輸入 `:abort` 來回到頂層。

```
> (/ 1 0)
Error: Division by zero.
       Options: :abort, :backtrace
>> :abort
>
```

在這些情況裡，實際上的輸入取決於實現。

當你在中斷迴圈裡，如果一個錯誤發生的話，你會到另一個中斷迴圈。多數的 Lisp 會指出你是在第幾層的中斷迴圈，要嘛通過印出多個提示符，不然就是在提示符前印出數字：

```
>> (/ 2 0)
Error: Division by zero.
       Options: :abort, :backtrace, :previous
>>>
```

現在我們位於兩層深的中斷迴圈。此時我們可以選擇回到前一個中斷迴圈，或是直接返回頂層。

追蹤與回溯 (Traces and Backtraces)

當你的程式不如你預期的那樣工作時，有時候第一件該解決的事情是，它在做什麼？如果你輸入 `(trace foo)`，則 Lisp 會在每次呼叫或返回 `foo` 時顯示一個資訊，顯示傳給 `foo` 的參數，或是 `foo` 返回的值。你可以追蹤任何自己定義的 (user-defined)函數。

一個追蹤通常會根據呼叫樹來縮進。在一個做遍歷的函數，像下面這個函數，它給一個樹的每一個非空元素加上 1，

```
(defun tree1+ (tr)
  (cond ((null tr) nil)
        ((atom tr) (1+ tr))
        (t (cons (tree1+ (car tr))
                 (tree1+ (cdr tr))))))
```

一個樹的形狀會因此反映出它被遍歷時的資料結構：

```
> (trace tree1+)
(tree1+)
> (tree1+ '((1 . 3) 5 . 7))
1 Enter TREE1+ ((1 . 3) 5 . 7)
  2 Enter TREE1+ (1.3)
    3 Enter TREE1+ 1
    3 Exit TREE1+ 2
    3 Enter TREE1+ 3
    3 Exit TREE1+ 4
  2 Exit TREE1+ (2 . 4)
  2 Enter TREE1+ (5 . 7)
    3 Enter TREE1+ 5
    3 Exit TREE1+ 6
    3 Enter TREE1+ 7
    3 Exit TREE1+ 8
  2 Exit TREE1+ (6 . 8)
1 Exit TREE1+ ((2 . 4) 6 . 8)
((2 . 4) 6 . 8)
```

要關掉 `foo` 的追蹤，輸入 `(untrace foo)`；要關掉所有正在追蹤的函數，只要輸入 `(untrace)` 就好。

一個更靈活的追蹤辦法是在你的程式碼裡插入診斷性的打印語句。如果已經知道結果了，這個經典的方法大概會與複雜的調適工具一樣被使用數十次。這也是爲什麼可以互動地重定義函數式多麼有用的原因。

一個回溯 (*backtrace*)是一個當前存在棧的呼叫的列表，當一個錯誤中止求值時，會由一個中斷迴圈生成此列表。如果追蹤像是"讓我看看你在做什麼"，一個回溯像是詢問"我們是怎麼到達這裡的？" 在某方面上，追蹤與回溯是互補的。一個追蹤會顯示在一個程式的呼叫樹裡，選定函數的呼叫。一個回溯會顯示在一個程式部分的呼叫樹裡，所有函數的呼叫（路徑爲從頂層呼叫到發生錯誤的地方）。

在一個典型的實現裡，我們可通過在中斷迴圈裡輸入 `:backtrace` 來獲得一個回溯，看起來可能像下面這樣：

```
> (tree1+ ' ( ( 1 . 3) 5 . A))
```

```
Error: A is not a valid argument to 1+.
       Options: :abort, :backtrace
» :backtrace
(1+ A)
(TREE1+ A)
(TREE1+ (5 . A))
(TREE1+ ((1 . 3) 5 . A))
```

出現在回溯裡的臭蟲較容易被發現。你可以僅往回檢視呼叫鏈，直到你找到第一個不該發生的事情。另一個函數式編程 (2.12 節)的好處是所有的臭蟲都會在回溯裡出現。在純函數式程式碼裡，每一個可能出錯的呼叫，在錯誤發生時，一定會在棧出現。

一個回溯每個實現所提供的資訊量都不同。某些實現會完整顯示一個所有待呼叫的歷史，並顯示參數。其他實現可能僅顯示呼叫歷史。一般來說，追蹤與回溯解釋型的程式碼會得到較多的資訊，這也是爲什麼你要在確定你的程式可以工作之後，再來編譯。

傳統上我們在解釋器裡除錯程式碼，且只在工作的情況下才編譯。但這個觀點也是可以改變的：至少有兩個 Common Lisp 實現沒有包含解釋器。

當什麼事都沒發生時 (When Noting Happens)

不是所有的 bug 都會打斷求值過程。另一個常見並可能更危險的情況是，當 Lisp 好像不鳥你一樣。通常這是程式進入無窮迴圈的徵兆。

如果你懷疑你進入了無窮迴圈，解決方法是中止執行，並跳出中斷迴圈。

如果迴圈是用迭代寫成的程式碼，Lisp 會開心地執行到天荒地老。但若是用遞迴寫成的程式碼（沒有做尾遞迴優化），你最終會獲得一個資訊，資訊說 Lisp 把棧的空間給用光了：

```
> (defun blow-stack () (1+ (blow-stack)))
BLOW-STACK
> (blow-stack)
Error: Stack Overflow
```

在這兩個情況裡，如果你懷疑進入了無窮迴圈，解決辦法是中斷執行，並跳出由於中斷所產生的中斷迴圈。

有時候程式在處理一個非常龐大的問題時，就算沒有進入無窮迴圈，也會把棧的空間用光。雖然這很少見。通常把棧空間用光是編程錯誤的徵兆。

遞迴函數最常見的錯誤是忘記了基本用例 (base case)。用英語來描述遞迴，通常會忽略基本用例。不嚴謹地說，我們可能說“obj 是列表的成員，如果它是列表的第一個元素，或是剩餘列表的成員” 嚴格上來講，應該添加一句“若列表爲空，則 obj 不是列表的成

員”。不然我們描述的就是個無窮遞迴了。

在 Common Lisp 裡，如果給入 `nil` 作爲參數， `car` 與 `cdr` 皆返回 `nil`：

```
> (car nil)
NIL
> (cdr nil)
NIL
```

所以若我們在 `member` 函數裡忽略了基本用例：

```
(defun our-member (obj lst)
  (if (eql (car lst) obj)
      lst
      (our-member obj (cdr lst))))
```

要是我們找的物件不在列表裡的話，則會陷入無窮迴圈。當我們到達列表底端而無所獲時，遞迴呼叫會等價於：

```
(our-member obj nil)
```

在正確的定義中（第十六頁「譯註: 2.7 節」），基本用例在此時會停止遞迴，並返回 `nil`。但在上面錯誤的定義裡，函數愚昧地尋找 `nil` 的 `car`，是 `nil`，並將 `nil` 拿去跟我們尋找的物件比較。除非我們要找的物件剛好是 `nil`，不然函數會繼續在 `nil` 的 `cdr` 裡尋找，剛好也是 `nil` —— 整個過程又重來了。

如果一個無窮迴圈的起因不是那麼直觀，可能可以通過看看追蹤或回溯來診斷出來。無窮迴圈有兩種。簡單發現的那種是依賴程式結構的那種。一個追蹤或回溯會即刻示範出，我們的 `our-member` 究竟哪裡出錯了。

比較難發現的那種，是因爲資料結構有缺陷才發生的無窮迴圈。如果你無意中創建了環狀結構（見 199頁「12.3 節」），遍歷結構的程式碼可能會掉入無窮迴圈裡。這些 bug 很難發現，因爲不在後面不會發生，看起來像沒有錯誤的程式碼一樣。最佳的解決辦法是預防，如同 199 頁所描述的：避免使用破壞性操作，直到程式已經正常工作，且你已準備好要調優程式碼來獲得效率。

如果 Lisp 有不鳥你的傾向，也有可能是等待你完成輸入什麼。在多數系統裡，按下 Enter 是沒有效果的，直到你輸入了一個完整的表達式。這個方法的好事是它允許你輸入多行的表達式。壞事是如果你無意中少了一個閉括號，或是一個閉引號，Lisp 會一直等你，直到你真正完成輸入完整的表達式：

```
> (format t "for example ~A~% 'this)
```

這裡我們在控制字串的最後忽略了閉引號。在此時按下回車是沒用的，因爲 Lisp 認爲我們還在輸入一個字串。

在某些實現裡，你可以回到上一行，並插入閉引號。在不允許你回到前行的系統，最佳辦法通常是中斷執行，並從中斷迴圈回到頂層。

沒有值或未綁定 (No Value/Unbound)

一個你最常聽到 Lisp 的抱怨是一個符號沒有值或未綁定。數種不同的問題都用這種方式呈現。

區域變數，如 `let` 與 `defun` 設置的那些，只在創建它們的表達式主體裡合法。所以要是我們試著在 創建變數的 `let` 外部引用它，

```
> (progn
    (let ((x 10))
      (format t "Here x = ~A. ~%" x))
    (format t "But now it's gone...~%")
    x)
Here x = 10.
But now it's gone...
Error: X has no value.
```

我們獲得一個錯誤。當 Lisp 抱怨某些東西沒有值或未綁定時，祂的意思通常是你無意間引用了一個不存在的變數。因爲沒有叫做 `x` 的區域變數，Lisp 假定我們要引用一個有著這個名字的全局變數或常數。錯誤會發生是因爲當 Lisp 試著要查找它的值的時候，卻發現根本沒有給值。打錯變數的名字通常會給出同樣的結果。

一個類似的問題發生在我們無意間將函數引用成變數。舉例來說：

```
> defun foo (x) (+ x 1))
Error: DEFUN has no value
```

這在第一次發生時可能會感到疑惑： `defun` 怎麼可能會沒有值？問題的癥結點在於我們忽略了最初的左括號，導致 Lisp 把符號 `defun` 解讀錯誤，將它視爲一個全局變數的引用。

有可能你真的忘記初始化某個全局變數。如果你沒有給 `defvar` 第二個參數，你的全局變數會被宣告出來，但沒有初始化；這可能是問題的根源。

意料之外的 Nil (Unexpected Nils)

當函數抱怨傳入 `nil` 作爲參數時，通常是程式先前出錯的徵兆。數個內建運算子返回 `nil` 來指出失敗。但由於 `nil` 是一個合法的 Lisp 物件，問題可能之後才發生，在程式某部分試著要使用這個信以爲真的返回值時。

舉例來說，返回一個月有多少天的函數有一個 bug；假設我們忘記十月份了：

```
(defun month-length (mon)
  (case mon
    ((jan mar may jul aug dec) 31)
    ((apr jun sept nov) 30)
    (feb (if (leap-year) 29 28))))
```

如果有另一個函數，企圖想計算出一個月當中有幾個禮拜，

```
(defun month-weeks (mon) (/ (month-length mon) 7.0))
```

則會發生下面的情形：

```
> (month-weeks 'oct)
Error: NIL is not a valud argument to /.
```

問題發生的原因是因爲 `month-length` 在 `case` 找不到匹配 。當這個情形發生時， `case` 返回 `nil` 。然後 `month-weeks` ，認爲獲得了一個數字，將值傳給 `/` ， `/` 就抱怨了。

在這裡最起碼 bug 與 bug 的臨牀表現是挨著發生的。這樣的 bug 在它們相距很遠時很難找到。要避免這個可能性，某些 Lisp 方言讓跑完 `case` 或 `cond` 又沒匹配的情形，產生一個錯誤。在 Common Lisp 裡，在這種情況裡可以做的是使用 `ecase` ，如 14.6 節所描述的。

重新命名 (Renaming)

在某些場合裡（但不是全部場合），有一種特別狡猾的 bug ，起因於重新命名函數或變數，。舉例來說，假設我們定義下列（低效的） 函數來找出雙重巢狀列表的深度：

```
(defun depth (x)
  (if (atom x)
      1
      (1+ (apply #'max (mapcar #'depth x)))))
```

測試函數時，我們發現它給我們錯誤的答案（應該是 1）：

```
> (depth '((a)))
3
```

起初的 1 應該是 0 才對。如果我們修好這個錯誤，並給這個函數一個較不模糊的名稱：

```
(defun nesting-depth (x)
  (if (atom x)
      0
      (1+ (apply #'max (mapcar #'depth x)))))
```

當我們再測試上面的例子，它返回同樣的結果：

```
> (nesting-depth '((a)))
3
```

我們不是修好這個函數了嗎？沒錯，但答案不是來自我們修好的程式碼。我們忘記也改掉遞迴呼叫中的名稱。在遞迴用例裡，我們的新函數仍呼叫先前的 depth ，這當然是不對的。

作爲選擇性參數的關鍵字 (Keywords as Optional Parameters)

若函數同時接受關鍵字與選擇性參數，這通常是個錯誤，無心地提供了關鍵字作爲選擇性參數。舉例來說，函數 read-from-string 有著下列的參數列表：

```
(read-from-string string &optional eof-error eof-value
                         &key start end preserve-whitespace)
```

這樣一個函數你需要依序提供值，給所有的選擇性參數，再來才是關鍵字參數。如果你忘記了選擇性參數，看看下面這個例子，

```
> (read-from-string "abcd" :start 2)
ABCD
4
```

則 :start 與 2 會成爲前兩個選擇性參數的值。若我們想要 read 從第二個字元開始讀取，我們應該這麼說：

```
> (read-from-string "abcd" nil nil :start 2)
CD
4
```

錯誤宣告 (Misdeclarations)

第十三章解釋了如何給變數及資料結構做型別宣告。通過給變數做型別宣告，你保證變

數只會包含某種型別的值。當產生程式碼時，Lisp 編譯器會依賴這個假定。舉例來說，這個函數的兩個參數都宣告爲 `double-floats`，

```
(defun df* (a b)
  (declare (double-float a b))
  (* a b))
```

因此編譯器在產生程式碼時，被授權直接將浮點乘法直接硬連接 (hard-wire)到程式碼裡。

如果呼叫 `df*` 的參數不是宣告的型別時，可能會捕捉一個錯誤，或單純地返回垃圾。在某個實現裡，如果我們傳入兩個定長數，我們獲得一個硬體中斷：

```
> (df* 2 3)
Error: Interrupt.
```

如果獲得這樣嚴重的錯誤，通常是由於數值不是先前宣告的型別。

警告 (Warnings)

有些時候 Lisp 會抱怨一下，但不會中斷求值過程。許多這樣的警告是錯誤的警鐘。一種最常見的可能是由編譯器所產生的，關於未宣告或未使用的變數。舉例來說，在 66 頁「譯註: 6.4 節」， `map-int` 的第二個呼叫，有一個 `x` 變數沒有使用到。如果想要編譯器在每次編譯程式時，停止通知你這些事，使用一個忽略宣告：

```
(map-int #'(lambda (x)
             (declare (ignore x))
             (random 100))
         10)
```

附錄 B：Lisp in Lisp

這個附錄包含了 58 個最常用的 Common Lisp 運算子。因爲如此多的 Lisp 是（或可以）用 Lisp 所寫成，而由於 Lisp 程式（或可以）相當精簡，這是一種方便解釋語言的方式。

這個練習也證明了，概念上 Common Lisp 不像看起來那樣龐大。許多 Common Lisp 運算子是有用的函式庫；要寫出所有其它的東西，你所需要的運算子相當少。在這個附錄的這些定義只需要:

```
apply aref backquote block car cdr ceiling char= cons defmacro documentation eq
error expt fdefinition function floor gensym get-setf-expansion if imagpart
labels length multiple-value-bind nth-value quote realpart symbol-function
tagbody type-of typep = + - / < >
```

這裡給出的程式碼作爲一種解釋 Common Lisp 的方式，而不是實現它的方式。在實際的實現上，這些運算子可以更高效，也會做更多的錯誤檢查。爲了方便參找，這些運算子本身按字母順序排列。如果你真的想要這樣定義 Lisp，每個宏的定義需要在任何呼叫它們的程式碼之前。

```
(defun -abs (n)
  (if (typep n 'complex)
      (sqrt (+ (expt (realpart n) 2) (expt (imagpart n) 2)))
      (if (< n 0) (- n) n)))
```

```
(defun -adjoin (obj lst &rest args)
  (if (apply #'member obj lst args) lst (cons obj lst)))
```

```
(defmacro -and (&rest args)
  (cond ((null args) t)
        ((cdr args)  `(if ,(car args) (-and ,@(cdr args))))
        (t           (car args))))
```

```
(defun -append (&optional first &rest rest)
  (if (null rest)
      first
      (nconc (copy-list first) (apply #'-append rest))))
```

```
(defun -atom (x) (not (consp x)))
```

```
(defun -butlast (lst &optional (n 1))
  (nreverse (nthcdr n (reverse lst))))
```

```
(defun -cadr (x) (car (cdr x)))
```

```
(defmacro -case (arg &rest clauses)
  (let ((g (gensym)))
    `(let ((,g ,arg))
       (cond ,@(mapcar #'(lambda (cl)
                           (let ((k (car cl)))
                             `(,(cond ((member k '(t otherwise))
                                       t)
                                      ((consp k)
                                       `(member ,g ',k))
                                      (t `(eql ,g ',k)))
                               (progn ,@(cdr cl)))))
                       clauses)))))
```

```
(defun -cddr (x) (cdr (cdr x)))
```

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

```
(defmacro -cond (&rest args)
  (if (null args)
      nil
      (let ((clause (car args)))
        (if (cdr clause)
            `(if ,(car clause)
                 (progn ,@(cdr clause))
                 (-cond ,@(cdr args)))
            `(or ,(car clause)
                 (-cond ,@(cdr args)))))))
```

```
(defun -consp (x) (typep x 'cons))
```

```
(defun -constantly (x) #'(lambda (&rest args) x))
```

```
(defun -copy-list (lst)
  (labels ((cl (x)
             (if (atom x)
                 x
                 (cons (car x)
                       (cl (cdr x))))))
    (cons (car lst)
          (cl (cdr lst)))))
```

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

```
(defmacro -defun (name parms &rest body)
  (multiple-value-bind (dec doc bod) (analyze-body body)
    `(progn
       (setf (fdefinition ',name)
             #'(lambda ,parms
                 ,@dec
                 (block ,(if (atom name) name (second name))
                   ,@bod))
             (documentation ',name 'function)
             ,doc)
       ',name)))
```

```
(defun analyze-body (body &optional dec doc)
  (let ((expr (car body)))
    (cond ((and (consp expr) (eq (car expr) 'declare))
           (analyze-body (cdr body) (cons expr dec) doc))
          ((and (stringp expr) (not doc) (cdr body))
           (if dec
               (values dec expr (cdr body))
               (analyze-body (cdr body) dec expr)))
          (t (values dec doc body)))))
```

這個定義不完全正確，參見 `let`

```
(defmacro -do (binds (test &rest result) &rest body)
  (let ((fn (gensym)))
    `(block nil
       (labels ((,fn ,(mapcar #'car binds)
                  (cond (,test ,@result)
                        (t (tagbody ,@body)
                           (,fn ,@(mapcar #'third binds))))))
         (,fn ,@(mapcar #'second binds))))))
```

```
(defmacro -dolist ((var lst &optional result) &rest body)
  (let ((g (gensym)))
    `(do ((,g ,lst (cdr ,g)))
         ((atom ,g) (let ((,var nil)) ,result))
       (let ((,var (car ,g)))
         ,@body))))
```

```
(defun -eql (x y)
  (typecase x
    (character (and (typep y 'character) (char= x y)))
    (number    (and (eq (type-of x) (type-of y))
                    (= x y)))
    (t         (eq x y))))
```

```
(defun -evenp (x)
  (typecase x
    (integer (= 0 (mod x 2)))
```

```
    (t        (error "non-integer argument"))))
```

```
(defun -funcall (fn &rest args) (apply fn args))
```

```
(defun -identity (x) x)
```

這個定義不完全正確：表達式 `(let ((&key 1) (&optional 2)))` 是合法的，但它產生的表達式不合法。

```
(defmacro -let (parms &rest body)
  `((lambda ,(mapcar #'(lambda (x)
                         (if (atom x) x (car x)))
                     parms)
      ,@body)
    ,@(mapcar #'(lambda (x)
                  (if (atom x) nil (cadr x)))
              parms)))
```

```
(defun -list (&rest elts) (copy-list elts))
```

```
(defun -listp (x) (or (consp x) (null x)))
```

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

```
(defun -mapcar (fn &rest lsts)
  (cond ((member nil lsts) nil)
        ((null (cdr lsts))
         (let ((lst (car lsts)))
           (cons (funcall fn (car lst))
                 (-mapcar fn (cdr lst)))))
        (t
         (cons (apply fn (-mapcar #'car lsts))
               (apply #'-mapcar fn
                      (-mapcar #'cdr lsts))))))
```

```
(defun -member (x lst &key test test-not key)
  (let ((fn (or test
                (if test-not
                    (complement test-not))
                #'eql)))
    (member-if #'(lambda (y)
                   (funcall fn x y))
               lst
               :key key)))
```

```
(defun -member-if (fn lst &key (key #'identity))
  (cond ((atom lst) nil)
```

```
        ((funcall fn (funcall key (car lst))) lst)
        (t (-member-if fn (cdr lst) :key key))))
```

```
(defun -mod (n m)
  (nth-value 1 (floor n m)))
```

```
(defun -nconc (&optional lst &rest rest)
  (if rest
      (let ((rest-conc (apply #'-nconc rest)))
        (if (consp lst)
            (progn (setf (cdr (last lst)) rest-conc)
                   lst)
            rest-conc))
      lst))
```

```
(defun -not (x) (eq x nil))
(defun -nreverse (seq)
  (labels ((nrl (lst)
             (let ((prev nil))
               (do ()
                   ((null lst) prev)
                 (psetf (cdr lst) prev
                        prev      lst
                        lst       (cdr lst)))))
           (nrv (vec)
             (let* ((len (length vec))
                    (ilimit (truncate (/ len 2))))
               (do ((i 0 (1+ i))
                    (j (1- len) (1- j)))
                   ((>= i ilimit) vec)
                 (rotatef (aref vec i) (aref vec j))))))
    (if (typep seq 'vector)
        (nrv seq)
        (nrl seq))))
```

```
(defun -null (x) (eq x nil))
```

```
(defmacro -or (&optional first &rest rest)
  (if (null rest)
      first
      (let ((g (gensym)))
        `(let ((,g ,first))
           (if ,g
               ,g
               (-or ,@rest))))))
```

這兩個 Common Lisp 沒有，但這裡有幾的定義會需要用到。

```
(defun pair (lst)
  (if (null lst)
```

```
      nil
      (cons (cons (car lst) (cadr lst))
            (pair (cddr lst)))))

(defun -pairlis (keys vals &optional alist)
  (unless (= (length keys) (length vals))
    (error "mismatched lengths"))
  (nconc (mapcar #'cons keys vals) alist))
```

```
(defmacro -pop (place)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-expansion place)
    (let ((g (gensym)))
      `(let* (,@(mapcar #'list vars forms)
              (,g ,access)
              (,(car var) (cdr ,g)))
         (prog1 (car ,g)
                ,set))))))
```

```
(defmacro -prog1 (arg1 &rest args)
  (let ((g (gensym)))
    `(let ((,g ,arg1))
       ,@args
       ,g)))
```

```
(defmacro -prog2 (arg1 arg2 &rest args)
  (let ((g (gensym)))
    `(let ((,g (progn ,arg1 ,arg2)))
       ,@args
       ,g)))
```

```
(defmacro -progn (&rest args) `(let nil ,@args))
```

```
(defmacro -psetf (&rest args)
  (unless (evenp (length args))
    (error "odd number of arguments"))
  (let* ((pairs (pair args))
         (syms (mapcar #'(lambda (x) (gensym))
                       pairs)))
    `(let ,(mapcar #'list
                   syms
                   (mapcar #'cdr pairs))
       (setf ,@(mapcan #'list
                       (mapcar #'car pairs)
                       syms))))))
```

```
(defmacro -push (obj place)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-expansion place)
    (let ((g (gensym)))
      `(let* ((,g ,obj)
```

```
              ,@(mapcar #'list vars forms)
              (,(car var) (cons ,g ,access)))
         ,set))))
```

```
(defun -rem (n m)
  (nth-value 1 (truncate n m)))

(defmacro -rotatef (&rest args)
  `(psetf ,@(mapcan #'list
                    args
                    (append (cdr args)
                            (list (car args))))))
```

```
(defun -second (x) (cadr x))

(defmacro -setf (&rest args)
  (if (null args)
      nil
      `(setf2 ,@args)))
```

```
(defmacro setf2 (place val &rest args)
  (multiple-value-bind (vars forms var set)
                       (get-setf-expansion place)
    `(progn
       (let* (,@(mapcar #'list vars forms)
              (,(car var) ,val))
         ,set)
       ,@(if args `((setf2 ,@args)) nil))))
```

```
(defun -signum (n)
  (if (zerop n) 0 (/ n (abs n))))
```

```
(defun -stringp (x) (typep x 'string))
```

```
(defun -tailp (x y)
  (or (eql x y)
      (and (consp y) (-tailp x (cdr y)))))
```

```
(defun -third (x) (car (cdr (cdr x))))
```

```
(defun -truncate (n &optional (d 1))
  (if (> n 0) (floor n d) (ceiling n d)))
```

```
(defmacro -typecase (arg &rest clauses)
  (let ((g (gensym)))
    `(let ((,g ,arg))
       (cond ,@(mapcar #'(lambda (cl)
                           `((typep ,g ',(car cl))
                             (progn ,@(cdr cl))))
```

```
                        clauses)))))
```

```
(defmacro -unless (arg &rest body)
  `(if (not ,arg)
       (progn ,@body)))
```

```
(defmacro -when (arg &rest body)
  `(if ,arg (progn ,@body)))
```

```
(defun -1+ (x) (+ x 1))
```

```
(defun -1- (x) (- x 1))
```

```
(defun ->= (first &rest rest)
  (or (null rest)
      (and (or (> first (car rest)) (= first (car rest)))
           (apply #'->= rest))))
```

附錄 C：Common Lisp 的改變

目前的 ANSI Common Lisp 與 1984 年由 Guy Steele 一書 *Common Lisp: the Language* 所定義的 Common Lisp 有著本質上的不同。同時也與 1990 年該書的第二版大不相同，雖然差別比較小。本附錄總結了重大的改變。1990年之後的改變獨自列在最後一節。

附錄 D：語言參考手冊

備註

本節既是備註亦作爲參考文獻。所有列於此的書籍與論文皆值得閱讀。

譯註: 備註後面跟隨的數字即書中的頁碼

備註 viii (Notes viii)

Steele, Guy L., Jr. [http://en.wikipedia.org/wiki/Guy_L._Steele,_Jr.], Scott E. Fahlman [http://en.wikipedia.org/wiki/Scott_Fahlman], Richard P. Gabriel [http://en.wikipedia.org/wiki/Richard_P._Gabriel], David A. Moon [http://en.wikipedia.org/wiki/David_Moon], Daniel L. Weinreb [http://en.wikipedia.org/wiki/Daniel_Weinreb] , Daniel G. Bobrow [http://en.wikipedia.org/wiki/Daniel_G._Bobrow], Linda G. DeMichiel [http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/d/DeMichiel:Linda_G=.html], Sonya E. Keene [http://www.amazon.com/Sonya-E.-Keene/e/B001ITVL6O], Gregor Kiczales [http://en.wikipedia.org/wiki/Gregor_Kiczales], Crispin Perdue [http://perdues.com/CrisPerdueResume.html], Kent M. Pitman [http://en.wikipedia.org/wiki/Kent_Pitman], Richard C. Waters [http://www.rcwaters.org/], 以及 John L White。 Common Lisp: the Language, 2nd Edition. [http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html] Digital Press, Bedford (MA), 1990.

備註 1 (Notes 1)

McCarthy, John. [http://en.wikipedia.org/wiki/John_McCarthy_(computer_scientist)] Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I. [http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.4527&rep=rep1&type=pdf] CACM, 3:4 (April 1960), pp. 184-195.

McCarthy, John. [http://en.wikipedia.org/wiki/John_McCarthy_(computer_scientist)] History of Lisp. [http://www-formal.stanford.edu/jmc/history/lisp/lisp.html] In Wexelblat, Richard L. [http://en.wikipedia.org/wiki/Richard_Wexelblat] (Ed.) Histroy of Programming Languages. [http://cs305.com/book/programming_languages/Conf-01/HOPLII/frontmatter.pdf] Academic Press, New York, 1981, pp. 173-197.

備註 3 (Notes 3)

Brooks, Frederick P [http://en.wikipedia.org/wiki/Frederick_Brooks]. The Mythical Man-Month [http://www.amazon.com/Mythical-Man-Month-Software-Engineering-Anniversary/dp/0201835959]. Addison-Wesley, Reading (MA), 1975, p. 16.

Rapid prototyping is not just a way to write programs faster or better. It is a way to write programs that otherwise might not get written at all. Even the most ambitious people shrink from big undertakings. It's easier to start something if one can convince oneself (however speciously) that it won't be too much work. That's why so many big things have begun as small things. Rapid prototyping lets us start small.

備註 4 (Notes 4)

同上， 第 i 頁。

備註 5 (Notes 5)

Murray, Peter and Linda. The Art of the Renaissance [http://www.amazon.com/Art-Renaissance-World/dp/0500200084]. Thames and Hudson, London, 1963, p.85.

備註 5-2 (Notes 5-2)

Janson, W.J. History of Art [http://www.amazon.com/History-Art-H-W-Janson/dp/0810934019/ref=sr_1_1?s=books&ie=UTF8&qid=1365042074&sr=1-1&keywords=History+of+Art], 3rd Edition. Abrams, New York, 1986, p. 374.

The analogy applies, of course, only to paintings done on panels and later on canvases. Well-paintings continued to be done in fresco. Nor do I mean to suggest that painting styles were driven by technological change; the opposite seems more nearly true.

備註 12 (Notes 12)

`car` 與 `cdr` 的名字來自最早的 Lisp 實現裡，列表內部的表示法：car 代表“寄存器位址部分的內容”、cdr 代表“寄存器遞減部分的內容”。

備註 17 (Notes 17)

對遞迴概念有困擾的讀者，可以查閱下列的書籍：

Touretzky, David S. Common Lisp: A Gentle Introduction to Symbolic Computation

[http://www.amazon.com/Common-Lisp-Introduction-Computation-Benjamin-Cummings/dp/B008T1B8WQ/ref=sr_1_3?s=books&ie=UTF8&qid=1365042108&sr=1-3&keywords=A+Gentle+Introduction+to+Symbolic+Computation]. Benjamin/Cummings, Redwood City (CA), 1990, Chapter 8.

Friedman, Daniel P., and Matthias Felleisen. The Little Lisper. MIT Press, Cambridge, 1987.

譯註：這本書有再版，可在這裡 [http://www.amazon.com/Common-LISP-Introduction-Symbolic-Computation/dp/0486498204/ref=sr_1_1?s=books&ie=UTF8&qid=1365042108&sr=1-1&keywords=A+Gentle+Introduction+to+Symbolic+Computation]找到。

備註 26 (Notes 26)

In ANSI Common Lisp there is also a `lambda` macro that allows you to write `(lambda (x) x)` for `#'(lambda (x) x)` . Since the use of this macro obscures the symmetry between lambda expressions and symbolic function names (where you still have to use sharp-quote), it yields a specious sort of elegance at best.

備註 28 (Notes 28)

Gabriel, Richard P. Lisp Good News, Bad News, How to Win Big [http://www.dreamsongs.com/Files/LispGoodNewsBadNews.pdf] *AI Expert*, June 1991, p.34.

備註 46 (Notes 46)

Another thing to be aware of when using sort: it does not guarantee to preserve the order of elements judged equal by the comparison function. For example, if you sort `(2 1 1.0)` by `<` , a valid Common Lisp implementation could return either `(1 1.0 2)` or `(1.0 1 2)` . To preserve as much as possible of the original order, use instead the slower `stable-sort` (also destructive), which could only return the first value.

備註 61 (Notes 61)

A lot has been said about the benefits of comments, and little or nothing about their cost. But they do have a cost. Good code, like good prose, comes from constant rewriting. To evolve, code must be malleable and compact. Interlinear comments make programs stiff and diffuse, and so inhibit the evolution of what they describe.

備註 62 (Notes 62)

Though most implementations use the ASCII character set, the only ordering that Common Lisp guarantees for characters is as follows: the 26 lowercase letters are in alphabetically ascending order, as are the uppercase letters, and the digits from 0 to 9.

備註 76 (Notes 76)

The standard way to implement a priority queue is to use a structure called a heap. See: Sedgewick, Robert. Algorithms [http://www.amazon.com/Algorithms-4th-Robert-Sedgewick/dp/032157351X/ref=sr_1_1?s=books&ie=UTF8&qid=1365042619&sr=1-1&keywords=algorithms+sedgewick]. Addison-Wesley, Reading (MA), 1988.

備註 81 (Notes 81)

The definition of progn sounds a lot like the evaluation rule for Common Lisp function calls (page 9). Though `progn` is a special operator, we could define a similar function:

```
(defun our-progn (ftrest args)
  (car (last args)))
```

This would be horribly inefficient, but functionally equivalent to the real `progn` if the last argument returned exactly one value.

備註 84 (Notes 84)

The analogy to a lambda expression breaks down if the variable names are symbols that have special meanings in a parameter list. For example,

```
(let ((&key 1) (&optional 2)))
```

is correct, but the corresponding lambda expression

```
((lambda (ftkey ftoptional)) 1 2)
```

is not. The same problem arises if you try to define do in terms of `labels` . Thanks to David Kuznick for pointing this out.

備註 89 (Notes 89)

Steele, Guy L., Jr., and Richard P. Gabriel. The Evolution of Lisp [http://www.dreamsongs.com/Files/HOPL2-Uncut.pdf]. ACM SIGPLANNotices 28:3 (March 1993). The example in the quoted passage was translated from Scheme into Common Lisp.

備註 91 (Notes 91)

To make the time look the way people expect, you would want to ensure that minutes and seconds are represented with two digits, as in:

```
(defun get-time-string ()
  (multiple-value-bind (s m h) (get-decoded-time)
    (format nil "~A:~2,,,'0@A:~2,,,'0@A" h m s)))
```

備註 94 (Notes 94)

In a letter of March 18 (old style) 1751, Chesterfield writes:

“It was notorious, that the Julian Calendar was erroneous, and had overcharged the solar year with eleven days. Pope Gregory the Thirteenth corrected this error [in 1582]; his reformed calendar was immediately received by all the Catholic powers of Europe, and afterwards adopted by all the Protestant ones, except Russia, Sweden, and England. It was not, in my opinion, very honourable for England to remain in a gross and avowed error, especially in such company; the inconveniency of it was likewise felt by all those who had foreign correspondences, whether political or mercantile. I determined, therefore, to attempt the reformation; I consulted the best lawyers, and the most skillful astronomers, and we cooked up a bill for that purpose. But then my difficulty began; I was to bring in this bill, which was necessarily composed of law jargon and astronomical calculations, to both of which I am an utter stranger. However, it was absolutely necessary to make the House of Lords think that I knew something of the matter; and also to make them believe that they knew something of it themselves, which they do not. For my own part, I could just as soon have talked Celtic or Sclavonian to them, as astronomy, and they would have understood me full as well; so I resolved to do better than speak to the purpose, and to please instead of informing them. I gave them, therefore, only an historical account of calendars, from the Egyptian down to the Gregorian, amusing them now and then with little episodes; but I was particularly attentive to the choice of my words, to the harmony and roundness of my periods, to my elocution, to my action. This succeeded, and ever will succeed; they thought I informed them, because I pleased them; and many of them said I had made the whole very clear to them; when, God knows, I had not even attempted it.”

See: Roberts, David (Ed.) Lord Chesterfield’s Letters

[http://books.google.com.tw/books/about/Lord_Chesterfield_s_Letters.html?id=nFZP1WQ6XDoC&redir_esc=y]. Oxford University Press, Oxford, 1992.

備註 95 (Notes 95)

In Common Lisp, a universal time is an integer representing the number of seconds since the beginning of 1900. The functions `encode-universal-time` and `decode-universal-time` translate dates into and out of this format. So for dates after 1900, there is a simpler way to do date arithmetic in Common Lisp:

```
(defun num->date (n)
  (multiple-value-bind (ig no re d m y)
                       (decode-universal-time n)
    (values d m y)))

(defun date->num (d m y)
  (encode-universal-time 1 0 0 d m y))

(defun date+ (d m y n)
  (num->date (+ (date->num d m y)
                (* 60 60 24 n))))
```

Besides the range limit, this approach has the disadvantage that dates tend not to be fixnums.

備註 100 (Notes 100)

Although a call to `setf` can usually be understood as a reference to a particular place, the underlying machinery is more general. Suppose that a marble is a structure with a single field called color:

```
(defstruct marble
  color)
```

The following function takes a list of marbles and returns their color, if they all have the same color, or n i l if they have different colors:

```
(defun uniform-color (lst)
  (let ((c (marble-color (car lst))))
    (dolist (m (cdr lst))
      (unless (eql (marble-color m) c)
        (return nil)))
    c))
```

Although `uniform-color` does not refer to a particular place, it is both reasonable and possible

to have a call to it as the first argument to `setf` . Having defined

```
(defun (setf uniform-color) (val 1st)
  (dolist (m 1st)
    (setf (marble-color m) val)))
```

we can say

```
(setf (uniform-color *marbles*) 'red)
```

to make the color of each element of `*marbles*` be red.

備註 100-2 (Notes 100-2)

In older Common Lisp implementations, you have to use `defsetf` to define how a call should be treated when it appears as the first argument to setf. Be careful when translating, because the parameter representing the new value comes last in the definition of a function whose name is given as the second argument to `defsetf` . That is, the call

```
(defun (setf primo) (val 1st) (setf (car 1st) val))
```

is equivalent to

```
(defsetf primo set-primo)
```

```
(defun set-primo (1st val) (setf (car 1st) val))
```

備註 106 (Notes 106)

C, for example, lets you pass a pointer to a function, but there's less you can pass in a function (because C doesn't have closures) and less the recipient can do with it (because C has no equivalent of apply). What's more, you are in principle supposed to declare the type of the return value of the function you pass a pointer to. How, then, could you write `map-int` or `filter` , which work for functions that return anything? You couldn't, really. You would have to suppress the type-checking of arguments and return values, which is dangerous, and even so would probably only be practical for 32-bit values.

備註 109 (Notes 109)

For many examples of the versatility of closures, see: Abelson, Harold, and Gerald Jay

Sussman, with Julie Sussman. Structure and Interpretation of Computer Programs [http://mitpress.mit.edu/sicp/]. MIT Press, Cambridge, 1985.

備註 109-2 (Notes 109-2)

For more information about Dylan, see: Shalit, Andrew, with Kim Barrett, David Moon, Orca Starbuck, and Steve Strassmann. Dylan Interim Reference Manual [http://jim.studt.net/dirm/interim-contents.html]. Apple Computer, 1994.

At the time of printing this document was accessible from several sites, including http://www.harlequin.com and http://www.apple.com. Scheme is a very small, clean dialect of Lisp. It was invented by Guy L. Steele Jr. and Gerald J. Sussman in 1975, and is currently defined by: Clinger, William, and Jonathan A. Rees (Eds.) \(Revised^4\) Report on the Algorithmic Language Scheme. 1991.

This report, and various implementations of Scheme, were at the time of printing available by anonymous FTP from swiss-ftp.ai.mit.edu:pub.

There are two especially good textbooks that use Scheme—Structure and Interpretation (see preceding note) and: Springer, George and Daniel P. Friedman. Scheme and the Art of Programming [http://www.amazon.com/Scheme-Art-Programming-George-Springer/dp/0262192888]. MIT Press, Cambridge, 1989.

備註 112 (Notes 112)

The most horrible Lisp bugs may be those involving dynamic scope. Such errors almost never occur in Common Lisp, which has lexical scope by default. But since so many of the Lisps used as extension languages still have dynamic scope, practicing Lisp programmers should be aware of its perils.

One bug that can arise with dynamic scope is similar in spirit to variable capture (page 166). You pass one function as an argument to another. The function passed as an argument refers to some variable. But within the function that calls it, the variable has a new and unexpected value.

Suppose, for example, that we wrote a restricted version of mapcar as follows:

```
(defun our-mapcar (fn x)
  (if (null x)
      nil (cons (funcall fn (car x))
                (our-mapcar fn (cdr x)))))
```

Then suppose that we used this function in another function, `add-to-all` , that would take a number and add it to every element of a list:

```
(defun add-to-all (lst x)
  (our-mapcar #'(lambda (num) (+ num x))
              lst))
```

In Common Lisp this code works fine, but in a Lisp with dynamic scope it would generate an error. The function passed as an argument to `our-mapcar` refers to `x` . At the point where we send this function to `our-mapcar` , `x` would be the number given as the second argument to `add-to-all` . But where the function will be called, within `our-mapcar` , `x` would be something else: the list passed as the second argument to `our-mapcar` . We would get an error when this list was passed as the second argument to `+` .

備註 123 (Notes 123)

Newer implementations of Common Lisp include avariable `*read-eval*` that can be used to turn off the `#` . read-macro. When calling `read-from-string` on user input, it is wise to bind `*read-eval*` to `nil` . Otherwise the user could cause side-effects by using `#` . in the input.

備註 125 (Notes 125)

There are a number of ingenious algorithms for fast string-matching, but string-matching in text files is one of the cases where the brute-force approach is still reasonably fast. For more on string-matching algorithms, see: Sedgewick, Robert. Algorithms [http://www.amazon.com/Algorithms-4th-Robert-Sedgewick/dp/032157351X/ref=sr_1_1?s=books&ie=UTF8&qid=1365042619&sr=1-1&keywords=algorithms+sedgewick]. Addison-Wesley, Reading (MA), 1988.

備註 141 (Notes 141)

In 1984 CommonLisp, reduce did not take a `:key` argument, so `random-next` would be defined:

```
(defun random-next (prev)
  (let* ((choices (gethash prev *words*))
         (i (random (let ((x 0))
                      (dolist (c choices)
                        (incf x (cdr c)))
                      x))))
    (dolist (pair choices)
      (if (minusp (decf i (cdr pair)))
```

```
        (return (car pair))))))
```

備註 141-2 (Notes 141-2)

In 1989, a program like Henley was used to simulate netnews postings by well-known flamers. The fake postings fooled a significant number of readers. Like all good hoaxes, this one had an underlying point. What did it say about the content of the original flames, or the attention with which they were read, that randomly generated postings could be mistaken for the real thing?

One of the most valuable contributions of artificial intelligence research has been to teach us which tasks are really difficult. Some tasks turn out to be trivial, and some almost impossible. If artificial intelligence is concerned with the latter, the study of the former might be called artificial stupidity. A silly name, perhaps, but this field has real promise—it promises to yield programs that play a role like that of control experiments.

Speaking with the appearance of meaning is one of the tasks that turn out to be surprisingly easy. People's predisposition to find meaning is so strong that they tend to overshoot the mark. So if a speaker takes care to give his sentences a certain kind of superficial coherence, and his audience are sufficiently credulous, they will make sense of what he says.

This fact is probably as old as human history. But now we can give examples of genuinely random text for comparison. And if our randomly generated productions are difficult to distinguish from the real thing, might that not set people to thinking?

The program shown in Chapter 8 is about as simple as such a program could be, and that is already enough to generate "poetry" that many people (try it on your friends) will believe was written by a human being. With programs that work on the same principle as this one, but which model text as more than a simple stream of words, it will be possible to generate random text that has even more of the trappings of meaning.

For a discussion of randomly generated poetry as a legitimate literary form, see: Low, Jackson M. Poetry, Chance, Silence, Etc. In Hall, Donald (Ed.) Claims for Poetry. University of Michigan Press, Ann Arbor, 1982. You bet.

Thanks to the Online Book Initiative, ASCII versions of many classics are available online. At the time of printing, they could be obtained by anonymous FTP from ftp.std.com:obi.

See also the Emacs Dissociated Press feature, which uses an equivalent algorithm to scramble a buffer.

備註 150 (Notes 150)

下面這個函數會顯示在一個給定實現中，16　　個用來標示浮點表示法的限制的全局常數：

```
(defun float-limits ()
  (dolist (m '(most least))
    (dolist (s '(positive negative))
      (dolist (f '(short single double long))
        (let ((n (intern (string-upcase
                          (format nil "~A-~A-~A-float"
                                  m  s  f)))))
          (format t "~30A ~A ~%" n (symbol-value n)))))))
```

備註 164 (Notes 164)

快速排序演算法 [http://zh.wikipedia.org/zh-cn/%E5%BF%AB%E9%80%9F%E6%8E%92%E5%BA%8F]由霍爾 [http://zh.wikipedia.org/zh-cn/%E6%9D%B1%E5%B0%BC%C2%B7%E9%9C%8D%E7%88%BE]於 1962 年發表，並被描述在 Knuth, D. E. *Sorting and Searching*. Addison-Wesley, Reading (MA), 1973.一書中。

備註 173 (Notes 173)

Foderaro, John K. Introduction to the Special Lisp Section. CACM 34:9 (Setember 1991), p.27 [http://www.informatik.uni-trier.de/~ley/db/journals/cacm/cacm34.html]

備註 176 (Notes 176)

關於 CLOS 更詳細的資訊，參考下列書目：

Keene, Sonya E. Object Oriented Programming in Common Lisp [http://en.wikipedia.org/wiki/Object-Oriented_Programming_in_Common_Lisp:_A_Programmer's_Guide_to_CLOS] , Addison-Wesley, Reading (MA), 1989

Kiczales, Gregor, Jim des Rivieres, and Daniel G. Bobrow. The Art of the Metaobject Protocol [http://en.wikipedia.org/wiki/The_Art_of_the_Metaobject_Protocol] MIT Press, Cambridge, 1991

備註 178 (Notes 178)

讓我們再回放剛剛的句子一次：我們甚至不需要看程式中其他的程式碼一眼，就可以完成種種的改動。這個想法或許對某些讀者聽起來擔憂地熟悉。這是寫出麵條式程式碼[http://zh.wikipedia.org/wiki/%E9%9D%A2%E6%9D%A1%E5%BC%8F%E4%BB%A3%E7%A的食譜。

物件導向模型使得通過一點一點的來構造程式變得簡單。但這通常意味著，在實踐上它提供了一種有結構的方法來寫出麵條式程式碼。這不一定是壞事，但也不會是好事。

很多現實世界中的程式碼是麵條式程式碼，這也許不能很快改變。針對那些終將成爲麵條式程式碼的程式來說，物件導向模型是好的：它們最起碼會是有結構的麵條。但針對那些也許可以避免誤入崎途的程式來說，面向物件抽象只是更加危險的，而不是有用的。

備註 183 (Notes 183)

When an instance would inherit a slot with the same name from several of its superclasses, the instance inherits a single slot that combines the properties of the slots in the superclasses. The way combination is done varies from property to property:

1. The `:allocation` , `:initform` (if any), and `:documentation` (if any), will be those of the most specific classes.
2. The `:initargs` will be the union of the `:initargs` of all the superclasses. So will the `:accessors` , `:readers` , and `:writers` , effectively.
3. The `:type` will be the intersection of the `:types` of all the superclasses.

備註 191 (Notes 191)

You can avoid explicitly uninterning the names of slots that you want to be encapsulated by using uninterned symbols as the names to start with:

```
(progn
  (defclass counter () ((#1=#:state :initform 0)))

  (defmethod increment ((c counter))
    (incf (slot-value c '#1#)))

  (defmethod clear ((c counter))
    (setf (slot-value c '#1#) 0)))
```

The `progn` here is a no-op; it is used to ensure that all the references to the uninterned symbol occur within the same expression. If this were inconvenient, you could use the following read-macro instead:

```
(defvar *symtab* (make-hash-table :test #'equal))

(defun pseudo-intern (name)
  (or (gethash name *symtab*)
      (setf (gethash name *symtab*) (gensym))))

(set-dispatch-macro-character #\# #\[
  #'(lambda (stream char1 char2)
      (do ((acc nil (cons char acc))
           (char (read-char stream) (read-char stream)))
          ((eql char #\]) (pseudo-intern acc)))))
```

Then it would be possible to say just:

```
(defclass counter () ((#[state] :initform 0)))

(defmethod increment ((c counter))
  (incf (slot-value c '#[state])))

(defmethod clear ((c counter))
  (setf (slot-value c '#[state]) 0))
```

備註 204 (Notes 204)

下面這個宏將新元素推入二元搜索樹：

```
(defmacro bst-push (obj bst <)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-expansion bst)
    (let ((g (gensym)))
      `(let* ((,g ,obj)
              ,@(mapcar #'list vars forms)
              (,(car var) (bst-insert! ,g ,access ,<)))
         ,set))))
```

備註 213 (Notes 213)

Knuth, Donald E. Structured Programming with goto Statements. [http://sbel.wisc.edu/Courses/ME964/Literature/knuthProgramming1974.pdf] *Computing Surveys* , 6:4 (December 1974), pp. 261-301

備註 214 (Notes 214)

Knuth, Donald E. Computer Programming as an Art [http://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&cad=rja&ved=0CC4QFjAB&url=http%3A%2F%2F

ife4DB4BR2CPORBQ] *In ACM Turing Award Lectures: The First Twenty Years*. ACM Press, 1987

This paper and the preceding one are reprinted in: Knuth, Donald E. Literate Programming. CSLI Lecture Notes #27, Stanford University Center for the Study of Language and Information, Palo Alto, 1992.

備註 216 (Notes 216)

Steele, Guy L., Jr. Debunking the "Expensive Procedure Call" Myth or, Procedural Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO. Proceedings of the National Conference of the ACM, 1977, p. 157.

Tail-recursion optimization should mean that the compiler will generate the same code for a tail-recursive function as it would for the equivalent `do`. The unfortunate reality, at least at the time of printing, is that many compilers generate slightly faster code for `do`s.

備註 217 (Notes 217)

For some examples of calls to disassemble on various processors, see: Norvig, Peter. Paradigms ofArtificial Intelligence Programming: Case Studies in Common Lisp. Morgan Kaufmann, San Mateo (CA), 1992.

備註 218 (Notes 218)

A lot of the increased popularity of object-oriented programming is more specifically the increased popularity of C++, and this in turn has a lot to do with typing. C++ gives you something that seems like a miracle in the conceptual world of C: the ability to define operators that work for different types of arguments. But you don't need an object-oriented language to do this—all you need is run-time typing. And indeed, if you look at the way people use C++, the class hierarchies tend to be flat. C++ has become so popular not because people need to write programs in terms of classes and methods, but because people need a way around the restrictions imposed by C's approach to typing.

備註 219 (Notes 219)

Macros can make declarations easier. The following macro expects a type name and an expression (probably numeric), and expands the expression so that all arguments, and all intermediate results, are declared to be of that type. If you wanted to ensure that an expression e

was evaluated using only fixnum arithmetic, you could say `(with-type fixnum e)`.

```
(defmacro with-type (type expr)
  `(the ,type ,(if (atom expr)
                   expr
                 (expand-call type (binarize expr)))))

(defun expand-call (type expr)
  `(,(car expr) ,@(mapcar #'(lambda (a)
                              `(with-type ,type ,a))
                          (cdr expr))))

(defun binarize (expr)
  (if (and (nthcdr 3 expr)
           (member (car expr) '(+ - * /)))
      (destructuring-bind (op a1 a2 . rest) expr
        (binarize `(,op (,op ,a1 ,a2) ,@rest)))
    expr))
```

The call to binarize ensures that no arithmetic operator is called with more than two arguments. As the Lucid reference manual points out, a call like

```
(the fixnum (+ (the fixnum a)
               (the fixnum b)
               (the fixnum c)))
```

still cannot be compiled into fixnum additions, because the intermediate results (e.g. a + b) might not be fixnums.

Using `with-type`, we could duplicate the fully declared version of `poly` on page 219 with:

```
(defun poly (a b x)
  (with-type fixnum (+ (* a (expt x 2)) (* b x))))
```

If you wanted to do a lot of fixnum arithmetic, you might even want to define a read-macro that would expand into a `(with-type fixnum ...)`.

備註 224 (Notes 224)

在許多 Unix 系統裡， `/usr/dict/words` 是個合適的單詞檔案。

備註 226 (Notes 229)

T is a dialect of Scheme with many useful additions, including support for pools. For more on T,

see: Rees, Jonathan A., Norman I. Adams, and James R. Meehan. The T Manual, 5th Edition. Yale University Computer Science Department, New Haven, 1988.

The T manual, and T itself, were at the time of printing available by anonymous FTP from hing.lcs.mit.edu:pub/t3.1 .

備註 229 (Notes 229)

The difference between specifications and programs is a difference in degree, not a difference in kind. Once we realize this, it seems strange to require that one write specifications for a program before beginning to implement it. If the program has to be written in a low-level language, then it would be reasonable to require that it be described in high-level terms first. But as the programming language becomes more abstract, the need for specifications begins to evaporate. Or rather, the implementation and the specifications can become the same thing.

If the high-level program is going to be re-implemented in a lower-level language, it starts to look even more like specifications. What Section 13.7 is saying, in other words, is that the specifications for C programs could be written in Lisp.

備註 230 (Notes 230)

Benvenuto Cellini's story of the casting of his Perseus is probably the most famous (and the funniest) account of traditional bronze-casting: Cellini, Benvenuto. Autobiography. Translated by George Bull, Penguin Books, Harmondsworth, 1956.

備註 239 (Notes 239)

Even experienced Lisp hackers find packages confusing. Is it because packages are gross, or because we are not used to thinking about what happens at read-time?

There is a similar kind of uncertainty about def macro, and there it does seem that the difficulty is in the mind of the beholder. A good deal of work has gone into finding a more abstract alternative to def macro. But def macro is only gross if you approach it with the preconception (common enough) that defining a macro is like defining a function. Then it seems shocking that you suddenly have to worry about variable capture. When you think of macros as what they are, transformations on source code, then dealing with variable capture is no more of a problem than dealing with division by zero at run-time.

So perhaps packages will turn out to be a reasonable way of providing modularity. It is prima facie evidence on their side that they resemble the techniques that programmers naturally use in

the absence of a formal module system.

備註 242 (Notes 242)

It might be argued that `loop` is more general, and that we should not define many operators to do what we can do with one. But it's only in a very legalistic sense that loop is one operator. In that sense, `eval` is one operator too. Judged by the conceptual burden it places on the user, `loop` is at least as many operators as it has clauses. What's more, these operators are not available separately, like real Lisp operators: you can't break off a piece of loop and pass it as an argument to another function, as you could `map-int` .

備註 248 (Notes 248)

關於更深入講述邏輯推論的資料，參見：Stuart Russell [http://www.cs.berkeley.edu/~russell/] 及 Peter Norvig [http://www.norvig.com/] 所著的 Artificial Intelligence: A Modern Approach [http://aima.cs.berkeley.edu/]。

備註 273 (Notes 273)

Because the program in Chapter 17 takes advantage of the possibility of having a `setf` form as the first argument to `defun` , it will only work in more recent Common Lisp implementations. If you want to use it in an older implementation, substitute the following code in the final version:

```
(proclaim '(inline lookup set-lookup))

(defsetf lookup set-lookup)

(defun set-lookup (prop obj val)
  (let ((off (position prop (layout obj) :test #'eq)))
    (if off
        (setf (svref obj (+ off 3)) val)
        (error "Can't set ~A of ~A." val obj))))

(defmacro defprop (name &optioanl meth?)
  `(progn
     (defun ,name (obj &rest args)
       ,(if meth?
          `(run-methods obj ',name args)
          `(rget ',name obj nil)))
     (defsetf ,name (obj) (val)
       `(setf (lookip ',',name ,obj) ,val))))
```

備註 276 (Notes 276)

If `defmeth` were defined as

```
(defmacro defmeth (name obj parms &rest body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (setf (gethash ',name ,gobj)
             #'(lambda ,parms
                 (labels ((next ()
                            (funcall (get-next ,gobj ',name)
                                     ,@parms)))
                   ,@body))))))
```

then it would be possible to invoke the next method simply by calling `next` :

```
(defmeth area grumpy-circle (c)
  (format t "How dare you stereotype me!""/,")
  (next))
```

備註 284 (Notes 284)

For really fast access to slots we would use the following macro:

```
(defmacro with-slotref ((name prop class) &rest body)
  (let ((g (gensym)))
    `(let ((,g (+ 3 (position ,prop (layout ,class)
                              :test #'eq))))
       (macrolet ((,name (obj) `(svref ,obj ,',g)))
         ,@body))))
```

It defines a local macro that refers directly to the vector element corresponding to a slot. If in some segment of code you wanted to refer to the same slot in many instances of the same class, with this macro the slot references would be straight `svref`s.

For example, if the balloon class is defined as follows,

```
(setf balloon-class (class nil size))
```

then this function pops (in the old sense) a list of ballons:

```
(defun popem (ballons)
  (with-slotref (bsize 'size balloon-class)
    (dolist (b ballons)
      (setf (bsize b) 0))))
```

備註 284-2 (Notes 284-2)

Gabriel, Richard P. Lisp Good News, Bad News, How to Win Big [http://www.dreamsongs.com/Files/LispGoodNewsBadNews.pdf] *AI Expert*, June 1991, p.35.

早在 1973 年， Richard Fateman [http://en.wikipedia.org/wiki/Richard_Fateman] 已經能證明在 PDP-10 [http://en.wikipedia.org/wiki/PDP-10] 主機上， MacLisp [http://en.wikipedia.org/wiki/Maclisp] 編譯器比製造商的 FORTRAN 編譯器，產生出更快速的程式碼。

譯註: 該篇 MacLisp 編譯器在 PDP-10 可產生比 Fortran 快的程式碼的論文在這可以找到 [http://dl.acm.org/citation.cfm?doid=1086803.1086804]

備註 399 (Notes 399)

It's easiest to understand backquote if we suppose that backquote and comma are like quote, and that `` `,x `` simply expands into `(bq (comma x))`. If this were so, we could handle backquote by augmenting `eval` as in this sketch:

```
(defun eval2 (expr)
  (case (and (consp expr) (car expr))
    (comma (error "unmatched comma"))
    (bq (eval-bq (second expr) 1))
    (t  (eval expr))))

(defun eval-bq (expr n)
  (cond ((atom expr)
         expr)
        ((eql (car expr) 'comma)
         (if (= n 1)
             (eval2 (second expr))
             (list 'comma (eval-bq (second expr)
                                   (1- n)))))
        ((eql (car expr) 'bq)
         (list 'bq (eval-bq (second expr) (1+ n))))
        (t
         (cons (eval-bq (car expr) n)
               (eval-bq (cdr expr) n)))))
```

In `eval-bq` , the parameter `n` is used to determine which commas match the current backquote. Each backquote increments it, and each comma decrements it. A comma encountered when n = 1 is a matching comma. Here is the example from page 400:

```
> (setf x 'a a 1 y 'b b 2)
2
```

```
> (eval2 '(bq (bq (w (comma x) (comma (comma y))))))
(BQ (W (COMMA X) (COMMA B)))
> (eval2 *)
(W A 2)
```

At some point a particularly remarkable molecule was formed by accident. We will call it the Replicator. It may not necessarily have been the biggest or the most complex molecule around, but it had the extraordinary property of being able to create copies of itself.

Richard Dawkins

The Selfish Gene

We shall first define a class of symbolic expressions in terms of ordered pairs and lists. Then we shall define five elementary functions and predicates, and build from them by composition, conditional expressions, and recursive definitions an extensive class of functions of which we shall give a number of examples. We shall then show how these functions themselves can be expressed as symbolic expressions, and we shall define a universal function apply that allows us to compute from the expression for a given function its value for given arguments.

John McCarthy

Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I

简体中文

前言

本书的目的是快速及全面的教你 Common Lisp 的有关知识。它实际上包含两本书。前半部分用大量的例子来解释 Common Lisp 里面重要的概念。后半部分是一个最新 Common Lisp 辞典，涵盖了所有 ANSI Common Lisp 的操作符。

这本书面向的读者

ANSI Common Lisp 这本书适合学生或者是专业的程序员去读。本书假设读者阅读前没有 Lisp 的相关知识。有别的程序语言的编程经验也许对读本书有帮助，但也不是必须的。本书从解释 Lisp 中最基本的概念开始，并对于 Lisp 最容易迷惑初学者的地方进行特别的强调。

本书也可以作为教授 Lisp 编程的课本，也可以作为人工智能课程和其他编程语言课程中，有关 Lisp 部分的参考书。想要学习 Lisp 的专业程序员肯定会很喜欢本书所采用的直截了当、注重实践的方法。那些已经在使用 Lisp 编程的人士将会在本书中发现许多有用的实例，此外，本书也是一本方便的 ANSI Common Lisp 参考书。

如何使用这本书

学习 Lisp 最好的办法就是拿它来编程。况且在学习的同时用你学到的技术进行编程，也是非常有趣的一件事。编写本书的目的就是让读者尽快的入门，在对 Lisp 进行简短的介绍之后， 第 2 章开始用 21 页的内容，介绍了着手编写 Lisp 程序时可能会用到的所有知识。 3-9 章讲解了 Lisp 里面一些重要的知识点。这些章节特别强调了一些重要的概念，比如指针在 Lisp 中扮演的角色，如何使用递归来解决问题，以及第一级函数的重要性等等。

针对那些想要更深入了解 Lisp 的读者： 10-14 章包含了宏、CLOS、列表操作、程序优化，以及一些更高级的课题，比如包和读取宏。

15-17 章通过 3 个 Common Lisp 的实际应用，总结了之前章节所讲解的知识：一个是进行逻辑推理的程序，另一个是 HTML 生成器，最后一个是针对面向对象编程的嵌入式语言。

本书的最后一部分包含了 4 个附录，这些附录应该对所有的读者都有用： 附录 A-D 包括了一个如何调试程序的指南， 58 个 Common Lisp 操作符的源程序，一个关于 ANSI Common Lisp 和以前的 Lisp 语言区别的总结，以及一个包括所有 ANSI Common Lisp 的参考手册。

本书还包括一节备注。这些备注包括一些说明，一些参考条目，一些额外的代码，以及一些对偶然出现的不正确表述的纠正。备注在文中用一个小圆圈来表示，像这样：○

Note: 译注: 由于小圈圈 ○ 实在太不明显了，译文中使用 λ 符号来表示备注。

λ [http://ansi-common-lisp.readthedocs.org/en/latest/zhCN/notes-cn.html#viii-notes-viii]

代码

虽然本书介绍的是 ANSI Common Lisp ，但是本书中的代码可以在任何版本的 Common Lisp 中运行。那些依赖 Lisp 语言新特性的例子的旁边，会有注释告诉你如何把它们运行于旧版本的 Lisp 中。

本书中所有的代码都可以在互联网上下载到。你可以在网络上找到这些代码，它们还附带着一个免费软件的链接，一些过去的论文，以及 Lisp 的 FAQ 。还有很多有关 Lisp 的资源可以在此找到： http://www.eecs.harvard.edu/onlisp/ 源代码可以在此 FTP 服务器上下载： ftp://ftp.eecs.harvard.edu:/pub/onlisp/ 读者的问题和意见可以发送到 pg@eecs.harvard.edu 。

Tip: 译注：下载的链接都坏掉了，本书的代码可以到此下载：https://raw.github.com/acl-translation/acl-chinese/master/code/acl2.lisp

On Lisp

在整本 On Lisp 书中，我一直试着指出一些 Lisp 独一无二的特性，这些特性使得 Lisp 更像 “Lisp” 。并展示一些 Lisp 能让你完成的新事情。比如说宏： Lisp 程序员能够并且经常编写一些能够写程序的程序。对于程序生成程序这种特性，因为 Lisp 是主流语言中唯一一个提供了相关抽象使得你能够方便地实现这种特性的编程语言，所以 Lisp 是主流语言中唯一一个广泛运用这个特性的语言。我非常乐意邀请那些想要更进一步了解宏和其他高级 Lisp 技术的读者，读一下本书的姐妹篇： On Lisp [http://www.paulgraham.com/onlisp.html] 。

Tip: On Lisp 已经由知名 Lisp 黑客 —— 田春 —— 翻译完成，可以在网络上找到。 —— 田春（知名 Lisp 黑客、Practical Common Lisp 译者）

鸣谢

在所有帮助我完成这本的朋友当中，我想特别的感谢一下 Robert Morris 。他的重要影

响反应在整本书中。他的良好影响使这本书更加优秀。本书中好一些实例程序都源自他手。这些程序包括 138 页的 Henley 和 249 页的模式匹配器。

我很高兴能有一个高水平的技术审稿小组：Skona Brittain, John Foderaro, Nick Levine, Peter Norvig 和 Dave Touretzky。本书中几乎所有部分都得益于它们的意见。 John Foderaro 甚至重写了本书 5.7 节中一些代码。

另外一些人通篇阅读了本书的手稿，它们是：Ken Anderson, Tom Cheatham, Richard Fateman, Steve Hain, Barry Margolin, Waldo Pacheco, Wheeler Ruml 和 Stuart Russell。特别要提一下，Ken Anderson 和 Wheeler Ruml 给予了很多有用的意见。

我非常感谢 Cheatham 教授，更广泛的说，哈佛，提供我编写这本书的一些必要条件。另外也要感谢 Aiken 实验室的人员：Tony Hartman, Dave Mazieres, Janusz Juda, Harry Bochner 和 Joanne Klys。

我非常高兴能再一次有机会和 Alan Apt 合作。还有这些在 Prentice Hall 工作的人士：Alan, Mona, Pompili Shirley McGuire 和 Shirley Michaels, 能与你们共事我很高兴。

本书用 Leslie Lamport 写的 LaTeX 进行排版。LaTeX 是在 Donald Knuth 编写的 TeX 的基础上，又加了 L.A.Carr, Van Jacobson 和 Guy Steele 所编写的宏完成。书中的图表是由 John Vlissides 和 Scott Stanton 编写的 Idraw 完成的。整本书的预览是由 Tim Theisen 写的 Ghostview 完成的。 Ghostview 是根据 L. Peter Deutsch 的 Ghostscript 创建的。

我还需要感谢其他的许多人，包括：Henry Baker, Kim Barrett, Ingrid Bassett, Trevor Blackwell, Paul Becker, Gary Bisbee, Frank Deutschmann, Frances Dickey, Rich 和 Scott Draves, Bill Dubuque, Dan Friedman, Jenny Graham, Alice Hartley, David Hendler, Mike Hewett, Glenn Holloway, Brad Karp, Sonya Keene, Ross Knights, Mutsumi Komuro, Steffi Kutzia, David Kuznick, Madi Lord, Julie Mallozzi, Paul McNamee, Dave Moon, Howard Mullings, Mark Nitzberg, Nancy Parmet 和其家人, Robert Penny, Mike Plusch, Cheryl Sacks, Hazem Sayed, Shannon Spires, Lou Steinberg, Paul Stoddard, John Stone, Guy Steele, Steve Strassmann, Jim Veitch, Dave Watkins, Idelle and Julian Weber, the Weickers, Dave Yost 和 Alan Yuille。

另外，着重感谢我的父母和 Jackie。

高德纳 [http://zh.wikipedia.org/zh-cn/%E9%AB%98%E5%BE%B7%E7%BA%B3]给他的经典丛书起名为《计算机程序设计艺术》。在他的图灵奖获奖感言中，他解释说这本书的书名源自于内心深处的潜意识 —— 潜意识告诉他，编程其实就是追求编写最优美的程序。

就像建筑设计一样，编程既是一门工程技艺也是一门艺术。一个程序要遵循数学原理也要符合物理定律。但是建筑师的目的不仅仅是建一个不会倒塌的建筑。更重要的是，他

们要建一个优美的建筑。

像高德纳一样，很多程序员认为编程的真正目的，不仅仅是编写出正确的程序，更重要的是写出优美的代码。几乎所有的 Lisp 黑客也是这么想的。 Lisp 黑客精神可以用两句话来概括：编程应该是有趣的。程序应该是优美的。这就是我在这本书中想要传达的精神。

保罗•格雷厄姆 (Paul Graham) [http://paulgraham.com/]

第一章：简介

约翰麦卡锡 [http://zh.wikipedia.org/zh-cn/%E7%BA%A6%E7%BF%B0%C2%B7%E9%BA%A6%E5%8D%A1%E9%94%A1]和他的学生于 1958 年展开 Lisp 的初次实现工作。 Lisp 是继 FORTRAN 之后，仍在使用的最古老的程序语言。 λ [http://acl.readthedocs.org/en/latest/zhCN/notes-cn.html#notes-1] 更值得注意的是，它仍走在程序语言技术的最前面。懂 Lisp 的程序员会告诉你，有某种东西使 Lisp 与众不同。

Lisp 与众不同的部分原因是，它被设计成能够自己进化。你能用 Lisp 定义新的 Lisp 操作符。当新的抽象概念风行时（如面向对象程序设计），我们总是发现这些新概念在 Lisp 是最容易来实现的。Lisp 就像生物的 DNA 一样，这样的语言永远不会过时。

1.1 新的工具 (New Tools)

为什么要学 Lisp？因为它让你能做一些其它语言做不到的事情。如果你只想写一个函数来返回小于 `n` 的数字总和，那么用 Lisp 和 C 是差不多的：

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

如果你只想做这种简单的事情，那用什么语言都不重要。假设你想写一个函数，输入一个数 `n` ，返回把 `n` 与传入参数 (argument)相加的函数。

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

在 C 语言中 `addn` 怎么实现？你根本写不出来。

你可能会想，谁会想做这样的事情？程序语言教你不要做它们没有提供的事情。你得针对每个程序语言，用其特定的思维来写程序，而且想得到你所不能描述的东西是很困难的。当我刚开始编程时 —— 用 Baisc —— 我不知道什么是递归，因为我根本不知道有这个东西。我是用 Basic 在思考。我只能用迭代的概念表达算法，所以我怎么会知道递归呢？

如果你没听过词法闭包 「Lexical Closure」 [http://zh.wikipedia.org/zh-cn/%E9%97%AD%E5%8C%85_(%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E (上述 addn 的范例)，相信我， Lisp 程序员一直在使用它。很难找到任何长度的 Common Lisp 程序，没有用到闭包的好处。在 112 页前，你自己会持续使用它。

闭包仅是其中一个我们在别的语言找不到的抽象概念之一。另一个更有价值的 Lisp 特点是， Lisp 程序是用 Lisp 的数据结构来表示。这表示你可以写出会写程序的程序。人们真的需要这个吗？没错 —— 它们叫做宏，有经验的程序员也一直在使用它。学到 173 页你就可以自己写出自己的宏了。

有了宏、闭包以及运行期类型，Lisp 凌驾在面向对象程序设计之上。如果你了解上面那句话，也许你不应该阅读此书。你得充分了解 Lisp 才能明白为什么此言不虚。但这不是空泛之言。这是一个重要的论点，并且在 17 章用程序相当明确的证明了这点。

第二章到第十三章会循序渐进地介绍所有你需要理解第 17 章程序的概念。你的努力会有所回报：你会感到在 C++ 编程是窒碍难行的，就像有经验的 C++ 程序员用 Basic 编程会感到窒息一样。更加鼓舞人心的是，如果我们思考为什么会有这种感觉。 编写 Basic 对于平常用 C++ 编程是令人感到窒息的，是因为有经验的 C++ 程序员知道一些用 Basic 不可能表达出来的技术。同样地，学习 Lisp 不仅教你学会一门新的语言 —— 它教你崭新的并且更强大的程序思考方法。

1.2 新的技术 (New Techniques)

如上一节所提到的， Lisp 赋予你别的语言所没有的工具。不仅仅如此，就 Lisp 带来的新特性来说 —— 自动内存管理 (automatic memory management)，显式类型 (manifest typing)，闭包 (closures)等 —— 每一项都使得编程变得如此简单。结合起来，它们组成了一个关键的部分，使得一种新的编程方式是有可能的。

Lisp 被设计成可扩展的：让你定义自己的操作符。这是可能的，因为 Lisp 是由和你程序一样的函数与宏所构成的。所以扩展 Lisp 就和写一个 Lisp 程序一样简单。事实上，它是如此的容易（和有用），以至于扩展语言自身成了标准实践。当你在用 Lisp 语言编程时，你也在创造一个适合你的程序的语言。你由下而上地，也由上而下地工作。

几乎所有的程序，都可以从订作适合自己所需的语言中受益。然而越复杂的程序，由下而上的程序设计就显得越有价值。一个由下而上所设计出来的程序，可写成一系列的层，每层担任上一层的程序语言。 TeX [http://en.wikipedia.org/wiki/TeX] 是最早使用这种方法所写的程序之一。你可以用任何语言由下而上地设计程序，但 Lisp 是本质上最适合这种方法的工具。

由下而上的编程方法，自然发展出可扩展的软件。如果你把由下而上的程序设计的原则，想成你程序的最上层，那这层就成为使用者的程序语言。正因可扩展的思想深植于

Lisp 当中，使得 Lisp 成为实现可扩展软件的理想语言。三个 1980 年代最成功的程序提供 Lisp 作为扩展自身的语言: GNU Emacs [http://www.gnu.org/software/emacs/]， Autocad [http://www.autodesk.com.tw/adsk/servlet/pc/index?siteID=1170616&id=14977606] ，和 Interleaf [http://en.wikipedia.org/wiki/Interleaf] 。

由下而上的编程方法，也是得到可重用软件的最好方法。写可重用软件的本质是把共同的地方从细节中分离出来，而由下而上的编程方法本质地创造这种分离。与其努力撰写一个庞大的应用，不如努力创造一个语言，用相对小的努力在这语言上撰写你的应用。和应用相关的特性集中在最上层，以下的层可以组成一个适合这种应用的语言 —— 还有什么比程序语言更具可重用性的呢？

Lisp 让你不仅编写出更复杂的程序，而且写的更快。 Lisp 程序通常很简短 —— Lisp 给了你更高的抽象化，所以你不用写太多代码。就像 Frederick Brooks [http://en.wikipedia.org/wiki/Fred_Brooks] 所指出的，编程所花的时间主要取决于程序的长度。因此仅仅根据这个单独的事实，就可以推断出用 Lisp 编程所花的时间较少。这种效果被 Lisp 的动态特点放大了：在 Lisp 中，编辑-编译-测试循环短到使编程像是即时的。

更高的抽象化与互动的环境，能改变各个机构开发软件的方式。术语快速建型描述了一种始于 Lisp 的编程方法：在 Lisp 里，你可以用比写规格说明更短的时间，写一个原型出来，而这种原型是高度抽象化的，可作为一个比用英语所写的更好的规格说明。而且 Lisp 让你可以轻易的从原型转成产品软件。当写一个考虑到速度的 Common Lisp 程序时，通过现代编译器的编译，Lisp 与其他的高阶语言所写的程序运行得一样快。

除非你相当熟悉 Lisp ，这个简介像是无意义的言论和冠冕堂皇的声明。*Lisp* 凌驾面向对象程序设计？ 你创造适合你程序的语言？ *Lisp* 编程是即时的？ 这些说法是什么意思？现在这些说法就像是枯竭的湖泊。随着你学到更多实际的 Lisp 特色，见过更多可运行的程序，这些说法就会被实际经验之水所充满，而有了明确的形状。

1.3 新的方法 (New Approach)

本书的目标之一是不仅是教授 Lisp 语言，而是教授一种新的编程方法，这种方法因为有了 Lisp 而有可能实现。这是一种你在未来会见得更多的方法。随着开发环境变得更强大，程序语言变得更抽象， Lisp 的编程风格正逐渐取代旧的规划-然后-实现 (*plan-and-implement*)的模式。

在旧的模式中，错误永远不应该出现。事前辛苦订出缜密的规格说明，确保程序完美的运行。理论上听起来不错。不幸地，规格说明是人写的，也是人来实现的。实际上结果是， 规划-然后-实现 模型不太有效。

身为 OS/360 的项目经理， Frederick Brooks [http://en.wikipedia.org/wiki/Fred_Brooks] 非

常熟悉这种传统的模式。他也非常熟悉它的后果：

> 任何 OS/360 的用户很快的意识到它应该做得更好...再者，产品推迟，用了更多的内存，成本是估计的好几倍，效能一直不好，直到第一版后的好几个版本更新，效能才算还可以。

而这却描述了那个时代最成功系统之一。

旧模式的问题是它忽略了人的局限性。在旧模式中，你打赌规格说明不会有严重的缺失，实现它们不过是把规格转成代码的简单事情。经验显示这实在是非常坏的赌注。打赌规格说明是误导的，程序到处都是臭虫 (bug) 会更保险一点。

这其实就是新的编程模式所假设的。设法尽量降低错误的成本，而不是希望人们不犯错。错误的成本是修补它所花费的时间。使用强大的语言跟好的开发环境，这种成本会大幅地降低。编程风格可以更多地依靠探索，较少地依靠事前规划。

规划是一种必要之恶。它是评估风险的指标：越是危险，预先规划就显得更重要。强大的工具降低了风险，也降低了规划的需求。程序的设计可以从最有用的信息来源中受益：过去实作程序的经验。

Lisp 风格从 1960 年代一直朝着这个方向演进。你在 Lisp 中可以如此快速地写出原型，以致于你已历经好几个设计和实现的循环，而在旧的模式当中，你可能才刚写完规格说明。你不必担心设计的缺失，因为你将更快地发现它们。你也不用担心有那么多臭虫。当你用函数式风格来编程，你的臭虫只有局部的影响。当你使用一种很抽象的语言，某些臭虫(如迷途指针 [http://zh.wikipedia.org/zh-cn/%E8%BF%B7%E9%80%94%E6%8C%87%E9%92%88])不再可能发生，而剩下的臭虫很容易找出，因为你的程序更短了。当你有一个互动的开发环境，你可以即时修补臭虫，不必经历 编辑，编译，测试的漫长过程。

Lisp 风格会这么演进是因为它产生的结果。听起来很奇怪，少的规划意味著更好的设计。技术史上相似的例子不胜枚举。一个相似的变革发生在十五世纪的绘画圈里。在油画流行前，画家使用一种叫做蛋彩 [http://zh.wikipedia.org/zh-cn/%E8%9B%8B%E5%BD%A9%E7%95%AB]的材料来作画。蛋彩不能被混和或涂掉。犯错的代价非常高，也使得画家变得保守。后来随着油画颜料的出现，作画风格有了大幅地改变。油画“允许你再来一次”这对困难主题的处理，像是画人体，提供了决定性的有利条件。

新的材料不仅使画家更容易作画了。它使新的更大胆的作画方式成为可能。 Janson 写道：

> 如果没有油画颜料，弗拉芒大师们的征服可见的现实的口号就会大打折扣。于是，从技术的角度来说，也是如此，但他们当之无愧地称得上是“现代绘画之父”，油画

颜料从此以后成为画家的基本颜料。

做为一种介质，蛋彩与油画颜料一样美丽。但油画颜料的弹性给想像力更大的发挥空间 —— 这是决定性的因素。

程序设计正经历着相同的改变。新的介质像是“动态的面向对象语言” —— 即 Lisp 。这不是说我们所有的软件在几年内都要用 Lisp 来写。从蛋彩到油画的转变也不是一夜完成的；油彩一开始只在领先的艺术中心流行，而且经常混合着蛋彩来使用。我们现在似乎正处于这个阶段。 Lisp 被大学，研究室和某些顶尖的公司所使用。同时，从 Lisp 借鉴的思想越来越多地出现在主流语言中：交互式编程环境 (interactive programming environment)、垃圾回收(garbage collection) [http://zh.wikipedia.org/zh-cn/%E5%9E%83%E5%9C%BE%E5%9B%9E%E6%94%B6_(%E8%A8%88%E7%AE%97%E6 运行期类型 (run-time typing)，仅举其中几个。

强大的工具正降低探索的风险。这对程序员来说是好消息，因为意味者我们可以从事更有野心的项目。油画的确有这个效果。采用油画后的时期正是绘画的黄金时期。类似的迹象正在程序设计的领域中发生。

第二章：欢迎来到 Lisp

本章的目的是让你尽快开始编程。本章结束时，你会掌握足够多的 Common Lisp 知识来开始写程序。

2.1 形式 (Form)

人可以通过实践来学习一件事，这对于 Lisp 来说特别有效，因为 Lisp 是一门交互式的语言。任何 Lisp 系统都含有一个交互式的前端，叫做顶层(toplevel)。你在顶层输入 Lisp 表达式，而系统会显示它们的值。

Lisp 通常会打印一个提示符告诉你，它正在等待你的输入。许多 Common Lisp 的实现用 `>` 作为顶层提示符。本书也沿用这个符号。

一个最简单的 Lisp 表达式是整数。如果我们在提示符后面输入 `1`，

```
> 1
1
>
```

系统会打印出它的值，接着打印出另一个提示符，告诉你它在等待更多的输入。

在这个情况里，打印的值与输入的值相同。数字 `1` 称之为对自身求值。当我们输入需要做某些计算来求值的表达式时，生活变得更加有趣了。举例来说，如果我们想把两个数相加，我们输入像是：

```
> (+ 2 3)
5
```

在表达式 `(+ 2 3)` 里，`+` 称为操作符，而数字 `2` 跟 `3` 称为实参。

在日常生活中，我们会把表达式写作 `2 + 3`，但在 Lisp 里，我们把 `+` 操作符写在前面，接著写实参，再把整个表达式用一对括号包起来：`(+ 2 3)`。这称为前序表达式。一开始可能觉得这样写表达式有点怪，但事实上这种表示法是 Lisp 最美妙的东西之一。

举例来说，我们想把三个数加起来，用日常生活的表示法，要写两次 `+` 号，

```
2 + 3 + 4
```

而在 Lisp 里，只需要增加一个实参：

```
(+ 2 3 4)
```

日常生活中用 + 时，它必须有两个实参，一个在左，一个在右。前序表示法的灵活性代表著，在 Lisp 里， + 可以接受任意数量的实参，包含了没有实参：

```
> (+)
0
> (+ 2)
2
> (+ 2 3)
5
> (+ 2 3 4)
9
> (+ 2 3 4 5)
14
```

由于操作符可接受不定数量的实参，我们需要用括号来标明表达式的开始与结束。

表达式可以嵌套。即表达式里的实参，可以是另一个复杂的表达式：

```
> (/ (- 7 1) (- 4 2))
3
```

上面的表达式用中文来说是， (七减一) 除以 (四减二) 。

Lisp 表示法另一个美丽的地方是：它就是如此简单。所有的 Lisp 表达式，要么是 1 这样的数原子，要么是包在括号里，由零个或多个表达式所构成的列表。以下是合法的 Lisp 表达式：

```
2 (+ 2 3) (+ 2 3 4) (/ (- 7 1) (- 4 2))
```

稍后我们将理解到，所有的 Lisp 程序都采用这种形式。而像是 C 这种语言，有着更复杂的语法：算术表达式采用中序表示法；函数调用采用某种前序表示法，实参用逗号隔开；表达式用分号隔开；而一段程序用大括号隔开。

在 Lisp 里，我们用单一的表示法，来表达所有的概念。

2.2 求值 (Evaluation)

上一小节中，我们在顶层输入表达式，然后 Lisp 显示它们的值。在这节里我们深入理解一下表达式是如何被求值的。

在 Lisp 里， `+` 是函数，然而如 `(+ 2 3)` 的表达式，是函数调用。

当 Lisp 对函数调用求值时，它做下列两个步骤：

1. 首先从左至右对实参求值。在这个例子当中，实参对自身求值，所以实参的值分别是 `2` 跟 `3` 。
2. 实参的值传入以操作符命名的函数。在这个例子当中，将 `2` 跟 `3` 传给 `+` 函数，返回 `5` 。

如果实参本身是函数调用的话，上述规则同样适用。以下是当 `(/ (- 7 1) (- 4 2))` 表达式被求值时的情形：

1. Lisp 对 `(- 7 1)` 求值: `7` 求值为 `7` ， `1` 求值为 `1` ，它们被传给函数 `-` ，返回 `6` 。
2. Lisp 对 `(- 4 2)` 求值: `4` 求值为 `4` ， `2` 求值为 `2` ，它们被传给函数 `-` ，返回 `2` 。
3. 数值 `6` 与 `2` 被传入函数 `/` ，返回 `3` 。

但不是所有的 Common Lisp 操作符都是函数，不过大部分是。函数调用都是这么求值。由左至右对实参求值，将它们的数值传入函数，来返回整个表达式的值。这称为 Common Lisp 的求值规则。

Note: 逃离麻烦

如果你试着输入 Lisp 不能理解的东西，它会打印一个错误讯息，接著带你到一种叫做中断循环（break loop）的顶层。 中断循环给予有经验的程序员一个机会，来找出错误的原因，不过最初你只会想知道如何从中断循环中跳出。 如何返回顶层取决于你所使用的 Common Lisp 实现。在这个假定的实现环境中，输入 `:abort` 跳出：

```
> (/ 1 0)
Error: Division by zero
      Options: :abort, :backtrace
>> :abort
>
```

附录 A 演示了如何调试 Lisp 程序，并给出一些常见的错误例子。

一个不遵守 Common Lisp 求值规则的操作符是 `quote` 。 `quote` 是一个特殊的操作符，意味着它自己有一套特别的求值规则。这个规则就是：什么也不做。 `quote` 操作符接受一个实参，并完封不动地返回它。

```
> (quote (+ 3 5))
```

```
(+ 3 5)
```

为了方便起见，Common Lisp 定义 ' 作为 quote 的缩写。你可以在任何的表达式前，贴上一个 '，与调用 quote 是同样的效果：

```
> '(+ 3 5)
(+ 3 5)
```

使用缩写 ' 比使用整个 quote 表达式更常见。

Lisp 提供 quote 作为一种保护表达式不被求值的方式。下一节将解释为什么这种保护很有用。

2.3 数据 (Data)

Lisp 提供了所有在其他语言找的到的，以及其他语言所找不到的数据类型。一个我们已经使用过的类型是整数（integer），整数用一系列的数字来表示，比如： 256 。另一个 Common Lisp 与多数语言有关，并很常见的数据类型是字符串（string），字符串用一系列被双引号包住的字符串表示，比如： "ora et labora" [3] 。整数与字符串一样，都是对自身求值的。

[3] “ora et labora” 是拉丁文，意思是祷告与工作。

有两个通常在别的语言所找不到的 Lisp 数据类型是符号（symbol）与列表（lists），符号是英语的单词 (words)。无论你怎么输入，通常会被转换为大写：

```
> 'Artichoke
ARTICHOKE
```

符号（通常）不对自身求值，所以要是想引用符号，应该像上例那样用 ' 引用它。

列表是由被括号包住的零个或多个元素来表示。元素可以是任何类型，包含列表本身。使用列表必须要引用，不然 Lisp 会以为这是个函数调用：

```
> '(my 3 "Sons")
(MY 3 "Sons")
> '(the list (a b c) has 3 elements)
(THE LIST (A B C) HAS 3 ELEMENTS)
```

注意引号保护了整个表达式（包含内部的子表达式）被求值。

你可以调用 list 来创建列表。由于 list 是函数，所以它的实参会被求值。这里我们看一个在函数 list 调用里面，调用 + 函数的例子：

```
> (list 'my (+ 2 1) "Sons")
(MY 3 "Sons")
```

我们现在来到领悟 Lisp 最卓越特性的地方之一。*Lisp*的程序是用列表来表示的。如果实参的优雅与弹性不能说服你 Lisp 表示法是无价的工具，这里应该能使你信服。这代表着 Lisp 程序可以写出 Lisp 代码。 Lisp 程序员可以（并且经常）写出能为自己写程序的程序。

不过得到第 10 章，我们才来考虑这种程序，但现在了解到列表和表达式的关系是非常重要的，而不是被它们搞混。这也就是为什么我们需要 `quote` 。如果一个列表被引用了，则求值规则对列表自身来求值；如果没有被引用，则列表被视为是代码，依求值规则对列表求值后，返回它的值。

```
> (list '(+ 2 1) (+ 2 1))
((+ 2 1) 3)
```

这里第一个实参被引用了，所以产生一个列表。第二个实参没有被引用，视为函数调用，经求值后得到一个数字。

在 Common Lisp 里有两种方法来表示空列表。你可以用一对不包括任何东西的括号来表示，或用符号 `nil` 来表示空表。你用哪种表示法来表示空表都没关系，但它们都会被显示为 `nil` ：

```
> ()
NIL
> nil
NIL
```

你不需要引用 `nil` （但引用也无妨），因为 `nil` 是对自身求值的。

2.4 列表操作 (List Operations)

用函数 `cons` 来构造列表。如果传入的第二个实参是列表，则返回由两个实参所构成的新列表，新列表为第一个实参加上第二个实参：

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

可以通过把新元素建立在空表之上，来构造一个新列表。上一节所看到的函数 `list` ，不过就是一个把几个元素加到 `nil` 上的快捷方式：

```
> (cons 'a (cons 'b nil))
(A B)
```

```
> (list 'a 'b)
(A B)
```

取出列表元素的基本函数是 car 和 cdr 。对列表取 car 返回第一个元素，而对列表取 cdr 返回第一个元素之后的所有元素：

```
> (car '(a b c))
A
> (cdr '(a b c))
(B C)
```

你可以把 car 与 cdr 混合使用来取得列表中的任何元素。如果我们想要取得第三个元素，我们可以：

```
> (car (cdr (cdr '(a b c d))))
C
```

不过，你可以用更简单的 third 来做到同样的事情：

```
> (third '(a b c d))
C
```

2.5 真与假 (Truth)

在 Common Lisp 里，符号 t 是表示逻辑 真 的缺省值。与 nil 相同， t 也是对自身求值的。如果实参是一个列表，则函数 listp 返回 真 ：

```
> (listp '(a b c))
T
```

函数的返回值将会被解释成逻辑 真 或逻辑 假 时，则称此函数为谓词（*predicate*）。在 Common Lisp 里，谓词的名字通常以 p 结尾。

逻辑 假 在 Common Lisp 里，用 nil ，即空表来表示。如果我们传给 listp 的实参不是列表，则返回 nil 。

```
> (listp 27)
NIL
```

由于 nil 在 Common Lisp 里扮演两个角色，如果实参是一个空表，则函数 null 返回 真 。

```
> (null nil)
```

```
T
```

而如果实参是逻辑 假 ，则函数 not 返回 真 ：

```
> (not nil)
T
```

null 与 not 做的是一样的事情。

在 Common Lisp 里，最简单的条件式是 if 。通常接受三个实参：一个 *test* 表达式，一个 *then* 表达式和一个 *else* 表达式。若 test 表达式求值为逻辑 真 ，则对 then 表达式求值，并返回这个值。若 test 表达式求值为逻辑 假 ，则对 else 表达式求值，并返回这个值：

```
> (if (listp '(a b c))
      (+ 1 2)
      (+ 5 6))
3
> (if (listp 27)
      (+ 1 2)
      (+ 5 6))
11
```

与 quote 相同， if 是特殊的操作符。不能用函数来实现，因为实参在函数调用时永远会被求值，而 if 的特点是，只有最后两个实参的其中一个会被求值。 if 的最后一个实参是选择性的。如果忽略它的话，缺省值是 nil ：

```
> (if (listp 27)
    (+ 1 2))
NIL
```

虽然 t 是逻辑 真 的缺省表示法，任何非 nil 的东西，在逻辑的上下文里通通被视为 真 。

```
> (if 27 1 2)
1
```

逻辑操作符 and 和 or 与条件式类似。两者都接受任意数量的实参，但仅对能影响返回值的几个实参求值。如果所有的实参都为 真 （即非 nil ），那么 and 会返回最后一个实参的值：

```
> (and t (+ 1 2))
3
```

如果其中一个实参为 假 ，那之后的所有实参都不会被求值。 or 也是如此，只要碰到一个为 真 的实参，就停止对之后所有的实参求值。

以上这两个操作符称为宏。宏和特殊的操作符一样，可以绕过一般的求值规则。第十章解释了如何编写你自己的宏。

2.6 函数 (Functions)

你可以用 defun 来定义新函数。通常接受三个以上的实参：一个名字，一组用列表表示的实参，以及一个或多个组成函数体的表达式。我们可能会这样定义 third：

```
> (defun our-third (x)
    (car (cdr (cdr x))))
OUR-THIRD
```

第一个实参说明此函数的名称将是 our-third 。第二个实参，一个列表 (x) ，说明这个函数会接受一个形参： x 。这样使用的占位符符号叫做变量。当变量代表了传入函数的实参时，如这里的 x ，又被叫做形参。

定义的剩余部分， (car (cdr (cdr x))) ，即所谓的函数主体。它告诉 Lisp 该怎么计算此函数的返回值。所以调用一个 our-third 函数，对于我们作为实参传入的任何 x ，会返回 (car (cdr (cdr x)))：

```
> (our-third '(a b c d))
C
```

既然我们已经讨论过了变量，理解符号是什么就更简单了。符号是变量的名字，符号本身就是以对象的方式存在。这也是为什么符号，必须像列表一样被引用。列表必须被引用，不然会被视为代码。符号必须要被引用，不然会被当作变量。

你可以把函数定义想成广义版的 Lisp 表达式。下面的表达式测试 1 和 4 的和是否大于 3：

```
> (> (+ 1 4) 3)
T
```

通过将这些数字替换为变量，我们可以写个函数，测试任两数之和是否大于第三个数：

```
> (defun sum-greater (x y z)
    (> (+ x y) z))
SUM-GREATER
> (sum-greater 1 4 3)
T
```

Lisp 不对程序、过程以及函数作区别。函数做了所有的事情（事实上，函数是语言的主要部分）。如果你想要把你的函数之一作为主函数（*main* function），可以这么做，但平常你就能在顶层中调用任何函数。这表示当你编程时，你可以把程序拆分成一小块一小块地来做调试。

2.7 递归 (Recursion)

上一节我们所定义的函数，调用了别的函数来帮它们做事。比如 `sum-greater` 调用了 `+` 和 `>` 。函数可以调用任何函数，包括自己。自己调用自己的函数是递归的。 Common Lisp 函数 `member` ，测试某个东西是否为列表的成员。下面是定义成递归函数的简化版：

```
> (defun our-member (obj lst)
   (if (null lst)
       nil
   (if (eql (car lst) obj)
       lst
       (our-member obj (cdr lst)))))
OUR-MEMBER
```

谓词 `eql` 测试它的两个实参是否相等；此外，这个定义的所有东西我们之前都学过了。下面是运行的情形：

```
> (our-member 'b '(a b c))
(B C)
> (our-member 'z '(a b c))
NIL
```

下面是 `our-member` 的定义对应到英语的描述。为了知道一个对象 `obj` 是否为列表 `lst` 的成员，我们

1. 首先检查 `lst` 列表是否为空列表。如果是空列表，那 `obj` 一定不是它的成员，结束。
2. 否则，若 `obj` 是列表的第一个元素时，则它是列表的成员。
3. 不然只有当 `obj` 是列表其余部分的元素时，它才是列表的成员。

当你想要了解递归函数是怎么工作时，把它翻成这样的叙述有助于你理解。

起初，许多人觉得递归函数很难理解。大部分的理解难处，来自于对函数使用了错误的比喻。人们倾向于把函数理解为某种机器。原物料像实参一样抵达；某些工作委派给其它函数；最后组装起来的成品，被作为返回值运送出去。如果我们用这种比喻来理解函数，那递归就自相矛盾了。机器怎可以把工作委派给自己？它已经在忙碌中了。

较好的比喻是，把函数想成一个处理的过程。在过程里，递归是在自然不过的事情了。日常生活中我们经常看到递归的过程。举例来说，假设一个历史学家，对欧洲历史上的人口变化感兴趣。研究文献的过程很可能是：

1. 取得一个文献的复本
2. 寻找关于人口变化的资讯
3. 如果这份文献提到其它可能有用的文献，研究它们。

过程是很容易理解的，而且它是递归的，因为第三个步骤可能带出一个或多个同样的过程。

所以，别把 `our-member` 想成是一种测试某个东西是否为列表成员的机器。而是把它想成是，决定某个东西是否为列表成员的规则。如果我们从这个角度来考虑函数，那么递归的矛盾就不复存在了。

2.8 阅读 Lisp (Reading Lisp)

上一节我们所定义的 `our-member` 以五个括号结尾。更复杂的函数定义更可能以七、八个括号结尾。刚学 Lisp 的人看到这么多括号会感到气馁。这叫人怎么读这样的程序，更不用说编了？怎么知道哪个括号该跟哪个匹配？

答案是，你不需要这么做。 Lisp 程序员用缩排来阅读及编写程序，而不是括号。当他们在写程序时，他们让文字编辑器显示哪个括号该与哪个匹配。任何好的文字编辑器，特别是 Lisp 系统自带的，都应该能做到括号匹配（paren-matching）。在这种编辑器中，当你输入一个括号时，编辑器指出与其匹配的那一个。如果你的编辑器不能匹配括号，别用了，想想如何让它做到，因为没有这个功能，你根本不可能编 Lisp 程序 [1] 。

有了好的编辑器之后，括号匹配不再会是问题。而且由于 Lisp 缩排有通用的惯例，阅读程序也不是个问题。因为所有人都使用一样的习惯，你可以忽略那些括号，通过缩排来阅读程序。

任何有经验的 Lisp 黑客，会发现如果是这样的 `our-member` 的定义很难阅读：

```
(defun our-member (obj lst) (if (null lst) nil (if
(eql (car lst) obj) lst (our-member obj (cdr lst)))))
```

但如果程序适当地缩排时，他就没有问题了。可以忽略大部分的括号而仍能读懂它：

```
defun our-member (obj lst)
 if null lst
    nil
    if eql (car lst) obj
```

```
        lst
        our-member obj (cdr lst)
```

事实上，这是你在纸上写 Lisp 程序的实用方法。等输入程序至计算机的时候，可以利用编辑器匹配括号的功能。

2.9 输入输出 (Input and Output)

到目前为止，我们已经利用顶层偷偷使用了 I/O 。对实际的交互程序来说，这似乎还是不太够。在这一节，我们来看几个输入输出的函数。

最普遍的 Common Lisp 输出函数是 `format` 。接受两个或两个以上的实参，第一个实参决定输出要打印到哪里，第二个实参是字符串模版，而剩余的实参，通常是要插入到字符串模版，用打印表示法（printed representation）所表示的对象。下面是一个典型的例子：

```
> (format t "~A plus ~A equals ~A. ~%" 2 3 (+ 2 3))
2 plus 3 equals 5.
NIL
```

注意到有两个东西被打印出来。第一行是 `format` 印出来的。第二行是调用 `format` 函数的返回值，就像平常顶层会打印出来的一样。通常像 `format` 这种函数不会直接在顶层调用，而是在程序内部里使用，所以返回值不会被看到。

`format` 的第一个实参 `t` ，表示输出被送到缺省的地方去。通常是顶层。第二个实参是一个用作输出模版的字符串。在这字符串里，每一个 `~A` 表示了被填入的位置，而 `~%` 表示一个换行。这些被填入的位置依序由后面的实参填入。

标准的输入函数是 `read` 。当没有实参时，会读取缺省的位置，通常是顶层。下面这个函数，提示使用者输入，并返回任何输入的东西：

```
(defun askem (string)
 (format t "~A" string)
 (read))
```

它的行为如下：

```
> (askem "How old are you?")
How old are you?29

29
```

记住 `read` 会一直永远等在这里，直到你输入了某些东西，并且（通常要）按下回车。

因此，不打印明确的提示信息是很不明智的，程序会给人已经死机的印象，但其实它是在等待输入。

第二件关于 `read` 所需要知道的事是，它很强大： `read` 是一个完整的 Lisp 解析器（parser）。不仅是可以读入字符，然后当作字符串返回它们。它解析它所读入的东西，并返回产生出来的 Lisp 对象。在上述的例子，它返回一个数字。

`askem` 的定义虽然很短，但体现出一些我们在之前的函数没看过的东西。函数主体可以有不只一个表达式。函数主体可以有任意数量的表达式。当函数被调用时，会依序求值，函数会返回最后一个的值。

在之前的每一节中，我们坚持所谓“纯粹的” Lisp —— 即没有副作用的 Lisp 。副作用是指，表达式被求值后，对外部世界的状态做了某些改变。当我们对一个如 `(+ 1 2)` 这样纯粹的 Lisp 表达式求值时，没有产生副作用。它只返回一个值。但当我们调用 `format` 时，它不仅返回值，还印出了某些东西。这就是一种副作用。

当我们想要写没有副作用的程序时，则定义多个表达式的函数主体就没有意义了。最后一个表达式的值，会被当成函数的返回值，而之前表达式的值都被舍弃了。如果这些表达式没有副作用，你没有任何理由告诉 Lisp ，为什么要去对它们求值。

2.10 变量 (Variables)

`let` 是一个最常用的 Common Lisp 的操作符之一，它让你引入新的局部变量（local variable）：

```
> (let ((x 1) (y 2))
      (+ x y))
3
```

一个 `let` 表达式有两个部分。第一个部分是一组创建新变量的指令，指令的形式为 *(variable expression)* 。每一个变量会被赋予相对应表达式的值。上述的例子中，我们创造了两个变量， `x` 和 `y` ，分别被赋予初始值 `1` 和 `2` 。这些变量只在 `let` 的函数体内有效。

一组变量与数值之后，是一个有表达式的函数体，表达式依序被求值。但这个例子里，只有一个表达式，调用 `+` 函数。最后一个表达式的求值结果作为 `let` 的返回值。以下是一个用 `let` 所写的，更有选择性的 `askem` 函数：

```
(defun ask-number ()
 (format t "Please enter a number. ")
 (let ((val (read)))
   (if (numberp val)
```

```
        val
        (ask-number))))
```

这个函数创建了变量 `val` 来储存 `read` 所返回的对象。因为它知道该如何处理这个对象，函数可以先观察你的输入，再决定是否返回它。你可能猜到了， `numberp` 是一个谓词，测试它的实参是否为数字。

如果使用者不是输入一个数字， `ask-number` 会持续调用自己。最后得到一个只接受数字的函数：

```
> (ask-number)
Please enter a number. a
Please enter a number. (ho hum)
Please enter a number. 52
52
```

我们已经看过的这些变量都叫做局部变量。它们只在特定的上下文里有效。另外还有一种变量叫做全局变量（global variable），是在任何地方都是可视的。 [2]

你可以给 `defparameter` 传入符号和值，来创建一个全局变量：

```
> (defparameter *glob* 99)
*GLOB*
```

全局变量在任何地方都可以存取，除了在定义了相同名字的区域变量的表达式里。为了避免这种情形发生，通常我们在给全局变量命名时，以星号作开始与结束。刚才我们创造的变量可以念作 “星-glob-星” (star-glob-star)。

你也可以用 `defconstant` 来定义一个全局的常量：

```
(defconstant limit (+ *glob* 1))
```

我们不需要给常量一个独一无二的名字，因为如果有相同名字存在，就会有错误产生(error)。如果你想要检查某些符号，是否为一个全局变量或常量，使用 `boundp` 函数：

```
> (boundp '*glob*)
T
```

2.11 赋值 (Assignment)

在 Common Lisp 里，最普遍的赋值操作符（assignment operator）是 `setf` 。可以用来给全局或局部变量赋值：

```
> (setf *glob* 98)
98
> (let ((n 10))
    (setf n 2)
    n)
2
```

如果 setf 的第一个实参是符号（symbol），且符号不是某个局部变量的名字，则 setf 把这个符号设为全局变量：

```
> (setf x (list 'a 'b 'c))
(A B C)
```

也就是说，通过赋值，你可以隐式地创建全局变量。 不过，一般来说，还是使用 defparameter 明确地创建全局变量比较好。

你不仅可以给变量赋值。传入 setf 的第一个实参，还可以是表达式或变量名。在这种情况下，第二个实参的值被插入至第一个实参所引用的位置：

```
> (setf (car x) 'n)
N
> x
(N B C)
```

setf 的第一个实参几乎可以是任何引用到特定位置的表达式。所有这样的操作符在附录 D 中被标注为 “可设置的”（“settable”）。你可以给 setf 传入（偶数）个实参。一个这样的表达式

```
(setf a 'b
      c 'd
      e 'f)
```

等同于依序调用三个单独的 setf 函数：

```
(setf a 'b)
(setf c 'd)
(setf e 'f)
```

2.12 函数式编程 (Functional Programming)

函数式编程意味着撰写利用返回值而工作的程序，而不是修改东西。它是 Lisp 的主流范式。大部分 Lisp 的内置函数被调用是为了取得返回值，而不是副作用。

举例来说，函数 remove 接受一个对象和一个列表，返回不含这个对象的新列表：

```
> (setf lst '(c a r a t))
(C A R A T)
> (remove 'a lst)
(C R T)
```

为什么不干脆说 `remove` 从列表里移除一个对象？因为它不是这么做的。原来的表没有被改变：

```
> lst
(C A R A T)
```

若你真的想从列表里移除某些东西怎么办？在 Lisp 通常你这么做，把这个列表当作实参，传入某个函数，并使用 `setf` 来处理返回值。要移除所有在列表 `x` 的 `a`，我们可以说：

```
(setf x (remove 'a x))
```

函数式编程本质上意味着避免使用如 `setf` 的函数。起初可能觉得这根本不可能，更遑论去做了。怎么可以只凭返回值来建立程序？

完全不用到副作用是很不方便的。然而，随着你进一步阅读，会惊讶地发现需要用到副作用的地方很少。副作用用得越少，你就更上一层楼。

函数式编程最重要的优点之一是，它允许交互式测试（interactive testing）。在纯函数式的程序里，你可以测试每个你写的函数。如果它返回你预期的值，你可以有信心它是对的。这额外的信心，集结起来，会产生巨大的差别。当你改动了程序里的任何一个地方，会得到即时的改变。而这种即时的改变，使我们有一种新的编程风格。类比于电话与信件，让我们有一种新的通讯方式。

2.13 迭代 (Iteration)

当我们想重复做一些事情时，迭代比递归来得更自然。典型的例子是用迭代来产生某种表格。这个函数

```
(defun show-squares (start end)
  (do ((i start (+ i 1)))
      ((> i end) 'done)
    (format t "~A ~A~%" i (* i i))))
```

列印从 `start` 到 `end` 之间的整数的平方：

```
> (show-squares 2 5)
2 4
```

```
3 9
4 16
5 25
DONE
```

do 宏是 Common Lisp 里最基本的迭代操作符。和 let 类似， do 可以创建变量，而第一个实参是一组变量的规格说明列表。每个元素可以是以下的形式

```
(variable initial update)
```

其中 *variable* 是一个符号， *initial* 和 *update* 是表达式。最初每个变量会被赋予 *initial* 表达式的值；每一次迭代时，会被赋予 *update* 表达式的值。在 show-squares 函数里， do 只创建了一个变量 i 。第一次迭代时， i 被赋与 start 的值，在接下来的迭代里， i 的值每次增加 1 。

第二个传给 do 的实参可包含一个或多个表达式。第一个表达式用来测试迭代是否结束。在上面的例子中，测试表达式是 (> i end) 。接下来在列表中的表达式会依序被求值，直到迭代结束。而最后一个值会被当作 do 的返回值来返回。所以 show-squares 总是返回 done 。

do 的剩余参数组成了循环的函数体。在每次迭代时，函数体会依序被求值。在每次迭代过程里，变量被更新，检查终止测试条件，接著（若测试失败）求值函数体。

作为对比，以下是递归版本的 show-squares ：

```
(defun show-squares (i end)
   (if (> i end)
     'done
     (progn
       (format t "~A ~A~%" i (* i i))
       (show-squares (+ i 1) end))))
```

唯一的新东西是 progn 。 progn 接受任意数量的表达式，依序求值，并返回最后一个表达式的值。

为了处理某些特殊情况， Common Lisp 有更简单的迭代操作符。举例来说，要遍历列表的元素，你可能会使用 dolist 。以下函数返回列表的长度：

```
(defun our-length (lst)
  (let ((len 0))
    (dolist (obj lst)
      (setf len (+ len 1)))
    len))
```

这里 dolist 接受这样形式的实参*(variable expression)*，跟着一个具有表达式的函数主体。函数主体会被求值，而变量相继与表达式所返回的列表元素绑定。因此上面的循环说，对于列表 lst 里的每一个 obj ，递增 len 。很显然这个函数的递归版本是：

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

也就是说，如果列表是空表，则长度为 0 ；否则长度就是对列表取 cdr 的长度加一。递归版本的 our-length 比较易懂，但由于它不是尾递归（tail-recursive）的形式 (见 13.2 节)，效率不是那么高。

2.14 函数作为对象 (Functions as Objects)

函数在 Lisp 里，和符号、字符串或列表一样，是稀松平常的对象。如果我们把函数的名字传给 function ，它会返回相关联的对象。和 quote 类似， function 是一个特殊操作符，所以我们无需引用（quote）它的实参：

```
> (function +)
#<Compiled-Function + 17BA4E>
```

这看起来很奇怪的返回值，是在典型的 Common Lisp 实现里，函数可能的打印表示法。

到目前为止，我们仅讨论过，不管是 Lisp 打印它们，还是我们输入它们，看起来都是一样的对象。但这个惯例对函数不适用。一个像是 + 的内置函数 ，在内部可能是一段机器语言代码（machine language code）。每个 Common Lisp 实现，可以选择任何它喜欢的外部表示法（external representation）。

如同我们可以用 ' 作为 quote 的缩写，也可以用 #' 作为 function 的缩写：

```
> #'+
#<Compiled-Function + 17BA4E>
```

这个缩写称之为升引号（sharp-quote）。

和别种对象类似，可以把函数当作实参传入。有个接受函数作为实参的函数是 apply 。apply 接受一个函数和实参列表，并返回把传入函数应用在实参列表的结果：

```
> (apply #'+ '(1 2 3))
6
> (+ 1 2 3)
6
```

apply 可以接受任意数量的实参，只要最后一个实参是列表即可：

```
> (apply #'+ 1 2 '(3 4 5))
15
```

函数 funcall 做的是一样的事情，但不需要把实参包装成列表。

```
> (funcall #'+ 1 2 3)
6
```

Note: 什么是 lambda ？

lambda 表达式里的 lambda 不是一个操作符。而只是个符号。 在早期的 Lisp 方言里，lambda 存在的原因是：由于函数在内部是用列表来表示， 因此辨别列表与函数的方法，就是检查第一个元素是否为 lambda 。

在 Common Lisp 里，你可以用列表来表达函数， 函数在内部会被表示成独特的函数对象。因此不再需要 *lambda* 了。 如果需要把函数记为

```
((x) (+ x 100))
```

而不是

```
(lambda (x) (+ x 100))
```

也是可以的。

但 Lisp 程序员习惯用符号 lambda ，来撰写函数， 因此 Common Lisp 为了传统，而保留了 lambda 。

defun 宏，创建一个函数并给函数命名。但函数不需要有名字，而且我们不需要 defun 来定义他们。和大多数的 Lisp 对象一样，我们可以直接引用函数。

要直接引用整数，我们使用一系列的数字；要直接引用一个函数，我们使用所谓的 *lambda* 表达式。一个 lambda 表达式是一个列表，列表包含符号 lambda ，接著是形参列表，以及由零个或多个表达式所组成的函数体。

下面的 lambda 表达式，表示一个接受两个数字并返回两者之和的函数：

```
(lambda (x y)
 (+ x y))
```

列表 `(x y)` 是形参列表，跟在它后面的是函数主体。

一个 `lambda` 表达式可以作为函数名。和普通的函数名称一样， lambda 表达式也可以是函数调用的第一个元素，

```
> ((lambda (x) (+ x 100)) 1)
101
```

而通过在 `lambda` 表达式前面贴上 `#'` ，我们得到对应的函数，

```
> (funcall #'(lambda (x) (+ x 100))
           1)
```

`lambda` 表示法除上述用途以外，还允许我们使用匿名函数。

2.15 类型 (Types)

Lisp 处理类型的方法非常灵活。在很多语言里，变量是有类型的，得声明变量的类型才能使用它。在 Common Lisp 里，数值才有类型，而变量没有。你可以想像每个对象，都贴有一个标明其类型的标签。这种方法叫做显式类型（*manifest typing*）。你不需要声明变量的类型，因为变量可以存放任何类型的对象。

虽然从来不需要声明类型，但出于效率的考量，你可能会想要声明变量的类型。类型声明在第 13.3 节时讨论。

Common Lisp 的内置类型，组成了一个类别的层级。对象总是不止属于一个类型。举例来说，数字 27 的类型，依普遍性的增加排序，依序是 `fixnum` 、 `integer` 、 `rational` 、 `real` 、 `number` 、 `atom` 和 `t` 类型。（数值类型将在第 9 章讨论。）类型 `t` 是所有类型的基类（supertype）。所以每个对象都属于 `t` 类型。

函数 `typep` 接受一个对象和一个类型，然后判定对象是否为该类型，是的话就返回真：

```
> (typep 27 'integer)
T
```

我们会在遇到各式内置类型时来讨论它们。

2.16 展望 (Looking Forward)

本章仅谈到 Lisp 的表面。然而，一种非比寻常的语言形象开始出现了。首先，这个语言用单一的语法，来表达所有的程序结构。语法基于列表，列表是一种 Lisp 对象。函

数本身也是 Lisp 对象，函数能用列表来表示。而 Lisp 本身就是 Lisp 程序。几乎所有你定义的函数，与内置的 Lisp 函数没有任何区别。

如果你对这些概念还不太了解，不用担心。 Lisp 介绍了这么多新颖的概念，在你能驾驭它们之前，得花时间去熟悉它们。不过至少要了解一件事：在这些概念当中，有着优雅到令人吃惊的概念。

Richard Gabriel [http://en.wikipedia.org/wiki/Richard_P._Gabriel] 曾经半开玩笑的说， C 是拿来写 Unix 的语言。我们也可以说， Lisp 是拿来写 Lisp 的语言。但这是两种不同的论述。一个可以用自己编写的语言和一种适合编写某些特定类型应用的语言，是有着本质上的不同。这开创了新的编程方法：你不但在语言之中编程，还把语言改善成适合程序的语言。如果你想了解 Lisp 编程的本质，理解这个概念是个好的开始。

Chapter 2 总结 (Summary)

1. Lisp 是一种交互式语言。如果你在顶层输入一个表达式， Lisp 会显示它的值。
2. Lisp 程序由表达式组成。表达式可以是原子，或一个由操作符跟着零个或多个实参的列表。前序表示法代表操作符可以有任意数量的实参。
3. Common Lisp 函数调用的求值规则： 依序对实参从左至右求值，接著把它们的值传入由操作符表示的函数。 `quote` 操作符有自己的求值规则，它完封不动地返回实参。
4. 除了一般的数据类型， Lisp 还有符号跟列表。由于 Lisp 程序是用列表来表示的，很轻松就能写出能编程的程序。
5. 三个基本的列表函数是 `cons` ，它创建一个列表； `car` ，它返回列表的第一个元素；以及 `cdr` ，它返回第一个元素之后的所有东西。
6. 在 Common Lisp 里， `t` 表示逻辑 `真` ，而 `nil` 表示逻辑 `假` 。在逻辑的上下文里，任何非 `nil` 的东西都视为 `真` 。基本的条件式是 `if` 。 `and` 与 `or` 是相似的条件式。
7. Lisp 主要由函数所组成。可以用 `defun` 来定义新的函数。
8. 自己调用自己的函数是递归的。一个递归函数应该要被想成是过程，而不是机器。
9. 括号不是问题，因为程序员通过缩排来阅读与编写 Lisp 程序。
10. 基本的 I/O 函数是 `read` ，它包含了一个完整的 Lisp 语法分析器，以及 `format` ，它通过字符串模板来产生输出。
11. 你可以用 `let` 来创造新的局部变量，用 `defparameter` 来创造全局变量。
12. 赋值操作符是 `setf` 。它的第一个实参可以是一个表达式。
13. 函数式编程代表避免产生副作用，也是 Lisp 的主导思维。
14. 基本的迭代操作符是 `do` 。
15. 函数是 Lisp 的对象。可以被当成实参传入，并且可以用 lambda 表达式来表示。
16. 在 Lisp 里，是数值才有类型，而不是变量。

Chapter 2 习题 (Exercises)

1. 描述下列表达式求值之后的结果：

```
(a) (+ (- 5 1) (+ 3 7))

(b) (list 1 (+ 2 3))

(c) (if (listp 1) (+ 1 2) (+ 3 4))

(d) (list (and (listp 3) t) (+ 1 2))
```

2. 给出 3 种不同表示 `(a b c)` 的 `cons` 表达式。
3. 使用 `car` 与 `cdr` 来定义一个函数，返回一个列表的第四个元素。
4. 定义一个函数，接受两个实参，返回两者当中较大的那个。
5. 这些函数做了什么？

```
(a) (defun enigma (x)
      (and (not (null x))
           (or (null (car x))
               (enigma (cdr x)))))

(b) (defun mystery (x y)
      (if (null y)
          nil
          (if (eql (car y) x)
              0
              (let ((z (mystery x (cdr y))))
                (and z (+ z 1))))))
```

6. 下列表达式， `x` 该是什么，才会得到相同的结果？

```
(a) > (car (x (cdr '(a (b c) d))))
    B
(b) > (x 13 (/ 1 0))
    13
(c) > (x #'list 1 nil)
    (1)
```

7. 只使用本章所介绍的操作符，定义一个函数，它接受一个列表作为实参，如果有一个元素是列表时，就返回真。
8. 给出函数的迭代与递归版本：

a. 接受一个正整数，并打印出数字数量的点。
b. 接受一个列表，并返回 `a` 在列表里所出现的次数。

9. 一位朋友想写一个函数，返回列表里所有非 `nil` 元素的和。他写了此函数的两个版本，但两个都不能工作。请解释每一个的错误在哪里，并给出正确的版本。

```
(a) (defun summit (lst)
      (remove nil lst)
      (apply #'+ lst))

(b) (defun summit (lst)
      (let ((x (car lst)))
        (if (null x)
            (summit (cdr lst))
            (+ x (summit (cdr lst))))))
```

脚注

[1] 在 vi，你可以用 :set sm 来启用括号匹配。在 Emacs，M-x lisp-mode 是一个启用的好方法。

[2] 真正的区别是词法变量（lexical）与特殊变量（special variable），但到第六章才会讨论这个主题。

第三章：列表

列表是 Lisp 的基本数据结构之一。在最早的 Lisp 方言里，列表是唯一的数据结构：“Lisp” 这个名字起初是 “LISt Processor” 的缩写。但 Lisp 已经超越这个缩写很久了。Common Lisp 是一个有着各式各样数据结构的通用性程序语言。

Lisp 程序开发通常呼应着开发 Lisp 语言自身。在最初版本的 Lisp 程序，你可能使用很多列表。然而之后的版本，你可能换到快速、特定的数据结构。本章描述了你可以用列表所做的很多事情，以及使用它们来演示一些普遍的 Lisp 概念。

3.1 构造 (Conses)

在 2.4 节我们介绍了 `cons` , `car` , 以及 `cdr` ，基本的 List 操作函数。 `cons` 真正所做的事情是，把两个对象结合成一个有两部分的对象，称之为 *Cons* 对象。概念上来说，一个 *Cons* 是一对指针；第一个是 `car` ，第二个是 `cdr` 。

Cons 对象提供了一个方便的表示法，来表示任何类型的对象。一个 *Cons* 对象里的一对指针，可以指向任何类型的对象，包括 *Cons* 对象本身。它利用到我们之后可以用 `cons` 来构造列表的可能性。

我们往往不会把列表想成是成对的，但它们可以这样被定义。任何非空的列表，都可以被视为一对由列表第一个元素及列表其余元素所组成的列表。 Lisp 列表体现了这个概念。我们使用 *Cons* 的一半来指向列表的第一个元素，然后用另一半指向列表其余的元素(可能是别的 *Cons* 或 `nil`)。 Lisp 的惯例是使用 `car` 代表列表的第一个元素，而用 `cdr` 代表列表的其余的元素。所以现在 `car` 是列表的第一个元素的同义词，而 `cdr` 是列表的其余的元素的同义词。列表不是不同的对象，而是像 *Cons* 这样的方式连结起来。

当我们想在 `nil` 上面建立东西时，

```
> (setf x (cons 'a nil))
(A)
```

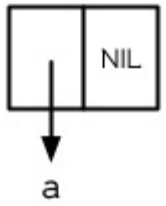

图 3.1 一个元素的列表

产生的列表由一个 *Cons* 所组成，见图 3.1。这种表达 *Cons* 的方式叫做箱子表示法 (box notation)，因为每一个 Cons 是用一个箱子表示，内含一个 `car` 和 `cdr` 的指针。当我们调用 `car` 与 `cdr` 时，我们得到指针指向的地方：

```
> (car x)
A
> (cdr x)
NIL
```

当我们构造一个多元素的列表时，我们得到一串 *Cons* (a chain of conses):

```
> (setf y (list 'a 'b 'c))
(A B C)
```

产生的结构见图 3.2。现在当我们想得到列表的 `cdr` 时，它是一个两个元素的列表。

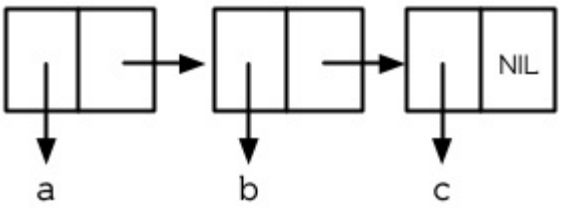

图 3.2 三个元素的列表

```
> (cdr y)
(B C)
```

在一个有多个元素的列表中， `car` 指针让你取得元素，而 `cdr` 让你取得列表内其余的东西。

一个列表可以有任何类型的对象作为元素，包括另一个列表：

```
> (setf z (list 'a (list 'b 'c) 'd))
(A (B C) D)
```

当这种情况发生时，它的结构如图 3.3 所示；第二个 *Cons* 的 `car` 指针也指向一个列表：

```
> (car (cdr z))
(B C)
```

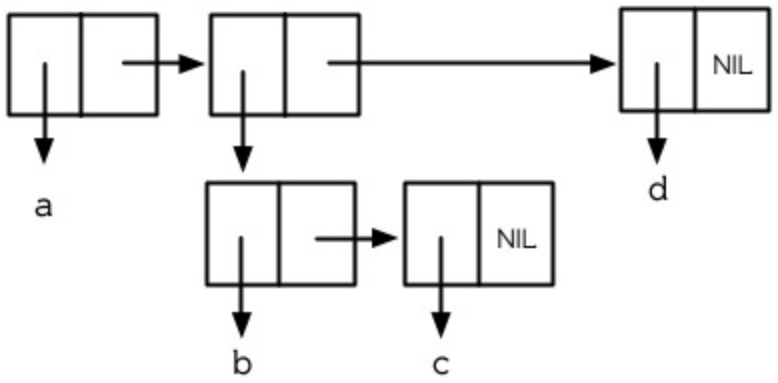

图 3.3 嵌套列表

前两个我们构造的列表都有三个元素；只不过 `z` 列表的第二个元素也刚好是一个列表。像这样的列表称为嵌套列表，而像 `y` 这样的列表称之为平坦列表 (*flat*list)。

如果参数是一个 *Cons* 对象，函数 `consp` 返回真。所以我们可以这样定义 `listp`：

```
(defun our-listp (x)
  (or (null x) (consp x)))
```

因为所有不是 *Cons* 对象的东西，就是一个原子 (atom)，判断式 `atom` 可以这样定义：

```
(defun our-atom (x) (not (consp x)))
```

注意， `nil` 既是一个原子，也是一个列表。

3.2 等式 (Equality)

每一次你调用 `cons` 时， Lisp 会配置一块新的内存给两个指针。所以如果我们用同样的参数调用 `cons` 两次，我们得到两个数值看起来一样，但实际上是两个不同的对象：

```
> (eql (cons 'a nil) (cons 'a nil))
NIL
```

如果我们也可以询问两个列表是否有相同元素，那就很方便了。 Common Lisp 提供了这种目的另一个判断式： `equal` 。而另一方面 `eql` 只有在它的参数是相同对象时才返回真，

```
> (setf x (cons 'a nil))
(A)
> (eql x x)
T
```

本质上 `equal` 若它的参数打印出的值相同时，返回真：

```
> (equal x (cons 'a nil))
T
```

这个判断式对非列表结构的别种对象也有效，但一种仅对列表有效的版本可以这样定义：

```
> (defun our-equal (x y)
    (or (eql x y)
        (and (consp x)
             (consp y)
             (our-equal (car x) (car y))
             (our-equal (cdr x) (cdr y)))))
```

这个定义意味着，如果某个 `x` 和 `y` 相等(`eql`)，那么他们也相等(`equal`)。

勘误：这个版本的 `our-equal` 可以用在符号的列表 (list of symbols)，而不是列表 (list)。

3.3 为什么 Lisp 没有指针 (Why Lisp Has No Pointers)

一个理解 Lisp 的秘密之一是意识到变量是有值的，就像列表有元素一样。如同 *Cons* 对象有指针指向他们的元素，变量有指针指向他们的值。

你可能在别的语言中使用过显式指针 (explicitly pointer)。在 Lisp，你永远不用这么做，因为语言帮你处理好指针了。我们已经在列表看过这是怎么实现的。同样的事情发生在变量身上。举例来说，假设我们想要把两个变量设成同样的列表：

```
 > (setf x '(a b c))
(A B C)
> (setf y x)
(A B C)
```

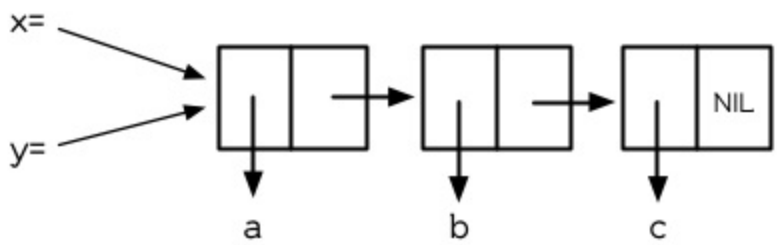

图 3.4 两个变量设为相同的列表

当我们把 `x` 的值赋给 `y` 时，究竟发生什么事呢？内存中与 `x` 有关的位置并没有包含这个

列表，而是一个指针指向它。当我们给 y 赋一个相同的值时， Lisp 复制的是指针，而不是列表。（图 3.4 显式赋值 x 给 y 后的结果）无论何时，你将某个变量的值赋给另个变量时，两个变量的值将会是 eql 的：

```
> (eql x y)
T
```

Lisp 没有指针的原因是因为每一个值，其实概念上来说都是一个指针。当你赋一个值给变量或将这个值存在数据结构中，其实被储存的是指向这个值的指针。当你要取得变量的值，或是存在数据结构中的内容时， Lisp 返回指向这个值的指针。但这都在台面下发生。你可以不加思索地把值放在结构里，或放“在”变量里。

为了效率的原因， Lisp 有时会选择一个折衷的表示法，而不是指针。举例来说，因为一个小整数所需的内存空间，少于一个指针所需的空间，一个 Lisp 实现可能会直接处理这个小整数，而不是用指针来处理。但基本要点是，程序员预设可以把任何东西放在任何地方。除非你声明你不愿这么做，不然你能够在任何的数据结构，存放任何类型的对象，包括结构本身。

3.4 建立列表 (Building Lists)

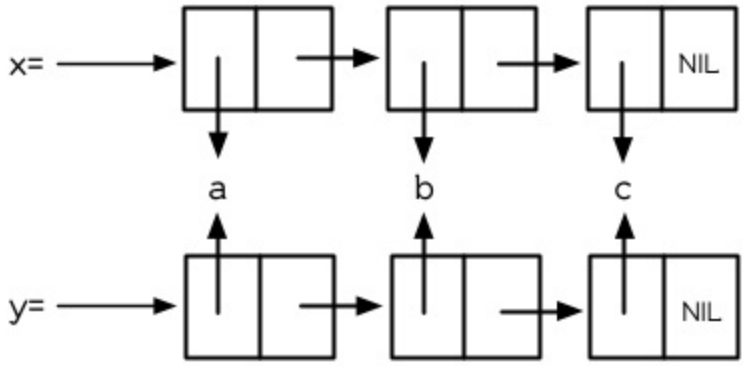

图 3.5 复制的结果

函数 copy-list 接受一个列表，然后返回此列表的复本。新的列表会有同样的元素，但是装在新的 *Cons* 对象里：

```
> (setf x '(a b c)
        y (copy-list x))
(A B C)
```

图 3.5 展示出结果的结构; 返回值像是有着相同乘客的新公交。我们可以把 copy-list 想成是这么定义的：

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

这个定义暗示着 x 与 (copy-list x) 会永远 equal，并永远不 eql，除非 x 是 NIL。

最后，函数 append 返回任何数目的列表串接 (concatenation)：

```
> (append '(a b) '(c d) 'e)
(A B C D . E)
```

通过这么做，它复制所有的参数，除了最后一个

3.5 示例：压缩 (Example: Compression)

作为一个例子，这节将演示如何实现简单形式的列表压缩。这个算法有一个令人印象深刻的名字，游程编码(run-length encoding)。

```
(defun compress (x)
  (if (consp x)
      (compr (car x) 1 (cdr x))
      x))

(defun compr (elt n lst)
  (if (null lst)
      (list (n-elts elt n))
      (let ((next (car lst)))
        (if (eql next elt)
            (compr elt (+ n 1) (cdr lst))
            (cons (n-elts elt n)
                  (compr next 1 (cdr lst)))))))

(defun n-elts (elt n)
  (if (> n 1)
      (list n elt)
      elt))
```

图 3.6 游程编码 (Run-length encoding)：压缩

在餐厅的情境下，这个算法的工作方式如下。一个女服务生走向有四个客人的桌子。“你们要什么？” 她问。“我要特餐，” 第一个客人说。 “我也是，” 第二个客人说。“听起来不错，” 第三个客人说。每个人看着第四个客人。 “我要一个 cilantro soufflé，” 他小声地说。 (译注：蛋奶酥上面洒香菜跟酱料）

瞬息之间，女服务生就转身踩着高跟鞋走回柜台去了。 “三个特餐，” 她大声对厨师

说，“还有一个香菜蛋奶酥。”

图 3.6 展示了如何实现这个压缩列表演算法。函数 compress 接受一个由原子组成的列表，然后返回一个压缩的列表：

```
> (compress '(1 1 1 0 1 0 0 0 0 1))
((3 1) 0 1 (4 0) 1)
```

当相同的元素连续出现好几次，这个连续出现的序列 (sequence)被一个列表取代，列表指明出现的次数及出现的元素。

主要的工作是由递归函数 compr 所完成。这个函数接受三个参数： elt ， 上一个我们看过的元素； n ，连续出现的次数；以及 lst ，我们还没检查过的部分列表。如果没有东西需要检查了，我们调用 n-elts 来取得 n elts 的表示法。如果 lst 的第一个元素还是 elt ，我们增加出现的次数 n 并继续下去。否则我们得到，到目前为止的一个压缩的列表，然后 cons 这个列表在 compr 处理完剩下的列表所返回的东西之上。

要复原一个压缩的列表，我们调用 uncompress (图 3.7)

```
> (uncompress '((3 1) 0 1 (4 0) 1))
(1 1 1 0 1 0 0 0 0 1)
```

```
(defun uncompress (lst)
  (if (null lst)
      nil
      (let ((elt (car lst))
            (rest (uncompress (cdr lst))))
        (if (consp elt)
            (append (apply #'list-of elt)
                    rest)
            (cons elt rest)))))

(defun list-of (n elt)
  (if (zerop n)
      nil
      (cons elt (list-of (- n 1) elt))))
```

图 3.7 游程编码 (Run-length encoding)：解压缩

这个函数递归地遍历这个压缩列表，逐字复制原子并调用 list-of ，展开成列表。

```
> (list-of 3 'ho)
(HO HO HO)
```

我们其实不需要自己写 list-of 。内置的 make-list 可以办到一样的事情 —— 但它使用

了我们还没介绍到的关键字参数 (keyword argument)。

图 3.6 跟 3.7 这种写法不是一个有经验的Lisp 程序员用的写法。它的效率很差，它没有尽可能的压缩，而且它只对由原子组成的列表有效。在几个章节内，我们会学到解决这些问题的技巧。

```
载入程序

在这节的程序是我们第一个实质的程序。
当我们想要写超过数行的函数时，
通常我们会把程序写在一个文件，
然后使用 load 让 Lisp 读取函数的定义。
如果我们把图 3.6 跟 3.7 的程序，
存在一个文件叫做，“compress.lisp”然后输入

(load "compress.lisp")

到顶层，或多或少的，
我们会像在直接输入顶层一样得到同样的效果。

注意：在某些实现中，Lisp 文件的扩展名会是“.lsp”而不是“.lisp”。
```

3.6 存取 (Access)

Common Lisp 有额外的存取函数，它们是用 car 跟 cdr 所定义的。要找到列表特定位置的元素，我们可以调用 nth ，

```
> (nth 0 '(a b c))
A
```

而要找到第 n 个 cdr ，我们调用 nthcdr ：

```
> (nthcdr 2 '(a b c))
(C)
```

nth 与 nthcdr 都是零索引的 (zero-indexed); 即元素从 0 开始编号，而不是从 1 开始。在 Common Lisp 里，无论何时你使用一个数字来参照一个数据结构中的元素时，都是从 0 开始编号的。

两个函数几乎做一样的事; nth 等同于取 nthcdr 的 car 。没有检查错误的情况下，nthcdr 可以这么定义：

```
(defun our-nthcdr (n lst)
  (if (zerop n)
      lst
```

```
      (our-nthcdr (- n 1) (cdr lst))))
```

函数 `zerop` 仅在参数为零时，才返回真。

函数 `last` 返回列表的最后一个 *Cons* 对象：

```
> (last '(a b c))
(C)
```

这跟取得最后一个元素不一样。要取得列表的最后一个元素，你要取得 `last` 的 `car` 。

Common Lisp 定义了函数 `first` 直到 `tenth` 可以取得列表对应的元素。这些函数不是 零索引的 (zero-indexed)：

`(second x)` 等同于 `(nth 1 x)` 。

此外， Common Lisp 定义了像是 `caddr` 这样的函数，它是 `cdr` 的 `cdr` 的 `car` 的缩写 (`car` of `cdr` of `cdr`)。所有这样形式的函数 `cxr` ，其中 x 是一个字符串，最多四个 `a` 或 `d` ，在 Common Lisp 里都被定义好了。使用 `cadr` 可能会有异常 (exception)产生，在所有人都可能会读的代码里使用这样的函数，可能不是个好主意。

3.7 映射函数 (Mapping Functions)

Common Lisp 提供了数个函数来对一个列表的元素做函数调用。最常使用的是 `mapcar` ，接受一个函数以及一个或多个列表，并返回把函数应用至每个列表的元素的结果，直到有的列表没有元素为止：

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

> (mapcar #'list
          '(a b c)
          '(1 2 3 4))
((A 1) (B 2) (C 3))
```

相关的 `maplist` 接受同样的参数，将列表的渐进的下一个 `cdr` 传入函数。

```
> (maplist #'(lambda (x) x)
           '(a b c))
((A B C) (B C) (C))
```

其它的映射函数，包括 `mapc` 我们在 89 页讨论（译注：5.4 节最后），以及 `mapcan` 在 202 页（译注：12.4 节最后）讨论。

3.8 树 (Trees)

Cons 对象可以想成是二叉树， `car` 代表左子树，而 `cdr` 代表右子树。举例来说，列表 `(a (b c) d)` 也是一棵由图 3.8 所代表的树。 （如果你逆时针旋转 45 度，你会发现跟图 3.3 一模一样）

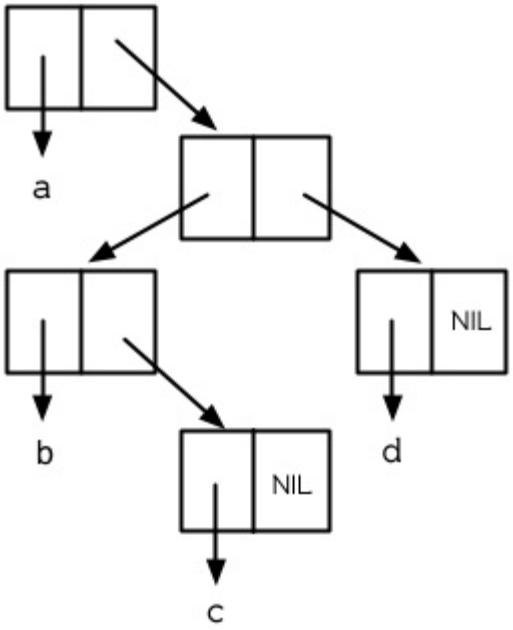

图 3.8 二叉树 (Binary Tree)

Common Lisp 有几个内置的操作树的函数。举例来说， `copy-tree` 接受一个树，并返回一份副本。它可以这么定义：

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

把这跟 36 页的 `copy-list` 比较一下； `copy-tree` 复制每一个 *Cons* 对象的 `car` 与 `cdr` ，而 `copy-list` 仅复制 `cdr` 。

没有内部节点的二叉树没有太大的用处。 Common Lisp 包含了操作树的函数，不只是因为我们需要树这个结构，而是因为我们需要一种方法，来操作列表及所有内部的列表。举例来说，假设我们有一个这样的列表：

```
(and (integerp x) (zerop (mod x 2)))
```

而我们想要把各处的 `x` 都换成 `y` 。调用 `substitute` 是不行的，它只能替换序列(sequence)中的元素：

```
> (substitute 'y 'x '(and (integerp x) (zerop (mod x 2))))
(AND (INTEGERP X) (ZEROP (MOD X 2)))
```

这个调用是无效的，因为列表有三个元素，没有一个元素是 `x` 。我们在这所需要的是 `subst` ，它替换树之中的元素。

```
> (subst 'y 'x '(and (integerp x) (zerop (mod x 2))))
(AND (INTEGERP Y) (ZEROP (MOD Y 2)))
```

如果我们定义一个 `subst` 的版本，它看起来跟 `copy-tree` 很相似：

```
> (defun our-subst (new old tree)
    (if (eql tree old)
        new
        (if (atom tree)
            tree
            (cons (our-subst new old (car tree))
                  (our-subst new old (cdr tree))))))
```

操作树的函数通常有这种形式， `car` 与 `cdr` 同时做递归。这种函数被称之为是 双重递归 (doubly recursive)。

3.9 理解递归 (Understanding Recursion)

学生在学习递归时，有时候是被鼓励在纸上追踪 (trace)递归程序调用 (invocation)的过程。 (288页「译注：附录 A 追踪与回溯 [http://acl.readthedocs.org/en/latest/zhCN/appendix-A-cn.html]」可以看到一个递归函数的追踪过程。)但这种练习可能会误导你：一个程序员在定义一个递归函数时，通常不会特别地去想函数的调用顺序所导致的结果。

如果一个人总是需要这样子思考程序，递归会是艰难的、没有帮助的。递归的优点是它精确地让我们更抽象地来设计算法。你不需要考虑真正函数时所有的调用过程，就可以判断一个递归函数是否是正确的。

要知道一个递归函数是否做它该做的事，你只需要问，它包含了所有的情况吗？举例来说，下面是一个寻找列表长度的递归函数：

```
> (defun len (lst)
    (if (null lst)
        0
        (+ (len (cdr lst)) 1)))
```

我们可以借由检查两件事情，来确信这个函数是正确的：

1. 对长度为 0 的列表是有效的。
2. 给定它对于长度为 n 的列表是有效的，它对长度是 n+1 的列表也是有效的。

如果这两点是成立的，我们知道这个函数对于所有可能的列表都是正确的。

我们的定义显然地满足第一点：如果列表(lst) 是空的(nil)，函数直接返回 0 。现在假定我们的函数对长度为 n 的列表是有效的。我们给它一个 n+1 长度的列表。这个定义说明了，函数会返回列表的 cdr 的长度再加上 1 。 cdr 是一个长度为 n 的列表。我们经由假定可知它的长度是 n 。所以整个列表的长度是 n+1 。

我们需要知道的就是这些。理解递归的秘密就像是处理括号一样。你怎么知道哪个括号对上哪个？你不需要这么做。你怎么想像那些调用过程？你不需要这么做。

更复杂的递归函数，可能会有更多的情况需要讨论，但是流程是一样的。举例来说，41 页的 our-copy-tree ，我们需要讨论三个情况： 原子，单一的 *Cons* 对象， n+1 的 *Cons* 树。

第一个情况（长度零的列表）称之为基本用例(*base case*)。当一个递归函数不像你想的那样工作时，通常是处理基本用例就错了。下面这个不正确的 member 定义，是一个常见的错误，整个忽略了基本用例：

```
(defun our-member (obj lst)
  (if (eql (car lst) obj)
      lst
      (our-member obj (cdr lst))))
```

我们需要初始一个 null 测试，确保在到达列表底部时，没有找到目标时要停止递归。如果我们要找的对象没有在列表里，这个版本的 member 会陷入无穷循环。附录 A 更详细地讨论了这种问题。

能够判断一个递归函数是否正确只不过是理解递归的上半场，下半场是能够写出一个做你想做的事情的递归函数。 6.9 节讨论了这个问题。

3.10 集合 (Sets)

列表是表示小集合的好方法。列表中的每个元素都代表了一个集合的成员：

```
> (member 'b '(a b c))
(B C)
```

当 member 要返回“真”时，与其仅仅返回 t ，它返回由寻找对象所开始的那部分。逻辑上来说，一个 *Cons* 扮演的角色和 t 一样，而经由这么做，函数返回了更多资讯。

一般情况下， member 使用 eql 来比较对象。你可以使用一种叫做关键字参数的东西来重写缺省的比较方法。多数的 Common Lisp 函数接受一个或多个关键字参数。这些关键字参数不同的地方是，他们不是把对应的参数放在特定的位置作匹配，而是在函数调用中用特殊标签，称为关键字，来作匹配。一个关键字是一个前面有冒号的符号。

一个 member 函数所接受的关键字参数是 :test 参数。

如果你在调用 member 时，传入某个函数作为 :test 参数，那么那个函数就会被用来比较是否相等，而不是用 eql 。所以如果我们想找到一个给定的对象与列表中的成员是否相等(equal)，我们可以：

```
> (member '(a) '((a) (z)) :test #'equal)
((A) (Z))
```

关键字参数总是选择性添加的。如果你在一个调用中包含了任何的关键字参数，他们要摆在最后; 如果使用了超过一个的关键字参数，摆放的顺序无关紧要。

另一个 member 接受的关键字参数是 :key 参数。借由提供这个参数，你可以在作比较之前，指定一个函数运用在每一个元素：

```
> (member 'a '((a b) (c d)) :key #'car)
((A B) (C D))
```

在这个例子里，我们询问是否有一个元素的 car 是 a 。

如果我们想要使用两个关键字参数，我们可以使用其中一个顺序。下面这两个调用是等价的：

```
> (member 2 '((1) (2)) :key #'car :test #'equal)
((2))
> (member 2 '((1) (2)) :test #'equal :key #'car)
((2))
```

两者都询问是否有一个元素的 car 等于(equal) 2。

如果我们想要找到一个元素满足任意的判断式像是—— oddp ，奇数返回真——我们可以使用相关的 member-if ：

```
> (member-if #'oddp '(2 3 4))
(3 4)
```

我们可以想像一个限制性的版本 member-if 是这样写成的：

```
(defun our-member-if (fn lst)
  (and (consp lst)
       (if (funcall fn (car lst))
           lst
           (our-member-if fn (cdr lst)))))
```

函数 adjoin 像是条件式的 cons 。它接受一个对象及一个列表，如果对象还不是列表的成员，才构造对象至列表上。

```
> (adjoin 'b '(a b c))
(A B C)
> (adjoin 'z '(a b c))
(Z A B C)
```

通常的情况下它接受与 member 函数同样的关键字参数。

集合论中的并集 (union)、交集 (intersection)以及补集 (complement)的实现，是由函数 union 、 intersection 以及 set-difference 。

这些函数期望两个（正好 2 个）列表（一样接受与 member 函数同样的关键字参数）。

```
> (union '(a b c) '(c b s))
(A C B S)
> (intersection '(a b c) '(b b c))
(B C)
> (set-difference '(a b c d e) '(b e))
(A C D)
```

因为集合中没有顺序的概念，这些函数不需要保留原本元素在列表被找到的顺序。举例来说，调用 set-difference 也有可能返回 (d c a) 。

3.11 序列 (Sequences)

另一种考虑一个列表的方式是想成一系列有特定顺序的对象。在 Common Lisp 里，序列(*sequences*)包括了列表与向量 (vectors)。本节介绍了一些可以运用在列表上的序列函数。更深入的序列操作在 4.4 节讨论。

函数 length 返回序列中元素的数目。

```
> (length '(a b c))
3
```

我们在 24 页 (译注：2.13节 `our-length`)写过这种函数的一个版本（仅可用于列表）。

要复制序列的一部分，我们使用 `subseq` 。第二个（需要的）参数是第一个开始引用进来的元素位置，第三个（选择性）参数是第一个不引用进来的元素位置。

```
> (subseq '(a b c d) 1 2)
(B)
>(subseq '(a b c d) 1)
(B C D)
```

如果省略了第三个参数，子序列会从第二个参数给定的位置引用到序列尾端。

函数 `reverse` 返回与其参数相同元素的一个序列，但顺序颠倒。

```
> (reverse '(a b c))
(C B A)
```

一个回文 (palindrome) 是一个正读反读都一样的序列 —— 举例来说， `(abba)` 。如果一个回文有偶数个元素，那么后半段会是前半段的镜射 (mirror)。使用 `length` 、 `subseq` 以及 `reverse` ，我们可以定义一个函数

```
(defun mirror? (s)
  (let ((len (length s)))
    (and (evenp len)
         (let ((mid (/ len 2)))
           (equal (subseq s 0 mid)
                  (reverse (subseq s mid)))))))
```

来检测是否是回文：

```
> (mirror? '(a b b a))
T
```

Common Lisp 有一个内置的排序函数叫做 `sort` 。它接受一个序列及一个比较两个参数的函数，返回一个有同样元素的序列，根据比较函数来排序：

```
> (sort '(0 2 1 3 8) #'>)
(8 3 2 1 0)
```

你要小心使用 `sort` ，因为它是破坏性的(*destructive*)。考虑到效率的因素， `sort` 被允许修改传入的序列。所以如果你不想你本来的序列被改动，传入一个副本。

使用 `sort` 及 `nth` ，我们可以写一个函数，接受一个整数 `n` ，返回列表中第 `n` 大的元素：

```
(defun nthmost (n lst)
  (nth (- n 1)
       (sort (copy-list lst) #'>)))
```

我们把整数减一因为 nth 是零索引的，但如果 nthmost 是这样的话，会变得很不直观。

```
(nthmost 2 '(0 2 1 3 8))
```

多努力一点，我们可以写出这个函数的一个更有效率的版本。

函数 every 和 some 接受一个判断式及一个或多个序列。当我们仅输入一个序列时，它们测试序列元素是否满足判断式：

```
> (every #'oddp '(1 3 5))
T
> (some #'evenp '(1 2 3))
T
```

如果它们输入多于一个序列时，判断式必须接受与序列一样多的元素作为参数，而参数从所有序列中一次提取一个：

```
> (every #'> '(1 3 5) '(0 2 4))
T
```

如果序列有不同的长度，最短的那个序列，决定需要测试的次数。

3.12 栈 (Stacks)

用 *Cons* 对象来表示的列表，很自然地我们可以拿来实现下推栈 (pushdown stack)。这太常见了，以致于 Common Lisp 提供了两个宏给堆使用： (push x y) 把 x 放入列表 y 的前端。而 (pop x) 则是将列表 x 的第一个元素移除，并返回这个元素。

两个函数都是由 setf 定义的。如果参数是常数或变量，很简单就可以翻译出对应的函数调用。

表达式

```
(push obj lst)
```

等同于

```
(setf lst (cons obj lst))
```

而表达式

```
(pop lst)
```

等同于

```
(let ((x (car lst)))
  (setf lst (cdr lst))
  x)
```

所以，举例来说：

```
> (setf x '(b))
(B)
> (push 'a x)
(A B)
> x
(A B)
> (setf y x)
(A B)
> (pop x)
(A)
> x
(B)
> y
(A B)
```

以上，全都遵循上述由 `setf` 所给出的相等式。图 3.9 展示了这些表达式被求值后的结构。

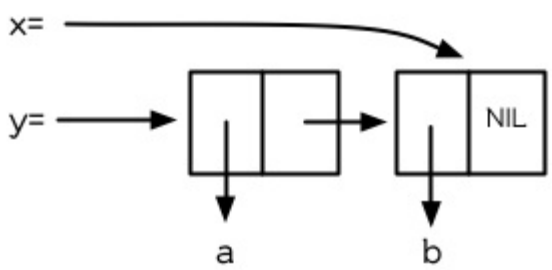

图 3.9 push 及 pop 的效果

你可以使用 `push` 来定义一个给列表使用的互动版 `reverse` 。

```
(defun our-reverse (lst)
  (let ((acc nil))
    (dolist (elt lst)
      (push elt acc))
    acc))
```

在这个版本，我们从一个空列表开始，然后把 lst 的每一个元素放入空表里。等我们完成时，lst 最后一个元素会在最前端。

pushnew 宏是 push 的变种，使用了 adjoin 而不是 cons：

```
 > (let ((x '(a b)))
     (pushnew 'c x)
     (pushnew 'a x)
     x)
(C A B)
```

在这里， c 被放入列表，但是 a 没有，因为它已经是列表的一个成员了。

3.13 点状列表 (Dotted Lists)

调用 list 所构造的列表，这种列表精确地说称之为正规列表(*proper*list)。一个正规列表可以是 NIL 或是 cdr 是正规列表的 *Cons* 对象。也就是说，我们可以定义一个只对正规列表返回真的判断式： [3]

```
(defun proper-list? (x)
  (or (null x)
      (and (consp x)
           (proper-list? (cdr x)))))
```

至目前为止，我们构造的列表都是正规列表。

然而， cons 不仅是构造列表。无论何时你需要一个具有两个字段 (field)的列表，你可以使用一个 *Cons* 对象。你能够使用 car 来参照第一个字段，用 cdr 来参照第二个字段。

```
> (setf pair (cons 'a 'b))
(A . B)
```

因为这个 *Cons* 对象不是一个正规列表，它用点状表示法来显示。在点状表示法，每个 *Cons* 对象的 car 与 cdr 由一个句点隔开来表示。这个 *Cons* 对象的结构展示在图 3.10 。

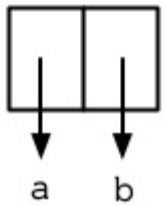

图3.10 一个成对的 *Cons* 对象 (A cons used as a pair)

一个非正规列表的 *Cons* 对象称之为点状列表 (dotted list)。这不是个好名字，因为非正规列表的 Cons 对象通常不是用来表示列表：`(a . b)` 只是一个有两部分的数据结构。

你也可以用点状表示法表示正规列表，但当 Lisp 显示一个正规列表时，它会使用普通的列表表示法：

```
> '(a . (b . (c . nil)))
(A B C)
```

顺道一提，注意列表由点状表示法与图 3.2 箱子表示法的关联性。

还有一个过渡形式 (intermediate form)的表示法，介于列表表示法及纯点状表示法之间，对于 `cdr` 是点状列表的 *Cons* 对象：

```
> (cons 'a (cons 'b (cons 'c 'd)))
(A B C . D)
```

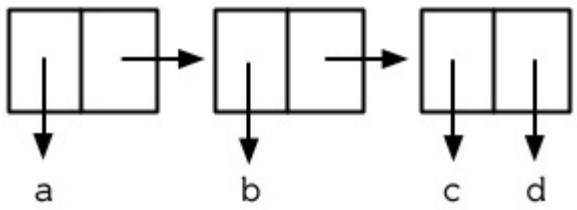

图 3.11 一个点状列表 (A dotted list)

这样的 *Cons* 对象看起来像正规列表，除了最后一个 cdr 前面有一个句点。这个列表的结构展示在图 3.11 ; 注意它跟图3.2 是多么的相似。

所以实际上你可以这么表示列表 `(a b)`，

```
(a . (b . nil))
(a . (b))
(a b . nil)
(a b)
```

虽然 Lisp 总是使用后面的形式，来显示这个列表。

3.14 关联列表 (Assoc-lists)

用 *Cons* 对象来表示映射 (mapping)也是很自然的。一个由 *Cons* 对象组成的列表称之为

关联列表(*assoc-list*or *alist*)。这样的列表可以表示一个翻译的集合，举例来说：

```
> (setf trans '((+ . "add") (- . "subtract")))
((+ . "add") (- . "subtract"))
```

关联列表很慢，但是在初期的程序中很方便。 Common Lisp 有一个内置的函数 `assoc`，用来取出在关联列表中，与给定的键值有关联的 *Cons* 对：

```
> (assoc '+ trans)
(+ . "add")
> (assoc '* trans)
NIL
```

如果 `assoc` 没有找到要找的东西时，返回 `nil` 。

我们可以定义一个受限版本的 `assoc`：

```
(defun our-assoc (key alist)
  (and (consp alist)
       (let ((pair (car alist)))
        (if (eql key (car pair))
            pair
            (our-assoc key (cdr alist))))))
```

和 `member` 一样，实际上的 `assoc` 接受关键字参数，包括 `:test` 和 `:key` 。 Common Lisp 也定义了一个 `assoc-if` 之于 `assoc`，如同 `member-if` 之于 `member` 一样。

3.15 示例：最短路径 (Example: Shortest Path)

图 3.12 包含一个搜索网络中最短路径的程序。函数 `shortest-path` 接受一个起始节点，目的节点以及一个网络，并返回最短路径，如果有的话。

在这个范例中，节点用符号表示，而网络用含以下元素形式的关联列表来表示：

(node . neighbors)

所以由图 3.13 展示的最小网络 (minimal network)可以这样来表示：

```
(setf min '((a b c) (b c) (c d)))
```

```
(defun shortest-path (start end net)
  (bfs end (list (list start)) net))

(defun bfs (end queue net)
  (if (null queue)
```

```
      nil
      (let ((path (car queue)))
        (let ((node (car path)))
          (if (eql node end)
              (reverse path)
              (bfs end
                   (append (cdr queue)
                           (new-paths path node net))
                   net))))))

(defun new-paths (path node net)
  (mapcar #'(lambda (n)
              (cons n path))
          (cdr (assoc node net))))
```

图 3.12 广度优先搜索(breadth-first search)

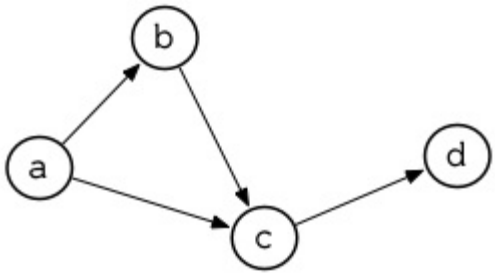

图 3.13 最小网络

要找到从节点 a 可以到达的节点，我们可以：

```
> (cdr (assoc 'a min))
(B C)
```

图 3.12 程序使用广度优先的方式搜索网络。要使用广度优先搜索，你需要维护一个含有未探索节点的队列。每一次你到达一个节点，检查这个节点是否是你要的。如果不是，你把这个节点的子节点加入队列的尾端，并从队列起始选一个节点，从这继续搜索。借由总是把较深的节点放在队列尾端，我们确保网络一次被搜索一层。

图 3.12 中的代码较不复杂地表示这个概念。我们不仅想要找到节点，还想保有我们怎么到那的纪录。所以与其维护一个具有节点的队列 (queue)，我们维护一个已知路径的队列，每个已知路径都是一列节点。当我们从队列取出一个元素继续搜索时，它是一个含有队列前端节点的列表，而不只是一个节点而已。

函数 `bfs` 负责搜索。起初队列只有一个元素，一个表示从起点开始的路径。所以 `shortest-path` 调用 `bfs`，并传入 `(list (list start))` 作为初始队列。

bfs 函数第一件要考虑的事是，是否还有节点需要探索。如果队列为空， bfs 返回 nil 指出没有找到路径。如果还有节点需要搜索， bfs 检查队列前端的节点。如果节点的 car 部分是我们要找的节点，我们返回这个找到的路径，并且为了可读性的原因我们反转它。如果我们没有找到我们要找的节点，它有可能在现在节点之后，所以我们把它的子节点（或是每一个子路径）加入队列尾端。然后我们递回地调用 bfs 来继续搜寻剩下的队列。

因为 bfs 广度优先地搜索，第一个找到的路径会是最短的，或是最短之一：

```
> (shortest-path 'a 'd min)
(A C D)
```

这是队列在我们连续调用 bfs 看起来的样子：

```
((A))
((B A) (C A))
((C A) (C B A))
((C B A) (D C A))
((D C A) (D C B A))
```

在队列中的第二个元素变成下一个队列的第一个元素。队列的第一个元素变成下一个队列尾端元素的 cdr 部分。

在图 3.12 的代码不是搜索一个网络最快的方法，但它给出了列表具有多功能的概念。在这个简单的程序中，我们用三种不同的方式使用了列表：我们使用一个符号的列表来表示路径，一个路径的列表来表示在广度优先搜索中的队列 [4] ，以及一个关联列表来表示网络本身。

3.16 垃圾 (Garbages)

有很多原因可以使列表变慢。列表提供了顺序存取而不是随机存取，所以列表取出一个指定的元素比数组慢，同样的原因，录音带取出某些东西比在光盘上慢。电脑内部里，*Cons* 对象倾向于用指针表示，所以走访一个列表意味着走访一系列的指针，而不是简单地像数组一样增加索引值。但这两个所花的代价与配置及回收 *Cons* 核 (cons cells)比起来小多了。

自动内存管理(*Automatic memory management*)是 Lisp 最有价值的特色之一。 Lisp 系统维护着一段内存称之为堆(*Heap*)。系统持续追踪堆当中没有使用的内存，把这些内存发放给新产生的对象。举例来说，函数 cons ，返回一个新配置的 *Cons* 对象。从堆中配置内存有时候通称为 *consing* 。

如果内存永远没有释放， Lisp 会因为创建新对象把内存用完，而必须要关闭。所以系

统必须周期性地通过搜索堆 (heap)，寻找不需要再使用的内存。不需要再使用的内存称之为垃圾 (*garbage*)，而清除垃圾的动作称为垃圾回收 (*garbage collection*或 GC)。

垃圾是从哪来的？让我们来创造一些垃圾：

```
> (setf lst (list 'a 'b 'c))
(A B C)
> (setf lst nil)
NIL
```

一开始我们调用 `list`， `list` 调用 `cons`，在堆上配置了一个新的 *Cons* 对象。在这个情况我们创出三个 *Cons* 对象。之后当我们把 `lst` 设为 `nil`，我们没有任何方法可以再存取 `lst`，列表 `(a b c)`。 [5]

因为我们没有任何方法再存取列表，它也有可能是不存在的。我们不再有任何方式可以存取的对象叫做垃圾。系统可以安全地重新使用这三个 *Cons* 核。

这种管理内存的方法，给程序員带来极大的便利性。你不用显式地配置 (allocate)或释放 (dellocate)内存。这也表示了你不需要处理因为这么做而可能产生的臭虫。内存泄漏 (Memory leaks)以及迷途指针 (dangling pointer)在 Lisp 中根本不可能发生。

但是像任何的科技进步，如果你不小心的话，自动内存管理也有可能对你不利。使用及回收堆所带来的代价有时可以看做 `cons` 的代价。这是有理的，除非一个程序从来不丢弃任何东西，不然所有的 *Cons* 对象终究要变成垃圾。 Consing 的问题是，配置空间与清除内存，与程序的常规运作比起来花费昂贵。近期的研究提出了大幅改善内存回收的演算法，但是 consing 总是需要代价的，在某些现有的 Lisp 系统中，代价是昂贵的。

除非你很小心，不然很容易写出过度显式创建 cons 对象的程序。举例来说， `remove` 需要复制所有的 `cons` 核，直到最后一个元素从列表中移除。你可以借由使用破坏性的函数避免某些 consing，它试着去重用列表的结构作为参数传给它们。破坏性函数会在 12.4 节讨论。

当写出 `cons` 很多的程序是如此简单时，我们还是可以写出不使用 `cons` 的程序。典型的方法是写出一个纯函数风格，使用很多列表的第一版程序。当程序进化时，你可以在代码的关键部分使用破坏性函数以及/或别种数据结构。但这很难给出通用的建议，因为有些 Lisp 实现，内存管理处理得相当好，以致于使用 `cons` 有时比不使用 `cons` 还快。这整个议题在 13.4 做更进一步的细部讨论。

无论如何 consing 在原型跟实验时是好的。而且如果你利用了列表给你带来的灵活性，你有较高的可能写出后期可存活下来的程序。

Chapter 3 总结 (Summary)

1. 一个 *Cons* 是一个含两部分的数据结构。列表用链结在一起的 *Cons* 组成。
2. 判断式 `equal` 比 `eql` 来得不严谨。基本上，如果传入参数印出来的值一样时，返回真。
3. 所有 Lisp 对象表现得像指针。你永远不需要显式操作指针。
4. 你可以使用 `copy-list` 复制列表，并使用 `append` 来连接它们的元素。
5. 游程编码是一个餐厅中使用的简单压缩演算法。
6. Common Lisp 有由 `car` 与 `cdr` 定义的多种存取函数。
7. 映射函数将函数应用至逐项的元素，或逐项的列表尾端。
8. 嵌套列表的操作有时被考虑为树的操作。
9. 要判断一个递归函数是否正确，你只需要考虑是否包含了所有情况。
10. 列表可以用来表示集合。数个内置函数把列表当作集合。
11. 关键字参数是选择性的，并不是由位置所识别，是用符号前面的特殊标签来识别。
12. 列表是序列的子类型。 Common Lisp 有大量的序列函数。
13. 一个不是正规列表的 *Cons* 称之为点状列表。
14. 用 cons 对象作为元素的列表，可以拿来表示对应关系。这样的列表称为关联列表 (assoc-lists)。
15. 自动内存管理拯救你处理内存配置的烦恼，但制造过多的垃圾会使程序变慢。

Chapter 3 习题 (Exercises)

1. 用箱子表示法表示以下列表：

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

2. 写一个保留原本列表中元素顺序的 `union` 版本：

```
> (new-union '(a b c) '(b a d))
(A B C D)
```

3. 定义一个函数，接受一个列表并返回一个列表，指出相等元素出现的次数，并由最常见至最少见的排序：

```
> (occurrences '(a b a d a c d c a))
((A . 4) (C . 2) (D . 2) (B . 1))
```

4. 为什么 `(member '(a) '((a) (b)))` 返回 nil?
5. 假设函数 `pos+` 接受一个列表并返回把每个元素加上自己的位置的列表：

```
> (pos+ '(7 5 1 4))
(7 6 3 7)
```

使用 (a) 递归 (b) 迭代 (c) `mapcar` 来定义这个函数。

6. 经过好几年的审议，政府委员会决定列表应该由 `cdr` 指向第一个元素，而 `car` 指向剩下的列表。定义符合政府版本的以下函数：

```
(a) cons
(b) list
(c) length (for lists)
(d) member (for lists; no keywords)
```

勘误：要解决 3.6 (b)，你需要使用到 6.3 节的参数 `&rest` 。

7. 修改图 3.6 的程序，使它使用更少 cons 核。（提示：使用点状列表）
8. 定义一个函数，接受一个列表并用点状表示法印出：

```
> (showdots '(a b c))
(A . (B . (C . NIL)))
NIL
```

9. 写一个程序来找到 3.15 节里表示的网络中，最长有限的路径 (不重复)。网络可能包含循环。

脚注

[3] 这个叙述有点误导，因为只要是对任何东西都不返回 nil 的函数，都不是正规列表。如果给定一个环状 cdr 列表(cdr-circular list)，它会无法终止。环状列表在 12.7 节 讨论。

[4] 12.3 小节会展示更有效率的队列实现方式。

[5] 事实上，我们有一种方式来存取列表。全局变量 `*` , `**` , 以及 `***` 总是设定为最后三个顶层所返回的值。这些变量在除错的时候很有用。

第四章：特殊数据结构

在之前的章节里，我们讨论了列表，Lisp 最多功能的数据结构。本章将演示如何使用 Lisp 其它的数据结构：数组（包含向量与字符串），结构以及哈希表。它们或许不像列表这么灵活，但存取速度更快并使用了更少空间。

Common Lisp 还有另一种数据结构：实例（instance）。实例将在 11 章讨论，讲述 CLOS。

4.1 数组 (Array)

在 Common Lisp 里，你可以调用 `make-array` 来构造一个数组，第一个实参为一个指定数组维度的列表。要构造一个 `2 x 3` 的数组，我们可以：

```
> (setf arr (make-array '(2 3) :initial-element nil))
#<Simple-Array T (2 3) BFC4FE>
```

Common Lisp 的数组至少可以达到七个维度，每个维度至少可以容纳 1023 个元素。

`:initial-element` 实参是选择性的。如果有提供这个实参，整个数组会用这个值作为初始值。若试著取出未初始化的数组内的元素，其结果为未定义（undefined）。

用 `aref` 取出数组内的元素。与 Common Lisp 的存取函数一样， `aref` 是零索引的（zero-indexed）：

```
> (aref arr 0 0)
NIL
```

要替换数组的某个元素，我们使用 `setf` 与 `aref`：

```
> (setf (aref arr 0 0) 'b)
B
> (aref arr 0 0)
B
```

要表示字面常量的数组（literal array），使用 `#na` 语法，其中 `n` 是数组的维度。举例来说，我们可以这样表示 `arr` 这个数组：

```
#2a((b nil nil) (nil nil nil))
```

如果全局变量 *print-array* 为真，则数组会用以下形式来显示：

```
> (setf *print-array* t)
T
> arr
#2A((B NIL NIL) (NIL NIL NIL))
```

如果我们只想要一维的数组，你可以给 make-array 第一个实参传一个整数，而不是一个列表：

```
> (setf vec (make-array 4 :initial-element nil))
#(NIL NIL NIL NIL)
```

一维数组又称为向量（*vector*）。你可以通过调用 vector 来一步骤构造及填满向量，向量的元素可以是任何类型：

```
> (vector "a" 'b 3)
#("a" b 3)
```

字面常量的数组可以表示成 #na ，字面常量的向量也可以用这种语法表达。

可以用 aref 来存取向量，但有一个更快的函数叫做 svref ，专门用来存取向量。

```
> (svref vec 0)
NIL
```

在 svref 内的 “sv” 代表“简单向量”（“simple vector”），所有的向量缺省是简单向量。[1]

4.2 示例：二叉搜索 (Example: Binary Search)

作为一个示例，这小节演示如何写一个在排序好的向量里搜索对象的函数。如果我们知道一个向量是排序好的，我们可以比（65页） find 做的更好， find 必须依序检查每一个元素。我们可以直接跳到向量中间开始找。如果中间的元素是我们要找的对象，搜索完毕。要不然我们持续往左半部或往右半部搜索，取决于对象是小于或大于中间的元素。

图 4.1 包含了一个这么工作的函数。其实这两个函数： bin-search 设置初始范围及发送控制信号给 finder ， finder 寻找向量 vec 内 obj 是否介于 start 及 end 之间。

```
(defun bin-search (obj vec)
  (let ((len (length vec)))
    (and (not (zerop len))
```

```
          (finder obj vec 0 (- len 1)))))

(defun finder (obj vec start end)
  (let ((range (- end start)))
    (if (zerop range)
        (if (eql obj (aref vec start))
            obj
            nil)
        (let ((mid (+ start (round (/ range 2)))))
          (let ((obj2 (aref vec mid)))
            (if (< obj obj2)
                (finder obj vec start (- mid 1))
                (if (> obj obj2)
                    (finder obj vec (+ mid 1) end)
                    obj)))))))
```

图 4.1: 搜索一个排序好的向量

如果要找的 range 缩小至一个元素，而如果这个元素是 obj 的话，则 finder 直接返回这个元素，反之返回 nil 。如果 range 大于 1 ，我们設置 middle (round 返回离实参最近的整数) 为 obj2 。如果 obj 小于 obj2 ，则递归地往向量的左半部寻找。如果 obj 大于 obj2 ，则递归地往向量的右半部寻找。剩下的一个选择是 obj=obj2 ，在这个情况我们找到要找的元素，直接返回这个元素。

如果我们插入下面这行至 finder 的起始处：

```
(format t "~A~%" (subseq vec start (+ end 1)))
```

我们可以观察被搜索的元素的数量，是每一步往左减半的：

```
> (bin-search 3 #(0 1 2 3 4 5 6 7 8 9))
#(0 1 2 3 4 5 6 7 8 9)
#(0 1 2 3)
#(3)
3
```

4.3 字符与字符串 (Strings and Characters)

字符串是字符组成的向量。我们用一系列由双引号包住的字符，来表示一个字符串常量，而字符 c 用 #\c 表示。

每个字符都有一个相关的整数 —— 通常是 ASCII 码，但不一定是。在多数的 Lisp 实现里，函数 char-code 返回与字符相关的数字，而 code-char 返回与数字相关的字符。

字符比较函数 char< （小于）， char<= （小于等于)， char= （等于)， char>= （大于

等于)， char> （大于)，以及 char/= （不同)。他们的工作方式和 146 页（译注 9.3 节)比较数字用的操作符一样。

```
> (sort "elbow" #'char<)
"below"
```

由于字符串是字符向量，序列与数组的函数都可以用在字符串。你可以用 aref 来取出元素，举例来说，

```
> (aref "abc" 1)
#\b
```

但针对字符串可以使用更快的 char 函数：

```
> (char "abc" 1)
#\b
```

可以使用 setf 搭配 char （或 aref ）来替换字符串的元素：

```
> (let ((str (copy-seq "Merlin")))
   (setf (char str 3) #\k)
   str)
```

如果你想要比较两个字符串，你可以使用通用的 equal 函数，但还有一个比较函数，是忽略字母大小写的 string-equal：

```
> (equal "fred" "fred")
T
> (equal "fred" "Fred")
NIL
>(string-equal "fred" "Fred")
T
```

Common Lisp 提供大量的操控、比较字符串的函数。收录在附录 D，从 364 页开始。

有许多方式可以创建字符串。最普遍的方式是使用 format 。将第一个参数设为 nil 来调用 format ，使它返回一个原本会印出来的字符串：

```
> (format nil "~A or ~A" "truth" "dare")
"truth or dare"
```

但若你只想把数个字符串连结起来，你可以使用 concatenate ，它接受一个特定类型的符号，加上一个或多个序列：

```
> (concatenate 'string "not " "to worry")
"not to worry"
```

4.4 序列 (Sequences)

在 Common Lisp 里，序列类型包含了列表与向量（因此也包含了字符串）。有些用在列表的函数，实际上是序列函数，包括 `remove`、 `length`、 `subseq`、 `reverse`、 `sort`、 `every` 以及 `some`。所以 46 页（译注 3.11 小节的 `mirror?` 函数）我们所写的函数，也可以用在其他种类的序列上：

```
> (mirror? "abba")
T
```

我们已经看过四种用来取出序列元素的函数： 给列表使用的 `nth`， 给向量使用的 `aref` 及 `svref`，以及给字符串使用的 `char`。 Common Lisp 也提供了通用的 `elt`，对任何种类的序列都有效：

```
> (elt '(a b c) 1)
B
```

针对特定类型的序列，特定的存取函数会比较快，所以使用 `elt` 是没有意义的，除非在代码当中，有需要支持通用序列的地方。

使用 `elt`，我们可以写一个针对向量来说更有效率的 `mirror?` 版本：

```
(defun mirror? (s)
  (let ((len (length s)))
    (and (evenp len)
         (do ((forward 0 (+ forward 1))
              (back (- len 1) (- back 1)))
             ((or (> forward back)
                  (not (eql (elt s forward)
                            (elt s back))))
              (> forward back))))))
```

这个版本也可用在列表，但这个实现更适合给向量使用。频繁的对列表调用 `elt` 的代价是昂贵的，因为列表仅允许顺序存取。而向量允许随机存取，从任何元素来存取每一个元素都是廉价的。

许多序列函数接受一个或多个，由下表所列的标准关键字参数：

参数	用途	缺省值
:key	应用至每个元素的函数	identity

:test	作来比较的函数	eql
:from-end	若为真，反向工作。	nil
:start	起始位置	0
:end	若有给定，结束位置。	nil

一个接受所有关键字参数的函数是 `position` ，返回序列中一个元素的位置，未找到元素时则返回 `nil` 。我们使用 `position` 来演示关键字参数所扮演的角色。

```
> (position #\a "fantasia")
1
> (position #\a "fantasia" :start 3 :end 5)
4
```

第二个例子我们要找在第四个与第六个字符间，第一个 `a` 所出现的位置。 `:start` 关键字参数是第一个被考虑的元素位置，缺省是序列的第一个元素。 `:end` 关键字参数，如果有给的话，是第一个不被考虑的元素位置。

如果我们给入 `:from-end` 关键字参数，

```
> (position #\a "fantasia" :from-end t)
7
```

我们得到最靠近结尾的 `a` 的位置。但位置是像平常那样计算；而不是从尾端算回来的距离。

`:key` 关键字参数是序列中每个元素在被考虑之前，应用至元素上的函数。如果我们说，

```
> (position 'a '((c d) (a b)) :key #'car)
1
```

那么我们要找的是，元素的 `car` 部分是符号 `a` 的第一个元素。

`:test` 关键字参数接受需要两个实参的函数，并定义了怎样是一个成功的匹配。缺省函数为 `eql` 。如果你想要匹配一个列表，你也许想使用 `equal` 来取代：

```
> (position '(a b) '((a b) (c d)))
NIL
> (position '(a b) '((a b) (c d)) :test #'equal)
0
```

`:test` 关键字参数可以是任何接受两个实参的函数。举例来说，给定 `<` ，我们可以询问第一个使第一个参数比它小的元素位置：

```
> (position 3 '(1 0 7 5) :test #'<)
2
```

使用 subseq 与 position，我们可以写出分开序列的函数。举例来说，这个函数

```
(defun second-word (str)
  (let ((p1 (+ (position #\  str) 1)))
    (subseq str p1 (position #\  str :start p1))))
```

返回字符串中第一个单字空格后的第二个单字：

```
> (second-word "Form follows function")
"follows"
```

要找到满足谓词的元素，其中谓词接受一个实参，我们使用 position-if 。它接受一个函数与序列，并返回第一个满足此函数的元素：

```
> (position-if #'oddp '(2 3 4 5))
1
```

position-if 接受除了 :test 之外的所有关键字参数。

有许多相似的函数，如给序列使用的 member 与 member-if 。分别是， find （接受全部关键字参数）与 find-if （接受除了 :test 之外的所有关键字参数）：

```
> (find #\a "cat")
#\a

> (find-if #'characterp "ham")
#\h
```

不同于 member 与 member-if ，它们仅返回要寻找的对象。

通常一个 find-if 的调用，如果解读为 find 搭配一个 :key 关键字参数的话，会显得更清楚。举例来说，表达式

```
(find-if #'(lambda (x)
             (eql (car x) 'complete))
         lst)
```

可以更好的解读为

```
(find 'complete lst :key #'car)
```

函数 remove （22 页）以及 remove-if 通常都可以用在序列。它们跟 find 与 find-if

是一样的关系。另一个相关的函数是 `remove-duplicates` ，仅保留序列中每个元素的最后一次出现。

```
> (remove-duplicates "abracadabra")
"cdbra"
```

这个函数接受前表所列的所有关键字参数。

函数 `reduce` 用来把序列压缩成一个值。它至少接受两个参数，一个函数与序列。函数必须是接受两个实参的函数。在最简单的情况下，一开始函数用序列前两个元素作为实参来调用，之后接续的元素作为下次调用的第二个实参，而上次返回的值作为下次调用的第一个实参。最后调用最终返回的值作为 `reduce` 整个函数的返回值。也就是说像是这样的表达式：

```
(reduce #'fn '(a b c d))
```

等同于

```
(fn (fn (fn 'a 'b) 'c) 'd)
```

我们可以使用 `reduce` 来扩充只接受两个参数的函数。举例来说，要得到三个或多个列表的交集(intersection)，我们可以：

```
> (reduce #'intersection '((b r a d 's) (b a d) (c a t)))
(A)
```

4.5 示例：解析日期 (Example: Parsing Dates)

作为序列操作的示例，本节演示了如何写程序来解析日期。我们将编写一个程序，可以接受像是 “16 Aug 1980” 的字符串，然后返回一个表示日、月、年的整数列表。

```
(defun tokens (str test start)
  (let ((p1 (position-if test str :start start)))
    (if p1
        (let ((p2 (position-if #'(lambda (c)
                                   (not (funcall test c)))
                               str :start p1)))
          (cons (subseq str p1 p2)
                (if p2
                    (tokens str test p2)
                    nil)))
        nil)))

(defun constituent (c)
```

```
  (and (graphic-char-p c)
       (not (char= c #\ ))))
```

图 4.2 辨别符号 (token)

图 4.2 里包含了某些在这个应用里所需的通用解析函数。第一个函数 `tokens` ，用来从字符串中取出语元 （token）。给定一个字符串及测试函数，满足测试函数的字符组成子字符串，子字符串再组成列表返回。举例来说，如果测试函数是对字母返回真的 `alpha-char-p` 函数，我们得到：

```
> (tokens "ab12 3cde.f" #'alpha-char-p 0)
("ab" "cde" "f")
```

所有不满足此函数的字符被视为空白 —— 他们是语元的分隔符，但永远不是语元的一部分。

函数 `constituent` 被定义成用来作为 `tokens` 的实参。

在 Common Lisp 里，图形字符是我们可见的字符，加上空白字符。所以如果我们用 `constituent` 作为测试函数时，

```
> (tokens "ab12 3cde.f gh" #'constituent 0)
("ab12" "3cde.f" "gh")
```

则语元将会由空白区分出来。

图 4.3 包含了特别为解析日期打造的函数。函数 `parse-date` 接受一个特别形式组成的日期，并返回代表这个日期的整数列表：

```
> (parse-date "16 Aug 1980")
(16 8 1980)
```

```
(defun parse-date (str)
  (let ((toks (tokens str #'constituent 0)))
    (list (parse-integer (first toks))
          (parse-month (second toks))
          (parse-integer (third toks)))))

(defconstant month-names
  #("jan" "feb" "mar" "apr" "may" "jun"
    "jul" "aug" "sep" "oct" "nov" "dec"))

(defun parse-month (str)
  (let ((p (position str month-names
                     :test #'string-equal)))
    (if p
```

```
        (+ p 1)
        nil)))
```

图 4.3 解析日期的函数

parse-date 使用 tokens 来解析日期字符串，接著调用 parse-month 及 parse-integer 来转译年、月、日。要找到月份，调用 parse-month ，由于使用的是 string-equal 来匹配月份的名字，所以输入可以不分大小写。要找到年和日，调用内置的 parse-integer， parse-integer 接受一个字符串并返回对应的整数。

如果需要自己写程序来解析整数，也许可以这么写：

```
(defun read-integer (str)
  (if (every #'digit-char-p str)
      (let ((accum 0))
        (dotimes (pos (length str))
          (setf accum (+ (* accum 10)
                         (digit-char-p (char str pos)))))
        accum)
    nil))
```

这个定义演示了在 Common Lisp 中，字符是如何转成数字的 —— 函数 digit-char-p 不仅测试字符是否为数字，同时返回了对应的整数。

4.6 结构 (Structures)

结构可以想成是豪华版的向量。假设你要写一个程序来追踪长方体。你可能会想用三个向量元素来表示长方体：高度、宽度及深度。与其使用原本的 svref ，不如定义像是下面这样的抽象，程序会变得更容易阅读，

```
(defun block-height (b) (svref b 0))
```

而结构可以想成是，这些函数通通都替你定义好了的向量。

要想定义结构，使用 defstruct 。在最简单的情况下，只要给出结构及字段的名字便可以了：

```
(defstruct point
  x
  y)
```

这里定义了一个 point 结构，具有两个字段 x 与 y 。同时隐式地定义了 make-point 、 point-p 、 copy-point 、 point-x 及 point-y 函数。

2.3 节提过， Lisp 程序可以写出 Lisp 程序。这是目前所见的明显例子之一。当你调用 defstruct 时，它自动生成了其它几个函数的定义。有了宏以后，你将可以自己来办到同样的事情（如果需要的话，你甚至可以自己写出 defstruct ）。

每一个 make-point 的调用，会返回一个新的 point 。可以通过给予对应的关键字参数，来指定单一字段的值：

```
(setf p (make-point :x 0 :y 0))
#S(POINT X 0 Y 0)
```

存取 point 字段的函数不仅被定义成可取出数值，也可以搭配 setf 一起使用。

```
> (point-x p)
0
> (setf (point-y p) 2)
2
> p
#S(POINT X 0 Y 2)
```

定义结构也定义了以结构为名的类型。每个点的类型层级会是，类型 point ，接著是类型 structure ，再来是类型 atom ，最后是 t 类型。所以使用 point-p 来测试某个东西是不是一个点时，也可以使用通用性的函数，像是 typep 来测试。

```
> (point-p p)
T
> (typep p 'point)
T
```

我们可以在本来的定义中，附上一个列表，含有字段名及缺省表达式，来指定结构字段的缺省值。

```
(defstruct polemic
  (type (progn
          (format t "What kind of polemic was it? ")
          (read)))
  (effect nil))
```

如果 make-polemic 调用没有给字段指定初始值，则字段会被设成缺省表达式的值：

```
> (make-polemic)
What kind of polemic was it? scathing
#S(POLEMIC :TYPE SCATHING :EFFECT NIL)
```

结构显示的方式也可以控制，以及结构自动产生的存取函数的字首。以下是做了前述两件事的 point 定义：

```
(defstruct (point (:conc-name p)
                  (:print-function print-point))
  (x 0)
  (y 0))

(defun print-point (p stream depth)
  (format stream "#<~A, ~A>" (px p) (py p)))
```

`:conc-name` 关键字参数指定了要放在字段前面的名字，并用这个名字来生成存取函数。预设是 `point-` ；现在变成只有 `p` 。不使用缺省的方式使代码的可读性些微降低了，只有在需要常常用到这些存取函数时，你才会想取个短点的名字。

`:print-function` 是在需要显示结构出来看时，指定用来打印结构的函数 —— 需要显示的情况比如，要在顶层显示时。这个函数需要接受三个实参：要被印出的结构，在哪里被印出，第三个参数通常可以忽略。 [2] 我们会在 7.1 节讨论流（stream）。现在来说，只要知道流可以作为参数传给 `format` 就好了。

函数 `print-point` 会用缩写的形式来显示点：

```
> (make-point)
#<0,0>
```

4.7 示例：二叉搜索树 (Example: Binary Search Tree)

由于 `sort` 本身系统就有了，极少需要在 Common Lisp 里编写排序程序。本节将演示如何解决一个与此相关的问题，这个问题尚未有现成的解决方案：维护一个已排序的对象集合。本节的代码会把对象存在二叉搜索树里（ *binary search tree* ）或称作 BST。当二叉搜索树平衡时，允许我们可以在与时间成 `log n` 比例的时间内，来寻找、添加或是删除元素，其中 `n` 是集合的大小。

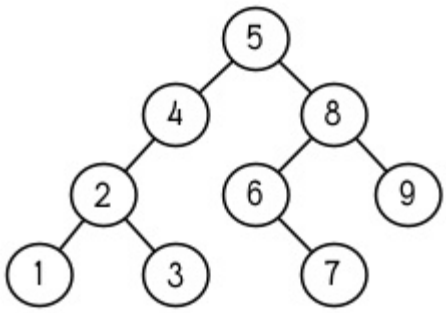

图 4.4: 二叉搜索树

二叉搜索树是一种二叉树，给定某个排序函数，比如 `<` ，每个元素的左子树都 `<` 该元素，而该元素 `<` 其右子树。图 4.4 展示了根据 `<` 排序的二叉树。

图 4.5 包含了二叉搜索树的插入与寻找的函数。基本的数据结构会是 node （节点），节点有三个部分：一个字段表示存在该节点的对象，以及各一个字段表示节点的左子树及右子树。可以把节点想成是有一个 car 和两个 cdr 的一个 cons 核（cons cell）。

```
(defstruct (node (:print-function
                  (lambda (n s d)
                    (format s "#<~A>" (node-elt n)))))
  elt (l nil) (r nil))

(defun bst-insert (obj bst <)
  (if (null bst)
      (make-node :elt obj)
      (let ((elt (node-elt bst)))
        (if (eql obj elt)
            bst
            (if (funcall < obj elt)
                (make-node
                   :elt elt
                   :l (bst-insert obj (node-l bst) <)
                   :r (node-r bst))
                (make-node
                   :elt elt
                   :r (bst-insert obj (node-r bst) <)
                   :l (node-l bst)))))))

(defun bst-find (obj bst <)
  (if (null bst)
      nil
      (let ((elt (node-elt bst)))
        (if (eql obj elt)
            bst
            (if (funcall < obj elt)
                (bst-find obj (node-l bst) <)
                (bst-find obj (node-r bst) <))))))

(defun bst-min (bst)
  (and bst
       (or (bst-min (node-l bst)) bst)))

(defun bst-max (bst)
  (and bst
       (or (bst-max (node-r bst)) bst)))
```

图 4.5 二叉搜索树：查询与插入

一棵二叉搜索树可以是 nil 或是一个左子、右子树都是二叉搜索树的节点。如同列表可由连续调用 cons 来构造，二叉搜索树将可以通过连续调用 bst-insert 来构造。这个函数接受一个对象，一棵二叉搜索树及一个排序函数，并返回将对象插入的二叉搜索树。和 cons 函数一样， bst-insert 不改动做为第二个实参所传入的二叉搜索树。以下是如何使用这个函数来构造一棵叉搜索树：

```
> (setf nums nil)
NIL
> (dolist (x '(5 8 4 2 1 9 6 7 3))
    (setf nums (bst-insert x nums #'<)))
NIL
```

图 4.4 显示了此时 nums 的结构所对应的树。

我们可以使用 bst-find 来找到二叉搜索树中的对象，它与 bst-insert 接受同样的参数。先前叙述所提到的 node 结构，它像是一个具有两个 cdr 的 cons 核。如果我们把 16 页的 our-member 拿来与 bst-find 比较的话，这样的类比更加明确。

与 member 相同， bst-find 不仅返回要寻找的元素，也返回了用寻找元素做为根节点的子树：

```
> (bst-find 12 nums #'<)
NIL
> (bst-find 4 nums #'<)
#<4>
```

这使我们可以区分出无法找到某个值，以及成功找到 nil 的情况。

要找到二叉搜索树的最小及最大的元素是很简单的。要找到最小的，我们沿着左子树的路径走，如同 bst-min 所做的。要找到最大的，沿着右子树的路径走，如同 bst-max 所做的：

```
> (bst-min nums)
#<1>
> (bst-max nums)
#<9>
```

要从二叉搜索树里移除元素一样很快，但需要更多代码。图 4.6 演示了如何从二叉搜索树里移除元素。

```
(defun bst-remove (obj bst <)
  (if (null bst)
      nil
      (let ((elt (node-elt bst)))
        (if (eql obj elt)
            (percolate bst)
            (if (funcall < obj elt)
                (make-node
                   :elt elt
                   :l (bst-remove obj (node-l bst) <)
                   :r (node-r bst))
                (make-node
                   :elt elt
```

```
                    :r (bst-remove obj (node-r bst) <)
                    :l (node-l bst)))))))

(defun percolate (bst)
  (cond ((null (node-l bst))
         (if (null (node-r bst))
             nil
             (rperc bst)))
        ((null (node-r bst)) (lperc bst))
        (t (if (zerop (random 2))
               (lperc bst)
               (rperc bst)))))

(defun rperc (bst)
  (make-node :elt (node-elt (node-r bst))
             :l (node-l bst)
             :r (percolate (node-r bst))))
```

图 4.6 二叉搜索树：移除

勘误：此版 `bst-remove` 的定义已被汇报是坏掉的，请参考 这里 [https://gist.github.com/2868263] 获得修复版。

函数 `bst-remove` 接受一个对象，一棵二叉搜索树以及排序函数，并返回一棵与本来的二叉搜索树相同的树，但不包含那个要移除的对象。和 `remove` 一样，它不改动做为第二个实参所传入的二叉搜索树：

```
> (setf nums (bst-remove 2 nums #'<))
#<5>
> (bst-find 2 nums #'<)
NIL
```

此时 `nums` 的结构应该如图 4.7 所示。（另一个可能性是 `1` 取代了 `2` 的位置。）

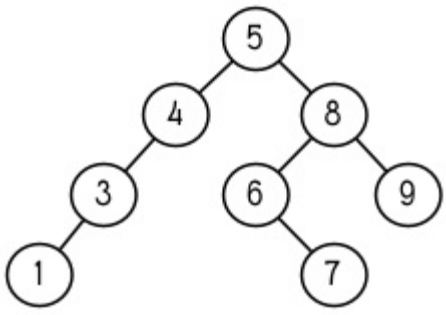

图 4.7: 二叉搜索树

移除需要做更多工作，因为从内部节点移除一个对象时，会留下一个空缺，需要由其中一个孩子来填补。这是 `percolate` 函数的用途。当它替换一个二叉搜索树的树根

（topmost element）时，会找其中一个孩子来替换，并用此孩子的孩子来填补，如此这般一直递归下去。

为了要保持树的平衡，如果有两个孩子时， `perlocate` 随机择一替换。表达式 `(random 2)` 会返回 `0` 或 `1` ，所以 `(zerop (random 2))` 会返回真或假。

```
(defun bst-traverse (fn bst)
  (when bst
    (bst-traverse fn (node-l bst))
    (funcall fn (node-elt bst))
    (bst-traverse fn (node-r bst))))
```

图 4.8 二叉搜索树：遍历

一旦我们把一个对象集合插入至二叉搜索树时，中序遍历会将它们由小至大排序。这是图 4.8 中， `bst-traverse` 函数的用途：

```
> (bst-traverse #'princ nums)
13456789
NIL
```

（函数 `princ` 仅显示单一对象）

本节所给出的代码，提供了一个二叉搜索树实现的脚手架。你可能想根据应用需求，来充实这个脚手架。举例来说，这里所给出的代码每个节点只有一个 `elt` 字段；在许多应用里，有两个字段会更有意义， `key` 与 `value` 。本章的这个版本把二叉搜索树视为集合看待，从这个角度看，重复的插入是被忽略的。但是代码可以很简单地改动，来处理重复的元素。

二叉搜索树不仅是维护一个已排序对象的集合的方法。他们是否是最好的方法，取决于你的应用。一般来说，二叉搜索树最适合用在插入与删除是均匀分布的情况。有一件二叉搜索树不擅长的事，就是用来维护优先队列（priority queues）。在一个优先队列里，插入也许是均匀分布的，但移除总是在一个另一端。这会导致一个二叉搜索树变得不平衡，而我们期望的复杂度是 `O(log(n))` 插入与移除操作，将会变成 `O(n)` 。如果用二叉搜索树来表示一个优先队列，也可以使用一般的列表，因为二叉搜索树最终会作用的像是个列表。

4.8 哈希表 (Hash Table)

第三章演示过列表可以用来表示集合（sets）与映射（mappings）。但当列表的长度大幅上升时（或是 10 个元素），使用哈希表的速度比较快。你通过调用 `make-hash-table` 来构造一个哈希表，它不需要传入参数：

```
> (setf ht (make-hash-table))
#<Hash-Table BF0A96>
```

和函数一样，哈希表总是用 `#<...>` 的形式来显示。

一个哈希表，与一个关联列表类似，是一种表达对应关系的方式。要取出与给定键值有关的数值，我们调用 `gethash` 并传入一个键值与哈希表。预设情况下，如果没有与这个键值相关的数值， `gethash` 会返回 `nil` 。

```
> (gethash 'color ht)
NIL
NIL
```

在这里我们首次看到 Common Lisp 最突出的特色之一：一个表达式竟然可以返回多个数值。函数 `gethash` 返回两个数值。第一个值是与键值有关的数值，第二个值说明了哈希表是否含有任何用此键值来储存的数值。由于第二个值是 `nil` ，我们知道第一个 `nil` 是缺省的返回值，而不是因为 `nil` 是与 `color` 有关的数值。

大部分的实现会在顶层显示一个函数调用的所有返回值，但仅期待一个返回值的代码，只会收到第一个返回值。 5.5 节会说明，代码如何接收多个返回值。

要把数值与键值作关联，使用 `gethash` 搭配 `setf` ：

```
> (setf (gethash 'color ht) 'red)
RED
```

现在如果我们再次调用 `gethash` ，我们会得到我们刚插入的值：

```
> (gethash 'color ht)
RED
T
```

第二个返回值证明，我们取得了一个真正储存的对象，而不是预设值。

存在哈希表的对象或键值可以是任何类型。举例来说，如果我们要保留函数的某种讯息，我们可以使用哈希表，用函数作为键值，字符串作为词条（entry）：

```
> (setf bugs (make-hash-table))
#<Hash-Table BF4C36>
> (push "Doesn't take keyword arguments."
        (gethash #'our-member bugs))
("Doesn't take keyword arguments.")
```

由于 `gethash` 缺省返回 `nil` ，而 `push` 是 `setf` 的缩写，可以轻松的给哈希表新添一个词

条。（有困扰的 our-member 定义在 16 页。）

可以用哈希表来取代用列表表示集合。当集合变大时，哈希表的查询与移除会来得比较快。要新增一个成员到用哈希表所表示的集合，把 gethash 用 setf 设成 t：

```
> (setf fruit (make-hash-table))
#<Hash-Table BFDE76>
> (setf (gethash 'apricot fruit) t)
T
```

然后要测试是否为成员，你只要调用：

```
> (gethash 'apricot fruit)
T
T
```

由于 gethash 缺省返回真，一个新创的哈希表，会很方便地是一个空集合。

要从集合中移除一个对象，你可以调用 remhash，它从一个哈希表中移除一个词条：

```
> (remhash 'apricot fruit)
T
```

返回值说明了是否有词条被移除；在这个情况里，有。

哈希表有一个迭代函数： maphash ，它接受两个实参，接受两个参数的函数以及哈希表。该函数会被每个键值对调用，没有特定的顺序：

```
> (setf (gethash 'shape ht) 'spherical
        (gethash 'size ht) 'giant)
GIANT

> (maphash #'(lambda (k v)
               (format t "~A = ~A~%" k v))
           ht)
SHAPE = SPHERICAL
SIZE = GIANT
COLOR = RED
NIL
```

maphash 总是返回 nil，但你可以通过传入一个会累积数值的函数，把哈希表的词条存在列表里。

哈希表可以容纳任何数量的元素，但当哈希表空间用完时，它们会被扩张。如果你想要确保一个哈希表，从特定数量的元素空间大小开始时，可以给 make-hash-table 一个选择性的 :size 关键字参数。做这件事情有两个理由：因为你知道哈希表会变得很大，你

想要避免扩张它；或是因为你知道哈希表会是很小，你不想要浪费内存。 `:size` 参数不仅指定了哈希表的空间，也指定了元素的数量。平均来说，在被扩张前所能够容纳的数量。所以

```
(make-hash-table :size 5)
```

会返回一个预期存放五个元素的哈希表。

和任何牵涉到查询的结构一样，哈希表一定有某种比较键值的概念。预设是使用 `eql`，但你可以提供一个额外的关键字参数 `:test` 来告诉哈希表要使用 `eq`， `equal`，还是 `equalp`：

```
> (setf writers (make-hash-table :test #'equal))
#<Hash-Table C005E6>
> (setf (gethash '(ralph waldo emerson) writers) t)
T
```

这是一个让哈希表变得有效率的取舍之一。有了列表，我们可以指定 `member` 为判断相等性的谓词。有了哈希表，我们可以预先决定，并在哈希表构造时指定它。

大多数 Lisp 编程的取舍（或是生活，就此而论）都有这种特质。起初你想要事情进行得流畅，甚至赔上效率的代价。之后当代码变得沉重时，你牺牲了弹性来换取速度。

Chapter 4 总结 (Summary)

1. Common Lisp 支持至少 7 个维度的数组。一维数组称为向量。
2. 字符串是字符的向量。字符本身就是对象。
3. 序列包括了向量与列表。许多序列函数都接受标准的关键字参数。
4. 处理字符串的函数非常多，所以用 Lisp 来解析字符串是小菜一碟。
5. 调用 `defstruct` 定义了一个带有命名字段的结构。它是一个程序能写出程序的好例子。
6. 二叉搜索树见长于维护一个已排序的对象集合。
7. 哈希表提供了一个更有效率的方式来表示集合与映射 (mappings)。

Chapter 4 习题 (Exercises)

1. 定义一个函数，接受一个平方数组（square array，一个相同维度的数组 `(n n)`），并将它顺时针转 90 度。

```
> (quarter-turn #2A((a b) (c d)))
#2A((C A) (D B))
```

你会需要用到 361 页的 `array-dimensions` 。

2. 阅读 368 页的 `reduce` 说明，然后用它来定义：

```
(a) copy-list
(b) reverse（针对列表）
```

3. 定义一个结构来表示一棵树，其中每个节点包含某些数据及三个小孩。定义：

```
(a) 一个函数来复制这样的树（复制完的节点与本来的节点是不相等（ `eql` ）的）
(b) 一个函数，接受一个对象与这样的树，如果对象与树中各节点的其中一个字段相等时，返回真。
```

4. 定义一个函数，接受一棵二叉搜索树，并返回由此树元素所组成的，一个由大至小排序的列表。
5. 定义 `bst-adjoin` 。这个函数应与 `bst-insert` 接受相同的参数，但应该只在对象不等于任何树中对象时将其插入。

勘误：`bst-adjoin` 的功能与 `bst-insert` 一模一样。

6. 任何哈希表的内容可以由关联列表（assoc-list）来描述，其中列表的元素是 `(k . v)` 的形式，对应到哈希表中的每一个键值对。定义一个函数：

```
(a) 接受一个关联列表，并返回一个对应的哈希表。
(b) 接受一个哈希表，并返回一个对应的关联列表。
```

脚注

[1] 一个简单数组大小是不可调整、元素也不可替换的，并不含有填充指针（fill-pointer）。数组缺省是简单的。简单向量是个一维的简单数组，可以含有任何类型的元素。

[2] 在 ANSI Common Lisp 里，你可以给一个 `:print-object` 的关键字参数来取代，它只需要两个实参。也有一個宏叫做 `print-unreadable-object` ，能用则用，可以用 `#<...>` 的语法来显示对象。

第五章：控制流

2.2 节介绍过 Common Lisp 的求值规则，现在你应该很熟悉了。本章的操作符都有一个共同点，就是它们都违反了求值规则。这些操作符让你决定在程序当中何时要求值。如果普通的函数调用是 Lisp 程序的树叶的话，那这些操作符就是连结树叶的树枝。

5.1 区块 (Blocks)

Common Lisp 有三个构造区块（block）的基本操作符： `progn` 、 `block` 以及 `tagbody` 。我们已经看过 `progn` 了。在 `progn` 主体中的表达式会依序求值，并返回最后一个表达式的值：

```
> (progn
    (format t "a")
    (format t "b")
    (+ 11 12))
ab
23
```

由于只返回最后一个表达式的值，代表著使用 `progn` （或任何区块）涵盖了副作用。

一个 `block` 像是带有名字及紧急出口的 `progn` 。第一个实参应为符号。这成为了区块的名字。在主体中的任何地方，可以停止求值，并通过使用 `return-from` 指定区块的名字，来立即返回数值：

```
> (block head
    (format t "Here we go.")
    (return-from head 'idea)
    (format t "We'll never see this."))
Here we go.
IDEA
```

调用 `return-from` 允许你的程序，从代码的任何地方，突然但优雅地退出。第二个传给 `return-from` 的实参，用来作为以第一个实参为名的区块的返回值。在 `return-from` 之后的表达式不会被求值。

也有一个 `return` 宏，它把传入的参数当做封闭区块 `nil` 的返回值：

```
> (block nil
    (return 27))
27
```

许多接受一个表达式主体的 Common Lisp 操作符，皆隐含在一个叫做 nil 的区块里。比如，所有由 do 构造的迭代函数：

```
> (dolist (x '(a b c d e))
    (format t "~A " x)
    (if (eql x 'c)
        (return 'done)))
A B C
DONE
```

使用 defun 定义的函数主体，都隐含在一个与函数同名的区块，所以你可以：

```
(defun foo ()
  (return-from foo 27))
```

在一个显式或隐式的 block 外，不论是 return-from 或 return 都不会工作。

使用 return-from，我们可以写出一个更好的 read-integer 版本：

```
(defun read-integer (str)
  (let ((accum 0))
    (dotimes (pos (length str))
      (let ((i (digit-char-p (char str pos))))
        (if i
            (setf accum (+ (* accum 10) i))
            (return-from read-integer nil))))
    accum))
```

68 页的版本在构造整数之前，需检查所有的字符。现在两个步骤可以结合，因为如果遇到非数字的字符时，我们可以舍弃计算结果。出现在主体的原子（atom）被解读为标签（labels)；把这样的标签传给 go ，会把控制权交给标签后的表达式。以下是一个非常丑的程序片段，用来印出一至十的数字：

```
> (tagbody
    (setf x 0)
    top
      (setf x (+ x 1))
      (format t "~A " x)
      (if (< x 10) (go top)))
1 2 3 4 5 6 7 8 9 10
NIL
```

这个操作符主要用来实现其它的操作符，不是一般会用到的操作符。大多数迭代操作符都隐含在一个 tagbody，所以是可能可以在主体里（虽然很少想要）使用标签及 go 。

如何决定要使用哪一种区块建构子呢（block construct）？几乎任何时候，你会使用

`progn` 。如果你想要突然退出的话，使用 `block` 来取代。多数程序员永远不会显式地使用 `tagbody` 。

5.2 语境 (Context)

另一个我们用来区分表达式的操作符是 `let` 。它接受一个代码主体，但允许我们在主体内设置新变量：

```
> (let ((x 7)
        (y 2))
    (format t "Number")
    (+ x y))
Number
9
```

一个像是 `let` 的操作符，创造出一个新的词法语境（lexical context）。在这个语境里有两个新变量，然而在外部语境的变量也因此变得不可视了。

概念上说，一个 `let` 表达式等同于函数调用。在 2.14 节证明过，函数可以用名字来引用，也可以通过使用一个 lambda 表达式从字面上来引用。由于 lambda 表达式是函数的名字，我们可以像使用函数名那样，把 lambda 表达式作为函数调用的第一个实参：

```
> ((lambda (x) (+ x 1)) 3)
4
```

前述的 `let` 表达式，实际上等同于：

```
((lambda (x y)
   (format t "Number")
   (+ x y))
 7
 2)
```

如果有关于 `let` 的任何问题，应该是如何把责任交给 `lambda` ，因为进入一个 `let` 等同于执行一个函数调用。

这个模型清楚的告诉我们，由 `let` 创造的变量的值，不能依赖其它由同一个 `let` 所创造的变量。举例来说，如果我们试着：

```
(let ((x 2)
      (y (+ x 1)))
  (+ x y))
```

在 `(+ x 1)` 中的 `x` 不是前一行所设置的值，因为整个表达式等同于：

```
((lambda (x y) (+ x y)) 2
                        (+ x 1))
```

这里明显看到 `(+ x 1)` 作为实参传给函数，不能引用函数内的形参 `x` 。

所以如果你真的想要新变量的值，依赖同一个表达式所设立的另一个变量？在这个情况下，使用一个变形版本 `let*` ：

```
> (let* ((x 1)
         (y (+ x 1)))
    (+ x y))
3
```

一个 `let*` 功能上等同于一系列嵌套的 `let` 。这个特别的例子等同于：

```
(let ((x 1))
  (let ((y (+ x 1)))
    (+ x y)))
```

`let` 与 `let*` 将变量初始值都设为 `nil` 。 `nil` 为初始值的变量，不需要依附在列表内:

```
> (let (x y)
    (list x y))
(NIL NIL)
```

`destructuring-bind` 宏是通用化的 `let` 。其接受单一变量，一个模式 (pattern) —— 一个或多个变量所构成的树 —— 并将它们与某个实际的树所对应的部份做绑定。举例来说：

```
> (destructuring-bind (w (x y) . z) '(a (b c) d e)
    (list w x y z))
(A B C (D E))
```

若给定的树（第二个实参）没有与模式匹配（第一个参数）时，会产生错误。

5.3 条件 (Conditionals)

最简单的条件式是 `if` ；其余的条件式都是基于 `if` 所构造的。第二简单的条件式是 `when` ，它接受一个测试表达式（test expression）与一个代码主体。若测试表达式求值返回真时，则对主体求值。所以

```
(when (oddp that)
  (format t "Hmm, that's odd.")
  (+ that 1))
```

等同于

```
(if (oddp that)
    (progn
      (format t "Hmm, that's odd.")
      (+ that 1)))
```

when 的相反是 unless ；它接受相同的实参，但仅在测试表达式返回假时，才对主体求值。

所有条件式的母体 (从正反两面看) 是 cond ， cond 有两个新的优点：允许多个条件判断，与每个条件相关的代码隐含在 progn 里。 cond 预期在我们需要使用嵌套 if 的情况下使用。 举例来说，这个伪 member 函数

```
(defun our-member (obj lst)
  (if (atom lst)
      nil
      (if (eql (car lst) obj)
          lst
          (our-member obj (cdr lst)))))
```

也可以定义成：

```
(defun our-member (obj lst)
  (cond ((atom lst) nil)
        ((eql (car lst) obj) lst)
        (t (our-member obj (cdr lst)))))
```

事实上，Common Lisp 实现大概会把 cond 翻译成 if 的形式。

总得来说呢， cond 接受零个或多个实参。每一个实参必须是一个具有条件式，伴随着零个或多个表达式的列表。当 cond 表达式被求值时，测试条件式依序求值，直到某个测试条件式返回真才停止。当返回真时，与其相关联的表达式会被依序求值，而最后一个返回的数值，会作为 cond 的返回值。如果符合的条件式之后没有表达式的话：

```
> (cond (99))
99
```

则会返回条件式的值。

由于 cond 子句的 t 条件永远成立，通常我们把它放在最后，作为缺省的条件式。如果没有子句符合时，则 cond 返回 nil ，但利用 nil 作为返回值是一种很差的风格 (这种问题可能发生的例子，请看 292 页)。译注: **Appendix A, unexpected nil** 小节。

当你想要把一个数值与一系列的常量比较时，有 case 可以用。我们可以使用 case 来定义一个函数，返回每个月份中的天数：

```
(defun month-length (mon)
  (case mon
    ((jan mar may jul aug oct dec) 31)
    ((apr jun sept nov) 30)
    (feb (if (leap-year) 29 28))
    (otherwise "unknown month")))
```

一个 case 表达式由一个实参开始，此实参会被拿来与每个子句的键值做比较。接着是零个或多个子句，每个子句由一个或一串键值开始，跟随着零个或多个表达式。键值被视为常量；它们不会被求值。第一个参数的值被拿来与子句中的键值做比较 (使用 eql)。如果匹配时，子句剩余的表达式会被求值，并将最后一个求值作为 case 的返回值。

缺省子句的键值可以是 t 或 otherwise 。如果没有子句符合时，或是子句只包含键值时，

```
> (case 99 (99))
NIL
```

则 case 返回 nil 。

typecase 宏与 case 相似，除了每个子句中的键值应为类型修饰符 (type specifiers)，以及第一个实参与键值比较的函数使用 typep 而不是 eql (一个 typecase 的例子在 107 页)。 译注: **6.5** 小节。

5.4 迭代 (Iteration)

最基本的迭代操作符是 do ，在 2.13 小节介绍过。由于 do 包含了隐式的 block 及 tagbody ，我们现在知道是可以在 do 主体内使用 return 、 return-from 以及 go 。

2.13 节提到 do 的第一个参数必须是说明变量规格的列表，列表可以是如下形式：

```
(variable  initial  update)
```

initial 与 update 形式是选择性的。若 update 形式忽略时，每次迭代时不会更新变量。若 initial 形式也忽略时，变量会使用 nil 来初始化。

在 23 页的例子中（译注: 2.13 节），

```
(defun show-squares (start end)
  (do ((i start (+ i 1)))
```

```
      ((> i end) 'done)
    (format t "~A ~A~%" i (* i i))))
```

update 形式引用到由 do 所创造的变量。一般都是这么用。如果一个 do 的 update 形式，没有至少引用到一个 do 创建的变量时，反而很奇怪。

当同时更新超过一个变量时，问题来了，如果一个 update 形式，引用到一个拥有自己的 update 形式的变量时，它会被更新呢？或是获得前一次迭代的值？使用 do 的话，它获得后者的值：

```
> (let ((x 'a))
    (do ((x 1 (+ x 1))
         (y x x))
        ((> x 5))
      (format t "(~A ~A)  " x y)))
(1 A)  (2 1)  (3 2)  (4 3)  (5 4)
NIL
```

每一次迭代时， x 获得先前的值，加上一； y 也获得 x 的前一次数值。

但也有一个 do* ，它有着和 let 与 let* 一样的关系。任何 initial 或 update 形式可以参照到前一个子句的变量，并会获得当下的值：

```
> (do* ((x 1 (+ x 1))
        (y x x))
       ((> x 5))
  (format t "(~A ~A) " x y))
(1 1) (2 2) (3 3) (4 4) (5 5)
NIL
```

除了 do 与 do* 之外，也有几个特别用途的迭代操作符。要迭代一个列表的元素，我们可以使用 dolist :

```
> (dolist (x '(a b c d) 'done)
    (format t "~A " x))
A B C D
DONE
```

当迭代结束时，初始列表内的第三个表达式 (译注: done) ，会被求值并作为 dolist 的返回值。缺省是 nil 。

有着同样的精神的是 dotimes ，给定某个 n ，将会从整数 0 ，迭代至 n-1 :

```
(dotimes (x 5 x)
  (format t "~A " x))
0 1 2 3 4
```

```
5
```

`dolist` 与 `dotimes` 初始列表的第三个表达式皆可省略，省略时为 ``nil 。注意该表达式可引用到迭代过程中的变量。

（译注：第三个表达式即上例之 `x` ，可以省略，省略时 `dotimes` 表达式的返回值为 `nil` 。）

Note: do 的重点 (THE POINT OF do)

在 “The Evolution of Lisp” 里，Steele 与 Garbriel 陈述了 do 的重点， 表达的实在太好了，值得整个在这里引用过来：

撇开争论语法不谈，有件事要说明的是，在任何一个编程语言中，一个循环若一次只能更新一个变量是毫无用处的。 几乎在任何情况下，会有一个变量用来产生下个值，而另一个变量用来累积结果。如果循环语法只能产生变量， 那么累积结果就得借由赋值语句来“手动”实现...或有其他的副作用。具有多变量的 do 循环，体现了产生与累积的本质对称性，允许可以无副作用地表达迭代过程：

```
(defun factorial (n)
  (do ((j n (- j 1))
       (f 1 (* j f)))
    ((= j 0) f)))
```

当然在 step 形式里实现所有的实际工作，一个没有主体的 do 循环形式是较不寻常的。

函数 `mapc` 和 `mapcar` 很像，但不会 `cons` 一个新列表作为返回值，所以使用的唯一理由是为了副作用。它们比 `dolist` 来得灵活，因为可以同时遍历多个列表：

```
> (mapc #'(lambda (x y)
            (format t "~A ~A  " x y))
      '(hip flip slip)
      '(hop flop slop))
HIP HOP  FLIP FLOP  SLIP SLOP
(HIP FLIP SLIP)
```

总是返回 `mapc` 的第二个参数。

5.5 多值 (Multiple Values)

曾有人这么说，为了要强调函数式编程的重要性，每个 Lisp 表达式都返回一个值。现在事情不是这么简单了；在 Common Lisp 里，一个表达式可以返回零个或多个数值。最

多可以返回几个值取决于各家实现，但至少可以返回 19 个值。

多值允许一个函数返回多件事情的计算结果，而不用构造一个特定的结构。举例来说，内置的 `get-decoded-time` 返回 9 个数值来表示现在的时间：秒，分，时，日期，月，年，天，以及另外两个数值。

多值也使得查询函数可以分辨出 `nil` 与查询失败的情况。这也是为什么 `gethash` 返回两个值。因为它使用第二个数值来指出成功还是失败，我们可以在哈希表里储存 `nil` ，就像我们可以储存别的数值那样。

`values` 函数返回多个数值。它一个不少地返回你作为数值所传入的实参：

```
> (values 'a nil (+ 2 4))
A
NIL
6
```

如果一个 `values` 表达式，是函数主体最后求值的表达式，它所返回的数值变成函数的返回值。多值可以原封不地通过任何数量的返回来传递：

```
> ((lambda () ((lambda () (values 1 2)))))
1
2
```

然而若只预期一个返回值时，第一个之外的值会被舍弃：

```
> (let ((x (values 1 2)))
    x)
1
```

通过不带实参使用 `values` ，是可能不返回值的。在这个情况下，预期一个返回值的话，会获得 `nil` ：

```
> (values)
> (let ((x (values)))
    x)
NIL
```

要接收多个数值，我们使用 `multiple-value-bind` ：

```
> (multiple-value-bind (x y z) (values 1 2 3)
    (list x y z))
(1 2 3)

> (multiple-value-bind (x y z) (values 1 2)
```

```
    (list x y z))
(1 2 NIL)
```

如果变量的数量大于数值的数量，剩余的变量会是 `nil` 。如果数值的数量大于变量的数量，多余的值会被舍弃。所以只想印出时间我们可以这么写:

```
> (multiple-value-bind (s m h) (get-decoded-time)
    (format t "~A:~A:~A" h m s))
"4:32:13"
```

你可以借由 `multiple-value-call` 将多值作为实参传给第二个函数：

```
> (multiple-value-call #'+ (values 1 2 3))
6
```

还有一个函数是 `multiple-value-list`：

```
> (multiple-value-list (values 'a 'b 'c))
(A B C)
```

看起来像是使用 `#'list` 作为第一个参数的来调用 `multiple-value-call` 。

5.6 中止 (Aborts)

你可以使用 `return` 在任何时候离开一个 `block` 。有时候我们想要做更极端的事，在数个函数调用里将控制权转移回来。要达成这件事，我们使用 `catch` 与 `throw` 。一个 `catch` 表达式接受一个标签（tag），标签可以是任何类型的对象，伴随着一个表达式主体：

```
(defun super ()
  (catch 'abort
    (sub)
    (format t "We'll never see this.")))

(defun sub ()
  (throw 'abort 99))
```

表达式依序求值，就像它们是在 `progn` 里一样。在这段代码里的任何地方，一个带有特定标签的 `throw` 会导致 `catch` 表达式直接返回：

```
> (super)
99
```

一个带有给定标签的 `throw` ，为了要到达匹配标签的 `catch` ，会将控制权转移 (因此杀

掉进程)给任何有标签的 catch 。如果没有一个 catch 符合欲匹配的标签时， throw 会产生一个错误。

调用 error 同时中断了执行，本来会将控制权转移到调用树（calling tree）的更高点，取而代之的是，它将控制权转移给 Lisp 错误处理器（error handler）。通常会导致调用一个中断循环（break loop）。以下是一个假定的 Common Lisp 实现可能会发生的事情：

```
> (progn
    (error "Oops!")
    (format t "After the error."))
Error: Oops!
       Options: :abort, :backtrace
>>
```

译注：2 个 >> 显示进入中断循环了。

关于错误与状态的更多讯息，参见 14.6 小节以及附录 A。

有时候你想要防止代码被 throw 与 error 打断。借由使用 unwind-protect ，可以确保像是前述的中断，不会让你的程序停在不一致的状态。一个 unwind-protect 接受任何数量的实参，并返回第一个实参的值。然而即便是第一个实参的求值被打断时，剩下的表达式仍会被求值：

```
> (setf x 1)
1
> (catch 'abort
    (unwind-protect
      (throw 'abort 99)
      (setf x 2)))
99
> x
2
```

在这里，即便 throw 将控制权交回监测的 catch ， unwind-protect 确保控制权移交时，第二个表达式有被求值。无论何时，一个确切的动作要伴随着某种清理或重置时，unwind-protect 可能会派上用场。在 121 页提到了一个例子。

5.7 示例： 日期运算 (Example: Date Arithmetic)

在某些应用里，能够做日期的加减是很有用的 —— 举例来说，能够算出从 1997 年 12 月 17 日，六十天之后是 1998 年 2 月 15 日。在这个小节里，我们会编写一个实用的工具来做日期运算。我们会将日期转成整数，起始点设置在 2000 年 1 月 1 日。我们会使用内置的 + 与 - 函数来处理这些数字，而当我们转换完毕时，再将结果转回日期。

要将日期转成数字，我们需要从日期的单位中，算出总天数有多少。举例来说，2004 年 11 月 13 日的天数总和，是从起始点至 2004 年有多少天，加上从 2004 年到 2004 年 11 月有多少天，再加上 13 天。

有一个我们会需要的东西是，一张列出非润年每月份有多少天的表格。我们可以使用 Lisp 来推敲出这个表格的内容。我们从列出每月份的长度开始：

```
> (setf mon '(31 28 31 30 31 30 31 31 30 31 30 31))
(31 28 31 30 31 30 31 31 30 31 30 31)
```

我们可以通过应用 + 函数至这个列表来测试总长度：

```
> (apply #'+ mon)
365
```

现在如果我们反转这个列表并使用 `maplist` 来应用 + 函数至每下一个 `cdr` 上，我们可以获得从每个月份开始所累积的天数：

```
> (setf nom (reverse mon))
(31 30 31 30 31 31 30 31 30 31 28 31)
> (setf sums (maplist #'(lambda (x)
                          (apply #'+ x))
                      nom))
(365 334 304 273 243 212 181 151 120 90 59 31)
```

这些数字体现了从二月一号开始已经过了 31 天，从三月一号开始已经过了 59 天……等等。

我们刚刚建立的这个列表，可以转换成一个向量，见图 5.1，转换日期至整数的代码。

```
(defconstant month
  #(0 31 59 90 120 151 181 212 243 273 304 334 365))

(defconstant yzero 2000)

(defun leap? (y)
  (and (zerop (mod y 4))
       (or (zerop (mod y 400))
           (not (zerop (mod y 100))))))

(defun date->num (d m y)
  (+ (- d 1) (month-num m y) (year-num y)))

(defun month-num (m y)
  (+ (svref month (- m 1))
     (if (and (> m 2) (leap? y)) 1 0)))

(defun year-num (y)
```

```
  (let ((d 0))
    (if (>= y yzero)
        (dotimes (i (- y yzero) d)
          (incf d (year-days (+ yzero i))))
        (dotimes (i (- yzero y) (- d))
          (incf d (year-days (+ y i))))))))

(defun year-days (y) (if (leap? y) 366 365))
```

图 **5.1** 日期运算：转换日期至数字

典型 Lisp 程序的生命周期有四个阶段：先写好，然后读入，接着编译，最后执行。有件 Lisp 非常独特的事情之一是，在这四个阶段时， Lisp 一直都在那里。可以在你的程序编译 (参见 10.2 小节)或读入时 (参见 14.3 小节) 来调用 Lisp。我们推导出 `month` 的过程演示了，如何在撰写一个程序时使用 Lisp。

效率通常只跟第四个阶段有关系，运行期（run-time）。在前三个阶段，你可以随意的使用列表拥有的威力与灵活性，而不需要担心效率。

若你使用图 5.1 的代码来造一个时光机器（time machine），当你抵达时，人们大概会不同意你的日期。即使是相对近的现在，欧洲的日期也曾有过偏移，因为人们会获得更精准的每年有多长的概念。在说英语的国家，最后一次的不连续性出现在 1752 年，日期从 9 月 2 日跳到 9 月 14 日。

每年有几天取决于该年是否是润年。如果该年可以被四整除，我们说该年是润年，除非该年可以被 100 整除，则该年非润年 —— 而要是它可以被 400 整除，则又是润年。所以 1904 年是润年，1900 年不是，而 1600 年是。

要决定某个数是否可以被另个数整除，我们使用函数 `mod` ，返回相除后的余数：

```
> (mod 23 5)
3
> (mod 25 5)
0
```

如果第一个实参除以第二个实参的余数为 0，则第一个实参是可以被第二个实参整除的。函数 `leap?` 使用了这个方法，来决定它的实参是否是一个润年：

```
> (mapcar #'leap? '(1904 1900 1600))
(T NIL T)
```

我们用来转换日期至整数的函数是 `date->num` 。它返回日期中每个单位的天数总和。要找到从某月份开始的天数和，我们调用 `month-num` ，它在 `month` 中查询天数，如果是在润年的二月之后，则加一。

要找到从某年开始的天数和， date->num 调用 year-num ，它返回某年一月一日相对于起始点（2000.01.01）所代表的天数。这个函数的工作方式是从传入的实参 y 年开始，朝着起始年（2000）往上或往下数。

```
(defun num->date (n)
  (multiple-value-bind (y left) (num-year n)
    (multiple-value-bind (m d) (num-month left y)
      (values d m y))))

(defun num-year (n)
  (if (< n 0)
      (do* ((y (- yzero 1) (- y 1))
            (d (- (year-days y)) (- d (year-days y))))
           ((<= d n) (values y (- n d))))
      (do* ((y yzero (+ y 1))
            (prev 0 d)
            (d (year-days y) (+ d (year-days y))))
           ((> d n) (values y (- n prev))))))

(defun num-month (n y)
  (if (leap? y)
      (cond ((= n 59) (values 2 29))
            ((> n 59) (nmon (- n 1)))
            (t        (nmon n)))
      (nmon n)))

(defun nmon (n)
  (let ((m (position n month :test #'<)))
    (values m (+ 1 (- n (svref month (- m 1)))))))

(defun date+ (d m y n)
  (num->date (+ (date->num d m y) n)))
```

图 **5.2** 日期运算：转换数字至日期

图 5.2 展示了代码的下半部份。函数 num->date 将整数转换回日期。它调用了 num-year 函数，以日期的格式返回年，以及剩余的天数。再将剩余的天数传给 num-month ，分解出月与日。

和 year-num 相同， num-year 从起始年往上或下数，一次数一年。并持续累积天数，直到它获得一个绝对值大于或等于 n 的数。如果它往下数，那么它可以返回当前迭代中的数值。不然它会超过年份，然后必须返回前次迭代的数值。这也是为什么要使用 prev ， prev 在每次迭代时会存入 days 前次迭代的数值。

函数 num-month 以及它的子程序（subroutine） nmon 的行为像是相反地 month-num 。他们从常数向量 month 的数值到位置，然而 month-num 从位置到数值。

图 5.2 的前两个函数可以合而为一。与其返回数值给另一个函数， num-year 可以直接

调用 num-month 。现在分成两部分的代码，比较容易做交互测试，但是现在它可以工作了，下一步或许是把它合而为一。

有了 date->num 与 num->date ，日期运算是很简单的。我们在 date+ 里使用它们，可以从特定的日期做加减。如果我们想透过 date+ 来知道 1997 年 12 月 17 日六十天之后的日期:

```
> (multiple-value-list (date+ 17 12 1997 60))
(15 2 1998)
```

我们得到，1998 年 2 月 15 日。

Chapter 5 总结 (Summary)

1. Common Lisp 有三个基本的区块建构子： progn ； 允许返回的 block ； 以及允许 goto 的 tagbody 。很多内置的操作符隐含在区块里。
2. 进入一个新的词法语境，概念上等同于函数调用。
3. Common Lisp 提供了适合不同情况的条件式。每个都可以使用 if 来定义。
4. 有数个相似迭代操作符的变种。
5. 表达式可以返回多个数值。
6. 计算过程可以被中断以及保护，保护可使其免于中断所造成的后果。

Chapter 5 练习 (Exercises)

1. 将下列表达式翻译成没有使用 let 与 let* ，并使同样的表达式不被求值 2 次。

```
(a) (let ((x (car y)))
      (cons x x))
(b) (let* ((w (car x))
           (y (+ w z)))
      (cons w y))
```

2. 使用 cond 重写 29 页的 mystery 函数。（译注: 第二章的练习第 5 题的 (b) 部分)
3. 定义一个返回其实参平方的函数，而当实参是一个正整数且小于等于 5 时，不要计算其平方。
4. 使用 case 与 svref 重写 month-num (图 5.1)。
5. 定义一个迭代与递归版本的函数，接受一个对象 x 与向量 v ，并返回一个列表，包含了向量 v 当中，所有直接在 x 之前的对象：

```
> (precedes #\a "abracadabra")
(#\c #\d #\r)
```

6. 定义一个迭代与递归版本的函数，接受一个对象与列表，并返回一个新的列表，在原本列表的对象之间加上传入的对象：

```
> (intersperse '- '(a b c d))
(A - B - C - D)
```

7. 定义一个接受一系列数字的函数，并在若且唯若每一对（pair）数字的差为一时，返回真，使用

```
(a) 递归
(b) do
(c) mapc 与 return
```

8. 定义一个单递归函数，返回两个值，分别是向量的最大与最小值。
9. 图 3.12 的程序在找到一个完整的路径时，仍持续遍历伫列。在搜索范围大时，这可能会产生问题。

```
(a) 使用 catch 与 throw 来变更程序，使其找到第一个完整路径时，直接返回它。
(b) 重写一个做到同样事情的程序，但不使用 catch 与 throw。
```

第六章：函数

理解函数是理解 Lisp 的关键之一。概念上来说，函数是 Lisp 的核心所在。实际上呢，函数是你手边最有用的工具之一。

6.1 全局函数 (Global Functions)

谓词 `fboundp` 告诉我们，是否有个函数的名字与给定的符号绑定。如果一个符号是函数的名字，则 `symbol-function` 会返回它：

```
> (fboundp '+)
T
> (symbol-function '+)
#<Compiled-function + 17BA4E>
```

可通过 `symbol-function` 给函数配置某个名字：

```
(setf (symbol-function 'add2)
  #'(lambda (x) (+ x 2)))
```

新的全局函数可以这样定义，用起来和 `defun` 所定义的函数一样：

```
> (add2 1)
3
```

实际上 `defun` 做了稍微多的工作，将某些像是

```
(defun add2 (x) (+ x 2))
```

翻译成上述的 `setf` 表达式。使用 `defun` 让程序看起来更美观，并或多或少帮助了编译器，但严格来说，没有 `defun` 也能写程序。

通过把 `defun` 的第一个实参变成这种形式的列表 `(setf f)`，你定义了当 `setf` 第一个实参是 `f` 的函数调用时，所会发生的事情。下面这对函数把 `primo` 定义成 `car` 的同义词：

```
(defun primo (lst) (car lst))

(defun (setf primo) (val lst)
  (setf (car lst) val))
```

在函数名是这种形式 `(setf f)` 的函数定义中，第一个实参代表新的数值，而剩余的实

参代表了传给 f 的参数。

现在任何 primo 的 setf，会是上面后者的函数调用：

```
> (let ((x (list 'a 'b 'c)))
    (setf (primo x) 480)
    x)
(480 b c)
```

不需要为了定义 (setf primo) 而定义 primo，但这样的定义通常是成对的。

由于字符串是 Lisp 表达式，没有理由它们不能出现在代码的主体。字符串本身是没有副作用的，除非它是最后一个表达式，否则不会造成任何差别。如果让字符串成为 defun 定义的函数主体的第一个表达式，

```
(defun foo (x)
  "Implements an enhanced paradigm of diversity"
  x)
```

那么这个字符串会变成函数的文档字符串（documentation string）。要取得函数的文档字符串，可以通过调用 documentation 来取得：

```
> (documentation 'foo 'function)
"Implements an enhanced paradigm of diversity"
```

6.2 局部函数 (Local Functions)

通过 defun 或 symbol-function 搭配 setf 定义的函数是全局函数。你可以像存取全局变量那样，在任何地方存取它们。定义局部函数也是有可能的，局部函数和局部变量一样，只在某些上下文内可以访问。

局部函数可以使用 labels 来定义，它是一种像是给函数使用的 let 。它的第一个实参是一个新局部函数的定义列表，而不是一个变量规格说明的列表。列表中的元素为如下形式：

```
(name parameters . body)
```

而 labels 表达式剩余的部份，调用 name 就等于调用 (lambda parameters . body) 。

```
> (labels ((add10 (x) (+ x 10))
           (consa  (x) (cons 'a x)))
    (consa (add10 3)))
(A . 13)
```

labels 与 let 的类比在一个方面上被打破了。由 labels 表达式所定义的局部函数，可以被其他任何在此定义的函数引用，包括自己。所以这样定义一个递归的局部函数是可能的：

```
> (labels ((len (lst)
             (if (null lst)
                 0
                 (+ (len (cdr lst)) 1))))
    (len '(a b c)))
3
```

5.2 节展示了 let 表达式如何被理解成函数调用。 do 表达式同样可以被解释成调用递归函数。这样形式的 do：

```
(do ((x a (b x))
     (y c (d y)))
    ((test x y) (z x y))
  (f x y))
```

等同于

```
(labels ((rec (x y)
           (cond ((test x y)
                  (z x y))
                 (t
                  (f x y)
                  (rec (b x) (d y))))))
  (rec a c))
```

这个模型可以用来解决，任何你对于 do 行为仍有疑惑的问题。

6.3 参数列表 (Parameter Lists)

2.1 节我们演示过，有了前序表达式， + 可以接受任何数量的参数。从那时开始，我们看过许多接受不定数量参数的函数。要写出这样的函数，我们需要使用一个叫做剩余（*rest*）参数的东西。

如果我们在函数的形参列表里的最后一个变量前，插入 &rest 符号，那么当这个函数被调用时，这个变量会被设成一个带有剩余参数的列表。现在我们可以明白 funcall 是如何根据 apply 写成的。它或许可以定义成：

```
(defun our-funcall (fn &rest args)
  (apply fn args))
```

我们也看过操作符中，有的参数可以被忽略，并可以缺省设成特定的值。这样的参数称为选择性参数（optional parameters）。（相比之下，普通的参数有时称为必要参数「required parameters」）如果符号 `&optional` 出现在一个函数的形参列表时，

```
(defun philosoph (thing &optional property)
  (list thing 'is property))
```

那么在 `&optional` 之后的参数都是选择性的，缺省为 `nil`：

```
> (philosoph 'death)
(DEATH IS NIL)
```

我们可以明确指定缺省值，通过将缺省值附在列表里给入。这版的 `philosoph`

```
(defun philosoph (thing &optional (property 'fun))
  (list thing 'is property))
```

有着更鼓舞人心的缺省值：

```
> (philosoph 'death)
(DEATH IS FUN)
```

选择性参数的缺省值可以不是常量。可以是任何的 Lisp 表达式。若这个表达式不是常量，它会在每次需要用到缺省值时被重新求值。

一个关键字参数（keyword parameter）是一种更灵活的选择性参数。如果你把符号 `&key` 放在一个形参列表，那在 `&key` 之后的形参都是选择性的。此外，当函数被调用时，这些参数会被识别出来，参数的位置在哪不重要，而是用符号标签（译注: : ）识别出来：

```
> (defun keylist (a &key x y z)
    (list a x y z))
KEYLIST

> (keylist 1 :y 2)
(1 NIL 2 NIL)

> (keylist 1 :y 3 :x 2)
(1 2 3 NIL)
```

和普通的选择性参数一样，关键字参数缺省值为 `nil`，但可以在形参列表中明确地指定缺省值。

关键字与其相关的参数可以被剩余参数收集起来，并传递给其他期望收到这些参数的函

数。举例来说，我们可以这样定义 adjoin：

```
(defun our-adjoin (obj lst &rest args)
  (if (apply #'member obj lst args)
      lst
      (cons obj lst)))
```

由于 adjoin 与 member 接受一样的关键字，我们可以用剩余参数收集它们，再传给 member 函数。

5.2 节介绍过 destructuring-bind 宏。在通常情况下，每个模式（pattern）中作为第一个参数的子树，可以与函数的参数列表一样复杂：

```
(destructuring-bind ((&key w x) &rest y) '((:w 3) a)
  (list w x y))
(3 NIL (A))
```

6.4 示例：实用函数 (Example: Utilities)

2.6 节提到过，Lisp 大部分是由 Lisp 函数组成，这些函数与你可以自己定义的函数一样。这是程序语言中一个有用的特色：你不需要改变你的想法来配合语言，因为你可以改变语言来配合你的想法。如果你想要 Common Lisp 有某个特定的函数，自己写一个，而这个函数会成为语言的一部分，就跟内置的 + 或 eql 一样。

有经验的 Lisp 程序员，由上而下（top-down）也由下而上 (bottom-up)地工作。当他们朝着语言撰写程序的同时，也打造了一个更适合他们程序的语言。通过这种方式，语言与程序结合的更好，也更好用。

写来扩展 Lisp 的操作符称为实用函数（utilities）。当你写了更多 Lisp 程序时，会发现你开发了一系列的程序，而在一个项目写过许多的实用函数，下个项目里也会派上用场。

专业的程序员常发现，手边正在写的程序，与过去所写的程序有很大的关联。这就是软件重用让人听起来很吸引人的原因。但重用已经被联想成面向对象程序设计。但软件不需要是面向对象的才能重用 —— 这是很明显的，我们看看程序语言（换言之，编译器），是重用性最高的软件。

要获得可重用软件的方法是，由下而上地写程序，而程序不需要是面向对象的才能够由下而上地写出。实际上，函数式风格相比之下，更适合写出重用软件。想想看 sort 。在 Common Lisp 你几乎不需要自己写排序程序； sort 是如此的快与普遍，以致于它不值得我们烦恼。这才是可重用软件。

```
(defun single? (lst)
  (and (consp lst) (null (cdr lst))))

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

(defun map-int (fn n)
  (let ((acc nil))
    (dotimes (i n)
      (push (funcall fn i) acc))
    (nreverse acc)))

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

(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)
              (setf wins obj
                    max  score))))
        (values wins max))))
```

图 **6.1** 实用函数

你可以通过撰写实用函数，在程序里做到同样的事情。图 6.1 挑选了一组实用的函数。前两个 `single?` 与 `append1` 函数，放在这的原因是要演示，即便是小程序也很有用。前一个函数 `single?`，当实参是只有一个元素的列表时，返回真。

```
> (single? '(a))
T
```

而后一个函数 `append1` 和 `cons` 很像，但在列表后面新增一个元素，而不是在前面:

```
> (append1 '(a b c) 'd)
(A B C D)
```

下个实用函数是 `map-int`，接受一个函数与整数 `n`，并返回将函数应用至整数 `0` 到 `n-1` 的结果的列表。

这在测试的时候非常好用（一个 Lisp 的优点之一是，互动环境让你可以轻松地写出测

试）。如果我们只想要一个 0 到 9 的列表，我们可以：

```
> (map-int #'identity 10)
(0 1 2 3 4 5 6 7 8 9)
```

然而要是我们想要一个具有 10 个随机数的列表，每个数介于 0 至 99 之间（包含 99），我们可以忽略参数并只要:

```
> (map-int #'(lambda (x) (random 100))
           10)
(85 50 73 64 28 21 40 67 5 32)
```

map-int 的定义说明了 Lisp 构造列表的标准做法（idiom）之一。我们创建一个累积器 acc，初始化是 nil，并将之后的对象累积起来。当累积完毕时，反转累积器。 [1]

我们在 filter 中看到同样的做法。 filter 接受一个函数与一个列表，将函数应用至列表元素上时，返回所有非 nil 元素:

```
> (filter #'(lambda (x)
              (and (evenp x) (+ x 10)))
          '(1 2 3 4 5 6 7))
(12 14 16)
```

另一种思考 filter 的方式是用通用版本的 remove-if 。

图 6.1 的最后一个函数， most ，根据某个评分函数（scoring function），返回列表中最高分的元素。它返回两个值，获胜的元素以及它的分数:

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

如果平手的话，返回先驰得点的元素。

注意图 6.1 的最后三个函数，它们全接受函数作为参数。 Lisp 使得将函数作为参数传递变得便捷，而这也是为什么，Lisp 适合由下而上程序设计的原因之一。成功的实用函数必须是通用的，当你可以将细节作为函数参数传递时，要将通用的部份抽象起来就变得容易许多。

本节给出的函数是通用的实用函数。可以用在任何种类的程序。但也可以替特定种类的程序撰写实用函数。确实，当我们谈到宏时，你可以凌驾于 Lisp 之上，写出自己的特定语言，如果你想这么做的话。如果你想要写可重用软件，看起来这是最靠谱的方式。

6.5 闭包 (Closures)

函数可以如表达式的值，或是其它对象那样被返回。以下是接受一个实参，并依其类型返回特定的结合函数：

```
(defun combiner (x)
  (typecase x
    (number #'+)
    (list #'append)
    (t #'list)))
```

在这之上，我们可以创建一个通用的结合函数:

```
(defun combine (&rest args)
  (apply (combiner (car args))
         args))
```

它接受任何类型的参数，并以适合它们类型的方式结合。（为了简化这个例子，我们假定所有的实参，都有着一样的类型。）

```
> (combine 2 3)
5
> (combine '(a b) '(c d))
(A B C D)
```

2.10 小节提过词法变量（lexical variables）只在被定义的上下文内有效。伴随这个限制而来的是，只要那个上下文还有在使用，它们就保证会是有效的。

如果函数在词法变量的作用域里被定义时，函数仍可引用到那个变量，即便函数被作为一个值返回了，返回至词法变量被创建的上下文之外。下面我们创建了一个把实参加上 3 的函数：

```
> (setf fn (let ((i 3))
             #'(lambda (x) (+ x i))))
#<Interpreted-Function C0A51E>
> (funcall fn 2)
5
```

当函数引用到外部定义的变量时，这外部定义的变量称为自由变量（free variable）。函数引用到自由的词法变量时，称之为闭包（closure）。 [2] 只要函数还存在，变量就必须一起存在。

闭包结合了函数与环境（environment）；无论何时，当一个函数引用到周围词法环境的某个东西时，闭包就被隐式地创建出来了。这悄悄地发生在像是下面这个函数，是一样

的概念:

```
(defun add-to-list (num lst)
  (mapcar #'(lambda (x)
              (+ x num))
          lst))
```

这函数接受一个数字及列表，并返回一个列表，列表元素是元素与传入数字的和。在 lambda 表达式里的变量 `num` 是自由的，所以像是这样的情况，我们传递了一个闭包给 `mapcar` 。

一个更显着的例子会是函数在被调用时，每次都返回不同的闭包。下面这个函数返回一个加法器（adder）:

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

它接受一个数字，并返回一个将该数字与其参数相加的闭包（函数）。

```
> (setf add3 (make-adder 3))
#<Interpreted-Function COEBF6>
> (funcall add3 2)
5
> (setf add27 (make-adder 27))
#<Interpreted-Function C0EE4E>
> (funcall add27 2)
29
```

我们可以产生共享变量的数个闭包。下面我们定义共享一个计数器的两个函数:

```
(let ((counter 0))
  (defun reset ()
    (setf counter 0))
  (defun stamp ()
    (setf counter (+ counter 1))))
```

这样的一对函数或许可以用来创建时间戳章（time-stamps）。每次我们调用 `stamp` 时，我们获得一个比之前高的数字，而调用 `reset` 我们可以将计数器归零:

```
> (list (stamp) (stamp) (reset) (stamp))
(1 2 0 1)
```

你可以使用全局计数器来做到同样的事情，但这样子使用计数器，可以保护计数器被非预期的引用。

Common Lisp 有一个内置的函数 `complement` 函数，接受一个谓词，并返回谓词的补数（complement）。比如：

```
> (mapcar (complement #'oddp)
          '(1 2 3 4 5 6))
(NIL T NIL T NIL T)
```

有了闭包以后，很容易就可以写出这样的函数：

```
(defun our-complement (f)
  #'(lambda (&rest args)
      (not (apply f args))))
```

如果你停下来好好想想，会发现这是个非凡的小例子；而这仅是冰山一角。闭包是 Lisp 特有的美妙事物之一。闭包开创了一种在别的语言当中，像是不可思议的程序设计方法。

6.6 示例：函数构造器 (Example: Function Builders)

Dylan 是 Common Lisp 与 Scheme 的混合物，有着 Pascal 一般的语法。它有着大量返回函数的函数：除了上一节我们所看过的 *complement* ，Dylan 包含: `compose` 、 `disjoin` 、 `conjoin` 、 `curry` 、 `rcurry` 以及 `always` 。图 6.2 有这些函数的 Common Lisp 实现，而图 6.3 演示了一些从定义延伸出的等价函数。

```
(defun compose (&rest fns)
  (destructuring-bind (fn1 . rest) (reverse fns)
    #'(lambda (&rest args)
        (reduce #'(lambda (v f) (funcall f v))
                rest
                :initial-value (apply fn1 args)))))

(defun disjoin (fn &rest fns)
  (if (null fns)
      fn
      (let ((disj (apply #'disjoin fns)))
        #'(lambda (&rest args)
            (or (apply fn args) (apply disj args))))))

(defun conjoin (fn &rest fns)
  (if (null fns)
      fn
      (let ((conj (apply #'conjoin fns)))
        #'(lambda (&rest args)
            (and (apply fn args) (apply conj args))))))

(defun curry (fn &rest args)
  #'(lambda (&rest args2)
```

```
      (apply fn (append args args2))))

(defun rcurry (fn &rest args)
  #'(lambda (&rest args2)
      (apply fn (append args2 args))))

(defun always (x) #'(lambda (&rest args) x))
```

图 **6.2 Dylan** 函数建构器

首先， compose 接受一个或多个函数，并返回一个依序将其参数应用的新函数，即，

```
(compose #'a #'b #'c)
```

返回一个函数等同于

```
#'(lambda (&rest args) (a (b (apply #'c args))))
```

这代表着 compose 的最后一个实参，可以是任意长度，但其它函数只能接受一个实参。

下面我们建构了一个函数，先给取参数的平方根，取整后再放回列表里，接著返回:

```
> (mapcar (compose #'list #'round #'sqrt)
          '(4 9 16 25))
((2) (3) (4) (5))
```

接下来的两个函数， disjoin 及 conjoin 同接受一个或多个谓词作为参数： disjoin 当任一谓词返回真时，返回真，而 conjoin 当所有谓词返回真时，返回真。

```
> (mapcar (disjoin #'integerp #'symbolp)
          '(a "a" 2 3))
(T NIL T T)
```

```
> (mapcar (conjoin #'integerp #'symbolp)
          '(a "a" 2 3))
(NIL NIL NIL T)
```

若考虑将谓词定义成集合， disjoin 返回传入参数的联集（union），而 conjoin 则是返回传入参数的交集（intersection）。

```
      cddr = (compose #'cdr #'cdr)
      nth  = (compose #'car #'nthcdr)
      atom = (compose #'not #'consp)
           = (rcurry #'typep 'atom)
        <= = (disjoin #'< #'=)
     listp = (disjoin #'< #'=)
```

```
           = (rcurry #'typep 'list)
        1+ = (curry #'+ 1)
           = (rcurry #'+ 1)
        1- = (rcurry #'- 1)
    mapcan = (compose (curry #'apply #'nconc) #'mapcar
complement = (curry #'compose #'not)
```

图 **6.3** 某些等价函数

函数 `curry` 与 `rcurry` （“right curry”）精神上与前一小节的 `make-adder` 相同。两者皆接受一个函数及某些参数，并返回一个期望剩余参数的新函数。下列任一个函数等同于 `(make-adder 3)`：

```
(curry #'+ 3)
(rcurry #'+ 3)
```

当函数的参数顺序重要时，很明显可以看出 `curry` 与 `rcurry` 的差别。如果我们 `curry #'-`，我们得到一个用其参数减去某特定数的函数，

```
(funcall (curry #'- 3) 2)
1
```

而当我们 `rcurry #'-` 时，我们得到一个用某特定数减去其参数的函数:

```
(funcall (rcurry #'- 3) 2)
-1
```

最后， `always` 函数是 Common Lisp 函数 `constantly` 。接受一个参数并原封不动返回此参数的函数。和 `identity` 一样，在很多需要传入函数参数的情况下很有用。

6.7 动态作用域 (Dynamic Scope)

2.11 小节解释过局部与全局变量的差别。实际的差别是词法作用域（lexical scope）的词法变量（lexical variable），与动态作用域（dynamic scope）的特别变量（special variable）的区别。但这俩几乎是没有区别，因为局部变量几乎总是是词法变量，而全局变量总是是特别变量。

在词法作用域下，一个符号引用到上下文中符号名字出现的地方。局部变量缺省有着词法作用域。所以如果我们在一个环境里定义一个函数，其中有一个变量叫做 `x`，

```
(let ((x 10))
  (defun foo ()
    x))
```

则无论 foo 被调用时有存在其它的 x，主体内的 x 都会引用到那个变量:

```
> (let ((x 20)) (foo))
10
```

而动态作用域，我们在环境中函数被调用的地方寻找变量。要使一个变量是动态作用域的，我们需要在任何它出现的上下文中声明它是 special 。如果我们这样定义 foo：

```
(let ((x 10))
  (defun foo ()
    (declare (special x))
    x))
```

则函数内的 x 就不再引用到函数定义里的那个词法变量，但会引用到函数被调用时，当下所存在的任何特别变量 x：

```
> (let ((x 20))
    (declare (special x))
    (foo))
20
```

新的变量被创建出来之后， 一个 declare 调用可以在代码的任何地方出现。 special 声明是独一无二的，因为它可以改变程序的行为。 13 章将讨论其它种类的声明。所有其它的声明，只是给编译器的建议；或许可以使程序运行的更快，但不会改变程序的行为。

通过在顶层调用 setf 来配置全局变量，是隐式地将变量声明为特殊变量:

```
> (setf x 30)
30
> (foo)
30
```

在一个文件里的代码，如果你不想依赖隐式的特殊声明，可以使用 defparameter 取代，让程序看起来更简洁。

动态作用域什么时候会派上用场呢？通常用来暂时给某个全局变量赋新值。举例来说，有 11 个变量来控制对象印出的方式，包括了 *print-base* ，缺省是 10 。如果你想要用 16 进制显示数字，你可以重新绑定 *print-base*：

```
> (let ((*print-base* 16))
    (princ 32))
20
32
```

这里显示了两件事情，由 princ 产生的输出，以及它所返回的值。他们代表着同样的数字，第一次在被印出时，用 16 进制显示，而第二次，因为在 let 表达式外部，所以是用十进制显示，因为 *print-base* 回到之前的数值， 10 。

6.8 编译 (Compilation)

Common Lisp 函数可以独立被编译或挨个文件编译。如果你只是在顶层输入一个 defun 表达式：

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

许多实现会创建一个直译的函数（interpreted function）。你可以将函数传给 compiled-function-p 来检查一个函数是否有被编译:

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

若你将 foo 函数名传给 compile :

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

则这个函数会被编译，而直译的定义会被编译出来的取代。编译与直译函数的行为一样，只不过对 compiled-function-p 来说不一样。

你可以把列表作为参数传给 compile 。这种 compile 的用法在 161 页 (译注: 10.1 小节)。

有一种函数你不能作为参数传给 compile : 一个像是 stamp 或是 reset 这种，在顶层明确使用词法上下文输入的函数 (即 let) [3] 在一个文件里面定义这些函数，接着编译然后载入文件是可以的。这么限制直译的代码的是实作的原因，而不是因为在词法上下文里明确定义函数有什么问题。

通常要编译 Lisp 代码不是挨个函数编译，而是使用 compile-file 编译整个文件。这个函数接受一个文件名，并创建一个原始码的编译版本 —— 通常会有同样的名称，但不同的扩展名。当编译过的文件被载入时， compiled-function-p 应给所有定义在文件内的函数返回真。

当一个函数包含在另一个函数内时，包含它的函数会被编译，而且内部的函数也会被编译。所以 make-adder (108 页)被编译时，它会返回编译的函数:

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

6.9 使用递归 (Using Recursion)

比起多数别的语言，递归在 Lisp 中扮演了一个重要的角色。这主要有三个原因：

1. 函数式程序设计。递归演算法有副作用的可能性较低。
2. 递归数据结构。 Lisp 隐式地使用了指标，使得递归地定义数据结构变简单了。最常见的是用在列表：一个列表的递归定义，列表为空表，或是一个 `cons` ，其中 `cdr` 也是个列表。
3. 优雅性。Lisp 程序员非常关心它们的程序是否美丽，而递归演算法通常比迭代演算法来得优雅。

学生们起初会觉得递归很难理解。但 3.9 节指出了，如果你想要知道是否正确，不需要去想递归函数所有的调用过程。

同样的如果你想写一个递归函数。如果你可以描述问题是怎么递归解决的，通常很容易将解法转成代码。要使用递归来解决一个问题，你需要做两件事：

1. 你必须要示范如何解决问题的一般情况，通过将问题切分成有限小并更小的子问题。
2. 你必须要示范如何通过 —— 有限的步骤，来解决最小的问题 —— 基本用例。

如果这两件事完成了，那问题就解决了。因为递归每次都将问题变得更小，而一个有限的问题终究会被解决的，而最小的问题仅需几个有限的步骤就能解决。

举例来说，下面这个找到一个正规列表（proper list）长度的递归算法，我们每次递归时，都可以找到更小列表的长度：

1. 在一般情况下，一个正规列表的长度是它的 `cdr` 加一。
2. 基本用例，空列表长度为 `0` 。

当这个描述翻译成代码时，先处理基本用例；但公式化递归演算法时，我们通常从一般情况下手。

前述的演算法，明确地描述了一种找到正规列表长度的方法。当你定义一个递归函数时，你必须要确定你在分解问题时，问题实际上越变越小。取得一个正规列表的 `cdr` 会给出 `length` 更小的子问题，但取得环状列表（circular list）的 `cdr` 不会。

这里有两个递归算法的示例。假定参数是有限的。注意第二个示例，我们每次递归时，将问题分成两个更小的问题：

第一个例子， `member` 函数，我们说某物是列表的成员，需满足：如果它是第一个元素的成员或是 `member` 的 `cdr` 的成员。但空列表没有任何成员。

第二个例子， `copy-tree` 一个 `cons` 的 `copy-tree` ，是一个由 `cons` 的 `car` 的 `copy-tree` 与 `cdr` 的 `copy-tree` 所组成的。一个原子的 `copy-tree` 是它自己。

一旦你可以这样描述算法，要写出递归函数只差一步之遥。

某些算法通常是这样表达最自然，而某些算法不是。你可能需要翻回前面，试试不使用递归来定义 `our-copy-tree` (41 页，译注: 3.8 小节)。另一方面来说，23 页 (译注: 2.13 节) 迭代版本的 `show-squares` 可能更容易比 24 页的递归版本要容易理解。某些时候是很难看出哪个形式比较自然，直到你试着去写出程序来。

如果你关心效率，有两个你需要考虑的议题。第一，尾递归（tail-recursive），会在 13.2 节讨论。一个好的编译器，使用循环或是尾递归的速度，应该是没有或是区别很小的。然而如果你需要使函数变成尾递归的形式时，或许直接用迭代会更好。

另一个需要铭记在心的议题是，最显而易见的递归算法，不一定是最有效的。经典的例子是费氏函数。它是这样递归地被定义的，

1. Fib(0) = Fib(1) = 1
2. Fib(n) = Fib(n-1)+Fib(n-2)

直接翻译这个定义，

```
(defun fib (n)
  (if (<= n 1)
      1
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

这样是效率极差的。一次又一次的重复计算。如果你要找 `(fib 10)` ，这个函数计算 `(fib 9)` 与 `(fib 8)` 。但要计算出 `(fib 9)` ，它需要再次计算 `(fib 8)` ，等等。

下面是一个算出同样结果的迭代版本:

```
(defun fib (n)
  (do ((i n (- i 1))
       (f1 1 (+ f1 f2))
       (f2 1 f1))
      ((<= i 1) f1)))
```

迭代的版本不如递归版本来得直观，但是效率远远高出许多。这样的事情在实践中常发生吗？非常少 —— 这也是为什么所有的教科书都使用一样的例子 —— 但这是需要注意的事。

Chapter 6 总结 (Summary)

1. 命名函数是一个存在符号的 `symbol-function` 部分的函数。 `defun` 宏隐藏了这样的细节。它也允许你定义文档字符串（documentation string），并指定 `setf` 要怎么处理函数调用。
2. 定义局部函数是有可能的，与定义局部变量有相似的精神。
3. 函数可以有选择性参数（optional）、剩余（rest）以及关键字（keyword）参数。
4. 实用函数是 Lisp 的扩展。他们是由下而上编程的小规模示例。
5. 只要有某物引用到词法变量时，它们会一直存在。闭包是引用到自由变量的函数。你可以写出返回闭包的函数。
6. Dylan 提供了构造函数的函数。很简单就可以使用闭包，然后在 Common Lisp 中实现它们。
7. 特别变量（special variable）有动态作用域 (dynamic scope)。
8. Lisp 函数可以单独编译，或（更常见）编译整个文件。
9. 一个递归演算法通过将问题细分成更小、更小的子问题来解决问题。

Chapter 6 练习 (Exercises)

1. 定义一个 `tokens` 版本 (67 页)，接受 `:test` 与 `:start` 参数，缺省分别是 `#'constituent` 与 `0` 。(译注: 67 页在 4.5 小节)
2. 定义一个 `bin-search` (60 页)的版本，接受 `:key` , `:test` , `start` 与 `end` 参数，有着一般的意义与缺省值。(译注: 60 页在 4.1 小节)
3. 定义一个函数，接受任何数目的参数，并返回传入的参数。
4. 修改 `most` 函数 (105 页)，使其返回 2 个数值，一个列表中最高分的两个元素。(译注: 105 页在 6.4 小节)
5. 用 `filter` (105 页) 来定义 `remove-if` （没有关键字）。(译注: 105 页在 6.4 小节)
6. 定义一个函数，接受一个参数、一个数字，并返回目前传入参数中最大的那个。
7. 定义一个函数，接受一个参数、一个数字，若传入参数比上个参数大时，返回真。函数第一次调用时应返回 `nil` 。
8. 假设 `expensive` 是一个接受一个参数的函数，一个介于 0 至 100 的整数（包含 100)，返回一个耗时的计算结果。定义一个函数 `frugal` 来返回同样的答案，但仅在没见过传入参数时调用 `expensive` 。
9. 定义一个像是 `apply` 的函数，但在任何数字印出前，缺省用 8 进制印出。

脚注

[1] 在这个情况下， `nreverse` (在 222 页描述)和 `reverse` 做一样的事情，但更有效率。

[2] “闭包”这个名字是早期的 Lisp 方言流传而来。它是从闭包需要在动态作用域里实现的方式衍生而来。

[3] 以前的 ANSI Common Lisp， `compile` 的第一个参数也不能是一个已经编译好的函数。

第七章：输入与输出

Common Lisp 有着威力强大的 I/O 工具。针对输入以及一些普遍读取字符的函数，我们有 `read`，包含了一个完整的解析器 (parser)。针对输出以及一些普遍写出字符的函数，我们有 `format`，它自己几乎就是一个语言。本章介绍了所有基本的概念。

Common Lisp 有两种流 (streams)，字符流与二进制流。本章描述了字符流的操作；二进制流的操作涵盖在 14.2 节。

7.1 流 (Streams)

流是用来表示字符来源或终点的 Lisp 对象。要从文件读取或写入，你将文件作为流打开。但流与文件是不一样的。当你在顶层读入或印出时，你也可以使用流。你甚至可以创建可以读取或写入字符串的流。

输入缺省是从 `*standard-input*` 流读取。输出缺省是在 `*standard-output*` 流。最初它们大概会在相同的地方：一个表示顶层的流。

我们已经看过 `read` 与 `format` 是如何在顶层读取与印出。前者接受一个应是流的选择性参数，缺省是 `*standard-input*`。`format` 的第一个参数也可以是一个流，但当它是 `t` 时，输出被送到 `*standard-output*`。所以我们目前为止都只用到缺省的流而已。我们可以在任何流上面做同样的 I/O 操作。

路径名（pathname）是一种指定一个文件的可移植方式。路径名包含了六个部分：host、device、directory、name、type 及 version。你可以通过调用 `make-pathname` 搭配一个或多个对应的关键字参数来产生一个路径。在最简单的情况下，你可以只指明名字，让其他的部分留为缺省：

```
> (setf path (make-pathname :name "myfile"))
      #P"myfile"
```

开启一个文件的基本函数是 `open`。它接受一个路径名 [1] 以及大量的选择性关键字参数，而若是开启成功时，返回一个指向文件的流。

你可以在创建流时，指定你想要怎么使用它。　无论你是要写入流、从流读取或者同时进行读写操作，都可以通过 `direction` 参数设置。三个对应的数值是 `:input`，`:output`，`:io`。如果是用来输出的流，`if-exists` 参数说明了如果文件已经存在时该怎么做；通常它应该是 `:supersede` (译注: 取代)。所以要创建一个可以写至 `"myfile"` 文件的流，你可以：

```
> (setf str (open path :direction :output
                       :if-exists :supersede))
#<Stream C017E6>
```

流的打印表示法因实现而异。

现在我们可以把这个流作为第一个参数传给 `format`，它会在流印出，而不是顶层：

```
> (format str "Something~%")
NIL
```

如果我们在此时检查这个文件，可能有输出，也可能没有。某些实现会将输出累积成一块 (chunks)再输出。直到我们将流关闭，它也许一直不会出现：

```
> (close str)
NIL
```

当你使用完时，永远记得关闭文件；在你还没关闭之前，内容是不保证会出现的。现在如果我们检查文件 “myfile”，应该有一行：

```
Something
```

如果我们只想从一个文件读取，我们可以开启一个具有 `:direction :input` 的流：

```
> (setf str (open path :direction :input))
#<Stream C01C86>
```

我们可以对一个文件使用任何输入函数。7.2 节会更详细的描述输入。这里作为一个示例，我们将使用 `read-line` 从文件来读取一行文字：

```
> (read-line str)
"Something"
> (close str)
NIL
```

当你读取完毕时，记得关闭文件。

大部分时间我们不使用 `open` 与 `close` 来操作文件的 I/O。`with-open-file` 宏通常更方便。它的第一个参数应该是一个列表，包含了变数名、伴随着你想传给 `open` 的参数。在这之后，它接受一个代码主体，它会被绑定至流的变数一起被求值，其中流是通过将剩余的参数传给 `open` 来创建的。之后这个流会被自动关闭。所以整个文件写入动作可以表示为：

```
(with-open-file (str path :direction :output
```

```
                              :if-exists :supersede)
  (format str "Something~%"))
```

with-open-file 宏将 close 放在 unwind-protect 里 (参见 92 页，译注: 5.6 节)，即使一个错误打断了主体的求值，文件是保证会被关闭的。

7.2 输入 (Input)

两个最受欢迎的输入函数是 read-line 及 read 。前者读入换行符 (newline)之前的所有字符，并用字符串返回它们。它接受一个选择性流参数 (optional stream argument)；若流忽略时，缺省为 *standard-input* :

```
> (progn
    (format t "Please enter your name: ")
    (read-line))
Please enter your name: Rodrigo de Bivar
"Rodrigo de Bivar"
NIL
```

译注：Rodrigo de Bivar 人称熙德 (El Cid)，十一世纪的西班牙民族英雄。

如果你想要原封不动的输出，这是你该用的函数。(第二个返回值只在 read-line 在遇到换行符之前，用尽输入时返回真。)

在一般情况下， read-line 接受四个选择性参数: 一个流；一个参数用来决定遇到 end-of-file 时，是否产生错误；若前一个参数为 nil 时，该返回什么；第四个参数 (在 235 页讨论)通常可以省略。

所以要在顶层显示一个文件的内容，我们可以使用下面这个函数：

```
(defun pseudo-cat (file)
  (with-open-file (str file :direction :input)
    (do ((line (read-line str nil 'eof)
               (read-line str nil 'eof)))
        ((eql line 'eof))
      (format t "~A~%" line))))
```

如果我们想要把输入解析为 Lisp 对象，使用 read 。这个函数恰好读取一个表达式，在表达式结束时停止读取。所以可以读取多于或少于一行。而当然它所读取的内容必须是合法的 Lisp 语法。

如果我们在顶层使用 read ，它会让我们在表达式里面，想用几个换行符就用几个：

```
> (read)
```

```
(a
b
c)
(A B C)
```

换句话说，如果我们在一行里面输入许多表达式， `read` 会在第一个表达式之后，停止处理字符，留下剩余的字符给之后读取这个流的函数处理。所以如果我们在一行输入多个表达式，来回应 `ask-number` (20 页。译注：2.10 小节)所印出提示符，会发生如下情形:

```
> (ask-number)
Please enter a number. a b
Please enter a number. Please enter a number. 43
43
```

两个连续的提示符 (successive prompts)在第二行被印出。第一个 `read` 调用会返回 `a` ，而它不是一个数字，所以函数再次要求一个数字。但第一个 `read` 只读取到 `a` 的结尾。所以下一个 `read` 调用返回 `b` ，导致了下一个提示符。

你或许想要避免使用 `read` 来直接处理使用者的输入。前述的函数若使用 `read-line` 来获得使用者输入会比较好，然后对结果字符串调用 `read-from-string` 。这个函数接受一个字符串，并返回第一个读取的表达式:

```
> (read-from-string "a b c")
A
2
```

它同时返回第二个值，一个指出停止读取字符串时的位置的数字。

在一般情况下， `read-from-string` 可以接受两个选择性参数与三个关键字参数。两个选择性参数是 `read` 的第三、第四个参数: 一个 end-of-file (这个情况是字符串) 决定是否报错，若不报错该返回什么。关键字参数 `:start` 及 `:end` 可以用来划分从字符串的哪里开始读。

所有的这些输入函数是由基本函数 (primitive) `read-char` 所定义的，它读取一个字符。它接受四个与 `read` 及 `read-line` 一样的选择性参数。Common Lisp 也定义一个函数叫做 `peek-char` ，跟 `read-char` 类似，但不会将字符从流中移除。

7.3 输出 (Output)

三个最简单的输出函数是 `prin1` , `princ` 以及 `terpri` 。这三个函数的最后一个参数皆为选择性的流参数，缺省是 `*standard-output*` 。

prin1 与 princ 的差别大致在于 prin1 给程序产生输出，而 princ 给人类产生输出。所以举例来说， prin1 会印出字符串左右的双引号，而 princ 不会:

```
> (prin1 "Hello")
"Hello"
"Hello"
> (princ "Hello")
Hello
"Hello"
```

两者皆返回它们的第一个参数 (译注: 第二个值是返回值) —— 顺道一提，是用 prin1 印出。 terpri 仅印出一新行。

有这些函数的背景知识在解释更为通用的 format 是很有用的。这个函数几乎可以用在所有的输出。他接受一个流 (或 t 或 nil)、一个格式化字符串 (format string)以及零个或多个额外的参数。格式化字符串可以包含特定的格式化指令 (format directives)，这些指令前面有波浪号 ~ 。某些格式化指令作为字符串的占位符 (placeholder)使用。这些位置会被格式化字符串之后，所给入参数的表示法所取代。

如果我们把 t 作为第一个参数，输出会被送至 *standard-output* 。如果我们给 nil ， format 会返回一个它会如何印出的字符串。为了保持简短，我们会在所有的示例里演示怎么做。

由于每人的观点不同， format 可以是令人惊讶的强大或是极为可怕的复杂。有大量的格式化指令可用，而只有少部分会被大多数程序设计师使用。两个最常用的格式化指令是 ~A 以及 ~% 。(你使用 ~a 或 ~A 都没关系，但后者较常见，因为它让格式化指令看起来一目了然。) 一个 ~A 是一个值的占位符，它会像是用 princ 印出一般。一个 ~% 代表着一个换行符 (newline)。

```
> (format nil "Dear ~A, ~% Our records indicate..."
                                              "Mr. Malatesta")
      "Dear Mr. Malatesta,
         Our records indicate..."
```

这里 format 返回了一个值，由一个含有换行符的字符串组成。

~S 格式化指令像是 ~A ，但它使用 prin1 印出对象，而不是 princ 印出:

```
> (format t "~S  ~A" "z" "z")
"z" z
NIL
```

格式化指令可以接受参数。 ~F 用来印出向右对齐 (right-justified)的浮点数，可接受五个参数:

1. 要印出字符的总数。缺省是数字的长度。
2. 小数之后要印几位数。缺省是全部。
3. 小数点要往右移几位 (即等同于将数字乘 10)。缺省是没有。
4. 若数字太长无法满足第一个参数时，所要印出的字符。如果没有指定字符，一个过长的数字会尽可能使用它所需的空间被印出。
5. 数字开始印之前左边的字符。缺省是空白。

下面是一个有五个参数的罕见例子:

```
? (format nil "~10,2,0,'*,' F" 26.21875)
"     26.22"
```

这是原本的数字取至小数点第二位、(小数点向左移 0 位)、在 10 个字符的空间里向右对齐，左边补满空白。注意作为参数给入是写成 `'*` 而不是 `#\*` 。由于数字塞得下 10 个字符，不需要使用第四个参数。

所有的这些参数都是选择性的。要使用缺省值你可以直接忽略对应的参数。如果我们想要做的是，印出一个小数点取至第二位的数字，我们可以说:

```
> (format nil "~,2,,,F" 26.21875)
"26.22"
```

你也可以忽略一系列的尾随逗号 (trailing commas)，前面指令更常见的写法会是:

```
> (format nil "~,2F" 26.21875)
"26.22"
```

警告：当 `format` 取整数时，它不保证会向上进位或向下舍入。就是说 `(format nil "~,1F" 1.25)` 可能会是 `"1.2"` 或 `"1.3"` 。所以如果你使用 `format` 来显示资讯时，而使用者期望看到某种特定取整数方式的数字 (如: 金额数量)，你应该在印出之前先显式地取好整数。

7.4 示例：字符串代换 (Example: String Substitution)

作为一个 I/O 的示例，本节演示如何写一个简单的程序来对文本文件做字符串替换。我们即将写一个可以将一个文件中，旧的字符串 `old` 换成某个新的字符串 `new` 的函数。最简单的实现方式是将输入文件里的每一个字符与 `old` 的第一个字符比较。如果没有匹配，我们可以直接印出该字符至输出。如果匹配了，我们可以将输入的下一个字符与 `old` 的第二个字符比较，等等。如果输入字符与 `old` 完全相等时，我们有一个成功的匹配，则我们印出 `new` 至文件。

而要是 `old` 在匹配途中失败了，会发生什么事呢？举例来说，假设我们要找的模式(pattern)是 `"abac"` ，而输入文件包含的是 `"ababac"` 。输入会一直到第四个字符才发现不匹配，也就是在模式中的 `c` 以及输入的 `b` 才发现。在此时我们可以将原本的 `a` 写至输出文件，因为我们已经知道这里没有匹配。但有些我们从输入读入的字符还是需要留着: 举例来说，第三个 `a` ，确实是成功匹配的开始。所以在我们要实现这个算法之前，我们需要一个地方来储存，我们已经从输入读入的字符，但之后仍然需要的字符。

一个暂时储存输入的队列 (queue)称作缓冲区 (buffer)。在这个情况里，因为我们知道我们不需要储存超过一个预定的字符量，我们可以使用一个叫做环状缓冲区 `ring buffer` 的资料结构。一个环状缓冲区实际上是一个向量。是使用的方式使其成为环状: 我们将之后的元素所输入进来的值储存起来，而当我们到达向量结尾时，我们重头开始。如果我们不需要储存超过 `n` 个值，则我们只需要一个长度为 `n` 或是大于 `n` 的向量，这样我们就不需要覆写正在用的值。

在图 7.1 的代码，实现了环状缓冲区的操作。 `buf` 有五个字段 (field): 一个包含存入缓冲区的向量，四个其它字段用来放指向向量的索引 (indices)。两个索引是 `start` 与 `end` ，任何环状缓冲区的使用都会需要这两个索引: `start` 指向缓冲区的第一个值，当我们取出一个值时， `start` 会递增 (incremented)； `end` 指向缓冲区的最后一个值，当我们插入一个新值时， `end` 会递增。

另外两个索引， `used` 以及 `new` ，是我们需要给这个应用的基本环状缓冲区所加入的东西。它们会介于 `start` 与 `end` 之间。实际上，它总是符合

```
start ≤ used ≤ new ≤ end
```

你可以把 `used` 与 `new` 想成是当前匹配 (current match) 的 `start` 与 `end` 。当我们开始一轮匹配时， `used` 会等于 `start` 而 `new` 会等于 `end` 。当下一个字符 (successive character)匹配时，我们需要递增 `used` 。当 `used` 与 `new` 相等时，我们将开始匹配时，所有存在缓冲区的字符读入。我们不想要使用超过从匹配时所存在缓冲区的字符，或是重复使用同样的字符。因此这个 `new` 索引，开始等于 `end` ，但它不会在一轮匹配我们插入新字符至缓冲区一起递增。

函数 `bref` 接受一个缓冲区与一个索引，并返回索引所在位置的元素。借由使用 `index` 对向量的长度取 `mod` ，我们可以假装我们有一个任意长的缓冲区。调用 `(new-buf n)` 会产生一个新的缓冲区，能够容纳 `n` 个对象。

要插入一个新值至缓冲区，我们将使用 `buf-insert` 。它将 `end` 递增，并把新的值放在那个位置 (译注: 递增完的位置)。相反的 `buf-pop` 返回一个缓冲区的第一个数值，接着将 `start` 递增。任何环状缓冲区都会有这两个函数。

```
(defstruct buf
  vec (start -1) (used -1) (new -1) (end -1))

(defun bref (buf n)
  (svref (buf-vec buf)
         (mod n (length (buf-vec buf)))))

(defun (setf bref) (val buf n)
  (setf (svref (buf-vec buf)
               (mod n (length (buf-vec buf))))
        val))

(defun new-buf (len)
  (make-buf :vec (make-array len)))

(defun buf-insert (x b)
  (setf (bref b (incf (buf-end b))) x))

(defun buf-pop (b)
  (prog1
    (bref b (incf (buf-start b)))
    (setf (buf-used b) (buf-start b)
          (buf-new  b) (buf-end   b))))

(defun buf-next (b)
  (when (< (buf-used b) (buf-new b))
    (bref b (incf (buf-used b)))))

(defun buf-reset (b)
  (setf (buf-used b) (buf-start b)
        (buf-new  b) (buf-end   b)))

(defun buf-clear (b)
  (setf (buf-start b) -1 (buf-used  b) -1
        (buf-new   b) -1 (buf-end   b) -1))

(defun buf-flush (b str)
  (do ((i (1+ (buf-used b)) (1+ i)))
      ((> i (buf-end b)))
    (princ (bref b i) str)))
```

图 **7.1** 环状缓冲区的操作

接下来我们需要两个特别为这个应用所写的函数: `buf-next` 从缓冲区读取一个值而不取出，而 `buf-reset` 重置 `used` 与 `new` 到初始值，分别是 `start` 与 `end` 。如果我们已经把至 `new` 的值全部读取完毕时， `buf-next` 返回 `nil` 。区别这个值与实际的值不会产生问题，因为我们只把值存在缓冲区。

最后 `buf-flush` 透过将所有作用的元素，写至由第二个参数所给入的流，而 `buf-clear` 通过重置所有的索引至 `-1` 将缓冲区清空。

在图 7.1 定义的函数被图 7.2 所使用，包含了字符串替换的代码。函数 file-subst 接受四个参数；一个查询字符串，一个替换字符串，一个输入文件以及一个输出文件。它创建了代表每个文件的流，然后调用 stream-subst 来完成实际的工作。

第二个函数 stream-subst 使用本节开始所勾勒的算法。它一次从输入流读一个字符。直到输入字符匹配要寻找的字符串时，直接写至输出流 (1)。当一个匹配开始时，有关字符在缓冲区 buf 排队等候 (2)。

变数 pos 指向我们想要匹配的字符在寻找字符串的所在位置。如果 pos 等于这个字符串的长度，我们有一个完整的匹配，则我们将替换字符串写至输出流，并清空缓冲区 (3)。如果在这之前匹配失败，我们可以将缓冲区的第一个元素取出，并写至输出流，之后我们重置缓冲区，并从 pos 等于 0 重新开始 (4)。

```
(defun file-subst (old new file1 file2)
  (with-open-file (in file1 :direction :input)
     (with-open-file (out file2 :direction :output
                                :if-exists :supersede)
       (stream-subst old new in out))))

(defun stream-subst (old new in out)
  (let* ((pos 0)
         (len (length old))
         (buf (new-buf len))
         (from-buf nil))
    (do ((c (read-char in nil :eof)
            (or (setf from-buf (buf-next buf))
                (read-char in nil :eof))))
        ((eql c :eof))
      (cond ((char= c (char old pos))
             (incf pos)
             (cond ((= pos len)                ; 3
                    (princ new out)
                    (setf pos 0)
                    (buf-clear buf))
                   ((not from-buf)             ; 2
                    (buf-insert c buf))))
            ((zerop pos)                       ; 1
             (princ c out)
             (when from-buf
               (buf-pop buf)
               (buf-reset buf)))
            (t                                 ; 4
             (unless from-buf
               (buf-insert c buf))
             (princ (buf-pop buf) out)
             (buf-reset buf)
             (setf pos 0))))
    (buf-flush buf out)))
```

图 **7.2** 字符串替换

下列表格展示了当我们将文件中的 `"baro"` 替换成 `"baric"` 所发生的事，其中文件只有一个单字 `"barbarous"`：

CHARACTER	SOURCE	MATCH	CASE	OUTPUT	BUFFER
b	file	b	2		b
a	file	a	2		b a
r	file	r	2		b a r
b	file	o	4	b	b.a r b.
a	buffer	b	1	a	a.r b.
r	buffer	b	1	r	r.b.
b	buffer	b	1		r b:
a	file	a	2		r b:a
r	file	r	2		r b:a
o	file	o	3	baric	r b:a r
u	file	b	1	u	
a	file	b	1	s	

第一栏是当前字符 —— `c` 的值；第二栏显示是从缓冲区或是直接从输入流读取；第三栏显示需要匹配的字符 —— `old` 的第 **posth** 字符；第四栏显示那一个条件式 (case)被求值作为结果；第五栏显示被写至输出流的字符；而最后一栏显示缓冲区之后的内容。在最后一栏里， `used` 与 `new` 的位置一样，由一个冒号 (: colon)表示。

在文件 `"test1"` 里有如下文字：

```
The struggle between Liberty and Authority is the most conspicuous feature
in the portions of history with which we are earliest familiar, particularly
in that of Greece, Rome, and England.
```

在我们对 `(file-subst " th" " z" "test1" "test2")` 求值之后，读取文件 `"test2"` 为:

```
The struggle between Liberty and Authority is ze most conspicuous feature
in ze portions of history with which we are earliest familiar, particularly
in zat of Greece, Rome, and England.
```

为了使这个例子尽可能的简单，图 7.2 的代码只将一个字符串换成另一个字符串。很容易扩展为搜索一个模式而不是一个字面字符串。你只需要做的是，将 `char=` 调用换成一个你想要的更通用的匹配函数调用。

7.5 宏字符 (Macro Characters)

一个宏字符 (macro character)是获得 `read` 特别待遇的字符。比如小写的 `a` ，通常与小写 `b` 一样处理，但一个左括号就不同了: 它告诉 Lisp 开始读入一个列表。

一个宏字符或宏字符组合也称作 `read-macro` (读取宏) 。许多 Common Lisp 预定义的读取宏是缩写。比如说引用 (Quote): 读入一个像是 `'a` 的表达式时，它被读取器展开成 `(quote a)` 。当你输入引用的表达式 (quoted expression)至顶层时，它们在读入之时就会被求值，所以一般来说你看不到这样的转换。你可以透过显式调用 `read` 使其现形:

```
> (car (read-from-string "'a"))
QUOTE
```

引用对于读取宏来说是不寻常的，因为它用单一字符表示。有了一个有限的字符集，你可以在 Common Lisp 里有许多单一字符的读取宏，来表示一个或更多字符。

这样的读取宏叫做派发 (dispatching)读取宏，而第一个字符叫做派发字符 (dispatching character)。所有预定义的派发读取宏使用井号 (`#`)作为派发字符。我们已经见过好几个。举例来说， `#'` 是 `(function ...)` 的缩写，同样的 `'` 是 `(quote ...)` 的缩写。

其它我们见过的派发读取宏包括 `#(...)` ，产生一个向量； `#nA(...)` 产生数组； `#\` 产生一个字符； `#S(n ...)` 产生一个结构。当这些类型的每个对象被 `prin1` 显示时 (或是 `format` 搭配 `~S`)，它们使用对应的读取宏 [2] 。这表示着你可以写出或读回这样的对象:

```
> (let ((*print-array* t))
    (vectorp (read-from-string (format nil "~S"
                                       (vector 1 2)))))
T
```

当然我们拿回来的不是同一个向量，而是具有同样元素的新向量。

不是所有对象被显示时都有着清楚 (distinct)、可读的形式。举例来说，函数与哈希表，倾向于这样 `#<...>` 被显示。实际上 `#<...>` 也是一个读取宏，但是特别用来产生当遇到 `read` 的错误。函数与哈希表不能被写出与读回来，而这个读取宏确保使用者不会有这样的幻觉。 [3]

当你定义你自己的事物表示法时 (举例来说，结构的印出函数)，你要将此准则记住。要不使用一个可以被读回来的表示法，或是使用 `#<...>` 。

Chapter 7 总结 (Summary)

1. 流是输入的来源或终点。在字符流里，输入输出是由字符组成。
2. 缺省的流指向顶层。新的流可以由开启文件产生。
3. 你可以解析对象、字符组成的字符串、或是单独的字符。
4. `format` 函数提供了完整的输出控制。
5. 为了要替换文本文件中的字符串，你需要将字符读入缓冲区。
6. 当 `read` 遇到一个宏字符像是 `'`，它调用相关的函数。

Chapter 7 练习 (Exercises)

1. 定义一个函数，接受一个文件名并返回一个由字符串组成的列表，来表示文件里的每一行。
2. 定义一个函数，接受一个文件名并返回一个由表达式组成的列表，来表示文件里的每一行。
3. 假设有某种格式的文件文件，注解是由 `%` 字符表示。从这个字符开始直到行尾都会被忽略。定义一个函数，接受两个文件名称，并拷贝第一个文件的内容去掉注解，写至第二个文件。
4. 定义一个函数，接受一个二维浮点数组，将其用简洁的栏位显示。每个元素应印至小数点二位，一栏十个字符宽。（假设所有的字符可以容纳）。你会需要 `array-dimensions` (参见 361 页，译注: Appendix D)。
5. 修改 `stream-subst` 来允许万用字符 (wildcard) 可以在模式中使用。若字符 `+` 出现在 `old` 里，它应该匹配任何输入字符。
6. 修改 `stream-subst` 来允许模式可以包含一个用来匹配任何数字的元素，以及一个可以匹配任何英文字符的元素或是一个可以匹配任何字符的元素。模式必须可以匹配任何特定的输入字符。(提示: `old` 可以不是一个字符串。)

脚注

[1] 你可以给一个字符串取代路径名，但这样就不可携了 (portable)。
[2] 要让向量与数组这样被显示，将 `*print-array*` 设为真。
[3] Lisp 不能只用 `#'` 来表示函数，因为 `#'` 本身无法提供表示闭包的方式。

第八章：符号

我们一直在使用符号。符号，在看似简单的表面之下，又好像没有那么简单。起初最好不要纠结于背后的实现机制。可以把符号当成数据对象与名字那样使用，而不需要理解两者是如何关联起来的。但到了某个时间点，停下来思考背后是究竟是如何工作会是很有用的。本章解释了背后实现的细节。

8.1 符号名 (Symbol Names)

第二章描述过，符号是变量的名字，符号本身以对象所存在。但 Lisp 符号的可能性，要比在多数语言仅允许作为变量名来得广泛许多。实际上，符号可以用任何字符串当作名字。可以通过调用 `symbol-name` 来获得符号的名字：

```
> (symbol-name 'abc)
"ABC"
```

注意到这个符号的名字，打印出来都是大写字母。缺省情况下， Common Lisp 在读入时，会把符号名字所有的英文字母都转成大写。代表 Common Lisp 缺省是不分大小写的：

```
> (eql 'abc 'Abc)
T
> (CaR '(a b c))
A
```

一个名字包含空白，或其它可能被读取器认为是重要的字符的符号，要用特殊的语法来引用。任何存在垂直杠 (vertical bar)之间的字符序列将被视为符号。可以如下这般在符号的名字中，放入任何字符：

```
> (list '|Lisp 1.5| '|| '|abc| '|ABC|)
(|Lisp 1.5| || |abc| ABC)
```

当这种符号被读入时，不会有大小写转换，而宏字符与其他的字符被视为一般字符。

那什么样的符号不需要使用垂直杠来参照呢？基本上任何不是数字，或不包含读取器视为重要的字符的符号。一个快速找出你是否可以不用垂直杠来引用符号的方法，是看看 Lisp 如何印出它的。如果 Lisp 没有用垂直杠表示一个符号，如上述列表的最后一个，那么你也可以不用垂直杠。

记得，垂直杠是一种表示符号的特殊语法。它们不是符号的名字之一：

```
> (symbol-name '|a b c|)
"a b c"
```

(如果想要在符号名称内使用垂直杠，可以放一个反斜线在垂直杠的前面。)

译注: 反斜线是 \ (backslash)。

8.2 属性列表 (Property Lists)

在 Common Lisp 里，每个符号都有一个属性列表（property-list）或称为 `plist` 。函数 `get` 接受符号及任何类型的键值，然后返回在符号的属性列表中，与键值相关的数值：

```
> (get 'alizarin 'color)
NIL
```

它使用 `eql` 来比较各个键。若某个特定的属性没有找到时， `get` 返回 `nil` 。

要将值与键关联起来时，你可以使用 `setf` 及 `get` :

```
> (setf (get 'alizarin 'color) 'red)
RED
> (get 'alizarin 'color)
RED
```

现在符号 `alizarin` 的 `color` 属性是 `red` 。

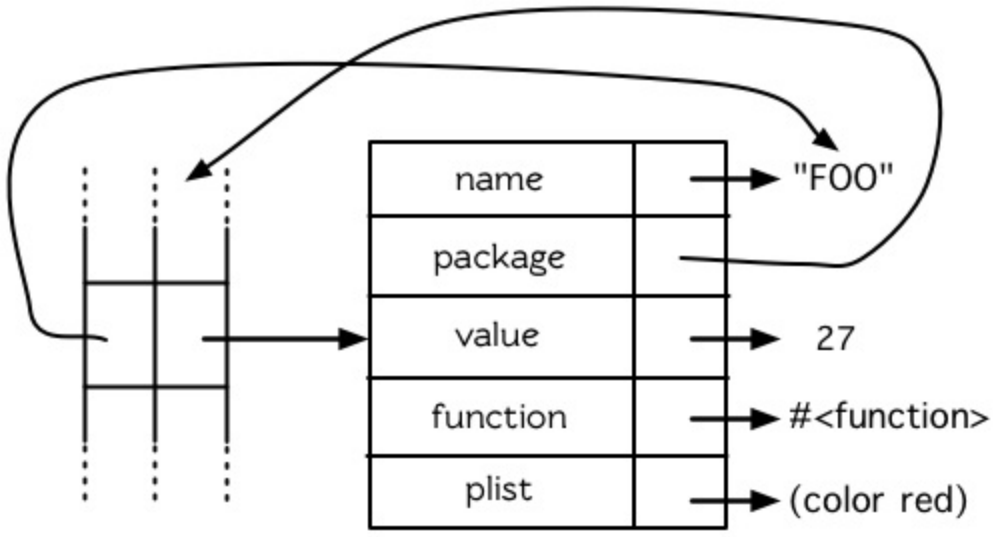

图 **8.1** 符号的结构

```
> (setf (get 'alizarin 'transparency) 'high)
HIGH
```

```
> (symbol-plist 'alizarin)
(TRANSPARENCY HIGH COLOR RED)
```

注意，属性列表不以关联列表（assoc-lists）的形式表示，虽然用起来感觉是一样的。

在 Common Lisp 里，属性列表用得不多。他们大部分被哈希表取代了（4.8 小节）。

8.3 符号很不简单 (Symbols Are Big)

当我们输入名字时，符号就被悄悄地创建出来了，而当它们被显示时，我们只看的到符号的名字。某些情况下，把符号想成是表面所见的东西就好，别想太多。但有时候符号不像看起来那么简单。

从我们如何使用和检查符号的方式来看，符号像是整数那样的小对象。而符号实际上确实是一个对象，差不多像是由 `defstruct` 定义的那种结构。符号可以有名字、主包（home package）、作为变量的值、作为函数的值以及带有一个属性列表。图 8.1 演示了符号在内部是如何表示的。

很少有程序会使用很多符号，以致于值得用其它的东西来代替符号以节省空间。但需要记住的是，符号是实际的对象，不仅是名字而已。当两个变量设成相同的符号时，与两个变量设成相同列表一样：两个变量的指针都指向同样的对象。

8.4 创建符号 (Creating Symbols)

8.1 节演示了如何取得符号的名字。另一方面，用字符串生成符号也是有可能的。但比较复杂一点，因为我们需要先介绍包（package）。

概念上来说，包是将名字映射到符号的符号表（symbol-tables）。每个普通的符号都属于一个特定的包。符号属于某个包，我们称为符号被包扣押（intern）了。函数与变量用符号作为名称。包借由限制哪个符号可以访问来实现模块化（modularity），也是因为这样，我们才可以引用到函数与变量。

大多数的符号在读取时就被扣押了。在第一次输入一个新符号的名字时，Lisp 会产生一个新的符号对象，并将它扣押到当下的包里（缺省是 `common-lisp-user` 包)。但也可以通过给入字符串与选择性包参数给 `intern` 函数，来扣押一个名称为字符串名的符号:

```
> (intern "RANDOM-SYMBOL")
RANDOM-SYMBOL
NIL
```

选择性包参数缺省是当前的包，所以前述的表达式，返回当前包里的一个符号，此符号

的名字是 “RANDOM-SYMBOL”，若此符号尚未存在时，会创建一个这样的符号出来。第二个返回值告诉我们符号是否存在；在这个情况，它不存在。

不是所有的符号都会被扣押。有时候有一个自由的（uninterned）符号是有用的，这和公用电话本是一样的原因。自由的符号叫做 *gensyms* 。我们将会在第 10 章讨论宏（Macro）时，理解 `gensym` 的作用。

8.5 多重包 (Multiple Packages)

大的程序通常切分为多个包。如果程序的每个部分都是一个包，那么开发程序另一个部分的某个人，将可以使用符号来作为函数名或变量名，而不必担心名字在别的地方已经被用过了。

在没有提供定义多个命名空间的语言里，工作于大项目的程序员，通常需要想出某些规范（convention），来确保他们不会使用同样的名称。举例来说，程序员写显示相关的代码（display code）可能用 `disp_` 开头的名字，而写数学相关的代码（math code）的程序员仅使用由 `math_` 开始的代码。所以若是数学相关的代码里，包含一个做快速傅立叶转换的函数时，可能会叫做 `math_fft` 。

包不过是提供了一种便捷方式来自动办到此事。如果你将函数定义在单独的包里，可以随意使用你喜欢的名字。只有你明确导出（ `export` ）的符号会被别的包看到，而通常前面会有包的名字(或修饰符)。

举例来说，假设一个程序分为两个包， `math` 与 `disp` 。如果符号 `fft` 被 `math` 包导出，则 `disp` 包里可以用 `math:fft` 来参照它。在 `math` 包里，可以只用 `fft` 来参照。

下面是你可能会放在文件最上方，包含独立包的代码：

```
(defpackage "MY-APPLICATION"
            (:use "COMMON-LISP" "MY-UTILITIES")
            (:nicknames "APP")
            (:export "WIN" "LOSE" "DRAW"))

(in-package my-application)
```

`defpackage` 定义一个新的包叫做 `my-application` [1] 它使用了其他两个包， `common-lisp` 与 `my-utilities` ，这代表着可以不需要用包修饰符（package qualifiers）来存取这些包所导出的符号。许多包都使用了 `common-lisp` 包 —— 因为你不会想给 Lisp 自带的操作符与变量再加上修饰符。

`my-application` 包本身只输出三个符号: `WIN` 、 `LOSE` 以及 `DRAW` 。由于调用 `defpackage` 给了 `my-application` 一个匿称 `app` ，则别的包可以这样引用到这些符号，比如

app:win 。

defpackage 伴随着一个 in-package ，确保当前包是 my-application 。所有其它未修饰的符号会被扣押至 my-application —— 除非之后有别的 in-package 出现。当一个文件被载入时，当前的包总是被重置成载入之前的值。

8.6 关键字 (Keywords)

在 keyword 包的符号 (称为关键字)有两个独特的性质：它们总是对自己求值，以及可以在任何地方引用它们，如 :x 而不是 keyword:x 。我们首次在 44 页 (译注: 3.10 小节）介绍关键字参数时， (member '(a) '((a) (z)) test: #'equal) 比 (member '(a) '((a) (z)) :test #'equal) 读起来更自然。现在我们知道为什么第二个较别扭的形式才是对的。 test 前的冒号字首，是关键字的识别符。

为什么使用关键字而不用一般的符号？因为关键字在哪都可以存取。一个函数接受符号作为实参，应该要写成预期关键字的函数。举例来说，这个函数可以安全地在任何包里调用:

```
(defun noise (animal)
  (case animal
    (:dog :woof)
    (:cat :meow)
    (:pig :oink)))
```

但如果是用一般符号写成的话，它只在被定义的包内正常工作，除非关键字也被导出了。

8.7 符号与变量 (Symbols and Variables)

Lisp 有一件可能会使你困惑的事情是，符号与变量的从两个非常不同的层面互相关联。当符号是特别变量（special variable）的名字时，变量的值存在符号的 value 栏位（图 8.1）。 symbol-value 函数引用到那个栏位，所以在符号与特殊变量的值之间，有直接的连接关系。

而对于词法变量（lexical variables）来说，事情就完全不一样了。一个作为词法变量的符号只不过是个占位符（placeholder）。编译器会将其转为一个寄存器（register）或内存位置的引用位址。在最后编译出来的代码中，我们无法追踪这个符号 (除非它被保存在调试器「debugger」的某个地方)。因此符号与词法变量的值之间是没有连接的；只要一有值，符号就消失了。

8.8 示例：随机文本 (Example: Random Text)

如果你要写一个操作单词的程序，通常使用符号会比字符串来得好，因为符号概念上是原子性的（atomic）。符号可以用 `eql` 一步比较完成，而字符串需要使用 `string=` 或 `string-equal` 逐一字符做比较。作为一个示例，本节将演示如何写一个程序来产生随机文本。程序的第一部分会读入一个示例文件（越大越好），用来累积之后所给入的相关单词的可能性（likeilhood）的信息。第二部分在每一个单词都根据原本的示例，产生一个随机的权重（weight）之后，随机走访根据第一部分所产生的网络。

产生的文字将会是部分可信的（locally plausible），因为任两个出现的单词也是输入文件里，两个同时出现的单词。令人惊讶的是，获得看起来是 —— 有意义的整句 —— 甚至整个段落是的频率相当高。

图 8.2 包含了程序的上半部，用来读取示例文件的代码。

```
(defparameter *words* (make-hash-table :size 10000))

(defconstant maxword 100)

(defun read-text (pathname)
  (with-open-file (s pathname :direction :input)
    (let ((buffer (make-string maxword))
          (pos 0))
      (do ((c (read-char s nil :eof)
              (read-char s nil :eof)))
          ((eql c :eof))
        (if (or (alpha-char-p c) (char= c #\'))
            (progn
              (setf (aref buffer pos) c)
              (incf pos))
            (progn
              (unless (zerop pos)
                (see (intern (string-downcase
                              (subseq buffer 0 pos))))
                (setf pos 0))
              (let ((p (punc c)))
                (if p (see p)))))))))

(defun punc (c)
  (case c
    (#\. '|.|) (#\, '|,|) (#\; '|;|)
    (#\! '|!|) (#\? '|?|) ))

(let ((prev `|.|))
  (defun see (symb)
    (let ((pair (assoc symb (gethash prev *words*))))
      (if (null pair)
          (push (cons symb 1) (gethash prev *words*))
```

```
            (incf (cdr pair))))
      (setf prev symb)))
```

图 **8.2** 读取示例文件

从图 8.2 所导出的数据，会被存在哈希表 `*words*` 里。这个哈希表的键是代表单词的符号，而值会像是下列的关联列表（assoc-lists）：

```
((|sin| . 1) (|wide| . 2) (|sights| . 1))
```

使用弥尔顿的失乐园 [http://zh.wikipedia.org/wiki/%E5%A4%B1%E6%A8%82%E5%9C%92]作为示例文件时，这是与键 `|discover|` 有关的值。它指出了 “discover” 这个单词，在诗里面用了四次，与 “wide” 用了两次，而 “sin” 与 ”sights” 各一次。(译注: 诗可以在这里找到 http://www.paradiselost.org/)

函数 `read-text` 累积了这个信息。这个函数接受一个路径名（pathname），然后替每一个出现在文件中的单词，生成一个上面所展示的关联列表。它的工作方式是，逐字读取文件的每个字符，将累积的单词存在字符串 `buffer` 。 `maxword` 设成 `100` ，程序可以读取至多 100 个单词，对英语来说足够了。

只要下个字符是一个字（由 `alpha-char-p` 决定）或是一撇 (apostrophe)，就持续累积字符。任何使单词停止累积的字符会送给 `see` 。数种标点符号（punctuation）也被视为是单词；函数 `punc` 返回标点字符的伪单词（pseudo-word）。

函数 `see` 注册每一个我们看过的单词。它需要知道前一个单词，以及我们刚确认过的单词 —— 这也是为什么要有变量 `prev` 存在。起初这个变量设为伪单词里的句点；在 `see` 函数被调用后， `prev` 变量包含了我们最后见过的单词。

在 `read-text` 返回之后， `*words*` 会包含输入文件的每一个单词的条目（entry）。通过调用 `hash-table-count` 你可以了解有多少个不同的单词存在。鲜少有英文文件会超过 10000 个单词。

现在来到了有趣的部份。图 8.3 包含了从图 8.2 所累积的数据来产生文字的代码。 `generate-text` 函数导出整个过程。它接受一个要产生几个单词的数字，以及选择性传入前一个单词。使用缺省值，会让产生出来的文件从句子的开头开始。

```
(defun generate-text (n &optional (prev '|.|))
  (if (zerop n)
      (terpri)
      (let ((next (random-next prev)))
        (format t "~A " next)
        (generate-text (1- n) next))))
```

```
(defun random-next (prev)
  (let* ((choices (gethash prev *words*))
         (i (random (reduce #'+ choices
                            :key #'cdr))))
    (dolist (pair choices)
      (if (minusp (decf i (cdr pair)))
          (return (car pair))))))
```

图 **8.3** 产生文字

要取得一个新的单词， `generate-text` 使用前一个单词，接著调用 `random-next` 。 `random-next` 函数根据每个单词出现的机率加上权重，随机选择伴随输入文本中 `prev` 之后的单词。

现在会是测试运行下程序的好时机。但其实你早看过一个它所产生的示例： 就是本书开头的那首诗，是使用弥尔顿的失乐园作为输入文件所产生的。

(译注: 诗可在这里看，或是浏览书的第 vi 页)

Half lost on my firmness gains more glad heart,

Or violent and from forage drives

A glimmering of all sun new begun

Both harp thy discourse they match’d,

Forth my early, is not without delay;

For their soft with whirlwind; and balm.

Undoubtedly he scornful turn’d round ninefold,

Though doubled now what redounds,

And chains these a lower world devote, yet inflicted?

Till body or rare, and best things else enjoy’d in heav’n

To stand divided light at ev’n and poise their eyes,

Or nourish, lik’ning spiritual, I have thou appear.

— Henley

Chapter 8 总结 (Summary)

1. 符号的名字可以是任何字符串，但由 `read` 创建的符号缺省会被转成大写。
2. 符号带有相关联的属性列表，虽然他们不需要是相同的形式，但行为像是 assoc-lists 。
3. 符号是实质的对象，比较像结构，而不是名字。
4. 包将字符串映射至符号。要在包里给符号创造一个条目的方法是扣留它。符号不需要被扣留。
5. 包通过限制可以引用的名称增加模块化。缺省的包会是 user 包，但为了提高模块化，大的程序通常分成数个包。
6. 可以让符号在别的包被存取。关键字是自身求值并在所有的包里都可以存取。
7. 当一个程序用来操作单词时，用符号来表示单词是很方便的。

Chapter 8 练习 (Exercises)

1. 可能有两个同名符号，但却不 `eql` 吗？
2. 估计一下用字符串表示 “FOO” 与符号表示 foo 所使用内存空间的差异。
3. 只使用字符串作为实参 来调用 137 页的 `defpackage` 。应该使用符号比较好。为什么使用字符串可能比较危险呢？
4. 加入需要的代码，使图 7.1 的代码可以放在一个叫做 `"RING"` 的包里，而图 7.2 的代码放在一个叫做 `"FILE"` 包里。不需要更动现有的代码。
5. 写一个确认引用的句子是否是由 Henley 生成的程序 (8.8 节)。
6. 写一版 Henley，接受一个单词，并产生一个句子，该单词在句子的中间。

脚注

[1] 调用 `defpackage` 里的名字全部大写的缘故在 8.1 节提到过，符号的名字缺省被转成大写。

第九章：数字

处理数字是 Common Lisp 的强项之一。Common Lisp 有着丰富的数值类型，而 Common Lisp 操作数字的特性与其他语言比起来更受人喜爱。

9.1 类型 (Types)

Common Lisp 提供了四种不同类型的数字：整数、浮点数、比值与复数。本章所讲述的函数适用于所有类型的数字。有几个不能用在复数的函数会特别说明。

整数写成一串数字：如 `2001` 。浮点数是可以写成一串包含小数点的数字，如 `253.72` ，或是用科学表示法，如 `2.5372e2` 。比值是写成由整数组成的分数：如 `2/3` 。而复数 `a+bi` 写成 `#c(a b)` ，其中 `a` 与 `b` 是任两个类型相同的实数。

谓词 `integerp` 、 `floatp` 以及 `complexp` 针对相应的数字类型返回真。图 9.1 展示了数值类型的层级。

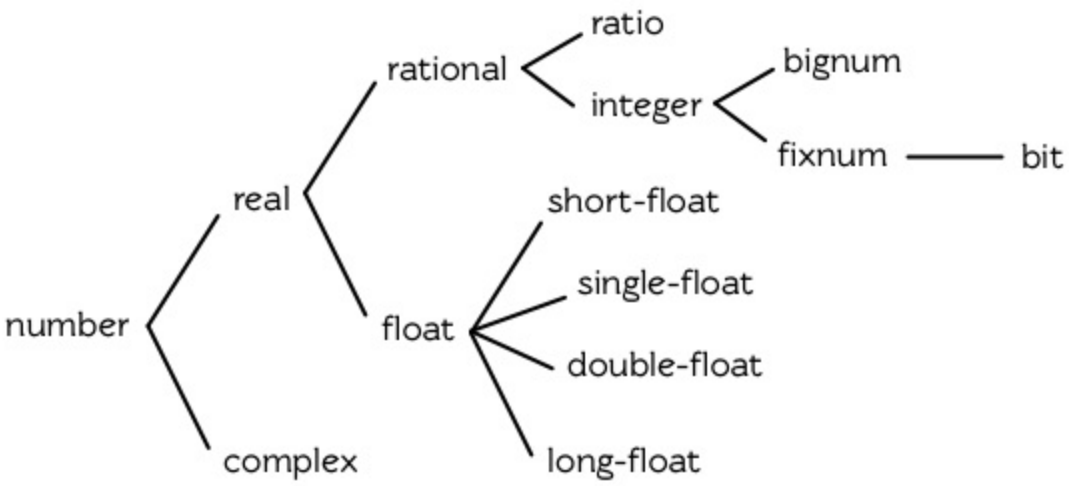

图 **9.1:** 数值类型

要决定计算过程会返回何种数字，以下是某些通用的经验法则：

1. 如果数值函数接受一个或多个浮点数作为参数，则返回值会是浮点数 (或是由浮点数组成的复数)。所以 `(+ 1.0 2)` 求值为 `3.0` ，而 `(+ #c(0 1.0) 2)` 求值为 `#c(2.0 1.0)` 。
2. 可约分的比值会被转换成最简分数。所以 `(/ 10 2)` 会返回 `5` 。
3. 若计算过程中复数的虚部变成 `0` 时，则复数会被转成实数 。所以 `(+ #c(1 -1) #c(2 1))` 求值成 `3` 。

第二、第三个规则可以在读入参数时直接应用，所以：

```
> (list (ratiop 2/2) (complexp #c(1 0)))
(NIL NIL)
```

9.2 转换及取出 (Conversion and Extraction)

Lisp 提供四种不同类型的数字的转换及取出位数的函数。函数 `float` 将任何实数转换成浮点数:

```
> (mapcar #'float '(1 2/3 .5))
(1.0 0.6666667 0.5)
```

将数字转成整数未必需要转换，因为它可能牵涉到某些资讯的丧失。函数 `truncate` 返回任何实数的整数部分:

```
> (truncate 1.3)
1
0.29999995
```

第二个返回值 `0.29999995` 是传入的参数减去第一个返回值。(会有 0.00000005 的误差是因为浮点数的计算本身就不精确。)

函数 `floor` 与 `ceiling` 以及 `round` 也从它们的参数中导出整数。使用 `floor` 返回小于等于其参数的最大整数，而 `ceiling` 返回大于或等于其参数的最小整数，我们可以将 `mirror?` (46 页，译注: 3.11 节)改成可以找出所有回文（palindromes）的版本:

```
(defun palindrome? (x)
  (let ((mid (/ (length x) 2)))
    (equal (subseq x 0 (floor mid))
           (reverse (subseq x (ceiling mid))))))
```

和 `truncate` 一样， `floor` 与 `ceiling` 也返回传入参数与第一个返回值的差，作为第二个返回值。

```
> (floor 1.5)
1
0.5
```

实际上，我们可以把 `truncate` 想成是这样定义的:

```
(defun our-truncate (n)
    (if (> n 0)
        (floor n)
```

```
        (ceiling n)))
```

函数 round 返回最接近其参数的整数。当参数与两个整数的距离相等时， Common Lisp 和很多程序语言一样，不会往上取（round up）整数。而是取最近的偶数：

```
> (mapcar #'round '(-2.5 -1.5 1.5 2.5))
(-2 -2 2 2)
```

在某些数值应用中这是好事，因为舍入误差（rounding error）通常会互相抵消。但要是用户期望你的程序将某些值取整数时，你必须自己提供这个功能。 [1] 与其他的函数一样， round 返回传入参数与第一个返回值的差，作为第二个返回值。

函数 mod 仅返回 floor 返回的第二个返回值；而 rem 返回 truncate 返回的第二个返回值。我们在 94 页（译注： 5.7 节）曾使用 mod 来决定一个数是否可被另一个整除，以及 127 页（译注： 7.4 节）用来找出环状缓冲区（ring buffer）中，元素实际的位置。

关于实数，函数 signum 返回 1 、 0 或 -1 ，取决于它的参数是正数、零或负数。函数 abs 返回其参数的绝对值。因此 (* (abs x) (signum x)) 等于 x 。

```
> (mapcar #'signum '(-2 -0.0 0.0 0 .5 3))
(-1 -0.0 0.0 0 1.0 1)
```

在某些应用里， -0.0 可能自成一格（in its own right），如上所示。实际上功能上几乎没有差别，因为数值 -0.0 与 0.0 有着一样的行为。

比值与复数概念上是两部分的结构。(译注：像 **Cons** 这样的两部分结构) 函数 numerator 与 denominator 返回比值或整数的分子与分母。（如果数字是整数，前者返回该数，而后者返回 1 。）函数 realpart 与 imgpart 返回任何数字的实数与虚数部分。（如果数字不是复数，前者返回该数字，后者返回 0 。）

函数 random 接受一个整数或浮点数。这样形式的表达式 (random n) ，会返回一个大于等于 0 并小于 n 的数字，并有着与 n 相同的类型。

9.3 比较 (Comparison)

谓词 = 比较其参数，当数值上相等时 —— 即两者的差为零时，返回真。

```
> (= 1 1.0)
T
> (eql 1 1.0)
NIL
```

= 比起 eql 来得宽松，但参数的类型需一致。

用来比较数字的谓词为 < （小于）、 <= （小于等于）、 = （等于）、 >= （大于等于）、 > (大于) 以及 /= （不相等）。以上所有皆接受一个或多个参数。只有一个参数时，它们全返回真。

```
(<= w x y z)
```

等同于二元操作符的结合（conjunction），应用至每一对参数上:

```
(and (<= w x) (<= x y) (<= y z))
```

由于 /= 若它的两个参数不等于时会返回真，表达式

```
(/= w x y z)
```

等同于

```
(and (/= w x) (/= w y) (/= w z)
     (/= x y) (/= y z) (/= y z))
```

特殊的谓词 zerop 、 plusp 与 minusp 接受一个参数，分别于参数 = 、 > 、 < 零时，返回真。虽然 -0.0 （如果实现有使用它）前面有个负号，但它 = 零，

```
> (list (minusp -0.0) (zerop -0.0))
(NIL T)
```

因此对 -0.0 使用 zerop ，而不是 minusp 。

谓词 oddp 与 evenp 只能用在整数。前者只对奇数返回真，后者只对偶数返回真。

本节定义的谓词中，只有 = 、 /= 与 zerop 可以用在复数。

函数 max 与 min 分别返回其参数的最大值与最小值。两者至少需要给一个参数:

```
> (list (max 1 2 3 4 5) (min 1 2 3 4 5))
(5 1)
```

如果参数含有浮点数的话，结果的类型取决于各家实现。

9.4 算术 (Arithematic)

用来做加减的函数是 + 与 - 。两者皆接受任何数量的参数，包括没有参数，在没有参数的情况下返回 0 。（译注: - 在没有参数的情况下会报错，至少要一个参数）一个这样形式的表达式 (- n) 返回 -n 。一个这样形式的表达式

```
(- x y z)
```

等同于

```
(- (- x y) z)
```

有两个函数 1+ 与 1- ，分别将参数加 1 与减 1 后返回。 1- 有一点误导，因为 (1- x) 返回 x-1 而不是 1-x 。

宏 incf 及 decf 分别递增与递减数字。这样形式的表达式 (incf x n) 类似于 (setf x (+ x n)) 的效果，而 (decf x n) 类似于 (setf x (- x n)) 的效果。这两个形式里，第二个参数皆是选择性给入的，缺省值为 1 。

用来做乘法的函数是 * 。接受任何数量的参数。没有参数时返回 1 。否则返回参数的乘积。

除法函数 / 至少要给一个参数。这样形式的调用 (/ n) 等同于 (/ 1 n) ，

```
> (/ 3)
1/3
```

而这样形式的调用

```
(/ x y z)
```

等同于

```
(/ (/ x y) z)
```

注意 - 与 / 两者在这方面的相似性。

当给定两个整数时， / 若第一个不是第二个的倍数时，会返回一个比值:

```
> (/ 365 12)
365/12
```

举例来说，如果你试着找出平均每一个月有多长，可能会有解释器在逗你玩的感觉。在这个情况下，你需要的是，对比值调用 float ，而不是对两个整数做 / 。

```
> (float 365/12)
30.416666
```

9.5 指数 (Exponentiation)

要找到 \(x^n\) 调用 `(expt x n)`，

```
> (expt 2 5)
32
```

而要找到 \(log_nx\) 调用 `(log x n)`：

```
> (log 32 2)
5.0
```

通常返回一个浮点数。

要找到 \(e^x\) 有一个特别的函数 `exp`，

```
> (exp 2)
7.389056
```

而要找到自然对数，你可以使用 `log` 就好，因为第二个参数缺省为 `e`：

```
> (log 7.389056)
2.0
```

要找到立方根，你可以调用 `expt` 用一个比值作为第二个参数，

```
> (expt 27 1/3)
3.0
```

但要找到平方根，函数 `sqrt` 会比较快:

```
> (sqrt 4)
2.0
```

9.6 三角函数 (Trigometric Functions)

常量 `pi` 是 `π` 的浮点表示法。它的精度取决于各家实现。函数 `sin` 、 `cos` 及 `tan` 分别可以找到正弦、余弦及正交函数，其中角度以径度表示：

```
> (let ((x (/ pi 4)))
    (list (sin x) (cos x) (tan x)))
(0.7071067811865475d0 0.7071067811865476d0 1.0d0)
;;; 译注: CCL 1.8  SBCL 1.0.55 下的结果是
;;; (0.7071067811865475D0 0.7071067811865476D0 0.9999999999999999D0)
```

这些函数都接受负数及复数参数。

函数 `asin` 、 `acos` 及 `atan` 实现了正弦、余弦及正交的反函数。参数介于 `-1` 与 `1` 之间（包含）时， `asin` 与 `acos` 返回实数。

双曲正弦、双曲余弦及双曲正交分别由 `sinh` 、 `cosh` 及 `tanh` 实现。它们的反函数同样为 `asinh` 、 `acosh` 以及 `atanh` 。

9.7 表示法 (Representations)

Common Lisp 没有限制整数的大小。可以塞进一个字（word）内存的小整数称为定长数(fixnums)。在计算过程中，整数无法塞入一个字时，Lisp 切换至使用多个字的表示法（一个大数「bignum」）。所以整数的大小限制取决于实体内存，而不是语言。

常量 `most-positive-fixnum` 与 `most-negative-fixnum` 表示一个实现不使用大数所可表示的最大与最小的数字大小。在很多实现里，它们为：

```
> (values most-positive-fixnum most-negative-fixnum)
536870911
-536870912
;;; 译注: CCL 1.8 的结果为
1152921504606846975
-1152921504606846976
;;; SBCL 1.0.55 的结果为
4611686018427387903
-4611686018427387904
```

谓词 `typep` 接受一个参数及一个类型名称，并返回指定类型的参数。所以，

```
> (typep 1 'fixnum)
T
> (type (1+ most-positive-fixnum) 'bignum)
T
```

浮点数的数值限制是取决于各家实现的。 Common Lisp 提供了至多四种类型的浮点数：短浮点 `short-float` 、 单浮点 `single-float` 、双浮点 `double-float` 以及长浮点 `long-float` 。Common Lisp 的实现是不需要用不同的格式来表示这四种类型（很少有实现这么干）。

一般来说，短浮点应可塞入一个字，单浮点与双浮点提供普遍的单精度与双精度浮点数的概念，而长浮点，如果想要的话，可以是很大的数。但实现可以不对这四种类型做区别，也是完全没有问题的。

你可以指定你想要何种格式的浮点数，当数字是用科学表示法时，可以通过将 `e` 替换为 `s f d l` 来得到不同的浮点数。（你也可以使用大写，这对长浮点来说是个好主意，因为 `l` 看起来太像 `1` 了。）所以要表示最大的 `1.0` 你可以写 `1L0` 。

（译注: `s` 为短浮点、 `f` 为单浮点、 `d` 为双浮点、 `l` 为长浮点。）

在给定的实现里，用十六个全局常量标明了每个格式的限制。它们的名字是这种形式: `m-s-f` ，其中 `m` 是 `most` 或 `least` ， `s` 是 `positive` 或 `negative` ，而 `f` 是四种浮点数之一。 λ [http://acl.readthedocs.org/en/latest/zhCN/notes-cn.html#notes-150]

浮点数下溢（underflow）与溢出（overflow），都会被 Common Lisp 视为错误 :

```
> (* most-positive-long-float 10)
Error: floating-point-overflow
```

9.8 范例：追踪光线 (Example: Ray-Tracing)

作为一个数值应用的范例，本节示范了如何撰写一个光线追踪器 (ray-tracer)。光线追踪是一个高级的 (deluxe)渲染算法: 它产生出逼真的图像，但需要花点时间。

要产生一个 3D 的图像，我们至少需要定义四件事: 一个观测点 (eye)、一个或多个光源、一个由一个或多个平面所组成的模拟世界 (simulated world)，以及一个作为通往这个世界的窗户的平面 (图像平面「image plane」)。我们产生出的是模拟世界投影在图像平面区域的图像。

光线追踪独特的地方在于，我们如何找到这个投影: 我们一个一个像素地沿着图像平面走，追踪回到模拟世界里的光线。这个方法带来三个主要的优势: 它让我们容易得到现实世界的光学效应 (optical effect)，如透明度 (transparency)、反射光 (reflected light)以及产生阴影 (cast shadows)；它让我们可以直接用任何我们想要的几何的物体，来定义出模拟的世界，而不需要用多边形 (polygons)来建构它们；以及它很简单实现。

```
(defun sq (x) (* x x))

(defun mag (x y z)
  (sqrt (+ (sq x) (sq y) (sq z))))

(defun unit-vector (x y z)
  (let ((d (mag x y z)))
    (values (/ x d) (/ y d) (/ z d))))
```

```
(defstruct (point (:conc-name nil))
  x y z)

(defun distance (p1 p2)
  (mag (- (x p1) (x p2))
       (- (y p1) (y p2))
       (- (z p1) (z p2))))

(defun minroot (a b c)
  (if (zerop a)
      (/ (- c) b)
      (let ((disc (- (sq b) (* 4 a c))))
        (unless (minusp disc)
          (let ((discrt (sqrt disc)))
            (min (/ (+ (- b) discrt) (* 2 a))
                 (/ (- (- b) discrt) (* 2 a))))))))
```

图 **9.2** 实用数学函数

图 9.2 包含了我们在光线追踪器里会需要用到的一些实用数学函数。第一个 `sq` ，返回其参数的平方。下一个 `mag` ，返回一个给定 `x y z` 所组成向量的大小 (magnitude)。这个函数被接下来两个函数用到。我们在 `unit-vector` 用到了，此函数返回三个数值，来表示与单位向量有着同样方向的向量，其中向量是由 `x y z` 所组成的:

```
> (multiple-value-call #'mag (unit-vector 23 12 47))
1.0
```

我们在 `distance` 也用到了 `mag` ，它返回三维空间中，两点的距离。（定义 `point` 结构来有一个 `nil` 的 `conc-name` 意味着栏位存取的函数会有跟栏位一样的名字: 举例来说，`x` 而不是 `point-x` 。）

最后 `minroot` 接受三个实数， `a` , `b` 与 `c` ，并返回满足等式 \(ax^2+bx+c=0\) 的最小实数 `x` 。当 `a` 不为 \(0\) 时，这个等式的根由下面这个熟悉的式子给出:

\[x = \dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}\]

图 9.3 包含了定义一个最小光线追踪器的代码。 它产生通过单一光源照射的黑白图像，与观测点 (eye)处于同个位置。 (结果看起来像是闪光摄影术 (flash photography)拍出来的)

`surface` 结构用来表示模拟世界中的物体。更精确的说，它会被 `included` 至定义具体类型物体的结构里，像是球体 (spheres)。 `surface` 结构本身只包含一个栏位: 一个 `color` 范围从 0 (黑色) 至 1 (白色)。

```
(defstruct surface color)
```

```
(defparameter *world* nil)
(defconstant eye (make-point :x 0 :y 0 :z 200))

(defun tracer (pathname &optional (res 1))
  (with-open-file (p pathname :direction :output)
    (format p "P2 ~A ~A 255" (* res 100) (* res 100))
    (let ((inc (/ res)))
      (do ((y -50 (+ y inc)))
          ((< (- 50 y) inc))
        (do ((x -50 (+ x inc)))
            ((< (- 50 x) inc))
          (print (color-at x y) p))))))

(defun color-at (x y)
  (multiple-value-bind (xr yr zr)
                       (unit-vector (- x (x eye))
                                    (- y (y eye))
                                    (- 0 (z eye)))
    (round (* (sendray eye xr yr zr) 255))))

(defun sendray (pt xr yr zr)
  (multiple-value-bind (s int) (first-hit pt xr yr zr)
    (if s
        (* (lambert s int xr yr zr) (surface-color s))
        0)))

(defun first-hit (pt xr yr zr)
  (let (surface hit dist)
    (dolist (s *world*)
      (let ((h (intersect s pt xr yr zr)))
        (when h
          (let ((d (distance h pt)))
            (when (or (null dist) (< d dist))
              (setf surface s hit h dist d))))))
    (values surface hit)))

(defun lambert (s int xr yr zr)
  (multiple-value-bind (xn yn zn) (normal s int)
    (max 0 (+ (* xr xn) (* yr yn) (* zr zn)))))
```

图 **9.3** 光线追踪。

图像平面会是由 x 轴与 y 轴所定义的平面。观测者 (eye) 会在 z 轴，距离原点 200 个单位。所以要在图像平面可以被看到，插入至 `*worlds*` 的表面 (一开始为 `nil`)会有着负的 z 座标。图 9.4 说明了一个光线穿过图像平面上的一点，并击中一个球体。

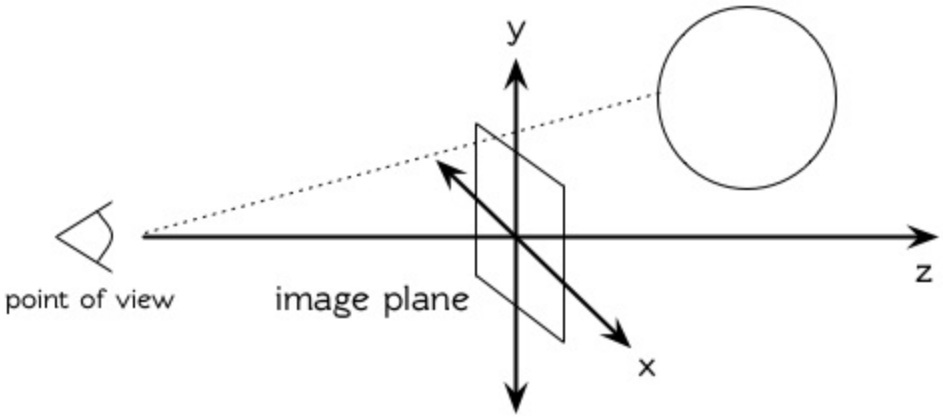

图 **9.4:** 追踪光线。

函数 `tracer` 接受一个路径名称，并写入一张图片至对应的文件。图片文件会用一种简单的 ASCII 称作 PGM 的格式写入。默认情况下，图像会是 100x100 。我们 PGM 文件的标头 (headers) 会由标签 `P2` 组成，伴随着指定图片宽度 (breadth)与高度 (height)的整数，初始为 100，单位为 pixel，以及可能的最大值 (255)。文件剩余的部份会由 10000 个介于 0 (黑)与 1 (白)整数组成，代表着 100 条 100 像素的水平线。

图片的解析度可以通过给入明确的 `res` 来调整。举例来说，如果 `res` 是 `2` ，则同样的图像会被渲染成 200x200 。

图片是一个在图像平面 100x100 的正方形。每一个像素代表着穿过图像平面抵达观测点的光的数量。要找到每个像素光的数量， `tracer` 调用 `color-at` 。这个函数找到从观测点至该点的向量，并调用 `sendray` 来追踪这个向量回到模拟世界的轨迹； `sandray` 会返回一个数值介于 0 与 1 之间的亮度 (intensity)，之后会缩放成一个 0 至 255 的整数来显示。

要决定一个光线的亮度， `sendray` 需要找到光是从哪个物体所反射的。要办到这件事，我们调用 `first-hit` ，此函数研究在 `*world*` 里的所有平面，并返回光线最先抵达的平面（如果有的话）。如果光没有击中任何东西， `sendray` 仅返回背景颜色，按惯例是 `0` (黑色)。如果光线有击中某物的话，我们需要找出在光击中时，有多少数量的光照在该平面。

朗伯定律 [http://zh.wikipedia.org/zh-tw/%E6%AF%94%E5%B0%94%EF%BC%8D%E6%9C%97%E4%BC%AF%E5%AE%9A%E5 告诉我们，由平面上一点所反射的光的强度，正比于该点的单位法向量 (unit normal vector) *N* (这里是与平面垂直且长度为一的向量)与该点至光源的单位向量 *L* 的点积 (dot-product):

\[i = N·L\]

如果光刚好照到这点， N 与 L 会重合 (coincident)，则点积会是最大值， `1` 。如果将在这时候将平面朝光转 90 度，则 N 与 L 会垂直，则两者点积会是 `0` 。如果光在平面后面，则点积会是负数。

在我们的程序里，我们假设光源在观测点 (eye)，所以 `lambert` 使用了这个规则来找到平面上某点的亮度 (illumination)，返回我们追踪的光的单位向量与法向量的点积。

在 `sendray` 这个值会乘上平面的颜色 (即便是有好的照明，一个暗的平面还是暗的)来决定该点之后总体亮度。

为了简单起见，我们在模拟世界里会只有一种物体，球体。图 9.5 包含了与球体有关的代码。球体结构包含了 `surface` ，所以一个球体会有一种颜色以及 `center` 和 `radius` 。调用 `defsphere` 添加一个新球体至世界里。

```
(defstruct (sphere (:include surface))
  radius center)

(defun defsphere (x y z r c)
  (let ((s (make-sphere
             :radius r
             :center (make-point :x x :y y :z z)
             :color  c)))
    (push s *world*)
    s))

(defun intersect (s pt xr yr zr)
  (funcall (typecase s (sphere #'sphere-intersect))
           s pt xr yr zr))

(defun sphere-intersect (s pt xr yr zr)
  (let* ((c (sphere-center s))
         (n (minroot (+ (sq xr) (sq yr) (sq zr))
                     (* 2 (+ (* (- (x pt) (x c)) xr)
                             (* (- (y pt) (y c)) yr)
                             (* (- (z pt) (z c)) zr)))
                     (+ (sq (- (x pt) (x c)))
                        (sq (- (y pt) (y c)))
                        (sq (- (z pt) (z c)))
                        (- (sq (sphere-radius s)))))))
    (if n
        (make-point :x  (+ (x pt) (* n xr))
                    :y  (+ (y pt) (* n yr))
                    :z  (+ (z pt) (* n zr))))))

(defun normal (s pt)
  (funcall (typecase s (sphere #'sphere-normal))
           s pt))

(defun sphere-normal (s pt)
  (let ((c (sphere-center s)))
```

```
    (unit-vector (- (x c) (x pt))
                 (- (y c) (y pt))
                 (- (z c) (z pt)))))
```

图 **9.5** 球体。

函数 `intersect` 判断与何种平面有关，并调用对应的函数。在此时只有一种， `sphere-intersect` ，但 `intersect` 是写成可以容易扩展处理别种物体。

我们要怎么找到一束光与一个球体的交点 (intersection)呢？光线是表示成点 \(p = 〈x_0,y_0,x_0〉\) 以及单位向量 \(v = 〈x_r,y_r,x_r〉\) 。每个在光上的点可以表示为 \(p+nv\)，对于某个 *n* —— 即 \(〈x_0+nx_r,y_0+ny_r,z_0+nz_r〉\) 。光击中球体的点的距离至中心 \(〈x_c,y_c,z_c〉\) 会等于球体的半径 *r* 。所以在下列这个交点的方程序会成立:

\[r = \sqrt{ (x_0 + nx_r - x_c)^2 + (y_0 + ny_r - y_c)^2 + (z_0 + nz_r - z_c)^2 }\]

这会给出

\[an^2 + bn + c = 0\]

其中

\[\begin{split}a = x_r^2 + y_r^2 + z_r^2\\b = 2((x_0-x_c)x_r + (y_0-y_c)y_r + (z_0-z_c)z_r)\\c = (x_0-x_c)^2 + (y_0-y_c)^2 + (z_0-z_c)^2 - r^2\end{split}\]

要找到交点我们只需要找到这个二次方程序的根。它可能是零、一个或两个实数根。没有根代表光没有击中球体；一个根代表光与球体交于一点 (擦过 「grazing hit」)；两个根代表光与球体交于两点 (一点交于进入时、一点交于离开时)。在最后一个情况里，我们想要两个根之中较小的那个； *n* 与光离开观测点的距离成正比，所以先击中的会是较小的 *n* 。所以我们调用 `minroot` 。如果有一个根， `sphere-intersect` 返回代表该点的 \(〈x_0+nx_r,y_0+ny_r,z_0+nz_r〉\) 。

图 9.5 的另外两个函数， `normal` 与 `sphere-normal` 类比于 `intersect` 与 `sphere-intersect` 。要找到垂直于球体很简单 —— 不过是从该点至球体中心的向量而已。

图 9.6 示范了我们如何产生图片； `ray-test` 定义了 38 个球体（不全都看的见）然后产生一张图片，叫做 “sphere.pgm” 。

(译注：PGM 可移植灰度图格式，更多信息参见 wiki [http://en.wikipedia.org/wiki/Portable_graymap])

```
(defun ray-test (&optional (res 1))
  (setf *world* nil)
  (defsphere 0 -300 -1200 200 .8)
```

```
(defsphere -80 -150 -1200 200 .7)
(defsphere 70 -100 -1200 200 .9)
(do ((x -2 (1+ x)))
    ((> x 2))
  (do ((z 2 (1+ z)))
      ((> z 7))
    (defsphere (* x 200) 300 (* z -400) 40 .75)))
(tracer (make-pathname :name "spheres.pgm") res))
```

图 **9.6** 使用光线追踪器

图 9.7 是产生出来的图片，其中 `res` 参数为 10。

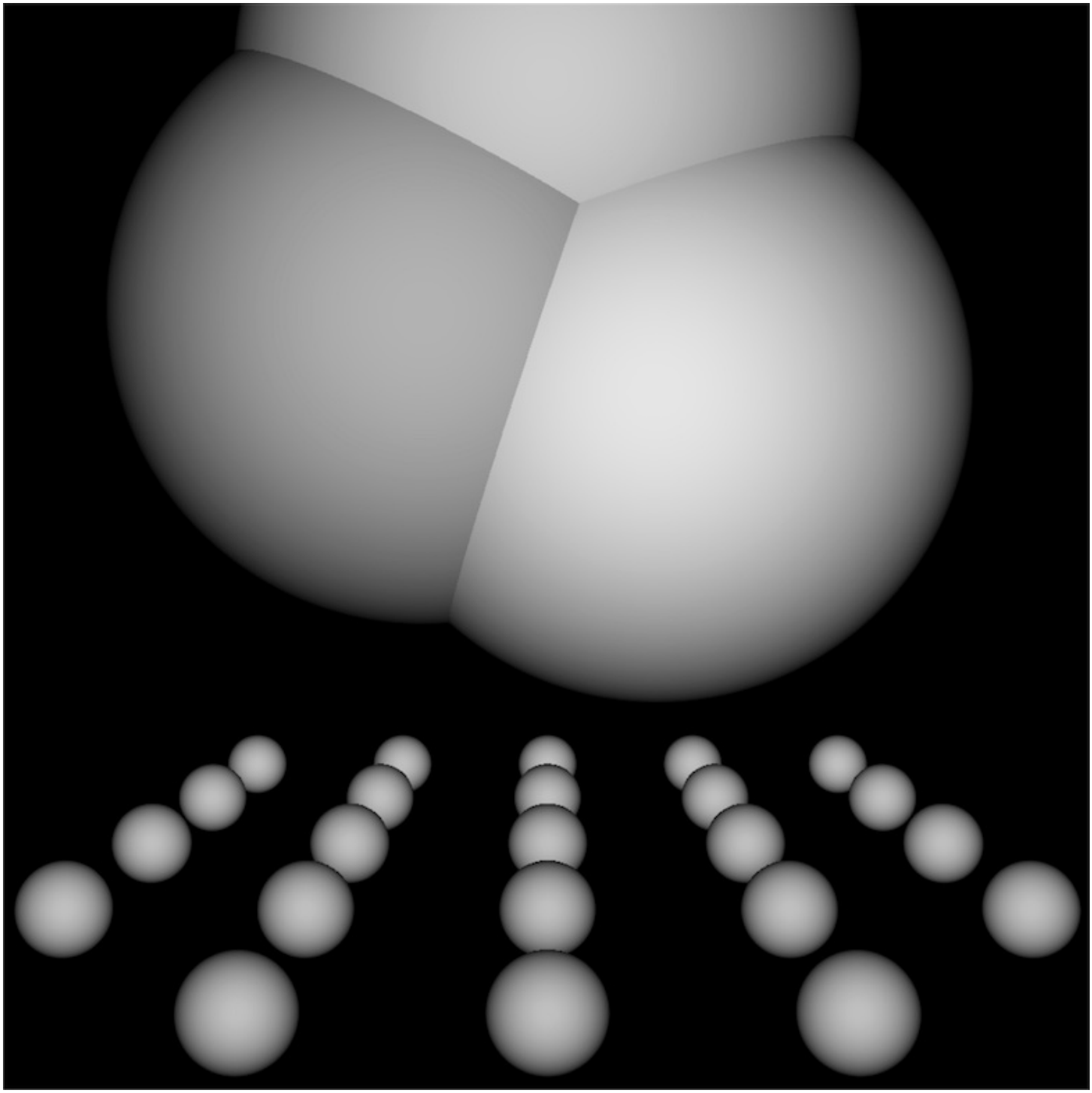

图 **9.7:** 追踪光线的图

一个实际的光线追踪器可以产生更复杂的图片，因为它会考虑更多，我们只考虑了单一光源至平面某一点。可能会有多个光源，每一个有不同的强度。它们通常不会在观测点，在这个情况程序需要检查至光源的向量是否与其他平面相交，这会在第一个相交的平面上产生阴影。将光源放置于观测点让我们不需要考虑这麼复杂的情况，因为我们看不见在阴影中的任何点。

一个实际的光线追踪器不仅追踪光第一个击中的平面，也会加入其它平面的反射光。一个实际的光线追踪器会是有颜色的，并可以模型化出透明或是闪耀的平面。但基本的算法会与图 9.3 所演示的差不多，而许多改进只需要递回的使用同样的成分。

一个实际的光线追踪器可以是高度优化的。这里给出的程序为了精简写成，甚至没有如 Lisp 程序员会最佳化的那样，就仅是一个光线追踪器而已。仅加入类型与行内宣告 (13.3 节)就可以让它变得两倍以上快。

Chapter 9 总结 (Summary)

1. Common Lisp 提供整数 (integers)、比值 (ratios)、浮点数 (floating-point numbers)以及复数 (complex numbers)。
2. 数字可以被约分或转换 (converted)，而它们的位数 (components)可以被取出。
3. 用来比较数字的谓词可以接受任意数量的参数，以及比较下一数对 (successive pairs) ── /= 函数除外，它是用来比较所有的数对 (pairs)。
4. Common Lisp 几乎提供你在低阶科学计算机可以看到的数值函数。同样的函数普遍可应用在多种类型的数字上。
5. Fixnum 是小至可以塞入一个字 (word)的整数。它们在必要时会悄悄但花费昂贵地转成大数 (bignum)。Common Lisp 提供最多四种浮点数。每一个浮点表示法的限制是实现相关的 (implementation-dependent)常量。
6. 一个光线追踪器 (ray-tracer)通过追踪光线来产生图像，使得每一像素回到模拟的世界。

Chapter 9 练习 (Exercises)

1. 定义一个函数，接受一个实数列表，若且唯若 (iff)它们是非递减 (nondecreasing)顺序时返回真。
2. 定义一个函数，接受一个整数 `cents` 并返回四个值，将数字用 `25-` , `10-` , `5-` , `1-` 来显示，使用最少数量的硬币。(译注: `25-` 是 25 美分，以此类推)
3. 一个遥远的星球住着两种生物， wigglies 与 wobblies 。 Wigglies 与 wobblies 唱歌一样厉害。每年都有一个比赛来选出十大最佳歌手。下面是过去十年的结果:

YEAR	1	2	3	4	5	6	7	8	9	10
WIGGLIES	6	5	6	4	5	5	4	5	6	5
WOBBLIES	4	5	4	6	5	5	6	5	4	5

写一个程序来模拟这样的比赛。你的结果实际上有建议委员会每年选出 10 个最佳歌手吗？

4. 定义一个函数，接受 8 个表示二维空间中两个线段端点的实数，若线段没有相交，则返回假，或返回两个值表示相交点的 `x` 座标与 `y` 座标。
5. 假设 `f` 是一个接受一个 (实数) 参数的函数，而 `min` 与 `max` 是有着不同正负号的非零实数，使得 `f` 对于参数 `i` 有一个根 (返回零)并满足 `min < i < max` 。定义一个函数，接受四个参数， `f` , `min` , `max` 以及 `epsilon` ，并返回一个 `i` 的近似值，准确至正负 `epsilon` 之内。
6. *Honer's method* 是一个有效率求出多项式的技巧。要找到 \(ax^3+bx^2+cx+d\) 你对 `x(x(ax+b)+c)+d` 求值。定义一个函数，接受一个或多个参数 —— x 的值伴随着 *n* 个实数，用来表示 `(n-1)` 次方的多项式的系数 —— 并用 *Honer's method* 计算出多项式的值。

译注: Honer's method on wiki [http://en.wikipedia.org/wiki/Horner's_method]

7. 你的 Common Lisp 实现使用了几个位元来表示定长数？
8. 你的 Common Lisp 实现提供几种不同的浮点数？

脚注

[1] 当 `format` 取整显示时，它不保证会取成偶数或奇数。见 125 页 (译注: 7.4 节)。

第十章：宏

Lisp 代码是由 Lisp 对象的列表来表示。2.3 节宣称这让 Lisp 可以写出可自己写程序的程序。本章将示范如何跨越表达式与代码的界线。

10.1 求值 (Eval)

如何产生表达式是很直观的：调用 `list` 即可。我们没有考虑到的是，如何使 Lisp 将列表视为代码。这之间缺少的一环是函数 `eval` ，它接受一个表达式，将其求值，然后返回它的值:

```
> (eval '(+ 1 2 3))
6
> (eval '(format t "Hello"))
Hello
NIL
```

如果这看起很熟悉的话，这是应该的。这就是我们一直交谈的那个 `eval` 。下面这个函数实现了与顶层非常相似的东西:

```
(defun our-toplevel ()
  (do ()
      (nil)
    (format t "~%> ")
    (print (eval (read)))))
```

也是因为这个原因，顶层也称为读取—求值—打印循环 (read-eval-print loop, REPL)。

调用 `eval` 是跨越代码与列表界线的一种方法。但它不是一个好方法:

1. 它的效率低下： `eval` 处理的是原始列表 (raw list)，或者当下编译它，或者用直译器求值。两种方法都比执行编译过的代码来得慢许多。
2. 表达式在没有词法语境 (lexical context)的情况下被求值。举例来说，如果你在一个 `let` 里调用 `eval` ，传给 `eval` 的表达式将无法引用由 `let` 所设置的变量。

有许多更好的方法 (下一节叙述)来利用产生代码的这个可能性。当然 `eval` 也是有用的，唯一合法的用途像是在顶层循环使用它。

对于程序员来说， `eval` 的主要价值大概是作为 Lisp 的概念模型。我们可以想像 Lisp 是由一个长的 `cond` 表达式定义而成:

```
(defun eval (expr env)
  (cond ...
        ((eql (car expr) 'quote) (cdr expr))
        ...
        (t (apply (symbol-function (car expr))
                  (mapcar #'(lambda (x)
                              (eval x env))
                          (cdr expr))))))
```

许多表达式由预设子句 (default clause)来处理，预设子句获得 car 所引用的函数，将 cdr 所有的参数求值，并返回将前者应用至后者的结果。 [1]

但是像 (quote x) 那样的句子就不能用这样的方式来处理，因为 quote 就是为了防止它的参数被求值而存在的。所以我们需要给 quote 写一个特别的子句。这也是为什么本质上将其称为特殊操作符 (special operator): 一个需要被实现为 eval 的一个特殊情况的操作符。

函数 coerce 与 compile 提供了一个类似的桥梁，让你把列表转成代码。你可以 coerce 一个 lambda 表达式，使其成为函数，

```
> (coerce '(lambda (x) x) 'function)
#<Interpreted-Function BF9D96>
```

而如果你将 nil 作为第一个参数传给 compile ，它会编译作为第二个参数传入的 lambda 表达式。

```
> (compile nil '(lambda (x) (+ x 2)))
#<Compiled-Function BF55BE>
NIL
NIL
```

由于 coerce 与 compile 可接受列表作为参数，一个程序可以在动态执行时 (on the fly)构造新函数。但与调用 eval 比起来，这不是一个从根本解决的办法，并且需抱有同样的疑虑来检视这两个函数。

函数 eval , coerce 与 compile 的麻烦不是它们跨越了代码与列表之间的界线，而是它们在执行期做这件事。跨越界线的代价昂贵。大多数情况下，在编译期做这件事是没问题的，当你的程序执行时，几乎不用成本。下一节会示范如何办到这件事。

10.2 宏 (Macros)

写出能写程序的程序的最普遍方法是通过定义宏。宏是通过转换 (transformation)而实现的操作符。你通过说明你一个调用应该要翻译成什么，来定义一个宏。这个翻译称为宏

展开(macro-expansion)，宏展开由编译器自动完成。所以宏所产生的代码，会变成程序的一个部分，就像你自己输入的程序一样。

宏通常通过调用 `defmacro` 来定义。一个 `defmacro` 看起来很像 `defun` 。但是与其定义一个函数调用应该产生的值，它定义了该怎么翻译出一个函数调用。举例来说，一个将其参数设为 `nil` 的宏可以定义成如下:

```
(defmacro nil! (x)
  (list 'setf x nil))
```

这定义了一个新的操作符，称为 `nil!` ，它接受一个参数。一个这样形式 `(nil! a)` 的调用，会在求值或编译前，被翻译成 `(setf a nil)` 。所以如果我们输入 `(nil! x)` 至顶层，

```
> (nil! x)
NIL
> x
NIL
```

完全等同于输入表达式 `(setf x nil)` 。

要测试一个函数，我们调用它，但要测试一个宏，我们看它的展开式 (expansion)。

函数 `macroexpand-1` 接受一个宏调用，并产生它的展开式:

```
> (macroexpand-1 '(nil! x))
(SETF X NIL)
T
```

一个宏调用可以展开成另一个宏调用。当编译器（或顶层）遇到一个宏调用时，它持续展开它，直到不可展开为止。

理解宏的秘密是理解它们是如何被实现的。在台面底下，它们只是转换成表达式的函数。举例来说，如果你传入这个形式 `(nil! a)` 的表达式给这个函数

```
(lambda (expr)
  (apply #'(lambda (x) (list 'setf x nil))
         (cdr expr)))
```

它会返回 `(setf a nil)` 。当你使用 `defmacro` ，你定义一个类似这样的函数。`macroexpand-1` 全部所做的事情是，当它看到一个表达式的 `car` 是宏时，将表达式传给对应的函数。

10.3 反引号 (Backquote)

反引号读取宏 (read-macro)使得从模版 (templates)建构列表变得有可能。反引号广泛使用在宏定义中。一个平常的引用是键盘上的右引号 (apostrophe)，然而一个反引号是一个左引号。(译注: open quote 左引号，closed quote 右引号)。它称作“反引号”是因为它看起来像是反过来的引号 (titled backwards)。

(译注: 反引号是键盘左上方数字 1 左边那个: ` ，而引号是 enter 左边那个 ')

一个反引号单独使用时，等于普通的引号:

```
> `(a b c)
(A B C)
```

和普通引号一样，单一个反引号保护其参数被求值。

反引号的优点是，在一个反引号表达式里，你可以使用 , （逗号）与 ,@ （comma-at）来重启求值。如果你在反引号表达式里，在某个东西前面加逗号，则它会被求值。所以我们可以使用反引号与逗号来建构列表模版:

```
> (setf a 1 b 2)
2
> `(a is ,a and b is ,b)
(A IS 1 AND B IS 2)
```

通过使用反引号取代调用 `list` ，我们可以写出会产生出展开式的宏。举例来说 `nil!` 可以定义为:

```
(defmacro nil! (x)
  `(setf ,x nil))
```

,@ 与逗号相似，但将（本来应该是列表的）参数扒开。将列表的元素插入模版来取代列表。

```
> (setf lst '(a b c))
(A B C)
> `(lst is ,lst)
(LST IS (A B C))
> `(its elements are ,@lst)
(ITS ELEMENTS ARE A B C)
```

,@ 在宏里很有用，举例来说，在用剩余参数表示代码主体的宏。假设我们想要一个 `while` 宏，只要初始测试表达式为真，对其主体求值:

```
> (let ((x 0))
    (while (< x 10)
       (princ x)
       (incf x)))
0123456789
NIL
```

我们可以通过使用一个剩余参数 (rest parameter)，搜集主体的表达式列表，来定义一个这样的宏，接着使用 comma-at 来扒开这个列表放至展开式里:

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

10.4 示例：快速排序法(Example: Quicksort)

图 10.1 包含了重度依赖宏的一个示例函数 —— 一个使用快速排序演算法 λ [http://acl.readthedocs.org/en/latest/zhCN/notes-cn.html#notes-164] 来排序向量的函数。这个函数的工作方式如下:

```
(defun quicksort (vec l r)
  (let ((i l)
        (j r)
        (p (svref vec (round (+ l r) 2))))     ; 1
    (while (<= i j)                            ; 2
      (while (< (svref vec i) p) (incf i))
      (while (> (svref vec j) p) (decf j))
      (when (<= i j)
        (rotatef (svref vec i) (svref vec j))
        (incf i)
        (decf j)))
    (if (>= (- j l) 1) (quicksort vec l j))     ; 3
    (if (>= (- r i) 1) (quicksort vec i r)))
  vec)
```

图 **10.1** 快速排序。

1. 开始你通过选择某个元素作为主键（ *pivot* ）。许多实现选择要被排序的序列中间元素。
2. 接着你分割（partition）向量，持续交换元素，直到所有主键左边的元素小于主键，右边的元素大于主键。
3. 最后，如果左右分割之一有两个或更多元素时，你递归地应用这个算法至向量的那些分割上。

每一次递归时，分割越变越小，直到向量完整排序为止。

在图 10.1 的实现里，接受一个向量以及标记欲排序范围的两个整数。这个范围当下的中间元素被选为主键 (p)。接着从左右两端开始产生分割，并将左边太大或右边太小的元素交换过来。(将两个参数传给 rotatef 函数，交换它们的值。)最后，如果一个分割含有多个元素时，用同样的流程来排序它们。

除了我们前一节定义的 while 宏之外，图 10.1 也用了内置的 when , incf , decf 以及 rotatef 宏。使用这些宏使程序看起来更加简洁与清晰。

10.5 设计宏 (Macro Design)

撰写宏是一种独特的程序设计，它有着独一无二的目标与问题。能够改变编译器所看到的东西，就像是能够重写它一样。所以当你开始撰写宏时，你需要像语言设计者一样思考。

本节快速给出宏所牵涉问题的概要，以及解决它们的技巧。作为一个例子，我们会定义一个称为 ntimes 的宏，它接受一个数字 *n* 并对其主体求值 *n* 次。

```
> (ntimes 10
    (princ "."))
..........
NIL
```

下面是一个不正确的 ntimes 定义，说明了宏设计中的某些议题:

```
(defmacro ntimes (n &rest body)
  `(do ((x 0 (+ x 1)))
       ((>= x ,n))
     ,@body))
```

这个定义第一眼看起来可能没问题。在上面这个情况，它会如预期的工作。但实际上它在两个方面坏掉了。

一个宏设计者需要考虑的问题之一是，不小心引入的变量捕捉 (variable capture)。这发生在当一个在宏展开式里用到的变量，恰巧与展开式即将插入的语境里，有使用同样名字作为变量的情况。不正确的 ntimes 定义创造了一个变量 x 。所以如果这个宏在已经有 x 作为名字的地方被调用时，它可能无法做到我们所预期的:

```
> (let ((x 10))
    (ntimes 5
       (setf x (+ x 1)))
    x)
10
```

如果 ntimes 如我们预期般的执行，这个表达式应该会对 x 递增五次，最后返回 15 。但因为宏展开刚好使用 x 作为迭代变量， setf 表达式递增那个 x ，而不是我们要递增的那个。一旦宏调用被展开，前述的展开式变成:

```
> (let ((x 10))
    (do ((x 0 (+ x 1)))
        ((>= x 5))
      (setf x (+ x 1)))
    x)
```

最普遍的解法是不要使用任何可能会被捕捉的一般符号。取而代之的我们使用 gensym (8.4 小节)。因为 read 函数 intern 每个它见到的符号，所以在一个程序里，没有可能会有任何符号会 eql gensym。如果我们使用 gensym 而不是 x 来重写 ntimes 的定义，至少对于变量捕捉来说，它是安全的:

```
(defmacro ntimes (n &rest body)
  (let ((g (gensym)))
    `(do ((,g 0 (+ ,g 1)))
         ((>= ,g ,n))
       ,@body)))
```

但这个宏在另一问题上仍有疑虑: 多重求值 (multiple evaluation)。因为第一个参数被直接插入 do 表达式，它会在每次迭代时被求值。当第一个参数是有副作用的表达式，这个错误非常清楚地表现出来:

```
> (let ((v 10))
    (ntimes (setf v (- v 1))
      (princ ".")))
.....
NIL
```

由于 v 一开始是 10 ， 而 setf 返回其第二个参数的值，应该印出九个句点。实际上它只印出五个。

如果我们看看宏调用所展开的表达式，就可以知道为什么:

```
> (let ((v 10))
    (do ((#:g1 0 (+ #:g1 1)))
        ((>= #:g1 (setf v (- v 1))))
      (princ ".")))
```

每次迭代我们不是把迭代变量 (gensym 通常印出前面有 #: 的符号)与 9 比较，而是与每次求值时会递减的表达式比较。这如同每次我们查看地平线时，地平线都越来越近。

避免非预期的多重求值的方法是设置一个变量，在任何迭代前将其设为有疑惑的那个表

达式。这通常牵扯到另一个 gensym:

```
(defmacro ntimes (n &rest body)
  (let ((g (gensym))
        (h (gensym)))
    `(let ((,h ,n))
       (do ((,g 0 (+ ,g 1)))
           ((>= ,g ,h))
         ,@body))))
```

终于，这是一个 ntimes 的正确定义。

非预期的变量捕捉与多重求值是折磨宏的主要问题，但不只有这些问题而已。有经验后，要避免这样的错误与避免更熟悉的错误一样简单，比如除以零的错误。

你的 Common Lisp 实现是一个学习更多有关宏的好地方。借由调用展开至内置宏，你可以理解它们是怎么写的。下面是大多数实现对于一个 cond 表达式会产生的展开式:

```
> (pprint (macroexpand-1 '(cond (a b)
                                (c d e)
                                (t f))))
(IF A
    B
    (IF C
        (PROGN D E)
        F))
```

函数 pprint 印出像代码一样缩排的表达式，这在检视宏展开式时特别有用。

10.6 通用化引用 (Generalized Reference)

由于一个宏调用可以直接在它出现的地方展开成代码，任何展开为 setf 表达式的宏调用都可以作为 setf 表达式的第一个参数。 举例来说，如果我们定义一个 car 的同义词，

```
(defmacro cah (lst) `(car ,lst))
```

然后因为一个 car 调用可以是 setf 的第一个参数，而 cah 一样可以:

```
> (let ((x (list 'a 'b 'c)))
    (setf (cah x) 44)
    x)
(44 B C)
```

撰写一个展开成一个 setf 表达式的宏是另一个问题，是一个比原先看起来更为困难的

问题。看起来也许你可以这样实现 incf ，只要

```
(defmacro incf (x &optional (y 1)) ; wrong
  `(setf ,x (+ ,x ,y)))
```

但这是行不通的。这两个表达式不相等:

```
(setf (car (push 1 lst)) (1+ (car (push 1 lst))))

(incf (car (push 1 lst)))
```

如果 lst 是 nil 的话，第二个表达式会设成 (2) ，但第一个表达式会设成 (1 2) 。

Common Lisp 提供了 define-modify-macro 作为写出对于 setf 限制类别的宏的一种方法 它接受三个参数: 宏的名字，额外的参数 (隐含第一个参数 place)，以及产生出 place 新数值的函数名。所以我们可以将 incf 定义为

```
(define-modify-macro our-incf (&optional (y 1)) +)
```

另一版将元素推至列表尾端的 push 可写成:

```
(define-modify-macro append1f (val)
  (lambda (lst val) (append lst (list val))))
```

后者会如下工作:

```
> (let ((lst '(a b c)))
    (append1f lst 'd)
    lst)
(A B C D)
```

顺道一提， push 与 pop 都不能定义为 modify-macros，前者因为 place 不是其第一个参数，而后者因为其返回值不是更改后的对象。

10.7 示例：实用的宏函数 (Example: Macro Utilities)

6.4 节介绍了实用函数 (utility)的概念，一种像是构造 Lisp 的通用操作符。我们可以使用宏来定义不能写作函数的实用函数。我们已经见过几个例子: nil! , ntimes 以及 while ，全部都需要写成宏，因为它们全都需要某种控制参数求值的方法。本节给出更多你可以使用宏写出的多种实用函数。图 10.2 挑选了几个实践中证实值得写的实用函数。

```
(defmacro for (var start stop &body body)
  (let ((gstop (gensym)))
```

```
    `(do ((,var ,start (1+ ,var))
          (,gstop ,stop))
         ((> ,var ,gstop))
       ,@body)))

(defmacro in (obj &rest choices)
  (let ((insym (gensym)))
    `(let ((,insym ,obj))
       (or ,@(mapcar #'(lambda (c) `(eql ,insym ,c))
                     choices)))))

(defmacro random-choice (&rest exprs)
  `(case (random ,(length exprs))
     ,@(let ((key -1))
         (mapcar #'(lambda (expr)
                     `(,(incf key) ,expr))
                 exprs))))

(defmacro avg (&rest args)
  `(/ (+ ,@args) ,(length args)))

(defmacro with-gensyms (syms &body body)
  `(let ,(mapcar #'(lambda (s)
                     `(,s (gensym)))
                 syms)
     ,@body))

(defmacro aif (test then &optional else)
  `(let ((it ,test))
     (if it ,then ,else)))
```

图 **10.2:** 实用宏函数

第一个 `for`，设计上与 `while` 相似 (164 页，译注: 10.3 节)。它是给需要使用一个绑定至一个值的范围的新变量来对主体求值的循环:

```
> (for x 1 8
          (princ x))
12345678
NIL
```

这比写出等效的 `do` 来得省事，

```
(do ((x 1 (+ x 1)))
    ((> x 8))
  (princ x))
```

这非常接近实际的展开式:

```
(do ((x 1 (1+ x))
```

```
      (#:g1 8))
    ((> x #:g1))
  (princ x))
```

宏需要引入一个额外的变量来持有标记范围 (range)结束的值。 上面在例子里的 `8` 也可是个函数调用，这样我们就不需要求值好几次。额外的变量需要是一个 gensym ，为了避免非预期的变量捕捉。

图 10.2 的第二个宏 `in` ，若其第一个参数 `eql` 任何自己其他的参数时，返回真。表达式我们可以写成:

```
(in (car expr) '+ '- '*)
```

我们可以改写成:

```
(let ((op (car expr)))
  (or (eql op '+)
      (eql op '-)
      (eql op '*)))
```

确实，第一个表达式展开后像是第二个，除了变量 `op` 被一个 gensym 取代了。

下一个例子 `random-choice` ，随机选取一个参数求值。在 74 页 (译注: 第 4 章的图 4.6) 我们需要随机在两者之间选择。 `random-choice` 宏实现了通用的解法。一个像是这样的调用:

```
(random-choice (turn-left) (turn-right))
```

会被展开为:

```
(case (random 2)
  (0 (turn-left))
  (1 (turn-right)))
```

下一个宏 `with-gensyms` 主要预期用在宏主体里。它不寻常，特别是在特定应用中的宏，需要 gensym 几个变量。有了这个宏，与其

```
(let ((x (gensym)) (y (gensym)) (z (gensym)))
       ...)
```

我们可以写成

```
(with-gensyms (x y z)
       ...)
```

到目前为止，图 10.2 定义的宏，没有一个可以定义成函数。作为一个规则，写成宏是因为你不能将它写成函数。但这个规则有几个例外。有时候你或许想要定义一个操作符来作为宏，好让它在编译期完成它的工作。宏 avg 返回其参数的平均值，

```
> (avg 2 4 8)
14/3
```

是一个这种例子的宏。我们可以将 avg 写成函数，

```
(defun avg (&rest args)
  (/ (apply #'+ args) (length args)))
```

但它会需要在执行期找出参数的数量。只要我们愿意放弃应用 avg ，为什么不在编译期调用 length 呢？

图 10.2 的最后一个宏是 aif ，它在此作为一个故意变量捕捉的例子。它让我们可以使用变量 it 来引用到一个条件式里的测试参数所返回的值。也就是说，与其写成

```
(let ((val (calculate-something)))
  (if val
      (1+ val)
      0))
```

我们可以写成

```
(aif (calculate-something)
     (1+ it)
     0)
```

小心使用 (*Use judiciously*)，预期的变量捕捉可以是一个无价的技巧。Common Lisp 本身在多处使用它: 举例来说 next-method-p 与 call-next-method 皆依赖于变量捕捉。

像这些宏明确演示了为何要撰写替你写程序的程序。一旦你定义了 for ，你就不需要写整个 do 表达式。值得写一个宏只为了节省打字吗？非常值得。节省打字是程序设计的全部；一个编译器的目的便是替你省下使用机械语言输入程序的时间。而宏允许你将同样的优点带到特定的应用里，就像高阶语言带给程序语言一般。通过审慎的使用宏，你也许可以使你的程序比起原来大幅度地精简，并使程序更显着地容易阅读、撰写及维护。

如果仍对此怀疑，考虑看看如果你没有使用任何内置宏时，程序看起来会是怎么样。所有宏产生的展开式，你会需要用手产生。你也可以将这个问题用在另一方面。当你在撰写一个程序时，扪心自问，我需要撰写宏展开式吗？如果是的话，宏所产生的展开式就是你需要写的东西。

10.8 源自 Lisp (On Lisp)

现在宏已经介绍过了，我们看过更多的 Lisp 是由超乎我们想像的 Lisp 写成。许多不是函数的 Common Lisp 操作符是宏，而他们全部用 Lisp 写成的。只有二十五个 Common Lisp 内置的操作符是特殊操作符。

John Foderaro [http://www.franz.com/about/bios/jkf.lhtml] 将 Lisp 称为“可程序的程序语言。” λ [http://acl.readthedocs.org/en/latest/zhCN/notes-cn.html#notes-173] 通过撰写你自己的函数与宏，你将 Lisp 变成任何你想要的语言。 (我们会在 17 章看到这个可能性的图形化示范)无论你的程序适合何种形式，你确信你可以将 Lisp 塑造成适合它的语言。

宏是这个灵活性的主要成分之一。它们允许你将 Lisp 变得完全认不出来，但仍然用一种有原则且高效的方法来实作。在 Lisp 社区里，宏是个越来越感兴趣的主题。可以使用宏办到惊人之事是很清楚的，但更确信的是宏背后还有更多需要被探索。如果你想的话，可以通过你来发现。Lisp 永远将进化放在程序员手里。这是它为什么存活的原因。

Chapter 10 总结 (Summary)

1. 调用 `eval` 是让 Lisp 将列表视为代码的一种方法，但这是不必要而且效率低落的。
2. 你通过叙说一个调用会展开成什么来定义一个宏。台面底下，宏只是返回表达式的函数。
3. 一个使用反引号定义的主体看起来像它会产生出的展开式 (expansion)。
4. 宏设计者需要注意变量捕捉及多重求值。宏可以通过漂亮印出 (pretty-printing)来测试它们的展开式。
5. 多重求值是大多数展开成 `setf` 表达式的问题。
6. 宏比函数来得灵活，可以用来定义许多实用函数。你甚至可以使用变量捕捉来获得好处。
7. Lisp 存活的原因是它将进化交给程序员的双手。宏是使其可能的部分原因之一。

Chapter 10 练习 (Exercises)

1. 如果 `x` 是 `a` ， `y` 是 `b` 以及 `z` 是 `(c d)` ，写出反引用表达式仅包含产生下列结果之一的变量:

```
(a) ((C D) A Z)

(b) (X B C D)

(c) ((C D A) Z)
```

2. 使用 `cond` 来定义 `if` 。
3. 定义一个宏，接受一个数字 *n* ，伴随着一个或多个表达式，并返回第 *n* 个表达式的值:

```
> (let ((n 2))
    (nth-expr n (/ 1 0) (+ 1 2) (/ 1 0)))
3
```

4. 定义 `ntimes` (167 页，译注: 10.5 节)使其展开成一个 (区域)递归函数，而不是一个 `do` 表达式。
5. 定义一个宏 `n-of` ，接受一个数字 *n* 与一个表达式，返回一个 *n* 个渐进值:

```
> (let ((i 0) (n 4))
    (n-of n (incf i)))
(1 2 3 4)
```

6. 定义一个宏，接受一变量列表以及一个代码主体，并确保变量在代码主体被求值后恢复 (revert)到原本的数值。
7. 下面这个 `push` 的定义哪里错误?

```
(defmacro push (obj lst)
  `(setf ,lst (cons ,obj ,lst)))
```

举出一个不会与实际 `push` 做一样事情的函数调用例子。

8. 定义一个将其参数翻倍的宏:

```
> (let ((x 1))
    (double x)
    x)
2
```

脚注

[1] 要真的复制一个 Lisp 的话， `eval` 会需要接受第二个参数 (这里的 `env`) 来表示词法环境 (lexical enviroment)。这个模型的 `eval` 是不正确的，因为它在对参数求值前就取出函数，然而 Common Lisp 故意没有特别指出这两个操作的顺序。

第十一章：Common Lisp 对象系统

Common Lisp 对象系统，或称 CLOS，是一组用来实现面向对象编程的操作集。由于它们有着同样的历史，通常将这些操作视为一个群组。 λ [http://acl.readthedocs.org/en/latest/zhCN/notes-cn.html#notes-176] 技术上来说，它们与其他部分的 Common Lisp 没什么大不同： `defmethod` 和 `defun` 一样，都是整合在语言中的一个部分。

11.1 面向对象编程 Object-Oriented Programming

面向对象编程意味著程序组织方式的改变。这个改变跟已经发生过的处理器运算处理能力分配的变化雷同。在 1970 年代，一个多用户的计算机系统代表著，一个或两个大型机连接到大量的哑终端 [http://zh.wikipedia.org/wiki/%E5%93%91%E7%BB%88%E7%AB%AF](dumb terminal)。现在更可能的是大量相互通过网络连接的工作站 (workstation)。系统的运算处理能力现在分布至个体用户上，而不是集中在一台大型的计算机上。

面向对象编程所带来的变革与上例非常类似，前者打破了传统程序的组织方式。不再让单一的程序去操作那些数据，而是告诉数据自己该做什么，程序隐含在这些新的数据“对象”的交互过程之中。

举例来说，假设我们要算出一个二维图形的面积。一个办法是写一个单独的函数，让它检查其参数的类型，然后视类型做处理，如图 11.1 所示。

```
(defstruct rectangle
  height width)

(defstruct circle
  radius)

(defun area (x)
  (cond ((rectangle-p x)
         (* (rectangle-height x) (rectangle-width x)))
        ((circle-p x)
         (* pi (expt (circle-radius x) 2)))))

> (let ((r (make-rectangle)))
    (setf (rectangle-height r) 2
          (rectangle-width r) 3)
    (area r))
6
```

图 **11.1:** 使用结构及函数来计算面积

使用 CLOS 我们可以写出一个等效的程序，如图 11.2 所示。在面向对象模型里，我们的程序被拆成数个独一无二的方法，每个方法为某些特定类型的参数而生。图 11.2 中的两个方法，隐性地定义了一个与图 11.1 相似作用的 `area` 函数，当我们调用 `area` 时，Lisp 检查参数的类型，并调用相对应的方法。

```
(defclass rectangle ()
  (height width))

(defclass circle ()
  (radius))

(defmethod area ((x rectangle))
  (* (slot-value x 'height) (slot-value x 'width)))

(defmethod area ((x circle))
  (* pi (expt (slot-value x 'radius) 2)))

> (let ((r (make-instance 'rectangle)))
    (setf (slot-value r 'height) 2
          (slot-value r 'width) 3)
    (area r))
6
```

图 **11.2:** 使用类型与方法来计算面积

通过这种方式，我们将函数拆成独一无二的方法，面向对象暗指继承 (*inheritance*) —— 槽（slot）与方法（method）皆有继承。在图 11.2 中，作为第二个参数传给 `defclass` 的空列表列出了所有基类。假设我们要定义一个新类，上色的圆形 (colored-circle)，则上色的圆形有两个基类，`colored` 与 `circle`：

```
(defclass colored ()
  (color))

(defclass colored-circle (circle colored)
  ())
```

当我们创造 `colored-circle` 类的实例 (instance)时，我们会看到两个继承：

1. `colored-circle` 的实例会有两个槽：从 `circle` 类继承而来的 `radius` 以及从 `colored` 类继承而来的 `color`。
2. 由于没有特别为 `colored-circle` 定义的 `area` 方法存在，若我们对 `colored-circle` 实例调用 `area`，我们会获得替 `circle` 类所定义的 `area` 方法。

从实践层面来看，面向对象编程代表著以方法、类、实例以及继承来组织程序。为什么

你会想这么组织程序？面向对象方法的主张之一说这样使得程序更容易改动。如果我们想要改变 ob 类对象所显示的方式，我们只需要改动 ob 类的 display 方法。如果我们希望创建一个新的类，大致上与 ob 相同，只有某些方面不同，我们可以创建一个 ob 类的子类。在这个子类里，我们仅改动我们想要的属性，其他所有的属性会从 ob 类默认继承得到。要是我们只是想让某个 ob 对象和其他的 ob 对象不一样，我们可以新建一个 ob 对象，直接修改这个对象的属性即可。若是当时的程序写的很讲究，我们甚至不需要看程序中其他的代码一眼，就可以完成种种的改动。 λ
[http://acl.readthedocs.org/en/latest/zhCN/notes-cn.html#notes-178]

11.2 类与实例 (Class and Instances)

在 4.6 节时，我们看过了创建结构的两个步骤：我们调用 defstruct 来设计一个结构的形式，接著通过一个像是 make-point 这样特定的函数来创建结构。创建实例 (instances) 同样需要两个类似的步骤。首先我们使用 defclass 来定义一个类别 (Class):

```
(defclass circle ()
  (radius center))
```

这个定义说明了 circle 类别的实例会有两个槽 (*slot*)，分别名为 radius 与 center （槽类比于结构里的字段「field」）。

要创建这个类的实例，我们调用通用的 make-instance 函数，而不是调用一个特定的函数，传入的第一个参数为类别名称：

```
> (setf c (make-instance 'circle))
#<CIRCLE #XC27496>
```

要给这个实例的槽赋值，我们可以使用 setf 搭配 slot-value：

```
> (setf (slot-value c 'radius) 1)
1
```

与结构的字段类似，未初始化的槽的值是未定义的 (undefined)。

11.3 槽的属性 (Slot Properties)

传给 defclass 的第三个参数必须是一个槽定义的列表。如上例所示，最简单的槽定义是一个表示其名称的符号。在一般情况下，一个槽定义可以是一个列表，第一个是槽的名称，伴随著一个或多个属性 (property)。属性像关键字参数那样指定。

通过替一个槽定义一个访问器 (accessor)，我们隐式地定义了一个可以引用到槽的函

数，使我们不需要再调用 slot-value 函数。如果我们如下更新我们的 circle 类定义，

```
(defclass circle ()
  ((radius :accessor circle-radius)
   (center :accessor circle-center)))
```

那我们能够分别通过 circle-radius 及 circle-center 来引用槽：

```
> (setf c (make-instance 'circle))
#<CIRCLE #XC5C726>

> (setf (circle-radius c) 1)
1

> (circle-radius c)
1
```

通过指定一个 :writer 或是一个 :reader ， 而不是 :accessor ， 我们可以获得访问器的写入或读取行为。

要指定一个槽的缺省值，我们可以给入一个 :initform 参数。若我们想要在 make-instance 调用期间就将槽初始化，我们可以用 :initarg 定义一个参数名。 [1] 加入刚刚所说的两件事，现在我们的类定义变成：

```
(defclass circle ()
  ((radius :accessor circle-radius
           :initarg :radius
           :initform 1)
   (center :accessor circle-center
           :initarg :center
           :initform (cons 0 0))))
```

现在当我们创建一个 circle 类的实例时，我们可以使用关键字参数 :initarg 给槽赋值，或是将槽的值设为 :initform 所指定的缺省值。

```
> (setf c (make-instance 'circle :radius 3))
#<CIRCLE #XC2DE0E>
> (circle-radius c)
3
> (circle-center c)
(0 . 0)
```

注意 initarg 的优先级比 initform 要高。

我们可以指定某些槽是共享的 —— 也就是每个产生出来的实例，共享槽的值都会是一样的。我们通过声明槽拥有 :allocation :class 来办到此事。（另一个办法是让一个槽

有 :allocation :instance，但由于这是缺省设置，不需要特别再声明一次。）当我们在一个实例中，改变了共享槽的值，则其它实例共享槽也会获得相同的值。所以我们会想要使用共享槽来保存所有实例都有的相同属性。

举例来说，假设我们想要模拟一群成人小报 (a flock of tabloids)的行为。（译注：可以看看什么是 tabloids [http://tinyurl.com/9n4dckk]。）在我们的模拟中，我们想要能够表示一个事实，也就是当一家小报采用一个头条时，其它小报也会跟进的这个行为。我们可以通过让所有的实例共享一个槽来实现。若 tabloid 类别像下面这样定义，

```
(defclass tabloid ()
  ((top-story :accessor tabloid-story
              :allocation :class)))
```

那么如果我们创立两家小报，无论一家的头条是什么，另一家的头条也会是一样的：

```
> (setf daily-blab (make-instance 'tabloid)
        unsolicited-mail (make-instance 'tabloid))
#<TABLOID #x302000EFE5BD>
> (setf (tabloid-story daily-blab) 'adultery-of-senator)
ADULTERY-OF-SENATOR
> (tabloid-story unsolicited-mail)
ADULTERY-OF-SENATOR
```

译注： ADULTERY-OF-SENATOR 参议员的性丑闻。

若有给入 :documentation 属性的话，用来作为 slot 的文档字符串。通过指定一个 :type，你保证一个槽里只会有这种类型的元素。类型声明会在 13.3 节讲解。

11.4 基类 (Superclasses)

defclass 接受的第二个参数是一个列出其基类的列表。一个类别继承了所有基类槽的联集。所以要是我们将 screen-circle 定义成 circle 与 graphic 的子类，

```
(defclass graphic ()
  ((color :accessor graphic-color :initarg :color)
   (visible :accessor graphic-visible :initarg :visible
            :initform t)))

(defclass screen-circle (circle graphic) ())
```

则 screen-circle 的实例会有四个槽，分别从两个基类继承而来。一个类别不需要自己创建任何新槽； screen-circle 的存在，只是为了提供一个可创建同时从 circle 及 graphic 继承的实例。

访问器及 :initargs 参数可以用在 screen-circle 的实例，就如同它们也可以用在 circle 或 graphic 类别那般：

```
> (graphic-color (make-instance 'screen-circle
                                :color 'red :radius 3))
RED
```

我们可以使每一个 screen-circle 有某种缺省的颜色，通过在 defclass 里替这个槽指定一个 :initform：

```
(defclass screen-circle (circle graphic)
  ((color :initform 'purple)))
```

现在 screen-circle 的实例缺省会是紫色的：

```
> (graphic-color (make-instance 'screen-circle))
PURPLE
```

11.5 优先级 (Precedence)

我们已经看过类别是怎样能有多个基类了。当一个实例的方法同时属于这个实例所属的几个类时，Lisp 需要某种方式来决定要使用哪个方法。优先级的重点在于确保这一切是以一种直观的方式发生的。

每一个类别，都有一个优先级列表：一个将自身及自身的基类从最具体到最不具体所排序的列表。在目前看过的例子中，优先级还不是需要讨论的议题，但在更大的程序里，它会是一个需要考虑的议题。

以下是一个更复杂的类别层级：

```
(defclass sculpture () (height width depth))

(defclass statue (sclpture) (subject))

(defclass metalwork () (metal-type))

(defclass casting (metalwork) ())

(defclass cast-statue (statue casting) ())
```

图 11.3 包含了一个表示 cast-statue 类别及其基类的网络。

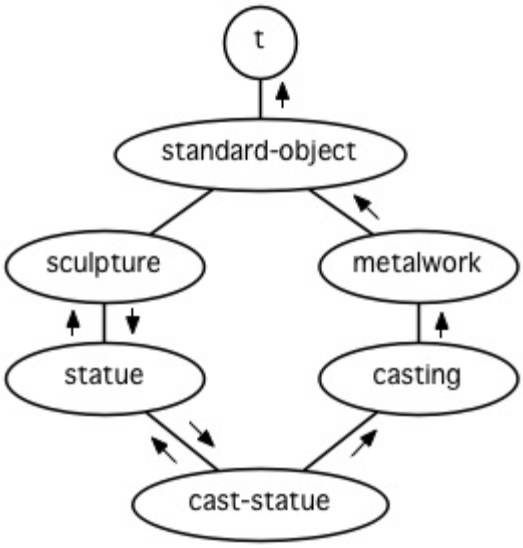

图 **11.3:** 类别层级

要替一个类别建构一个这样的网络，从最底层用一个节点表示该类别开始。接著替类别最近的基类画上节点，其顺序根据 `defclass` 调用里的顺序由左至右画，再来给每个节点重复这个过程，直到你抵达一个类别，这个类别最近的基类是 `standard-object` ——即传给 `defclass` 的第二个参数为 `()` 的类别。最后从这些类别往上建立链接，到表示 `standard-object` 节点为止，接著往上加一个表示类别 `t` 的节点与一个链接。结果会是一个网络，最顶与最下层各为一个点，如图 11.3 所示。

一个类别的优先级列表可以通过如下步骤，遍历对应的网络计算出来：

1. 从网络的底部开始。
2. 往上走，遇到未探索的分支永远选最左边。
3. 如果你将进入一个节点，你发现此节点右边也有一条路同样进入该节点时，则从该节点退后，重走刚刚的老路，直到回到一个节点，这个节点上有尚未探索的路径。接著返回步骤 2。
4. 当你抵达表示 `t` 的节点时，遍历就结束了。你第一次进入每个节点的顺序就决定了节点在优先级列表的顺序。

这个定义的结果之一（实际上讲的是规则 3）在优先级列表里，类别不会在其子类别出现前出现。

图 11.3 的箭头演示了一个网络是如何遍历的。由这个图所决定出的优先级列表为：`cast-statue` , `statue` , `sculpture` , `casting` , `metalwork` , `standard-object` , `t` 。有时候会用 *specific* 这个词，作为在一个给定的优先级列表中来引用类别的位置的速记法。优先级列表从最高优先级排序至最低优先级。

优先级的主要目的是，当一个通用函数 (generic function)被调用时，决定要用哪个方

法。这个过程在下一节讲述。另一个优先级重要的地方是，当一个槽从多个基类继承时。408 页的备注解释了当这情况发生时的应用规则。 λ
[http://acl.readthedocs.org/en/latest/zhCN/notes-cn.html#notes-183]

11.6 通用函数 (Generic Functions)

一个通用函数 (generic function) 是由一个或多个方法组成的一个函数。方法可用 `defmethod` 来定义，与 `defun` 的定义形式类似：

```
(defmethod combine (x y)
  (list x y))
```

现在 `combine` 有一个方法。若我们在此时调用 `combine` ，我们会获得由传入的两个参数所组成的一个列表：

```
> (combine 'a 'b)
(A B)
```

到现在我们还没有做任何一般函数做不到的事情。一个通用函数不寻常的地方是，我们可以继续替它加入新的方法。

首先，我们定义一些可以让新的方法引用的类别：

```
(defclass stuff () ((name :accessor name :initarg :name)))
(defclass ice-cream (stuff) ())
(defclass topping (stuff) ())
```

这里定义了三个类别： `stuff` ，只是一个有名字的东西，而 `ice-cream` 与 `topping` 是 `stuff` 的子类。

现在下面是替 `combine` 定义的第二个方法：

```
(defmethod combine ((ic ice-cream) (top topping))
  (format nil "~A ice-cream with ~A topping."
          (name ic)
          (name top)))
```

在这次 `defmethod` 的调用中，参数被特化了 (*specialized*)：每个出现在列表里的参数都有一个类别的名字。一个方法的特化指出它是应用至何种类别的参数。我们刚定义的方法仅能在传给 `combine` 的参数分别是 `ice-cream` 与 `topping` 的实例时。

而当一个通用函数被调用时， Lisp 是怎么决定要用哪个方法的？Lisp 会使用参数的类别与参数的特化匹配且优先级最高的方法。这表示若我们用 `ice-cream` 实例与 `topping`

实例去调用 combine 方法，我们会得到我们刚刚定义的方法：

```
> (combine (make-instance 'ice-cream :name 'fig)
           (make-instance 'topping :name 'treacle))
"FIG ice-cream with TREACLE topping"
```

但使用其他参数时，我们会得到我们第一次定义的方法：

```
> (combine 23 'skiddoo)
(23 SKIDDOO)
```

因为第一个方法的两个参数皆没有特化，它永远只有最低优先权，并永远是最后一个调用的方法。一个未特化的方法是一个安全手段，就像 case 表达式中的 otherwise 子句。

一个方法中，任何参数的组合都可以特化。在这个方法里，只有第一个参数被特化了：

```
(defmethod combine ((ic ice-cream) x)
  (format nil "~A ice-cream with ~A."
          (name ic)
          x))
```

若我们用一个 ice-cream 的实例以及一个 topping 的实例来调用 combine ，我们仍然得到特化两个参数的方法，因为它是最具体的那个：

```
> (combine (make-instance 'ice-cream :name 'grape)
           (make-instance 'topping :name 'marshmallow))
"GRAPE ice-cream with MARSHMALLOW topping"
```

然而若第一个参数是 ice-cream 而第二个参数不是 topping 的实例的话，我们会得到刚刚上面所定义的那个方法：

```
> (combine (make-instance 'ice-cream :name 'clam)
           'reluctance)
"CLAM ice-cream with RELUCTANCE"
```

当一个通用函数被调用时，参数决定了一个或多个可用的方法 (*applicable* methods)。如果在调用中的参数在参数的特化约定内，我们说一个方法是可用的。

如果没有可用的方法，我们会得到一个错误。如果只有一个，它会被调用。如果多于一个，最具体的会被调用。最具体可用的方法是由调用传入参数所属类别的优先级所决定的。由左往右审视参数。如果有一个可用方法的第一个参数，此参数特化给某个类，其类的优先级高于其它可用方法的第一个参数，则此方法就是最具体的可用方法。平手时比较第二个参数，以此类推。 [2]

在前面的例子里，很容易看出哪个是最具体的可用方法，因为所有的对象都是单继承的。一个 ice-cream 的实例是，按顺序来， ice-cream ， stuff ， standard-object ，以及 t 类别的成员。

方法不需要在由 defclass 定义的类别层级来做特化。他们也可以替类型做特化（更精准的说，可以反映出类型的类别）。以下是一个给 combine 用的方法，对数字做了特化：

```
(defmethod combine ((x number) (y number))
  (+ x y))
```

方法甚至可以对单一的对象做特化，用 eql 来决定：

```
(defmethod combine ((x (eql 'powder)) (y (eql 'spark)))
  'boom)
```

单一对象特化的优先级比类别特化来得高。

方法可以像一般 Common Lisp 函数一样有复杂的参数列表，但所有组成通用函数方法的参数列表必须是一致的 (*congruent*)。参数的数量必须一致，同样数量的选择性参数（如果有的话），要嘛一起使用 &rest 或是 &key 参数，或者一起不要用。下面的参数列表对是全部一致的，

```
(x)              (a)
(x &optional y)  (a &optional b)
(x y &rest z)    (a b &key c)
(x y &key z)     (a b &key c d)
```

而下列的参数列表对不是一致的：

```
(x)              (a b)
(x &optional y)  (a &optional b c)
(x &optional y)  (a &rest b)
(x &key x y)     (a)
```

只有必要参数可以被特化。所以每个方法都可以通过名字及必要参数的特化独一无二地识别出来。如果我们定义另一个方法，有着同样的修饰符及特化，它会覆写掉原先的。所以通过说明

```
(defmethod combine ((x (eql 'powder)) (y (eql 'spark)))
  'kaboom)
```

我们重定义了当 combine 方法的参数是 powder 与 spark 时， combine 方法干了什么事

儿。

11.7 辅助方法 (Auxiliary Methods)

方法可以通过如 `:before`， `:after` 以及 `:around` 等辅助方法来增强。 `:before` 方法允许我们说：“嘿首先，先做这个。” 最具体的 `:before` 方法优先被调用，作为其它方法调用的序幕 (prelude)。 `:after` 方法允许我们说 “P.S. 也做这个。” 最具体的 `:after` 方法最后被调用，作为其它方法调用的闭幕 (epilogue)。在这之间，我们运行的是在这之前仅视为方法的方法，而准确地说应该叫做主方法 (*primary method*)。这个主方法调用所返回的值为方法的返回值，甚至 `:after` 方法在之后被调用也不例外。

`:before` 与 `:after` 方法允许我们将新的行为包在调用主方法的周围。 `:around` 方法提供了一个更戏剧的方式来办到这件事。如果 `:around` 方法存在的话，会调用的是 `:around` 方法而不是主方法。则根据它自己的判断， `:around` 方法自己可能会调用主方法（通过函数 `call-next-method`，这也是这个函数存在的目的）。

这称为标准方法组合机制 (*standard method combination*)。在标准方法组合机制里，调用一个通用函数会调用

1. 最具体的 `:around` 方法，如果有的话。
2. 否则，依序，
    a. 所有的 `:before` 方法，从最具体到最不具体。
    b. 最具体的主方法
    c. 所有的 `:after` 方法，从最不具体到最具体

返回值为 `:around` 方法的返回值（情况 1）或是最具体的主方法的返回值（情况 2）。

辅助方法通过在 `defmethod` 调用中，在方法名后加上一个修饰关键字 (qualifying keyword)来定义。如果我们替 `speaker` 类别定义一个主要的 `speak` 方法如下：

```
(defclass speaker () ())

(defmethod speak ((s speaker) string)
        (format t "~A" string))
```

则使用 `speaker` 实例来调用 `speak` 仅印出第二个参数：

```
> (speak (make-instance 'speaker)
         "I'm hungry")
I'm hungry
NIL
```

通过定义一个 intellectual 子类，将主要的 speak 方法用 :before 与 :after 方法包起来，

```
(defclass intellectual (speaker) ())

(defmethod speak :before ((i intellectual) string)
  (princ "Perhaps "))

(defmethod speak :after ((i intellectual) string)
  (princ " in some sense"))
```

我们可以创建一个说话前后带有惯用语的演讲者：

```
> (speak (make-instance 'intellectual)
         "I am hungry")
Perhaps I am hungry in some sense
NIL
```

如同先前标准方法组合机制所述，所有的 :before 及 :after 方法都被调用了。所以如果我们替 speaker 基类定义 :before 或 :after 方法，

```
(defmethod speak :before ((s speaker) string)
  (princ "I think "))
```

无论是哪个 :before 或 :after 方法被调用，整个通用函数所返回的值，是最具体主方法的返回值 —— 在这个情况下，为 format 函数所返回的 nil 。

而在有 :around 方法时，情况就不一样了。如果有一个替传入通用函数特别定义的 :around 方法，则优先调用 :around 方法，而其它的方法要看 :around 方法让不让它们被运行。一个 :around 或主方法，可以通过调用 call-next-method 来调用下一个方法。在调用下一个方法前，它使用 next-method-p 来检查是否有下个方法可调用。

有了 :around 方法，我们可以定义另一个，更谨慎的， speaker 的子类别：

```
(defclass courtier (speaker) ())

(defmethod speak :around ((c courtier) string)
  (format t "Does the King believe that ~A?" string)
  (if (eql (read) 'yes)
      (if (next-method-p) (call-next-method))
      (format t "Indeed, it is a preposterous idea. ~%"))
  'bow)
```

当传给 speak 的第一个参数是 courtier 类的实例时，朝臣 (courtier)的舌头有了

:around 方法保护，就不会被割掉了：

```
> (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) "kings will last")
Does the King believe that kings will last? no
Indeed, it is a preposterous idea.
BOW
```

记得由 :around 方法所返回的值即通用函数的返回值，这与 :before 与 :after 方法的返回值不一样。

11.8 方法组合机制 (Method Combination)

在标准方法组合中，只有最具体的主方法会被调用（虽然它可以通过 call-next-method 来调用其它方法）。但我们可能会想要把所有可用的主方法的结果汇总起来。

用其它组合手段来定义方法也是有可能的 —— 举例来说，一个返回所有可用主方法的和的通用函数。操作符 (*Operator*)方法组合可以这么理解，想像它是 Lisp 表达式的求值后的结果，其中 Lisp 表达式的第一个元素是某个操作符，而参数是按照具体性调用可用主方法的结果。如果我们定义 price 使用 + 来组合数值的通用函数，并且没有可用的 :around 方法，它会如它所定义的方式动作：

```
(defun price (&rest args)
  (+ (apply 〈most specific primary method〉 args)
     .
     .
     .
     (apply 〈least specific primary method〉 args)))
```

如果有可用的 :around 方法的话，它们根据优先级决定，就像是标准方法组合那样。在操作符方法组合里，一个 around 方法仍可以通过 call-next-method 调用下个方法。然而主方法就不可以使用 call-next-method 了。

我们可以指定一个通用函数的方法组合所要使用的类型，借由在 defgeneric 调用里加入一个 method-combination 子句：

```
(defgeneric price (x)
  (:method-combination +))
```

现在 price 方法会使用 + 方法组合；任何替 price 定义的 defmethod 必须有 + 来作为第二个参数。如果我们使用 price 来定义某些类型，

```
(defclass jacket () ())
(defclass trousers () ())
(defclass suit (jacket trousers) ())

(defmethod price + ((jk jacket)) 350)
(defmethod price + ((tr trousers)) 200)
```

则可获得一件正装的价钱，也就是所有可用方法的总和：

```
> (price (make-instance 'suit))
550
```

下列符号可以用来作为 `defmethod` 的第二个参数或是作为 `defgeneric` 调用中，`method-combination` 的选项：

```
+    and    append    list    max    min    nconc    or    progn
```

你也可以使用 `standard`，yields 标准方法组合。

一旦你指定了通用函数要用何种方法组合，所有替该函数定义的方法必须用同样的机制。而现在如果我们试著使用另个操作符（ `:before` 或 `after` ）作为 `defmethod` 给 `price` 的第二个参数，则会抛出一个错误。如果我们想要改变 `price` 的方法组合机制，我们需要通过调用 `fmakunbound` 来移除整个通用函数。

11.9 封装 (Encapsulation)

面向对象的语言通常会提供某些手段，来区别对象的表示法以及它们给外在世界存取的介面。隐藏实现细节带来两个优点：你可以改变实现方式，而不影响对象对外的样子，而你可以保护对象在可能的危险方面被改动。隐藏细节有时候被称为封装 (*encapsulated*)。

虽然封装通常与面向对象编程相关联，但这两个概念其实是没相干的。你可以只拥有其一，而不需要另一个。我们已经在 108 页 (译注： 6.5 小节。)看过一个小规模的封装例子。函数 `stamp` 及 `reset` 通过共享一个计数器工作，但调用时我们不需要知道这个计数器，也保护我们不可直接修改它。

在 Common Lisp 里，包是标准的手段来区分公开及私有的信息。要限制某个东西的存取，我们将它放在另一个包里，并且针对外部介面，仅输出需要用的名字。

我们可以通过输出可被改动的名字，来封装一个槽，但不是槽的名字。举例来说，我们可以定义一个 `counter` 类别，以及相关的 `increment` 及 `clear` 方法如下：

```
(defpackage "CTR"
  (:use "COMMON-LISP")
  (:export "COUNTER" "INCREMENT" "CLEAR"))

(in-package ctr)

(defclass counter () ((state :initform 0)))

(defmethod increment ((c counter))
  (incf (slot-value c 'state)))

(defmethod clear ((c counter))
  (setf (slot-value c 'state) 0))
```

在这个定义下，在包外部的代码只能够创造 `counter` 的实例，并调用 `increment` 及 `clear` 方法，但不能够存取 `state` 。

如果你想要更进一步区别类的内部及外部介面，并使其不可能存取一个槽所存的值，你也可以这么做。只要在你将所有需要引用它的代码定义完，将槽的名字 unintern：

```
(unintern 'state)
```

则没有任何合法的、其它的办法，从任何包来引用到这个槽。 λ [http://acl.readthedocs.org/en/latest/zhCN/notes-cn.html#notes-191]

11.10 两种模型 (Two Models)

面向对象编程是一个令人疑惑的话题，部分的原因是因为有两种实现方式：消息传递模型 (message-passing model)与通用函数模型 (generic function model)。一开始先有的消息传递。通用函数是广义的消息传递。

在消息传递模型里，方法属于对象，且方法的继承与槽的继承概念一样。要找到一个物体的面积，我们传给它一个 `area` 消息：

```
tell obj area
```

而这调用了任何对象 `obj` 所拥有或继承来的 area 方法。

有时候我们需要传入额外的参数。举例来说，一个 `move` 方法接受一个说明要移动多远的参数。如我我们想要告诉 `obj` 移动 10 个单位，我们可以传下面的消息：

```
(move obj 10)
```

消息传递模型的局限性变得清晰。在消息传递模型里，我们仅特化 (specialize) 第一个

参数。 牵扯到多对象时，没有规则告诉方法该如何处理 —— 而对象回应消息的这个模型使得这更加难处理了。

在消息传递模型里，方法是对象所有的，而在通用函数模型里，方法是特别为对象打造的 (specialized)。 如果我们仅特化第一个参数，那么通用函数模型和消息传递模型就是一样的。但在通用函数模型里，我们可以更进一步，要特化几个参数就几个。这也表示了，功能上来说，消息传递模型是通用函数模型的子集。如果你有通用函数模型，你可以仅特化第一个参数来模拟出消息传递模型。

Chapter 11 总结 (Summary)

1. 在面向对象编程中，函数 `f` 通过定义拥有 `f` 方法的对象来隐式地定义。对象从它们的父母继承方法。
2. 定义一个类别就像是定义一个结构，但更加啰嗦。一个共享的槽属于一整个类别。
3. 一个类别从基类中继承槽。
4. 一个类别的祖先被排序成一个优先级列表。理解优先级算法最好的方式就是通过视觉。
5. 一个通用函数由一个给定名称的所有方法所组成。一个方法通过名称及特化参数来识别。参数的优先级决定了当调用一个通用函数时会使用哪个方法。
6. 方法可以通过辅助方法来增强。标准方法组合机制意味著如果有 `:around` 方法的话就调用它；否则依序调用 `:before` ，最具体的主方法以及 `:after` 方法。
7. 在操作符方法组合机制中，所有的主方法都被视为某个操作符的参数。
8. 封装可以通过包来实现。

10. 面向对象编程有两个模型。通用函数模型是广义的消息传递模型。

Chapter 11 练习 (Exercises)

1. 替图 11.2 所定义的类定义访问器、 initforms 以及 initargs 。重写相关的代码使其再也不用调用 `slot-value` 。
2. 重写图 9.5 的代码，使得球体与点为类别，而 `intersect` 及 `normal` 为通用函数。
3. 假设有若干类别定义如下：

```
(defclass a (c d)   ...)  (defclass e ()   ...)
(defclass b (d c)   ...)  (defclass f (h)  ...)
(defclass c ()      ...)  (defclass g (h)  ...)
(defclass d (e f g) ...)  (defclass h ()   ...)
```

a. 画出表示类别 `a` 祖先的网络以及列出 `a` 的实例归属的类别，从最相关至最不相关排列。

b. 替类别 `b` 也做 (a) 小题的要求。

4. 假定你已经有了下列函数：

`precedence` ：接受一个对象并返回其优先级列表，列表由最具体至最不具体的类组成。

`methods`：接受一个通用函数并返回一个列出所有方法的列表。

`specializations` ：接受一个方法并返回一个列出所有特化参数的列表。返回列表中的每个元素是类别或是这种形式的列表 `(eql x)`，或是 `t`（表示该参数没有被特化）。

使用这些函数（不要使用 `compute-applicable-methods` 及 `find-method` ），定义一个函数 `most-spec-app-meth` ，该函数接受一个通用函数及一个列出此函数被调用过的参数，如果有最相关可用的方法的话，返回它。

5. 不要改变通用函数 `area` 的行为（图 11.2），
6. 举一个只有通用函数的第一个参数被特化会很难解决的问题的例子。

脚注

[1] Initarg 的名称通常是关键字，但不需要是。

[2] 我们不可能比较完所有的参数而仍有平手情形存在，因为这样我们会有两个有着同样特化的方法。这是不可能的，因为第二个的定义会覆写掉第一个。

第十二章：结构

3.3 节中介绍了 Lisp 如何使用指针允许我们将任何值放到任何地方。这种说法是完全有可能的，但这并不一定都是好事。

例如，一个对象可以是它自己的一个元素。这是好事还是坏事，取决于程序员是不是有意这样设计的。

12.1 共享结构 (Shared Structure)

多个列表可以共享 cons 。在最简单的情况下，一个列表可以是另一个列表的一部分。

```
> (setf part (list 'b 'c))
(B C)
> (setf whole (cons 'a part))
(A B C)
```

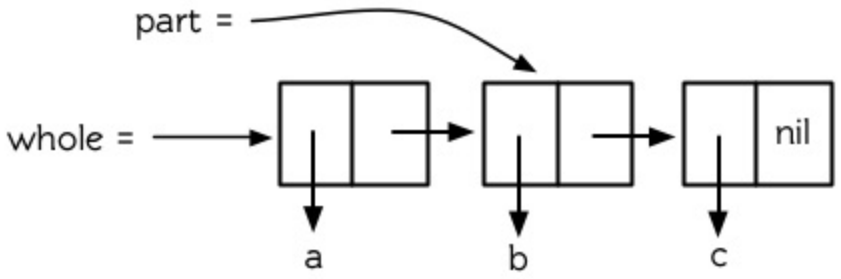

图 **12.1** 共享结构

执行上述操作后，第一个 cons 是第二个 cons 的一部分 (事实上，是第二个 cons 的 cdr)。在这样的情况下，我们说，这两个列表是共享结构 (Share Structure)。这两个列表的基本结构如图 12.1 所示。

其中，第一个 cons 是第二个 cons 的一部分 (事实上，是第二个 cons 的 cdr)。在这样的情况下，我们称这两个列表为共享结构 (Share Structure)。这两个列表的基本结构如图 12.1 所示。

使用 tailp 判断式来检测一下。将两个列表作为它的输入参数，如果第一个列表是第二个列表的一部分时，则返回 T ：

```
> (tailp part whole)
T
```

我们可以把它想像成：

```
(defun our-tailp (x y)
  (or (eql x y)
      (and (consp y)
                          (our-tailp x (cdr y)))))
```

如定义所表明的，每个列表都是它自己的尾端， `nil` 是每一个正规列表的尾端。

在更复杂的情况下，两个列表可以是共享结构，但彼此都不是对方的尾端。在这种情况下，他们都有一个共同的尾端，如图 12.2 所示。我们像这样构建这种情况：

```
(setf part (list 'b 'c)
                        whole1 (cons 1 part)
                        whole2 (cons 2 part))
```

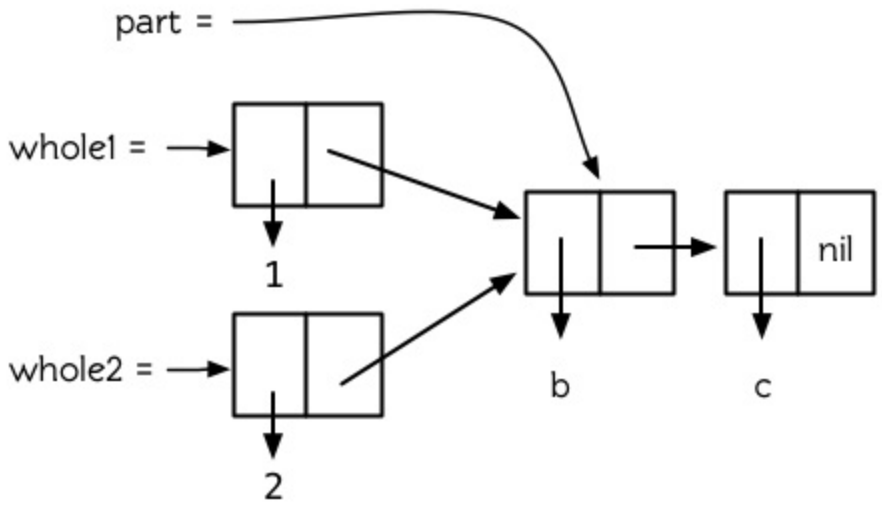

图 **12.2** 被共享的尾端

现在 `whole1` 和 `whole2` 共享结构，但是它们彼此都不是对方的一部分。

当存在嵌套列表时，重要的是要区分是列表共享了结构，还是列表的元素共享了结构。顶层列表结构指的是，直接构成列表的那些 `cons` ，而不包含那些用于构造列表元素的 `cons` 。图 12.3 是一个嵌套列表的顶层列表结构 (译者注：图 12.3 中上面那三个有黑色阴影的 `cons` 即构成顶层列表结构的 `cons`)。

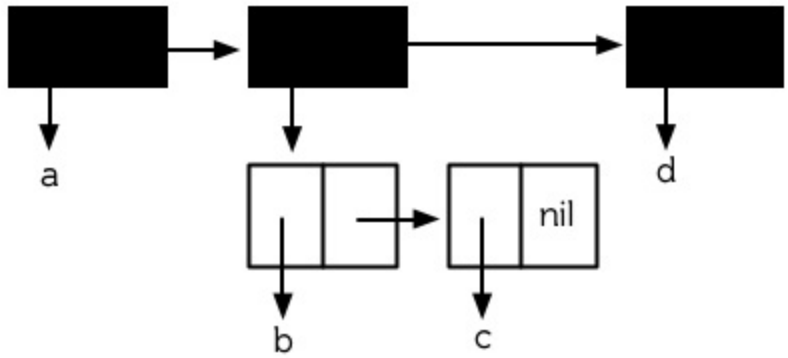

图 **12.3** 顶层列表结构

两个 `cons` 是否共享结构，取决于我们把它们看作是列表还是树[http://zh.wikipedia.org/wiki/%E6%A0%91_(%E6%95%B0%E6%8D%AE%E7%BB%93%E6%可能存在两个嵌套列表，当把它们看作树时，它们共享结构，而看作列表时，它们不共享结构。图 12.4 构建了这种情况，两个列表以一个元素的形式包含了同一个列表，代码如下：

```
(setf element (list 'a 'b)
      holds1 (list 1 element 2)
      holds2 (list element 3))
```

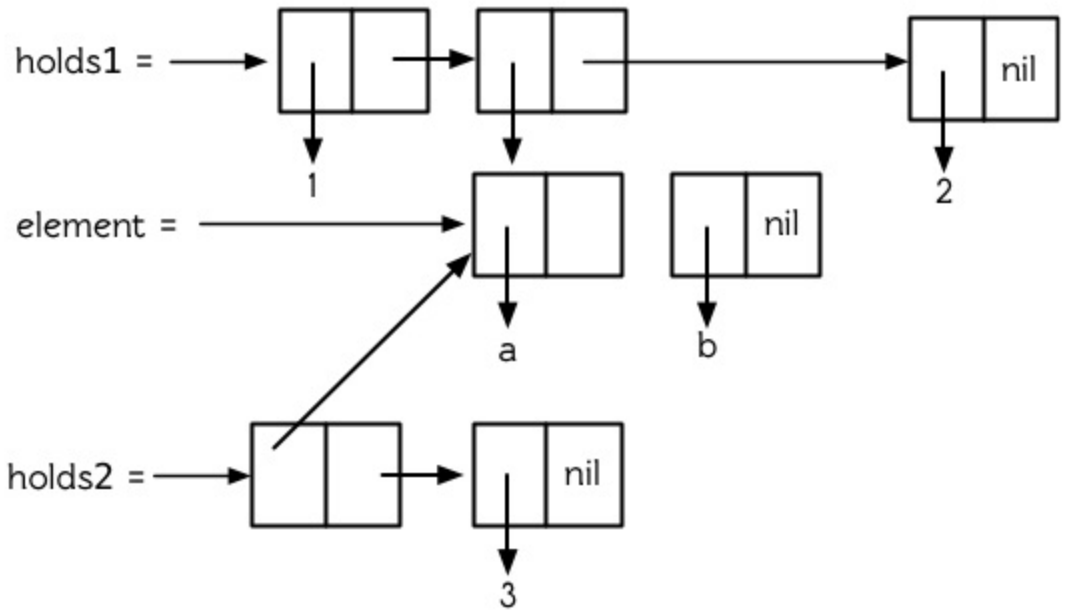

图 **12.4** 共享子树

虽然 `holds1` 的第二个元素和 `holds2` 的第一个元素共享结构 (其实是相同的)，但如果把 `holds1` 和 `holds2` 看成是列表时，它们不共享结构。仅当两个列表共享顶层列表结构时，才能说这两个列表共享结构，而 `holds1` 和 `holds2` 没有共享顶层列表结构。

如果我们想避免共享结构，可以使用复制。函数 copy-list 可以这样定义：

```
(defun our-copy-list (lst)
   (if (null lst)
       nil
       (cons (car lst) (our-copy-list (cdr lst)))))
```

它返回一个不与原始列表共享顶层列表结构的新列表。函数 copy-tree 可以这样定义：

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

它返回一个连原始列表的树型结构也不共享的新列表。图 12.5 显示了对一个嵌套列表使用 copy-list 和 copy-tree 的区别。

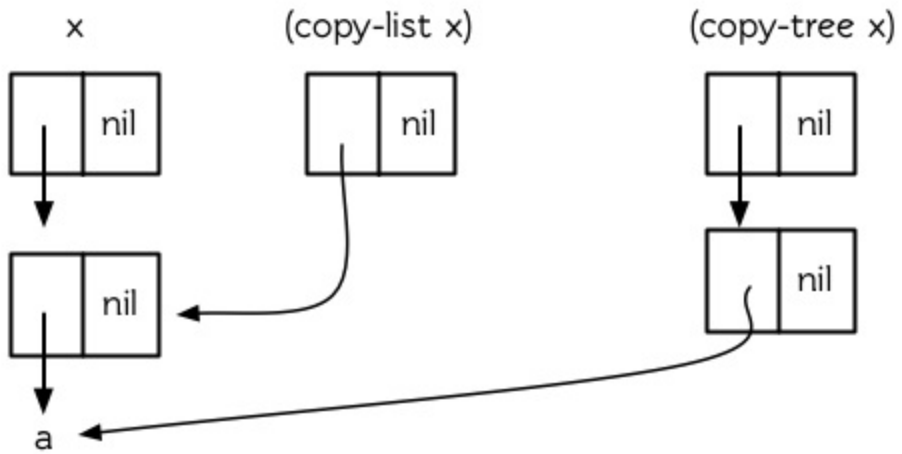

图 **12.5** 两种复制

12.2 修改 (Modification)

为什么要避免共享结构呢？之前讨论的共享结构问题仅仅是个智力练习，到目前为止，并没使我们在实际写程序的时候有什么不同。当修改一个被共享的结构时，问题出现了。如果两个列表共享结构，当我们修改了其中一个，另外一个也会无意中被修改。

上一节中，我们介绍了怎样构建一个是其它列表的尾端的列表：

```
(setf whole (list 'a 'b 'c)
      tail (cdr whole))
```

因为 whole 的 cdr 与 tail 是相等的，无论是修改 tail 还是 whole 的 cdr ，我们修改的

都是同一个 cons：

```
> (setf (second tail ) 'e)
E
> tail
(B E)
> whole
(A B E)
```

同样的，如果两个列表共享同一个尾端，这种情况也会发生。

一次修改两个对象并不总是错误的。有时候这可能正是你想要的。但如果无意的修改了共享结构，将会引入一些非常微妙的 bug。Lisp 程序员要培养对共享结构的意识，并且在这类错误发生时能够立刻反应过来。当一个列表神秘的改变了的时候，很有可能是因为改变了其它与之共享结构的对象。

真正危险的不是共享结构，而是改变被共享的结构。为了安全起见，干脆避免对结构使用 setf (以及相关的运算，比如： pop ， rplaca 等)，这样就不会遇到问题了。如果某些时候不得不修改列表结构时，要搞清楚要修改的列表的来源，确保它不要和其它不需要改变的对象共享结构。如果它和其它不需要改变的对象共享了结构，或者不能预测它的来源，那么复制一个副本来进行改变。

当你调用别人写的函数的时候要加倍小心。除非你知道它内部的操作，否则，你传入的参数时要考虑到以下的情况：

1.它对你传入的参数可能会有破坏性的操作

2.你传入的参数可能被保存起来，如果你调用了一个函数，然后又修改了之前作为参数传入该函数的对象，那么你也就改变了函数已保存起来作为它用的对象[1]。

在这两种情况下，解决的方法是传入一个拷贝。

在 Common Lisp 中，一个函数调用在遍历列表结构 (比如， mapcar 或 remove-if 的参数)的过程中不允许修改被遍历的结构。关于评估这样的代码的重要性并没有明确的规定。

12.3 示例：队列 (Example: Queues)

共享结构并不是一个总让人担心的特性。我们也可以对其加以利用的。这一节展示了怎样用共享结构来表示队列 [http://zh.wikipedia.org/wiki/%E9%98%9F%E5%88%97]。队列对象是我们可以按照数据的插入顺序逐个检出数据的仓库，这个规则叫做先进先出 (FIFO, first in, first out) [http://zh.wikipedia.org/zh-

cn/%E5%85%88%E9%80%B2%E5%85%88%E5%87%BA]。

用列表表示栈 (stack) [http://zh.wikipedia.org/wiki/%E6%A0%88]比较容易，因为栈是从同一端插入和检出。而表示队列要困难些，因为队列的插入和检出是在不同端。为了有效的实现队列，我们需要找到一种办法来指向列表的两个端。

图 12.6 给出了一种可行的策略。它展示怎样表示一个含有 a，b，c 三个元素的队列。一个队列就是一对列表，最后那个 `cons` 在相同的列表中。这个列表对由被称作头端 (front)和尾端 (back)的两部分组成。如果要从队列中检出一个元素，只需在其头端 `pop`，要插入一个元素，则创建一个新的 `cons`，把尾端的 `cdr` 设置成指向这个 `cons`，然后将尾端指向这个新的 `cons`。

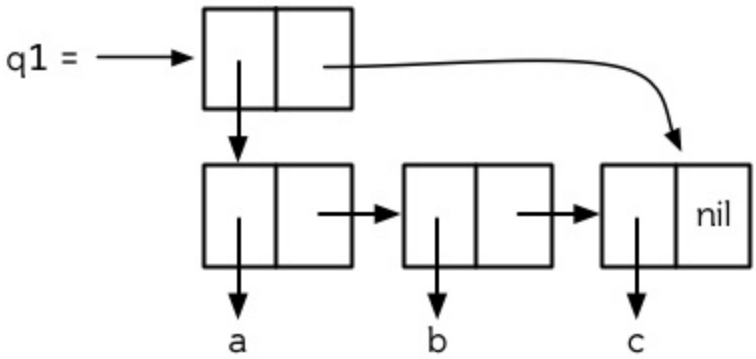

图 **12.6** 一个队列的结构

```
(defun make-queue () (cons nil nil))

(defun enqueue (obj q)
  (if (null (car q))
      (setf (cdr q) (setf (car q) (list obj)))
      (setf (cdr (cdr q)) (list obj)
            (cdr q) (cdr (cdr q)))))
  (car q))

(defun dequeue (q)
  (pop (car q)))
```

图 **12.7** 队列实现

图 12.7 中的代码实现了这一策略。其用法如下：

```
> (setf q1 (make-queue))
(NIL)
> (progn (enqueue 'a q1)
         (enqueue 'b q1)
         (enqueue 'c q1))
(A B C)
```

现在， q1 的结构就如图 12.6 那样：

```
> q1
((A B C) C)
```

从队列中检出一些元素：

```
> (dequeue q1)
A
> (dequeue q1)
B
> (enqueue 'd q1)
(C D)
```

12.4 破坏性函数 (Destructive Functions)

Common Lisp 包含一些允许修改列表结构的函数。为了提高效率，这些函数是具有破坏性的。虽然它们可以回收利用作为参数传给它们的 cons ，但并不是因为想要它们的副作用而调用它们 (译者注：因为这些函数的副作用并没有任何保证，下面的例子将说明问题)。

比如， delete 是 remove 的一个具有破坏性的版本。虽然它可以破坏作为参数传给它的列表，但它并不保证什么。在大多数的 Common Lisp 的实现中，会出现下面的情况：

```
> (setf lst '(a r a b i a) )
(A R A B I A)
> (delete 'a lst )
(R B I)
> lst
(A R B I)
```

正如 remove 一样，如果你想要副作用，应该对返回值使用 setf ：

```
(setf lst (delete 'a lst))
```

破坏性函数是怎样回收利用传给它们的列表的呢？比如，可以考虑 nconc —— append 的破坏性版本。[2]下面是两个参数版本的实现，其清楚地展示了两个已知列表是怎样被缝在一起的：

```
(defun nconc2 ( x y)
    (if (consp x)
        (progn
           (setf (cdr (last x)) y)
            x)
```

```
          y))
```

我们找到第一个列表的最后一个 *Cons* 核 (cons cells)，把它的 `cdr` 设置成指向第二个列表。一个正规的多参数的 `nconc` 可以被定义成像附录 B 中的那样。

函数 `mapcan` 类似 `mapcar` ，但它是用 `nconc` 把函数的返回值 (必须是列表) 拼接在一起的：

```
> (mapcan #'list
          '(a b c)
          '(1 2 3 4))
( A 1 B 2 C 3)
```

这个函数可以定义如下：

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

使用 `mapcan` 时要谨慎，因为它具有破坏性。它用 `nconc` 拼接返回的列表，所以这些列表最好不要再在其它地方使用。

这类函数在处理某些问题的时候特别有用，比如，收集树在某层上的所有子结点。如果 `children` 函数返回一个节点的孩子节点的列表，那么我们可以定义一个函数返回某节点的孙子节点的列表如下：

```
(defun grandchildren (x)
   (mapcan #'(lambda (c)
               (copy-list (children c)))
           (children x)))
```

这个函数调用 `copy-list` 时存在一个假设 —— `chlidren` 函数返回的是一个已经保存在某个地方的列表，而不是构建了一个新的列表。

一个 `mapcan` 的无损变体可以这样定义：

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

如果使用 `mappend` 函数，那么 `grandchildren` 的定义就可以省去 `copy-list` ：

```
(defun grandchildren (x)
   (mappend #'children (children x)))
```

12.5 示例：二叉搜索树 (Example: Binary Search Trees)

在某些情况下，使用破坏性操作比使用非破坏性的显得更自然。第 4.7 节中展示了如何维护一个具有二分搜索格式的有序对象集 (或者说维护一个二叉搜索树 (BST) [http://zh.wikipedia.org/zh-cn/%E4%BA%8C%E5%85%83%E6%90%9C%E5%B0%8B%E6%A8%B9])。第 4.7 节中给出的函数都是非破坏性的，但在我们真正使用BST的时候，这是一个不必要的保护措施。本节将展示如何定义更符合实际应用的具有破坏性的插入函数和删除函数。

图 12.8 展示了如何定义一个具有破坏性的 `bst-insert` (第 72 页「译者注：第 4.7 节」)。相同的输入参数，能够得到相同返回值。唯一的区别是，它将修改作为第二个参数输入的 BST。 在第 2.12 节中说过，具有破坏性并不意味着一个函数调用具有副作用。的确如此，如果你想使用 `bst-insert!` 构造一个 BST，你必须像调用 `bst-insert` 那样调用它：

```
> (setf *bst* nil)
NIL
> (dolist (x '(7 2 9 8 4 1 5 12))
(setf *bst* (bst-insert! x *bst* #'<)))
NIL
```

```
(defun bst-insert! (obj bst <)
  (if (null bst)
      (make-node :elt obj)
      (progn (bsti obj bst <)
             bst)))

(defun bsti (obj bst <)
  (let ((elt (node-elt bst)))
    (if (eql obj elt)
        bst
        (if (funcall < obj elt)
            (let ((l (node-l bst)))
              (if l
                  (bsti obj l <)
                  (setf (node-l bst)
                        (make-node :elt obj))))
            (let ((r (node-r bst)))
              (if r
                  (bsti obj r <)
                  (setf (node-r bst)
                        (make-node :elt obj)))))))))
```

图 **12.8:** 二叉搜索树：破坏性插入

你也可以为 BST 定义一个类似 push 的功能，但这超出了本书的范围。(好奇的话，可以

参考第 409 页 「译者注：即备注 204 」 的宏定义。)

与 bst-remove (第 74 页「译者注：第 4.7 节」) 对应，图 12.9 展示了一个破坏性版本的 bst-delete 。同 delete 一样，我们调用它并不是因为它的副作用。你应该像调用 bst-remove 那样调用 bst-delete :

```
> (setf *bst* (bst-delete 2 *bst* #'<) )
#<7>
> (bst-find 2 *bst* #'<)
NIL
```

```
(defun bst-delete (obj bst <)
  (if bst (bstd obj bst nil nil <))
  bst)

(defun bstd (obj bst prev dir <)
  (let ((elt (node-elt bst)))
    (if (eql elt obj)
        (let ((rest (percolate! bst)))
          (case dir
            (:l (setf (node-l prev) rest))
            (:r (setf (node-r prev) rest))))
      (if (funcall < obj elt)
          (if (node-l bst)
              (bstd obj (node-l bst) bst :l <))
          (if (node-r bst)
              (bstd obj (node-r bst) bst :r <))))))

(defun percolate! (bst)
  (cond ((null (node-l bst))
         (if (null (node-r bst))
             nil
             (rperc! bst)))
        ((null (node-r bst)) (lperc! bst))
        (t (if (zerop (random 2))
               (lperc! bst)
               (rperc! bst)))))

(defun lperc! (bst)
  (setf (node-elt bst) (node-elt (node-l bst)))
  (percolate! (node-l bst)))

(defun rperc! (bst)
  (setf (node-elt bst) (node-elt (node-r bst)))
  (percolate! (node-r bst)))
```

图 **12.9:** 二叉搜索树：破坏性删除

译注：此范例已被回报为错误的，一个修复的版本请造访这里 [https://gist.github.com/2868339]。

12.6 示例：双向链表 (Example: Doubly-Linked Lists)

普通的 Lisp 列表是单向链表，这意味着其指针指向一个方向：我们可以获取下一个元素，但不能获取前一个。在双向链表[http://zh.wikipedia.org/wiki/%E5%8F%8C%E5%90%91%E9%93%BE%E8%A1%A8]中，指针指向两个方向，我们获取前一个元素和下一个元素都很容易。这一节将介绍如何创建和操作双向链表。

图 12.10 展示了如何用结构来实现双向链表。将 cons 看成一种结构，它有两个字段：指向数据的 car 和指向下一个元素的 cdr 。要实现一个双向链表，我们需要第三个字段，用来指向前一个元素。图 12.10 中的 defstruct 定义了一个含有三个字段的对象 dl (用于“双向链接”)，我们将用它来构造双向链表。dl 的 data 字段对应一个 cons 的 car，next 字段对应 cdr 。 prev 字段就类似一个 cdr ，指向另外一个方向。(图 12.11 是一个含有三个元素的双向链表。) 空的双向链表为 nil ，就像空的列表一样。

```
(defstruct (dl (:print-function print-dl))
  prev data next)

(defun print-dl (dl stream depth)
  (declare (ignore depth))
  (format stream "#<DL ~A>" (dl->list dl)))

(defun dl->list (lst)
  (if (dl-p lst)
      (cons (dl-data lst) (dl->list (dl-next lst)))
      lst))

(defun dl-insert (x lst)
  (let ((elt (make-dl :data x :next lst)))
    (when (dl-p lst)
      (if (dl-prev lst)
          (setf (dl-next (dl-prev lst)) elt
                (dl-prev elt) (dl-prev lst)))
      (setf (dl-prev lst) elt))
    elt))

(defun dl-list (&rest args)
  (reduce #'dl-insert args
          :from-end t :initial-value nil))

(defun dl-remove (lst)
  (if (dl-prev lst)
      (setf (dl-next (dl-prev lst)) (dl-next lst)))
  (if (dl-next lst)
      (setf (dl-prev (dl-next lst)) (dl-prev lst)))
  (dl-next lst))
```

图 **12.10:** 构造双向链表

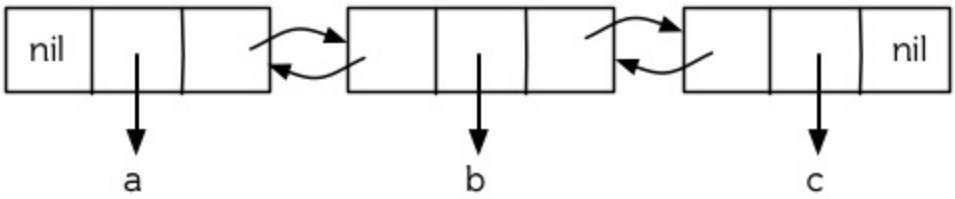

图 **12.11:** 一个双向链表。

为了便于操作，我们为双向链表定义了一些实现类似 `car` ， `cdr` ， `consp` 功能的函数：`dl-data` ， `dl-next` 和 `dl-p` 。 `dl->list` 是 `dl` 的打印函数(`print-function`)，其返回一个包含 `dl` 所有元素的普通列表。

函数 `dl-insert` 就像针对双向链表的 `cons` 操作。至少，它就像 `cons` 一样，是一个基本构建函数。与 `cons` 不同的是，它实际上要修改作为第二个参数传递给它的双向链表。在这种情况下，这是自然而然的。我们 `cons` 内容到普通列表前面，不需要对普通列表的 `rest` (译者注： `rest` 即 `cdr` 的另一种表示方法，这里的 `rest` 是对通过 `cons` 构建后列表来说的，即修改之前的列表) 做任何修改。但是要在双向链表的前面插入元素，我们不得不修改列表的 `rest` (这里的 `rest` 即指没修改之前的双向链表) 的 `prev` 字段来指向这个新元素。

几个普通列表可以共享同一个尾端。因为双向链表的尾端不得不指向它的前一个元素，所以不可能存在两个双向链表共享同一个尾端。如果 `dl-insert` 不具有破坏性，那么它不得不复制其第二个参数。

单向链表(普通列表)和双向链表另一个有趣的区别是，如何持有它们。我们使用普通列表的首端，来表示单向链表，如果将列表赋值给一个变量，变量可以通过保存指向列表第一个 `cons` 的指针来持有列表。但是双向链表是双向指向的，我们可以用任何一个点来持有双向链表。 `dl-insert` 另一个不同于 `cons` 的地方在于 `dl-insert` 可以在双向链表的任何位置插入新元素，而 `cons` 只能在列表的首端插入。

函数 `dl-list` 是对于 `dl` 的类似 `list` 的功能。它接受任意多个参数，它会返回一个包含以这些参数作为元素的 `dl` ：

```
> (dl-list 'a 'b 'c)
#<DL (A B C)>
```

它使用了 `reduce` 函数 (并设置其 `from-end` 参数为 `true`， `initial-value` 为 `nil`)，其功能等价于

```
(dl-insert 'a (dl-insert 'b (dl-insert 'c nil)) )
```

如果将 dl-list 定义中的 #'dl-insert 换成 #'cons，它就相当于 list 函数了。下面是 dl-list 的一些常见用法：

```
> (setf dl (dl-list 'a 'b))
#<DL (A B)>
> (setf dl (dl-insert 'c dl))
#<DL (C A B)>
> (dl-insert 'r (dl-next dl))
#<DL (R A B)>
> dl
#<DL (C R A B)>
```

最后，dl-remove 的作用是从双向链表中移除一个元素。同 dl-insert 一样，它也是具有破坏性的。

12.7 环状结构 (Circular Structure)

将列表结构稍微修改一下，就可以得到一个环形列表。存在两种环形列表。最常用的一种是其顶层列表结构是一个环的，我们把它叫做 cdr-circular ，因为环是由一个 cons 的 cdr 构成的。

构造一个单元素的 cdr-circular 列表，可以将一个列表的 cdr 设置成列表自身：

```
> (setf x (list 'a))
(A)
> (progn (setf (cdr x) x) nil)
NIL
```

这样 x 就是一个环形列表，其结构如图 12.12 (左) 所示。

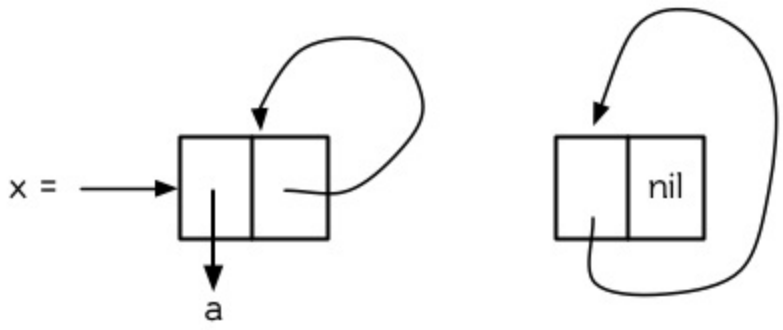

图 **12.12** 环状列表。

如果 Lisp 试着打印我们刚刚构造的结构，将会显示 (a a a a a ..... —— 无限个 a)。但如果设置全局变量 *print-circle* 为 t 的话，Lisp 就会采用一种方式打印出一个能代表环形结构的对象：

```
> (setf *print-circle* t )
T
> x
#1=(A . #1#)
```

如果你需要，你也可以使用 #n= 和 #n# 这两个读取宏，来自己表示共享结构。

cdr-cicular 列表十分有用，比如，可以用来表示缓冲区、池。下面这个函数，可以将一个普通的非空列表，转换成一个对应的 cdr-cicular 列表：

```
(defun circular (lst)
        (setf (cdr (last lst)) lst))
```

另外一种环状列表叫做 car-circular 列表。car-circular 列表是一个树，并将其自身当作自己的子树的结构。因为环是通过一个 cons 的 car 形成的，所以叫做 car-circular。这里构造了一个 car-circular，它的第二个元素是它自身：

```
> (let ((y (list 'a )))
(setf (car y) y)
      y)
#i=(#i#)
```

图 12.12 (右) 展示了其结构。这个 car-circular 是一个正规列表。cdr-circular 列表都不是正规列表，除开一些特殊情况 car-circular 列表是正规列表。

一个列表也可以既是 car-circular，又是 cdr-circular。一个 cons 的 car 和 cdr 均是其自身：

```
> (let ((c (cons 11)) )
      (setf (car c) c
            (cdr c) c)
      c)
#1=(#1# . #1#)
```

很难想像这样的一个列表有什么用。实际上，了解环形列表的主要目的就是为了避免因为偶然因素构造出了环形列表，因为，将一个环形列表传给一个函数，如果该函数遍历这个环形列表，它将进入死循环。

环形结构的这种问题在列表以外的其他对象中也存在。比如，一个数组可以将数组自身当作其元素：

```
> (setf *print-array* t )
T
> (let ((a (make-array 1)) )
          (setf (aref a 0) a)
          a)
#1=#(#1#)
```

实际上，任何可以包含元素的对象都可能包含其自身作为元素。

用 defstruct 构造出环形结构是相当常见的。比如，一个结构 c 是一颗树的元素，它的 parent 字段所指向的结构 p 的 child 字段也恰好指向 c 。

```
> (progn (defstruct elt
          (parent nil ) (child nil) )
     (let ((c (make-elt) )
                (p (make-elt)) )
          (setf (elt-parent c) p
                     (elt-child p) c)
          c) )
#1=#S(ELT PARENT #S(ELT PARENT NIL CHILD #1#) CHILD NIL)
```

要实现像这样一个结构的打印函数 (print-function)，我们需要将全局变量 *print-circle* 绑定为 t ，或者避免打印可能构成环的字段。

12.8 常量结构 (Constant Structure)

因为常量实际上是程序代码的一部分，所以我们也不应该修改他们，或者是不经意地写了自重写的代码。一个通过 quote 引用的列表是一个常量，所以一定要小心，不要修改被引用的列表的任何 cons。比如，如果我们用下面的代码，来测试一个符号是不是算术运算符：

```
(defun arith-op (x)
(member x '(+ - * /)))
```

如果被测试的符号是算术运算符，它的返回值将至少一个被引用列表的一部分。如果我们修改了其返回值，

```
> (nconc (arith-op '*) '(as i t were))
(* / AS IT WERE)
```

那么我就会修改 arith-op 函数中的一个列表，从而改变了这个函数的功能：

```
> (arith-op 'as )
(AS IT WERE)
```

写一个返回常量结构的函数，并不一定是错误的。但当你考虑使用一个破坏性的操作是否安全的时候，你必须考虑到这一点。

有几个其它方法来实现 `arith-op`，使其不返回被引用列表的部分。一般地，我们可以通过将其中的所有引用(`quote`) 替换成 `list` 来确保安全，这使得它每次被调用都将返回一个新的列表：

```
(defun arith-op (x)
        (member x (list '+ '- '* '/)))
```

这里，使用 `list` 是一种低效的解决方案，我们应该使用 `find` 来替代 `member`：

```
(defun arith-op (x)
        (find x '(+ - * /)))
```

这一节讨论的问题似乎只与列表有关，但实际上，这个问题存在于任何复杂的对象中：数组，字符串，结构，实例等。你不应该逐字地去修改程序的代码段。

即使你想写自修改程序，通过修改常量来实现并不是个好办法。编译器将常量编译成了代码，破坏性的操作可能修改它们的参数，但这些都是没有任何保证的事情。如果你想写自修改程序，正确的方法是使用闭包 (见 6.5 节)。

Chapter 12 总结 (Summary)

1. 两个列表可以共享一个尾端。多个列表可以以树的形式共享结构，而不是共享顶层列表结构。可通过拷贝方式来避免共用结构。
2. 共享结构通常可以被忽略，但如果你要修改列表，则需要特别注意。因为修改一个含共享结构的列表可能修改所有共享该结构的列表。
3. 队列可以被表示成一个 `cons` ，其的 `car` 指向队列的第一个元素， `cdr` 指向队列的最后一个元素。
4. 为了提高效率，破坏性函数允许修改其输入参数。
5. 在某些应用中，破坏性的实现更适用。
6. 列表可以是 `car-circular` 或 `cdr-circular` 。 Lisp 可以表示圆形结构和共享结构。
7. 不应该去修改的程序代码段中的常量形式。

Chapter 12 练习 (Exercises)

1. 画三个不同的树，能够被打印成 `((A) (A) (A))` 。写一个表达式来生成它们。
2. 假设 `make-queue` ， `enqueue` 和 `dequeue` 是按照图 12.7 中的定义，用箱子表式法画

出下面每一步所得到的队列的结构图：

```
> (setf q (make-queue))
(NIL)
> (enqueue 'a q)
(A)
> (enqueue 'b q)
(A B)
> (dequeue q)
A
```

3. 定义一个函数 `copy-queue`，可以返回一个 queue 的拷贝。
4. 定义一个函数，接受两个输入参数 `object` 和 `queue`，能将 `object` 插入到 `queue` 的首端。
5. 定义一个函数，接受两个输入参数 `object` 和 `queue`，能具有破坏性地将 `object` 的第一个实例 (`eql` 等价地) 移到 `queue` 的首端。
6. 定义一个函数，接受两个输入参数 `object` 和 `lst` (`lst` 可能是 `cdr-circular` 列表)，如果 `object` 是 `lst` 的成员时返回真。
7. 定义一个函数，如果它的参数是一个 `cdr-circular` 则返回真。
8. 定义一个函数，如果它的参数是一个 `car-circular` 则返回真。

脚注

[1] 比如，在 Common Lisp 中，修改一个被用作符号名的字符串被认为是一种错误，因为内部的定义并没声明它是从参数复制来的，所以必须假定修改传入内部的任何参数中的字符串来创建新的符号是错误的。

[2] 函数名称中 n 的含义是 “non-consing”。一些具有破坏性的函数以 n 开头。

第十三章：速度

Lisp 实际上是两种语言：一种能写出快速执行的程序，一种则能让你快速的写出程序。在程序开发的早期阶段，你可以为了开发上的便捷舍弃程序的执行速度。一旦程序的结构开始固化，你就可以精炼其中的关键部分以使得它们执行的更快。

由于各个 Common Lisp 实现间的差异，很难针对优化给出通用的建议。在一个实现上使程序变快的修改也许在另一个实现上会使得程序变慢。这是难免的事儿。越强大的语言，离机器底层就越远，离机器底层越远，语言的不同实现沿着不同路径趋向它的可能性就越大。因此，即便有一些技巧几乎一定能够让程序运行的更快，本章的目的也只是建议而不是规定。

13.1 瓶颈规则 (The Bottleneck Rule)

不管是什么实现，关于优化都可以整理出三点规则：它应该关注瓶颈，它不应该开始的太早，它应该始于算法。

也许关于优化最重要的事情就是要意识到，程序中的大部分执行时间都是被少数瓶颈所消耗掉的。 正如高德纳 [http://en.wikipedia.org/wiki/Donald_Knuth]所说，“在一个与 I/O 无关 (Non-I/O bound) 的程序中，大部分的运行时间集中在大概 3% 的源代码中。” λ [http://acl.readthedocs.org/en/latest/zhCN/notes-cn.html#notes-213] 优化程序的这一部分将会使得它的运行速度明显的提升；相反，优化程序的其他部分则是在浪费时间。

因此，优化程序时关键的第一步就是找到瓶颈。许多 Lisp 实现都提供性能分析器 (profiler) 来监视程序的运行并报告每一部分所花费的时间量。 为了写出最为高效的代码，性能分析器非常重要，甚至是必不可少的。 如果你所使用的 Lisp 实现带有性能分析器，那么请在进行优化时使用它。另一方面，如果实现没有提供性能分析器的话，那么你就不得不通过猜测来寻找瓶颈，而且这种猜测往往都是错的！

瓶颈规则的一个推论是，不应该在程序的初期花费太多的精力在优化上。高德纳 [http://en.wikipedia.org/wiki/Donald_Knuth]对此深信不疑：“过早的优化是一切 (至少是大多数) 问题的源头。” λ [http://acl.readthedocs.org/en/latest/zhCN/notes-cn.html#notes-214] 在刚开始写程序的时候，通常很难看清真正的瓶颈在哪，如果这个时候进行优化，你很可能是在浪费时间。优化也会使程序的修改变得更加困难，边写程序边优化就像是在用风干非常快的颜料来画画一样。

在适当的时候做适当的事情，可以让你写出更优秀的程序。 Lisp 的一个优点就是能让你用两种不同的工作方式来进行开发：很快地写出运行较慢的代码，或者，放慢写程序

的速度，精雕细琢，从而得出运行得较快的代码。

在程序开发的初期阶段，工作通常在第一种模式下进行，只有当性能成为问题的时候，才切换到第二种模式。 对于非常底层的语言，比如汇编，你必须优化程序的每一行。但这么做会浪费你大部分的精力，因为瓶颈仅仅是其中很小的那部分代码。一个更加抽象的语言能够让你把主要精力集中在瓶颈上， 达到事半功倍的效果。

当真正开始优化的时候，还必须从最顶端入手。 在使用各种低层次的编码技巧 (low-level coding tricks) 之前，请先确保你已经使用了最为高效的算法。 这么做的潜在好处相当大 —— 甚至可能大到你都不再需要玩那些奇淫技巧。 当然本规则还是要和前一个规则保持平衡。 有些时候，关于算法的决策必须尽早进行。

13.2 编译 (Compilation)

有五个参数可以控制代码的编译方式： *speed* (速度)代表编译器产生代码的速度；*compilation-speed* (编译速度)代表程序被编译的速度； *safety* (安全) 代表要对目标代码进行错误检查的数量； *space* (空间)代表目标代码的大小和内存需求量；最后， *debug* (调试)代表为了调试而保留的信息量。

> **Note:** 交互与解释 (INTERACTIVE VS. INTERPRETED)
>
> Lisp 是一种交互式语言 (Interactive Language)，但是交互式的语言不必都是解释型的。早期的 Lisp 都通过解释器实现，因此认为 Lisp 的特质都依赖于它是被解释的想法就这么产生了。但这种想法是错误的：Common Lisp 既是编译型语言，又是解释型语言。
>
> 至少有两种 Common Lisp 实现甚至都不包含解释器。在这些实现中，输入到顶层的表达式在求值前会被编译。因此，把顶层叫做解释器的这种说法，不仅是落伍的，甚至还是错误的。

编译参数不是真正的变量。它们在声明中被分配从 `0` (最不重要) 到 `3` (最重要) 的权值。如果一个主要的瓶颈发生在某个函数的内层循环中，我们或许可以添加如下的声明：

```
(defun bottleneck (...)
  (do (...)
      (...)
    (do (...)
        (...)
      (declare (optimize (speed 3) (safety 0)))
      ...)))
```

一般情况下，应该在代码写完并且经过完善测试之后，才考虑加上那么一句声明。

要让代码在任何情况下都尽可能地快，可以使用如下声明：

```
(declaim (optimize (speed 3)
                   (compilation-speed 0)
                   (safety 0)
                   (debug 0)))
```

考虑到前面提到的瓶颈规则 [1]，这种苛刻的做法可能并没有什么必要。

另一类特别重要的优化就是由 Lisp 编译器完成的尾递归优化。当 *speed* (速度)的权值最大时，所有支持尾递归优化的编译器都将保证对代码进行这种优化。

如果在一个调用返回时调用者中没有残余的计算，该调用就被称为尾递归。下面的代码返回列表的长度：

```
(defun length/r (lst)
  (if (null lst)
      0
      (1+ (length/r (cdr lst)))))
```

这个递归调用不是尾递归，因为当它返回以后，它的值必须传给 `1+` 。相反，这是一个尾递归的版本，

```
(defun length/rt (lst)
  (labels ((len (lst acc)
             (if (null lst)
                 acc
                 (len (cdr lst) (1+ acc)))))
    (len lst 0)))
```

更准确地说，局部函数 `len` 是尾递归调用，因为当它返回时，调用函数已经没什么事情可做了。 和 `length/r` 不同的是，它不是在递归回溯的时候构建返回值，而是在递归调用的过程中积累返回值。 在函数的最后一次递归调用结束之后， `acc` 参数就可以作为函数的结果值被返回。

出色的编译器能够将一个尾递归编译成一个跳转 (goto)，因此也能将一个尾递归函数编译成一个循环。在典型的机器语言代码中，当第一次执行到表示 `len` 的指令片段时，栈上会有信息指示在返回时要做些什么。由于在递归调用后没有残余的计算，该信息对第二层调用仍然有效：第二层调用返回后我们要做的仅仅就是从第一层调用返回。 因此，当进行第二层调用时，我们只需给参数设置新的值，然后跳转到函数的起始处继续执行就可以了，没有必要进行真正的函数调用。

另一个利用函数调用抽象，却没有开销的方法是使函数内联编译。对于那些调用开销比函数体的执行代价还高的小型函数来说，这种技术非常有价值。例如，以下代码用于判

断列表是否仅有一个元素：

```
(declaim (inline single?))

(defun single? (lst)
  (and (consp lst) (null (cdr lst))))
```

因为这个函数是在全局被声明为内联的，引用了 `single?` 的函数在编译后将不需要真正的函数调用。 [2] 如果我们定义一个调用它的函数，

```
(defun foo (x)
  (single? (bar x)))
```

当 `foo` 被编译后， `single?` 函数体中的代码将会被编译进 `foo` 的函数体，就好像我们直接写以下代码一样：

```
(defun foo (x)
  (let ((lst (bar x)))
    (and (consp lst) (null (cdr lst)))))
```

内联编译有两个限制： 首先，递归函数不能内联。 其次，如果一个内联函数被重新定义，我们就必须重新编译调用它的任何函数，否则调用仍然使用原来的定义。

在一些早期的 Lisp 方言中，有时候会使用宏（ 10.2 节）来避免函数调用。这种做法在 Common Lisp 中通常是没有必要的。

不同 Lisp 编译器的优化方式千差万别。 如果你想了解你的编译器为某个函数生成的代码，试着调用 `disassemble` 函数：它接受一个函数或者函数名，并显示该函数编译后的形式。 即便你看到的东西是完全无法理解的，你仍然可以使用 `disassemble` 来判断声明是否起效果：编译函数的两个版本，一个使用优化声明，另一个不使用优化声明，然后观察由 `disassemble` 显示的两组代码之间是否有差异。 同样的技巧也可以用于检验函数是否被内联编译。 不论情况如何，都请优先考虑使用编译参数，而不是手动调优的方式来优化代码。

13.3 类型声明 (Type Declarations)

如果 Lisp 不是你所学的第一门编程语言，那么你也许会感到困惑，为什么这本书还没说到类型声明这件事来？毕竟，在很多流行的编程语言中，类型声明是必须要做的。

在不少编程语言里，你必须为每个变量声明类型，并且变量也只可以持有与该类型相一致的值。 这种语言被称为强类型(*strongly typed*) 语言。 除了给程序员们徒增了许多负担外，这种方式还限制了你能做的事情。 使用这种语言，很难写出那些需要多种类型

的参数一起工作的函数，也很难定义出可以包含不同种类元素的数据结构。 当然，这种方式也有它的优势，比如无论何时当编译器碰到一个加法运算，它都能够事先知道这是一个什么类型的加法运算。如果两个参数都是整数类型，编译器可以直接在目标代码中生成一个固定 (hard-wire) 的整数加法运算。

正如 2.15 节所讲，Common Lisp 使用一种更加灵活的方式：显式类型 (manifest typing) [3] 。有类型的是值而不是变量。变量可以用于任何类型的对象。

当然，这种灵活性需要付出一定的速度作为代价。 由于 + 可以接受好几种不同类型的数，它不得不在运行时查看每个参数的类型来决定采用哪种加法运算。

在某些时候，如果我们要执行的全都是整数的加法，那么每次查看参数类型的这种做法就说不上高效了。 Common Lisp 处理这种问题的方法是：让程序员尽可能地提示编译器。 比如说，如果我们提前就能知道某个加法运算的两个参数是定长数 (fixnums) ，那么就可以对此进行声明，这样编译器就会像 C 语言的那样为我们生成一个固定的整数加法运算。

因为显式类型也可以通过声明类型来生成高效的代码，所以强类型和显式类型两种方式之间的差别并不在于运行速度。 真正的区别是，在强类型语言中，类型声明是强制性的，而显式类型则不强加这样的要求。 在 Common Lisp 中，类型声明完全是可选的。它们可以让程序运行的更快，但(除非错误)不会改变程序的行为。

全局声明以 `declaim` 伴随一个或多个声明的形式来实现。一个类型声明是一个列表，包含了符号 `type` ，后跟一个类型名，以及一个或多个变量组成。举个例子，要为一个全局变量声明类型，可以这么写：

```
(declaim (type fixnum *count*))
```

在 ANSI Common Lisp 中，可以省略 `type` 符号，将声明简写为：

```
(declaim (fixnum *count*))
```

局部声明通过 `declare` 完成，它接受的参数和 `declaim` 的一样。 声明可以放在那些创建变量的代码体之前：如 `defun` 、 `lambda` 、 `let` 、 `do` ，诸如此类。 比如说，要把一个函数的参数声明为定长数，可以这么写：

```
(defun poly (a b x)
  (declare (fixnum a b x))
  (+ (* a (expt x 2)) (* b x)))
```

在类型声明中的变量名指的就是该声明所在的上下文中的那个变量 —— 那个通过赋值可以改变它的值的变量。

你也可以通过 `the` 为某个表达式的值声明类型。 如果我们提前就知道 `a` 、 `b` 和 `x` 是足够小的定长数，并且它们的和也是定长数的话，那么可以进行以下声明：

```
(defun poly (a b x)
  (declare (fixnum a b x))
  (the fixnum (+ (the fixnum (* a (the fixnum (expt x 2))))
                 (the fixnum (* b x)))))
```

看起来是不是很笨拙啊？幸运的是有两个原因让你很少会这样使用 `the` 把你的数值运算代码变得散乱不堪。其一是很容易通过宏，来帮你插入这些声明。其二是某些实现使用了特殊的技巧，即便没有类型声明的定长数运算也能足够快。

Common Lisp 中有相当多的类型 —— 恐怕有无数种类型那么多，如果考虑到你可以自己定义新的类型的话。 类型声明只在少数情况下至关重要，可以遵照以下两条规则来进行：

1. 当函数可以接受若干不同类型的参数(但不是所有类型)时，可以对参数的类型进行声明。如果你知道一个对 `+` 的调用总是接受定长数类型的参数，或者一个对 `aref` 的调用第一个参数总是某种特定种类的数组，那么进行类型声明是值得的。
2. 通常来说，只有对类型层级中接近底层的类型进行声明，才是值得的：将某个东西的类型声明为 `fixnum` 或者 `simple-array` 也许有用，但将某个东西的类型声明为 `integer` 或者 `sequence` 或许就没用了。

类型声明对内容复杂的对象特别重要，这包括数组、结构和对象实例。这些声明可以在两个方面提升效率：除了可以让编译器来决定函数参数的类型以外，它们也使得这些对象可以在内存中更高效地表示。

如果对数组元素的类型一无所知的话，这些元素在内存中就不得不用一块指针来表示。但假如预先就知道数组包含的元素仅仅是 —— 比方说 —— 双精度浮点数 (double-floats)，那么这个数组就可以用一组实际的双精度浮点数来表示。这样数组将占用更少的空间，因为我们不再需要额外的指针指向每一个双精度浮点数；同时，对数组元素的访问也将更快，因为我们不必沿着指针去读取和写元素。

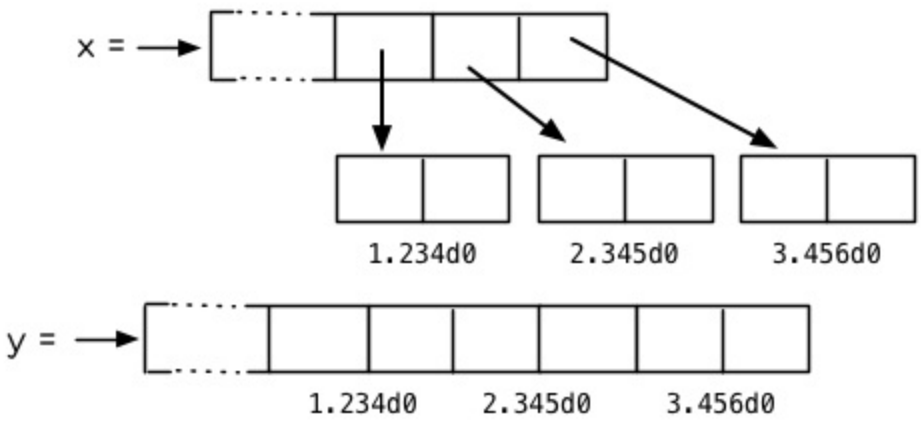

图 **13.1**：指定元素类型的效果

你可以通过 `make-array` 的 `:element-type` 参数指定数组包含值的种类。这样的数组被称为特化数组(specialized array)。 图 13.1 为我们展示了如下代码在多数实现上求值后发生的事情：

```
(setf x (vector 1.234d0 2.345d0 3.456d0)
      y (make-array 3 :element-type 'double-float)
      (aref y 0) 1.234d0
      (aref y 1) 2.345d0
      (aref y 2) 3.456d0))
```

图 13.1 中的每一个矩形方格代表内存中的一个字 (a word of memory)。这两个数组都由未特别指明长度的头部 (header) 以及后续 三个元素的某种表示构成。对于 `x` 来说，每个元素都由一个指针表示。此时每个指针碰巧都指向双精度浮点数，但实际上我们可以存储任何类型的对象到这个向量中。对 `y` 来说，每个元素实际上都是双精度浮点数。 `y` 更快而且占用更少空间，但意味着它的元素只能是双精度浮点数。

注意我们使用 `aref` 来引用 `y` 的元素。一个特化的向量不再是一个简单向量，因此我们不再能够通过 `svref` 来引用它的元素。

除了在创建数组时指定元素的类型，你还应该在使用数组的代码中声明数组的维度以及它的元素类型。一个完整的向量声明如下：

```
(declare (type (vector fixnum 20) v))
```

以上代码声明了一个仅含有定长数，并且长度固定为 `20` 的向量。

```
(setf a (make-array '(1000 1000)
                    :element-type 'single-float
                    :initial-element 1.0s0))

(defun sum-elts (a)
  (declare (type (simple-array single-float (1000 1000))
                 a))
  (let ((sum 0.0s0))
    (declare (type single-float sum))
    (dotimes (r 1000)
      (dotimes (c 1000)
        (incf sum (aref a r c))))
    sum))
```

图 **13.2** 对数组元素求和

最为通用的数组声明形式由数组类型以及紧接其后的元素类型和一个维度列表构成：

```
(declare (type (simple-array fixnum (4 4)) ar))
```

图 13.2 展示了如何创建一个 1000×1000 的单精度浮点数数组，以及如何编写一个将该数组元素相加的函数。数组以行主序 (row-major order)存储，遍历时也应尽可能按此顺序进行。

我们将用 `time` 来比较 `sum-elts` 在有声明和无声明两种情况下的性能。 `time` 宏显示表达式求值所花费时间的某种度量(取决于实现)。对被编译的函数求取时间才是有意义的。在某个实现中，如果我们以获取最快速代码的编译参数编译 `sum-elts` ，它将在不到半秒的时间内返回：

```
> (time (sum-elts a))
User Run Time = 0.43 seconds
1000000.0
```

如果我们把 *sum-elts* 中的类型声明去掉并重新编译它，同样的计算将花费超过5秒的时间：

```
> (time (sum-elts a))
User Run Time = 5.17 seconds
1000000.0
```

类型声明的重要性 —— 特别是对数组和数来说 —— 怎么强调都不过分。上面的例子中，仅仅两行代码就可以让 `sum-elts` 变快 12 倍。

13.4 避免垃圾 (Garbage Avoidance)

Lisp 除了可以让你推迟考虑变量的类型以外，它还允许你推迟对内存分配的考虑。 在程序的早期阶段，暂时忽略内存分配和臭虫等问题，将有助于解放你的想象力。 等到程序基本固定下来以后，就可以开始考虑怎么减少动态分配，从而让程序运行得更快。

但是，并不是构造（consing）用得少的程序就一定快。 多数 Lisp 实现一直使用着差劲的垃圾回收器，在这些实现中，过多的内存分配容易让程序运行变得缓慢。 因此，『高效的程序应该尽可能地减少 `cons` 的使用』这种观点，逐渐成为了一种传统。 最近这种传统开始有所改变，因为一些实现已经用上了相当先进（sophisticated）的垃圾回收器，它们实行一种更为高效的策略：创建新的对象，用完之后抛弃而不是进行回收。

本节介绍了几种方法，用于减少程序中的构造。 但构造数量的减少是否有利于加快程序的运行，这一点最终还是取决于实现。 最好的办法就是自己去试一试。

减少构造的办法有很多种。 有些办法对程序的修改非常少。 例如，最简单的方法就是使用破坏性函数。 下表罗列了一些常用的函数，以及这些函数对应的破坏性版本。

安全	破坏性
append	nconc
reverse	nreverse
remove	delete
remove-if	delete-if
remove-duplicates	delete-duplicates
subst	nsubst
subst-if	nsubst-if
union	nunion
intersection	nintersection
set-difference	nset-difference

当确认修改列表是安全的时候，可以使用 `delete` 替换 `remove` ，用 `nreverse` 替换 `reverse` ，诸如此类。

即便你想完全摆脱构造，你也不必放弃在运行中 (on the fly)创建对象的可能性。 你需要做的是避免在运行中为它们分配空间和通过垃圾回收收回空间。通用方案是你自己预先分配内存块 (block of memory)，以及明确回收用过的块。预先可能意味着在编译期或者某些初始化例程中。具体情况还应具体分析。

例如，当情况允许我们利用一个有限大小的堆栈时，我们可以让堆栈在一个已经分配了空间的向量中增长或缩减，而不是构造它。Common Lisp 内置支持把向量作为堆栈使用。如果我们传给 `make-array` 可选的 `fill-pointer` 参数，我们将得到一个看起来可扩展的向量。 `make-array` 的第一个参数指定了分配给向量的存储量，而 `fill-pointer` 指定了初始有效长度：

```
> (setf *print-array* t)
T
> (setf vec (make-array 10 :fill-pointer 2
                           :initial-element nil))
#(NIL NIL)
```

我们刚刚制造的向量对于操作序列的函数来说，仍好像只含有两个元素，

```
> (length vec)
2
```

但它能够增长直到十个元素。因为 `vec` 有一个填充指针，我们可以使用 `vector-push` 和 `vector-pop` 函数推入和弹出元素，就像它是一个列表一样：

```
> (vector-push 'a vec)
2
> vec
```

```
#(NIL NIL A)
> (vector-pop vec)
A
> vec
#(NIL NIL)
```

当我们调用 vector-push 时，它增加填充指针并返回它过去的值。只要填充指针小于 make-array 的第一个参数，我们就可以向这个向量中推入新元素；当空间用尽时，vector-push 返回 nil 。目前我们还可以向 vec 中推入八个元素。

使用带有填充指针的向量有一个缺点，就是它们不再是简单向量了。我们不得不使用 aref 来代替 svref 引用元素。代价需要和潜在的收益保持平衡。

```
(defconstant dict (make-array 25000 :fill-pointer 0))

(defun read-words (from)
  (setf (fill-pointer dict) 0)
  (with-open-file (in from :direction :input)
    (do ((w (read-line in nil :eof)
            (read-line in nil :eof)))
        ((eql w :eof))
      (vector-push w dict))))

(defun xform (fn seq) (map-into seq fn seq))

(defun write-words (to)
  (with-open-file (out to :direction :output
                          :if-exists :supersede)
    (map nil #'(lambda (x)
                 (fresh-line out)
                 (princ x out))
           (xform #'nreverse
                  (sort (xform #'nreverse dict)
                        #'string<)))))
```

图 **13.3** 生成同韵字辞典

当应用涉及很长的序列时，你可以用 map-into 代替 map 。 map-into 的第一个参数不是一个序列类型，而是用来存储结果的，实际的序列。这个序列可以是该函数接受的其他序列参数中的任何一个。所以，打个比方，如果你想为一个向量的每个元素加 1，你可以这么写：

```
(setf v (map-into v #'1+ v))
```

图 13.3 展示了一个使用大向量应用的例子：一个生成简单的同韵字辞典 (或者更确切的说，一个不完全韵辞典)的程序。函数 read-line 从一个每行仅含有一个单词的文件中读取单词，而函数 write-words 将它们按照字母的逆序打印出来。比如，输出的起始可

能是

```
a amoeba alba samba marimba...
```

结束是

```
...megahertz gigahertz jazz buzz fuzz
```

利用填充指针和 `map-into`，我们可以把程序写的既简单又高效。

在数值应用中要当心大数 (bignums)。大数运算需要构造，因此也就会比较慢。 即使程序的最后结果为大数，但是，通过调整计算，将中间结果保存在定长数中，这种优化也是有可能的。

另一个避免垃圾回收的方法是，鼓励编译器在栈上分配对象而不是在堆上。 如果你知道只是临时需要某个东西，你可以通过将它声明为 `dynamic extent` 来避免在堆上分配空间。

通过一个动态范围 (dynamic extent)变量声明，你告诉编译器，变量的值应该和变量保持相同的生命期。 什么时候值的生命期比变量长呢？这里有个例子：

```
(defun our-reverse (lst)
  (let ((rev nil))
    (dolist (x lst)
      (push x rev))
    rev))
```

在 `our-reverse` 中，作为参数传入的列表以逆序被收集到 `rev` 中。当函数返回时，变量 `rev` 将不复存在。 然而，它的值 —— 一个逆序的列表 —— 将继续存活：它被送回调用函数，一个知道它的命运何去何从的地方。

相比之下，考虑如下 `adjoin` 实现：

```
(defun our-adjoin (obj lst &rest args)
  (if (apply #'member obj lst args)
      lst
      (cons obj lst)))
```

在这个例子里，我们可以从函数的定义看出， `args` 参数中的值 (列表) 哪儿也没去。它不必比存储它的变量活的更久。在这种情形下把它声明为动态范围的就比较有意义。如果我们加上这样的声明：

```
(defun our-adjoin (obj lst &rest args)
  (declare (dynamic-extent args))
```

```
  (if (apply #'member obj lst args)
      lst
      (cons obj lst)))
```

那么编译器就可以 (但不是必须)在栈上为 `args` 分配空间，在 `our-adjoin` 返回后，它将自动被释放。

13.5 示例: 存储池 (Example: Pools)

对于涉及数据结构的应用，你可以通过在一个存储池 (pool)中预先分配一定数量的结构来避免动态分配。当你需要一个结构时，你从池中取得一份，当你用完后，再把它送回池中。为了演示存储池的使用，我们将快速的编写一段记录港口中船舶数量的程序原型 (prototype of a program)，然后用存储池的方式重写它。

```
(defparameter *harbor* nil)

(defstruct ship
  name flag tons)

(defun enter (n f d)
  (push (make-ship :name n :flag f :tons d)
        *harbor*))

(defun find-ship (n)
  (find n *harbor* :key #'ship-name))

(defun leave (n)
  (setf *harbor*
        (delete (find-ship n) *harbor*)))
```

图 **13.4** 港口

图 13.4 中展示的是第一个版本。 全局变量 `harbor` 是一个船只的列表， 每一艘船只由一个 `ship` 结构表示。 函数 `enter` 在船只进入港口时被调用； `find-ship` 根据给定名字 (如果有的话) 来寻找对应的船只；最后， `leave` 在船只离开港口时被调用。

一个程序的初始版本这么写简直是棒呆了，但它会产生许多的垃圾。当这个程序运行时，它会在两个方面构造：当船只进入港口时，新的结构将会被分配；而 `harbor` 的每一次增大都需要使用构造。

我们可以通过在编译期分配空间来消除这两种构造的源头 (sources of consing)。图 13.5 展示了程序的第二个版本，它根本不会构造。

```
(defconstant pool (make-array 1000 :fill-pointer t))
```

```
(dotimes (i 1000)
  (setf (aref pool i) (make-ship)))

(defconstant harbor (make-hash-table :size 1100
                                     :test #'eq))

(defun enter (n f d)
  (let ((s (if (plusp (length pool))
               (vector-pop pool)
               (make-ship))))
    (setf (ship-name s)        n
          (ship-flag s)        f
          (ship-tons s)        d
          (gethash n harbor) s)))

(defun find-ship (n) (gethash n harbor))

(defun leave (n)
  (let ((s (gethash n harbor)))
    (remhash n harbor)
    (vector-push s pool)))
```

图 **13.5** 港口（第二版）

严格说来，新的版本仍然会构造，只是不在运行期。在第二个版本中， `harbor` 从列表变成了哈希表，所以它所有的空间都在编译期分配了。 一千个 `ship` 结构体也会在编译期被创建出来，并被保存在向量池(vector pool) 中。(如果 `:fill-pointer` 参数为 `t` ，填充指针将指向向量的末尾。) 此时，当 `enter` 需要一个新的结构时，它只需从池中取来一个便是，无须再调用 `make-ship` 。 而且当 `leave` 从 `harbor` 中移除一艘 `ship` 时，它把它送回池中，而不是抛弃它。

我们使用存储池的行为实际上是肩负起内存管理的工作。这是否会让我们的程序更快仍取决于我们的 Lisp 实现怎样管理内存。总的说来，只有在那些仍使用着原始垃圾回收器的实现中，或者在那些对 GC 的不可预见性比较敏感的实时应用中才值得一试。

13.6 快速操作符 (Fast Operators)

本章一开始就宣称 Lisp 是两种不同的语言。就某种意义来讲这确实是正确的。如果你仔细看过 Common Lisp 的设计，你会发现某些特性主要是为了速度，而另外一些主要为了便捷性。

例如，你可以通过三个不同的函数取得向量给定位置上的元素： `elt` 、 `aref` 、 `svref` 。如此的多样性允许你把一个程序的性能提升到极致。 所以如果你可以使用 `svref` ，完事儿！ 相反，如果对某段程序来说速度很重要的话，或许不应该调用 `elt` ，它既可以用于数组也可以用于列表。

对于列表来说，你应该调用 `nth` ，而不是 `elt` 。然而只有单一的一个函数 —— `length` —— 用于计算任何一个序列的长度。为什么 Common Lisp 不单独为列表提供一个特定的版本呢？因为如果你的程序正在计算一个列表的长度，它在速度上已经输了。在这个例子中，就像许多其他的例子一样，语言的设计暗示了哪些会是快速的而哪些不是。

另一对相似的函数是 `eql` 和 `eq` 。前者是验证同一性 (identity) 的默认判断式，但如果你知道参数不会是字符或者数字时，使用后者其实更快。两个对象 *eq* 只有当它们处在相同的内存位置上时才成立。数字和字符可能不会与任何特定的内存位置相关，因此 `eq` 不适用于它们 (即便多数实现中它仍然能用于定长数)。对于其他任何种类的参数， `eq` 和 `eql` 将返回相同的值。

使用 `eq` 来比较对象总是最快的，因为 Lisp 所需要比较的仅仅是指向对象的指针。因此 `eq` 哈希表 (如图 13.5 所示) 应该会提供最快的访问。 在一个 `eq` 哈希表中， `gethash` 可以只根据指针查找，甚至不需要查看它们指向的是什么。然而，访问不是唯一要考虑的因素； *eq* 和 *eql* 哈希表在拷贝型垃圾回收算法 (copying garbage collection algorithm)中会引起额外的开销，因为垃圾回收后需要对一些哈希值重新进行计算 (rehashing)。如果这变成了一个问题，最好的解决方案是使用一个把定长数作为键值的 `eql` 哈希表。

当被调函数有一个余留参数时，调用 `reduce` 可能是比 `apply` 更高效的一种方式。例如，相比

```
(apply #'+ '(1 2 3))
```

写成如下可以更高效：

```
(reduce #'+ '(1 2 3))
```

它不仅有助于调用正确的函数，还有助于按照正确的方式调用它们。余留、可选和关键字参数 是昂贵的。只使用普通参数，函数调用中的参量会被调用者简单的留在被调者能够找到的地方。但其他种类的参数涉及运行时的处理。关键字参数是最差的。针对内置函数，优秀的编译器采用特殊的办法把使用关键字参量的调用编译成快速代码 (fast code)。但对于你自己编写的函数，避免在程序中对速度敏感的部分使用它们只有好处没有坏处。另外，不把大量的参量都放到余留参数中也是明智的举措，如果这可以避免的话。

不同的编译器有时也会有一些它们独到优化。例如，有些编译器可以针对键值是一个狭小范围中的整数的 `case` 语句进行优化。查看你的用户手册来了解那些实现特有的优化的建议吧。

13.7 二阶段开发 (Two-Phase Development)

在以速度至上的应用中，你也许想要使用诸如 C 或者汇编这样的低级语言来重写一个 Lisp 程序的某部分。你可以对用任何语言编写的程序使用这一技巧 —— C 程序的关键部分经常用汇编重写 —— 但语言越抽象，用两阶段（two phases）开发程序的好处就越明显。

Common Lisp 没有规定如何集成其他语言所编写的代码。这部分留给了实现决定，而几乎所有的实现都提供了某种方式来实现它。

使用一种语言编写程序然后用另一种语言重写它其中部分看起来可能是一种浪费。事实上，经验显示这是一种好的开发软件的方式。先针对功能、然后是速度比试着同时达成两者来的简单。

如果编程完全是一个机械的过程 —— 简单的把规格说明翻译为代码 —— 在一步中把所有的事情都搞定也许是合理的。但编程永远不是如此。不论规格说明多么精确， 编程总是涉及一定量的探索 —— 通常比任何人能预期到的还多的多。

一份好的规格说明，也许会让编程看起来像是简单的把它们翻译成代码的过程。这是一个普遍的误区。编程必定涉及探索，因为规格说明必定含糊不清。如果它们不含糊的话，它们就都算不上规格说明。

在其他领域，尽可能精准的规格说明也许是可取的。如果你要求一块金属被切割成某种形状，最好准确的说出你想要的。但这个规则不适用于软件，因为程序和规格说明由相同的东西构成：文本。你不可能编写出完全合意的规格说明。如果规格说明有那么精确的话，它们就变成程序了。 λ [http://acl.readthedocs.org/en/latest/zhCN/notes-cn.html#notes-229]

对于存在着可观数量的探索的应用 (再一次，比任何人承认的还要多，将实现分成两个阶段是值得的。而且在第一阶段中你所使用的手段 (medium) 不必就是最后的那个。例如，制作铜像的标准方法是先从粘土开始。你先用粘土做一个塑像出来，然后用它做一个模子，在这个模子中铸造铜像。在最后的塑像中是没有丁点粘土的，但你可以从铜像的形状中认识到它发挥的作用。试想下从一开始就只用一块儿铜和一个凿子来制造这么个一模一样的塑像要多难啊！出于相同的原因，首先用 Lisp 来编写程序，然后用 C 改写它，要比从头开始就用 C 编写这个程序要好。

Chapter 13 总结 (Summary)

1. 不应过早开始优化，应该关注瓶颈，而且应该从算法开始。
2. 有五个不同的参数控制编译。它们可以在本地声明也可以在全局声明。
3. 优秀的编译器能够优化尾递归，将一个尾递归的函数转换为一个循环。内联编译是另一种避免函数调用的方法。
4. 类型声明并不是必须的，但它们可以让一个程序更高效。类型声明对于处理数值和

数组的代码特别重要。
5. 少的构造可以让程序更快，特别是在使用着原始的垃圾回收器的实现中。解决方案是使用破坏性函数、预先分配空间块、以及在栈上分配。
6. 某些情况下，从预先分配的存储池中提取对象可能是有价值的。
7. Common Lisp 的某些部分是为了速度而设计的，另一些则为了灵活性。
8. 编程必定存在探索的过程。探索和优化应该被分开 —— 有时甚至需要使用不同的语言。

Chapter 13 练习 (Exercises)

1. 检验你的编译器是否支持 (observe)内敛声明。
2. 将下述函数重写为尾递归形式。它被编译后能快多少？

```
(defun foo (x)
  (if (zerop x)
      0
      (1+ (foo (1- x)))))
```

注意：你需要增加额外的参数。

3. 为下述程序增加声明。你能让它们变快多少？

```
(a) 在 5.7 节中的日期运算代码。
(b) 在 9.8 节中的光线跟踪器 (ray-tracer)。
```

4. 重写 3.15 节中的广度优先搜索的代码让它尽可能减少使用构造。
5. 使用存储池修改 4.7 节中的二叉搜索的代码。

脚注

[1] 较早的实现或许不提供 `declaim`； 需要使用 `proclaim` 并且引用这些参量 (quote the argument)。

[2] 为了让内联声明 (inline declaration) 有效，你同时必须设置编译参数，告诉它你想获得最快的代码。

[3] 有两种方法可以描述 Lisp 声明类型 (typing) 的方式：从类型信息被存放的位置或者从它被使用的时间。显示类型 (manifest typing) 的意思是类型信息与数据对象 (data objects) 绑定，而运行时类型(run-time typing) 的意思是类型信息在运行时被使用。实际上，两者是一回事儿。

第十四章：进阶议题

本章是选择性阅读的。本章描述了 Common Lisp 里一些更深奥的特性。Common Lisp 像是一个冰山：大部分的功能对于那些永远不需要他们的多数用户是看不见的。你或许永远不需要自己定义包 (Package)或读取宏 (read-macros)，但当你需要时，有些例子可以让你参考是很有用的。

14.1 类型标识符 (Type Specifiers)

类型在 Common Lisp 里不是对象。举例来说，没有对象对应到 `integer` 这个类型。我们像是从 `type-of` 函数里所获得的，以及作为传给像是 `typep` 函数的参数，不是一个类型，而是一个类型标识符 (type specifier)。

一个类型标识符是一个类型的名称。最简单的类型标识符是像是 `integer` 的符号。这些符号形成了 Common Lisp 里的类型层级。在层级的最顶端是类型 `t` —— 所有的对象皆为类型 `t` 。而类型层级不是一棵树。从 `nil` 至顶端有两条路，举例来说：一条从 `atom` ，另一条从 `list` 与 `sequence` 。

一个类型实际上只是一个对象集合。这意味著有多少类型就有多少个对象的集合：一个无穷大的数目。我们可以用原子的类型标识符 (atomic type specifiers)来表示某些集合：比如 `integer` 表示所有整数集合。但我们也可以建构一个复合类型标识符 (compound type specifiers)来参照到任何对象的集合。

举例来说，如果 `a` 与 `b` 是两个类型标识符，则 `(or a b)` 表示分别由 `a` 与 `b` 类型所表示的联集 (union)。也就是说，一个类型 `(or a b)` 的对象是类型 `a` 或 类型 `b` 。

如果 `circular?` 是一个对于 `cdr` 为环状的列表返回真的函数，则你可以使用适当的序列集合来表示： [1]

```
(or vector (and list (not (satisfies circular?))))
```

某些原子的类型标识符也可以出现在复合类型标识符。要表示介于 1 至 100 的整数（包含），我们可以用：

```
(integer 1 100)
```

这样的类型标识符用来表示一个有限的类型 (finite type)。

在一个复合类型标识符里，你可以通过在一个参数的位置使用 `*` 来留下某些未指定的信

息。所以

```
(simple-array fixnum (* *))
```

描述了指定给 `fixnum` 使用的二维简单数组 (simple array)集合，而

```
(simple-array fixnum *)
```

描述了指定给 `finxnum` 使用的简单数组集合 (前者的超类型 「supertype」)。尾随的星号可以省略，所以上个例子可以写为：

```
(simple-array fixnum)
```

若一个复合类型标识符没有传入参数，你可以使用一个原子。所以 `simple-array` 描述了所有简单数组的集合。

如果有某些复合类型标识符你想重复使用，你可以使用 `deftype` 定义一个缩写。这个宏与 `defmacro` 相似，但会展开成一个类型标识符，而不是一个表达式。通过表达

```
(deftype proseq ()
         '(or vector (and list (not (satisfies circular?)))))
```

我们定义了 `proseq` 作为一个新的原子类型标识符：

```
> (typep #(1 2) 'proseq)
T
```

如果你定义一个接受参数的类型标识符，参数会被视为 Lisp 形式（即没有被求值），与 `defmacro` 一样。所以

```
(deftype multiple-of (n)
  `(and integer (satisfies (lambda (x)
                             (zerop (mod x ,n))))))
```

(译注: 注意上面代码是使用反引号 `)

定义了 *(multiple-of n)* 当成所有 `n` 的倍数的标识符：

```
> (type 12 '(multiple-of 4))
T
```

类型标识符会被直译 (interpreted)，因此很慢，所以通常你最好定义一个函数来处理这类的测试。

14.2 二进制流 (Binary Streams)

第 7 章曾提及的流有二进制流 (binary streams)以及字符流 (character streams)。一个二进制流是一个整数的来源及/或终点，而不是字符。你通过指定一个整数的子类型来创建一个二进制流 —— 当你打开流时，通常是用 `unsigned-byte` —— 来作为 `:element-type` 的参数。

关于二进制流的 I/O 函数仅有两个， `read-byte` 以及 `write-byte` 。所以下面是如何定义复制一个文件的函数：

```
(defun copy-file (from to)
  (with-open-file (in from :direction :input
                           :element-type 'unsigned-byte)
    (with-open-file (out to :direction :output
                            :element-type 'unsigned-byte)
      (do ((i (read-byte in nil -1)
              (read-byte in nil -1)))
          ((minusp i))
        (declare (fixnum i))
        (write-byte i out)))))
```

仅通过指定 `unsigned-byte` 给 `:element-type` ，你让操作系统选择一个字节 (byte)的长度。举例来说，如果你明确地想要读写 7 比特的整数，你可以使用：

```
(unsigned-byte 7)
```

来传给 `:element-type` 。

14.3 读取宏 (Read-Macros)

7.5 节介绍过宏字符 (macro character)的概念，一个对于 `read` 有特别意义的字符。每一个这样的字符，都有一个相关联的函数，这函数告诉 `read` 当遇到这个字符时该怎么处理。你可以变更某个已存在宏字符所相关联的函数，或是自己定义新的宏字符。

函数 `set-macro-character` 提供了一种方式来定义读取宏 (read-macros)。它接受一个字符及一个函数，因此当 `read` 碰到该字符时，它返回调用传入函数后的结果。

Lisp 中最古老的读取宏之一是 `'` ，即 `quote` 。我们可以定义成：

```
(set-macro-character #\'
        #'(lambda (stream char)
                (list (quote quote) (read stream t nil t))))
```

当 read 在一个普通的语境下遇到 ' 时，它会返回在当前流和字符上调用这个函数的结果。(这个函数忽略了第二个参数，第二个参数永远是引用字符。)所以当 read 看到 'a 时，会返回 (quote a) 。

译注: read 函数接受的参数 (read &optional stream eof-error eof-value recursive)

现在我们明白了 read 最后一个参数的用途。它表示无论 read 调用是否在另一个 read 里。传给 read 的参数在几乎所有的读取宏里皆相同：传入参数有流 (stream)；接著是第二个参数， t ，说明了 read 若读入的东西是 end-of-file 时，应不应该报错；第三个参数说明了不报错时要返回什么，因此在这里也就不重要了；而第四个参数 t 说明了这个 read 调用是递归的。

(译注：困惑的话可以看看 read 的定义 [https://gist.github.com/3467235])

你可以（通过使用 make-dispatch-macro-character ）来定义你自己的派发宏字符（dispatching macro character），但由于 # 已经是一个宏字符，所以你也可以直接使用。六个 # 打头的组合特别保留给你使用： #! 、 #? 、 ##[、 ##] 、 #{ 、 #} 。

你可以通过调用 set-dispatch-macro-character 定义新的派发宏字符组合，与 set-macro-character 类似，除了它接受两个字符参数外。下面的代码定义了 #? 作为返回一个整数列表的读取宏。

```
(set-dispatch-macro-character #\# #\?
  #'(lambda (stream char1 char2)
      (list 'quote
            (let ((lst nil))
              (dotimes (i (+ (read stream t nil t) 1))
                (push i lst))
              (nreverse lst)))))
```

现在 #?n 会被读取成一个含有整数 0 至 n 的列表。举例来说：

```
> #?7
(1 2 3 4 5 6 7)
```

除了简单的宏字符，最常定义的宏字符是列表分隔符 (list delimiters)。另一个保留给用户的字符组是 #{ 。以下我们定义了一种更复杂的左括号：

```
(set-macro-character #\} (get-macro-character #\)))

(set-dispatch-macro-character #\# #\{
  #'(lambda (stream char1 char2)
      (let ((accum nil)
```

```
            (pair (read-delimited-list #\} stream t)))
        (do ((i (car pair) (+ i 1)))
            ((> i (cadr pair))
             (list 'quote (nreverse accum)))
          (push i accum)))))
```

这定义了一个这样形式 `#{x y}` 的表达式，使得这样的表达式被读取为所有介于 `x` 与 `y` 之间的整数列表，包含 `x` 与 `y`：

```
> #{2 7}
(2 3 4 4 5 6 7)
```

函数 `read-delimited-list` 正是为了这样的读取宏而生的。它的第一个参数是被视为列表结束的字符。为了使 `}` 被识别为分隔符，必须先给它这个角色，所以程序在开始的地方调用了 `set-macro-character`。

如果你想要在定义一个读取宏的文件里使用该读取宏，则读取宏的定义应要包在一个 `eval-when` 表达式里，来确保它在编译期会被求值。不然它的定义会被编译，但不会被求值，直到编译文件被载入时才会被求值。

14.4 包 (Packages)

一个包是一个将名字映对到符号的 Lisp 对象。当前的包总是存在全局变量 `*package*` 里。当 Common Lisp 启动时，当前的包会是 `*common-lisp-user*`，通常称为用户包 (user package)。函数 `package-name` 返回包的名字，而 `find-package` 返回一个给定名称的包:

```
> (package-name *package*)
"COMMON-LISP-USER"
> (find-package "COMMON-LISP-USER")
#<Package "COMMON-LISP-USER" 4CD15E>
```

通常一个符号在读入时就被 interned 至当前的包里面了。函数 `symbol-package` 接受一个符号并返回该符号被 interned 的包。

```
(symbol-package 'sym)
#<Package "COMMON-LISP-USER" 4CD15E>
```

有趣的是，这个表达式返回它该返回的值，因为表达式在可以被求值前必须先被读入，而读取这个表达式导致 `sym` 被 interned。为了之后的用途，让我们给 `sym` 一个值:

```
> (setf sym 99)
99
```

现在我们可以创建及切换至一个新的包：

```
> (setf *package* (make-package 'mine
                                :use '(common-lisp)))
#<Package "MINE" 63390E>
```

现在应该会听到诡异的背景音乐，因为我们来到一个不一样的世界了： 在这里 sym 不再是本来的 sym 了。

```
MINE> sym
Error: SYM has no value
```

为什么会这样？因为上面我们设为 99 的 sym 与 mine 里的 sym 是两个不同的符号。 [2] 要在用户包之外参照到原来的 sym ，我们必须把包的名字加上两个冒号作为前缀：

```
MINE> common-lisp-user::sym
99
```

所以有着相同打印名称的不同符号能够在不同的包内共存。可以有一个 sym 在 common-lisp-user 包，而另一个 sym 在 mine 包，而他们会是不一样的符号。这就是包存在的意义。如果你在分开的包内写你的程序，你大可放心选择函数与变量的名字，而不用担心某人使用了同样的名字。即便是他们使用了同样的名字，也不会是相同的符号。

包也提供了信息隐藏的手段。程序应通过函数与变量的名字来参照它们。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参照的对象。

通常使用两个冒号作为包的前缀也是很差的风格。这么做你就违反了包本应提供的模块性。如果你不得不使用一个双冒号来参照到一个符号，这是因为某人根本不想让你用。

通常我们应该只参照被输出 (*exported*)的符号。如果我们回到用户包里，并输出一个被 interned 的符号，

```
MINE> (in-package common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (export 'bar)
T
> (setf bar 5)
5
```

我们使这个符号对于其它的包是可视的。现在当我们回到 mine ，我们可以仅使用单冒号来参照到 bar ，因为他是一个公开可用的名字：

```
> (in-package mine)
```

```
#<Package "MINE" 63390E>
MINE> common-lisp-user:bar
5
```

通过把 bar 输入 (import)至 mine 包，我们就能进一步让 mine 和 user 包可以共享 bar 这个符号：

```
MINE> (import 'common-lisp-user:bar)
T
MINE> bar
5
```

在输入 bar 之后，我们根本不需要用任何包的限定符 (package qualifier)，就能参照它了。这两个包现在共享了同样的符号；不可能会有一个独立的 mine:bar 了。

要是已经有一个了怎么办？在这种情况下， import 调用会产生一个错误，如下面我们试著输入 sym 时便知：

```
MINE> (import 'common-lisp-user::sym)
Error: SYM is already present in MINE.
```

在此之前，当我们试着在 mine 包里对 sym 进行了一次不成功的求值，我们使 sym 被 interned 至 mine 包里。而因为它没有值，所以产生了一个错误，但输入符号名的后果就是使这个符号被 intern 进这个包。所以现在当我们试著输入 sym 至 mine 包里，已经有一个相同名称的符号了。

另一个方法来获得别的包内符号的存取权是使用(use)它：

```
MINE> (use-package 'common-lisp-user)
T
```

现在所有由用户包 (译注: common-lisp-user 包）所输出的符号，可以不需要使用任何限定符在 mine 包里使用。(如果 sym 已经被用户包输出了，这个调用也会产生一个错误。)

含有自带操作符及变量名字的包叫做 common-lisp 。由于我们将这个包的名字在创建 mine 包时作为 make-package 的 :use 参数，所有的 Common Lisp 自带的名字在 mine 里都是可视的:

```
MINE> #'cons
#<Compiled-Function CONS 462A3E>
```

在编译后的代码中， 通常不会像这样在顶层进行包的操作。更常见的是包的调用会包含

在源文件里。通常，只要把 in-package 和 defpackage 放在源文件的开头就可以了，正如 137 页所示。

这种由包所提供的模块性实际上有点奇怪。我们不是对象的模块 (modules)，而是名字的模块。

每一个使用了 common-lisp 的包，都可以存取 cons ，因为 common-lisp 包里有一个叫这个名字的函数。但这会导致一个名字为 cons 的变量也会在每个使用了 common-lisp 包里是可视的。如果包使你困惑，这就是主要的原因；因为包不是基于对象而是基于名字。

14.5 Loop 宏 (The Loop Facility)

loop 宏最初是设计来帮助无经验的 Lisp 用户来写出迭代的代码。与其撰写 Lisp 代码，你用一种更接近英语的形式来表达你的程序，然后这个形式被翻译成 Lisp。不幸的是，loop 比原先设计者预期的更接近英语：你可以在简单的情况下使用它，而不需了解它是如何工作的，但想在抽象层面上理解它几乎是不可能的。

如果你是曾经计划某天要理解 loop 怎么工作的许多 Lisp 程序员之一，有一些好消息与坏消息。好消息是你并不孤单：几乎没有人理解它。坏消息是你永远不会理解它，因为 ANSI 标准实际上并没有给出它行为的正式规范。

这个宏唯一的实际定义是它的实现方式，而唯一可以理解它（如果有人可以理解的话）的方法是通过实例。ANSI 标准讨论 loop 的章节大部分由例子组成，而我们将会使用同样的方式来介绍相关的基础概念。

第一个关于 loop 宏我们要注意到的是语法 (*syntax*)。一个 loop 表达式不是包含子表达式而是子句 (*clauses*)。這些子句不是由括号分隔出来；而是每种都有一个不同的语法。在这个方面上， loop 与传统的 Algol-like 语言相似。但其它 loop 独特的特性，使得它与 Algol 不同，也就是在 loop 宏里调换子句的顺序与会发生的事情没有太大的关联。

一个 loop 表达式的求值分为三个阶段，而一个给定的子句可以替多于一个的阶段贡献代码。这些阶段如下：

1. 序幕 (*Prologue*)。 被求值一次来做为迭代过程的序幕。包括了将变量设至它们的初始值。
2. 主体 (*Body*) 每一次迭代时都会被求值。
3. 闭幕 (*Epilogue*) 当迭代结束时被求值。决定了 loop 表达式的返回值（可能返回多个值）。

我们会看几个 loop 子句的例子，并考虑何种代码会贡献至何个阶段。

举例来说，最简单的 loop 表达式，我们可能会看到像是下列的代码：

```
> (loop for x from 0 to 9
        do (princ x))
0123456789
NIL
```

这个 loop 表达式印出从 0 至 9 的整数，并返回 nil 。第一个子句，

```
for x from 0 to 9
```

贡献代码至前两个阶段，导致 x 在序幕中被设为 0 ，在主体开头与 9 来做比较，在主体结尾被递增。第二个子句，

```
do (princ x)
```

贡献代码给主体。

一个更通用的 for 子句说明了起始与更新的形式 (initial and update form)。停止迭代可以被像是 while 或 until 子句来控制。

```
> (loop for x = 8 then (/ x 2)
        until (< x 1)
        do (princ x))
8421
NIL
```

你可以使用 and 来创建复合的 for 子句，同时初始及更新两个变量：

```
> (loop for x from 1 to 4
        and y from 1 to 4
        do (princ (list x y)))
(1 1)(2 2)(3 3)(4 4)
NIL
```

要不然有多重 for 子句时，变量会被循序更新。

另一件在迭代代码通常会做的事是累积某种值。举例来说：

```
> (loop for x in '(1 2 3 4)
        collect (1+ x))
(2 3 4 5)
```

在 for 子句使用 in 而不是 from ，导致变量被设为一个列表的后续元素，而不是连续的整数。

在这个情况里， collect 子句贡献代码至三个阶段。在序幕，一個匿名累加器(anonymous accumulator)設為 nil ；在主体裡， (1+ x) 被累加至這個累加器，而在闭幕时返回累加器的值。

这是返回一个特定值的第一个例子。有用来明确指定返回值的子句，但没有这些子句时，一个 collect 子句决定了返回值。所以我们在这里所做的其实是重复了 mapcar 。

loop 最常见的用途大概是蒐集调用一个函数数次的结果：

```
> (loop for x from 1 to 5
        collect (random 10))
(3 8 6 5 0)
```

这里我们获得了一个含五个随机数的列表。这跟我们定义过的 map-int 情况类似 (105 页「译注: 6.4 小节。」)。如果我们有了 loop ，为什么还需要 map-int ？另一个人也可以说，如果我们有了 map-int ，为什么还需要 loop ？

一个 collect 子句也可以累积值到一个有名字的变量上。下面的函数接受一个数字的列表并返回偶数与奇数列表：

```
(defun even/odd (ns)
  (loop for n in ns
        if (evenp n)
           collect n into evens
           else collect n into odds
        finally (return (values evens odds))))
```

一个 finally 子句贡献代码至闭幕。在这个情况它指定了返回值。

一个 sum 子句和一个 collect 子句类似，但 sum 子句累积一个数字，而不是一个列表。要获得 1 至 n 的和，我们可以写：

```
(defun sum (n)
  (loop for x from 1 to n
        sum x))
```

loop 更进一步的细节在附录 D 讨论，从 325 页开始。举个例子，图 14.1 包含了先前章节的两个迭代函数，而图 14.2 演示了将同样的函数翻译成 loop 。

```
(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)
              (setf wins obj
                    max  score))))
        (values wins max))))

(defun num-year (n)
  (if (< n 0)
      (do* ((y (- yzero 1) (- y 1))
            (d (- (year-days y)) (- d (year-days y))))
           ((<= d n) (values y (- n d))))
      (do* ((y yzero (+ y 1))
            (prev 0 d)
            (d (year-days y) (+ d (year-days y))))
           ((> d n) (values y (- n prev))))))
```

图 **14.1** 不使用 **loop** 的迭代函数

```
(defun most (fn lst)
  (if (null lst)
      (values nil nil)
      (loop with wins = (car lst)
            with max = (funcall fn wins)
            for obj in (cdr lst)
            for score = (funcall fn obj)
            when (> score max)
              (do (setf wins obj
                        max score)
            finally (return (values wins max))))))

(defun num-year (n)
  (if (< n 0)
      (loop for y downfrom (- yzero 1)
            until (<= d n)
            sum (- (year-days y)) into d
            finally (return (values (+ y 1) (- n d))))
      (loop with prev = 0
            for y from yzero
            until (> d n)
            do (setf prev d)
            sum (year-days y) into d
            finally (return (values (- y 1)
                                    (- n prev))))))
```

图 **14.2** 使用 **loop** 的迭代函数

一个 `loop` 的子句可以参照到由另一个子句所设置的变量。举例来说，在 `even/odd` 的定义里面， `finally` 子句参照到由两个 `collect` 子句所创建的变量。这些变量之间的关系，是 `loop` 定义最含糊不清的地方。考虑下列两个表达式：

```
(loop for y = 0 then z
```

```
      for x from 1 to 5
      sum 1 into z
      finally (return y z))

(loop for x from 1 to 5
      for y = 0 then z
      sum 1 into z
      finally (return y z))
```

它们看起来够简单 —— 每一个有四个子句。但它们返回同样的值吗？它们返回的值多少？你若试着在标准中想找答案将徒劳无功。每一个 `loop` 子句本身是够简单的。但它们组合起来的方式是极为复杂的 —— 而最终，甚至标准里也没有明确定义。

由于这类原因，使用 `loop` 是不推荐的。推荐 `loop` 的理由，你最多可以说，在像是图 14.2 这般经典的例子中， `loop` 让代码看起来更容易理解。

14.6 状况 (Conditions)

在 Common Lisp 里，状况 (condition)包括了错误以及其它可能在执行期发生的情况。当一个状况被捕捉时 (signalled)，相应的处理程序 (handler)会被调用。处理错误状况的缺省处理程序通常会调用一个中断循环 (break-loop)。但 Common Lisp 提供了多样的操作符来捕捉及处理错误。要覆写缺省的处理程序，甚至是自己写一个新的处理程序也是有可能的。

多数的程序员不会直接处理状况。然而有许多更抽象的操作符使用了状况，而要了解这些操作符，知道背后的原理是很有用的。

Common lisp 有数个操作符用来捕捉错误。最基本的是 `error` 。一个调用它的方法是给入你会给 `format` 的相同参数：

```
> (error "Your report uses ~A as a verb." 'status)
Error: Your report uses STATUS as a verb
                        Options: :abort, :backtrace
>>
```

如上所示，除非这样的状况被处理好了，不然执行就会被打断。

用来捕捉错误的更抽象操作符包括了 `ecase` 、 `check-type` 以及 `assert` 。前者与 `case` 相似，要是没有键值匹配时会捕捉一个错误：

```
> (ecase 1 (2 3) (4 5))
Error: No applicable clause
                        Options: :abort, :backtrace
>>
```

普通的 case 在没有键值匹配时会返回 nil ，但由于利用这个返回值是很差的编码风格，你或许会在当你没有 otherwise 子句时使用 ecase 。

check-type 宏接受一个位置，一个类型名以及一个选择性字符串，并在该位置的值不是预期的类型时，捕捉一个可修正的错误 (correctable error)。一个可修正错误的处理程序会给我们一个机会来提供一个新的值：

```
> (let ((x '(a b c)))
                (check-type (car x) integer "an integer")
                x)
Error: The value of (CAR X), A, should be an integer.
Options: :abort, :backtrace, :continue
>> :continue
New value of (CAR X)? 99
(99 B C)
>
```

在这个例子里， (car x) 被设为我们提供的新值，并重新执行，返回了要是 (car x) 本来就包含我们所提供的值所会返回的结果。

这个宏是用更通用的 assert 所定义的， assert 接受一个测试表达式以及一个有着一个或多个位置的列表，伴随着你可能传给 error 的参数：

```
> (let ((sandwich '(ham on rye)))
    (assert (eql (car sandwich) 'chicken)
            ((car sandwich))
            "I wanted a ~A sandwich." 'chicken)
    sandwich)
Error: I wanted a CHICKEN sandwich.
Options: :abort, :backtrace, :continue
>> :continue
New value of (CAR SANDWICH)? 'chicken
(CHICKEN ON RYE)
```

要建立新的处理程序也是可能的，但大多数程序员只会间接的利用这个可能性，通过使用像是 ignore-errors 的宏。如果它的参数没产生错误时像在 progn 里求值一样，但要是在求值过程中，不管什么参数报错，执行是不会被打断的。取而代之的是， ignore-errors 表达式会直接返回两个值： nil 以及捕捉到的状况。

举例来说，如果在某个时候，你想要用户能够输入一个表达式，但你不想要在输入是语法上不合时中断执行，你可以这样写：

```
(defun user-input (prompt)
  (format t prompt)
  (let ((str (read-line)))
    (or (ignore-errors (read-from-string str))
```

```
        nil)))
```

若输入包含语法错误时，这个函数仅返回 `nil`：

```
> (user-input "Please type an expression")
Please type an expression> #%@#+!!
NIL
```

脚注

[1] 虽然标准没有提到这件事，你可以假定 `and` 以及 `or` 类型标示符仅考虑它们所要考虑的参数，与 `or` 及 `and` 宏类似。

[2] 某些 Common Lisp 实现，当我们不在用户包下时，会在顶层提示符前打印包的名字。

第十五章：示例：推论

接下来三章提供了大量的 Lisp 程序例子。选择这些例子来说明那些较长的程序所采取的形式，和 Lisp 所擅长解决的问题类型。

在这一章中我们将要写一个基于一组 `if-then` 规则的推论程序。这是一个经典的例子——不仅在于其经常出现在教科书上，还因为它反映了 Lisp 作为一个“符号计算”语言的本意。这个例子散发着很多早期 Lisp 程序的气息。

15.1 目标 (The Aim)

在这个程序中，我们将用一种熟悉的形式来表示信息：包含单个判断式，以及跟在之后的零个或多个参数所组成的列表。要表示 Donald 是 Nancy 的家长，我们可以这样写：

```
(parent donald nancy)
```

事实上，我们的程序是要表示一些从已有的事实作出推断的规则。我们可以这样来表示规则：

```
(<- head body)
```

其中，`head` 是 那么...部分 (then-part)，`body` 是 如果...部分 (if-part)。在 `head` 和 `body` 中我们使用以问号为前缀的符号来表示变量。所以下面这个规则：

```
(<- (child ?x ?y) (parent ?y ?x))
```

表示：如果 y 是 x 的家长，那么 x 是 y 的孩子；更恰当地说，我们可以通过证明 `(parent y x)` 来证明 `(child x y)` 的所表示的事实。

可以把规则中的 *body* 部分(if-part) 写成一个复杂的表达式，其中包含 `and` , `or` 和 `not` 等逻辑操作。所以当我们想要表达 “如果 x 是 y 的家长，并且 x 是男性，那么 x 是 y 的父亲” 这样的规则，我们可以写：

```
(<- (father ?x ?y) (and (parent ?x ?y) (male ?x)))
```

一些规则可能依赖另一些规则所产生的事实。比如，我们写的第一个规则是为了证明 `(child x y)` 的事实。如果我们定义如下规则：

```
(<- (daughter ?x ?y) (and (child ?x ?y) (female ?x)))
```

然后使用它来证明 `(daughter x y)` 可能导致程序使用第一个规则去证明 `(child x y)`。

表达式的证明可以回溯任意数量的规则，只要它最终结束于给出的已知事实。这个过程有时候被称为反向链接 (backward-chaining)。之所以说 反向 (backward) 是因为这一类推论先考虑 *head* 部分，这是为了在继续证明 *body* 部分之前检查规则是否有效。链接 (chaining) 来源于规则之间的依赖关系，从我们想要证明的内容到我们的已知条件组成一个链接 (尽管事实上它更像一棵树)。 λ [http://acl.readthedocs.org/en/latest/zhCN/notes-cn.html#notes-248]

15.2 匹配 (Matching)

我们需要有一个函数来做模式匹配以完成我们的反向链接 (back-chaining) 程序，这个函数能够比较两个包含变量的列表，它会检查在给变量赋值后是否可以使两个列表相等。举例，如果 `?x` 和 `?y` 是变量，那么下面两个列表：

```
(p ?x ?y c ?x)
(p  a  b c  a)
```

当 `?x = a` 且 `?y = b` 时匹配，而下面两个列表：

```
(p ?x b ?y a)
(p ?y b  c a)
```

当 `?x = ?y = c` 时匹配。

我们有一个 `match` 函数，它接受两棵树，如果这两棵树能匹配，则返回一个关联列表（assoc-list）来显示他们是如何匹配的：

```
(defun match (x y &optional binds)
  (cond
   ((eql x y) (values binds t))
   ((assoc x binds) (match (binding x binds) y binds))
   ((assoc y binds) (match x (binding y binds) binds))
   ((var? x) (values (cons (cons x y) binds) t))
   ((var? y) (values (cons (cons y x) binds) t))
   (t
    (when (and (consp x) (consp y))
      (multiple-value-bind (b2 yes)
                           (match (car x) (car y) binds)
        (and yes (match (cdr x) (cdr y) b2)))))))

(defun var? (x)
  (and (symbolp x)
       (eql (char (symbol-name x) 0) #\?)))
```

```
(defun binding (x binds)
  (let ((b (assoc x binds)))
    (if b
        (or (binding (cdr b) binds)
            (cdr b)))))
```

图 **15.1:** 匹配函数。

```
> (match '(p a b c a) '(p ?x ?y c ?x))
((?Y . B) (?X . A))
T
> (match '(p ?x b ?y a) '(p ?y b c a))
((?Y . C) (?X . ?Y))
T
> (match '(a b c) '(a a a))
NIL
```

当 match 函数逐个元素地比较它的参数时候，它把 binds 参数中的值分配给变量，这被称为绑定 (bindings)。如果成功匹配， match 函数返回生成的绑定；否则，返回 nil 。当然并不是所有成功的匹配都会产生绑定，我们的 match 函数就像 gethash 函数那样返回第二个值来表明匹配成功：

```
> (match '(p ?x) '(p ?x))
NIL
T
```

如果 match 函数像上面那样返回 nil 和 t ，表明这是一个没有产生绑定的成功匹配。下面用中文来描述 match 算法是如何工作的：

1. 如果 x 和 y 在 eql 上相等那么它们匹配；否则，
2. 如果 x 是一个已绑定的变量，并且绑定匹配 y ，那么它们匹配；否则，
3. 如果 y 是一个已绑定的变量，并且绑定匹配 x ，那么它们匹配；否则，
4. 如果 x 是一个未绑定的变量，那么它们匹配，并且为 x 建立一个绑定；否则，
5. 如果 y 是一个未绑定的变量，那么它们匹配，并且为 y 建立一个绑定；否则，
6. 如果 x 和 y 都是 cons ，并且它们的 car 匹配，由此产生的绑定又让 cdr 匹配，那么它们匹配。

下面是一个例子，按顺序来说明以上六种情况：

```
> (match '(p ?v  b ?x  d (?z ?z))
         '(p  a ?w  c ?y ( e  e))
         '((?v . a) (?w . b)))
((?Z . E) (?Y . D) (?X . C) (?V . A) (?W . B))
T
```

match 函数通过调用 binding 函数在一个绑定列表中寻找变量（如果有的话）所关联的值。这个函数必须是递归的，因为有这样的情况 “匹配建立一个绑定列表，而列表中变量只是间接关联到它的值： ?x 可能被绑定到一个包含 (?x . ?y) 和 (?y . a) 的列表”：

```
> (match '(?x a) '(?y ?y))
((?Y . A) (?X . ?Y))
T
```

先匹配 ?x 和 ?y ，然后匹配 ?y 和 a ，我们间接确定 ?x 是 a 。

15.3 回答查询 (Answering Queries)

在介绍了绑定的概念之后，我们可以更准确的说一下我们的程序将要做什么：它得到一个可能包含变量的表达式，根据我们给定的事实和规则返回使它正确的所有绑定。比如，我们只有下面这个事实：

```
(parent donald nancy)
```

然后我们想让程序证明：

```
(parent ?x ?y)
```

它会返回像下面这样的表达：

```
(((?x . donald) (?y . nancy)))
```

它告诉我们只有一个可以让这个表达式为真的方法： ?x 是 donald 并且 ?y 是 nancy 。

在通往目标的路上，我们已经有了一个的重要部分：一个匹配函数。 下面是用来定义规则的一段代码：

```
(defvar *rules* (make-hash-table))

(defmacro <- (con &optional ant)
  `(length (push (cons (cdr ',con) ',ant)
                 (gethash (car ',con) *rules*))))
```

图 **15.2** 定义规则

规则将被包含于一个叫做 *rules* 的哈希表，通过头部 (head) 的判断式构建这个哈系表。这样做加强了我们无法使用判断式中的变量的限制。虽然我们可以通过把所有这样的规则放在分离的列表中来消除限制，但是如果这样做，当我们需要证明某件事的时侯

不得不和每一个列表进行匹配。

我们将要使用同一个宏 <- 去定义事实 (facts)和规则 (rules)。一个事实将被表示成一个没有 *body* 部分的规则。这和我们对规则的定义保持一致。一个规则告诉我们你可以通过证明 *body* 部分来证明 *head* 部分，所以没有 *body* 部分的规则意味着你不需要通过证明任何东西来证明 *head* 部分。这里有两个对应的例子：

```
> (<- (parent donald nancy))
1
> (<- (child ?x ?y) (parent ?y ?x))
1
```

调用 <- 返回的是给定判断式下存储的规则数量；用 `length` 函数来包装 `push` 能使我们免于看到顶层中的一大堆返回值。

下面是我们的推论程序所需的大多数代码：

```
(defun prove (expr &optional binds)
  (case (car expr)
    (and (prove-and (reverse (cdr expr)) binds))
    (or  (prove-or (cdr expr) binds))
    (not (prove-not (cadr expr) binds))
    (t   (prove-simple (car expr) (cdr expr) binds))))

(defun prove-simple (pred args binds)
  (mapcan #'(lambda (r)
              (multiple-value-bind (b2 yes)
                                   (match args (car r)
                                          binds)
                (when yes
                  (if (cdr r)
                      (prove (cdr r) b2)
                      (list b2)))))
          (mapcar #'change-vars
                  (gethash pred *rules*))))

(defun change-vars (r)
  (sublis (mapcar #'(lambda (v) (cons v (gensym "?")))
                  (vars-in r))
          r))

(defun vars-in (expr)
  (if (atom expr)
      (if (var? expr) (list expr))
    (union (vars-in (car expr))
           (vars-in (cdr expr)))))
```

图 **15.3:** 推论。

上面代码中的 prove 函数是推论进行的枢纽。它接受一个表达式和一个可选的绑定列表作为参数。如果表达式不包含逻辑操作，它调用 prove-simple 函数，前面所说的链接(chaining)正是在这个函数里产生的。这个函数查看所有拥有正确判断式的规则，并尝试对每一个规则的 *head* 部分和它想要证明的事实做匹配。对于每一个匹配的 *head* ，使用匹配所产生的新的绑定在 *body* 上调用 prove 。对 prove 的调用所产生的绑定列表被 mapcan 收集并返回：

```
> (prove-simple 'parent '(donald nancy) nil)
(NIL)
> (prove-simple 'child '(?x ?y) nil)
(((#:?6 . NANCY) (#:?5 . DONALD) (?Y . #:?5) (?X . #:?6)))
```

以上两个返回值指出有一种方法可以证明我们的问题。（一个失败的证明将返回 nil。）第一个例子产生了一组空的绑定，第二个例子产生了这样的绑定： ?x 和 ?y 被（间接）绑定到 nancy 和 donald 。

顺便说一句，这是一个很好的例子来实践 2.13 节提出的观点。因为我们用函数式的风格来写这个程序，所以可以交互式地测试每一个函数。

第二个例子返回的值里那些 *gensyms* 是怎么回事？如果我们打算使用含有变量的规则，我们需要避免两个规则恰好包含相同的变量。如果我们定义如下两条规则：

```
(<- (child ?x ?y) (parent ?y ?x))

(<- (daughter ?y ?x) (and (child ?y ?x) (female ?y)))
```

第一条规则要表达的意思是：对于任何的 x 和 y ， 如果 y 是 x 的家长，则 x 是 y 的孩子。第二条则是：对于任何的 x 和 y ， 如果 y 是 x 的孩子并且 y 是女性，则 y 是 x 的女儿。在每一条规则内部，变量之间的关系是显著的，但是两条规则使用了相同的变量并非我们刻意为之。

如果我们使用上面所写的规则，它们将不会按预期的方式工作。如果我们尝试证明“ a 是 b 的女儿”，匹配到第二条规则的 *head* 部分时会将 a 绑定到 ?y ，将 b 绑定到 ?x。我们无法用这样的绑定匹配第一条规则的 *head* 部分：

```
> (match '(child ?y ?x)
         '(child ?x ?y)
         '((?y . a) (?x . b)))
NIL
```

为了保证一条规则中的变量只表示规则中各参数之间的关系，我们用 *gensyms* 来代替规则中的所有变量。这就是 change-vars 函数的目的。一个 *gensym* 不可能在另一个规则中作为变量出现。但是因为规则可以是递归的，我们必须防止出现一个规则和自身冲突

的可能性，所以在定义和使用一个规则时都要调用 chabge-vars 函数。

现在只剩下定义用以证明复杂表达式的函数了。下面就是需要的函数：

```
(defun prove-and (clauses binds)
  (if (null clauses)
      (list binds)
      (mapcan #'(lambda (b)
                  (prove (car clauses) b))
              (prove-and (cdr clauses) binds))))

(defun prove-or (clauses binds)
  (mapcan #'(lambda (c) (prove c binds))
          clauses))

(defun prove-not (clause binds)
  (unless (prove clause binds)
    (list binds)))
```

图 **15.4** 逻辑操作符 **(Logical operators)**

操作一个 or 或者 not 表达式是非常简单的。操作 or 时，我们提取在 or 之间的每一个表达式返回的绑定。操作 not 时，当且仅当在 not 里的表达式产生 none 时，返回当前的绑定。

prove-and 函数稍微复杂一点。它像一个过滤器，它用之后的表达式所建立的每一个绑定来证明第一个表达式。这将导致 and 里的表达式以相反的顺序被求值。除非调用 prove 中的 prove-and 函数则会先逆转它们。

现在我们有了一个可以工作的程序，但它不是很友好。必须要解析 prove-and 返回的绑定列表是令人厌烦的，它们会变得更长随着规则变得更加复杂。下面有一个宏来帮助我们更愉快地使用这个程序：

```
(defmacro with-answer (query &body body)
  (let ((binds (gensym)))
    `(dolist (,binds (prove ',query))
       (let ,(mapcar #'(lambda (v)
                         `(,v (binding ',v ,binds)))
                     (vars-in query))
         ,@body))))
```

图 **15.5** 介面宏 **(Interface macro)**

它接受一个 query （不被求值）和若干表达式构成的 body 作为参数，把 query 所生成的每一组绑定的值赋给 query 中对应的模式变量，并计算 body 。

```
> (with-answer (parent ?x ?y)
    (format t "~A is the parent of ~A.~%" ?x ?y))
DONALD is the parent of NANCY.
NIL
```

这个宏帮我们做了解析绑定的工作，同时为我们在程序中使用 `prove` 提供了一个便捷的方法。下面是这个宏展开的情况：

```
(with-answer (p ?x ?y)
  (f ?x ?y))

;;将被展开成下面的代码

(dolist (#:g1 (prove '(p ?x ?y)))
  (let ((?x (binding '?x #:g1))
        (?y (binding '?y #:g1)))
    (f ?x ?y)))
```

图 **15.6: with-answer** 调用的展开式

下面是使用它的一个例子：

```
(<- (parent donald nancy))
(<- (parent donald debbie))
(<- (male donald))
(<- (father ?x ?y) (and (parent ?x ?y) (male ?x)))
(<- (= ?x ?y))
(<- (sibling ?x ?y) (and (parent ?z ?x)
                         (parent ?z ?y)
                         (not (= ?x ?y)))))

;;我们可以像下面这样做出推论

> (with-answer (father ?x ?y)
    (format t "~A is the father of ~A.~%" ?x ?y))
DONALD is the father of DEBBIE.
DONALD is the father of NANCY.
NIL
> (with-answer (sibling ?x ?y))
    (format t "~A is the sibling of ~A.~%" ?x ?y))
DEBBLE is the sibling of NANCY.
NANCY is the  sibling of DEBBIE.
NIL
```

图 **15.7:** 使用中的程序

15.4 分析 (Analysis)

看上去，我们在这一章中写的代码，是用简单自然的方式去实现这样一个程序。事实上，它的效率非常差。我们在这里是其实是做了一个解释器。我们能够把这个程序做得像一个编译器。

这里做一个简单的描述。基本的思想是把整个程序打包到两个宏 `<-` 和 `with-answer` ，把已有程序中在运行期做的多数工作搬到宏展开期（在 10.7 节的 `avg` 可以看到这种构思的雏形) 用函数取代列表来表示规则，我们不在运行时用 `prove` 和 `prove-and` 这样的函数来解释表达式，而是用相应的函数把表达式转化成代码。当一个规则被定义的时候就有表达式可用。为什么要等到使用的时候才去分析它呢？这同样适用于和 `<-` 调用了相同的函数来进行宏展开的 `with-answer` 。

听上去好像比我们已经写的这个程序复杂很多，但其实可能只是长了两三倍。想要学习这种技术的读者可以看 *On Lisp* 或者 *Paradigms of Artificial Intelligence Programming* ，这两本书有一些使用这种风格写的示例程序。

第十六章：示例：生成 HTML

本章的目标是完成一个简单的 HTML 生成器 —— 这个程序可以自动生成一系列包含超文本链接的网页。除了介绍特定 Lisp 技术之外，本章还是一个典型的自底向上编程（bottom-up programming）的例子。 我们以一些通用 HTML 实用函数作为开始，继而将这些例程看作是一门编程语言，从而更好地编写这个生成器。

16.1 超文本标记语言 (HTML)

HTML （HyperText Markup Language，超文本标记语言）用于构建网页，是一种简单、易学的语言。本节就对这种语言作概括性介绍。

当你使用网页浏览器阅览网页时，浏览器从远程服务器获取 HTML 文件，并将它们显示在你的屏幕上。每个 HTML 文件都包含任意多个标签（tag），这些标签相当于发送给浏览器的指令。

```
<center>
<h2>Your Fortune</h2>
</center>
<br><br>
Welcome to the home page of the Fortune Cookie
Institute.  FCI is a non-profit institution
dedicated to the development of more realistic
fortunes.  Here are some examples of fortunes
that fall within our guidelines:
<ol>
<li>Your nostril hairs will grow longer.
<li>You will never learn how to dress properly.
<li>Your car will be stolen.
<li>You will gain weight.
</ol>
Click <a href="research.html">here</a> to learn
more about our ongoing research projects.
```

图 **16.1** 一个 **HTML** 文件

图 16.1 给出了一个简单的 HTML 文件，图 16.2 展示了这个 HTML 文件在浏览器里显示时大概是什么样子。

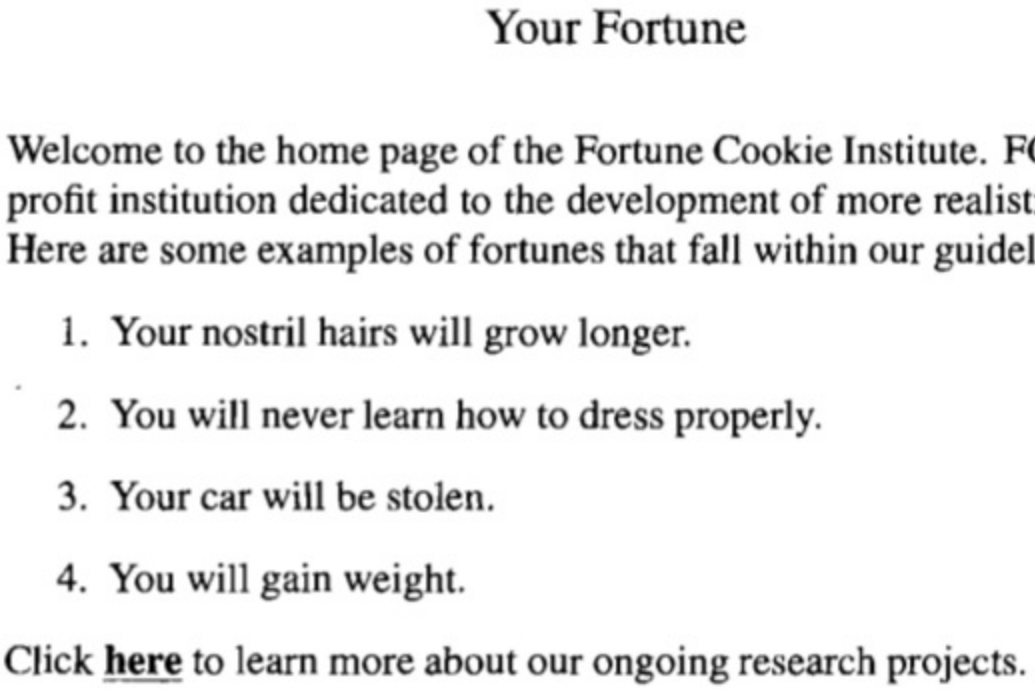

图 **16.2** 一个网页

注意在尖角括号之间的文本并没有被显示出来，这些用尖角括号包围的文本就是标签。HTML 的标签分为两种，一种是成双成对地出现的：

```
<tag>...</tag>
```

第一个标签标志着某种情景（environment）的开始，而第二个标签标志着这种情景的结束。 这种标签的一个例子是 `<h2>` ：所有被 `<h2>` 和 `</h2>` 包围的文本，都会使用比平常字体尺寸稍大的字体来显示。

另外一些成双成对出现的标签包括：创建带编号列表的 `<ol>` 标签（ol 代表 ordered list，有序表），令文本居中的 `<center>` 标签，以及创建链接的 `<a>` 标签（a 代表 anchor，锚点）。

被 `<a>` 和 `</a>` 包围的文本就是超文本（hypertext）。 在大多数浏览器上，超文本都会以一种与众不同的方式被凸显出来 —— 它们通常会带有下划线 —— 并且点击这些文本会让浏览器跳转到另一个页面。 在标签 `a` 之后的部分，指示了链接被点击时，浏览器应该跳转到的位置。

一个像

```
<a href="foo.html">
```

这样的标签，就标识了一个指向另一个 HTML 文件的链接，其中这个 HTML 文件和当

前网页的文件夹相同。 当点击这个链接时，浏览器就会获取并显示 `foo.html` 这个文件。

当然，链接并不一定都要指向相同文件夹下的 HTML 文件，实际上，一个链接可以指向互联网的任何一个文件。

和成双成对出现的标签相反，另一种标签没有结束标记。 在图 16.1 里有一些这样的标签，包括：创建一个新文本行的 `<br>` 标签（br 代表 break ，断行），以及在列表情景中，创建一个新列表项的 `<li>` 标签（li 代表 list item ，列表项）。

HTML 还有不少其他的标签，但是本章要用到的标签，基本都包含在图 16.1 里了。

16.2 HTML 实用函数 (HTML Utilities)

```
(defmacro as (tag content)
  `(format t "<~(~A~)>~A</~(~A~)>"
           ',tag ,content ',tag))

(defmacro with (tag &rest body)
  `(progn
     (format t "~&<~(~A~)>~%" ',tag)
     ,@body
     (format t "~&</~(~A~)>~%" ',tag)))

(defmacro brs (&optional (n 1))
  (fresh-line)
  (dotimes (i n)
    (princ "<br>"))
  (terpri))
```

图 **16.3** 标签生成例程

本节会定义一些生成 HTML 的例程。 图 16.3 包含了三个基本的、生成标签的例程。 所有例程都将它们的输出发送到 `*standard-output*` ；可以通过重新绑定这个变量，将输出重定向到一个文件。

宏 `as` 和 `with` 都用于在标签之间生成表达式。其中 `as` 接受一个字符串，并将它打印在两个标签之间：

```
> (as center "The Missing Lambda")
<center>The Missing Lambda</center>
NIL
```

`with` 则接受一个代码体（body of code），并将它放置在两个标签之间：

```
> (with center
    (princ "The Unbalanced Parenthesis"))
<center>
The Unbalanced Parenthesis
</center>
NIL
```

两个宏都使用了 ~(...~) 来进行格式化，从而将标签转化为小写字母的标签。 HTML 并不介意标签是大写还是小写，但是在包含许许多多标签的 HTML 文件中，小写字母的标签可读性更好一些。

除此之外， as 倾向于将所有输出都放在同一行，而 with 则将标签和内容都放在不同的行里。 （使用 ~& 来进行格式化，以确保输出从一个新行中开始。） 以上这些工作都只是为了让 HTML 更具可读性，实际上，标签之外的空白并不影响页面的显示方式。

图 16.3 中的最后一个例程 brs 用于创建多个文本行。 在很多浏览器中，这个例程都可以用于控制垂直间距。

```
(defun html-file (base)
  (format nil "~(~A~).html" base))

(defmacro page (name title &rest body)
  (let ((ti (gensym)))
    `(with-open-file (*standard-output*
                      (html-file ,name)
                      :direction :output
                      :if-exists :supersede)
       (let ((,ti ,title))
         (as title ,ti)
         (with center
           (as h2 (string-upcase ,ti)))
         (brs 3)
         ,@body))))
```

图 **16.4 HTML** 文件生成例程

图 16.4 包含用于生成 HTML 文件的例程。 第一个函数根据给定的符号（symbol）返回一个文件名。 在一个实际应用中，这个函数可能会返回指向某个特定文件夹的路径（path）。 目前来说，这个函数只是简单地将 .html 后缀追加到给定符号名的后边。

宏 page 负责生成整个页面，它的实现和 with-open-file 很相似： body 中的表达式会被求值，求值的结果通过 *standard-output* 所绑定的流，最终被写入到相应的 HTML 文件中。

6.7 小节展示了如何临时性地绑定一个特殊变量。 在 113 页的例子中，我们在 let 的体内将 *print-base* 绑定为 16 。 这一次，通过将 *standard-output* 和一个指向 HTML

文件的流绑定，只要我们在 page 的函数体内调用 as 或者 princ ，输出就会被传送到 HTML 文件里。

page 宏的输出先在顶部打印 title ，接着求值 body 中的表达式，打印 body 部分的输出。

如果我们调用

```
(page 'paren "The Unbalanced Parenthesis"
  (princ "Something in his expression told her..."))
```

这会产生一个名为 paren.html 的文件（文件名由 html-file 函数生成），文件中的内容为：

```
<title>The Unbalanced Parenthesis</title>
<center>
<h2>THE UNBALANCED PARENTHESIS</h2>
</center>
<br><br><br>
Something in his expression told her...
```

除了 title 标签以外，以上输出的所有 HTML 标签在前面已经见到过了。 被 <title> 标签包围的文本并不显示在网页之内，它们会显示在浏览器窗口，用作页面的标题。

```
(defmacro with-link (dest &rest body)
  `(progn
     (format t "<a href=\"~A\">" (html-file ,dest))
     ,@body
     (princ "</a>")))

(defun link-item (dest text)
  (princ "<li>")
  (with-link dest
    (princ text)))

(defun button (dest text)
  (princ "[ ")
  (with-link dest
    (princ text))
  (format t " ]~%"))
```

图 **16.5** 生成链接的例程

图片 16.5 给出了用于生成链接的例程。 with-link 和 with 很相似：它根据给定的地址 dest ，创建一个指向 HTML 文件的链接。 而链接内部的文本，则通过求值 body 参数中的代码段得出：

```
> (with-link 'capture
    (princ "The Captured Variable"))
<a href="capture.html">The Captured Variable</a>
"</a>"
```

`with-link` 也被用在 `link-item` 当中，这个函数接受一个字符串，并创建一个带链接的列表项：

```
> (link-item 'bq "Backquote!")
<li><a href="bq.html">Backquote!</a>
"</a>"
```

最后， `button` 也使用了 `with-link` ，从而创建一个被方括号包围的链接：

```
> (button 'help "Help")
[ <a href="help.html">Help</a> ]
NIL
```

16.3 迭代式实用函数 (An Iteration Utility)

在这一节，我们先暂停一下编写 HTML 生成器的工作，转到编写迭代式例程的工作上来。

你可能会问，怎样才能知道，什么时候应该编写主程序，什么时候又应该编写子例程？

实际上，这个问题，没有答案。

通常情况下，你总是先开始写一个程序，然后发现需要写一个新的例程，于是你转而去编写新例程，完成它，接着再回过头去编写原来的程序。 时间关系，要在这里演示这个开始-完成-又再开始的过程是不太可能的，这里只展示这个迭代式例程的最终形态，需要注意的是，这个程序的编写并不如想象中的那么简单。 程序通常需要经历多次重写，才会变得简单。

```
(defun map3 (fn lst)
  (labels ((rec (curr prev next left)
             (funcall fn curr prev next)
             (when left
               (rec (car left)
                    curr
                    (cadr left)
                    (cdr left)))))
    (when lst
      (rec (car lst) nil (cadr lst) (cdr lst)))))
```

图 **16.6** 对树进行迭代

图 16.6 里定义的新例程是 mapc 的一个变种。它接受一个函数和一个列表作为参数，对于传入列表中的每个元素，它都会用三个参数来调用传入函数，分别是元素本身，前一个元素，以及后一个元素。（当没有前一个元素或者后一个元素时，使用 nil 代替。）

```
> (map3 #'(lambda (&rest args) (princ args))
        '(a b c d))
(A NIL B) (B A C) (C B D) (D C NIL)
NIL
```

和 mapc 一样， map3 总是返回 nil 作为函数的返回值。需要这类例程的情况非常多。在下一个小节就会看到，这个例程是如何让每个页面都实现“前进一页”和“后退一页”功能的。

map3 的一个常见功能是，在列表的两个相邻元素之间进行某些处理：

```
> (map3 #'(lambda (c p n)
            (princ c)
            (if n (princ " | ")))
        '(a b c d))
A | B | C | D
NIL
```

程序员经常会遇到上面的这类问题，但只要花些功夫，定义一些例程来处理它们，就能为后续工作节省不少时间。

16.4 生成页面 (Generating Pages)

一本书可以有任意数量的大章，每个大章又有任意数量的小节，而每个小节又有任意数量的分节，整本书的结构呈现出一棵树的形状。

尽管网页使用的术语和书本不同，但多个网页同样可以被组织成树状。

本节要构建的是这样一个程序，它生成多个网页，这些网页带有以下结构： 第一页是一个目录，目录中的链接指向各个节点（section）页面。 每个节点包含一些指向项（item）的链接。 而一个项就是一个包含纯文本的页面。

除了页面本身的链接以外，根据页面在树状结构中的位置，每个页面都会带有前进、后退和向上的链接。 其中，前进和后退链接用于在同级（sibling）页面中进行导航。 举个例子，点击一个项页面中的前进链接时，如果这个项的同一个节点下还有下一个项，那么就跳到这个新项的页面里。 另一方面，向上链接将页面跳转到树形结构的上一层 —— 如果当前页面是项页面，那么返回到节点页面；如果当前页面是节点页面，那么返回到目录页面。 最后，还会有索引页面：这个页面包含一系列链接，按字母顺序排列所有项。

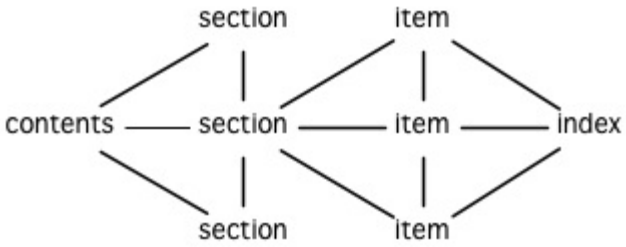

图 **16.7** 网站的结构

图 16.7 展示了生成程序创建的页面所形成的链接结构。

```
(defparameter *sections* nil)

(defstruct item
  id title text)

(defstruct section
  id title items)

(defmacro defitem (id title text)
  `(setf ,id
         (make-item :id     ',id
                    :title  ,title
                    :text   ,text)))

(defmacro defsection (id title &rest items)
  `(setf ,id
         (make-section :id    ',id
                       :title ,title
                       :items (list ,@items))))

(defun defsite (&rest sections)
  (setf *sections* sections))
```

图 **16.8** 定义一个网站

图 16.8 包含定义页面所需的数据结构。程序需要处理两类对象：项和节点。这两类对象的结构很相似，不过节点包含的是项的列表，而项包含的是文本块。

节点和项两类对象都带有 `id` 域。 标识符（id）被用作符号（symbol），并达到以下两个目的：在 `defitem` 和 `defsection` 的定义中， 标识符会被设置到被创建的项或者节点当中，作为我们引用它们的一种手段；另一方面，标识符还会作为相应文件的前缀名（base name），比如说，如果项的标识符为 `foo` ，那么项就会被写到 `foo.html` 文件当中。

节点和项也同时带有 `title` 域。这个域的值应该为字符串，并且被用作相应页面的标

题。

在节点里，项的排列顺序由传给 `defsection` 的参数决定。 与此类似，在目录里，节点的排列顺序由传给 `defsite` 的参数决定。

```
(defconstant contents "contents")
(defconstant index    "index")

(defun gen-contents (&optional (sections *sections*))
  (page contents contents
    (with ol
      (dolist (s sections)
        (link-item (section-id s) (section-title s))
        (brs 2))
      (link-item index (string-capitalize index)))))

(defun gen-index (&optional (sections *sections*))
  (page index index
    (with ol
      (dolist (i (all-items sections))
        (link-item (item-id i) (item-title i))
        (brs 2)))))

(defun all-items (sections)
  (let ((is nil))
    (dolist (s sections)
      (dolist (i (section-items s))
        (setf is (merge 'list (list i) is #'title<))))
    is))

(defun title< (x y)
  (string-lessp (item-title x) (item-title y)))
```

图 **16.9** 生成索引和目录

图 16.9 包含的函数用于生成索引和目录。 常量 `contents` 和 `index` 都是字符串，它们分别用作 `contents` 页面的标题和 `index` 页面的标题；另一方面，如果有其他页面包含了目录和索引这两个页面，那么这两个常量也会作为这些页面文件的前缀名。

函数 `gen-contents` 和 `gen-index` 非常相似。 它们都打开一个 HTML 文件，生成标题和链接列表。 不同的地方是，索引页面的项必须是有序的。 有序列表通过 `all-items` 函数生成，它遍历各个项并将它加入到保存已知项的列表当中，并使用 `title<` 函数作为排序函数。 注意，因为 `title<` 函数对大小写敏感，所以在对比标题前，输入必须先经过 `string-lessp` 处理，从而忽略大小写区别。

实际程序中的对比操作通常更复杂一些。举个例子，它们需要忽略无意义的句首词汇，比如 `"a"` 和 `"the"` 。

```
(defun gen-site ()
  (map3 #'gen-section *sections*)
  (gen-contents)
  (gen-index))

(defun gen-section (sect <sect sect>)
  (page (section-id sect) (section-title sect)
    (with ol
      (map3 #'(lambda (item <item item>)
                (link-item (item-id item)
                           (item-title item))
                (brs 2)
                (gen-item sect item <item item>))
            (section-items sect)))
    (brs 3)
    (gen-move-buttons (if <sect (section-id <sect))
                      contents
                      (if sect> (section-id sect>)))))

(defun gen-item (sect item <item item>)
  (page (item-id item) (item-title item)
    (princ (item-text item))
    (brs 3)
    (gen-move-buttons (if <item (item-id <item))
                      (section-id sect)
                      (if item> (item-id item>)))))

(defun gen-move-buttons (back up forward)
  (if back (button back "Back"))
  (if up (button up "Up"))
  (if forward (button forward "Forward")))
```

图 **16.10** 生成网站、节点和项

图 16.10 包含其余的代码： `gen-site` 生成整个页面集合，并调用相应的函数，生成节点和项。

所有页面的集合包括目录、索引、各个节点以及各个项的页面。 目录和索引的生成由图 16.9 中的代码完成。 节点和项由分别由生成节点页面的 `gen-section` 和生成项页面的 `gen-item` 完成。

这两个函数的开头和结尾非常相似。 它们都接受一个对象、对象的左兄弟、对象的右兄弟作为参数；它们都从对象的 `title` 域中提取标题内容；它们都以调用 `gen-move-buttons` 作为结束，其中 `gen-move-buttons` 创建指向左兄弟的后退按钮、指向右兄弟的前进按钮和指向双亲（parent）对象的向上按钮。 它们的不同在于函数体的中间部分： `gen-section` 创建有序列表，列表中的链接指向节点包含的项，而 `gen-item` 创建的项则链接到相应的文本页面。

项所包含的内容完全由用户决定。 比如说，将 HTML 标签作为内容也是完全没问题的。 项的文本当然也可以由其他程序来生成。

图 16.11 演示了如何手工地定义一个微型网页。 在这个例子中，列出的项都是 Fortune 饼干公司新推出的产品。

```
(defitem des "Fortune Cookies: Dessert or Fraud?" "...")

(defitem case "The Case for Pessimism" "...")

(defsection position "Position Papers" des case)

(defitem luck "Distribution of Bad Luck" "...")

(defitem haz "Health Hazards of Optimism" "...")

(defsection abstract "Research Abstracts" luck haz)

(defsite position abstract)
```

图 **16.11** 一个微型网站

第十七章：示例：对象

在本章里，我们将使用 Lisp 来自己实现面向对象语言。这样子的程序称为嵌入式语言 (*embedded language*)。嵌入一个面向对象语言到 Lisp 里是一个绝佳的例子。同時作为一个 Lisp 的典型用途，並演示了面向对象的抽象是如何多自然地在 Lisp 基本的抽象上构建出来。

17.1 继承 (Inheritance)

11.10 小节解释过通用函数与消息传递的差别。

在消息传递模型里，

1. 对象有属性，
2. 并回应消息，
3. 并从其父类继承属性与方法。

当然了，我们知道 CLOS 使用的是通用函数模型。但本章我们只对于写一个迷你的对象系统 (minimal object system)感兴趣，而不是一个可与 CLOS 匹敌的系统，所以我们将使用消息传递模型。

我们已经在 Lisp 里看过许多保存属性集合的方法。一种可能的方法是使用哈希表来代表对象，并将属性作为哈希表的条目保存。接著可以通过 `gethash` 来存取每个属性：

```
(gethash 'color obj)
```

由于函数是数据对象，我们也可以将函数作为属性保存起来。这表示我们也可以有方法；要调用一个对象特定的方法，可以通过 `funcall` 一下哈希表里的同名属性：

```
(funcall (gethash 'move obj) obj 10)
```

我们可以在这个概念上，定义一个 Smalltalk 风格的消息传递语法，

```
(defun tell (obj message &rest args)
  (apply (gethash message obj) obj args))
```

所以想要一个对象 `obj` 移动 10 单位，我们可以说：

```
(tell obj 'move 10)
```

事实上，纯 Lisp 唯一缺少的原料是继承。我们可以通过定义一个递归版本的 gethash 来实现一个简单版，如图 17.1 。现在仅用共 8 行代码，便实现了面向对象编程的 3 个基本元素。

```
(defun rget (prop obj)
  (multiple-value-bind (val in) (gethash prop obj)
    (if in
        (values val in)
        (let ((par (gethash :parent obj)))
          (and par (rget prop par))))))

(defun tell (obj message &rest args)
  (apply (rget message obj) obj args))
```

图 **17.1**：继承

让我们用这段代码，来试试本来的例子。我们创建两个对象，其中一个对象是另一个的子类：

```
> (setf circle-class (make-hash-table)
        our-circle   (make-hash-table)
        (gethash :parent our-circle) circle-class
        (gethash 'radius our-circle) 2)
2
```

circle-class 对象会持有给所有圆形使用的 area 方法。它是接受一个参数的函数，该参数为传来原始消息的对象：

```
> (setf (gethash 'area circle-class)
        #'(lambda (x)
            (* pi (expt (rget 'radius x) 2))))
#<Interpreted-Function BF1EF6>
```

现在当我们询问 our-circle 的面积时，会根据此类所定义的方法来计算。我们使用 rget 来读取一个属性，用 tell 来调用一个方法：

```
> (rget 'radius our-circle)
2
T
> (tell our-circle 'area)
12.566370614359173
```

在开始改善这个程序之前，值得停下来想想我们到底做了什么。仅使用 8 行代码，我们使纯的、旧的、无 CLOS 的 Lisp ，转变成一个面向对象语言。我们是怎么完成这项壮举的？应该用了某种秘诀，才会仅用了 8 行代码，就实现了面向对象编程。

的确有一个秘诀存在，但不是编程的奇技淫巧。这个秘诀是，Lisp 本来就是一个面向对象的语言了，甚至说，是种更通用的语言。我们需要做的事情，不过就是把本来就存在的抽象，再重新包装一下。

17.2 多重继承 (Multiple Inheritance)

到目前为止我们只有单继承 —— 一个对象只可以有一个父类。但可以通过使 `parent` 属性变成一个列表来获得多重继承，并重新定义 `rget` ，如图 17.2 所示。

在只有单继承的情况下，当我们想要从对象取出某些属性，只需要递归地延著祖先的方向往上找。如果对象本身没有我们想要属性的有关信息，可以检视其父类，以此类推。有了多重继承后，我们仍想要执行同样的搜索，但这件简单的事，却被对象的祖先可形成一个图，而不再是简单的树给复杂化了。不能只使用深度优先来搜索这个图。有多个父类时，可以有如图 17.3 所示的层级存在： `a` 起源于 `b` 及 `c` ，而他们都是 `d` 的子孙。一个深度优先（或说高度优先）的遍历结果会是 `a` , `b` , `d`, `c` , `d` 。而如果我们想要的属性在 `d` 与 `c` 都有的话，我们会获得存在 `d` 的值，而不是存在 `c` 的值。这违反了子类可覆写父类提供缺省值的原则。

如果我们想要实现普遍的继承概念，就不应该在检查其子孙前，先检查该对象。在这个情况下，适当的搜索顺序会是 `a` , `b` , `c` , `d` 。那如何保证搜索总是先搜子孙呢？最简单的方法是用一个对象，以及按正确优先顺序排序的，由祖先所构成的列表。通过调用 `traverse` 开始，建构一个列表，表示深度优先遍历所遇到的对象。如果任一个对象有共享的父类，则列表中会有重复元素。如果仅保存最后出现的复本，会获得一般由 CLOS 定义的优先级列表。（删除所有除了最后一个之外的复本，根据 183 页所描述的算法，规则三。）Common Lisp 函数 `delete-duplicates` 定义成如此作用的，所以我们只要在深度优先的基础上调用它，我们就会得到正确的优先级列表。一旦优先级列表创建完成， `rget` 根据需要的属性搜索第一个符合的对象。

我们可以通过利用优先级列表的优点，举例来说，一个爱国的无赖先是一个无赖，然后才是爱国者：

```
> (setf scoundrel            (make-hash-table)
        patriot              (make-hash-table)
        patriotic-scoundrel (make-hash-table)
        (gethash 'serves scoundrel) 'self
        (gethash 'serves patriot) 'country
        (gethash :parents patriotic-scoundrel)
                 (list scoundrel patriot))
(#<Hash-Table C41C7E> #<Hash-Table C41F0E>)
> (rget 'serves patriotic-scoundrel)
SELF
T
```

到目前为止，我们有一个强大的程序，但极其丑陋且低效。在一个 Lisp 程序生命周期的第二阶段，我们将这个初步框架提炼成有用的东西。

17.3 定义对象 (Defining Objects)

第一个我们需要改善的是，写一个用来创建对象的函数。我们程序表示对象以及其父类的方式，不需要给用户知道。如果我们定义一个函数来创建对象，用户将能够一个步骤就创建出一个对象，并指定其父类。我们可以在创建一个对象的同时，顺道构造优先级列表，而不是在每次当我们需要找一个属性或方法时，才花费庞大代价来重新构造。

如果我们要维护优先级列表，而不是在要用的时候再构造它们，我们需要处理列表会过时的可能性。我们的策略会是用一个列表来保存所有存在的对象，而无论何时当某些父类被改动时，重新给所有受影响的对象生成优先级列表。这代价是相当昂贵的，但由于查询比重定义父类的可能性来得高许多，我们会省下许多时间。这个改变对我们的程序的灵活性没有任何影响；我们只是将花费从频繁的操作转到不频繁的操作。

图 17.4 包含了新的代码。 λ [http://acl.readthedocs.org/en/latest/zhCN/notes-cn.html#notes-273] 全局的 `*objs*` 会是一个包含所有当前对象的列表。函数 `parents` 取出一个对象的父类；相反的 `(setf parents)` 不仅配置一个对象的父类，也调用 `make-precedence` 来重新构造任何需要变动的优先级列表。这些列表与之前一样，由 `precedence` 来构造。

用户现在不用调用 `make-hash-table` 来创建对象，调用 `obj` 来取代， `obj` 一步完成创建一个新对象及定义其父类。我们也重定义了 `rget` 来利用保存优先级列表的好处。

```
(defvar *objs* nil)

(defun parents (obj) (gethash :parents obj))

(defun (setf parents) (val obj)
  (prog1 (setf (gethash :parents obj) val)
         (make-precedence obj)))

(defun make-precedence (obj)
  (setf (gethash :preclist obj) (precedence obj))
  (dolist (x *objs*)
    (if (member obj (gethash :preclist x))
        (setf (gethash :preclist x) (precedence x)))))

(defun obj (&rest parents)
  (let ((obj (make-hash-table)))
    (push obj *objs*)
    (setf (parents obj) parents)
    obj))

(defun rget (prop obj)
  (dolist (c (gethash :preclist obj))
```

```
    (multiple-value-bind (val in) (gethash prop c)
      (if in (return (values val in))))))
```

图 **17.4:** 创建对象

17.4 函数式语法 (Functional Syntax)

另一个可以改善的空间是消息调用的语法。 `tell` 本身是无谓的杂乱不堪，这也使得动词在第三顺位才出现，同时代表著我们的程序不再可以像一般 Lisp 前序表达式那样阅读:

```
(tell (tell obj 'find-owner) 'find-owner)
```

我们可以使用图 17.5 所定义的 `defprop` 宏，通过定义作为函数的属性名称来摆脱这种 `tell` 语法。若选择性参数 `meth?` 为真的话，会将此属性视为方法。不然会将属性视为槽，而由 `rget` 所取回的值会直接返回。一旦我们定义了属性作为槽或方法的名字，

```
(defmacro defprop (name &optional meth?)
  `(progn
     (defun ,name (obj &rest args)
       ,(if meth?
          `(run-methods obj ',name args)
          `(rget ',name obj)))
     (defun (setf ,name) (val obj)
       (setf (gethash ',name obj) val))))

(defun run-methods (obj name args)
  (let ((meth (rget name obj)))
    (if meth
        (apply meth obj args)
        (error "No ~A method for ~A." name obj))))
```

图 **17.5:** 函数式语法

```
(defprop find-owner t)
```

我们就可以在函数调用里引用它，则我们的代码读起来将会再次回到 Lisp 本来那样：

```
(find-owner (find-owner obj))
```

我们的前一个例子在某种程度上可读性变得更高了：

```
> (progn
    (setf scoundrel           (obj)
          patriot             (obj)
```

```
          patriotic-scoundrel (obj scoundrel patriot))
    (defprop serves)
    (setf (serves scoundrel) 'self
          (serves patriot) 'country)
    (serves patriotic-scoundrel))
SELF
T
```

17.5 定义方法 (Defining Methods)

到目前为止，我们借由叙述如下的东西来定义一个方法：

```
(defprop area t)

(setf circle-class (obj))

(setf (area circle-class)
      #'(lambda (c) (* pi (expt (radius c) 2))))
```

```
(defmacro defmeth (name obj parms &rest body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (setf (gethash ',name ,gobj)
             (labels ((next () (get-next ,gobj ',name)))
               #'(lambda ,parms ,@body))))))

(defun get-next (obj name)
  (some #'(lambda (x) (gethash name x))
        (cdr (gethash :preclist obj))))
```

图 **17.6** 定义方法。

在一个方法里，我们可以通过给对象的 `:preclist` 的 `cdr` 获得如内置 `call-next-method` 方法的效果。所以举例来说，若我们想要定义一个特殊的圆形，这个圆形在返回面积的过程中印出某个东西，我们可以说：

```
(setf grumpt-circle (obj circle-class))

(setf (area grumpt-circle)
      #'(lambda (c)
          (format t "How dare you stereotype me!~%")
          (funcall (some #'(lambda (x) (gethash 'area x))
                         (cdr (gethash :preclist c)))
                   c)))
```

这里 `funcall` 等同于一个 `call-next-method` 调用，但他..

图 17.6 的 defmeth 宏提供了一个便捷方式来定义方法，并使得调用下个方法变得简单。一个 defmeth 的调用会展开成一个 setf 表达式，但 setf 在一個 labels 表达式里定义了 next 作为取出下个方法的函数。这个函数与 next-method-p 类似（第 188 页「譯註: 11.7 節」），但返回的是我们可以调用的东西，同時作為 call-next-method 。λ [http://acl.readthedocs.org/en/latest/zhCN/notes-cn.html#notes-273] 前述两个方法可以被定义成：

```
(defmeth area circle-class (c)
  (* pi (expt (radius c) 2)))

(defmeth area grumpy-circle (c)
  (format t "How dare you stereotype me!~%")
  (funcall (next) c))
```

顺道一提，注意 defmeth 的定义也利用到了符号捕捉。方法的主体被插入至函数 next 是局部定义的一个上下文里。

17.6 实例 (Instances)

到目前为止，我们还没有将类别与实例做区别。我们使用了一个术语来表示两者，对象(*object*)。将所有的对象视为一体是优雅且灵活的，但这非常没效率。在许多面向对象应用里，继承图的底部会是复杂的。举例来说，模拟一个交通情况，我们可能有少于十个对象来表示车子的种类，但会有上百个对象来表示特定的车子。由于后者会全部共享少数的优先级列表，创建它们是浪费时间的，并且浪费空间来保存它们。

图 17.7 定义一个宏 inst ，用来创建实例。实例就像其他对象一样（现在也可称为类别），有区别的是只有一个父类且不需维护优先级列表。它们也没有包含在列表 *objs** 里。在前述例子里，我们可以说：

```
(setf grumpy-circle (inst circle-class))
```

由于某些对象不再有优先级列表，函数 rget 以及 get-next 现在被重新定义，检查这些对象的父类来取代。获得的效率不用拿灵活性交换。我们可以对一个实例做任何我们可以给其它种对象做的事，包括创建一个实例以及重定义其父类。在后面的情况里，(setf parents) 会有效地将对象转换成一个“类别”。

17.7 新的实现 (New Implementation)

我们到目前为止所做的改善都是牺牲灵活性交换而来。在这个系统的开发后期，一个 Lisp 程序通常可以牺牲些许灵活性来获得好处，这里也不例外。目前为止我们使用哈希表来表示所有的对象。这给我们带来了超乎我们所需的灵活性，以及超乎我们所想的花

费。在这个小节里，我们会重写我们的程序，用简单向量来表示对象。

```
(defun inst (parent)
  (let ((obj (make-hash-table)))
    (setf (gethash :parents obj) parent)
    obj))

(defun rget (prop obj)
  (let ((prec (gethash :preclist obj)))
    (if prec
        (dolist (c prec)
          (multiple-value-bind (val in) (gethash prop c)
            (if in (return (values val in)))))
      (multiple-value-bind (val in) (gethash prop obj)
        (if in
            (values val in)
            (rget prop (gethash :parents obj)))))))

(defun get-next (obj name)
  (let ((prec (gethash :preclist obj)))
    (if prec
        (some #'(lambda (x) (gethash name x))
              (cdr prec))
      (get-next (gethash obj :parents) name))))
```

图 **17.7:** 定义实例

这个改变意味著放弃动态定义新属性的可能性。目前我们可通过引用任何对象，给它定义一个属性。现在当一个类别被创建时，我们会需要给出一个列表，列出该类有的新属性，而当实例被创建时，他们会恰好有他们所继承的属性。

在先前的实现里，类别与实例没有实际区别。一个实例只是一个恰好有一个父类的类别。如果我们改动一个实例的父类，它就变成了一个类别。在新的实现里，类别与实例有实际区别；它使得将实例转成类别不再可能。

在图 17.8-17.10 的代码是一个完整的新实现。图片 17.8 给创建类别与实例定义了新的操作符。类别与实例用向量来表示。表示类别与实例的向量的前三个元素包含程序自身要用到的信息，而图 17.8 的前三个宏是用来引用这些元素的：

```
(defmacro parents (v) `(svref ,v 0))
(defmacro layout (v) `(the simple-vector (svref ,v 1)))
(defmacro preclist (v) `(svref ,v 2))

(defmacro class (&optional parents &rest props)
  `(class-fn (list ,@parents) ',props))

(defun class-fn (parents props)
  (let* ((all (union (inherit-props parents) props))
         (obj (make-array (+ (length all) 3)
```

```
                              :initial-element :nil)))
    (setf (parents obj)  parents
          (layout obj)   (coerce all 'simple-vector)
          (preclist obj) (precedence obj))
    obj))

(defun inherit-props (classes)
  (delete-duplicates
    (mapcan #'(lambda (c)
                (nconc (coerce (layout c) 'list)
                       (inherit-props (parents c))))
            classes)))

(defun precedence (obj)
  (labels ((traverse (x)
             (cons x
                   (mapcan #'traverse (parents x)))))
    (delete-duplicates (traverse obj))))

(defun inst (parent)
  (let ((obj (copy-seq parent)))
    (setf (parents obj)  parent
          (preclist obj) nil)
    (fill obj :nil :start 3)
    obj))
```

图 **17.8:** 向量实现：创建

1. `parents` 字段取代旧实现中，哈希表条目里 `:parents` 的位置。在一个类别里，`parents` 会是一个列出父类的列表。在一个实例里，`parents` 会是一个单一的父类。
2. `layout` 字段是一个包含属性名字的向量，指出类别或实例的从第四个元素开始的设计 (layout)。
3. `preclist` 字段取代旧实现中，哈希表条目里 `:preclist` 的位置。它会是一个类别的优先级列表，实例的话就是一个空表。

因为这些操作符是宏，他们全都可以被 `setf` 的第一个参数使用（参考 10.6 节）。

`class` 宏用来创建类别。它接受一个含有其基类的选择性列表，伴随著零个或多个属性名称。它返回一个代表类别的对象。新的类别会同时有自己本身的属性名，以及从所有基类继承而来的属性。

```
> (setf *print-array* nil
        gemo-class (class nil area)
        circle-class (class (geom-class) radius))
#<Simple-Vector T 5 C6205E>
```

这里我们创建了两个类别：`geom-class` 没有基类，且只有一个属性，`area`；`circle-`

class 是 gemo-class 的子类，并添加了一个属性， radius 。 [1] circle-class 类的设计

```
> (coerce (layout circle-class) 'list)
(AREA RADIUS)
```

显示了五个字段里，最后两个的名称。 [2]

class 宏只是一个 class-fn 的介面，而 class-fn 做了实际的工作。它调用 inherit-props 来汇整所有新对象的父类，汇整成一个列表，创建一个正确长度的向量，并适当地配置前三个字段。（ preclist 由 precedence 创建，本质上 precedence 没什么改变。）类别余下的字段设置为 :nil 来指出它们尚未初始化。要检视 circle-class 的 area 属性，我们可以：

```
> (svref circle-class
         (+ (position 'area (layout circle-class)) 3))
:NIL
```

稍后我们会定义存取函数来自动办到这件事。

最后，函数 inst 用来创建实例。它不需要是一个宏，因为它仅接受一个参数：

```
> (setf our-circle (inst circle-class))
#<Simple-Vector T 5 C6464E>
```

比较 inst 与 class-fn 是有益学习的，它们做了差不多的事。因为实例仅有一个父类，不需要决定它继承什么属性。实例可以仅拷贝其父类的设计。它也不需要构造一个优先级列表，因为实例没有优先级列表。创建实例因此与创建类别比起来来得快许多，因为创建实例在多数应用里比创建类别更常见。

```
(declaim (inline lookup (setf lookup)))

(defun rget (prop obj next?)
  (let ((prec (preclist obj)))
    (if prec
        (dolist (c (if next? (cdr prec) prec) :nil)
          (let ((val (lookup prop c)))
            (unless (eq val :nil) (return val))))
        (let ((val (lookup prop obj)))
          (if (eq val :nil)
              (rget prop (parents obj) nil)
              val)))))

(defun lookup (prop obj)
  (let ((off (position prop (layout obj) :test #'eq)))
    (if off (svref obj (+ off 3)) :nil)))
```

```
(defun (setf lookup) (val prop obj)
  (let ((off (position prop (layout obj) :test #'eq)))
    (if off
        (setf (svref obj (+ off 3)) val)
        (error "Can't set ~A of ~A." val obj))))
```

图 **17.9:** 向量实现：存取

现在我们可以创建所需的类别层级及实例，以及需要的函数来读写它们的属性。图 17.9 的第一个函数是 `rget` 的新定义。它的形状与图 17.7 的 `rget` 相似。条件式的两个分支，分别处理类别与实例。

1. 若对象是一个类别，我们遍历其优先级列表，直到我们找到一个对象，其中欲找的属性不是 `:nil` 。如果没有找到，返回 `:nil` 。
2. 若对象是一个实例，我们直接查找属性，并在没找到时递回地调用 `rget` 。

`rget` 与 `next?` 新的第三个参数稍后解释。现在只要了解如果是 `nil` ， `rget` 会像平常那样工作。

函数 `lookup` 及其反相扮演著先前 `rget` 函数里 `gethash` 的角色。它们使用一个对象的 `layout` ，来取出或设置一个给定名称的属性。这条查询是先前的一个复本：

```
> (lookup 'area circle-class)
:NIL
```

由于 `lookup` 的 `setf` 也定义了，我们可以给 `circle-class` 定义一个 `area` 方法，通过：

```
(setf (lookup 'area circle-class)
      #'(lambda (c)
          (* pi (expt (rget 'radius c nil) 2))))
```

在这个程序里，和先前的版本一样，没有特别区别出方法与槽。一个“方法”只是一个字段，里面有着一个函数。这将很快会被一个更方便的前端所隐藏起来。

```
(declaim (inline run-methods))

(defmacro defprop (name &optional meth?)
  `(progn
     (defun ,name (obj &rest args)
       ,(if meth?
            `(run-methods obj ',name args)
            `(rget ',name obj nil)))
     (defun (setf ,name) (val obj)
       (setf (lookup ',name obj) val))))
```

```
(defun run-methods (obj name args)
  (let ((meth (rget name obj nil)))
    (if (not (eq meth :nil))
        (apply meth obj args)
        (error "No ~A method for ~A." name obj))))

(defmacro defmeth (name obj parms &rest body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (defprop ,name t)
       (setf (lookup ',name ,gobj)
             (labels ((next () (rget ,gobj ',name t)))
               #'(lambda ,parms ,@body))))))
```

图 **17.10:** 向量实现：宏介面

图 17.10 包含了新的实现的最后部分。这个代码没有给程序加入任何威力，但使程序更容易使用。宏 `defprop` 本质上没有改变；现在仅调用 `lookup` 而不是 `gethash` 。与先前相同，它允许我们用函数式的语法来引用属性：

```
> (defprop radius)
(SETF RADIUS)
> (radius our-circle)
:NIL
> (setf (radius our-circle) 2)
2
```

如果 `defprop` 的第二个选择性参数为真的话，它展开成一个 `run-methods` 调用，基本上也没什么改变。

最后，函数 `defmeth` 提供了一个便捷方式来定义方法。这个版本有三件新的事情：它隐含了 `defprop` ，它调用 `lookup` 而不是 `gethash` ，且它调用 `regt` 而不是 278 页的 `get-next` (译注: 图 17.7 的 `get-next`)来获得下个方法。现在我们理解给 `rget` 添加额外参数的理由。它与 `get-next` 非常相似，我们同样通过添加一个额外参数，在一个函数里实现。若这额外参数为真时， `rget` 取代 `get-next` 的位置。

现在我们可以达到先前方法定义所有的效果，但更加清晰：

```
(defmeth area circle-class (c)
  (* pi (expt (radius c) 2)))
```

注意我们可以直接调用 `radius` 而无须调用 `rget` ，因为我们使用 `defprop` 将它定义成一个函数。因为隐含的 `defprop` 由 `defmeth` 实现，我们也可以调用 `area` 来获得 `our-circle` 的面积：

```
> (area our-circle)
```

```
12.566370614359173
```

17.8 分析 (Analysis)

我们现在有了一个适合撰写实际面向对象程序的嵌入式语言。它很简单，但就大小来说相当强大。而在典型应用里，它也会是快速的。在一个典型的应用里，操作实例应比操作类别更常见。我们重新设计的重点在于如何使得操作实例的花费降低。

在我们的程序里，创建类别既慢且产生了许多垃圾。如果类别不是在速度为关键考量时创建，这还是可以接受的。会需要速度的是存取函数以及创建实例。这个程序里的没有做编译优化的存取函数大约与我们预期的一样快。 λ [http://acl.readthedocs.org/en/latest/zhCN/notes-cn.html#notes-284] 而创建实例也是如此。且两个操作都没有用到构造 (consing)。除了用来表达实例的向量例外。会自然的以为这应该是动态地配置才对。但我们甚至可以避免动态配置实例，如果我们使用像是 13.4 节所提出的策略。

我们的嵌入式语言是 Lisp 编程的一个典型例子。只不过是一个嵌入式语言就可以是一个例子了。但 Lisp 的特性是它如何从一个小的、受限版本的程序，进化成一个强大但低效的版本，最终演化成快速但稍微受限的版本。

Lisp 恶名昭彰的缓慢不是 Lisp 本身导致（Lisp 编译器早在 1980 年代就可以产生出与 C 编译器一样快的代码），而是由于许多程序员在第二个阶段就放弃的事实。如同 Richard Gabriel 所写的，

> 要在 Lisp 撰写出性能极差的程序相当简单；而在 C 这几乎是不可能的。 λ [http://acl.readthedocs.org/en/latest/zhCN/notes-cn.html#notes-284-2]

这完全是一个真的论述，但也可以解读为赞扬或贬低 Lisp 的论点：

1. 通过牺牲灵活性换取速度，你可以在 Lisp 里轻松地写出程序；在 C 语言里，你没有这个选择。
2. 除非你优化你的 Lisp 代码，不然要写出缓慢的软件根本易如反掌。

你的程序属于哪一种解读完全取决于你。但至少在开发初期，Lisp 使你有牺牲执行速度来换取时间的选择。

有一件我们示例程序没有做的很好的事是，它不是一个称职的 CLOS 模型（除了可能没有说明难以理解的 `call-next-method` 如何工作是件好事例外）。如大象般庞大的 CLOS 与这个如蚊子般微小的 70 行程序之间，存在多少的相似性呢？当然，这两者的差别是出自于教育性，而不是探讨有多相似。首先，这使我们理解到“面向对象”的广度。我们的程序比任何被称为是面向对象的都来得强大，而这只不过是 CLOS 的一小部

分威力。

我们程序与 CLOS 不同的地方是，方法是属于某个对象的。这个方法的概念使它们与对第一个参数做派发的函数相同。而当我们使用函数式语法来调用方法时，这看起来就跟 Lisp 的函数一样。相反地，一个 CLOS 的通用函数，可以派发它的任何参数。一个通用函数的组件称为方法，而若你将它们定义成只对第一个参数特化，你可以制造出它们是某个类或实例的方法的错觉。但用面向对象编程的消息传递模型来思考 CLOS 最终只会使你困惑，因为 CLOS 凌驾在面向对象编程之上。

CLOS 的缺点之一是它太庞大了，并且 CLOS 费煞苦心的隐藏了面向对象编程，其实只不过是改写 Lisp 的这个事实。本章的例子至少阐明了这一点。如果我们满足于旧的消息传递模型，我们可以用一页多一点的代码来实现。面向对象编程不过是 Lisp 可以做的小事之一而已。更发人深省的问题是，Lisp 除此之外还能做些什么？

脚注

[1] 当类别被显示时，`*print-array*` 应当是 `nil` 。 任何类别的 `preclist` 的第一个元素都是类别本身，所以试图显示类别的内部结构会导致一个无限循环。

[2] 这个向量被 coerced 成一个列表，只是为了看看里面有什么。有了 `*print-array*` 被设成 `nil` ，一个向量的内容应该不会显示出来。

附录 A：调试

这个附录演示了如何调试 Lisp 程序，并给出你可能会遇到的常见错误。

中断循环 (Breakloop)

如果你要求 Lisp 做些它不能做的事，求值过程会被一个错误讯息中断，而你会发现你位于一个称为中断循环的地方。中断循环工作的方式取决于不同的实现，但通常它至少会显示三件事：一个错误信息，一组选项，以及一个特别的提示符。

在中断循环里，你也可以像在顶层那样给表达式求值。在中断循环里，你或许能够找出错误的起因，甚至是修正它，并继续你程序的求值过程。然而，在一个中断循环里，你想做的最常见的事是跳出去。多数的错误起因于打错字或是小疏忽，所以通常你只会想终止程序并返回顶层。在下面这个假定的实现里，我们输入 `:abort` 来回到顶层。

```
> (/ 1 0)
Error: Division by zero.
       Options: :abort, :backtrace
>> :abort
>
```

在这些情况里，实际上的输入取决于实现。

当你在中断循环里，如果一个错误发生的话，你会到另一个中断循环。多数的 Lisp 会指出你是在第几层的中断循环，要嘛通过印出多个提示符，不然就是在提示符前印出数字：

```
>> (/ 2 0)
Error: Division by zero.
       Options: :abort, :backtrace, :previous
>>>
```

现在我们位于两层深的中断循环。此时我们可以选择回到前一个中断循环，或是直接返回顶层。

追踪与回溯 (Traces and Backtraces)

当你的程序不如你预期的那样工作时，有时候第一件该解决的事情是，它在做什么？如果你输入 `(trace foo)`，则 Lisp 会在每次调用或返回 `foo` 时显示一个信息，显示传给 `foo` 的参数，或是 `foo` 返回的值。你可以追踪任何自己定义的 (user-defined)函数。

一个追踪通常会根据调用树来缩进。在一个做遍历的函数，像下面这个函数，它给一个树的每一个非空元素加上 1，

```
(defun tree1+ (tr)
  (cond ((null tr) nil)
        ((atom tr) (1+ tr))
        (t (cons (tree1+ (car tr))
                 (tree1+ (cdr tr))))))
```

一个树的形状会因此反映出它被遍历时的数据结构：

```
> (trace tree1+)
(tree1+)
> (tree1+ '((1 . 3) 5 . 7))
1 Enter TREE1+ ((1 . 3) 5 . 7)
  2 Enter TREE1+ (1.3)
    3 Enter TREE1+ 1
    3 Exit TREE1+ 2
    3 Enter TREE1+ 3
    3 Exit TREE1+ 4
  2 Exit TREE1+ (2 . 4)
  2 Enter TREE1+ (5 . 7)
    3 Enter TREE1+ 5
    3 Exit TREE1+ 6
    3 Enter TREE1+ 7
    3 Exit TREE1+ 8
  2 Exit TREE1+ (6 . 8)
1 Exit TREE1+ ((2 . 4) 6 . 8)
((2 . 4) 6 . 8)
```

要关掉 `foo` 的追踪，输入 `(untrace foo)`；要关掉所有正在追踪的函数，只要输入 `(untrace)` 就好。

一个更灵活的追踪办法是在你的代码里插入诊断性的打印语句。如果已经知道结果了，这个经典的方法大概会与复杂的调适工具一样被使用数十次。这也是为什么可以互动地重定义函数式多么有用的原因。

一个回溯 (*backtrace*)是一个当前存在栈的调用的列表，当一个错误中止求值时，会由一个中断循环生成此列表。如果追踪像是"让我看看你在做什么"，一个回溯像是询问"我们是怎么到达这里的？" 在某方面上，追踪与回溯是互补的。一个追踪会显示在一个程序的调用树里，选定函数的调用。一个回溯会显示在一个程序部分的调用树里，所有函数的调用（路径为从顶层调用到发生错误的地方）。

在一个典型的实现里，我们可通过在中断循环里输入 `:backtrace` 来获得一个回溯，看起来可能像下面这样：

```
> (tree1+ ' ( ( 1 . 3) 5 . A))
```

```
Error: A is not a valid argument to 1+.
       Options: :abort, :backtrace
» :backtrace
(1+ A)
(TREE1+ A)
(TREE1+ (5 . A))
(TREE1+ ((1 . 3) 5 . A))
```

出现在回溯里的臭虫较容易被发现。你可以仅往回检查调用链，直到你找到第一个不该发生的事情。另一个函数式编程 (2.12 节)的好处是所有的臭虫都会在回溯里出现。在纯函数式代码里，每一个可能出错的调用，在错误发生时，一定会在栈出现。

一个回溯每个实现所提供的信息量都不同。某些实现会完整显示一个所有待调用的历史，并显示参数。其他实现可能仅显示调用历史。一般来说，追踪与回溯解释型的代码会得到较多的信息，这也是为什么你要在确定你的程序可以工作之后，再来编译。

传统上我们在解释器里调试代码，且只在工作的情况下才编译。但这个观点也是可以改变的：至少有两个 Common Lisp 实现没有包含解释器。

当什么事都没发生时 (When Noting Happens)

不是所有的 bug 都会打断求值过程。另一个常见并可能更危险的情况是，当 Lisp 好像不鸟你一样。通常这是程序进入无穷循环的徵兆。

如果你怀疑你进入了无穷循环，解决方法是中止执行，并跳出中断循环。

如果循环是用迭代写成的代码，Lisp 会开心地执行到天荒地老。但若是用递归写成的代码（没有做尾递归优化），你最终会获得一个信息，信息说 Lisp 把栈的空间给用光了：

```
> (defun blow-stack () (1+ (blow-stack)))
BLOW-STACK
> (blow-stack)
Error: Stack Overflow
```

在这两个情况里，如果你怀疑进入了无穷循环，解决办法是中断执行，并跳出由于中断所产生的中断循环。

有时候程序在处理一个非常庞大的问题时，就算没有进入无穷循环，也会把栈的空间用光。虽然这很少见。通常把栈空间用光是编程错误的徵兆。

递归函数最常见的错误是忘记了基本用例 (base case)。用英语来描述递归，通常会忽略基本用例。不严谨地说，我们可能说“obj 是列表的成员，如果它是列表的第一个元素，或是剩余列表的成员” 严格上来讲，应该添加一句“若列表为空，则 obj 不是列表的成

员”。不然我们描述的就是个无穷递归了。

在 Common Lisp 里，如果给入 nil 作为参数， car 与 cdr 皆返回 nil ：

```
> (car nil)
NIL
> (cdr nil)
NIL
```

所以若我们在 member 函数里忽略了基本用例：

```
(defun our-member (obj lst)
  (if (eql (car lst) obj)
      lst
      (our-member obj (cdr lst))))
```

要是我们找的对象不在列表里的话，则会陷入无穷循环。当我们到达列表底端而无所获时，递归调用会等价于：

```
(our-member obj nil)
```

在正确的定义中（第十六页「译注: 2.7 节」），基本用例在此时会停止递归，并返回 nil 。但在上面错误的定义里，函数愚昧地寻找 nil 的 car ，是 nil ，并将 nil 拿去跟我们寻找的对象比较。除非我们要找的对象刚好是 nil ，不然函数会继续在 nil 的 cdr 里寻找，刚好也是 nil —— 整个过程又重来了。

如果一个无穷循环的起因不是那么直观，可能可以通过看看追踪或回溯来诊断出来。无穷循环有两种。简单发现的那种是依赖程序结构的那种。一个追踪或回溯会即刻演示出，我们的 our-member 究竟哪里出错了。

比较难发现的那种，是因为数据结构有缺陷才发生的无穷循环。如果你无意中创建了环状结构（见 199页「12.3 节」），遍历结构的代码可能会掉入无穷循环里。这些 bug 很难发现，因为不在后面不会发生，看起来像没有错误的代码一样。最佳的解决办法是预防，如同 199 页所描述的：避免使用破坏性操作，直到程序已经正常工作，且你已准备好要调优代码来获得效率。

如果 Lisp 有不鸟你的倾向，也有可能是等待你完成输入什么。在多数系统里，按下回车是没有效果的，直到你输入了一个完整的表达式。这个方法的好事是它允许你输入多行的表达式。坏事是如果你无意中少了一个闭括号，或是一个闭引号，Lisp 会一直等你，直到你真正完成输入完整的表达式：

```
> (format t "for example ~A~% 'this)
```

这里我们在控制字符串的最后忽略了闭引号。在此时按下回车是没用的，因为 Lisp 认为我们还在输入一个字符串。

在某些实现里，你可以回到上一行，并插入闭引号。在不允许你回到前行的系统，最佳办法通常是中断执行，并从中断循环回到顶层。

没有值或未绑定 (No Value/Unbound)

一个你最常听到 Lisp 的抱怨是一个符号没有值或未绑定。数种不同的问题都用这种方式呈现。

局部变量，如 `let` 与 `defun` 设置的那些，只在创建它们的表达式主体里合法。所以要是我们试著在 创建变量的 `let` 外部引用它，

```
> (progn
    (let ((x 10))
      (format t "Here x = ~A. ~%" x))
    (format t "But now it's gone...~%")
    x)
Here x = 10.
But now it's gone...
Error: X has no value.
```

我们获得一个错误。当 Lisp 抱怨某些东西没有值或未绑定时，它的意思通常是你无意间引用了一个不存在的变量。因为没有叫做 `x` 的局部变量，Lisp 假定我们要引用一个有着这个名字的全局变量或常量。错误会发生是因为当 Lisp 试著要查找它的值的时候，却发现根本没有给值。打错变量的名字通常会给出同样的结果。

一个类似的问题发生在我们无意间将函数引用成变量。举例来说：

```
> defun foo (x) (+ x 1))
Error: DEFUN has no value
```

这在第一次发生时可能会感到疑惑： `defun` 怎么可能会没有值？问题的症结点在于我们忽略了最初的左括号，导致 Lisp 把符号 `defun` 解读错误，将它视为一个全局变量的引用。

有可能你真的忘记初始化某个全局变量。如果你没有给 `defvar` 第二个参数，你的全局变量会被宣告出来，但没有初始化；这可能是问题的根源。

意料之外的 Nil (Unexpected Nils)

当函数抱怨传入 `nil` 作为参数时，通常是程序先前出错的徵兆。数个内置操作符返回 `nil` 来指出失败。但由于 `nil` 是一个合法的 Lisp 对象，问题可能之后才发生，在程序某部分试著要使用这个信以为真的返回值时。

举例来说，返回一个月有多少天的函数有一个 bug；假设我们忘记十月份了：

```
(defun month-length (mon)
  (case mon
    ((jan mar may jul aug dec) 31)
    ((apr jun sept nov) 30)
    (feb (if (leap-year) 29 28))))
```

如果有另一个函数，企图想计算出一个月当中有几个礼拜，

```
(defun month-weeks (mon) (/ (month-length mon) 7.0))
```

则会发生下面的情形：

```
> (month-weeks 'oct)
Error: NIL is not a valud argument to /.
```

问题发生的原因是因为 `month-length` 在 `case` 找不到匹配 。当这个情形发生时， `case` 返回 `nil` 。然后 `month-weeks` ，认为获得了一个数字，将值传给 `/` ， `/` 就抱怨了。

在这里最起码 bug 与 bug 的临床表现是挨著发生的。这样的 bug 在它们相距很远时很难找到。要避免这个可能性，某些 Lisp 方言让跑完 `case` 或 `cond` 又没匹配的情形，产生一个错误。在 Common Lisp 里，在这种情况里可以做的是使用 `ecase` ，如 14.6 节所描述的。

重新命名 (Renaming)

在某些场合里（但不是全部场合），有一种特别狡猾的 bug ，起因于重新命名函数或变量，。举例来说，假设我们定义下列（低效的） 函数来找出双重嵌套列表的深度：

```
(defun depth (x)
  (if (atom x)
      1
      (1+ (apply #'max (mapcar #'depth x)))))
```

测试函数时，我们发现它给我们错误的答案（应该是 1）：

```
> (depth '((a)))
3
```

起初的 1 应该是 0 才对。如果我们修好这个错误，并给这个函数一个较不模糊的名称：

```
(defun nesting-depth (x)
  (if (atom x)
      0
      (1+ (apply #'max (mapcar #'depth x)))))
```

当我们再测试上面的例子，它返回同样的结果：

```
> (nesting-depth '((a)))
3
```

我们不是修好这个函数了吗？没错，但答案不是来自我们修好的代码。我们忘记也改掉递归调用中的名称。在递归用例里，我们的新函数仍调用先前的 depth ，这当然是不对的。

作为选择性参数的关键字 (Keywords as Optional Parameters)

若函数同时接受关键字与选择性参数，这通常是个错误，无心地提供了关键字作为选择性参数。举例来说，函数 read-from-string 有着下列的参数列表：

```
(read-from-string string &optional eof-error eof-value
                         &key start end preserve-whitespace)
```

这样一个函数你需要依序提供值，给所有的选择性参数，再来才是关键字参数。如果你忘记了选择性参数，看看下面这个例子，

```
> (read-from-string "abcd" :start 2)
ABCD
4
```

则 :start 与 2 会成为前两个选择性参数的值。若我们想要 read 从第二个字符开始读取，我们应该这么说：

```
> (read-from-string "abcd" nil nil :start 2)
CD
4
```

错误声明 (Misdeclarations)

第十三章解释了如何给变量及数据结构做类型声明。通过给变量做类型声明，你保证变

量只会包含某种类型的值。当产生代码时，Lisp 编译器会依赖这个假定。举例来说，这个函数的两个参数都声明为 `double-floats`，

```
(defun df* (a b)
  (declare (double-float a b))
  (* a b))
```

因此编译器在产生代码时，被授权直接将浮点乘法直接硬连接 (hard-wire)到代码里。

如果调用 `df*` 的参数不是声明的类型时，可能会捕捉一个错误，或单纯地返回垃圾。在某个实现里，如果我们传入两个定长数，我们获得一个硬体中断：

```
> (df* 2 3)
Error: Interrupt.
```

如果获得这样严重的错误，通常是由于数值不是先前声明的类型。

警告 (Warnings)

有些时候 Lisp 会抱怨一下，但不会中断求值过程。许多这样的警告是错误的警钟。一种最常见的可能是由编译器所产生的，关于未宣告或未使用的变量。举例来说，在 66 页「译注: 6.4 节」，`map-int` 的第二个调用，有一个 `x` 变量没有使用到。如果想要编译器在每次编译程序时，停止通知你这些事，使用一个忽略声明：

```
(map-int #'(lambda (x)
             (declare (ignore x))
             (random 100))
         10)
```

附录 B：Lisp in Lisp

这个附录包含了 58 个最常用的 Common Lisp 操作符。因为如此多的 Lisp 是（或可以）用 Lisp 所写成，而由于 Lisp 程序（或可以）相当精简，这是一种方便解释语言的方式。

这个练习也证明了，概念上 Common Lisp 不像看起来那样庞大。许多 Common Lisp 操作符是有用的函式库；要写出所有其它的东西，你所需要的操作符相当少。在这个附录的这些定义只需要:

```
apply aref backquote block car cdr ceiling char= cons defmacro documentation eq
error expt fdefinition function floor gensym get-setf-expansion if imagpart
labels length multiple-value-bind nth-value quote realpart symbol-function
tagbody type-of typep = + - / < >
```

这里给出的代码作为一种解释 Common Lisp 的方式，而不是实现它的方式。在实际的实现上，这些操作符可以更高效，也会做更多的错误检查。为了方便参找，这些操作符本身按字母顺序排列。如果你真的想要这样定义 Lisp，每个宏的定义需要在任何调用它们的代码之前。

```
(defun -abs (n)
  (if (typep n 'complex)
      (sqrt (+ (expt (realpart n) 2) (expt (imagpart n) 2)))
      (if (< n 0) (- n) n)))
```

```
(defun -adjoin (obj lst &rest args)
  (if (apply #'member obj lst args) lst (cons obj lst)))
```

```
(defmacro -and (&rest args)
  (cond ((null args) t)
        ((cdr args)  `(if ,(car args) (-and ,@(cdr args))))
        (t           (car args))))
```

```
(defun -append (&optional first &rest rest)
  (if (null rest)
      first
      (nconc (copy-list first) (apply #'-append rest))))
```

```
(defun -atom (x) (not (consp x)))
```

```
(defun -butlast (lst &optional (n 1))
  (nreverse (nthcdr n (reverse lst))))
```

```
(defun -cadr (x) (car (cdr x)))
```

```
(defmacro -case (arg &rest clauses)
  (let ((g (gensym)))
    `(let ((,g ,arg))
       (cond ,@(mapcar #'(lambda (cl)
                           (let ((k (car cl)))
                             `(,(cond ((member k '(t otherwise))
                                       t)
                                      ((consp k)
                                       `(member ,g ',k))
                                      (t `(eql ,g ',k)))
                               (progn ,@(cdr cl)))))
                       clauses)))))
```

```
(defun -cddr (x) (cdr (cdr x)))
```

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

```
(defmacro -cond (&rest args)
  (if (null args)
      nil
      (let ((clause (car args)))
        (if (cdr clause)
            `(if ,(car clause)
                 (progn ,@(cdr clause))
                 (-cond ,@(cdr args)))
            `(or ,(car clause)
                 (-cond ,@(cdr args)))))))
```

```
(defun -consp (x) (typep x 'cons))
```

```
(defun -constantly (x) #'(lambda (&rest args) x))
```

```
(defun -copy-list (lst)
  (labels ((cl (x)
             (if (atom x)
                 x
                 (cons (car x)
                       (cl (cdr x))))))
    (cons (car lst)
          (cl (cdr lst)))))
```

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

```
(defmacro -defun (name parms &rest body)
  (multiple-value-bind (dec doc bod) (analyze-body body)
    `(progn
       (setf (fdefinition ',name)
             #'(lambda ,parms
                 ,@dec
                 (block ,(if (atom name) name (second name))
                   ,@bod))
             (documentation ',name 'function)
             ,doc)
       ',name)))
```

```
(defun analyze-body (body &optional dec doc)
  (let ((expr (car body)))
    (cond ((and (consp expr) (eq (car expr) 'declare))
           (analyze-body (cdr body) (cons expr dec) doc))
          ((and (stringp expr) (not doc) (cdr body))
           (if dec
               (values dec expr (cdr body))
               (analyze-body (cdr body) dec expr)))
          (t (values dec doc body)))))
```

这个定义不完全正确，参见 `let`

```
(defmacro -do (binds (test &rest result) &rest body)
  (let ((fn (gensym)))
    `(block nil
       (labels ((,fn ,(mapcar #'car binds)
                  (cond (,test ,@result)
                        (t (tagbody ,@body)
                           (,fn ,@(mapcar #'third binds))))))
         (,fn ,@(mapcar #'second binds))))))
```

```
(defmacro -dolist ((var lst &optional result) &rest body)
  (let ((g (gensym)))
    `(do ((,g ,lst (cdr ,g)))
         ((atom ,g) (let ((,var nil)) ,result))
       (let ((,var (car ,g)))
         ,@body))))
```

```
(defun -eql (x y)
  (typecase x
    (character (and (typep y 'character) (char= x y)))
    (number    (and (eq (type-of x) (type-of y))
                    (= x y)))
    (t         (eq x y))))
```

```
(defun -evenp (x)
  (typecase x
    (integer (= 0 (mod x 2)))
```

```
    (t        (error "non-integer argument"))))
```

```
(defun -funcall (fn &rest args) (apply fn args))
```

```
(defun -identity (x) x)
```

这个定义不完全正确：表达式 `(let ((&key 1) (&optional 2)))` 是合法的，但它产生的表达式不合法。

```
(defmacro -let (parms &rest body)
  `((lambda ,(mapcar #'(lambda (x)
                         (if (atom x) x (car x)))
                     parms)
      ,@body)
    ,@(mapcar #'(lambda (x)
                  (if (atom x) nil (cadr x)))
              parms)))
```

```
(defun -list (&rest elts) (copy-list elts))
```

```
(defun -listp (x) (or (consp x) (null x)))
```

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

```
(defun -mapcar (fn &rest lsts)
  (cond ((member nil lsts) nil)
        ((null (cdr lsts))
         (let ((lst (car lsts)))
           (cons (funcall fn (car lst))
                 (-mapcar fn (cdr lst)))))
        (t
         (cons (apply fn (-mapcar #'car lsts))
               (apply #'-mapcar fn
                      (-mapcar #'cdr lsts))))))
```

```
(defun -member (x lst &key test test-not key)
  (let ((fn (or test
                (if test-not
                    (complement test-not))
                #'eql)))
    (member-if #'(lambda (y)
                   (funcall fn x y))
               lst
               :key key)))
```

```
(defun -member-if (fn lst &key (key #'identity))
  (cond ((atom lst) nil)
```

```
        ((funcall fn (funcall key (car lst))) lst)
        (t (-member-if fn (cdr lst) :key key))))
```

```
(defun -mod (n m)
  (nth-value 1 (floor n m)))
```

```
(defun -nconc (&optional lst &rest rest)
  (if rest
      (let ((rest-conc (apply #'-nconc rest)))
        (if (consp lst)
            (progn (setf (cdr (last lst)) rest-conc)
                   lst)
            rest-conc))
      lst))
```

```
(defun -not (x) (eq x nil))
(defun -nreverse (seq)
  (labels ((nrl (lst)
             (let ((prev nil))
               (do ()
                   ((null lst) prev)
                 (psetf (cdr lst) prev
                        prev      lst
                        lst       (cdr lst)))))
           (nrv (vec)
             (let* ((len (length vec))
                    (ilimit (truncate (/ len 2))))
               (do ((i 0 (1+ i))
                    (j (1- len) (1- j)))
                   ((>= i ilimit) vec)
                 (rotatef (aref vec i) (aref vec j))))))
    (if (typep seq 'vector)
        (nrv seq)
        (nrl seq))))
```

```
(defun -null (x) (eq x nil))
```

```
(defmacro -or (&optional first &rest rest)
  (if (null rest)
      first
      (let ((g (gensym)))
        `(let ((,g ,first))
           (if ,g
               ,g
               (-or ,@rest))))))
```

这两个 Common Lisp 没有，但这里有几的定义会需要用到。

```
(defun pair (lst)
  (if (null lst)
```

```
      nil
      (cons (cons (car lst) (cadr lst))
            (pair (cddr lst)))))

(defun -pairlis (keys vals &optional alist)
  (unless (= (length keys) (length vals))
    (error "mismatched lengths"))
  (nconc (mapcar #'cons keys vals) alist))
```

```
(defmacro -pop (place)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-expansion place)
    (let ((g (gensym)))
      `(let* (,@(mapcar #'list vars forms)
              (,g ,access)
              (,(car var) (cdr ,g)))
         (prog1 (car ,g)
                ,set))))))
```

```
(defmacro -prog1 (arg1 &rest args)
  (let ((g (gensym)))
    `(let ((,g ,arg1))
       ,@args
       ,g)))
```

```
(defmacro -prog2 (arg1 arg2 &rest args)
  (let ((g (gensym)))
    `(let ((,g (progn ,arg1 ,arg2)))
       ,@args
       ,g)))
```

```
(defmacro -progn (&rest args) `(let nil ,@args))
```

```
(defmacro -psetf (&rest args)
  (unless (evenp (length args))
    (error "odd number of arguments"))
  (let* ((pairs (pair args))
         (syms (mapcar #'(lambda (x) (gensym))
                       pairs)))
    `(let ,(mapcar #'list
                   syms
                   (mapcar #'cdr pairs))
       (setf ,@(mapcan #'list
                       (mapcar #'car pairs)
                       syms))))))
```

```
(defmacro -push (obj place)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-expansion place)
    (let ((g (gensym)))
      `(let* ((,g ,obj)
```

```
              ,@(mapcar #'list vars forms)
              (,(car var) (cons ,g ,access)))
         ,set))))
```

```
(defun -rem (n m)
  (nth-value 1 (truncate n m)))

(defmacro -rotatef (&rest args)
  `(psetf ,@(mapcan #'list
                    args
                    (append (cdr args)
                            (list (car args))))))
```

```
(defun -second (x) (cadr x))

(defmacro -setf (&rest args)
  (if (null args)
      nil
      `(setf2 ,@args)))
```

```
(defmacro setf2 (place val &rest args)
  (multiple-value-bind (vars forms var set)
                       (get-setf-expansion place)
    `(progn
       (let* (,@(mapcar #'list vars forms)
              (,(car var) ,val))
         ,set)
       ,@(if args `((setf2 ,@args)) nil))))
```

```
(defun -signum (n)
  (if (zerop n) 0 (/ n (abs n))))
```

```
(defun -stringp (x) (typep x 'string))
```

```
(defun -tailp (x y)
  (or (eql x y)
      (and (consp y) (-tailp x (cdr y)))))
```

```
(defun -third (x) (car (cdr (cdr x))))
```

```
(defun -truncate (n &optional (d 1))
  (if (> n 0) (floor n d) (ceiling n d)))
```

```
(defmacro -typecase (arg &rest clauses)
  (let ((g (gensym)))
    `(let ((,g ,arg))
       (cond ,@(mapcar #'(lambda (cl)
                           `((typep ,g ',(car cl))
                             (progn ,@(cdr cl))))
```

```
                        clauses)))))
```

```
(defmacro -unless (arg &rest body)
  `(if (not ,arg)
       (progn ,@body)))
```

```
(defmacro -when (arg &rest body)
  `(if ,arg (progn ,@body)))
```

```
(defun -1+ (x) (+ x 1))
```

```
(defun -1- (x) (- x 1))
```

```
(defun ->= (first &rest rest)
  (or (null rest)
      (and (or (> first (car rest)) (= first (car rest)))
           (apply #'->= rest))))
```

附录 C：Common Lisp 的改变

目前的 ANSI Common Lisp 与 1984 年由 Guy Steele 一书 *Common Lisp: the Language* 所定义的 Common Lisp 有着本质上的不同。同时也与 1990 年该书的第二版大不相同，虽然差别比较小。本附录总结了重大的改变。1990年之后的改变独自列在最后一节。

附录 D：语言参考手册

备注

本节既是备注亦作为参考文献。所有列于此的书籍与论文皆值得阅读。

译注：备注后面跟随的数字即书中的页码

备注 viii (Notes viii)

Steele, Guy L., Jr. [http://en.wikipedia.org/wiki/Guy_L._Steele,_Jr.], Scott E. Fahlman [http://en.wikipedia.org/wiki/Scott_Fahlman], Richard P. Gabriel [http://en.wikipedia.org/wiki/Richard_P._Gabriel], David A. Moon [http://en.wikipedia.org/wiki/David_Moon], Daniel L. Weinreb [http://en.wikipedia.org/wiki/Daniel_Weinreb] , Daniel G. Bobrow [http://en.wikipedia.org/wiki/Daniel_G._Bobrow], Linda G. DeMichiel [http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/d/DeMichiel:Linda_G=.html], Sonya E. Keene [http://www.amazon.com/Sonya-E.-Keene/e/B001ITVL6O], Gregor Kiczales [http://en.wikipedia.org/wiki/Gregor_Kiczales], Crispin Perdue [http://perdues.com/CrisPerdueResume.html], Kent M. Pitman [http://en.wikipedia.org/wiki/Kent_Pitman], Richard C. Waters [http://www.rcwaters.org/], 以及 John L White。Common Lisp: the Language, 2nd Edition. [http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html] Digital Press, Bedford (MA), 1990.

备注 1 (Notes 1)

McCarthy, John. [http://en.wikipedia.org/wiki/John_McCarthy_(computer_scientist)] Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I. [http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.4527&rep=rep1&type=pdf] CACM, 3:4 (April 1960), pp. 184-195.

McCarthy, John. [http://en.wikipedia.org/wiki/John_McCarthy_(computer_scientist)] History of Lisp. [http://www-formal.stanford.edu/jmc/history/lisp/lisp.html] In Wexelblat, Richard L. [http://en.wikipedia.org/wiki/Richard_Wexelblat] (Ed.) Histroy of Programming Languages. [http://cs305.com/book/programming_languages/Conf-01/HOPLII/frontmatter.pdf] Academic Press, New York, 1981, pp. 173-197.

备注 3 (Notes 3)

Brooks, Frederick P [http://en.wikipedia.org/wiki/Frederick_Brooks]. The Mythical Man-Month [http://www.amazon.com/Mythical-Man-Month-Software-Engineering-Anniversary/dp/0201835959]. Addison-Wesley, Reading (MA), 1975, p. 16.

Rapid prototyping is not just a way to write programs faster or better. It is a way to write programs that otherwise might not get written at all. Even the most ambitious people shrink from big undertakings. It's easier to start something if one can convince oneself (however speciously) that it won't be too much work. That's why so many big things have begun as small things. Rapid prototyping lets us start small.

备注 4 (Notes 4)

同上， 第 i 页。

备注 5 (Notes 5)

Murray, Peter and Linda. The Art of the Renaissance [http://www.amazon.com/Art-Renaissance-World/dp/0500200084]. Thames and Hudson, London, 1963, p.85.

备注 5-2 (Notes 5-2)

Janson, W.J. History of Art [http://www.amazon.com/History-Art-H-W-Janson/dp/0810934019/ref=sr_1_1?s=books&ie=UTF8&qid=1365042074&sr=1-1&keywords=History+of+Art], 3rd Edition. Abrams, New York, 1986, p. 374.

The analogy applies, of course, only to paintings done on panels and later on canvases. Well-paintings continued to be done in fresco. Nor do I mean to suggest that painting styles were driven by technological change; the opposite seems more nearly true.

备注 12 (Notes 12)

`car` 与 `cdr` 的名字来自最早的 Lisp 实现里，列表内部的表示法：car 代表“寄存器位址部分的内容”、cdr 代表“寄存器递减部分的内容”。

备注 17 (Notes 17)

对递归概念有困扰的读者，可以查阅下列的书籍：

Touretzky, David S. Common Lisp: A Gentle Introduction to Symbolic Computation

[http://www.amazon.com/Common-Lisp-Introduction-Computation-Benjamin-Cummings/dp/B008T1B8WQ/ref=sr_1_3?s=books&ie=UTF8&qid=1365042108&sr=1-3&keywords=A+Gentle+Introduction+to+Symbolic+Computation]. Benjamin/Cummings, Redwood City (CA), 1990, Chapter 8.

Friedman, Daniel P., and Matthias Felleisen. The Little Lisper. MIT Press, Cambridge, 1987.

譯註：這本書有再版，可在這裡 [http://www.amazon.com/Common-LISP-Introduction-Symbolic-Computation/dp/0486498204/ref=sr_1_1?s=books&ie=UTF8&qid=1365042108&sr=1-1&keywords=A+Gentle+Introduction+to+Symbolic+Computation]找到。

备注 26 (Notes 26)

In ANSI Common Lisp there is also a `lambda` macro that allows you to write `(lambda (x) x)` for `#'(lambda (x) x)` . Since the use of this macro obscures the symmetry between lambda expressions and symbolic function names (where you still have to use sharp-quote), it yields a specious sort of elegance at best.

备注 28 (Notes 28)

Gabriel, Richard P. Lisp Good News, Bad News, How to Win Big [http://www.dreamsongs.com/Files/LispGoodNewsBadNews.pdf] *AI Expert*, June 1991, p.34.

备注 46 (Notes 46)

Another thing to be aware of when using sort: it does not guarantee to preserve the order of elements judged equal by the comparison function. For example, if you sort `(2 1 1.0)` by `<` , a valid Common Lisp implementation could return either `(1 1.0 2)` or `(1.0 1 2)` . To preserve as much as possible of the original order, use instead the slower `stable-sort` (also destructive), which could only return the first value.

备注 61 (Notes 61)

A lot has been said about the benefits of comments, and little or nothing about their cost. But they do have a cost. Good code, like good prose, comes from constant rewriting. To evolve, code must be malleable and compact. Interlinear comments make programs stiff and diffuse, and so inhibit the evolution of what they describe.

备注 62 (Notes 62)

Though most implementations use the ASCII character set, the only ordering that Common Lisp guarantees for characters is as follows: the 26 lowercase letters are in alphabetically ascending order, as are the uppercase letters, and the digits from 0 to 9.

备注 76 (Notes 76)

The standard way to implement a priority queue is to use a structure called a heap. See: Sedgewick, Robert. Algorithms [http://www.amazon.com/Algorithms-4th-Robert-Sedgewick/dp/032157351X/ref=sr_1_1?s=books&ie=UTF8&qid=1365042619&sr=1-1&keywords=algorithms+sedgewick]. Addison-Wesley, Reading (MA), 1988.

备注 81 (Notes 81)

The definition of progn sounds a lot like the evaluation rule for Common Lisp function calls (page 9). Though `progn` is a special operator, we could define a similar function:

```
(defun our-progn (ftrest args)
  (car (last args)))
```

This would be horribly inefficient, but functionally equivalent to the real `progn` if the last argument returned exactly one value.

备注 84 (Notes 84)

The analogy to a lambda expression breaks down if the variable names are symbols that have special meanings in a parameter list. For example,

```
(let ((&key 1) (&optional 2)))
```

is correct, but the corresponding lambda expression

```
((lambda (ftkey ftoptional)) 1 2)
```

is not. The same problem arises if you try to define do in terms of `labels` . Thanks to David Kuznick for pointing this out.

备注 89 (Notes 89)

Steele, Guy L., Jr., and Richard P. Gabriel. The Evolution of Lisp [http://www.dreamsongs.com/Files/HOPL2-Uncut.pdf]. ACM SIGPLANNotices 28:3 (March 1993). The example in the quoted passage was translated from Scheme into Common Lisp.

备注 91 (Notes 91)

To make the time look the way people expect, you would want to ensure that minutes and seconds are represented with two digits, as in:

```
(defun get-time-string ()
  (multiple-value-bind (s m h) (get-decoded-time)
    (format nil "~A:~2,,,'0@A:~2,,,'0@A" h m s)))
```

备注 94 (Notes 94)

In a letter of March 18 (old style) 1751, Chesterfield writes:

“It was notorious, that the Julian Calendar was erroneous, and had overcharged the solar year with eleven days. Pope Gregory the Thirteenth corrected this error [in 1582]; his reformed calendar was immediately received by all the Catholic powers of Europe, and afterwards adopted by all the Protestant ones, except Russia, Sweden, and England. It was not, in my opinion, very honourable for England to remain in a gross and avowed error, especially in such company; the inconveniency of it was likewise felt by all those who had foreign correspondences, whether political or mercantile. I determined, therefore, to attempt the reformation; I consulted the best lawyers, and the most skillful astronomers, and we cooked up a bill for that purpose. But then my difficulty began; I was to bring in this bill, which was necessarily composed of law jargon and astronomical calculations, to both of which I am an utter stranger. However, it was absolutely necessary to make the House of Lords think that I knew something of the matter; and also to make them believe that they knew something of it themselves, which they do not. For my own part, I could just as soon have talked Celtic or Sclavonian to them, as astronomy, and they would have understood me full as well; so I resolved to do better than speak to the purpose, and to please instead of informing them. I gave them, therefore, only an historical account of calendars, from the Egyptian down to the Gregorian, amusing them now and then with little episodes; but I was particularly attentive to the choice of my words, to the harmony and roundness of my periods, to my elocution, to my action. This succeeded, and ever will succeed; they thought I informed them, because I pleased them; and many of them said I had made the whole very clear to them; when, God knows, I had not even attempted it.”

See: Roberts, David (Ed.) Lord Chesterfield’s Letters

[http://books.google.com.tw/books/about/Lord_Chesterfield_s_Letters.html?id=nFZP1WQ6XDoC&redir_esc=y]. Oxford University Press, Oxford, 1992.

备注 95 (Notes 95)

In Common Lisp, a universal time is an integer representing the number of seconds since the beginning of 1900. The functions `encode-universal-time` and `decode-universal-time` translate dates into and out of this format. So for dates after 1900, there is a simpler way to do date arithmetic in Common Lisp:

```
(defun num->date (n)
  (multiple-value-bind (ig no re d m y)
                       (decode-universal-time n)
    (values d m y)))

(defun date->num (d m y)
  (encode-universal-time 1 0 0 d m y))

(defun date+ (d m y n)
  (num->date (+ (date->num d m y)
                (* 60 60 24 n))))
```

Besides the range limit, this approach has the disadvantage that dates tend not to be fixnums.

备注 100 (Notes 100)

Although a call to `setf` can usually be understood as a reference to a particular place, the underlying machinery is more general. Suppose that a marble is a structure with a single field called color:

```
(defstruct marble
  color)
```

The following function takes a list of marbles and returns their color, if they all have the same color, or n i l if they have different colors:

```
(defun uniform-color (lst)
  (let ((c (marble-color (car lst))))
    (dolist (m (cdr lst))
      (unless (eql (marble-color m) c)
        (return nil)))
    c))
```

Although `uniform-color` does not refer to a particular place, it is both reasonable and possible

to have a call to it as the first argument to `setf` . Having defined

```
(defun (setf uniform-color) (val 1st)
  (dolist (m 1st)
    (setf (marble-color m) val)))
```

we can say

```
(setf (uniform-color *marbles*) 'red)
```

to make the color of each element of `*marbles*` be red.

备注 100-2 (Notes 100-2)

In older Common Lisp implementations, you have to use `defsetf` to define how a call should be treated when it appears as the first argument to setf. Be careful when translating, because the parameter representing the new value comes last in the definition of a function whose name is given as the second argument to `defsetf` . That is, the call

```
(defun (setf primo) (val 1st) (setf (car 1st) val))
```

is equivalent to

```
(defsetf primo set-primo)
```

```
(defun set-primo (1st val) (setf (car 1st) val))
```

备注 106 (Notes 106)

C, for example, lets you pass a pointer to a function, but there's less you can pass in a function (because C doesn't have closures) and less the recipient can do with it (because C has no equivalent of apply). What's more, you are in principle supposed to declare the type of the return value of the function you pass a pointer to. How, then, could you write `map-int` or `filter` , which work for functions that return anything? You couldn't, really. You would have to suppress the type-checking of arguments and return values, which is dangerous, and even so would probably only be practical for 32-bit values.

备注 109 (Notes 109)

For many examples of the versatility of closures, see: Abelson, Harold, and Gerald Jay

Sussman, with Julie Sussman. Structure and Interpretation of Computer Programs [http://mitpress.mit.edu/sicp/]. MIT Press, Cambridge, 1985.

备注 109-2 (Notes 109-2)

For more information about Dylan, see: Shalit, Andrew, with Kim Barrett, David Moon, Orca Starbuck, and Steve Strassmann. Dylan Interim Reference Manual [http://jim.studt.net/dirm/interim-contents.html]. Apple Computer, 1994.

At the time of printing this document was accessible from several sites, including http://www.harlequin.com and http://www.apple.com. Scheme is a very small, clean dialect of Lisp. It was invented by Guy L. Steele Jr. and Gerald J. Sussman in 1975, and is currently defined by: Clinger, William, and Jonathan A. Rees (Eds.) \(Revised^4\) Report on the Algorithmic Language Scheme. 1991.

This report, and various implementations of Scheme, were at the time of printing available by anonymous FTP from swiss-ftp.ai.mit.edu:pub.

There are two especially good textbooks that use Scheme—Structure and Interpretation (see preceding note) and: Springer, George and Daniel P. Friedman. Scheme and the Art of Programming [http://www.amazon.com/Scheme-Art-Programming-George-Springer/dp/0262192888]. MIT Press, Cambridge, 1989.

备注 112 (Notes 112)

The most horrible Lisp bugs may be those involving dynamic scope. Such errors almost never occur in Common Lisp, which has lexical scope by default. But since so many of the Lisps used as extension languages still have dynamic scope, practicing Lisp programmers should be aware of its perils.

One bug that can arise with dynamic scope is similar in spirit to variable capture (page 166). You pass one function as an argument to another. The function passed as an argument refers to some variable. But within the function that calls it, the variable has a new and unexpected value.

Suppose, for example, that we wrote a restricted version of mapcar as follows:

```
(defun our-mapcar (fn x)
  (if (null x)
      nil (cons (funcall fn (car x))
                (our-mapcar fn (cdr x)))))
```

Then suppose that we used this function in another function, `add-to-all` , that would take a number and add it to every element of a list:

```
(defun add-to-all (1st x)
  (our-mapcar #'(lambda (num) (+ num x))
              1st))
```

In Common Lisp this code works fine, but in a Lisp with dynamic scope it would generate an error. The function passed as an argument to `our-mapcar` refers to `x` . At the point where we send this function to `our-mapcar` , `x` would be the number given as the second argument to `add-to-all` . But where the function will be called, within `our-mapcar` , `x` would be something else: the list passed as the second argument to `our-mapcar` . We would get an error when this list was passed as the second argument to `+` .

備注 123 (Notes 123)

Newer implementations of Common Lisp include avariable `*read-eval*` that can be used to turn off the `#` . read-macro. When calling `read-from-string` on user input, it is wise to bind `*read-eval*` to `nil` . Otherwise the user could cause side-effects by using `#` . in the input.

備注 125 (Notes 125)

There are a number of ingenious algorithms for fast string-matching, but string-matching in text files is one of the cases where the brute-force approach is still reasonably fast. For more on string-matching algorithms, see: Sedgewick, Robert. Algorithms [http://www.amazon.com/Algorithms-4th-Robert-Sedgewick/dp/032157351X/ref=sr_1_1?s=books&ie=UTF8&qid=1365042619&sr=1-1&keywords=algorithms+sedgewick]. Addison-Wesley, Reading (MA), 1988.

備注 141 (Notes 141)

In 1984 CommonLisp, reduce did not take a `:key` argument, so `random-next` would be defined:

```
(defun random-next (prev)
  (let* ((choices (gethash prev *words*))
         (i (random (let ((x 0))
                      (dolist (c choices)
                        (incf x (cdr c)))
                      x))))
    (dolist (pair choices)
      (if (minusp (decf i (cdr pair)))
```

```
        (return (car pair))))))
```

備注 141-2 (Notes 141-2)

In 1989, a program like Henley was used to simulate netnews postings by well-known flamers. The fake postings fooled a significant number of readers. Like all good hoaxes, this one had an underlying point. What did it say about the content of the original flames, or the attention with which they were read, that randomly generated postings could be mistaken for the real thing?

One of the most valuable contributions of artificial intelligence research has been to teach us which tasks are really difficult. Some tasks turn out to be trivial, and some almost impossible. If artificial intelligence is concerned with the latter, the study of the former might be called artificial stupidity. A silly name, perhaps, but this field has real promise—it promises to yield programs that play a role like that of control experiments.

Speaking with the appearance of meaning is one of the tasks that turn out to be surprisingly easy. People's predisposition to find meaning is so strong that they tend to overshoot the mark. So if a speaker takes care to give his sentences a certain kind of superficial coherence, and his audience are sufficiently credulous, they will make sense of what he says.

This fact is probably as old as human history. But now we can give examples of genuinely random text for comparison. And if our randomly generated productions are difficult to distinguish from the real thing, might that not set people to thinking?

The program shown in Chapter 8 is about as simple as such a program could be, and that is already enough to generate "poetry" that many people (try it on your friends) will believe was written by a human being. With programs that work on the same principle as this one, but which model text as more than a simple stream of words, it will be possible to generate random text that has even more of the trappings of meaning.

For a discussion of randomly generated poetry as a legitimate literary form, see: Low, Jackson M. Poetry, Chance, Silence, Etc. In Hall, Donald (Ed.) Claims for Poetry. University of Michigan Press, Ann Arbor, 1982. You bet.

Thanks to the Online Book Initiative, ASCII versions of many classics are available online. At the time of printing, they could be obtained by anonymous FTP from ftp.std.com:obi.

See also the Emacs Dissociated Press feature, which uses an equivalent algorithm to scramble a buffer.

備注 150 (Notes 150)

下面这个函数会显示在一个给定实现中，16　　个用来标示浮点表示法的限制的全局常量：

```
(defun float-limits ()
  (dolist (m '(most least))
    (dolist (s '(positive negative))
      (dolist (f '(short single double long))
        (let ((n (intern (string-upcase
                          (format nil "~A-~A-~A-float"
                                  m  s  f)))))
          (format t "~30A ~A ~%" n (symbol-value n)))))))
```

备注 164 (Notes 164)

快速排序演算法 [http://zh.wikipedia.org/zh-cn/%E5%BF%AB%E9%80%9F%E6%8E%92%E5%BA%8F]由霍尔 [http://zh.wikipedia.org/zh-cn/%E6%9D%B1%E5%B0%BC%C2%B7%E9%9C%8D%E7%88%BE]于 1962 年发表，并被描述在 Knuth, D. E. *Sorting and Searching*. Addison-Wesley, Reading (MA), 1973.一书中。

备注 173 (Notes 173)

Foderaro, John K. Introduction to the Special Lisp Section. CACM 34:9 (Setember 1991), p.27 [http://www.informatik.uni-trier.de/~ley/db/journals/cacm/cacm34.html]

备注 176 (Notes 176)

关于 CLOS 更详细的信息，参考下列书目：

Keene, Sonya E. Object Oriented Programming in Common Lisp [http://en.wikipedia.org/wiki/Object-Oriented_Programming_in_Common_Lisp:_A_Programmer's_Guide_to_CLOS] , Addison-Wesley, Reading (MA), 1989

Kiczales, Gregor, Jim des Rivieres, and Daniel G. Bobrow. The Art of the Metaobject Protocol [http://en.wikipedia.org/wiki/The_Art_of_the_Metaobject_Protocol] MIT Press, Cambridge, 1991

备注 178 (Notes 178)

让我们再回放刚刚的句子一次：我们甚至不需要看程序中其他的代码一眼，就可以完成种种的改动。这个想法或许对某些读者听起来担忧地熟悉。这是写出面条式代码[http://zh.wikipedia.org/wiki/%E9%9D%A2%E6%9D%A1%E5%BC%8F%E4%BB%A3%E7%A的食谱。

面向对象模型使得通过一点一点的来构造程序变得简单。但这通常意味著，在实践上它提供了一种有结构的方法来写出面条式代码。这不一定是坏事，但也不会是好事。

很多现实世界中的代码是面条式代码，这也许不能很快改变。针对那些终将成为面条式代码的程序来说，面向对象模型是好的：它们最起码会是有结构的面条。但针对那些也许可以避免误入崎途的程序来说，面向对象抽象只是更加危险的，而不是有用的。

备注 183 (Notes 183)

When an instance would inherit a slot with the same name from several of its superclasses, the instance inherits a single slot that combines the properties of the slots in the superclasses. The way combination is done varies from property to property:

1. The `:allocation` , `:initform` (if any), and `:documentation` (if any), will be those of the most specific classes.
2. The `:initargs` will be the union of the `:initargs` of all the superclasses. So will the `:accessors` , `:readers` , and `:writers` , effectively.
3. The `:type` will be the intersection of the `:types` of all the superclasses.

备注 191 (Notes 191)

You can avoid explicitly uninterning the names of slots that you want to be encapsulated by using uninterned symbols as the names to start with:

```
(progn
  (defclass counter () ((#1=#:state :initform 0)))

  (defmethod increment ((c counter))
    (incf (slot-value c '#1#)))

  (defmethod clear ((c counter))
    (setf (slot-value c '#1#) 0)))
```

The `progn` here is a no-op; it is used to ensure that all the references to the uninterned symbol occur within the same expression. If this were inconvenient, you could use the following read-macro instead:

```
(defvar *symtab* (make-hash-table :test #'equal))

(defun pseudo-intern (name)
  (or (gethash name *symtab*)
      (setf (gethash name *symtab*) (gensym))))

(set-dispatch-macro-character #\# #\[
  #'(lambda (stream char1 char2)
      (do ((acc nil (cons char acc))
           (char (read-char stream) (read-char stream)))
          ((eql char #\]) (pseudo-intern acc)))))
```

Then it would be possible to say just:

```
(defclass counter () ((#[state] :initform 0)))

(defmethod increment ((c counter))
  (incf (slot-value c '#[state])))

(defmethod clear ((c counter))
  (setf (slot-value c '#[state]) 0))
```

备注 204 (Notes 204)

下面这个宏将新元素推入二叉搜索树：

```
(defmacro bst-push (obj bst <)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-expansion bst)
    (let ((g (gensym)))
      `(let* ((,g ,obj)
              ,@(mapcar #'list vars forms)
              (,(car var) (bst-insert! ,g ,access ,<)))
         ,set))))
```

备注 213 (Notes 213)

Knuth, Donald E. Structured Programming with goto Statements. [http://sbel.wisc.edu/Courses/ME964/Literature/knuthProgramming1974.pdf] *Computing Surveys* , 6:4 (December 1974), pp. 261-301

备注 214 (Notes 214)

Knuth, Donald E. Computer Programming as an Art [http://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&cad=rja&ved=0CC4QFjAB&url=http%3A%2F%2F

ife4DB4BR2CPORBQ] *In ACM Turing Award Lectures: The First Twenty Years*. ACM Press, 1987

This paper and the preceding one are reprinted in: Knuth, Donald E. Literate Programming. CSLI Lecture Notes #27, Stanford University Center for the Study of Language and Information, Palo Alto, 1992.

备注 216 (Notes 216)

Steele, Guy L., Jr. Debunking the "Expensive Procedure Call" Myth or, Procedural Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO. Proceedings of the National Conference of the ACM, 1977, p. 157.

Tail-recursion optimization should mean that the compiler will generate the same code for a tail-recursive function as it would for the equivalent `do`. The unfortunate reality, at least at the time of printing, is that many compilers generate slightly faster code for `do`s.

备注 217 (Notes 217)

For some examples of calls to disassemble on various processors, see: Norvig, Peter. Paradigms ofArtificial Intelligence Programming: Case Studies in Common Lisp. Morgan Kaufmann, San Mateo (CA), 1992.

备注 218 (Notes 218)

A lot of the increased popularity of object-oriented programming is more specifically the increased popularity of C++, and this in turn has a lot to do with typing. C++ gives you something that seems like a miracle in the conceptual world of C: the ability to define operators that work for different types of arguments. But you don't need an object-oriented language to do this—all you need is run-time typing. And indeed, if you look at the way people use C++, the class hierarchies tend to be flat. C++ has become so popular not because people need to write programs in terms of classes and methods, but because people need a way around the restrictions imposed by C's approach to typing.

备注 219 (Notes 219)

Macros can make declarations easier. The following macro expects a type name and an expression (probably numeric), and expands the expression so that all arguments, and all intermediate results, are declared to be of that type. If you wanted to ensure that an expression e

was evaluated using only fixnum arithmetic, you could say `(with-type fixnum e)`.

```
(defmacro with-type (type expr)
  `(the ,type ,(if (atom expr)
                   expr
                 (expand-call type (binarize expr)))))

(defun expand-call (type expr)
  `(,(car expr) ,@(mapcar #'(lambda (a)
                              `(with-type ,type ,a))
                          (cdr expr))))

(defun binarize (expr)
  (if (and (nthcdr 3 expr)
           (member (car expr) '(+ - * /)))
      (destructuring-bind (op a1 a2 . rest) expr
        (binarize `(,op (,op ,a1 ,a2) ,@rest)))
    expr))
```

The call to binarize ensures that no arithmetic operator is called with more than two arguments. As the Lucid reference manual points out, a call like

```
(the fixnum (+ (the fixnum a)
               (the fixnum b)
               (the fixnum c)))
```

still cannot be compiled into fixnum additions, because the intermediate results (e.g. a + b) might not be fixnums.

Using `with-type`, we could duplicate the fully declared version of `poly` on page 219 with:

```
(defun poly (a b x)
  (with-type fixnum (+ (* a (expt x 2)) (* b x))))
```

If you wanted to do a lot of fixnum arithmetic, you might even want to define a read-macro that would expand into a `(with-type fixnum ...)`.

备注 224 (Notes 224)

在许多 Unix 系统里，`/usr/dict/words` 是个合适的单词文件。

备注 226 (Notes 229)

T is a dialect of Scheme with many useful additions, including support for pools. For more on T,

see: Rees, Jonathan A., Norman I. Adams, and James R. Meehan. The T Manual, 5th Edition. Yale University Computer Science Department, New Haven, 1988.

The T manual, and T itself, were at the time of printing available by anonymous FTP from hing.lcs.mit.edu:pub/t3.1 .

备注 229 (Notes 229)

The difference between specifications and programs is a difference in degree, not a difference in kind. Once we realize this, it seems strange to require that one write specifications for a program before beginning to implement it. If the program has to be written in a low-level language, then it would be reasonable to require that it be described in high-level terms first. But as the programming language becomes more abstract, the need for specifications begins to evaporate. Or rather, the implementation and the specifications can become the same thing.

If the high-level program is going to be re-implemented in a lower-level language, it starts to look even more like specifications. What Section 13.7 is saying, in other words, is that the specifications for C programs could be written in Lisp.

备注 230 (Notes 230)

Benvenuto Cellini's story of the casting of his Perseus is probably the most famous (and the funniest) account of traditional bronze-casting: Cellini, Benvenuto. Autobiography. Translated by George Bull, Penguin Books, Harmondsworth, 1956.

备注 239 (Notes 239)

Even experienced Lisp hackers find packages confusing. Is it because packages are gross, or because we are not used to thinking about what happens at read-time?

There is a similar kind of uncertainty about def macro, and there it does seem that the difficulty is in the mind of the beholder. A good deal of work has gone into finding a more abstract alternative to def macro. But def macro is only gross if you approach it with the preconception (common enough) that defining a macro is like defining a function. Then it seems shocking that you suddenly have to worry about variable capture. When you think of macros as what they are, transformations on source code, then dealing with variable capture is no more of a problem than dealing with division by zero at run-time.

So perhaps packages will turn out to be a reasonable way of providing modularity. It is prima facie evidence on their side that they resemble the techniques that programmers naturally use in

the absence of a formal module system.

备注 242 (Notes 242)

It might be argued that `loop` is more general, and that we should not define many operators to do what we can do with one. But it's only in a very legalistic sense that loop is one operator. In that sense, `eval` is one operator too. Judged by the conceptual burden it places on the user, `loop` is at least as many operators as it has clauses. What's more, these operators are not available separately, like real Lisp operators: you can't break off a piece of loop and pass it as an argument to another function, as you could `map-int` .

备注 248 (Notes 248)

关于更深入讲述逻辑推论的资料，参见：Stuart Russell [http://www.cs.berkeley.edu/~russell/] 及 Peter Norvig [http://www.norvig.com/] 所著的 Artificial Intelligence: A Modern Approach [http://aima.cs.berkeley.edu/]。

备注 273 (Notes 273)

Because the program in Chapter 17 takes advantage of the possibility of having a `setf` form as the first argument to `defun` , it will only work in more recent Common Lisp implementations. If you want to use it in an older implementation, substitute the following code in the final version:

```
(proclaim '(inline lookup set-lookup))

(defsetf lookup set-lookup)

(defun set-lookup (prop obj val)
  (let ((off (position prop (layout obj) :test #'eq)))
    (if off
        (setf (svref obj (+ off 3)) val)
        (error "Can't set ~A of ~A." val obj))))

(defmacro defprop (name &optioanl meth?)
  `(progn
     (defun ,name (obj &rest args)
       ,(if meth?
          `(run-methods obj ',name args)
          `(rget ',name obj nil)))
     (defsetf ,name (obj) (val)
       `(setf (lookip ',',name ,obj) ,val))))
```

备注 276 (Notes 276)

If `defmeth` were defined as

```
(defmacro defmeth (name obj parms &rest body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (setf (gethash ',name ,gobj)
             #'(lambda ,parms
                 (labels ((next ()
                            (funcall (get-next ,gobj ',name)
                                     ,@parms)))
                   ,@body))))))
```

then it would be possible to invoke the next method simply by calling `next` :

```
(defmeth area grumpy-circle (c)
  (format t "How dare you stereotype me!""/,")
  (next))
```

備注 284 (Notes 284)

For really fast access to slots we would use the following macro:

```
(defmacro with-slotref ((name prop class) &rest body)
  (let ((g (gensym)))
    `(let ((,g (+ 3 (position ,prop (layout ,class)
                              :test #'eq))))
       (macrolet ((,name (obj) `(svref ,obj ,',g)))
         ,@body))))
```

It defines a local macro that refers directly to the vector element corresponding to a slot. If in some segment of code you wanted to refer to the same slot in many instances of the same class, with this macro the slot references would be straight `svref`s.

For example, if the balloon class is defined as follows,

```
(setf balloon-class (class nil size))
```

then this function pops (in the old sense) a list of ballons:

```
(defun popem (ballons)
  (with-slotref (bsize 'size balloon-class)
    (dolist (b ballons)
      (setf (bsize b) 0))))
```

备注 284-2 (Notes 284-2)

Gabriel, Richard P. Lisp Good News, Bad News, How to Win Big [http://www.dreamsongs.com/Files/LispGoodNewsBadNews.pdf] *AI Expert*, June 1991, p.35.

早在 1973 年， Richard Fateman [http://en.wikipedia.org/wiki/Richard_Fateman] 已经能证明在 PDP-10 [http://en.wikipedia.org/wiki/PDP-10] 主机上， MacLisp [http://en.wikipedia.org/wiki/Maclisp] 编译器比制造商的 FORTRAN 编译器，产生出更快速的代码。

译注：该篇 MacLisp 编译器在 PDP-10 可产生比 Fortran 快的代码的论文在这可以找到 [http://dl.acm.org/citation.cfm?doid=1086803.1086804]

备注 399 (Notes 399)

It's easiest to understand backquote if we suppose that backquote and comma are like quote, and that `` `,x `` simply expands into `(bq (comma x))`. If this were so, we could handle backquote by augmenting `eval` as in this sketch:

```
(defun eval2 (expr)
  (case (and (consp expr) (car expr))
    (comma (error "unmatched comma"))
    (bq (eval-bq (second expr) 1))
    (t  (eval expr))))

(defun eval-bq (expr n)
  (cond ((atom expr)
         expr)
        ((eql (car expr) 'comma)
         (if (= n 1)
             (eval2 (second expr))
             (list 'comma (eval-bq (second expr)
                                   (1- n)))))
        ((eql (car expr) 'bq)
         (list 'bq (eval-bq (second expr) (1+ n))))
        (t
         (cons (eval-bq (car expr) n)
               (eval-bq (cdr expr) n)))))
```

In `eval-bq`, the parameter `n` is used to determine which commas match the current backquote. Each backquote increments it, and each comma decrements it. A comma encountered when n = 1 is a matching comma. Here is the example from page 400:

```
> (setf x 'a a 1 y 'b b 2)
2
```

```
> (eval2 '(bq (bq (w (comma x) (comma (comma y))))))
(BQ (W (COMMA X) (COMMA B)))
> (eval2 *)
(W A 2)
```

At some point a particularly remarkable molecule was formed by accident. We will call it the Replicator. It may not necessarily have been the biggest or the most complex molecule around, but it had the extraordinary property of being able to create copies of itself.

Richard Dawkins

The Selfish Gene

We shall first define a class of symbolic expressions in terms of ordered pairs and lists. Then we shall define five elementary functions and predicates, and build from them by composition, conditional expressions, and recursive definitions an extensive class of functions of which we shall give a number of examples. We shall then show how these functions themselves can be expressed as symbolic expressions, and we shall define a universal function apply that allows us to compute from the expression for a given function its value for given arguments.

John McCarthy

Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I

Index