Chapter 1

簡介 (Introduction)

約翰麥卡錫 (John McCarthy 1927-2011 R.I.P.) 和他的學生於 1958 年開始 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。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 與眾不同。

Lisp 與眾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 運算元。當新的抽象概念風行時(如物件導向程式設計 (Object-oriented programming)),我們總是發現這些新概念在 Lisp 是最容易來實現的。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)

為什麼要學 Lisp? 因為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回傳小於 n 的數字總和,那麼用 Lisp 和 C 是差不多的:

```
; Lisp
(defun sum (n)
  (let ((s 0))
    (dotimes (i n s)
      (incf s i))))
```

```
/* C */
int sum(int n){
  int i, s = 0;
  for(i = 0; i < n; 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 ,回傳把 n 與傳入引數 (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。當我剛開始寫程式時 —用 Baisc —我不知道有遞迴 (recursion),因為我根本不知道有這個東西。我是用 Basic 在思考。我只能用疊代 (iteration) 的概念表達算法,所以我怎會知道遞迴呢?

如果你不知道 詞法閉包 (Lexical Closure) (上述 addn 的範例),相信我,Lisp 程式設計師一直使用它。很難找到任何長度的 Common Lisp 程式沒有使用閉包的好處。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 特質是,Lisp 程式是用 Lisp 的資料結構來表示的。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 —它們叫做巨集 (Macro),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巨集了。

有了巨集 (macros)、閉包 (closures) 以及執行期型態 (run-time typing), 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 才知道為什麼此言不虛。但這不是空泛之言。這是一個重要的論點,而在 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為了理解 17 章程式的概念。你的努力會有所回報:你會感到在 C++ 寫程式是窒礙難行的,就像有經驗的 C++ 程式設計師用 Basic 寫程式會感到窒息一樣。更加鼓舞人心的是,如果我們思考為什麼會有這種感覺。用 Basic 寫程式對於平常用 C++ 寫程式是令人感到窒息的,是因為有經驗的 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 給你別的語言所不能提供的工具。但更多的是,獨立地說,伴隨 Lisp 的新特性 —自動記憶體管理 (automatic memory management),顯式型態 (manifest typing),閉包 (closures),等等 —每一項都使寫程式變得如此簡單。結合起來,它們組成了一個關鍵的部分,使得一種新的寫程式的方式是有可能的。

Lisp 被設計成可擴展的:它讓你定義自己的運算元。這是可能的,因為 Lisp 是由和你的程式一樣的函數與巨集所構成的。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 語言寫程式時,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 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。

由下而上的程式設計,自然地發展出可擴展的軟體。如果你把由下而上地程序設計的原則想成你程式的最上層,那這層就成為使用者的程式語言。因為可擴展的思想深植於 Lisp 當中,使得 Lisp 成為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了 Lisp 作為擴展自身的語言: GNU Emacs ,Autocad ,和 Interleaf 。

由下而上的程式設計,也是得到可重複使用軟體的最好方法。寫可重用軟體的本質是把共同的地方從細節中分離出來,而由下而上的程式設計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。就像 Frederick Brooks 所指出的,寫程式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 寫程式所花的時間較少。這種效果被 Lisp 的動態特質放大了:在 Lisp 中,編輯 –編譯 –測試循環短到使寫程式像是即時的。

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 (*rapid prototyping*) 描述了一種從 Lisp 開始的寫程式方法:在 Lisp ,你可以用比寫規格說明更短的時間,寫出一個原型來,而這種原型是高度抽象化的,可作為一個比用英語所寫的更好的規格說明。而且 Lisp 讓你可以輕易地從原型轉成產品軟體。當寫一個考慮到速度的 Common Lisp 程式時,透過現代編譯器的編譯,它們和用其他的高階語言寫的程式執行得一樣快。

除非你相當熟悉 Lisp ,否則這個簡介像是無意義的言論和冠冕堂皇的聲明。Lisp 凌駕物件導向程式設計? 你創造適合你程式的語言? 用 Lisp 寫程式是即時的? 這些說法是什麼意思? 現在這些說法就像是空淡的湖泊。隨著你學到更多實際的 Lisp 特色,見過更多可執行的程式,這些概念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你 Lisp 語言,而是教你一種新的程式設計方法,這種方法因為有了 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 的寫程式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,錯誤永遠不應出現。事前辛苦訂出縝密的規格說明,確保

程式完美的執行。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。

身為 OS/360 的專案經理,Frederick 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,產品延期,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局限性。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。

這其實就是新的寫程式模式所假設的。設法盡量降低錯誤的成本,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,較少地依靠事前規畫。

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消息來源中受益:過去撰寫程式所獲得的經驗。

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,以致於你以歷經好幾個設計和實現的循環,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,因為你將更快地

發現它們。你也不用擔心那麼多臭蟲。當你用函數式風格來寫程式,你的臭蟲只有局部的影響。當你使用一種很抽象的語言,某些臭蟲 (如 迷途指標 (Dangling Pointer)) 不再可能發生,而剩下的臭蟲很容易找出,因為你的程式更短了。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為它產生的結果。聽起來很奇怪,少的計畫意味著更好的設計。技術史上相似的例子不勝枚舉。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 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為可能。Janson 寫道:

如果沒有油畫顏料,佛萊明大師們 (Flemish masters) 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為畫家的基本顏料。

做為一種介質,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:交互式開發環境,垃圾回收 (Garbage collection),執行期類別,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,因為意味者我們可以從事更有野心的專案。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。