

函數編程與推論

Functional Program Construction and Reasoning

Shin-Cheng Mu

September 10, 2019

DRAFT

DRAFT

目錄

目錄	1
0 符號、演算、與抽象化	3
0.1 騎士與惡棍之島	3
0.2 讓符號為你工作	5
0.3 抽象化	6
0.4 抽象化與表達力	7
0.5 正確性	8
0.6 可演算的程式語言	9
0.7 相關資料	11
1 值、函數、與定義	13
1.1 值與求值	13
1.2 函數定義	18
1.3 高階函數	21
1.4 函數合成	25
1.5 λ 算式	28
1.6 簡單資料型態	30
1.7 弱首範式	35
1.8 串列	36
1.9 全麥編程	44
1.10 自訂資料型別	46
1.11 參考資料	47
2 歸納定義與證明	51
2.1 數學歸納法	52
2.2 自然數上之歸納定義	53
2.3 自然數上之歸納證明	56
2.4 串列與其歸納定義	59
2.5 更多歸納定義與證明	63
2.6 由集合論看歸納法	70
2.7 歸納定義的簡單變化	73
2.8 完全歸納	75
2.9 良基歸納	77

2.10 詞典序歸納	81
2.11 交互歸納	83
2.12 參考資料	83
Bibliography	85
Index	87

DRAFT

符號、演算、與抽象化

我們先從一個故事開始。

一個島上住著兩種人，騎士 (knight) 與惡棍 (knave)。騎士總說實話，惡棍總說謊話 — 所謂「謊話」的意思是實話的相反。從外表看不出一個人是騎士或惡棍。你站在一個山洞前。傳說中，山洞盡頭藏著價值連城的黃金。但也有可能並非如此，洞裡其實是一頭龍，一進去就會被吃了。洞口站著一位老先生，看不出是騎士或是惡棍。你該如何設計一個問題問他，得知山洞裡到底有沒有金子呢？

我們不妨先做個熱身題。

例 0.1. 假設在你面前站著兩位居民， A 與 B 。居民 A 說：「如果你問 B 他自己是不是騎士，他會說是。」由此你可知道 A 是騎士或是惡棍嗎？ B 呢？

在讀下去之前，請先嘗試解解看。

0.1 騎士與惡棍之島

剛剛你是怎麼解這問題的？。許多人解這類問題用的是窮舉法：假設 A 是騎士、 B 也是騎士，看看會不會有矛盾，接著假設 A 是騎士、 B 是惡棍，看看是否有矛盾... 依此類推。

我們想介紹另一種做法。我們將「 A 是騎士」簡寫成 A 。如果 A 說了某個陳述 S ，我們無法知道 S 是否成立，但我們知道

$$A \equiv S .$$

其中的 (\equiv) 可以理解為邏輯中「若且唯若」的關係，也可以簡單理解為（真假值的）「等於」。若用口語說，當我們寫 $P \equiv Q$ ，表示 P 和 Q 的真假值相同。確

實，假定 A 說了陳述 S ，如果 A 是騎士， S 便是真的；如果 A 不是騎士， S 便是假的。因此「 A 說了陳述 S 」可寫成 $A \equiv S$ 。

「若且唯若」這符號有幾個性質。首先，對任何陳述 P ，可知 $P \equiv P$ 必定為真：

$$(P \equiv P) \equiv \text{true} .$$

我們將這條規則命名為「自反恆真」。其次，既然 (\equiv) 可以理解為「等於」，理所當然該有交換律：¹

$$\begin{aligned} P &\equiv Q \\ &= Q \equiv P . \end{aligned}$$

最後，很少人注意到，真假值的「等於」是有結合律的：對任何 P, Q , 和 R ，我們有這樣的性質：²

$$\begin{aligned} (P \equiv Q) &\equiv R \\ &= P \equiv (Q \equiv R) . \end{aligned}$$

為了練習，我們多考慮一些情況。假設我們問 A ：「你是騎士嗎？」而 A 回答「是的！」這可以記成 $A \equiv A$ 。然而，根據自反恆真， $A \equiv A$ 可以化簡為 true 。這意味著：在這島上不管你問誰「你是騎士嗎」，對方都會回答「是」。

如果 A 說：「 B 是騎士」呢？這可以記成 $A \equiv B$ 。由此我們無法知道 B 到底是騎士還是惡棍，但我們可以知道 A 和 B 是同一種人。

現在想想問題 0.1：居民 A 說「如果你問 B 他自己是不是騎士，他會說是。」這可以記成 $A \equiv (B \equiv B)$ 。我們來演算看看：

$$\begin{aligned} A &\equiv (B \equiv B) \\ &= \{ \text{自反恆真} \} \\ A &\equiv \text{true} \\ &= \{ \text{結合律與自反恆真} \} \\ A & . \end{aligned}$$

也就是說，我們不知道 B 到底是騎士還是惡棍，卻可知道 A 一定是騎士！這段演算中，把 $A \equiv \text{true}$ 換成 A 的最後一步，直覺上似乎很直觀。如果要探究理由，其實用了結合律與自反恆真： $(A \equiv A) \equiv \text{true}$ 根據結合律可代換為 $A \equiv (A \equiv \text{true})$ ，因此 $A \equiv \text{true}$ 可代換成 A 。

再考慮一個熱身題：如果 A 說「我和 B 是同一種人」呢？這可記成 $A \equiv (A \equiv B)$ 。我們算算看：

$$\begin{aligned} A &\equiv (A \equiv B) \\ &= \{ \text{結合律} \} \end{aligned}$$

¹ (\equiv) 和 $(=)$ 都是「等於」。我們在此處使用不同符號，僅是為了區分出先後次序：同一行內部的相等用 (\equiv) ，行與行之間的相等用 $(=)$ 。本書其他地方的用法可能不同。

² $(P \equiv Q) \equiv R$ 只在 P, Q , 和 R 是真假值（或邏輯陳述）時有意義，否則 P, Q , 和 R 無法用 (\equiv) 串起。反例： $(3 = 3) = 3$ 可化簡成 $\text{true} = 3$ ，但後式中等號兩邊的值型別不同，不是合法的句子。也許因為我們較常考慮數字的相等，而較少用符號表達真假值的相等， (\equiv) 的結合律並不廣為人知。

$$\begin{aligned}
 & (A \equiv A) \equiv B \\
 = & \{ \text{自反恆真} \} \\
 & \text{true} \equiv B \\
 = & \{ \text{交換律與自反恆真} \} \\
 & B .
 \end{aligned}$$

這次，我們不知 A 是騎士還是惡棍，卻可知 B 一定是騎士。

最後，我們該回到本章開頭的問題了。你站在山洞口，面對洞口的老人。姑且稱呼他為 A 。要怎麼想出一個問題問他，用來判斷島上到底有沒有金子呢？之前介紹邏輯符號的目的是讓我們可避免瞎猜，而可用代數解未知數的方式把問題推演出來。把「島上有金子」這個命題記為 G 。我們希望問某個問題 Q 。如果 A 對問題 Q 回答「是」，就表示島上有金子，反之則否。

- 「 A 對問題 Q 回答『是』」記為 $A \equiv Q$ ；
- 「 A 對問題 Q 回答『是』，島上便有金子」其實應該是一個若且唯若的命題，寫成 $G \equiv (A \equiv Q)$ 。

而根據交換律和結合律，我們可以推演：

$$\begin{aligned}
 & G \equiv (A \equiv Q) \\
 = & \{ \text{結合律} \} \\
 & (G \equiv A) \equiv Q \\
 = & \{ \text{交換律} \} \\
 & Q \equiv (G \equiv A) .
 \end{aligned}$$

我們可得知 $Q \equiv (G \equiv A)$ ，也就是說我們要問 A 的問題 Q 就是 $G \equiv A$ ：「請問，『島上有金子』和『你是騎士』是不是等價的呢？」³

希望這個島上不論騎士或惡棍，邏輯都蠻不錯，才聽得懂這種問題囉！

0.2 讓符號為你工作

回顧看看，我們方才解各種「騎士與惡棍」問題的方法都分為兩個步驟：第一步先把情況以一組符號描述出來，第二步則是依照這些符號的規則下去推演。理想上，第二步比第一步容易，因為我們在進行推演時已不用回頭去思考這些符號的意思，只需照著符號本身的規則不加思索地推演。這麼的好處是能減輕我們思考的負擔。

探究怎樣的符號組成是合法的、一組合法的符號能否經由給定的規則轉換成另一組符號，是語法 (syntax) 層次上的問題。問這些符號的「意思」是什麼，則是語意 (semantics) 的層次。以騎士與惡棍問題為例，如果我們回到第一原則去窮舉「如果 A 是騎士， B 也是騎士.. 如果 A 是惡棍， B 也是惡棍..」等等種種可能，這是在語意上思考。至少對這個問題來說，語意上的解法不僅較繁瑣，也只能用來回答「某人是騎士還是惡棍」類型的問題，而難以用來推演出「該如何問老人，才能得知山洞中是否有黃金？」。上一節的解法則是利用語法

³不妨試試看能否把這句子講得不那麼繞口？

幫助我們：將問題寫下，利用交換律、結合律等等規則時，我們其實沒有回頭去思考「等等，這個式子是什麼意思？」但在符號的幫助之下，我們以更簡潔的方式解了許多個問題。

再舉一個「以符號思考」的例子：你會如何算 28×18 ？如果你的思路是「28乘以19是10個28加上8個28，而10個28是280，8個28是... 10個28減掉2個28！」那麼這種思考方式比較接近語意上的。你不斷在思考這個式子「代表什麼」，並試圖尋找較簡單的捷徑。如果你拿起紙筆，甚至在心中畫出了這樣的符號：

$$\begin{array}{r} 28 \\ \times 17 \\ \hline \end{array}$$

然後開始照背誦的九九乘法表去推演，此時你用的是語法式的解法。你也許不會思考許多，只以熟悉的規則推著演算的進行。以乘法而言，這兩種作法各有用處。但不可否認地，這套語法式解法的存在使我們可安心地把乘法當作已經解決的問題：我們可以不用思考地作乘法。

許多人見到數學符號就覺得難。事實上，數學符號是發明來簡化問題的。符號與符號之間的規則讓我們可僅用語法、不需在語意上思考，我們因而可把腦力用在更難的問題上。大家看到數學符號就覺得難，其實正是因為它們常被用在在不靠數學符號就難以解決的難題上。難的是這些問題，而符號給了我們解決它們的能力。

這一切和程式語言有什麼關係呢？因為，一套好的程式語言就是一套設計良好的符號。它能幫助我們描述問題，並找出解決問題的方法。一些程式語言學者提出的口號「讓符號為你工作 (let the symbols do the work!)」恰好地描述了這門學問的精神。

0.3 抽象化

如前所述，我們解決問題的第一步是以符號將它表達出來。這一步稱作「抽象化 (abstraction)」，通常是較難的一步。

例 0.2. Mary 有的蘋果數目是 John 的兩倍。Mary 發現她的蘋果中有一半已經壞掉了，於是丟了它們。John 則吃了一顆蘋果。現在 Mary 有的蘋果數目仍是 John 的兩倍。請問他們最初各有多少顆蘋果？

你會如何解這問題呢？如果回到第一原則，我們也許可以一步步嘗試：試試看 John 最初有一顆蘋果，Mary 有兩顆的情況是不是合理解答，然後試試看 John 有兩顆，Mary 有四顆的情況... 這是在語意上解問題。就如同我們解騎士與惡棍問題時用窮舉法一樣。

但許多人也許會用代數：把 Mary 最初的蘋果數目用 m 表示，John 的蘋果數目用 j 表示，寫成這樣的式子：

$$\begin{aligned} m &= 2j, \\ m/2 &= 2(j-1). \end{aligned}$$

接下來，用高中程度的代數方法，就可以找出 m 和 j 的值了。

代數方法是純粹基於語法的技術：我們觀察式子的結構，決定該怎麼作（例如把第二個式子乘以二，兩式相減；或著把 m 換成 $2j$ ），而不需記得 m 和 j 分別是什麼意思。

不需依賴「意義」在此是個重要的優點：這表示這套代數方法和特定問題的意義無關，可應用在許許多多場合。可用來解這個問題，也可用來解難免問題... 只要能把問題表示成代數式子，就有一個機械化、不需費神思考的解法。

從「Mary 有的蘋果數目是 John 的兩倍...」到 $m = 2j; m/2 = 2(j-1)$ 的轉變是一個「抽象化」。抽象化在此的意義是將不重要的資訊拋棄，只『抽取』此問題中最關鍵的元素。由於此處我們的目的是找出「Mary 與 John 最初各有多少顆蘋果」，我們可猜想關於數字的資訊（例如 $m = 2j$ ）是重要的，並猜想其他的一些資訊（蘋果是壞掉還是被吃掉了？兩人丟/吃蘋果的先後順序？）可能是不重要的。因此我們決定只留下關於數字的資訊，並幸運地確實靠此便解出了問題。

電腦是個只會處理符號的機器。實體世界的問題之所以能用電腦來解決，仰賴的就是「抽象化」這一步。有人說，整個計算科學就是關於抽象化的學問。

日常用語中，當我們說某事物「很抽象」，通常意味該事物模糊而難以理解。在計算科學中，「抽象」的事物才是最關鍵、最確實、最該把握的。我們前面才說過「數學符號讓事情變容易」，現在又說「抽象過的事物是最確實的」。看來學程式語言久了，讓我們越來越難與常人溝通。最後只好離群索居，躲到騎士與惡棍之島上捉弄來訪的遊客囉！

0.4 抽象化與表達力

在開發較具規模的軟體前，許多軟體方法建議我們分析問題、需求，並寫成形式化的規格。⁴ 這是一種抽象化。面對真實的問題時，這一步不容易。

我們再回到蘋果的例子，回想起在該例中，我們選擇抽取和值有關的資訊，而拋棄了許多其他：因果、先後順序... 等等。

在某些問題或應用中，因果或時間順序是否有可能是重要的？確實有。解這些問題時，我們可能在中途發現選錯了抽象的方式，使得留下的資訊不足以解決該問題。於是我們只好重新開始。有時，我們甚至會發現現有的符號不足以表達這些問題，得設計另一套符號。

這讓我們討論到解決問題重要的第零步：在我們能用符號表達問題之前，得先有人設計出一套完善的、足以表達許多問題的符號及其運算規則。為不同目的，我們可能設計不同的符號。一套符號也伴隨著該符號可如何轉變、操作（例如我們見過的交換律、結合律，某些符號碰在一起可化簡、等等）的規則。符號及其規則合起來成為一個「形式 (formal) 系統」。使用設計過的符號與其規則解決問題的研究被稱作「形式方法 (formal method)」。⁵ 此處的“formal”一詞，意指我們利用符號的「形式 (form)」來解決問題。⁵

⁴例如，Abrial [1996] 的 B method 是以一套基於集合論與一階邏輯的形式語言描述軟體規格的方法。

⁵大學資訊系所的一門必修課“formal language”中的“formal”也為「形式」之意。早期台灣將該門課稱為「正規語言」，可能是將 formal 誤解為相對於 casual 的「正式/正規」。

如果我們希望電腦幫我們解問題，需設計的符號就是一套程式語言。這不是一件容易的事：一個程式語言的表達力得強到足以描述所有該語言被設計來解決的問題。一套程式語言是設計者看待世界的抽象觀點。有些程式語言主張世界的狀態可以被一個個指令改變（「把 x 的值更新為 $x + y$ ；在螢幕上畫一個方塊...」）；有些語言認為世界應該看待為一個個物件；有些語言認為世界是許多彼此傳送訊息的共時程序；有些語言認為只要描述出每個個體之前的邏輯關係，交給電腦推論結果，就是很好的計算模型；本書中將介紹的抽象觀點則主張把一切都看成函數：寫程式是定義函數，圖形是座標到顏色的函數，動畫則是時間到圖形的函數...

晚近的語言設計，尤其是型別系統的設計，採用了許多邏輯學界的結果。本章談「騎士與惡棍之島」時，使用的是命題邏輯 (propositional logic) 的一種分支。一套「邏輯」也可視作一群符號以及其規則。許多人可能不知道，邏輯有許多種。命題邏輯是一套簡單的邏輯，其優點是給定任一合法的句子，都有一套機械性作法可得知其成立與否——這個性質稱作「可判定性 (decidability)」。但命題邏輯的表達力並不強：許多事情無法在命題邏輯中表達出來。

想表達更細緻的陳述，可使用表達力更強的邏輯。有些邏輯能表達「對所有」、「存在」；有些邏輯能表達先後順序以及「將一直成立」、「將在未來某時間成立」等觀念；有些邏輯可用來表達概念與概念之間的關係。但有得必有失，一套邏輯的表達力只要強到某個程度，就不再有可判定性了，沒有一套固定的方法可知道任意一個邏輯式子的真假。這是理性的限制。在知道形式系統的侷限之下盡力探索並發揮其極限，便是這套學問迷人之處。

因此，設計程式語言時總得掌握這樣的平衡：我們希望語言的表達力強到足以描述我們希望表達的運算程序，但又不希望強到無法掌握其性質。

0.5 正確性

程式語言是設計者選擇用來看待世界的抽象方式。這樣的選擇必然是基於一些考量：如果設計者希望程式能很容易分割、重用，他可能選擇易於將大問題分解成小問題的觀點。如果設計者希望該語言能有效率很高的實作，這語言看待世界的方式可能就和機器的世界觀很接近。

本書選擇的方向則是「程式語言應能幫助我們確保程式的正確性」。

但什麼是「正確」？直觀說來，一個程式「正確」的意思是該程式「做到了我們要它做的事。」但，電腦本來不就只能一個指令一個動作地做我們要它做的事嗎？那麼「正確」的意思到底是什麼？

我們考慮一下這個問題：

例 0.3 (最大區段和). 給定一串（有正有負）的數字。請找出一段連續的數字，使其總和越大越好，並傳回這個總和。

這是經典的「最大區段和」問題。如前面的章節所述，面對一個問題，我們先把它描述成符號。如果 a 是給定的那串數字， N 是其長度， $a[i]$ 是數字中的第 i 個，而 m 是我們想求出的、最大的那個和，最大區段和問題可以描

述成這樣：

$$\begin{aligned} m &= \max\{sum(i, j) \mid 0 \leq i \leq j \leq N\} , \\ sum(i, j) &= \sum_{k=i}^{j-1} a[k] , \end{aligned} \quad (1)$$

其中 $sum(i, j)$ 代表 $a[i]..a[j-1]$ 的和，而 $\max S$ 找出集合 S 中最大的那個。

但（許多讀者可能也知道）這個問題其實有個線性時間內可執行完畢的解：將這串數字 a 從頭到尾看過一遍，維持兩個變數 s 和 m 。每看到一個新數字 $a[j]$ ，將 s 更新為 0 與 $s+a[k]$ 中較大的那個，並把 m 更新為 s 與 m 中較大的那個。寫成程式的話，可能是這樣：

```
s, m = 0, 0
for k in range(0, n-1)
    s = max(0, s + a[k])
    m = max(s, m)
```

那麼，明顯的問題來了。上面的程式和問題描述(1)一點也不像。我們怎麼知道程式真正實作了(1)的要求？

由此談「正確性」，相信清楚多了。數學式(1)給的是一個規格 (specification)，談的是我們要的結果 (*what*)，而程式描述的則是怎麼做 (*how*)。「正確性」總是相對於一個規格而言的：描述「怎麼做」的程式是否真做到了規格的要求？

「最大區段和」的例子可以給我們幾個啟示。首先，把「要什麼」和「怎麼做」牽上關聯，有時是很困難的。當我們說一個程式「很難懂」，其中一個意思是很難看出「為什麼這個程式實作出了這個規格？」（也就是說「為什麼這個程式是正確的？」）看懂一個程式，也就是了解他為何是正確的。

「最大區段和」的例子也告訴我們，「難懂」的程式不一定要很長。上面的程式短短四行，但若沒有輔助解釋，一般人可能很難「看懂」它。⁶

如果規格已經不見了呢？這樣的情形更糟。當我們在沒有規格的情況下問一個程式「是做什么的」，我們真正問的是「這個程式符合的規格是什麼？」這可能是個難題。回頭看看上面的程式，如果沒有給(1)，你能說得出這個程式在做什麼嗎？

0.6 可演算的程式語言

正確性是如此重要又難以掌握的性質，我們因此希望程式語言能幫助我們確保程式的正確性。我們採用的方法是：給定一個規格，我們希望正確的程式能夠由其規格演算、推導出來。就如同在騎士與惡棍問題中，我們把希望 Q 滿足的性質寫下，然後利用符號的規則算出 Q 。我們希望程式也能如此從其規格被算出來。

以最大區段和問題為例，用我們之後將介紹的符號，我們可以把(1)改寫成：

⁶理解這個程式的關鍵是： m 永遠是 $a[0]..a[k]$ 中所有連續區段最大的和，而 s 永遠是 $a[0]..a[k]$ 中最右端為 $a[k]$ 的連續區段中最大的和。這兩個條件是該迴圈的不變式 (loop invariant)。不變式是了解一個迴圈最重要的資訊。

α	β	γ	δ	ε	ς	ζ	η	θ	ι	κ	λ	μ	ν	ξ	ο	π	
1	2	3	4	5	6	7	8	9	10	20	30	40	50	60	70	80	90
ρ	σ	τ	υ	φ	χ	ψ	ω	ϗ	α	β	γ						
100	200	300	400	500	600	700	800	900	1000	2000	3000						

Figure 0.1: 希臘數字。

$mss = \text{max} \cdot \text{map sum} \cdot \text{segments}$,

其意思大約是「給定一串數字，找出它的所有連續區段，算每一個區段的和，然後傳回其中最大的。」接下來的問題是， mss 有沒有比較快的實作？我們希望可以代數方法，由 mss 的規格出發：

$\text{max} \cdot \text{map sum} \cdot \text{segments}$
 $= \{ \text{segments 的定義} \}$
 $\text{max} \cdot \text{map sum} \cdot \text{concat} \cdot \text{map inits} \cdot \text{tails}$
 $= \{ \text{某定理} \}$
 \dots
 $= \{ \text{另一些定理} \}$
 $\text{max} \cdot \text{scanr} (\oplus) 0$.

最後的 $\text{max} \cdot \text{scanr} (\oplus) 0$ 相當於前一節的程式，只是以不同的符號表示。

我們可以由規格出發，在符號上操作，將程式如同求代數的解一樣地算出來。只要每個小步驟都正確，最後的程式就是正確的。這套技術稱作「程式推導 (program derivation)」或「程式演算 (program calculation)」，將是本書重要的主題。而在本書的觀點中，函數編程相較於其他典範的特色與優勢是函數語言是一套適合演算的符號系統。

適合演算的符號 我們舉個例子，談談符號「適合演算」是怎麼回事。圖 0.1 中顯示的是西元前四世紀左右希臘人使用的數字表示法。以希臘字母表為順序，1, 2,...到 9 分別寫成 α, β,...θ. 每遇到 10 的乘幂便換一組符號，如 10, 20, ...寫成 ι, κ...。11, 12, 13 分別寫成 ια, ιβ, ιγ, 21, 22, 23 則寫成 κα, κβ, κγ. 653 可寫成 χνγ. 這套表示法的缺點包括不易表示大數，以及不易做演算：我們需要知道 δ 加上 β 等於 ς, 以及 μ 加上 κ 等於 ξ。⁷

與之對比的是瑪雅數字。這套數字系統的年代不詳，據估計可能在西元四至世紀採用。一個點代表 1，一條橫杠代表 5。例如 19 寫成 𐄀𐄀𐄀。瑪雅數字採用 20 進位，較大的位數寫在上方。因此



𐄀𐄀𐄀

代表 $7(\equiv) \times 20 + 11(\equiv)$ ，也就是 151。另有個例外：第三個數字為 18（而非 20）的乘幂——因為 $18 \times 20 = 360$ 接近一年的日數。圖 0.2 中左邊由上至下是以

⁷西元前八世紀，希臘人曾使用以 Ι, Π, Δ, Η 分別表示 1, 5, 10, 100 的表示法。後來被圖 0.1 中的系統取代。無論如何，希臘人仍以此發展出了進步的數學。






$$\begin{array}{rcl}
 \cdot & 1 \times 18 \times 20 \times 20 \times 20 \times 20 & = 2880000 \\
 \cdots & 2 \times 18 \times 20 \times 20 \times 20 & = 288000 \\
 \div & 6 \times 18 \times 20 \times 20 & = 43200 \\
 \cdots & 2 \times 18 \times 20 & = 720 \\
 \equiv & 13 \times 20 & = 260 \\
 \equiv & 19 & \\
 & \text{總和: } 3212199 &
 \end{array}$$

Figure 0.2: 以瑪雅數字表示 3212199.

瑪雅數字寫成的 3212199。值得注意的是，此類以符號的位置表示其量級的數字表示法大都需要一個相當於「零」的佔位符號。在瑪雅數字中，該符號寫成 。因此 20 可寫成 ，與只有一個點的 1 區分。⁸

如何用瑪雅數字作加法？以下顯示的是 $151 + 55$ ：


$$\begin{array}{ccccccc}
 \cdot & & \cdot & & \cdot & & \cdot \\
 \cdot & + & \cdot & = & \cdot & = & \cdot \\
 \cdot & & \cdot & & \cdot & & \cdot \\
 \cdot & & \cdot & & \cdot & & \cdot \\
 151 & & 55 & & 206 & & 206
 \end{array}$$

我們可先把兩個個位數（與）和兩個 20 位數（與）分別相加。其中，個位數相加的結果是——超過了 20！於是我們把其中四個橫槓（即一個 20）往上進位成一個點，和 20 位數的四個點結合成另一個橫槓。

有趣的是，我相信大部分的讀者即使可能是第一次看到這種數系，也從未使用過 20 進位，還是可迅速理解以上的加法，甚至可以舉一反三地自己試試看。這就是一個適合演算的符號系統給我們的便利。

對本書來說，函數語言的價值便是——它是個適於演算的語言。

0.7 相關資料

本章中的騎士與惡棍問題所使用的邏輯形式稱作演算邏輯 (calculational logic)，其主要精神是將命題邏輯與述語邏輯表達為適合用於等式推導與演算的形式。Gries and Schneider [2003] 認為大部分形式邏輯系統是為了研究邏輯本身的特性而設計，而演算邏輯則是為了以邏輯解決問題而開發的。演算邏輯由程式語言學者們於 80 年代初期漸漸發展出來。⁹ Dijkstra and Scholten [1990] 將演算邏輯用於指令式程式語言的語意中。本章中的騎士與惡棍問題由 Backhouse [2003] 中節錄而來。這是一本介紹以形式方式構思演算法以解決問題的教科書。本章只用到了 () 一個運算子，該書中則有更完整的介紹。Gries and Schneider [October 22, 1993] 則是一本以演算邏輯為基礎的離散數學教科書。

「讓符號為你工作」這句口號可在 Dijkstra 的許多著作中見到（例如 Dijkstra and van Gasteren [1986], Dijkstra [2000]）。Dijkstra [2004] 寫道：「設計程式的

⁸一些早期數字表示法中有此種佔位符號，但並不見得把「零」視為可獨立存在的一個數。「將零視為一個數字」是數學史上的重要發明，許多數學性質都需有它的存在才得以描述。程式語言中也有類似的情況：有一個「不做任何事的指令」或一個「將輸入照樣傳回的函數（即 *id*）」對於描述程式的性質是很重要的。

⁹Gries and Schneider [2003] 將此歸功於 Roland Backhouse, Edsger W. Dijkstra, Wim H.J. Feijen, David Gries, Carel S. Scholten, 以及 Netty van Gasteren 等人。他們的共同研究主題是以形式方法開發程式。

人最好把程式當作精巧的算式。而我們知道只有一種設計精巧算式的可靠方法：用符號操作來推導它。我們得讓符號為我們工作，因為這是已知唯一在大尺度下仍可行的方法。」Misra 則在 1988 年 Marktoberdorf 暑期課程的演講 [Misra, 1989] 中虛構了一個故事：使用羅馬數字的羅馬人被推銷比較易於演算的印度-阿拉伯數字系統，卻不以為然地說「我們已經有奴隸為我們工作了！」

希臘與瑪雅數字的例子取自 Mazur [2014]。該書對於種種符號的演進有更詳盡的介紹。

DRAFT

值、函數、與定義

語言是概念的載體。如第 0 章所述，程式語言不僅用來表達概念，也用於演算，分擔我們思考的負擔。在本書中，為便於精確說明，我們也不得不選一個語言。

本書中使用的語言大致上基於 Haskell，但為適合本書使用而經過簡化、修改。我們將在本章初步介紹 Haskell 語言的一小部分，包括在 Haskell 中何謂「計算」、值與函數的定義、常見的資料結構，以及串列上的常用函數。目的是讓讀者具備足夠的基本概念，以便進入之後的章節。也因此，本書中所介紹的語言並非完整的 Haskell，本書也不應視作 Haskell 語言的教材。對於有此需求的讀者，我將在本章結尾推薦一些適合的教科書。

1.1 值與求值

Haskell 是個可被編譯 (compile)、也可被直譯 (interpret) 的語言。Haskell 直譯器延續了 LISP 系列語言的傳統，是個「讀、算、印 (read, evaluate, print)」的迴圈 — 電腦從使用者端讀取一個算式，算出結果，把結果印出，然後再等使用者輸入下一個算式。一段與 Haskell 直譯器的對話可能是這樣：

```
Main> 3 + 4
7
Main> sum [1..100]
5050
Main>
```

在此例中，`Main>` 是 Haskell 直譯器的提示符號。¹ 使用者輸入 `3 + 4`，Haskell 算出其值 `7` 並印出。接著，使用者想得知 `1` 到 `100` 的和，Haskell 算出 `5050`。

¹此處的畫面擷取自 GHCi (Glasgow Haskell Compiler Interactive)。GHC 為目前最被廣泛使用的 Haskell 實作。

演算格式

本書中將使用如下的格式表達（不）等式演算或推論：

$$\begin{array}{l} \text{expr}_0 \\ = \{ \text{reason}_0 \} \\ \text{expr}_1 \\ \geq \{ \text{reason}_1 \} \\ \text{expr}_2 \\ : \\ = \text{expr}_n . \end{array}$$

這是一個 $\text{expr}_0 \geq \text{expr}_n$ 的證明。式子 $\text{expr}_0 \dots \text{expr}_n$ 用具有遞移律的運算子（如 $(=)$, (\geq) 等等）串起。放在大括號中的是註解，例如 reason_0 是 $\text{expr}_0 = \text{expr}_1$ 的原因， reason_1 是 $\text{expr}_1 \geq \text{expr}_2$ 的原因。

根據 van de Snepscheut [1993, p19], 此格式最早由 W.H.J. Feijen 所建議。

上例中的算式只用到 Haskell 已知的函數（如 $(+)$, sum 等）。使用者若要自己定義新函數，通常得寫在另一個檔案中，命令 Haskell 直譯器去讀取。例如，我們可把如下的定義寫在一個檔案中：

```
double :: Int → Int
double x = x + x .
```

上述定義的第一行是個型別宣告。當我們寫 $e :: t$ ，代表 e 具有型別 t 。Int 是整數的型別，而箭號 (\rightarrow) 為函數型別的建構元。第一行 $\text{double} :: \text{Int} \rightarrow \text{Int}$ 便是告知電腦我們將定義一個新識別字 *double*，其型別為「從整數 (Int) 到整數的函數」。² 至於該函數的定義本體則在第二行 $\text{double } x = x + x$ ，告知電腦「凡看到 *double x*，均可代換成 $x + x$ 。」

求值 第 0 章曾提及：一套程式語言是設計者看待世界的抽象觀點。程式通常用來表達計算，因此程式語言也得告訴我們：在其假想的世界中，「計算」是怎麼一回事。指令式語言的世界由許多容器般的變數組成，計算是將值在變數之間搬來搬去。邏輯編程中，描述一個問題便是寫下事物間的邏輯關係，計算是邏輯規則「歸結 (resolution)」的附帶效果。共時 (concurrent) 程式語言著眼於描述多個同時執行的程式如何透過通道傳遞訊息，計算也靠此達成。

函數語言中，一個程式便是一個數學式，而「計算」便是依照該式子中各個符號的定義，反覆地展開、歸約，直到算出一個「值」為止。這個過程又稱作「求值 (evaluation)」。在 Haskell 直譯器中，若 *double* 的定義已被讀取，輸入 $\text{double } (9 \times 3)$ ，電腦會算出 54：

```
Main> double (9 × 3)
54
```

但 54 這個值是怎麼被算出來的？以下是其中一種可能：

²Haskell 內建至少兩種整數：Int 為有限大小（通常為該電腦中一個字組 (word)）的整數，Integer 則是所謂的大整數或任意精度整數，無大小限制。本書中只使用 Int。

$$\begin{aligned}
 & \text{double } (9 \times 3) \\
 = & \{ (\times) \text{ 的定義} \} \\
 & \text{double } 27 \\
 = & \{ \text{double 的定義} \} \\
 & 27 + 27 \\
 = & \{ (+) \text{ 的定義} \} \\
 & 54 .
 \end{aligned}$$

上述演算的第一步將 9×3 歸約成 27 — 我們尚未定義 (\times) 與 $(+)$ ，但目前只需知道它們與大家所熟悉的整數乘法、加法相同。第二步將 $\text{double } 27$ 變成 $27 + 27$ ，根據的是 double 的定義： $\text{double } x = x + x$ 。最後， $27 + 27$ 理所當然地歸約成 54 。「歸約」一詞由 β -reduction 而來，在此處指將函數本體中的形式參數代換為實際參數。³ 在上述例子中，我們遇到如 $\text{double } (9 \times 3)$ 的函數呼叫，先將參數 (9×3) 化簡，再展開函數定義，可說是由內到外的歸約方式。大部分程式語言都依這樣的順序求值，讀者可能也對這種順序較熟悉。

但這並不是唯一的求值順序。我們能否由外到內，先把 double 展開呢？

$$\begin{aligned}
 & \text{double } (9 \times 3) \\
 = & \{ \text{double 的定義} \} \\
 & (9 \times 3) + (9 \times 3) \\
 = & \{ (\times) \text{ 的定義} \} \\
 & 27 + (9 \times 3) \\
 = & \{ (\times) \text{ 的定義} \} \\
 & 27 + 27 \\
 = & \{ (+) \text{ 的定義} \} \\
 & 54 .
 \end{aligned}$$

以這個順序求值，同樣得到 54 。

一般說來，一個式子有許多種可能的求值順序：可能是由內往外、由外往內、或其他更複雜的順序。我們自然想到一個問題：這些不同的順序都會把該式子化簡成同一個值嗎？有沒有可能做出一個式子，使用一個順序會被化簡成 54 ，另一個順序化簡成 53 ？

我們看看如下的例子。函數 three 顧名思義，不論得到什麼參數，都傳回 3 ； inf 則是一個整數：

$$\begin{aligned}
 & \text{three} :: \text{Int} \rightarrow \text{Int} \\
 & \text{three } x = 3 , \\
 & \text{inf} :: \text{Int} \\
 & \text{inf} = 1 + \text{inf} .
 \end{aligned}$$

在指令式語言中， inf 的定義可能會被讀解為「將變數 inf 的值加一」。但函數語言中「變數」的值是不能更改的。此處的意義應是： inf 是一個和 $1 + \text{inf}$ 相等的數值。我們來看看 three inf 會被為甚麼？如果我們由內往外求值：

³Reduction 的另一個常見譯名是「化簡」，然而，許多情況下，一個式子被 reduce 後變得更長而不「簡」，因此本書譯為「歸約」。

```

three inf
= { inf 的定義 }
three (1 + inf)
= { inf 的定義 }
three (1 + (1 + inf))
= { inf 的定義 }
...

```

看來永遠停不下來！但如果我們由外往內，*three inf* 第一步就可變成 3：

```

three inf
= { three 的定義 }
3 .

```

我們該如何理解、討論這樣的現象呢？

範式與求值順序 為描述、討論相關的現象，我們得非正式地介紹一些術語。用較直觀、不形式化的方式理解，一個式子中「接下來可歸約之處」稱作其歸約點 (*redex*)。例如 $(9 \times 3) + (4 \times 6)$ 中， 9×3 與 4×6 都是歸約點。如果一個式子已沒有歸約點、無法再歸約了，我們說該式已是個範式 (*normal form*)。

回頭看，經由之前的例子我們已得知：

- 有些式子有範式 (如 *double* (9×3) 有個範式 54)，有些沒有 (如 *inf*)。
- 同一個式子可用許多順序求值。有些求值順序會碰到範式，有些不會。

給一個式子，我們很自然地希望知道它有沒有值，並算出其值。如果一個式子有很多個範式，我們便難說哪一個才是該式子的「值」。如此一來，立刻衍生出幾個問題。給定一個式子，我們是否能得知它有沒有範式呢？如果有，用哪個求值順序才能得到那個範式？以及，有沒有可能用一個求值順序會得到某範式，換另一個求值順序卻得到另一個範式？

很不幸地，第一個問題的答案是否定的：沒有一套固定的演算法可判定任意一個式子是否有範式。這相當於計算理論中常說到的停機問題 (*halting problem*) — 沒有一個演算法能準確判斷任意一個演算法 (對某個輸入) 是否能正常終止。

但對於另兩個問題，計算科學中有個重要的 *Church-Rosser* 定理。非常粗略地說，該定理告訴我們：在我們目前討論的這類語言中⁴

- 一個式子最多只有一個範式。
- 如果一個式子有範式，使用由外到內的求值順序可得到該範式。

如此一來，給定任一個式子，我們都可用由外到內的方式算算看。假設一算之下得到 (例如) 54。用其他的求值順序可能得不到範式，但若有了範式，必定也是 54。如果由外到內的順序得不到範式，用其他任何順序也得不到。

由於「由外到內」的求值順序有「最能保證得到範式」的性質，又被稱作「範式順序 (*normal order evaluation*)」。「由內到外」的則被稱作「應用順序 (*applicative order evaluation*)」。以本書的目的而言，我們可大致把 Haskell 使

⁴此處討論的可粗略說是以 λ -calculus 為基礎的函數語言。基於其他概念設計的程式語言當然可能不遵守 Church-Rosser 定理。

用的求值方式視為範式順序。但請讀者記得這是個很粗略、不盡然正確的說法——Haskell 實作上使用的求值方式可說是經過了許多最佳化。正式的 λ -calculus 教科書中對於歸約點、求值順序、Church-Rosser 定理等概念會有更精確的定義。

被迫求值 型別 `Bool` 表示真假，有兩個值 `False` 和 `True`。常用的函數 `not` 可定義如下：

```
not :: Bool → Bool
not False = True
not True  = False .
```

此處 `not` 的定義依輸入值不同而寫成兩個條款。這種定義方式在 Haskell 中稱作樣式配對 (*pattern matching*)：`False` 與 `True` 在此處都是樣式 (*patterns*)。遇到這樣的定義時，Haskell 將輸入依照順序與樣式們一個個比對。如果對得上，便傳回相對應的右手邊。本例中，若輸入為 `False`，傳回值為 `True`，否則傳回值為 `False`。

我們來看看 `not (5 ≤ 3)` 該怎麼求值？若依照範式順序，照理來說應先將 `not` 的定義展開。但若不知 `5 ≤ 3` 的值究竟是 `False` 還是 `True`，我們不知該展開哪行定義！因此，要計算 `not (5 ≤ 3)`，也只好先算出 `5 ≤ 3` 了：

```
not (5 ≤ 3)
= { (≤) 之定義 }
not False
= { not 之定義 }
True .
```

求值過程中若必須得知某子算式的值才能決定如何進行，只好先算那個子算式。在 Haskell 中有不少（有清楚定義的）場合得如此，包括遇上 (`≤`)、(`≥`) 等運算子、樣式配對、`case` 算式（將於第 1.6.1 節中介紹）... 等等。

習題 1.1 — 定義一個函數 `myeven :: Int → Bool`，判斷其輸入是否為偶數。你可能用得到以下函數：^a

```
mod :: Int → Int → Int ,
(==) :: Int → Int → Bool .
```

其中 `mod x y` 為 `x` 除以 `y` 之餘數，`(==)` 則用於判斷兩數是否相等。

習題 1.2 — 定義一個函數 `area :: Float → Float`，給定一個圓的半徑，計算其面積。（可粗略地將 `22/7` 當作圓周率。）

^a此處所給的並非這些函數最一般的型別。

惰性求值 實作上，Haskell 求值的方式還經過了更多的最佳化：例如將歸約過的算式改寫為它的值，避免重複計算。這套求值方式稱為惰性求值 (*lazy evaluation*)。

技術上說來，惰性求值和範式順序求值並不一樣。但前者可視為後者的最佳化實作 — 惰性求值的結果必須和範式順序求值相同。因此，在本書之中大部分地方可忽略他們的差異。本書中談到偏向實作面的議題時會用「惰性求值」一詞，在談不牽涉到特定實作的理論時則使用「範式順序求值」。

1.2 函數定義

考慮如下的函數定義：

```
smaller :: Int → Int → Int
smaller x y = if x ≤ y then x else y .
```

我們可粗略地理解為：*smaller* 是一個函數，拿兩個參數 *x* 與 *y*，傳回其中較小的那個。如 *smaller* (*double* 6) (3 + 4) 的值為 7。

習題 1.3 — 用前一節介紹的求值順序，將 *smaller* (*double* 6) (3 + 4) 化簡為範式。

守衛 如果函數本體做的第一件事就是條件判斷，Haskell 提供另一種語法：

```
smaller :: Int → Int → Int
smaller x y | x ≤ y = x
            | x > y = y .
```

這麼寫出的 *smaller* 的行為仍相同：如果 $x \leq y$ 成立，傳回 *x*；如果 $x > y$ ，傳回 *y*。但這種語法較接近一些數學教科書中定義函數的寫法。其中，放在 $x \leq y$ 和 $x > y$ 等位置的必須是型別為 *Bool* 的算式。它們擋在那兒，只在值為 *True* 的時候才讓執行「通過」，因此被稱為守衛 (*guard*)。如果有三個以上的選項，如下述例子，使用守衛比 *if..then..else* 更清晰可讀：

```
sign :: Int → Int
sign x | x > 0 = 1
      | x == 0 = 0
      | x < 0 = -1 .
```

遇到數個守衛時，Haskell 將依照順序嘗試每個選項，直到碰到第一個為真的守衛，然後只執行該項定義。也就是說，即使我們把 *sign* 定義中的 $x == 0$ 改為 $x \geq 0$ ，*sign* 10 的值仍會是 1。若每個守衛都是 *False*，程式則將中斷執行（並傳回一個錯誤訊息）。在許多程式中，我們會希望最後一個守衛能捕捉所有的漏網之魚：如果其他條款都不適用，就執行最後一個。一種做法是讓一個守衛是 *True*。或著，在 Haskell 中有個 *otherwise* 關鍵字可讓定義讀來更口語化些：

```
sign :: Int → Int
sign x | x > 0      = 1
      | x == 0      = 0
      | otherwise   = -1 .
```

事實上，*otherwise* 的定義就是 *otherwise* = *True*。

區域識別字 Haskell 提供兩種宣告區域識別字的語法：**let** 與 **where**。也許最容易理解的方式是用例子說明。

例 1.1. 工讀生每小時的時薪為新台幣 130 元。假設一週有五個工作天，每天有八小時上班時間。定義一個函數 $payment :: \text{Int} \rightarrow \text{Int}$ ，輸入某學生工作的週數，計算其薪資。

答. 我們當然可直接用一個式子算出薪資。但為清楚起見，我們可用兩個區域識別字 *days* 和 *hours*，分別計算該學生工作的日數和時數。如果用 **let**，可這麼做。

```
payment :: Int → Int
payment weeks = let days = 5 × weeks
                  hours = 8 × days
                  in 130 × hours .
```

let 算式的語法為

```
let  $x_1 = e_1$ 
     $x_2 = e_2 \dots$ 
in  $e$  .
```

其中 e 為整個算式的值，而 x_1, x_2 等等為區域識別字。兩個區域識別字的有效範圍包括 e ，以及 e_1 與 e_2 。

另一種語法是 **where** 子句。若用它定義 *payment*，可寫成這樣：

```
payment :: Int → Int
payment weeks = 130 × hours ,
  where hours = 8 × days
        days  = 5 × weeks .
```

□

該用 **let** 或是 **where**？大部份時候這可依個人習慣，看寫程式的人覺得怎麼說一件事情比較順。使用 **let** 時，敘事的順序是由小到大，先給「工作日數」、「工作時數」等小元件的定義，再用他們組出最後的式子 $130 \times \text{hours}$ 。使用 **where** 時則是由大到小，先說我們要算「工作時數乘以 130」，然後補充「其中，工作時數的定義是...」。

但，Haskell 之所以保留了兩種語法，是為了因應不同的用途。語法上，**let** 是一個算式，可出現在算式中。如下的算式是合法的，其值為 24:

```
(1 + 1) × (let y = double 5 in y + 2) .
```

where 子句的一般語法則如下例所示：

```
f  $x_0 = d_0$ 
  where  $y_0 = e_0$ 
```

$$f\ x_1 = d_1$$

$$\text{where } y_1 = e_1 \ .$$

由語法設計上即可看出，子句 **where** $y_0 = e_0$ 只能放在 $f\ x_0 = \dots$ 的一旁當作補述，不能出現在 d_0 之中。這個例子中， y_0 的有效範圍含括 d_0 與 e_0 。另， e_0 可以使用 x_0 。

算式中只能用 **let**。相對地，也有些只能使用 **where** 的場合。我們來看我們來看一個只能使用 **where** 的例子：

例 1.2. 延續例 1.1。今年起，新勞動法規規定工作超過 19 週的工讀生必須視為正式雇員，學校除了薪資外，也必須付給勞保、健保費用。學校需負擔的勞健保金額為雇員薪資的百分之六。請更新函數 *payment*，輸入某工讀生工作週數，計算在新規定之下，學校需為工讀生付出的總額。

答. 一種可能寫法是先使用守衛，判斷工作週數是否大於 19：

```
payment :: Int → Int
payment weeks | weeks > 19 = round (fromIntegral baseSalary × 1.06)
               | otherwise = baseSalary ,
where baseSalary = 130 × hours
      hours      = 8 × days
      days       = 5 × weeks .
```

在 **where** 子句中，我們先算出不含勞健保費用的薪資，用識別字 *baseSalary* 表達。如果 *weeks* 大於 19，我們得將 *baseSalary* 乘以 1.06；否則即傳回 *baseSalary*。函數 *fromIntegral* 把整數轉為浮點數，*round* 則把浮點數四捨五入為整數。請注意：兩個被守衛的算式都用到了 *baseSalary* — **where** 子句中定義的識別字是可以跨越守衛的。相較之下，**let** 算式只能出現在等號的右邊，而在守衛 *weeks > 19 = ...* 之後出現的 **let** 所定義出的區域識別字，顯然無法在 *otherwise = ...* 之中被使用，反之亦然。 □

巢狀定義 **let** 算式之中還可有 **let** 算式，**where** 子句中定義的識別字也可有自己的 **where** 子句。我們看看兩個關於 **let** 的例子：

例 1.3. 猜猜看 *nested* 和 *recursive* 的值分別是什麼。將他們載入 *Haskell* 直譯器，看看和你的猜測是否相同。

```
nested :: Int
nested = let x = 3
         in (let x = 5
             in x + x) + x ,

recursive :: Int
recursive = let x = 3
            in let x = x + 1
               in x .
```

答. *nested* 的值是 13，因為 $x + x$ 之中的 x 在 **let** $x = 5$ 的範圍中，而 $.. + x$ 中的 x 則在 **let** $x = 3$ 的範圍中。⁵ 至於 *recursive* 的值，關鍵在於 $x = x + 1$ 中右

⁵在各種語言中，範圍的設計都是為了給程式員方便：在寫子算式時，可不用擔心是否與外層的識別字撞名。在教學時，我們難免舉各種撞名的例子作為說明。若把這些刁鑽例子當作考題，就是違反設計者本意的發展了。

一級公民

在程式語言中，若說某物/某概念是一級公民 (*first-class citizen*) 或「一級的」，通常指它和其他事物被同等對待：如果其他事物可被當作參數、可被當作傳返回值、可被覆寫... 那麼它也可以。這是一個沒有嚴格形式定義的說法，由 Christopher Strachey 在 1960 年代提出，可用在型別、值、物件、模組... 等等之上。

例如：OCaml 是個有「一級模組」的語言，因為 OCaml 模組也可當作參數，可定義從模組到模組的函數 (OCaml 中稱之為 *functor*)。在 C 語言之中函數是次級的，因為函數不能當參數傳 (能傳的是函數的指標，而非函數本身)。Strachey 指出，在 Algol 中實數是一級的，而程序是次級的。

手邊的 x 指的是哪個。若是 $x = 3$ 的那個 x ，整個算式的值將是 4。若 $x = x + 1$ 中，等號右手邊的 x 也是左手邊的 x ，*recursive* 就是 $((\dots) + 1) + 1$ ，沒有範式。這兩種設計都有其道理。Haskell 選了後者：在 `let $x = e$ in ...` 之中， x 的有效範圍包括 e 。因此 *recursive* 在 Haskell 中會無窮地求值下去。但也有些函數語言中的 `let` 採用前者的設計。通常這類語言中會另有一個 `letrec` 結構，和 Haskell 的 `let` 功能相同。 □

1.3 高階函數

目前為止，我們看過由整數到整數的函數、由整數到真假值的函數... 那麼，可以有由函數到函數的函數嗎？函數語言將函數視為重要的構成元件，因此函數也被視為一級公民。如果整數、真假值... 可以當作參數、可以被函數傳回，函數當然也可以。一個「輸入或輸出也是函數」的函數被稱為高階函數 (*higher order function*)。Haskell 甚至設計了許多鼓勵我們使用高階函數的機制。本書中我們將見到許多高階函數。其實，我們已經看過一個例子了。

Currying 回顧 *smaller* 的定義：

```
smaller :: Int → Int → Int
smaller  $x$   $y$  = if  $x \leq y$  then  $x$  else  $y$  .
```

1.2 節中說「*smaller* 是一個函數，拿兩個參數 x 與 y 」。但這僅是口語上方便的說法。事實上，在 Haskell 中（如同在 λ -calculus 中），所有函數都只有一個參數。函數 *smaller* 其實是一個傳回函數的函數：

- *smaller* 的型別 $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ 其實應看成 $\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$ ：這個函數拿到一個 Int 後，會傳回一個型別為 $\text{Int} \rightarrow \text{Int}$ 的函數。
- *smaller* 3 的型別是 $\text{Int} \rightarrow \text{Int}$ 。這個函數還可拿一個 Int 參數，將之和 3 比大小，傳回較小的那個。
- *smaller* 3 4 是一個 Int 。它其實是將函數 *smaller* 3 作用在 4 之上。也就是說，*smaller* 3 4 其實應看成 (*smaller* 3) 4。根據定義，它可展開為 `if 3 ≤ 4 then 3 else 4`，然後化簡為 3。

習題 1.4 — 將 *smaller* 的定義鍵入一個檔案，載入 Haskell 直譯器中。

1. *smaller 3 4* 的型別是什麼？在 GHCi 中可用 `:t e` 指令得到算式 *e* 的型別。
2. *smaller 3* 的型別是什麼？
3. 在檔案中定義 *st3 = smaller 3*. 函數 *st3* 的型別是什麼？
4. 給 *st3* 一些參數，觀察其行為。

「用『傳回函數的函數』模擬『接收多個參數的函數』」這種做法稱作 *currying*.⁶ Haskell 鼓勵大家使用 *currying* — 它的內建函數大多依此配合設計，語法設計上也很給 *currying* 方便。當型別中出現連續的 (\rightarrow) 時，預設為往右邊結合，例如 $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ 應看成 $\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$. 這使得「傳回函數的函數」容易寫。而在值的層次，連續的函數應用往左結合。例如，*(smaller 3) 4* 可寫成 *smaller 3 4*。這讓我們能很方便地將參數一個個餵給 *curried* 函數。

另一方面，如果我們想使用 *double* 兩次，計算 *x* 的四倍，應寫成 *double (double x)*. 若寫 *double double x*，會被視為 *(double double) x* — *double* 作用在自身之上，而這顯然是個型別錯誤。

我們再看一個使用 *currying* 的練習：

例 1.4. 給定 *a, b, c, x*，下述函數 *poly* 計算 $ax^2 + bx + c$:

```
poly :: Int → Int → Int → Int → Int
poly a b c x = a × x × x + b × x + c
```

請定義一個函數 *poly₁*，使得 $\text{poly}_1 x = x^2 + 2x + 1$. 函數 *poly₁* 需使用 *poly*.

答. 一種作法是：

```
poly1 :: Int → Int
poly1 x = poly 1 2 1 x
```

但這相當於 $\text{poly}_1 x = (\text{poly } 1 \ 2 \ 1) x$ — *poly* 拿到 *x* 後，立刻把 *x* 傳給 *poly 1 2 1* 這個函數。因此 *poly₁* 可更精簡地寫成：

```
poly1 :: Int → Int
poly1 = poly 1 2 1
```

兩種寫法都有人使用。有提及 *x* 的寫法著重於描述拿到參數 *x* 之後要對它進行什麼操作。而省略 *x* 的寫法則是在函數的層次上思考：我們要定義一個函數，稱作 *poly₁*。這個函數是什麼呢？就是 *poly* 拿到 1, 2, 1 之後傳回的那個函數。

如果我們想用 *poly* 定義出另一個函數 $\text{poly}_2 a b c = a \times 2 + b \times 2 + c$ 呢？最好理解的可能是 *poly₂* 的寫法：

```
poly2 :: Int → Int → Int → Int
poly2 a b c = poly a b c 2
```

我們可以用一些技巧使 *a, b*, 和 *c* 不出現在定義中，但如此得到的程式並不會更好懂。 □

⁶Currying 為動名詞，形容詞則為 *curried*。此詞來自於邏輯學家 Haskell B. Curry 的姓氏。詳見第 1.11 節。

二元運算子 在進入其他主題前，我們交代一些語法細節。Haskell 鼓勵 currying 的使用，也把二元運算子都設計成 curried 的。例如加法的型別是 $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ 。Haskell 也配套設計了種種關於二元運算子的特殊語法，希望讓它們更好用。但這些語法規則的存在都僅是為了方便我們寫出（主觀上）語法漂亮的程式，而不是非有不可、非學不可的規定。

假設某二元運算子 (\oplus) 的型別是 $a \rightarrow b \rightarrow c$ ， $(x\oplus)$ 是給定了 (\oplus) 的第一個參數後得到的函數； $(\oplus y)$ 則是給定了 (\oplus) 的第二個參數後得到的函數：⁷

$$\begin{aligned} (x\oplus)y &= x\oplus y & \{ (x\oplus) \text{ 的型別為 } b \rightarrow c; \} \\ (\oplus y)x &= x\oplus y & \{ (\oplus y) \text{ 的型別為 } a \rightarrow c. \} \end{aligned}$$

例如：

- $(2\times)$ 和 $(\times 2)$ 都是把一個數字乘以二的函數；
- $(/2)$ 則把輸入數字除以二；
- $(1/)$ 計算輸入數字的倒數。

名字以英文字母開頭的函數預設為前序的。例如，計算餘數的函數 *mod* 使用時得寫成 *mod 6 4*。若把它放在「倒引號 (backquote)」中，表示將其轉為中序 — 如果我們比較喜歡把 *mod* 放到中間，可以寫成 *6`mod`4*。首字元非英文字母的函數（如 $(+)$, $(/)$ 等）則會被預設為中序的二元運算子。若把一個中序二元運算子放在括號中，表示將其轉為前序運算子。例如， $(+) 1 2$ 和 $1+2$ 的意思相同。

在 Haskell 的設計中，函數應用的優先順序比中序運算子高。因此 *double 3 + 4* 會被視作 $(\text{double } 3) + 4$ ，而不是 $\text{double } (3 + 4)$ 。將中序運算子放在括號中也有「讓它不再是個中序運算子，只是個一般識別字」的意思。例如算式 $f + x$ 中， f 和 x 是中序運算子 $(+)$ 的參數。但在 $f (+) x$ 中， $(+)$ 和 x 都是 f 的參數（這個式子可以讀解為 $(f (+)) x$ ）。

以函數為參數 下述函數 *square* 計算輸入的平方：

```
square :: Int → Int
square x = x × x .
```

我們可另定義一個函數 *quad* :: $\text{Int} \rightarrow \text{Int}$ ，把 *square* 用兩次，使得 *quad x* 算出 x^4 。

```
quad :: Int → Int
quad x = square (square x) .
```

但，「把某函數用兩次」是個常見的編程模式。我們能不能把 *quad* 與 *square* 抽象掉，單獨談「用兩次」這件事呢？下面的函數 *twice* 把參數 f 在 x 之上用兩次：

```
twice :: (a → a) → (a → a)
twice f x = f (f x) .
```

⁷根據 Hudak et al. [2007]，此種「切片」(sectioning) 語法最早見於 David Wile 的博士論文。後來被包括 Richard Bird 在內的 IFIP WG 2.1 成員使用，並由 David A. Turner 實作在他的語言 Miranda 中

有了 *twice*, 我們可以這麼定義 *quad*:

```
quad :: Int → Int
quad = twice square .
```

函數 *twice* 是本書中第一個「以函數為參數」的函數。我們可看到「讓函數可作為參數」對於抽象化是有益的：我們可以把「做兩次」這件事單獨拿出來說，把「做什麼」抽象掉。

「函數可以當作參數」意味著我們可以定義作用在函數上的運算子。*twice* 就是這麼一個運算子：它拿一個函數 *f*，把它加工一下，做出另一個函數（後者的定義是把 *f* 用兩次）。

參數式多型 函數 *twice* 也是本書中第一個多型函數。在 Haskell 的型別中，小寫開頭的識別字（如其中的 *a*）是型別層次的參數。讀者可想像成在 *twice* 的型別最外層有一個省略掉的 $\forall a.$ 也就是說，*twice* 的完整型別是 $\forall a. (a \rightarrow a) \rightarrow (a \rightarrow a)$ — 對所有的型別 *a*, *twice* 都可拿一個型別為 $a \rightarrow a$ 的函數，然後傳回一個型別為 $a \rightarrow a$ 的函數。

在 *twice* 的型別 $(a \rightarrow a) \rightarrow (a \rightarrow a)$ 中，

- 第一個 $a \rightarrow a$ 是參數 *f* 的型別，
- 在第二個 $a \rightarrow a$ 中，第一個 *a* 是參數 *x* 的型別，
- 第二個 *a* 則是整個計算結果的型別。

參數 *f* 的型別必須是 $a \rightarrow a$ ：輸出入型別必須一樣，因為 *f* 的結果必須可當作 *f* 自己的輸入。

在 *twice* 被使用時，型別參數 *a* 會依照上下文被特化 (*instantiate*) 成別的型別。例如 *twice square* 中，因為 *square* 的型別是 $\text{Int} \rightarrow \text{Int}$ ，這一個 *twice* 的型別變成了 $(\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int})$ — *a* 被特化成 Int 。若某函數 *k* 的型別是 $\text{Float} \rightarrow \text{Float}$ ，在 *twice k* 中，*twice* 的型別是 $(\text{Float} \rightarrow \text{Float}) \rightarrow (\text{Float} \rightarrow \text{Float})$ 。同一個函數 *twice* 可能依其上下文而有許多不同的型別，但都是 $(a \rightarrow a) \rightarrow (a \rightarrow a)$ 的特例。「一段程式可能有許多不同型別」的現象稱作多型 (*polymorphism*)。多型又有許多種類，此處為其中一種。詳情見...[todo:]

例 1.5. 考慮下述的函數 *forktimes*:

```
forktimes f g x = f x × g x .
```

算式 *forktimes f g x* 把 *f x* 和 *g x* 的結果乘起來。

1. 請想想 *forktimes* 的型別該是什麼？
2. 試定義函數 *compute :: Int → Int*，使用 *forktimes* 計算 $x^2 + 3 \times x + 2$ 。提示：
 $x^2 + 3 \times x + 2 = (x + 1) \times (x + 2)$.

答. 如同 *twice*, *forktimes* 可以有很多型別，但都應該是 $(a \rightarrow \text{Int}) \rightarrow (a \rightarrow \text{Int}) \rightarrow a \rightarrow \text{Int}$ 的特例：在 *forktimes f g x* 中，*f* 和 *g* 的型別可以是 $a \rightarrow \text{Int}$ ，其中 *a* 可以是任何型別 *a*，而 *x* 的型別必須也是同一個 *a*。函數 *compute* 可定義如下：

```
compute :: Int → Int
compute = forktimes (+1) (+2) .
```

其中 *forktimes* 型別中的 *a* 被特化為 Int 。 □

如前所述， $\text{forktimes } f \ g \ x$ 把 $f \ x$ 和 $g \ x$ 的結果乘起來。但，一定得是乘法嗎？我們當然可以再多做一點點抽象化。

例 1.6. 考慮函數 $\text{lift}_2 \ h \ f \ g \ x = h \ (f \ x) \ (g \ x)$.

1. lift_2 的型別是什麼？
2. 用 lift_2 定義 forktimes .
3. 用 lift_2 計算 $x^2 + 3 \times x + 2$.

答. 我們把 lift_2 最泛用的型別和其定義重複如下：

$$\begin{aligned} \text{lift}_2 &:: (a \rightarrow b \rightarrow c) \rightarrow (d \rightarrow a) \rightarrow (d \rightarrow b) \rightarrow d \rightarrow c \ . \\ \text{lift}_2 \ h \ f \ g \ x &= h \ (f \ x) \ (g \ x) \ . \end{aligned}$$

有了 lift_2 , forktimes 可定義為：

$$\begin{aligned} \text{forktimes} &:: (a \rightarrow \text{Int}) \rightarrow (a \rightarrow \text{Int}) \rightarrow a \rightarrow \text{Int} \\ \text{forktimes} &= \text{lift}_2 \ (\times) \ , \end{aligned}$$

請讀者觀察： lift 型別中的 a, b, c 都特化成 Int , d 則改名為 a .

我們也可用 lift_2 定義 compute :

$$\begin{aligned} \text{compute} &:: \text{Int} \rightarrow \text{Int} \\ \text{compute} &= \text{lift}_2 \ (\times) \ (+1) \ (+2) \ . \end{aligned}$$

函數 lift_2 可以看作一個作用在二元運算子上的運算子，功用是把二元運算子「提升」到函數層次。例如，原本 (\times) 只能拿兩個 Int 當作參數，(例： 1×2 是「把 1 和 2 乘起來」)，但現在 $\text{lift}_2 \ (\times)$ 可將函數 $(+1)$ 和 $(+2)$ 當參數了，意思為「把 $(+1)$ 和 $(+2)$ 的結果乘起來」。

1.4 函數合成

拿到一個函數 f ，我們能做的基本操作包括把 f 作用在某個參數上、把 f 傳給別的函數... 此外，另一個常用的基本操作是將 f 和別的函數合成 (compose) 為一個新函數。⁸ 「合成」運算子在 Haskell 中寫成 (\cdot) 。這個運算子的形式定義如下（我們先看定義本體，待會兒再看型別）：

$$(f \cdot g) \ x = f \ (g \ x) \ .$$

若用口語說， $f \cdot g$ 是將 f 和 g 兩個函數「串起來」得到的新函數：輸入 x 先丟給 g ，後者算出的結果再傳給 f 。

例 1.7. $\text{square} \cdot \text{double}$ 與 $\text{double} \cdot \text{square}$ 都是由 Int 到 Int 的函數。直覺上，前者把輸入先給 double ，其結果再給 square 。後者則反過來。如何了解它們的行為？既然它們是函數，我們便餵給它們一個參數，看看會展開成什麼！兩者分別展開如下：

⁸Robert Glück 認為函數上應有三個基本操作：函數應用、函數合成、以及求一個函數的反函數。前兩者已經提到，第三者則是大部分語言欠缺的。

$$\begin{array}{ll}
(\text{square} \cdot \text{double}) x & (\text{double} \cdot \text{square}) x \\
= \{ (\cdot) \text{ 的定義} \} & = \{ (\cdot) \text{ 的定義} \} \\
\text{square} (\text{double } x) & \text{double} (\text{square } x) \\
= (x+x) \times (x+x) , & = (x \times x) + (x \times x) .
\end{array}$$

所以，如果輸入為 x ， $(\text{square} \cdot \text{double}) x$ 計算 $(2x)^2$ ； $(\text{double} \cdot \text{square}) x$ 則是 $2x^2$ 。

但，並不是所有函數都可以串在一起： $f \cdot g$ 之中， g 的輸出型別和 f 的輸入型別必須一致才行。運算子 (\cdot) 包括型別的完整定義為：

$$\begin{array}{l}
(\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c \\
(f \cdot g) x = f (g x) .
\end{array}$$

如果 g 的型別是 $a \rightarrow b$ ， f 的型別是 $b \rightarrow c$ ，將他們串接起來後，便得到一個型別為 $a \rightarrow c$ 的函數。

有了 (\cdot) ，函數 twice 可以定義如下：

$$\begin{array}{l}
\text{twice} :: (a \rightarrow a) \rightarrow (a \rightarrow a) \\
\text{twice } f = f \cdot f .
\end{array}$$

確實，根據 (\cdot) 的定義， $\text{twice } f x = (f \cdot f) x = f (f x)$ ，和 twice 原來的定義相同。為了討論函數合成的性質，我們先介紹一個函數 id ：

$$\begin{array}{l}
\text{id} :: a \rightarrow a \\
\text{id } x = x .
\end{array}$$

它又稱作單位函數或恆等函數。這是一個似乎沒有在做什麼的函數：給任何輸入， id 都原封不動地把它送到輸出——這也反映在他的型別 $a \rightarrow a$ 上。這個函數有什麼重要性呢？原來， (\cdot) 滿足結合律，並且以 id 為單位元素（這也是「單位函數」這名字的由來）：

$$\begin{array}{l}
\text{id} \cdot f = f = f \cdot \text{id} , \\
(f \cdot g) \cdot h = f \cdot (g \cdot h) .
\end{array}$$

用數學術語來說的話， id 與 (\cdot) 形成一個幺半群 (*monoid*)。函數 id 的重要性就如同 0 在代數中的重要性（ 0 與 $(+)$ 也是一個幺半群）。我們在許多計算、證明中都會見到它。

以下我們試著證明 (\cdot) 的結合律。我們想論證 $(f \cdot g) \cdot h = f \cdot (g \cdot h)$ ，但該如何下手？該等式的等號左右兩邊都是函數。當我們說兩個整數相等，意思很清楚：如果等號左邊是 0 ，右邊也是 0 ；如果左邊是 1 ，右邊也是 1 ... 但說兩個函數「相等」，是什麼意思呢？

定義 1.8 (外延相等 (extensional equality)). 給定兩個型別相同的函數 f 和 g ，當我們說它們外延相等 (extensionally equal)，意思是給任何一個輸入， f 和 g 都算出相等的輸出。也就是： $(\forall x. f x = g x)$ 。

本書中，當我們寫兩個函數相等 ($f = g$) 時，指的便是外延相等，除非例外註明。

在外延相等的假設下，證明 $(f \cdot g) \cdot h = f \cdot (g \cdot h)$ 也就是證明對任何一個 x , $((f \cdot g) \cdot h) x = (f \cdot (g \cdot h)) x$ 均成立。我們推論如下：

$$\begin{aligned}
 & ((f \cdot g) \cdot h) x \\
 = & \{ (\cdot) \text{ 的定義} \} \\
 & (f \cdot g) (h x) \\
 = & \{ (\cdot) \text{ 的定義} \} \\
 & f (g (h x)) \\
 = & \{ (\cdot) \text{ 的定義} \} \\
 & f ((g \cdot h) x) . \\
 = & \{ (\cdot) \text{ 的定義} \} \\
 & (f \cdot (g \cdot h)) x .
 \end{aligned}$$

既然 $(f \cdot g) \cdot h = f \cdot (g \cdot h)$ ，我們便可統一寫成 $f \cdot g \cdot h$ ，不用加括號了。

習題 1.5 — 證明 $id \cdot f = f = f \cdot id$ 。

合成 (\cdot) 也是一個中序運算子。和其他中序運算子一樣，其優先性低於函數應用。因此，當我們寫 $f \cdot g x$ ，指的是 $f \cdot (g x)$ — $g x$ 為一個函數，和 f 合成，而不是 $(f \cdot g) x$ （後者根據 (\cdot) 的定義，是 $f(g x)$ ）。

例 1.9. 下列程式中，有些是合法的 *Haskell* 式子、有些則有型別錯誤。對每個程式，如果它是合法的，請找出它的型別，並說說看該程式做什麼。如果有型別錯誤，請簡述為什麼。

1. `square · smaller 3`;
2. `smaller 3 · square`;
3. `smaller (square 3)`;
4. `smaller · square 3`.

答. 1. — 3. 都是 $\text{Int} \rightarrow \text{Int}$ 。

1. 根據 (\cdot) 的定義， $(\text{square} \cdot \text{smaller } 3) x = \text{square} (\text{smaller } 3 x)$ 。因此 `square · smaller 3` 是一個函數，將其輸入和 3 比較，取較小者的平方。
2. $(\text{smaller } 3 \cdot \text{square}) x = \text{smaller } 3 (\text{square } x)$ 。因此它讀入 x ，並在 3 或 $x \uparrow 2$ 之中選較小的那個。
3. `smaller (square 3)` 是一個函數，讀入 x 之後，選擇 x 與 $3 \uparrow 2$ 之中較小的那個。
4. `smaller · square 3` 有型別錯誤：`square 3` 不是一個函數（而是一個整數），無法和 `smaller` 合成。

□

函數應用運算子 本書中有些時候會將許多函數組合成一串，例如 `square · double · (+1) · smaller 3`。由於函數應用的優先順序比一般二元運算元高，把上述式子應用在參數 5 之上時得寫成

$$(\text{square} \cdot \text{double} \cdot (+1) \cdot \text{smaller } 7) 5 ,$$

(這個式子的值為 $(2 \times (5 + 1))^2$)。每次都得加一對括號似乎有些累贅。Haskell 另有一個運算子 $(\$)$, 唸作 “apply”, 代表函數應用：

$$\begin{aligned}(\$) &:: (a \rightarrow b) \rightarrow a \rightarrow b \\ f \$ x &= f x \end{aligned}$$

$f \$ x$ 和 $f x$ 意思一樣。那麼我們為何需要這個運算子呢？原因之一是 $(\$)$ 的優先度比 (\cdot) 低，因此上式可省去括號改寫如下：

$$square \cdot double \cdot (+1) \cdot smaller \$ 5 \text{ .}$$

運算子 $(\$)$ 的另一個重要意義是：「函數應用」這個動作有了符號，成為可以獨立討論的事物。例如， $(\$)$ 可以當作參數。一個這麼做的例子是習題 1.10。

常量函數 既然介紹了 *id*, 本節也順便介紹一個以後將使用到的基本組件。給定 x 之後，函數 *const x* 是一個不論拿到什麼函數，都傳回 x 的函數。函數 *const* 的定義如下：

$$\begin{aligned}const &:: a \rightarrow b \rightarrow a \\ const \ x \ y &= x \end{aligned}$$

第 1.1 節開頭的範例 *three* 可定義為 *three = const 3*。

「無論如何都傳回 x 」聽來好像是個沒用的函數，但和 *id* 一樣，我們日後會看到它在演算、證明中時常用上。事實上，組件邏輯理論告訴我們：所有函數都可以由 *id*, *const*, 和下述的 *subst* 三個函數組合出來。

$$\begin{aligned}subst &:: (a \rightarrow b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c) \\ subst \ f \ g \ x &= f \ x \ (g \ x) \end{aligned}$$

1.5 λ 算式

雖說函數是一級市民，在本書之中，目前為止，仍有一項功能是所有其他型別擁有、函數卻還沒有的：寫出一個未命名的值的能力。整數、真假值都能不經命名、直接寫在算式中，例如，我們可寫 *smaller (square 3) 4*, 而不需要先定義好

$$\begin{aligned}num_1, num_2 &:: \text{Int} \\ num_1 &= 3 \\ num_2 &= 4 \end{aligned}$$

才能說 *smaller (square num₁) num₂*。但使用函數時，似乎非得先給個名字，才能使用它：

$$\begin{aligned}square &:: \text{Int} \rightarrow \text{Int} \\ square \ x &= \dots \text{ ,} \\ quad &= twice \ square \end{aligned}$$

如果為了某些原因（例如，如果在我們的程式中 *square* 只會被用到一次），我們不想給 *square* 一個名字，我們能不能直接把它寫出來呢？

λ 算式便是允許我們這麼做的語法。以直觀的方式解釋， $\lambda x \rightarrow e$ 便是一個函數，其中 x 是參數， e 是函數本體。例如， $(\lambda x \rightarrow x \times x)$ 是一個函數，計算其輸入 (x) 的平方。如果我們不想給 *square* 一個名字，我們可將 *quad* 定義為：

$$quad = twice (\lambda x \rightarrow x \times x) .$$

寫成 λ 算式的函數也可直接作用在參數上，例如 $(\lambda x \rightarrow e_1) e_2$ 。這個式子歸約的結果通常表示為 $e_1 [e_2/x]$ ，意思是「將 e_1 之中的 x 代換為 e_2 」。例如，算式 $(\lambda x \rightarrow x \times x) (3+4)$ 可歸約為 $(3+4) \times (3+4)$ 。

例 1.10. 以下是一些 λ 算式的例子：

- 函數 $(\lambda x \rightarrow 1+x)$ 把輸入遞增 — 和 $(1+)$ 相同。其實，把 $(1+)$ 的語法糖去掉後，得到的就是這個 λ 算式。
- $(\lambda x \rightarrow \lambda y \rightarrow x + 2 \times x \times y)$ 是一個傳回 λ 算式的函數。
 - $(\lambda x \rightarrow \lambda y \rightarrow x + 2 \times x \times y) (3+4)$ 可歸約為 $(\lambda y \rightarrow (3+4) + 2 \times (3+4) \times y)$ 。注意 λx 不見了，函數本體中的 x 被代換成 $3+4$ ， $\lambda y \rightarrow ..$ 則仍留著。
 - $(\lambda x \rightarrow \lambda y \rightarrow x + 2 \times x \times y) (3+4) 5$ 可歸約為 $(3+4) + 2 \times (3+4) \times 5$ 。
- 由於傳回函數的函數是常見的，*Haskell* (如同 *λ -calculus*) 提供較短的語法。上述例子中的函數也可簡寫成： $(\lambda x y \rightarrow x + 2 \times x \times y)$ 。
- 函數也可以當參數。例如， $(\lambda x \rightarrow x 3 3) (+)$ 可歸約為 $(+) 3 3$ ，或 $3+3$ 。
- 以下是 $(\lambda f x \rightarrow f x x) (\lambda y \rightarrow 2 \times y) 3$ 的求值過程：

$$\begin{aligned} & (\lambda f x \rightarrow f x x) (\lambda y \rightarrow 2 \times y + z) 3 \\ &= (\lambda x \rightarrow (\lambda y \rightarrow 2 \times y + z) x x) 3 \\ &= (\lambda y \rightarrow 2 \times y + z) 3 3 \\ &= 2 \times 3 + 3 \\ &= 9 . \end{aligned}$$

- 在 $\lambda x \rightarrow e$ 之中， x 是範圍限於 e 的區域識別字。因此：

$$\begin{aligned} & (\lambda f x \rightarrow x + f x) (\lambda x \rightarrow x + x) 3 \\ &= (\lambda x \rightarrow x + (\lambda x \rightarrow x + x) x) 3 \\ &= 3 + (\lambda x \rightarrow x + x) 3 \\ &= 3 + 3 + 3 \\ &= 9 . \end{aligned}$$

有了 λ 算式後，函數 *smaller* 又有另一種寫法：

$$\begin{aligned} & smaller :: Int \rightarrow Int \rightarrow Int \\ & smaller = \lambda x y \rightarrow \text{if } x \leq y \text{ then } x \text{ else } y . \end{aligned}$$

事實上， λ 算式可視為更基礎的機制 — 目前為止我們所介紹的種種語法結構都僅是 λ 算式的語法糖，都可展開、轉譯為 λ 算式。*Haskell* 的 λ 算式源於

一套稱為 λ 演算 (λ calculus) 的形式語言 — 這是一個為了研究計算本質而發展出的理論，也是函數語言的理論核心。我們將在爾後的章節中做更詳盡的介紹。[\[todo: which?\]](#)

1.6 簡單資料型態

藉由一些例子，我們已經看過 Haskell 的一些數值型別：`Int`、`Float` 等等。在本節中我們將簡短介紹我們將用到的一些其他型別。

1.6.1 布林值

布林值 (Boolean) 常用於程式中表達真和假。在 Haskell 中，我們可假想有這樣的一個型別定義：

```
data Bool = False | True .
```

其中，**data** 是 Haskell 宣告新資料型別的保留字。上述定義可用口語描述成「定義一個稱作 `Bool` 的新資料型別，有兩個可能的值，分別為 `False` 和 `True`。」`False` 和 `True` 是型別 `Bool` 的唯二兩個建構元 — 任何型別為 `Bool` 的值，如果有正規式，必定是它們兩者之一。在 Haskell 之中，建構元必須以大寫英文字母或冒號 (`:`) 開頭。

樣式配對 有了資料，我們來看看怎麼定義該型別上的函數。以布林值為輸入的函數中，最簡單又常用的可能是 `not`:

```
not :: Bool → Bool
not False = True
not True  = False .
```

這和我們的直覺理解一致：`not False` 是 `True`，`not True` 是 `False`。我們看到這個定義寫成兩行（正式說來是兩個「子句」），每一個子句分別對應到 `Bool` 的一個可能的值。以下則是邏輯上的「且」和「或」（分別寫作 \wedge 與 \vee ）的定義：⁹

```
 $\wedge, \vee$  :: Bool → Bool → Bool
False  $\wedge$  y = False
True   $\wedge$  y = y ,
False  $\vee$  y = y
True   $\vee$  y = True .
```

運算子 \wedge 與 \vee 的定義同樣是各兩個子句，每個子句分別考慮其第一個參數的值。以 $x \wedge y$ 為例：如果 x 是 `False`，不論 y 的值為何， $x \wedge y$ 都是 `False`；如果 x 是 `True`， $x \wedge y$ 的值和 y 相同。 \vee 的情況類似。

⁹邏輯「且」又稱作合取 (conjunction)；邏輯「或」又稱作析取 (disjunction)。在 Haskell 中，「且」與「或」需分別寫成 `(&&)` 和 `(||)`。本書中採用數學與邏輯領域較常使用的 \wedge 與 \vee 。

例 1.11. 以下函數判斷給定年份 y 是否為閏年。

```
leapyear :: Int → Bool
leapyear y = (y `mod` 4 == 0) ∧
              (y `mod` 100 /= 0 ∨ y `mod` 400 == 0) .
```

我們來算算看 `leapyear 2016`。依照定義展開為

$$(2016 \text{ `mod` } 4 == 0) \wedge (2016 \text{ `mod` } 100 \neq 0 \vee 2016 \text{ `mod` } 400 == 0) .$$

接下來該怎麼做呢？函數 (\wedge) 的定義有兩個子句，我們得知道 `2016 `mod` 4 == 0` 的值才能得知該歸約成哪個。因此只好先算 `2016 `mod` 4 == 0`，得到 `True`：

$$\text{True} \wedge (2016 \text{ `mod` } 100 \neq 0 \vee 2016 \text{ `mod` } 400 == 0) ,$$

然後依照 (\wedge) 的定義歸約為 `2016 `mod` 100 /= 0 ∨ 2016 `mod` 400 == 0`。接下來也依此類推。

我們發現這是第 1.1 節中所提及的被迫求值的例子：我們得先把參數算出，才知道接下來如何走。函數 `not`, (\wedge) , (\vee) 定義成許多個子句，每個都分析其參數的可能外觀，據此決定該怎麼走。這種定義方式稱作樣式配對 (*pattern matching*)：等號左手邊的 `False`, `True` 等等在此是樣式 (*pattern*)。使用這些函數時，例如 $x \wedge y$ 中， x 得先被算到可以和這些樣式配對上的程度，才能決定接下來的計算如何進行。

樣式配對也可用在不止一個參數上。例如，以下的運算元 $(==)$ 判斷兩個布林值是否相等。

```
(==) :: Bool → Bool → Bool
False == False = True
False == True  = False
True  == False = False
True  == True  = True .
```

讀者可能注意到我們用了同一個符號 $(==)$ 來表示整數與布林值的相等測試。請讀者暫且接受，相信 Haskell 有某些方式可得知任一個算式中的 $(==)$ 到底是什麼型別的相等。詳情 [\[todo: where?\]](#)

Haskell 中另有一個專用來做樣式配對的 `case` 算式。例如， (\wedge) 也可寫成如下的形式：

```
(\wedge) :: Bool → Bool
x \wedge y = case x of
  False → False
  True  → y .
```

由於 `case` 是算式，如同 `let` 一樣可出現在其他算式中，也可巢狀出現。

習題 1.6 — 以 `case` 算式定義 `not`, `(∨)`, 和 `(=)`.

習題 1.7 — 另一個定義 `(=) :: Bool → Bool → Bool` 的方式是

$$x == y = (x \wedge y) \vee (\text{not } x \wedge \text{not } y) .$$

請將 $(x,y) := (\text{False}, \text{False})$, $(x,y) := (\text{False}, \text{True})$ 等四種可能分別代入化簡，看看是否和本節之前的 `(=)` 定義相同。

1.6.2 字元

我們可把「字元」這個型別想成一個很長的 `data` 宣告：

```
data Char = 'a' | 'b' | ... | 'z' | 'A' | 'B' .....
```

其中包括所有字母、符號、空白... 目前的 Haskell 甚至有處理 Unicode 字元的能力。但無論如何，`Char` 之中的字元數目是有限的。我們可用樣式配對定義字元上的函數。注意：字元以單引號括起來。

我們也可假設字元是有順序的，每個字元對應到一個內碼。關於 `Char` 的常用函數中，`ord` 將字元的內碼找出，`chr` 則將內碼轉為字元：

```
ord :: Char → Int ,
chr  :: Int  → Char .
```

例 1.12. 下列函數 `isUpper` 判斷一個字元是否為大寫英文字母；`toLower` 則將大寫字母轉成小寫字母，若輸入並非大寫字母則不予以變動。

```
isUpper :: Char → Bool
isUpper c = let x = ord c in ord 'A' ≤ x ∧ x ≤ ord 'Z' ,
toLower :: Char → Char
toLower c | isUpper c = chr (ord c - ord 'A' + ord 'a')
           | otherwise = c .
```

1.6.3 序對

數學上，將兩個值（如 3 和 'a'）放在一起，就成了一個有序對 (*ordered pair*)，可寫成 $(3, 'a')$ 。之所以稱作「有序」對，因為其中兩個元素的順序是不可忽略的 — $(3, 'a')$ 與 $('a', 3)$ 是不同的有序對。另一個常見譯名是「數對」。由於我們處理的不只是數字，本書將之簡稱為「序對」。

給兩個集合 A 和 B ，從 A 之中任取一元素 x ，從 B 之中也任取一元素 y ，兩者的序對 (x,y) 形成的集合稱作 A 和 B 的笛卡兒積 (*Cartesian product*)，寫成 $A \times B$ ：

$$A \times B = \{ (x,y) \mid x \in A, y \in B \} .$$

Haskell 之中也有類似的構造。給定型別 a 與 b ，它們的序對的型別是 $(a \times b)$ 。¹⁰ 我們可以想像 Haskell 有這麼一個型別定義：

data $(a \times b) = (a, b)$.

以口語說的話， $(a \times b)$ 是一個新型別，而具有此型別的值若有範式，必定是 (x, y) 的形式，其中 x 的型別是 a ， y 的型別是 b 。¹¹ 序對的建構元寫成 $(,)$ ，型別為 $a \rightarrow b \rightarrow (a \times b)$ 。例如 $(,) 4 'b' = (4, 'b')$ 。

兩個常用的函數 *fst* 與 *snd* 分別取出序對中的第一個和第二個元素：

$$\begin{aligned} \text{fst} &:: (a \times b) \rightarrow a & \text{snd} &:: (a \times b) \rightarrow b \\ \text{fst } (x, y) &= x & \text{snd } (x, y) &= y \end{aligned}$$

函數 *fst* 與 *snd* 的定義方式也是樣式配對：輸入值必須先計算成 (x, y) 的形式。

例 1.13. 以下是一些序對與其相關函數的例子。

- $(3, 'a')$ 是一個型別為 $(\text{Int} \times \text{Char})$ 的序對。
- $\text{fst } (3, 'a') = 3$, $\text{snd } (3, 'a') = 'a'$
- 函數 *swap* 將序對中的元素調換：

$$\begin{aligned} \text{swap} &:: (a \times b) \rightarrow (b \times a) \\ \text{swap } (x, y) &= (y, x) \end{aligned}$$

另一個定義方式是 $\text{swap } p = (\text{snd } p, \text{fst } p)$ 。但這兩個定義並不盡然相同。詳見第 1.7 節。

序對也可以巢狀構成。例如 $((\text{True}, 3), 'c')$ 是一個型別為 $((\text{Bool} \times \text{Int}) \times \text{Char})$ 的序對，而 $\text{snd } (\text{fst } ((\text{True}, 3), 'c')) = 3$ 。在 Haskell 之中， $((a \times b) \times c)$ 與 $(a \times (b \times c))$ 被視為不同的型別，但他們是同構的——我們可定義一對函數在這兩個型別之間作轉換：

$$\begin{aligned} \text{assocr} &:: ((a \times b) \times c) \rightarrow (a \times (b \times c)) \\ \text{assocr } ((x, y), z) &= (x, (y, z)) \text{ ,} \\ \text{assocl} &:: (a \times (b \times c)) \rightarrow ((a \times b) \times c) \\ \text{assocl } (x, (y, z)) &= ((x, y), z) \text{ ,} \end{aligned}$$

並且滿足 $\text{assocr} \cdot \text{assocl} = \text{id}$ ，和 $\text{assocl} \cdot \text{assocr} = \text{id}$ 。

習題 1.8 — 試試看不用樣式配對，而以 *fst* 和 *snd* 定義 *assocl* 和 *assocr*:

$$\begin{aligned} \text{assocl } p &= \dots \\ \text{assocr } p &= \dots \end{aligned}$$

另外可一提的是，Haskell 允許我們在 λ 算式中做樣式配對。例如 *fst* 的另一種寫法是：

¹⁰然而，由於「程式可能不終止」這個因素作怪， $a \times b$ 的元素並不僅是 a 與 b （如果視做集合）的笛卡兒積。詳見 [todo: where?]

¹¹其實這個定義並不符合 Haskell 的語法，因此只是方便理解的想像。另，型別 $(a \times b)$ 在 Haskell 中寫成 (a, b) 。我的經驗中，讓型別與值的語法太接近，反易造成困惑。

同構

兩個集合 A 與 B 同構 (isomorphic)，意思是 A 之中的 每個元素都唯一地對應到 B 之中的一個元素，反之亦然。

一個形式定義是： A 與 B 同構意謂我們能找到兩個全 (total) 函數 $to :: A \rightarrow B$ 和 $from :: B \rightarrow A$ ，滿足

$$\begin{aligned} from \cdot to &= id \text{ ,} \\ to \cdot from &= id \text{ .} \end{aligned}$$

此處的兩個 id 型別依序分別為 $A \rightarrow A$ 和 $B \rightarrow B$ 。將定義展開，也就是說，對所有 $x :: A$ ， $from (to x) = x$ ；對所有 $y :: B$ ， $to (from y) = y$ 。這個定義迫使對每個 x 都存在一個唯一的 $to x$ ，反之亦然。

我們已有兩個例子： $((a \times b) \times c)$ 與 $(a \times (b \times c))$ 同構，此外， $(a \times b)$ 與 $(b \times a)$ 也同構，因為 $swap \cdot swap = id$ 。

如果集合 A 與 B 同構，不僅 A 之中的每個元素都有個在 B 之中相對的元素，給任一個定義在 A 之上的函數 f ，我們必可構造出一個 B 之上的函數，具有和 f 相同的性質。即使 A 與 B 並不真正相等，我們也可把它們視為基本上沒有差別的。在許多無法談「相等」的領域中，同構是和「相等」地位一樣的觀念。

$$fst = \lambda(x,y) \rightarrow x \text{ .}$$

Haskell 另有提供更多個元素形成的有序組，例如 $(True, 3, 'c')$ 是一個型別為 $(Bool \times Int \times Char)$ 的值。但本書暫時不使用他們。

分裂與積 在我們將介紹的編程風格中，以下兩個產生序對的運算子相當好用。第一個運算子利用兩個函數產生一個序對：

$$\begin{aligned} \langle \cdot, \cdot \rangle &:: (a \rightarrow b) \rightarrow (a \rightarrow c) \rightarrow a \rightarrow (b \times c) \\ \langle f, g \rangle x &= (f x, g x) \text{ .} \end{aligned}$$

給定兩個函數 $f :: a \rightarrow b$ 和 $g :: a \rightarrow c$ ， $\langle f, g \rangle :: a \rightarrow (b \times c)$ 是一個新函數，將 f 和 g 的結果收集在一個序對中。我們借用範疇論的詞彙，將此稱作分裂 (split) — $\langle f, g \rangle$ 可讀成「 f 與 g 的分裂」。

如果我們已經有了一個序對，我們可用 $(f \times g)$ 算出一個新序對：

$$\begin{aligned} (\times) &:: (a \rightarrow b) \rightarrow (c \rightarrow d) \rightarrow (a \times c) \rightarrow (b \times d) \\ (f \times g) (x,y) &= (f x, g y) \text{ .} \end{aligned}$$

函數 $(f \times g)$ 將 f 和 g 分別作用在序對 (x,y) 的兩個元素上。這個操作稱作「 f 和 g 的乘積」，同樣是借用範疇論的詞彙。

Currying 與 Uncurrying 如前所述，Haskell 的每個函數都只拿一個參數。拿多個參數的函數可以傳回函數的函數來模擬，稱作 currying。有了序對之後，另一種模擬多參數的方式是把參數都包到一個序對中。例如，型別為 $(a \times b) \rightarrow c$ 的函數可視為拿了兩個型別為 a 與 b 的參數。

函數 `curry` 與 `uncurry` 幫助我們在這兩種表示法之間轉換 — `curry` 將拿序對的函數轉換成 curried 函數，`uncurry` 則讓 curried 函數改拿序對當作參數：

$$\begin{aligned} \text{curry} &:: (a \times b \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c) \\ \text{curry } f \ x \ y &= f \ (x, y) \ , \\ \text{uncurry} &:: (a \rightarrow b \rightarrow c) \rightarrow ((a \times b) \rightarrow c) \\ \text{uncurry } f \ (x, y) &= f \ x \ y \ . \end{aligned}$$

例：如果 $(=)$ 的型別為 $\text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}$, $\text{uncurry } (=)$ 的型別為 $(\text{Int} \times \text{Int}) \rightarrow \text{Bool}$ 。後者檢查一個序對中的兩個值是否相等（例： $\text{uncurry } (=) \ (3, 3) = \text{True}$ ）。

習題 1.9 — 事實上， curry 與 uncurry 的存在證明了 $(a \times b) \rightarrow c$ 與 $a \rightarrow b \rightarrow c$ 是同構的。試證明 $\text{curry} \cdot \text{uncurry} = \text{id}$ ，以及 $\text{uncurry} \cdot \text{curry} = \text{id}$ 。

習題 1.10 — 請說明 $\text{map } (\text{uncurry } \$)$ 的型別與功能。關於 $(\$)$ 請參考第 27 頁。

1.7 弱首範式

我們現在可把第 1.1 節中提及的求值與範式談得更仔細些。第一次閱讀的讀者可把本節跳過。回顧 fst 使用樣式配對的定義 $\text{fst } (x, y) = x$ 。假設我們把 swap 定義如下：

$$\text{swap } p = (\text{snd } p, \text{fst } p) \ .$$

考慮 $\text{fst } (\text{swap } (3, 'a'))$ 該怎麼求值：

$$\begin{aligned} & \text{fst } (\text{swap } (3, 'a')) \\ &= \{ \text{swap 的定義} \} \\ & \text{fst } (\text{snd } (3, 'a'), \text{fst } (3, 'a')) \\ &= \{ \text{fst 的定義} \} \\ & \text{snd } (3, 'a') \\ &= \{ \text{snd 的定義} \} \\ & 'a' \ . \end{aligned}$$

在第一步中，由於 fst 使用樣式配對，我們得先把 $\text{swap } (3, 'a')$ 算出來。若把 $\text{swap } (3, 'a')$ 算到底，得到的範式是 $('a', 3)$ 。但如第一步中所顯示，如果目的只是為了配對 (x, y) 這個樣式，我們並不需要把 $\text{swap } (3, 'a')$ 算完，只需算到 $(\text{snd } (3, 'a'), \text{fst } (3, 'a'))$ 即可 — x 可對應到 $\text{snd } (3, 'a')$, y 可對應到 $\text{fst } (3, 'a')$, fst 的計算便可以進行。在下一步中，子算式 $\text{fst } (3, 'a')$ 便被丟棄了，並沒有必要算出來。

做樣式配對時，Haskell 只會把算式歸約到剛好足以與樣式配對上的程度。當樣式的深度只有一層（如 (x, y) ）時，與之配對的式子會被歸約成一種稱作弱首範式（weak head normal form）的形式。弱首範式有其嚴格定義，但本書讀者只需知道：算式會被歸約到露出最外面的建構元，（在此例中是被歸約成 $(-, -)$ 的形式），然後便停下來。

例 1.14. 回顧 swap 的兩種定義方式，分別命名為

$$\begin{aligned} \text{swap}_1(x, y) &= (y, x) , \\ \text{swap}_2 p &= (\text{snd } p, \text{fst } p) , \end{aligned}$$

若考慮不終止的參數，兩者的行為並不盡然相同。定義 $\text{three}(x, y) = 3$ ，並假設 \perp 是一個沒有範式的式子——一旦開始對 \perp 求值，便停不下來。試計算 $\text{three}(\text{swap}_1 \perp)$ 和 $\text{three}(\text{swap}_2 \perp)$ 。

答. 由於 three 使用樣式配對，計算 $\text{three}(\text{swap}_1 \perp)$ 時得把 $\text{swap}_1 \perp$ 歸約成弱首範式。同理，計算 $\text{swap}_1 \perp$ 時，第一步便是先試圖把 \perp 歸約成弱首範式，然後便停不下來了。

至於 $\text{three}(\text{swap}_2 \perp)$ 則可如下地求值：

$$\begin{aligned} &\text{three}(\text{swap}_2 \perp) \\ &= \{ \text{swap}_2 \text{ 之定義} \} \\ &\quad \text{three}(\text{snd } \perp, \text{fst } \perp) \\ &= \{ \text{three 之定義} \} \\ &\quad 3 . \end{aligned}$$

第一步中， $\text{swap}_2 \perp$ 則依照範式順序求值的原則展開為 $(\text{snd } \perp, \text{fst } \perp)$ ——這是一個序對，只是該序對含有兩個沒有範式的元素。該序對可對應到樣式 (x, y) ，因此整個式子歸約為 3。

附帶一提， $\text{three } \perp$ 是一個不停止的計算。 □

1.8 串列

一個串列 (list) 抽象說來便是將零個或多個值放在一起變成一串。串列是函數語言傳統上的重要資料結構：早期的函數語言 LISP 便是 LIsT Processing 的縮寫。Haskell 中的串列多了一個限制：串列中的每個元素必須有同樣的型別。本書中將「元素型別均為 a 的串列」的型別寫成 $\text{List } a$ 。¹² Haskell 以中括號表示串列，其中的元素以逗號分隔。例如， $[1, 2, 3, 4]$ 是一個型別為 List Int 的串列，其中有四個元素； $[\text{True}, \text{False}, \text{True}]$ 是一個型別為 List Bool 的串列，有三個元素。至於 $[]$ 則是沒有元素的空串列（通常唸做“nil”），其最通用的型別為 $\text{List } a$ ，其中 a 可以是任何型別。

串列的元素也可以是串列。例如 $[[1, 2, 3], [], [4, 5]]$ 的型別是 List (List Int) ，含有三個元素，分別為 $[1, 2, 3]$ ， $[]$ ，和 $[4, 5]$ 。

事實上，上述的寫法只是語法糖。我們可想像 Haskell 有這樣的型別定義：

$$\text{data List } a = [] \mid a : \text{List } a .$$

意謂一個元素型別為 a 的串列只有兩種可能構成方式：可能是空串列 $[]$ ，也可能是一個元素 (a) 接上另一個串列 $(\text{List } a)$ 。後者的情況中，元素和串列之間用符號 $(:)$ 銜接。

符號 $(:)$ 唸作“cons”，為「建構 (construct)」的字首。其型別為 $a \rightarrow \text{List } a \rightarrow \text{List } a$ ——它總是將一個型別為 a 的元素接到一個 $\text{List } a$ 之上，造出另一個

¹²Haskell 中的寫法是 $[a]$ 。同樣地，在我的教學經驗中，將中括號同時使用在值與型別上造成不少誤解。例如學生可能認為 $[1, 2]$ 的型別是 $[\text{Int}, \text{Int}]$ ——其實應該是 $[\text{Int}]$ 。

(:) 與 (::)

大部分有型別的函數語言（如 ML, Agda 等）之中，`(:)` 表示型別關係，`::` 則是串列的建構元。Haskell 的前身之一是 David A. Turner 的語言 Miranda。在其 Hindley-Milner 型別系統中，Miranda 使用者幾乎不需寫出程式的型別 — 型別可由電腦自動推導。而串列是重要資料結構。把兩個符號調換過來，使常用的符號較短一些，似乎是合理的設計。

Haskell 繼承了 Miranda 的語法。然而，後來 Haskell 的型別發展得越來越複雜，使用者偶爾需要寫出型別來幫助編譯器。即使型別簡單，程式語言界也漸漸覺得將函數的型別寫出是好習慣。而串列建構元的使用量並不見得比型別關係多。但此時想改符號也為時已晚了。

List a. 上述的 `[1,2,3,4]` 其實是 `1:(2:(3:(4:[])))` 的簡寫：由空串列 `[]` 開始，將元素一個個接上去。為了方便，Haskell 將 `(:)` 運算元視做右相依的，因此我們可將括號省去，寫成 `1:2:3:4:[]`。

無論如何，這樣的串列表示法是偏一邊的 — 元素總是從左邊放入，最左邊的元素也最容易取出。如果一個串列不是空的，其最左邊的元素稱作該串列的頭 (*head*)，剩下的元素稱作其尾 (*tail*)。例如，`[1,2,3,4]` 的頭是 `1`，尾是 `[2,3,4]`。

Haskell 中將字串當作字元形成的串列。標準函式庫中這麼定義著：

```
type String = List Char .
```

意謂 `String` 就是 `List Char`。在 Haskell 中，`data` 用於定義新型別，而 `type` 並不產生一個新的型別，只是給現有的型別一個較方便或更顯出當下意圖的名字。此外，Haskell 另提供一個語法糖，用雙引號表達字串。因此，`"fun"` 是 `['f','u','n']` 的簡寫，後者又是 `'f':'u':'n':[]` 的簡寫。

本節接下來將介紹許多與串列相關的內建函數。

1.8.1 串列解構

我們先從拆解串列的函數開始。函數 `head` 和 `tail` 分別取出一個串列的頭和尾：

```
head :: List a → a           tail :: List a → List a
head (x:xs) = x ,           tail (x:xs) = xs .
```

注意其型別：`head` 傳回一個元素，`tail` 則傳回一個串列。例：`head "fun"` 和 `tail "fun"` 分別是字元 `'f'` 和字串 `"un"`。函數 `head` 和 `tail` 都可用樣式配對定義，但此處的樣式並不完整，尚缺 `[]` 的情況。如果將空串列送給 `head` 或 `tail`，則會出現執行時錯誤。因此，`head` 和 `tail` 都是部分函數 — 它們只將某些值（非空的串列）對應到輸出，某些值（空串列）則沒有。

函數 `null` 判斷一個串列是否為空串列。它也可用樣式配對定義如下：

```
null :: List a → Bool
null []      = True
null (x:xs) = False .
```


本書依循 Bird [1998] 中的變數命名習慣，將型別為串列的變數以 *s* 做結尾，例如 *xs*, *ys* 等等。至於「元素為串列的串列」則命名為 *xss*, *yss* 等等。但這只是為方便理解而設計的習慣。Haskell 本身並無此規定。

除了 *head* 與 *tail*，也有另一組函數 *last* 與 *init* 分別取出一個串列最右邊的元素，以及剩下的串列：

```
last :: List a → a ,
init  :: List a → List a .
```

例：*last "fun"* 與 *init "fun"* 分別為字元 'n' 與字串 "fu"。但 *last* 與 *init* 的定義比起 *head* 與 *tail* 來得複雜：記得我們的串列表示法是偏向一邊的，從左邊存取元素容易，從右邊存取元素則較麻煩。我們會在 [todo: where] 之中談到 *last* 與 *init* 的定義。

1.8.2 串列生成

第 1.8.1 節中的函數均將串列拆開。本節之中我們來看一些生成串列的方法。如果元素的型別是有順序的（例如 *Int*, *Char* 等型別），Haskell 提供了一個方便我們依序生成串列的語法。以例子說明：

例 1.15. 以下為 *Haskell* 的列舉語法的一些例子：

- $[0..10]$ 可展開為 $[0,1,2,3,4,5,6,7,8,9,10]$.
- 可用頭兩個元素來指定間隔量。例如 $[0,3..10] = [0,3,6,9]$. 注意該串列的元素不超過右界 10
- 在 $[10..0]$ 之中，10 一開始就超過了右界 0，因此得到 $[]$. 如果想要產生由 10 倒數到 0 的串列，可這樣指定間隔： $[10,9..0]$.
- 字元也是有順序的，因此 $['a'..'z']$ 可展開為含所有英文小寫字母的串列。
- 至於沒有右界的 $[0..]$ 則會展開為含 $[0,1,2,3\dots]$ 的無限長串列。

函數 $iterate :: (a \rightarrow a) \rightarrow a \rightarrow List\ a$ 用於產生無限長的串列： $iterate\ f\ x$ 可展開為 $[x, f\ x, f\ (f\ x), f\ (f\ (f\ x))\dots]$.

例 1.16. 一些 *iterate* 的例子：

- $iterate\ (1+)\ 0$ 展開為 $[0,1,2,3\dots]$. 其實 $[n..]$ 可視為 $iterate\ (1+)\ n$ 的簡寫。
- 在例 1.24 中我們會看到 $[m..n]$ 也可用 *iterate* 與其他函數做出。
- $iterate\ not\ False$ 可得到無窮串列 $[False, True, False\dots]$.

數學中描述集合時常使用一種稱作集合建構式 (set comprehension) 的語法。例如， $\{x \times x \mid x \in S, odd\ x\}$ 表示收集所有 $x \times x$ 形成的集合，其中 x 由集合 S 中取出，並且必須為奇數。Haskell 將類似的語法用在串列上。同樣以例子說明：

例 1.17. 串列建構式 (*list comprehension*) 的例子：

- $[x \mid x \leftarrow [0..9]]$ 表示「從 $[0..9]$ 之中取出 x ，並收集 x 」，可展開為 $[0,1,2,3,4,5,6,7,8,9]$.

- $[x \times x \mid x \leftarrow [0..10]]$ 的 x 來源和之前相同，但收集的是 $x \times x$ ，得到 $[0, 1, 4, 9, 25, 36, 49, 64, 81]$.
- $[(x, y) \mid x \leftarrow [0..2], y \leftarrow "abc"]$ 展開得到 $[(0, 'a'), (0, 'b'), (0, 'c'), (1, 'a'), (1, 'b'), (1, 'c'), (2, 'a'), (2, 'b'), (2, 'c')]$. 注意序對出現的順序：先固定 x ，將 y 跑過一遍，再換成下一個 x .
- $[x \times x \mid x \leftarrow [0..10], \text{odd } x]$ 從 $[0..10]$ 之中取出 x ，但只挑出滿足 $\text{odd } x$ 的那些，得到 $[1, 9, 25, 49, 81]$.

例 1.18. 以下算式的值分別為何？

1. $[(a, b) \mid a \leftarrow [1..3], b \leftarrow [1..2]]$.
2. $[(a, b) \mid b \leftarrow [1..2], a \leftarrow [1..3]]$.
3. $[(i, j) \mid i \leftarrow [1..4], j \leftarrow [(i+1)..4]]$. 這是一個有了 $i \leftarrow \dots$ 之後， i 即可在右方被使用的例子。
4. $[(i, j) \mid i \leftarrow [1..4], \text{even } i, j \leftarrow [(i+1)..4], \text{odd } j]$.
5. $['a' \mid i \leftarrow [0..10]]$. 這個例子顯示 i 並不一定非得出現在被收集項目中。

答. 分別展開如下：

1. $[(1, 1), (1, 2), (2, 1), (2, 2), (3, 1), (3, 2)]$.
2. $[(1, 1), (2, 1), (3, 1), (1, 2), (2, 2), (3, 2)]$.
3. $[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]$.
4. $[(2, 3)]$.
5. $"aaaaaaaaaa"$.

□

串列建構式在寫程式時相當好用，但它也僅是個語法糖 — 所有的串列建構式都可轉換為後面的章節將介紹的 *map*, *concat*, *filter* 等函數的組合。

1.8.3 串列上的種種組件函數

我們將在本節介紹大量與串列有關的函數。它們常被稱做組件 (*combinators*) 函數。每一個組件都負責一項單一、但具通用性而容易重用的功能。它們常用來彼此結合以組出更大的程式。

介紹這些函數有兩個原因。首先，它們可用來組出許多有趣的程式。我們將一邊逐一介紹這些函數，一邊以例子示範，同時也逐漸帶出本章鼓勵的一種特殊編程風格。另一個原因是在日後的章節中我們也都將以這些函數作為例子，討論並證明關於它們的性質。

第一次閱讀的讀者可能訝異：這麼多函數，怎麼記得住？事實上，這些組件大都僅是把常見的編程模式具體地以形式方式表達出來。辨識出這些模式後，不僅會發現它們其實很熟悉，對於我們日後了解其他程式也有助益。

長度 函數 $\text{length} :: \text{List } a \rightarrow \text{Int}$ 計算串列的長度。空串列 `[]` 的長度為 0。例：
 $\text{length } "function" = 8$.

索引 函數 $(!!)$ 的型別為 $\text{List } a \rightarrow \text{Int} \rightarrow a$. 給定串列 xs 和整數 i , 如果 $0 \leq i < \text{length } xs$, $xs !! i$ 為 xs 中的第 i 個元素, 但由 0 起算。例 `"function" !! 0 = 'f'`, `"function" !! 3 = 'c'`. 如果 $i > \text{length } xs$, 則會成為執行期錯誤。注意: 如果 $\text{length } xs = n$, 其中的元素編號分別為 $0, 1 \dots n-1$.

在指令式語言中, 索引是處理陣列常用的基本操作。處理陣列的常見模式是用一個變數 i 指向目前正被處理的元素, 將 $a[i]$ 的值讀出或覆寫, 然後更新 i 的值。但由接下來的許多範例中, 讀者會發現本章盡量避免這種做法。也因此 $(!!)$ 在本章中使用的機會不多。

連接 函數 $(++) :: \text{List } a \rightarrow \text{List } a \rightarrow \text{List } a$ 將兩個串列相接。例: `[1,2,3] ++ [4,5] = [1,2,3,4,5]`.

函數 $(++)$ 和 $(:)$ 似乎都是把串列接上東西。兩者有什麼不同呢? 答案是: $(:) :: a \rightarrow \text{List } a \rightarrow \text{List } a$ 永遠把一個元素接到串列的左邊, 而 $(++)$ 把兩個串列接在一起, 兩個串列都有可能含有零個或多個元素。例: `[] ++ [4,5] = [4,5]`. 事實上, $(:)$ 是比 $(++)$ 更基礎的操作。在第 2.4 節中, 我們會看到 $(++)$ 是用 $(:)$ 定義而成的。

另一個關於連接的函數是 $\text{concat} :: \text{List } (\text{List } a) \rightarrow \text{List } a$: 它以一個元素都是串列的串列作為輸入, 將其中的串列接在一起。例: `concat [[1,2,3], [], [4], [5,6]] = [1,2,3,4,5,6]`. 它和 $(++)$ 的不同之處在哪呢? 顯然, $(++)$ 總把兩個串列接在一起, 而 concat 的參數中可含有零個或多個串列。在第 2.4 節中, 我們會看到 concat 是用 $(++)$ 定義而成的。

取與丟 take 的型別為 $\text{Int} \rightarrow \text{List } a \rightarrow \text{List } a$. $\text{take } n \text{ } xs$ 取 xs 的前 n 個元素。若 xs 的長度不到 n , $\text{take } n \text{ } xs$ 能拿幾個就拿幾個。例: `take 3 "function" = "fun"`, `take 5 "ox" = "ox"`.

相對地, $\text{drop } n \text{ } xs$ 丟掉 xs 的前 n 個元素。若 xs 的長度不到 n , $\text{drop } n \text{ } xs$ 能丟幾個就拿幾個。例: `drop 3 "function" = "ction"`, `take 5 "ox" = ""`. 函數 drop 的型別也是 $\text{Int} \rightarrow \text{List } a \rightarrow \text{List } a$.

函數 take 和 drop 顯然有些關聯, 但它們的關聯該怎麼具體地寫下來呢? 一個可能是: 對所有的 n 和 xs ,

$$\text{take } n \text{ } xs ++ \text{drop } n \text{ } xs = xs \quad .$$

乍看之下似乎言之成理。但這個性質真的成立嗎? 我們將在第 2 章中討論到。

映射 $\text{map} :: (a \rightarrow b) \rightarrow \text{List } a \rightarrow \text{List } b$ 是串列上一個很重要的高階函數: $\text{map } f \text{ } xs$ 將 f 作用在 xs 的每一個元素上。例:

$$\begin{aligned} \text{map square } [1,2,3,4] &= [1,4,9,16] \quad , \\ \text{map } (1+) [2,3,4] &= [3,4,5] \quad . \end{aligned}$$

回憶我們之前關於高階函數的討論, 另一個理解方式是: map 是一個處理函數的操作。給一個「將 a 變成 b 」的函數 $f :: a \rightarrow b$, map 將這個函數提升到串列的層次, 得到一個「將 $\text{List } a$ 變成 $\text{List } b$ 」的函數 $\text{map } f :: \text{List } a \rightarrow \text{List } b$.

例 1.19. 如果一個串列 xs 可分解為 $ys ++ zs$ ，我們說 ys 是 xs 的一個前段 (prefix)， zs 則是 xs 的一個後段 (suffix)。例如，串列 $[1,2,3]$ 的前段包括 $[], [1], [1,2]$ ，與 $[1,2,3]$ （注意： $[]$ 是一個前段， $[1,2,3]$ 本身也是），後段則包括 $[1,2,3], [2,3], [3]$ ，與 $[]$ 。

試定義函數 $inits :: List a \rightarrow List (List a)$ ，計算輸入串列的所有前段。¹³ 提示：目前我們可以用 map ， $take$ 和其他函數組出 $inits$ 。在第 2.5 節中將會介紹另一個做法。

答. 一種使用 map 和 $take$ 的可能作法如下：

```
inits :: List a -> List (List a)
inits xs = map (\n -> take n xs) [0..length xs] .
```

或著也可用串列建構式寫成 $[take\ n\ xs \mid n \leftarrow [0..length\ xs]]$ 。讀者可發現： $[f\ x \mid x \leftarrow xs]$ 就是 $map\ f\ xs$ 。□

習題 1.11 — 定義函數 $tails :: List a \rightarrow List (List a)$ ，計算輸入串列的所有後段。

過濾 一個型別為 $a \rightarrow Bool$ 的函數稱作一個「述語」(predicate)。給定述語 p ， $filter\ p\ xs$ 將 xs 之中滿足 p 的元素挑出。函數 $filter$ 的型別為 $(a \rightarrow Bool) \rightarrow List\ a \rightarrow List\ a$ 。例： $filter\ even\ [2,5,1,7,6] = [2,6]$ 。

例 1.20. 該怎麼得知一個字串中大寫字母的個數？將大寫字母過濾出來，計算所得串列的長度即可。如下所示：

```
numUpper :: String -> Int
numUpper = length . filter isUpper .
```

例 1.21. 下列算式求出 0^2 到 50^2 的平方數（能寫成 n^2 的數字）中，結尾為 25 的數字。

```
filter ((== 25) . ('mod' 100)) (map square [0..50]) .
```

歸約後得到 $[25, 225, 625, 1225, 2025]$ 。其中 $((== 25) \cdot ('mod' 100))$ 部分使用了第 23 頁中提到的語法。如果覺得不習慣，也可用 λ 算式寫成：

```
filter (\n -> n `mod` 100 == 25) (map square [0..50]) .
```

例 1.22. 接續上例。另一個可能寫法是先過濾出「平方之後結尾為 25」的數字，再算這些數字的平方：

```
map square (filter ((== 25) . ('mod' 100) . square) [0..50]) .
```

這個算式也歸約出一樣的結果： $[25, 225, 625, 1225, 2025]$ 。

稍微推廣一些，這個例子暗示我們 $filter\ p \cdot map\ f$ 和 $map\ f \cdot filter\ (p \cdot f)$ 似乎是等價的。但確實如此嗎？我們也將在第 2 章中討論。

¹³請注意該函數的名字是 *inits*，和之前介紹過的 *init* 不同。這是 Haskell 函式庫中使用的命名。

例 1.23. 接續上例。如果我們不僅希望找到結尾為 25 的平方數，也希望知道它們是什麼數字的平方，一種寫法如下：

$$\text{filter } ((= 25) \cdot ('mod' 100) \cdot \text{snd}) (\text{map } \langle id, \text{square} \rangle [0..50])$$

我們用 $\text{map } \langle id, \text{square} \rangle$ 將每個數字與他們的平方放在一個序對中，得到 $[(0,0), (1,1), (2,4), (3,9), \dots]$ 。而 filter 的述語多了一個 snd ，表示我們只要那些「第二個元素符合條件」的序對。上式化簡後可得到 $[(5,25), (15,225), (25,625), (35,1225), (45,2025)]$ 。運算元 $\langle \cdot, \cdot \rangle$ 的定義詳見第 34 頁。

述語 $((= 25) \cdot ('mod' 100) \cdot \text{snd})$ 可以展開為 $(\lambda(i,n) \rightarrow n \cdot 'mod' 100 == 25)$ 。

取、丟、與過濾 函數 takeWhile , dropWhile 和 filter 有一樣的型別。

$$\begin{aligned} \text{takeWhile} &:: (a \rightarrow \text{Bool}) \rightarrow \text{List } a \rightarrow \text{List } a \\ \text{dropWhile} &:: (a \rightarrow \text{Bool}) \rightarrow \text{List } a \rightarrow \text{List } a \end{aligned}$$

它們之間的差異也許用例子解釋得最清楚：

$$\begin{aligned} \text{filter } \text{even } [6,2,4,1,7,8,2] &= [6,2,4,8,2] \\ \text{takeWhile } \text{even } [6,2,4,1,7,8,2] &= [6,2,4] \\ \text{dropWhile } \text{even } [6,2,4,1,7,8,2] &= [1,7,8,2] \end{aligned}$$

$\text{filter } p$ 挑出所有滿足 p 的元素； $\text{takeWhile } p$ 由左往右逐一取出元素，直到遇上第一個不滿足 p 的元素，並將剩下的串列丟棄； $\text{dropWhile } p$ 則與 $\text{takeWhile } p$ 相對，將元素丟棄，直到遇上第一個不滿足 p 的元素。直覺上，後兩者似乎也應該滿足 $\text{takeWhile } p \text{ } xs ++ \text{dropWhile } p \text{ } xs = xs$ ，但這仍尚待驗證。

例 1.24. 給定整數 m 與 n ， $[m..n]$ 可視為 $\text{takeWhile } (\leq n) (\text{iterate } (1+) m)$ 的簡寫。

例 1.25. 讀者也許覺得 takeWhile 或 dropWhile 似乎和迴圈有密切關係。確實，利用 iterate 與 dropWhile ，我們可定義出類似 while 迴圈的操作：

$$\begin{aligned} \text{until} &:: (a \rightarrow \text{Bool}) \rightarrow (a \rightarrow a) \rightarrow a \rightarrow a \\ \text{until } p \text{ } f &= \text{head} \cdot \text{dropWhile } (\text{not} \cdot p) \cdot \text{iterate } f \end{aligned}$$

$\text{until } p \text{ } f \text{ } x$ 由 x 算出 $f \text{ } x$ ，由 $f \text{ } x$ 算出 $f (f \text{ } x)$... 直到 $p (f (f \dots x))$ 成立為止。例： $\text{until } ((>50) \cdot \text{square}) (1+) 0$ 得到 8，因為 $8^2 = 64$ ，是第一個平方大於 50 的非負整數。由於惰性求值， $\text{iterate } f$ 在意義上雖然是個無窮串列，但只會被執行到 $\text{dropWhile } (\text{not} \cdot p)$ 擷取的長度為止。

下述函數則實作了用輾轉相減法求最大公因數的古典演算法。函數 minus 不斷將大數減去小數，直到兩數相等為止：

$$\begin{aligned} \text{gcd} &:: (\text{Int} \times \text{Int}) \rightarrow \text{Int} \\ \text{gcd} &= \text{fst} \cdot \text{until } (\text{uncurry } (=)) \text{ } \text{minus} \\ &\quad \text{where } \text{minus } (x,y) \mid x > y = (y, x-y) \\ &\quad \quad \mid x < y = (y-x, x) \end{aligned}$$

關於 uncurry 詳見第 34 頁。

習題 1.12 — 試定義一個函數 $squaresUpTo :: Int \rightarrow List\ Int$ ，使得 $squaresUpTo\ n$ 傳回所有不大於 n 的平方數。例： $squaresUpTo\ 10 = [1, 4, 9]$ ， $squaresUpTo\ (-1) = []$ 。

拉鍊 函數 $zip :: List\ a \rightarrow List\ b \rightarrow List\ (a \times b)$ 的作用可由下述例子示範：

```
zip [1,2,3] "abc" = [(1, 'a'), (2, 'b'), (3, 'c')] ,
zip [1,2]   "abc" = [(1, 'a'), (2, 'b')] ,
zip [1,2,3] "ab"  = [(1, 'a'), (2, 'b')] ,
zip [1..]   "abc" = [(1, 'a'), (2, 'b'), (3, 'c')] ,
zip [1..]   [2..] = [(1,2), (2,3), (3,4)..] ,
```

$zip\ xs\ ys$ 將串列 xs 與 ys 相對應的元素放在序對中。如果兩個串列長度不同， zip 將其中一個用完後即停止。 zip 也能處理無限長的串列。由於這個動作看來像是把 xs 與 ys 當作拉鍊的兩側「拉起來」，因此用拉拉鍊的狀態詞“zip”命名。

相對地，也有一個函數 $unzip :: List\ (a \times b) \rightarrow (List\ a \times List\ b)$ ，將「拉鍊」拉開。例： $unzip\ [(1, 'a'), (2, 'b'), (3, 'c')]$ 可得到 $([1,2,3], "abc")$ 。

許多情況下，我們不想要把兩兩對應的元素放到序對中，而是分別餵給一個二元運算子。這時可用另一個相關函數 $zipWith$ ，例： $zipWith\ (+)\ [1,2,3]\ [4,5,6] = [5,7,9]$ 函數 $zipWith$ 可以這樣定義：

```
zipWith :: (a -> b -> c) -> List a -> List b -> List c
zipWith f = map (uncurry f) . zip .
```

習題 1.13 — 用 $zipWith$ 定義 zip 。

例 1.26. 試定義函數 $positions :: Char \rightarrow String \rightarrow List\ Int$ ，使得 $positions\ z\ xs$ 傳回 z 在 xs 中出現的所有位置。例： $positions\ 'o'\ "hoola\ hooligans" = [1,2,7,8]$ 。

答. 一種可能寫法如下：

```
positions z = map fst . filter ((== z) . snd) . zip [0..] .
```

我們用 $zip\ [0..]$ 為輸入串列標上位置，用 $filter\ ((== z) . snd)$ 取出第二個元素等於 z 的序對，最後用 $map\ fst$ 取出所有位置。注意函數合成與 currying 的使用。□

例 1.27. 接續上例。如果我們僅想要 z 出現的第一個位置呢？我們可以定義：

```
pos :: Char -> String -> Int
pos z = head . positions z .
```

這是一個部分函數， $pos\ z\ xs$ 傳回 $positions\ z\ xs$ 的第一個結果。如果 z 沒有出現， $positions\ z\ xs$ 傳回 $[]$ ， $pos\ z\ xs$ 會得到執行期錯誤。如果 z 出現在 xs 中，由於惰性求值， pos 得到第一個位置後 $positions$ 便會停下，不會把串列整個產生。

如果我們希望 *pos* 在 *z* 沒有出現時傳回 -1 ，可以這麼做：

```
pos :: Char → String → Int
pos z xs = case positions z xs of
  []      → -1
  (i:is)  → i
```

1.9 全麥編程

讀者至此應已注意到本章採用的特殊編程風格。一般說到串列，大家會先想到資料結構課程中常提到的連結串列 (linked list)。介紹連結串列的範例程式大多用迴圈或遞迴追蹤著指標，一個一個地處理串列中的元素。在指令式語言中做關於陣列的操作時，也常用變數或指標指著「目前的」元素，並在一個迴圈中將該變數逐次遞增或減。總之，我們處理聚合型資料結構時，總是將其中元素一個個取出來處理。但本章的做法不同：我們將整個串列視為一個整體，對整個串列做 *map*, *filter*, *dropWhile* 等動作，或將它和另一個串列整個 *zip* 起來...

這種編程方式被稱作全麥編程 (*wholemeal programming*)，第 1.11 節中將解釋此詞的由來。全麥編程的提倡者們認為：一個個地處理元素太瑣碎，而鼓勵我們拉遠些，使用組件，以更抽象的方式組織程式的結構。

諸如 *map*, *filter*, *iterate*, *zipWith* 等等組件其實都是常見的編程模式。它們可被視為為了特定目的已先寫好的迴圈。拜高階函數與惰性求值之賜，這些組件能容易地被重用在許多不同脈絡中。這麼做的好處之一是：諸如 *map*, *filter*, *zip* 等組件的意義清楚，整個程式的意義也因此會比起自行在迴圈中一個個處理元素來得容易理解。事實上，這麼做可以養成我們思考演算法的新習慣。一些常見的編程模式現在是有名字的，我們把編程模式抽象出來了。而如同第 0 章所述，抽象化是我們理解、掌握、操作事物的重要方法。我們現在有了更多詞彙去理解、討論程式與演算法：「這個演算法其實就是先做個 *map*，把結果 *concat* 起來，然後做 *filter*...

在本書其他章節中我們也將看到：這些抽象化方便我們去操作、轉換程式。具體說來，如果程式用這些組件拼湊成，我們對這些組件知道的性質都可用在我們的程式上。例如，如果我們知道 $\text{map } f \cdot \text{map } g = \text{map } (f \cdot g)$ ，當我們看到程式中有兩個相鄰的 *map*，我們可用已知的性質把他們合併成一個——這相當於合併兩個迴圈。或著我們可以將一個 *map* 拆成兩個，以方便後續的其他處理。程式的建構方法使得程式含有更多資訊，使我們有更多可操作的空間。

全麥編程之所以成為可能，有賴程式語言的支援。例如，高階函數使得我們能將與特定問題相關的部分（如 *map f* 與 *filter p* 中的 *f* 與 *p*）抽象出來；惰性求值使我們勇於使用大串列或無限串列作為中間值，不用擔心它們被不必要地真正算出。

此外，全麥編程也需要豐富的組件函式庫。設計良好的組件捕捉了常見的編程模式，有了它們的幫忙，我們的程式可寫得簡潔明瞭——本章之中大部分的程式都是都是一行搞定的“one-liner”。但，這些組件不可能窮舉所有的編程模式。我們仍會需要自行從頭寫些函數。受到全麥編程影響，在自行寫函數

時，我們也常會希望將它們寫得更通用些，藉此發現常見的編程模式，設計出可重用的組件。

全麥編程能寫出多實用的程式？第 1.11 節中會提及其他學者嘗試過的，包含解密碼、解數獨在內的有趣例子。在本節，我們則想示範一個小練習：由下至上的合併排序 (merge sort)。

合併排序 假設我們已有一個函數 $merge' :: (List\ Int \times List\ Int) \rightarrow List\ Int$ ，如果 xs 與 ys 已經排序好， $merge' (xs, ys)$ 將它們合併為一個排序好的串列。¹⁴ 函數 $merge'$ 可用第 2.10 節的方式歸納寫成，也可使用將在第 [todo: where] 章提及的組件 $unfoldr$ 做出。我們如何用 $merge'$ 將整個串列排序好呢？

一般書中較常提及由上至下的合併排序：將輸入串列（或陣列）切成長度大致相等的兩半，分別排序，然後合併。本節則以由下至上的方式試試看。如果輸入串列為 $[4, 2, 3, 5, 8, 0, 1, 7]$ ，我們先把每個元素都單獨變成串列，也就是變成 $[[4], [2], [3], [5], [8], [0], [1], [7]]$ 。然後把相鄰的串列兩兩合併： $[[2, 4], [3, 5], [8, 0], [1, 7]]$ ，再兩兩合併成為 $[[2, 3, 4, 5], [0, 1, 8, 7]]$ ，直到只剩下一個大串列為止。

如果我們定義兩個輔助函數： $wrap$ 將一個元素包成一個串列， $isSingle$ 判斷一個串列是否只剩下一個元素，

```
wrap :: a → List a
wrap x = [x] ,
```

```
isSingle :: List a → Bool
isSingle [x] = True
isSingle xs  = False .
```

那麼上述的合併排序可以寫成：

```
msort = head · until isSingle mergeAdj · map wrap .
```

這幾乎只是把口語描述逐句翻譯：先把每個元素都包成串列，反覆做 $mergeAdj$ 直到只剩下一個大串列，然後將那個大串列取出來。

下一項工作是定義 $mergeAdj :: List (List\ Int) \rightarrow List (List\ Int)$ ，其功能是将相鄰的串列兩兩合併。如果我們能訂出一個函數 $adjs :: List\ a \rightarrow List (a \times a)$ ，將相鄰的元素放在序對中， $mergeAdj$ 就可以寫成：

```
mergeAdj = map merge' · adjs .
```

但 $adjs$ 該怎麼定義呢？對大部分讀者來說，最自然的方式也許是用第 2 章將討論的歸納法。但作為練習，我們姑且用現有的組件試試看。先弄清楚我們對 $adjs$ 的期待。當 $xs = [x_0, x_1, x_2, x_3]$ ，我們希望 $adjs\ xs = [(x_0, x_1), (x_2, x_3)]$ 。但當 xs 有奇數個元素時，例如 $xs = [x_0, x_1, x_2, x_3, x_4]$ ，最後一個元素 x_4 便落單了。如果是為了合併排序，我們也許可以把 x_4 和 $[]$ 放在一起， $adjs\ xs = [(x_0, x_1), (x_2, x_3), (x_4, [])]$ 。但為使 $adjs$ 適用於更多的情況，也許我們應該讓它多拿一個參數，當作落單的元素的配對。因此我們把 $adjs$ 的型別改為 $a \rightarrow List\ a \rightarrow List (a \times a)$ ，希望 $adjs\ z\ xs = [(x_0, x_1), (x_2, x_3), (x_4, z)]$ 。

我們試著看看這可如何辦到。

¹⁴之所以取名為 $merge'$ ，因為在第 2.9 節中我們將使用一個類似且相關的函數 $merge :: List\ Int \rightarrow List\ Int \rightarrow List\ Int$ 。

- 首先，`zip xs (tail xs)` 可把 `xs` 的每個元素和其下一個放在序對中。例：當 `xs = [x0, x1, x2, x3, x4]` 時，`zip xs (tail xs)` 的值是 `[(x0, x1), (x1, x2), (x2, x3), (x3, x4)]`。
- 如果我們為 `zip` 的第二個參數補上一個 `z`，成為 `zip xs (tail (xs ++ [z]))`，這可歸約為 `[(x0, x1), (x1, x2), (x2, x3), (x3, x4), (x4, z)]`。
- 再將位置（由 0 算起）為奇數的元素丟棄，我們便得到原先希望的 `[(x0, x1), (x2, x3), (x4, z)]` 了！

讀者可試試看當 `xs` 有偶數個元素時的情況。總之，`adjs` 可定義成：

```
adjs :: a → List a → List (a × a)
adjs z xs = everyother (zip xs (tail xs ++ [z])) ,
```

其中 `everyother ys` 把 `ys` 中位置為奇數的元素丟棄。

最後，考慮如何把串列中位置為奇數的元素丟棄。一種做法是：一直從串列中丟掉頭兩個元素，直到串列用完：

```
everyother :: List a → List a
everyother = map head · takeWhile (not · null) · iterate (drop 2) .
```

總而言之，由下至上的合併排序可寫成：

```
msort :: List Int → List Int
msort = head · until isSingle mergeAdj · map wrap ,
```

其中 `mergeAdj` 的定義是：

```
mergeAdj :: List (List Int) → List (List Int)
mergeAdj = map merge' · adjs [] .
```

如果我們想看到合併排序完成前的每一步驟，可將 `msort` 中（以 `iterate` 與 `dropWhile` 定義出）的 `until` 改為 `iterate` 與 `takeWhile`：

```
msortSteps :: List Int → List (List (List Int))
msortSteps = takeWhile (not · isSingle) · iterate mergeAdj · map wrap .
```

例如，`msortSteps [9, 2, 5, 3, 6, 4, 7, 0, 5, 1, 8, 2, 3, 1]` 可得到

```
[[[9], [2], [5], [3], [6], [4], [7], [0], [5], [1], [8], [2], [3], [1]],
 [2, 9], [3, 5], [4, 6], [0, 7], [1, 5], [2, 8], [1, 3]],
 [2, 3, 5, 9], [0, 4, 6, 7], [1, 2, 5, 8], [1, 3]],
 [0, 2, 3, 4, 5, 6, 7, 9], [1, 1, 2, 3, 5, 8]] .
```

最後兩個串列合併為 `[0, 1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 7, 8, 9]`，即為 `msort` 的結果。

1.10 自訂資料型別

本章目前為止給讀者看到的 `data` 定義其實都是 Haskell 已內建的型別。使用者也可自己定義新資料型別。例如，我們可能定義一個新型別表達四個方向：

```
data Direction = North | East | South | West ,
```

或著定義一個表示顏色的型別，用三個浮點數表達紅、綠、藍的比例：

```
data RGBColor = RGB Float Float Float ,
```

例：土耳其藍 (turquoise) 可寫成 `RGB 0.25 0.875 0.8125`。下列函數則降低一個顏色的彩度：¹⁵

```
desaturate :: Float → RGBColor → RGBColor
desaturate p (RGB r g b) =
  RGB (r + p × (gr - r)) (g + p × (gr - g)) (b + p × (gr - b)) ,
  where gr = r × 0.299 + g × 0.587 + b × 0.144 .
```

我們也可定義如 `List` 一樣的遞迴資料型別。例如，資料結構中可能談到兩種二元樹狀結構，一種僅在內部節點有標示（稱作 *internally labelled*），另一種僅在葉節點有表示（稱作 *externally labelled*）。這兩種二元樹可分別表示如下：

```
data ITree a = Null | Node a (ITree a) (ITree a) ,
data ETree a = Tip a | Bin (ETree a) (ETree a) .
```

怎麼編寫這種資料結構上的程式呢？我們將在下一章中說到。

1.11 參考資料

本章中的許多想法取自 Bird [1998]，該書是我相當推薦的 Haskell 教材。

Currying Moses Schönfinkel [1924] 提出多參數函數可用單參數函數表達。Haskell Curry 在許多著作中（例：Curry [1980]）使用 currying，但當時並沒有 currying 一詞。為何此概念最後會以 Curry 命名呢？David A. Turner（Haskell 語言的前身之一 Miranda 的設計人）在一次網路討論 [Sankar et al., 1997] 中表示 currying 一詞由 Christopher Strachey 取名，於 1967 年前後使用在其上課資料中。這種說法目前廣被大家接受，但我目前尚未找可佐證的上課資料。相反地，Strachey [1967] 之中明確表示他認為 currying 的概念是由 Schönfinkel 發明的，並稱之為「Schönfinkel 的裝置」。¹⁶ 但 currying 的想法可追溯得比 Schönfinkel 或 Curry 都早。F. L. Gottlob Frege 1891 年的 *Über Funktion und Begriff* (英譯 *Function and Concept*) [Frege, 1960] 結尾幾頁的概念即是 currying。

全麥編程 「全麥編程」一詞由牛津大學 Geraint Jones 取名，由來可能是模仿健康食物的說詞。如 Bird [2010, 第 19 章] 便寫道，「全麥編程好處多多，可預防『索引症』(indexitis)，鼓勵合法的程式建構。」在該章之中，Richard Bird 以大量使用串列組件函數的全麥編程為起點，推導出能相當迅速地解數獨的程

¹⁵這是一個簡便的做法：算出該顏色的灰度 *gr*，然後計算每個原色與該灰度的線性內插。更準確的作法應將 RGB 轉成 HSV，以後者調整飽和度。

¹⁶原文：“There is a device originated by Schönfinkel, for reducing operators with several operands to the successive application of single operand operators.”

Haskell 為何叫 Haskell?

1980 年代中期，程式語言學者們已各自開發出了許多個語法、語意類似但卻稍有不同、大都只在出生機構被使用的情性純函數語言。沒有一個語言取得壓倒性的優勢。為溝通方便、以及為了讓整個領域能走向下一步，大家有了該設計個統合、共通的情性純函數語言的共識。1988 年一月，新語言設計小組在耶魯大學開會，眾多討論項目中包括幫語言取個名字。以下軼事節錄自 Hudak et al. [2007].

當天被提出的選項包括 Haskell, Vivaldi, Mozart, CFL (Common Functional Language), Curry, Frege, Peano, Nice, Fun, Light... 等等。最後經程序選出的名字是 “Curry”，紀念邏輯學家 Haskell B. Curry — 他在組件邏輯 (combinatorial logic)、Curry-Howard 同構等領域的研究一直深遠影響函數語言學界。

但當天晚上就有人覺得這名字會招惹太多雙關語笑話。除了咖喱之外，小組成員覺得實在不行的是：TIM (three instruction machine) 是函數語言用的一種抽象機器，但 Tim Curry 則成了電影洛基恐怖秀 (Rocky Horror Picture Show, 1975) 的男主角。

於是新語言的名字就改成 Haskell 了。

小組成員 Paul Hudak 和 David Wise 寫信給 Curry 的遺孀 Virginia Curry，徵求她的同意。Hudak 後來親自登門拜訪，Virginia Curry 和他聊了之前的訪客（包含 Church 與 Kleene）的故事；後來她也去聽了 Hudak 關於 Haskell（語言）的演講，表現得十分友善。臨別前，她說：「其實呀，Haskell 一直都不喜歡他的名字。」

式。Hinze [2009] 以全麥編程為工具，示範了河內塔問題 (Tower of Hanoi) 的許多性質，以及其與謝爾賓斯基 (Sierpiński) 三角形的關係。其中寫道「函數語言擅長全麥編程。這個詞彙由 Geraint Jones 命名。全麥編程意謂由大處去想：處理整個串列，而不是一連串的元素；發展出整個解答的空間，而不是個別的解答；想像整個圖，而不是單一的路徑。對於已知的問題，全麥編程常給我們新洞察與新觀點。」Hutton [2016, 第五章] 則以編、解密碼為例。凱撒加密 (Caesar cipher) 為一種簡單的加密方式：將明文中的每個字母都往前或後偏移固定的量，例如當偏移量為 2 時，'a' 變成 'c'，'b' 變成 'd' ... 一種解凱撒密碼的有效方式是計算密文中每個字母的分佈，和一般英文文章中的平均字母分佈做比較，藉以猜出偏移量。Graham Hutton 在書中示範如何用組件函數、完全不用遞迴地寫出編碼與解碼程式。以上都是相當值得一看的例子。

LISP 的串列 誕生於 1958 年的 LISP 是目前仍被使用的高階程式語言中歷史第二悠久的 — 最早的是 FORTRAN. 但 LISP 與 FORTRAN 是風格截然不同的語言。雖然具有含副作用的指令，LISP 仍被認為是函數語言的先驅。

LISP 為「串列處理 (list processing)」的縮寫。但事實上，LISP 中的聚合資料結構「S 算式 (S-expression)」不只可用來表達串列。**CONS** 函數做出的是一個序對，其中第一個元素被稱作 **CAR** (contents of the address part of register)，第二個稱作 **CDR** (contents of the decrement part of register)。如果 **CDR** 的部分仍是一個 **CONS** 做出的序對，或是特殊值 **NIL**，整個結構表達的就是一個串列。**S** 算式也可用來做出二元樹、語法樹... 等等。Haskell 串列的建構元 `[]` 可唸成 “nil”，`(:)` 唸成 “cons”，這兩個詞彙都從 LISP 而來。

在前幾波人工智慧熱潮時，大家認為符號與邏輯的處理是人工智慧的基礎。但早期的程式語言大多針對數值運算而設計，會處理串列的 LISP 便被視

為最適合做符號處理的語言 — 「人工智慧用的語言」。另一個被視為「人工智慧專用語言」的是莫基於述語邏輯與歸結 (resolution) 的 PROLOG. 今日的人工智慧技術以神經網路為基礎，「人工智慧專用語言」的頭銜則給了 Python。

DRAFT

DRAFT

歸納定義與證明

「全麥編程」的觀念鼓勵我們以小組件組織出大程式。但這些個別組件該如何實作呢？或著，沒有合用的組件時該怎麼辦？我們可以回到更基礎的層次，用遞迴 (recursion) 定義它們。「遞迴」意指一個值的定義又用到它本身，是數學中常見的定義方式。在早期的編程教材中，遞迴常被視為艱澀、難懂、進階的主題。但在函數編程中，遞迴是使程式可不定次數地重複一項計算的唯一方法。一旦跨過了門檻，遞迴其實是個很清晰、簡潔地描述事情的方式。

對於遞迴，許多初學者一方面不習慣、覺得如此構思程式很違反「直覺」，另一方面也納悶：以自己定義自己，到底是什麼意思？這兩個難處其實都談到了好問題。對於前者，我們希望發展一些引導我們構思遞迴程式的思路，希望有了這些依據，能使寫遞迴程式變得直覺而自然。關於後者，其實並非所有遞迴定義都有「意思」——有些「不好」的遞迴並沒有定義出東西。我們討論遞迴的意義時必須限定在「好」的、有意義的程式上。最好有些方式確保我們寫出的遞迴定義是好的。

在本章我們將討論一種型式較單純的遞迴：歸納 (induction)。對上述兩個問題，本章的回應是：先有歸納定義出的資料結構，再依附著該資料結構撰寫歸納定義的程式，是一種思考、解決程式問題的好方法，也是一種理解遞迴程式的方式。此外，依循這種方法也能確保該定義是「好」的。我們將從數學歸納法出發，發現歸納程式與數學歸納法的相似性——寫程式和證明其實是很相似的活動。

本書以 Haskell 為學習工具，但在之後的幾章，我們僅使用 Haskell 的一小部分。Haskell 支援無限大的資料結構，允許我們寫出不會終止的程式。但我們將假設所有資料結構都是有限的（除非特別指明），所有函數皆會好好終止（也就是說函數都是「全函數 (total function)」——對每一個值，都會好好地算出一個結果，不會永遠算下去，也不會丟回一個錯誤）。這麼做的理由將在本章解釋。

2.1 數學歸納法

在討論怎麼寫程式之前，我們得先複習一下數學歸納法——晚點我們就會明白理由。回顧：自然數在此指的是 $0, 1, 2, \dots$ 等所有非負整數¹。自然數有無限多個，但每個自然數都是有限大的。自然數的型別記為 \mathbb{N} 。若 a 是一個型別， a 之上的述語 (predicate) 可想成型別為 $a \rightarrow \text{Bool}$ 的函數，常用來表示某性質對某特定的 a 是否成立。自然數上的述語便是 $\mathbb{N} \rightarrow \text{Bool}$ 。數學歸納法可用來證明某性質對所有自然數都成立：

給定述語 $P :: \mathbb{N} \rightarrow \text{Bool}$ 。若

1. P 對 0 成立，並且
2. 若 P 對 n 成立， P 對 $1+n$ 亦成立，

我們可得知 P 對所有自然數皆成立。

為何上述的論證是對的？我們在 2.6 節將提供一個解釋。但此處我們可以提供一個和編程較接近的理解方式。自然數可被想成如下的一個資料結構：

data $\mathbb{N} = 0 \mid 1_+ \mathbb{N}$.

這行定義有幾種讀解法，目前我們考慮其中一種——該定義告訴我們：

1. 0 的型別是 \mathbb{N} ;
2. 如果 n 的型別是 \mathbb{N} , $1_+ n$ 的型別也是 \mathbb{N} ;
3. 此外，沒有其他型別是 \mathbb{N} 的東西。

這種定義方式稱作歸納定義 (inductive definition)。其中「沒有其他型別是 \mathbb{N} 的東西」一句話很重要——這意味著任一個自然數只可能是 0 ，或是另一個自然數加一，沒有別的可能。任一個自然數都是這麼做出來的：由 0 開始，套上有限個 1_+ 。反過來說，給任意一個自然數，我們將包覆其外的 1_+ 一層層拆掉，在有限時間內一定會碰到 0 。有人主張將 inductive definition 翻譯為迭構定義，著重在從基底（此處為 0 ）開始，一層層堆積上去（此處為套上 1_+ ）的概念。

本書中，我們把自然數的 0 寫成粗體，表明它是資料建構元；把 1_+ 的加號寫得小些並和 1 放得很近，以強調「加一」是資料建構元、是一個不可分割的動作（和我們之後將介紹的一般自然數加法 $(+)$ 不同）。數字 2 其實是 $1_+(1_+ 0)$ 的簡寫， 3 其實是 $1_+(1_+(1_+ 0))$ 的簡寫。

歸納定義出的資料型別允許我們做歸納證明。由於 P 是 \mathbb{N} 到 Bool 的函數，「 P 對 0 成立」可記為 $P\ 0$ ，「若 P 對 n 成立， P 對 $1+n$ 亦成立」可記為 $P\ (1_+ n) \Leftarrow P\ n$ 。²我們假設兩者都已被證明。那麼，我們來證明 $P\ 3$ ：

$$\begin{aligned} & P\ (1_+(1_+(1_+ 0))) \\ \Leftarrow & \{ \text{因 } P\ (1_+ n) \Leftarrow P\ n \} \\ & P\ (1_+(1_+ 0)) \\ \Leftarrow & \{ \text{因 } P\ (1_+ n) \Leftarrow P\ n \} \\ & P\ (1_+ 0) \end{aligned}$$

¹有些數學派別的「自然數」是從 1 算起的。計算科學中則通常習慣以 0 起算。

² $P \Leftarrow Q$ 意思是「若 Q 則 P 」。依此順序寫，有「為證明 P ，我們想辦法讓 Q 成立」的感覺。許多人習慣由右到左的箭頭，但不論數學上或日常生活中，這都是常使用的論證思考方向。

$$\Leftarrow \{ \text{因 } P(1_+ n) \Leftarrow P n \} \\ P 0 .$$

第一步中，我們希望 $P(1_+(1_+(1_+ 0)))$ 成立，根據 $P(1_+ n) \Leftarrow P n$ ，只要 $P(1_+(1_+ 0))$ 即可。第二步中，我們希望 $P(1_+(1_+ 0))$ 成立，同樣根據 $P(1_+ n) \Leftarrow P n$ ，只要 $P(1_+ 0)$ 成立即可... 最後，只要 $P 0$ 成立， $P(1_+ 0)$ 即成立，但 $P 0$ 是已知的。因此我們已論證出 $P 3$ 成立！

由上述推演中，我們發現：數學歸納法的兩個前提 $P 0$ 與 $P(1_+ n) \Leftarrow P n$ 給了我們一個對任一個自然數 m ，生成一個 $P m$ 之證明的方法。這是由於自然數本就是一個歸納定義出的資料型別：任一個自然數 m 都是有限個 1_+ 套在 0 之上的結果，因此，只要反覆用 $P(1_+ n) \Leftarrow P n$ 拆，總有碰到 $P 0$ 的一天。既然對任何 m ，都做得出一個 $P m$ 的證明，我們就可安心相信 P 對任何自然數都成立了。

為了之後討論方便，我們將前述的數學歸納法寫得更形式化些：

$$\text{自然數上之歸納法：} (\forall n. P n) \Leftarrow P 0 \wedge (\forall n. P(1_+ n) \Leftarrow P n) .$$

這只是把之前的文字描述改寫成二階邏輯，但可清楚看出：給定 P ，我們希望證明它對所有自然數都成立，只需要提供 $P 0$ 和 $P(1_+ n) \Leftarrow P n$ 兩個證明。其中 $P 0$ 是確定 P 對 0 成立的基底 (base case)， $P(1_+ n) \Leftarrow P n$ 則被稱作歸納步驟 (inductive step)：在假設 $P n$ 成立的前提下，想辦法「多做一步」，論證 $P(1_+ n)$ 也成立。餘下的就可交給數學歸納法這個架構了。

2.2 自然數上之歸納定義

數學歸納法和編程有什麼關係呢？考慮一個例子：給定 $b, n :: \mathbb{N}$ ，我們希望寫個函數 exp 計算乘幂，使得 $\text{exp } b \ n = b^n$ 。我們先把型別寫下：

$$\text{exp} :: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ \text{exp } b \ n = ?$$

問號部分該怎麼寫？沒有其他線索很難進行，因此我們回想： n 是自然數，而任何自然數只可能是 0 或 1_+ 做出的。我們便分成這兩個狀況考慮吧：

$$\text{exp} :: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ \text{exp } b \ 0 = ? \\ \text{exp } b \ (1_+ n) = ?$$

其中， $\text{exp } b \ 0$ 較簡單：顯然應該是 $b^0 = 1$ 。至於 $\text{exp } b \ (1_+ n)$ 的右手邊該怎麼寫？似乎很難一步定出來。但假設 $\text{exp } b \ n$ 已經順利算出了 b^n ，由於 $b^{1_+ n} = b \times b^n$ ， $\text{exp } b \ (1_+ n)$ 與之的關係可寫成：

$$\text{exp } b \ (1_+ n) = b \times \text{exp } b \ n .$$

如此一來我們便完成了一個計算乘幂的程式：

Haskell v.s Math

很不幸地，Haskell 並不接受 2.2 節中 *exp* 的定義。首先，Haskell 並沒有獨立的自然數型別。我們可自己定（並將其宣告為 `Num` 類別的一員），或著直接使用 Haskell 內建的 `Int` 型別。其次，Haskell 原本允許我們在定義的左手邊寫 *exp b (n+1)*，但這套稱作 “*n+k* pattern” 的語法已在 Haskell 2010 中被移除。目前我們得將 *exp* 寫成：

```
exp :: Int → Int → Int
exp b 0 = 1
exp b n = b × exp b (n - 1) .
```

n+k pattern 曾引起激烈討論。支持者主要著眼於它在教學上的方便：這方便我們討論數學歸納法、做證明、並讓我們更明顯地看出自然數與串列的相似性。反對者則批評它與 `type class` 的衝突。後來由反方勝出。

有些 Haskell 教科書堅持書中出現的程式碼須是能在一個字一個字地鍵入電腦後即可執行的。本書的定位並非 Haskell 教材，而是函數編程概念的入門書。為此目的，我們希望選擇適合清晰表達概念、易於操作、演算、證明的符號。而一個實用目的的語言得在許多設計上妥協尋求平衡，基於種種考量，往往得犧牲符號的簡潔與便利性（這點我們完全能理解）。因此本書中的程式語法偶爾會和 Haskell 語法有所不同。我們會盡量指明這些不同處，使讀者知道如何將本書中的程式轉換成現下的 Haskell 語法。

```
exp :: ℕ → ℕ → ℕ
exp b 0 = 1
exp b (1+ n) = b × exp b n .
```

回顧一下剛剛的思路：我們難以一步登天地對任何 *n* 寫出 *exp b n*，但我們提供 *exp b 0* 該有的值，並在假設 *exp b n* 已算出該有的值的前提下，試著做一點加工、多算那一步，想法做出 *exp b (1+ n)* 該有的值。這和前述的數學歸納法是一樣的！寫歸納程式和做歸納證明是很類似的行為。使用數學歸納法證明 *P* 需要提供一個基底 *P 0* 和歸納步驟 *P (1+ n) ⇐ P n*。歸納定義程式也一樣。在 *exp b n* 的定義中，基底是 *exp b 0*，歸納步驟則是由 *exp b n* 想法變出 *exp b (1+ n)*。有這兩個元件，我們便有了一個對任何自然數 *n*，保證算出 *exp b n* 的方法。作為例子，我們看看 *exp 2 3* 是怎麼被算出來的：

```
exp 2 (1+ (1+ (1+ 0)))
= { exp 之歸納步驟 }
2 × exp (1+ (1+ 0))
= { exp 之歸納步驟 }
2 × 2 × exp (1+ 0)
= { exp 之歸納步驟 }
2 × 2 × 2 × exp 0
= { exp 之基底 }
2 × 2 × 2 × 1 .
```

第一步中，要算出 *exp 2 (1+ (1+ (1+ 0)))*，我們得先算出 *exp (1+ (1+ 0))*。要算出後者，在第二步中我們得先算出 *exp (1+ 0)*... 直到我們碰到 *exp b 0*。

自然數上的歸納定義 我們將 b 固定，稍微抽象一點地看 $\exp b :: \mathbb{N} \rightarrow \mathbb{N}$ 這個函數。該定義符合這樣的模式：

$$\begin{aligned} f &:: \mathbb{N} \rightarrow a \\ f\ 0 &= e \\ f\ (1_+ n) &= \dots f\ n \dots \end{aligned}$$

這類函數的輸入是 \mathbb{N} ，其定義中 $f\ (1_+ n)$ 的狀況以 $f\ n$ 定出，此外沒有其他對 f 的呼叫。若一個函數符合這樣的模式，我們說它是在自然數上歸納定義出的，其中 $f\ 0$ 那條稱作其基底， $f\ (1_+ n)$ 那條稱作其歸納步驟。我們日後將看到的許多程式都符合這個模式。

數學上，若一個函數能為其值域內的每個值都找到一個輸出，我們說它是個全函數 (total function)，否則是部分函數 (partial function)。計算上，當我們說 f 是一個全函數，意謂只要 x 型別正確並可算出值， $f\ x$ 便能終止並算出一個值，不會永久跑下去，也不會丟出錯誤。

如前所述的、在自然數上歸納定義的 f 會是全函數嗎？首先，任何自然數都可拆成 0 或是 $1_+ n$ ，而這兩個情況已被 f 的兩行定義涵括，不會出現漏失的錯誤。其次， f 每次呼叫自己，其參數都少了一層 1_+ 。長此以往，不論輸入多大，總有一天會遇到基底 $f\ 0$ — 因為任何自然數都是從 0 開始，套上有限個 1_+ 。只要基底狀況的 e 以及在歸納步驟中 $f\ n$ 前後的計算都正常終止，對任何輸入， f 都會正常終止。因此 f 是個全函數。

「程式會終止」是很重要的性質，我們之後會常談到。在本書目前為止示範的編程方法中，「一個函數若呼叫自己，只能給它更小的參數」是個單純但重要的規範（例如 $f\ (1_+ n)$ 的右手邊可以有 $f\ n$ ，不能有 $f\ (1_+ n)$ 或 $f\ (1_+ (1_+ n))$ ）。操作上這確保程式會終止，而在第 [todo: where?] 章之中，我們將提到這也確保該遞迴定義是「好」的、有意義的。

順便一提：在 $f\ (1_+ n)$ 的右手邊中， $f\ n$ 可以出現不只一次 — 因為 $\dots f\ n \dots f\ n \dots$ 可看成 $(\lambda x \rightarrow \dots x \dots x \dots) (f\ n)$ 。

乘法、加法 我們多看一些歸納定義的例子。在 \exp 中我們用到乘法，但假若我們的程式語言中只有加法、沒有乘法呢？我們可自己定定看：

$$\begin{aligned} (\times) &:: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ m \times n &= ? \end{aligned}$$

若不用組件，我們目前會的寫程式方法只有歸納法，也只有這招可試試看了。但， (\times) 有兩個參數，我們該把 $(m \times) :: \mathbb{N} \rightarrow \mathbb{N}$ 視為一個函數，分別考慮 n 是 0 或 $1_+ \dots$ 的情況，還是把 $(\times n) :: \mathbb{N} \rightarrow \mathbb{N}$ 視為一個函數，考慮 m 是 0 或 $1_+ \dots$ 的情況？答案是兩者皆可，並無根本性的差異。只是現在我們做的選擇會影響到之後與 (\times) 相關的證明怎麼寫（見第 2.3 節）。本書中的習慣是拆左手邊的參數，因此我們考慮以下兩種情況。

$$\begin{aligned} (\times) &:: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ 0 \times n &= ? \\ (1_+ m) \times n &= \dots m \times n \dots \end{aligned}$$

基底狀況中， $0 \times n$ 的合理結果應是 0 。歸納步驟中，我們得想法算出 $(1_+ m) \times n$ ，但我們可假設 $m \times n$ 已經算出了。稍作思考後，讀者應可同意以下的做法：

$$\begin{aligned} (\times) &:: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ 0 &\quad \times n = 0 \\ (1_+ m) \times n &= n + (m \times n) , \end{aligned}$$

如果已有 $m \times n$ ，多做一個 (n_+) ，就可得到 $(1_+ m) \times n$ 了。

如果我們的程式語言中連加法都沒有呢？加法可看成連續地做 1_+ ：

$$\begin{aligned} (+) &:: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ 0 &\quad + n = n \\ (1_+ m) + n &= 1_+ (m + n) . \end{aligned}$$

此處 $(+)$ 是我們定義的、可將任意兩個自然數相加的加法，而 1_+ 只做「加一」，是基本的資料建構元。為求一致，我們同樣在左邊的參數上做歸納。基底狀況中， $0 + n$ 只應是 n 。想計算 $(1_+ m) + n$ ，先假設 $m + n$ 已經算出，再多套一個 1_+ 。不難看出 $m + n$ 是把 n 當做基底，在外面套上 m 個 1_+ 的結果。³

2.3 自然數上之歸納證明

上一節中我們定出了函數 exp 。如果定義正確， $\text{exp } b \ n$ 算的應是 b^n 。例如，我們知道 $b^{m+n} = b^m \times b^n$ 。我們定出的函數 exp 是否真有此性質呢？

定理 2.1. 對任何 $b, m, n :: \mathbb{N}$, $\text{exp } b \ (m+n) = \text{exp } b \ m \times \text{exp } b \ n$.

我們試著證明定理 2.1。數學歸納法是我們目前唯一的工具，而要使用它，第一個問題是：該用 b , m , 或 n 的哪一個來做歸納呢（意即把哪一個拆解）？

觀察定理 2.1 中待證明式的等號兩邊，並參照 exp , $(+)$, 與 (\times) 的定義。等號左手邊的 $\text{exp } b \ (m+n)$ 之中，化簡 $\text{exp } b$ 前得知道 $m+n$ 究竟是 0 還是 $1_+ k$ 。而根據 $(+)$ 的定義，化簡 $m+n$ 前需知道 m 究竟是 0 還是 $1_+ k$ 。再看右手邊，根據 (\times) 的定義，要化簡 $\text{exp } b \ m \times \text{exp } b \ n$ 得先化簡 $\text{exp } b \ m$ ，而後者也得知道 m 是什麼。對兩邊的分析都指向：我們應針對 m 做歸納！

策略擬定後，我們便試試看吧！

證明 2.1. 欲證明 $\text{exp } b \ (m+n) = \text{exp } b \ m \times \text{exp } b \ n$ ，我們在 m 之上做歸納。 m 要不就是 0 ，要不就是 $1_+ k$ 。

情況 $m := 0$ 。此時需證明 $\text{exp } b \ (0+n) = \text{exp } b \ 0 \times \text{exp } b \ n$ 。推論如下：

$$\begin{aligned} &\text{exp } b \ (0+n) \\ &= \{ (+) \text{ 之定義 } \} \\ &\quad \text{exp } b \ n \\ &= \{ \text{因 } 1 \times k = k \} \\ &\quad 1 \times \text{exp } b \ n \end{aligned}$$

³ 「這樣做不是很慢嗎？」是的。本章的自然數表示法，以及其引申出的運算元都不應看作有效率的實作，而是理論工具。了解加法與乘法可這樣看待後，許多其相關性質都可依此推導出來。

$$= \{ \text{exp 之定義} \} \\ \text{exp } b \, 0 \times \text{exp } b \, n .$$

情況 $m := 1_+ m$. 此時需證明 $\text{exp } b \, ((1_+ m) + n) = \text{exp } b \, (1_+ m) \times \text{exp } b \, n$, 但可假設 $\text{exp } b \, (m + n) = \text{exp } b \, m \times \text{exp } b \, n$ 已成立。推論如下：

$$\begin{aligned} & \text{exp } b \, ((1_+ m) + n) \\ = & \{ (+) \text{ 之定義} \} \\ & \text{exp } b \, (1_+ (m + n)) \\ = & \{ \text{exp 之定義} \} \\ & b \times \text{exp } b \, (m + n) \\ = & \{ \text{歸納假設} \} \\ & b \times (\text{exp } b \, m \times \text{exp } b \, n) \\ = & \{ (\times) \text{ 之遞移律} \} \\ & (b \times \text{exp } b \, m) \times \text{exp } b \, n \\ = & \{ \text{exp 之定義} \} \\ & \text{exp } b \, (1_+ m) \times \text{exp } b \, n . \end{aligned}$$

□

對這個證明，讀者是否有所懷疑？最大的疑問可能在「假設 $\text{exp } b \, (m + n) = \text{exp } b \, m \times \text{exp } b \, n$ 成立」這句上。這不就是我們要證明的性質嗎？在證明中假設它成立，似乎是用該性質自己在證明自己。這是可以的嗎？

為清楚說明，我們回顧一下第 2.1 節中的數學歸納法（並把區域識別字改為 k 以避免混淆）：

$$\text{自然數上之歸納法：} (\forall k. P \, k) \Leftarrow P \, 0 \wedge (\forall k. P \, (1_+ k) \Leftarrow P \, k) .$$

證明 2.1 欲證的是 $\text{exp } b \, (m + n) = \text{exp } b \, m \times \text{exp } b \, n$ ，並在 m 上做歸納。更精確地說，就是選用了下述的 P ：⁴

$$P \, m \equiv (\text{exp } b \, (m + n) = \text{exp } b \, m \times \text{exp } b \, n) ,$$

在證明中改變的是 m ，而 b 與 n 是固定的。數學歸納法可證明 $(\forall m. P \, m)$ ，展開後正是 $(\forall m. \text{exp } b \, (m + n) = \text{exp } b \, m \times \text{exp } b \, n)$ 。而根據數學歸納法，我們需提供 $P \, 0$ 與 $(\forall m. P \, (1_+ m) \Leftarrow P \, m)$ 的證明。

證明 2.1 中「情況 $m := 0$ 」的部分，就是 $P \, 0$ 的證明。而「情況 $m := 1_+ m$ 」則是 $(\forall m. P \, (1_+ m) \Leftarrow P \, m)$ 的證明。「假設 $\text{exp } b \, (m + n) = \text{exp } b \, m \times \text{exp } b \, n$ 成立」指的是假設 $P \, m$ 成立，我們在此前提之下試圖證明 $P \, (1_+ m)$ 。因此，證明 2.1 並沒有「用該性質自己證明自己」。我們是以一個比較小的結果 ($P \, m$) 證明稍大一點的結果 ($P \, (1_+ m)$)。就如同我們寫歸納程式時，假設 $f \, n$ 已經算出，試著用它定出 $f \, (1_+ n)$ 。

在證明之中，如 $P \, m$ 這種在歸納步驟假設成立的性質被稱作歸納假設 (induction hypothesis)。

⁴在程式推導圈子中， (\equiv) 常用來代表「只用在真假值上、且滿足遞移律的等號」。本書中使用 (\equiv) 以和 $(=)$ 做區分。

程式與證明 證明 2.1 還有一些能啟發我們之處。方才，我們看到 $\text{exp } b(m+n) = \text{exp } b m \times \text{exp } b n$ ，決定以數學歸納法證明，但接下來怎麼著手？怎麼選定在哪個變數上做歸納？

答案是：分析該式中式式的行為。程式怎麼拆其參數，我們在證明中就怎麼拆。我們試圖證明某些程式的性質，但程式本身便提供了證明可怎麼進行的提示。「使證明的結構符合程式的結構」是許多證明的秘訣。並非所有證明都可以如此完成，但本原則在許多情況下適用。

再看看 exp , $(+)$, (\times) 等函數的定義。為何他們都分出了兩個情況： 0 與 $1+n$ ，並且 $1+n$ 的情況使用到該函數對於 n 的值？因為自然數的資料型別是這麼定的！自然數只可能是 0 或 $1+n$ ，而後者是由 n 做出來的。因此程式也如此寫。程式的結構依循與其處理的資料型別之結構。

資料、程式、與證明原來有著這樣的關係：證明的結構依循著程式的結構，而程式的結構又依循著資料型別的結構。歸納定義出了一個型別後，自然知道怎麼在上面寫歸納程式；歸納程式有了，自然知道如何做關於這些程式的歸納證明。一切由定義資料結構開始。掌握這個原則，大部分的證明就不是難事。

讓符號為你工作 再考慮證明 2.1 中的狀況 $m := 1+n$ 。假想由你做這個證明，由第一行 $\text{exp } b((1+n)+n)$ 開始。接下來該怎麼進行？

既然我們已經打定主意用數學歸納法，在證明的某處必定會使用 $\text{exp } b(m+n) = \text{exp } b m \times \text{exp } b n$ 這個歸納假設。因此，證明前幾行的目的便是想辦法將 $\text{exp } b((1+n)+n)$ 中的 $1+n$ 往外提，將 $\text{exp } b$ 往內側推，使得式子中出現 $\text{exp } b(m+n)$ 。一旦成功，就可運用歸納假設，將其改寫成 $\text{exp } b m \times \text{exp } b n$ ！接下的就是機械化地收尾、將式子整理成 $\text{exp } b(1+n) \times \text{exp } b n$ 了。

這呼應到第 0.2 節所說的「讓符號為你工作」。光從語意上想，我們不易理解為何 $\text{exp } b((1+n)+n)$ 能夠等於 $\text{exp } b(1+n) \times \text{exp } b n$ 。但符號給我們線索。我們可觀察式子的結構，暫時不去想語意；我們的目標是操作、移動這些符號，將它們轉換成可使用歸納假設的形式。因此，接下來的演算推導便有所依據而非盲目進行：已知目標是把某符號往外提或往內推，我們就可尋找、使用可達到此目的的規則。這些規則包括已知函數的定義、或諸如分配律、遞移律、結合律等等數學性質。若很明顯地缺了一個想要的性質，也許可以把它當作引理另外證證看。符號幫助我們，使我們的思考清晰而有方向。

習題 2.1 — 證明 $1 \times k = k$ 。這個證明並不需要歸納。

習題 2.2 — 證明 $(+)$ 之遞移律: $m + (n+k) = (m+n) + k$ 。此證明中你使用的述語是什麼？

習題 2.3 — 證明 $k+1 = k$ 。你需要使用歸納法嗎？用什麼述語？

最後，說到自然數上的歸納定義，似乎不得不提階層 (factorial)。用非正式的寫法， $\text{fact } n = n \times (n-1) \times (n-2) \times \dots \times 1$ 。形式化的定義如下：

$$\begin{aligned} \text{fact} &:: \mathbb{N} \rightarrow \mathbb{N} \\ \text{fact } 0 &= 1 \\ \text{fact } (1+n) &= (1+n) \times \text{fact } n . \end{aligned}$$

我們在定理 2.6 中將會談到階層與排列的關係。

2.4 串列與其歸納定義

如同第 1.8 所述，「元素型別為 a 的串列」可定義成如下的資料型別：

```
data List a = [] | a : List a .
```

這個定義可以理解為

- `[]` 是一個串列，
- 若 xs 是一個元素型別為 a 的串列， x 型別為 a ，則 $x:xs$ 也是一個元素型別為 a 的串列，
- 此外沒有其他元素型別為 a 的串列。

我們不難發現 `List a` 和 \mathbb{N} 是相當類似的資料結構：`[]` 相當於 0 ，`(:)` 則類似 1_+ ，只是此處我們不只「加一」，添加的那個東西多了一些資訊，是一個型別為 a 的元素。或著我們可反過來說，串列「只是」在每個 1_+ 上都添了一些資訊的自然數！既然自然數與串列有類似的結構，不難想像許多自然數上的函數、自然數的性質，都有串列上的類似版本，

串列上的歸納定義 和自然數類似，許多串列上的函數可歸納地定義出來。由於串列只可能由 `[]` 或 `(:)` 做出，定義串列上的函數時也分別處理這兩個情況。基底情況為 `[]`，而欲定義 $f(x:xs)$ 的值時，可假設 $f\ xs$ 已算出來了：

```
f :: List a → b
f []      = e
f (x:xs) = ...f xs ...
```

來看些例子吧！「算一個陣列的和」可能是許多人學到陣列後得寫的頭幾個練習程式。串列版的和可以這麼寫：

```
sum :: List Int → Int
sum []      = 0
sum (x:xs) = x + sum xs .
```

基底狀況中，空串列的和應是 0 。歸納步驟中，我們要算 $x:xs$ 的和，可假設我們已算出 xs 的和，再加上 x 即可。計算串列長度的 `length` 有很類似的定義：

```
length :: List a → ℕ
length []      = 0
length (x:xs) = 1+ (length xs) .
```

在歸納步驟中，我們想計算 $x:xs$ 的長度，只需假設我們已知 xs 的長度，然後加一。事實上，`length` 剛好體現了前述「`List a` 只是在每個 1_+ 上添了資訊的自然數」一事：`length` 把串列走過一遍，將 `[]` 代換成 0 ，並將每個 `(:)` 中附加的資訊拋棄，代換成 1_+ 。

函數 `map f :: List a → List b`，也就是 `map` 給定函數 f 的結果，也可在串列上歸納定義：

$$\begin{aligned} \text{map} &:: (a \rightarrow b) \rightarrow \text{List } a \rightarrow \text{List } b \\ \text{map } f [] &= [] \\ \text{map } f (x:xs) &= f x : \text{map } f xs \end{aligned}$$

基底狀況的合理結果是 $[]$ 。歸納步驟中，要對 $x:xs$ 中的每個元素都做 f ，我們可假設已經知道如何對 xs 中的每個元素都做 f ，把其結果接上 $f x$ 即可。

函數 $(++)$ 把兩個串列接起來。如果我們在其左邊的參數上做歸納定義，可得到：⁵

$$\begin{aligned} (++) &:: \text{List } a \rightarrow \text{List } a \rightarrow \text{List } a \\ [] ++ ys &= ys \\ (x:xs) ++ ys &= x : (xs ++ ys) \end{aligned}$$

空串列接上 ys 仍是 ys 。歸納步驟中，要把 $x:xs$ 接上 ys ，我們可假設已有辦法把 xs 接上 ys ，然後只需添上 x 即可。

請讀者比較一下 $(++)$ 與自然數加法 $(+)$ 的定義，會發現兩者的結構一模一樣！如果串列是在每個 1_+ 中加上資料的自然數， $(++)$ 就是串列上的加法了。在定理 2.2 中我們將更形式化地寫出兩者的關係。

串列上之歸納法 如果 $\text{List } a$ 是一個歸納定義出的資料結構，我們應可以在 $\text{List } a$ 之上做歸納證明。確實，串列上的歸納法可寫成：

$$\text{串列上之歸納法} : (\forall xs. P \text{ } xs) \Leftarrow P [] \wedge (\forall x \text{ } xs. P (x:xs) \Leftarrow P xs) \text{ .}$$

以文字敘述的話：給定一個述語 $P :: \text{List } a \rightarrow \text{Bool}$ ，若要證明 $P \text{ } xs$ 對所有 xs 都成立，只需證明 $P []$ 和「對所有 x 和 xs ，若 $P \text{ } xs$ 則 $P (x:xs)$ 」。

前面說到 $(++)$ 與 $(+)$ 的相似性。若要形式化地把 $\text{List } a$, \mathbb{N} , $(++)$, 與 $(+)$ 牽上關係，連接他們的橋樑就是 length — $xs ++ ys$ 的長度，應是 xs 與 ys 的長度之和！我們試著證明看看。

定理 2.2. 對所有 xs 與 ys , $\text{length } (xs ++ ys) = \text{length } xs + \text{length } ys$.

Proof. 檢視 length , $(++)$, 與 $(+)$ 的定義，會發現等號兩邊都須對 xs 做分析才能化簡。因此我們對 xs 做歸納。

狀況 $xs := []$.

$$\begin{aligned} &\text{length } ([] ++ ys) \\ &= \{ (++) \text{ 之定義 } \} \\ &\quad \text{length } ys \\ &= \{ (+) \text{ 之定義 } \} \\ &\quad 0 + \text{length } ys \\ &= \{ \text{length 之定義 } \} \\ &\quad \text{length } [] + \text{length } ys \end{aligned}$$

⁵依照 Haskell 的運算元優先順序， $x:(xs ++ ys)$ 其實可寫成 $x:xs ++ ys$ ，一般也常如此寫。此處為了清楚而加上括號。

等式證明的步驟該多詳細？

本書中目前為止的等式證明相當細：每一個定義展開都成為獨立的步驟。這是為了教學目的，實務上不一定得如此。以我而言，自己的研究手稿中可能會將步驟寫得極詳細，為確保每個細節正確，並讓他人（或幾年後已經忘記細節的自己）在不需知道上下文的情況下也能機械化地檢查每個步驟。但在論文中，因篇幅有限，及考量讀者一次能處理的資訊量有限，發表出的證明可能會省略許多步驟。

實務上，被認為簡單、不寫出也不妨礙理解的步驟或說明都可被省略。但何謂簡單則很依靠作者的判斷與習慣。一般說來，僅展開定義的步驟用電腦便可自動做到，通常是可精簡掉的。最好寫出的步驟則可能是決定整個證明之結構的、不易以電腦決定而得靠人類智慧與經驗的，等等。這可能包括使用歸納假設的那步，或使用較特別的引理時。例如，定理 2.2 的歸納步驟證明可能被精簡如下：

$$\begin{aligned}
 & \text{length } ((x:xs) ++ ys) \\
 &= \mathbf{1}_+ (\text{length } (xs ++ ys)) \\
 &= \{ \text{歸納假設} \} \\
 & \quad \mathbf{1}_+ (\text{length } xs + \text{length } ys) \\
 &= \text{length } (x:xs) + \text{length } ys .
 \end{aligned}$$

狀況 $xs := x:xs$.

$$\begin{aligned}
 & \text{length } ((x:xs) ++ ys) \\
 &= \{ (+) \text{ 之定義} \} \\
 & \quad \text{length } (x: (xs ++ ys)) \\
 &= \{ \text{length 之定義} \} \\
 & \quad \mathbf{1}_+ (\text{length } (xs ++ ys)) \\
 &= \{ \text{歸納假設} \} \\
 & \quad \mathbf{1}_+ (\text{length } xs + \text{length } ys) \\
 &= \{ (+) \text{ 之定義} \} \\
 & \quad (\mathbf{1}_+ (\text{length } xs)) + \text{length } ys \\
 &= \{ \text{length 之定義} \} \\
 & \quad \text{length } (x:xs) + \text{length } ys .
 \end{aligned}$$

□

讀者可觀察到類似的技巧：在 $xs := x:xs$ 的狀況中，頭幾步的目的是將 length 往裡推，製造出 $\text{length } (xs ++ ys)$ ，以便使用歸納假設。

討論自然數時，習題 2.2 曾請讀者證明加法都滿足遞移律。此處示範證明類似定理的串列版：

定理 2.3. $(++)$ 滿足遞移律。意即，對任何 xs , ys , 和 zs , $(xs ++ ys) ++ zs = xs ++ (ys ++ zs)$.

Proof. 在 xs 之上做歸納。

狀況 $xs := []$:

$$\begin{aligned}
& ([] ++ ys) ++ zs \\
&= \{ (++) \text{ 之定義} \} \\
& \quad ys ++ zs \\
&= \{ (++) \text{ 之定義} \} \\
& \quad [] ++ (ys ++ zs) .
\end{aligned}$$

狀況 $xs := x : xs$:

$$\begin{aligned}
& (x : xs) ++ ys ++ zs \\
&= \{ (++) \text{ 之定義} \} \\
& \quad (x : (xs ++ ys)) ++ zs \\
&= \{ (++) \text{ 之定義} \} \\
& \quad x : ((xs ++ ys) ++ zs) \\
&= \{ \text{歸納假設} \} \\
& \quad x : (xs ++ (ys ++ zs)) \\
&= \{ (++) \text{ 之定義} \} \\
& \quad (x : xs) ++ (ys ++ zs) .
\end{aligned}$$

□

基底狀況的證明很簡單。至於歸納步驟，同樣地，前兩步都是為了湊出 $(xs ++ ys) ++ zs$ ，以便使用歸納假設。既然 $(++)$ 滿足遞移律，日後我們寫 $xs ++ ys ++ zs$ 就可不加括號了。

習題 2.4 — 證明對所有 f, xs ，與 ys ， $\text{map } f (xs ++ ys) = \text{map } f xs ++ \text{map } f ys$ 。

習題 2.5 — 證明對所有 xs ， $xs ++ [] = xs$ 。比較本題與習題 2.3 的證明。

下述的 *map* 融合定理 (*map-fusion theorem*) 是關於 *map* 極常用的定理之一。所謂「融合」在此處是把兩個 *map* 融合為一。我們日後會見到更多的融合定理。

定理 2.4. 對任何 f 與 g ， $\text{map } f \cdot \text{map } g = \text{map } (f \cdot g)$ 。

Proof. 我們目前只會用歸納證明，但是 $\text{map } f \cdot \text{map } g = \text{map } (f \cdot g)$ 的左右邊都是函數，沒有出現串列也沒有出現自然數。該拿什麼東西來歸納呢？

回顧外延相等 (定義 1.8)：當 h, k 均是函數， $h = k$ 的意思是對任何參數 x ， $hx = kx$ 。因此，將待證式左右邊各補上參數，並將 (\cdot) 展開，可得知其意義為對任何 xs ，

$$\text{map } f (\text{map } g xs) = \text{map } (f \cdot g) xs .$$

我們便可以在 xs 上做歸納了！

□

習題 2.6 — 完成定理 2.4 的證明。

習題 2.7 — 證明對所有 f , $\text{length} \cdot \text{map } f = \text{length}$.

習題 2.8 — 證明 $\text{length} \cdot \text{concat} = \text{sum} \cdot \text{map length}$.

習題 2.9 — 證明 $\text{sum} \cdot \text{concat} = \text{sum} \cdot \text{map sum}$.

習題 2.10 — 證明對所有 f , $\text{map } f \cdot \text{concat} = \text{concat} \cdot \text{map} (\text{map } f)$.

習題 2.11 — 證明對所有 xs , $\text{sum} (\text{map} (\mathbf{1}_+) xs) = \text{length } xs + \text{sum } xs$.

習題 2.12 — 證明對所有 xs 與 y , $\text{sum} (\text{map} (\text{const } y) xs) = y \times \text{length } xs$.

2.5 更多歸納定義與證明

為讓讀者熟悉，本節中我們多看一些自然數或串列上的歸納定義例子。

filter 我們曾見過的函數 *filter* 可寫成如下的歸納定義：

```
filter :: (a → Bool) → List a → List a
filter p []      = []
filter p (x:xs) = if p x then x:filter p xs else filter p xs
```

在 *filter* 的許多性質中，我們證明這個例子：

定理 2.5. $\text{filter } p \cdot \text{map } f = \text{map } f \cdot \text{filter} (p \cdot f)$.

Proof. 和定理 2.4 一樣，待證式的左右邊都是函數。根據外延相等，我們將左右邊各補上參數 xs ，並在 xs 上做歸納：

$$\text{filter } p (\text{map } f xs) = \text{map } f (\text{filter} (p \cdot f) xs) \quad .$$

情況 $x := []$ 中左右邊都可化簡成 $[]$ 。我們看看 $xs := x:xs$ 的情況：

```
filter p (map f (x:xs))
= { map 之定義 }
  filter p (f x:map f xs)
= { filter 之定義 }
  if p (f x) then f x:filter p (map f xs) else filter p (map f xs)
= { 歸納假設 }
  if p (f x) then f x:map f (filter (p · f) xs) else map f (filter (p · f) xs)
= { map 之定義 }
  if p (f x) then map f (x:filter (p · f) xs) else map f (filter (p · f) xs)
= { 因 f (if p then e1 else e2) = if p then f e1 else f e2, 如後述 }
  map f (if p (f x) then x:filter (p · f) xs else filter (p · f) xs)
= { filter 之定義 }
  map f (filter (p · f) (x:xs)) .
```

□

終止與證明 上述證明的倒數第二步為將 $\text{map } f$ 提到外面，用了一個關於 **if** 的性質：

$$f(\text{if } p \text{ then } e_1 \text{ else } e_2) = \text{if } p \text{ then } f e_1 \text{ else } f e_2 . \quad (2.1)$$

這性質對嗎？若 p 成立，左右手邊都化簡為 $f e_1$ ，若 p 不成立，左右手邊都化簡為 $f e_2$ 。因此 (2.1) 應該成立，是嗎？

答案是：如果我們假設的世界中有不終止的程式，(2.1) 便不正確了。例如，當 f 是 $\text{three } x = 3$ ，而 p 是個永遠執行、不終止的算式（例： $\text{let } b = \text{not } b \text{ in } b$ ）：

$$\text{three}(\text{if } p \text{ then } e_1 \text{ else } e_2) \stackrel{?}{=} \text{if } p \text{ then } \text{three } e_1 \text{ else } \text{three } e_2$$

上述式子的左手邊直接化簡成 3，但右手邊卻不會終止，因為 **if** 得知道 p 的值。我們找到了 (2.1) 的反例！

在允許可能不終止的程式存在的世界中，(2.1) 得多些附加條件。通常的做法是限定 f 須是個嚴格函數，意即 f 的輸入若不終止， f 也不會終止。但 (2.1) 並不是唯一帶著附加條件的性質 — 許多常用性質都得加上類似的附加條件。所有狀況分析也都得將不終止考慮進去，例如，自然數除了 0 與 $1+n$ 之外，還多了第三種情況「不終止」。⁶ 推論與證明變得更複雜。有些人因此較喜歡另一條路：藉由種種方法確保我們只寫出會終止的程式，便可假設我們確實活在所有程式都正常終止的世界中。

保護式 v.s. 條件分支 有些人喜歡用保護式語法定義 **filter**：

$$\begin{aligned} \text{filter } p [] &= [] \\ \text{filter } p (x:xs) &= \begin{cases} p x & \text{if } p x \\ \text{otherwise} & \text{if } \text{not } p x \end{cases} \\ &= \begin{cases} x:\text{filter } p xs & \text{if } p x \\ \text{filter } p xs & \text{if } \text{not } p x \end{cases} \end{aligned}$$

若在此定義下證明定理 2.5，依「證明的結構與程式的結構相同」的原則，順理成章地，我們可在 $xs := x:xs$ 中再分出 $p(f x)$ 成立與不成立的兩個子狀況：

$$\begin{aligned} \text{狀況 } xs &= [] : \dots \\ \text{狀況 } xs &= x:xs : \dots \\ \text{狀況 } p(f x) : & \\ & \text{filter } p(\text{map } f(x:xs)) \\ &= \{ p(f x) \text{ 成立} \} \\ & \quad f x : \text{filter } p(\text{map } f xs) \\ &= \dots \\ \text{狀況 } \text{not}(p(f x)) : & \\ & \text{filter } p(\text{map } f(x:xs)) \\ &= \{ (\text{not } p(f x)) \} \\ & \quad \text{filter } p(\text{map } f xs) \\ &= \dots \end{aligned}$$

這個定義中不用 **if**，因此證明中也用不上 (2.1)，但該證明要成立仍須假設所有程式都正常終止 — 我們少證了一個「 $p(f x)$ 不終止」的情況（而確實，在此情況下 (2.1) 並不成立）。喜歡用哪個方式純屬個人偏好。

⁶Bird 就採用這種作法。

前幾章提過的 *takeWhile* 與 *dropWhile* 兩函數型別與 *filter* 相同。他們可寫成如下的歸納定義：

```
takeWhile :: (a → Bool) → List a → List a
takeWhile p [] = []
takeWhile p (x:xs) = if p x then x:takeWhile p xs else [] ,

dropWhile :: (a → Bool) → List a → List a
dropWhile p [] = []
dropWhile p (x:xs) = if p x then x:dropWhile p xs else [] .
```

兩者都是在輸入串列上做歸納。兩者也都可用保護式語法定義。

習題 2.13 — 證明 $\text{takeWhile } p \text{ } xs ++ \text{dropWhile } p \text{ } xs = xs$ 。

習題 2.14 — 以保護式語法定義 *takeWhile* 與 *dropWhile*，以此定義做做看習題 2.13。

不等式證明 給定如下的定義， $\text{elem } x \text{ } xs$ 判斷 x 是否出現在串列 xs 中：

```
elem x [] = False
elem x (y:xs) = (x == y) ∨ elem x xs .
```

目前為止，我們所練習的都是以 $(=)$ 將式子串起的等式證明。以下以 *elem* 為例，我們嘗試證明一個「不等式」：

$$\text{elem } z \text{ } xs \Rightarrow \text{elem } z \text{ } (xs ++ ys) .$$

以口語說出的話：「若 z 出現在 xs 中， z 也出現在 $xs ++ ys$ 中」。欲證明上式，該從哪一側推到哪一側呢？一般認為從式子較長、或結構較複雜的那側開始，化簡成較短、較簡單的那側，是較容易的。因此我們嘗試由右側推到左側：由 $\text{elem } z \text{ } (xs ++ ys)$ 開始，尋找使之成立的條件，並希望 $\text{elem } z \text{ } xs$ 是足夠的。

Proof. 在 xs 上做歸納。基底 $xs := []$ 的狀況在此省略，看 $xs := x:xs$ 的狀況：

```
elem z ((x:xs) ++ ys)
≡ { (++) 之定義 }
elem z (x:(xs ++ ys))
≡ { elem 之定義 }
(z == x) ∨ elem z (xs ++ ys)
⇐ { 歸納假設 }
(z == x) ∨ elem z xs
≡ { elem 之定義 }
elem z (x:xs) .
```

□

讀者可注意：第 1, 2, 4 步使用的邏輯關係都是 (\equiv) ，第 3 步卻是 (\Leftarrow) ，因此整個證明建立了「若 $\text{elem } z \text{ } (x:xs)$ ，則 $\text{elem } z \text{ } ((x:xs) ++ ys)$ 」。

習題 2.15 — 證明 $\text{not } (\text{elem } z \ (xs \ ++ \ ys)) \Rightarrow \text{not } (\text{elem } z \ xs)$.

習題 2.16 — 證明 $\text{elem } z \ xs \Rightarrow \text{elem } z \ (ys \ ++ \ xs)$.

習題 2.17 — 證明 $(\forall x. p \ x \Rightarrow q \ x) \Rightarrow \text{all } p \ xs \Rightarrow \text{all } q \ xs$. 其中 all 的定義為：

```
all :: (a → Bool) → List a → Bool
all p []      = True
all p (x:xs) = p x ∧ all p xs .
```

習題 2.18 — 證明 $\text{all } (\in xs) \ (\text{filter } p \ xs)$. 其中 $x \in xs$ 是 $\text{elem } x \ xs$ 的中序寫法。我們可能需要習題 2.16 和 2.17 的結果，以及下述性質：

$$\text{if } p \text{ then } x \text{ else } x = x . \quad (2.2)$$

前段與後段 本章目前為止討論的歸納定義都依循著這樣的模式：欲定義 $f :: \text{List } a \rightarrow b$, 只要為 $f []$ 與 $f (x:xs)$ 找到定義。在定義後者時，只需定義出由 $f \ xs$ 做出 $f (x:xs)$ 的關鍵一步。目前為止，這關鍵一步都是加一、加上一個元素等簡單的動作。現在我們來看些更複雜的例子。

例 1.19 中曾提及：如果一個串列 xs 可分解為 $ys \ ++ \ zs$, 我們說 ys 是 xs 的一個前段 (*prefix*), zs 是 xs 的一個後段 (*suffix*). 例如，串列 $[1,2,3]$ 的前段包括 $[], [1], [1,2]$, 與 $[1,2,3]$ (注意： $[]$ 是一個前段，串列 $[1,2,3]$ 本身也是)，後段則包括 $[1,2,3], [2,3], [3]$, 與 $[]$ 。我們是否能定義一個函數 $\text{inits} :: \text{List } a \rightarrow \text{List } (\text{List } a)$, 計算給定串列的所有前段呢？例 1.19 給的答案是：

```
inits xs = map (\n → take n xs) [0..length xs] .
```

如果不用組件，改用歸納定義呢？我們試試看：

```
inits :: List a → List (List a)
inits []      = ?
inits (x:xs) = ?
```

基底狀況 $\text{inits } []$ 的可能選擇是 $[[]]$ (見後述)。至於歸納步驟該怎麼寫？我們用例子來思考。比較 $\text{inits } [2,3]$ 與 $\text{inits } [1,2,3]$:

```
inits [2,3] = [[],[2],[2,3]] ,
inits [1,2,3] = [[],[1],[1,2],[1,2,3]] .
```

假設我們已算出 $\text{inits } [2,3]$, 如何把它加工變成 $\text{inits } [1,2,3]$? 請讀者暫停一下，思考看看！

一個思路是：如果在 $[[],[2],[2,3]]$ 中的每個串列前面都補一個 1, 我們就有了 $[[1],[1,2],[1,2,3]]$. 再和 $\text{inits } [1,2,3]$ 比較，就只差一個空串列了！因此 inits 的一種定義方式是：


```

inits :: List a → List (List a)
inits []      = [[]]
inits (x:xs) = [] : map (x:) (inits xs) .

```

在此得提醒：有些讀者認為基底狀況 $\text{inits } []$ 的值選為 $[[]]$ ，是因為結果的型別是 $\text{List (List } a)$ （直覺地把每個 List 都對應到一組中括號，或認為 $[[]]$ 是型別為 $\text{List (List } a)$ 的最簡單的值）。但事實並非如此：畢竟， $[]$ 的型別也可以是 $\text{List (List } a)$ ！我們讓 $\text{inits } [] = [[]]$ 的原因是空串列 $[]$ 的「所有前段」只有一個，恰巧也是 $[]$ 。就如同在自然數上的歸納函數定義中，有些基底狀況是 0，有些是 1，有些是別的值，此處我們也依我們的意圖，選定最合適的基底值。

習題 2.19 — 試定義 $\text{inits}^+ :: \text{List } a \rightarrow \text{List (List } a)$ ，計算一個串列的所有非空前段。例如 $\text{inits}^+ [1,2,3]$ 是 $[[]], [1], [1,2], [1,2,3]$ 。當然，其中一個定義方式是 $\text{inits}^+ = \text{tail} \cdot \text{inits}$ 。你能以歸納方式定義出 inits^+ 嗎？

```

inits+ []      = ?
inits+ (x:xs) = ?

```

習題 2.20 — 我們驗證一下 inits 在例 1.19 中的組件定義與本章的歸納定義是相等的。定義 $\text{upto} :: \mathbb{N} \rightarrow \text{List } \mathbb{N}$:

```

upto 0      = [0]
upto (1+ n) = 0 : map (1+) (upto n) .

```

使得 $\text{upto } n = [0..n]$ 。假設 inits 已如本節一般地歸納定義，證明對所有 xs ， $\text{inits } xs = \text{map } (\lambda n \rightarrow \text{take } n \text{ } xs) (\text{upto } (\text{length } xs))$ 。您可能會需要 map 融合定理（定理 2.4），

定義傳回後段的函數 tails 時可依循類似的想法：如何把 $\text{tails } [2,3] = [[2,3], [3], []]$ 加工，得到 $\text{tails } [1,2,3] = [[1,2,3], [2,3], [3], []]$ ？這次較簡單：加上 $[1,2,3]$ 即可。的確，串列 $x:xs$ 的後段包括 $x:xs$ 自己，以及 xs 的後段：

```

tails :: List a → List (List a)
tails []      = [[]]
tails (x:xs) = (x:xs) : tails xs .

```

在習題 2.34 中我們將證明一個將 inits 與 tails 牽上關係的定理：將 inits 傳回的前段與 tails 傳回的後段照其順序對應，每對接起來都是原來的串列。

連續區段 給定一個串列，許多傳統最佳化問題的目標是計算符合某條件的連續區段（簡稱「區段」）。例如， $[1,2,3]$ 的區段包括 $[], [1], [2], [3], [1,2], [2,3]$ ，以及 $[1,2,3]$ 本身。我們可用 inits 與 tails 得到一個串列的所有區段。

```

segments :: List a → List (List a)
segments = concat · map inits · tails .

```

但 segments 無法寫成本章目前這種形式的歸納定義。我們將在以後的章節再討論到 segments 。

習題 2.21 — 試著把 *segments* 寫成如下的歸納定義：

```
segments :: List a → List (List a)
segments [] = ?
segments (x:xs) = ...segments xs...
```

在歸納步驟中希望由 *segments xs* 湊出 *segments (x:xs)*。這是能在不對輸入串列（型別為 *List a*）做任何限制之下做得到的嗎？如果做不到，為什麼？

排列 函數 *fan x xs* 把 *x* 插入 *xs* 的每一個可能空隙。例如，*fan 1 [2,3,4]* 可得到 *[[1,2,3,4],[2,1,3,4],[2,3,1,4],[2,3,4,1]]*。讀者不妨想想它該怎麼定義？一種可能方式如下：

```
fan :: a → List a → List (List a)
fan x [] = [[x]]
fan x (y:xs) = (x:y:xs) : map (y:) (fan x xs)
```

有了 *fan*，我們不難定義 *perms :: List a → List (List a)*，計算一個串列所有可能的排列。例如，*perms [1,2,3] = [[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]*：

```
perms :: List a → List (List a)
perms [] = [[]]
perms (x:xs) = concat (map (fan x) (perms xs))
```

讀者可思考為何我們需要 *concat*？如果沒有，會出現什麼錯誤？

基於 *perm* 的這個定義，我們證明一個定理：長度為 *n* 的串列有 *fact n* 種排列。這個證明將使用到許多輔助性質與引理，有些已經是我們之前證明過的習題，有些則可作為接下來的習題。在本證明之中我們也練習將連續的函數應用 *f (g (h x))* 寫成函數合成 *f · g · h \$ x* 以方便計算。

定理 2.6. 對任何 *xs*, *length (perms xs) = fact (length xs)*.

Proof. 在 *xs* 上做歸納。

基底狀況 *xs := []*:

```
length (perms [])
= length [[]]
= 1
= fact (length [])
```

歸納步驟 *xs := x:xs*:

```
length (perms (x:xs))
= { perms, (·), 與 ($) 之定義 }
length · concat · map (fan x) · perms $ xs
= { 因 length · concat = sum · map length (習題 2.8), map-融合 }
```

$$\begin{aligned}
& \text{sum} \cdot \text{map} (\text{length} \cdot \text{fan } x) \cdot \text{perms } \$ xs \\
= & \{ \text{因 } \text{length} \cdot \text{fan } x = (\mathbf{1}_+) \cdot \text{length} \text{ (習題 2.24)} \} \\
& \text{sum} \cdot \text{map} ((\mathbf{1}_+) \cdot \text{length}) \cdot \text{perms } \$ xs \\
= & \{ \text{因 } \text{map length (perms } xs) = \text{map (const (length } xs)) \text{ (perms } xs) \text{ (習題 2.25)} \} \\
& \text{sum} \cdot \text{map} ((\mathbf{1}_+) \cdot \text{const (length } xs)) \cdot \text{perms } \$ xs \\
= & \{ \text{因 } \text{sum (map } (\mathbf{1}_+) xs) = \text{length } xs + \text{sum } xs \text{ (習題 2.11)} \} \\
& \text{length (perms } xs) + \text{sum (map (const (length } xs)) \text{ (perms } xs)) \\
= & \{ \text{因 } \text{sum (map (const } y) xs) = y \times \text{length } xs \text{ (習題 2.12)} \} \\
& \text{length (perms } xs) + \text{length } xs \times \text{length (perms } xs) \\
= & \{ \text{四則運算: } x + y \times x = (1 + y) \times x \} \\
& (\mathbf{1}_+ (\text{length } xs)) \times \text{length (perms } xs) \\
= & \{ \text{歸納假設} \} \\
& (\mathbf{1}_+ (\text{length } xs)) \times \text{fact (length } xs) \\
= & \{ \text{fact 之定義} \} \\
& \text{fact } (\mathbf{1}_+ (\text{length } xs)) \\
= & \{ \text{length 之定義} \} \\
& \text{fact (length (x:xs))} .
\end{aligned}$$

□

習題 2.22 — 證明 $\text{map } f \cdot \text{fan } x = \text{fan } (f \ x) \cdot \text{map } f$.

習題 2.23 — 證明 $\text{perm} \cdot \text{map } f = \text{map } (\text{map } f) \cdot \text{perm}$.

習題 2.24 — 證明 $\text{length (fan } x \ xs) = \mathbf{1}_+ (\text{length } xs)$.

習題 2.25 — 證明 $\text{perms } xs$ 傳回的每個串列都和 xs 一樣長，也就是 $\text{map length (perms } xs) = \text{map (const (length } xs)) \text{ (perms } xs)$. 其中 const 定義於第 28 頁 — $\text{const } y$ 是一個永遠傳回 y 的函數。

子串列 本節最後一個例子：函數 $\text{sublists} :: \text{List } a \rightarrow \text{List (List } a)$ 計算一個串列的所有子串列。後者是類似子集合的概念，只是把順序也考慮進去： ys 是 xs 的子串列，如果將 xs 中的零個或數個元素移除後可得到 ys ：例如 $\text{sublists } [1, 2, 3]$ 的結果可能是： $[[], [3], [2], [2, 3], [1], [1, 3], [1, 2], [1, 2, 3]]$ 。怎麼定義 sublists 呢？在基底狀況中，空串列仍有一個子串列 $[]$ 。歸納步驟中， $x:xs$ 的子串列可分為兩種：不含 x 的，以及含 x 的。不含 x 的子串列就是 xs 的所有子串列（以下稱作 yss ），而含 x 的子串列就是 yss 中的每個串列接上 x 。因此我們可定義：

$$\begin{aligned}
& \text{sublists} :: \text{List } a \rightarrow \text{List (List } a) \\
& \text{sublists } [] = [[]] \\
& \text{sublists } (x:xs) = yss \mathbin{++} \text{map } (x:) yss \ , \\
& \text{where } yss = \text{sublists } xs \ .
\end{aligned}$$

習題 2.26 — 定義 $splits :: List\ a \rightarrow List\ (List\ a \times List\ a)$ ，使 $splits\ xs$ 傳回所有滿足 $ys ++ zs$ 的 (ys, zs) 。例：

$$splits\ [1, 2, 3] = [([], [1, 2, 3]), ([1], [2, 3]), ([1, 2], [3]), ([1, 2, 3], [])]$$

另一種說法是 $splits\ xs = zip\ (inits\ xs)\ (tails\ xs)$ 。

習題 2.27 — 證明 $length \cdot sublists = exp\ 2 \cdot length$ 。也就是說，長度為 n 的串列的子串列數目為 2^n 。你會需要的性質可能包括定理 2.2 ($length\ (xs ++ ys) = length\ xs + length\ ys$)，以及 $length \cdot map\ f = length$ 。

2.6 由集合論看歸納法

序理論回顧 為了本節的完整性，我們在這兒回顧一些重要定義。對學過序理論或抽象代數的讀者來說，以下概念應不陌生。如果您還不熟悉這些定義，由於它們在程式語言語意中常常使用，值得花些時間弄懂。

定義 2.7 (前序、偏序)。給定集合 S ，令 (\leq) 為 S 上的一個二元關係。如果 (\leq) 滿足：

1. 自反律：對所有 $x \in S$ ， $x \leq x$ 。
2. 遞移律：對所有 $x, y, z \in S$ ， $x \leq y \wedge y \leq z \Rightarrow x \leq z$ 。

則 (\leq) 被稱為 S 上的一個前序 (preorder)。如果 (\leq) 除上述兩性質外還滿足：

3. 反對稱律：對所有 $x, y \in S$ ， $x \leq y \wedge y \leq x \Rightarrow x = y$ 。

則 (\leq) 被稱為 S 上的一個偏序 (partial order)。如果 S 上有偏序 (\leq) ，我們常把它們放在一起，稱 $(S, (\leq))$ 為一個偏序集合 (partially ordered set, 或 poset)。

定義 2.8 (最小上界、最大下界)。給定偏序集合 $(S, (\leq))$ ，考慮其子集 $T \subseteq S$ ：

- 如果 $x \in S$ 滿足 $(\forall y \in T : y \leq x)$ ，則 x 是 T 的一個上界 (upper bound)。
- 如果 T 的所有上界中存在「最小」的，該值稱作 T 的最小上界 (supremum, 或 least upper bound)，記為 $\sup T$ 。依據定義， $\sup T$ 滿足 $(\forall y \in T : y \leq \sup T) \Rightarrow \sup T \leq x$ 。
- 如果 $x \in S$ 滿足 $(\forall y \in T : y \geq x)$ ，則 x 是 T 的一個下界 (lower bound)。
- 如果 T 的所有下界存在「最大」的，該值稱作 T 的最大下界 (infimum, 或 greatest lower bound)，記為 $\inf T$ 。依據定義， $\inf T$ 滿足 $(\forall y \in T : y \geq \inf T) \Rightarrow \inf T \geq x$ 。

定義 2.9 (格、完全格)。考慮偏序集合 $(S, (\leq))$ ：

- 如果對任何 $x, y \in S$ ， $\sup \{x, y\}$ 和 $\inf \{x, y\}$ 均存在且都在 S 之中，則 $(S, (\leq))$ 是一個格 (lattice)。
- 如果對任何 $T \subseteq S$ ， $\sup T$ 和 $\inf T$ 均存在且都在 S 之中，則 $(S, (\leq))$ 是一個完全格 (complete lattice)。

在本節之中我們只會考慮一種格。令 D 代表所有範式（如 0 ， $1 + 0$ ， True ， $1 : 2 : []$ ， $(\lambda x \rightarrow x) \dots$ ）的集合，我們考慮的格是 $\mathcal{P}D$ ，即 D 的所有子集形成的集合。其上的偏序就是子集關係 (\subseteq) 。

定點 再回顧一些與定點相關的理論。

定義 2.10 (定點). 給定完全格 $(A, (\leq))$ 和函數 $f :: A \rightarrow A$,

1. 如果 $f x \leq x$, 我們說 x 是 f 的一個前定點 (prefixed point).
2. 如果 $f x = x$, 我們說 x 是 f 的一個定點 (fixed point).
3. 如果 $f x \geq x$, 我們說 x 是 f 的一個後定點 (postfixed point).

定義 2.11 (最小前定點、最大後定點). 給定完全格 $(A, (\leq))$ 和函數 $f :: A \rightarrow A$,

- 如果 f 的前定點之中存在最小值, 我們將它記為 μf . 根據此定義, μf 滿足 $f x \leq x \Rightarrow \mu f \leq x$.
- 如果 f 的後定點之中存在最大值, 我們將它記為 νf . 根據此定義, νf 滿足 $x \leq f x \Rightarrow x \leq \nu f$.

定理 2.12. 給定完全格 $(A, (\leq))$ 和函數 $f :: A \rightarrow A$,

- f 的最小前定點 μf 也是最小的定點,
- f 的最小前定點 νf 也是最大的定點。

歸納定義 回顧自然數的定義：

data $\mathbb{N} = 0 \mid 1_+ \mathbb{N}$.

第 2.1 節對這行定義的解釋是：

1. 0 的型別是 \mathbb{N} ;
2. 如果 n 的型別是 \mathbb{N} , $1_+ n$ 的型別也是 \mathbb{N} ;
3. 此外, 沒有其他型別是 \mathbb{N} 的東西。

如果我們把一個型別視作一個集合, 上述條件定出了怎麼樣的集合呢?⁷ 用 \mathbb{N} 表示我們定出的這個新型別。上述第 1. 點告訴我們 0 是 \mathbb{N} 的成員, 也就是 $\{0\} \subseteq \mathbb{N}$. 第 2. 點則表示, 從 \mathbb{N} 這個集合中取出任一個元素 n , 加上 1_+ , 得到的結果仍會在 \mathbb{N} 之中。也就是說 $\{1_+ n \mid n \in \mathbb{N}\} \subseteq \mathbb{N}$. 集合基本定理告訴我們 $X \subseteq Z \wedge Y \subseteq Z$ 和 $X \cup Y \subseteq Z$ 是等價的, 所以:⁸

$$\{0\} \cup \{1_+ n \mid n \in \mathbb{N}\} \subseteq \mathbb{N} . \quad (2.3)$$

意思是說, 如果我們定義一個集合到集合的函數 NatF :

$$\text{NatF } X = \{0\} \cup \{1_+ n \mid n \in X\} ,$$

那麼 (2.3) 可以改寫為

$$\text{NatF } \mathbb{N} \subseteq \mathbb{N} ,$$

也就是說, \mathbb{N} 是 NatF 的一個前定點!

至於 3. 呢? 它告訴我們 \mathbb{N} 僅含恰巧能滿足 1. 和 2. 的元素, 沒有多餘。意即, \mathbb{N} 是滿足 1. 和 2. 的集合之中最小的。若有另一個集合 Z 也滿足 1. 和 2., 我們必定有 $\mathbb{N} \subseteq Z$. 也就是說 \mathbb{N} 是 NatF 的最小前定點: $\mathbb{N} = \mu \text{NatF}$!

⁷請注意:「把型別視為集合」僅在簡單的語意之中成立。本書後來將會討論更複雜的語意, 屆時型別並不只是集合。

⁸ $X \subseteq Z \wedge Y \subseteq Z \equiv X \cup Y \subseteq Z$ 被稱為「 (\cup) 的泛性質」。

給定述語 P ，我們把所有「滿足 P 的值形成的集合」也記為 P 。⁹ 數學歸納法的目的是證明所有自然數都滿足 P 。但，「所有自然數都滿足 P 」其實就是 $\mathbb{N} \subseteq P$ 。

如何證明 $\mathbb{N} \subseteq P$ ？如前所述， \mathbb{N} 是 NatF 的最小前定點。如果 P 恰巧也是 NatF 的一個前定點， $\mathbb{N} \subseteq P$ 一定得成立。寫成推論如下：

$$\begin{aligned}
 & \mathbb{N} \subseteq P \\
 \Leftarrow & \{ \mathbb{N} \text{ 是 } \text{NatF} \text{ 的最小前定點, 定義 2.11} \} \\
 & \text{NatF } P \subseteq P \\
 \equiv & \{ \text{NatF 的定義} \} \\
 & \{0\} \cup \{1+n \mid n \in P\} \subseteq P \\
 \equiv & \{ (\cup) \text{ 的泛性質: } X \cup Y \subseteq Z \equiv X \subseteq Z \wedge Y \subseteq Z \} \\
 & \{0\} \subseteq P \wedge \{1+n \mid n \in P\} \subseteq P .
 \end{aligned}$$

也就是說，如果證出 $\{0\} \subseteq P$ 和 $\{1+n \mid n \in P\} \subseteq P$ ，我們就有 $\mathbb{N} \subseteq P$ 。其中，

1. $\{0\} \subseteq P$ 翻成口語便是「 P 對 0 成立」，
2. $\{1+n \mid n \in P\} \subseteq P$ 則是「若 P 對 n 成立， P 對 $1+n$ 亦成立」。

正是數學歸納法的兩個前提！

原來，數學歸納法之所以成立，是因為自然數被定義為某函數的最小前定點。事實上，當我們說某型別是「歸納定義」的，意思便是它是某個函數的最小前定點。

以串列為例。為單純起見，先考慮元素皆為 \mathbb{N} 的串列。如下的定義

data ListNat = [] | N : ListNat ,

可理解為 $\text{ListNat} = \mu \text{ListNatF}$ ，而 ListNatF 定義為：

$\text{ListNatF } X = \{\{\}\} \cup \{n:xs \mid xs \in X, n \in \mathbb{N}\}$.

至於如下定義的、有型別參數的串列，

data List a = [] | a : List a ,

則可理解為 $\text{List } a = \mu (\text{ListF } a)$ — $\text{List } a$ 是 $\text{ListF } a$ 的最小前定點，其中 ListF 定義如下：

$\text{ListF } A X = \{\{\}\} \cup \{x:xs \mid xs \in X, x \in A\}$.

給定某型別 A ，當我們要證明某性質 P 對所有 $\text{List } A$ 都成立，實質上是想要證明 $\text{List } A \subseteq P$ （同樣地，此處 P 代表所有使述語 P 成立的值之集合）。我們推論如下：

$$\begin{aligned}
 & \text{List } A \subseteq P \\
 \Leftarrow & \{ \text{List } a = \mu (\text{ListF } a) \} \\
 & \text{ListF } A P \subseteq P
 \end{aligned}$$

⁹對任何 A ，函數 $P :: A \rightarrow \text{Bool}$ 和「滿足 P 的 A 形成的集合」是同構的。我們可把它們等同視之。

$$\begin{aligned}
&\equiv \{ \text{ListF 之定義} \} \\
&\quad \{ [] \} \cup \{ x:xs \mid xs \in P, x \in A \} \subseteq P \\
&\equiv \{ (\cup) \text{ 的泛性質} \} \\
&\quad \{ [] \} \subseteq P \wedge \{ x:xs \mid xs \in P, x \in A \} \subseteq P .
\end{aligned}$$

其中 $\{ [] \} \subseteq P$ 翻成口語即是「 $P []$ 成立」； $\{ x:xs \mid xs \in P, x \in A \} \subseteq P$ 則是「若 $P xs$ 成立，對任何 $x::A$, $P (x:xs)$ 成立」。

我們之所以能做自然數與串列的歸納證明，因為它們都是歸納定義出的型別 — 它們都被定義成某函數的最小前定點。若非如此，歸納證明就不適用了。那麼，有不是歸納定義出的型別嗎？

讀者可能注意到，本節起初同時談到最小前定點與最大後定點，但後來只討論了前者。事實上，我們也可以把一個資料型別定義為某函數的最大後定點。這時我們說該資料型別是個餘歸納 (coinductive) 定義，如此訂出的型別稱為餘資料 (codata)。歸納定義出的資料結構是有限的，而餘歸納定義的型別可能包括無限長的資料結構。寫餘資料相關的證明，另有一套稱作餘歸納 (coinduction) 的方法，而餘歸納也影響到我們如何寫與餘資料相關的程式。餘歸納和歸納的相關理論剛好是漂亮的對偶。我們將在 [todo: where] 介紹餘歸納。

習題 2.28 — 回顧第 1.10 節中介紹的兩種樹狀結構：

```

data ITree a = Null | Node a (ITree a) (ITree a) ,
data ETree a = Tip a | Bin (ETree a) (ETree a) .

```

說說看它們分別是什麼函數的最小前定點，並找出它們的歸納原則。

2.7 歸納定義的簡單變化

目前為止，我們所認定「良好」的函數定義是這種形式：

$$\begin{aligned}
f \mathbf{0} &= \dots \\
f (\mathbf{1}_+ n) &= \dots f n \dots
\end{aligned}$$

我們知道這樣定義出的函數是個全函數、對所有輸入都會終止、和歸納法有密切關係...。以後的幾個章節中，我們將逐步放鬆限制，允許更有彈性的函數定義模式。我們先從歸納法的一些較簡單的變化開始。

基底狀況的變化 有些函數的值域是「正整數」或「大於 b 的整數」。定義這些函數時我們可用這樣的模式：

$$\begin{aligned}
f_1 \mathbf{1} &= e & f_b b &= e \\
f_1 (\mathbf{1}_+ n) &= \dots f_1 n \dots , & f_b (\mathbf{1}_+ n) &= \dots f_b n \dots .
\end{aligned}$$

我們可把 f_1 理解為：另外訂了一個資料型別 $\text{data } \mathbb{N}^+ = \mathbf{1} \mid \mathbf{1}_+ \mathbb{N}^+$ ，以 $\mathbf{1}$ 為基底狀況，而 f_1 是 \mathbb{N}^+ 之上的全函數。 f_b 的情況也類似。與使用 \mathbb{N} 的函數混用時，

我們就得在這兩個型別之間作轉換。這相當於檢查給 f_1 的輸入都是大於 1 的整數。實務上為了方便，我們仍用同一個型別實作 \mathbb{N} 與 \mathbb{N}^+ ，就如同實務上用 `Int` 實作 \mathbb{N} 一樣。

串列的情形也類似。有些函數若能只定義在「不是空的串列」上，其定義會比較合理。對於這些函數，我們可想像有這麼一個「非空串列」資料型別 `data List+ a = [a] | a : List+ a`，只是為了方便，我們用普通串列實作它。¹⁰

例 2.13. 假設 $x \uparrow y$ 傳回 x 與 y 中較大的值。函數 `maximum+` 傳回一個非空串列中的最大元素：

$$\begin{aligned} \text{maximum}^+ &:: \text{List}^+ \text{ Int} \rightarrow \text{Int} \\ \text{maximum}^+ [x] &= x \\ \text{maximum}^+ (x : xs) &= x \uparrow \text{maximum}^+ xs \end{aligned}$$

`Haskell` 標準函式庫中另有一個函數 `maximum :: List Int → Int`，但該函數需假設 `Int` 中有一個相當於 $-\infty$ 的值存在，以便當作 `maximum []` 的結果。

多個參數的歸納定義 函數 `take/drop` 的功能與 `takeWhile/dropWhile` 很類似，但其定義方式卻有些不同：

$$\begin{aligned} \text{take} &:: \mathbb{N} \rightarrow \text{List } a \rightarrow \text{List } a \\ \text{take } 0 \quad xs &= [] \\ \text{take } (1_+ n) [] &= [] \\ \text{take } (1_+ n) (x : xs) &= x : \text{take } n \, xs, \\ \text{drop} &:: \mathbb{N} \rightarrow \text{List } a \rightarrow \text{List } a \\ \text{drop } 0 \quad xs &= xs \\ \text{drop } (1_+ n) [] &= [] \\ \text{drop } (1_+ n) (x : xs) &= \text{drop } n \, xs. \end{aligned}$$

我們可把 `take/drop` 想成在自然數上歸納定義成的高階函數：`take (1+ n)` 的值是一個 `List a → List a` 的函數。定義 `take (1+ n)` 時，我們假設 `take n` 已有定義。唯一的特殊處是我們另分出兩個子情況：串列為 `[]`，或串列為 `x : xs`。

根據「使證明的結構符合程式的結構」的原則，如果要做關於 `take/drop` 的證明，一種可能做法是也依循著它們的結構：先拆解自然數，在 $n := 1_+ n$ 的情況中再分析串列的值。作為例子，我們來驗證第 40 頁提到的這個性質：

定理 2.14. 對所有 n 與 xs ， $\text{take } n \, xs \mathbin{++} \text{drop } n \, xs = xs$ 。

Proof. 在 n 上做歸納。在 $n := []$ 的情況下，等號兩邊都化簡為 `[]`。在 $n := 1_+ n$ 的情況中，我們再細分出兩種情形：

狀況 $n := 1_+ n$ ， $xs := []$ 。顯然等號兩邊都化簡為 `[]`。

狀況 $n := 1_+ n$ ， $xs := x : xs$ ：

$$\begin{aligned} &\text{take } (1_+ n) (x : xs) \mathbin{++} \text{drop } (1_+ n) (x : xs) \\ &= \{ \text{take 與 drop 之定義} \} \end{aligned}$$

¹⁰在一些型別系統更強大的語言中、進行更注重證明的應用時，我確實有將 \mathbb{N}^+ 與 `List+` 做成與 \mathbb{N} 和 `List` 不同的型別的經驗與需求。

$$\begin{aligned}
 & x : take\ n\ xs ++ drop\ n\ xs \\
 = & \quad \{ \text{歸納假設} \} \\
 & x : xs \quad .
 \end{aligned}$$

□

然而，本章討論的是所有資料結構都是歸納定義、所有函數也都是全函數的世界。如果我們的世界中有不終止程式存在，上式便不見得成立了。

習題 2.29 — 請舉一個在允許不終止程式的 Haskell 中， $take\ n\ xs ++ drop\ n\ xs = xs$ 不成立的例子。

習題 2.30 — 對任何串列 xs ， $head\ xs : tail\ xs = xs$ 都成立嗎？請舉一個反例。

函數 zip 是另一個例子。我們可把 $zip\ xs\ ys$ 視為 xs 之上的歸納定義：

$$\begin{aligned}
 zip & :: List\ a \rightarrow List\ b \rightarrow List\ (a \times b) \\
 zip\ []\ ys & = [] \\
 zip\ (x:xs)\ [] & = [] \\
 zip\ (x:xs)\ (y:yz) & = (x,y) : zip\ xs\ yz \quad .
 \end{aligned}$$

習題 2.31 — 試定義 $zipWith :: (a \rightarrow b \rightarrow c) \rightarrow List\ a \rightarrow List\ b \rightarrow List\ c$ ，並證明 $zipWith\ f\ xs\ ys = map\ (uncurry\ f)\ (zip\ xs\ ys)$ 。

2.8 完全歸納

說到遞迴定義，費氏數 (Fibonacci number) 是最常見的教科書例子之一。簡而言之，第零個費氏數是 0，第一個費氏數是 1，之後的每個費氏數是之前兩個的和。寫成遞迴定義如下：

$$\begin{aligned}
 fib & :: \mathbb{N} \rightarrow \mathbb{N} \\
 fib\ 0 & = 0 \\
 fib\ 1 & = 1 \\
 fib\ (2+n) & = fib\ (1+n) + fib\ n \quad .
 \end{aligned}$$

但這和我們之前談到的歸納定義稍有不同。我們已知定義 $f\ (1+n)$ 時可假設 f 在 n 之上已有定義。但在 fib 的定義中， $fib\ (2+n)$ 用到了 fib 的前兩個值。這樣的定義是可以的嗎？

函數 fib 的定義可以視為完全歸納（又稱作強歸納）的例子。回顧：先前介紹的數學歸納法中，使 $P\ n$ 成立的前提之一是「對所有 n ，若 $P\ n$ 成立， $P\ (1+n)$ 亦成立」。完全歸納則把這個前提增強如下：

給定述語 $P :: \mathbb{N} \rightarrow \text{Bool}$ 。若

- 對所有小於 n 的值 i ， $P\ i$ 皆成立，則 $P\ n$ 亦成立，

則我們可得知 P 對所有自然數皆成立。

以更形式化的方式可寫成：

完全歸納： $(\forall n. P n) \Leftarrow (\forall n. P n \Leftarrow (\forall i < n. P i))$.

請注意：前提 $P n \Leftarrow (\forall i < n. P i)$ 隱含 $P 0$ 成立，因為當 $n := 0$ ，由於沒有自然數 i 滿足 $i < n$ ，算式 $(\forall i < n. P i)$ 可化簡為 **True**。

在完全歸納法之中，證明 $P n$ 時，我們可假設 P 對所有小於 n 的值都已成立了。對寫程式的人來說，有了完全歸納法，表示我們日後定義自然數上的函數 $f :: \mathbb{N} \rightarrow a$ 時，每個 $f n$ 都可以自由使用 f 在所有小於 n 的輸入之上的值。因此 $\text{fib}(2+n)$ 可以用到 $\text{fib}(1+n)$ 與 $\text{fib } n$ ，因為 $n < 1+n < 2+n$ 。

例 2.15. 關於完全歸納，離散數學教科書中的一個常見例子是「試證明所有自然數都可寫成不相同的二的乘冪的和」，例如 $50 = 2^5 + 2^4 + 2$ 。這可用完全歸納證明。我們以半形式的方式論述如下：令 $P n$ 為「 n 可寫成一串不相同的二的乘冪的和」。對所有 n ，我們想要證明 $P n \Leftarrow (\forall i < n. P i)$ 。當 n 為 **0**，這一串數字即是空串列。當 n 大於零，

- 我們可找到最接近 n 但不超過 n 的二之乘冪，稱之為 m （也就是 $m = 2^k$ 而且 $m \leq n < 2 \times m$ ）。
- 由於 n 不是 **0**， m 也不是 **0**，也因此 $n - m < n$ 。
- 依據歸納假設 $(\forall i < n. P i)$ ，以及 $n - m < n$ ， $P(n - m)$ 成立 — $n - m$ 可以寫成不相同的二的乘冪的和。
- 也因此 n 可寫成不相同的二的乘冪的和 — 把 $n - m$ 加上 m 即可。

上述證明僅用口語描述，因為如果形式化地寫下這個證明，就等同於寫個將自然數轉成一串二的乘冪的程式，並證明其正確性！下述函數 *binary* 做這樣的轉換，例如， $\text{binary } 50 = [32, 16, 2]$ ：

```
binary :: N -> List N
binary 0 = []
binary n = m : binary (n - m) ,
  where m = last (takeWhile (<=) n) twos
        twos = iterate (2*) 1 .
```

函數 *binary* 是一個完全歸納定義，和上述的證明對應得相當密切：串列 *twos* 是 $[1, 2, 4, 8, \dots]$ 等等所有二的乘冪， m 是其中最接近而不超過 n 的。遞迴呼叫 $\text{binary}(n - m)$ 是許可的，因為 $n - m < n$ ，而根據完全歸納，我們已假設對所有 $i < n$ ， $\text{binary } i$ 皆有定義。

有了完全歸納法，我們在定義自然數上的函數時可允許更靈活的函數定義：

```
f :: N -> a
f b = .... { 一些基底情況 }
f n = ..f m...f k... { 如果 m < n 且 k < n }
```

$f n$ 的右手邊可以出現不只一個遞迴呼叫，只要參數都小於 n 。但我們必須確定上述定義中的幾個子句足以包含所有狀況，沒有狀況被遺漏。例如，我們若把 *fib* 定義中的 $\text{fib } 1 = \dots$ 基底狀況去掉，計算 $\text{fib } 2 = \text{fib } 1 + \text{fib } 0$ 時便會出錯。

習題 2.32 — 證明 $\text{sum}(\text{binary } n) = n$.

習題 2.33 — 證明當 $n \geq 1$, $\text{fib}(2+n) > \alpha^n$, 其中 $\alpha = (1 + \sqrt{5})/2$. 這個證明可用 $n:=1$ 和 $n:=2$ 當基底狀況。

完全歸納 vs 簡單歸納 完全歸納有個較早的稱呼：強歸納 (strong induction)。原本的歸納法則相對被稱呼為簡單歸納或弱歸納。強/弱歸納的稱呼可能使人以為完全歸納比簡單歸納更強 — 意謂前者能證明出一些後者無法證明的定理。事實上，完全歸納與簡單歸納是等價的：能用一個方法證出的定理，用另一個方法也能證出。因此，使用哪一個純粹只是方便性的考量。

反應在編程上，給任一個完全歸納定義函數 $f :: \mathbb{N} \rightarrow A$, 我們總能做出一個以簡單歸納定義的函數 $fs :: \mathbb{N} \rightarrow \text{List } A$, 滿足 $fs\ n = \text{map } f\ [n, n-1, \dots, 0]$. 例如，函數 $\text{fibs } n$ 傳回 $[\text{fib } n, \text{fib } (n-1) \dots, \text{fib } 0]$.

```
fibs :: N → List N
fibs 0      = [0]
fibs 1      = [1, 0]
fibs (1+ n) = (x1 + x0) : x1 : x0 : xs ,
  where (x1 : x0 : xs) = fibs n .
```

由 fib 到 fibs 的轉換可能令讀者想起演算法中的動態規劃 (dynamic programming)。我們將在日後談到這個話題。

串列上的完全歸納 串列與自然數是類似的資料結構。串列上的完全歸納原則便是將 0 代換為 $[]$, 將 $1+$ 代換為 $(x:)$. 至於「小於」的關係，可定義為：

$$ys < xs \equiv ys \in \text{tails } xs \wedge ys \neq xs .$$

也就是說 ys 是 xs 的一個後段，但不是 xs 自己。有了如上定義，串列上的完全歸納法是：

$$\text{串列的完全歸納: } (\forall xs \cdot P\ xs) \Leftarrow (\forall xs \cdot P\ xs \Leftarrow (\forall ys < xs \cdot P\ ys)) .$$

應用在編程上，當定義 $f\ (xs:)$ 時，遞迴呼叫可作用在 xs 的任何後段上。

但對許多串列上的函數而言，這樣的模式還不夠靈活。我們得用下一節說到的良基歸納。

2.9 良基歸納

良基歸納 (well-founded induction) 可視為完全歸納的再推廣。如果說完全歸納的主角是自然數，使用的是自然數上的「小於 ($<$)」關係，良基歸納則將其推廣到任何型別，使用任一個良基序。

定義 2.16. 給定某型別 A 之上的二元關係 (\triangleleft)。如果從任意一個 $a_0 :: A$ 開始，均不存在無限多個滿足如下關係的 $a_1, a_2 \dots$:

$$\dots \triangleleft a_2 \triangleleft a_1 \triangleleft a_0 ,$$

則 (\triangleleft) 可稱為一個良基序 (well-founded ordering)。

把 $b \triangleleft a$ 簡稱為「 b 小於 a 」。上述定義可以這麼地直覺理解：給定任一個 $a_0 :: A$ ，我們找一個滿足 $a_1 \triangleleft a_0$ 的值 a_1 。可能已經不存在這種 a_1 ，但如果存在，我們再找一個滿足 $a_2 \triangleleft a_1$ 的 a_2 。說 (\triangleleft) 是個良基序的意思便是前述過程不可能永遠做下去：總有一天我們得停在一個「最小」的某基底 $a_n :: A$ 。舉例說明：自然數上的「小於 $(<)$ 」關係是個良基序，但整數上的 $(<)$ 關係則不是——由於負數的存在。實數上的 $(<)$ 關係也不是。良基序並非得是個全序 (total order)。例如，我們可定義序對上的比較關係如下：

$$(x_1, y_1) \triangleleft (x_2, y_2) \equiv x_1 < x_2 \wedge y_1 < y_2 ,$$

其中 x_1, y_1, x_2, y_2 都是自然數。這麼一來，不論 $(1, 4) \triangleleft (2, 3)$ 或 $(2, 3) \triangleleft (1, 4)$ 都不成立，但 (\triangleleft) 仍是個良基序——任何兩個自然數形成的序對不論以什麼方式遞減，最晚也得停在 $(0, 0)$ 。

如果 (\triangleleft) 是個良基序，我們便可在其上做歸納。以直覺來理解的話，如果某函數定義成如此的形式（假設這幾個子句已經包括參數的所有可能情況）：

$$\begin{aligned} f &:: A \rightarrow B \\ f b &= \dots & \{ \text{一些基底情況} \} \\ f x &= \dots f y \dots f z \dots & \{ \text{如果 } y \triangleleft x \text{ 且 } z \triangleleft x \} \end{aligned}$$

由任何 $f x$ 開始，若 x 不是基底情況之一，我們需遞迴呼叫 $f y$ 和 $f z$ 。但 y 和 z 在 (\triangleleft) 這個序上比 x 「小」了一點。此後即使再做遞迴呼叫，每次使用的參數又更小了一點。而由於 (\triangleleft) 是良基序， f 的參數不可能永遠「小」下去—— f 非得停在某個基底情況不可。因此 f 必須正常終止。同樣的原則也用在證明上：

給定述語 $P :: A \rightarrow \text{Bool}$ 以及 A 之上的良基序 (\triangleleft) 。若

- 對所有滿足 $y \triangleleft x$ 的值 y ， $P y$ 皆成立，則 $P x$ 亦成立，

則我們可得知 P 對所有 A 皆成立。

或著可寫成如下形式：

$$\begin{aligned} \text{良基歸納: } (\forall x. P x) &\Leftarrow (\forall x. P x \Leftarrow (\forall y \triangleleft x. P y)) , \\ &\text{其中 } (\triangleleft) \text{ 為一個良基序。} \end{aligned}$$

終止證明與良基歸納 我們已在許多地方強調：確定程式正常終止是很重要的。我們也知道以簡單歸納與完全歸納定義出的程式均是會正常終止的。但這兩種歸納定義的限制很多。雖然我們已舉了許多例子，仍有些程式難以套入它們所要求的模板中。相較之下，良基歸納寬鬆許多。大部分我們已知、會終止的程式都可視為良基歸納定義。

或著，上述段落應該反過來說。在函數編程中，欲證明某個遞迴定義的函數會終止，最常見的方式是證明該函數每次遞迴呼叫時使用的參數都在某個度量上「變小」了，而這個度量又不可能一直變小下去。因此該函數遲早得碰到

基底狀況。換句話說，該函數每次遞迴呼叫的參數符合某個良基序；當我們如此證明一個函數會終止，其實就相當於在論證該函數是一個良基歸納定義。在指令式編程中證明某迴圈會終止的做法也類似。最常見的方式是證明該迴圈每多執行一次，某個量值就會變小，而該量值是不可能一直變小的。也就是說這些量值在每趟迴圈執行時的值符合某個良基序。

以下我們將看幾個遞迴定義的例子。請讀者們想想：這些函數總會正常終止嗎？為何？如果它們是良基歸納，使用的良基序是什麼？

例 2.17. 以下是大家熟悉的快速排序 (*quicksort*) [Hoare, 1962]:

```
qsrt :: List Int → List Int
qsrt [] = []
qsrt (x:xs) = qsrt ys ++ [x] ++ qsrt zs ,
  where (ys,zs) = (filter (≤ x) xs, filter (< x) xs) .
```

空串列是已經排序好的。當輸入為非空串列 $x:xs$ ，我們將 xs 分為小於等於 x 的，以及大於 x 的，分別遞迴排序，再將結果接在一起。

函數 *qsrt* 會正常終止，因為每次遞迴呼叫時，作為參數的串列都會減少至少一個元素（因為 x 被取出了），而串列的長度又不可能小於 0。若要稍微形式地談這件事，可從良基歸納的觀點來看。如果定義：

$$ys \triangleleft xs \equiv \text{length } ys < \text{length } xs ,$$

在 *qsrt* ($x:xs$) 子句中， $ys \triangleleft xs$ 和 $zs \triangleleft xs$ 均被滿足，而 (\triangleleft) 是一個良基序。因此 *qsrt* 是一個奠立在 (\triangleleft) 之上的良基歸納定義。

例 2.18. 在串列上，合併排序也是很常使用的排序方式。我們在第 1.9 節中示範過以全麥編程方式寫成、由下往上的合併排序。此處的寫法則更接近大家一般的認知：拿到一個長度為 n 的串列，將之分割為長度大致為 $n/2$ 的兩段，分別排序之後合併。同樣地，假設我們已有一個函數 $\text{merge} :: \text{List Int} \rightarrow \text{List Int} \rightarrow \text{List Int}$ ，如果 xs 與 ys 已經排序好， $\text{merge } xs \ ys$ 將它們合併為一個排序好的串列。合併排序可寫成：

```
msort :: List Int → List Int
msort [] = []
msort [x] = [x]
msort xs = merge (msort ys) (msort zs) ,
  where (ys,zs) = (take (n `div` 2) xs, drop (n `div` 2) xs) .
```

要論證 *msort* 會正常終止，或著說，要將 *msort* 視為一個良基歸納定義，我們可用和例 2.17 中一樣的良基序 (\triangleleft) 。

但此處請讀者小心檢查：在 *msort* xs 子句中， $ys \triangleleft xs$ 和 $zs \triangleleft xs$ 有被滿足嗎？當 $\text{length } xs = n$ ，串列 ys 與 zs 的長度分別是 $n \text{ `div` } 2$ 和 $n - n \text{ `div` } 2$ 。當 $\text{length } xs = 1$ 時， ys 與 zs 的長度分別是... 0 和 1 — zs 並沒有變短！

這是為何我們需要 *msort* $[x]$ 這個子句把 $\text{length } xs = 1$ 的情況分開處理。如果沒有這個子句，*msort* 將有可能不終止 — 讀者不妨試試看。

例 2.19. 歐幾里得 (*Euclid*) 的《幾何原本》成書於西元前三百年，其中描述「計算最大公因數」的做法可能是世界上最古老的演算法。以下函數計算兩個自然數 (m, n) 的最大公因數。如果兩數相等，它們的最大公因數也是自身。若兩數不相等，其最大公因數會是「大數減小數」與「小數」的最大公因數：

$$\begin{aligned} \text{gcd} &:: (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N} \\ \text{gcd } (m, n) &| m == n = n \\ &| \text{otherwise} = \text{gcd } ((m \uparrow n) - (m \downarrow n), m \downarrow n) . \end{aligned}$$

這個程式總會正常終止嗎？為什麼？

事實上，若 m 或 n 其中之一為 0， $\text{gcd } (m, n)$ 是不會終止的——讀者不妨也試試看。若 m, n 均為正整數呢？首先我們先確立：如果初始的 m, n 均為正整數， gcd 每次遞迴呼叫拿到的參數也都是正整數——確實如此，因為如果 $m > 0$ ， $n > 0$ ，且 $m \neq n$ ，那麼 $(m \uparrow n) - (m \downarrow n)$ 與 $m \downarrow n$ 都不會是零或負數。接下來我們可論證：如果 m, n 均為正整數，每次遞迴呼叫中，兩參數的和都變小了一些。確實：

$$\begin{aligned} &(m \uparrow n) - (m \downarrow n) + (m \downarrow n) \\ &= m \uparrow n \\ &< \{ m, n \text{ 均為正整數} \} \\ & \quad m + n . \end{aligned}$$

因此，我們可得知 gcd 在 m, n 均為正整數時會正常終止。如果把 gcd 當作一個良基歸納，我們用了如下的良基序：

$$(m_1, n_1) \triangleleft (m_2, n_2) \equiv m_1 + n_1 < m_2 + n_2 ,$$

其中 m_1, n_1, m_2, n_2 均為正整數。

例 2.20 (Curried 函數). 下述函數 *interleave* 將兩個參數中的元素交錯放置。例如 *interleave* $[1, 2, 3] [4, 5] = [1, 4, 2, 5, 3]$ 。

$$\begin{aligned} \text{interleave} &:: \text{List } a \rightarrow \text{List } a \rightarrow \text{List } a \\ \text{interleave } [] & \quad \text{ys} = \text{ys} \\ \text{interleave } xs & \quad [] = xs \\ \text{interleave } (x:xs) & \text{ys} = x:\text{interleave } \text{ys } xs . \end{aligned}$$

這可視為一個良基歸納定義嗎？若將 *interleave* 做為傳回函數的高階函數看待，我們比較難看出它是定義在什麼良基序上的。但若把 *interleave* 的兩個參數一起考慮，我們不難看出什麼度量在遞迴呼叫後「變小」了：兩個參數長度的和！

凡是遇到像 *interleave* 的 *curried* 函數，我們也可考慮它的 *uncurried* 版本：

$$\begin{aligned} \text{interleave}' &:: (\text{List } a \times \text{List } a) \rightarrow \text{List } a \\ \text{interleave}' ((), \text{ys}) &= \text{ys} \\ \text{interleave}' (xs, ()) &= xs \\ \text{interleave}' (x:xs, \text{ys}) &= x:\text{interleave}' (\text{ys}, xs) . \end{aligned}$$

函數 *interleave'* 是個良基定義 — 參數中的兩個串列雖然交換位置，但它們長度的總和會變小。也就是說 *interleave'* 可視為定義在這個良基序上的函數：

$$(xs_1, ys_1) \triangleleft (xs_2, ys_2) \equiv \text{length } xs_1 + \text{length } ys_1 < \text{length } xs_2 + \text{length } ys_2 \text{ .}$$

凡是 *interleave'* 有的性質，不難找出 *interleave* 的相對應版本；證明 *interleave* 的性質時，可當成是在證明 *interleave'* 的相對性質。因此我們也會比較寬鬆地說 *interleave* 也是 (\triangleleft) 之上的良基歸納定義。

例 2.21. 下列函數被稱作「McCarthy 91 函數」：

$$\begin{aligned} mc91 &:: \mathbb{N} \rightarrow \mathbb{N} \\ mc91 \ n \mid n > 100 &= n - 10 \\ &\mid otherwise = mc91 (mc91 (n + 11)) \text{ .} \end{aligned}$$

讀者不妨先猜猜看 *mc91* 會傳回什麼？答案是，*mc* 和以下函數是等價的：

$$\begin{aligned} mc91' \mid n > 100 &= n - 10 \\ &\mid otherwise = 91 \text{ .} \end{aligned}$$

[todo: finish this.]

2.10 詞典序歸納

我們終於要定義第 1.9 節與 2.9 節中都提到的「合併」函數：將兩個已排序好的串列合而為一。函數 *merge* 最自然的寫法可能是：

$$\begin{aligned} merge &:: \text{List Int} \rightarrow \text{List Int} \rightarrow \text{List Int} \\ merge [] \quad ys &= ys \\ merge (x:xs) [] &= x:xs \\ merge (x:xs) (y:ys) &= \text{if } x \leq y \text{ then } x:merge \ xs \ (y:ys) \\ &\quad \text{else } y:merge \ (x:xs) \ ys \text{ .} \end{aligned}$$

如果兩個串列之中有一個為空串列，合併的結果是另一個。如果兩個都不是空串列，我們比較其第一個元素，以便決定將哪個當作合併後的第一個元素。

但，*merge* 最後一個子句的第一個遞迴呼叫中，*y:ys* 沒有變短；第二個遞迴呼叫中，*x:xs* 沒有變短。這種程式會終止嗎？如果會，是哪種歸納定義呢？

一種看法是將 *merge* 視作和例 2.20 中的 *interleave* 類似的歸納定義：兩個參數的長度和在遞迴呼叫中變小了。另一個可能是將函數 *merge* 視作詞典序歸納 (lexicographic induction) 的例子 — 詞典序歸納也是良基歸納的一個特例。

先介紹詞典序。我們怎麼決定兩個英文單字在詞典中的先後順序呢？通常是先比較其第一個字母，如果第一個字母便分出了大小，就以此大小為準，不論剩下的字母為何。如果第一個字母一樣，便從第二個字母開始比起。若要以形式化的方式寫下詞典序的定義，我們考慮一個較簡單的狀況：如何比較 $x_1 y_1$ 和 $x_2 y_2$ 兩個長度均為二的字串？如果 ($<$) 是比較單一字元大小的順序，我們把兩個字元的詞典序寫成 ($<; <$)，定義如下：

$$x_1 y_1 (<; <) x_2 y_2 \equiv x_1 < x_2 \vee (x_1 = x_2 \wedge y_1 < y_2) \text{ .}$$

如前所述，先比較 x_1 與 x_2 ，如果相等，再比較 y_1 與 y_2 。

我們可以稍微擴充一些，考慮 x_i 與 y_i 型別不同的情況：

定義 2.22. 給定義在型別 A 之上的序 (\triangleleft) 和型別 B 之上的序 (\prec) ，它們的詞典序 (lexicographic ordering)，寫做 $(\triangleleft; \prec)$ ，是 $(A \times B)$ 上的一個序，定義為：

$$(x_1, y_1)(\triangleleft; \prec)(x_2, y_2) \equiv x_1 \triangleleft x_2 \vee (x_1 = x_2 \wedge y_1 \prec y_2) .$$

上述定義也可擴充到三個、四個... 元素的序對上。此處便不把他們寫出來了。

關於詞典序的有趣性質相當多，此處僅用到下述性質

定理 2.23. 如果 (\triangleleft) 與 (\prec) 均為良基序， $(\triangleleft; \prec)$ 也是良基序。

因此， $(\triangleleft; \prec)$ 也可用來做歸納定義。我們把使用詞典序的良基序歸納稱作「詞典序歸納」。

回頭看 *merge* 的定義。先考慮下述、在第 1.9 節中出現的 uncurried 版本：

```
merge' :: (List Int × List Int) → List Int
merge' ([], ys) = ys
merge' (x:xs, []) = x:xs
merge' (x:xs, y:ys) = if x ≤ y then x:merge' (xs, y:ys)
                      else y:merge' (x:xs, ys) .
```

如果 (\triangleleft) 是比較串列長度的良基序，我們可說 *merge'* 是在 $(\triangleleft; \triangleleft)$ 之上的歸納定義。確實，

- $(xs, y:ys)(\triangleleft; \triangleleft)(x:xs, y:ys)$ ，因為 $xs \triangleleft x:xs$;
- $(x:xs, ys)(\triangleleft; \triangleleft)(x:xs, y:ys)$ ，因為 $x:xs = x:xs$ 且 $ys \triangleleft y:ys$ 。

至於 *merge* 則是 *merge'* 的 curried 版本，因此也是定義良好的。

如前所述，函數 *merge* 的定義不一定得看成辭典序歸納 — 它也可和 *interleave* 一樣看成另一種較簡單的良基歸納 — 比較兩參數的長度之和。接下來的例子就得倚靠辭典序歸納了。

例 2.24. Ackermann 函數 (Ackermann's function) 有許多知名之處：它是一個遞增得相當快的函數。

```
ack :: ℕ → ℕ → ℕ
ack 0      n      = 1 + n
ack (1 + m) 0      = ack m 1
ack (1 + m) (1 + n) = ack m (ack (1 + m) n) .
```

該函數定義上的特殊處之一是 $ack(1+m)n$ 的結果又被當作 $ack m$ 的參數，因此較難以用理解 *interleave* 的方式理解。但它可視為詞典序 $(\triangleleft; \triangleleft)$ 上的歸納：

- $(m, 1)(\triangleleft; \triangleleft)(1+m, 0)$ ，因為 $m < 1+m$;
- $(1+m, n)(\triangleleft; \triangleleft)(1+m, 1+n)$ ，因為 $n < 1+n$;
- $(m, ack(1+m)n)(\triangleleft; \triangleleft)(1+m, 1+n)$ ，因為 $m < 1+m$ 。

2.11 交互歸納

許多工作無法由一個函數獨立完成，而需要許多函數彼此呼叫。本章最後談談這類的交互歸納 (mutual induction) 定義。下列函數定義中，*even* 判斷其輸入是否為偶數。第二個子句告訴我們：如果 *n* 是奇數， $1_+ n$ 便是偶數。但如何判斷一個數字是否為奇數？如果 *n* 是偶數， $1_+ n$ 便是奇數：

$$\begin{array}{ll} \text{even} :: \mathbb{N} \rightarrow \text{Bool} & \text{odd} :: \mathbb{N} \rightarrow \text{Bool} \\ \text{even } 0 &= \text{True} & \text{odd } 0 &= \text{False} \\ \text{even } (1_+ n) &= \text{odd } n, & \text{odd } (1_+ n) &= \text{even } n. \end{array}$$

這類彼此呼叫的定義可以視為一整個大定義。為讓讀者習慣，我們先把 *even* 與 *odd* 的定義改寫為 λ 算式與 **case**：

$$\begin{array}{ll} \text{even} = \lambda n \rightarrow \text{case } n \text{ of} & \text{odd} = \lambda n \rightarrow \text{case } n \text{ of} \\ \quad 0 &\rightarrow \text{True} & \quad 0 &\rightarrow \text{False} \\ \quad 1_+ n &\rightarrow \text{odd } n, & \quad 1_+ n &\rightarrow \text{even } n. \end{array}$$

上述的定義可以合併成一個：*evenOdd* 是一個序對，其中有兩個函數 $\mathbb{N} \rightarrow \text{Bool}$ ，其中是 *fst evenOdd* 就是 *even*，*snd evenOdd* 就是 *odd*：

$$\begin{array}{l} \text{evenOdd} :: ((\mathbb{N} \rightarrow \text{Bool}) \times (\mathbb{N} \rightarrow \text{Bool})) \\ \text{evenOdd} = (\lambda n \rightarrow \text{case } n \text{ of } \quad 0 \quad \rightarrow \text{True} \\ \quad \quad \quad 1_+ n \rightarrow \text{snd evenOdd } n, \\ \quad \lambda n \rightarrow \text{case } n \text{ of } \quad 0 \quad \rightarrow \text{False} \\ \quad \quad \quad 1_+ n \rightarrow \text{fst evenOdd } n). \end{array}$$

習題 2.34 — 證明 $\text{all } (xs ==) (\text{zipWith } (++) (\text{inits } xs) (\text{tails } xs))$. (unfinished)

2.12 參考資料

快速排序最初由 Hoare 在 Communications of the ACM 的演算法專欄中發表為兩個獨立的演算法：將陣列分割為大、小兩塊的「演算法 63: PARTITION」[Hoare, 1961b]，以及用前述演算法將陣列排序的「演算法 64: QUICKSORT」[Hoare, 1961c]。至於「演算法 65: FIND」[Hoare, 1961a] 的功能則是：給定 *k*，尋找一個陣列中第 *k* 小的元素。該演算法也使用了 PARTITION，我們現在常把它稱為「快速選擇 (quickselect)」。該專欄要求作者使用 Algol 語言。Algol 支援遞迴，Hoare 也大方地用了遞迴。當時遞迴仍是新觀念，Hoare 在另一篇論文 [Hoare, 1962] 中（以大量文字）描述不用遞迴的作法。

例 2.17 中的快速排序常被當作「函數編程（或 Haskell）漂亮又簡潔」的例證：快速排序用其他語言得寫得落落長，用 Haskell 可在兩三行內清楚地寫完。但這樣的比較並不很公平：例 2.17 排序的是串列，而拿來比較的對象通常是將陣列做排序的指令式語言程式。快速排序的主要挑戰之一，也是「演算法 63: PARTITION」的重點，是在 $O(n)$ 的時間、 $O(1)$ 的額外空間之內完成陣列的分塊。這點在串列版本中並沒有（或不須）表達出來。

Manna and McCarthy [1969] Manna and Pnueli [1970]

DRAFT

Bibliography

- J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- R. C. Backhouse. *Program Construction: Calculating Implementations from Specifications*. John Wiley & Sons, Ltd., 2003.
- R. S. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998. ISBN 0-13-484346-0.
- R. S. Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, 2010.
- H. B. Curry. Some philosophical aspects of combinatory logic. In J. Barwise, H. J. Keisler, and K. Kunen, editors, *The Kleene Symposium*, pages 85–101. North-Holland, 1980.
- E. W. Dijkstra. The notational conventions I adopted, and why. EWD 1300, 2000.
- E. W. Dijkstra. The next fifty years. EWD 1243, 2004.
- E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
- E. W. Dijkstra and A. J. M. van Gasteren. On naming. AvG 67 / EWD 958, 1986.
- F. L. G. Frege. Function and concept. translated by Peter T. Geach. In P. T. Geach and M. Black, editors, *Translations from the Philosophical Writings of Gottlob Frege*, pages 21–41. Basil Blackwell, 1960.
- D. Gries and F. B. Schneider. Calculational logic. <https://www.cs.cornell.edu/gries/Logic/intro.html>, 2003.
- D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math*. Springer, October 22, 1993.
- R. Hinze. La tour d'Hanoï. In A. Tolmach, editor, *International Conference on Functional Programming*, pages 3–10. ACM Press, 2009.
- C. A. R. Hoare. Algorithm 65: Find. *Communications of the ACM*, 4(7):321–322, 1961a.

- C. A. R. Hoare. Algorithm 63: Partition. *Communications of the ACM*, 4(7):321, 1961b.
- C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961c.
- C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, January 1962.
- P. Hudak, R. J. M. Hughes, S. L. Peyton Jones, and P. L. Wadler. A history of Haskell: being lazy with class. In B. Ryder and B. Hailpern, editors, *History of Programming Languages III*, pages 1–55. ACM Press, 2007.
- G. Hutton. *Programming in Haskell, 2nd Edition*. Cambridge University Press, 2016.
- Z. Manna and J. McCarthy. Properties of programs and partial function logic. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages 27–37. Edinburgh University Press, 1969.
- Z. Manna and A. Pnueli. Formalization of properties of functional programs. *Journal of the ACM*, 17(3):555–569, 1970.
- J. Mazur. *Enlightening Symbols: A Short History of Mathematical Notation and Its Hidden Powers*. Princeton University Press, 2014. 中譯本：啟蒙的符號：數學符號的誕生、演化和隱藏的力量。譯者：洪萬生，洪贊天，英家銘，黃俊瑋，黃美倫，鄭宜瑾。臉譜出版社，2015。
- J. Misra. A visionary decision. In M. Broy, editor, *Constructive Methods in Computing Science: International Summer School directed by F.L. Bauer, M. Broy, E.W. Dijkstra, C.A.R. Hoare*, pages 1–3. Springer-Verlag, 1989.
- N. Sankar, L. Allis, J. Seldi, J. Camp, S. Kahr, G. Leave, R. Botti, A. Dill, D. A. Turner, and J. Picton-Warlow. Curryng, or schonfinkeling? <http://computer-programming-forum.com/23-functional/976f118bb90d8b15.htm>, May 1997.
- M. Schönfinkel. Über die bausteinen der mathematische logik. *Mathematische Annalen*, 92:305–316, 1924. Translated by Stefan Bauer-Mengelberg as “On the building blocks of mathematical logic” in Jean van Heijenoort, 1967. *A Source Book in Mathematical Logic*, 1879–1931.
- C. Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13:11–49, 1967.
- J. L. van de Snepscheut. *What Computing Is All About*. Springer, 1993.

Index

- (\cdot), 25
- ($:$), 36, 37
- ($::$), 14, 37
- [], 36
- ($\$$), 28

- abstraction 抽象化, 6
- Ackermann's function Ackermann 函數, 82

- Boolean 布林值, 30

- calculational logic 演算邏輯, 11
- Cartesian product 笛卡兒積, 32
- Church-Rosser Theorem, 16
- combinator 組件, 39
- complete lattices 完全格, 70
- concurrency 共時, 14
- conjunction 合取、且, 30
- currying, 22, 34
 - uncurrying, 34

- disjunction 析取、或, 30
- dynamic programming 動態規劃, 77

- evaluation 求值, 14
 - applicative order 應用順序, 16
 - lazy evaluation 惰性求值, 17
 - normal order 範式順序, 16
- extensional equality 外延相等, 26

- Fibonacci number 費氏數, 75
- first-class citizen 一級公民, 21
- fixed point 定點, 71
 - postfixed point 後定點, 71
 - prefixed point 前定點, 71

- function composition 函數合成, 25
- function 函數
 - partial 部分函數, 37
 - total 全函數, 34

- greatest common divisor 最大公因數, 80
- greatest lower bound 最大下界, 70
- greatest postfix point 最大後定點, 71
- guard 守衛, 18

- higher-order function 高階函數, 21

- identity function 單位函數恆等函數, 26

- induction 歸納, 51
 - base case 基底, 53
 - complete induction 完全歸納, 75
 - induction hypothesis 歸納假設, 57
 - inductive step 歸納步驟, 53
 - lexicographic induction 詞典序歸納, 81
 - mutual 交互, 83
 - strong induction 強歸納, 77
 - well-founded induction 良基歸納, 77

- infimum 最大下界, 70
- isomorphism 同構, 34

- lattice 格, 70
- least prefix point 最小前定點, 71
- least upper bound 最小上界, 70

LISP, 48
list 串列, 36
 comprehension 串列建構式, 38
 prefix 前段, 41, 66
 segment 區段, 67
 suffix 後段, 41, 66
logic programming 邏輯編程, 14
 resolution 歸結, 14, 49
lower bound 下界, 70

map-fusion *map* 融合定理, 62
merge sort 合併排序, 79
 bottom-up 由下至上, 45
monoid 幺半群, 26

natural number 自然數, 52
normal form 範式, 16
 weak head 弱首範式, 35

pair 序對, 32
partial order 偏序, 70
partially ordered set 偏序集合, 70
pattern matching 樣式配對, 30
polymorphism 多型, 24
poset 偏序集合, 70
predicate 述語, 41, 52
preorder 前序, 70
program calculation 程式演算, 10
program derivation 程式推導, 10

quickselect 快速選擇, 83
quicksort 快速排序, 79, 83

recursion 遞迴, 51
redex 歸約點, 16
reduction 歸約, 15

semantics 語意, 5
supremum 最小上界, 70
syntax 語法, 5

total function 全函數, 51

upper bound 上界, 70

well-founded ordering 良基序, 78
wholemeal programming 全麥編程,