

有貓！

# Haskell 趣學指南

打不倒的空氣人，學不會的 Haskell

Gitbook

*Miran Lipovaca* 著  
MnO2 訳

# 目錄

介紹	0
Introduction	1
Ready Go	2
Type And Typeclass	3
Syntax in Function	4
Recursion	5
High Order Function	6
Module	7
Build Our Own Type and Typeclass	8
Input and Output	9
Functionally Solving Problems	10
Functors, Applicative Functors 與 Monoids	11
A Fistful of Monad	12
For a Few Monad More	13
Zippers	14
FAQ	15
Resource	16

# HASKELL 趣學指南

## 簡介

LEARN YOU A HASKELL FOR GREAT GOOD 中文版

## Top 貢獻者

- Fleuria
- letoh
- jiyingyiyong
- douglarek

## 社群

- [functional thursday](#)
- [haskell.tw](#)
- [haskell.sg](#)

## 貢獻

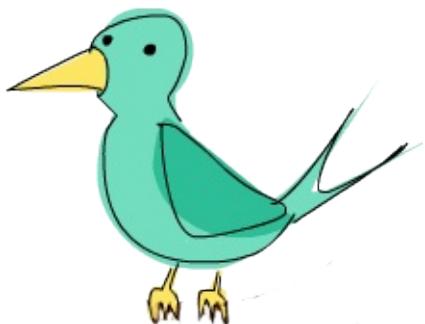
- [github repo](#)

# 簡介

## 關於這份教學

歡迎來到 Haskell 趣學指南！會想看這篇文章表示你對學習 Haskell 有很大的興趣。你來對地方了，來讓我簡單介紹一下這個教學。

撰寫這份教學，一方面是讓我自己對 Haskell 更熟練，另一方面是希望能夠分享我的學習經驗，幫助初學者更快進入狀況。網路上已經有無數 Haskell 的教學文件，在我學習的過程中，我並不限於只參考一份來源。我常常閱讀不同的教學文章，他們每個都從不同的角度出發。參考這些資源讓我能將知識化整為零。這份教學是希望提供更多的機會能讓你找到你想要得到的解答。



這份教學主要針對已經有使用命令式程式語言 (imperative programming languages) 寫程式經驗 (C, C++, Java, Python ...)、卻未曾接觸過函數式程式語言 (functional programming languages) (Haskell, ML, OCaml ...) 的讀者。就算沒有寫程式經驗也沒關係，會想學 Haskell 的人我相信都是很聰明的。

若在學習中遇到什麼地方不懂的，Freenode IRC 上的 #Haskell 頻道是提問的絕佳去處。那裡的人都很友善，有耐心且能體諒初學者。（譯註：Stackoverflow 上的 #haskell tag 也有很多 Haskell 神人們耐心地回答問題，提供給不習慣用 IRC 的人的另一個選擇。）

我經歷了不少挫折才學會 Haskell，在初學的時候它看起來是如此奇怪的語言。但有一天我突然開竅了，之後的學習便如魚得水。我想要表達的是：儘管 Haskell 乍看下如此地詭異，但假如你對程式設計十分有興趣，他非常值得你學習。學習 Haskell 讓你想起你第一次寫程式的感覺。非常有趣，而且強迫你 Think different。

## 什麼是 Haskell？

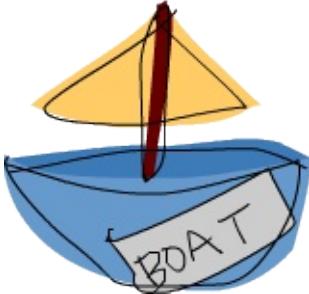


Haskell 與其他語言不同，是一門純粹函數式程式語言 (*purely functional programming language*)。在一般常見的命令式語言中，要執行操作的話是給電腦一組命令，而狀態會隨著命令的執行而改變。例如你指派變數 `a` 的值為 5，而隨後做了其它一些事情之後 `a` 就可能變成的其它值。有控制流程 (control flow)，你就可以重複執行操作。然而在純粹函數式程式語言中，你不是像命令式語言那樣命令電腦「要做什麼」，而是通過用函數來描述出問題「是什麼」，如「階乘是指從 1 到某個數的乘積」，「一個串列中數字的和」是指把第一個數字跟剩餘數字的和相加。你用宣告函數是什麼的形式來寫程式。另外，變數 (variable) 一旦被指定，就不可以更改了，你已經說了 `a` 就是 5，就不能再另說 `a` 是別的什麼數。（譯註：其實用 `variable` 來表達造成字義的 *overloading*，會讓人聯想到 *imperative languages* 中 `variable` 是代表狀態，但在 *functional languages* 中 `variable` 是相近於數學中使用的 `variable`。`x=5` 代表 `x` 就是 5，不是說 `x` 在 5 這個狀態。）所以說，在純粹函數式程式語言中的函數能做的唯一事情就是利用引數計算結果，不會產生所謂的“副作用 (side effect)”（譯註：也就是改變非函數內部的狀態，像是 *imperative languages* 裡面動到 *global variable* 就是 *side effect*）。一開始會覺得這限制很大，不過這也是他的優點所在：若以同樣的參數呼叫同一個函數兩次，得到的結果一定是相同。這被稱作“引用透明 (Referential Transparency)\_”（譯註：這就跟數學上函數的使用一樣）。如此一來編譯器就可以理解程式的行為，你也很容易就能驗證一個函數的正確性，繼而可以將一些簡單的函數組合成更複雜的函數。



Haskell 是惰性 (*lazy*) 的。也就是說若非特殊指明，函數在真正需要結果以前不會被求值。再加上引用透明，你就可以把程式僅看作是數據的一系列變形。如此一來就有了很多有趣的特性，如無限長度的資料結構。假設你有一個 List: `xs = [1, 2, 3, 4, 5, 6, 7, 8]`，還有一個函數 `doubleMe`，它可以將一個 List 中的所有元素都乘以二，返回一個新的 List。若是在命令式語言中，把一個 List 乘以 8，執行 `doubleMe(doubleMe(doubleMe(xs)))`，得遍歷三遍 `xs` 才會得到結果。而在惰性語言中，調用 `doubleMe` 時並不會立即求值，它會說“嗯嗯，待會兒再做！”。不過一旦要看結果，第一個 `doubleMe` 就會對第二個說“給我結果，快！”第二個

`doubleMe` 就會把同樣的話傳給第三個 `doubleMe`，第三個 `doubleMe` 只能將 1 乘以 2 得 2 後交給第二個，第二個再乘以 2 得 4 交給第一個，最終得到第一個元素 8。也就是說，這一切只需要遍歷一次 list 即可，而且僅在你真正需要結果時才會執行。惰性語言中的計算只是一組初始數據和變換公式。



Haskell 是靜態類型 (*statically typed*) 的。當你編譯程式時，編譯器需要明確哪個是數字，哪個是字串。這就意味着很大一部分錯誤都可以在編譯時被發現，若試圖將一個數字和字串相加，編譯器就會報錯。Haskell 擁有一套強大的類型系統，支持自動類型推導 (type inference)。這一來你就不需要在每段程式碼上都標明它的類型，像計算 `a=5+4`，你就不需另告訴編譯器“`a` 是一個數值”，它可以自己推導出來。類型推導可以讓你的程式更加簡練。假設有個函數是將兩個數值相加，你不需要聲明其類型，這個函數可以對一切可以相加的值進行計算。

Haskell 採納了很多高階程式語言的概念，因而它的程式碼優雅且簡練。與同層次的命令式語言相比，Haskell 的程式碼往往會更短，更短就意味着更容易理解，bug 也就更少。

Haskell 這語言是一群非常聰明的人設計的 (他們每個人都有 PhD 學位)。最初的工作始於 1987 年，一群學者聚在一起想設計一個屢到爆的程式語言。到了 2003 年，他們公開了 Haskell Report，這份報告描述了 Haskell 語言的一個穩定版本。(譯註：這份報告是 Haskell 98 標準的修訂版，Haskell 98 是在 1999 年公開的，是目前 Haskell 各個編譯器實現預設支援的標準。在 2010 年又公開了另一份 Haskell 2010 標準，詳情可見穆信成老師所撰寫的[簡介](#)。

## 要使用 Haskell 有哪些要求呢？

一句話版本的答案是：你只需要一個編輯器和一個編譯器。在這裡我們不會對編輯器多加著墨，你可以用任何你喜歡的編輯器。至於編譯器，在這份教學中我們會使用目前最流行的版本：GHC。而安裝 GHC 最方便的方法就是去下載 Haskell Platform，他包含了許多現成 Runtime Library 讓你方便寫程式。(譯註：Ubuntu 的使用者有現成的套件可以使用，可以直接 `apt-get install Haskell-platform` 來安裝。但套件的版本有可能比較舊。)

GHC 可以解釋執行 Haskell Script (通常是以 `.hs` 作為結尾)，也可以編譯。它還有個互動模式，你可以在裡面呼叫 Script 裡定義的函數，即時得到結果。對於學習而言，這比每次修改都編譯執行要方便的多。想進入互動模式，只要打開控制台輸入 `ghci` 即可。假設你在 `myfunctions.hs` 裡定義了一些函數，在 `ghci` 中輸入 `:l myfunctions.hs`，`ghci` 便會載入

`myfunctions.hs`。之後你便可以呼叫你定義的函數。一旦修改了這個 `.hs` 檔案的內容，再次執行 `:l myfunctions.hs` 或者相同作用的 `:r`，都可以重新載入該檔案。我自己通常就是在 `.hs` 檔案中定義幾個函數，再到 ghci 載入，呼叫看看，再修改再重新載入。這也正是我們往後的基本流程。

# 從零開始

## 準備好了嗎？



準備來開始我們的旅程！如果你就是那種從不看說明書的人，我推薦你還是回頭看一下簡介的最後一節。那裡面講了這個教學中你需要用到的工具及基本用法。我們首先要做的就是進入 ghci 的互動模式，接着就可以寫幾個函數體驗一下 Haskell 了。打開終端機，輸入 `ghci`，你會看到下列歡迎訊息：

```
GHCI, version 6.8.2: http://www.haskell.org/ghc/
?: for help  Loading package base ... linking ... done.
Prelude>
```

恭喜您已經進入了 ghci 了！目前它的命令列提示是 `prelude>`，不過它在你裝載一些模組之後會變的比較長。為了美觀起見，我們會輸入指令 `:set prompt "ghci>"` 把它改成 `ghci>`。

首先來看一些簡單的運算

```
ghci> 2 + 15
17
ghci> 49 * 100
4900
ghci> 1892 - 1472
420
ghci> 5 / 2
2.5
ghci>
```

很簡單吧！你也可以在一行中使用多個運算子，他們會按照運算子優先順序執行計算，而使用括號可以改變執行的優先順序。

```
ghci> (50 * 100) - 4999
1
ghci> 50 * 100 - 4999
1
ghci> 50 * (100 - 4999)
-244950
```

但注意處理負數的時候有個小陷阱：我們執行 `5 * -3` 會 ghci 會回報錯誤。所以說，使用負數時最好將其置於括號之中，像 `5*(-3)` 就不會有問題。

要進行布林代數 (Boolean Algebra) 的演算也是很直覺的。你也許早就會猜，`&&` 指的是布林代數上的 AND，而 `||` 指的是布林代數上的 OR，`not` 會把 `True` 變成 `False`，`False` 變成 `True`。

```
ghci> True && False
False
ghci> True && True
True
ghci> False || True
True
ghci> not False
True
ghci> not (True && True)
False
```

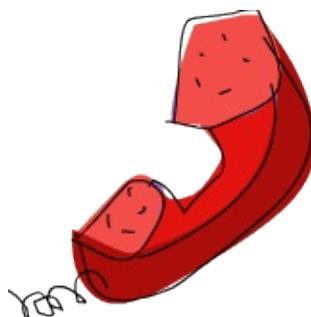
相等性可以這樣判定

```
ghci> 5 == 5
True
ghci> 1 == 0
False
ghci> 5 /= 5
False
ghci> 5 /= 4
True
ghci> "hello" == "hello"
True
```

那執行 `5+"llama"` 或者 `5==True` 會怎樣？如果我們真的試著在 ghci 中跑，會得到下列的錯誤訊息：

```
No instance for (Num [Char])
arising from a use of `+' at :1:0-9
Possible fix: add an instance declaration for (Num [Char])
In the expression: 5 + "llama"
In the definition of `it': it = 5 + "llama"
```

這邊 ghci 提示說 "11ama" 並不是數值型別，所以它不知道該怎樣才能給它加上 5。即便是 "four" 甚至是 "4" 也不可以，Haskell 不拿它當數值。執行 `True==5`，ghci 就會提示型別不匹配。`+` 運算子要求兩端都是數值，而 `==` 運算子僅對兩個可比較的值可用。這就要求他們的型別都必須一致，蘋果和橘子就無法做比較。我們會在後面深入地理解型別的概念。Note: `5+4.0` 是可以執行的，5 既可以做被看做整數也可以被看做浮點數，但 4.0 則不能被看做整數。



也許你並未察覺，不過從始至終我們一直都在使用函數。`*` 就是一個將兩個數相乘的函數，就像三明治一樣，用兩個參數將它夾在中央，這被稱作中綴函數。而其他大多數不能與數夾在一起的函數則被稱作前綴函數。絕大部分函數都是前綴函數，在接下來我們就不多做區別。大多數命令式程式語言中的函數呼叫形式通常就是函數名，括號，由逗號分隔的參數列。而在 Haskell 中，函數呼叫的形式是函數名，空格，空格分隔的參數列。簡單舉個例子，我們呼叫 Haskell 中最無趣的函數：

```
ghci> succ 8
9
```

`succ` 函數返回一個數的後繼 (successor)。而且如你所見，在 Haskell 中是用空格來將函數與參數分隔的。至於呼叫多個參數的函數也很容易，`min` 和 `max` 接受兩個可比較大小的參數，並返回較大或者較小的那個數。

```
ghci> min 9 10
9
ghci> min 3.4 3.2
3.2
ghci> max 100 101
101
```

函數呼叫擁有最高的優先順序，如下兩句是等效的

```
ghci> succ 9 + max 5 4 + 1
16
ghci> (succ 9) + (max 5 4) + 1
16
```

若要取 9 乘 10 的後繼，`succ 9*10` 是不行的，程式會先取 9 的後繼，然後再乘以 10 得 100。正確的寫法應該是 `succ(9*10)`，得 91。如果某函數有兩個參數，也可以用 ``` 符號將它括起，以中綴函數的形式呼叫它。

例如取兩個整數相除所得商的 `div` 函數，`div 92 10` 可得 9，但這種形式不容易理解：究竟是哪個數是除數，哪個數被除？使用中綴函數的形式 `92 `div` 10` 就更清晰了。

從命令式程式語言走過來的人們往往會覺得函數呼叫與括號密不可分，在 C 中，呼叫函數必加括號，就像 `foo()`，`bar(1)`，或者 `baz(3, "haha")`。而在 Haskell 中，函數的呼叫使用空格，例如 `bar (bar 3)`，它並不表示以 `bar` 和 3 兩個參數去呼叫 `bar`，而是以 `bar 3` 所得的結果作為參數去呼叫 `bar`。在 C 中，就相當於 `bar(bar(3))`。

## 初學者的第一個函數

在前一節中我們簡單介紹了函數的呼叫，現在讓我們編寫我們自己的函數！打開你最喜歡的編輯器，輸入如下程式碼，它的功能就是將一個數字乘以 2。

```
doubleMe x = x + x
```

函數的聲明與它的呼叫形式大致相同，都是先函數名，後跟由空格分隔的參數表。但在聲明中一定要在 `=` 後面定義函數的行為。

保存為 `baby.hs` 或任意名稱，然後轉至保存的位置，打開 ghci，執行 `:l baby.hs`。這樣我們的函數就裝載成功，可以呼叫了。

```
ghci> :l baby
[1 of 1] Compiling Main           ( baby.hs, interpreted )
Ok, modules loaded: Main.
ghci> doubleMe 9
18
ghci> doubleMe 8.3
16.6
```

+ 運算子對整數和浮點都可用(實際上所有有數字特徵的值都可以)，所以我們的函數可以處理一切數值。聲明一個包含兩個參數的函數如下：

```
doubleUs x y = x*2 + y*2
```

很簡單。將其寫成 `doubleUs x y = x + x + y + y` 也可以。測試一下(記住要保存為 `baby.hs` 併到 ghci 下邊執行 `:l baby.hs`)

```
ghci> doubleUs 4 9
26
ghci> doubleUs 2.3 34.2
73.0
ghci> doubleUs 28 88 + doubleMe 123
478
```

你可以在其他函數中呼叫你編寫的函數，如此一來我們可以將 `doubleUs` 函數改為：

```
doubleUs x y = doubleMe x + doubleMe y
```



這種情形在 Haskell 下邊十分常見：編寫一些簡單的函數，然後將其組合，形成一個較為複雜的函數，這樣可以減少重複工作。設想若是哪天有個數學家驗證說 2 應該是 3，我們只需要將 `doubleMe` 改為 `x+x+x` 即可，由於 `doubleUs` 呼叫到 `doubleMe`，於是整個程式便進入了 2 即是 3 的古怪世界。

Haskell 中的函數並沒有順序，所以先聲明 `doubleUs` 還是先聲明 `doubleMe` 都是同樣的。如下，我們編寫一個函數，它將小於 100 的數都乘以 2，因為大於 100 的數都已經足夠大了！

```
doubleSmallNumber x = if x > 100
    then x
    else x*2
```

接下來介紹 Haskell 的 `if` 語句。你也許會覺得和其他語言很像，不過存在一些不同。

Haskell 中 `if` 語句的 `else` 部分是不可省略。在命令式語言中，你可以通過 `if` 語句來跳過一段程式碼，而在 Haskell 中，每個函數和表達式都要返回一個結果。對於這點我覺得將 `if` 語句置於一行之中會更易理解。Haskell 中的 `if` 語句的另一個特點就是它其實是個表達式，表達式就是返回一個值的一段程式碼：`5` 是個表達式，它返回 `5`；`4+8` 是個表達式；`x+y` 也是個表達式，它返回 `x+y` 的結果。正由於 `else` 是強制的，`if` 語句一定會返回某個值，所以說 `if` 語句也是個表達式。如果要給剛剛定義的函數的結果都加上 1，可以如此修改：

```
doubleSmallNumber' x = (if x > 100 then x else x*2) + 1
```

若是去掉括號，那就會只在小於 100 的時候加 1。注意函數名最後的那個單引號，它沒有任何特殊含義，只是一個函數名的合法字元罷了。通常，我們使用單引號來區分一個稍經修改但差別不大的函數。定義這樣的函數也是可以的：

```
conanO'Brien = "It's a-me, Conan O'Brien!"
```

在這裡有兩點需要注意。首先就是我們沒有大寫 `conan` 的首字母，因為首字母大寫的函數是不允许許的，稍後我們將討論其原因；另外就是這個函數並沒有任何參數。沒有參數的函數通常被稱作“定義”(或者“名字”)，一旦定義，`conanO'Brien` 就與字串 `"It's a-me, Conan O'Brien!"` 完全等價，且它的值不可以修改。

## List 入門



在 Haskell 中，List 就像現實世界中的購物單一樣重要。它是最常用的資料結構，並且十分強大，靈活地使用它可以解決很多問題。本節我們將對 List，字串和 list comprehension 有個初步瞭解。在 Haskell 中，List 是一種單型別的資料結構，可以用來存儲多個型別相同的元素。我們可以在裡面裝一組數字或者一組字元，但不能把字元和數字裝在一起。

\*Note\*: 在 ghci 下，我們可以使用 ``let`` 關鍵字來定義一個常量。在 ghci 下執行 ``let a=1`` 與在腳本

```
ghci> let lostNumbers = [4, 8, 15, 16, 23, 48]
ghci> lostNumbers
[4, 8, 15, 16, 23, 48]
```

如你所見，一個 List 由方括號括起，其中的元素用逗號分隔開來。若試圖寫

`[1, 2, 'a', 3, 'b', 'c', 4]` 這樣的 List，Haskell 就會報出這幾個字元不是數字的錯誤。字串實際上就是一組字元的 List，`"Hello"` 只是 `['h', 'e', 'l', 'l', 'o']` 的語法糖而已。所以我們可以使用處理 List 的函數來對字串進行操作。將兩個 List 合併是很常見的操作，這可以通過 `++` 運算子實現。

```
ghci> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
ghci> "hello" ++ " " ++ "world"
"hello world"
ghci> ['w','o'] ++ ['o','t']
"woot"
```

在使用 `++` 運算子處理長字串時要格外小心(對長 List 也是同樣), Haskell 會遍歷整個的 List(`++` 符號左邊的那個)。在處理較短的字串時問題還不大, 但要是在一個 5000 萬長度的 List 上追加元素, 那可得執行好一會兒了。所以說, 用 `:` 運算子往一個 List 前端插入元素會是更好的選擇。

```
ghci> 'A':" SMALL CAT"
"A SMALL CAT"
ghci> 5:[1,2,3,4,5]
[5,1,2,3,4,5]
```

`:` 運算子可以連接一個元素到一個 List 或者字串之中, 而 `++` 運算子則是連接兩個 List。若要使用 `++` 運算子連接單個元素到一個 List 之中, 就用方括號把它括起使之成為單個元素的 List。`[1,2,3]` 實際上是 `1:2:3:[]` 的語法糖。`[]` 表示一個空 List, 若要從前端插入 3, 它就成了 `[3]`, 再插入 2, 它就成了 `[2,3]`, 以此類推。

\*Note\*: ``[], [[]], [[], []], []`` 是不同的。第一個是一個空的 List, 第二個是含有一個空 List 的 List,

若是要按照索引取得 List 中的元素, 可以使用 `!!` 運算子, 索引的下標為 0。

```
ghci> "Steve Buscemi" !! 6
'B'
ghci> [9.4,33.2,96.2,11.2,23.25] !! 1
33.2
```

但你若是試圖在一個只含有 4 個元素的 List 中取它的第 6 個元素, 就會報錯。要小心 !

List 同樣也可以用來裝 List, 甚至是 List 的 List 的 List :

```
ghci> let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b ++ [[1,1,1,1]]
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3],[1,1,1,1]]
ghci> [6,6,6]:b
[[6,6,6],[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b !! 2
[1,2,2,3,4]
```

List 中的 List 可以是不同長度，但必須得是相同的型別。如不可以將 List 中混合放置字元和數組相同，混合放置數值和字元的 List 也是同樣不可以的。當 List 內裝有可比較的元素時，使用 `>` 和 `>=` 可以比較 List 的大小。它會先比較第一個元素，若它們的值相等，則比較下一個，以此類推。

```
ghci> [3,2,1] > [2,1,0]
True
ghci> [3,2,1] > [2,10,100]
True
ghci> [3,4,2] > [3,4]
True
ghci> [3,4,2] > [2,4]
True
ghci> [3,4,2] == [3,4,2]
True
```

還可以對 List 做啥？如下是幾個常用的函數：

**head** 返回一個 List 的頭部，也就是 List 的首個元素。

```
ghci> head [5,4,3,2,1]
5
```

**tail** 返回一個 List 的尾部，也就是 List 除去頭部之後的部分。

```
ghci> tail [5,4,3,2,1]
[4,3,2,1]
```

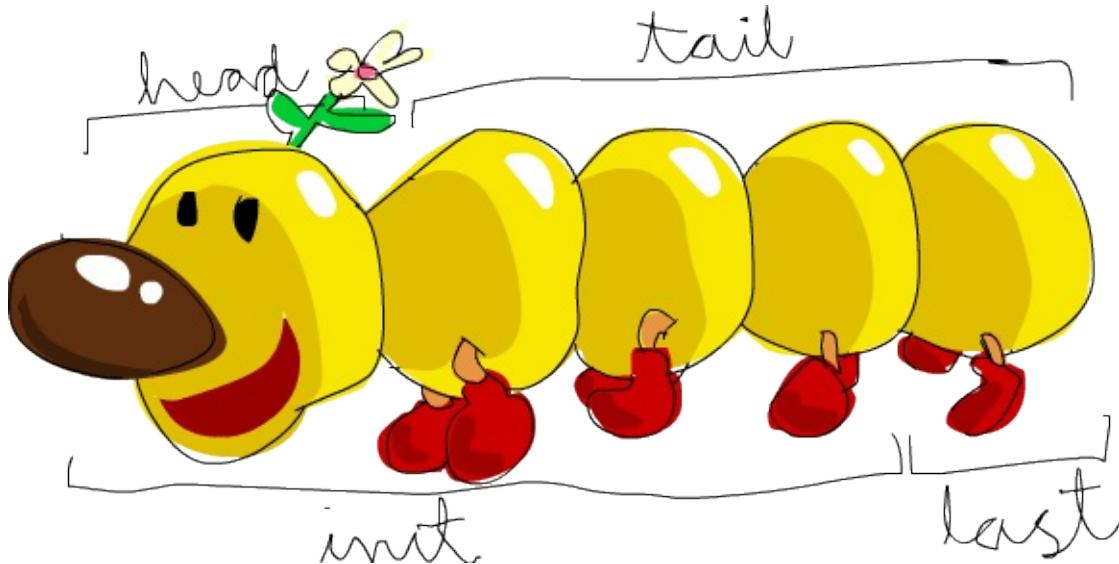
**last** 返回一個 List 的最後一個元素。

```
ghci> last [5,4,3,2,1]
1
```

**init** 返回一個 List 除去最後一個元素的部分。

```
ghci> init [5,4,3,2,1]
[5,4,3,2]
```

如果我們把 List 想象為一頭怪獸，那這就是它的樣子：



試一下，若是取一個空 List 的 `head` 又會怎樣？

```
ghci> head []
*** Exception: Prelude.head: empty list
```

糟糕，程式直接跳出錯誤。如果怪獸都不存在的話，那他的頭也不會存在。在使用 `head`, `tail`, `last` 和 `init` 時要小心別用到空的 List 上，這個錯誤不會在編譯時被捕獲。所以說做些工作以防止從空 List 中取值會是個好的做法。

**length** 返回一個 List 的長度。

```
ghci> length [5,4,3,2,1]
5
```

**null** 檢查一個 List 是否為空。如果是，則返回 `True`，否則返回 `False`。應當避免使用 `xs==[]` 之類的語句來判斷 List 是否為空，使用 `null` 會更好。

```
ghci> null [1,2,3]
False
ghci> null []
True
```

**reverse** 將一個 List 反轉：

```
ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]
```

**take** 返回一個 List 的前幾個元素，看：

```
ghci> take 3 [5,4,3,2,1]
[5,4,3]
ghci> take 1 [3,9,3]
[3]
ghci> take 5 [1,2]
[1,2]
ghci> take 0 [6,6,6]
[]
```

如上，若是圖取超過 List 長度的元素個數，只能得到原 List。若 `take 0` 個元素，則會得到一個空 List！`drop` 與 `take` 的用法大體相同，它會刪除一個 List 中的前幾個元素。

```
ghci> drop 3 [8,4,2,1,5,6]
[1,5,6]
ghci> drop 0 [1,2,3,4]
[1,2,3,4]
ghci> drop 100 [1,2,3,4]
[]
```

**maximum** 返回一個 List 中最大的那個元素。**minimum** 返回最小的。

```
ghci> minimum [8,4,2,1,5,6]
1
ghci> maximum [1,9,2,3,4]
9
```

**sum** 返回一個 List 中所有元素的和。**product** 返回一個 List 中所有元素的積。

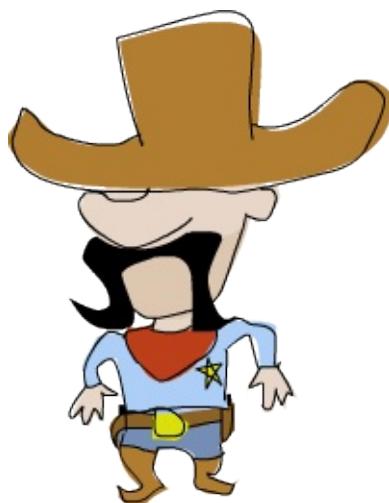
```
ghci> sum [5,2,1,6,3,2,5,7]
31
ghci> product [6,2,1,2]
24
ghci> product [1,2,5,6,7,9,2,0]
0
```

**elem** 判斷一個元素是否在包含於一個 List，通常以中綴函數的形式呼叫它。

```
ghci> 4 `elem` [3,4,5,6]
True
ghci> 10 `elem` [3,4,5,6]
False
```

這就是幾個基本的 List 操作函數，我們會在往後的一節中瞭解更多的函數。

## 使用 Range



今天如果想得到一個包含 1 到 20 之間所有數的 List，你會怎麼做？我們可以將它們一個一個用鍵盤打出來，但很明顯地這不是一個完美的方案，特別是你追求一個好的程式語言的時候。我們想用的是區間 (Range)。Range 是構造 List 方法之一，而其中的值必須是可枚舉的，像 1、2、3、4...字元同樣也可以枚舉，字母表就是 `A..Z` 所有字元的枚舉。而名字就不可以枚舉了，"john" 後面是誰？我不知道。

要得到包含 1 到 20 中所有自然數的 List，只要 `[1..20]` 即可，這與用手寫 `[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]` 是完全等價的。其實用手寫一兩個還不是什麼大事，但若是手寫一個非常長的 List 那就鐵定是個笨方法。

```
ghci> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> ['K'..'Z']
"KLMNOPQRSTUVWXYZ"
```

Range 的特點是他還允許你指定每一步該跨多遠。譬如說，今天的問題換成是要得到 1 到 20 間所有的偶數或者 3 的倍數該怎樣？

```
ghci> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
ghci> [3,6..20]
[3,6,9,12,15,18]
```

僅需用逗號將前兩個元素隔開，再標上上限即可。儘管 Range 很聰明，但它恐怕還滿足不了一些人對它的期許。你就不能通過 `[1,2,4..100]` 這樣的語句來獲得所有 2 的幕。一方面是因为步長只能標明一次，另一方面就是僅憑前幾項，數組的後項是不能確定的。要得到 20 到 1 的 List，`[20..1]` 是不可以的。必須得 `[20,19..1]`。在 Range 中使用浮點數要格外小心！出於定義的原因，浮點數並不精確。若是使用浮點數的話，你就會得到如下的糟糕結果

```
ghci> [0.1, 0.3 .. 1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
```

我的建議就是避免在 Range 中使用浮點數。

你也可以不標明 Range 的上限，從而得到一個無限長度的 List。在後面我們會講解關於無限 List 的更多細節。取前 24 個 13 的倍數該怎樣？恩，你完全可以 `[13,26..24*13]`，但有更好的方法：`take 24 [13,26..]`。

由於 Haskell 是惰性的，它不會對無限長度的 List 求值，否則會沒完沒了的。它會等着，看你會從它那兒取多少。在這裡它見你只要 24 個元素，便欣然交差。如下是幾個生成無限 List 的函數 `cycle` 接受一個 List 做參數並返回一個無限 List。如果你只是想看一下它的運算結果而已，它會運行個沒完的。所以應該在某處劃好範圍。

```
ghci> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
ghci> take 12 (cycle "LOL ")
"LOL LOL LOL "
```

`repeat` 接受一個值作參數，並返回一個僅包含該值的無限 List。這與用 `cycle` 處理單元素 List 差不多。

```
ghci> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
```

其實，你若只是想得到包含相同元素的 List，使用 `replicate` 會更簡單，如 `replicate 3 10`，得 `[10,10,10]`。

## List Comprehension



學過數學的你對集合的 comprehension (Set Comprehension) 概念一定不會陌生。通過它，可以從既有的集合中按照規則產生一個新集合。前十個偶數的 set comprehension 可以表示為  $S = \{2 \cdot x \mid x \in \mathbb{N}, x \leq 10\}$ ，豎綫左端的部分是輸出函數，`x` 是變數，`N` 是輸入集合。在 Haskell 下，我們可以通過類似 `take 10 [2,4..]` 的程式碼來實現。但若是把簡單的乘 2 改成更複雜的函數操作該怎麼辦呢？用 list comprehension，它與 set comprehension 十分的相似，用它取前十個偶數輕而易舉。這個 list comprehension 可以表示為：

```
ghci> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
```

如你所見，結果正確。給這個 comprehension 再添個限制條件 (predicate)，它與前面的條件由一個逗號分隔。在這裡，我們要求只取乘以 2 後大於等於 12 的元素。

```
ghci> [x*2 | x <- [1..10], x*2 >= 12]
[12,14,16,18,20]
```

cool，靈了。若是取 50 到 100 間所有除7的餘數為 3 的元素該怎麼辦？簡單：

```
ghci> [x | x <- [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]
```

成功！從一個 List 中篩選出符合特定限制條件的操作也可以稱為過濾 (filtering)。即取一組數並且按照一定的限制條件過濾它們。再舉個例子吧，假如我們想要一個 comprehension，它能夠使 List 中所有大於 10 的奇數變為 "BANG"，小於 10 的奇數變為 "BOOM"，其他則統統扔掉。方便重用起見，我們將這個 comprehension 置於一個函數之中。

```
boomBangs xs = [if x < 10 then "BOOM!" else "BANG!" | x <- xs, odd x]
```

這個 comprehension 的最後部分就是限制條件，使用 `odd` 函數判斷是否為奇數：返回 `True`，就是奇數，該 List 中的元素才被包含。

```
ghci> boomBangs [7..13]
["BOOM!","BOOM!","BANG!","BANG!"]
```

也可以加多個限制條件。若要達到 10 到 20 間所有不等於 13, 15 或 19 的數，可以這樣：

```
ghci> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20]
```

除了多個限制條件之外，從多個 List 中取元素也是可以的。這樣的話 comprehension 會把所有的元素組合交付給我們的輸出函數。在不過濾的前提下，取自兩個長度為 4 的集合的 comprehension 會產生一個長度為 16 的 List。假設有兩個 List, [2,5,10] 和 [8,10,11]，要取它們所有組合的積，可以這樣：

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

意料之中，得到的新 List 長度為 9。若只取乘積大於 50 的結果該如何？

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]
[55,80,100,110]
```

取個包含一組名詞和形容詞的 List comprehension 吧，寫詩的話也許用得着。

```
ghci> let nouns = ["hobo", "frog", "pope"]
ghci> let adjectives = ["lazy", "grouchy", "scheming"]
ghci> [adjective ++ " " ++ noun | adjective <- adjectives, noun <- nouns]
["lazy hobo", "lazy frog", "lazy pope", "grouchy hobo", "grouchy frog", "grouchy pope", "scheming hobo", "scheming frog", "scheming pope"]
```

明白！讓我們編寫自己的 `length` 函數吧！就叫做 `length'`！

```
length' xs = sum [1 | _ <- xs]
```

`_` 表示我們並不關心從 List 中取什麼值，與其弄個永遠不用的變數，不如直接一個 `_`。這個函數將一個 List 中所有元素置換為 1，並且使其相加求和。得到的結果便是我們的 List 長度。友情提示：字串也是 List，完全可以使用 list comprehension 來處理字串。如下是個除去字串中所有非大寫字母的函數：

```
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

測試一下：

```
ghci> removeNonUppercase "Hahaha! Ahahaha!"
"HA"
ghci> removeNonUppercase "IdontLIKEFROGS"
"ILIKEFROGS"
```

在這裡，限制條件做了所有的工作。它說：只有在 `['A'..'z']` 之間的字元才可以被包含。

若操作含有 List 的 List，使用嵌套的 List comprehension 也是可以的。假設有個包含許多數值的 List 的 List，讓我們在不拆開它的前提下除去其中的所有奇數：

```
ghci> let xxs = [[1,3,5,2,3,1,2,4,5],[1,2,3,4,5,6,7,8,9],[1,2,4,2,1,6,3,1,3,2,3,6]]
ghci> [ [ x | x <- xs, even x ] | xs <- xxs]
[[2,2,4],[2,4,6,8],[2,4,2,6,2,6]]
```

將 List Comprehension 分成多行也是可以的。若非在 ghci 之下，還是將 List Comprehension 分成多行好，尤其是需要嵌套的時候。

## Tuple



從某種意義上講，Tuple (元組)很像 List --都是將多個值存入一個個體的容器。但它們卻有着本質的不同，一組數字的 List 就是一組數字，它們的型別相同，且不關心其中包含元素的數量。而 Tuple 則要求你對需要組合的數據的數目非常的明確，它的型別取決於其中項的數目與其各自的型別。Tuple 中的項由括號括起，並由逗號隔開。

另外的不同之處就是 Tuple 中的項不必為同一型別，在 Tuple 裡可以存入多型別項的組合。

動腦筋，在 Haskell 中表示二維向量該如何？使用 List 是一種方法，它倒也工作良好。若要將一組向量置於一個 List 中來表示平面圖形又該怎樣？我們可以寫類似 `[[1,2],[8,11],[4,5]]` 的程式碼來實現。但問題在於，`[(1,2),(8,11,5),(4,5)]` 也是同樣合法的，因為其中元素的型別都相同。儘管這樣並不靠譜，但編譯時並不會報錯。然而一個長度為 2 的 Tuple (也可以稱作序對，Pair)，是一個獨立的類型，這便意味着一個包含一組序對的 List 不能再加入一個三元組，所以說把原先的方括號改為圓括號使用 Tuple 會更好：`[(1,2),(8,11),(4,5)]`。若試圖表示這樣的圖形：`[(1,2),(8,11,5),(4,5)]`，就會報出以下的錯誤：

```
Couldn't match expected type `'(t, t1)'
against inferred type `'(t2, t3, t4)'
In the expression: (8, 11, 5)
In the expression: [(1, 2), (8, 11, 5), (4, 5)]
In the definition of `it': it = [(1, 2), (8, 11, 5), (4, 5)]
```

這告訴我們說程式在試圖將序對和三元組置於同一 List 中，而這是不允許的。同樣 `[(1, 2), ("one", 2)]` 這樣的 List 也不行，因為其中的第一個 Tuple 是一對數字，而第二個 Tuple 却成了一個字串和一個數字。Tuple 可以用來儲存多個數據，如，我們要表示一個人的名字與年齡，可以使用這樣的 Tuple: `("Christopher", "Walken", 55)`。從這個例子裡也可以看出，Tuple 中也可以存儲 List。

使用 Tuple 前應當事先明確一條數據中應該由多少個項。每個不同長度的 Tuple 都是獨立的型別，所以你就不可以寫個函數來給它追加元素。而唯一能做的，就是通過函數來給一個 List 追加序對，三元組或是四元組等內容。

可以有單元素的 List，但 Tuple 不行。想想看，單元素的 Tuple 本身就只有一個值，對我們又有啥意義？不靠譜。

同 List 相同，只要其中的項是可比較的，Tuple 也可以比較大小，只是你不可以像比較不同長度的 List 那樣比較不同長度的 Tuple。如下是兩個有用的序對操作函數：

**fst** 返回一個序對的首項。

```
ghci> fst (8, 11)
8
ghci> fst ("Wow", False)
"Wow"
```

**snd** 返回序對的尾項。

```
ghci> snd (8, 11)
11
ghci> snd ("Wow", False)
False
```

\*Note\*：這兩個函數僅對序對有效，而不能應用於三元組，四元組和五元組之上。稍後，我們將過一遍從 Tuple 中取數。

有個函數很 cool，它就是 `zip`。它可以用來生成一組序對 (Pair) 的 List。它取兩個 List，然後將它們交叉配對，形成一組序對的 List。它很簡單，卻很實用，尤其是你需要組合或是遍歷兩個 List 時。如下是個例子：

```
ghci> zip [1,2,3,4,5] [5,5,5,5,5]
[(1,5),(2,5),(3,5),(4,5),(5,5)]
ghci> zip [1 .. 5] ["one", "two", "three", "four", "five"]
[(1,"one"),(2,"two"),(3,"three"),(4,"four"),(5,"five")]
```

它把元素配對並返回一個新的 List。第一個元素配第一個，第二個元素配第二個..以此類推。注意，由於序對中可以含有不同的型別，`zip` 函數可能會將不同型別的序對組合在一起。若是兩個不同長度的 List 會怎麼樣？

```
ghci> zip [5,3,2,6,2,7,2,5,4,6,6] ["im","a","turtle"]
[(5,"im"),(3,"a"),(2,"turtle")]
```

較長的那個會在中間斷開，去匹配較短的那個。由於 Haskell 是惰性的，使用 `zip` 同時處理有限和無限的 List 也是可以的：

```
ghci> zip [1..] ["apple", "orange", "cherry", "mango"]
[(1,"apple"),(2,"orange"),(3,"cherry"),(4,"mango")]
```

接下來考慮一個同時應用到 List 和 Tuple 的問題：如何取得所有三邊長度皆為整數且小於等於 10，周長為 24 的直角三角形？首先，把所有三邊長度小於等於 10 的三角形都列出來：

```
ghci> let triangles = [ (a,b,c) | c <- [1..10], b <- [1..10], a <- [1..10] ]
```

剛才我們是從三個 List 中取值，並且通過輸出函數將其組合為一個三元組。只要在 ghci 下邊呼叫 `triangle`，你就會得到所有三邊都小於等於 10 的三角形。我們接下來給它添加一個限制條件，令其必須為直角三角形。同時也考慮上 `b` 邊要短於斜邊，`a` 邊要短於 `b` 邊情況：

```
ghci> let rightTriangles = [ (a,b,c) | c <- [1..10], b <- [1..c], a <- [1..b], a^2 + b^2
```

已經差不多了。最後修改函數，告訴它只要周長為 24 的三角形。

```
ghci> let rightTriangles' = [ (a,b,c) | c <- [1..10], b <- [1..c], a <- [1..b], a^2 + b^2
ghci> rightTriangles'
[(6,8,10)]
```

得到正確結果！這便是函數式程式語言的一般思路：先取一個初始的集合併將其變形，執行過濾條件，最終取得正確的結果。

# Types and Typeclasses

## Type



之前我們有說過 Haskell 是 Static Type，這表示在編譯時期每個表達式的型別都已經確定下來，這提高了程式碼的安全性。若程式碼中有讓布林值與數字相除的動作，就不會通過編譯。這樣的好處就是與其讓程序在運行時崩潰，不如在編譯時就找出可能的錯誤。Haskell 中所有東西都有型別，因此在編譯的時候編譯器可以做到很多事情。

與 Java 和 Pascal 不同，Haskell 支持型別推導。寫下一個數字，你就沒必要另告訴 Haskell 說"它是個數字"，它自己能推導出來。這樣我們就不必在每個函數或表達式上都標明其型別了。在前面我們只簡單涉及一下 Haskell 的型別方面的知識，但是理解這一型別系統對於 Haskell 的學習是至關重要的。

型別是每個表達式都有的某種標籤，它標明了這一表達式所屬的範疇。例如，表達式 `True` 是 `boolean` 型，`"hello"` 是個字串，等等。

可以使用 `ghci` 來檢測表達式的型別。使用 `:t` 命令後跟任何可用的表達式，即可得到該表達式的型別，先試一下：

```
ghci> :t 'a'
'a' :: Char
ghci> :t True
True :: Bool
ghci> :t "HELLO!"
"HELLO!" :: [Char]
ghci> :t (True, 'a')
(True, 'a') :: (Bool, Char)
ghci> :t 4 == 5
4 == 5 :: Bool
```



可以看出，`:t` 命令處理一個表達式的輸出結果為表達式後跟 `::` 及其型別，`::` 讀作"它的型別為"。凡是明確的型別，其首字母必為大寫。`'a'`，如它的樣子，是 `Char` 型別，易知是個字元 (character)。`True` 是 `Bool` 型別，也靠譜。不過這又是啥，檢測 `"hello"` 得一個 `[Char]` 這方括號表示一個 List，所以我們可以將其讀作"一組字元的 List"。而與 List 不同，每個 Tuple 都是獨立的型別，於是 `(True, 'a')` 的型別是 `(Bool, Char)`，而 `('a', 'b', 'c')` 的型別為 `(Char, Char, Char)`。`4==5` 一定回傳 `False`，所以它的型別為 `Bool`。

同樣，函數也有型別。編寫函數時，給它一個明確的型別聲明是個好習慣，比較短的函數就不用多此一舉了。還記得前面那個過濾大寫字母的 List Comprehension 嗎？給它加上型別聲明便是這個樣子：

```
removeNonUppercase :: [Char] -> [Char]
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

`removeNonUppercase` 的型別為 `[Char]->[Char]`，從它的參數和回傳值的型別上可以看出，它將一個字串映射為另一個字串。`[Char]` 與 `String` 是等價的，但使用 `String` 會更清晰：`removeNonUppercase :: String -> String`。編譯器會自動檢測出它的型別，我們還是標明了它的型別聲明。要是多個參數的函數該怎樣？如下便是一個將三個整數相加的簡單函數。

```
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

參數之間由 `->` 分隔，而與回傳值之間並無特殊差異。回傳值是最後一項，參數就是前三項。稍後，我們將講解為何只用 `->` 而不是 `Int, Int, Int->Int` 之類"更好看"的方式來分隔參數。

如果你打算給你編寫的函數加上個型別聲明卻拿不準它的型別是啥，只要先不寫型別聲明，把函數體寫出來，再使用 `:t` 命令測一下即可。函數也是表達式，所以 `:t` 對函數也是同樣可用的。

如下是幾個常見的型別：

`Int` 表示整數。`7` 可以是 `Int`，但 `7.2` 不可以。`Int` 是有界的，也就是說它由上限和下限。對 32 位的機器而言，上限一般是 `2147483647`，下限是 `-2147483648`。

**Integer** 表示...厄...也是整數，但它是無界的。這就意味着可以用它存放非常非常大的數，我是說非常大。它的效率不如 Int 高。

```
factorial :: Integer -> Integer
factorial n = product [1..n]
```

```
ghci> factorial 50
3041409320171337804361260816606476884437764156896051200000000000000
```

**Float** 表示單精度的浮點數。

```
circumference :: Float -> Float
circumference r = 2 * pi * r
```

```
ghci> circumference 4.0
25.132742
```

**Double** 表示雙精度的浮點數。

```
circumference' :: Double -> Double
circumference' r = 2 * pi * r
```

```
ghci> circumference' 4.0
25.132741228718345
```

**Bool** 表示布林值，它只有兩種值：`True` 和 `False`。

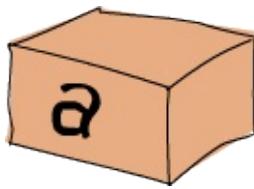
**Char** 表示一個字元。一個字元由單引號括起，一組字元的 List 即為字串。

**Tuple** 的型別取決於它的長度及其中項的型別。注意，空 Tuple 同樣也是個型別，它只有一種值：`()`。

## Type variables

你覺得 `head` 函數的型別是啥？它可以取任意型別的 List 的首項，是怎麼做到的呢？我們查一下！

```
ghci> :t head
head :: [a] -> a
```



嗯! `a` 是啥? 型別嗎? 想想我們在前面說過, 凡是型別其首字母必大寫, 所以它不會是個型別。它是個型別變數, 意味著 `a` 可以是任意的型別。這一點與其他語言中的泛型 (generic) 很相似, 但在 Haskell 中要更為強大。它可以讓我們輕而易舉地寫出型別無關的函數。使用到型別變數的函數被稱作"多態函數", `head` 函數的型別聲明裡標明了它可以取任意型別的 List 並回傳其中的第一個元素。

在命名上, 型別變數使用多個字元是合法的, 不過約定俗成, 通常都是使用單個字元, 如 `a`, `b`, `c`, `d` ...

還記得 `fst`? 我們查一下它的型別:

```
ghci> :t fst
fst :: (a, b) -> a
```

可以看到 `fst` 取一個包含兩個型別的 Tuple 作參數, 並以第一個項的型別作為回傳值。這便是 `fst` 可以處理一個含有兩種型別項的 pair 的原因。注意, `a` 和 `b` 是不同的型別變數, 但它們不一定非得是不同的型別, 它只是標明了首項的型別與回傳值的型別相同。

## Typeclasses 入門



型別定義行為的介面, 如果一個型別屬於某 Typeclass, 那它必實現了該 Typeclass 所描述的行為。很多從 OOP 走過來的人們往往會把 Typeclass 當成物件導向語言中的 `class` 而感到疑惑, 呃, 它們不是一回事。易於理解起見, 你可以把它看做是 Java 的 `interface`。

`==` 函數的型別聲明是怎樣的?

```
ghci> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

\*Note\*: 判斷相等的`==`運算子是函數，``^-*`/`之類的運算子也是同樣。在預設條件下，它們多為中綴函數。若要檢查

有意思。在這裡我們見到個新東西：`=>` 符號。它左邊的部分叫做型別約束。我們可以這樣閱讀這段型別聲明："相等函數取兩個相同型別的值作為參數並回傳一個布林值，而這兩個參數的型別同在 Eq 類之中(即型別約束)"

**Eq** 這一 Typeclass 提供了判斷相等性的介面，凡是可比較相等性的型別必屬於 Eq class。

```
ghci> 5 == 5
True
ghci> 5 /= 5
False
ghci> 'a' == 'a'
True
ghci> "Ho Ho" == "Ho Ho"
True
ghci> 3.432 == 3.432
True
```

`elem` 函數的型別為：`(Eq a)=>a->[a]->Bool`。這是它在檢測值是否存在於一個 List 時使用到了`==`的緣故。

幾個基本的 Typeclass：

**Eq** 包含可判斷相等性的型別。提供實現的函數是 `==` 和 `/=`。所以，只要一個函數有 Eq 類的型別限制，那麼它就必定在定義中用到了 `==` 和 `/=`。剛纔說了，除函數以外的所有型別都屬於 Eq，所以它們都可以判斷相等性。

**Ord** 包含可比較大小的型別。除了函數以外，我們目前所談到的所有型別都屬於 Ord 類。`Ord` 包中包含了 `<`, `>`, `<=`, `>=` 之類用於比較大小的函數。`compare` 函數取兩個 Ord 類中的相同型別的值作參數，回傳比較的結果。這個結果是如下三種型別之一：`GT`, `LT`, `EQ`。

```
ghci> :t (>)
(>) :: (Ord a) => a -> a -> Bool
```

型別若要成為Ord的成員，必先加入Eq家族。

```
ghci> "Abrakadabra" < "Zebra"
True
ghci> "Abrakadabra" `compare` "Zebra"
LT
ghci> 5 >= 2
True
ghci> 5 `compare` 3
GT
```

**Show** 的成員為可用字串表示的型別。目前為止，除函數以外的所有型別都是 `Show` 的成員。操作 `Show Typeclass`，最常用的函數表示 `show`。它可以取任一 `Show` 的成員型別並將其轉為字串。

```
ghci> show 3
"3"
ghci> show 5.334
"5.334"
ghci> show True
"True"
```

**Read** 是與 `show` 相反的 `Typeclass`。`read` 函數可以將一個字串轉為 `Read` 的某成員型別。

```
ghci> read "True" || False
True
ghci> read "8.2" + 3.8
12.0
ghci> read "5" - 2
3
ghci> read "[1,2,3,4]" ++ [3]
[1,2,3,4,3]
```

一切良好，如上的所有型別都屬於這一 `Typeclass`。嘗試 `read "4"` 又會怎樣？

```
ghci> read "4"
< interactive >:1:0:
    Ambiguous type variable `a' in the constraint:
      `Read a' arising from a use of `read' at <interactive>:1:0-7
    Probable fix: add a type signature that fixes these type variable(s)
```

`ghci` 跟我們說它搞不清楚我們想要的是什麼樣的回傳值。注意呼叫 `read` 後跟的那部分，`ghci` 通過它來辨認其型別。若要一個 `boolean` 值，他就知道必須得回傳一個 `Bool` 型別的值。但在這裡它只知道我們要的型別屬於 `Read Typeclass`，而不能明確到底是哪個。看一下 `read` 函數的型別聲明吧：

```
ghci> :t read
read :: (Read a) => String -> a
````haskell
```

看，它的回傳值屬於 `ReadTypeclass`，但我們若用不到這個值，它就永遠都不會得知該表達式的型別。所以我們需要在

```
````haskell
ghci> read "5" :: Int
5
ghci> read "5" :: Float
5.0
ghci> (read "5" :: Float) * 4
20.0
ghci> read "[1,2,3,4]" :: [Int]
[1,2,3,4]
ghci> read "(3, 'a')" :: (Int, Char)
(3, 'a')
```

編譯器可以辨認出大部分表達式的型別，但遇到 `read "5"` 的時候它就搞不清楚究竟該是 `Int` 還是 `Float` 了。只有經過運算，Haskell 才會明確其型別；同時由於 Haskell 是靜態的，它還必須得在 編譯前搞清楚所有值的型別。所以我們就最好提前給它打聲招呼："嘿，這個表達式應該是這個型別，省的你認不出來！"

**Enum** 的成員都是連續的型別 -- 也就是可枚舉。`Enum` 類存在的主要好處就在於我們可以在 `Range` 中用到它的成員型別：每個值都有後繼子 (`successer`) 和前置子 (`predecessor`)，分別可以通過 `succ` 函數和 `pred` 函數得到。該 `Typeclass` 包含的型別有：`()`，`Bool`，`Char`，`Ordering`，`Int`，`Integer`，`Float` 和 `Double`。

```
ghci> ['a'..'e']
"abcde"
ghci> [LT .. GT]
[LT,EQ,GT]
ghci> [3 .. 5]
[3,4,5]
ghci> succ 'B'
'C'
```

**Bounded** 的成員都有一個上限和下限。

```
ghci> minBound :: Int
-2147483648
ghci> maxBound :: Char
'\1114111'
ghci> maxBound :: Bool
True
ghci> minBound :: Bool
False
```

`minBound` 和 `maxBound` 函數很有趣，它們的型別都是 `(Bounded a) => a`。可以說，它們都是多態常量。

如果其中的項都屬於 `Bounded Typeclass`，那麼該 `Tuple` 也屬於 `Bounded`

```
ghci> maxBound :: (Bool, Int, Char)
(True, 2147483647, '\1114111')
```

`Num` 是表示數字的 `Typeclass`，它的成員型別都具有數字的特徵。檢查一個數字的型別：

```
ghci> :t 20
20 :: (Num t) => t
```

看樣子所有的數字都是多態常量，它可以作為所有 `Num Typeclass` 中的成員型別。以上便是 `Num Typeclass` 中包含的所有型別，檢測 `*` 運算子的型別，可以發現它可以處理一切的數字：

```
ghci> :t (*)
(*) :: (Num a) => a -> a -> a
```

它只取兩個相同型別的參數。所以 `(5 :: Int) * (6 :: Integer)` 會引發一個型別錯誤，而 `5 * (6 :: Integer)` 就不會有問題。

型別只有親近 `Show` 和 `Eq`，才可以加入 `Num`。

`Integral` 同樣是表示數字的 `Typeclass`。`Num` 包含所有的數字：實數和整數。而 `Integral` 僅包含整數，其中的成員型別有 `Int` 和 `Integer`。

`Floating` 僅包含浮點型別：`Float` 和 `Double`。

有個函數在處理數字時會非常有用，它便是 `fromIntegral`。其型別聲明為：`fromIntegral :: (Num b, Integral a) => a -> b`。從中可以看出，它取一個整數做參數並回傳一個更加通用的數字，這在同時處理整數和浮點時會尤為有用。舉例來說，`length` 函數的型別聲明為：`length :: [a] -> Int`，而非更通用的形式，如 `length :: (Num b) => [a] -> b`。這應

該是歷史原因吧，反正我覺得挺蠢。如果取了一個 List 長度的值再給它加 3.2 就會報錯，因為這是將浮點數和整數相加。面對這種情況，我們就用 `fromIntegral (length [1,2,3,4]) + 3.2` 來解決。

注意到，`fromIntegral` 的型別聲明中用到了多個型別約束。如你所見，只要將多個型別約束放到括號裡用逗號隔開即可。

# 函數的語法

## 模式匹配 (Pattern matching)



本章講的就是 Haskell 那套獨特的語法結構，先從模式匹配開始。模式匹配通過檢查數據的特定結構來檢查其是否匹配，並按模式從中取得數據。

在定義函數時，你可以為不同的模式分別定義函數本身，這就讓程式碼更加簡潔易讀。你可以匹配一切數據型別 --- 數字，字元，List，元組，等等。我們弄個簡單函數，讓它檢查我們傳給它的數字是不是 7。

```
lucky :: (Integral a) => a -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"
```

在呼叫 `lucky` 時，模式會從上至下進行檢查，一旦有匹配，那對應的函數體就被應用。這個模式中的唯一匹配是參數為 7，如果不是 7，就轉到下一個模式，它匹配一切數值並將其綁定為 `x`。這個函數完全可以使用 `if` 實現，不過我們若要個分辨 1 到 5 中的數字，而無視其它數的函數該怎麼辦？要是沒有模式匹配的話，那可得好大一棵 `if-else` 樹了！

```
sayMe :: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

注意下，如果我們把最後匹配一切的那個模式挪到最前，它的結果就全都是 "Not between 1 and 5" 了。因為它自己匹配了一切數字，不給後面的模式留機會。

記得前面實現的那個階乘函數麼？當時是把 `n` 的階乘定義成了 `product [1..n]`。也可以寫出像數學那樣的遞迴實現，先說明 0 的階乘是 1，再說明每個正整數的階乘都是這個數與它前驅 (predecessor) 對應的階乘的積。如下便是翻譯到 Haskell 的樣子：

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

這就是我們定義的第一個遞迴函數。遞迴在 Haskell 中十分重要，我們會在後面深入理解。如果拿一個數(如 3)呼叫 `factorial` 函數，這就是接下來的計算步驟：先計算 `3*factorial 2`，`factorial 2` 等於 `2*factorial 1`，也就是 `3*(2*(factorial 1))`。`factorial 1` 等於 `1*factorial 0`，好，得 `3*(2*(1*factorial 0))`，遞迴在這裡到頭了，嗯 --- 我們在萬能匹配前面有定義，0 的階乘是 1。於是最終的結果等於 `3*(2*(1*1))`。若是把第二個模式放在前面，它就會捕獲包括 0 在內的一切數字，這一來我們的計算就永遠都不會停止了。這便是為什麼說模式的順序是如此重要：它總是優先匹配最符合的那個，最後才是那個萬能的。

模式匹配也會失敗。假如這個函數：

```
charName :: Char -> String
charName 'a' = "Albert"
charName 'b' = "Broseph"
charName 'c' = "Cecil"
```

拿個它沒有考慮到的字元去呼叫它，你就會看到這個：

```
ghci> charName 'a'
"Albert"
ghci> charName 'b'
"Broseph"
ghci> charName 'h'
*** Exception: tut.hs:(53,0)-(55,21): Non-exhaustive patterns in function charName
```

它告訴我們說，這個模式不夠全面。因此，在定義模式時，一定要留一個萬能匹配的模式，這樣我們的程序就不會為了不可預料的輸入而崩潰了。

對 Tuple 同樣可以使用模式匹配。寫個函數，將二維空間中的向量相加該如何？將它們的 `x` 項和 `y` 項分別相加就是了。如果不瞭解模式匹配，我們很可能會寫出這樣的程式碼：

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors a b = (fst a + fst b, snd a + snd b)
```

嗯，可以運行。但有更好的方法，上模式匹配：

```
addVectors :: (Num a) => (a, a) -> (a, a)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

there we go！好多了！注意，它已經是個萬能的匹配了。兩個 `addVector` 的型別都是 `addVectors:: (Num a) => (a,a) -> (a,a)`，我們就能夠保證，兩個參數都是序對 (Pair) 了。

`fst` 和 `snd` 可以從序對中取出元素。三元組 (Triple) 呢？嗯，沒現成的函數，得自己動手：

```
first :: (a, b, c) -> a
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

third :: (a, b, c) -> c
third (_, _, z) = z
```

這裡的 `_` 就和 List Comprehension 中一樣。表示我們不關心這部分的具體內容。

說到 List Comprehension，我想起來在 List Comprehension 中也能用模式匹配：

```
ghci> let xs = [(1,3), (4,3), (2,4), (5,3), (5,6), (3,1)]
ghci> [a+b | (a,b) <- xs]
[4,7,6,8,11,4]
```

一旦模式匹配失敗，它就簡單挪到下個元素。

對 List 本身也可以使用模式匹配。你可以用 `[]` 或 `:`` 來匹配它。因為 `[1,2,3]` 本質就是 `1:2:3:[]` 的語法糖。你也可以使用前一種形式，像 `x:xs` 這樣的模式可以將 List 的頭部綁定為 `x`，尾部綁定為 `xs`。如果這 List 只有一個元素，那麼 `xs` 就是一個空 List。

\*Note\* : ```x:xs``` 這模式的應用非常廣泛，尤其是遞迴函數。不過它只能匹配長度大於等於 1 的 List。

如果你要把 List 的前三個元素都綁定到變數中，可以使用類似 `x:y:z:xs` 這樣的形式。它只能匹配長度大於等於 3 的 List。

我們已經知道了對 List 做模式匹配的方法，就實現個我們自己的 `head` 函數。

```
head' :: [a] -> a
head' [] = error "Can't call head on an empty list, dummy!"
head' (x:_)= x
```

看看管不管用：

```
ghci> head' [4,5,6]
4
ghci> head' "Hello"
'H'
```

漂亮！注意下，你若要綁定多個變數(用 `_` 也是如此)，我們必須用括號將其括起。同時注意下我們用的這個 `error` 函數，它可以生成一個運行時錯誤，用參數中的字串表示對錯誤的描述。它會直接導致程序崩潰，因此應謹慎使用。可是對一個空 List 取 `head` 真的不靠譜哇。

弄個簡單函數，讓它用非標準的英語給我們展示 List 的前幾項。

```
tell :: (Show a) => [a] -> String
tell [] = "The list is empty"
tell (x:[]) = "The list has one element: " ++ show x
tell (x:y:[]) = "The list has two elements: " ++ show x ++ " and " ++ show y
tell (x:y:_)= "This list is long. The first two elements are: " ++ show x ++ " and " ++
```

這個函數顧及了空 List，單元素 List，雙元素 List 以及較長的 List，所以這個函數很安全。`(x:[])` 與 `(x:y:[])` 也可以寫作 `[x]` 和 `[x,y]` (有了語法糖，我們不必多加括號)。不過 `(x:y:_)` 這樣的模式就不行了，因為它匹配的 List 長度不固定。

我們會用 List Comprehension 實現過自己的 `length` 函數，現在用模式匹配和遞迴重新實現它：

```
length' :: (Num b) => [a] -> b
length' [] = 0
length' (_:xs) = 1 + length' xs
```

這與先前寫的那個 `factorial` 函數很相似。先定義好未知輸入的結果 --- 空 List，這也叫作邊界條件。再在第二個模式中將這 List 分割為頭部和尾部。說，List 的長度就是其尾部的長度加 1。匹配頭部用的 `_`，因為我們並不關心它的值。同時也應明確，我們顧及了 List 所有可能的模式：第一個模式匹配空 List，第二個匹配任意的非空 List。

看下拿 `"ham"` 呼叫 `length'` 會怎樣。首先它會檢查它是否為空 List。顯然不是，於是進入下一模式。它匹配了第二個模式，把它分割為頭部和尾部並無視掉頭部的值，得長度就是 `1+length' "am"`。ok。以此類推，`"am"` 的 `length` 就是 `1+length' "m"`。好，現在我們有

了 `1+(1+length' "m")`。`length' "m"` 即 `1+length ""` (也就是 `1+length' []`)。根據定義，`length' []` 等於 `0`。最後得 `1+(1+(1+0))`。

再實現 `sum`。我們知道空 List 的和是 0，就把它定義為一個模式。我們也知道一個 List 的和就是頭部加上尾部的和的和。寫下來就成了：

```
sum' :: (Num a) => [a] -> a
sum' [] = 0
sum' (x:xs) = x + sum' xs
```

還有個東西叫做 `as` 模式，就是將一個名字和 `@` 置於模式前，可以在按模式分割什麼東西時仍保留對其整體的引用。如這個模式 `xs@(x:y:ys)`，它會匹配出與 `x:y:ys` 對應的東西，同時你也可以方便地通過 `xs` 得到整個 List，而不必在函數體中重複 `x:y:ys`。看下這個 quick and dirty 的例子：

```
capital :: String -> String
capital "" = "Empty string, whoops!"
capital all@(x:xs) = "The first letter of " ++ all ++ " is " ++ [x]
```

```
ghci> capital "Dracula"
"The first letter of Dracula is D"
```

我們使用 `as` 模式通常就是為了在較大的模式中保留對整體的引用，從而減少重複性的工作。

還有——你不可以在模式匹配中使用 `++`。若有個模式是 `(xs++ys)`，那麼這個 List 該從什麼地方分開呢？不靠譜吧。而 `(xs++[x,y,z])` 或只一個 `(xs++[x])` 或許還能說的過去，不過出於 List 的本質，這樣寫也是不可以的。

## 什麼是 Guards

模式用來檢查一個值是否合適並從中取值，而 guard 則用來檢查一個值的某項屬性是否為真。乍一聽有點像是 `if` 語句，實際上也正是如此。不過處理多個條件分支時 guard 的可讀性要高些，並且與模式匹配契合的很好。



在講解它的語法前，我們先看一個用到 guard 的函數。它會依據你的 BMI 值 (body mass index, 身體質量指數)來不同程度地侮辱你。BMI 值即為體重除以身高的平方。如果小於 18.5，就是太瘦；如果在 18.5 到 25 之間，就是正常；25 到 30 之間，超重；如果超過 30，肥胖。這就是那個函數(我們目前暫不為您計算 BMI，它只是直接取一個 BMI 值)。

```
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
| bmi <= 18.5 = "You're underweight, you emo, you!"
| bmi <= 25.0 = "You're supposedly normal. Pfffft, I bet you're ugly!"
| bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
| otherwise     = "You're a whale, congratulations!"
```

guard 由跟在函數名及參數後面的豎線標誌，通常他們都是靠右一個縮進排成一列。一個 guard 就是一個布爾表達式，如果為真，就使用其對應的函數體。如果為假，就送去見下一個 guard，如之繼續。如果我們用 24.3 呼叫這個函數，它就會先檢查它是否小於等於 18.5，顯然不是，於是見下一個 guard。24.3 小於 25.0，因此通過了第二個 guard 的檢查，就返回第二個字串。

在這裡則是相當的簡潔，不過不難想象這在命令式語言中又會是怎樣的一棵 if-else 樹。由於 if-else 的大樹比較雜亂，若是出現問題會很難發現，guard 對此則十分清楚。

最後的那個 guard 往往都是 `otherwise`，它的定義就是簡單一個 `otherwise = True`，捕獲一切。這與模式很相像，只是模式檢查的是匹配，而它們檢查的是布爾表達式。如果一個函數的所有 guard 都沒有通過(而且沒有提供 `otherwise` 作萬能匹配)，就轉入下一模式。這便是 guard 與模式契合的地方。如果始終沒有找到合適的 guard 或模式，就會發生一個錯誤。

當然，guard 可以在含有任意數量參數的函數中使用。省得用戶在使用這函數之前每次都自己計算 `bmi`。我們修改下這個函數，讓它取身高體重為我們計算。

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
| weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
| weight / height ^ 2 <= 25.0 = "You're supposedly normal. Pfffft, I bet you're ugly!"
| weight / height ^ 2 <= 30.0 = "You're fat! Lose some weight, fatty!"
| otherwise                   = "You're a whale, congratulations!"
```

你可以測試自己胖不胖。

```
ghci> bmiTell 85 1.90
"You're supposedly normal. Pfffft, I bet you're ugly!"
```

運行的結果是我不太胖。不過程式卻說我很醜。

要注意一點，函數的名字和參數的後面並沒有 `=`。許多初學者會造成語法錯誤，就是因為在後面加上了 `=`。

另一個簡單的例子：寫個自己的 `max` 函數。應該還記得，它是取兩個可比較的值，返回較大的那個。

```
max' :: (Ord a) => a -> a -> a
max' a b
| a > b      = a
| otherwise   = b
```

`guard` 也可以塞在一行裡面。但這樣會喪失可讀性，因此是不被鼓勵的。即使是較短的函數也是如此，不過出於展示，我們可以這樣重寫 `max'`：

```
max' :: (Ord a) => a -> a -> a
max' a b | a > b = a | otherwise = b
```

這樣的寫法根本一點都不容易讀。

我們再來試試用 `guard` 實現我們自己的 `compare` 函數：

```
myCompare :: (Ord a) => a -> a -> Ordering
a `myCompare` b
| a > b      = GT
| a == b     = EQ
| otherwise   = LT
```

```
ghci> 3 `myCompare` 2
GT
```

\*Note\*：通過反單引號，我們不僅可以以中綴形式呼叫函數，也可以在定義函數的時候使用它。有時這樣會更易讀。

## 關鍵字 Where

前一節中我們寫了這個 `bmi` 計算函數：

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
| weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
| weight / height ^ 2 <= 25.0 = "You're supposedly normal. Pfffft, I bet you're ugly!"
| weight / height ^ 2 <= 30.0 = "You're fat! Lose some weight, fatty!"
| otherwise                  = "You're a whale, congratulations!"
```

注意，我們重複了 3 次。我們重複了 3 次。程式設計師的字典裡不應該有"重複"這個詞。既然發現有重複，那麼給它一個名字來代替這三個表達式會更好些。嗯，我們可以這樣修改：

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
| bmi <= 18.5 = "You're underweight, you emo, you!"
| bmi <= 25.0 = "You're supposedly normal. Pfffft, I bet you're ugly!"
| bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
| otherwise    = "You're a whale, congratulations!"
where bmi = weight / height ^ 2
```

我們的 `where` 關鍵字跟在 `guard` 後面(最好是與豎線縮進一致)，可以定義多個名字和函數。這些名字對每個 `guard` 都是可見的，這一來就避免了重複。如果我們打算換種方式計算 `bmi`，只需進行一次修改就行了。通過命名，我們提升了程式碼的可讀性，並且由於 `bmi` 只計算了一次，函數的執行效率也有所提升。我們可以再做下修改：

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
| bmi <= skinny = "You're underweight, you emo, you!"
| bmi <= normal = "You're supposedly normal. Pfffft, I bet you're ugly!"
| bmi <= fat     = "You're fat! Lose some weight, fatty!"
| otherwise      = "You're a whale, congratulations!"
where bmi = weight / height ^ 2
      skinny = 18.5
      normal = 25.0
      fat = 30.0
```

函數在 `where` 綁定中定義的名字只對本函數可見，因此我們不必擔心它會污染其他函數的命名空間。注意，其中的名字都是一列垂直排開，如果不這樣規範，Haskell 就搞不清楚它們在哪個地方了。

`where` 綁定不會在多個模式中共享。如果你在一個函數的多個模式中重複用到同一名字，就應該把它置於全局定義之中。

`where` 綁定也可以使用模式匹配！前面那段程式碼可以改成：

```

...
where bmi = weight / height ^ 2
      (skinny, normal, fat) = (18.5, 25.0, 30.0)

```

我們再搞個簡單函數，讓它告訴我們姓名的首字母：

```

initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
  where (f:_)=firstname
        (l:_)=lastname

```

我們完全按可以在函數的參數上直接使用模式匹配(這樣更短更簡潔)，在這裡只是為了演示在 `where` 語句中同樣可以使用模式匹配：

`where` 綁定可以定義名字，也可以定義函數。保持健康的程式語言風格，我們搞個計算一組 `bmi` 的函數：

```

calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi w h | (w, h) <- xs]
  where bmi weight height = weight / height ^ 2

```

這就全了！在這裡將 `bmi` 搞成一個函數，是因為我們不能依據參數直接進行計算，而必須先從傳入函數的 List 中取出每個序對並計算對應的值。

`where` 綁定還可以一層套一層地來使用。有個常見的寫法是，在定義一個函數的時候也寫幾個輔助函數擺在 `where` 綁定中。而每個輔助函數也可以透過 `where` 擁有各自的輔助函數。

## 關鍵字 Let

`let` 綁定與 `where` 綁定很相似。`where` 綁定是在函數底部定義名字，對包括所有 guard 在內的整個函數可見。`let` 綁定則是個表達式，允許你在任何位置定義局部變數，而對不同的 guard 不可見。正如 Haskell 中所有賦值結構一樣，`let` 綁定也可以使用模式匹配。看下它的實際應用！這是個依據半徑和高度求圓柱體表面積的函數：

```

cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
  let sideArea = 2 * pi * r * h
      topArea = pi * r ^2
  in sideArea + 2 * topArea

```



`let` 的格式為 `let [bindings] in [expressions]`。在 `let` 中綁定的名字僅對 `in` 部分可見。`let` 裡面定義的名字也得對齊到一列。不難看出，這用 `where` 綁定也可以做到。那麼它倆有什麼區別呢？看起來無非就是，`let` 把綁定放在語句前面而 `where` 放在後面嘛。

不同之處在於，`let` 綁定本身是個表達式，而 `where` 綁定則是個語法結構。還記得前面我們講 if 語句時提到它是個表達式，因而可以隨處安放？

```
ghci> [if 5 > 3 then "Woo" else "Boo", if 'a' > 'b' then "Foo" else "Bar"]
["Woo", "Bar"]
ghci> 4 * (if 10 > 5 then 10 else 0) + 2
42
```

用 `let` 綁定也可以實現：

```
ghci> 4 * (let a = 9 in a + 1) + 2
42
```

`let` 也可以定義局部函數：

```
ghci> [let square x = x * x in (square 5, square 3, square 2)]
[(25,9,4)]
```

若要在一行中綁定多個名字，再將它們排成一列顯然是不可以的。不過可以用分號將其分開。

```
ghci> (let a = 100; b = 200; c = 300 in a*b*c, let foo="Hey "; bar = "there!" in foo ++ b
(6000000,"Hey there!"))
```

最後那個綁定後面的分號不是必須的，不過加上也沒關係。如我們前面所說，你可以在 `let` 綁定中使用模式匹配。這在從 Tuple 取值之類的操作中很方便。

```
ghci> (let (a,b,c) = (1,2,3) in a+b+c) * 100
600
```

你也可以把 `let` 綁定放到 List Comprehension 中。我們重寫下那個計算 `bmi` 值的函數，用個 `let` 替換掉原先的 `where`。

```
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2]
```

List Comprehension 中 `let` 綁定的樣子和限制條件差不多，只不過它做的不是過濾，而是綁定名字。`let` 中綁定的名字在輸出函數及限制條件中都可見。這一來我們就可以讓我們的函數隻返回胖子的 `bmi` 值：

```
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2, bmi >= 25.0]
```

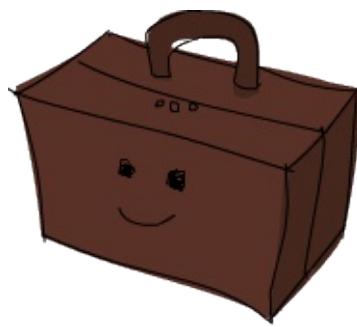
在 `(w, h) <- xs` 這裡無法使用 `bmi` 這名字，因為它在 `let` 綁定的前面。

在 List Comprehension 中我們忽略了 `let` 綁定的 `in` 部分，因為名字的可見性已經預先定義好了。不過，把一個 `let...in` 放到限制條件中也是可以的，這樣名字只對這個限制條件可見。在 ghci 中 `in` 部分也可以省略，名字的定義就在整個交互中可見。

```
ghci> let zoot x y z = x * y + z
ghci> zoot 3 9 2
29
ghci> let boot x y z = x * y + z in boot 3 4 2
14
ghci> boot
< interactive>:1:0: Not in scope: `boot'
```

你說既然 `let` 已經這麼好了，還要 `where` 幹嘛呢？嗯，`let` 是個表達式，定義域限制的相當小，因此不能在多個 guard 中使用。一些朋友更喜歡 `where`，因為它是跟在函數體後面，把主函數體距離型別聲明近一些會更易讀。

## Case expressions



有命令式程式語言 (C, C++, Java, etc.) 的經驗的同學一定會有所瞭解，很多命令式語言都提供了 `case` 語句。就是取一個變數，按照對變數的判斷選擇對應的程式碼塊。其中可能會存在一個萬能匹配以處理未預料的情況。

Haskell 取了這一概念融合其中。如其名，`case` 表達式就是，嗯，一種表達式。跟 `if..else` 和 `let` 一樣的表達式。用它可以對變數的不同情況分別求值，還可以使用模式匹配。Hmm，取一個變數，對它模式匹配，執行對應的程式碼塊。好像在哪兒聽過？啊，就是函數定義時參數的模式匹配！好吧，模式匹配本質上不過就是 `case` 語句的語法糖而已。這兩段程式碼就是完全等價的：

```
head' :: [a] -> a
head' [] = error "No head for empty lists!"
head' (x:_)= x
```

```
head' :: [a] -> a
head' xs = case xs of [] -> error "No head for empty lists!"
                  (x:_)-> x
```

看得出，`case` 表達式的語法十分簡單：

```
case expression of pattern -> result
                  pattern -> result
                  pattern -> result
                  ...
```

`expression` 匹配合適的模式。一如預期地，第一個模式若匹配，就執行第一個區塊的程式碼；否則就接下去比對下一個模式。如果到最後依然沒有匹配的模式，就會產生運行時錯誤。

函數參數的模式匹配只能在定義函數時使用，而 `case` 表達式可以用在任何地方。例如：

```
describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of [] -> "empty."
                           [x] -> "a singleton list."
                           xs -> "a longer list."
```

這在表達式中作模式匹配很方便，由於模式匹配本質上就是 `case` 表達式的語法糖，那麼寫成這樣也是等價的：

```
describeList :: [a] -> String
describeList xs = "The list is " ++ what xs
  where what [] = "empty."
        what [x] = "a singleton list."
        what xs = "a longer list."
```

# 遞迴

你好， 遞迴！



前面的章節中我們簡要談了一下遞迴。而在本章，我們會深入地瞭解到它為何在 Haskell 中是如此重要，能夠以遞迴思想寫出簡潔優雅的程式碼。

如果你還不知道什麼是遞迴，就讀這個句子。哈哈！開個玩笑而已！遞迴實際上是定義函數以呼叫自身的方式。在數學定義中，遞迴隨處可見，如斐波那契數列（fibonacci）。它先是定義兩個非遞迴的數： $F(0)=0, F(1)=1$ ，表示斐波那契數列的前兩個數為 0 和 1。然後就是對其他自然數，其斐波那契數就是它前面兩個數字的和，即  $F(N)=F(N-1)+F(N-2)$ 。這樣一來， $F(3)$  就是  $F(2)+F(1)$ ，進一步便是  $(F(1)+F(0))+F(1)$ 。已經下探到了前面定義的非遞迴斐波那契數，可以放心地說  $F(3)$  就是 2 了。在遞迴定義中聲明的一兩個非遞迴的值（如  $F(0)$  和  $F(1)$ ）也可以稱作邊界條件，這對遞迴函數的正確求值至關重要。要是前面沒有定義  $F(0)$  和  $F(1)$  的話，它下探到 0 之後就會進一步到負數，你就永遠都得不到結果了。一不留神它就算到了  $F(-2000)=F(-2001)+F(-2002)$ ，並且永遠都算不到頭！

遞迴在 Haskell 中非常重要。命令式語言要求你提供求解的步驟，Haskell 則傾向于讓你提供問題的描述。這便是 Haskell 沒有 `while` 或 `for` 循環的原因，遞迴是我們的替代方案。

## 實作 Maximum

`maximum` 函數取一組可排序的 List（屬於 `Ord Typeclass`）做參數，並回傳其中的最大值。想想，在命令式風格中這一函數該怎麼實現。很可能你會設一個變數來存儲當前的最大值，然後用循環遍歷該 List，若存在比這個值更大的元素，則修改變數為這一元素的值。到最後，變數的值就是運算結果。唔！描述如此簡單的算法還頗費了點口舌呢！

現在看看遞迴的思路是如何：我們先定下一個邊界條件，即處理單個元素的 List 時，回傳該元素。如果該 List 的頭部大於尾部的最大值，我們就可以假定較長的 List 的最大值就是它的頭部。而尾部若存在比它更大的元素，它就是尾部的最大值。就這麼簡單！現在，我們在

## Haskell 中實現它

```
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs)
| x > maxTail = x
| otherwise = maxTail
  where maxTail = maximum' xs
```

如你所見，模式匹配與遞迴簡直就是天造地設！大多數命令式語言中都沒有模式匹配，於是你就得造一堆 if-else 來測試邊界條件。而在這裡，我們僅需要使用模式將其表示出來。第一個模式說，如果該 List 為空，崩潰！就該這樣，一個空 List 的最大值能是啥？我不知道。第二個模式也表示一個邊緣條件，它說，如果這個 List 僅包含單個元素，就回傳該元素的值。

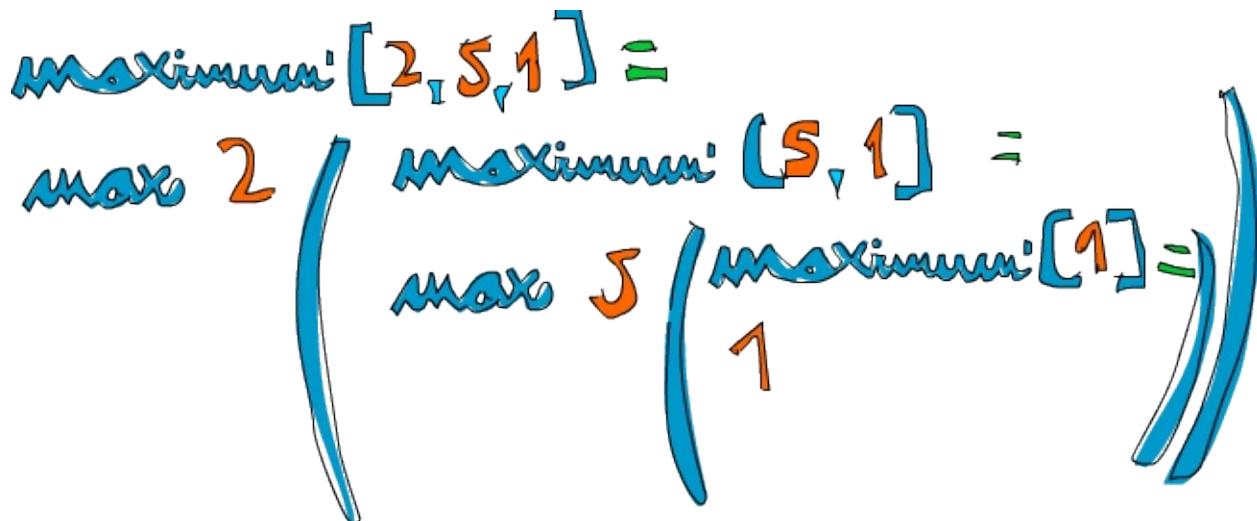
現在是第三個模式，執行動作的地方。通過模式匹配，可以取得一個 List 的頭部和尾部。這在使用遞迴處理 List 時是十分常見的。出於習慣，我們用個 where 語句來表示 maxTail 作為該 List 中尾部的最大值，然後檢查頭部是否大於尾部的最大值。若是，回傳頭部；若非，回傳尾部的最大值。

我們取個 List `[2, 5, 1]` 做例子來看看它的工作原理。當呼叫 `maximum'` 處理它時，前兩個模式不會被匹配，而第三個模式匹配了它並將其分為 `2` 與 `[5, 1]`。where 子句再取 `[5, 1]` 的最大值。於是再次與第三個模式匹配，並將 `[5, 1]` 分割為 `5` 和 `[1]`。繼續，where 子句取 `[1]` 的最大值，這時終於到了邊緣條件！回傳 `1`。進一步，將 `5` 與 `[1]` 中的最大值做比較，易得 `5`，現在我們就得到了 `[5, 1]` 的最大值。再進一步，將 `2` 與 `[5, 1]` 中的最大值相比較，可得 `5` 更大，最終得 `5`。

改用 `max` 函數會使程式碼更加清晰。如果你還記得，`max` 函數取兩個值做參數並回傳其中較大的值。如下便是用 `max` 函數重寫的 `maximum'`

```
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs) = max x (maximum' xs)
```

太漂亮了！一個 List 的最大值就是它的首個元素與它尾部中最大值相比較所得的結果，簡明扼要。



## 來看幾個遞迴函數

現在我們已經瞭解了遞迴的思路，接下來就使用遞迴來實現幾個函數。先實現下 `replicate` 函數，它取一個 `Int` 值和一個元素做參數，回傳一個包含多個重複元素的 List，如 `replicate 3 5` 回傳 `[5, 5, 5]`。考慮一下，我覺得它的邊界條件應該是負數。如果要 `replicate` 重複某元素零次，那就是空 List。負數也是同樣，不靠譜。

```
replicate' :: (Num i, Ord i) => i -> a -> [a]
replicate' n x
| n <= 0      = []
| otherwise = x:replicate' (n-1) x
```

在這裡我們使用了 guard 而非模式匹配，是因為這裡做的是布林判斷。如果 `n` 小於 0 就回傳一個空 List，否則，回傳以 `x` 作首個元素並後接重複 `n-1` 次 `x` 的 List。最後，`(n-1)` 的那部分就會令函數抵達邊緣條件。

\*Note\*: Num 不是 Ord 的子集，表示數字不一定得拘泥于排序，這就是在做加減法比較時要將 Num 與 Ord 型別統一

接下來實現 `take` 函數，它可以從一個 List 取出一定數量的元素。如 `take 3 [5, 4, 3, 2, 1]`，得到 `[5, 4, 3]`。若要取零或負數個的話就會得到一個空 List。同樣，若是從一個空 List 中取值，它會得到一個空 List。注意，這兒有兩個邊界條件，寫出來：

```
take' :: (Num i, Ord i) => i -> [a] -> [a]
take' n _
| n <= 0      = []
take' _ []      = []
take' n (x:xs) = x : take' (n-1) xs
```



首個模式辨認若為 0 或負數, 回傳空 List. 同時注意這裡用了一個 guard 却沒有指定 otherwise 部分, 這就表示 `n` 若大於 0, 會轉入下一模式. 第二個模式指明了若試圖從一個空 List 中取值, 則回傳空 List. 第三個模式將 List 分割為頭部和尾部, 然後表明從一個 List 中取多個元素等同於令 `x` 作頭部後接從尾部取 `n-1` 個元素所得的 List. 假如我們要從 `[4, 3, 2, 1]` 中取 3 個元素, 試着從紙上寫出它的推導過程

`reverse` 函數簡單地反轉一個 List, 動腦筋想一下它的邊界條件! 該怎樣呢? 想想...是空 List! 空 List 的反轉結果還是它自己. Okay, 接下來該怎麼辦? 好的, 你猜的出來. 若將一個 List 分割為頭部與尾部, 那它反轉的結果就是反轉後的尾部與頭部相連所得的 List.

```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]
```

繼續下去!

Haskell 支持無限 List, 所以我們的遞迴就不必添加邊界條件。這樣一來, 它可以對某值計算個沒完, 也可以產生一個無限的資料結構, 如無限 List。而無限 List 的好處就在於我們可以在任意位置將它斷開。

`repeat` 函數取一個元素作參數, 回傳一個僅包含該元素的無限 List. 它的遞迴實現簡單的很, 看:

```
repeat' :: a -> [a]
repeat' x = x:repeat' x
```

呼叫 `repeat 3` 會得到一個以 3 為頭部並無限數量的 3 為尾部的 List, 可以說 `repeat 3` 運行起來就是 `3:repeat 3`, 然後 `3:3:3:3` 等等. 若執行 `repeat 3`, 那它的運算永遠都不會停止。而 `take 5 (repeat 3)` 就可以得到 5 個 3, 與 `replicate 5 3` 差不多。

`zip` 取兩個 List 作參數並將其捆在一起。`zip [1,2,3] [2,3]` 回傳 `[(1,2),(2,3)]`，它會把較長的 List 從中間斷開，以匹配較短的 List. 用 `zip` 處理一個 List 與空 List 又會怎樣？嗯，會得一個空 List, 這便是我們的限制條件，由於 `zip` 取兩個參數，所以要有兩個邊緣條件

```
zip' :: [a] -> [b] -> [(a,b)]
zip' _ [] = []
zip' [] _ = []
zip' (x:xs) (y:ys) = (x,y):zip' xs ys
```

前兩個模式表示兩個 List 中若存在空 List, 則回傳空 List. 第三個模式表示將兩個 List 捆綁的行為，即將其頭部配對並後跟捆綁的尾部. 用 `zip` 處理 `[1,2,3]` 與 `['a','b']` 的話，就會在 `[3]` 與 `[]` 時觸及邊界條件，得到 `(1,'a'): (2,'b'): []` 的結果，與 `[(1,'a'),(2,'b')]` 等價.

再實現一個標準庫函數 -- `elem`！它取一個元素與一個 List 作參數，並檢測該元素是否包含于此 List. 而邊緣條件就與大多數情況相同，空 List. 大家都知道空 List 中不包含任何元素，便不必再做任何判斷

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' a [] = False
elem' a (x:xs)
| a == x    = True
| otherwise = a `elem'` xs
```

這很簡單明瞭。若頭部不是該元素，就檢測尾部，若為空 List 就回傳 `False`.

## "快速"排序



假定我們有一個可排序的 List, 其中元素的型別為 `Ord Typeclass` 的成員. 現在我們要給它排序! 有個排序算法非常的酷，就是快速排序 (quick sort)，睿智的排序方法. 儘管它在命令式語言中也不過 10 行，但在 Haskell 下邊要更短，更漂亮，儼然已經成了 Haskell 的招牌了. 嗯，我們在這裡也實現一下. 或許會顯得很俗氣，因為每個人都用它來展示 Haskell 究竟有多優雅!

它的型別聲明應為 `quicksort :: (Ord a) => [a] -> [a]`，沒啥奇怪的。邊界條件呢？如料，空 List。排過序的空 List 還是空 List。接下來便是算法的定義：排過序的 *List* 就是令所有小於等於頭部的元素在先(它們已經排過了序)，後跟大於頭部的元素(它們同樣已經拍過了序)。注意定義中有兩次排序，所以就得遞迴兩次！同時也需要注意算法定義的動詞為"是"什麼而非"做"這個，"做"那個，再"做"那個...這便是函數式編程之美！如何才能從 List 中取得比頭部小的那些元素呢？List Comprehension。好，動手寫出這個函數！

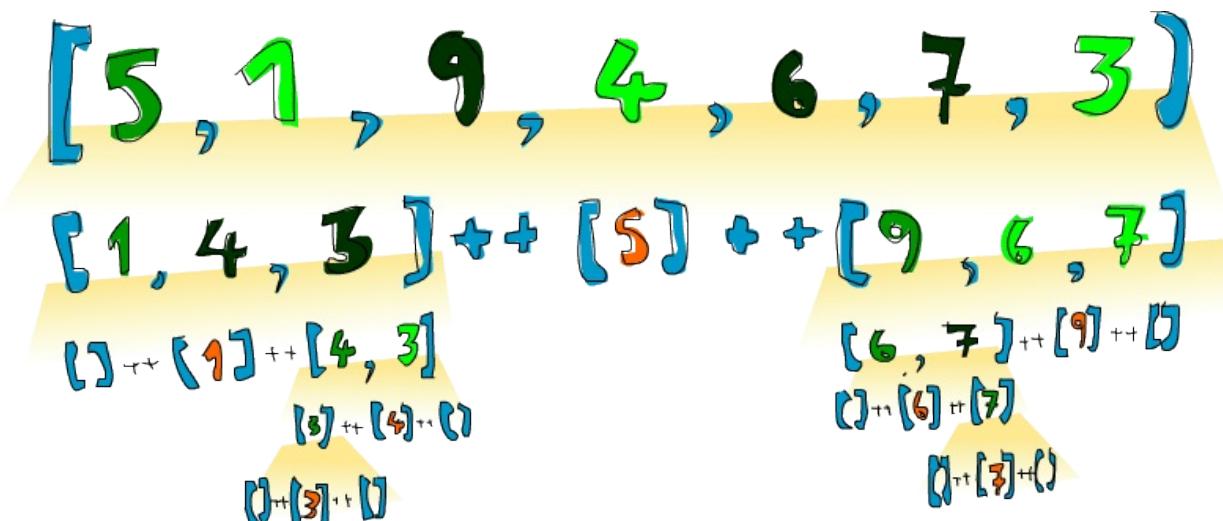
```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
  let smallerSorted = quicksort [a | a <- xs, a <= x]
      biggerSorted = quicksort [a | a <- xs, a > x]
  in smallerSorted ++ [x] ++ biggerSorted
```

小小的測試一下，看看結果是否正確~

```
ghci> quicksort [10,2,5,3,1,6,7,4,2,3,4,8,9]
[1,2,2,3,3,4,4,5,6,7,8,9,10]
ghci> quicksort "the quick brown fox jumps over the lazy dog"
" abcdeeffghijklmnooopqrstuvwxyz"
```

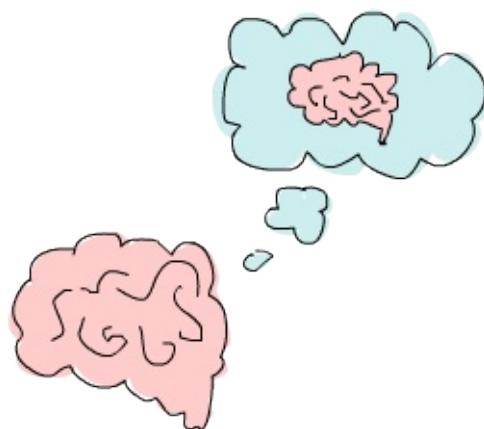
booyah! 如我所說的一樣！若給 `[5,1,9,4,6,7,3]` 排序，這個算法就會取出它的頭部，即 5。將其置於分別比它大和比它小的兩個 List 中間，得 `[1,4,3] ++ [5] ++ [9,6,7]`，我們便知道了當排序結束之時，5會在第四位，因為有3個數比它小每，也有三個數比它大。好的，接着排 `[1,4,3]` 與 `[9,6,7]`，結果就出來了！對它們的排序也是使用同樣的函數，將它們分成許多小塊，最終到達臨界條件，即空 List 經排序依然為空，有個插圖：

橙色的部分表示已定位並不再移動的元素。從左到右看，便是一個排過序的 List。在這裡我們將所有元素與 `head` 作比較，而實際上就快速排序算法而言，選擇任意元素都是可以的。被選擇的元素就被稱作錨（pivot），以方便模式匹配。小於錨的元素都在淺綠的部分，大於錨都在深綠部分，這個黃黃的坡就表示了快速排序的執行方式：



## 用遞迴來思考

我們已經寫了不少遞迴了，也許你已經發覺了其中的固定模式：先定義一個邊界條件，再定義個函數，讓它從一堆元素中取一個並做點事情後，把餘下的元素重新交給這個函數。這一模式對 List、Tree 等資料結構都是適用的。例如，`sum` 函數就是一個 List 頭部與其尾部的 `sum` 的和。一個 List 的積便是該 List 的頭與其尾部的積相乘的積，一個 List 的長度就是 1 與其尾部長度的和. 等等



再者就是邊界條件。一般而言，邊界條件就是為避免程序出錯而設置的保護措施，處理 List 時的邊界條件大部分都是空 List，而處理 Tree 時的邊界條件就是沒有子元素的節點。

處理數字時也與之相似。函數一般都得接受一個值並修改它。早些時候我們編寫過一個計算 Factorial 的函數，它便是某數與它減一的 Factorial 數的積。讓它乘以零就不行了，Factorial 數又都是非負數，邊界條件便可以定為 1，即乘法的單位元。因為任何數乘以 1 的結果還是這個數。而在 `sum` 中，加法的單位元就是 0。在快速排序中，邊界條件和單位元都是空 List，因為任一 List 與空 List 相加的結果依然是原 List。

使用遞迴來解決問題時應當先考慮遞迴會在什麼樣的條件下不可用，然後再找出它的邊界條件和單位元，考慮參數應該在何時切開(如對 List 使用模式匹配)，以及在何處執行遞迴。

# 高階函數



Haskell 中的函數可以接受函數作為參數也可以返回函數作為結果，這樣的函數就被稱作高階函數。高階函數可不只是某簡單特性而已，它貫穿于 Haskell 的方方面面。要拒絕循環與狀態的改變而通過定義問題"是什麼"來解決問題，高階函數必不可少。它們是編碼的得力工具。

## Curried functions

本質上，Haskell 的所有函數都只有一個參數，那麼我們先前編那麼多含有多個參數的函數又是怎麼回事？呵，小伎倆！所有多個參數的函數都是 Curried functions。什麼意思呢？取一個例子最好理解，就拿我們的好朋友 `max` 函數說事吧。它看起來像是取兩個參數，回傳較大的那個數。實際上，執行 `max 4 5` 時，它會首先回傳一個取一個參數的函數，其回傳值不是 4 就是該參數，取決於誰大。然後，以 5 為參數呼叫它，並取得最終結果。這聽著挺繞口的，不過這一概念十分的酷！如下的兩個呼叫是等價的：

```
ghci> max 4 5
5
ghci> (max 4) 5
5
```



把空格放到兩個東西之間，稱作函數呼叫。它有點像個運算符，並擁有最高的優先順序。看看 `max` 函數的型別: `max :: (Ord a) => a -> a -> a`。也可以寫作: `max :: (Ord a) => a -> (a -> a)`。可以讀作 `max` 取一個參數 `a`，並回傳一個函數(就是那個 `->`)，這個函數取一個 `a` 型別的參數，回傳一個 `a`。這便是為何只用箭頭來分隔參數和回傳值型別。

這樣的好處又是如何？簡言之，我們若以不全的參數來呼叫某函數，就可以得到一個不全呼叫的函數。如果你高興，構造新函數就可以如此便捷，將其傳給另一個函數也是同樣方便。

看下這個函數，簡單至極：

```
multThree :: (Num a) => a -> a -> a -> a
multThree x y z = x * y * z
```

我們若執行 `mulThree 3 5 9` 或 `((mulThree 3) 5) 9`，它背後是如何運作呢？首先，按照空格分隔，把 `3` 交給 `mulThree`。這回傳一個回傳函數的函數。然後把 `5` 交給它，回傳一個取一個參數並使之乘以 `15` 的函數。最後把 `9` 交給這一函數，回傳 `135`。想想，這個函數的型別也可以寫作 `mulThree :: (Num a) => a -> (a -> (a -> a))`，`->` 前面的東西就是函數取的參數，後面的東西就是其回傳值。所以說，我們的函數取一個 `a`，並回傳一個型別為 `(Num a) => a -> (a -> a)` 的函數，類似，這一函數回傳一個取一個 `a`，回傳一個型別為 `(Num a) => a -> a` 的函數。而最後的這個函數就只取一個 `a` 並回傳一個 `a`，如下：

```
ghci> let multTwoWithNine = mulThree 9
ghci> multTwoWithNine 2 3
54
ghci> let multWithEighteen = multTwoWithNine 2
ghci> multWithEighteen 10
180
```

前面提到，以不全的參數呼叫函數可以方便地創造新的函數。例如，搞個取一數與 `100` 比較大小的函數該如何？大可這樣：

```
compareWithHundred :: (Num a, Ord a) => a -> Ordering
compareWithHundred x = compare 100 x
```

用 `99` 呼叫它，就可以得到一個 `GT`。簡單。注意下在等號兩邊都有 `x`。想想 `compare 100` 會回傳什麼？一個取一數與 `100` 比較的函數。Wow，這不正是我們想要的？這樣重寫：

```
compareWithHundred :: (Num a, Ord a) => a -> Ordering
compareWithHundred = compare 100
```

型別聲明依然相同，因為 `compare 100` 回傳函數。`compare` 的型別為 `(Ord a) => a -> (a -> Ordering)`，用 `100` 呼叫它後回傳的函數型別為 `(Num a, Ord a) => a -> Ordering`，同時由於 `100` 還是 `Num` 型別類的實例，所以還得另留一個類約束。

Yo! 你得保證已經弄明白了 Curried functions 與不全呼叫的原理，它們很重要！

中綴函數也可以不全呼叫，用括號把它和一邊的參數括在一起就行了。這回傳一個取一參數並將其補到缺少的那一端的函數。一個簡單函數如下：

```
divideByTen :: (Floating a) => a -> a
divideByTen = (/10)
```

呼叫 `divideByTen 200` 就是 `(/10) 200`，和 `200 / 10` 等價。

一個檢查字元是否為大寫的函數：

```
isUpperAlphanum :: Char -> Bool
isUpperAlphanum = (`elem` ['A'..'Z'])
```

唯一的例外就是 `-` 運算符，按照前面提到的定義，`(-4)` 理應回傳一個並將參數減 4 的函數，而實際上，處於計算上的方便，`(-4)` 表示負 `4`。若你一定要弄個將參數減 4 的函數，就用 `subtract` 好了，像這樣 `(subtract 4)`。

若不用 `let` 給它命名或傳到另一函數中，在 ghci 中直接執行 `multThree 3 4` 會怎樣？

```
ghci> multThree 3 4
:1:0:
No instance for (Show (t -> t))
arising from a use of `print' at :1:0-12
Possible fix: add an instance declaration for (Show (t -> t))
In the expression: print it
In a 'do' expression: print it
```

ghci 說，這一表達式回傳了一個 `a -> a` 型別的函數，但它不知道該如何顯示它。函數不是 `Show` 型別類的實例，所以我們不能得到表示一函數內容的字串。若在 ghci 中計算 `1+1`，它會首先計算得 `2`，然後呼叫 `show 2` 得到該數值的字串表示，即 `"2"`，再輸出到屏幕。

## 是時候了，來點高階函數！

Haskell 中的函數可以取另一個函數做參數，也可以回傳函數。舉個例子，我們弄個取一個函數並呼叫它兩次的函數。

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```



首先注意這型別聲明。在此之前我們很少用到括號，因為 `(->)` 是自然的右結合，不過在這裡括號是必須的。它標明了首個參數是個參數與回傳值型別都是 `a` 的函數，第二個參數與回傳值的型別也都是 `a`。我們可以用 Curried functions 的思路來理解這一函數，不過免得自尋煩惱，我們姑且直接把它看作是取兩個參數回傳一個值，其首個參數是個型別為 `(a->a)` 的函數，第二個參數是個 `a`。該函數的型別可以是 `(Int->Int)`，也可以是 `(String->String)`，但第二個參數必須與之一致。

\*Note\*: 現在開始我們會直說某函數含有多個參數(除非它真的只有一個參數)。以簡潔之名，我們會說 ```(a->a->a)`

這個函數是相當的簡單，就拿參數 `f` 當函數，用 `x` 呼叫它得到的結果再去呼叫它。也就這樣玩：

```
ghci> applyTwice (+3) 10
16
ghci> applyTwice (++ " HAHA") "HEY"
"HEY HAHA HAHA"
ghci> applyTwice ("HAHA "++) "HEY"
"HAHA HAHA HEY"
ghci> applyTwice (multThree 2 2) 9
144
ghci> applyTwice (3:) [1]
[3,3,1]
```

看，不全呼叫多神奇！如果有個函數要我們給它傳個一元函數，大可以不全呼叫一個函數讓它剩一個參數，再把它交出去。

接下來我們用高階函數的編程思想來實現個標準庫中的函數，它就是 `zipWith`。它取一個函數和兩個 List 做參數，並把兩個 List 交到一起(使相應的元素去呼叫該函數)。如下就是我們的實現：

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

看下這個型別聲明，它的首個參數是個函數，取兩個參數處理交叉，其型別不必相同，不過相同也沒關係。第二三個參數都是 List，回傳值也是個 List。第一個 List 中元素的型別必須是 `a`，因為這個處理交叉的函數的第一個參數是 `a`。第二個 List 中元素的型別必為 `b`，因為這個處理交叉的函數第二個參數的型別是 `b`。回傳的 List 中元素型別為 `c`。如果一個函數說取一個型別為 `a->b->c` 的函數做參數，傳給它個 `a->a->c` 型別的也是可以的，但反過來就不行了。可以記下，若在使用高階函數的時候不清楚其型別為何，就先忽略掉它的型別聲明，再到 ghci 下用 `:t` 命令來看下 Haskell 的型別推導。

這函數的行為與普通的 `zip` 很相似，邊界條件也是相同，只不過多了個參數，即處理元素交叉的函數。它關不着邊界條件什麼事兒，所以我們就只留一個 `_`。後一個模式的函數體與 `zip` 也很像，只不過這裡是 `f x y` 而非 `(x,y)`。只要足夠通用，一個簡單的高階函數可以在不同的場合反覆使用。如下便是我們 `zipWith'` 函數本領的冰山一角：

```
ghci> zipWith' (+) [4,2,5,6] [2,6,2,3]
[6,8,7,9]
ghci> zipWith' max [6,3,2,1] [7,3,1,5]
[7,3,2,5]
ghci> zipWith' (++) ["foo ", "bar ", "baz "] ["fighters", "hoppers", "aldrin"]
["foo fighters","bar hoppers","baz aldrin"]
ghci> zipWith' (*) (replicate 5 2) [1..]
[2,4,6,8,10]
ghci> zipWith' (zipWith' (*)) [[1,2,3],[3,5,6],[2,3,4]] [[3,2,2],[3,4,5],[5,4,3]]
[[3,4,6],[9,20,30],[10,12,12]]
```

如你所見，一個簡單的高階函數就可以玩出很多花樣。命令式語言使用 `for`、`while`、賦值、狀態檢測來實現功能，再包起來留個介面，使之像個函數一樣呼叫。而函數式語言使用高階函數來抽象出常見的模式，像成對遍歷並處理兩個 List 或從中篩掉自己不需要的結果。

接下來實現標準庫中的另一個函數 `flip`，`flip` 簡單地取一個函數作參數並回傳一個相似的函數，只是它們的兩個參數倒了個。

```
flip' :: (a -> b -> c) -> (b -> a -> c)
flip' f = g
  where g x y = f y x
```

從這型別聲明中可以看出，它取一個函數，其參數型別分別為 `a` 和 `b`，而它回傳的函數的參數型別為 `b` 和 `a`。由於函數預設都是柯裡化的，`->` 為右結合，這裡的第二對括號其實並無必要，`(a -> b -> c) -> (b -> a -> c)` 與 `(a -> b -> c) -> (b -> (a -> c))` 等價，也與

`(a -> b -> c) -> b -> a -> c` 等價。前面我們寫了 `g x y = f y x`，既然這樣可行，那麼 `f y x = g x y` 不也一樣？這一來我們可以改成更簡單的寫法：

```
flip' :: (a -> b -> c) -> b -> a -> c
flip' f y x = f x y
```

在這裡我們就利用了 Curried functions 的優勢，只要呼叫 `flip' f` 而不帶 `y` 和 `x`，它就會回傳一個倆參數倒個的函數。`flip` 處理的函數往往都是用來傳給其他函數呼叫，於是我們可以發揮 Curried functions 的優勢，預先想好發生完全呼叫的情景並處理好回傳值。

```
ghci> flip' zip [1,2,3,4,5] "hello"
[('h',1),('e',2),('l',3),('l',4),('o',5)]
ghci> zipWith (flip' div) [2,2..] [10,8,6,4,2]
[5,4,3,2,1]
```

## map 與 filter

`map` 取一個函數和 List 做參數，遍歷該 List 的每個元素來呼叫該函數產生一個新的 List。看下它的型別聲明和實現：

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

從這型別聲明中可以看出，它取一個取 `a` 回傳 `b` 的函數和一組 `a` 的 List，並回傳一組 `b`。這就是 Haskell 的有趣之處：有時只看型別聲明就能對函數的行為猜個大致。`map` 函數多才多藝，有一百萬種用法。如下是其中一小部分：

```
ghci> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
ghci> map (++ "!")
["BIFF", "BANG", "POW"]
["BIFF!", "BANG!", "POW!"]
ghci> map (replicate 3) [3..6]
[[3,3,3],[4,4,4],[5,5,5],[6,6,6]]
ghci> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]
[[1,4],[9,16,25,36],[49,64]]
ghci> map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]
[1,3,6,2,2]
```

你可能會發現，以上的所有程式碼都可以用 List Comprehension 來替代。`map (+3)` `[1,5,3,1,6]` 與 `[x+3 | x <- [1,5,3,1,6]]` 完全等價。

`filter` 函數取一個限制條件和一個 List，回傳該 List 中所有符合該條件的元素。它的型別聲明及實現大致如下：

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
| p x      = x : filter p xs
| otherwise = filter p xs
```

很簡單。只要 `p x` 所得的結果為真，就將這一元素加入新 List，否則就無視之。幾個使用範例：

```
ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
ghci> filter (==3) [1,2,3,4,5]
[3]
ghci> filter even [1..10]
[2,4,6,8,10]
ghci> let notNull x = not (null x) in filter notNull [[1,2,3],[],[3,4,5],[2,2],[],[],[]]
[[1,2,3],[3,4,5],[2,2]]
ghci> filter (`elem` ['a'..'z']) "u LaUgH aT mE BeCaUsE I aM diFfeREnt"
"uagameasadifeent"
ghci> filter (`elem` ['A'..'Z']) "i lauGh At You BecAuse u r aLL the Same"
"GAYBALLS"
```

同樣，以上都可以用 List Comprehension 的限制條件來實現。並沒有教條規定你必須在什麼情況下用 `map` 和 `filter` 還是 List Comprehension，選擇權歸你，看誰舒服用誰就是。如果有多個限制條件，只能連着套好幾個 `filter` 或用 `&&` 等邏輯函數的組合之，這時就不如 List comprehension 來得爽了。

還記得上一章的那個 `quicksort` 函數麼？我們用到了 List Comprehension 來過濾大於或小於錨的元素。換做 `filter` 也可以實現，而且更加易讀：

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
  let smallerSorted = quicksort (filter ((<=x)) xs)
      biggerSorted = quicksort (filter ((>x)) xs)
  in  smallerSorted ++ [x] ++ biggerSorted
```



`map` 和 `filter` 是每個函數式程序員的麵包黃油(呃, `map` 和 `filter` 還是 List Comprehension 並不重要)。想想前面我們如何解決給定周長尋找合適直角三角形的問題的? 在命令式編程中, 我們可以套上三個循環逐個測試當前的組合是否滿足條件, 若滿足, 就打印到屏幕或其他類似的輸出。而在函數式編程中, 這行就都交給 `map` 和 `filter`。你弄個取一參數的函數, 把它交給 `map` 過一遍 List, 再 `filter` 之找到合適的結果。感謝 Haskell 的惰性, 即便是你多次 `map` 一個 `List` 也只會遍歷一遍該 List, 要找出小於 100000 的數中最大的 3829 的倍數, 只需過濾結果所在的 List 就行了。

要找出小於 100000 的 3829 的所有倍數, 我們應當過濾一個已知結果所在的 List.

```
largestDivisible :: (Integral a) => a
largestDivisible = head (filter p [100000, 99999..])
  where p x = x `mod` 3829 == 0
```

首先, 取一個降序的小於 100000 所有數的 List, 然後按照限制條件過濾它。由於這個 List 是降序的, 所以結果 List 中的首個元素就是最大的那個數。惰性再次行動! 由於我們只取這結果 List 的首個元素, 所以它並不關心這 List 是有限還是無限的, 在找到首個合適的結果處運算就停止了。

接下來, 我們就要找出所有小於 10000 且為奇的平方的和, 得先提下 `takeWhile` 函數, 它取一個限制條件和 List 作參數, 然後從頭開始遍歷這一 List, 並回傳符合限制條件的元素。而一旦遇到不符合條件的元素, 它就停止了。如果我們要取出字串 "elephants know how to party" 中的首個單詞, 可以 `takewhile (/=' ') "elephants know how to party"`, 回傳 "elephants"。okay, 要求所有小於 10000 的奇數的平方的和, 首先就用  $(^2)$  函數 `map` 掉這個無限的 List `[1..]`。然後過濾之, 只取奇數就是了。在大於 10000 處將它斷開, 最後前面的所有元素加到一起。這一切連寫函數都不用, 在 ghci 下直接搞定。

```
ghci> sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
166650
```

不錯! 先從幾個初始數據(表示所有自然數的無限 List), 再 `map` 它, `filter` 它, 切它, 直到它符合我們的要求, 再將其加起來。這用 List comprehension 也是可以的, 而哪種方式就全看你的個人口味。

```
ghci> sum (takeWhile (<10000) [m | m <- [n^2 | n <- [1..]], odd m])
166650
```

感謝 Haskell 的惰性特質, 這一切才得以實現。我們之所以可以 `map` 或 `filter` 一個無限 List, 是因為它的操作不會被立即執行, 而是拖延一下。只有我們要求 Haskell 交給我們 `sum` 的結果的時候, `sum` 函數才會跟 `takewhile` 說, 它要這些數。`takewhile` 就再去要求 `filter` 和 `map` 行動起來, 並在遇到大於等於 10000 時候停止。下個問題與 Collatz 序列有關, 取一個自然數, 若為偶數就除以 2。若為奇數就乘以 3 再加 1。再用相同的方式處理所

得的結果，得到一組數字構成的的鏈。它有個性質，無論任何以任何數字開始，最終的結果都會歸 1。所以若拿 13 當作起始數，就可以得到這樣一個序列 13, 40, 20, 10, 5, 16, 8, 4, 2, 1。13\*3+1 得 40, 40 除 2 得 20, 如是繼續，得到一個 10 個元素的鏈。

好的，我們想知道的是：以 1 到 100 之間的所有數作為起始數，會有多少個鏈的長度大於 15？

```
chain :: (Integral a) => a -> [a]
chain 1 = [1]
chain n
| even n = n:chain (n `div` 2)
| odd n = n:chain (n*3 + 1)
```

該鏈止於 1，這便是邊界條件。標準的遞歸函數：

```
ghci> chain 10
[10,5,16,8,4,2,1]
ghci> chain 1
[1]
ghci> chain 30
[30,15,46,23,70,35,106,53,160,80,40,20,10,5,16,8,4,2,1]
```

yay! 貌似工作良好。現在由這個函數來告訴我們結果：

```
numLongChains :: Int
numLongChains = length (filter isLong (map chain [1..100]))
  where isLong xs = length xs > 15
```

我們把 `chain` 函數 `map` 到 `[1..100]`，得到一組鏈的 List，然後用個限制條件過濾長度大於 15 的鏈。過濾完畢後就可以得出結果list中的元素個數。

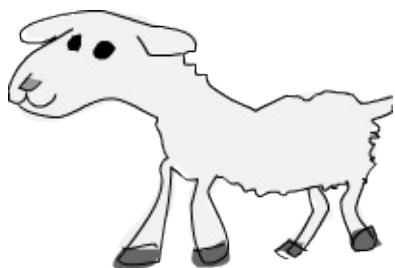
\*Note\*: 這函數的型別為 ```numLongChains :: Int```。這是由於歷史原因，```length``` 回傳一個 ```Int``` 而不是 `[Int]`。

用 `map`，我們可以寫出類似 `map (*) [0..]` 之類的程式碼。如果只是為了例證 Curried functions 和不全呼叫的函數是真正的值及其原理，那就是你可以把函數傳遞或把函數裝在 List 中(只是你還不能將它們轉換為字串)。迄今為止，我們還只是 `map` 單參數的函數到 List，如 `map (*2) [0..]` 可得一組型別為 `(Num a) => [a]` 的 List，而 `map (*) [0..]` 也是完全沒問題的。`*` 的型別為 `(Num a) => a -> a -> a`，用單個參數呼叫二元函數會回傳一個一元函數。如果用 `*` 來 `map` 一個 `[0..]` 的 List，就會得到一組一元函數組成的 List，即 `(Num a) => [a->a]`。`map (*) [0..]` 所得的結果寫起來大約就是 `[(0*), (1*), (2*)..]`。

```
ghci> let listOffuns = map (*) [0..]
ghci> (listOffuns !! 4) 5
20
```

取所得 List 的第五個元素可得一函數，與 `(*4)` 等價。然後用 `5` 呼叫它，與 `(* 4) 5` 或 `4*5` 都是等價的。

## lambda



lambda 就是匿名函數。有些時候我們需要傳給高階函數一個函數，而這函數我們只會用這一次，這就弄個特定功能的 lambda。編寫 lambda，就寫個 `\` (因為它看起來像是希臘字母的 lambda -- 如果你斜視的厲害)，後面是用空格分隔的參數，`->` 後面就是函數體。通常我們都是用括號將其括起，要不然它就會佔據整個右邊部分。

向上 5 英吋左右，你會看到我們在 `numLongChain` 函數中用 `where` 語句聲明了個 `isLong` 函數傳遞給了 `filter`。好的，用 lambda 代替它。

```
numLongChains :: Int
numLongChains = length (filter (\xs -> length xs > 15) (map chain [1..100]))
```



lambda 是個表達式，因此我們可以任意傳遞。表達式 `(\xs -> length xs > 15)` 回傳一個函數，它可以告訴我們一個 List 的長度是否大於 15。

不熟悉 Curried functions 與不全呼叫的人們往往會寫出很多 lambda，而實際上大部分都是沒必要的。例如，表達式 `map (+3) [1, 6, 3, 2]` 與 `map (\x -> x+3) [1, 6, 3, 2]` 等價，`(+3)` 和 `(\x -> x+3)` 都是給一個數加上 3。不用說，在這種情況下不用 lambda 要清爽的多。

和普通函數一樣，lambda 也可以取多個參數。

```
ghci> zipWith (\a b -> (a * 30 + 3) / b) [5,4,3,2,1] [1,2,3,4,5]
[153.0, 61.5, 31.0, 15.75, 6.6]
```

同普通函數一樣，你也可以在 lambda 中使用模式匹配，只是你無法為一個參數設置多個模式，如 `[]` 和 `(x:xs)`。lambda 的模式匹配若失敗，就會引發一個運行時錯誤，所以慎用！

```
ghci> map (\(a,b) -> a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)]
[3,8,9,8,7]
```

一般情況下，lambda 都是括在括號中，除非我們想要後面的整個語句都作為 lambda 的函數體。很有趣，由於有柯裡化，如下的兩段是等價的：

```
addThree :: (Num a) => a -> a -> a -> a
addThree x y z = x + y + z
```

```
addThree :: (Num a) => a -> a -> a -> a
addThree = \x -> \y -> \z -> x + y + z
```

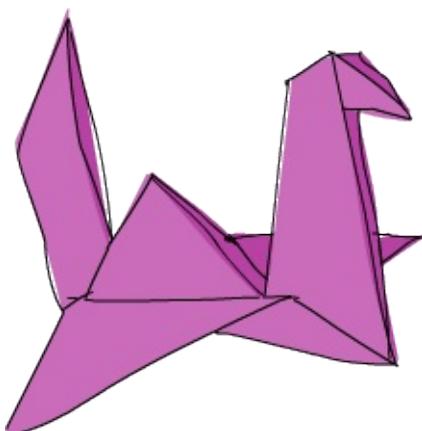
這樣的函數聲明與函數體中都有 `->`，這一來型別聲明的寫法就很明白了。當然第一段程式碼更易讀，不過第二個函數使得柯裡化更容易理解。

有些時候用這種語句寫還是挺酷的，我覺得這應該是最易讀的 `flip` 函數實現了：

```
flip' :: (a -> b -> c) -> b -> a -> c
flip' f = \x y -> f y x
```

儘管這與 `flip' f x y = f y x` 等價，但它可以更明白地表示出它會產生一個新的函數。`flip` 常用來處理一個函數，再將回傳的新函數傳遞給 `map` 或 `filter`。所以如此使用 lambda 可以更明確地表現出回傳值是個函數，可以用來傳遞給其他函數作參數。

## 關鍵字 fold



回到當初我們學習遞歸的情景。我們會發現處理 List 的許多函數都有固定的模式，通常我們會將邊界條件設置為空 List，再引入 `(x:xs)` 模式，對單個元素和餘下的 List 做些事情。這一模式是如此常見，因此 Haskell 引入了一組函數來使之簡化，也就是 `fold`。它們與 `map` 有點像，只是它們回傳的是單個值。

一個 `fold` 取一個二元函數，一個初始值(我喜歡管它叫累加值)和一個需要摺疊的 List。這個二元函數有兩個參數，即累加值和 List 的首項(或尾項)，回傳值是新的累加值。然後，以新的累加值和新的 List 首項呼叫該函數，如是繼續。到 List 遍歷完畢時，只剩下一個累加值，也就是最終的結果。

首先看下 `foldl` 函數，也叫做左摺疊。它從 List 的左端開始摺疊，用初始值和 List 的頭部呼叫這二元函數，得一新的累加值，並用新的累加值與 List 的下一個元素呼叫二元函數。如是繼續。

我們再實現下 `sum`，這次用 `fold` 替代那複雜的遞歸：

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs
```

測試下，一二三～

```
ghci> sum' [3,5,2,1]
11
```

**0+3**  
[3, 5, 2, 1]

**3+5**  
[5, 2, 1]

**8+2**  
[2, 1]

**10+1**  
[1]

**11**

我們深入看下 `fold` 的執行過程：`\acc x -> acc + x` 是個二元函數，`0` 是初始值，`xs` 是待摺疊的 List。一開始，累加值為 `0`，當前項為 `3`，呼叫二元函數 `0+3` 得 `3`，作新的累加值。接着來，累加值為 `3`，當前項為 `5`，得新累加值 `8`。再往後，累加值為 `8`，當前項為 `2`，得新累加值 `10`。最後累加值為 `10`，當前項為 `1`，得 `11`。恭喜，你完成了一次摺疊 (`fold`)！

左邊的這個圖表示了摺疊的執行過程，一步又一步(一天又一天!)。淺棕色的數字都是累加值，你可以從中看出 List 是如何從左端一點點加到累加值上的。唔對對對！如果我們考慮到函數的柯裡化，可以寫出更簡單的實現：

```
sum' :: (Num a) => [a] -> a
sum' = foldl (+) 0
```

這個 lambda 函數 `(\acc x -> acc + x)` 與 `(+)` 等價。我們可以把 `xs` 等一應參數省略掉，反正呼叫 `foldl (+) 0` 會回傳一個取 List 作參數的函數。通常，如果你的函數類似 `foo a = bar b a`，大可改為 `foo = bar b`。有柯裡化嘛。

呼呼，進入右摺疊前我們再實現個用到左摺疊的函數。大家肯定都知道 `elem` 是檢查某元素是否屬於某 List 的函數吧，我就不再提了(唔，剛提了)。用左摺疊實現它：

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' y ys = foldl (\acc x -> if x == y then True else acc) False ys
```

好好好，這裡我們有什麼？起始值與累加值都是布爾值。在處理 `fold` 時，累加值與最終結果的型別總是相同的。如果你不知道怎樣對待起始值，那我告訴你，我們先假設它不存在，以 `False` 開始。我們要是 `fold` 一個空 List，結果就是 `False`。然後我們檢查當前元素是

否為我們尋找的，如果是，就令累加值為 `True`，如果否，就保留原值不變。若 `False`，及表明當前元素不是。若 `True`，就表明已經找到了。

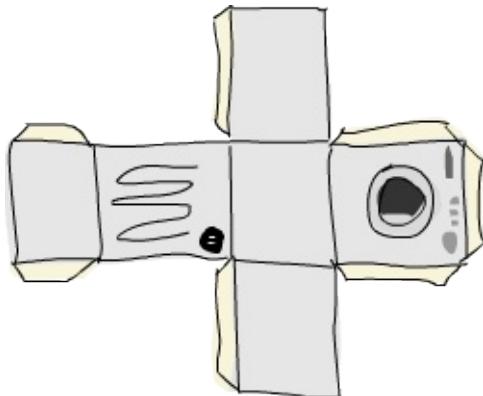
右摺疊 `foldr` 的行為與左摺疊相似，只是累加值是從 List 的右邊開始。同樣，左摺疊的二元函數取累加值作首個參數，當前值為第二個參數(即 `\acc x -> ...`)，而右摺疊的二元函數參數的順序正好相反(即 `\x acc -> ...`)。這倒也正常，畢竟是從右端開始摺疊。

累加值可以是任何型別，可以是數值，布爾值，甚至一個新的 List。我們可以用右 `fold` 實現 `map` 函數，累加值就是個 List。將 `map` 處理過的元素一個一個連到一起。很容易想到，起始值就是空 List。

```
map' :: (a -> b) -> [a] -> [b]
map' f xs = foldr (\x acc -> f x : acc) [] xs
```

如果我們用 `(+3)` 來映射 `[1, 2, 3]`，它就會先到達 List 的右端，我們取最後那個元素，也就是 `3` 來呼叫 `(+3)`，得 `6`。追加 `(:)` 到累加值上，`6:[]` 得 `[6]` 並成為新的累加值。用 `2` 呼叫 `(+3)`，得 `5`，追加到累加值，於是累加值成了 `[5, 6]`。再對 `1` 呼叫 `(+3)`，並將結果 `4` 追加到累加值，最終得結果 `[4, 5, 6]`。

當然，我們也完全可以用左摺疊來實現它，`map' f xs = foldl (\acc x -> acc ++ [f x]) [] xs` 就行了。不過問題是，使用 `(++)` 往 List 後面追加元素的效率要比使用 `(:)` 低得多。所以在生成新 List 的時候人們一般都是使用右摺疊。



反轉一個 List，既也可以通過右摺疊，也可以通過左摺疊。有時甚至不需要管它們的分別，如 `sum` 函數的左右摺疊實現都是十分相似。不過有個大的不同，那就是右摺疊可以處理無限長度的資料結構，而左摺疊不可以。將無限 List 從中斷開執行左摺疊是可以的，不過若是向右，就永遠到不了頭了。

所有遍歷 List 中元素並據此回傳一個值的操作都可以交給 `fold` 實現。無論何時需要遍歷 List 並回傳某值，都可以嘗試下 `fold`。因此，`fold` 的地位可以說與 `map` 和 `filter` 並駕齊驅，同為函數式編程中最常用的函數之一。

**foldl1** 與 **foldr1** 的行為與 **foldl** 和 **foldr** 相似，只是你無需明確提供初始值。他們假定 List 的首個(或末尾)元素作為起始值，並從旁邊的元素開始摺疊。這一來，**sum** 函數大可這樣實現：**sum = foldl1 (+)**。這裡待摺疊的 List 中至少要有一個元素，若使用空 List 就會產生一個運行時錯誤。不過 **foldl** 和 **foldr** 與空 List 相處的就很好。所以在使用 **fold** 前，應該先想下它會不會遇到空 List，如果不遇到，大可放心使用 **foldr1** 和 **foldl1**。

為了體會 **fold** 的威力，我們就用它實現幾個庫函數：

```
maximum' :: (Ord a) => [a] -> a
maximum' = foldr1 (\x acc -> if x > acc then x else acc)

reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []

product' :: (Num a) => [a] -> a
product' = foldr1 (*)

filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x acc -> if p x then x : acc else acc) []

head' :: [a] -> a
head' = foldr1 (\x _ -> x)

last' :: [a] -> a
last' = foldl1 (\_ x -> x)
```

僅靠模式匹配就可以實現 **head** 函數和 **last** 函數，而且效率也很高。這裡只是為了演示，用 **fold** 的實現方法。我覺得我們這個 **reverse'** 定義的相當聰明，用一個空 List 做初始值，並向左展開 List，從左追加到累加值，最後得到一個反轉的新 List。**\acc x -> x : acc** 有點像 **:** 函數，只是參數順序相反。所以我們可以改成 **foldl (flip (:)) []**。

有個理解摺疊的思路：假設我們有個二元函數 **f**，起始值 **z**，如果從右摺疊 **[3, 4, 5, 6]**，實際上執行的就是 **f 3 (f 4 (f 5 (f 6 z)))**。**f** 會被 List 的尾項和累加值呼叫，所得的結果會作為新的累加值傳入下一個呼叫。假設 **f** 是 **(+)**，起始值 **z** 是 **0**，那麼就是 **3 + (4 + (5 + (6 + 0)))**，或等價的首碼形式：**(+) 3 ((+) 4 ((+) 5 ((+) 6 0)))**。相似，左摺疊一個 List，以 **g** 為二元函數，**z** 為累加值，它就與 **g (g (g (g z 3) 4) 5) 6** 等價。如果用 **flip (:)** 作二元函數，**[]** 為累加值(看得出，我們是要反轉一個 List)，這就與 **flip (:)(flip (:)(flip (:)(flip ([] 3) 4) 5) 6** 等價。顯而易見，執行該表達式的結果為 **[6, 5, 4, 3]**。

**scanl** 和 **scanr** 與 **foldl** 和 **foldr** 相似，只是它們會記錄下累加值的所有狀態到一個 List。也有 **scanl1** 和 **scanr1**。

```
ghci> scanl (+) 0 [3,5,2,1]
[0,3,8,10,11]
ghci> scanr (+) 0 [3,5,2,1]
[11,8,3,1,0]
ghci> scanl1 (\acc x -> if x > acc then x else acc) [3,4,5,3,7,9,2,1]
[3,4,5,5,7,9,9,9]
ghci> scanl (flip (:)) [] [3,2,1]
[[], [3], [2, 3], [1, 2, 3]]
```

當使用 `scanl` 時，最終結果就是 List 的最後一個元素。而在 `scanr` 中則是第一個。

```
sqrtSums :: Int
sqrtSums = length (takeWhile (<1000) (scanl1 (+) (map sqrt [1..]))) + 1
```

```
ghci> sqrtSums
131
ghci> sum (map sqrt [1..131])
1005.0942035344083
ghci> sum (map sqrt [1..130])
993.6486803921487
```

`scan` 可以用來跟蹤 `fold` 函數的執行過程。想想這個問題，取所有自然數的平方根的和，尋找在何處超過 `1000`？先 `map sqrt [1..]`，然後用個 `fold` 來求它們的和。但在這裡我們想知道求和的過程，所以使用 `scan`，`scan` 完畢時就可以得到小於 `1000` 的所有和。所得結果 List 的第一個元素為 `1`，第二個就是 `1+根2`，第三個就是 `1+根2+根3`。若有 `x` 個和小於 `1000`，那結果就是 `x+1`。

## 有\$的函數呼叫

好的，接下來看看 `$` 函數。它也叫作函數呼叫符。先看下它的定義：

```
($) :: (a -> b) -> a -> b
f $ x = f x
```



什麼鬼東西？這沒啥意義的操作符？它只是個函數呼叫符罷了？好吧，不全是，但差不多。普通的函數呼叫符有最高的優先順序，而 `$` 的優先順序則最低。用空格的函數呼叫符是左結合的，如 `f a b c` 與 `((f a) b) c` 等價，而 `$` 則是右結合的。

聽著不錯。但有什麼用？它可以減少我們程式碼中括號的數目。試想有這個表達式：`sum (map sqrt [1..130])`。由於低優先順序的 `$`，我們可以將其改為 `sum $ map sqrt [1..130]`，可以省敲不少鍵！`sqrt 3 + 4 + 9` 會怎樣？這會得到 9, 4 和根 3 的和。若要取 `(3+4+9)` 的平方根，就得 `sqrt (3+4+9)` 或用 `$ : sqrt $ 3+4+9`。因為 `$` 有最低的優先順序，所以你可以把 `$` 看作是在右面寫一對括號的等價形式。

`sum (filter (> 10) (map (*2) [2..10]))` 該如何？嗯，`$` 是右結合，`f (g (z x))` 與 `f $ g $ z x` 等價。所以我麼可以將 `sum (filter (> 10) (map (*2) [2..10]))` 重寫為 `sum $ filter (> 10) $ map (*2) [2..10]`。

除了減少括號外，`$` 還可以將數據作為函數使用。例如映射一個函數呼叫符到一組函數組成的 List：

```
ghci> map ($ 3) [(4+),(10*),(^2),sqrt]
[7.0,30.0,9.0,1.7320508075688772]
```

## Function composition

在數學中，函數組合是這樣定義的： $(f \circ g)(x) = f(g(x))$ ，表示組合兩個函數成為一個函數。以 `x` 呼叫這一函數，就與用 `x` 呼叫 `g` 再用所得的結果呼叫 `f` 等價。

Haskell 中的函數組合與之很像，即 . 函數。其定義為：

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```



注意下這型別聲明，`f` 的參數型別必須與 `g` 的回傳型別相同。所以得到的組合函數的參數型別與 `g` 相同，回傳型別與 `f` 相同。表達式 `negate . (*3)` 回傳一個求一數字乘以 3 後的負數的函數。

函數組合的用處之一就是生成新函數，並傳遞給其它函數。當然我們可以用 lambda 實現，但大多數情況下，使用函數組合無疑更清楚。假設我們有一組由數字組成的 List，要將其全部轉為負數，很容易就想到應先取其絕對值，再取負數，像這樣：

```
ghci> map (\x -> negate (abs x)) [5, -3, -6, 7, -3, 2, -19, 24]
[-5, -3, -6, -7, -3, -2, -19, -24]
```

注意下這個 lambda 與那函數組合是多麼的相像。用函數組合，我們可以將程式碼改為：

```
ghci> map (negate . abs) [5, -3, -6, 7, -3, 2, -19, 24]
[-5, -3, -6, -7, -3, -2, -19, -24]
```

漂亮！函數組合是右結合的，我們同時組合多個函數。表達式 `f (g (z x))` 與 `(f . g . z)` 等價。按照這個思路，我們可以將

```
ghci> map (\xs -> negate (sum (tail xs))) [[1..5], [3..6], [1..7]]
[-14, -15, -27]
```

改為：

```
ghci> map (negate . sum . tail) [[1..5], [3..6], [1..7]]
[-14, -15, -27]
```

不過含多個參數的函數該怎麼辦？好，我們可以使用不全呼叫使每個函數都只剩下一個參數。`sum (replicate 5 (max 6.7 8.9))` 可以重寫為 `(sum . replicate 5 . max 6.7) 8.9` 或 `sum . replicate 5 . max 6.7 $ 8.9`。在這裡會產生一個函數，它取與 `max 6.7` 同樣的參數，並使用結果呼叫 `replicate 5` 再用 `sum` 求和。最後用 `8.9` 呼叫該函數。不過一般你可以這麼讀，用 `8.9` 呼叫 `max 6.7`，然後使它 `replicate 5`，再 `sum` 之。如果你打算用函數組合來替掉那堆括號，可以先在最靠近參數的函數後面加一個 `$`，接着就用 `.` 組合其所有函數呼叫，而不用管最後那個參數。如果有這樣一段程式碼：`replicate 100 (product (map (*3) (zipWith max [1,2,3,4,5] [4,5,6,7,8]))))`，可以改為：`replicate 100 . product . map (*3) . zipWith max [1,2,3,4,5] $ [4,5,6,7,8]`。如果表達式以 3 個括號結尾，就表示你可以將其修改為函數組合的形式。

函數組合的另一用途就是定義 point free style (也稱作 pointless style) 的函數。就拿我們之前寫的函數作例子：

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (+) 0 xs
```

等號的兩端都有個 `xs`。由於有柯裡化 (Currying)，我們可以省掉兩端的 `xs`。`foldl (+) 0` 回傳的就是一個取一 List 作參數的函數，我們把它修改為 `sum' = foldl (+) 0`，這就是 point free style。下面這個函數又該如何改成 point free style 呢？

```
fn x = ceiling (negate (tan (cos (max 50 x)))))
```

像剛才那樣簡單去掉兩端的 `x` 是不行的，函數定義中 `x` 的右邊還有括號。`cos (max 50)` 是有錯誤的，你不能求一個函數的餘弦。我們的解決方法就是，使用函數組合。

```
fn = ceiling . negate . tan . cos . max 50
```

漂亮！point free style 會令你去思考函數的組合方式，而非數據的傳遞方式，更加簡潔明瞭。你可以將一組簡單的函數組合在一起，使之形成一個複雜的函數。不過函數若過于複雜，再使用 point free style 往往會適得其反，因此構造較長的函數組合鏈是不被鼓勵的(雖然我本人熱衷于函數組合)。更好的解決方法，就是使用 `let` 語句給中間的運算結果綁定一個名字，或者說把問題分解成幾個小問題再組合到一起。這樣一來我們程式碼的讀者就可以輕鬆些，不必要糾結那巨長的函數組合鏈了。

在 `map` 和 `filter` 那節中，我們求了小於 10000 的所有奇數的平方的和。如下就是將其置於一個函數中的樣子：

```
oddSquareSum :: Integer
oddSquareSum = sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
```

身為函數組合狂人，我可能會這麼寫：

```
oddSquareSum :: Integer
oddSquareSum = sum . takeWhile (<10000) . filter odd . map (^2) $ [1..]
```

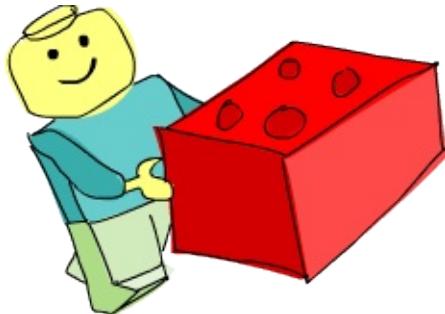
不過若是給別人看，我可能就這麼寫了：

```
oddSquareSum :: Integer
oddSquareSum =
  let oddSquares = filter odd $ map (^2) [1..]
      belowLimit = takeWhile (<10000) oddSquares
  in sum belowLimit
```

這段程式碼可贏不了程式碼花樣大賽，不過我們的讀者可能會覺得它比函數組合鏈更好看。

# 模組 (Modules)

## 裝載模組



Haskell 中的模組是含有一組相關的函數，型別和型別類的組合。而 Haskell 程序的本質便是從主模組中引用其它模組並呼叫其中的函數來執行操作。這樣可以把程式碼分成多塊，只要一個模組足夠的獨立，它裡面的函數便可以被不同的程序反覆重用。這就讓不同的程式碼各司其職，提高了程式碼的健壯性。

Haskell 的標準庫就是一組模組，每個模組都含有一組功能相近或相關的函數和型別。有處理 List 的模組，有處理並發的模組，也有處理複數的模組，等等。目前為止我們談及的所有函數、型別以及型別類都是 `Prelude` 模組的一部分，它預設自動裝載。在本章，我們看一下幾個常用的模組，在開始瀏覽其中的函數之前，我們先得知道如何裝載模組。

在 Haskell 中，裝載模組的語法為 `import`，這必須得在函數的定義之前，所以一般都是將它置於程式碼的頂部。無疑，一段程式碼中可以裝載很多模組，只要將 `import` 語句分行寫開即可。裝載 `Data.List` 試下，它裡面有很多實用的 List 處理函數。

執行 `import Data.List`，這樣一來 `Data.List` 中包含的所有函數就都進入了全局命名空間。也就是說，你可以在程式碼的任意位置呼叫這些函數。`Data.List` 模組中有個 `nub` 函數，它可以篩掉一個 List 中的所有重複元素。用點號將 `length` 和 `nub` 組合：`length . nub`，即可得到一個與 `(\xs -> length (nub xs))` 等價的函數。

```
import Data.List

numUniques :: (Eq a) => [a] -> Int
numUniques = length . nub
```

你也可以在 ghci 中裝載模組，若要呼叫 `Data.List` 中的函數，就這樣：

```
ghci> :m Data.List
```

若要在 ghci 中裝載多個模組，不必多次 `:m` 命令，一下就可以全部搞定：

```
ghci> :m Data.List Data.Map Data.Set
```

而你的程序中若已經有包含的程式碼，就不必再用 `:m` 了。

如果你只用得到某模組的兩個函數，大可僅包含它倆。若僅裝載 `Data.List` 模組 `nub` 和 `sort`，就這樣：

```
import Data.List (nub, sort)
```

也可以只包含除去某函數之外的其它函數，這在避免多個模組中函數的命名衝突很有用。假設我們的程式碼中已經有了一個叫做 `nub` 的函數，而裝入 `Data.List` 模組時就要把它裡面的 `nub` 除掉。

```
import Data.List hiding (nub)
```

避免命名衝突還有個方法，便是 `qualified import`，`Data.Map` 模組提供了一個按鍵索值的資料結構，它裡面有幾個和 `Prelude` 模組重名的函數。如 `filter` 和 `null`，裝入 `Data.Map` 模組之後再呼叫 `filter`，Haskell 就不知道它究竟是哪個函數。如下便是解決的方法：

```
import qualified Data.Map
```

這樣一來，再呼叫 `Data.Map` 中的 `filter` 函數，就必須得 `Data.Map.filter`，而 `filter` 依然是為我們熟悉喜愛的樣子。但是要在每個函數前面都加一個 `Data.Map` 實在是太煩人了！那就給它起個別名，讓它短些：

```
import qualified Data.Map as M
```

好，再呼叫 `Data.Map` 模組的 `filter` 函數的話僅需 `M.filter` 就行了

要瀏覽所有的標準庫模組，參考這個手冊。翻閱標準庫中的模組和函數是提升個人 Haskell 水平的重要途徑。你也可以各個模組的原始碼，這對 Haskell 的深入學習及掌握都是大有好處的。

檢索函數或搜尋函數位置就用 [<http://www.Haskell.org/hoogle/> Hoogle]，相當了不起的 Haskell 搜索引擎！你可以用函數名，模組名甚至型別聲明來作為檢索的條件。

## Data.List

顯而易見，`Data.List` 是關於 List 操作的模組，它提供了一組非常有用的 List 處理函數。在前面我們已經見過了其中的幾個函數(如 `map` 和 `filter`)，這是 `Prelude` 模組出於方便起見，導出了幾個 `Data.List` 裡的函數。因為這幾個函數是直接引用自 `Data.List`，所以就無需使用 `qualified import`。在下面，我們來看看幾個以前沒見過的函數：

**intersperse** 取一個元素與 List 作參數，並將該元素置於 List 中每對元素的中間。如下是個例子：

```
ghci> intersperse '.' "MONKEY"
"M.O.N.K.E.Y"
ghci> intersperse 0 [1,2,3,4,5,6]
[1,0,2,0,3,0,4,0,5,0,6]
```

**intercalate** 取兩個 List 作參數。它會將第一個 List 交叉插入第二個 List 中間，並返回一個 List.

```
ghci> intercalate " " ["hey","there","guys"]
"hey there guys"
ghci> intercalate [0,0,0] [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,0,0,0,4,5,6,0,0,0,7,8,9]
```

**transpose** 函數可以反轉一組 List 的 List。你若把一組 List 的 List 看作是個 2D 的矩陣，那 `transpose` 的操作就是將其列為行。

```
ghci> transpose [[1,2,3],[4,5,6],[7,8,9]]
[[1,4,7],[2,5,8],[3,6,9]]
ghci> transpose ["hey","there","guys"]
["htg", "ehu", "yey", "rs", "e"]
```

假如有兩個多項式  $3x^2 + 5x + 9$ ， $10x^3 + 9$  和  $8x^3 + 5x^2 + x - 1$ ，將其相加，我們可以列三個 List: `[0,3,5,9]`，`[10,0,0,9]` 和 `[8,5,1,-1]` 來表示。再用如下的方法取得結果。

```
ghci> map sum $ transpose [[0,3,5,9],[10,0,0,9],[8,5,1,-1]]
[18,8,6,17]
```



使用 `transpose` 處理這三個 List 之後，三次幕就到了第一行，二次幕到了第二行，以此類推。在用 `sum` 函數將其映射，即可得到正確的結果。

`foldl'` 和 `foldl1'` 是它們各自惰性實現的嚴格版本。在用 `fold` 處理較大的 List 時，經常會遇到堆棧溢出的問題。而這罪魁禍首就是 `fold` 的惰性：在執行 `fold` 時，累加器的值並不會被立即更新，而是做一個"在必要時會取得所需的結果"的承諾。每過一遍累加器，這一行為就重複一次。而所有的這堆"承諾"最終就會塞滿你的堆棧。嚴格的 `fold` 就不會有這一問題，它們不會作"承諾"，而是直接計算中間值的結果並繼續執行下去。如果用惰性 `fold` 時經常遇到溢出錯誤，就應換用它們的嚴格版。

**concat** 把一組 List 連接為一個 List。

```
ghci> concat ["foo", "bar", "car"]
"foobarcar"
ghci> concat [[3,4,5],[2,3,4],[2,1,1]]
[3,4,5,2,3,4,2,1,1]
```

它相當於移除一級嵌套。若要徹底地連接其中的元素，你得 `concat` 它兩次才行。

**concatMap** 函數與 `map` 一個 List 之後再 `concat` 它等價。

```
ghci> concatMap (replicate 4) [1..3]
[1,1,1,1,2,2,2,2,3,3,3,3]
```

**and** 取一組布林值 List 作參數。只有其中的值全為 `True` 的情況下才會返回 `True`。

```
ghci> and $ map (>4) [5,6,7,8]
True
ghci> and $ map (==4) [4,4,4,3,4]
False
```

**or** 與 `and` 相似，一組布林值 List 中若存在一個 `True` 它就返回 `True`。

```
ghci> or $ map (==4) [2,3,4,5,6,1]
True
ghci> or $ map (>4) [1,2,3]
False
```

**any** 和 **all** 取一個限制條件和一組布林值 List 作參數，檢查是否該 List 的某個元素或每個元素都符合該條件。通常較 `map` 一個 List 到 `and` 或 `or` 而言，使用 `any` 或 `all` 會更多些。

```
ghci> any (==4) [2,3,5,6,1,4]
True
ghci> all (>4) [6,9,10]
True
ghci> all (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
False
ghci> any (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
True
```

**iterate** 取一個函數和一個值作參數。它會用該值去呼叫該函數並用所得的結果再次呼叫該函數，產生一個無限的 List.

```
ghci> take 10 $ iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512]
ghci> take 3 $ iterate (++ "haha") "haha"
["haha","hahahaha","hahahahahaha"]
```

**splitAt** 取一個 List 和數值作參數，將該 List 在特定的位置斷開。返回一個包含兩個 List 的二元組.

```
ghci> splitAt 3 "heyman"
("hey", "man")
ghci> splitAt 100 "heyman"
("heyman", "")
ghci> splitAt (-3) "heyman"
("", "heyman")
ghci> let (a,b) = splitAt 3 "foobar" in b ++ a
"barfoo"
```

**takeWhile** 這一函數十分的實用。它從一個 List 中取元素，一旦遇到不符合條件的某元素就停止.

```
ghci> takeWhile (>3) [6,5,4,3,2,1,2,3,4,5,4,3,2,1]
[6,5,4]
ghci> takeWhile (/=' ') "This is a sentence"
"This"
```

如果要求所有三次方小於 1000 的數的和，用 `filter` 來過濾 `map (^3) [1..]` 所得結果中所有小於 1000 的數是不行的。因為對無限 List 執行的 `filter` 永遠都不會停止。你已經知道了這個 List 是單增的，但 Haskell 不知道。所以應該這樣：

```
ghci> sum $ takeWhile (<10000) $ map (^3) [1..]
53361
```

用 `(^3)` 處理一個無限 List，而一旦出現了大於 10000 的元素這個 List 就被切斷了，`sum` 到一起也就輕而易舉。

`dropWhile` 與此相似，不過它是扔掉符合條件的元素。一旦限制條件返回 `False`，它就返回 List 的餘下部分。方便實用！

```
ghci> dropWhile (/=' ') "This is a sentence"
" is a sentence"
ghci> dropWhile (<3) [1,2,2,2,3,4,5,4,3,2,1]
[3,4,5,4,3,2,1]
```

給一 `Tuple` 組成的 List，這 Tuple 的首項表示股票價格，第二三四項分別表示年,月,日。我們想知道它是在哪天首次突破 \$1000 的！

```
ghci> let stock = [(994.4,2008,9,1),(995.2,2008,9,2),(999.2,2008,9,3),(1001.4,2008,9,4),(1001.4,2008,9,4)]
```

`span` 與 `takewhile` 有點像，只是它返回兩個 List。第一個 List 與同參數呼叫 `takewhile` 所得的結果相同，第二個 List 就是原 List 中餘下的部分。

```
ghci> let (fw, rest) = span (/=' ') "This is a sentence" in "First word:" ++ fw ++ ", the
"First word: This, the rest: is a sentence"
```

`span` 是在條件首次為 `False` 時斷開 List，而 `break` 則是在條件首次為 `True` 時斷開 List。`break p` 與 `span (not . p)` 是等價的。

```
ghci> break (==4) [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
ghci> span (/=4) [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
```

`break` 返回的第二個 List 就會以第一個符合條件的元素開頭。

`sort` 可以排序一個 List，因為只有能夠作比較的元素才可以被排序，所以這一 List 的元素必須是 `Ord` 型別類的實例型別。

```
ghci> sort [8,5,3,2,1,6,4,2]
[1,2,2,3,4,5,6,8]
ghci> sort "This will be sorted soon"
" Tbdeehiillnooorssstw"
```

`group` 取一個 List 作參數，並將其中相鄰並相等的元素各自歸類，組成一個個子 List.

```
ghci> group [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[[1,1,1],[2,2,2],[3,3],[2,2,2],[5],[6],[7]]
```

若在 `group` 一個 List 之前給它排序就可以得到每個元素在該 List 中的出現次數。

```
ghci> map (\l@(x:xs) -> (x,length l)) . group . sort $ [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[(1,4),(2,7),(3,2),(5,1),(6,1),(7,1)]
```

`inits` 和 `tails` 與 `init` 和 `tail` 相似，只是它們會遞歸地呼叫自身直到什麼都不剩，看：

```
ghci> inits "w00t"
[ "", "w", "w0", "w00", "w00t" ]
ghci> tails "w00t"
[ "w00t", "00t", "0t", "t", "" ]
ghci> let w = "w00t" in zip (inits w) (tails w)
[ ("", "w00t"), ("w", "00t"), ("w0", "0t"), ("w00", "t"), ("w00t", "") ]
```

我們用 `fold` 實現一個搜索子 List 的函數：

```
search :: (Eq a) => [a] -> [a] -> Bool
search needle haystack =
  let nlen = length needle
  in foldl (\acc x -> if take nlen x == needle then True else acc) False (tails haystack)
```

首先，對搜索的 List 呼叫 `tails`，然後遍歷每個 List 來檢查它是不是我們想要的。

由此我們便實現了一個類似 `isInfixOf` 的函數，`isInfixOf` 從一個 List 中搜索一個子 List，若該 List 包含子 List，則返回 `True`。

```
ghci> "cat" `isInfixOf` "im a cat burglar"
True
ghci> "Cat" `isInfixOf` "im a cat burglar"
False
ghci> "cats" `isInfixOf` "im a cat burglar"
False
```

**isPrefixOf** 與 **isSuffixOf** 分別檢查一個 List 是否以某子 List 開頭或者結尾.

```
ghci> "hey" `isPrefixOf` "hey there!"
True
ghci> "hey" `isPrefixOf` "oh hey there!"
False
ghci> "there!" `isSuffixOf` "oh hey there!"
True
ghci> "there!" `isSuffixOf` "oh hey there"
False
```

**elem** 與 **notElem** 檢查一個 List 是否包含某元素.

**partition** 取一個限制條件和 List 作參數，返回兩個 List，第一個 List 中包含所有符合條件的元素，而第二個 List 中包含餘下的.

```
ghci> partition (`elem` ['A'..'Z']) "BOBsidneyMORGANeddy"
("BOBMORGAN", "sidneyeddy")
ghci> partition (>3) [1,3,5,6,3,2,1,0,3,7]
([5,6,7], [1,3,3,2,1,0,3])
```

瞭解這個與 `span` 和 `break` 的差異是很重要的.

```
ghci> span (`elem` ['A'..'Z']) "BOBsidneyMORGANeddy"
("BOB", "sidneyMORGANeddy")
```

`span` 和 `break` 會在遇到第一個符合或不符合條件的元素處斷開，而 `partition` 則會遍歷整個 List.

**find** 取一個 List 和限制條件作參數，並返回首個符合該條件的元素，而這個元素是個 `Maybe` 值。在下章，我們將深入地探討相關的算法和資料結構，但在這裡你只需瞭解 `Maybe` 值是 `Just something` 或 `Nothing` 就夠了。與一個 List 可以為空也可以包含多個元素相似，一個 `Maybe` 可以為空，也可以是單一元素。同樣與 List 類似，一個 Int 型的 List 可以寫作 `[Int]`，`Maybe` 有個 Int 型可以寫作 `Maybe Int`。先試一下 `find` 函數再說.

```
ghci> find (>4) [1,2,3,4,5,6]
Just 5
ghci> find (>9) [1,2,3,4,5,6]
Nothing
ghci> :t find
find :: (a -> Bool) -> [a] -> Maybe a
```

注意一下 `find` 的型別，它的返回結果為 `Maybe a`，這與 `[a]` 的寫法有點像，只是 `Maybe` 型的值只能為空或者單一元素，而 `List` 可以為空，一個元素，也可以是多個元素。

想想前面那段找股票的程式碼，`head (dropWhile (\(val,y,m,d) -> val < 1000) stock)`。但 `head` 並不安全！如果我們的股票沒漲過 \$1000 會怎樣？`dropWhile` 會返回一個空 `List`，而對空 `List` 取 `head` 就會引發一個錯誤。把它改成 `find (\(val,y,m,d) -> val > 1000) stock` 就安全多啦，若存在合適的結果就得到它，像 `Just (1001.4, 2008, 9, 4)`，若不存在合適的元素（即我們的股票沒有漲到過 \$1000），就會得到一個 `Nothing`。

`elemIndex` 與 `elem` 相似，只是它返回的不是布林值，它只是'可能' (`Maybe`) 返回我們找的元素的索引，若這一元素不存在，就返回 `Nothing`。

```
ghci> :t elemIndex
elemIndex :: (Eq a) => a -> [a] -> Maybe Int
ghci> 4 `elemIndex` [1,2,3,4,5,6]
Just 3
ghci> 10 `elemIndex` [1,2,3,4,5,6]
Nothing
```

`elemIndices` 與 `elemIndex` 相似，只不過它返回的是 `List`，就不需要 `Maybe` 了。因為不存在用空 `List` 就可以表示，這就與 `Nothing` 相似了。

```
ghci> ' ' `elemIndices` "Where are the spaces?"
[5, 9, 13]
```

`findIndex` 與 `find` 相似，但它返回的是可能存在的首個符合該條件元素的索引。`findIndices` 會返回所有符合條件的索引。

```
ghci> findIndex (==4) [5,3,2,1,6,4]
Just 5
ghci> findIndex (==7) [5,3,2,1,6,4]
Nothing
ghci> findIndices (`elem` ['A'..'Z']) "Where Are The Caps?"
[0, 6, 10, 14]
```

在前面，我們講過了 `zip` 和 `zipWith`，它們只能將兩個 List 組到一個二元組數或二參函數中，但若要組三個 List 該怎麼辦？好說~ 有 `zip3`, `zip4` ...和 `zipWith3`, `zipWith4` ...直到 7。這看起來像是個 hack，但工作良好。連着組 8 個 List 的情況很少遇到。還有個聰明辦法可以組起無限多個 List，但限於我們目前的水平，就先不談了。

```
ghci> zipWith3 (\x y z -> x + y + z) [1,2,3] [4,5,2,1] [2,2,3]
[7,9,8]
ghci> zip4 [2,3,3] [2,2,2] [5,5,3] [2,2,2]
[(2,2,5,2),(3,2,5,2),(3,2,3,2)]
```

與普通的 `zip` 操作相似，以返回的 List 中長度最短的那個為準。

在處理來自檔案或其它地方的輸入時，`lines` 會非常有用。它取一個字串作參數。並返回由其中的每一行組成的 List。

```
ghci> lines "first line\nsecond line\nthird line"
["first line","second line","third line"]
```

'\n' 表示unix下的換行符，在 Haskell 的字元中，反斜杠表示特殊字元。

`unlines` 是 `lines` 的反函數，它取一組字串的 List，並將其通過 '\n' 合併到一塊。

```
ghci> unlines ["first line", "second line", "third line"]
"first line\nsecond line\nthird line\n"
```

`words` 和 `unwords` 可以把一個字串分為一組單詞或執行相反的操作，很有用。

```
ghci> words "hey these are the words in this sentence"
["hey", "these", "are", "the", "words", "in", "this", "sentence"]
ghci> words "hey these are the words in this\nsentence"
["hey", "these", "are", "the", "words", "in", "this", "sentence"]
ghci> unwords ["hey", "there", "mate"]
"hey there mate"
```

我們前面講到了 `nub`，它可以將一個 List 中的重複元素全部篩掉，使該 List 的每個元素都如雪花般獨一無二，'nub' 的含義就是'一小塊'或'一部分'，用在這裡覺得很古怪。我覺得，在函數的命名上應該用更確切的詞語，而避免使用老掉牙的過時詞彙。

```
ghci> nub [1,2,3,4,3,2,1,2,3,4,3,2,1]
[1,2,3,4]
ghci> nub "Lots of words and stuff"
"Lots fwrdanu"
```

`delete` 取一個元素和 List 作參數，會刪掉該 List 中首次出現的這一元素。

```
ghci> delete 'h' "hey there ghang!"
"ey there ghang!"
ghci> delete 'h' . delete 'h' $ "hey there ghang!"
"ey tere ghang!"
ghci> delete 'h' . delete 'h' . delete 'h' $ "hey there ghang!"
"ey tere gang!"
```

\ 表示 List 的差集操作，這與集合的差集很相似，它會從左邊 List 中的元素扣除存在於右邊 List 中的元素一次。

```
ghci> [1..10] \\ [2,5,9]
[1,3,4,6,7,8,10]
ghci> "I'm a big baby" \\ "big"
"I'm a baby"
```

**union** 與集合的並集也是很相似，它返回兩個 List 的並集，即遍歷第二個 List 若存在某元素不屬於第一個 List，則追加到第一個 List。看，第二個 List 中的重複元素就都沒了！

```
ghci> "hey man" `union` "man what's up"
"hey manwt'sup"
ghci> [1..7] `union` [5..10]
[1,2,3,4,5,6,7,8,9,10]
```

**intersection** 相當於集合的交集。它返回兩個 List 的相同部分。

```
ghci> [1..7] `intersect` [5..10]
[5,6,7]
```

**insert** 可以將一個元素插入一個可排序的 List，並將其置於首個大於等於它的元素之前，如果使用 `insert` 來給一個排過序的 List 插入元素，返回的結果依然是排序的。

```
ghci> insert 4 [1,2,3,5,6,7]
[1,2,3,4,5,6,7]
ghci> insert 'g' $ ['a'..'f'] ++ ['h'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> insert 3 [1,2,4,3,2,1]
[1,2,3,4,3,2,1]
```

`length`, `take`, `drop`, `splitAt`, `!!` 和 `replicate` 之類的函數有個共同點。那就是它們的參數中都有個 Int 值（或者返回 Int 值），我覺得使用 `Integral` 或 `Num` 型別類會更好，但出於歷史原因，修改這些會破壞掉許多既有的程式碼。在 `Data.List` 中包含了更通用的替代版，如：`genericLength`, `genericTake`, `genericDrop`, `genericSplitAt`, `genericIndex` 和 `genericReplicate`。`length` 的型別聲明為 `length :: [a] -> Int`，而我們若要像這樣求它

的平均值，`let xs = [1..6] in sum xs / length xs`，就會得到一個型別錯誤，因為 `/` 運算符不能對 `Int` 型使用！而 `genericLength` 的型別聲明則為 `genericLength :: (Num a) => [b] -> a`，`Num` 既可以是整數又可以是浮點數，`let xs = [1..6] in sum xs / genericLength xs` 這樣再求平均數就不會有問題了。

`nub`, `delete`, `union`, `intsect` 和 `group` 函數也有各自的通用替代版 `nubBy`, `deleteBy`, `unionBy`, `intersectBy` 和 `groupBy`，它們的區別就是前一組函數使用 `(==)` 來測試是否相等，而帶 `By` 的那組則取一個函數作參數來判定相等性，`group` 就與 `groupBy (==)` 等價。

假如有個記錄某函數在每秒的值的 `List`，而我們要按照它小於零或者大於零的交界處將其分為一組子 `List`。如果用 `group`，它只能將相鄰並相等的元素組到一起，而在這裡我們的標準是它們是否互為相反數。`groupBy` 登場！它取一個含兩個參數的函數作為參數來判定相等性。

```
ghci> let values = [-4.3, -2.4, -1.2, 0.4, 2.3, 5.9, 10.5, 29.1, 5.3, -2.4, -14.5, 2.9, 2.3]
ghci> groupBy (\x y -> (x > 0) == (y > 0)) values
[[-4.3, -2.4, -1.2], [0.4, 2.3, 5.9, 10.5, 29.1, 5.3], [-2.4, -14.5], [2.9, 2.3]]
```

這樣一來我們就可以很清楚地看出哪部分是正數，哪部分是負數，這個判斷相等性的函數會在兩個元素同時大於零或同時小於零時返回 `True`。也可以寫作 `\x y -> (x > 0) && (y > 0) || (x <= 0) && (y <= 0)`。但我覺得第一個寫法的可讀性更高。`Data.Function` 中還有個 `on` 函數可以讓它的表達更清晰，其定義如下：

```
on :: (b -> b -> c) -> (a -> b) -> a -> a -> c
f `on` g = \x y -> f (g x) (g y)
```

執行 `(==) `on` (> 0)` 得到的函數就與 `\x y -> (x > 0) == (y > 0)` 基本等價。`on` 與帶 `By` 的函數在一起會非常好用，你可以這樣寫：

```
ghci> groupBy ((==) `on` (> 0)) values
[[-4.3, -2.4, -1.2], [0.4, 2.3, 5.9, 10.5, 29.1, 5.3], [-2.4, -14.5], [2.9, 2.3]]
```

可讀性很高！你可以大聲念出來：按照元素是否大於零，給它分類！

同樣，`sort`, `insert`, `maximum` 和 `min` 都有各自的通用版本。如 `groupBy` 類似，`sortBy`, `insertBy`, `maximumBy` 和 `minimumBy` 都取一個函數來比較兩個元素的大小。像 `sortBy` 的型別聲明為：`sortBy :: (a -> a -> Ordering) -> [a] -> [a]`。前面提過，`Ordering` 型別可以有三個值，`LT`, `EQ` 和 `GT`。`compare` 取兩個 `Ord` 型別類的元素作參數，所以 `sort` 與 `sortBy compare` 等價。

`List` 是可以比較大小的，且比較的依據就是其中元素的大小。如果按照其子 `List` 的長度為標準當如何？很好，你可能已經猜到了，`sortBy` 函數。

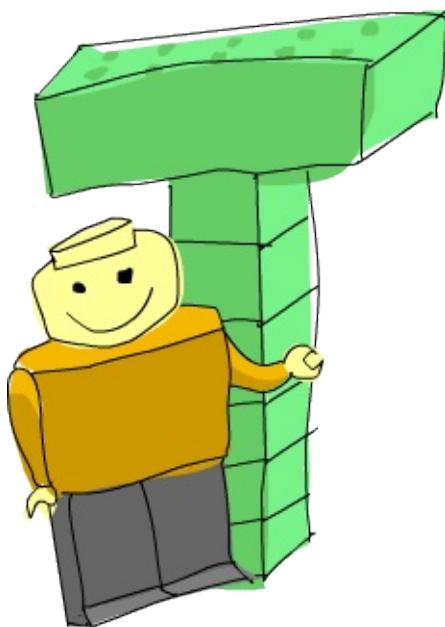
```
ghci> let xs = [[5,4,5,4,4],[1,2,3],[3,5,4,3],[],[2],[2,2]]
ghci> sortBy (compare `on` length) xs
[[], [2], [2, 2], [1, 2, 3], [3, 5, 4, 3], [5, 4, 5, 4, 4]]
```

太絕了! `compare `on` length`, 乖乖, 這簡直就是英文! 如果你搞不清楚 `on` 在這裡的原理, 就可以認為它與 `\x y -> length x `compare` length y` 等價。通常, 與帶 `By` 的函數打交道時, 若要判斷相等性, 則 `(==) `on` something`。若要判定大小, 則 `compare `on` something`。

## Data.Char

如其名, `Data.Char` 模組包含了一組用於處理字元的函數。由於字串的本質就是一組字元的 List, 所以往往會在 `filter` 或是 `map` 字串時用到它。

`Data.Char` 模組中含有一系列用於判定字元範圍的函數, 如下:



**isControl** 判斷一個字元是否是控制字元。**isSpace** 判斷一個字元是否是空格字元, 包括空格, tab, 換行符等。**isLower** 判斷一個字元是否為小寫。**isUpper** 判斷一個字元是否為大寫。**isAlpha** 判斷一個字元是否為字母。**isAlphaNum** 判斷一個字元是否為字母或數字。**isPrint** 判斷一個字元是否是可打印的。**isDigit** 判斷一個字元是否為數字。**isOctDigit** 判斷一個字元是否為八進制數字。**isHexDigit** 判斷一個字元是否為十六進制數字。**isLetter** 判斷一個字元是否為字母。**isMark** 判斷是否為 unicode 注音字元, 你如果是法國人就會經常用到的。**isNumber** 判斷一個字元是否為數字。**isPunctuation** 判斷一個字元是否為標點符號。**isSymbol** 判斷一個字元是否為貨幣符號。**isSeperator** 判斷一個字元是否為 unicode 空格或分隔符。**isAscii** 判斷一個字元是否在 unicode 字母表的前 128 位。**isLatin1** 判斷一個字元是否在 unicode 字母表的前 256 位。**isAsciiUpper** 判斷一個字元是否為大寫的 ascii 字元。**isAsciiLower** 判斷一個字元是否為小寫的 ascii 字元。

以上所有判斷函數的型別聲明皆為 `Char -> Bool`，用到它們的絕大多數情況都無非就是過濾字串或類似操作。假設我們在寫個程序，它需要一個由字元和數字組成的用戶名。要實現對用戶名的檢驗，我們可以結合使用 `Data.List` 模組的 `all` 函數與 `Data.Char` 的判斷函數。

```
ghci> all isAlphaNum "bobby283"
True
ghci> all isAlphaNum "eddy the fish!"
False
```

Kewl~ 免得你忘記，`all` 函數取一個判斷函數和一個 List 做參數，若該 List 的所有元素都符合條件，就返回 `True`。

也可以使用 `isSpace` 來實現 `Data.List` 的 `words` 函數。

```
ghci> words "hey guys its me"
["hey", "guys", "its", "me"]
ghci> groupBy ((==) `on` isSpace) "hey guys its me"
[["hey", " ", "guys", " ", "its", " ", "me"]]
ghci>
```

Hmm，不錯，有點 `words` 的樣子了。只是還有空格在裡面，恩，該怎麼辦？我知道，用 `filter` 濾掉它們！

```
ghci> filter (not . any isSpace) . groupBy ((==) `on` isSpace) $ "hey guys its me"
["hey", "guys", "its", "me"]
```

啊哈。

`Data.Char` 中也含有與 `Ordering` 相似的型別。`Ordering` 可以有三個值，`LT`，`GT` 和 `EQ`。這就是個枚舉，它表示了兩個元素作比較可能的結果。`GeneralCategory` 型別也是個枚舉，它表示了一個字元可能所在的分類。而得到一個字元所在分類的主要方法就是使用 `generalCategory` 函數。它的型別為：`generalCategory :: Char -> GeneralCategory`。那 31 個分類就不在此一一列出了，試下這個函數先：

```
ghci> generalCategory ' '
Space
ghci> generalCategory 'A'
UppercaseLetter
ghci> generalCategory 'a'
LowercaseLetter
ghci> generalCategory '.'
OtherPunctuation
ghci> generalCategory '9'
DecimalNumber
ghci> map generalCategory "\t\nA9?| "
[Space,Control,Control,UppercaseLetter,DecimalNumber,OtherPunctuation,MathSymbol]
```

由於 `GeneralCategory` 型別是 `Eq` 型別類的一部分，使用類似 `generalCategory c == Space` 的程式碼也是可以的。

`toUpper` 將一個字元轉為大寫字母，若該字元不是小寫字母，就按原值返回。`toLower` 將一個字元轉為小寫字母，若該字元不是大寫字母，就按原值返回。`toTitle` 將一個字元轉為 title-case，對大多數字元而言，title-case 就是大寫。`digitToInt` 將一個字元轉為 Int 值，而這一字元必須得在 `'1'..'9'`, `'a'..'f'` 或 `'A'..'F'` 的範圍之內。

```
ghci> map digitToInt "34538"
[3,4,5,3,8]
ghci> map digitToInt "FF85AB"
[15,15,8,5,10,11]
```

`intToDigit` 是 `digitToInt` 的反函數。它取一個 `0` 到 `15` 的 `Int` 值作參數，並返回一個小寫的字元。

```
ghci> intToDigit 15
'f'
ghci> intToDigit 5
'5'
```

`ord` 與 `char` 函數可以將字元與其對應的數字相互轉換。

```
ghci> ord 'a'
97
ghci> chr 97
'a'
ghci> map ord "abcdefghijklmnopqrstuvwxyz"
[97,98,99,100,101,102,103,104]
```

兩個字元的 `ord` 值之差就是它們在 unicode 字元表上的距離。

*Caesar cipher* 是加密的基礎算法，它將消息中的每個字元都按照特定的字母表進行替換。它的實現非常簡單，我們這裡就先不管字母表了。

```
encode :: Int -> String -> String
encode shift msg =
  let ords = map ord msg
      shifted = map (+ shift) ords
  in map chr shifted
```

先將一個字串轉為一組數字，然後給它加上某數，再轉回去。如果你是標準的組合牛仔，大可將函數寫為：`map (chr . (+ shift) . ord) msg`。試一下它的效果：

```
ghci> encode 3 "Heeeeey"
"Khhhh|"
ghci> encode 4 "Heeeeey"
"Liiiii}"
ghci> encode 1 "abcd"
"bcde"
ghci> encode 5 "Marry Christmas! Ho ho ho!"
"Rfww~%Hmwnxyrfx&%Mt%mt%mt&"
```

不錯。再簡單地將它轉成一組數字，減去某數後再轉回來就是解密了。

```
decode :: Int -> String -> String
decode shift msg = encode (negate shift) msg
```

```
ghci> encode 3 "I'm a little teapot"
"Lp#d#olwoh#whdsrw"
ghci> decode 3 "Lp#d#olwoh#whdsrw"
"I'm a little teapot"
ghci> decode 5 . encode 5 $ "This is a sentence"
"This is a sentence"
```

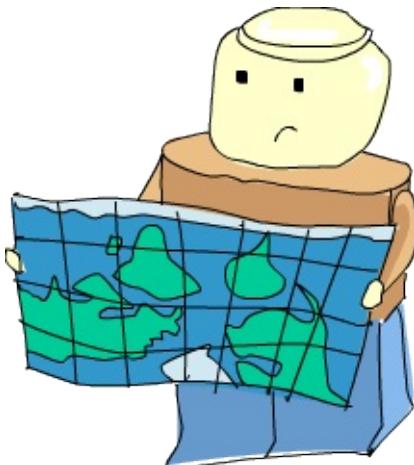
## Data.Map

關聯列表(也叫做字典)是按照鍵值對排列而沒有特定順序的一種 List。例如，我們用關聯列表儲存電話號碼，號碼就是值，人名就是鍵。我們並不關心它們的存儲順序，只要能按人名得到正確的號碼就好。在 Haskell 中表示關聯列表的最簡單方法就是弄一個二元組的 List，而這二元組就首項為鍵，後項為值。如下便是個表示電話號碼的關聯列表：

```
phoneBook = [("betty", "555-2938") ,
             ("bonnie", "452-2928") ,
             ("patsy", "493-2928") ,
             ("lucille", "205-2928") ,
             ("wendy", "939-8282") ,
             ("penny", "853-2492") ]
```

不理這貌似古怪的縮進，它就是一組二元組的 List 而已。話說對關聯列表最常見的操作就是按鍵索值，我們就寫個函數來實現它。

```
findKey :: (Eq k) => k -> [(k,v)] -> v
findKey key xs = snd . head . filter (\(k,v) -> key == k) $ xs
```



簡潔漂亮。這個函數取一個鍵和 List 做參數，過濾這一 List 僅保留鍵匹配的項，並返迴首個鍵值對。但若該關聯列表中不存在這個鍵那會怎樣？哼，那就會在試圖從空 List 中取 `head` 時引發一個運行時錯誤。無論如何也不能讓程序就這麼輕易地崩潰吧，所以就應該用 `Maybe` 型別。如果沒找到相應的鍵，就返回 `Nothing`。而找到了就返回 `Just something`。而這 `something` 就是鍵對應的值。

```
findKey :: (Eq k) => k -> [(k,v)] -> Maybe v
findKey key [] = Nothing
findKey key ((k,v):xs) =
    if key == k then
        Just v
    else
        findKey key xs
```

看這型別聲明，它取一個可判斷相等性的鍵和一個關聯列表做參數，可能 (`Maybe`) 得到一個值。聽起來不錯。這便是個標準的處理 List 的遞歸函數，邊界條件，分割 List，遞歸呼叫，都有了 -- 經典的 `fold` 模式。看看用 `fold` 怎樣實現吧。

```
findKey :: (Eq k) => k -> [(k,v)] -> Maybe v
findKey key = foldr (\(k,v) acc -> if key == k then Just v else acc) Nothing
```

\*Note\*: 通常，使用 ``fold`` 來替代類似的遞歸函數會更好些。用 ``fold`` 的程式碼讓人一目瞭然，而看明白這

```
ghci> findKey "penny" phoneBook
Just "853-2492"
ghci> findKey "betty" phoneBook
Just "555-2938"
ghci> findKey "wilma" phoneBook
Nothing
```

如魔咒般靈驗！只要我們有這姑娘的號碼就 `Just` 可以得到，否則就是 `Nothing`。方纔我們實現的函數便是 `Data.List` 模組的 `lookup`，如果要按鍵去尋找相應的值，它就必須得遍歷整個 `List`，直到找到為止。而 `Data.Map` 模組提供了更高效的方式（通過樹實現），並提供了一組好用的函數。從現在開始，我們扔掉關聯列表，改用 `Map`。由於 `Data.Map` 中的一些函數與 `Prelude` 和 `Data.List` 模組存在命名衝突，所以我們使用 `qualified import`。  
`import qualified Data.Map as Map` 在程式碼中加上這句，並 `load` 到 `ghci` 中繼續前進，看看 `Data.Map` 是如何的一座寶庫！如下便是其中函數的一瞥：

**fromList** 取一個關聯列表，返回一個與之等價的 `Map`。

```
ghci> Map.fromList [("betty", "555-2938"), ("bonnie", "452-2928"), ("lucille", "205-2928")]
fromList [("betty", "555-2938"), ("bonnie", "452-2928"), ("lucille", "205-2928")]
ghci> Map.fromList [(1,2), (3,4), (3,2), (5,5)]
fromList [(1,2), (3,2), (5,5)]
```

若其中存在重複的鍵，就將其忽略。如下即 `fromList` 的型別聲明。

```
Map.fromList :: (Ord k) => [(k, v)] -> Map.Map k v
```

這表示它取一組鍵值對的 `List`，並返回一個將 `k` 映射為 `v` 的 `map`。注意一下，當使用普通的關聯列表時，只需要鍵的可判斷相等性就行了。而在這裡，它還必須得是可排序的。這在 `Data.Map` 模組中是強制的。因為它會按照某順序將其組織在一棵樹中。在處理鍵值對時，只要鍵的型別屬於 `Ord` 型別類，就應該儘量使用 `Data.Map.empty` 返回一個空 `map`。

```
ghci> Map.empty
fromList []
```

**insert** 取一個鍵，一個值和一個 `map` 做參數，給這個 `map` 插入新的鍵值對，並返回一個新的 `map`。

```
ghci> Map.empty
fromList []
ghci> Map.insert 3 100 Map.empty
fromList [(3,100)]
ghci> Map.insert 5 600 (Map.insert 4 200 ( Map.insert 3 100 Map.empty))
fromList [(3,100),(4,200),(5,600)]
ghci> Map.insert 5 600 . Map.insert 4 200 . Map.insert 3 100 $ Map.empty
fromList [(3,100),(4,200),(5,600)]
```

通過 `empty`，`insert` 與 `fold`，我們可以編寫出自己的 `fromList`。

```
fromList' :: (Ord k) => [(k,v)] -> Map.Map k v
fromList' = foldr (\(k,v) acc -> Map.insert k v acc) Map.empty
```

簡潔明瞭的 `fold`！從一個空的 `map` 開始，然後從右摺疊，隨着遍歷不斷地往 `map` 中插入新的鍵值對。

**null** 檢查一個 `map` 是否為空。

```
ghci> Map.null Map.empty
True
ghci> Map.null $ Map.fromList [(2,3),(5,5)]
False
```

**size** 返回一個 `map` 的大小。

```
ghci> Map.size Map.empty
0
ghci> Map.size $ Map.fromList [(2,4),(3,3),(4,2),(5,4),(6,4)]
5
```

**singleton** 取一個鍵值對做參數，並返回一個只含有一個映射的 `map`。

```
ghci> Map.singleton 3 9
fromList [(3,9)]
ghci> Map.insert 5 9 $ Map.singleton 3 9
fromList [(3,9),(5,9)]
```

**lookup** 與 `Data.List` 的 `lookup` 很像，只是它的作用對象是 `map`，如果它找到鍵對應的值。就返回 `Just something`，否則返回 `Nothing`。

**member** 是個判斷函數，它取一個鍵與 `map` 做參數，並返回該鍵是否存在於該 `map`。

```
ghci> Map.member 3 $ Map.fromList [(3,6),(4,3),(6,9)]
True
ghci> Map.member 3 $ Map.fromList [(2,5),(4,5)]
False
```

**map** 與 **filter** 與其對應的 `List` 版本很相似:

```
ghci> Map.map (*100) $ Map.fromList [(1,1),(2,4),(3,9)]
fromList [(1,100),(2,400),(3,900)]
ghci> Map.filter isUpper $ Map.fromList [(1,'a'),(2,'A'),(3,'b'),(4,'B')]
fromList [(2,'A'),(4,'B')]
```

`toList` 是 `fromList` 的反函數。

```
ghci> Map.toList . Map.insert 9 2 $ Map.singleton 4 3
[(4,3),(9,2)]
```

**keys** 與 **elems** 各自返回一組由鍵或值組成的 `List`, `keys` 與 `map fst . Map.toList` 等價, `elems` 與 `map snd . Map.toList` 等價. `fromListWith` 是個很酷的小函數, 它與 `fromList` 很像, 只是它不會直接忽略掉重複鍵, 而是交給一個函數來處理它們。假設一個姑娘可以有多個號碼, 而我們有個像這樣的關聯列表:

```
phoneBook =
[("betty", "555-2938")
,("betty", "342-2492")
,("bonnie", "452-2928")
,("patsy", "493-2928")
,("patsy", "943-2929")
,("patsy", "827-9162")
,("lucille", "205-2928")
,("wendy", "939-8282")
,("penny", "853-2492")
,("penny", "555-2111")
]
```

如果用 `fromList` 來生成 `map`, 我們會丟掉許多號碼! 如下才是正確的做法:

```
phoneBookToMap :: (Ord k) => [(k, String)] -> Map.Map k String
phoneBookToMap xs = Map.fromListWith (\number1 number2 -> number1 ++ ", " ++ number2) xs
```

```
ghci> Map.lookup "patsy" $ phoneBookToMap phoneBook
"827-9162, 943-2929, 493-2928"
ghci> Map.lookup "wendy" $ phoneBookToMap phoneBook
"939-8282"
ghci> Map.lookup "betty" $ phoneBookToMap phoneBook
"342-2492, 555-2938"
```

一旦出現重複鍵，這個函數會將不同的值組在一起，同樣，也可以預設地將每個值放到一個單元素的 List 中，再用 `++` 將他們都連接在一起。

```
phoneBookToMap :: (Ord k) => [(k, a)] -> Map.Map k [a]
phoneBookToMap xs = Map.fromListWith (++) $ map (\(k,v) -> (k,[v])) xs
ghci> Map.lookup "patsy" $ phoneBookToMap phoneBook
["827-9162", "943-2929", "493-2928"]
```

很簡潔！它還有別的玩法，例如在遇到重複元素時，單選最大的那個值。

```
ghci> Map.fromListWith max [(2,3),(2,5),(2,100),(3,29),(3,22),(3,11),(4,22),(4,15)]
fromList [(2,100),(3,29),(4,22)]
```

或是將相同鍵的值都加在一起。

```
ghci> Map.fromListWith (+) [(2,3),(2,5),(2,100),(3,29),(3,22),(3,11),(4,22),(4,15)]
fromList [(2,108),(3,62),(4,37)]
```

`insertWith` 之於 `insert`，恰如 `fromListWith` 之於 `fromList`。它會將一個鍵值對插入一個 `map` 之中，而該 `map` 若已經包含這個鍵，就問問這個函數該怎麼辦。

```
ghci> Map.insertWith (+) 3 100 $ Map.fromList [(3,4),(5,103),(6,339)]
fromList [(3,104),(5,103),(6,339)]
```

`Data.Map` 裡面還有不少函數，  
[\[http://www.haskell.org/ghc/docs/latest/html/libraries/Data-Map.html 這個文檔\]](http://www.haskell.org/ghc/docs/latest/html/libraries/Data-Map.html) 中的列表就很全了。

## Data.Set



`Data.Set` 模組提供了對數學中集合的處理。集合既像 `List` 也像 `Map`：它裡面的每個元素都是唯一的，且內部的數據由一棵樹來組織(這和 `Data.Map` 模組的 `map` 很像)，必須得是可排序的。同樣是插入,刪除,判斷從屬關係之類的操作，使用集合要比 `List` 快得多。對一個集合而言，最常見的操作莫過于並集，判斷從屬或是將集合轉為 `List`.

由於 `Data.Set` 模組與 `Prelude` 模組和 `Data.List` 模組中存在大量的命名衝突，所以我們使用 `qualified import`

將 `import` 語句至于程式碼之中：

```
import qualified Data.Set as Set
```

然後在 ghci 中裝載

假定我們有兩個字串，要找出同時存在於兩個字串的字元

```
text1 = "I just had an anime dream. Anime... Reality... Are they so different?"
text2 = "The old man left his garbage can out and now his trash is all over my lawn!"
```

**fromList** 函數同你想的一樣，它取一個 `List` 作參數並將其轉為一個集合

```
ghci> let set1 = Set.fromList text1
ghci> let set2 = Set.fromList text2
ghci> set1
fromList ".?AIRadefhijlmnorstuy"
ghci> set2
fromList "!Tabcdefghilmnorstuvwxyz"
```

如你所見，所有的元素都被排了序。而且每個元素都是唯一的。現在我們取它的交集看看它們共同包含的元素：

```
ghci> Set.intersection set1 set2
fromList " adefhilmnorstuy"
```

使用 `difference` 函數可以得到存在於第一個集合但不在第二個集合的元素

```
ghci> Set.difference set1 set2
fromList ".?AIRj"
ghci> Set.difference set2 set1
fromList "!Tbcgvw"
```

也可以使用 `union` 得到兩個集合的並集

```
ghci> Set.union set1 set2
fromList ".?AIRAbcdefghijklmnorstuvwxyz"
```

`null`, `size`, `member`, `empty`, `singleton`, `insert`, `delete` 這幾個函數就跟你想的差不多啦

```
ghci> Set.null Set.empty
True
ghci> Set.null $ Set.fromList [3,4,5,5,4,3]
False
ghci> Set.size $ Set.fromList [3,4,5,3,4,5]
3
ghci> Set.singleton 9
fromList [9]
ghci> Set.insert 4 $ Set.fromList [9,3,8,1]
fromList [1,3,4,8,9]
ghci> Set.insert 8 $ Set.fromList [5..10]
fromList [5,6,7,8,9,10]
ghci> Set.delete 4 $ Set.fromList [3,4,5,4,3,4,5]
fromList [3,5]
```

也可以判斷子集與真子集，如果集合 A 中的元素都屬於集合 B，那麼 A 就是 B 的子集，如果 A 中的元素都屬於 B 且 B 的元素比 A 多，那 A 就是 B 的真子集

```
ghci> Set.fromList [2,3,4] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
True
ghci> Set.fromList [1,2,3,4,5] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
True
ghci> Set.fromList [1,2,3,4,5] `Set.isProperSubsetOf` Set.fromList [1,2,3,4,5]
False
ghci> Set.fromList [2,3,4,8] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
False
```

對集合也可以執行 `map` 和 `filter`：

```
ghci> Set.filter odd $ Set.fromList [3,4,5,6,7,2,3,4]
fromList [3,5,7]
ghci> Set.map (+1) $ Set.fromList [3,4,5,6,7,2,3,4]
fromList [3,4,5,6,7,8]
```

集合有一常見用途，那就是先 `fromList` 刪掉重複元素後再 `toList` 轉回去。儘管 `Data.List` 模組的 `nub` 函數完全可以完成這一工作，但在對付大 List 時則會明顯的力不從心。使用集合則會快很多，`nub` 函數只需 List 中的元素屬於 `Eq` 型別類就行了，而若要使用集合，它必須得屬於 `Ord` 型別類

```
ghci> let setNub xs = Set.toList $ Set.fromList xs
ghci> setNub "HEY WHATS CRACKALACKIN"
" ACEHIKLNRSTWY"
ghci> nub "HEY WHATS CRACKALACKIN"
"HEY WATSCRKLIN"
```

在處理較大的 List 時，`setNub` 要比 `nub` 快，但也可以從中看出，`nub` 保留了 List 中元素的原有順序，而 `setNub` 不。

## 建立自己的模組

我們已經見識過了幾個很酷的模組，但怎樣才能構造自己的模組呢？几乎所有的編程語言都允許你將程式碼分成多個檔案，Haskell 也不例外。在編程時，將功能相近的函數和型別至于同一模組中會是個很好的習慣。這樣一來，你就可以輕鬆地一個 `import` 來重用其中的函數。

接下來我們將構造一個由計算幾何圖形體積和面積組成的模組，先從新建一個 `Geometry.hs` 的檔案開始。

在模組的開頭定義模組的名稱，如果檔案名叫做 `Geometry.hs` 那它的名字就得是 `Geometry`。在聲明出它含有的函數名之後就可以編寫函數的實現啦，就這樣寫：

```
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidArea
, cuboidVolume
) where
```

如你所見，我們提供了對球體、立方體和立方體的面積和體積的解法。繼續進發，定義函數體：

```

module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidArea
, cuboidVolume
) where

sphereVolume :: Float -> Float
sphereVolume radius = (4.0 / 3.0) * pi * (radius ^ 3)

sphereArea :: Float -> Float
sphereArea radius = 4 * pi * (radius ^ 2)

cubeVolume :: Float -> Float
cubeVolume side = cuboidVolume side side side

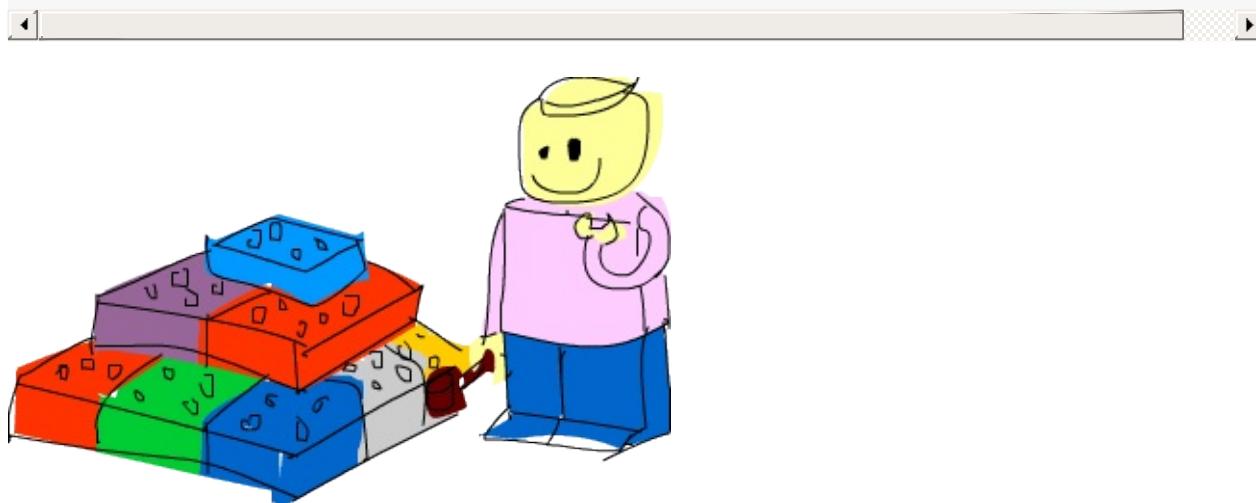
cubeArea :: Float -> Float
cubeArea side = cuboidArea side side side

cuboidVolume :: Float -> Float -> Float -> Float
cuboidVolume a b c = rectangleArea a b * c

cuboidArea :: Float -> Float -> Float -> Float
cuboidArea a b c = rectangleArea a b * 2 + rectangleArea a c * 2 + rectangleArea c b * 2

rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b

```



標準的幾何公式。有幾個地方需要注意一下，由於立方體只是長方體的特殊形式，所以在求它面積和體積的時候我們就將它當作是邊長相等的長方體。在這裡還定義了一個 `helper` 函數，`rectangleArea` 它可以通過長方體的兩條邊計算出長方體的面積。它僅僅是簡單的相乘而已，份量不大。但請注意我們可以在這一模組中呼叫這個函數，而它不會被導出！因為我們這個模組只與三維圖形打交道。

當構造一個模組的時候，我們通常只會導出那些行為相近的函數，而其內部的實現則是隱蔽的。如果有人用到了 `Geometry` 模組，就不需要關心它的內部實現是如何。我們作為編寫者，完全可以隨意修改這些函數甚至將其刪掉，沒有人會注意到裡面的變動，因為我們並不把它們導出。

要使用我們的模組，只需：

```
import Geometry
```

將 `Geometry.hs` 檔案至于用到它的程序檔案的同一目錄之下。

模組也可以按照分層的結構來組織，每個模組都可以含有多個子模組。而子模組還可以有自己的子模組。我們可以把 `Geometry` 分成三個子模組，而一個模組對應各自的圖形對象。

首先，建立一個 `Geometry` 檔案夾，注意首字母要大寫，在裡面新建三個檔案

如下就是各個檔案的內容：

`sphere.hs`

```
module Geometry.Sphere
( volume
, area
) where

volume :: Float -> Float
volume radius = (4.0 / 3.0) * pi * (radius ^ 3)

area :: Float -> Float
area radius = 4 * pi * (radius ^ 2)
```

`cuboid.hs`

```
module Geometry.Cuboid
( volume
, area
) where

volume :: Float -> Float -> Float -> Float
volume a b c = rectangleArea a b * c

area :: Float -> Float -> Float -> Float
area a b c = rectangleArea a b * 2 + rectangleArea a c * 2 + rectangleArea c b * 2

rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

## cube.hs

```
module Geometry.Cube
( volume
, area
) where

import qualified Geometry.Cuboid as Cuboid

volume :: Float -> Float
volume side = Cuboid.volume side side side

area :: Float -> Float
area side = Cuboid.area side side side
```

好的! 先是 `Geometry.Sphere`。注意，我們將它置於 `Geometry` 檔案夾之中並將它的名字定為 `Geometry.Sphere`。對 `Cuboid` 也是同樣，也注意下，在三個模組中我們定義了許多名稱相同的函數，因為所在模組不同，所以不會產生命名衝突。若要在 `Geometry.Cube` 使用 `Geometry.Cuboid` 中的函數，就不能直接 `import Geometry.Cuboid`，而必須得 `qualified import`。因為它們中間的函數名完全相同。

```
import Geometry.Sphere
```

然後，呼叫 `area` 和 `volume`，就可以得到球體的面積和體積，而若要用到兩個或更多此類模組，就必須得 `qualified import` 來避免重名。所以就得這樣寫：

```
import qualified Geometry.Sphere as Sphere
import qualified Geometry.Cuboid as Cuboid
import qualified Geometry.Cube as Cube
```

然後就可以呼叫 `Sphere.area`，`Sphere.volume`，`Cuboid.area` 了，而每個函數都只計算其對應物體的面積和體積。

以後你若發現自己的程式碼體積龐大且函數眾多，就應該試着找找目的相近的函數能否裝入各自的模組，也方便日後的重用。

# 構造我們自己的 Types 和 Typeclasses

## Algebraic Data Types 入門

在前面的章節中，我們談了一些 Haskell 內建的型別和 Typeclass。而在本章中，我們將學習構造型別和 Typeclass 的方法。

我們已經見識過許多型別，如 `Bool`、`Int`、`Char`、`Maybe` 等等，不過在 Haskell 中該如何構造自己的型別呢？好問題，一種方法是使用 `data` 關鍵字。首先我們來看看 `Bool` 在標準函式庫中的定義：

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

`data` 表示我們要定義一個新的型別。`=` 的左端標明型別的名稱即 `Bool`，`=` 的右端就是值構造子 (*Value Constructor*)，它們明確了該型別可能的值。`|` 讀作"或"，所以可以這樣閱讀該聲明：`Bool` 型別的值可以是 `True` 或 `False`。型別名和值構造子的首字母必大寫。

相似，我們可以假想 `Int` 型別的聲明：

```
data Int = -2147483648 | -2147483647 | ... | -1 | 0 | 1 | 2 | ... | 2147483647
```



頭尾兩個值構造子分別表示了 `Int` 型別的最小值和最大值，注意到真正的型別宣告不是長這個樣子的，這樣寫只是為了便於理解。我們用省略號表示中間省略的一大段數字。

我們想想 Haskell 中圖形的表示方法。表示圓可以用一個 Tuple，如 `(43.1, 55.0, 10.4)`，前兩項表示圓心的位置，末項表示半徑。聽著不錯，不過三維向量或其它什麼東西也可能是這種形式！更好的方法就是自己構造一個表示圖形的型別。假定圖形可以是圓 (Circle) 或長方形 (Rectangle)：

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
```

這是啥，想想？`Circle` 的值構造子有三個項，都是 `Float`。可見我們在定義值構造子時，可以在後面跟幾個型別表示它包含值的型別。在這裡，前兩項表示圓心的坐標，尾項表示半徑。`Rectangle` 的值構造子取四個 `Float` 項，前兩項表示其左上角的坐標，後兩項表示右下角的坐標。

談到「項」(field)，其實應為「參數」(parameters)。值構造子的本質是個函數，可以返回一個型別的值。我們看下這兩個值構造子的型別聲明：

```
ghci> :t Circle
Circle :: Float -> Float -> Float -> Shape
ghci> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
```

Cool，這麼說值構造子就跟普通函數並無二致囉，誰想得到？我們寫個函數計算圖形面積：

```
surface :: Shape -> Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

值得一提的是，它的型別聲明表示了該函數取一個 `Shape` 值並返回一個 `Float` 值。寫 `Circle -> Float` 是不可以的，因為 `Circle` 並非型別，真正的型別應該是 `Shape`。這與不能寫 `True->False` 的道理是一樣的。再就是，我們使用的模式匹配針對的都是值構造子。之前我們匹配過 `[]`、`False` 或 `5`，它們都是不包含參數的值構造子。

我們只關心圓的半徑，因此不需理會表示坐標的前兩項：

```
ghci> surface $ Circle 10 20 10
314.15927
ghci> surface $ Rectangle 0 0 100 100
10000.0
```

Yay, it works！不過我們若嘗試輸出 `Circle 10 20` 到控制台，就會得到一個錯誤。這是因為 Haskell 還不知道該型別的字元串表示方法。想想，當我們往控制台輸出值的時候，Haskell 會先呼叫 `show` 函數得到這個值的字元串表示才會輸出。因此要讓我們的 `Shape` 型別成為 `Show` 型別類的成員。可以這樣修改：

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float deriving (Show)
```

先不去深究 *deriving* (派生) , 可以先這樣理解：若在 `data` 單型的後面加上 `deriving (Show)` , 那 Haskell 就會自動將該型別至于 `show` 型別類之中。好了，由於值構造子是個函數，因此我們可以拿它交給 `map` , 拿它不全呼叫，以及普通函數能做的一切。

```
ghci> Circle 10 20 5
Circle 10.0 20.0 5.0
ghci> Rectangle 50 230 60 90
Rectangle 50.0 230.0 60.0 90.0
```

我們若要取一組不同半徑的同心圓，可以這樣：

```
ghci> map (Circle 10 20) [4,5,6,6]
[Circle 10.0 20.0 4.0,Circle 10.0 20.0 5.0,Circle 10.0 20.0 6.0,Circle 10.0 20.0 6.0]
```

我們的型別還可以更好。增加加一個表示二維空間中點的型別，可以讓我們的 `Shape` 更加容易理解：

```
data Point = Point Float Float deriving (Show)
data Shape = Circle Point Float | Rectangle Point Point deriving (Show)
```

注意下 `Point` 的定義，它的型別與值構造子用了相同的名字。沒啥特殊含義，實際上，在一個型別含有唯一值構造子時這種重名是很常見的。好的，如今我們的 `circle` 含有兩個項，一個是 `Point` 型別，一個是 `Float` 型別，好作區分。`Rectangle` 也是同樣，我們得修改 `surface` 函數以適應型別定義的變動。

```
surface :: Shape -> Float
surface (Circle _ r) = pi * r ^ 2
surface (Rectangle (Point x1 y1) (Point x2 y2)) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

唯一需要修改的地方就是模式。在 `circle` 的模式中，我們無視了整個 `Point` 。而在 `Rectangle` 的模式中，我們用了一個嵌套的模式來取得 `Point` 中的項。若出於某原因而需要整個 `Point` ，那麼直接匹配就是了。

```
ghci> surface (Rectangle (Point 0 0) (Point 100 100))
10000.0
ghci> surface (Circle (Point 0 0) 24)
1809.5574
```

表示移動一個圖形的函數該怎麼寫？它應當取一個 `Shape` 和表示位移的兩個數，返回一個位於新位置的圖形。

```
nudge :: Shape -> Float -> Float -> Shape
nudge (Circle (Point x y) r) a b = Circle (Point (x+a) (y+b)) r
nudge (Rectangle (Point x1 y1) (Point x2 y2)) a b = Rectangle (Point (x1+a) (y1+b)) (Point (x2+a) (y2+b))
```

簡潔明瞭。我們再給這一 `Shape` 的點加上位移的量。

```
ghci> nudge (Circle (Point 34 34) 10) 5 10
Circle (Point 39.0 44.0) 10.0
```

如果不想直接處理 `Point`，我們可以搞個輔助函數 (auxilliary function)，初始從原點創建圖形，再移動它們。

```
baseCircle :: Float -> Shape
baseCircle r = Circle (Point 0 0) r

baseRect :: Float -> Float -> Shape
baseRect width height = Rectangle (Point 0 0) (Point width height)
```

```
ghci> nudge (baseRect 40 100) 60 23
Rectangle (Point 60.0 23.0) (Point 100.0 123.0)
```

毫無疑問，你可以把你的數據型別導出到模組中。只要把你的型別與要導出的函數寫到一起就是了。再在後面跟個括號，列出要導出的值構造子，用逗號隔開。如要導出所有的值構造子，那就寫個`..`。

若要將這裡定義的所有函數和型別都導出到一個模組中，可以這樣：

```
module Shapes
( Point(..)
, Shape(..)
, surface
, nudge
, baseCircle
, baseRect
) where
```

一個 `Shape(..)`，我們就導出了 `Shape` 的所有值構造子。這一來無論誰導入我們的模組，都可以用 `Rectangle` 和 `Circle` 值構造子來構造 `Shape` 了。這與寫 `Shape(Rectangle,Circle)` 等價。

我們可以選擇不導出任何 `Shape` 的值構造子，這一來使用我們模組的人就只能用輔助函數 `baseCircle` 和 `baseRect` 來得到 `Shape` 了。`Data.Map` 就是這一套，沒有 `Map.Map [(1, 2), (3, 4)]`，因為它沒有導出任何一個值構造子。但你可以用，像 `Map.fromList` 這樣的輔助函數得到 `map`。應該記住，值構造子只是函數而已，如果不導出它們，就拒絕了使用我們模組的人呼叫它們。但可以使用其他返回該型別的函數，來取得這一型別的值。

不導出數據型別的值構造子隱藏了他們的內部實現，令型別的抽象度更高。同時，我們模組的使用者也就無法使用該值構造子進行模式匹配了。

## Record Syntax

OK，我們需要一個數據型別來描述一個人，得包含他的姓、名、年齡、身高、電話號碼以及最愛的冰淇淋。我不知你的想法，不過我覺得要瞭解一個人，這些資料就夠了。就這樣，實現出來！

```
data Person = Person String String Int Float String String deriving (Show)
```

O~Kay，第一項是名，第二項是姓，第三項是年齡，等等。我們造一個人：

```
ghci> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
ghci> guy
Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

貌似很酷，就是難讀了點兒。弄個函數得人的某項資料又該如何？如姓的函數，名的函數，等等。好吧，我們只能這樣：

```
firstName :: Person -> String
firstName (Person firstname _ _ _ _) = firstname

lastName :: Person -> String
lastName (Person _ lastname _ _ _) = lastname

age :: Person -> Int
age (Person _ _ age _ _) = age

height :: Person -> Float
height (Person _ _ _ height _) = height

phoneNumber :: Person -> String
phoneNumber (Person _ _ _ _ number _) = number

flavor :: Person -> String
flavor (Person _ _ _ _ flavor) = flavor
```

唔，我可不願意寫這樣的程式碼！雖然 it works，但也太無聊了哇。

```
ghci> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
ghci> firstName guy
"Buddy"
ghci> height guy
184.2
ghci> flavor guy
"Chocolate"
```

你可能會說，一定有更好的方法！呃，抱歉，沒有。

開個玩笑，其實有的，哈哈哈～Haskell 的發明者都是天才，早就料到了此類情形。他們引入了一個特殊的型別，也就是剛纔提到的更好的方法 -- *Record Syntax*。

```
data Person = Person { firstName :: String
                      , lastName :: String
                      , age :: Int
                      , height :: Float
                      , phoneNumber :: String
                      , flavor :: String
                      } deriving (Show)
```

與原先讓那些項一個挨一個的空格隔開不同，這裡用了花括號 {}。先寫出項的名字，如 `firstName`，後跟兩個冒號(也叫 Paamayim Nekudotayim，哈哈～(譯者不知道什麼意思～囧))，標明其型別，返回的數據型別仍與以前相同。這樣的好處就是，可以用函數從中直接按項取值。通過 Record Syntax，Haskell 就自動生成了這些函數：`firstName`，`lastName`，`age`，`height`，`phoneNumber` 和 `flavor`。

```
ghci> :t flavor
flavor :: Person -> String
ghci> :t firstName
firstName :: Person -> String
```

還有個好處，就是若派生 (deriving) 到 `Show` 型別類，它的顯示是不同的。假如我們有個型別表示一輛車，要包含生產商、型號以及出場年份：

```
data Car = Car String String Int deriving (Show)
```

```
ghci> Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967
```

若用 Record Syntax，就可以得到像這樣的新車：

```
data Car = Car {company :: String, model :: String, year :: Int} deriving (Show)
```

```
ghci> Car {company="Ford", model="Mustang", year=1967}
Car {company = "Ford", model = "Mustang", year = 1967}
```

這一來在造車時我們就不必關心各項的順序了。

表示三維向量之類簡單數據，`Vector = Vector Int Int Int` 就足夠明白了。但一個值構造子中若含有很多個項目不易區分，如一個人或者一輛車啥的，就應該使用 Record Syntax。

## Type parameters

值構造子可以取幾個參數產生一個新值，如 `car` 的構造子是取三個參數返回一個 `Car`。與之相似，型別構造子可以取型別作參數，產生新的型別。這乍一聽貌似有點深奧，不過實際上並不複雜。如果你對 C++ 的模板有瞭解，就會看到很多相似的地方。我們看一個熟悉的型別，好對型別參數有個大致印象：

```
data Maybe a = Nothing | Just a
```



這裡的 `a` 就是個型別參數。也正因為有了它，`Maybe` 就成為了一個型別構造子。在它的值不是 `Nothing` 時，它的型別構造子可以搞出 `Maybe Int`，`Maybe String` 等等諸多型別。但只一個 `Maybe` 是不行的，因為它不是型別，而是型別構造子。要成為真正的型別，必須得把它需要的型別參數全部填滿。

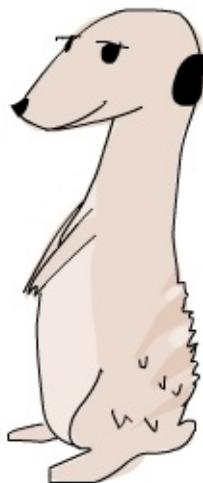
所以，如果拿 `Char` 作參數交給 `Maybe`，就可以得到一個 `Maybe Char` 的型別。如，`Just 'a'` 的型別就是 `Maybe Char`。

你可能並未察覺，在遇見 Maybe 之前我們早就接觸到型別參數了。它便是 List 型別。這裡面有點語法糖，List 型別實際上就是取一個參數來生成一個特定型別，這型別可以是 [Int]，[char] 也可以是 [String]，但不會跟在 [] 的後面。

把玩一下 Maybe !

```
ghci> Just "Haha"
Just "Haha"
ghci> Just 84
Just 84
ghci> :t Just "Haha"
Just "Haha" :: Maybe [Char]
ghci> :t Just 84
Just 84 :: (Num t) => Maybe t
ghci> :t Nothing
Nothing :: Maybe a
ghci> Just 10 :: Maybe Double
Just 10.0
```

型別參數很實用。有了它，我們就可以按照我們的需要構造出不同的型別。若執行 :t Just "Haha"，型別推導引擎就會認出它是個 Maybe [Char]，由於 Just a 裡的 a 是個字元串，那麼 Maybe a 裡的 a 一定也是個字元串。



注意下，Nothing 的型別為 Maybe a。它是多態的，若有函數取 Maybe Int 型別的參數，就大概可以傳給它一個 Nothing，因為 Nothing 中不包含任何值。Maybe a 型別可以有 Maybe Int 的行為，正如 5 可以是 Int 也可以是 Double。與之相似，空 List 的型別是 [a]，可以與一切 List 打交道。因此，我們可以 [1, 2, 3]++[]，也可以 ["ha", "ha, ", "ha"]++[]。

型別參數有很多好處，但前提是用對了地方才行。一般都是不關心型別裡面的內容，如 Maybe a。一個型別的行為若有點像是容器，那麼使用型別參數會是個不錯的選擇。我們完全可以把我們的 car 型別從

```
data Car = Car { company :: String
                , model :: String
                , year :: Int
            } deriving (Show)
```

改成：

```
data Car a b c = Car { company :: a
                        , model :: b
                        , year :: c
                    } deriving (Show)
```

但是，這樣我們又得到了什麼好處？回答很可能是，一無所得。因為我們只定義了處理 `Car` `String String Int` 型別的函數，像以前，我們還可以弄個簡單函數來描述車的屬性。

```
tellCar :: Car -> String
tellCar (Car {company = c, model = m, year = y}) = "This " ++ c ++ " " ++ m ++ " was made
```

```
ghci> let stang = Car {company="Ford", model="Mustang", year=1967}
ghci> tellCar stang
"This Ford Mustang was made in 1967"
```

可愛的小函數！它的型別聲明得很漂亮，而且工作良好。好，如果改成 `Car a b c` 又會怎樣？

```
tellCar :: (Show a) => Car String String a -> String
tellCar (Car {company = c, model = m, year = y}) = "This " ++ c ++ " " ++ m ++ " was made
```

我們只能強制性地給這個函數安一個 `(Show a) => Car String String a` 的型別約束。看得出來，這要繁複得多。而唯一的好處貌似就是，我們可以使用 `Show` 型別類的 `instance` 來作 `a` 的型別。

```
ghci> tellCar (Car "Ford" "Mustang" 1967)
"This Ford Mustang was made in 1967"
ghci> tellCar (Car "Ford" "Mustang" "nineteen sixty seven")
"This Ford Mustang was made in \"nineteen sixty seven\""
ghci> :t Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967 :: (Num t) => Car [Char] [Char] t
ghci> :t Car "Ford" "Mustang" "nineteen sixty seven"
Car "Ford" "Mustang" "nineteen sixty seven" :: Car [Char] [Char]
```

其實在現實生活中，使用 `Car String String Int` 在大多數情況下已經滿夠了。所以給 `Car` 型別加型別參數貌似並沒有什麼必要。通常我們都是都是是在一個型別中包含的型別並不影響它的行為時才引入型別參數。一組什麼東西組成的 `List` 就是一個 `List`，它不關心裡面東西的型別是啥，然而總是工作良好。若取一組數字的和，我們可以在後面的函數體中明確是一組數字的 `List`。`Maybe` 與之相似，它表示可以有什麼東西可以沒有，而不必關心這東西是啥。

我們之前還遇見過一個型別參數的應用，就是 `Data.Map` 中的 `Map k v`。`k` 表示 `Map` 中鍵的型別，`v` 表示值的型別。這是個好例子，`Map` 中型別參數的使用允許我們能夠用一個型別索引另一個型別，只要鍵的型別在 `Ord` 型別類就行。如果叫我們自己定義一個 `Map` 型別，可以在 `data` 聲明中加上一個型別類的約束。

```
data (Ord k) => Map k v = ...
```

然而 Haskell 中有一個嚴格的約定，那就是永遠不要在 `data` 聲明中添加型別約束。為什麼？嗯，因為這樣沒好處，反而得寫更多不必要的型別約束。`Map k v` 要是有 `Ord k` 的約束，那就相當於假定每個 `Map` 的相關函數都認為 `k` 是可排序的。若不給數據型別加約束，我們就不必給那些不關心鍵是否可排序的函數另加約束了。這類函數的一個例子就是 `toList`，它只是把一個 `Map` 轉換為關聯 `List` 罷了，型別聲明為 `toList :: Map k v -> [(k, v)]`。要是加上型別約束，就只能是 `toList :: (Ord k) => Map k a -> [(k, v)]`，明顯沒必要嘛。

所以說，永遠不要在 `data` 聲明中加型別約束 --- 即便看起來沒問題。免得在函數聲明中寫出過多無謂的型別約束。

我們實現個表示三維向量的型別，再給它加幾個處理函數。我麼那就給它個型別參數，雖然大多數情況都是數值型，不過這一來它就支持了多種數值型別。

```
data Vector a = Vector a a a deriving (Show)
vplus :: (Num t) => Vector t -> Vector t -> Vector t
(Vector i j k) `vplus` (Vector l m n) = Vector (i+l) (j+m) (k+n)
vectMult :: (Num t) => Vector t -> t -> Vector t
(Vector i j k) `vectMult` m = Vector (i*m) (j*m) (k*m)
scalarMult :: (Num t) => Vector t -> Vector t -> t
(Vector i j k) `scalarMult` (Vector l m n) = i*l + j*m + k*n
```

`vplus` 用來相加兩個向量，即將其所有對應的項相加。`scalarMult` 用來求兩個向量的標量積，`vectMult` 求一個向量和一個標量的積。這些函數可以處理 `Vector Int`，`Vector Integer`，`Vector Float` 等等型別，只要 `vector a` 裡的這個 `a` 在 `Num` 型別類中就行。同樣，如果你看下這些函數的型別聲明就會發現，它們只能處理相同型別的向量，其中包含的數字型別必須與另一個向量一致。注意，我們並沒有在 `data` 聲明中添加 `Num` 的類約束。反正無論怎麼著都是給函數加約束。

再度重申，型別構造子和值構造子的區分是相當重要的。在聲明數據型別時，等號=左端的那個是型別構造子，右端的(中間可能有分隔)都是值構造子。拿 `Vector t t t -> Vector t t t` 作函數的型別就會產生一個錯誤，因為在型別聲明中只能寫型別，而 `vector` 的型別構造子只有個參數，它的值構造子才是有三個。我們就慢慢耍：

```
ghci> Vector 3 5 8 `vplus` Vector 9 2 8
Vector 12 7 16
ghci> Vector 3 5 8 `vplus` Vector 9 2 8 `vplus` Vector 0 2 3
Vector 12 9 19
ghci> Vector 3 9 7 `vectMult` 10
Vector 30 90 70
ghci> Vector 4 9 5 `scalarMult` Vector 9.0 2.0 4.0
74.0
ghci> Vector 2 9 3 `vectMult` (Vector 4 9 5 `scalarMult` Vector 9 2 4)
Vector 148 666 222
```

## Derived instances



在 [types-and-type-classes.html#Typeclasses] 入門 Typeclass 101] 那一節裡面，我們瞭解了 Typeclass 的基礎內容。裡面提到，型別類就是定義了某些行為的介面。例如，`Int` 型別是 `Eq` 型別類的一個 instance，`Eq` 類就定義了判定相等性的行為。`Int` 值可以判斷相等性，所以 `Int` 就是 `Eq` 型別類的成員。它的真正威力體現在作為 `Eq` 介面的函數中，即 `==` 和 `/=`。只要一個型別是 `Eq` 型別類的成員，我們就可以使用 `==` 函數來處理這一型別。這便是為何 `4==4` 和 `"foo"/="bar"` 這樣的表達式都需要作型別檢查。

我們也會提到，人們很容易把型別類與 Java, Python, C++ 等語言的類混淆。很多人對此都倍感不解，在原先那些語言中，類就像是藍圖，我們可以根據它來創造對象、保存狀態並執行操作。而型別類更像是介面，我們不是靠它構造數據，而是給既有的數據型別描述行為。什麼東西若可以判定相等性，我們就可以讓它成為 `Eq` 型別類的 `instance`。什麼東西若可以比較大小，那就可以讓它成為 `Ord` 型別類的 `instance`。

在下一節，我們將看一下如何手工實現型別類中定義函數來構造 `instance`。現在呢，我們先瞭解下 Haskell 是如何自動生成這幾個型別類的 `instance`, `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, `Read`。只要我們在構造型別時在後面加個 `deriving` (派生)關鍵字，Haskell 就可以自動地給我們的型別加上這些行為。

看這個數據型別：

```
data Person = Person { firstName :: String
                      , lastName :: String
                      , age :: Int
                    }
```

這描述了一個人。我們先假定世界上沒有重名重姓又同齡的人存在，好，假如有兩個 record，有沒有可能是描述同一個人呢？當然可能，我麼可以判定姓名年齡的相等性，來判斷它倆是否相等。這一來，讓這個型別成為 `Eq` 的成員就很靠譜了。直接 derive 這個 `instance`：

```
data Person = Person { firstName :: String
                      , lastName :: String
                      , age :: Int
                    } deriving (Eq)
```

在一個型別 derive 為 `Eq` 的 `instance` 後，就可以直接使用 `==` 或 `/=` 來判斷它們的相等性了。Haskell 會先看下這兩個值的值構造子是否一致(這裡只是單值構造子)，再用 `==` 來檢查其中的所有數據(必須都是 `Eq` 的成員)是否一致。在這裡只有 `String` 和 `Int`，所以是沒有問題的。測試下我們的 `Eq` instance：

```
ghci> let mikeD = Person {firstName = "Michael", lastName = "Diamond", age = 43}
ghci> let adRock = Person {firstName = "Adam", lastName = "Horovitz", age = 41}
ghci> let mca = Person {firstName = "Adam", lastName = "Yauch", age = 44}
ghci> mca == adRock
False
ghci> mikeD == adRock
False
ghci> mikeD == mikeD
True
ghci> mikeD == Person {firstName = "Michael", lastName = "Diamond", age = 43}
True
```

自然，`Person` 如今已經成為了 `Eq` 的成員，我們就可以將其應用於所有在型別聲明中用到 `Eq` 類約束的函數了，如 `elem`。

```
ghci> let beastieBoys = [mca, adRock, mikeD]
ghci> mikeD `elem` beastieBoys
True
```

`Show` 和 `Read` 型別類處理可與字元串相互轉換的東西。同 `Eq` 相似，如果一個型別的構造子含有參數，那所有參數的型別必須都得屬於 `Show` 或 `Read` 才能讓該型別成為其 `instance`。就讓我們的 `Person` 也成為 `Read` 和 `Show` 的一員吧。

```
data Person = Person { firstName :: String
                      , lastName :: String
                      , age :: Int
                    } deriving (Eq, Show, Read)
```

然後就可以輸出一個 `Person` 到控制台了。

```
ghci> let mikeD = Person {firstName = "Michael", lastName = "Diamond", age = 43}
ghci> mikeD
Person {firstName = "Michael", lastName = "Diamond", age = 43}
ghci> "mikeD is: " ++ show mikeD
"mikeD is: Person {firstName = \"Michael\", lastName = \"Diamond\", age = 43}"
```

如果我們還沒讓 `Person` 型別作為 `show` 的成員就嘗試輸出它，Haskell 就會向我們抱怨，說它不知道該怎麼把它表示成一個字元串。不過現在既然已經 derive 成為了 `Show` 的一個 `instance`，它就知道了。

`Read` 几乎就是與 `show` 相對的型別類，`show` 是將一個值轉換成字元串，而 `read` 則是將一個字元串轉成某型別的值。還記得，使用 `read` 函數時我們必須得用型別註釋註明想要的型別，否則 Haskell 就不會知道如何轉換。

```
ghci> read "Person {firstName =\"Michael\", lastName =\"Diamond\", age = 43}" :: Person
Person {firstName = "Michael", lastName = "Diamond", age = 43}
```

如果我們 `read` 的結果會在後面用到參與計算，Haskell 就可以推導出是一個 `Person` 的行為，不加註釋也是可以的。

```
ghci> read "Person {firstName =\"Michael\", lastName =\"Diamond\", age = 43}" == mikeD
True
```

也可以 `read` 帶參數的型別，但必須填滿所有的參數。因此 `read "Just 't'" :: Maybe a` 是不可以的，`read "Just 't'" :: Maybe Char` 才對。

很容易想象 `Ord` 型別類 `derive instance` 的行為。首先，判斷兩個值構造子是否一致，如果是，再判斷它們的參數，前提是它們的參數都得是 `Ord` 的 `instance`。`Bool` 型別可以有兩種值，`False` 和 `True`。為了瞭解在比較中程序的行為，我們可以這樣想像：

```
data Bool = False | True deriving (Ord)
```

由於值構造子 `False` 安排在 `True` 的前面，我們可以認為 `True` 比 `False` 大。

```
ghci> True `compare` False
GT
ghci> True > False
True
ghci> True < False
False
```

在 `Maybe a` 數據型別中，值構造子 `Nothing` 在 `Just` 值構造子前面，所以一個 `Nothing` 總要比 `Just something` 的值小。即便這個 `something` 是 `-1000000000` 也是如此。

```
ghci> Nothing < Just 100
True
ghci> Nothing > Just (-49999)
False
ghci> Just 3 `compare` Just 2
GT
ghci> Just 100 > Just 50
True
```

不過類似 `Just (*3) > Just (*2)` 之類的程式碼是不可以的。因為 `(*3)` 和 `(*2)` 都是函數，而函數不是 `Ord` 類的成員。

作枚舉，使用數字型別就能輕易做到。不過使用 `Enum` 和 `Bounded` 型別類會更好，看下這個型別：

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

所有的值構造子都是 `nullary` 的(也就是沒有參數)，每個東西都有前置子和後繼子，我們可以讓它成為 `Enum` 型別類的成員。同樣，每個東西都有可能的最小值和最大值，我們也可以讓它成為 `Bounded` 型別類的成員。在這裡，我們就同時將它搞成其它可 `derive` 型別類的 `instance`。再看看我們能拿它做啥：

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
         deriving (Eq, Ord, Show, Read, Bounded, Enum)
```

由於它是 `Show` 和 `Read` 型別類的成員，我們可以將這個型別的值與字元串相互轉換。

```
ghci> Wednesday
Wednesday
ghci> show Wednesday
"Wednesday"
ghci> read "Saturday" :: Day
Saturday
```

由於它是 `Eq` 與 `Ord` 的成員，因此我們可以拿 `Day` 作比較。

```
ghci> Saturday == Sunday
False
ghci> Saturday == Saturday
True
ghci> Saturday > Friday
True
ghci> Monday `compare` Wednesday
LT
```

它也是 `Bounded` 的成員，因此有最早和最晚的一天。

```
ghci> minBound :: Day
Monday
ghci> maxBound :: Day
Sunday
```

它也是 `Enum` 的 `instance`，可以得到前一天和後一天，並且可以對此使用 List 的區間。

```
ghci> succ Monday
Tuesday
ghci> pred Saturday
Friday
ghci> [Thursday .. Sunday]
[Thursday, Friday, Saturday, Sunday]
ghci> [minBound .. maxBound] :: [Day]
[Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday]
```

那是相當的棒。

## Type synonyms

在前面我們提到在寫型別名的時候，`[Char]` 和 `String` 等價，可以互換。這就是由型別別名實現的。型別別名實際上什麼也沒做，只是給型別提供了不同的名字，讓我們的程式碼更容易理解。這就是 `[Char]` 的別名 `String` 的由來。

```
type String = [Char]
```

我們已經介紹過了 `type` 關鍵字，這個關鍵字有一定誤導性，它並不是用來創造新類（這是 `data` 關鍵字做的事情），而是給一個既有型別提供一個別名。

如果我們隨便搞個函數 `toUpperString` 或其他什麼名字，將一個字元串變成大寫，可以用這樣的型別聲明 `toUpperString :: [Char] -> [Char]`，也可以這樣 `toUpperString :: String -> String`，二者在本質上是完全相同的。後者要更易讀些。

在前面 `Data.Map` 那部分，我們用了一個關聯 `List` 來表示 `phoneBook`，之後才改成的 `Map`。我們已經發現了，一個關聯 `List` 就是一組鍵值對組成的 `List`。再看下我們 `phoneBook` 的樣子：

```
phoneBook :: [(String, String)]
phoneBook =
  [("betty", "555-2938")
  , ("bonnie", "452-2928")
  , ("patsy", "493-2928")
  , ("lucille", "205-2928")
  , ("wendy", "939-8282")
  , ("penny", "853-2492")
  ]
```

可以看出，`phoneBook` 的型別就是 `[(String, String)]`，這表示一個關聯 `List` 僅是 `String` 到 `String` 的映射關係。我們就弄個型別別名，好讓它型別聲明中能夠表達更多資訊。

```
type PhoneBook = [(String, String)]
```

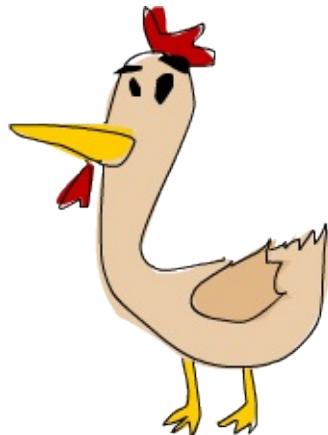
現在我們 `phoneBook` 的型別聲明就可以是 `phoneBook :: PhoneBook` 了。再給字元串加上別名：

```
type PhoneNumber = String
type Name = String
type PhoneBook = [(Name, PhoneNumber)]
```

Haskell 程序員給 `String` 加別名是為了讓函數中字元串的表達方式及用途更加明確。

好的，我們實現了一個函數，它可以取一名字和號碼檢查它是否存在於電話本。現在可以給它加一個相當好看明了的型別聲明：

```
inPhoneBook :: Name -> PhoneNumber -> PhoneBook -> Bool
inPhoneBook name pnumber pbook = (name, pnumber) `elem` pbook
```



如果不用型別別名，我們函數的型別聲明就只能是 `String -> String -> [(String, String)] -> Bool` 了。在這裡使用型別別名是為了讓型別聲明更加易讀，但你也不必拘泥于它。引入型別別名的動機既非單純表示我們函數中的既有型別，也不是為了替換掉那些重複率高的長名字型別(如 `[(String, String)]`)，而是為了讓型別對事物的描述更加明確。

型別別名也是可以有參數的，如果你想搞個型別來表示關聯 List，但依然要它保持通用，好讓它可以使用任意型別作 `key` 和 `value`，我們可以這樣：

```
type AssocList k v = [(k, v)]
```

好的，現在一個從關聯 List 中按鍵索值的函數型別可以定義為 `(Eq k) => k -> AssocList k v -> Maybe v. AssocList i`。`AssocList` 是個取兩個型別做參數生成一個具體型別的型別構造子，如 `Assoc Int String` 等等。

\*Fronzie 說\* : Hey ! 當我提到具體型別，那我就是說它是完全呼叫的，就像 ``Map Int String``。要不就是多態[

我們可以用不全呼叫來得到新的函數，同樣也可以使用不全呼叫得到新的型別構造子。同函數一樣，用不全的型別參數呼叫型別構造子就可以得到一個不全呼叫的型別構造子，如果我們要一個表示從整數到某東西間映射關係的型別，我們可以這樣：

```
type IntMap v = Map Int v
```

也可以這樣：

```
type IntMap = Map Int
```

無論怎樣，`IntMap` 的型別構造子都是取一個參數，而它就是這整數指向的型別。

Oh yeah, 如果要你去實現它，很可能會用個 `qualified import` 來導入 `Data.Map`。這時，型別構造子前面必須得加上模組名。所以應該寫個 `type IntMap = Map.Map Int`

你得保證真正弄明白了型別構造子和值構造子的區別。我們有了個叫 `IntMap` 或者 `AssocList` 的別名並不意味着我們可以執行類似 `AssocList [(1,2),(4,5),(7,9)]` 的程式碼，而是可以用不同的名字來表示原先的 List，就像 `[(1,2),(4,5),(7,9)] :: AssocList Int Int` 讓它裡面的型別都是 `Int`。而像處理普通的 Tuple 構成的那種 List 處理它也是可以的。型別別名(型別依然不變)，只可以在 Haskell 的型別部分中使用，像定義新型別或型別聲明或型別註釋中跟在::後面的部分。

另一個很酷的二參型別就是 `Either a b` 了，它大約是這樣定義的：

```
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)
```

它有兩個值構造子。如果用了 `Left`，那它內容的型別就是 `a`；用了 `Right`，那它內容的型別就是 `b`。我們可以用它來將可能是兩種型別的值封裝起來，從裡面取值時就同時提供 `Left` 和 `Right` 的模式匹配。

```
ghci> Right 20
Right 20
ghci> Left "woot"
Left "woot"
ghci> :t Right 'a'
Right 'a' :: Either a Char
ghci> :t Left True
Left True :: Either Bool b
```

到現在為止，`Maybe` 是最常見的表示可能失敗的計算的型別了。但有時 `Maybe` 也並不是十分的好用，因為 `Nothing` 中包含的信息還是太少。要是我們不關心函數失敗的原因，它還是不錯的。就像 `Data.Map` 的 `lookup` 只有在搜尋的項不在 `Map` 時才會失敗，對此我們一清二楚。但我們若想知道函數失敗的原因，那還得使用 `Either a b`，用 `a` 來表示可能的錯誤的型別，用 `b` 來表示一個成功運算的型別。從現在開始，錯誤一律用 `Left` 值構造子，而結果一律用 `Right`。

一個例子：有個學校提供了不少壁櫥，好給學生們地方放他們的 Gun'N'Rose 海報。每個壁櫥都有個密碼，哪個學生想用個壁櫥，就告訴管理員壁櫥的號碼，管理員就會告訴他壁櫥的密碼。但如果這個壁櫥已經讓別人用了，管理員就不能告訴他密碼了，得換一個壁櫥。我們就用 `Data.Map` 的一個 `Map` 來表示這些壁櫥，把一個號碼映射到一個表示壁櫥占用情況及密碼的 Tuple 裡。

```

import qualified Data.Map as Map

data LockerState = Taken | Free deriving (Show, Eq)

type Code = String

type LockerMap = Map.Map Int (LockerState, Code)

```

很簡單，我們引入了一個新的型別來表示壁櫥的占用情況。併為壁櫥密碼及按號碼找壁櫥的 Map 分別設置了一個別名。好，現在我們實現這個按號碼找壁櫥的函數，就用 Either String Code 型別表示我們的結果，因為 lookup 可能會以兩種原因失敗。櫥子已經讓別人用了或者壓根就沒有這個櫥子。如果 lookup 失敗，就用字元串表明失敗的原因。

```

lockerLookup :: Int -> LockerMap -> Either String Code
lockerLookup lockerNumber map =
  case Map.lookup lockerNumber map of
    Nothing -> Left $ "Locker number " ++ show lockerNumber ++ " doesn't exist!"
    Just (state, code) -> if state /= Taken
      then Right code
      else Left $ "Locker " ++ show lockerNumber ++ " is already"

```

我們在這裡個 Map 中執行一次普通的 lookup，如果得到一個 Nothing，就返回一個 Left String 的值，告訴他壓根就沒這個號碼的櫥子。如果找到了，就再檢查下，看這櫥子是不是已經讓別人用了，如果是，就返回個 Left String 說它已經讓別人用了。否則就返回個 Right Code 的值，通過它來告訴學生壁櫥的密碼。它實際上就是個 Right String，我們引入了個型別別名讓它這型別聲明更好看。

如下是個 Map 的例子：

```

lockers :: LockerMap
lockers = Map.fromList
  [(100, (Taken, "ZD39I"))
  ,(101, (Free, "JAH3I"))
  ,(103, (Free, "IQSA9"))
  ,(105, (Free, "QOTSA"))
  ,(109, (Taken, "893JJ"))
  ,(110, (Taken, "99292"))]

```

現在從裡面 lookup 某個櫥子號..

```
ghci> lockerLookup 101 lockers
Right "JAH3I"
ghci> lockerLookup 100 lockers
Left "Locker 100 is already taken!"
ghci> lockerLookup 102 lockers
Left "Locker number 102 doesn't exist!"
ghci> lockerLookup 110 lockers
Left "Locker 110 is already taken!"
ghci> lockerLookup 105 lockers
Right "QOTSA"
```

我們完全可以用 `Maybe a` 來表示它的結果，但這樣一來我們就對得不到密碼的原因不得而知了。而在這裡，我們的新型別可以告訴我們失敗的原因。

## Recursive data structures (遞迴地定義資料結構)

如我們先前看到的，一個 algebraic data type 的構造子可以有好幾個 field，其中每個 field 都必須有具體的型態。有了那個概念，我們能定義一個型態，其中他的構造子的 field 的型態是他自己。這樣我們可以遞迴地定義下去，某個型態的值便可能包含同樣型態的值，進一步下去他還可以再包含同樣型態的值。

考慮一下 `List: [5]`。他其實是 `5:[]` 的語法糖。在 `:` 的左邊是一個普通值，而在右邊是一串 List。只是在這個案例中是空的 List。再考慮 `[4,5]`。他可以看作 `4:(5:[])`。看看第一個 `:`，我們看到他也有一個元素在左邊，一串 List `5:[]` 在右邊。同樣的道理 `3:(4:(5:6:[]))` 也是這樣。

我們可以說一個 List 的定義是要碼是空的 List 或是一個元素，後面用 `:` 接了另一串 List。

我們用 algebraic data type 來實作我們自己的 List！

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
```

這讀起來好像我們前一段提及的定義。他要碼是空的 List，或是一個元素跟一串 List 的結合。如果你被搞混了，看看用 record syntax 定義的可能比較清楚。

```
data List a = Empty | Cons { listHead :: a, listTail :: List a} deriving (Show, Read, Eq,
```

你可能也對這邊的 `cons` 構造子不太清楚。`cons` 其實就是指 `:`。對 List 而言，`:` 其實是一個構造子，他接受一個值跟另一串 List 來構造一個 List。現在我們可以使用我們新定義的 List 型態。換句話說，他有兩個 field，其中一個 field 具有型態 `a`，另一個有型態 `[a]`。

```
ghci> Empty
Empty
ghci> 5 `Cons` Empty
Cons 5 Empty
ghci> 4 `Cons` (5 `Cons` Empty)
Cons 4 (Cons 5 Empty)
ghci> 3 `Cons` (4 `Cons` (5 `Cons` Empty))
Cons 3 (Cons 4 (Cons 5 Empty))
```

我們用中綴的方式呼叫 `Cons` 構造子，這樣你可以很清楚地看到他就是 `:`。`Empty` 代表 `[]`，而 `4 `Cons` (5 `Cons` Empty)` 就是 `4:(5:[])`。

我們可以只用特殊字元來定義函數，這樣他們就會自動具有中綴的性質。我們也能同樣的手法套用在構造子上，畢竟他們不過是回傳型態的函數而已。

```
infixr 5 :-:
data List a = Empty | a :-: (List a) deriving (Show, Read, Eq, Ord)
```

首先我們留意新的語法結構：`fixity` 告訴。當我們定義函數成 operator，我們能同時指定 fixity (但並不是必須的)。`fixity` 指定了他應該是 left-associative 或是 right-associative，還有他的優先順序。例如說，`*` 的 fixity 是 `infixl 7 *`，而 `+` 的 fixity 是 `infixl 6`。代表他們都是 left-associative。`(4 * 3 * 2)` 等於 `((4 * 3) * 2)`。但 `*` 擁有比 `+` 更高的優先順序。所以 `5 * 4 + 3` 會是 `(5 * 4) + 3`。

這樣我們就可以寫成 `a :-: (List a)` 而不是 `Cons a (List a)`：

```
ghci> 3 :-: 4 :-: 5 :-: Empty
(:-:) 3 ((:-:) 4 ((:-:) 5 Empty))
ghci> let a = 3 :-: 4 :-: 5 :-: Empty
ghci> 100 :-: a
(:-:) 100 ((:-:) 3 ((:-:) 4 ((:-:) 5 Empty)))
```

Haskell 在宣告 `deriving Show` 的時候，他會仍視構造子為前綴函數，因此必須要用括號括起來。

我們在來寫個函數來把兩個 List 連起來。一般 `++` 在操作普通 List 的時候是這樣的：

```
infixr 5 ++
(++) :: [a] -> [a] -> [a]
[]      ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

我們把他偷過來用在我們的 List 上，把函數命名成 `.++`：

```
infixr 5 .++
(.++) :: List a -> List a -> List a
Empty .++ ys = ys
(x :+: xs) .++ ys = x :+: (xs .++ ys)
```

來看看他如何運作：

```
ghci> let a = 3 :+: 4 :+: 5 :+: Empty
ghci> let b = 6 :+: 7 :+: Empty
ghci> a .++ b
(:-:) 3 ((:-:) 4 ((:-:) 5 ((:-:) 6 ((:-:) 7 Empty))))
```

如果我們想要的話，我們可以定義其他操作我們list的函數。

注意到我們是如何利用 `(x :+: xs)` 做模式匹配的。他運作的原理實際上就是利用到構造子。我們可以利用 `:+:` 做模式匹配原因就是他是構造子，同樣的 `:` 也是構造子所以可以用他做匹配。`[]` 也是同樣道理。由於模式匹配是用構造子來作的，所以我們才能對像 `8`，`'a'` 之類的做模式匹配。他們是數值與字元的構造子。

接下來我們要實作二元搜尋樹 (binary search tree)。如果你對二元搜尋樹不太清楚，我們來快速地解釋一遍。他的結構是每個節點指向兩個其他節點，一個在左邊一個在右邊。在左邊節點的元素會比這個節點的元素要小。在右邊的話則比較大。每個節點最多可以有兩棵子樹。而我們知道譬如說一棵包含 5 的節點的左子樹，裡面所有的元素都會小於 5。而節點的右子樹裡面的元素都會大於 5。如果我們想找找看 8 是不是在我們的樹裡面，我們就從 5 那個節點找起，由於 8 比 5 要大，很自然地就會往右搜尋。接著我們走到 7，又由於 8 比 7 要大，所以我們再往右走。我們在三步就找到了我們要的元素。如果這不是棵樹而是 List 的話，那就會需要花到七步才能找到 8。

`Data.Set` 跟 `Data.Map` 中的 `set` 和 `Map` 都是用樹來實現的，只是他們是用平衡二元搜尋樹而不是隨意的二元搜尋樹。不過這邊我們就只先寫一棵普通的二元搜尋樹就好了。

這邊我們來定義一棵樹的結構：他不是一棵空的樹就是帶有值並含有兩棵子樹。聽起來非常符合 algebraic data type 的結構！

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a) deriving (Show, Read, Eq)
```

我們不太想手動來建棵二元搜尋樹，所以我們要來寫一個函數，他接受一棵樹還有一個元素，把這個元素安插到這棵二元搜尋樹中。當拿這個元素跟樹的節點比較結果比較小的話，我們就往左走，如果比較大，就往右走。重複這個動作直到我們走到一棵空的樹。一旦碰到空的樹的話，我們就把元素插入節點。

在 C 語言中，我們是用修改指標的方式來達成這件事。但在 Haskell 中，我們沒辦法修改我們的樹。所以我們在決定要往左或往右走的時候就做一棵新的子樹，走到最後要安插節點的時候也是做一棵新的樹。因此我們插入函數的型態會是 `a -> Tree a -> Tree a`。他接受一個元素跟一棵樹，並回傳一棵包含了新元素的新的樹。這看起來很沒效率的樣子，但別擔心，惰性的特性可以讓我們不用擔心這個。

來看下列兩個函數。第一個做了一個單節點的樹，而第二個插入一個元素到一棵樹中。

```
singleton :: a -> Tree a
singleton x = Node x EmptyTree EmptyTree

treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node a left right)
| x == a = Node x left right
| x < a = Node a (treeInsert x left) right
| x > a = Node a left (treeInsert x right)
```

`singleton` 函數只是一個做一個含有兩棵空子樹的節點的函數的別名。在插入的操作中，我們先為終端條件定義了一個模式匹配。如果我們走到了一棵空的子樹，這表示我們到達了我們想要的地方，我們便建造一棵空的單元素的樹來放在那個位置。如果我們還沒走到一棵空的樹來插入我們的元素。那就必須要做一些檢查來往下走。如果我們要安插的元素跟 root 所含有的元素相等，那就直接回傳這棵樹。如果安插的元素比較小，就回傳一棵新的樹。這棵樹的 root 跟原來的相同，右子樹也相同，只差在我們要安插新的元素到左子樹中。如果安插的元素反而比較大，那整個過程就相反。

接下來，我們要寫一個函數來檢查某個元素是否已經在這棵樹中。首先我們定義終端條件。如果我們已經走到一棵空的樹，那這個元素一定不在這棵樹中。這跟我們搜尋 List 的情形是一致的。如果我們要在空的 List 中搜尋某一元素，那就代表他不在這個 List 裡面。假設我們現在搜尋一棵非空的樹，而且 root 中的元素剛好就是我們要的，那就找到了。那如果不是呢？我們就要利用在 root 節點左邊的元素都比 root 小的這個性質。如果我們的元素比 root 小，那就往左子樹中找。如果比較大，那就往右子樹中找。

```
treeElem :: (Ord a) => a -> Tree a -> Bool
treeElem x EmptyTree = False
treeElem x (Node a left right)
| x == a = True
| x < a = treeElem x left
| x > a = treeElem x right
```

我們要作的就是把之前段落所描述的事轉換成程式碼。首先我們不想手動一個個來創造一棵樹。我們想用一個 `fold` 來從一個 List 創造一棵樹。要知道走遍一個 List 並回傳某種值的操作都可以用 `fold` 來實現。我們先從一棵空的樹開始，然後從右邊走過 List 的每一個元素，一個一個丟到樹裡面。

```
ghci> let nums = [8,6,4,1,7,3,5]
ghci> let numsTree = foldr treeInsert EmptyTree nums
ghci> numsTree
Node 5 (Node 3 (Node 1 EmptyTree EmptyTree) (Node 4 EmptyTree EmptyTree)) (Node 7 (Node 6
```

在 `foldr` 中，`treeInsert` 是做 folding 操作的函數，而 `EmptyTree` 是起始的 accumulator，`nums` 則是要被走遍的 List。

當我們想把我們的樹印出來的時候，印出來的形式會不太容易讀。但如果我們能有結構地印出來呢？我們知道 root 是 5，他有兩棵子樹，其中一個的 root 是 3 另一個則是 7。

```
ghci> 8 `treeElem` numsTree
True
ghci> 100 `treeElem` numsTree
False
ghci> 1 `treeElem` numsTree
True
ghci> 10 `treeElem` numsTree
False
```

檢查元素是否屬於某棵樹的函數現在能正常運作了！

你可以看到 algebraic data structures 是非常有力的概念。我們可以使用這個結構來構造出布林值，週一到週五的概念，甚至還有二元樹。

## Typeclasses 的第二堂課

到目前為止我們學到了一些 Haskell 中的標準 typeclass，也學到了某些已經定義為他們 instance 的型別。我們知道如何讓我們自己定義的型別自動被 Haskell 所推導成標準 typeclass 的 instance。在這個章節中，我們會學到如何構造我們自己的 typeclass，並且如何構造這些 typeclass 的 type instance。

來快速複習一下什麼是 typeclass: typeclass 就像是 interface。一個 typeclass 定義了一些行為(像是比較相不相等，比較大小順序，能否窮舉)而我們會把希望滿足這些性質的型別定義成這些 typeclass 的 instance。typeclass 的行為是由定義的函數來描述。並寫出對應的實作。當我們把一個型別定義成某個 typeclass 的 instance，就表示我們可以對那個型別使用 typeclass 中定義的函數。

Typeclass 跟 Java 或 Python 中的 class 一點關係也沒有。這個概念讓很多人混淆，所以我希望你先忘掉所有在命令式語言中學到有關 class 的所有東西。

例如說，`Eq` 這個 typeclass 是描述可以比較相等的事物。他定義了 `==` 跟 `/=` 兩個函數。如果我們有一個型別 `car`，而且對他們做相等比較是有意義的，那把 `car` 作成是 `Eq` 的一個 instance 是非常合理的。

這邊來看看在 `Prelude` 之中 `Eq` 是怎麼被定義的。

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)
```

我們在這邊看到了一些奇怪的語法跟關鍵字。別擔心，你一下子就會瞭解他們的。首先，我們看到 `class Eq a where`，那代表我們定義了一個新的 typeclass 叫做 `Eq`。`a` 是一個型別變數，他代表 `a` 是任何我們在定義 instance 時的型別。他不一定要叫做 `a`。他也不一定非要一個字母不可，只要他是小寫就好。然後我們又定義了幾個函數。我們並不一定要實作函數的本體，不過必須要寫出函數的型別宣告。

如果我們寫成 `class Eq equatable where` 還有 `(==) :: equatable -> equatable -> Bool` 這樣的形式，對一些人可能比較容易理解。

總之我們實作了 `Eq` 中需要定義的函數本體，只是我們定義他的方式是用交互遞迴的形式。我們描述兩個 `Eq` 的 instance 要相等，那他們就不能不一樣，而他們如果不一樣，那他們就是不相等。我們其實不必這樣寫，但很快你會看到這其實是有用的。

如果我們說 `class Eq a where` 然後定義 `(==) :: a -> a -> Bool`，那我們之後檢查函數的型別時會發現他的型別是 `(Eq a) -> a -> a -> Bool`。

當我們有了 `class` 以後，可以用來做些什麼呢？說實話，不多。不過一旦我們為它寫一些 instance，就會有些好功能。來看看下面這個型別：

```
data TrafficLight = Red | Yellow | Green
```

這裡定義了紅綠燈的狀態。請注意這個型別並不是任何 `class` 的 instance，雖然可以透過 `derive` 讓它成為 `Eq` 或 `Show` 的 instance，但我們打算手工打造。下面展示了如何讓一個型別成為 `Eq` 的 instance：

```
instance Eq TrafficLight where
  Red == Red = True
  Green == Green = True
  Yellow == Yellow = True
  _ == _ = False
```

我們使用了 `instance` 這個關鍵字。`class` 是用來定義新的 typeclass，而 `instance` 是用來說明我們要定義某個 typeclass 的 `instance`。當我們要定義 `Eq`，我們會寫 `class Eq a where`，其中 `a` 代表任何型態。我們可以從 `instance` 的寫法：`instance Eq TrafficLight where` 看出來。我們會把 `a` 換成實際的型別。

由於 `==` 是用 `/=` 來定義的，同樣的 `/=` 也是用 `==` 來定義。所以我們只需要在 `instance` 定義中複寫其中一個就好了。我們這樣叫做定義了一個 minimal complete definition。這是說能讓型別符合 `class` 行為所最小需要實作的函數數量。而 `Eq` 的 minimal complete definition 需要 `==` 或 `/=` 其中一個。而如果 `Eq` 是這樣定義的：

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

當我們定義 `instance` 的時候必須要兩個函數都實作，因為 Haskell 並不知道這兩個函數是怎麼關聯在一起的。所以 minimal complete definition 在這邊是 `==` 跟 `/=`。

你可以看到我們是用模式匹配來實作 `==`。由於不相等的情況比較多，所以我們只寫出相等的，最後再用一個 `case` 接住說你不在前面相等的 `case` 的話，那就是不相等。

我們再來寫 `show` 的 `instance`。要滿足 `show` 的 minimal complete definition，我們必須實作 `show` 函數，他接受一個值並把他轉成字串。

```
instance Show TrafficLight where
  show Red = "Red light"
  show Yellow = "Yellow light"
  show Green = "Green light"
```

再一次地，我們用模式匹配來完成我們的任務。我們來看看他是如何運作的。

```
ghci> Red == Red
True
ghci> Red == Yellow
False
ghci> Red `elem` [Red, Yellow, Green]
True
ghci> [Red, Yellow, Green]
[Red light,Yellow light,Green light]
```

如果我們用 `derive` 來自動產生 `Eq` 的話，效果是一樣的。不過用 `derive` 來產生 `show` 的話，他會把值構造子轉換成字串。但我們這邊要的不太一樣，我們希望印出像 `"Red light"` 這樣的字串，所以我們就必須手動來寫出 `instance`。

你也可以把 typeclass 定義成其他 typeclass 的 subclass。像是 `Num` 的 `class` 告訴就有點冗長，但我們先看個雛型。

```
class (Eq a) => Num a where
  ...
```

正如我們先前提到的，我們可以在很多地方加上 class constraints。這不過就是在 `class Num a where` 中的 `a` 上，加上他必須要是 `Eq` 的 instance 的限制。這基本上就是在說我們在定義一個型別為 `Num` 之前，必須先為他定義 `Eq` 的 instance。在某個型別可以被視作 `Number` 之前，必須先能被比較相不相等其實是蠻合理的。這就是 subclass 在做的事：幫 class declaration 加上限制。也就是說當我們定義 typeclass 中的函數本體時，我們可以預設 `a` 是屬於 `Eq`，因此能使用 `==`。

但像是 `Maybe` 或是 `List` 是如何被定義成 typeclass 的 instance 呢？`Maybe` 的特別之處在於他跟 `TrafficLight` 不一樣，他不是一個具體的型別。他是一個型別構造子，接受一個型別參數（像是 `Char` 之類的）而構造出一個具體的型別（像是 `Maybe Char`）。讓我們再回顧一下 `Eq` 這個 typeclass：

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)
```

從型別宣告來看，可以看到 `a` 必須是一個具體型別，因為所有在函數中的型別都必須是具體型別。（你沒辦法寫一個函數，他的型別是 `a -> Maybe`，但你可以寫一個函數，他的型別是 `a -> Maybe a`，或是 `Maybe Int -> Maybe String`）這就是為什麼我們不能寫成像這樣：

```
instance Eq Maybe where
  ...
```

```
instance Eq (Maybe m) where
  Just x == Just y = x == y
  Nothing == Nothing = True
  _ == _ = False
```

這就好像在說我們要把 `Maybe something` 這種東西全部都做成 `Eq` 的 instance。我們的確可以寫成 `(Maybe something)`，但我們通常是只用一個字母，這樣比較像是 Haskell 的風格。`(Maybe m)` 這邊則取代了 `a` 在 `class Eq a where` 的位置。儘管 `Maybe` 不是一個具體的型別。`Maybe m` 却是。指定一個型別參數（在這邊是小寫的 `m`），我們說我們想要所有像是 `Maybe m` 的都成為 `Eq` 的 instance。

不過這仍然有一個問題。你能看出來嗎？我們用 `==` 來比較 `Maybe` 包含的東西，但我們並沒有任何保證說 `Maybe` 裝的東西可以是 `Eq`。這就是為什麼我們需要修改我們的 instance 定義：

```
instance (Eq m) => Eq (Maybe m) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

這邊我們必須要加上限制。在這個 `instance` 的宣告中，我們希望所有 `Maybe m` 形式的型別都屬於 `Eq`，但只有當 `m` 也屬於 `Eq` 的時候。這也是 Haskell 在 `derive` 的時候做的事。

在大部份情形下，在 `typeclass` 宣告中的 `class constraints` 都是要讓一個 `typeclass` 成為另一個 `typeclass` 的 `subclass`。而在 `instance` 宣告中的 `class constraint` 則是要表達型別的要求限制。舉裡來說，我們要求 `Maybe` 的內容物也要屬於 `Eq`。

當定義 `instance` 的時候，如果你需要提供具體型別（像是在 `a -> a -> Bool` 中的 `a`），那你必須要加上括號跟型別參數來構造一個具體型別。

要知道你在定義 `instance` 的時候，型別參數會被取代。`class Eq a where` 中的 `a` 會被取代成真實的型別。所以試著想像把型別放進型別宣告中。`(==) :: Maybe -> Maybe -> Bool` 並非合法。但 `(==) :: (Eq m) => Maybe m -> Maybe m -> Bool` 則是。這是不論我們要定義什麼，通用的型別宣告都是 `(==) :: (Eq a) => a -> a -> Bool`

還有一件事要確認。如果你想看看一個 `typeclass` 有定義哪些 `instance`。可以在 `ghci` 中輸入 `:info YourTypeClass`。所以輸入 `:info Num` 會告訴你這個 `typeclass` 定義了哪些函數，還有哪些型別屬於這個 `typeclass`。`:info` 也可以查詢型別跟型別構造子的資訊。如果你輸入 `:info Maybe`。他會顯示 `Maybe` 所屬的所有 `typeclass`。`:info` 也能告訴你函數的型別宣告。

## yes-no typeclass

在 Javascript 或是其他弱型別的程式語言，你能在 `if expression` 中擺上任何東西。舉例來說，你可以做像下列的事：`if (0) alert("YEAH!") else alert("NO!")`, `if ("") alert("YEAH!") else alert("NO!")`, `if (false) alert("YEAH") else alert("NO!)` 等等，而上述所有的片段執行後都會跳出 `NO!`。如果你寫 `if ("WHAT") alert ("YEAH") else alert("NO!")`，他會跳出 `YEAH!`，因為 Javascript 認為非空字串會是 `true`。

儘管使用 `Bool` 來表達布林的語意是比較好的作法。為了有趣起見，我們來試試看模仿 Javascript 的行為。我們先從 `typeclass` 宣告開始看：

```
class YesNo a where
    yesno :: a -> Bool
```

`YesNo` `typeclass` 定義了一個函數。這個函數接受一個可以判斷為真否的型別的值。而從我們寫 `a` 的位置，可以看出來 `a` 必須是一個具體型別。

接下來我們來定義一些 instance。對於數字，我們會假設任何非零的數字都會被當作 `true`，而 0 則當作 `false`。

```
instance YesNo Int where
    yesno 0 = False
    yesno _ = True
```

空的 List (包含字串)代表 `false`，而非空的 List 則代表 `true`。

```
instance YesNo [a] where
    yesno [] = False
    yesno _ = True
```

留意到我們加了一個型別參數 `a` 來讓整個 List 是一個具體型別，不過我們並沒有對包涵在 List 中的元素的型別做任何額外假設。我們還剩下 `Bool` 可以被作為真假值，要定義他們也很容易：

```
instance YesNo Bool where
    yesno = id
```

你說 `id` 是什麼？他不過是標準函式庫中的一個函數，他接受一個參數並回傳相同的東西。

我們也讓 `Maybe a` 成為 `YesNo` 的 instance。

```
instance YesNo (Maybe a) where
    yesno (Just _) = True
    yesno Nothing = False
```

由於我們不必對 `Maybe` 的內容做任何假設，因此並不需要 class constraint。我們只要定義遇到 `Just` 包裝過的值就代表 `true`，而 `Nothing` 則代表 `false`。這裡還是得寫出 `(Maybe a)` 而不是只有 `Maybe`，畢竟 `Maybe -> Bool` 的函式並不存在（因為 `Maybe` 並不是具體型別），而 `Maybe a -> Bool` 看起來就合理多了。現在有了這個定義，`Maybe something` 型式的型別都屬於 `YesNo` 了，不論 `something` 是什麼。

之前我們定義了 `Tree a`，那代表一個二元搜尋樹。我們可以說一棵空的樹是 `false`，而非空的樹則是 `true`。

```
instance YesNo (Tree a) where
    yesno EmptyTree = False
    yesno _ = True
```

而一個紅綠燈可以代表 yes or no 嗎？當然可以。如果他是紅燈，那你就會停下來，如果他是綠燈，那你就能走。但如果黃燈呢？只能說我通常會闖黃燈。

```
instance YesNo TrafficLight where
    yesno Red = False
    yesno _ = True
```

現在我們定義了許多 instance，來試著跑跑看！

```
ghci> yesno $ length []
False
ghci> yesno "haha"
True
ghci> yesno ""
False
ghci> yesno $ Just 0
True
ghci> yesno True
True
ghci> yesno EmptyTree
False
ghci> yesno []
False
ghci> yesno [0,0,0]
True
ghci> :t yesno
yesno :: (YesNo a) => a -> Bool
```

很好，統統是我們預期的結果。我們來寫一個函數來模仿 if statement 的行為，但他是運作在 YesNo 的型別上。

```
yesnoIf :: (YesNo y) => y -> a -> a -> a
yesnoIf yesnoVal yesResult noResult =
    if yesno yesnoVal then yesResult else noResult
```

很直覺吧！他接受一個 yes or no 的值還有兩個部份，如果值是代表 "yes"，那第一個部份就會被執行，而如果值是 "no"，那第二個部份就會執行。

```
ghci> yesnoIf [] "YEAH!" "NO!"
"NO!"
ghci> yesnoIf [2,3,4] "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf True "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf (Just 500) "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf Nothing "YEAH!" "NO!"
"NO!"
```

# Functor typeclass

到目前為止我們看過了許多在標準函式庫中的 typeclass。我們操作過 `Ord`，代表可以被排序的東西。我們也操作過 `Eq`，代表可以被比較相等性的事物。也看過 `Show`，代表可以被印成字串來表示的東西。至於 `Read` 則是我們可以把字串轉換成型別的動作。不過現在我們要來看一下 `Functor` 這個 typeclass，基本上就代表可以被 map over 的事物。聽到這個詞你可能會聯想到 `List`，因為 map over list 在 Haskell 中是很常見的操作。你沒想錯，`List` 的確是屬於 `Functor` 這個 typeclass。

來看看他的實作會是瞭解 `Functor` 的最佳方式：

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

我們看到他定義了一個函數 `fmap`，而且並沒有提供一個預設的實作。`fmap` 的型別蠻有趣的。到目前為止的我們看過的 typeclass 中的型別變數都是具體型別。就像是 `(==) :: (Eq a) => a -> a -> Bool` 中的 `a` 一樣。但現在碰到的 `f` 並不是一個具體型別（一個像是 `Int`, `Bool` 或 `Maybe String` 的型別），而是接受一個型別參數的型別構造子。如果要快速回顧的話可以看一下 `Maybe Int` 是一個具體型別，而 `Maybe` 是一個型別構造子，可接受一個型別作為參數。總之，我們知道 `fmap` 接受一個函數，這個函數從一個型別映射到另一個型別，還接受一個 functor 裝有原始的型別，然後會回傳一個 functor 裝有映射後的型別。

如果聽不太懂也沒關係。當我們看幾個範例之後會比較好懂。不過這邊 `fmap` 的型別宣告讓我們想起類似的東西，就是 `map :: (a -> b) -> [a] -> [b]`。

他接受一個函數，這函數把一個型別的東西映射成另一個。還有一串裝有某個型別的 `List` 變成裝有另一個型別的 `List`。到這邊聽起來實在太像 functor 了。實際上，`map` 就是針對 `List` 的 `fmap`。來看看 `List` 是如何被定義成 `Functor` 的 instance 的。

```
instance Functor [] where
    fmap = map
```

注意到我們不是寫成 `instance Functor [a] where`，因為從 `fmap :: (a -> b) -> f a -> f b` 可以知道 `f` 是一個型別構造子，他接受一個型別。而 `[a]` 則已經是一個具體型別（一個擁有某個型別的 `List`），其中 `[]` 是一個型別構造子，能接受某個型別而構造出像 `[Int]`、`[String]` 甚至是 `[[String]]` 的具體型別。

對於 `List`, `fmap` 只不過是 `map`，對 `List` 操作的時候他們都是一樣的。

```
map :: (a -> b) -> [a] -> [b]
ghci> fmap (*2) [1..3]
[2,4,6]
ghci> map (*2) [1..3]
[2,4,6]
```

至於當我們對空的 List 操作 `map` 或 `fmap` 呢？我們會得到一個空的 List。他把一個型別為 `[a]` 的空的 List 轉成型別為 `[b]` 的空的 List。

可以當作盒子的型別可能就是一個 functor。你可以把 List 想做是一個擁有無限小隔間的盒子。他們可能全部都是空的，已也可能有一部份是滿的其他是空的。所以作為一個盒子會具有什麼性質呢？例如說 `Maybe a`。他表現得像盒子在於他可能什麼東西都沒有，就是 `Nothing`，或是可以裝有一個東西，像是 `"HAHA"`，在這邊就是 `Just "HAHA"`。可以看到 `Maybe` 作為一個 functor 的定義：

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

注意到我們是寫 `instance Functor Maybe where` 而不是 `instance Functor (Maybe m) where`，就像我們在寫 `YesNo` 時的 `Maybe` 一樣。Functor 要的是一個接受一個型別參數的型別構造子而不是一個具體型別。如果你把 `f` 代換成 `Maybe`。`fmap` 就會像 `(a -> b) -> Maybe a -> Maybe b`。但如果你把 `f` 代換成 `(Maybe m)`，那他就會像 `(a -> b) -> Maybe m a -> Maybe m b`，這看起來並不合理，因為 `Maybe` 只接受一個型別參數。

總之，`fmap` 的實作是很簡單的。如果一個空值是 `Nothing`，那他就會回傳 `Nothing`。如果我們 `map over` 一個空的盒子，我們就會得到一個空的盒子。就像我們 `map over` 一個空的 List，那我們就會得到一個空的 List。如果他不是一個空值，而是包在 `Just` 中的某個值，那我們就會套用在包在 `Just` 中的值。

```
ghci> fmap (++) "HEY GUYS IM INSIDE THE JUST" (Just "Something serious.")
Just "Something serious. HEY GUYS IM INSIDE THE JUST"
ghci> fmap (++) "HEY GUYS IM INSIDE THE JUST" Nothing
Nothing
ghci> fmap (*2) (Just 200)
Just 400
ghci> fmap (*2) Nothing
Nothing
```

另外 `Tree a` 的型別也可以被 `map over` 且被定義成 `Functor` 的一個 `instance`。他可以被想成是一個盒子，而 `Tree` 的型別構造子也剛好接受單一個型別參數。如果你把 `fmap` 看作是一個特別為 `Tree` 寫的函數，他的型別宣告會長得像這樣 `(a -> b) -> Tree a -> Tree b`

b。不過我們在這邊會用到遞迴。map over 一棵空的樹會得到一棵空的樹。map over 一棵非空的樹會得到一棵被函數映射過的樹，他的 root 會先被映射，然後左右子樹都分別遞迴地被函數映射。

```
instance Functor Tree where
    fmap f EmptyTree = EmptyTree
    fmap f (Node x leftsub rightsub) =
        Node (f x) (fmap f leftsub) (fmap f rightsub)
```

```
ghci> fmap (*2) EmptyTree
EmptyTree
ghci> fmap (*4) (foldr treeInsert EmptyTree [5,7,3,2,1,7])
Node 28 (Node 4 EmptyTree (Node 8 EmptyTree (Node 12 EmptyTree (Node 20 EmptyTree EmptyTr
```

那 `Either a b` 又如何？他可以是一個 functor 嗎？`Functor` 限制型別構造子只能接受一個型別參數，但 `Either` 却接受兩個。聰明的你會想到我可以 partial apply `Either`，先餵給他一個參數，並把另一個參數當作 free parameter。來看看 `Either a` 在標準函式庫中是如何被定義的：

```
instance Functor (Either a) where
    fmap f (Right x) = Right (f x)
    fmap f (Left x) = Left x
```

我們在這邊做了些什麼？你可以看到我們把 `Either a` 定義成一個 instance，而不是 `Either`。那是因為 `Either a` 是一個接受單一型別參數的型別構造子，而 `Either` 則接受兩個。如果 `fmap` 是針對 `Either a`，那他的型別宣告就會像是  $(b \rightarrow c) \rightarrow Either a b \rightarrow Either a c$ ，他又等價於  $(b \rightarrow c) \rightarrow (Either a) b \rightarrow (Either a) c$ 。在實作中，我們碰到一個 `Right` 的時候會做 `map`，但在碰到 `Left` 的時候卻不這樣做，為什麼呢？如果我們回頭看看 `Either a b` 是怎麼定義的：

```
data Either a b = Left a | Right b
```

如果我們希望對他們兩個都做 `map` 的動作，那 `a` 跟 `b` 必須要是相同的型別。也就是說，如果我們的函數是接受一個字串然後回傳另一個字串，而且 `b` 是字串，`a` 是數字，這樣的情形是不可行的。而且從觀察 `fmap` 的型別也可以知道，當他運作在 `Either` 上的時候，第一個型別參數必須固定，而第二個則可以改變，而其中第一個參數正好就是 `Left` 用的。

我們持續用盒子的比喩也仍然貼切，我們可以把 `Left` 想做是空的盒子在他旁邊寫上錯誤訊息，說明為什麼他是空的。

在 `Data.Map` 中的 `Map` 也可以被定義成 functor，像是 `Map k v` 的情況下，`fmap` 可以用 `v -> v'` 這樣一個函數來 map over `Map k v`，並回傳 `Map k v'`。

注意到 ' 在這邊並沒有特別的意思，他只是用來表示他跟另一個東西有點像，只有一點點差別而已。

你可以自己試試看把 `Map k` 變成 `Functor` 的一個 instance。

看過了 `Functor` 這個 typeclass，我們知道 typeclass 可以拿來代表高階的概念。我們也練習過不少 partially applying type 跟定義 instance。在下幾章中，我們也會看到 functor 必須要遵守的定律。

還有一件事就是 functor 應該要遵守一些定律，這樣他們的一些性質才能被保證。如果我們用 `fmap (+1)` 來 map over `[1, 2, 3, 4]`，我們會期望結果會是 `[2, 3, 4, 5]` 而不是反過來變成 `[5, 4, 3, 2]`。如果我們使用 `fmap (\a -> a)` 來 map over 一個 list，我們會期待拿回相同的結果。例如說，如果我們給 `Tree` 定義了錯誤的 functor instance，對 tree 使用 `fmap` 可能會導致二元搜尋樹的性質喪失，也就是在 root 左邊的節點不再比 root 小，在 root 右邊的節點不再比 root 大。我們在下面幾章會多談 functor laws。

## Kind

型別構造子接受其他型別作為他的參數，來構造出一個具體型別。這樣的行為會讓我們想到函數，也是接受一個值當作參數，並回傳另一個值。我們也看過型別構造子可以 partially apply (`Either String` 是一個型別構造子，他接受一個型別來構造出一個具體型別，就像 `Either String Int`)。這些都是函數能辦到的事。在這個章節中，對於型別如何被套用到型別構造子上，我們會來看一下正式的定義。就像我們之前是用函數的型別來定義出函數是如何套用值的。如果你看不懂的話，你可以跳過這一章，這不會影響你後續的閱讀。然而如果你搞懂的話，你會對於型別系統有更進一步的了解。

像是 `3`, `"YEAH"` 或是 `takewhile` 的值他們都有自己的型別（函數也是值的一種，我們可以把他們傳來傳去）型別是一個標籤，值會把他帶著，這樣我們就可以推測出他的性質。但型別也有他們自己的標籤，叫做 kind。kind 是型別的型別。雖然聽起來有點玄妙，不過他的確是個有趣的概念。

那 kind 可以拿來做什麼呢？我們可以在 ghci 中用 `:k` 來得知一個型別的 kind。

```
ghci> :k Int
Int :: *
```

一個星星代表的是什麼意思？一個 \* 代表這個型別是具體型別。一個具體型別是沒有任何型別參數，而值只能屬於具體型別。而 \* 的讀法叫做 star 或是 type。

我們再看看 `Maybe` 的 kind：

```
ghci> :k Maybe
Maybe :: * -> *
```

`Maybe` 的型別構造子接受一個具體型別（像是 `Int`）然後回傳一個具體型別，像是 `Maybe Int`。這就是 `kind` 告訴我們的資訊。就像 `Int -> Int` 代表這個函數接受 `Int` 並回傳一個 `Int`。`* -> *` 代表這個型別構造子接受一個具體型別並回傳一個具體型別。我們再來對 `Maybe` 套用型別參數後再看看他的 `kind` 是什麼：

```
ghci> :k Maybe Int
Maybe Int :: *
```

正如我們預期的。我們對 `Maybe` 套用了型別參數後會得到一個具體型別（這就是 `* -> *` 的意思）這跟 `:t isUpper` 還有 `:t isUpper 'A'` 的差別有點類似。`isUpper` 的型別是 `Char -> Bool` 而 `isUpper 'A'` 的型別是 `Bool`。而這兩種型別，都是 `*` 的 `kind`。

我們對一個型別使用 `:k` 來得到他的 `kind`。就像我們對值使用 `:t` 來得到的他的型別一樣。就像我們先前說的，型別是值的標籤，而 `kind` 是型別的標籤。

我們再來看看其他的 `kind`

```
ghci> :k Either
Either :: * -> * -> *
```

這告訴我們 `Either` 接受兩個具體型別作為參數，並構造出一個具體型別。他看起來也像是一个接受兩個參數並回傳值的函數型別。型別構造子是可以做 `curry` 的，所以我們也能 `partially apply`。

```
ghci> :k Either String
Either String :: * -> *
ghci> :k Either String Int
Either String Int :: *
```

當我們希望定義 `Either` 成為 `Functor` 的 `instance` 的時候，我們必須先 `partial apply`，因為 `Functor` 預期有一個型別參數，但 `Either` 却有兩個。也就是說，`Functor` 希望型別的 `kind` 是 `* -> *`，而我們必須先 `partial apply` `Either` 來得到 `kind` `* -> *`，而不是最開始的 `* -> * -> *`。我們再來看看 `Functor` 的定義

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

我們看到 `f` 型別變數是接受一個具體型別且構造出一個具體型別的型別。知道他構造出具體型別是因為是作為函數參數的型別。從那裡我們可以推測出一個型別要是屬於 `Functor` 必須是 `* -> * Kind`。

現在我們來練習一下。來看看下面這個新定義的 typeclass。

```
class Tofu t where
    tofu :: j a -> t a j
```

這看起來很怪。我們幹嘛要為這個奇怪的 typeclass 定義 instance？我們可以來看看他的 kind 是什麼？由於 `j a` 被當作 `tofu` 這個函數的參數的型別，所以 `j a` 一定是 `* Kind`。我們假設 `a` 是 `* Kind`，那 `j` 就會是 `* -> *` 的 kind。我們看到 `t` 由於是函數的回傳值，一定是接受兩個型別參數的型別。而知道 `a` 是 `*`，`j` 是 `* -> *`，我們可以推測出 `t` 是 `* -> (* -> *) -> *`。也就是說他接受一個具體型別 `a`，一個接受單一參數的型別構造子 `j`，然後產生出一個具體型別。

我們再來定義出一個型別具有 `* -> (* -> *) -> *` 的 kind，下面是一種定義的方法：

```
data Frank a b = Frank {frankField :: b a} deriving (Show)
```

我們怎麼知道這個型別具有 `* -> (* -> *) -> *` 的 kind 呢？ADT 中的欄位是要來塞值的，所以他們必須是 `* Kind`。我們假設 `a` 是 `*`，那 `b` 就是接受一個型別參數的 kind `* -> *`。現在我們知道 `a` 跟 `b` 的 kind 了，而他們又是 `Frank` 的型別參數，所以我們知道 `Frank` 會有 `* -> (* -> *) -> *` 的 kind。第一個 `*` 代表 `a`，而 `(* -> *)` 代表 `b`。我們構造些 `Frank` 的值並檢查他們的型別吧：

```
ghci> :t Frank {frankField = Just "HAHA"}
Frank {frankField = Just "HAHA"} :: Frank [Char] Maybe
ghci> :t Frank {frankField = Node 'a' EmptyTree EmptyTree}
Frank {frankField = Node 'a' EmptyTree EmptyTree} :: Frank Char Tree
ghci> :t Frank {frankField = "YES"}
Frank {frankField = "YES"} :: Frank Char []
```

由於 `frankField` 具有 `a b` 的型別。他的值必定有一個類似的型別。他們可能是 `Just "HAHA"`，也就有 `Maybe [Char]` 的型別，或是他們可能是 `['Y', 'E', 'S']`，他的型別是 `[Char]`。（如果我們是用自己定義的 List 型別的話，那就會是 `List Char`）。我們看到 `Frank` 值的型別對應到 `Frank` 的 kind。`[Char]` 具有 `*` 的 kind，而 `Maybe` 則是 `* -> *`。由於結果必須是個值，也就是他必須要是具體型別，因使他必須 fully applied，因此每個 `Frank blah blaah` 的值都會是 `*` 的 kind。

要把 `Frank` 定義成 `Tofu` 的 instance 也是很簡單。我們看到 `tofu` 接受 `j a`（例如 `Maybe Int`）並回傳 `t a j`。所以我們將 `Frank` 代入 `t`，就得到 `Frank Int Maybe`。

```
instance Tofu Frank where
    tofu x = Frank x
```

```
ghci> tofu (Just 'a') :: Frank Char Maybe
Frank {frankField = Just 'a'}
ghci> tofu ["HELLO"] :: Frank [Char] []
Frank {frankField = ["HELLO"]}
```

這並不是很有用，但讓我們做了不少型別的練習。再來看看下面的型別：

```
data Barry t k p = Barry { yabba :: p, dabba :: t k }
```

我們想要把他定義成 `Functor` 的 `instance`。`Functor` 希望是  $* \rightarrow *$  的型別，但 `Barry` 並不是那種 `kind`。那 `Barry` 的 `kind` 是什麼呢？我們可以看到他接受三個型別參數，所以會是 `something -> something -> something -> *`。`p` 是一個具體型別因此是  $*$ 。至於 `k`，我們假設他是  $*$ ，所以 `t` 會是  $* \rightarrow *$ 。現在我們把這些代入 `something`，所以 `kind` 就變成  $(* \rightarrow *) \rightarrow * \rightarrow * \rightarrow *$ 。我們用 `ghci` 來檢查一下。

```
ghci> :k Barry
Barry :: (* -> *) -> * -> * -> *
```

我們猜對了！現在要把這個型別定義成 `Functor`，我們必須先 `partially apply` 頭兩個型別參數，這樣我們就會是  $* \rightarrow *$  的 `kind`。這代表 `instance` 定義會是 `instance Functor (Barry a b) where`。如果我們看 `fmap` 針對 `Barry` 的型別，也就是把 `f` 代換成 `Barry c d`，那就會是 `fmap :: (a -> b) -> Barry c d a -> Barry c d b`。第三個 `Barry` 的型別參數是對於任何型別，所以我們並不牽扯進他。

```
instance Functor (Barry a b) where
    fmap f (Barry {yabba = x, dabba = y}) = Barry {yabba = f x, dabba = y}
```

我們把 `f` `map` 到第一個欄位。

在這一個章節中，我們看到型別參數是怎麼運作的，以及正如我們用型別來定義出函數的參數，我們也用 `kind` 是來定義他。我們看到函數跟型別構造子有許多彼此相像的地方。然而他們是兩個完全不同的東西。當我們在寫一般實用的 Haskell 程式時，你幾乎不會碰到需要動到 `kind` 的東西，也不需要動腦去推敲 `kind`。通常你只需要在定義 `instance` 時 `partially apply` 你自己的  $* \rightarrow *$  或是  $*$  型別，但知道背後運作的原理也是很好的。知道型別本身也有自己的型別也是很有趣的。如果你實在不懂這邊講的東西，也可以繼續閱讀下去。但如果能理解，那你就會理解 Haskell 型別系統的一大部份。

# 輸入與輸出



我們已經說明了 Haskell 是一個純粹函數式語言。雖說在命令式語言中我們習慣給電腦執行一連串指令，在函數式語言中我們是用定義東西的方式進行。在 Haskell 中，一個函數不能改變狀態，像是改變一個變數的內容。（當一個函數會改變狀態，我們說這函數是有副作用的。）在 Haskell 中函數唯一可以做的事是根據我們給定的參數來算出結果。如果我們用同樣的參數呼叫兩次同一個函數，它會回傳相同的結果。儘管這從命令式語言的角度來看是蠻大的限制，我們已經看過它可以達成多麼酷的效果。在一個命令式語言中，程式語言沒辦法給你任何保證在一個簡單如打印出幾個數字的函數不會同時燒掉你的房子，綁架你的狗並刮傷你車子的烤漆。例如，當我們要建立一棵二元樹的時候，我們並不插入一個節點來改變原有的樹。由於我們無法改變狀態，我們的函數實際上回傳了一棵新的二元樹。

函數無法改變狀態的好處是它讓我們促進了我們理解程式的容易度，但同時也造成了一個問題。假如說一個函數無法改變現實世界的狀態，那它要如何打印出它所計算的結果？畢竟要告訴我們結果的話，它必須要改變輸出裝置的狀態（譬如說螢幕），然後從螢幕傳達到我們的腦，並改變我們心智的狀態。

不要太早下結論，Haskell 實際上設計了一個非常聰明的系統來處理有副作用的函數，它漂亮地將我們的程式區分成純粹跟非純粹兩部分。非純粹的部分負責跟鍵盤還有螢幕溝通。有了這區分的機制，在跟外界溝通的同時，我們還是能夠有效運用純粹所帶來的好處，像是惰性求值、容錯性跟模組性。

# Hello, world!



到目前為止我們都是將函數載入 GHCi 中來測試，像是標準函式庫中的一些函式。但現在我們要做些不一樣的，寫一個真實跟世界互動的 Haskell 程式。當然不例外，我們會來寫個 "hello world"。

現在，我們把下一行打到你熟悉的編輯器中

```
main = putStrLn "hello, world"
```

我們定義了一個 `main`，並在裡面以 `"hello, world"` 為參數呼叫了 `putStrLn`。看起來沒什麼大不了，但不久你就會發現它的奧妙。把這程式存成 `helloworld.hs`。

現在我們將做一件之前沒做過的事：編譯你的程式。打開你的終端並切換到包含 `helloworld.hs` 的目錄，並輸入下列指令。

```
$ ghc --make helloworld
[1 of 1] Compiling Main           ( helloworld.hs, helloworld.o )
Linking helloworld ...
```

順利的話你就會得到如上的訊息，接著你便可以執行你的程式 `./helloworld`

```
$ ./helloworld
hello, world
```

這就是我們第一個編譯成功並打印出字串到螢幕的程式。很簡單吧。

讓我們來看一下我們究竟做了些什麼，首先來看一下 `putStrLn` 函數的型態：

```
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t putStrLn "hello, world"
putStrLn "hello, world" :: IO ()
```

我們可以這麼解讀 `putStrLn` 的型態：`putStrLn` 接受一個字串並回傳一個 I/O action，這 I/O action 包含了 `()` 的型態。（即空的 tuple，或者是 unit 型態）。一個 I/O action 是一個會造成副作用的動作，常是指讀取輸入或輸出到螢幕，同時也代表會回傳某些值。在螢幕打印出幾個字串並沒有什麼有意義的回傳值可言，所以這邊用一個 `()` 來代表。

那究竟 I/O action 會在什麼時候被觸發呢？這就是 `main` 的功用所在。一個 I/O action 會在我們把它綁定到 `main` 這個名字並且執行程式的時候觸發。

把整個程式限制在只能有一個 I/O action 看似是個極大的限制。這就是為什麼我們需要 do 表示法來將所有 I/O action 繩成一個。來看看下面這個例子。

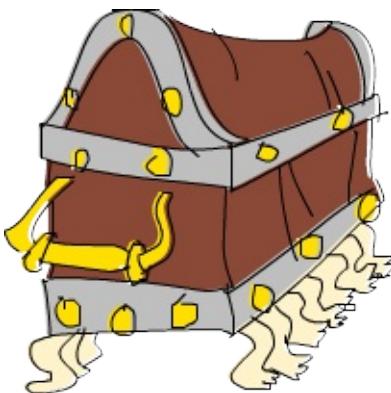
```
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

新的語法，有趣吧！它看起來就像一個命令式的程式。如果你編譯並執行它，它便會照你預期的方式執行。我們寫了一個 do 並且接著一連串指令，就像寫個命令式程式一般，每一步都是一個 I/O action。將所有 I/O action 用 do 繩在一起變成了一個大的 I/O action。這個大的 I/O action 的型態是 `IO ()`，這完全是由最後一個 I/O action 所決定的。

這就是為什麼 `main` 的型態永遠都是 `main :: IO something`，其中 `something` 是某個具體的型態。按照慣例，我們通常不會把 `main` 的型態在程式中寫出來。

另一個有趣的事情是第三行 `name <- getLine`。它看起來像是從輸入讀取一行並存到一個變數 `name` 之中。真的是這樣嗎？我們來看看 `getLine` 的型態吧

```
ghci> :t getLine
getLine :: IO String
```



我們可以看到 `getLine` 是一個回傳 `String` 的 I/O action。因為它會等使用者輸入某些字串，這很合理。那 `name <- getLine` 又是如何？你能這樣解讀它：執行一個 I/O action `getLine` 並將它的結果綁定到 `name` 這個名字。`getLine` 的型態是 `IO String`，所以 `name` 的型態會是 `String`。你能把 I/O action 想成是一個長了腳的盒子，它會跑到真實世界

中替你做某些事，像是在牆壁上塗鴉，然後帶回來某些資料。一旦它帶了某些資料給你，打開盒子的唯一辦法就是用 `<-`。而且如果我們要從 I/O action 拿出某些資料，就一定同時要在另一個 I/O action 中。這就是 Haskell 如何漂亮地分開純粹跟不純粹的程式的方法。`getLine` 在這樣的意義下是不純粹的，因為執行兩次的時候它沒辦法保證會回傳一樣的值。這也是為什麼它需要在一個 `IO` 的型態建構子中，那樣我們才能在 I/O action 中取出資料。而且任何一段程式一旦依賴著 I/O 資料的話，那段程式也會被視為 I/O code。

但這不表示我們不能在純粹的程式碼中使用 I/O action 回傳的資料。只要我們綁定它到一個名字，我們便可以暫時地使用它。像在 `name <- getLine` 中 `name` 不過是一個普通字串，代表在盒子中的內容。我們能將這個普通的字串傳給一個極度複雜的函數，並回傳你一生會有多少財富。像是這樣：

```
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn $ "Read this carefully, because this is your future: " ++ tellFortune name
```

`tellFortune` 並不知道任何 I/O 有關的事，它的型態只不過是 `String -> String`。

再來看看這段程式碼吧，他是合法的嗎？

```
nameTag = "Hello, my name is " ++ getLine
```

如果你回答不是，恭喜你。如果你說是，你答錯了。這麼做不對的理由是 `++` 要求兩個參數都必須是串列。他左邊的參數是 `String`，也就是 `[Char]`。然而 `getLine` 的型態是 `IO String`。你不能串接一個字串跟 I/O action。我們必須先把 `String` 的值從 I/O action 中取出，而唯一可行的方法就是在 I/O action 中使用 `name <- getLine`。如果我們需要處理一些非純粹的資料，那我們就要在非純粹的環境中做。所以我們最好把 I/O 的部分縮減到最小的比例。

每個 I/O action 都有一個值封裝在裡面。這也是為什麼我們之前的程式可以這麼寫：

```
main = do
    foo <- putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

然而，`foo` 只會有一個 `()` 的值，所以綁定到 `foo` 這個名字似乎是多餘的。另外注意到我們並沒有綁定最後一行的 `putStrLn` 給任何名字。那是因為在一個 do block 中，最後一個 action 不能綁定任何名字。我們在之後講解 Monad 的時候會說明為什麼。現在你可以先想成 do block 會自動從最後一個 action 取出值並綁定給他的結果。

除了最後一行之外，其他在 do 中沒有綁定名字的其實也可以寫成綁定的形式。所以 `putStrLn "BLAH"` 可以寫成 `_ <- putStrLn "BLAH"`。但這沒什麼實際的意義，所以我們寧願寫成 `putStrLn something`。

初學者有時候會想錯

```
name = getLine
```

以為這行會讀取輸入並給他綁定一個名字叫 `name` 但其實只是把 `getLine` 這個 I/O action 指定一個名字叫 `name` 罷了。記住，要從一個 I/O action 中取出值，你必須要在另一個 I/O action 中將他用 `<-` 綁定給一個名字。

I/O actions 只會在綁定給 `main` 的時候或是在另一個用 do 串起來的 I/O action 才會執行。你可以用 do 來串接 I/O actions，再用 do 來串接這些串接起來的 I/O actions。不過只有最外面的 I/O action 被指定給 `main` 才會觸發執行。

喔對，其實還有另外一個情況。就是在 GHCi 中輸入一個 I/O action 並按下 Enter 鍵，那也會被執行

```
ghci> putStrLn "HEEY"
HEEY
```

就算我們只是在 GHCi 中打幾個數字或是呼叫一個函數，按下 Enter 就會計算它並呼叫 `show`，再用 `putStrLn` 將字串打印出在終端上。

還記得 let binding 嗎？如果不記得，回去溫習一下這個章節。它們的形式是 `let bindings in expression`，其中 `bindings` 是 `expression` 中的名字、`expression` 則是被運用到這些名字的算式。我們也提到了 list comprehensions 中，`in` 的部份不是必需的。你能夠在 do blocks 中使用 let bindings 如同在 list comprehensions 中使用它們一樣，像這樣：

```
import Data.Char

main = do
    putStrLn "What's your first name?"
    firstName <- getLine
    putStrLn "What's your last name?"
    lastName <- getLine
    let bigFirstName = map toUpper firstName
        bigLastName = map toUpper lastName
    putStrLn $ "hey " ++ bigFirstName ++ " " ++ bigLastName ++ ", how are you?"
```

注意我們是怎麼編排在 do block 中的 I/O actions，也注意到我們是怎麼編排 let 跟其中的名字的，由於對齊在 Haskell 中並不會被無視，這麼編排才是好的習慣。我們的程式用 `map toUpper firstName` 將 `"John"` 轉成大寫的 `"JOHN"`，並將大寫的結果綁定到一個名字上，之

後在輸出的時候參考到了這個名字。

你也許會問究竟什麼時候要用 `<-`，什麼時候用 let bindings？記住，`<-` 是用來運算 I/O actions 並將他的結果綁定到名稱。而 `map toUpper firstName` 並不是一個 I/O action。他只是一個純粹的 expression。所以總結來說，當你要綁定 I/O actions 的結果時用 `<-`，而對於純粹的 expression 使用 let bindings。對於錯誤的 `let firstName = getLine`，我們只不過是把 `getLine` 這個 I/O actions 紕了一個不同的名字罷了。最後還是要用 `<-` 將結果取出。

現在我們來寫一個會一行一行不斷地讀取輸入，並將讀進來的字反過來輸出到螢幕上的程式。程式會在輸入空白行的時候停止。

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            putStrLn $ reverseWords line
            main

reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

在分析這段程式前，你可以執行看看來感受一下程式的運行。

首先，我們來看一下 `reverseWords`。他不過是一個普通的函數，假如接受了個字串 "hey there man"，他會先呼叫 `words` 來產生一個字的串列 ["hey", "there", "man"]。然後用 `reverse` 來 `map` 整個串列，得到 ["yeh", "ereht", "nam"]，接著用 `unwords` 來得到最終的結果 "yeh ereht nam"。這些用函數合成來簡潔的表達。如果沒有用函數合成，那就會寫成醜醜的樣子 `reverseWords st = unwords (map reverse (words st))`

那 `main` 又是怎麼一回事呢？首先，我們用 `getLine` 從終端讀取了一行，並把這行輸入取名叫 `line`。然後接著一個條件式 expression。記住，在 Haskell 中 if 永遠要伴隨一個 else，這樣每個 expression 才會有值。當 if 的條件是 true（也就是輸入了一個空白行），我們便執行一個 I/O action，如果 if 的條件是 false，那 else 底下的 I/O action 被執行。這也就是說當 if 在一個 I/O do block 中的時候，長的樣子是 `if condition then I/O action else I/O action`。

我們首先來看一下在 else 中發生了什麼事。由於我們在 else 中只能有一個 I/O action，所以我們用 do 來將兩個 I/O actions 綁成一個，你可以寫成這樣：

```
else (do
    putStrLn $ reverseWords line
    main)
```

這樣可以明顯看到整個 do block 可以看作一個 I/O action，只是比較醜。但總之，在 do block 裡面，我們依序呼叫了 `getLine` 以及 `reversewords`，在那之後，我們遞迴呼叫了 `main`。由於 `main` 也是一個 I/O action，所以這不會造成任何問題。呼叫 `main` 也就代表我們回到程式的起點。

那假如 `null line` 的結果是 true 呢？也就是說 then 的區塊被執行。我們看一下區塊裡面有 `then return ()`。如果你是從 C、Java 或 Python 過來的，你可能會認為 `return` 不過是作一樣的事情便跳過這一段。但很重要的：`return` 在 Haskell 裡面的意義跟其他語言的 `return` 完全不同！他們有相同的樣貌，造成了許多人搞錯，但確實他們是不一樣的。在命令式語言中，`return` 通常結束 method 或 subroutine 的執行，並且回傳某個值給呼叫者。在 Haskell 中，他的意義則是利用某個 pure value 造出 I/O action。用之前盒子的比喻來說，就是將一個 value 裝進箱子裡面。產生出的 I/O action 並沒有作任何事，只不過將 value 包起來而已。所以在 I/O 的情況下來說，`return "haha"` 的型態是 `IO String`。將 pure value 包成 I/O action 有什麼實質意義呢？為什麼要弄成 `IO` 包起來的值？這是因為我們一定要在 else 中擺上某些 I/O action，所以我們才用 `return ()` 做了一個沒作什麼事情的 I/O action。

在 I/O do block 中放一個 `return` 並不會結束執行。像下面這個程式會執行到底。

```
main = do
    return ()
    return "HAHAHA"
    line <- getLine
    return "BLAH BLAH BLAH"
    return 4
    putStrLn line
```

所有在程式中的 `return` 都是將 value 包成 I/O actions，而且由於我們沒有將他們綁定名稱，所以這些結果都被忽略。我們能用 `<-` 與 `return` 來達到綁定名稱的目的。

```
main = do
    a <- return "hell"
    b <- return "yeah!"
    putStrLn $ a ++ " " ++ b
```

可以看到 `return` 與 `<-` 作用相反。`return` 把 value 裝進盒子中，而 `<-` 將 value 從盒子拿出來，並綁定一個名稱。不過這麼做是有些多餘，因為你可以用 let bindings 來綁定

```
main = do
    let a = "hell"
        b = "yeah"
    putStrLn $ a ++ " " ++ b
```

在 I/O do block 中需要 `return` 的原因大致上有兩個：一個是我們需要一個什麼事都不做的 I/O action，或是我們不希望這個 do block 形成的 I/O action 的結果值是這個 block 中的最後一個 I/O action，我們希望有一個不同的結果值，所以我們用 `return` 來作一個 I/O action 包了我們想要的結果放在 do block 的最後。

在我們接下去講檔案之前，讓我們來看看有哪些實用的函數可以處理 I/O。

`putStr` 跟 `putStrLn` 幾乎一模一樣，都是接受一個字串當作參數，並回傳一個 I/O action 打印出字串到終端上，只差在 `putStrLn` 會換行而 `putStr` 不會罷了。

```
main = do putStrLn "Hey, "
          putStrLn "I'm "
          putStrLn "Andy!"
```

```
$ runhaskell putstr_test.hs
Hey, I'm Andy!
```

他的 type signature 是 `putStr :: String -> IO ()`，所以是一個包在 I/O action 中的 unit。也就是空值，沒有辦法綁定他。

`putChar` 接受一個字元，並回傳一個 I/O action 將他打印到終端上。

```
main = do putChar 't'
          putChar 'e'
          putChar 'h'
```

```
$ runhaskell putchar_test.hs
teh
```

`putStr` 實際上就是 `putChar` 遞迴定義出來的。`putStr` 的邊界條件是空字串，所以假設我們打印一個空字串，那他只是回傳一個什麼都不做的 I/O action，像 `return ()`。如果打印的不是空字串，那就先用 `putChar` 打印出字串的第一個字元，然後再用 `putStr` 打印出字串剩下部份。

```
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do
    putChar x
    putStr xs
```

看看我們如何在 I/O 中使用遞迴，就像我們在 pure code 中所做的一樣。先定義一個邊界條件，然後再思考剩下如何作。

`print` 接受任何是 `Show` typeclass 的 instance 的型態的值，這代表我們知道如何用字串表示他，呼叫 `show` 來將值變成字串然後將其輸出到終端上。基本上，他就是 `putStrLn`。`show`。首先呼叫 `show` 然後把結果餵給 `putStrLn`，回傳一個 I/O action 打印出我們的值。

```
main = do print True
          print 2
          print "haha"
          print 3.2
          print [3,4,3]
```

```
$ runhaskell print_test.hs
True
2
"haha"
3.2
[3,4,3]
```

就像你看到的，這是個很方便的函數。還記得我們提到 I/O actions 只有在 `main` 中才會被執行以及在 GHCI 中運算的事情嗎？當我們用鍵盤打了些值，像 `3` 或 `[1,2,3]` 並按下 Enter，GHCI 實際上就是用了 `print` 來將這些值輸出到終端。

```
ghci> 3
3
ghci> print 3
3
ghci> map (++"!") ["hey", "ho", "woo"]
["hey!", "ho!", "woo!"]
ghci> print (map (++"!") ["hey", "ho", "woo"])
["hey!", "ho!", "woo!"]
```

當我們需要打印出字串，我們會用 `putStrLn`，因為我們不想要周圍有引號，但對於輸出值來說，`print` 才是最常用的。

`getChar` 是一個從輸入讀進一個字元的 I/O action，因此他的 type signature 是 `getChar :: IO Char`，代表一個 I/O action 的結果是 `Char`。注意由於緩衝區的關係，只有當 Enter 被按下的時候才會觸發讀取字元的行為。

```
main = do
  c <- getChar
  if c /= ' '
    then do
      putChar c
      main
    else return ()
```

這程式看起來像是讀取一個字元並檢查他是否為一個空白。如果是的話便停止，如果不是的話便打印到終端上並重複之前的行為。在某種程度上來說也不能說錯，只是結果不如你預期而已。來看看結果吧。

```
$ runhaskell getchar_test.hs
hello sir
hello
```

上面的第二行是輸入。我們輸入了 `hello sir` 並按下了 Enter。由於緩衝區的關係，程式是在我們按了 Enter 後才執行而不是在某個輸入字元的時候。一旦我們按下了 Enter，那他就把我們直到目前輸入的一次做完。

`when` 這函數可以在 `Control.Monad` 中找到他（你必須 `import Control.Monad` 才能使用他）。他在一個 do block 中看起來就像一個控制流程的 statement，但實際上他的確是一個普通的函數。他接受一個 boolean 值跟一個 I/O action。如果 boolean 值是 `True`，便回傳我們傳給他的 I/O action。如果 boolean 值是 `False`，便回傳 `return ()`，即什麼都不做的 I/O action。我們接下來用 `when` 來改寫我們之前的程式。

```
import Control.Monad

main = do
    c <- getChar
    when (c /= ' ') $ do
        putChar c
        main
```

就像你看到的，他可以將 `if something then do some I/O action else return ()` 這樣的模式封裝起來。

`sequence` 接受一串 I/O action，並回傳一個會依序執行他們的 I/O action。運算的結果是包在一個 I/O action 的一連串 I/O action 的運算結果。他的 type signature 是 `sequence :: [IO a] -> IO [a]`

```
main = do
    a <- getLine
    b <- getLine
    c <- getLine
    print [a,b,c]
```

其實可以寫成

```
main = do
    rs <- sequence [getLine, getLine, getLine]
    print rs
```

所以 `sequence [getLine, getLine, getLine]` 作成了一個執行 `getLine` 三次的 I/O action。如果我們對他綁定一個名字，結果便是這串結果的串列。也就是說，三個使用者輸入的東西組成的串列。

一個常見的使用方式是我們將 `print` 或 `putStrLn` 之類的函數 `map` 到串列上。`map print [1,2,3,4]` 這個動作並不會產生一個 I/O action，而是一串 I/O action，就像是 `[print 1, print 2, print 3, print 4]`。如果我們將一串 I/O action 變成一個 I/O action，我們必須用 `sequence`

```
ghci> sequence (map print [1,2,3,4,5])
1
2
3
4
5
[(),(),(),(),()]
```

那 `[(),(),(),(),()]` 是怎麼回事？當我們在 GHCI 中運算 I/O action，他會被執行並把結果打印出來，唯一例外是結果是 `()` 的時候不會被打印出。這也是為什麼 `putStrLn "hehe"` 在 GHCI 中只會打印出 `hehe`（因為 `putStrLn "hehe"` 的結果是 `()`）。但當我們使用 `getLine` 時，由於 `getLine` 的型態是 `IO String`，所以結果會被打印出來。

由於對一個串列 `map` 一個回傳 I/O action 的函數，然後再 `sequence` 他這個動作太常用了。所以有一些函數在函式庫中 `mapM` 跟 `mapM_`。`mapM` 接受一個函數跟一個串列，將對串列用函數 `map` 然後 `sequence` 結果。`mapM_` 也作同樣的事，只是他把運算的結果丟掉而已。在我們不關心 I/O action 結果的情況下，`mapM_` 是最常被使用的。

```
ghci> mapM print [1,2,3]
1
2
3
[(),(),()]
ghci> mapM_ print [1,2,3]
1
2
3
```

`forever` 接受一個 I/O action 並回傳一個永遠作同一件事的 I/O action。你可以在 `Control.Monad` 中找到他。下面的程式會不斷地要使用者輸入些東西，並把輸入的東西轉成大寫輸出到螢幕上。

```

import Control.Monad
import Data.Char

main = forever $ do
    putStrLn "Give me some input: "
    l <- getLine
    putStrLn $ map toUpper l

```

在 `Control.Monad` 中的 `forM` 跟 `mapM` 的作用一樣，只是參數的順序相反而已。第一個參數是串列，而第二個則是函數。這有什麼用？在一些有趣的情況下還是有用的：

```

import Control.Monad

main = do
    colors <- forM [1,2,3,4] (\a -> do
        putStrLn $ "Which color do you associate with the number " ++ show a ++ "?"
        color <- getLine
        return color)
    putStrLn "The colors that you associate with 1, 2, 3 and 4 are: "
    mapM putStrLn colors

```

`(\a -> do ...)` 是接受一個數字並回傳一個 I/O action 的函數。我們必須用括號括住他，不然 lambda 會貪心 match 的策略會把最後兩個 I/O action 也算進去。注意我們在 do block 裡面 `return color`。我們那麼作是讓 do block 的結果是我們選的顏色。實際上我們並不需那麼作，因為 `getLine` 已經達到我們的目的。先 `color <- getLine` 再 `return color` 只不過是把值取出再包起來，其實是跟 `getLine` 效果相當。`forM` 產生一個 I/O action，我們把結果綁定到 `colors` 這名稱。`colors` 是一個普通包含字串的串列。最後，我們用 `mapM putStrLn colors` 打印出所有顏色。

你可以把 `forM` 的意思想成將串列中的每個元素作成一個 I/O action。至於每個 I/O action 實際作什麼就要看原本的元素是什麼。然後，執行這些 I/O action 並將結果綁定到某個名稱上。或是直接將結果忽略掉。

```
$ runhaskell from_test.hs
which color do you associate with the number 1?
white
Which color do you associate with the number 2?
blue
Which color do you associate with the number 3?
red
Which color do you associate with the number 4?
orange
The colors that you associate with 1, 2, 3 and 4 are:
white
blue
red
orange
```

其實我們也不是一定要用到 `form`，只是用了 `form` 程式會比較容易理解。正常來講是我們需要在 `map` 跟 `sequence` 的時候定義 I/O action 的時候使用 `form`，同樣地，我們也可以將最後一行寫成 `form colors putStrLn`。

在這一節，我們學會了輸入與輸出的基礎。我們也了解了什麼是 I/O action，他們是如何幫助我們達成輸入與輸出的目的。這邊重複一遍，I/O action 跟其他 Haskell 中的 value 沒有兩樣。我們能夠把他當參數傳給函式，或是函式回傳 I/O action。他們特別之處在於當他們是寫在 `main` 裡面或 GHCI 裡面的時候，他們會被執行，也就是實際輸出到你螢幕或輸出音效的時候。每個 I/O action 也能包著一個從真實世界拿回來的值。

不要把像是 `putStrLn` 的函式想成接受字串並輸出到螢幕。要想成一個函式接受字串並回傳一個 I/O action。當 I/O action 被執行的時候，會漂亮地打印出你想要的東西。

## 檔案與字符流



`getChar` 是一個讀取單一字元的 I/O action。`getLine` 是一個讀取一行的 I/O action。這是兩個非常直覺的函式，多數程式語言也有類似這兩個函式的 statement 或 function。但現在我們來看看 `getContents`。`getContents` 是一個從標準輸入讀取直到 end-of-file 字元的 I/O action。他的型態是 `getContents :: IO String`。最酷的是 `getContents` 是惰性 I/O (Lazy I/O)。當我們寫了 `foo <- getContents`，他並不會馬上讀取所有輸入，將他們存在 memory 裡面。他只有當你真的需要輸入資料的時候才會讀取。

當我們需要重導一個程式的輸出到另一個程式的輸入時，`getContents` 非常有用。假設我們有下面一個文字檔：

```
I'm a lil' teapot
What's with that airplane food, huh?
It's so small, tasteless
```

還記得我們介紹 `forever` 時寫的小程式嗎？會把所有輸入的東西轉成大寫的那一個。為了防止你忘記了，這邊再重複一遍。

```
import Control.Monad
import Data.Char

main = forever $ do
    putStrLn "Give me some input: "
    l <- getLine
    putStrLn $ map toUpper l
```

將我們的程式存成 `capslocker.hs` 然後編譯他。然後用 Unix 的 Pipe 將文字檔餵給我們的程式。我們使用的是 GNU 的 `cat`，會將指定的檔案輸出到螢幕。

```
$ ghc --make capslocker
[1 of 1] Compiling Main           ( capslocker.hs, capslocker.o )
Linking capslocker ...
$ cat haiku.txt
I'm a lil' teapot
What's with that airplane food, huh?
It's so small, tasteless
$ cat haiku.txt | ./capslocker
I'M A LIL' TEAPOT
WHAT'S WITH THAT AIRPLANE FOOD, HUH?
IT'S SO SMALL, TASTELESS
capslocker <stdin>: hGetLine: end of file
```

就如你看到的，我們是用 `|` 這符號來將某個程式的輸出 piping 到另一個程式的輸入。我們做的事相當於 run 我們的 `capslocker`，然後將 `haiku` 的內容用鍵盤打到終端上，最後再按 `Ctrl-D` 來代表 end-of-file。這就像執行 `cat haiku.txt` 後大喊，嘿，不要把內容打印到終端上，把內容塞到 `capslocker`！

我們用 `forever` 在做的事基本上就是將輸入經過轉換後變成輸出。用 `getContents` 的話可以讓我們的程式更加精鍊。

```
import Data.Char

main = do
    contents <- getContents
    putStrLn (map toUpper contents)
```

我們將 `getContents` 取回的字串綁定到 `contents`。然後用 `toUpper` `map` 到整個字串後打印到終端上。記住字串基本上就是一串惰性的串列 (list)，同時 `getContents` 也是惰性 I/O，他不會一口氣讀入內容然後將內容存在記憶體中。實際上，他會一行一行讀入並輸出大寫的版本，這是因為輸出才是真的需要輸入的資料的時候。

```
$ cat haiku.txt | ./capslocker
I'M A LIL' TEAPOT
WHAT'S WITH THAT AIRPLAN FOOD, HUH?
IT'S SO SMALL, TASTELESS
```

很好，程式運作正常。假如我們執行 `capslocker` 然後自己打幾行字呢？

```
$ ./capslocker
hey ho
HEY HO
lets go
LETS GO
```

按下 Ctrl-D 來離開環境。就像你看到的，程式是一行一行將我們的輸入打印出來。當 `getContent` 的結果被綁定到 `contents` 的時候，他不是被表示成在記憶體中的一個字串，反而比較像是他有一天會是字串的一個承諾。當我們將 `toUpper` `map` 到 `contents` 的時候，便是一個函數被承諾將會被 `map` 到內容上。最後 `putStr` 則要求先前的承諾說，給我一行大寫的字串吧。實際上還沒有任何一行被取出，所以便跟 `contents` 說，不如從終端那邊取出些字串吧。這才是 `getContents` 真正從終端讀入一行並把這一行交給程式的時候。程式便將這一行用 `toUpper` 處理並交給 `putStr`，`putStr` 則打印出他。之後 `putStr` 再說：我需要下一行。整個步驟便再重複一次，直到讀到 end-of-file 為止。

接著我們來寫個程式，讀取輸入，並只打印出少於十個字元的行。

```

main = do
    contents <- getContents
    putStrLn (shortLinesOnly contents)

shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filter (\line -> length line < 10) allLines
        result = unlines shortLines
    in result

```

我們把 I/O 部份的程式碼弄得很短。由於程式的行為是接某些輸入，作些處理然後輸出。我們可以把他想成讀取輸入，呼叫一個函數，然後把函數的結果輸出。

`shortLinesOnly` 的行為是這樣：拿到一個字串，像是 "short\nooooooooooooong\nshort again"。這字串有三行，前後兩行比較短，中間一行很長。他用 `lines` 把字串分成 `["short", "ooooooooooooong", "short again"]`，並把結果綁定成 `allLines`。然後過濾這些字串，只有少於十個字元的留下，`["short", "short again"]`，最後用 `unlines` 把這些字串用換行接起來，形成 "short\nshort again"

```

i'm short
so am i
i am a looooooooooong line!!!
yeah i'm long so what hahahaha!!!!!
short line
ooooooooooooooooooooooooooooong
short

```

```

$ ghc --make shortlinesonly
[1 of 1] Compiling Main           ( shortlinesonly.hs, shortlinesonly.o )
Linking shortlinesonly ...
$ cat shortlines.txt | ./shortlinesonly
i'm short
so am i
short

```

我們把 `shortlines.txt` 的內容經由 pipe 送給 `shortlinesonly`，結果就如你看到，我們只有得到比較短的行。

從輸入那一些字串，經由一些轉換然後輸出這樣的模式實在太常用了。常用到甚至建立了一個函數叫 `interact`。`interact` 接受一個 `String -> String` 的函數，並回傳一個 I/O action。那個 I/O action 會讀取一些輸入，呼叫提供的函數，然後把函數的結果打印出來。所以我們的程式可以改寫成這樣。

```
main = interact shortLinesOnly

shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filter (\line -> length line < 10) allLines
        result = unlines shortLines
    in result
```

我們甚至可以再讓程式碼更短一些，像這樣

```
main = interact $ unlines . filter ((<10) . length) . lines
```

看吧，我們讓程式縮到只剩一行了，很酷吧！

能應用 `interact` 的情況有幾種，像是從輸入 pipe 讀進一些內容，然後丟出一些結果的程式；或是從使用者獲取一行一行的輸入，然後丟回根據那一行運算的結果，再拿取另一行。這兩者的差別主要是取決於使用者使用他們的方式。

我們再來寫另一個程式，它不斷地讀取一行行並告訴我們那一行字串是不是一個回文字串 (palindrome)。我們當然可以用 `getLine` 讀取一行然後再呼叫 `main` 作同樣的事。不過同樣的事情可以用 `interact` 更簡潔地達成。當使用 `interact` 的時候，想像你是將輸入經有某些轉換成輸出。在這個情況當中，我們要將每一行輸入轉換成 "palindrome" 或 "not a palindrome"。所以我們必須寫一個函數將 "elephant\nABCBA\nwhatever" 轉換成 not a palindrome\nnot a palindrome"。來動手吧！

```
respondPalindromes contents = unlines (map (\xs ->
    if isPalindrome xs then "palindrome" else "not a palindrome") (lines contents))
    where isPalindrome xs = xs == reverse xs
```

再來將程式改寫成 point-free 的形式

```
respondPalindromes = unlines . map (\xs ->
    if isPalindrome xs then "palindrome" else "not a palindrome") . lines
    where isPalindrome xs = xs == reverse xs
```

很直覺吧！首先將 "elephant\nABCBA\nwhatever" 變成 ["elephant", "ABCBA", "whatever"] 然後將一個 lambda 函數 map 它， ["not a palindrome", "palindrome", "not a palindrome"] 然後用 `unlines` 變成一行字串。接著

```
main = interact respondPalindromes
```

來測試一下吧。

```
$ runhaskell palindrome.hs
hehe
not a palindrome
ABCBA
palindrome
cookie
not a palindrome
```

即使我們的程式是把一大把字串轉換成另一個，其實他表現得好像我們是一行一行做的。這是因為 Haskell 是惰性的，程式想要打印出第一行結果時，他必須要先有第一行輸入。所以一旦我們給了第一行輸入，他便打印出第一行結果。我們用 end-of-line 字元來結束程式。

我們也可以用 pipe 的方式將輸入餵給程式。假設我們有這樣一個檔案。

```
dogaroo
radar
rotor
madam
```

將他存為 `words.txt`，將他餵給程式後得到的結果

```
$ cat words.txt | runhaskell palindromes.hs
not a palindrome
palindrome
palindrome
palindrome
```

再一次地提醒，我們得到的結果跟我們自己一個一個字打進輸入的內容是一樣的。我們看不到 `palindrome.hs` 輸入的內容是因為內容來自於檔案。

你應該大致了解 Lazy I/O 是如何運作，並能善用他的優點。他可以從輸入轉換成輸出的角度方向思考。由於 Lazy I/O，沒有輸入在被用到之前是真的被讀入。

到目前為止，我們的示範都是從終端讀取某些東西或是打印出某些東西到終端。但如果我們想要讀寫檔案呢？其實從某個角度來說我們已經作過這件事了。我們可以把讀寫終端想成讀寫檔案。只是把檔案命名成 `stdout` 跟 `stdin` 而已。他們分別代表標準輸出跟標準輸入。我們即將看到的讀寫檔案跟讀寫終端並沒什麼不同。

首先來寫一個程式，他會開啟一個叫 `girlfriend.txt` 的檔案，檔案裡面有 Avril Lavigne 的暢銷名曲 Girlfriend，並將內容打印到終端上。接下來是 `girlfriend.txt` 的內容。

```
Hey! Hey! You! You!
I don't like your girlfriend!
No way! No way!
I think you need a new one!
```

這則是我們的主程式。

```
import System.IO

main = do
    handle <- openFile "girlfriend.txt" ReadMode
    contents <- hGetContents handle
    putStrLn contents
    hClose handle
```

執行他後得到的結果。

```
$ runhaskell girlfriend.hs
Hey! Hey! You! You!
I don't like your girlfriend!
No way! No way!
I think you need a new one!
```

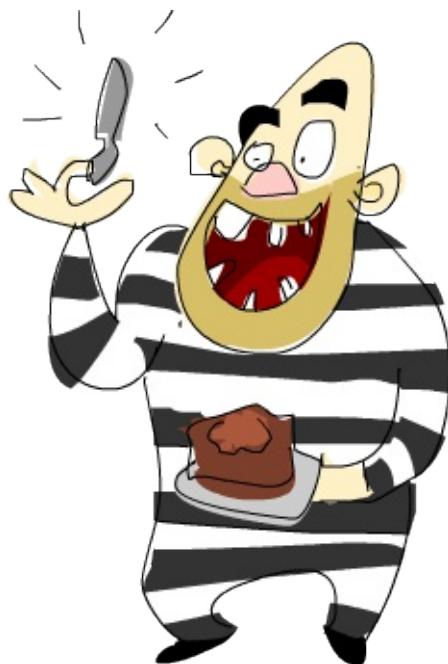
我們來一行行看一下程式。我們的程式用 do 把好幾個 I/O action 繩在一起。在 do block 的第一行，我們注意到有一個新的函數叫 **openFile**。他的 type signature 是 `openFile :: FilePath -> IOMode -> IO Handle`。他說了 `openFile` 接受一個檔案路徑跟一個 `IOMode`，並回傳一個 I/O action，他會打開一個檔案並把檔案關聯到一個 handle。

`FilePath` 不過是 `String` 的 type synonym。

```
type FilePath = String
```

`IOMode` 則是一個定義如下的型態

```
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```



就像我們之前定義的型態，分別代表一個星期的七天。這個型態代表了我們想對打開的檔案做什麼。很簡單吧。留意到我們的型態是 `IOMode` 而不是 `IO Mode`。`IO Mode` 代表的是一個 I/O action 包含了一個型態為 `Mode` 的值，但 `IOMode` 不過是一個陽春的 enumeration。

最後，他回傳一個 I/O action 會將指定的檔案用指定的模式打開。如果我們將 I/O action 繩定到某個東西，我們會得到一個 `Handle`。型態為 `Handle` 的值代表我們的檔案在哪裡。有了 `handle` 我們才知道要從哪個檔案讀取內容。想讀取檔案但不將檔案繩定到 `handle` 上這樣做是很蠢的。所以，我們將一個 `handle` 繩定到 `handle`。

接著一行，我們看到一個叫 **`hGetContents`** 的函數。他接了一個 `Handle`，所以他知道要從哪個檔案讀取內容並回傳一個 `IO String`。一個包含了檔案內容的 I/O action。這函數跟 `getContents` 差不多。唯一的差別是 `getContents` 會自動從標準輸入讀取內容（也就是終端），而 `hGetContents` 接了一個 `file handle`，這 `file handle` 告訴他讀取哪個檔案。除此之外，他們都是一樣的。就像 `getContents`，`hGetContents` 不會把檔案一次都拉到記憶體中，而是有必要才會讀取。這非常酷，因為我們把 `contents` 當作是整個檔案般用，但他實際上不在記憶體中。就算這是個很大的檔案，`hGetContents` 也不會塞爆你的記憶體，而是只有必要的時候才會讀取。

要留意檔案的 `handle` 還有檔案的內容兩個概念的差異，在我們的程式中他們分別被繩定到 `handle` 跟 `contents` 兩個名字。`handle` 是我們拿來區分檔案的依據。如果你把整個檔案系統想成一本厚厚的書，每個檔案分別是其中的一個章節，`handle` 就像是書籤一般標記了你現在正在閱讀（或寫入）哪一個章節，而內容則是章節本身。

我們使用 `putStr contents` 打印出內容到標準輸出，然後我們用了 **`hClose`**。他接受一個 `handle` 然後回傳一個關掉檔案的 I/O action。在用了 `openFile` 之後，你必須自己把檔案關掉。

要達到我們目的的另一種方式是使用 `withFile`, 他的 type signature 是 `withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a`。他接受一個檔案路徑, 一個 `IOMode` 以及一個函數, 這函數則接受一個 `handle` 跟一個 I/O action。`withFile` 最後回傳一個會打開檔案, 對檔案作某件事然後關掉檔案的 I/O action。處理的結果是包在最後的 I/O action 中, 這結果跟我們給的函數的回傳是相同的。這聽起來有些複雜, 但其實很簡單, 特別是我們有 `lambda`, 來看看我們用 `withFile` 改寫前面程式的一個範例：

```
import System.IO

main = do
    withFile "girlfriend.txt" ReadMode (\handle -> do
        contents <- hGetContents handle
        putStrLn contents)
```

正如你看到的, 程式跟之前的看起來很像。`(\handle -> ... )` 是一個接受 `handle` 並回傳 I/O action 的函數, 他通常都是用 `lambda` 來表示。我們需要一個回傳 I/O action 的函數的理由而不是一個本身作處理並關掉檔案的 I/O action, 是因為這樣一來那個 I/O action 不會知道他是對哪個檔案在做處理。用 `withFile` 的話, `withFile` 會打開檔案並把 `handle` 傳給我們給他的函數, 之後他則拿到一個 I/O action, 然後作成一個我們描述的 I/O action, 最後關上檔案。例如我們可以這樣自己作一個 `withFile` :

```
withFile' :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
withFile' path mode f = do
    handle <- openFile path mode
    result <- f handle
    hClose handle
    return result
```



我們知道要回傳的是一個 I/O action，所以我們先放一個 do。首先我們打開檔案，得到一個 handle。然後我們 apply handle 到我們的函數，並得到一個做事的 I/O action。我們綁定那個 I/O action 到 result 這個名字，關上 handle 並 return result。return 的作用把從 f 得到的結果包在 I/O action 中，這樣一來 I/O action 中就包含了 f handle 得到的結果。如果 f handle 回傳一個從標準輸入讀去數行並寫到檔案然後回傳讀入的行數的 I/O action，在 withFile' 的情形中，最後的 I/O action 就會包含讀入的行數。

就像 hGetContents 對應 getContents 一樣，只不過是針對某個檔案。我們也有 hGetLine、hPutStr、hPutStrLn、hGetChar 等等。他們分別是少了 h 的那些函數的對應。只不過他們要多拿一個 handle 當參數，並且是針對特定檔案而不是標準輸出或標準輸入。像是 putStrLn 是一個接受一個字串並回傳一個打印出加了換行字元的字串的 I/O action 的函數。hPutStrLn 接受一個 handle 跟一個字串，回傳一個打印出加了換行字元的字串到檔案的 I/O action。以此類推，hGetLine 接受一個 handle 然後回傳一個從檔案讀取一行的 I/O action。

讀取檔案並對他們的字串內容作些處理實在太常見了，常見到我們有三個函數來更進一步簡化我們的工作。

**readFile** 的 type signature 是 readFile :: FilePath -> IO String。記住，FilePath 不過是 String 的一個別名。readFile 接受一個檔案路徑，回傳一個惰性讀取我們檔案的 I/O action。然後將檔案的內容綁定到某個字串。他比起先 openFile，綁定 handle，然後 hGetContents 要好用多了。這邊是一個用 readFile 改寫之前例子的範例：

```
import System.IO

main = do
    contents <- readFile "girlfriend.txt"
    putStrLn contents
```

由於我們拿不到 handle，所以我們也無法關掉他。這件事 Haskell 的 readFile 在背後幫我們做了。

**writeFile** 的型態是 writeFile :: FilePath -> String -> IO ()。他接受一個檔案路徑，以及一個要寫到檔案中的字串，並回傳一個寫入動作的 I/O action。如果這個檔案已經存在了，他會先把檔案內容都砍了再寫入。下面示範了如何把 girlfriend.txt 的內容轉成大寫然後寫入到 friendcaps.txt 中

```
import System.IO
import Data.Char

main = do
    contents <- readFile "girlfriend.txt"
    writeFile "friendcaps.txt" (map toUpper contents)
```

```
$ runhaskell girlfriendtocaps.hs
$ cat girlfriendcaps.txt
HEY! HEY! YOU! YOU!
I DON'T LIKE YOUR GIRLFRIEND!
NO WAY! NO WAY!
I THINK YOU NEED A NEW ONE!
```

**appendFile** 的型態很像 `writeFile`，只是 `appendFile` 並不會在檔案存在時把檔案內容砍掉而是接在後面。

假設我們有一個檔案叫 `todo.txt`，裡面每一行是一件要做的事情。現在我們寫一個程式，從標準輸入接受一行將他加到我們的 to-do list 中。

```
import System.IO

main = do
    todoItem <- getLine
    appendFile "todo.txt" (todoItem ++ "\n")
```

```
$ runhaskell appendtodo.hs
Iron the dishes
$ runhaskell appendtodo.hs
Dust the dog
$ runhaskell appendtodo.hs
Take salad out of the oven
$ cat todo.txt
Iron the dishes
Dust the dog
Take salad out of the oven
```

由於 `getLine` 回傳的值不會有換行字元，我們需要在每一行最後加上 `"\n"`。

還有一件事，我們提到 `contents <- hGetContents handle` 是惰性 I/O，不會將檔案一次都讀到記憶體中。所以像這樣寫的話：

```
main = do
    withFile "something.txt" ReadMode (\handle -> do
        contents <- hGetContents handle
        putStrLn contents)
```

實際上像是用一個 pipe 把檔案弄到標準輸出。正如你可以把 `list` 想成 `stream` 一樣，你也可以把檔案想成 `stream`。他會每次讀一行然後打印到終端上。你也許會問這個 pipe 究竟一次可以塞多少東西，讀去硬碟的頻率究竟是多少？對於文字檔而言，預設的 buffer 通常是 line-

buffering。這代表一次被讀進來的大小是一行。這也是為什麼在這個 case 我們是一行一行處理。對於 binary file 而言，預設的 buffer 是 block-buffering。這代表我們是一個 chunk 一個 chunk 去讀得。而一個 chunk 的大小是根據作業系統不同而不同。

你能用 `hSetBuffering` 來控制 buffer 的行為。他接受一個 handle 跟一個 `BufferMode`，回傳一個會設定 buffer 行為的 I/O action。`BufferMode` 是一個 enumeration 型態，他可能的值有：`NoBuffering`，`LineBuffering` 或 `BlockBuffering (Maybe Int)`。其中 `Maybe Int` 是表示一個 chunck 有幾個 byte。如果他的值是 `Nothing`，則作業系統會幫你決定 chunk 的大小。`NoBuffering` 代表我們一次讀一個 character。一般來說 `NoBuffering` 的表現很差，因為他存取硬碟的頻率很高。

接下來是我們把之前的範例改寫成用 2048 bytes 的 chunk 讀取，而不是一行一行讀。

```
main = do
    withFile "something.txt" ReadMode (\handle -> do
        hSetBuffering handle $ BlockBuffering (Just 2048)
        contents <- hGetContents handle
        putStrLn contents)
```

用更大的 chunk 來讀取對於減少存取硬碟的次數是有幫助的，特別是我們的檔案其實是透過網路來存取。

我們也可以使用 `hFlush`，他接受一個 handle 並回傳一個會 flush buffer 到檔案的 I/O action。當我們使用 line-buffering 的時候，buffer 在每一行都會被 flush 到檔案。當我們使用 block-buffering 的時候，是在我們讀每一個 chunk 作 flush 的動作。flush 也會發生在關閉 handle 的時候。這代表當我們碰到換行字元的時候，讀或寫的動作都會停止並回報手邊的資料。但我們能使用 `hFlush` 來強迫回報所有已經在 buffer 中的資料。經過 flushing 之後，資料也就能被其他程式看見。

把 block-buffering 的讀取想成這樣：你的馬桶會在水箱有一加侖的水的時候自動沖水。所以你不斷灌水進去直到一加侖，馬桶就會自動沖水，在水裡面的資料也就會被看到。但你也可以手動地按下沖水鈕來沖水。他會讓現有的水被沖走。沖水這個動作就是 `hFlush` 這個名字的含意。

我們已經寫了一個將 item 加進 to-do list 裡面的程式，現在我們想加進移除 item 的功能。我先把程式碼貼上然後講解他。我們會使用一些新面孔像是 `System.Directory` 以及 `System.IO` 裡面的函數。

來看一下我們包含移除功能的程式：

```

import System.IO
import System.Directory
import Data.List

main = do
    handle <- openFile "todo.txt" ReadMode
    (tempName, tempHandle) <- openTempFile "." "temp"
    contents <- hGetContents handle
    let todoTasks = lines contents
    numberedTasks = zipWith (\n line -> show n ++ " - " ++ line) [0..] todoTasks
    putStrLn "These are your TO-DO items:"
    putStrLn $ unlines numberedTasks
    putStrLn "Which one do you want to delete?"
    numberString <- getLine
    let number = read numberString
    newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile "todo.txt"
    renameFile tempName "todo.txt"

```

一開始，我們用 `read mode` 打開 `todo.txt`，並把他綁定到 `handle`。

接著，我們使用了一個之前沒用過在 `System.IO` 中的函數 `openTempFile`。他的名字淺顯易懂。他接受一個暫存的資料夾跟一個樣板檔案名，然後打開一個暫存檔。我們使用 `."` 當作我們的暫存資料夾，因為 `.` 在幾乎任何作業系統中都代表了現在所在的資料夾。我們使用 `"temp"` 當作我們暫存檔的樣板名，他代表暫存檔的名字會是 `temp` 接上某串隨機字串。他回傳一個創建暫存檔的 I/O action，然後那個 I/O action 的結果是一個 pair：暫存檔的名字跟一個 `handle`。我們當然可以隨便開啟一個 `todo2.txt` 這種名字的檔案。但使用 `openTempFile` 會是比較好的作法，這樣你不會不小心覆寫任何檔案。

我們不用 `getCurrentDirectory` 的來拿到現在所在資料夾而用 `."` 的原因是 `.` 在 unix-like 系統跟 Windows 中都表示現在的資料夾。

然後，我們綁定 `todo.txt` 的內容成 `contents`。把字串斷成一串字串，每個字串代表一行。`todoTasks` 就變成 `["Iron the dishes", "Dust the dog", "Take salad out of the oven"]`。我們用一個會把 3 跟 `"hey"` 變成 `"3 - hey"` 的函數，然後從 0 開始把這個串列 `zip` 起來。所以 `numberedTasks` 就是 `["0 - Iron the dishes", "1 - Dust the dog" ... ]`。我們用 `unlines` 把這個串列變成一行，然後打印到終端上。注意我們也有另一種作法，就是用 `mapM putStrLn numberedTasks`。

我們問使用者他們想要刪除哪一個並且等著他們輸入一個數字。假設他們想要刪除 1 號，那代表 `Dust the dog`，所以他們輸入 `1`。於是 `numberString` 就代表 `"1"`。由於我們想要一個數字，而不是一個字串，所以我們用對 `1` 使用 `read`，並且綁定到 `number`。

還記得在 `Data.List` 中的 `delete` 跟 `!!` 嗎？`!!` 回傳某個 `index` 的元素，而 `delete` 刪除在串列中第一個發現的元素，然後回傳一個新的沒有那個元素的串列。`(todoTasks !! number)` (`number` 代表 `1`) 回傳 "Dust the dog"。我們把 `todoTasks` 去掉第一個 "Dust the dog" 後的串列綁定到 `newTodoItems`，然後用 `unlines` 變成一行然後寫到我們所打開的暫存檔。舊有的檔案並沒有變動，而暫存檔包含砍掉那一行後的所有內容。

在我們關掉原始檔跟暫存檔之後我們用 `removeFile` 來移除原本的檔案。他接受一個檔案路徑並且刪除檔案。刪除舊得 `todo.txt` 之後，我們用 `renameFile` 來將暫存檔重新命名成 `todo.txt`。特別留意 `removeFile` 跟 `renameFile` (兩個都在 `System.Directory` 中) 接受的是檔案路徑，而不是 `handle`。

這就是我們要的，實際上我們可以用更少行寫出同樣的程式，但我們很小心地避免覆寫任何檔案，並詢問作業系統我們可以把暫存檔擺在哪？讓我們來執行看看。

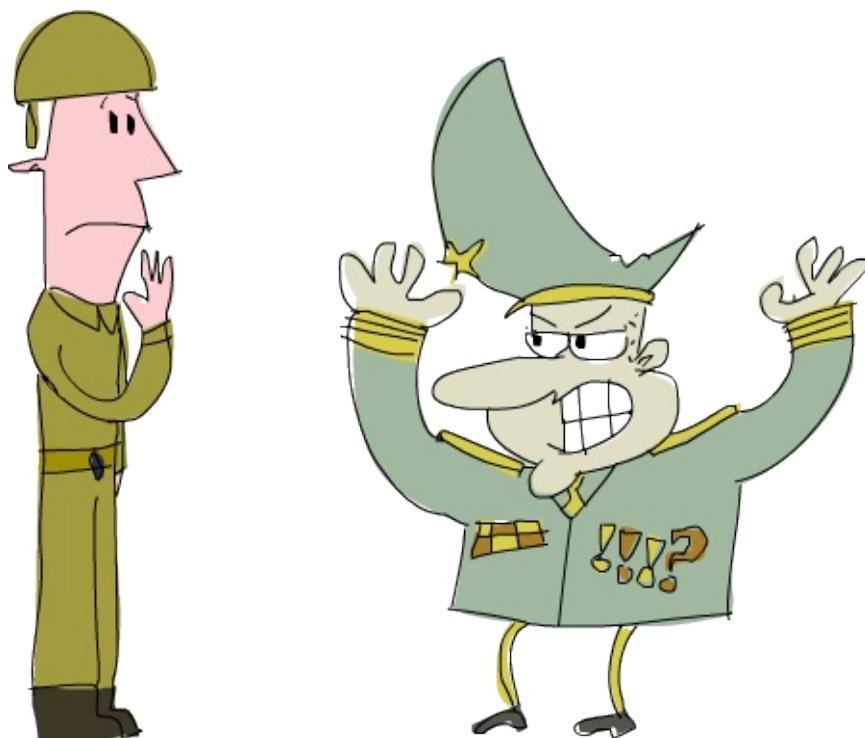
```
$ runhaskell deletetodo.hs
These are your TO-DO items:
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven
Which one do you want to delete?
1

$ cat todo.txt
Iron the dishes
Take salad out of the oven

$ runhaskell deletetodo.hs
These are your TO-DO items:
0 - Iron the dishes
1 - Take salad out of the oven
Which one do you want to delete?
0

$ cat todo.txt
Take salad out of the oven
```

## 命令列引數



如果你想要寫一個在終端裡運行的程式，處理命令列引數是不可或缺的。幸運的是，利用 Haskell 的 Standard Library 能讓我們有效地處理命令列引數。

在之前的章節中，我們寫了一個能將 to-do item 加進或移除 to-do list 的一個程式。但我們的寫法有兩個問題。第一個是我們把放 to-do list 的檔案名稱給寫死了。我們擅自決定使用者不會有很多個 to-do lists，就把檔案命名為 todo.txt。

一種解決的方法是每次都詢問使用者他們想將他們的 to-do list 放進哪個檔案。我們在使用者要刪除的時候也採用這種方式。這是一種可以運作的方式，但不太能被接受，因為他需要使用者運行程式，等待程式詢問才能回答。這被稱為互動式的程式，但討厭的地方在當你想要自動化執行程式的時候，好比說寫成 script，這會讓你的 script 寫起來比較困難。

這也是為什麼有時候讓使用者在執行的時候就告訴程式他們要什麼會比較好，而不是讓程式去問使用者要什麼。比較好的方式是讓使用者透過命令列引數告訴程式他們想要什麼。

在 `System.Environment` 模組當中有兩個很酷的 I/O actions，一個是 **getArgs**，他的 type 是 `getArgs :: IO [String]`，他是一個拿取命令列引數的 I/O action，並把結果放在包含的一個串列中。**getProgName** 的型態是 `getProgName :: IO String`，他則是一個 I/O action 包含了程式的名稱。

我們來看一個展現他們功能的程式。

```

import System.Environment
import Data.List

main = do
    args <- getArgs
    progName <- getProgName
    putStrLn "The arguments are:"
    mapM putStrLn args
    putStrLn "The program name is:"
    putStrLn progName

```

我們將 `getArgs` 跟 `progName` 分別綁定到 `args` 跟 `progName`。我們打印出 `The arguments are:` 以及在 `args` 中的每個引數。最後，我們打印出程式的名稱。我們把程式編譯成 `arg-test`。

```

$ ./arg-test first second w00t "multi word arg"
The arguments are:
first
second
w00t
multi word arg
The program name is:
arg-test

```

知道了這些函數現在你能寫幾個很酷的命令列程式。在之前的章節，我們寫了一個程式來加入待作事項，也寫了另一個程式刪除事項。現在我們要把兩個程式合起來，他會根據命令列引數來決定該做的事情。我們也會讓程式可以處理不同的檔案，而不是只有 `todo.txt`

我們叫這程式 `todo`，他會作三件事：

```

# 檢視待作事項
# 加入待作事項
# 刪除待作事項

```

我們暫不考慮不合法的輸入這件事。

我們的程式要像這樣運作：假如我們要加入 `Find the magic sword of power`，則我們會打 `todo add todo.txt "Find the magic sword of power"`。要檢視事項我們則會打 `todo view todo.txt`，如果要移除事項二則會打 `todo remove todo.txt 2`

我們先作一個分發的 `association list`。他會把命令列引數當作 `key`，而對應的處理函數當作 `value`。這些函數的型態都是 `[String] -> IO ()`。他們會接受命令列引數的串列並回傳對應的檢視，加入以及刪除的 I/O action。

```

import System.Environment
import System.Directory
import System.IO
import Data.List

dispatch :: [(String, [String] -> IO ())]
dispatch = [ ("add", add)
            , ("view", view)
            , ("remove", remove)
            ]

```

我們定義了 `main`, `add`, `view` 跟 `remove`, 就從 `main` 開始講吧：

```

main = do
    (command:args) <- getArgs
    let (Just action) = lookup command dispatch
    action args

```

首先，我們取出引數並把他們綁定到 `(command:args)`。如果你還記得 pattern matching，這麼做會把第一個引數綁定到 `command`，把其他的綁定到 `args`。如果我們像這樣執行程式 `todo add todo.txt "Spank the monkey"`，`command` 會變成 `"add"`，而 `args` 會變成 `["todo.txt", "Spank the monkey"]`。

在下一行，我們在一個分派的串列中尋到我們的指令是哪個。由於 `"add"` 指向 `add`，我們的結果便是 `Just add`。我們再度使用了 pattern matching 來把我們的函數從 `Maybe` 中取出。但如果我們想要的指令不在分派的串列中呢？那樣 `lookup` 就會回傳 `Nothing`，但我們這邊並不特別處理失敗的情況，所以 pattern matching 會失敗然後我們的程式就會當掉。

最後，我們用剩下的引數呼叫 `action` 這個函數。他會還傳一個加入 `item`，顯示所有 `items` 或者刪除 `item` 的 I/O action。由於這個 I/O action 是在 `main` 的 do block 中，他最後會被執行。如果我們的 `action` 函數是 `add`，他就會被餵 `args` 然後回傳一個加入 `Spank the monkey` 到 `todo.txt` 中的 I/O action。

我們剩下要做的就是實作 `add`, `view` 跟 `remove`，我們從 `add` 開始：

```

add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\n")

```

如果我們這樣執行程式 `todo add todo.txt "Spank the monkey"`，則 `"add"` 會被綁定到 `command`，而 `["todo.txt", "Spank the monkey"]` 會被帶到從 `dispatch` list 中拿到的函數。

由於我們不處理不合法的輸入，我們只針對這兩項作 pattern matching，然後回傳一個附加一行到檔案末尾的 I/O action。

接著，我們來實作檢視串列。如果我們想要檢視所有 items，我們會 `todo view todo.txt`。所以 `command` 會是 `"view"`，而 `args` 會是 `["todo.txt"]`。

```
view :: [String] -> IO ()
view [fileName] = do
    contents <- readFile fileName
    let todoTasks = lines contents
    numberedTasks = zipWith (\n line -> show n ++ " - " ++ line) [0..] todoTasks
    putStrLn $ unlines numberedTasks
```

這跟我們之前刪除檔案的程式差不多，只是我們是在顯示內容而已，

最後，我們要來實作 `remove`。他基本上跟之前寫的只有刪除功能的程式很像，所以如果你不知道刪除是怎麼做的，可以去看之前的解釋。主要的差別是我們不寫死 `todo.txt`，而是從參數取得。我們也不會提示使用者要刪除哪一號的 item，而是從參數取得。

```
remove :: [String] -> IO ()
remove [fileName, numberString] = do
    handle <- openFile fileName ReadMode
    (tempName, tempHandle) <- openTempFile "." "temp"
    contents <- hGetContents handle
    let number = read numberString
        todoTasks = lines contents
        newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile fileName
    renameFile tempName fileName
```

我們打開 `fileName` 的檔案以及一個暫存。刪除使用者要我們刪的那一行後，把檔案內容寫到暫存檔。砍掉原本的檔案然後把暫存檔重新命名成 `fileName`。

來看看完整的程式。

```

import System.Environment
import System.Directory
import System.IO
import Data.List

dispatch :: [(String, [String] -> IO ())]
dispatch = [ ("add", add)
           , ("view", view)
           , ("remove", remove)
           ]

main = do
    (command:args) <- getArgs
    let (Just action) = lookup command dispatch
    action args

add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\n")

view :: [String] -> IO ()
view [fileName] = do
    contents <- readFile fileName
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line) [0..] todoTasks
    putStrLn $ unlines numberedTasks

remove :: [String] -> IO ()
remove [fileName, numberString] = do
    handle <- openFile fileName ReadMode
    (tempName, tempHandle) <- openTempFile "." "temp"
    contents <- hGetContents handle
    let number = read numberString
        todoTasks = lines contents
        newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile fileName
    renameFile tempName fileName

```



總結我們的程式：我們做了一個 dispatch association，將指令對應到一些會接受命令列引數並回傳 I/O action 的函數。我們知道使用者下了什麼命令，並根據那個命令從 dispatch list 取出對應的函數。我們用剩下的命令列引數呼叫哪些函數而得到一些作相對應事情的 I/O action。然後便執行那些 I/O action。

在其他程式語言，我們可能會用一個大的 switch case 來實作，但使用高階函數讓我們可以要 dispatch list 紿我們要的函數，並要那些函數給我們適當的 I/O action。

讓我們看看執行結果。

```
$ ./todo view todo.txt
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven

$ ./todo add todo.txt "Pick up children from drycleaners"

$ ./todo view todo.txt
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven
3 - Pick up children from drycleaners

$ ./todo remove todo.txt 2

$ ./todo view todo.txt
0 - Iron the dishes
1 - Dust the dog
2 - Pick up children from drycleaners
```

要再另外加新的選項也是很容易。只要在 dispatch list 加入新的會作你要的事情函數。你可以試試實作一個 bump 函數，接受一個檔案跟一個 task number，他會回傳一個把那個 task 搬到 to-do list 頂端的 I/O action。

對於不合法的輸入你也可以讓程式結束地漂亮一點。(例如使用者輸入了 todo UP YOURS HAHAHAHA )可以作一個回報錯誤的 I/O action (例如 ``errorExist :: IO () )檢查有沒有不合法的輸入，如果有便執行這個回報錯誤的 I/O action。我們之後會談另一個可能，就是用 exception。

## 亂數



在許多情況下，你寫程式會需要些隨機的資料。或許你在製作一個遊戲，在遊戲中你需要擲骰子。或是你需要測試程式的測試資料。精準一點地說，我們需要 pseudo-random 的資料，我們知道真正的隨機資料好比是一隻猴子拿著起司跟奶油騎在單輪車上，任何事情都會發生。在這個章節，我們要看看如何讓 Haskell 產生些 pseudo-random 的資料。

在大多數其他的程式語言中，會給你一些函數能讓你拿到些隨機亂數。每呼叫一次他就會拿到一個不同的數字。那在 Haskell 中是如何？要記住 Haskell 是一個純粹函數式語言。代表任何東西都具有 referential transparency。那代表你餵給一個函數相同的參數，不管怎麼呼叫都是回傳相同的結果。這很新奇的原因是因為他讓我們理解程式的方式不同，而且可以讓我們延遲計算，直到我們真正需要他。如果我呼叫一個函數，我可以確定他不會亂來。我真正在乎的是他的結果。然而，這會造成在亂數的情況有點複雜。如果我有一個函數像這樣：

```
randomNumber :: (Num a) => a
randomNumber = 4
```

由於他永遠回傳 4，所以對於亂數的情形而言是沒什麼意義。就算 4 這個結果是擲骰子來的也沒有意義。

其他的程式語言是怎麼產生亂數的呢？他們可能隨便拿取一些電腦的資訊，像是現在的時間，你怎麼移動你的滑鼠，以及周圍的聲音。根據這些算出一個數值讓他看起來好像隨機的。那些要素算出來的結果可能在每個時間都不同，所以你會拿到不同的隨機數字。

所以說在 Haskell 中，假如我們能作一個函數，他會接受一個具隨機性的參數，然後根據那些資訊還傳一個數值。

在 `System.Random` 模組中。他包含所有滿足我們需求的函數。讓我們先來看其中一個，就是 **random**。他的型態是 `random :: (RandomGen g, Random a) => g -> (a, g)`。哇，出現了新的 typeclass。**RandomGen** typeclass 是指那些可以當作亂源的型態。而**Random** typeclass 則是可以裝亂數的型態。一個布林值可以是隨機值，不是 `True` 就是 `False`。一個整數可以是隨機的好多不同值。那你會問，函數可以是一個隨機值嗎？我不這麼認為。如果我們試著翻譯 `random` 的型態宣告，大概會是這樣：他接受一個 random generator (亂源所在)，然後回傳一個隨機值以及一個新的 random generator。為什麼他要回傳一個新的 random generator 呢？就是下面我們要講的。

要使用 `random` 函數，我們必須要了解 random generator。在 `System.Random` 中有一個很酷的型態，叫做 **StdGen**，他是 `RandomGen` 的一個 instance。我們可以自己手動作一個 `StdGen` 也可以告訴系統給我們一個現成的。

要自己做一個 random generator，要使用 **mkStdGen** 這個函數。他的型態是 `mkStdGen :: Int -> StdGen`。他接受一個整數，然後根據這個整數會給一個 random generator。讓我們來試一下 `random` 以及 `mkStdGen`，用他們產生一個亂數吧。

```
ghci> random (mkStdGen 100)
```

```
<interactive>:1:0:
  Ambiguous type variable `a' in the constraint:
    `Random a' arising from a use of `random' at <interactive>:1:0-20
  Probable fix: add a type signature that fixes these type variable(s) `
```

這是什麼？由於 `random` 函數會回傳 `Random` typeclass 中任何一種型態，所以我們必須告訴 Haskell 我們是要哪一種型態。不要忘了我們是回傳 random value 跟 random generator 的一個 pair

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624, 651872571 1655838864)
```

我們終於有了一個看起來像亂數的數字。tuple 的第一個部份是我們的亂數，而第二個部份是一個新的 random generator 的文字表示。如果我們用相同的 random generator 再呼叫 `random` 一遍呢？

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624, 651872571 1655838864)
```

不易外地我們得到相同的結果。所以我們試試用不同的 random generator 作為我們的參數。

```
ghci> random (mkStdGen 949494) :: (Int, StdGen)
(539963926, 466647808 1655838864)
```

很好，我們拿到了不同的數字。我們可以用不同的型態標誌來拿到不同型態的亂數

```
ghci> random (mkStdGen 949488) :: (Float, StdGen)
(0.8938442, 1597344447 1655838864)
ghci> random (mkStdGen 949488) :: (Bool, StdGen)
(False, 1485632275 40692)
ghci> random (mkStdGen 949488) :: (Integer, StdGen)
(1691547873, 1597344447 1655838864)
```

讓我們寫一個模擬丟三次銅板的函數。假如 `random` 不同時回傳一個亂數以及一個新的 `random generator`，我們就必須讓這函數接受三個 `random generators` 讓他們每個回傳一個擲銅板的結果。但那樣聽起來怪怪的，加入一個 `generator` 可以產生一個型態是 `Int` 的亂數，他應該可以產生擲三次銅板的結果（總共才八個組合）。這就是 `random` 為什麼要回傳一個新的 `generator` 的關鍵了。

我們將一個銅板表示成 `Bool`。`True` 代表反面，`False` 代表正面。

```
threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen =
  let (firstCoin, newGen) = random gen
  (secondCoin, newGen') = random newGen
  (thirdCoin, newGen') = random newGen'
  in (firstCoin, secondCoin, thirdCoin)
```

我們用我們拿來當參數的 `generator` 呼叫 `random` 並得到一個擲銅板的結果跟一個新的 `generator`。然後我們再用新的 `generator` 呼叫他一遍，來得到第二個擲銅板的結果。對於第三個擲銅板的結果也是如法炮製。如果我們一直都用同樣的 `generator`，那所有的結果都會是相同的值。也就是不是 `(False, False, False)` 就是 `(True, True, True)`。

```
ghci> threeCoins (mkStdGen 21)
(True, True, True)
ghci> threeCoins (mkStdGen 22)
(True, False, True)
ghci> threeCoins (mkStdGen 943)
(True, False, True)
ghci> threeCoins (mkStdGen 944)
(True, True, True)
```

留意我們不需要寫 `random gen :: (Bool, StdGen)`。那是因為我們已經在函數的型態宣告那邊就表明我們要的是布林。而 Haskell 可以推敲出我們要的是布林值。

假如我們要的是擲四次？甚至五次呢？有一個函數叫 `randoms`，他接受一個 generator 並回傳一個無窮序列。

```
ghci> take 5 $ randoms (mkStdGen 11) :: [Int]
[-1807975507, 545074951, -1015194702, -1622477312, -502893664]
ghci> take 5 $ randoms (mkStdGen 11) :: [Bool]
[True, True, True, True, False]
ghci> take 5 $ randoms (mkStdGen 11) :: [Float]
[7.904789e-2, 0.62691015, 0.26363158, 0.12223756, 0.38291094]
```

為什麼 `randoms` 不另外多回傳一個新的 generator 呢？我們可以這樣地實作 `randoms'`

```
randoms' :: (RandomGen g, Random a) => g -> [a]
randoms' gen = let (value, newGen) = random gen in value : randoms' newGen
```

一個遞迴的定義。我們由現在的 generator 拿到一個亂數跟一個新的 generator，然後製作一個 list，list 的第一個值是那個亂數，而 list 的其餘部份是根據新的 generator 產生出的其餘亂數們。由於我們可能產生出無限的亂數，所以不可能回傳一個新的 generator。

我們可以寫一個函數，他會回傳有限個亂數跟一個新的 generator

```
finiteRandoms :: (RandomGen g, Random a, Num n, Eq n) => n -> g -> ([a], g)
finiteRandoms 0 gen = ([], gen)
finiteRandoms n gen =
  let (value, newGen) = random gen
      (restOfList, finalGen) = finiteRandoms (n-1) newGen
  in (value : restOfList, finalGen)
```

又是一個遞迴的定義。我們說如果我們要 0 個亂數，我們便回傳一個空的 list 跟原本給我們的 generator。對於其他數量的亂數，我們先拿一個亂數跟一個新的 generator。這一個亂數便是 list 的第一個數字。然後 list 中剩下的便是 n-1 個由新的 generator 產生出的亂數。然後我們回傳整個 list 跟最後一個產生完 n-1 個亂數後 generator。

如果我們要的是在某個範圍內的亂數呢？現在拿到的亂數要不是太大就是太小。如果我們想要的是骰子上的數字呢？`randomR` 能滿足我們的需求。他的型態是 `randomR :: (RandomGen g, Random a) :: (a, a) -> g -> (a, g)`，代表他有點類似 `random`。只不過他的第一個參數是一對數目，定義了最後產生亂數的上界以及下界。

```
ghci> randomR (1, 6) (mkStdGen 359353)
(6, 1494289578 40692)
ghci> randomR (1, 6) (mkStdGen 35935335)
(3, 1250031057 40692)
```

另外也有一個 `randomRs` 的函數，他會產生一連串在給定範圍內的亂數：

```
ghci> take 10 $ randomRs ('a','z') (mkStdGen 3) :: [Char]
"ndkxbvmmomg"
```

這結果看起來像是一個安全性很好的密碼。

你會問你自己，這一單元跟 I/O 有關係嗎？到現在為止還沒出現任何跟 I/O 有關的東西。到現在為止我們都是手動地做我們的 random generator。但那樣的問題是，程式永遠都會回傳同樣的亂數。這在真實世界中的程式是不能接受的。這也是為什麼 `System.Random` 要提供 `getStdGen` 這個 I/O action，他的型態是 `IO StdGen`。當你的程式執行時，他會跟系統要一個 random generator，並存成一個 global generator。`getStdGen` 會替你拿那個 global random generator 並把他綁定到某個名稱上。

這裡有一個簡單的產生隨機字串的程式。

```
import System.Random

main = do
    gen <- getStdGen
    putStrLn $ take 20 (randomRs ('a','z') gen)
```

```
$ runhaskell random_string.hs
pybphhzzhuepknbykxhe
$ runhaskell random_string.hs
eiqgcxykivpudlsvvjpg
$ runhaskell random_string.hs
nzdceoconysdgcyqjruo
$ runhaskell random_string.hs
bakzhnnuzrkgvesqlrx
```

要當心當我們連續兩次呼叫 `getStdGen` 的時候，實際上都會回傳同樣的 global generator。像這樣：

```
import System.Random

main = do
    gen <- getStdGen
    putStrLn $ take 20 (randomRs ('a','z') gen)
    gen2 <- getStdGen
    putStrLn $ take 20 (randomRs ('a','z') gen2)
```

你會打印出兩次同樣的字串。要能得到兩個不同的字串是建立一個無限的 stream，然後拿前 20 個字當作第一個字串，拿下 20 個字當作第二個字串。要這麼做，我們需要在 `Data.List` 中的 `splitAt` 函數。他會把一個 list 根據給定的 index 切成一個 tuple，tuple 的第一部份就是切斷的前半，第二個部份就是切斷的後半。

```

import System.Random
import Data.List

main = do
    gen <- getStdGen
    let randomChars = randomRs ('a','z') gen
        (first20, rest) = splitAt 20 randomChars
        (second20, _) = splitAt 20 rest
    putStrLn first20
    putStr second20

```

另一種方法是用 **newStdGen** 這個 I/O action，他會把現有的 random generator 分成兩個新的 generators。然後會把其中一個指定成 global generator，並回傳另一個。

```

import System.Random

main = do
    gen <- getStdGen
    putStrLn $ take 20 (randomRs ('a','z') gen)
    gen' <- newStdGen
    putStr $ take 20 (randomRs ('a','z') gen')

```

當我們綁定 **newStdGen** 的時候我們不只是會拿到一個新的 generator，global generator 也會被重新指定。所以再呼叫一次 **getStdGen** 並綁定到某個名稱的話，我們就會拿到跟 **gen** 不一樣的 generator。

這邊有一個小程序會讓使用者猜數字：

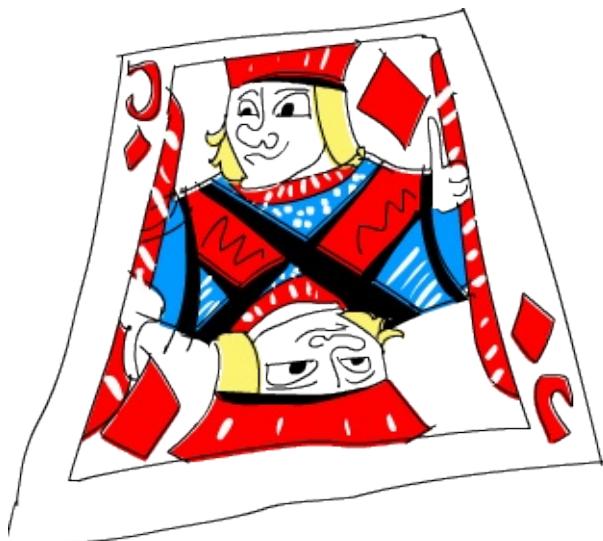
```

import System.Random
import Control.Monad(when)

main = do
    gen <- getStdGen
    askForNumber gen

askForNumber :: StdGen -> IO ()
askForNumber gen = do
    let (randNumber, newGen) = randomR (1,10) gen :: (Int, StdGen)
    putStrLn "Which number in the range from 1 to 10 am I thinking of? "
    numberString <- getLine
    when (not $ null numberString) $ do
        let number = read numberString
        if randNumber == number
            then putStrLn "You are correct!"
            else putStrLn $ "Sorry, it was " ++ show randNumber
    askForNumber newGen

```



我們寫了一個 `askForNumber` 的函數，他接受一個 random generator 並回傳一個問使用者要數字並回答是否正確的 I/O action。在那個函數裡面，我們先根據從參數拿到的 generator 產生一個亂數以及一個新的 generator，分別叫他們為 `randomNumber` 跟 `newGen`。假設那個產生的數字是 `7`。則我們要求使用者猜我們握有的數字是什麼。我們用 `getLine` 來將結果綁定到 `numberString` 上。當使用者輸入 `7`，`numberString` 就會是 `"7"`。接下來，我們用 `when` 來檢查使用者輸入的是否是空字串。如果是，那一個空的 I/O action `return ()` 就會被回傳。基本上就等於是結束程式的意思。如果不是，那 I/O action 就會被執行。我們用 `read` 來把 `numberString` 轉成一個數字，所以 `number` 便會是 `7`。

如果使用者給我們一些 ```read``` 沒辦法讀取的輸入（像是 ``"haha"``），我們的程式便會當掉並打印出錯誤訊息。

我們檢查如果輸入的數字跟我們隨機產生的數字一樣，便提示使用者恰當的訊息。然後再遞迴地呼叫 `askForNumber`，只是會拿到一個新的 generator。就像之前的 generator 一樣，他會給我們一個新的 I/O action。

`main` 的組成很簡單，就是由拿取一個 random generator 跟呼叫 `askForNumber` 組成罷了。

來看看我們的程式：

```
$ runhaskell guess_the_number.hs
Which number in the range from 1 to 10 am I thinking of? 4
Sorry, it was 3
Which number in the range from 1 to 10 am I thinking of? 10
You are correct!
Which number in the range from 1 to 10 am I thinking of? 2
Sorry, it was 4
Which number in the range from 1 to 10 am I thinking of? 5
Sorry, it was 10
Which number in the range from 1 to 10 am I thinking of?
```

用另一種方式寫的話像這樣：

```

import System.Random
import Control.Monad(when)

main = do
    gen <- getStdGen
    let (randNumber, _) = randomR (1,10) gen :: (Int, StdGen)
    putStrLn "Which number in the range from 1 to 10 am I thinking of? "
    numberString <- getLine
    when (not $ null numberString) $ do
        let number = read numberString
        if randNumber == number
            then putStrLn "You are correct!"
            else putStrLn $ "Sorry, it was " ++ show randNumber
    newStdGen
    main

```

他非常類似我們之前的版本，只是不是遞迴地呼叫，而是把所有的工作都在 `main` 裡面做掉。在告訴使用者他們猜得是否正確之後，便更新 global generator 然後再一次呼叫 `main`。兩種策略都是有效但我比較喜歡第一種方式。因為他在 `main` 裡面做的事比較少，並提供我們一個可以重複使用的函數。

## Bytestrings



List 是一種有用又酷的資料結構。到目前為止，我們幾乎無處不使用他。有好幾個函數是專門處理 List 的，而 Haskell 惰性的性質又讓我們可以用 filter 跟 map 來替換其他語言中的 for loop 跟 while loop。也由於 evaluation 只會發生在需要的時候，像 infinite list 也對於 Haskell

不成問題（甚至是 infinite list of infinite list）。這也是為什麼 list 能被用來表達 stream，像是讀取標準輸入或是讀取檔案。我們可以打開檔案然後讀取內容成字串，即便實際上我們是需要的時候才會真正取讀取。

然而，用字串來處理檔案有一個缺點：就是他很慢。就像你所知道的，`String` 是一個`[Char]` 的 type synonym。`Char` 沒有一個固定的大小，因為他可能由好幾個 byte 組成，好比說 Unicode。再加上 list 是惰性的。如果你有一個 list 像 `[1, 2, 3, 4]`，他只會在需要的時候被 evaluate。所以整個 list 實際比較像是一個"保證"你會有一個 list。要記住 `[1, 2, 3, 4]` 不過是 `1:2:3:4:[]` 的一個 syntactic sugar。當 list 的第一個元素被 evaluated 的時候，剩餘的部份 `2:3:4:[]` 一樣也只是一個"保證"你會有一個 list，以此類推。以此類推。所以你可以想像成 list 是保證在你需要的時候會給你第一個元素，以及保證你會有剩下的部份當你還需要更多的時候。其實不難說服你這樣做並不是一個最有效率的作法。

這樣額外的負擔在大多數時候不會造成困擾，但當我們要讀取一個很大的檔案的時候就是個問題了。這也是為什麼 Haskell 要有 `bytestrings`。Bytestrings 有點像 list，但他每一個元素都是一個 byte (8 bits)，而且他們惰性的程度也是不同。

Bytestrings 有兩種：strict 跟 lazy。Strict bytestrings 放在 `Data.ByteString`，他們把惰性的性質完全拿掉。不會有所謂任何的「保證」，一個 strict bytestring 就代表一連串的 bytes。因此你不會有一個無限長的 strict bytestrings。如果你 evaluate 第一個 byte，你就必須 evaluate 整個 bytestring。這麼做的優點是他會比較少 overhaed，因為他沒有 "Thunk"（也就是用 Haskell 術語來說的「保證」）。缺點就是他可能會快速消耗你的記憶體，因為你把他們一次都讀進了記憶體。

另一種 bytestring 是放在 `Data.ByteString.Lazy` 中。他們具有惰性，但又不像 list 那麼極端。就像我們之前說的，List 的 thunk 個數是跟 list 中有幾個元素一模一樣。這也是為什麼他們速度沒辦法滿足一些特殊需求。Lazy bytestrings 則用另一種作法，他們被存在 chunks 中（不要跟 Thunk 搞混），每一個 chunk 的大小是 64K。所以如果你 evaluate lazy bytestring 中的 byte，則前 64K 會被 evaluated。在那個 chunck 之後，就是一些「保證」會有剩餘的 chunk。lazy bytestrings 有點像裝了一堆大小為 64K 的 strict bytestrings 的 list。當你用 lazy bytestring 處理一個檔案的時候，他是一個 chunk 一個 chunk 去讀。這很棒是因為他不會讓我們一下使用大量的記憶體，而且 64K 有很高的可能性能夠裝進你 CPU 的 L2 Cache。

如果你大概看過 `Data.ByteString.Lazy` 的文件，你會看到到他有一堆函數的名稱跟 `Data.List` 中的函數名稱相同，只是出現的 type signature 是 `ByteString` 而不是 `[a]`，是 `Word8` 而不是 `a`。同樣名稱的函數基本上表現的行為跟 list 中的差不多。因為名稱是一樣的，所以必須用 qualified import 才不會在裝載進 GHCI 的時候造成衝突。

```
import qualified Data.ByteString.Lazy as B
import qualified Data.ByteString as S
```

`B` 中有 lazy bytestrings 跟對應的函數，而 `S` 中則有 strict 的版本。大多數時候我們是用 lazy 的版本。

**pack** 函數的 type signature 是 `pack :: [Word8] -> ByteString`。代表他接受一串型態為 `Word8` 的 bytes，並回傳一個 `ByteString`。你能想像一個 lazy 的 list，要讓他稍微不 lazy 一些，所以讓他對於 64K lazy。

那 `Word8` 型態又是怎麼一回事？。他就像 `Int`，只是他的範圍比較小，介於 0-255 之間。他代表一個 8-bit 的數字。就像 `Int` 一樣，他是屬於 `Num` 這個 typeclass。例如我們知道 `5` 是 polymorphic 的，他能夠表現成任何數值型態。其實 `Word8` 他也能表示。

```
ghci> B.pack [99, 97, 110]
Chunk "can" Empty
ghci> B.pack [98..120]
Chunk "bcdefghijklmnopqrstuvwxyz" Empty
```

正如你看到的，你其實不必特別在意 `Word8`，因為型態系統會選擇正確的型態。如果你試著用比較大的數字，像是 `336`。那對於 `Word8` 他就會變成 `80`。

我們把一些數值打包成 `ByteString`，使他們可以塞進一個 chunk 裡面。`Empty` 之於 `ByteString` 就像 `[]` 之於 `list` 一樣。

**unpack** 是 `pack` 的相反，他把一個 bytestring 變成一個 byte list。

**fromChunks** 接受一串 strict 的 bytestrings 並把他變成一串 lazy bytestring。**toChunks** 接受一個 lazy bytestrings 並將他變成一串 strict bytestrings。

```
ghci> B.fromChunks [S.pack [40, 41, 42], S.pack [43, 44, 45], S.pack [46, 47, 48]]
Chunk "(*" (Chunk "+, -" (Chunk "./0" Empty)))
```

如果你有很多小的 strict bytestrings 而且不想先將他們 join 起來（會耗損 memory）這樣的作法是不錯的。

bytestring 版本的 `:` 叫做 **cons**。他接受一個 byte 跟一個 bytestring，並把這個 byte 放到 bytestring 的前端。他是 lazy 的操作，即使 bytestring 的第一個 chunk 不是滿的，他也會新增一個 chunk。這也是為什麼當你要插入很多 bytes 的時候最好用 strict 版本的 `cons`，也就是 **cons'**。

```
ghci> B.cons 85 $ B.pack [80, 81, 82, 84]
Chunk "U" (Chunk "PQRT" Empty)
ghci> B.cons' 85 $ B.pack [80, 81, 82, 84]
Chunk "UPQRT" Empty
ghci> foldr B.cons B.empty [50..60]
Chunk "2" (Chunk "3" (Chunk "4" (Chunk "5" (Chunk "6" (Chunk "7" (Chunk "8" (Chunk "9" (C
Empty))))))))
ghci> foldr B.cons' B.empty [50..60]
Chunk "23456789:<" Empty
```

你可以看到 `empty` 製造了一個空的 bytestring。也注意到 `cons` 跟 `cons'` 的差異了嗎？有了 `foldr`，我們逐步地把一串數字從右邊開始，一個個放到 bytestring 的前頭。當我們用 `cons`，我們則得到一個 byte 一個 chunk 的結果，並不是我們要的。

bytestring 模組有一大票很像 `Data.List` 中的函數。包括了

```
head, tail, init, null, length, map, reverse, foldl, foldr, concat,
takeWhile, filter, 等等。
```

他也有表現得跟 `System.IO` 中一樣的函數，只有 `Strings` 被換成了 `ByteString` 而已。像是 `System.IO` 中的 `readFile`，他的型態是 `readFile :: FilePath -> IO String`，而 bytestring 模組中的 `readFile` 則是 `readFile :: FilePath -> IO ByteString`。小心，如果你用了 strict bytestring 來讀取一個檔案，他會把檔案內容都讀進記憶體中。而使用 lazy bytestring，他則會讀取 chunks。

讓我們來寫一個簡單的程式，他從命令列接受兩個檔案名，然後拷貝第一個檔案內容成第二個檔案。雖然 `System.Directory` 中已經有一個函數叫 `copyFile`，但我們想要實作自己的版本。

```
import System.Environment
import qualified Data.ByteString.Lazy as B

main = do
    (fileName1:fileName2:_) <- getArgs
    copyFile fileName1 fileName2

copyFile :: FilePath -> FilePath -> IO ()
copyFile source dest = do
    contents <- B.readFile source
    B.writeFile dest contents
```

我們寫了自己的函數，他接受兩個 `FilePath`（記住 `FilePath` 不過是 `String` 的同義詞。）並回傳一個 I/O action，他會用 bytestring 拷貝第一個檔案至另一個。在 `main` 函數中，我們做的只是拿到命令列引數然後呼叫那個函數來拿到一個 I/O action。

```
$ runhaskell bytestringcopy.hs something.txt ../../something.txt
```

就算我們不用 bytestring 來寫，程式最後也會長得像這樣。差別在於我們會用 `B.readFile` 跟 `B.writeFile` 而不是 `readFile` 跟 `writeFile`。有很大的可能性，就是你只要 import 檔案並在函數前加上 qualified 模組名，就可以把一個用正常 `String` 的程式改成用 `ByteString`。也有可能你是要反過來做，但那也不難。

當你需要更好的效能來讀取許多資料，嘗試用 bytestring，有很大的機會你會用很小的力氣改進很多效能。我通常用正常 `String` 來寫程式，然後在效能不好的時候把他們改成 `ByteString`。

# Exceptions (例外)



所有的程式語言都有要處理失敗的情形。這就是人生。不同的語言有不同的處理方式。在 C 裡面，我們通常用非正常範圍的回傳值（像是 `-1` 或 `null`）來回傳錯誤。Java 跟 C# 則傾向於使用 `exception` 來處理失敗的情況。當一個 `exception` 被丟出的時候，控制流程就會跳到我們做一些清理動作的地方，做完清理後 `exception` 被重新丟出，這樣一些處理錯誤的程式碼可以完成他們的工作。

Haskell 有一個很棒的型態系統。Algebraic data types 允許像是 `Maybe` 或 `Either` 這種型態，我們能用這些型態來代表一些可能有或沒有的結果。在 C 裡面，在失敗的時候回傳 `-1` 是很常見的事。但他只對寫程式的人有意義。如果我們不小心，我們有可能把這些錯誤碼當作正常值來處理，便造成一些混亂。Haskell 的型態系統賦予我們更安全的環境。一個 `a -> Maybe b` 的函數指出了他會產生一個包含 `b` 的 `Just`，或是回傳 `Nothing`。這型態跟 `a -> b` 是不同的，如果我們試著將兩個函數混用，compiler 便會警告我們。

儘管有表達力夠強的型態來輔助失敗的情形，Haskell 仍然支持 `exception`，因為 `exception` 在 I/O 的 contexts 下是比較合理的。在處理 I/O 的時候會有一堆奇奇怪怪的事情發生，環境是很不能被信賴的。像是打開檔案。檔案有可能被 lock 起來，也有可能檔案被移除了，或是整個硬碟都被拔掉。所以直接跳到處理錯誤的程式碼是很合理的。

我們了解到 I/O code 會丟出 `exception` 是件合理的事。至於 pure code 呢？其實他也能丟出 `Exception`。想想看 `div` 跟 `head` 兩個案例。他們的型態是 `(Integral a) => a -> a -> a` 以及 `[a] -> a`。`Maybe` 跟 `Either` 都沒有在他們的回傳型態中，但他們都有可能失敗。`div` 有可能除以零，而 `head` 有可能你傳給他一個空的 list。

```
ghci> 4 `div` 0
*** Exception: divide by zero
ghci> head []
*** Exception: Prelude.head: empty list
```



pure code 能丟出 Exception，但 Exception 只能在 I/O section 中被接到（也就是在 `main` 的 do block 中）這是因為在 pure code 中你不知道什麼東西什麼時候會被 evaluate。因為 lazy 特性的緣故，程式沒有一個特定的執行順序，但 I/O code 有。

先前我們談過為什麼在 I/O 部份的程式要越少越好。程式的邏輯部份盡量都放在 pure 的部份，因為 pure 的特性就是他們的結果只會根據函數的參數不同而改變。當思考 pure function 的時候，你只需要考慮他回傳什麼，因為除此之外他不會有任何副作用。這會讓事情簡單許多。儘管 I/O 的部份是難以避免的（像是打開檔案之類），但最好是把 I/O 部份降到最低。Pure functions 預設是 lazy，那代表我們不知道他什麼時候會被 evaluate，不過我們也不該知道。然而，一旦 pure functions 需要丟出 Exception，他們何時被 evaluate 就很重要了。那是因為我們只有在 I/O 的部份才能接到 Exception。這很糟糕，因為我們說過希望 I/O 的部份越少越好。但如果我們不接 Exception，我們的程式就會當掉。這問題有解決辦法嗎？答案是不要在 pure code 裡面使用 Exception。利用 Haskell 的型態系統，盡量使用 `Either` 或 `Maybe` 之類的型態來表示可能失敗的計算。

這也是為什麼我們要來看看怎麼使用 I/O Exception。I/O Exception 是當我們在 `main` 裡面跟外界溝通失敗而丟出的 Exception。例如我們嘗試打開一個檔案，結果發現他已經被刪掉或是其他狀況。來看看一個嘗試打開命令列引數所指定檔案名稱，並計算裡面有多少行的程式。

```

import System.Environment
import System.IO

main = do (fileName:_)<- getArgs
          contents <- readFile fileName
          putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"

```

一個很簡單的程式。我們使用 `getArgs` I/O action，並綁定第一個 string 到 `fileName`。然後我們綁定檔案內容到 `contents`。最後，我們用 `lines` 來取得 line 的 list，並計算 list 的長度，並用 `show` 來轉換數字成 string。他如我們想像的工作，但當我們給的檔案名稱不存在的時候呢？

```

$ runhaskell linecount.hs i_dont_exist.txt
linecount.hs: i_dont_exist.txt: openFile: does not exist (No such file or directory)

```

GHC 丟了錯誤訊息給我們，告訴我們檔案不存在。然後程式就掛掉了。假如我們希望打印出比較好一些的錯誤訊息呢？一種方式就是在打開檔案前檢查他存不存在。用

`System.Directory` 中的 **doesFileExist**。

```

import System.Environment
import System.IO
import System.Directory

main = do (fileName:_)<- getArgs
          fileExists <- doesFileExist fileName
          if fileExists
              then do contents <- readFile fileName
                      putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines"
              else do putStrLn "The file doesn't exist!"

```

由於 `doesFileExist` 的型態是 `doesFileExist :: FilePath -> IO Bool`，所以我們要寫成 `fileExists <- doesFileExist fileName`。那代表他回傳含有一個布林值告訴我們檔案存不存在的 I/O action。`doesFileExist` 是不能直接在 if expression 中使用的。

另一個解法是使用 Exception。在這個情境下使用 Exception 是沒問題的。檔案不存在這個 Exception 是在 I/O 中被丟出，所以在 I/O 中接起來也沒什麼不對。

要這樣使用 Exception，我們必須使用 `System.IO.Error` 中的 **catch** 函數。他的型態是 `catch :: IO a -> (IOError -> IO a) -> IO a`。他接受兩個參數，第一個是一個 I/O action。像是他可以接受一個打開檔案的 I/O action。第二個是 handler。如果第一個參數的 I/O action 丟出了 Exception，則他會被傳給 handler，他會決定要作些什麼。所以整個 I/O action 的結果不是如預期中做完第一個參數的 I/O action，就是 handler 處理的結果。



如果你對其他語言像是 Java, Python 中 try-catch 的形式很熟，那 catch 其實跟他們很像。第一個參數就是其他語言中的 try block。第二個參數就是其他語言中的 catch block。其中 handler 只有在 exception 被丟出時才會被執行。

handler 接受一個 IOError 型態的值，他代表的是一個 I/O exception 已經發生了。他也帶有一些 exception 本身的資訊。至於這型態在語言中使如何被實作則是要看編譯器。這代表我們沒辦法用 pattern matching 的方式來檢視 IOError。就像我們不能用 pattern matching 來檢視 IO something 的內容。但我們能用一些 predicate 來檢視他們。

我們來看看一個展示 catch 的程式

```
import System.Environment
import System.IO
import System.IO.Error

main = toTry `catch` handler

toTry :: IO ()
toTry = do (fileName:_)<- getArgs
           contents <- readFile fileName
           putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"

handler :: IOError -> IO ()
handler e = putStrLn "Whoops, had some trouble!"
```

首先你看到我們可以在關鍵字周圍加上 backticks 來把 catch 當作 infix function 用，因為他剛好接受兩個參數。這樣使用讓可讀性變好。toTry `catch` handler 跟 catch toTry handler 是一模一樣的。toTry 是一個 I/O action，而 handler 接受一個 IOError，並回傳一個當 exception 發生時被執行的 I/O action。

來看看執行的結果。

```
$ runhaskell count_lines.hs i_exist.txt
The file has 3 lines!

$ runhaskell count_lines.hs i_dont_exist.txt
Whoops, had some trouble!
```

在 `handler` 裡面我們並沒有檢查我們拿到的是什麼樣的 `IOError`，我們只是打印出 `"Whoops, had some trouble!"`。接住任何種類的 `Exception` 就跟其他語言一樣，在 Haskell 中也不是一個好的習慣。假如其他種類的 `Exception` 發生了，好比說我們送一個中斷指令，而我們沒有接到的話會發生什麼事？這就是為什麼我們要做跟其他語言一樣的事：就是檢查我們拿到的是什麼樣的 `Exception`。如果說是我們要的 `Exception`，那就做對應的處理。如果不是，我們再重新丟出 `Exception`。我們把我們的程式這樣修改，只接住檔案不存在的 `Exception`。

```
import System.Environment
import System.IO
import System.IO.Error

main = toTry `catch` handler

toTry :: IO ()
toTry = do (fileName:_)<- getArgs
           contents <- readFile fileName
           putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"

handler :: IOError -> IO ()
handler e
| isDoesNotExistError e = putStrLn "The file doesn't exist!"
| otherwise = ioError e
```

除了 `handler` 以外其他東西都沒變，我們只接住我們想要的 I/O exception。這邊使用了 `System.IO.Error` 中的函數 `isDoesNotExistError` 跟 `ioError`。`isDoesNotExistError` 是一個運作在 `IOError` 上的 predicate，他代表他接受一個 `IOError` 然後回傳 `True` 或 `False`，他的型態是 `isDoesNotExistError :: IOError -> Bool`。我們用他來判斷是否這個錯誤是檔案不存在所造成的。我們這邊使用 guard，但其實也可以用 `if else`。如果 `exception` 不是由於檔案不存在所造成的，我們就用 `ioError` 重新丟出接到的 `exception`。他的型態是 `ioError :: IOException -> IO a`，所以他接受一個 `IOError` 然後產生一個會丟出 `exception` 的 I/O action。那個 I/O action 的型態是 `IO a`，但他其實不會產生任何結果，所以他可以被當作是 `IO anything`。

所以有可能在 `toTry` 裡面丟出的 `exception` 並不是檔案不存在造成的，而 `toTry `catch` handler` 會接住再丟出來，很酷吧。

程式裡面有好幾個運作在 `IOError` 上的 I/O action，當其中一個沒有被 evaluate 成 `True` 時，就會掉到下一個 guard。這些 predicate 分別為：

```
* **isAlreadyExistsError**
* **isDoesNotExistError**
* **isFullError**
* **isEOFError**
* **isIllegalOperation**
* **isPermissionError**
* **isUserError**
```

大部分的意思都是顯而易見的。當我們用了 `userError` 來丟出 exception 的時候，`isUserError` 被 evaluate 成 `True`。例如說，你可以寫 `ioError $ userError "remote computer unplugged!"`，儘管用 `Either` 或 `Maybe` 來表示可能的錯誤會比自己丟出 exception 更好。

所以你可能寫一個像這樣的 handler

```
handler :: IOError -> IO ()
handler e
| isDoesNotExistError e = putStrLn "The file doesn't exist!"
| isFullError e = freeSomeSpace
| isIllegalOperation e = notifyCops
| otherwise = ioError e
```

其中 `notifyCops` 跟 `freeSomeSpace` 是一些你定義的 I/O action。如果 exception 不是你想要的，記得要把他們重新丟出，不然你的程式可能只會安靜地當掉。

`System.IO.Error` 也提供了一些能詢問 exception 性質的函數，像是哪些 handle 造成錯誤，或哪些檔案名造成錯誤。這些函數都是 `ioe` 當開頭。而且你可以在文件中看到一整串詳細資料。假設我們想要打印出造成錯誤的檔案名。我們不能直接打印出從 `getArgs` 那邊拿到的 `fileName`，因為只有 `IOError` 被傳進 `handler` 中，而 `handler` 並不知道其他事情。一個函數只依賴於他所被呼叫時的參數。這也是為什麼我們會用 `ioeGetFileName` 這函數，他的型態是 `ioeGetFileName :: IOError -> Maybe FilePath`。他接受一個 `IOError` 並回傳一個 `FilePath`（他是 `String` 的同義詞。）基本上他做的事就是從 `IOError` 中抽出檔案路徑。我們來修改一下我們的程式。

```

import System.Environment
import System.IO
import System.IO.Error

main = toTry `catch` handler

toTry :: IO ()
toTry = do (fileName:_)<- getArgs
    contents <- readFile fileName
    putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"

handler :: IOError -> IO ()
handler e
| isDoesNotExistError e =
    case ioeGetFileName e of Just path -> putStrLn $ "Whoops! File does not exist at:
                                                Nothing -> putStrLn "Whoops! File does not exist at unknown"
| otherwise = ioError e

```

在 `isDoesNotExistError` 是 `True` 的 guard 裡面，我們在 `case expression` 中用 `e` 來呼叫 `ioeGetFileName`，然後用 pattern matching 拆出 `Maybe` 中的值。當你想要用 pattern matching 却又不想要寫一個新的函數的時候，`case expression` 是你的好朋友。

你不想只用一個 `catch` 來接你 I/O part 中的所有 exception。你可以只在特定地方用 `catch` 接 exception，或你可以用不同的 handler。像這樣：

```

main = do toTry `catch` handler1
         thenTryThis `catch` handler2
         launchRockets

```

這邊 `toTry` 使用 `handler1` 當作 `handler`，而 `thenTryThis` 用了 `handler2`。`launchRockets` 並不是 `catch` 的參數，所以如果有任何一個 exception 被丟出都會讓我們的程式當掉，除非 `launchRockets` 使用 `catch` 來處理 exception。當然 `toTry`，`thenTryThis` 跟 `launchRockets` 都是 I/O actions，而且被 `do syntax` 繩在一起。這很像其他語言中的 try-catch blocks，你可以把一小段程式用 try-catch 包住，你可以自己調整該包多少進去。

現在你知道如何處理 I/O exception 了。我們並沒有提到如何從 pure code 中丟出 exception，這是因為正如我們先前提到的，Haskell 提供了更好的辦法來處理錯誤。就算是在可能會失敗的 I/O action 中，我也傾向用 `IO (Either a b)`，代表他們是 I/O action，但當他們被執行，他們結果的型態是 `Either a b`，意思是 `Left a` 就是 `Right b`。

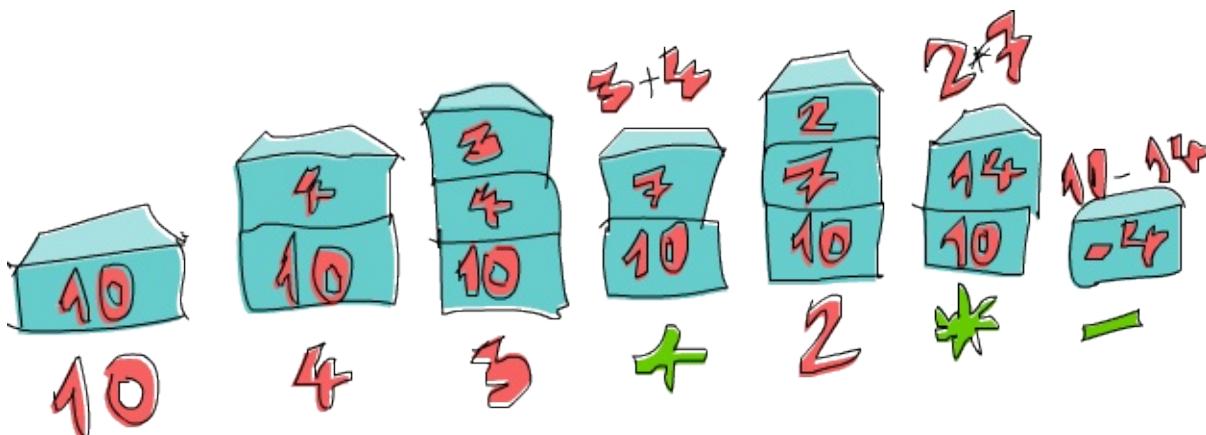
# 函數式地思考來解決問題

在這一章中，我們會檢視幾個有趣的問題，並嘗試用函數式的方式來漂亮地解決他們。我們並不會介紹新的概念，我們只是練習我們剛學到的寫程式的技巧。每一節都會探討不同的問題。會先描述問題，然後用最好的方式解決他。

## 運算逆波蘭表示法(Reverse Polish notation form)

我們在學校學習數學的時候，我們多半都是用中置(infix)的方式來寫數學式。例如說，我們會寫 `10 - (4 + 3) * 2`。`+`, `*`, `-` 是中置運算子(infix operators)。在 Haskell 中就像是 `+` 或 `elem` 一樣。這種寫法對於人類來說很容易閱讀與理解，但缺點是我們必須用括號來描述運算的優先順序。

逆波蘭表示法是另外一種數學式的描述方法。乍看之下顯得怪異，但他其實很容易理解並使用。因為我們不需要括弧來描述，也很容易放進計算機裡面運算。儘管現在的計算機都是用中置的方式讓你輸入，有些人仍堅持用 RPN 的計算機。前述的算式如果表達成 RPN 的話會是 `10 4 3 + 2 * -`。我們要如何計算他的結果呢？可以想想堆疊，基本上你是從左向右閱讀算式。每當碰到一個數值，就把他堆上堆疊。當我們碰到一個運算子。就把兩個數值從堆疊上拿下來，用運算子運算兩個數值然後把結果推回堆疊中。當你消耗完整個算式的時候，而且假設你的算式是合法的，那你就應該只剩一個數值在堆疊中，



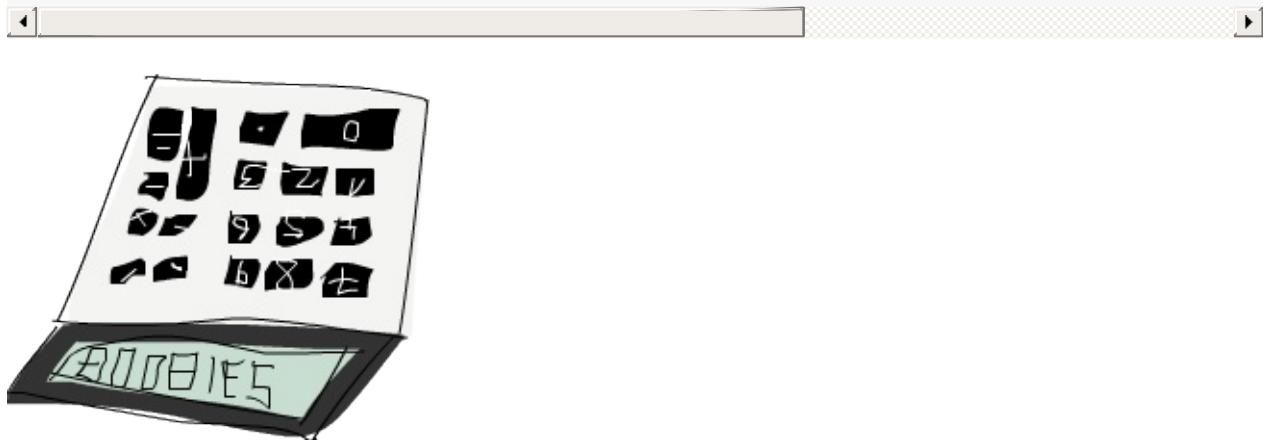
我們再接著看 `10 4 3 + 2 * -`。首先我們把 `10` 推到堆疊上，所以堆疊現在是 `10`。下一個接著的輸入是 `4`，我們也把他推上堆疊。堆疊的狀態便變成 `10, 4`。接著也對下一個輸入 `3` 做同樣的事，所以堆疊變成 `10, 4, 3`。然後便碰到了第一個運算子 `+`。我們把堆疊最上層的兩個數值取下來（所以堆疊變成 `10`）把兩個數值加起來然後推回堆疊上。堆疊的狀態便變成 `10, 7`。我們再把輸入 `2` 推上堆疊，堆疊變成 `10, 7, 2`。我們又碰到另一個運算子，所以把 `7` 跟 `2` 取下，把他們相乘起來然後推回堆疊上。`7` 跟 `2` 相乘的結果是

`14`，所以堆疊的狀態是 `10, 14`。最後我們碰到了 `-`。我們把 `10` 跟 `14` 取下，將他們相減然後推回堆疊上。所以現在堆疊的狀態變成 `-4`。而我們已經把所有數值跟運算子的消耗完了，所以 `-4` 便是我們的結果。

現在我們知道我們如何手算 RPN 運算式了，接下來可以思考一下我們寫一個 Haskell 的函數，當他接到一個 RPN 運算式，像是 `"10 4 3 + 2 * -"` 時，他可以給出結果。

這個函數的型別會是什麼樣呢？我們希望他接受一個字串當作參數，並產出一個數值作為結果。所以應該會是 `solveRPN :: (Num a) => String -> a`。

小建議：在你去實作函數之前，先想一下你會怎麼宣告這個函數的型別能夠幫助你釐清問題。在 Haskell 中由於我們有



當我們要實作一個問題的解法時，你可以先動手一步一步解看看，嘗試從裡面得到一些靈感。我們這邊把每一個用空白隔開的數值或運算子都當作獨立的一項。所以把 `"10 4 3 + 2 * -"` 這樣一個字串斷成一串 list `["10", "4", "3", "+", "2", "*", "-"]` 應該會有幫助。

接下來我們要如何應用這個斷好的 list 呢？我們從左至右來走一遍，並保存一個工作用的堆疊。這樣有讓你想到些什麼可以用的嗎？沒錯，在 `folds` 的那一章裡面，我們提到基本上當你需要從左至右或由右至左走過一遍 list 的時候並產生些結果的時候。我們都能用 `fold` 來實作他。

在這個 case 中由於我們是從左邊走到右邊，所以我們採取 left fold。accumulator 則是選用堆疊，而 `fold` 的結果也會是一個堆疊，只是裡面只有一個元素而已。

另外要多考慮一件事是我們用什麼來代表我們的堆疊？我們可以用 list 來代替，list 的 head 就可以當作是堆疊的頂端。畢竟要把一個元素加到 list 的 head 要比加到最後要有效率多。所以如果我們有一個堆疊，裡面有 `10, 4, 3`，那我們可以用 `[3, 4, 10]` 來代表他。

現在我們有了足夠的資訊來寫出我們的函數。他會接受一個字串 `"10 4 3 + 2 * -"`，隨即用 `words` 來斷成 list `["10", "4", "3", "+", "2", "*", "-"]`。接下來我們做一個 left fold 來產生出只有一個元素的堆疊，也就是 `[-4]`。我們把這個元素從 list 取出便是最後的結果。

來看看我們的實作：

```
import Data.List

solveRPN :: (Num a) => String -> a
solveRPN expression = head (foldl foldingFunction [] (words expression))
  where  foldingFunction stack item = ...
```

我們接受一個運算式並把他斷成一串 List。然後我們用一個 folding 函數來 fold 這串 list。注意到我們用 `[]` 來當作起始的 accumulator。這個 accumulator 就是我們的堆疊，所以 `[]` 代表一個空的堆疊。在運算之後我們得到一個只有一個元素的堆疊，我們呼叫 `head` 來取出他並用 `read` 來轉換他。

所以我們現在只缺一個接受堆疊的 folding 函數，像是可以接受 `[4, 10]` 跟 `"3"`，然後得到 `[3, 4, 10]`。如果是 `[4, 10]` 跟 `"+"`，那就會得到 `[40]`。但在實作之前，我們先把我們的函數改寫成 point-free style，這樣可以省下許多括號。

```
import Data.List

solveRPN :: (Num a) => String -> a
solveRPN = head . foldl foldingFunction [] . words
  where  foldingFunction stack item = ...
```

看起來好多了。我們的 folding 函數會接受一個堆疊、新的項，並回傳一個新的堆疊。我們使用模式匹配的方式來取出堆疊最上層的元素，然後對 `"+"` 跟 `"+"` 做匹配。

```
solveRPN :: (Num a, Read a) => String -> a
solveRPN = head . foldl foldingFunction [] . words
  where  foldingFunction (x:y:ys) "*" = (x * y):ys
         foldingFunction (x:y:ys) "+" = (x + y):ys
         foldingFunction (x:y:ys) "-" = (y - x):ys
         foldingFunction xs numberString = read numberString:xs
```

我們用展開成四個模式匹配。模式會從第一個開始嘗試匹配。所以 folding 函數會看看目前的項是否是 `"+"`。如果是，那就會將 `[3, 4, 9, 3]` 的頭兩個元素綁定到 `x`，`y` 兩個名稱。所以 `x` 會是 `3` 而 `y` 等於 `4`。`ys` 便會是 `[9, 3]`。他會回傳一個 list，只差在 `x` 跟 `y` 相乘的結果為第一個元素。也就是說會把最上層兩個元素取出，相乘後再放回去。如果第一個元素不是 `"+"`，那模式匹配就會比對到 `"+"`，以此類推。

如果項並未匹配到任何一個運算子，那我們就會假設這個字串是一個數值。如果他是一個數值，我們會用 `read` 來把字串轉換成數值。並把這個數值推到堆疊上。

另外注意到我們加了 `Read a` 這像 class constraint，畢竟我們要使用到 `read` 來轉換成數值。所以我們必須要宣告成他要屬於 `Num` 跟 `Read` 兩種 typeclass。（譬如說 `Int`，`Float` 等）

我們是從左至右走過 `["2", "3", "+"]`。一開始堆疊的狀態是 `[]`。首先他會用 `[]` 跟 `"2"` 來餵給 folding 函數。由於此項並不是一個運算子。他會用 `read` 讀取後加到 `[]` 的開頭。所以堆疊的狀態變成 `[2]`。接下來就是用 `[2]` 跟 `["3"]` 來餵給 folding 函數，而得到 `[3, 2]`。最後再用 `[3, 2]` 跟 `["+"]` 來呼叫 folding 函數。這會堆疊頂端的兩個數值，加起來後推回堆疊。最後堆疊變成 `[5]`，這就是我們回傳的數值。

我們來試試看我們新寫的函數：

```
ghci> solveRPN "10 4 3 + 2 * -"
-4
ghci> solveRPN "2 3 +"
5
ghci> solveRPN "90 34 12 33 55 66 + * - +"
-3947
ghci> solveRPN "90 34 12 33 55 66 + * - + -"
4037
ghci> solveRPN "90 34 12 33 55 66 + * - + -"
4037
ghci> solveRPN "90 3 -"
87
```

看起來運作良好。這個函數有一個特色就是他很容易改寫來支持額外的運算子。他們也不一定要是二元運算子。例如說我們可以寫一個運算子叫做 `"log"`，他會從堆疊取出一個數值算出他的 `log` 後推回堆疊。我們也可以用三元運算子來從堆疊取出三個數值，並把結果放回堆疊。甚至是像是 `"sum"` 這樣的運算子，取出所有數值並把他們的和推回堆疊。

我們來改寫一下我們的函數讓他多支援幾個運算子。為了簡單起見，我們改寫宣告讓他回傳 `Float` 型別。

```
import Data.List

solveRPN :: String -> Float
solveRPN = head . foldl foldingFunction [] . words
where  foldingFunction (x:y:ys) "*" = (x * y):ys
       foldingFunction (x:y:ys) "+" = (x + y):ys
       foldingFunction (x:y:ys) "-" = (y - x):ys
       foldingFunction (x:y:ys) "/" = (y / x):ys
       foldingFunction (x:y:ys) "^" = (y ** x):ys
       foldingFunction (x:xs) "ln" = log x:xs
       foldingFunction xs "sum" = [sum xs]
       foldingFunction xs numberString = read numberString:xs
```

看起來不錯，沒有疑問地 `/` 是除法而 `**` 是取 exponential。至於 `log` 運算子，我們只需要模式匹配一個元素，畢竟 `log` 只需要一個元素。而 `sum` 運算子，我們只回傳一個僅有一個元素的堆疊，包含了所有元素的和。

```
ghci> solveRPN "2.7 ln"
0.9932518
ghci> solveRPN "10 10 10 10 sum 4 /"
10.0
ghci> solveRPN "10 10 10 10 10 sum 4 /"
12.5
ghci> solveRPN "10 2 ^"
100.0
```

由於 `read` 知道如何轉換浮點數，我們也可在運算適中使用他。

```
ghci> solveRPN "43.2425 0.5 ^"
6.575903
```

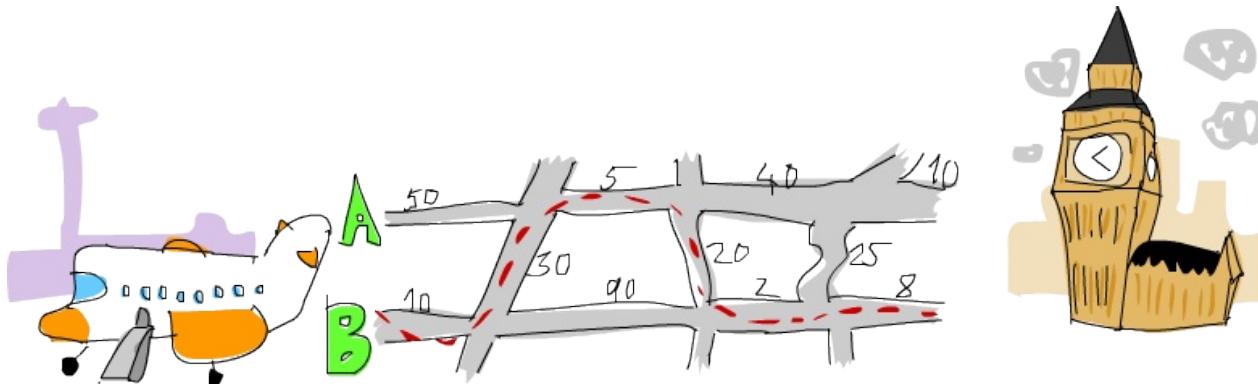
有這樣一個容易拓展到浮點數而且動到的程式碼又在十行以內的函數，我想是非常棒的。

有一件事要留意的是這個函數對於錯誤處理並不好。當我們碰到非法輸入的時候，他就會直接當掉。之後我們碰到 `Monad` 的時候我們會寫一個容錯的版本，他的型別會是 `solveRPN :: String -> Maybe Float`。當然我們現在也可以寫一個，不過那會有點麻煩，因為會有一大堆檢查 `Nothing` 的動作。如果你希望挑戰的話，也可以盡管嘗試。（提示：你可以用 `reads` 來看看一次 `read` 是否會成功）

## 路徑規劃

我們接下來的問題是：你的飛機剛剛降落在英格蘭的希思羅機場。你接下來有一個會議，你租了一台車希望盡速從機場前往倫敦市中心。

從希思羅機場到倫敦有兩條主要道路，他們中間有很多小路連接彼此。如果你要走小路的話都會花掉一定的時間。你的問題就是要選一條最佳路徑讓你可以盡快前往倫敦。你從圖的最左邊出發，中間可能穿越小路來前往右邊。



你可以從圖中看到，從希思羅機場到倫敦在這個路徑配置下的最短路徑是先選主要道路 B，經由小路到 A 之後，再走一小段，轉到 B 之後繼續往前走。如果採取這個路徑的話，會花去 75 分鐘。如果選其他道路的話，就會花更多時間。

我們任務就是要寫一個程式，他接受道路配置的輸入，然後印出對應的最短路徑。我們的輸入看起來像是這樣：

```
50
10
30
5
90
20
40
2
25
10
8
0
```

我們在心中可以把輸入的數值三個三個看作一組。每一組由道路 A, 道路 B, 還有交叉的小路組成。而要能夠這樣組成，我們必須讓最後有一條虛擬的交叉小路，只需要走 0 分鐘就可以穿越他。因為我們並不會在意在倫敦裡面開車的成本，畢竟我們已經到達倫敦了。

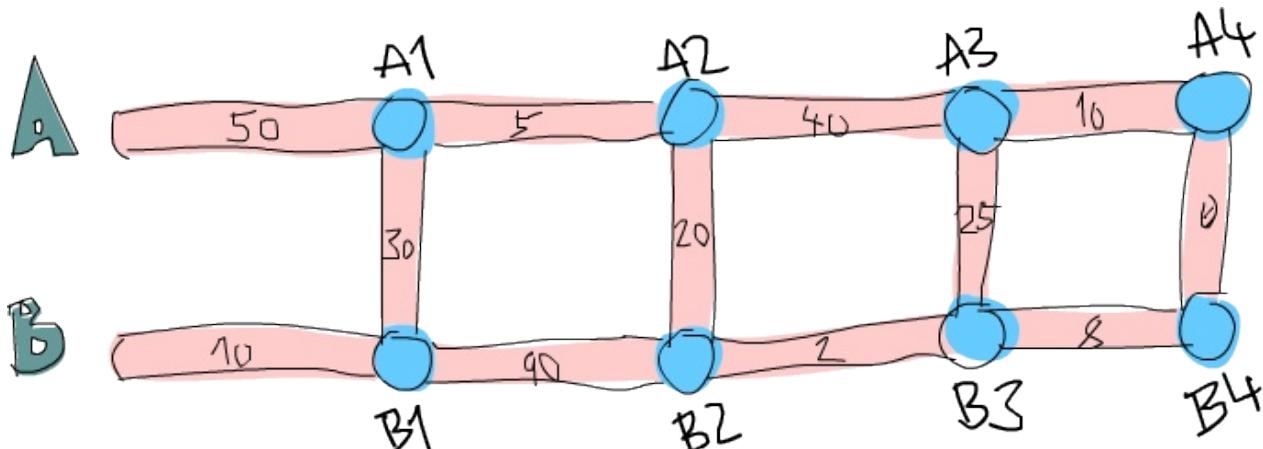
正如我們在解 RPN 計算機的問題的時候，我們是用三步驟來解題：

- 首先忘掉 Haskell，想想我們自己是怎麼一步步解題的。
- 想想如何在 Haskell 中表達我們的資料。
- 在 Haskell 中要如何對這些資料做運算來產生出解答。

在介紹 RPN 計算機的章節中，我們首先自己用人腦計算表達式，在心中維持一個堆疊然後一項一項處理。我們決定用一個字串來表達我們的表達式。最後，我們用 left fold 來走過我們這一串 list，並算出結果。

究竟我們要怎麼用手算出從希思羅機場到倫敦的最短路徑呢？我們可以觀察整章圖片，猜測哪一條是最短路徑然後希望我們有猜對。這樣的作法對於很小的輸入可以成功，但如果我們的路徑超過 10000 組呢？這樣我們不知道我們的解法是不是最佳解，我們只能說可能是。

所以那並不是一個好作法。這邊有一張簡化過後的圖。



你能想出來到道路 A 上第一個交叉點的最短路徑嗎？（標記成 A1 的點）這太容易了。我們只要看看從道路 A 出發或是從道路 B 出發穿越至道路 A 兩種作法哪種比較短就好。很明顯的，從道路 B 出發的比較短，只要花費 40 分鐘，然而從道路 A 則要花費 50 分鐘。那到交叉點 B1 呢？同樣的作法可以看出從道路 B 出發只要花費 10 分鐘，遠比從道路 A 出發然後穿越小路要花費少，後者要花費 80 分鐘！

現在我們知道要到達 A1 的最短路徑是經由 B 然後鄒小路到達，共花費 40。而我們知道要達到 B1 的最短路徑則是直接走 B，花費 10。這樣的知識有辦法幫助我們得知到下一個交叉點的最短路徑嗎？可以的。

我們來看看到達 A2 的最短路徑是什麼。要到達 A2，我們必須要從 A1 走到 A2 或是從 B1 走小路。由於我們知道到達 A1 跟 B1 的成本，我們可以很容易的想出到達 A2 的最佳路徑。到達 A1 要花費 40，而從 A1 到 A2 需要 5。所以 B, C, A 總共要 45。而要到達 B1 只要 10，但需要額外花費 110 分鐘來到達 B2 然後走小路到達 A2。所以最佳路徑就是 B, C, A。同樣地到達 B2 最好的方式就是走 A1 然後走小路。

也許你會問如果先在 B1 跨到道路 A 然後走到 A2 的情況呢？我們已經考慮過了從 B1 到 A1 的情況，所以我們不需

現在我們有了至 A2 跟 B2 的最佳路徑，我們可以一直重複這個過程直到最右邊。一旦我們到達了 A4 跟 B4，那其中比較短的就是我們的最佳路徑了。

基本上對於第二組而言，我們只是不斷地重複之前的步驟，只是我們考慮進在前面的最佳路徑而已。當然我們也可以說在第一步就考慮進了前面的最佳路徑，只是他們都是 0 而已。

總結一下。要得到從希思羅機場到倫敦的最短路徑。我們首先看看到達下一個道路 A 上的交叉點的最短路徑。共有兩種選擇的路徑，一是直接從道路 A 出發然後走到交叉點，要不然就是從道路 B 出發，走到第一個交叉點然後走小路。得到結果後記住結果。接著再用同樣的方法來得到走到道路 B 上下一個交叉點的最短路徑，並也記住結果。然後我們看看要走到再下一個道路 A 上的交叉點，究竟是從這個道路 A 上的交叉點往前走，或是從對應的道路 B 上的交叉點往前走再走到對面，兩種選擇哪種比較好。記下比較好的選擇，然後也對對應的道路 B 上的交叉點做一次這個過程。做完全部組之後就到達最右邊。一旦到達最右邊，最佳的選擇就是我們的最短路徑了。

基本上當我們到達最右邊的時候，我們記下了最後停在道路 A 的最短路徑跟最後停在道路 B 的最短路徑。其中比較短的是我們真正的最短路徑。現在我們已經知道怎麼用手算出答案。如果你有閒工夫，你可以拿紙筆對於任何一組道路配置算出他的最短路徑。

接下來的問題是，我們要如何用 Haskell 的型別來代表這裡的道路配置呢？一種方式就是把起始點跟交叉點都當作圖的節點，並連到其他的交叉點。如果我們想像其實起點也有一條長度為 1 的虛擬道路連接彼此，那每個交叉點或是節點就都連接對面的節點了。同時他們也連到下一個交叉點。唯一的例外是最後一個節點，他們只連接到對面。

```
data Node = Node Road Road | EndNode Road
data Road = Road Int Node
```

一個節點要碼是一個普通的節點，他包含有通往下一個交叉點的路徑資訊，還有往對面道路的資訊。或是一個終端點，只包含往對面節點的道路資訊。一條道路包含他多長，還有他指向哪裡。例如說，道路 A 的第一個部份就可寫成 `Road 50 a1`。其中 `a1` 是 `Node x y` 這樣一個節點。而 `x` 跟 `y` 則分別指向 `B1` 跟 `A2`。

另一種方式就是用 `Maybe` 來代表往下一個交叉點走的路。每個節點有指到對面節點的路徑部份，但只有不是終端節點的節點才有指向下一個交叉點的路。

```
data Node = Node Road (Maybe Road)
data Road = Road Int Node
```

這些是用 Haskell 來代表道路系統的方式，而我們也能靠他們來解決問題。但也許我們可以想出更簡單的模型？如果我們想想之前手算的方式，我們每次檢查都只有檢查三條路徑的長度而已。在道路 A 的部份，跟在道路 B 的部份，還有接觸兩個部份並將他們連接起來的部份。當我們觀察到 `A1` 跟 `B1` 的最短路徑時，我們只考慮第一組的三個部份，他們分別花費 `50, 10, 30`，`5, 90, 20`，`40, 2, 25` 跟 `10, 8, 0`。

讓我們資料型別越簡單越好，不過這樣已經是極限了。

```
data Section = Section { getA :: Int, getB :: Int, getC :: Int } deriving (Show)
type RoadSystem = [Section]
```

這樣很完美，而且對於我們的實作也有幫助。`Section` 是一個 algebraic data type，包含三個整數，分別代表三個不同部份的道路長。我們也定義了型別同義字，說 `RoadSystem` 代表包含 `section` 的 list。

當然我們也可以用一個 tuple ```(Int, Int, Int)``` 來代表一個 `section`。使用 tuple 對於一些簡單的情況是。

從希思羅機場到倫敦的道路系統便可以這樣表示：

```
heathrowToLondon :: RoadSystem
heathrowToLondon = [Section 50 10 30, Section 5 90 20, Section 40 2 25, Section 10 8 0]
```

我們現在要做的就是用 Haskell 實作我們先前的解法。所以我們應該怎樣宣告我們計算最短路徑函數的型別呢？他應該接受一個道路系統作為參數，然後回傳一個路徑。我們會用一個 list 來代表我們的路徑。我們定義了 `Label` 來表示 `A, B` 或 `C`。並且也定義一個同義詞

Path :

```
data Label = A | B | C deriving (Show)
type Path = [(Label, Int)]
```

而我們的函數 `optimalPath` 應該要有 `optimalPath :: RoadSystem -> Path` 這樣的型別。如果被餵給 `heathrowToLondon` 這樣的道路系統，他應該要回傳下列的路徑：

```
[(B, 10), (C, 30), (A, 5), (C, 20), (B, 2), (B, 8)]
```

我們接下來就從左至右來走一遍 list，並沿路上記下 A 的最佳路徑跟 B 的最佳路徑。我們會 `accumulate` 我們的最佳路徑。這聽起來有沒有很熟悉？沒錯！就是 left fold。

當我們手動做解答的時候，有一個步驟是我們不斷重複的。就是檢查現有 A 跟 B 的最佳路徑以及目前的 section，產生出新的 A 跟 B 的最佳路徑。舉例來說，最開始我們的最佳路徑是 `[]` 跟 `[]`。我們看過 `Section 50 10 30` 後就得到新的到 A1 的最佳路徑為 `[(B, 10), (C, 30)]`，而到 B1 的最佳路徑是 `[(B, 10)]`。如果你們把這個步驟看作是一個函數，他接受一對路徑跟一個 section，並產生出新的一對路徑。所以型別是 `(Path, Path) -> Section -> (Path, Path)`。我們接下來繼續實作這個函數。

提示：把 ```(Path, Path) -> Section -> (Path, Path)``` 當作 left fold 用的二元函數，fold 要求的型

```
roadStep :: (Path, Path) -> Section -> (Path, Path)
roadStep (pathA, pathB) (Section a b c) =
    let priceA = sum $ map snd pathA
        priceB = sum $ map snd pathB
        forwardPriceToA = priceA + a
        crossPriceToA = priceB + b + c
        forwardPriceToB = priceB + b
        crossPriceToB = priceA + a + c
        newPathToA = if forwardPriceToA <= crossPriceToA
                    then (A, a):pathA
                    else (C, c):(B, b):pathB
        newPathToB = if forwardPriceToB <= crossPriceToB
                    then (B, b):pathB
                    else (C, c):(A, a):pathA
    in (newPathToA, newPathToB)
```



上面的程式究竟寫了些什麼？首先他根據先前 A 的最佳解計算出道路 A 的最佳解，之後也如法炮製計算 B 的最佳解。使用 `sum $ map snd pathA`，所以如果 `pathA` 是 `[(A, 100), (C, 20)]`。`priceA` 就是 120。`forwardPriceToA` 就會是我們要付的成本。如果我們是從先前在 A 上的交叉點前往。那他就會等於我們至先前交叉點的最佳解加上目前 `section` 中 A 的部份。`crossPriceToA` 則是我們從先前在 B 上的交叉點前往 A 所要付出的代價。他是先前 B 的最佳解加上 `section` 中 B 的部份加上 C 的長。同樣地方式也可以決定 `forwardPriceToB` 跟 `crossPriceToB`。

現在我們知道了到 A 跟 B 的最佳路徑，我們需要根據這些資訊來構造到 A 跟 B 的整體路徑。如果直接走到 A 耗費較少的話，我們就把 `newPathToA` 設定成 `(A, a):pathA`。這樣做的事就是把 `Label A` 跟 `section` 的長度 `a` 接到最佳路徑的前面。要記得 `A` 是一個 `label`，而 `a` 的型別是 `Int`。我們為什麼要接在前面而不是 `pathA ++ [(A, a)]` 呢？因為接在 `list` 的前面比起接在後端要有效率多了。不過這樣產生出來的 `list` 就會相反。但要把 `list` 再反過來並不難。如果先走到 B 再穿越小路走到 A 比較短的話，那 `newPathToA` 就會包含這樣走的路線。同樣的道理也可以套用在 `newPathToB` 上。

最後我們回傳 `newPathToA` 跟 `newPathToB` 這一對結果。

我們把 `heathrowToLondon` 的第一個 `section` 餌給我們的函數。由於他是第一個 `section`，所以到 A 跟 B 的最佳路徑就是一對空的 `list`。

```
ghci> roadStep ([] , []) (head heathrowToLondon)
([(C, 30), (B, 10)], [(B, 10)])
```

要記住包含的路徑是反過來的，要從右邊往左邊讀。所以到 A 的最佳路徑可以解讀成從 B 出發，然後穿越到道路 A。而 B 的最佳路徑則是直接從 B 出發走到下一個交叉點。

優化小技巧：當我們寫 ```priceA = sum $ map snd pathA``` 的時候。我們是在計算每步的成本。如果我們實作



現在我們有了一個函數他接受一對路徑跟一個 section，並產生新的最佳路徑。我們可以用一個 left fold 來做。我們用 `([], [])` 跟第一個 section 來餵給 `roadStep` 並得到一對最佳路徑。然後他又被餵給這個新得到的最佳路徑跟下一個 section。以此類推。當我們走過全部的 section 的時候，我們就會得到一對最佳路徑，而其中比較短的那個就是解答。有了這樣的想法，我們便可以實作 `optimalPath`。

```
optimalPath :: RoadSystem -> Path
optimalPath roadSystem =
    let (bestAPath, bestBPath) = foldl roadStep (,[],[]) roadSystem
        in if sum (map snd bestAPath) <= sum (map snd bestBPath)
            then reverse bestAPath
            else reverse bestBPath
```

我們對 `roadSystem` 做 left fold。而用的起始 accumulator 是一對空的路徑。fold 的結果也是一對路徑，我們用模式匹配的方式來把路徑從結果取出。然後我們檢查哪一個路徑比較短便回傳他。而且在回傳之前也順便把整個結果反過來。因為我們先前提到的我們是用接在前頭的方式來構造結果的。

我們來測試一下吧！

```
ghci> optimalPath heathrowToLondon
[(B,10),(C,30),(A,5),(C,20),(B,2),(B,8),(C,0)]
```

這正是我們應該得到的結果！不過跟我們預期的結果仍有點差異，在最後有一步 `(C,0)`，那代表我們已經在倫敦了仍然跨越小路。不過由於他的成本是 0，所以依然可以算做正確的結果。

我們找出最佳路徑的函數，現在要做的只需要從標準輸入讀取文字形式道路系統，並把他轉成 `RoadSystem`，然後用 `optimalPath` 來把他跑一遍就好了。

首先，我們寫一個函數，他接受一串 list 並把他切成同樣大小的 group。我們命名他為 `groupOf`。當參數是 `[1..10]` 時，`groupOf 3` 應該回傳 `[[1,2,3],[4,5,6],[7,8,9],[10]]`。

```
groupsOf :: Int -> [a] -> [[a]]
groupsOf 0 _ = undefined
groupsOf _ [] = []
groupsOf n xs = take n xs : groupsOf n (drop n xs)
```

一個標準的遞迴函數。對於 `xs` 等於 `[1..10]` 且 `n` 等於 `3`，這可以寫成 `[1,2,3]`：  
`groupsOf 3 [4,5,6,7,8,9,10]`。當這個遞迴結束的時候，我們的 `list` 就三個三個分好組。而下列是我們的 `main` 函數，他從標準輸入讀取資料，構造 `RoadSystem` 並印出最短路徑。

```
import Data.List

main = do
    contents <- getContents
    let threes = groupsOf 3 (map read $ lines contents)
        roadSystem = map (\[a,b,c] -> Section a b c) threes
        path = optimalPath roadSystem
        pathString = concat $ map (show . fst) path
        pathPrice = sum $ map snd path
    putStrLn $ "The best path to take is: " ++ pathString
    putStrLn $ "The price is: " ++ show pathPrice
```

首先，我們從標準輸入獲取所有的資料。然後我們呼叫 `lines` 來把 `"50\n10\n30\n..."` 轉換成 `["50", "10", "30"...,`，然後我們 `map read` 來把這些轉成包含數值的 `list`。我們呼叫 `groupsOf 3` 來把 `list` 的 `list`，其中子 `list` 長度為 3。我們接著對這個 `list` 來 `map` 一個 `lambda` (`\[a,b,c] -> Section a b c`)。正如你看到的，這個 `lambda` 接受一個長度為 3 的 `list` 然後把他變成 `Section`。所以 `roadSystem` 現在就是我們的道路配置，而且是正確的型別 `RoadSystem`。我們呼叫 `optimalPath` 而得到一個路徑跟對應的代價，之後再印出來。

我們將下列文字存成檔案。

```
50
10
30
5
90
20
40
2
25
10
8
0
```

存成一個叫 `paths.txt` 的檔案然後餵給我們的程式。

```
$ cat paths.txt | runhaskell heathrow.hs
The best path to take is: BCACBBC
The price is: 75
```

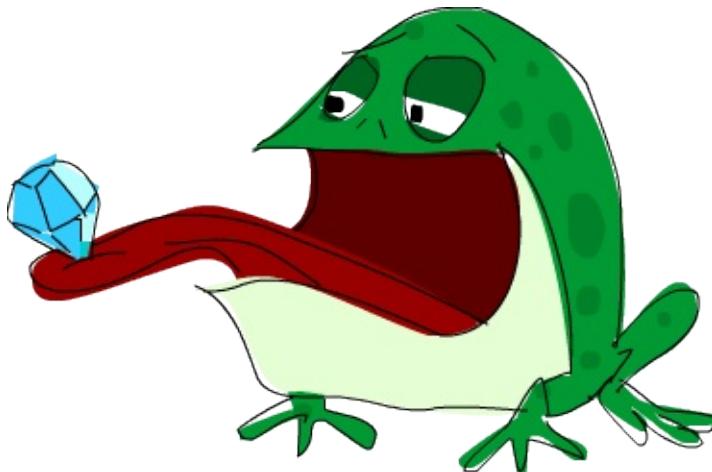
執行成功！你可以用你對 `Data.Random` 的瞭解來產生一個比較大的路徑配置，然後你可以把產生的亂數資料餵給你的程式。如果你碰到堆疊溢出，試試看用 `foldl'` 而不要用 `foldl`。`foldl'` 是 strict 的可以減少記憶體消耗。

# Functors, Applicative Functors 與 Monoids

Haskell 的一些特色，像是純粹性，高階函數，algebraic data types，typeclasses，這些讓我們可以從更高的角度來看到 polymorphism 這件事。不像 OOP 當中需要從龐大的型態階層來思考。我們只需要看看手邊的型態的行為，將他們跟適當地 typeclass 對應起來就可以了。像 `Int` 的行為跟很多東西很像。好比說他可以比較相不相等，可以從大到小排列，也可以將他們一一窮舉出來。

Typeclass 的運用是很隨意的。我們可以定義自己的資料型態，然後描述他可以怎樣被操作，跟 typeclass 關聯起來便定義了他的行為。由於 Haskell 強大的型態系統，這讓我們只要讀函數的型態宣告就可以知道很多資訊。typeclass 可以定義得很抽象很 general。我們之前有看過 typeclass 定義了可以比較兩個東西是否相等，或是定義了可以比較兩個東西的大小。這些是既抽象但又描述簡潔的行為，但我們不會認為他們有什麼特別之處，因為我們時常碰到他們。最近我們看過了 functor，基本上他們是一群可以被 map over 的物件。這是其中一個例子能夠抽象但又漂亮地描述行為。在這一章中，我們會詳加闡述 functors，並會提到比較強一些的版本，也就是 applicative functors。我們也會提到 monoids。

## 溫習 Functors



我們已經在之前的章節提到 functors。如果你還沒讀那個章節，也許你應該先去看看。或是你直接假裝你已經讀過了。

來快速複習一下：Functors 是可以被 map over 的物件，像是 lists, `Maybe`, `trees` 等等。在 Haskell 中我們是用 `Functor` 這個 typeclass 來描述他。這個 typeclass 只有一個 method，叫做 `fmap`，他的型態是 `fmap :: (a -> b) -> f a -> f b`。這型態說明了如果給我一個從 `a` 映到 `b` 的函數，以及一個裝了 `a` 的盒子，我會回給你一個裝了 `b` 的盒子。就好像用這個函數將每個元素都轉成 `b` 一樣

\*給一點建議\*。這盒子的比喻嘗試讓你抓到些 functors 是如何運作的感覺。在之後我們也會用相同的比喻來比喻 applicative functors。

如果一個 type constructor 要是 Functor 的 instance，那他的 kind 必須是  $* \rightarrow *$ ，這代表他必須剛好接受一個 type 當作 type parameter。像是 Maybe 可以是 Functor 的一個 instance，因為他接受一個 type parameter，來做成像是 Maybe Int，或是 Maybe String。如果一個 type constructor 接受兩個參數，像是 Either，我們必須給他兩個 type parameter。所以我們不能這樣寫：`instance Functor Either where`，但我們可以寫 `instance Functor (Either a) where`，如果我們把 `fmap` 限縮成只是 `Either a` 的，那他的型態就是 `fmap :: (b -> c) -> Either a b -> Either a c`。就像你看到的，`Either a` 的是固定的一部分，因為 `Either a` 只恰好接受一個 type parameter，但 `Either` 則要接受兩個 type parameters。這樣 `fmap` 的型態變成 `fmap :: (b -> c) -> Either b -> Either c`，這不太合理。

我們知道有許多型態都是 Functor 的 instance，像是 `[]`，`Maybe`，`Either a` 以及我們自己寫的 `Tree`。我們也看到了如何用一個函數 map 他們。在這一章節，我們再多舉兩個例子，也就是 `IO` 跟 `(->) r`。

如果一個值的型態是 `IO String`，他代表的是一個會被計算成 `String` 結果的 I/O action。我們可以用 do syntax 來把結果綁定到某個名稱。我們之前把 I/O action 比喻做長了腳的盒子，會到真實世界幫我們取一些值回來。我們可以檢視他們取了什麼值，但一旦看過，我們必須要把值放回盒子中。用這個比喻，`IO` 的行為就像是一個 functor。

我們來看看 `IO` 是怎麼樣的一個 Functor instance。當我們 `fmap` 用一個 function 來 map over I/O action 時，我們會想要拿回一個裝著已經用 function 映射過值的 I/O action。

```
instance Functor IO where
    fmap f action = do
        result <- action
        return (f result)
```

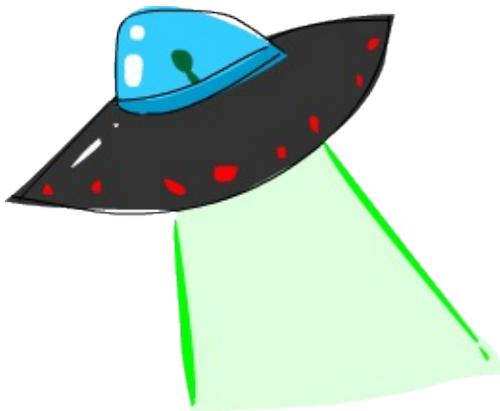
對一個 I/O action 做 map over 動作的結果仍會是一個 I/O action，所以我們才用 do syntax 來把兩個 I/O action 黏成一個。在 `fmap` 的實作中，我們先執行了原本傳進的 I/O action，並把結果綁定成 `result`。然後我們寫了 `return (f result)`。`return` 就如你所知道的，是一個只會回傳包了你傳給他東西的 I/O action。還有一個 do block 的回傳值一定是他最後一個 I/O action 的回傳值。這也是為什麼我們需要 `return`。其實他只是回傳包了 `f result` 的 I/O action。

我們可以再多實驗一下來找到些感覺。來看看這段 code：

```
main = do line <- getLine
          let line' = reverse line
              putStrLn $ "You said " ++ line' ++ " backwards!"
              putStrLn $ "Yes, you really said" ++ line' ++ " backwards!"
```

這程式要求使用者輸入一行文字，然後印出一行反過來的。我們可以用 `fmap` 來改寫：

```
main = do line <- fmap reverse getLine
          putStrLn $ "You said " ++ line ++ " backwards!"
          putStrLn $ "Yes, you really said" ++ line ++ " backwards!"
```



就像我們用 `fmap` `reverse` 來 map over `Just "blah"` 會得到 `Just "halb"`，我們也可以用 `fmap` `reverse` 來 map over `getLine`。`getLine` 是一個 I/O action，他的 type 是 `IO String`，而用 `reverse` 來 map over 他會回傳一個取回一個字串並 `reverse` 他的 I/O action。就像我們 apply 一個 function 到一個 `Maybe` 一樣，我們也可以 apply 一個 function 到一個 `IO`，只是這個 `IO` 會跑去外面拿回某些值。然後我們把結果用 `<-` 綁定到某個名稱，而這個名稱綁定的值是已經 `reverse` 過了。

而 `fmap (++!"") getLine` 這個 I/O action 表現得就像 `getLine`，只是他的結果多了一個 `"!"` 在最後。

如果我們限縮 `fmap` 到 `IO` 型態上，那 `fmap` 的型態是 `fmap :: (a -> b) -> IO a -> IO b`。`fmap` 接受一個函數跟一個 I/O action，並回傳一個 I/O action 包含了已經 apply 過 function 的結果。

如果你曾經注意到你想要將一個 I/O action 綁定到一個名稱上，只是為了要 apply 一個 function。你可以考慮使用 `fmap`，那會更漂亮地表達這件事。或者你想要對 functor 中的資料做 transformation，你可以先將你要用的 function 寫在 top level，或是把他作成一個 lambda function，甚至用 function composition。

```
import Data.Char
import Data.List

main = do line <- fmap (intersperse '-' . reverse . map toUpper) getLine
          putStrLn line
```

```
$ runhaskell fmapping_io.hs
hello there
E-R-E-H-T- -O-L-L-E-H
```

正如你想的，`intersperse '-' . reverse . map toUpper` 合成了一個 function，他接受一個字串，將他轉成大寫，然後反過來，再用 `intersperse '-'` 安插'-'。他是比較漂亮版本的 `(\xs -> intersperse '-' (reverse (map toUpper xs)))`。

另一個 `Functor` 的案例是 `(->) r`，只是我們先前沒有注意到。你可能會困惑到底 `(->) r` 究竟代表什麼？一個 `r -> a` 的型態可以寫成 `(->) r a`，就像是 `2 + 3` 可以寫成 `(+) 2 3` 一樣。我們可以從一個不同的角度來看待 `(->) r a`，他其實只是一個接受兩個參數的 type constructor，好比 `Either`。但記住我們說過 `Functor` 只能接受一個 type constructor。這也是為什麼 `(->)` 不是 `Functor` 的一個 instance，但 `(->) r` 則是。如果程式的語法允許的話，你也可以將 `(->) r` 寫成 `(r ->)`。就如 `(2+)` 代表的其實是 `(+) 2`。至於細節是如何呢？我們可以看看 `Control.Monad.Instances`。

我們通常說一個接受任何東西以及回傳隨便一個東西的函數型態是 ```a -> b```。```r -> a``` 是同樣意思，只是把符號換了。

```
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))
```

如果語法允許的話，他可以被寫成

```
instance Functor (r ->) where
    fmap f g = (\x -> f (g x))
```

但其實是不允許的，所以我們必須寫成第一種的樣子。

首先我們來看看 `fmap` 的型態。他的型態是 `fmap :: (a -> b) -> f a -> f b`。我們把所有的 `f` 在心裡代換成 `(->) r`。則 `fmap` 的型態就變成 `fmap :: (a -> b) -> ((->) r a) -> ((->) r b)`。接著我們把 `(->) r a` 跟 `(->) r b` 換成 `r -> a` 跟 `r -> b`。則我們得到 `fmap :: (a -> b) -> (r -> a) -> (r -> b)`。

從上面的結果看到將一個 function map over 一個 function 會得到另一個 function，就如 map over 一個 function 到 Maybe 會得到一個 Maybe，而 map over 一個 function 到一個 list 會得到一個 list。而 `fmap :: (a -> b) -> (r -> a) -> (r -> b)` 告訴我們什麼？他接受一個從 a 到 b 的 function，跟一個從 r 到 a 的 function，並回傳一個從 r 到 b 的 function。這根本就是 function composition。把 `r -> a` 的輸出接到 `a -> b` 的輸入，的確是 function composition 在做的事。如果你再仔細看看 instance 的定義，會發現真的就是一個 function composition。

```
instance Functor ((->) r) where
    fmap = (.)
```

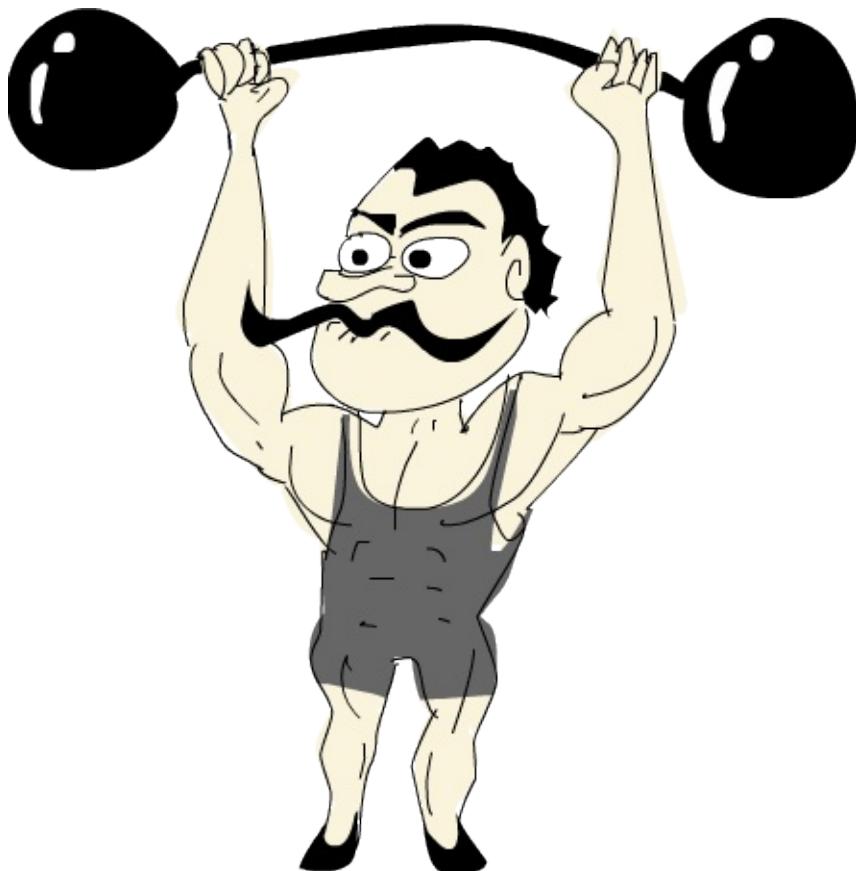
這很明顯就是把 `fmap` 當 composition 在用。可以用 `:m + Control.Monad.Instances` 把模組裝載進來，並做一些嘗試。

```
ghci> :t fmap (*3) (+100)
fmap (*3) (+100) :: (Num a) => a -> a
ghci> fmap (*3) (+100) 1
303
ghci> (*3) `fmap` (+100) $ 1
303
ghci> (*3) . (+100) $ 1
303
ghci> fmap (show . (*3)) (*100) 1
"300"
```

我們呼叫 `fmap` 的方式是 infix 的方式，這跟 `.` 很像。在第二行，我們把 `(*3)` map over 到 `(+100)` 上，這會回傳一個先把輸入值 `(+100)` 再 `(*3)` 的 function，我們再用 `1` 去呼叫他。

到這邊為止盒子的比喻還適用嗎？如果你硬是要解釋的話還是解釋得通。當我們將 `fmap` `(*3)` map over `Just 3` 的時候，對於 `Maybe` 我們很容易把他想成是裝了值的盒子，我們只是對盒子裡面的值 `(*3)`。但對於 `fmap (*3) (+100)` 呢？你可以把 `(+100)` 想成是一個裝了值的盒子。有點像把 I/O action 想成長了腳的盒子一樣。對 `(+100)` 使用 `fmap (*3)` 會產生另一個表現得像 `(+100)` 的 function。只是在算出值之前，會再多計算 `(*3)`。這樣我們可以看出來 `fmap` 表現得就像 `.` 一樣。

`fmap` 等同於 function composition 這件事對我們來說並不是很實用，但至少是一個有趣的觀點。這也讓我們打開視野，看到盒子的比喻不是那麼恰當，functors 實際比較像 computation。function 被 map over 到一個 computation 會產生經由那個 function 映射過後的 computation。



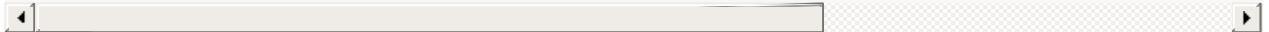
在我們繼續看 `fmap` 該遵守的規則之前，我們再看一次 `fmap` 的型態，他是 `fmap :: (a -> b) -> f a -> f b`。很明顯我們是在討論 Functor，所以為了簡潔，我們就不寫 `(Functor f) =>` 的部份。當我們在學 curry 的時候，我們說過 Haskell 的 function 實際上只接受一個參數。一個型態是 `a -> b -> c` 的函數實際上是接受 `a` 然後回傳 `b -> c`，而 `b -> c` 實際上接受一個 `b` 然後回傳一個 `c`。如果我們用比較少的參數呼叫一個函數，他就會回傳一個函數需要接受剩下的參數。所以 `a -> b -> c` 可以寫成 `a -> (b -> c)`。這樣 curry 可以明顯一些。

同樣的，我們可以不要把 `fmap` 想成是一個接受 function 跟 functor 並回傳一個 function 的 function。而是想成一個接受 function 並回傳一個新的 function 的 function，回傳的 function 接受一個 functor 並回傳一個 functor。他接受 `a -> b` 並回傳 `f a -> f b`。這動作叫做 lifting。我們用 GHCI 的 `:t` 來做的實驗。

```
ghci> :t fmap (*2)
fmap (*2) :: (Num a, Functor f) => f a -> f a
ghci> :t fmap (replicate 3)
fmap (replicate 3) :: (Functor f) => f a -> f [a]
```

`fmap (*2)` 接受一個 functor `f`，並回傳一個基於數字的 functor。那個 functor 可以是 list，可以是 `Maybe`，可以是 `Either String`。`fmap (replicate 3)` 可以接受一個基於任何型態的 functor，並回傳一個基於 list 的 functor。

當我們提到 functor over numbers 的時候，你可以想像他是一個 functor 包含有許多數字在裡面。前面一種說法



這樣的觀察在我們只有綁定一個部份套用的函數，像是 `fmap (++!"!")`，的時候會顯得更清楚，

你可以把 `fmap` 想做是一個函數，他接受另一個函數跟一個 functor，然後把函數對 functor 每一個元素做映射，或你可以想做他是一個函數，他接受一個函數並把他 lift 到可以在 functors 上面操作。兩種想法都是正確的，而且在 Haskell 中是等價。

`fmap (replicate 3) :: (Functor f) => f a -> f [a]` 這樣的型態代表這個函數可以運作在任何 functor 上。至於確切的行為則要看究竟我們操作的是什麼樣的 functor。如果我們是用 `fmap (replicate 3)` 對一個 list 操作，那我們會選擇 `fmap` 針對 list 的實作，也就是只是一個 `map`。如果我們是碰到 `Maybe a`。那他在碰到 `Just` 型態的時候，會對裡面的值套用 `replicate 3`。而碰到 `Nothing` 的時候就回傳 `Nothing`。

```
ghci> fmap (replicate 3) [1,2,3,4]
[[1,1,1],[2,2,2],[3,3,3],[4,4,4]]
ghci> fmap (replicate 3) (Just 4)
Just [4,4,4]
ghci> fmap (replicate 3) (Right "blah")
Right ["blah","blah","blah"]
ghci> fmap (replicate 3) Nothing
Nothing
ghci> fmap (replicate 3) (Left "foo")
Left "foo"
```

接下來我們來看看 functor laws。一個東西要成為 functor，必須要遵守某些定律。不管任何一個 functor 都被要求具有某些性質。他們必須是能被 map over 的。對他們呼叫 `fmap` 應該是要用一個函數 map 每一個元素，不多做任何事情。這些行為都被 functor laws 所描述。對於 `Functor` 的 instance 來說，總共兩條定律應該被遵守。不過他們不會在 Haskell 中自動被檢查，所以你必須自己確認這些條件。

functor law 的第一條說明，如果我們對 functor 做 map `id`，那得到的新的 functor 應該要跟原來的一樣。如果寫得正式一點，他代表 `fmap id = id`。基本上他就是說對 functor 呼叫 `fmap id`，應該等同於對 functor 呼叫 `id` 一樣。畢竟 `id` 只是 identity function，他只會把參數照原樣丟出。他也可以被寫成 `\x -> x`。如果我們對 functor 的概念就是可以被 map over 的物件，那 `fmap id = id` 的性就顯而易見。

我們來看看這個定律的幾個案例：

```
ghci> fmap id (Just 3)
Just 3
ghci> id (Just 3)
Just 3
ghci> fmap id [1..5]
[1,2,3,4,5]
ghci> id [1..5]
[1,2,3,4,5]
ghci> fmap id []
[]
ghci> fmap id Nothing
Nothing
```

如果我們看看 `Maybe` 的 `fmap` 的實作，我們不難發現第一定律為何被遵守。

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

我們可以想像在 `f` 的位置擺上 `id`。我們看到 `fmap id` 拿到 `Just x` 的時候，結果只不過是 `Just (id x)`，而 `id` 有只回傳他拿到的東西，所以可以知道 `Just (id x)` 等價於 `Just x`。所以說我們可以知道對 `Maybe` 中的 `Just` 用 `id` 做 `map over` 的動作，會拿回一樣的值。

而將 `id` `map over` `Nothing` 會拿回 `Nothing` 並不稀奇。所以從這兩個 `fmap` 的實作，我們可以看到的確 `fmap id = id` 有被遵守。



第二定律描述說先將兩個函數合成並將結果 *map over* 一個 *functor* 的結果，應該跟先將第一個函數 *map over* 一個 *functor*，再將第二個函數 *map over* 那個 *functor* 的結果是一樣的。正式地寫下來的話就是  $fmap(f \cdot g) = fmap f \cdot fmap g$ 。或是用另外一種寫法，對於任何一個 *functor*  $F$ ，下面這個式子應該要被遵守： $fmap(f \cdot g) F = fmap f (fmap g F)$ 。

如果我們能夠證明某個型別遵守兩個定律，那我們就可以保證他跟其他 *functor* 對於映射方面都擁有相同的性質。我們知道如果對他用 `fmap`，我們知道不會有除了 *mapping* 以外的事會發生，而他就僅僅會表現成某個可以被 *map over* 的東西。也就是一個 *functor*。你可以再仔細檢視 `fmap` 對於某些型別的實作來了解第二定律。正如我們先前對 `Maybe` 檢視第一定律一般。

如果你需要的話，我們能在這邊演練一下 `Maybe` 是如何遵守第二定律的。首先 `fmap(f \cdot g)` 來 *map over* `Nothing` 的話，我們會得到 `Nothing`。因為用任何函數來 `fmap Nothing` 的話都會回傳 `Nothing`。如果我們 `fmap f (fmap g Nothing)`，我們會得到 `Nothing`。可以看到當面對 `Nothing` 的時候，`Maybe` 很顯然是遵守第二定律的。那對於 `Just something` 呢？如果我們使用 `fmap(f \cdot g)(Just x)` 的話，從實作的程式碼中我可以看到 `Just(f \cdot g x)`，也就是 `Just(f(g x))`。如果我們使用 `fmap f (fmap g (Just x))` 的話我們可以從實作知道 `fmap g (Just x)` 會是 `Just(g x)`。`fmap f (fmap g (Just x))` 跟 `fmap f (Just(g x))` 相等。而從實作上這又會相等於 `Just(f(g x))`。

如果你不太理解這邊的說明，別擔心。只要確定你了解什麼是函數合成就好。在多數的情況下你可以直覺地對應到這些型別表現得就像 *containers* 或函數一樣。或是也可以換種方法，只要多嘗試對型別中不同的值做操作你就可以看看型別是否有遵守定律。

我們來看一些經典的例子。這些型別建構子雖然是 `Functor` 的 `instance`，但實際上他們並不是 `functor`，因為他們並不遵守這些定律。我們來看看其中一個型別。

```
data CMaybe a = CNothing | CJust Int a deriving (Show)
```

`C` 這邊代表的是計數器。他是一種看起來像是 `Maybe a` 的型別，只差在 `Just` 包含了兩個 `field` 而不是一個。在 `CJust` 中的第一個 `field` 是 `Int`，他是扮演計數器用的。而第二個 `field` 則為型別 `a`，他是從型別參數來的，而他確切的型別當然會依據我們選定的 `CMaybe a` 而定。我們來對他作些操作來獲得些操作上的直覺吧。

```
ghci> CNothing
CNothing
ghci> CJust 0 "haha"
CJust 0 "haha"
ghci> :t CNothing
CNothing :: CMaybe a
ghci> :t CJust 0 "haha"
CJust 0 "haha" :: CMaybe [Char]
ghci> CJust 100 [1,2,3]
CJust 100 [1,2,3]
```

如果我們使用 `CNothing`，就代表不含有 `field`。如果我們用的是 `CJust`，那第一個 `field` 是整數，而第二個 `field` 可以為任何型別。我們來定義一個 `Functor` 的 `instance`，這樣每次我們使用 `fmap` 的時候，函數會被套用在第二個 `field`，而第一個 `field` 會被加一。

```
instance Functor CMaybe where
  fmap f CNothing = CNothing
  fmap f (CJust counter x) = CJust (counter+1) (f x)
```

這種定義方式有點像是 `Maybe` 的定義方式，只差在當我們使用 `fmap` 的時候，如果碰到的不是空值，那我們不只會套用函數，還會把計數器加一。我們可以來看一些範例操作。

```
ghci> fmap (++) "ha" (CJust 0 "ho")
CJust 1 "hoha"
ghci> fmap (++) "he" (fmap (++) "ha" (CJust 0 "ho"))
CJust 2 "hohahe"
ghci> fmap (++) "blah" CNothing
CNothing
```

這些會遵守 `functor laws` 嗎？要知道有不遵守的情形，只要找到一個反例就好了。

```
ghci> fmap id (CJust 0 "haha")
CJust 1 "haha"
ghci> id (CJust 0 "haha")
CJust 0 "haha"
```

我們知道 functor law 的第一定律描述當我們用 `id` 來 map over 一個 functor 的時候，他的結果應該跟只對 functor 呼叫 `id` 的結果一樣。但我們可以看到這個例子中，這對於 `cMaybe` 並不遵守。儘管他的確是 `Functor` typeclass 的一個 instance。但他並不遵守 functor law 因此不是一個 functor。如果有人使用我們的 `cMaybe` 型別，把他當作 functor 用，那他就會期待 functor laws 會被遵守。但 `cMaybe` 並沒辦法滿足，便會造成錯誤的程式。當我們使用一個 functor 的時候，函數合成跟 map over 的先後順序不應該有影響。但對於 `cMaybe` 他是有影響的，因為他紀錄了被 map over 的次數。如果我們希望 `cMaybe` 遵守 functor law，我們必須要讓 `Int` 欄位在做 `fmap` 的時候維持不變。

乍看之下 functor laws 看起來不是很必要，也容易讓人搞不懂，但我們知道如果一個型別遵守 functor laws，那我們就能對他作些基本的假設。如果遵守了 functor laws，我們知道對他做 `fmap` 不會做多餘的事情，只是用一個函數做映射而已。這讓寫出來的程式碼足夠抽象也容易擴展。因為我們可以用定律來推論型別的行為。

所有在標準函式庫中的 `Functor` 的 instance 都遵守這些定律，但你可以自己檢查一遍。下一次你定義一個型別為 `Functor` 的 instance 的時候，花點時間確認他確實遵守 functor laws。一旦你操作過足夠多的 functors 時，你就會獲得直覺，知道他們會有什麼樣的性質跟行為。而且 functor laws 也會覺得顯而易見。但就算沒有這些直覺，你仍然可以一行一行地來找看看有沒有反例讓這些定律失效。

我們可以把 functor 看作輸出具有 context 的值。例如說 `Just 3` 就是輸出 `3`，但他又帶有一個可能沒有值的 context。`[1, 2, 3]` 輸出三個值，`1`，`2` 跟 `3`，同時也帶有可能有多個值或沒有值的 context。`(+3)` 則會帶有一個依賴於參數的 context。

如果你把 functor 想做是輸出值這件事，那你可以把 map over 一個 functor 這件事想成在 functor 輸出的後面再多加一層轉換。當我們做 `fmap (+3) [1, 2, 3]` 的時候，我們是把 `(+3)` 接到 `[1, 2, 3]` 後面，所以當我們檢視任何一個 list 的輸出的時候，`(+3)` 也會被套用在上面。另一個例子是對函數做 map over。當我們做 `fmap (+3) (*3)`，我們是把 `(+3)` 這個轉換套用在 `(*3)` 後面。這樣想的話會很自然就會把 `fmap` 跟函數合成關聯起來 (`fmap (+3) (*3)` 等價於 `(+3) . (*3)`，也等價於 `\x -> ((x * 3) + 3)`)，畢竟我們是接受一個函數 `(*3)` 然後套用 `(+3)` 轉換。最後的結果仍然是一個函數，只是當我們餵給他一個數字的時候，他會先乘上三然後做轉換加上三。這基本上就是函數合成在做的事。

## Applicative functors



在這個章節中，我們會學到 applicative functors，也就是加強版的 functors，在 Haskell 中是用在 `Control.Applicative` 中的 `Applicative` 這個 typeclass 來定義的。

你還記得 Haskell 中函數預設就是 Curried 的，那代表接受多個參數的函數實際上是接受一個參數然後回傳一個接受剩餘參數的函數，以此類推。如果一個函數的型別是 `a -> b -> c`，我們通常會說這個函數接受兩個參數並回傳 `c`，但他實際上是接受 `a` 並回傳一個 `b -> c` 的函數。這也是為什麼我們可以用 `(f x) y` 的方式呼叫 `f x y`。這個機制讓我們可以 partially apply 一個函數，可以用比較少的參數呼叫他們。可以做成一個函數再餵給其他函數。

到目前為止，當我們要對 functor map over 一個函數的時候，我們用的函數都是只接受一個參數的。但如果我們要 map 一個接受兩個參數的函數呢？我們來看幾個具體的例子。如果我們有 `Just 3` 然後我們做 `fmap (*) (Just 3)`，那我們會獲得什麼樣的結果？從 `Maybe` 對 `Functor` 的 instance 實作來看，我們知道如果他是 `Just something`，他會對在 `Just` 中的 `something` 做映射。因此當 `fmap (*) (Just 3)` 會得到 `Just ((* 3))`，也可以寫做 `Just (* 3)`。我們得到了一個包在 `Just` 中的函數。

```
ghci> :t fmap (++) (Just "hey")
fmap (++) (Just "hey") :: Maybe ([Char] -> [Char])
ghci> :t fmap compare (Just 'a')
fmap compare (Just 'a') :: Maybe (Char -> Ordering)
ghci> :t fmap compare "A LIST OF CHARS"
fmap compare "A LIST OF CHARS" :: [Char -> Ordering]
ghci> :t fmap (\x y z -> x + y / z) [3,4,5,6]
fmap (\x y z -> x + y / z) [3,4,5,6] :: (Fractional a) => [a -> a -> a]
```

如果我們 map `compare` 到一個包含許多字元的 list 呢？他的型別是 `(Ord a) => a -> a -> Ordering`，我們會得到包含許多 `Char -> Ordering` 型別函數的 list，因為 `compare` 被 partially apply 到 list 中的字元。他不是包含許多 `(Ord a) => a -> Ordering` 的函數，因為第一個 `a` 碰到的型別是 `Char`，所以第二個 `a` 也必須是 `Char`。

我們看到如何用一個多參數的函數來 map functor，我們會得到一個包含了函數的 functor。那現在我們能對這個包含了函數的 functor 做什麼呢？我們能用一個吃這些函數的函數來 map over 這個 functor，這些在 functor 中的函數都會被當作參數丟給我們的函數。

```
ghci> let a = fmap (*) [1,2,3,4]
ghci> :t a
a :: [Integer -> Integer]
ghci> fmap (\f -> f 9) a
[9,18,27,36]
```

但如果我們的有一個 functor 裡面是 `Just (3 *)` 還有另一個 functor 裡面是 `Just 5`，但我們想要把第一個 `Just (3 *)` map over `Just 5` 呢？如果是普通的 functor，那就沒救了。因為他們只允許 map 一個普通的函數。即使我們用 `\f -> f 9` 來 map 一個裝了很多函數的 functor，我們也是使用了普通的函數。我們是無法單純用 `fmap` 來把包在一個 functor 的函數 map 另一個包在 functor 中的值。我們能用模式匹配 `Just` 來把函數從裡面抽出來，然後再 map `Just 5`，但我們是希望有一個一般化的作法，對任何 functor 都有效。

我們來看看 `Applicative` 這個 typeclass。他位在 `Control.Applicative` 中，在其中定義了兩個函數 `pure` 跟 `<*>`。他並沒有提供預設的實作，如果我們想使用他必須要為他們 applicative functor 的實作。typeclass 定義如下：

```
class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

這簡簡單單的三行可以讓我們學到不少。首先來看第一行。他開啟了 `Applicative` 的定義，並加上 class constraint。描述了一個型別構造子要是 `Applicative`，他必須也是 `Functor`。這就是為什麼我們說一個型別構造子屬於 `Applicative` 的話，他也會是 `Functor`，因此我們能對他使用 `fmap`。

第一個定義的是 `pure`。他的型別宣告是 `pure :: a -> f a`。`f` 代表 applicative functor 的 instance。由於 Haskell 有一個優秀的型別系統，其中函數又是將一些參數映射成結果，我們可以從型別宣告中讀出許多訊息。`pure` 應該要接受一個值，然後回傳一個包含那個值的 applicative functor。我們這邊是用盒子來作比喻，即使有一些比喻不完全符合現實的情況。儘管這樣，`a -> f a` 仍有許多豐富的資訊，他確實告訴我們他會接受一個值並回傳一個 applicative functor，裡面裝有結果。

對於 `pure` 比較好的說法是把一個普通值放到一個預設的 context 下，一個最小的 context 但仍然包含這個值。

`<*>` 也非常有趣。他的型別是 `f (a -> b) -> f a -> f b`。這有讓你聯想到什麼嗎？沒錯！就是 `fmap :: (a -> b) -> f a -> f b`。他有點像加強版的 `fmap`。然而 `fmap` 接受一個函數跟一個 functor，然後套用 functor 之中的函數。`<*>` 則是接受一個裝有函數的 functor 跟另

一個 functor，然後取出第一個 functor 中的函數將他對第二個 functor 中的值做 map。

我們來看看 `Maybe` 的 `Applicative` 實作：

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something
```

從 class 的定義我們可以看到 `f` 作為 applicative functor 會接受一個具體型別當作參數，所以我們是寫成 `instance Applicative Maybe where` 而不是寫成 `instance Applicative (Maybe a) where`。

首先看到 `pure`。他只不過是接受一個東西然後包成 applicative functor。我們寫成 `pure = Just` 是因為 `Just` 不過就是一個普通函數。我們其實也可以寫成 `pure x = Just x`。

接著我們定義了 `<*>`。我們無法從 `Nothing` 中抽出一個函數，因為 `Nothing` 並不包含一個函數。所以我們說如果我們要嘗試從 `Nothing` 中取出一個函數，結果必定是 `Nothing`。如果你看看 `Applicative` 的定義，你會看到他有 `Functor` 的限制，他代表 `<*>` 的兩個參數都會是 functors。如果第一個參數不是 `Nothing`，而是一個裝了函數的 `Just`，而且我們希望將這個函數對第二個參數做 map。這個也考慮到第二個參數是 `Nothing` 的情況，因為 `fmap` 任何一個函數至 `Nothing` 會回傳 `Nothing`。

對於 `Maybe` 而言，如果左邊是 `Just`，那 `<*>` 會從其中抽出了一個函數來 map 右邊的值。如果有任何一個參數是 `Nothing`。那結果便是 `Nothing`。

來試試看吧！

```
ghci> Just (+3) <*> Just 9
Just 12
ghci> pure (+3) <*> Just 10
Just 13
ghci> pure (+3) <*> Just 9
Just 12
ghci> Just (++"hahah") <*> Nothing
Nothing
ghci> Nothing <*> Just "woot"
Nothing
```

我們看到 `pure (+3)` 跟 `Just (+3)` 在這個 case 下是一樣的。如果你是在 applicative context 底下跟 `Maybe` 打交道的話請用 `pure`，要不然就用 `Just`。前四個輸入展示了函數是如何被取出並做 map 的動作，但在這個 case 底下，他們同樣也可以用 `unwrap` 函數來 map over functors。最後一行比較有趣，因為我們試著從 `Nothing` 取出函數並將他 map 到某個值。結果當然是 `Nothing`。

對於普通的 functors，你可以用一個函數 map over 一個 functors，但你可能沒辦法拿到結果。而 applicative functors 則讓你可以用單一一個函數操作好幾個 functors。看看下面一段程式碼：

```
ghci> pure (+) <*> Just 3 <*> Just 5
Just 8
ghci> pure (+) <*> Just 3 <*> Nothing
Nothing
ghci> pure (+) <*> Nothing <*> Just 5
Nothing
```



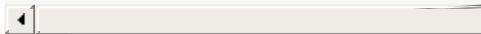
究竟我們寫了些什麼？我們來一步一步看一下。`<*>` 是 left-associative，也就是說 `pure (+) <*> Just 3 <*> Just 5` 可以寫成 `(pure (+) <*> Just 3) <*> Just 5`。首先 `+` 是擺在一個 functor 中，在這邊剛好他是一個 Maybe。所以首先，我們有 `pure (+)`，他等價於 `Just (+)`。接下來由於 partial application 的關係，`Just (+) <*> Just 3` 等價於 `Just (3+)`。把一個 `3` 餵給 `+` 形成另一個只接受一個參數的函數，他的效果等於加上 `3`。最後 `Just (3+) <*> Just 5` 被運算，其結果是 `Just 8`。

這樣很棒吧！用 applicative style 的方式來使用 applicative functors。像是 `pure f <*> x <*> y <*> ...` 就讓我們可以拿一個接受多個參數的函數，而且這些參數不一定是被包在 functor 中。就這樣來套用在多個在 functor context 的值。這個函數可以吃任意多的參數，畢竟 `<*>` 只是做 partial application 而已。

如果我們考慮到 `pure f <*> x` 等於 `fmap f x` 的話，這樣的用法就更方便了。這是 applicative laws 的其中一條。我們稍後會更仔細地檢視這條定律。現在我們先依直覺來使用他。就像我們先前所說的，`pure` 把一個值放進一個預設的 context 中。如果我們要把一個函數放在一個預設的 context，然後把他取出並套用在放在另一個 applicative functor 的值。我們會做的事就是把函數 map over 那個 applicative functor。但我們不會寫成 `pure f <*> x <*> y <*> ...`，而是寫成 `fmap f x <*> y <*> ...`。這也是為什麼 `Control.Applicative` 會 export 一個函數 `<$>`，他基本上就是中綴版的 `fmap`。他是這麼被定義的：

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
f <$> x = fmap f x
```

要記住型別變數跟參數的名字還有值綁定的名稱不衝突。```f``` 在函數的型別宣告中是型別變數，說明 ```f``` 應該要滿



`<$>` 的使用顯示了 applicative style 的好處。如果我們想要將 `f` 套用三個 applicative functor。我們可以寫成 `f <$> x <*> y <*> z`。如果參數不是 applicative functor 而是普通值的話。我們則寫成 `f x y z`。

我們再仔細看看他是如何運作的。我們有一個 `Just "johntra"` 跟 `Just "volta"` 這樣的值，我們希望將他們結合成一個 `String`，並且包含在 `Maybe` 中。我們會這樣做：

```
ghci> (++) <$> Just "johntra" <*> Just "volta"
Just "johntravolta"
```

可以將上面的跟下面這行比較一下：

```
ghci> (++) "johntra" "volta"
"johntravolta"
```

可以將一個普通的函數套用在 applicative functor 上真不錯。只要稍微寫一些 `<$>` 跟 `<*>` 就可以把函數變成 applicative style，可以操作 applicatives 並回傳 applicatives。

總之當我們在做 `(++) <$> Just "johntra" <*> Just "volta"` 時，首先我們將 `(++)` map over 到 `Just "johntra"`，然後產生 `Just ("johntra"++)`，其中 `(++)` 的型別為 `(++) :: [a] -> [a] -> [a]`，`Just ("johntra"++)` 的型別為 `Maybe ([Char] -> [Char])`。注意到 `(++)` 是如何吃掉第一個參數，以及我們是怎麼決定 `a` 是 `Char` 的。當我們做 `Just ("johntra"++) <*> Just "volta"`，他接受一個包在 `Just` 中的函數，然後 map over `Just "volta"`，產生了 `Just "johntravolta"`。如果兩個值中有任意一個為 `Nothing`，那整個結果就會是 `Nothing`。

到目前為止我們只有用 `Maybe` 當作我們的案例，你可能也會想說 applicative functor 差不多就等於 `Maybe`。不過其實有許多其他 `Applicative` 的 instance。我們來看看有哪些。

`List` 也是 applicative functor。很驚訝嗎？來看看我們是怎麼定義 `[]` 為 `Applicative` 的 instance 的。

```
instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x <- xs]
```

早先我們說過 `pure` 是把一個值放進預設的 context 中。換種說法就是一個會產生那個值的最小 context。而對 list 而言最小 context 就是 `[]`，但由於空的 list 並不包含一個值，所以我們沒辦法把他當作 `pure`。這也是為什麼 `pure` 其實是接受一個值然後回傳一個包含單元素的

list。同樣的，`Maybe` 的最小 context 是 `Nothing`，但他其實表示的是沒有值。所以 `pure` 實際上是被實作成 `Just` 的。

```
ghci> pure "Hey" :: [String]
["Hey"]
ghci> pure "Hey" :: Maybe String
Just "Hey"
```

至於 `<*>` 呢？如果我們假定 `<*>` 的型別是限制在 list 上的話，我們會得到 `(<*>) :: [a -> b] -> [a] -> [b]`。他是用 list comprehension 來實作的。`<*>` 必須要從左邊的參數取出函數，將他 map over 右邊的參數。但左邊的 list 有可能不包含任何函數，也可能包含一個函數，甚至是多個函數。而右邊的 list 有可能包含多個值。這也是為什麼我們用 list comprehension 的方式來從兩個 list 取值。我們要對左右任意的組合都做套用的動作。而得到的結果就會是左右兩者任意組合的結果。

```
ghci> [(*0,+100),(^2) ] <*> [1,2,3]
[0,0,0,101,102,103,1,4,9]
```

左邊的 list 包含三個函數，而右邊的 list 有三個值。所以結果會是有九個元素的 list。在左邊 list 中的每一個函數都被套用到右邊的值。如果我們今天在 list 中的函數是接收兩個參數的，我們也可以套用到兩個 list 上。

```
ghci> [(+),(*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]
```

由於 `<*>` 是 left-associative，也就是說 `[(+),(*)] <*> [1,2]` 會先運作，產生 `[(1+),(2+),  
(1*),(2*)]`。由於左邊的每一個函數都套用至右邊的每一個值。也就產生 `[(1+),(2+),(1*),  
(2*)] <*> [3,4]`，其便是最終結果。

list 的 applicative style 是相當有趣的：

```
ghci> (++) <$> ["ha","heh","hmm"] <*> ["?","!","."]
["ha?", "ha!", "ha.", "heh?", "heh!", "heh.", "hmm?", "hmm!", "hmm."]
```

看看我們是如何將一個接受兩個字串參數的函數套用到兩個 applicative functor 上的，只要用適當的 applicative 運算子就可以達成。

你可以將 list 看作是一個 non-deterministic 的計算。而對於像 `100` 或是 `"what"` 這樣的值則是 deterministic 的計算，只會有一個結果。而 `[1,2,3]` 則可以看作是沒有確定究竟是哪一種結果。所以他代表的是所有可能的結果。當你在做 `(+) <$> [1,2,3] <*> [4,5,6]`，你可以想做是把兩個 non-deterministic 的計算做 `+`，只是他會產生另一個 non-deterministic 的計算，而且結果更加不確定。

Applicative style 對於 list 而言是一個取代 list comprehension 的好方式。在第二章中，我們想要看到 `[2, 5, 10]` 跟 `[8, 10, 11]` 相乘的結果，所以我們這樣做：

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

我們只是從兩個 list 中取出元素，並將一個函數套用在任何元素的組合上。這也可以用 applicative style 的方式來寫：

```
ghci> (*) <$> [2,5,10] <*> [8,10,11]
[16,20,22,40,50,55,80,100,110]
```

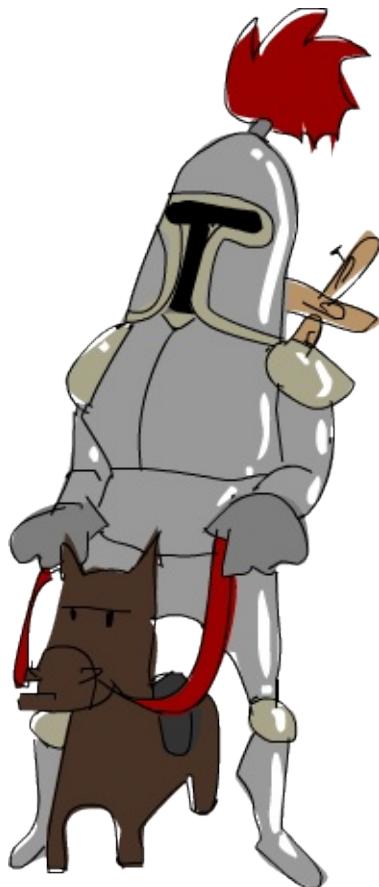
這寫法對我來說比較清楚。可以清楚表達我們是要對兩個 non-deterministic 的計算做 `*`。如果我們想要所有相乘大於 50 可能的計算結果，我們會這樣寫：

```
ghci> filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]
[55,80,100,110]
```

很容易看到 `pure f <*> xs` 等價於 `fmap f xs`。而 `pure f` 就是 `[f]`，而且 `[f] <*> xs` 可將左邊的每個函數套用至右邊的每個值。但左邊其實只有一個函數，所以他做起來就像是 mapping。

另一個我們已經看過的 `Applicative` 的 instance 是 `IO`，來看看他是怎麼實作的：

```
instance Applicative IO where
    pure = return
    a <*> b = do
        f <- a
        x <- b
        return (f x)
```



由於 `pure` 是把一個值放進最小的 context 中，所以將 `return` 定義成 `pure` 是很合理的。因為 `return` 也是做同樣的事情。他做了一個不做任何事情的 I/O action，他可以產生某些值來作為結果，但他實際上並沒有做任何 I/O 的動作，例如說印出結果到終端或是檔案。

如果 `<*>` 被限定在 `IO` 上操作的話，他的型別會是 `(<*>) :: IO (a -> b) -> IO a -> IO b`。他接受一個產生函數的 I/O action，還有另一個 I/O action，並從以上兩者創造一個新的 I/O action，也就是把第二個參數餵給第一個參數。而得到回傳的結果，然後放到新的 I/O action 中。我們用 `do` 的語法來實作他。你還記得的話 `do` 就是把好幾個 I/O action 黏在一起，變成一個大的 I/O action。

而對於 `Maybe` 跟 `[]` 而言，我們可以把 `<*>` 想做是從左邊的參數取出一個函數，然後套用到右邊的參數上。至於 `IO`，這種取出的類比方式仍然適用，但我們必須多加一個 `sequencing` 的概念，因為我們是從兩個 I/O action 中取值，也是在 `sequencing`，把他們黏成一個。我們從第一個 I/O action 中取值，但要取出 I/O action 的結果，他必須要先被執行過。

考慮下面這個範例：

```
myAction :: IO String
myAction = do
    a <- getLine
    b <- getLine
    return $ a ++ b
```

這是一個提示使用者輸入兩行並產生將兩行輸入串接在一起結果的一個 I/O action。我們先把兩個 `getLine` 黏在一起，然後用一個 `return`，這是因為我們想要這個黏成的 I/O action 包含 `a ++ b` 的結果。我們也可以用 applicative style 的方式來描述：

```
myAction :: IO String
myAction = (++) <$> getLine <*> getLine
```

我們先前的作法是將兩個 I/O action 的結果餵給函數。還記得 `getLine` 的型別是 `getLine :: IO String`。當我們對 applicative functor 使用 `<*>` 的時候，結果也會是 applicative functor。

如果我們再使用盒子的類比，我們可以把 `getLine` 想做是一個去真實世界中拿取字串的盒子。而 `(++) <$> getLine <*> getLine` 會創造一個比較大的盒子，這個大盒子會派兩個盒子去終端拿取字串，並把結果串接起來放進自己的盒子中。

`(++) <$> getLine <*> getLine` 的型別是 `IO String`，他代表這個表達式是一個再普通不過的 I/O action，他裡面也裝著某種值。這也是為什麼我們可以這樣寫：

```
main = do
    a <- (++) <$> getLine <*> getLine
    putStrLn $ "The two lines concatenated turn out to be: " ++ a
```

如果你發現你是在做 binding I/O action 的動作，而且在 binding 之後還呼叫一些函數，最後用 `return` 來將結果包起來。那你可以考慮使用 applicative style，這樣可以更簡潔。

另一個 `Applicative` 的 instance 是 `((->) r)`。雖然他們通常是用在 code golf 的情況，但他們還是十分有趣的例子。所以我們還是來看一下他們是怎麼被實作的。

如果你忘記 `((->) r)` 的意思，回去翻翻前一章節我們介紹 `((->) r)` 作為一個 functor 的範例。

```
instance Applicative ((->) r) where
    pure x = (\_ -> x)
    f <*> g = \x -> f x (g x)
```

當我們用 `pure` 將一個值包成 applicative functor 的時候，他產生的結果永遠都會是那個值。也就是最小的 context。那也是為什麼對於 function 的 `pure` 實作來講，他就是接受一個值，然後造一個函數永遠回傳那個值，不管他被餵了什麼參數。如果你限定 `pure` 的型別至 `((->) r)` 上，他就會是 `pure :: a -> ((->) r)`。

```
ghci> (pure 3) "blah"
3
```

由於 currying 的關係，函數套用是 left-associative，所以我們忽略掉括弧。

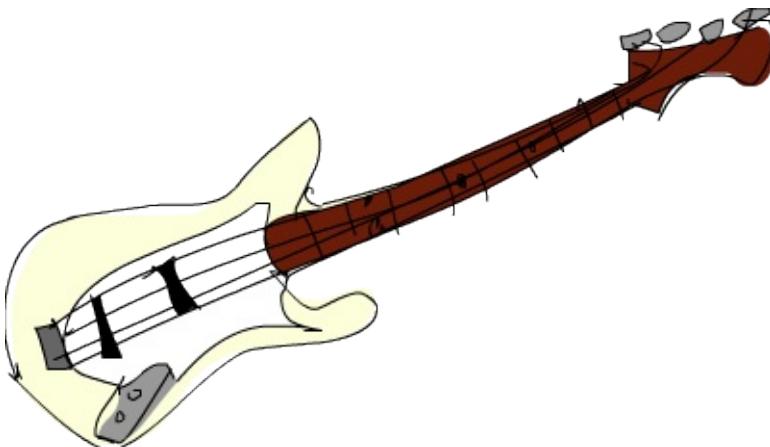
```
ghci> pure 3 "blah"
3
```

而 `<*>` 的實作是比較不容易瞭解的，我們最好看一下怎麼用 applicative style 的方式來使用作為 applicative functor 的 function。

```
ghci> :t (+) <$> (+3) <*> (*100)
(+) <$> (+3) <*> (*100) :: (Num a) => a -> a
ghci> (+) <$> (+3) <*> (*100) $ 5
508
```

將兩個 applicative functor 餵給 `<*>` 可以產生一個新的 applicative functor，所以如果我們丟給他兩個函數，我們能得到一個新的函數。所以是怎麼一回事呢？當我們做 `(+) <$> (+3) <*> (*100)`，我們是在實作一個函數，他會將 `(+3)` 跟 `(*100)` 的結果再套用 `+`。要看一個實際的範例的話，可以看一下 `(+) <$> (+3) <*> (*100) $ 5` 首先 `5` 被丟給 `(+3)` 跟 `(*100)`，產生 `8` 跟 `500`。然後 `+` 被套用到 `8` 跟 `500`，得到 `508`。

```
ghci> (\x y z -> [x,y,z]) <$> (+3) <*> (*2) <*> (/2) $ 5
[8.0,10.0,2.5]
```



這邊也一樣。我們創建了一個函數，他會呼叫 `\x y z -> [x,y,z]`，而丟的參數是 `(+3)`，`(*2)` 跟 `(/2)`。`5` 被丟給以上三個函數，然後他們結果又接到 `\x y z -> [x, y, z]`。

你可以將函數想做是裝著最終結果的盒子，所以 `k <$> f <*> g` 會製造一個函數，他會將 `f` 跟 `g` 的結果丟給 `k`。當我們做 `(+) <$> Just 3 <*> Just 5`，我們是用 `+` 套用在一些可能有或可能沒有的值上，所以結果也會是可能有或沒有。當我們做 `(+) <$> (+10) <*> (+5)`，我們是將 `+` 套用在 `(+10)` 跟 `(+5)` 的結果上，而結果也會是一個函數，當被餵給一個參數的時候會產生結果。

我們通常不會將函數當作 applicative 用，不過仍然值得當作練習。對於 `(->) r` 怎麼定義成 `Applicative` 的並不是真的那麼重要，所以如果你不是很懂的話也沒關係。這只是讓你獲得一些操作上的直覺罷了。

一個我們之前還沒碰過的 `Applicative` 的 instance 是 `ZipList`，他是包含在 `Control.Applicative` 中。

對於 list 要作為一個 applicative functor 可以有多種方式。我們已經介紹過其中一種。如果套用 `<*>`，左邊是許多函數，而右邊是許多值，那結果會是函數套用到值的所有組合。如果我們做 `[(+3), (*2)] <*> [1, 2]`。那 `(+3)` 會先套用至 `1` 跟 `2`。接著 `(*2)` 套用至 `1` 跟 `2`。而得到 `[4, 5, 2, 4]`。

然而 `[(+3), (*2)] <*> [1, 2]` 也可以這樣運作：把左邊第一個函數套用至右邊第一個值，接著左邊第二個函數套用右邊第二個值，以此類推。這樣得到的會是 `[4, 4]`。或是 `[1 + 3, 2 * 2]`。

由於一個型別不能對同一個 typeclass 定義兩個 instance，所以才會定義了 `ZipList a`，他只有一個構造子 `ZipList`，他只包含一個欄位，他的型別是 `list`。

```
instance Applicative ZipList where
    pure x = ZipList (repeat x)
    ZipList fs <*> ZipList xs = ZipList (zipWith (\f x -> f x) fs xs)
```

`<*>` 做的就是我們之前說的。他將第一個函數套用至第一個值，第二個函數套用第二個值。這也是 `zipWith (\f x -> f x) fs xs` 做的事。由於 `zipWith` 的特性，所以結果會跟 `list` 中比較短的那個一樣長。

`pure` 也值得我們討論一下。他接受一個值，把他重複地放進一個 `list` 中。`pure "haha"` 就會是 `ZipList ("haha", "haha", "haha"...)`。這可能會造成些混淆，畢竟我們說過 `pure` 是把一個值放進一個最小的 context 中。而你會想說無限長的 `list` 不可能會是一個最小的 context。但對於 `zip list` 來說這是很合理的，因為他必須在 `list` 的每個位置都有值。這也遵守了 `pure f <*> xs` 必須要等價於 `fmap f xs` 的特性。如果 `pure 3` 只是回傳 `ZipList [3]`，那 `pure (*2) <*> ZipList [1, 5, 10]` 就只會算出 `ZipList [2]`，因為兩個 `zip list` 算出結果的長度會是比較短的那個的長度。如果我們 `zip` 一個有限長的 `list` 以及一個無限長的 `list`，那結果的長會是有限長的 `list` 的長度。

那 `zip list` 是怎麼用 applicative style 操作的呢？我們來看看，`ZipList a` 型別並沒有定義成 `Show` 的 instance，所以我們必須用 `getZipList` 函數來從 `zip list` 取出一個普通的 `list`。

```
ghci> getZipList $ (+) <$> ZipList [1,2,3] <*> ZipList [100,100,100]
[101,102,103]
ghci> getZipList $ (+) <$> ZipList [1,2,3] <*> ZipList [100,100..]
[101,102,103]
ghci> getZipList $ max <$> ZipList [1,2,3,4,5,3] <*> ZipList [5,3,1,2]
[5,3,3,4]
ghci> getZipList $ (,,) <$> ZipList "dog" <*> ZipList "cat" <*> ZipList "rat"
[('d','c','r'),('o','a','a'),('g','t','t')]
```

`((,,))` 函數跟 `\x y z -> (x,y,z)` 是等價的，而 `((,))` 跟 `\x y -> (x,y)` 是等價的。

除了 `zipWith`，標準函式庫中也有 `zipWith3`, `zipWith4` 之類的函數，最多支援到 7。 `zipWith` 接受一個接受兩個參數的函數，並把兩個 list zip 起來。`zipWith3` 則接受一個接受三個參數的函數，然後把三個 list zip 起來。以此類推。用 applicative style 的方式來操作 zip list 的話，我們就不需要對每個數量的 list 都定義一個獨立的 zip 函數來 zip 他們。我們只需要用 applicative style 的方式來把任意數量的 list zip 起來就可以了。

`Control.Applicative` 定義了一個函數叫做 `liftA2`，他的型別是 `liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c`。他定義如下：

```
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b
```

並沒有太難理解的東西，他不過就是對兩個 applicatives 套用函數而已，而不用我們剛剛熟悉的 applicative style。我們提及他的理由只是要展示為什麼 applicative functors 比起一般的普通 functor 要強。如果只是普通的 functor 的話，我們只能將一個函數 map over 這個 functor。但有了 applicative functor，我們可以對好多個 functor 套用一個函數。看看這個函數的型別，他會是 `(a -> b -> c) -> (f a -> f b -> f c)`。當我們從這樣的角度來看他的話，我們可以說 `liftA2` 接受一個普通的二元函數，並將他升級成一個函數可以運作在兩個 functor 之上。

另外一個有趣的概念是，我們可以接受兩個 applicative functor 並把他們結合成一個 applicative functor，這個新的將這兩個 applicative functor 裝在 list 中。舉例來說，我們現在有 `Just 3` 跟 `Just 4`。我們假設後者是一個只包含單元素的 list。

```
ghci> fmap (\x -> [x]) (Just 4)
Just [4]
```

所以假設我們有 `Just 3` 跟 `Just [4]`。我們有怎麼得到 `Just [3,4]` 呢？很簡單。

```
ghci> liftA2 (:) (Just 3) (Just [4])
Just [3,4]
ghci> (:) <$> Just 3 <*> Just [4]
Just [3,4]
```

還記得 `:` 是一個函數，他接受一個元素跟一個 list，並回傳一個新的 list，其中那個元素已經接在前面。現在我們有了 `Just [3,4]`，我們能夠將他跟 `Just 2` 繩在一起變成 `Just [2,3,4]` 嗎？當然可以。我們可以將任意數量的 applicative 繩在一起變成一個 applicative，裡面包含一個裝有結果的 list。我們試著實作一個函數，他接受一串裝有 applicative 的 list，然後回傳一個 applicative 裡面有一個裝有結果的 list。我們稱呼他為 `sequenceA`。

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA [] = pure []
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs
```

居然用到了遞迴！首先我們來看一下他的型別。他將一串 applicative 的 list 轉換成一個 applicative 裝有一個 list。從這個資訊我們可以推測出邊界條件。如果我們要將一個空的 list 變成一個裝有 list 的 applicative。我們只要把這個空的 list 放進一個預設的 context。現在來看一下我們怎麼用遞迴的。如果們有一個可以分成頭跟尾的 list (`x` 是一個 applicative 而 `xs` 是一串 applicative)，我們可以對尾巴呼叫 `sequenceA`，便會得到一個裝有 list 的 applicative。然後我們只要將在 `x` 中的值把他接到裝有 list 的 applicative 前面就可以了。

所以如果我們做 `sequenceA [Just 1, Just 2]`，也就是 `(:) <$> Just 1 <*> sequenceA [Just 2]`。那會等價於 `(:) <$> Just 1 <*> ((:) <$> Just 2 <*> sequenceA [])`。我們知道 `sequenceA []` 算出來會是 `Just []`，所以運算式就變成 `(:) <$> Just 1 <*> ((:) <$> Just 2 <*> Just [])`，也就是 `(:) <$> Just 1 <*> Just [2]`，算出來就是 `Just [1,2]`。

另一種實作 `sequenceA` 的方式是用 fold。要記得幾乎任何需要走遍整個 list 並 accumulate 成一個結果的都可以用 fold 來實作。

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA = foldr (liftA2 (:)) (pure [])
```

我們從右往左走，並且起始的 accumulator 是用 `pure []`。我們是用 `liftA2 (:)` 來結合 accumulator 跟 list 中最後的元素，而得到一個 applicative，裡面裝有一個單一元素的一個 list。然後我們再用 `liftA2 (:)` 來結合 accumulator 跟最後一個元素，直到我們只剩下 accumulator 為止，而得到一個 applicative，裡面裝有所有結果。

我們來試試看套用在不同 applicative 上。

```
ghci> sequenceA [Just 3, Just 2, Just 1]
Just [3,2,1]
ghci> sequenceA [Just 3, Nothing, Just 1]
Nothing
ghci> sequenceA [(+3),(+2),(+1)] 3
[6,5,4]
ghci> sequenceA [[1,2,3],[4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
ghci> sequenceA [[1,2,3],[4,5,6],[3,4,4],[]]
[]
```

很酷吧。當我們套用在 `Maybe` 上時，`sequenceA` 創造一個新的 `Maybe`，他包含了一個 list 裝有所有結果。如果其中一個值是 `Nothing`，那整個結果就會是 `Nothing`。如果你有一串 `Maybe` 型別的值，但你只在乎當結果不包含任何 `Nothing` 的情況，這樣的特性就很方便。

當套用在函數時，`sequenceA` 接受裝有一堆函數的 list，並回傳一個回傳 list 的函數。在我們的範例中，我們寫了一個函數，他只接受一個數值作為參數，他會把他套用至 list 中的每一個函數，並回傳一個包含結果的 list。`sequenceA [(+3),(+2),(+1)] 3` 會將 `3` 餵給 `(+3)`，`(+2)` 跟 `(+1)`，然後將所有結果裝在一個 list 中。

而 `(+) <$> (+3) <*> (*2)` 會創見一個接受單一參數的一函數，將他同時餵給 `(+3)` 跟 `(*2)`，然後呼叫 `+` 來將兩者加起來。同樣的道理，`sequenceA [(+3),(*2)]` 是製造一個接受單一參數的函數，他會將他餵給所有包含在 list 中的函數。但他最後不是呼叫 `+`，而是呼叫 `:` 跟 `pure []` 來把結果接成一個 list，得到最後的結果。

當我們有一串函數，我們想要將相同的輸入都餵給他們並檢視結果的時候，`sequenceA` 非常好用。例如說，我們手上有一個數值，但不知道他是否滿足一串 predicate。一種實作的方式是像這樣：

```
ghci> map (\f -> f 7) [(>4),(<10),odd]
[True,True,True]
ghci> and $ map (\f -> f 7) [(>4),(<10),odd]
True
```

記住 `and` 接受一串布林值，並只有在全部都是 `True` 的時候才回傳 `True`。另一種實作方式是用 `sequenceA`：

```
ghci> sequenceA [(>4),(<10),odd] 7
[True,True,True]
ghci> and $ sequenceA [(>4),(<10),odd] 7
True
```

`sequenceA [(>4),(<10),odd]` 接受一個函數，他接受一個數值並將他餵給所有的 predicate，包含 `[(>4),(<10),odd]`。然後回傳一串布林值。他將一個型別為 `(Num a) => [a -> Bool]` 的 list 變成一個型別為 `(Num a) => a -> [Bool]` 的函數，很酷吧。

由於 list 要求裡面元素的型別要一致，所以包含在 list 中的所有函數都是同樣型別。你不能創造一個像是 `[ord, (+3)]` 這樣的 list，因為 `ord` 接受一個字元並回傳一個數值，然而 `(+3)` 接受一個數值並回傳一個數值。

當跟 `[]` 一起使用的時候，`sequenceA` 接受一串 list，並回傳另一串 list。他實際上是創建一個包含所有可能組合的 list。為了方便說明，我們比較一下使用 `sequenceA` 跟 list comprehension 的差異：

```
ghci> sequenceA [[1,2,3],[4,5,6]]
[[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
 ghci> [[x,y] | x <- [1,2,3], y <- [4,5,6]]
 [[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
 ghci> sequenceA [[1,2],[3,4]]
 [[[1,3],[1,4],[2,3],[2,4]]
 ghci> [[x,y] | x <- [1,2], y <- [3,4]]
 [[[1,3],[1,4],[2,3],[2,4]]
 ghci> sequenceA [[1,2],[3,4],[5,6]]
 [[[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],[2,4,5],[2,4,6]]
 ghci> [[x,y,z] | x <- [1,2], y <- [3,4], z <- [5,6]]
 [[[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],[2,4,5],[2,4,6]]]
```

這可能有點難以理解，但如果你多做點嘗試，你會比較能看出來些眉目。假設我們在做 `sequenceA [[1,2],[3,4]]`。要知道這是怎麼回事，我們首先用 `sequenceA` 的定義 `sequenceA (x:xs) = (:) <$> x <*> sequenceA xs` 還有邊界條件 `sequenceA [] = pure []` 來看看。你不需要實際計算，但他可以幫助你理解 `sequenceA` 是怎麼運作在一串 list 上，畢竟這有點複雜。

```
# 我們從 ``sequenceA [[1,2],[3,4]]`` 開始
# 那可以被計算成 ``(:) <$> [1,2] <*> sequenceA [[3,4]]``
# 計算內層的 ``sequenceA``，會得到 ``(:) <$> [1,2] <*> ((:) <$> [3,4] <*> sequenceA [])``
# 我們碰到了邊界條件，所以會是 ``(:) <$> [1,2] <*> ((:) <$> [3,4] <*> [[]])``
# 現在我們計算 ``((:) <$> [3,4] <*> [[]])`` 的部份，我們會對左邊 list 中的每一個值（也就是 ``3`` 跟
# 而對於左邊的每一個值(``1`` 跟 ``2``)以及右邊可能的值(``[3]`` 跟 ``[4]``)我們套用 ``:`` 而得到
```

計算 `(+) <$> [1,2] <*> [4,5,6]` 會得到一個 non-deterministic 的結果 `x + y`，其中 `x` 代表 `[1,2]` 中的每一個值，而 `y` 代表 `[4,5,6]` 中的每一個值。我們用 list 來表示每一種可能的情形。同樣的，當我們在做 `sequence [[1,2],[3,4],[5,6],[7,8]]`，他的結果會是 non-deterministic 的 `[x,y,z,w]`，其中 `x` 代表 `[1,2]` 中的每一個值，而 `y` 代表 `[3,4]` 中的每一個值。以此類推。我們用 list 代表 non-deterministic 的計算，每一個元素都是一個可能的情形。這也是為什麼會用到 list of list。

當使用在 I/O action 上的時候，`sequenceA` 跟 `sequence` 是等價的。他接受一串 I/O action 並回傳一個 I/O action，這個 I/O action 會計算 list 中的每一個 I/O action，並把結果放在一個 list 中。要將型別為 `[IO a]` 的值轉換成 `IO [a]` 的值，也就是會產生一串 list 的一個 I/O action，那這些 I/O action 必須要一個一個地被計算，畢竟對於這些 I/O action 你沒辦法不計算就得到結果。

```
ghci> sequenceA [getLine, getLine, getLine]
heyh
ho
woo
["heyh", "ho", "woo"]
```

就像普通的函數一樣，applicative functors 也遵循一些定律。其中最重要的一個是我們之前提過的 `pure f <*> x = fmap f x`。你可以證明一些我們之前介紹過的 applicative functor 遵守這個定律當作練習。其他的 functors law 有：

```
# ``pure id <*> v = v``
# ``pure (.) <*> u <*> v <*> w = u <*> (v <*> w)``
# ``pure f <*> pure x = pure (f x)``
# ``u <*> pure y = pure ($ y) <*> u``
```

我們不會一項一項地細看，那樣會花費很大的篇幅而且對讀者來說很無聊，但如果你有興趣，你可以針對某些 instance 看看他們會不會遵守。

結論就是 applicative functor 不只是有趣而且實用，他允許我們結合不同種類的計算，像是 I/O 計算，non-deterministic 的計算，有可能失敗的計算等等。而使用 `<$>` 跟 `<*>` 我們可以將普通的函數來運作在任意數量的 applicative functors 上。

## 關鍵字"newtype"



到目前為止，我們已經看過了如何用 `data` 關鍵字定義自己的 algebraic data type。我們也學習到了如何用 `type` 來定義 type synonyms。在這個章節中，我們會看一下如何使用 `newtype` 來從一個現有的型別中定義出新的型別，並說明我們為什麼會想要那麼做。

在之前的章節中，我們瞭解到其實 list 有很多種方式可以被視為一種 applicative functor。一中方式是定義 `<*>` 將左邊的每一個值跟右邊的每一個值組合，而得到各種組合的結果。

```
ghci> [(+1),(*100),(*5)] <*> [1,2,3]
[2,3,4,100,200,300,5,10,15]
```

第二種方式是將 `<*>` 定義成將左邊的第一個函數套用至右邊的第一個值，然後將左邊第二個函數套用至右邊第二個值。以此類推。最終，這表現得有點像將兩個 list 用一個拉鍊拉起來一樣。但由於 list 已經被定義成 `Applicative` 的 instance 了，所以我們要怎麼要讓 list 可以被定義成第二種方式呢？如果你還記得我們說過我們是有很好的理由定義了 `ZipList a`，其中他裡面只包含一個值構造子跟只包含一個欄位。其實他的理由就是要讓 `ZipList` 定義成用拉鍊的方式來表現 applicative 行為。我們只不過用 `ZipList` 這個構造子將他包起來，然後用 `getZipList` 來解開來。

```
ghci> getZipList $ ZipList [(+1),(*100),(*5)] <*> ZipList [1,2,3]
[2,200,15]
```

所以這跟 `newtype` 這個關鍵字有什麼關係呢？想想看我們是怎麼宣告我們的 `ZipList a` 的，一種方式是像這樣：

```
data ZipList a = ZipList [a]
```

也就是一個只有一個值構造子的型別而且那個構造子裡面只有一個欄位。我們也可以用 record syntax 來定義一個解開的函數：

```
data ZipList a = ZipList { getZipList :: [a] }
```

這樣聽起來不錯。這樣我們就有兩種方式來讓一個型別來表現一個 typeclass，我們可以用 `data` 關鍵字來把一個型別包在另一個裡面，然後再將他定義成第二種表現方式。

而在 Haskell 中 `newtype` 正是為了這種情形，我們想將一個型別包在另一個型別中。在實際的函式庫中 `ZipList a` 是這樣定義了：

```
newtype ZipList a = ZipList { getZipList :: [a] }
```

這邊我們不用 `data` 關鍵字反而是用 `newtype` 關鍵字。這是為什麼呢？第一個理由是 `newtype` 比較快速。如果你用 `data` 關鍵字來包一個型別的話，在你執行的時候會有一些包起來跟解開來的成本。但如果你用 `newtype` 的話，Haskell 會知道你只是要將一個現有的型別包成一個新的型別，你想要內部運作完全一樣但只是要一個全新的型別而已。有了這個概念，Haskell 可以將包裹跟解開來的成本都去除掉。

那為什麼我們不是一直使用 `newtype` 呢？當你用 `newtype` 來製作一個新的型別時，你只能定義單一個值構造子，而且那個構造子只能有一個欄位。但使用 `data` 的話，你可以讓那個型別有好幾個值構造子，並且每個構造子可以有零個或多個欄位。

```

data Profession = Fighter | Archer | Accountant

data Race = Human | Elf | Orc | Goblin

data PlayerCharacter = PlayerCharacter Race Profession

```

當使用 `newtype` 的時候，你是被限制只能用一個值構造子跟單一欄位。

對於 `newtype` 我們也能使用 `deriving` 關鍵字。我們可以 `derive` 像是 `Eq` , `Ord` , `Enum` , `Bounded` , `Show` 跟 `Read` 的 `instance`。如果我們想要對新的型別做 `derive`，那原本的型別必須已經在那個 `typeclass` 中。這樣很合理，畢竟 `newtype` 就是要將現有的型別包起來。如果我們按照下面的方式定義的話，我們就能對我們的型別做印出以及比較相等性的操作：

```

newtype CharList = CharList { getCharList :: [Char] } deriving (Eq, Show)

```

我們來跑跑看：

```

ghci> CharList "this will be shown!"
CharList {getCharList = "this will be shown!"}
ghci> CharList "benny" == CharList "benny"
True
ghci> CharList "benny" == CharList "oisters"
False

```

對於這個 `newtype`，他的值構造子有下列型別：

```

CharList :: [Char] -> CharList

```

他接受一個 `[Char]` 的值，例如 `"my sharona"` 並回傳一個 `CharList` 的值。從上面我們使用 `CharList` 的值構造子的範例中，我們可以看到的確是這樣。相反地，`getCharList` 具有下列的型別。

```

getCharList :: CharList -> [Char]

```

他接受一個 `CharList` 的值並將他轉成 `[Char]`。你可以將這個想成包裝跟解開的動作，但你也可以將他想成從一個型別轉成另一個型別。

## Using newtype to make type class instances

有好幾次我們想要讓我們的型別屬於某個 `typeclass`，但型別變數並沒有符合我們想要的。要把 `Maybe` 定義成 `Functor` 的 `instance` 很容易，因為 `Functor` 這個 `typeclass` 被定義如下：

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

我們先定義如下：

```
instance Functor Maybe where
```

然後我們實作 `fmap`。當所有的型別變數被填上時，由於 `Maybe` 取代了 `Functor` 中 `f` 的位置，所以如果我們看看 `fmap` 運作在 `Maybe` 上時是什麼樣，他會像這樣：

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```



看起來不錯吧？現在我們想要 `tuple` 成為 `Functor` 的一個 `instance`，所以當我們用 `fmap` 來 map over 一個 `tuple` 時，他會先套用到 `tuple` 中的第一個元素。這樣當我們做 `fmap (+3)` `(1,1)` 會得到 `(4,1)`。不過要定義出這樣的 `instance` 有些困難。對於 `Maybe`，我們只要寫 `instance Functor Maybe where`，這是因為對於只吃一個參數的型別構造子我們很容易定義成 `Functor` 的 `instance`。但對於 `(a,b)` 這樣的就沒辦法。要繞過這樣的困境，我們可以用 `newtype` 來重新定義我們的 `tuple`，這樣第二個型別參數就代表了 `tuple` 中的第一個元素部份。

```
newtype Pair b a = Pair { getPair :: (a,b) }
```

現在我們可以將他定義成 `Functor` 的 `instance`，所以函數被 map over `tuple` 中的第一個部份。

```
instance Functor (Pair c) where
    fmap f (Pair (x,y)) = Pair (f x, y)
```

正如你看到的，我們可以對 newtype 定義的型別做模式匹配。我們用模式匹配來拿到底層的 tuple，然後我們將 `f` 來套用至 tuple 的第一個部份，然後我們用 `Pair` 這個值構造子來將 tuple 轉換成 `Pair b a`。如果我們問 `fmap` 的型別究竟是什麼，他會是：

```
fmap :: (a -> b) -> Pair c a -> Pair c b
```

我們說過 `instance Functor (Pair c) where` 跟 `Pair c` 取代了 `Functor` 中 `f` 的位置：

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

如果我們將一個 tuple 轉換成 `Pair b a`，我們可以用 `fmap` 來 map over 第一個部份。

```
ghci> getPair $ fmap (*100) (Pair (2,3))
(200,3)
ghci> getPair $ fmap reverse (Pair ("london calling", 3))
("gnillac nodnol",3)
```

## On newtype laziness

我們提到 `newtype` 一般來講比 `data` 來得有效率。`newtype` 能做的唯一一件事就是將現有的型別包成新的型別。這樣 Haskell 在內部就能將新的型別的值用舊的方式來操作。只是要記住他們還是不同的型別。這代表 `newtype` 並不只是有效率，他也具備 `lazy` 的特性。我們來說明一下這是什麼意思。

就像我們之前說得，Haskell 預設是具備 `lazy` 的特性，這代表只有當我們要將函數的結果印出來的時候計算才會發生。或者說，只有當我們真的需要結果的時候計算才會發生。在 Haskell 中 `undefined` 代表會造成錯誤的計算。如果我們試著計算他，也就是將他印到終端中，Haskell 會丟出錯誤。

```
ghci> undefined
*** Exception: Prelude.undefined
```

然而，如果我們做一個 list，其中包含一些 `undefined` 的值，但卻要求一個不是 `undefined` 的 `head`，那一切都會順利地被計算，因為 Haskell 並不需要 list 中其他元素來得到結果。我們僅僅需要看到第一個元素而已。

```
ghci> head [3,4,5,undefined,2,undefined]
3
```

現在們考慮下面的型別：

```
data CoolBool = CoolBool { getCoolBool :: Bool }
```

這是一個用 `data` 關鍵字定義的 algebraic data type。他有一個值建構子並只有一個型別為 `Bool` 的欄位。我們寫一個函數來對 `CoolBool` 做模式匹配，並回傳一個 `"hello"` 的值。他並不會管 `CoolBool` 中裝的究竟是 `True` 或 `False`。

```
helloMe :: CoolBool -> String
helloMe (CoolBool _) = "hello"
```

這次我們不餵給這個函數一個普通的 `CoolBool`，而是丟給他一個 `undefined`。

```
ghci> helloMe undefined
*** Exception: Prelude.undefined "
```

結果收到了一個 `Exception`。是什麼造成這個 `Exception` 的呢？用 `data` 定義的型別可以有好幾個值構造子（儘管 `CoolBool` 只有一個）所以當我們要看看餵給函數的值是否是 `(CoolBool _)` 的形式，Haskell 會需要做一些基本的計算來看看是哪個值構造子被用到。但當我們計算 `undefined` 的時候，就算是一點也會丟出 `Exception`。

我們不用 `data` 來定義 `CoolBool` 而用 `newtype`：

```
newtype CoolBool = CoolBool { getCoolBool :: Bool }
```

我們不用修改 `helloMe` 函數，因為對於模式匹配使用 `newtype` 或 `data` 都是一樣。我們再來將 `undefined` 餵給 `helloMe`。

```
ghci> helloMe undefined
"hello"
```

居然正常運作！為什麼呢？正如我們說過得，當我們使用 `newtype` 的時候，Haskell 內部可以將新的型別用舊的型別來表示。他不必加入另一層 box 來包住舊有的型別。他只要注意他是不同的型別就好了。而且 Haskell 會知道 `newtype` 定義的型別一定只會有一個構造子，他不必計算餵給函數的值就能確定他是 `(CoolBool _)` 的形式，因為 `newtype` 只有一個可能的值跟單一欄位！

這樣行為的差異可能沒什麼關係，但實際上他非常重要。因為他讓我們認知到儘管從撰寫程式的觀點來看沒什麼差異，但他們的確是兩種不同的機制。儘管 `data` 可以讓你從無到有定義型別，`newtype` 是從一個現有的型別做出來的。對 `newtype` 做模式匹配並不是像從盒子中取出東西，他比較像是將一個型別轉換成另一個型別。

## type vs newtype vs data

到目前為止，你也許對於 `type` , `data` 跟 `newtype` 之間的差異還不是很瞭解，讓我們快速複習一遍。

`type` 關鍵字是讓我們定義 type synonyms。他代表我們只是要給一個現有的型別另一個名字，假設我們這樣做：

```
type IntList = [Int]
```

這樣做可以允許我們用 `IntList` 的名稱來指稱 `[Int]`。我們可以交換地使用他們。但我們並不會因此有一個 `IntList` 的值構造子。因為 `[Int]` 跟 `IntList` 只是兩種指稱同一個型別的方式。我們在指稱的時候用哪一個並無所謂。

```
ghci> ([1,2,3] :: IntList) ++ ([1,2,3] :: [Int])
[1,2,3,1,2,3]
```

當我們想要讓 type signature 更清楚一些，給予我們更瞭解函數的 context 的時候，我們會定義 type synonyms。舉例來說，當我們用一個型別為 `[(String, String)]` 的 association list 來代表一個電話簿的時候，我們可以定義一個 `PhoneBook` 的 type synonym，這樣 type signature 會比較容易讀。

`newtype` 關鍵字將現有的型別包成一個新的型別，大部分是為了要讓他們可以是特定 typeclass 的 instance 而這樣做。當我們使用 `newtype` 來包裹一個現有的型別時，這個型別跟原有的型別是分開的。如果我們將下面的型別用 `newtype` 定義：

```
newtype CharList = CharList { getCharList :: [Char] }
```

我們不能用 `++` 來將 `CharList` 跟 `[Char]` 接在一起。我們也不能用 `++` 來將兩個 `CharList` 接在一起，因為 `++` 只能套用在 list 上，而 `CharList` 並不是 list，儘管你會說他包含一個 list。但我們可以將兩個 `CharList` 轉成 list，將他們 `++` 然後再轉回 `CharList`。

當我們在 `newtype` 告中使用 record syntax 的時候，我們會得到將新的型別轉成舊的型別的函數，也就是我們 `newtype` 的值構造子，以及一個函數將他的欄位取出。新的型別並不會被自動定義成原有型別所屬的 typeclass 的一個 instance，所以我們必須自己來 derive 他們。

實際上你可以將 `newtype` 想成是只能定義一個構造子跟一個欄位的 `data` 告。如果你碰到這種情形，可以考慮使用 `newtype`。

使用 `data` 關鍵字是為了定義自己的型別。他們可以在 algebraic data type 中放任意數量的構造子跟欄位。可以定義的東西從 list, Maybe 到 tree。

如果你只是希望你的 type signature 看起來比較乾淨，你可以只需要 type synonym。如果你想要將現有的型別包起來並定義成一個 type class 的 instance，你可以嘗試使用 newtype。如果你想要定義完全新的型別，那你應該使用 `data` 關鍵字。

## Monoids

Haskell 中 typeclass 是用來表示一個型別之間共有的行為，是一種 interface。我們介紹過 `Eq`，他定義型別是否可以比較相等性，以及 `Ord`，他表示可以被排序的型別。還介紹了更有趣的像是 `Functor` 跟 `Applicative`。

當我們定義一個型別時，我們會想說他應該要支援的行為。也就是表現的行為是什麼，並且要讓他屬於哪些 typeclass。如果希望他可以比較相等與否，那我們就應該定義他成為 `Eq` 的一個 instance。如果我們想要看看型別是否是一種 functor，我們可以定義他是 `Functor` 的一個 instance。以此類推。

考慮 `*` 是一個將兩個數值相乘的一個函數。如果我們將一個數值乘上 `1`，那就會得到自身的數值。我們實際上是做 `1 * x` 或 `x * 1` 並沒有差別。結果永遠會是 `x`。同樣的，`++` 是一個接受兩個參數並回傳新的值的一個函數。只是他不是相乘而是將兩個 list 接在一起。而類似 `*`，他也有一個特定的值，當他跟其他值使用 `++` 時會得到同樣的值。那個值就是空的 list `[]`。

```
ghci> 4 * 1
4
ghci> 1 * 9
9
ghci> [1,2,3] ++ []
[1,2,3]
ghci> [] ++ [0.5, 2.5]
[0.5,2.5]
```

看起來 `*` 之於 `1` 跟 `++` 之於 `[]` 有類似的性質：

```
# 函數同樣接受兩個參數
# 參數跟回傳值是同樣的型別
# 同樣存在某些值當套用二元函數時並不會改變其他值
```

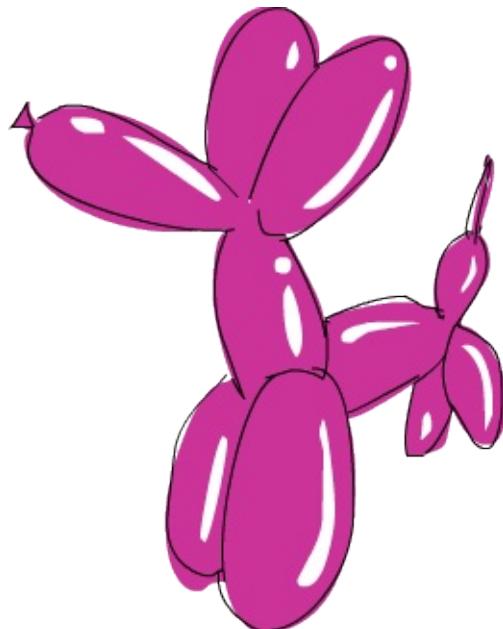
關於這兩種操作還有另一個比較難察覺的性質就是，當我們對這個二元函數對三個以上的值操作並化簡，函數套用的順序並不會影響到結果。不論是 `(3 * 4) * 5` 或是 `3 * (4 * 5)`，兩種方式都會得到 `60`。而 `++` 也是相同的。

```
ghci> (3 * 2) * (8 * 5)
240
ghci> 3 * (2 * (8 * 5))
240
ghci> "la" ++ ("di" ++ "da")
"ladida"
ghci> ("la" ++ "di") ++ "da"
"ladida"
```

我們稱呼這樣的性質為結合律(associativity)。`*` 遵守結合律，`++` 也是。但 `-` 就不遵守。`(5 - 3) - 4` 跟 `5 - (3 - 4)` 得到的結果是不同的。

注意到這些性質並具體地寫下來，就可以得到 monoid。一個 monoid 是你有一個遵守結合律的二元函數還有一個可以相對於那個函數作為 identity 的值。當某個值相對於一個函數是一個 identity，他表示當我們將這個值丟給函數時，結果永遠會是另外一邊的那個值本身。`1` 是相對於 `*` 的 identity，而 `[]` 是相對於 `++` 的 identity。在 Haskell 中還有許多其他的 monoid，這也是為什麼我們定義了 `Monoid` 這個 typeclass。他描述了表現成 monoid 的那些型別。我們來看看這個 typeclass 是怎麼被定義的：

```
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```



`Monoid` typeclass 被定義在 `import Data.Monoid` 中。我們來花些時間好好瞭解他。

首先我們看到只有具體型別才能定義成 `Monoid` 的 instance。由於在 typeclass 定義中的 `m` 並不接受任何型別參數。這跟 `Functor` 以及 `Applicative` 不同，他們要求他們的 instance 必須是一個接受單一型別參數的型別構造子。

第一個函數是 `mempty`，由於他不接受任何參數，所以他並不是一個函數，而是一個 polymorphic 的常數。有點像是 `Bounded` 中的 `minBound` 一樣。`mempty` 表示一個特定 monoid 的 identity。

再來我們看到 `mappend`，你可能已經猜到，他是一個接受兩個相同型別的值的二元函數，並回傳同樣的型別。不過要注意的是他的名字不太符合他真正的意思，他的名字隱含了我們要將兩個東西接在一起。儘管在 list 的情況下 `++` 的確將兩個 list 接起來，但 `*` 則否。他只不過將兩個數值做相乘。當我們再看到其他 `Monoid` 的 instance 時，我們會看到他們大部分都沒有接起來的做，所以不要用接起來的概念來想像 `mappend`，只要想像他們是接受兩個 monoid 的值並回傳另外一個就好了。

在 typeclass 定義中的最後一個函數是 `mconcat`。他接受一串 monoid 值，並將他們用 `mappend` 簡化成單一的值。他有一個預設的實作，就是從 `mempty` 作為起始值，然後用 `mappend` 來 fold。由於對於大部分的 instance 預設的實作就沒什麼問題，我們不會想要實作自己的 `mconcat`。當我們定義一個型別屬於 `Monoid` 的時候，多半實作 `mempty` 跟 `mappend` 就可以了。而 `mconcat` 就是因為對於一些 instance，有可能有比較有效率的方式來實作 `mconcat`。不過大多數情況都不需要。

在我們繼續接下去看幾個 `Monoid` 的例子前，我們來看一下 monoid law。我們提過必須有一個值作為 identity 以及一個遵守結合律的二元函數當作前提。我們是可以定義一個 `Monoid` 的 instance 卻不遵守這些定律的，但這樣寫出來的 instance 就沒有用了，因為我們在使用 `Monoid` 的時候都是依靠這些定律才可以稱作實質上的 monoid。所以我們必須確保他們遵守：

```
# ``mempty `mappend` x = x``
# ``x `mappend` mempty = x``
# ``(``x `mappend` y) `mappend` z = x `mappend` (y `mappend` z)`'
```

前兩個描述了 `mempty` 相對於 `mappend` 必須要表現成 identity。而第三個定律說了 `mappend` 必須要遵守結合律。也就是說我們做 `mappend` 順序並不重要。Haskell 不會自己檢查這些定律是否有被遵守。所以你必須自己小心地檢查他們。

## Lists are monoids

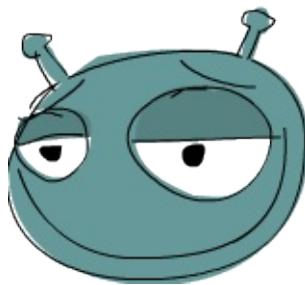
沒錯，list 是一種 monoid。正如我們先前看到的，`++` 跟空的 list `[]` 共同形成了一個 monoid。他的 instance 很簡單：

```
instance Monoid [a] where
    mempty = []
    mappend = (++)
```

`list` 是 `Monoid` typeclass 的一個 `instance`, 這跟他們裝的元素的型別無關。注意到我們寫 `instance Monoid [a]` 而非 `instance Monoid []`, 這是因為 `Monoid` 要求 `instance` 必須是具體型別。

我們試著跑跑看, 得到我們預期中的結果：

```
ghci> [1,2,3] `mappend` [4,5,6]
[1,2,3,4,5,6]
ghci> ("one" `mappend` "two") `mappend` "tree"
"onetwotree"
ghci> "one" `mappend` ("two" `mappend` "tree")
"onetwotree"
ghci> "one" `mappend` "two" `mappend` "tree"
"onetwotree"
ghci> "pang" `mappend` mempty
"pang"
ghci> mconcat [[1,2],[3,6],[9]]
[1,2,3,6,9]
ghci> mempty :: [a]
[]
```



注意到最後一行我們明白地標記出型別。這是因為如果只些 `mempty` 的話, GHCi 不會知道他是哪一個 `instance` 的 `mempty`, 所以我們必須清楚說出他是 `list instance` 的 `mempty`。我們可以使用一般化的型別 `[a]`, 因為空的 `list` 可以看作是屬於任何型別。

由於 `mconcat` 有一個預設的實作, 我們將某個型別定義成 `Monoid` 的型別時就可以自動地得到預設的實作。但對於 `list` 而言, `mconcat` 其實就是 `concat`。他接受一個裝有 `list` 的 `list`, 並把他用 `++` 來扁平化他。

`list` 的 `instance` 也遵守 monoid law。當我們有好幾個 `list` 並且用 `mappend` 來把他們串起來, 先後順序並不是很重要, 因為他們都是接在最後面。而且空的 `list` 也表現得如 `identity` 一樣。注意到 monoid 並不要求 `a `mappend` b` 等於 `b `mappend` a`。在 `list` 的情況下, 他們明顯不相等。

```
ghci> "one" `mappend` "two"
"onetwo"
ghci> "two" `mappend` "one"
"twoone"
```

這樣並沒有關係。`3 * 5` 跟 `5 * 3` 會相等只不過是乘法的性質而已，但沒有保證所有 monoid 都要遵守。

## Product and Sum

我們已經描述過將數值表現成一種 monoid 的方式。只要將 `*` 當作二元函數而 `1` 當作 identity 就好了。而且這不是唯一一種方式，另一種方式是將 `+` 作為二元函數而 `0` 作為 identity。

```
ghci> 0 + 4
4
ghci> 5 + 0
5
ghci> (1 + 3) + 5
9
ghci> 1 + (3 + 5)
9
```

他也遵守 monoid law，因為將 `0` 加上其他數值，都會是另外一者。而且加法也遵守結合律。所以現在我們有兩種方式來將數值表現成 monoid，那要選哪一個呢？其實我們不必要強迫定下來，還記得當同一種型別有好幾種表現成某個 typeclass 的方式時，我們可以用 `newtype` 來包裹現有的型別，然後再定義新的 instance。這樣就行了。

`Data.Monoid` 這個模組匯出了兩種型別，`Product` 跟 `Sum`。`Product` 定義如下：

```
newtype Product a = Product { getProduct :: a }
    deriving (Eq, Ord, Read, Show, Bounded)
```

簡單易懂，就是一個單一型別參數的 `newtype`，並 derive 一些性質。他的 `Monoid` 的 instance 長得像這樣：

```
instance Num a => Monoid (Product a) where
    mempty = Product 1
    Product x `mappend` Product y = Product (x * y)
```

`mempty` 只不過是將 `1` 包在 `Product` 中。`mappend` 則對 `Product` 的構造子做模式匹配，將兩個取出的數值相乘後再將結果放回去。就如你看到的，typeclass 定義前面有 `Num a` 的條件限制。所以他代表 `Product a` 對於所有屬於 `Num` 的 `a` 是一個 `Monoid`。要將 `Product`

a 作為一個 monoid 使用，我們需要用 newtype 來做包裹跟解開的動作。

```
ghci> getProduct $ Product 3 `mappend` Product 9
27
ghci> getProduct $ Product 3 `mappend` mempty
3
ghci> getProduct $ Product 3 `mappend` Product 4 `mappend` Product 2
24
ghci> getProduct . mconcat . map Product $ [3,4,2]
24
```

這當作 `Monoid` 的一個演練還不錯，但並不會有人覺得這會比 `3 * 9` 跟 `3 * 1` 這種方式來做乘法要好。但我們稍後會說明儘管像這種顯而易見的定義還是有他方便的地方。

`Sum` 跟 `Product` 定義的方式類似，我們也可以用類似的方式操作：

```
ghci> getSum $ Sum 2 `mappend` Sum 9
11
ghci> getSum $ mempty `mappend` Sum 3
3
ghci> getSum . mconcat . map Sum $ [1,2,3]
6
```

## Any and ALL

另一種可以有兩種表示成 monoid 方式的型別是 `Bool`。第一種方式是將 `||` 當作二元函數，而 `False` 作為 identity。這樣的意思是只要有任何一個參數是 `True` 他就回傳 `True`，否則回傳 `False`。所以如果我們使用 `False` 作為 identity，他會在跟 `False` 做 OR 時回傳 `False`，跟 `True` 做 OR 時回傳 `True`。`Any` 這個 newtype 是 `Monoid` 的一個 instance，並定義如下：

```
newtype Any = Any { getAny :: Bool }
    deriving (Eq, Ord, Read, Show, Bounded)
```

他的 instance 長得像這樣：

```
instance Monoid Any where
    mempty = Any False
    Any x `mappend` Any y = Any (x || y)
```

他叫做 `Any` 的理由是 `x `mappend` y` 當有任何一個是 `True` 時就會是 `True`。就算是更多個用 `mappend` 串起來的 `Any`，他也會在任何一個是 `True` 回傳 `True`。

```
ghci> getAny $ Any True `mappend` Any False
True
ghci> getAny $ mempty `mappend` Any True
True
ghci> getAny . mconcat . map Any $ [False, False, False, True]
True
ghci> getAny $ mempty `mappend` mempty
False
```

另一種 `Bool` 表現成 `Monoid` 的方式是用 `&&` 作為二元函數，而 `True` 作為 `identity`。只有當所有都是 `True` 的時候才會回傳 `True`。下面是他的 `newtype` 定義：

```
newtype All = All { getAll :: Bool }
    deriving (Eq, Ord, Read, Show, Bounded)
```

而這是他的 `instance`：

```
instance Monoid All where
    mempty = All True
    All x `mappend` All y = All (x && y)
```

當我們用 `mappend` 來串起 `All` 型別的值時，結果只有當所有 `mappend` 的值是 `True` 時才會是 `True`：

```
ghci> getAll $ mempty `mappend` All True
True
ghci> getAll $ mempty `mappend` All False
False
ghci> getAll . mconcat . map All $ [True, True, True]
True
ghci> getAll . mconcat . map All $ [True, True, False]
False
```

就如乘法跟加法一樣，我們通常寧願用二元函數來操作他們也不會用 `newtype` 來將他們包起來。不會將他們包成 `Any` 或 `All` 然後用 `mappend`, `mempty` 或 `mconcat` 來操作。通常使用 `or` 跟 `and`，他們接受一串 `Bool`，並只有當任意一個或是所有都是 `True` 的時候才回傳 `True`。

## The Ordering monoid

還記得 `Ordering` 型別嗎？他是比較運算之後得到的結果，包含三個值：`LT`，`EQ` 跟 `GT`，分別代表小於，等於跟大於：

```
ghci> 1 `compare` 2
LT
ghci> 2 `compare` 2
EQ
ghci> 3 `compare` 2
GT
```

針對 list, 數值跟布林值而言, 要找出 monoid 的行為只要去檢視已經定義的函數, 然後看看有沒有展現出 monoid 的特性就可以了, 但對於 `Ordering`, 我們就必須要更仔細一點才能看出來是否是一個 monoid, 但其實他的 `Monoid` instance 還蠻直覺的：

```
instance Monoid Ordering where
    mempty = EQ
    LT `mappend` _ = LT
    EQ `mappend` y = y
    GT `mappend` _ = GT
```



這個 `instance` 定義如下：當我們用 `mappend` 兩個 `Ordering` 型別的值時，左邊的會被保留下來。除非左邊的值是 `EQ`，那我們就會保留右邊的當作結果。而 `identity` 就是 `EQ`。乍看之下有點隨便，但實際上他是我們比較兩個英文字時所用的方法。我們先比較兩個字母是否相等，如果他們不一樣，那我們就知道那一個字在字典中會在前面。而如果兩個字母相等，那我們就繼續比較下一個字母，以此類推。

舉例來說，如果我們字典順序地比較 `"ox"` 跟 `"on"` 的話。我們會先比較兩個字的首個字母，看看他們是否相等，然後繼續比較第二個字母。我們看到 `'x'` 是比 `'n'` 要來得大，所以我們就知道如何比較兩個字了。而要瞭解為何 `EQ` 是 `identity`，我們可以注意到如果我們在兩個字中間的同樣位置塞入同樣的字母，那他們之間的字典順序並不會改變。`"oix"` 仍然比 `"oin"` 要大。

很重要的一件事是在 `Ordering` 的 `Monoid` 定義裡 `x `mappend` y` 並不等於 `y `mappend` x`。因為除非第一個參數是 `EQ`，不然結果就會是第一個參數。所以 `LT `mappend` GT` 等於 `LT`，然而 `GT `mappend` LT` 等於 `GT`。

```
ghci> LT `mappend` GT
LT
ghci> GT `mappend` LT
GT
ghci> mempty `mappend` LT
LT
ghci> mempty `mappend` GT
GT
```

所以這個 `monoid` 在什麼情況下會有用呢？假設你要寫一個比較兩個字串長度的函數，並回傳 `Ordering`。而且當字串一樣長的時候，我們不直接回傳 `EQ`，反而繼續用字典順序比較他們。一種實作的方式如下：

```
lengthCompare :: String -> String -> Ordering
lengthCompare x y = let a = length x `compare` length y
                     b = x `compare` y
                     in if a == EQ then b else a
```

我們稱呼比較長度的結果為 `a`，而比較字典順序的結果為 `b`，而當長度一樣時，我們就回傳字典順序。

如果善用我們 `Ordering` 是一種 `monoid` 這項知識，我們可以把我們的函數寫得更簡單些：

```
import Data.Monoid

lengthCompare :: String -> String -> Ordering
lengthCompare x y = (length x `compare` length y) `mappend`
                    (x `compare` y)
```

我們可以試著跑跑看：

```
ghci> lengthCompare "zen" "ants"
LT
ghci> lengthCompare "zen" "ant"
GT
```

要記住當我們使用 `mappend`。他在左邊不等於 `EQ` 的情況下都會回傳左邊的值。相反地則回傳右邊的值。這也是為什麼我們將我們認為比較重要的順序放在左邊的參數。如果我們要繼續延展這個函數，要讓他們比較母音的順序，並把這順序列為第二重要，那我們可以這樣修改他：

```
import Data.Monoid

lengthCompare :: String -> String -> Ordering
lengthCompare x y = (length x `compare` length y) `mappend`
    (vowels x `compare` vowels y) `mappend`
    (x `compare` y)
  where vowels = length . filter (`elem` "aeiou")
```

我們寫了一個輔助函數，他接受一個字串並回傳他有多少母音。他是先用 `filter` 來把字母濾到剩下 `"aeiou"`，然後再用 `length` 計算長度。

```
ghci> lengthCompare "zen" "anna"
LT
ghci> lengthCompare "zen" "ana"
LT
ghci> lengthCompare "zen" "ann"
GT
```

在第一個例子中我們看到長度不同所以回傳 `LT`，明顯地 `"zen"` 要短於 `"anna"`。在第二個例子中，長度是一樣的，但第二個字串有比較多的母音，所以結果仍然是 `LT`。在第三個範例中，兩個長度都相等，他們也有相同個數的母音，經由字典順序比較後得到 `"zen"` 比較大。

`Ordering` 的 `monoid` 允許我們用不同方式比較事物，並將這些順序也定義了依重要度不同的一个順序。

## Maybe the monoid

我們來看一下 `Maybe a` 是怎樣有多種方式來表現成 `Monoid` 的，並且說明哪些是比較有用的。一種將 `Maybe a` 當作 `monoid` 的方式就是他的 `a` 也是一個 `monoid`，而我們將 `mappend` 實作成使用包在 `Just` 裡面的值對應的 `mappend`。並且用 `Nothing` 當作 `identity`。所以如果我 `mappend` 兩個參數中有一個是 `Nothing`。那結果就會是另一邊的值。他的 `instance` 定義如下：

```
instance Monoid a => Monoid (Maybe a) where
  mempty = Nothing
  Nothing `mappend` m = m
  m `mappend` Nothing = m
  Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)
```

留意到 `class constraint`。他說明 `Maybe a` 只有在 `a` 是 `Monoid` 的情況下才會是一個 `Monoid`。如果我們 `mappend` 某個東西跟 `Nothing`。那結果就會是某個東西。如果我們 `mappend` 兩個 `Just`，那 `Just` 包住的結果就會 `mappended` 在一起並放回 `Just`。我們能

這麼做是因為 class constraint 保證了在 Just 中的值是 Monoid。

```
ghci> Nothing `mappend` Just "andy"
Just "andy"
ghci> Just LT `mappend` Nothing
Just LT
ghci> Just (Sum 3) `mappend` Just (Sum 4)
Just (Sum {getSum = 7})
```

這當你在處理有可能失敗的 monoid 的時候比較有用。有了這個 instance，我們就不必一一去檢查他們是否失敗，是否是 Nothing 或是 Just，我們可以直接將他們當作普通的 monoid。

但如果在 Maybe 中的型別不是 Monoid 呢？注意到在先前的 instance 定義中，唯一有依賴於 monoid 限制的情況就是在 mappend 兩個 Just 的時候。但如果我們不知道包在 Just 裡面的值究竟是不是 monoid，我們根本無法用 mappend 操作他們，所以該怎麼辦呢？一種方式就是直接丟掉第二個值而留下第一個值。這就是 First a 存在的目的，而這是他的定義：

```
newtype First a = First { getFirst :: Maybe a }
    deriving (Eq, Ord, Read, Show)
```

我們接受一個 Maybe a 並把他包成 newtype，Monoid 的定義如下：

```
instance Monoid (First a) where
    mempty = First Nothing
    First (Just x) `mappend` _ = First (Just x)
    First Nothing `mappend` x = x
```

正如我們說過得，mempty 就是包在 First 中的 Nothing。如果 mappend 的第一個參數是 Just，我們就直接忽略第二個參數。如果第一個參數是 Nothing，那我們就將第二個參數當作結果。並不管他究竟是 Just 或是 Nothing：

```
ghci> getFirst $ First (Just 'a') `mappend` First (Just 'b')
Just 'a'
ghci> getFirst $ First Nothing `mappend` First (Just 'b')
Just 'b'
ghci> getFirst $ First (Just 'a') `mappend` First Nothing
Just 'a'
```

First 在我們有一大串 Maybe 而且想知道他們之中就竟有沒有 Just 的時候很有用。可以利用 mconcat：

```
ghci> getFirst . mconcat . map First $ [Nothing, Just 9, Just 10]
Just 9
```

如果我們希望定義一個 `Maybe a` 的 monoid，讓他當 `mappend` 的兩個參數都是 `Just` 的時候將第二個參數當作結果。`Data.Monoid` 中有一個現成的 `Last a`，他很像是 `First a`，只差在 `mappend` 跟 `mconcat` 會保留最後一個非 `Nothing` 的值。

```
ghci> getLast . mconcat . map Last $ [Nothing, Just 9, Just 10]
Just 10
ghci> getLast $ Last (Just "one") `mappend` Last (Just "two")
Just "two"
```

## Using monoids to fold data structures

另一種有趣的 monoid 使用方式就是讓他來幫助我們 fold 一些資料結構。到目前為止我們只有 fold list。但 list 並不是唯一一種可以 fold 的資料結構。我們幾乎可以 fold 任何一種資料結構。像是 tree 也是一種常見的可以 fold 的資料結構。

由於有太多種資料結構可以 fold 了，所以我們定義了 `Foldable` 這個 typeclass。就像 `Functor` 是定義可以 map over 的結構。`Foldable` 是定義可以 fold 的結構。在 `Data.Foldable` 中有定義了一些有用的函數，但他們名稱跟 `Prelude` 中的名稱衝突。所以最好是用 qualified 的方式 import 他們：

```
import qualified Foldable as F
```

為了少打一些字，我們將他們 import qualified 成 `F`。所以這個 typeclass 中定義了哪些函數呢？有 `foldr`，`foldl`，`foldr1` 跟 `foldl1`。你會說我們已經知道這些函數了，他們有什麼不一樣的地方嗎？我們來比較一下 `Foldable` 中的 `foldr` 跟 `Prelude` 中的 `foldr` 的型別異同：

```
ghci> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
ghci> :t F.foldr
F.foldr :: (F.Foldable t) => (a -> b -> b) -> b -> t a -> b
```

儘管 `foldr` 接受一個 list 並將他 fold 起來，`Data.Foldable` 中的 `foldr` 接受任何可以 fold 的型別。並不只是 list。而兩個 `foldr` 對於 list 的結果是相同的：

```
ghci> foldr (*) 1 [1,2,3]
6
ghci> F.foldr (*) 1 [1,2,3]
6
```

那有哪些資料結構支援 fold 呢？首先我們有 `Maybe`：

```
ghci> F.foldl (+) 2 (Just 9)
11
ghci> F.foldr (||) False (Just True)
True
```

但 fold 一個 `Maybe` 並沒什麼新意。畢竟當他是 `Just` 的時候表現得像是只有單一元素的 list，而當他是 `Nothing` 的時候就像是空的 list 一樣。所以我們來看一些比較複雜的資料結構。

還記得 Making Our Own Types and Typeclass 章節中的樹狀的資料結構嗎？我們是這樣定義的：

```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show, Read, Eq)
```

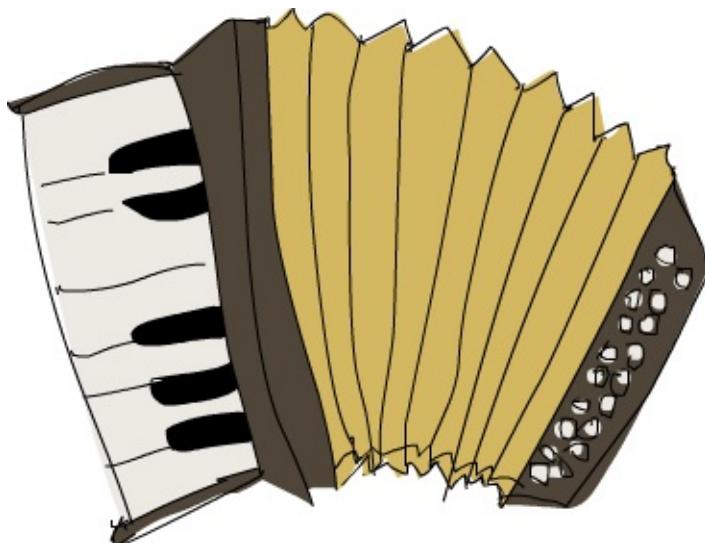
我們說一棵樹要不就是一棵空的樹要不然就是一個包含值的節點，並且還指向另外兩棵樹。定義他之後，我們將他定義成 `Functor` 的 instance，因此可以 `fmap` 他。現在我們要將他定義成 `Foldable` 的 instance，這樣我們就可以 fold 他。要定義成 `Foldable` 的一種方式就是實作 `foldr`。但另一種比較簡單的方式就是實作 `foldMap`，他也屬於 `Foldable` typeclass。`foldMap` 的型別如下：

```
foldMap :: (Monoid m, Foldable t) => (a -> m) -> t a -> m
```

第一個參數是一個函數，這個函數接受 foldable 資料結構中包含的元素的型別，並回傳一個 monoid。他第二個參數是一個 foldable 的結構，並包含型別 `a` 的元素。他將第一個函數來 map over 這個 foldable 的結構，因此得到一個包含 monoid 的 foldable 結構。然後用 `mappend` 來簡化這些 monoid，最後得到單一的一個 monoid。這個函數聽起來不太容易理解，但我們下面會看到他其實很容易實作。而且好消息是只要實作了這個函數就可以讓我們的函數成為 `Foldable`。所以我們只要實作某個型別的 `foldMap`，我們就可以得到那個型別的 `foldr` 跟 `foldl`。

這就是我們如何定義 `Tree` 成為 `Foldable` 的：

```
instance F.Foldable Tree where
    foldMap f Empty = mempty
    foldMap f (Node x l r) = F.foldMap f l `mappend`  
                                f x           `mappend`  
                                F.foldMap f r
```



我們是這樣思考的：如果我們寫一個函數，他接受樹中的一個元素並回傳一個 monoid，那我們要怎麼簡化整棵樹到只有單一一個 monoid？當我們在對樹做 `fmap` 的時候，我們將那函數套用至節點上，並遞迴地套用至左子樹以及右子樹。這邊我們不只是 `map` 一個函數而已，我們還要求要把結果用 `mappend` 簡化成只有單一一個 monoid 值。首先我們考慮樹為空的情形，一棵沒有值也沒有子樹的情形。由於沒有值我們也沒辦法將他套用上面轉換成 monoid 的函數，所以當樹為空的時候，結果應該要是 `mempty`。

在非空節點的情形下比較有趣，他包含一個值跟兩棵子樹。在這種情況下，我們遞迴地做 `foldMap`，用 `f` 來套用到左子樹跟右子樹上。要記住我們的 `foldMap` 只會得到單一的 monoid 值。我們也會套用 `f` 到節點中的值。這樣我們就得到三個 monoid 值，有兩個來自簡化子樹的結果，還有一個是套用 `f` 到節點中的值的結果。而我們需要將這三個值整合成單一個值。要達成這個目的我們使用 `mappend`，而且自然地會想到照左子樹，節點值以及右子樹的順序來簡化。

注意到我們並不一定要提供一個將普通值轉成 monoid 的函數。我們只是把他當作是 `foldMap` 的參數，我們要決定的只是如何套用那個函數，來把得到的 monoid 們簡化成單一結果。

現在我們有樹的 `Foldable` instance，而 `foldr` 跟 `foldl` 也有預設的實作了。考慮下面這棵樹：

```
testTree = Node 5
  (Node 3
    (Node 1 Empty Empty)
    (Node 6 Empty Empty)
  )
  (Node 9
    (Node 8 Empty Empty)
    (Node 10 Empty Empty)
  )
```

他的 root 是 5，而他左邊下來分別是 3，再來是 1 跟 6。而右邊下來是 9，再來是 8 跟 10。有了 Foldable 的定義，我們就能像對 list 做 fold 一樣對樹做 fold：

```
ghci> F.foldl (+) 0 testTree
42
ghci> F.foldl (*) 1 testTree
64800
```

`foldMap` 不只是定義 Foldable 新的 instance 有用。他也對簡化我們的結構至單一 monoid 值有用。舉例來說，如果我們想要知道我們的樹中有沒有 3，我們可以這樣做：

```
ghci> getAny $ F.foldMap (\x -> Any $ x == 3) testTree
True
```

這邊 `\x -> Any $ x == 3` 是一個接受一個數值並回傳一個 monoid 的函數，也就是一個包在 Any 中的 Bool。`foldMap` 將這個函數套用至樹的每一個節點，並把結果用 `mappend` 簡化成單一 monoid。如果我們這樣做：

```
ghci> getAny $ F.foldMap (\x -> Any $ x > 15) testTree
False
```

經過套用 lambda 之後我們所有的節點都會是 Any False。但 `mappend` 必須要至少吃到一個 True 才能讓最後的結果變成 True。這也是為什麼結果會是 False，因為我們樹中所有的值都小於等於 15。

我們也能將 `foldMap` 配合 `\x -> [x]` 使用來將我們的樹轉成 list。經過套用那個函數後，所有節點都變成包含單一元素的 list。最後用 `mappend` 將這些單一元素的 list 轉成一個裝有全部元素的 list：

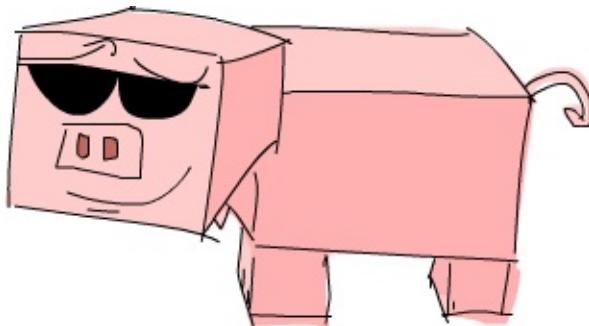
```
ghci> F.foldMap (\x -> [x]) testTree
[1,3,6,5,8,9,10]
```

這個小技巧並不限於樹而已，他可以被套用在任何 Foldable 上。

# 來看看幾種 Monad

當我們第一次談到 Functor 的時候，我們了解到他是一個抽象概念，代表是一種可以被 map over 的值。然後我們再將其概念提升到 Applicative Functor，他代表一種帶有 context 的型態，我們可以用函數操作他而且同時還保有他的 context。

在這一章，我們會學到 Monad，基本上他是一種加強版的 Applicative Functor，正如 Applicative Functor 是 Functor 的加強版一樣。



我們介紹到 Functor 是因為我們觀察到有許多型態都可以被 function 給 map over，了解到這個目的，便抽象化了 Functor 這個 typeclass 出來。但這讓我們想問：如果給定一個 `a -> b` 的函數以及 `f a` 的型態，我們要如何將函數 map over 這個型態而得到 `f b`？我們知道要如何 map over `Maybe a`, `[a]` 以及 `IO a`。我們甚至還知道如何用 `a -> b` map over `r -> a`，並且會得到 `r -> b`。要回答這個問題，我們只需要看 `fmap` 的型態就好了：

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

然後只要針對 `Functor` instance 撰寫對應的實作。

之後我們又看到一些可以針對 Functor 改進的地方，例如 `a -> b` 也被包在一個 Functor value 裡面呢？像是 `Just (*3)`，我們要如何 apply `Just 5` 紿他？如果我們不要 apply `Just 5` 而是 `Nothing` 呢？甚至給定 `[(*2), (+4)]`，我們要如何 apply 他們到 `[1, 2, 3]` 呢？對於此，我們抽象出 Applicative typeclass，這就是我們想要問的問題：

```
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
```

我們也看到我們可以將一個正常的值包在一個資料型態中。例如說我們可以拿一個 `1` 然後把他包成 `Just 1`。或是把他包成 `[1]`。也可以是一個 I/O action 會產生一個 `1`。這樣包裝的 function 我們叫他做 `pure`。

如我們說得，一個 applicative value 可以被看作一個有附加 context 的值。例如說，'a' 只是一個普通的字元，但 Just 'a' 是一個附加了 context 的字元。他不是 char 而是 Maybe char，這型態告訴我們這個值可能是一個字元，也可能什麼都沒有。

來看看 Applicative typeclass 怎樣讓我們用普通的 function 操作他們，同時還保有 context：

```
ghci> (*) <$> Just 2 <*> Just 8
Just 16
ghci> (++) <$> Just "klingon" <*> Nothing
Nothing
ghci> (-) <$> [3,4] <*> [1,2,3]
[2,1,0,3,2,1]
```

所以我們可以視他們為 applicative values，Maybe a 代表可能會失敗的 computation，[a] 代表同時有好多結果的 computation (non-deterministic computation)，而 IO a 代表會有 side-effects 的 computation。

Monad 是一個從 Applicative functors 很自然的一個演進結果。對於他們我們主要考量的點是：如果你有一個具有 context 的值 m a，你能如何把他丟進一個只接受普通值 a 的函數中，並回傳一個具有 context 的值？也就是說，你如何套用一個型態為 a -> m b 的函數至 m a？基本上，我們要求的函數是：

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

如果我們有一個漂亮的值跟一個函數接受普通的值但回傳漂亮的值，那我們要如何要把漂亮的值丟進函數中？這就是我們使用 Monad 時所要考量的事情。我們不寫成 f a 而寫成 m a 是因為 m 代表的是 Monad，但 monad 不過就是支援 >>= 操作的 applicative functors。>>= 我們稱呼他為 bind。

當我們有一個普通值 a 跟一個普通函數 a -> b，要套用函數是一件很簡單的事。但當你在處理具有 context 的值時，就需要多考慮些東西，要如何把漂亮的值餵進函數中，並如何考慮他們的行為，但你將會了解到他們其實不難。

## 動手做做看：Maybe Monad



現在對於什麼是 Monad 已經有了些模糊的概念，我們來看看要如何讓這概念更具體一些。

不意外地，Maybe 是一個 Monad，所以讓我們對於他多探討些，看看是否能跟我們所知的 Monad 概念結合起來。

到這邊要確定你了解什麼是 Applicatives。如果你知道好幾種 ``Applicative`` 的 instance 還有他們代表的意

一個 `Maybe a` 型態的值代表型態為 `a` 的值而且具備一個可能造成錯誤的 context。而 `Just "dharma"` 的值代表他不是一個 `"dharma"` 的字串就是字串不見時的 `Nothing`。如果你把字串當作計算的結果，`Nothing` 就代表計算失敗了。

當我們把 `Maybe` 視作 functor，我們其實要的是一個 `fmap` 來把一個函數針對其中的元素做套用。他會對 `Just` 中的元素進行套用，要不然就是保留 `Nothing` 的狀態，其代表裡面根本沒有元素。

```
ghci> fmap (++)!"") (Just "wisdom")
Just "wisdom!"
ghci> fmap (++)!"") Nothing
Nothing
```

或者視為一個 applicative functor，他也有類似的作用。只是 applicative 也把函數包了起來。`Maybe` 作為一個 applicative functor，我們能用 `<*>` 來套用一個存在 `Maybe` 中的函數至包在另外一個 `Maybe` 中的值。他們都必須是包在 `Just` 來代表值存在，要不然其實就是 `Nothing`。當你在想套用函數到值上面的時候，缺少了函數或是值都會造成錯誤，所以這樣做是很合理的。

```
ghci> Just (+3) <*> Just 3
Just 6
ghci> Nothing <*> Just "greed"
Nothing
ghci> Just ord <*> Nothing
Nothing
```

當我們用 applicative 的方式套用函數至 `Maybe` 型態的值時，就跟上面描述的差不多。過程中所有值都必須是 `Just`，要不然結果一定會是 `Nothing`。

```
ghci> max <$> Just 3 <*> Just 6
Just 6
ghci> max <$> Just 3 <*> Nothing
Nothing
```

我們來思考一下要怎麼為 `Maybe` 實作 `>=`。正如我們之前提到的，`>=` 接受一個 monadic value，以及一個接受普通值的函數，這函數會回傳一個 monadic value。`>=` 會幫我們套用這個函數到這個 monadic value。在函數只接受普通值的情況下，函數是如何作到這件事的呢？要作到這件事，他必須要考慮到 monadic value 的 context。

在這個案例中，`>=` 會接受一個 `Maybe a` 以及一個型態為 `a -> Maybe b` 的函數。他會套用函數到 `Maybe a`。要釐清他怎麼作到的，首先我們注意到 `Maybe` 的 applicative functor 特性。假設我們有一個函數 `\x -> Just (x+1)`。他接受一個數字，把他加 `1` 後再包回 `Just`。

```
ghci> (\x -> Just (x+1)) 1
Just 2
ghci> (\x -> Just (x+1)) 100
Just 101
```

如果我們餵給函數 `1`，他會計算成 `Just 2`。如果我們餵給函數 `100`，那結果便是 `Just 101`。但假如我們餵一個 `Maybe` 的值給函數呢？如果我們把 `Maybe` 想成一個 applicative functor，那答案便很清楚。如果我們拿到一個 `Just`，就把包在 `Just` 裡面的值餵給函數。如果我們拿到一個 `Nothing`，我們就說結果是 `Nothing`。

我們呼叫 `applyMaybe` 而不呼叫 `>=`。他接受 `Maybe a` 跟一個回傳 `Maybe b` 的函數，並套用函數至 `Maybe a`。

```
applyMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
applyMaybe Nothing f = Nothing
applyMaybe (Just x) f = f x
```

我們套用一個 infix 函數，這樣 `Maybe` 的值可以寫在左邊且函數是在右邊：

```
ghci> Just 3 `applyMaybe` \x -> Just (x+1)
Just 4
ghci> Just "smile" `applyMaybe` \x -> Just (x ++ ":")"
Just "smile :"
ghci> Nothing `applyMaybe` \x -> Just (x+1)
Nothing
ghci> Nothing `applyMaybe` \x -> Just (x ++ ":")
Nothing
```

在上述的範例中，我們看到在套用 `applyMaybe` 的時候，函數是套用在 `Just` 裡面的值。當我們試圖套用到 `Nothing`，那整個結果便是 `Nothing`。假如函數回傳 `Nothing` 呢？

```
ghci> Just 3 `applyMaybe` \x -> if x > 2 then Just x else Nothing
Just 3
ghci> Just 1 `applyMaybe` \x -> if x > 2 then Just x else Nothing
Nothing
```

這正是我們期待的結果。如果左邊的 monadic value 是 `Nothing`，那整個結果就是 `Nothing`。如果右邊的函數是 `Nothing`，那結果也會是 `Nothing`。這跟我們之前把 `Maybe` 當作 applicative 時，過程中有任何一個 `Nothing` 整個結果就會是 `Nothing` 一樣。

對於 `Maybe` 而言，我們已經找到一個方法處理漂亮值的方式。我們作到這件事的同時，也保留了 `Maybe` 代表可能造成錯誤的計算的意義。

你可能會問，這樣的結果有用嗎？由於 applicative functors 讓我們可以拿一個接受普通值的函數，並讓他可以操作具有 context 的值，這樣看起來 applicative functors 好像比 monad 強。但我們會看到 monad 也能作到，因為他只是 applicative functors 的升級版。他們同時也能作到 applicative functors 不能作到的事情。

稍候我們會再繼續探討 `Maybe`，但我們先來看看 monad 的 type class。

## Monad type class

正如 functors 有 `Functor` 這個 type class，而 applicative functors 有一個 `Applicative` 這個 type class，monad 也有他自己的 type class：`Monad` 他看起來像這樣：

```
class Monad m where
    return :: a -> m a

    (=>) :: m a -> (a -> m b) -> m b

    (=>) :: m a -> m b -> m b
    x >> y = x => \_ -> y

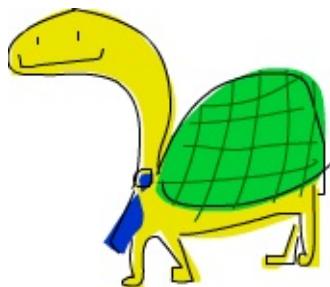
    fail :: String -> m a
    fail msg = error msg
```



我們從第一行開始看。他說 `class Monad m where`。但我們之前不是提到 monad 是 applicative functors 的加強版嗎？不是應該有一個限制說一個型態必須先是一個 applicative functor 才可能是一個 monad 嗎？像是 `class (Applicative m) = > Monad m where`。他的確應該要有，但當 Haskell 被創造的早期，人們沒有想到 applicative functor 適合被放進語言中，所以最後沒有這個限制。但的確每個 monad 都是 applicative functor，即使 `Monad` 並沒有這麼宣告。

在 `Monad` typeclass 中定義的第一個函數是 `return`。他其實等價於 `pure`，只是名字不同罷了。他的型態是 `(Monad m) => a -> m a`。他接受一個普通值並把他放進一個最小的 context 中。也就是說他把普通值包進一個 monad 裡面。他跟 Applicative 裡面 `pure` 函數做的事情一樣，所以說其實我們已經認識了 `return`。我們已經用過 `return` 來處理一些 I/O。我們用他來做一些假的 I/O，印出一些值。對於 `Maybe` 來說他就是接受一個普通值然後包進 `Just`。

提醒一下：```return``` 跟其他語言中的 ```return``` 是完全不一樣的。他並不是結束一個函數的執行，他只不過是把



接下來定義的函數是 `bind`: `>>=`。他就像是函數套用一樣，只差在他不接受普通值，他是接受一個 monadic value（也就是具有 context 的值）並且把他餵給一個接受普通值的函數，並回傳一個 monadic value。

接下來，我們定義了 `>>`。我們不會介紹他，因為他有一個事先定義好的實作，基本上我們在實作 `Monad typeclass` 的時候都不會去理他。

最後一個函數是 `fail`。我們通常在我們程式中不會具體寫出來。他是被 Haskell 用在處理語法錯誤的情況。我們目前不需要太在意 `fail`。

我們知道了 `Monad typeclass` 長什麼樣子，我們來看一下 `Maybe` 的 `Monad instance`。

```
instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
    Just x >>= f = f x
    fail _ = Nothing
```

`return` 跟 `pure` 是等價的。這沒什麼困難的。我們跟我們在定義 `Applicative` 的時候做一樣的事，只是把他用 `Just` 包起來。

`>>=` 跟我們的 `applyMaybe` 是一樣的。當我們將 `Maybe a` 塞給我們的函數，我們保留住 `context`，並且在輸入是 `Nothing` 的時候回傳 `Nothing`。畢竟當沒有值的時候套用我們的函數是沒有意義的。當輸入是 `Just` 的時候則套用 `f` 並將他包在 `Just` 裡面。

我們可以試著感覺一下 `Maybe` 是怎樣表現成 `Monad` 的。

```
ghci> return "WHAT" :: Maybe String
Just "WHAT"
ghci> Just 9 >>= \x -> return (x*10)
Just 90
ghci> Nothing >>= \x -> return (x*10)
Nothing
```

第一行沒什麼了不起，我們已經知道 `return` 就是 `pure` 而我們又對 `Maybe` 操作過 `pure` 了。至於下兩行就比較有趣點。

留意我們是如何把 `Just 9` 餵給 `\x -> return (x*10)`。在函數中 `x` 綁定到 `9`。他看起來好像我們能不用 pattern matching 的方式就從 `Maybe` 中抽取出值。但我們並沒有喪失掉 `Maybe` 的 context，當他是 `Nothing` 的時候，`>>=` 的結果也會是 `Nothing`。

## 走鋼索



我們已經知道要如何把 `Maybe a` 餵進 `a -> Maybe b` 這樣的函數。我們可以看看我們如何重複使用 `>>=` 來處理多個 `Maybe a` 的值。

首先來說個小故事。皮爾斯決定要辭掉他的工作改行試著走鋼索。他對走鋼索蠻在行的，不過仍有個小問題。就是鳥會停在他拿的平衡竿上。他們會飛過來停一小會兒，然後再飛走。這樣的情況在兩邊的鳥的數量一樣時並不是個太大的問題。但有時候，所有的鳥都會想要停在同一邊，皮爾斯就失去了平衡，就會讓他從鋼索上掉下去。

我們這邊假設兩邊的鳥差異在三個之內的時候，皮爾斯仍能保持平衡。所以如果是右邊有一隻，左邊有四隻的話，那還撐得住。但如果左邊有五隻，那就會失去平衡。

我們要寫個程式來模擬整個情況。我們想看看皮爾斯究竟在好幾隻鳥來來去去後是否還能撐住。例如說，我們想看看先來了一隻鳥停在左邊，然後來了四隻停在右邊，然後左邊那隻飛走了。之後會是什麼情形。

我們用一對整數來代表我們的平衡竿狀態。頭一個位置代表左邊的鳥的數量，第二個位置代表右邊的鳥的數量。

```
type Birds = Int
type Pole = (Birds,Birds)
```

由於我們用整數來代表有多少隻鳥，我們便先來定義 `Int` 的同義型態，叫做 `Birds`。然後我們把 `(Birds, Birds)` 定義成 `Pole`。

接下來，我們定義一個函數他接受一個數字，然後把他放在竿子的左邊，還有另外一個函數放在右邊。

```
landLeft :: Birds -> Pole -> Pole
landLeft n (left,right) = (left + n,right)

landRight :: Birds -> Pole -> Pole
landRight n (left,right) = (left,right + n)
```

我們來試著執行看看：

```
ghci> landLeft 2 (0,0)
(2,0)
ghci> landRight 1 (1,2)
(1,3)
ghci> landRight (-1) (1,2)
(1,1)
```

要模擬鳥飛走的話我們只要給定一個負數就好了。由於這些操作是接受 `Pole` 並回傳 `Pole`，所以我們可以把函數串在一起。

```
ghci> landLeft 2 (landRight 1 (landLeft 1 (0,0)))
(3,1)
```

當我們餵 `(0,0)` 給 `landLeft 1` 時，我們會得到 `(1,0)`。接著我們模擬右邊又停了一隻鳥，狀態就變成 `(1,1)`。最後又有兩隻鳥停在左邊，狀態變成 `(3,1)`。我們這邊的寫法是先寫函數名稱，然後再套用參數。但如果先寫 `pole` 再寫函數名稱會比較清楚，所以我們會想定義一個函數

```
x -: f = f x
```

我們能先套用參數然後再寫函數名稱：

```
ghci> 100 -: (*3)
300
ghci> True -: not
False
ghci> (0,0) -: landLeft 2
(2,0)
```

有了這個函數，我們便能寫得比較好讀一些：

```
ghci> (0,0) -: landLeft 1 -: landRight 1 -: landLeft 2
(3,1)
```

這個範例跟先前的範例是等價的，只不過好讀許多。很清楚的看出我們是從 `(0,0)` 開始，然後停了一隻在左邊，接著右邊又有一隻，最後左邊多了兩隻。

到目前為止沒什麼問題，但如果我們要停 10 隻在左邊呢？

```
ghci> landLeft 10 (0,3)
(10,3)
```

你說左邊有 10 隻右邊卻只有 3 隻？那不是早就應該掉下去了？這個例子太明顯了，如果換個比較不明顯的例子。

```
ghci> (0,0) -: landLeft 1 -: landRight 4 -: landLeft (-1) -: landRight (-2)
(0,2)
```

表面看起來沒什麼問題，但如果你仔細看的話，有一瞬間是右邊有四隻，但左邊沒有鳥。要修正這個錯誤，我們要重新檢視 `landLeft` 跟 `landRight`。我們其實是希望這些函數產生失敗的情況。那就是在維持平衡的時候回傳新的 pole，但失敗的時候告訴我們失敗了。這時候 `Maybe` 就剛剛好是我們要的 context 了。我們用 `Maybe` 重新寫一次：

```
landLeft :: Birds -> Pole -> Maybe Pole
landLeft n (left,right)
| abs ((left + n) - right) < 4 = Just (left + n, right)
| otherwise = Nothing

landRight :: Birds -> Pole -> Maybe Pole
landRight n (left,right)
| abs (left - (right + n)) < 4 = Just (left, right + n)
| otherwise = Nothing
```

現在這些函數不回傳 `Pole` 而回傳 `Maybe Pole` 了。他們仍接受鳥的數量跟舊的的 `pole`，但他們現在會檢查是否有太多鳥會造成皮爾斯失去平衡。我們用 guards 來檢查是否有差異超過三的情況。如果沒有，那就包一個在 `Just` 中的新的 `pole`，如果是，那就回傳 `Nothing`。

再來執行看看：

```
ghci> landLeft 2 (0,0)
Just (2,0)
ghci> landLeft 10 (0,3)
Nothing
```

一如預期，當皮爾斯不會掉下去的時候，我們就得到一個包在 `Just` 中的新 `pole`。當太多鳥停在同一邊的時候，我們就會拿到 `Nothing`。這樣很棒，但我們卻不知道怎麼把東西串在一起了。我們不能做 `landLeft 1 (landRight 1 (0,0))`，因為當我們對 `(0,0)` 使用 `landRight 1` 時，我們不是拿到 `Pole` 而是拿到 `Maybe Pole`。`landLeft 1` 會拿到 `Pole` 而不是拿到 `Maybe Pole`。

我們需要一種方法可以把拿到的 `Maybe Pole` 塞到拿 `Pole` 的函數中，然後回傳 `Maybe Pole`。而我們有 `>>=`，他對 `Maybe` 做的事就是我們要的

```
ghci> landRight 1 (0,0) >>= landLeft 2
Just (2,1)
```

`landLeft 2` 的型態是 `Pole -> Maybe Pole`。我們不能餵給他 `Maybe Pole` 的東西。而 `landRight 1 (0,0)` 的結果就是 `Maybe Pole`，所以我們用 `>>=` 來接受一個有 context 的值然後拿給 `landLeft 2`。`>>=` 的確讓我們把 `Maybe` 當作有 context 的值，因為當我們丟 `Nothing` 給 `landLeft 2` 的時候，結果會是 `Nothing`。

```
ghci> Nothing >>= landLeft 2
Nothing
```

這樣我們可以把這些新寫的用 `>>=` 串在一起。讓 monadic value 可以餵進只吃普通值的函數。

來看看些例子：

```
ghci> return (0,0) >>= landRight 2 >>= landLeft 2 >>= landRight 2
Just (2,4)
```

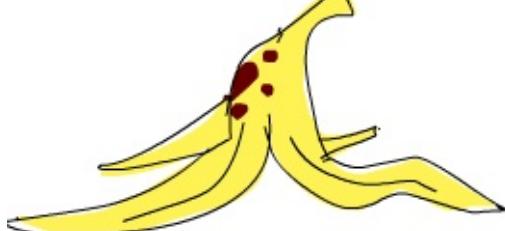
我們最開始用 `return` 回傳一個 `pole` 並把他包在 `Just` 裡面。我們可以像往常套用 `landRight 2`，不過我們不那麼做，我們改用 `>>=`。`Just (0,0)` 被餵到 `landRight 2`，得到 `Just (0,2)`。接著被餵到 `landLeft 2`，得到 `Just (2,2)`。

還記得我們之前引入失敗情況的例子嗎？

```
ghci> (0,0) -: landLeft 1 -: landRight 4 -: landLeft (-1) -: landRight (-2)
(0,2)
```

之前的例子並不會反應失敗的情況。但如果我們用 `>=` 的話就可以得到失敗的結果。

```
ghci> return (0,0) >= landLeft 1 >= landRight 4 >= landLeft (-1) >= landRight (-2)
Nothing
```



正如預期的，最後的情形代表了失敗的情況。我們再進一步看看這是怎麼產生的。首先 `return` 把 `(0,0)` 放到一個最小的 context 中，得到 `Just (0,0)`。然後是 `Just (0,0) >= landLeft 1`。由於 `Just (0,0)` 是一個 `Just` 的值。`landLeft 1` 被套用至 `(0,0)` 而得到 `Just (1,0)`。這反應了我們仍保持在平衡的狀態。接著是 `Just (1,0) >= landright 4` 而得到了 `Just (1,4)`。距離不平衡只有一步之遙了。他又被餵給 `landLeft (-1)`，這組合成了 `landLeft (-1) (1,4)`。由於失去了平衡，我們變得到了 `Nothing`。而我們把 `Nothing` 餵給 `landRight (-2)`，由於他是 `Nothing`，也就自動得到了 `Nothing`。

如果只把 `Maybe` 當作 applicative 用的話是沒有辦法達到我們要的效果的。你試著做一遍就會卡住。因為 applicative functor 並不允許 applicative value 之間有彈性的互動。他們最多就是讓我們可以用 applicative style 來傳遞參數給函數。applicative operators 能拿到他們的結果並把他用 applicative 的方式餵給另一個函數，並把最終的 applicative 值放在一起。但在每一步之間並沒有太多允許我們作手腳的機會。而我們的範例需要每一步都倚賴前一步的結果。當每一隻鳥降落的時候，我們都會把前一步的結果拿出來看看。好知道結果到底應該成功或失敗。

我們也能寫出一個函數，完全不管現在究竟有幾隻鳥停在竿子上，只是要害皮爾斯滑倒。我們可以稱呼這個函數叫做 `banana`：

```
banana :: Pole -> Maybe Pole
banana _ = Nothing
```

現在我們能把香蕉皮串到我們的過程中。他絕對會讓遇到的人滑倒。他完全不管前面的狀態是什麼都會產生失敗。

```
ghci> return (0,0) >>= landLeft 1 >>= banana >>= landRight 1
Nothing
```

Just (1,0) 被餵給 banana , 而產生了 Nothing , 之後所有的結果便都是 Nothing 了。

要同樣表示這種忽略前面的結果，只注重眼前的 monadic value 的情況，其實我們可以用 >> 來表達。

```
(>>) :: (Monad m) => m a -> m b -> m b
m >> n = m >>= \_ -> n
```

一般來講，碰到一個完全忽略前面狀態的函數，他就應該只會回傳他想回傳的值而已。但碰到 Monad，他們的 context 還是必須要被考慮到。來看一下 >> 串接 Maybe 的情況。

```
ghci> Nothing >> Just 3
Nothing
ghci> Just 3 >> Just 4
Just 4
ghci> Just 3 >> Nothing
Nothing
```

如果你把 >> 換成 >>= \\_ -> , 那就很容易看出他的意思。

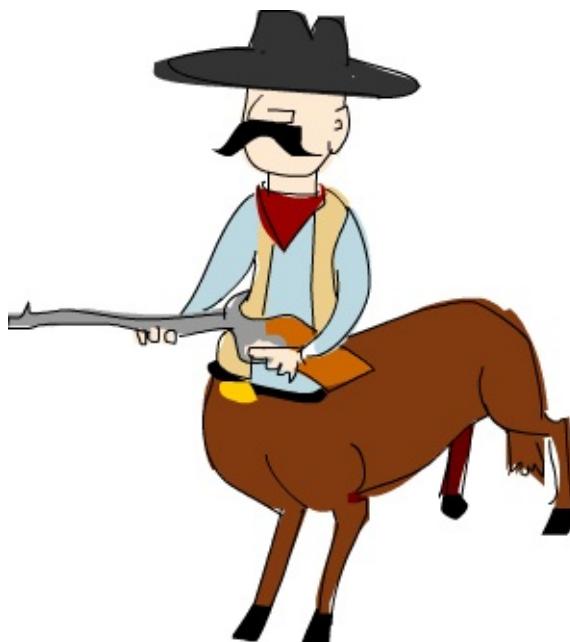
我們也可以把 banana 改用 >> 跟 Nothing 來表達：

```
ghci> return (0,0) >>= landLeft 1 >> Nothing >>= landRight 1
Nothing
```

我們得到了保證的失敗。

我們也可以看看假如我們故意不用把 Maybe 視為有 context 的值的寫法。他會長得像這樣：

```
routine :: Maybe Pole
routine = case landLeft 1 (0,0) of
    Nothing -> Nothing
    Just pole1 -> case landRight 4 pole1 of
        Nothing -> Nothing
        Just pole2 -> case landLeft 2 pole2 of
            Nothing -> Nothing
            Just pole3 -> landLeft 1 pole3
```



左邊先停了一隻鳥，然後我們停下來檢查有沒有失敗。當失敗的時候我們回傳 `Nothing`。當成功的時候，我們在右邊停一隻鳥，然後再重複前面做的事情。把這些瑣事轉換成 `>=` 證明了 `Maybe` `Monad` 的力量，可以省去我們不少的時間。

注意到 `Maybe` 對 `>=` 的實作，他其實就是在做碰到 `Nothing` 就會傳 `Nothing`，碰到正確值就繼續用 `Just` 傳遞值。

在這個章節中，我們看過了好幾個函數，也見識了用 `Maybe` `monad` 來表示失敗的 `context` 的力量。把普通的函數套用換成了 `>=`，讓我們可以輕鬆地應付可能會失敗的情況，並幫我們傳遞 `context`。這邊的 `context` 就代表失敗的可能性，當我們套用函數到 `context` 的時候，就代表考慮進了失敗的情況。

## do 表示法

`Monad` 在 Haskell 中是十分重要的，所以我們還特別為了操作他設置了特別的語法：`do` 表示法。我們在介紹 I/O 的時候已經用過 `do` 來把小的 I/O action 串在一起了。其實 `do` 並不只是可以用在 `IO`，他可以用在任何 `monad` 上。他的原則是簡單明瞭，把 `monadic value` 串成一串。我們這邊來細看 `do` 是如何使用，以及為什麼我們十分倚賴他。

來看一下熟悉的例子：

```
ghci> Just 3 >>= (\x -> Just (show x ++ "!"))
Just "3!"
```

你說這沒什麼了不起，不過就是把 `monadic value` 餵給一個函數罷了。其中 `x` 就指定成 `3`。也從 `monadic value` 變成了普通值。那如果我們要在 `lambda` 中使用 `>=` 呢？

```
ghci> Just 3 >>= (\x -> Just "!" >>= (\y -> Just (show x ++ y)))
Just "3!"
```

我們嵌一個 `>>=` 在另外一個 `>>=` 中。在外層的 lambda，我們把 `Just "!"` 餵給 `\y -> Just (show x ++ y)`。在內層的 lambda，`y` 被指定成 `"!"`。`x` 仍被指定成 `3`，是因為我們是從外層的 lambda 取值的。這些行為讓我們回想到下列式子：

```
ghci> let x = 3; y = "!" in show x ++ y
"3!"
```

差別在於前述的值是 monadic，具有失敗可能性的 context。我們可以把其中任何一步代換成失敗的狀態：

```
ghci> Nothing >>= (\x -> Just "!" >>= (\y -> Just (show x ++ y)))
Nothing
ghci> Just 3 >>= (\x -> Nothing >>= (\y -> Just (show x ++ y)))
Nothing
ghci> Just 3 >>= (\x -> Just "!" >>= (\y -> Nothing))
Nothing
```

第一行中，把 `Nothing` 餵給一個函數，很自然地會回傳 `Nothing`。第二行裡，我們把 `Just 3` 餵給一個函數，所以 `x` 就成了 `3`。但我們把 `Nothing` 餵給內層的 lambda 所有的結果就成了 `Nothing`，這也進一步使得外層的 lambda 成了 `Nothing`。這就好比我們在 `let expression` 中來把值指定給變數一般。只差在我們這邊的值是 monadic value。

要再說得更清楚點，我們來把 script 改寫成每行都處理一個 `Maybe`：

```
foo :: Maybe String
foo = Just 3 >>= (\x ->
    Just "!" >>= (\y ->
        Just (show x ++ y)))
```

為了擺脫這些煩人的 lambda，Haskell 允許我們使用 `do` 表示法。他讓我們可以把先前的程式寫成這樣：

```
foo :: Maybe String
foo = do
  x <- Just 3
  y <- Just "!"
  Just (show x ++ y)
```



這看起來好像讓我們不用在每一步都去檢查 `Maybe` 的值究竟是 `Just` 或 `Nothing`。這蠻方便的，如果在任何一個步驟我們取出了 `Nothing`。那整個 `do` 的結果就會是 `Nothing`。我們把整個責任都交給 `>=`，他會幫我們處理所有 `context` 的問題。這邊的 `do` 表示法不過是另外一種語法的形式來串連所有的 monadic value 罷了。

在 `do expression` 中，每一行都是一個 monadic value。要檢查處理的結果的話，就要使用 `<-`。如果我們拿到一個 `Maybe String`，並用 `<-` 來綁定給一個變數，那個變數就會是一個 `String`，就像是使用 `>=` 來將 monadic value 帶給 lambda 一樣。至於 `do expression` 中的最後一個值，好比說 `Just (show x ++ y)`，就不能用 `<-` 來綁定結果，因為那樣的寫法當轉換成 `>=` 的結果時並不合理。他必須要是所有 monadic value 黏起來後的總結果，要考慮到前面所有可能失敗的情形。

舉例來說，來看看下面這行：

```
ghci> Just 9 >= (\x -> Just (x > 8))
Just True
```

由於 `>=` 左邊的參數是一個 `Just` 型態的值，當 lambda 被套用至 `9` 就會得到 `Just True`。如果我們重寫整個式子，改用 `do` 表示法：我們會得到：

```
marySue :: Maybe Bool
marySue = do
  x <- Just 9
  Just (x > 8)
```

如果我們比較這兩種寫法，就很容易看出為什麼整個 monadic value 的結果會是在 `do` 表示法中最後一個 monadic value 的值。他串連了全面所有的結果。

我們走鋼索的模擬程式也可以改用 `do` 表示法重寫。`landLeft` 跟 `landRight` 接受一個鳥的數字跟一個竿子來產生一個包在 `Just` 中新的竿子。而在失敗的情況會產生 `Nothing`。我們使用 `>=` 來串連所有的步驟，每一步都倚賴前一步的結果，而且都帶有可能失敗的 `context`。這邊有一個範例，先是有兩隻鳥停在左邊，接著有兩隻鳥停在右邊，然後是一隻鳥停在左邊：

```
routine :: Maybe Pole
routine = do
    start <- return (0,0)
    first <- landLeft 2 start
    second <- landRight 2 first
    landLeft 1 second
```

我們來看看成功的結果：

```
ghci> routine
Just (3,2)
```

當我們要把這些 `routine` 用具體寫出的 `>=`，我們會這樣寫：`return (0,0) >= landLeft 2`，而有了 `do` 表示法，每一行都必須是一個 monadic value。所以我們清楚地把前一個 `Pole` 傳給 `landLeft` 跟 `landRight`。如果我們檢視我們綁定 `Maybe` 的變數，`start` 就是 `(0,0)`，而 `first` 就會是 `(2,0)`。

由於 `do` 表示法是一行一行寫，他們會看起來很像是命令式的寫法。但實際上他們只是代表序列而已，每一步的值都倚賴前一步的結果，並帶著他們的 `context` 繼續下去。

我們再重新來看看如果我們沒有善用 `Maybe` 的 monad 性質的程式：

```
routine :: Maybe Pole
routine =
  case Just (0,0) of
    Nothing -> Nothing
    Just start -> case landLeft 2 start of
      Nothing -> Nothing
      Just first -> case landRight 2 first of
        Nothing -> Nothing
        Just second -> landLeft 1 second
```

在成功的情形下，`Just (0,0)` 變成了 `start`，而 `landLeft 2 start` 的結果成了 `first`。

如果我們想在 `do` 表示法裡面對皮爾斯丟出香蕉皮，我們可以這樣做：

```

routine :: Maybe Pole
routine = do
    start <- return (0,0)
    first <- landLeft 2 start
    Nothing
    second <- landRight 2 first
    landLeft 1 second

```

當我們在 `do` 表示法寫了一行運算，但沒有用到 `<-` 來綁定值的話，其實實際上就是用了 `>>`，他會忽略掉計算的結果。我們只是要讓他們有序，而不是要他們的結果，而且他比寫成 `_ <- Nothing` 要來得漂亮的多。

你會問究竟我們何時要使用 `do` 表示法或是 `>>=`，這完全取決於你的習慣。在這個例子由於有每一步都倚賴於前一步結果的特性，所以我們使用 `>>=`。如果用 `do` 表示法，我們就必須清楚寫出鳥究竟是停在哪根竿子上，但其實每一次都是前一次的結果。不過他還是讓我們了解到怎麼使用 `do`。

在 `do` 表示法中，我們其實可以用模式匹配來綁定 monadic value，就好像我們在 `let` 表達式，跟函數參數中使用模式匹配一樣。這邊來看一個在 `do` 表示法中使用模式匹配的範例：

```

justH :: Maybe Char
justH = do
    (x:xs) <- Just "hello"
    return x

```

我們用模式匹配來取得 `"hello"` 的第一個字元，然後回傳結果。所以 `justH` 計算會得到 `Just 'h'`。

如果模式匹配失敗怎麼辦？當定義一個函數的時候，一個模式不匹配就會跳到下一個模式。如果所有都不匹配，那就會造成錯誤，整個程式就當掉。另一方面，如果在 `let` 中進行模式匹配失敗會直接造成錯誤。畢竟在 `let` 表達式的情況下並沒有失敗就跳下一個的設計。至於在 `do` 表示法中模式匹配失敗的話，那就會呼叫 `fail` 函數。他定義在 `Monad` 的 type class 定義裡。他允許在現在的 monad context 底下，失敗只會造成失敗而不會讓整個程式當掉。他預設的實作如下：

```

fail :: (Monad m) => String -> m a
fail msg = error msg

```

可見預設的實作的確是讓程式掛掉，但在某些考慮到失敗的可能性的 `Monad`（像是 `Maybe`）常常會有他們自己的實作。對於 `Maybe`，他的實作像是這樣：

```

fail _ = Nothing

```

他忽略錯誤訊息，並直接回傳 `Nothing`。所以當在 `do` 表示法中的 `Maybe` 模式匹配失敗的時候，整個結果就會是 `Nothing`。這種方式比起讓程式掛掉要好多了。這邊來看一下 `Maybe` 模式匹配失敗的範例：

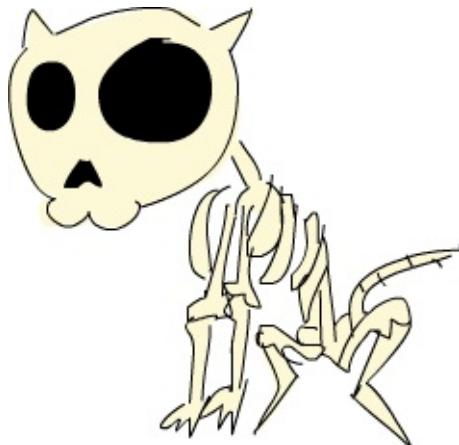
```
wopwop :: Maybe Char
wopwop = do
    (x:xs) <- Just ""
    return x
```

模式匹配的失敗，所以那一行的效果相當於一個 `Nothing`。我們來看看執行結果：

```
ghci> wopwop
Nothing
```

這樣模式匹配的失敗只會限制在我們 monad 的 context 中，而不是整個程式的失敗。這種處理方式要好多了。

## List Monad



我們已經了解了 `Maybe` 可以被看作具有失敗可能性 context 的值，也見識到如何用 `>=` 來把這些具有失敗考量的值傳給函數。在這一個章節中，我們要看一下如何利用 list 的 monadic 的性質來寫 non-deterministic 的程式。

我們已經討論過在把 list 當作 applicatives 的時候他們具有 non-deterministic 的性質。像 `5` 這樣一個值是 deterministic 的。他只有一種結果，而且我們清楚的知道他是什麼結果。另一方面，像 `[3, 8, 9]` 這樣的值包含好幾種結果，所以我們能把他看作是同時具有好幾種結果的值。把 list 當作 applicative functors 展示了這種特性：

```
ghci> (*) <$> [1, 2, 3] <*> [10, 100, 1000]
[10, 100, 1000, 20, 200, 2000, 30, 300, 3000]
```

將左邊 list 中的元素乘上右邊 list 中的元素這樣所有的組合全都被放進結果的 list 中。當處理 non-determinism 的時候，這代表我們有好幾種選擇可以選，我們也會每種選擇都試試看，因此最終的結果也會是一個 non-deterministic 的值。只是包含更多不同可能罷了。

non-determinism 這樣的 context 可以被漂亮地用 monad 來考慮。所以我們這就來看看 list 的 Monad instance 的定義：

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
    fail _ = []
```

`return` 跟 `pure` 是做同樣的事，所以我們應該算已經理解了 `return` 的部份。他接受一個值，並把他放進一個最小的一個 context 中。換種說法，就是他做了一個只包含一個元素的 list。這樣對於我們想要操作普通值的時候很有用，可以直接把他包起來變成 non-deterministic value。

要理解 `>=` 在 list monad 的情形下是怎麼運作的，讓我們先來回歸基本。`>=` 基本上就是接受一個有 context 的值，把他餵進一個只接受普通值的函數，並回傳一個具有 context 的值。如果操作的函數只會回傳普通值而不是具有 context 的值，那 `>=` 在操作一次後就會失效，因為 context 不見了。讓我們來試著把一個 non-deterministic value 塞到一個函數中：

```
ghci> [3,4,5] >= \x -> [x,-x]
[3,-3,4,-4,5,-5]
```

當我們對 `Maybe` 使用 `>=`，是有考慮到可能失敗的 context。在這邊 `>=` 則是有考慮到 non-determinism。`[3,4,5]` 是一個 non-deterministic value，我們把他餵給一個回傳 non-deterministic value 的函數。那結果也會是 non-deterministic。而且他包含了所有從 `[3,4,5]` 取值，套用 `\x -> [x,-x]` 後的結果。這個函數他接受一個數值並產生兩個數值，一個原來的數值與取過負號的數值。當我們用 `>=` 來把一個 list 餵給這個函數，所有在 list 中的數值都保留了原有的跟取負號過的版本。`x` 會針對 list 中的每個元素走過一遍。

要看看結果是如何算出來的，只要看看實作就好了。首先我們從 `[3,4,5]` 開始。然後我們用 lambda 映射過所有元素得到：

```
[[3,-3],[4,-4],[5,-5]]
```

lambda 會掃過每個元素，所以我們有一串包含一堆 list 的 list，最後我們在把這些 list 壓扁，得到一層的 list。這就是我們得到 non-deterministic value 的過程。

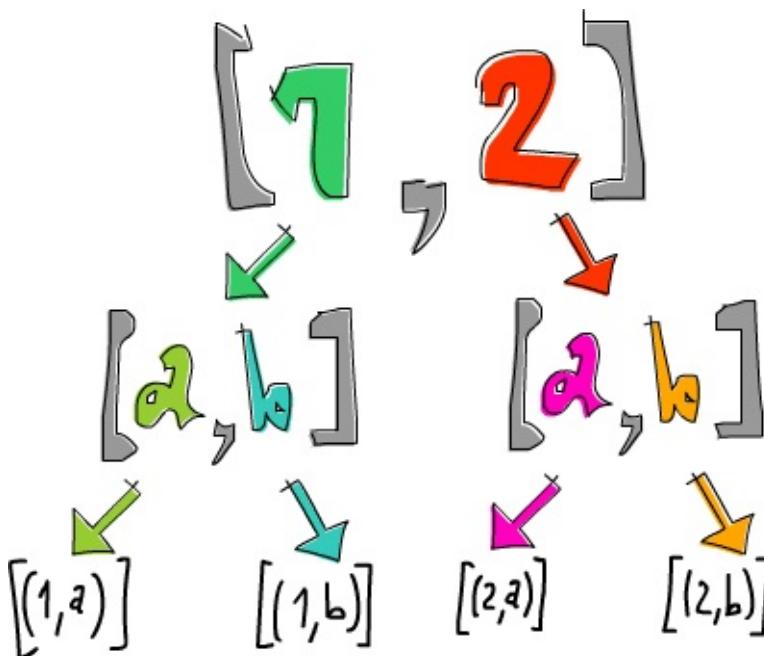
non-determinism 也有考慮到失敗的可能性。`[]` 其實等價於 `Nothing`，因為他什麼結果也沒有。所以失敗等同於回傳一個空的 list。所有的錯誤訊息都不用。讓我們來看看範例：

```
ghci> [] >>= \x -> ["bad", "mad", "rad"]
[]
ghci> [1,2,3] >>= \x -> []
[]
```

第一行裡面，一個空的 list 被丟給 lambda。因為 list 沒有任何元素，所以函數收不到任何東西而產生空的 list。這跟把 `Nothing` 餵給函數一樣。第二行中，每一個元素都被餵給函數，但所有元素都被丟掉，而只回傳一個空的 list。因為所有的元素都造成了失敗，所以整個結果也代表失敗。

就像 `Maybe` 一樣，我們可以用 `>>=` 把他們串起來：

```
ghci> [1,2] >>= \n -> ['a','b'] >>= \ch -> return (n, ch)
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]
```



`[1,2]` 被綁定到 `n` 而 `['a','b']` 被綁定到 `ch`。最後我們用 `return (n, ch)` 來把他放到一個最小的 context 中。在這個案例中，就是把 `(n, ch)` 放到 list 中，這代表最低程度的 non-determinism。整套結構要表達的意思就是對於 `[1,2]` 的每個元素，以及 `['a','b']` 的每個元素，我們產生一個 tuple，每項分別取自不同的 list。

一般來說，由於 `return` 接受一個值並放到最小的 context 中，他不會多做什麼額外的東西僅僅是展示出結果而已。

當你要處理 non-deterministic value 的時候，你可以把 list 中的每個元素想做計算路線的一個 branch。

這邊把先前的表達式用 `do` 重寫：

```
listOfTuples :: [(Int,Char)]
listOfTuples = do
    n <- [1,2]
    ch <- ['a','b']
    return (n, ch)
```

這樣寫可以更清楚看到 `n` 走過 `[1,2]` 中的每一個值，而 `ch` 則取過 `['a','b']` 中的每個值。正如 `Maybe` 一般，我們從 monadic value 中取出普通值然後餵給函數。`>=` 會幫我們處理好一切 context 相關的問題，只差在這邊的 context 指的是 non-determinism。

使用 `do` 來對 list 操作讓我們回想起之前看過的一些東西。來看看下列的片段：

```
ghci> [(n,ch) | n <- [1,2], ch <- ['a','b'] ]
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

沒錯，就是 list comprehension。在先前的範例中，`n` 會走過 `[1,2]` 的每個元素，而 `ch` 會走過 `['a','b']` 的每個元素。同時我們又把 `(n, ch)` 放進一個 context 中。這跟 list comprehension 的目的一樣，只是我們在 list comprehension 裡面不用在最後寫一個 `return` 來得到 `(n, ch)` 的結果。

實際上，list comprehension 不過是一個語法糖。不論是 list comprehension 或是用 `do` 表示法來表示，他都會轉換成用 `>=` 來做計算。

List comprehension 允許我們 filter 我們的結果。舉例來說，我們可以只要包含 `7` 在表示位數裡面的數值。

```
ghci> [ x | x <- [1..50], '7' `elem` show x ]
[7,17,27,37,47]
```

我們用 `show` 跟 `x` 來把數值轉成字串，然後檢查 `'7'` 是否包含在字串裡面。要看看 filtering 要如何轉換成用 list monad 來表達，我們可以考慮使用 `guard` 函數，還有 `MonadPlus` 這個 type class。`MonadPlus` 這個 type class 是用來針對可以同時表現成 monoid 的 monad。下面是他的定義：

```
class Monad m => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a
```

`mzero` 是其實是 `Monoid` 中 `mempty` 的同義詞，而 `mplus` 則對應到 `mappend`。因為 list 同時是 monoid 跟 monad，他們可以是 `MonadPlus` 的 instance。

```
instance MonadPlus [] where
    mzero = []
    mplus = (++)
```

對於 list 而言，`mzero` 代表的是不產生任何結果的 non-deterministic value，也就是失敗的結果。而 `mplus` 則把兩個 non-deterministic value 結合成一個。`guard` 這個函數被定義成下列形式：

```
guard :: (MonadPlus m) => Bool -> m ()
guard True = return ()
guard False = mzero
```

這函數接受一個布林值，如果他是 `True` 就回傳一個包在預設 context 中的 `()`。如果他失敗就產生 `mzero`。

```
ghci> guard (5 > 2) :: Maybe ()
Just ()
ghci> guard (1 > 2) :: Maybe ()
Nothing
ghci> guard (5 > 2) :: [()]
[()]
ghci> guard (1 > 2) :: [()]
[]
```

看起來蠻有趣的，但用起來如何呢？我們可以用他來過濾 non-deterministic 的計算。

```
ghci> [1..50] >>= (\x -> guard ('7' `elem` show x) >> return x)
[7,17,27,37,47]
```

這邊的結果跟我們之前 list comprehension 的結果一致。究竟 `guard` 是如何辦到的？我們先看看 `guard` 跟 `>>` 是如何互動：

```
ghci> guard (5 > 2) >> return "cool" :: [String]
["cool"]
ghci> guard (1 > 2) >> return "cool" :: [String]
[]
```

如果 `guard` 成功的話，結果就會是一個空的 tuple。接著我們用 `>>` 來忽略掉空的 tuple，而呈現不同的結果。另一方面，如果 `guard` 失敗的話，後面的 `return` 也會失敗。這是因為用 `>>=` 把空的 list 餵給函數總是會回傳空的 list。基本上 `guard` 的意思就是：如果一個布林值是 `False` 那就產生一個失敗狀態，不然的話就回傳一個基本的 `()`。這樣計算就可以繼續進行。

這邊我們把先前的範例用 `do` 改寫：

```
sevensOnly :: [Int]
sevensOnly = do
  x <- [1..50]
  guard ('7' `elem` show x)
  return x
```

如果我們不寫最後一行 `return x`，那整個 list 就會是包含一堆空 tuple 的 list。

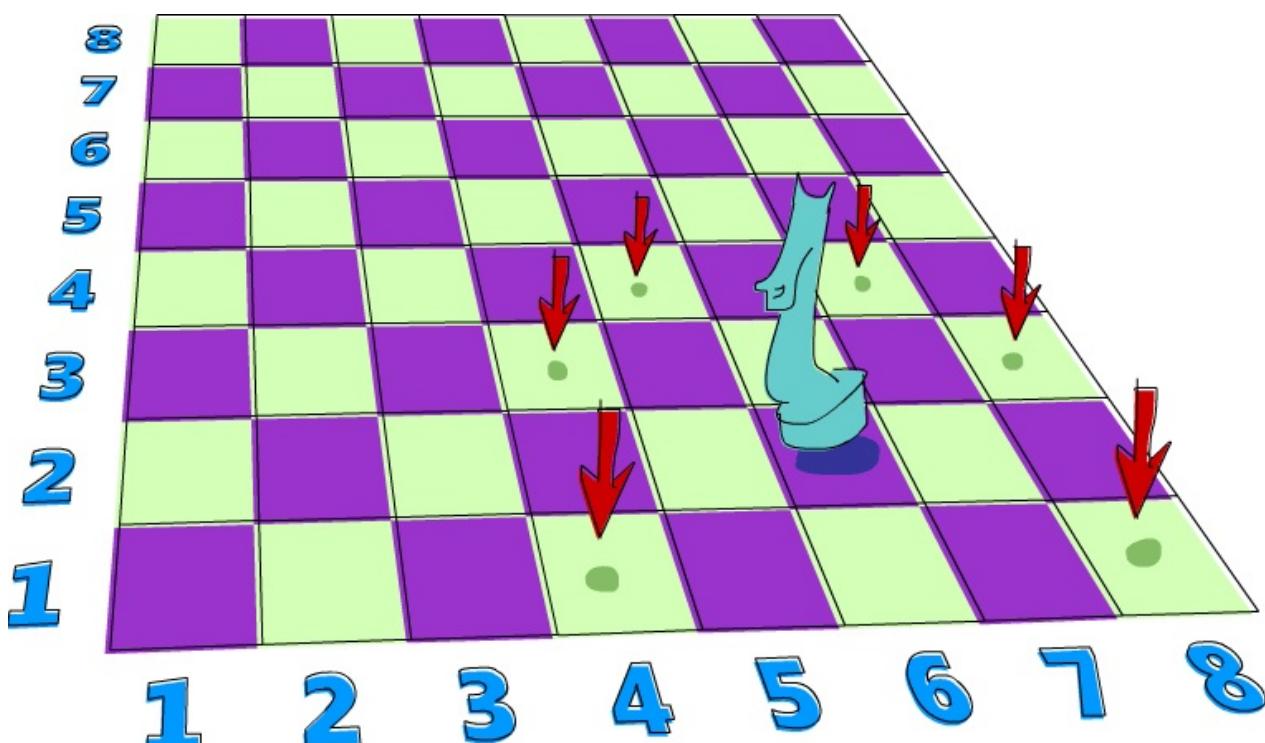
把上述範例寫成 list comprehension 的話就會像這樣：

```
ghci> [ x | x <- [1..50], '7' `elem` show x ]
[7,17,27,37,47]
```

所以 list comprehension 的 filtering 基本上跟 `guard` 是一致的。

## A knight's quest

這邊來看一個可以用 non-determinism 解決的問題。假設你有一個西洋棋盤跟一隻西洋棋中的騎士擺在上面。我們希望知道是否這隻騎士可以在三步之內移到我們想要的位置。我們只要用一對數值來表示騎士在棋盤上的位置。第一個數值代表棋盤的行，而第二個數值代表棋盤的列。



我們先幫騎士的位置定義一個 type synonym。

```
type KnightPos = (Int, Int)
```

假設騎士現在是在 `(6, 2)`。究竟他能不能夠在三步內移動到 `(6, 1)` 呢？你可能會先考慮究竟哪一步是最佳的一步。但不如全部一起考慮吧！要好好利用所謂的 non-determinism。所以我們不是只選擇一步，而是選擇全部。我們先寫一個函數回傳所有可能的下一步：

```
moveKnight :: KnightPos -> [KnightPos]
moveKnight (c, r) = do
  (c', r') <- [(c+2, r-1), (c+2, r+1), (c-2, r-1), (c-2, r+1),
                 , (c+1, r-2), (c+1, r+2), (c-1, r-2), (c-1, r+2)
                 ]
  guard (c' `elem` [1..8] && r' `elem` [1..8])
  return (c', r')
```

騎士有可能水平或垂直移動一步或二步，但問題是他們必須要同時水平跟垂直移動。`(c', r')` 走過 list 中的每一個元素，而 `guard` 會保證產生的結果會停留在棋盤上。如果沒有，那就會產生一個空的 list，表示失敗的結果，`return (c', r')` 也就不會被執行。

這個函數也可以不用 list monad 來寫，但我們這邊只是寫好玩的。下面是一個用 `filter` 實現的版本：

```
moveKnight :: KnightPos -> [KnightPos]
moveKnight (c, r) = filter onBoard
  [(c+2, r-1), (c+2, r+1), (c-2, r-1), (c-2, r+1),
   , (c+1, r-2), (c+1, r+2), (c-1, r-2), (c-1, r+2)
   ]
  where onBoard (c, r) = c `elem` [1..8] && r `elem` [1..8]
```

兩個函數做的都是相同的事，所以選個你喜歡的吧。

```
ghci> moveKnight (6, 2)
[(8, 1), (8, 3), (4, 1), (4, 3), (7, 4), (5, 4)]
ghci> moveKnight (8, 1)
[(6, 2), (7, 3)]
```

我們接受一個位置然後產生所有可能的移動方式。所以我們有一個 non-deterministic 的下一個位置。我們用 `>=` 來餵給 `moveKnight`。接下來我們就可以寫一個三步內可以達到的所有位置：

```
in3 :: KnightPos -> [KnightPos]
in3 start = do
  first <- moveKnight start
  second <- moveKnight first
  moveKnight second
```

如果你傳 `(6, 2)`，得到的 list 會很大，因為會有不同種方式來走到同樣的一個位置。我們也可以不用 `do` 來寫：

```
in3 start = return start >>= moveKnight >>= moveKnight >>= moveKnight
```

第一次 `>>=` 紿我們移動一步的所有結果，第二次 `>>=` 紿我們移動兩步的所有結果，第三次則給我們移動三步的所有結果。

用 `return` 來把一個值放進預設的 context 然後用 `>>=` 餵給一個函數其實跟函數呼叫是同樣的，只是用不同的寫法而已。接著我們寫一個函數接受兩個位置，然後可以測試是否可以在三步內從一個位置移到另一個位置：

```
canReachIn3 :: KnightPos -> KnightPos -> Bool
canReachIn3 start end = end `elem` in3 start
```

我們產生所有三步的可能位置，然後看看其中一個位置是否在裡面。所以我們可以看看是否可以在三步內從 `(6, 2)` 走到 `(6, 1)`：

```
ghci> (6,2) `canReachIn3` (6,1)
True
```

那從 `(6, 2)` 到 `(7, 3)` 呢？

```
ghci> (6,2) `canReachIn3` (7,3)
False
```

答案是不行。你可以修改函數改成當可以走到的時候，他還會告訴你實際的步驟。之後你也可以改成不只限定成三步，可以任意步。

## Monad laws (單子律)



正如 applicative functors 以及 functors, Monad 也有一些要遵守的定律。我們定義一個 `Monad` 的 instance 並不代表他是一個 monad, 只代表他被定義成那個 type class 的 instance。一個型態要是 monad, 則必須遵守單子律。這些定律讓我們可以對這個型態的行為做一些合理的假設。

Haskell 允許任何型態是任何 type class 的 instance。但他不會檢查單子律是否有被遵守，所以如果我們要寫一個 `Monad` 的 instance, 那最好我們確定他有遵守單子律。我們可以不用擔心標準函式庫中的型態是否有遵守單子律。但之後我們定義自己的型態時，我們必須自己檢查是否有遵守單子律。不用擔心，他們不會很複雜。

## Left identity

單子律的第一項說當我們接受一個值，將他用 `return` 放進一個預設的 context 並把他用 `>>=` 餵進一個函數的結果，應該要跟我們直接做函數呼叫的結果一樣。

- `return x >>= f` 應該等於 `f x`

如果你是把 monadic value 視為把一個值放進最小的 context 中，僅僅是把同樣的值放進結果中的話，那這個定律應該很直覺。因為把這個值放進 context 中然後丟給函數，應該要跟直接把這個值丟給函數做呼叫應該沒有差別。

對於 `Maybe` monad, `return` 被定義成 `Just`。`Maybe` monad 講的是失敗的可能性，如果我們有普通值要把他放進 context 中，那把這個動作當作是計算成功應該是很合理的，畢竟我們都知道那個值是很具體的。這邊有些範例：

```
ghci> return 3 >>= (\x -> Just (x+100000))
Just 100003
ghci> (\x -> Just (x+100000)) 3
Just 100003
```

對於 list monad 而言，`return` 是把值放進一個 list 中，變成只有一個元素的 list。`>>=` 則會走過 list 中的每個元素，並把他們丟給函數做運算，但因為在單一元素的 list 中只有一個值，所以跟直接對那元素做運算是等價的：

```
ghci> return "WoM" >>= (\x -> [x,x,x])
["WoM", "WoM", "WoM"]
ghci> (\x -> [x,x,x]) "WoM"
["WoM", "WoM", "WoM"]
```

至於 `IO`，我們已經知道 `return` 並不會造成副作用，只不過是在結果中呈現原有值。所以這個定律對於 `IO` 也是有效的。

## Right identity

單子律的第二個規則是如果我們有一個 monadic value，而且我們把他用 `>>=` 餵給 `return`，那結果就會是原有的 monadic value。

- `m >>= return` 會等於 `m`

這一個可能不像第一定律那麼明顯，但我們還是來看看為什麼會遵守這條。當我們把一個 monadic value 用 `>>=` 餵給函數，那些函數是接受普通值並回傳具有 context 的值。`return` 也是在他們其中。如果你仔細看他的型態，`return` 是把一個普通值放進一個最小 context 中。這就表示，對於 `Maybe` 他並沒有造成任何失敗的狀態，而對於 `list` 他也沒有多加 non-determinism。

```
ghci> Just "move on up" >>= (\x -> return x)
Just "move on up"
ghci> [1,2,3,4] >>= (\x -> return x)
[1,2,3,4]
ghci> putStrLn "Wah!" >>= (\x -> return x)
Wah!
```

如果我們仔細檢視 `list monad` 的範例，會發現 `>>=` 的實作是：

```
xs >>= f = concat (map f xs)
```

所以當我們將 `[1,2,3,4]` 丟給 `return`，第一個 `return` 會把 `[1,2,3,4]` 映射成 `[[1], [2], [3], [4]]`，然後再把這些小 `list` 串接成我們原有的 `list`。

`Left identity` 跟 `right identity` 是描述 `return` 的行為。他重要的原因是因為他把普通值轉換成具有 context 的值，如果他出錯的話會很頭大。

## Associativity

單子律最後一條是說當我們用 `>>` 把一串 monadic function 串在一起，他們的先後順序不應該影響結果：

- `(m >>= f) >>= g` 跟 `m >>= (\x -> f x >>= g)` 是相等的

究竟這邊說的是什麼呢？我們有一個 monadic value `m`，以及兩個 monadic function `f` 跟 `g`。當我們寫下 `(m >>= f) >>= g`，代表的是我們把 `m` 餵給 `f`，他的結果是一個 monadic value。然後我們把這個結果餵給 `g`。而在 `m >>= (\x -> f x >>= g)` 中，我們接受一個 monadic value 然後餵給一個函數，這個函數會把 `f x` 的結果丟給 `g`。我們不太容易直接看出兩者相同，所以先來看個範例比較好理解。

還記得之前皮爾斯的範例嗎？要模擬鳥停在他的平衡竿上，我們把好幾個函數串在一起

```
ghci> return (0,0) >>= landRight 2 >>= landLeft 2 >>= landRight 2
Just (2,4)
```

從 `Just (0,0)` 出發，然後把值傳給 `landRight 2`。他的結果又被綁到下一個 monadic function，以此類推。如果我們用括號清楚標出優先順序的話會是這樣：

```
ghci> ((return (0,0) >>= landRight 2) >>= landLeft 2) >>= landRight 2
Just (2,4)
```

我們也可以改寫成這樣：

```
return (0,0) >>= (\x ->
  landRight 2 x >>= (\y ->
    landLeft 2 y >>= (\z ->
      landRight 2 z)))
```

`return (0,0)` 等價於 `Just (0,0)`，當我們把他餵給 lambda，裡面的 `x` 就等於 `(0,0)`。`landRight` 接受一個數值跟 pole，算出來的結果是 `Just (0,2)` 然後把他餵給另一個 lambda，裡面的 `y` 就變成了 `(0,2)`。這樣的 operation 持續下去，直到最後一隻鳥降落，而得到 `Just (2,4)` 的結果，這也是整個操作的總結果。

這些 monadic function 的優先順序並不重要，重點是他們的意義。從另一個角度來看這個定律：考慮兩個函數 `f` 跟 `g`，將兩個函數組合起來的定義像是這樣：

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = (\x -> f (g x))
```

如果 `g` 的型態是 `a -> b` 且 `f` 的型態是 `b -> c`，我們可以把他們合成一個型態是 `a -> c` 的新函數。所以中間的參數都有自動帶過。現在假設這兩個函數是 monadic function，也就是說如果他們的回傳值是 monadic function？如果我們有一個函數他的型態是 `a -> m b`，我們並不能直接把結果丟給另一個型態為 `b -> m c` 的函數，因為後者只接受型態為 `b` 的普通值。然而，我們可以用 `>>=` 來做到我們想要的事。有了 `>>=`，我們可以合成兩個 monadic function：

```
(<=<) :: (Monad m) => (b -> m c) -> (a -> m b) -> (a -> m c)
f <=< g = (\x -> g x >>= f)
```

所以現在我們可以合成兩個 monadic functions：

```
ghci> let f x = [x,-x]
ghci> let g x = [x^3,x^2]
ghci> let h = f <=< g
ghci> h 3
[9, -9, 6, -6]
```

至於這跟結合律有什麼關係呢？當我們把這定律看作是合成的定律，他就只是說了 `f <=< (g <=< h)` 跟 `(f <=< g) <=< h` 應該等價。只是他是針對 monad 而已。

如果我們把頭兩個單子律用 `<=<` 改寫，那 left identity 不過就是說對於每個 monadic function `f`，`f <=< return` 跟 `f` 是等價，而 right identity 說 `return <=< f` 跟 `f` 是等價。

如果看看普通函數的情形，就會發現很像，`(f . g) . h` 等價於 `f . (g . h)`，`f . id` 跟 `f` 等價，且 `id . f` 等價於 `f`。

在這一章中，我們檢視了 monad 的基本性質，而且也了解了 `Maybe` monad 跟 `list` monad 的運作方式。在下一章，我們會看看其他一些有特色的 monad，我們也會學到如何定義自己的 monad。

## 再來看看更多 Monad



我們已經看過 Monad 是如何接受具有 context 的值，並如何用函數操作他們還有如何用 `>=>` 跟 `do` 來減輕我們對 context 的關注，集中精神在 value 本身。

我們也看過了 `Maybe` 是如何把值加上一個可能會失敗的 context。我們學習到 List Monad 是如何加進多重結果的 context。我們也了解 `IO` Monad 如何運作，而且我們在知道什麼是 Monad 之前就已經知道他了。

在這個章節，我們會介紹一些其他的 Monad。他們可以把值變成 monadic value，因此可以讓我們的程式更簡潔清晰。多見識幾個 Monad 也可以敏銳我們對 Monad 的直覺。

我們即將要介紹的 Monad 都包含在 `mtl` 這個套建中。一個 Haskell package 包含了一堆模組。而 `mtl` 已經包含在 Haskell Platform 中，所以你可能不用另外安裝。要檢查你有沒有這套件，你可以下 `ghc-pkg list`。這會列出你已經安裝的套件，其中應該包含 `mtl` 後面接著對應的版號。

## 你所不知道的 Writer Monad

我們已經看過 `Maybe`, `list` 以及 `IO` Monad。現在我們要來看看 `Writer` Monad。

相對於 `Maybe` 是加入可能失敗的 context, `list` 是加入 non-deterministic 的 context, `writer` 則是加進一個附加值的 context, 好比 `log` 一般。`writer` 可以讓我們在計算的同時蒐集所有 `log` 紀錄, 並匯集成一個 `log` 並附加在結果上。

例如我們想要附加一個 `String` 好說明我們的值在幹麼（有可能是為了除錯）。想像有一個函數接受一個代表幫派人數的數字, 然後會回傳值告訴我們這是否算是一個龐大的幫派：

```
isBigGang :: Int -> Bool
isBigGang x = x > 9
```

現在我們希望他不只是回傳 `True` 或 `False`, 我們還希望他能夠多回傳一個字串代表 `log`。這很容易, 只要多加一個 `String` 在 `Bool` 旁邊就好了。

```
isBigGang :: Int -> (Bool, String)
isBigGang x = (x > 9, "Compared gang size to 9.")
```

我們現在回傳了一個 Tuple, 第一個元素是原來的布林值, 第二個元素是一個 `String`。現在我們的值有了一個 context。

```
ghci> isBigGang 3
(False, "Compared gang size to 9.")
ghci> isBigGang 30
(True, "Compared gang size to 9.")
```



到目前為止都還不錯, `isBigGang` 回傳一個值跟他的 context。對於正常的數值來說這樣的寫法都能運作良好。但如果我們想要把一個已經具有 context 的值, 像是 `(3, "Smallish gang.")`, 餵給 `isBigGang` 呢？我們又面對了同樣的問題：如果我們有一個能接受正常數值並回傳一個具有 context 值的 function, 那我們要如何餵給他一個具有 context 的值？

當我們在研究 `Maybe monad` 的時候，我們寫了一個 `applyMaybe`。他接受一個 `Maybe a` 值跟一個 `a -> Maybe b` 型態的函數，他會把 `Maybe a` 餵給這個 function，即便這個 function 其實是接受 `a` 而非 `Maybe a`。`applyMaybe` 有針對這樣的 context 做處理，也就是會留意有可能發生的失敗情況。但在 `a -> Maybe b` 裡面，我們可以只專心處理正常數值即可。因為 `applyMaybe` (之後變成了 `>>=`) 會幫我們處理需要檢查 `Nothing` 或 `Just` 的情況。

我們再來寫一個接受附加 log 值的函數，也就是 `(a, String)` 型態的值跟 `a -> (b, String)` 型態的函數。我們稱呼這個函數為 `applyLog`。這個函數有的 context 是附加 log 值，而不是一個可能會失敗的 context，因此 `applyLog` 會確保原有的 log 被保留，並附上從函數產生出的新的 log。這邊我們來看一下實作：

```
applyLog :: (a, String) -> (a -> (b, String)) -> (b, String)
applyLog (x, log) f = let (y, newLog) = f x in (y, log ++ newLog)
```

當我們想把一個具有 context 的值餵給一個函數的時候，我們會嘗試把值跟他的 context 分開，然後把值餵給函數再重新接回 context。在 `Maybe monad` 的情況，我們檢查值是否為 `Just x`，如果是，便將 `x` 餵給函數。而在 log 的情況，我們知道 pair 的其中一個 component 是 log 而另一個是值。所以我們先取出值 `x`，將 `f` apply 到 `x`，便獲取 `(y, newLog)`，其中 `y` 是新的值而 `newLog` 則是新的 log。但如果我們回傳 `newLog`，舊的 log 便不會包含進去，所以我們要回傳的是 `(y, log ++ newLog)`。我們用 `++` 來把新的 log 接到舊的上面。

來看看 `applyLog` 運作的情形：

```
ghci> (3, "Smallish gang.") `applyLog` isBigGang
(False, "Smallish gang.Compared gang size to 9")
ghci> (30, "A freaking platoon.") `applyLog` isBigGang
(True, "A freaking platoon.Compared gang size to 9")
```

跟之前的結果很像，只差在我們多了伴隨產生的 log。再來多看幾個例子：

```
ghci> ("Tobin", "Got outlaw name.") `applyLog` (\x -> (length x, "Applied length."))
(5, "Got outlaw name.Applied length.")
ghci> ("Bathcat", "Got outlaw name.") `applyLog` (\x -> (length x, "Applied length"))
(7, "Got outlaw name.Applied length")
```

可以看到在 lambda 裡面 `x` 只是個正常的字串而不是 tuple，且 `applyLog` 幫我們處理掉附加 log 的動作。

## Monoids 的好處

請確定你了解什麼是 Monoids。

到目前為止 `applyLog` 接受 `(a,[String])` 型態的值，但為什麼 `log` 一定要是 `String` 呢？我們使用 `++` 來附加新的 `log`，難道 `++` 並不能運作在任何形式的 `list`，而一定要限制我們在 `String` 上呢？我們當然可以擺脫 `String`，我們可以如下改變他的型態：

```
applyLog :: (a,[c]) -> (a -> (b,[c])) -> (b,[c])
```

我們用一個 `List` 來代表 `Log`。包含在 `List` 中的元素型態必須跟原有的 `List` 跟回傳的 `List` 型態相同，否則我們沒辦法用 `++` 來把他們接起來。

這能夠運作在 `bytestring` 上嗎？絕對沒問題。只是我們現在的型態只對 `List` 有效。我們必須要另外做一個 `bytestring` 版本的 `applyLog`。但我們注意到 `List` 跟 `bytestring` 都是 `monoids`。因此他們都是 `Monoid type class` 的 `instance`，那代表他們都有實作 `mappend`。對 `List` 以及 `bytestring` 而言，`mappend` 都是拿來串接的。

```
ghci> [1,2,3] `mappend` [4,5,6]
[1,2,3,4,5,6]
ghci> B.pack [99,104,105] `mappend` B.pack [104,117,97,104,117,97]
Chunk "chi" (Chunk "huahua" Empty)
```

修改後我們的 `applyLog` 可以運作在任何 `monoid` 上。我們必須要修改型態宣告來表示這件事，同時也要在實作中把 `++` 改成 `mappend`：

```
applyLog :: (Monoid m) => (a,m) -> (a -> (b,m)) -> (b,m)
applyLog (x,log) f = let (y,newLog) = f x in (y,log `mappend` newLog)
```

由於現在包含的值可以是任何 `monoid`，我們不再需要把 `tuple` 想成包含一個值跟對應的 `log`，我們可以想成他包含一個值跟一個對應的 `monoid`。舉例來說，可以說我們有一個 `tuple` 包含一個產品名稱跟一個符合 `monoid` 特性的產品價格。我們可以定義一個 `Sum` 的 `newtype` 來保證我們在操作產品的時候也會把價錢跟著加起來。

```
import Data.Monoid

type Food = String
type Price = Sum Int

addDrink :: Food -> (Food,Price)
addDrink "beans" = ("milk", Sum 25)
addDrink "jerky" = ("whiskey", Sum 99)
addDrink _ = ("beer", Sum 30)
```

我們用 `string` 來代表食物，用 `newtype` 重新定義 `nInt` 為 `Sum`，來追蹤總共需要花多少錢。可以注意到我們用 `mappend` 來操作 `Sum` 的時候，價錢會被一起加起來。

```
ghci> Sum 3 `mappend` Sum 9
Sum {getSum = 12}
```

`addDrink` 的實作很簡單，如果我們想吃豆子，他會回傳 "milk" 以及伴隨的 `Sum 25`，同樣的如果我們要吃 "jerky"，他就會回傳 "whiskey"，要吃其他東西的話，就會回傳 "beer"。乍看之下這個函數沒什麼特別，但如果用 `applyLog` 的話就會有趣些。

```
ghci> ("beans", Sum 10) `applyLog` addDrink
("milk",Sum {getSum = 35})
ghci> ("jerky", Sum 25) `applyLog` addDrink
("whiskey",Sum {getSum = 124})
ghci> ("dogmeat", Sum 5) `applyLog` addDrink
("beer",Sum {getSum = 35})
```

牛奶價值 25 美分，但如果我們也吃了價值 10 美分的豆子的話，總共需要付 35 美分。這樣很清楚地展示了伴隨的值不一定需要是 log，他可以是任何 monoid。至於兩個值要如何結合，那要看 monoid 中怎麼定義。當我們需要的是 log 的時候，他們是串接，但這個 case 裡面，數字是被加起來。

由於 `addDrink` 回傳一個 `(Food,Price)`，我們可以再把結果重新餵給 `addDrink`，這可以很容易告訴我們總共喝了多少錢：

```
ghci> ("dogmeat", Sum 5) `applyLog` addDrink `applyLog` addDrink
("beer",Sum {getSum = 65})
```

將狗食跟 30 美分的啤酒加在一起會得到 `("beer", Sum 35)`。如果我們用 `applyLog` 將上面的結果再餵給 `addDrink`，我們會得到 `("beer", Sum 65)` 這樣的結果。

## The Writer type

我們認識了一個附加 monoid 的值其實表現出來的是一個 monad，我們來再來看看其他類似的 `Monad` instance。`Control.Monad.Writer` 這模組含有 `Writer w a` 的一個型態，裏面定義了他 `Monad` 的 instance，還有一些操作這些值的函數。

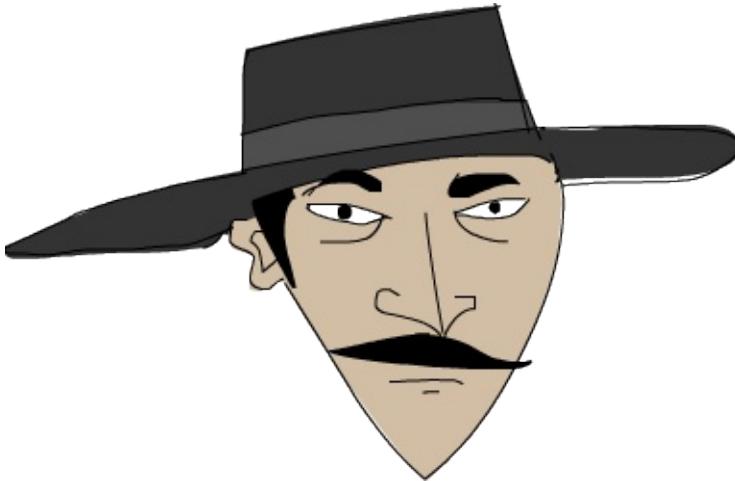
首先，我們來看一下型態。要把一個 monoid 附加給一個值，只需要定義一個 tuple 就好了。`Writer w a` 這型態其實是一個 `newtype` wrapper。他的定義很簡單：

```
newtype Writer w a = Writer { runWriter :: (a, w) }
```

他包在一個 `newtype` 裏面，並且可以是一個 `Monad` 的 instance，而且這樣定義的好處是可以跟單純 tuple 的型態區分開來。`a` 這個型態參數代表是包含的值的型態，而 `w` 則是附加的 monoid 的型態。

他 `Monad` instance 的定義如下：

```
instance (Monoid w) => Monad (Writer w) where
    return x = Writer (x, mempty)
    (Writer (x, v)) >>= f = let (Writer (y, v')) = f x in Writer (y, v `mappend` v')
```



首先，我們來看看 `>>=`。他的實作基本上就是 `applyLog`，只是我們的 tuple 現在是包在一個 `Writer` 的 newtype 中，我們可以用 pattern matching 的方式把他給 unwrap。我們將 `x` 餵給 `f`。這會回給我們 `Writer w a`。接著可以用 `let` expression 來做 pattern matching。把結果綁定到 `y` 這個名字上，然後用 `mappend` 來結合舊的 monoid 值跟新的 monoid 值。最後把結果跟 monoid 值用 `Writer` constructor 包起來，形成我們最後的 `Writer value`。

那 `return` 呢？回想 `return` 的作用是接受一個值，並回傳一個具有意義的最小 context 來裝我們的值。那究竟什麼樣的 context 可以代表我們的 `Writer` 呢？如果我們希望 monoid 值所造成的影響愈小愈好，那 `mempty` 是個合理的選擇。`mempty` 是被當作 identity monoid value，像是 `""` 或 `Sum 0`，或是空的 bytestring。當我們對 `mempty` 用 `mappend` 跟其他 monoid 值結合，結果會是其他的 monoid 值。所以如果我們用 `return` 來做一個 `Writer`，然後用 `>>=` 來餵給其他的函數，那函數回傳的便是算出來的 monoid。下面我們試著用 `return` 搭配不同 context 來回傳 `3`：

```
ghci> runWriter (return 3 :: Writer String Int)
(3,"")
ghci> runWriter (return 3 :: Writer (Sum Int) Int)
(3,Sum {getSum = 0})
ghci> runWriter (return 3 :: Writer (Product Int) Int)
(3,Product {getProduct = 1})
```

因為 `Writer` 並沒有定義成 `Show` 的 instance，我們必須用 `runWriter` 來把我們的 `Writer` 轉成正常的 tuple。對於 `String`，monoid 的值就是空字串。而對於 `Sum` 來說則是 `0`，因為 `0` 加上其他任何值都會是對方。而對 `Product` 來說，則是 `1`。

這裡的 `Writer` instance 並沒有定義 `fail`，所以如果 pattern matching 失敗的話，就會呼叫 `error`。

## Using do notation with Writer

既然我們定義了 `Monad` 的 `instance`，我們自然可以用 `do` 串接 `Writer` 型態的值。這在我們需要對一群 `Writer` 型態的值做處理時顯得特別方便。就如其他的 `monad`，我們可以把他們當作具有 `context` 的值。在現在這個 `case` 中，所有的 `monoid` 的值都會用 `mappend` 來連接起來並得到最後的結果。這邊有一個簡單的範例，我們用 `Writer` 來相乘兩個數。

```
import Control.Monad.Writer

logNumber :: Int -> Writer [String] Int
logNumber x = Writer (x, ["Got number: " ++ show x])

multWithLog :: Writer [String] Int
multWithLog = do
    a <- logNumber 3
    b <- logNumber 5
    return (a*b)
```

`logNumber` 接受一個數並把這個數做成一個 `Writer`。我們再用一串 `string` 來當作我們的 `monoid` 值，每一個數都跟著一個只有一個元素的 `list`，說明我們只有一個數。`multWithLog` 式一個 `Writer`，他將 `3` 跟 `5` 相乘並確保相乘的紀錄有寫進最後的 `log` 中。我們用 `return` 來做成 `a*b` 的結果。我們知道 `return` 會接受某個值並加上某個最小的 `context`，我們可以確定他不會多添加額外的 `log`。如果我們執行程式會得到：

```
ghci> runWriter multWithLog
(15, ["Got number: 3", "Got number: 5"])
```

有時候我們就是想要在某個時間點放進某個 `Monoid value`。`tell` 正是我們需要的函數。他實作了 `MonadWriter` 這個 `type class`，而且在當 `Writer` 用的時候也能接受一個 `monoid value`，好比說 `["This is going on"]`。我們能用他來把我們的 `monoid value` 接到任何一個 `dummy value` `()` 上來形成一個 `Writer`。當我們拿到的結果是 `()` 的時候，我們不會把他綁定到變數上。來看一個 `multWithLog` 的範例：

```
multWithLog :: Writer [String] Int
multWithLog = do
    a <- logNumber 3
    b <- logNumber 5
    tell ["Gonna multiply these two"]
    return (a*b)
```

`return (a*b)` 是我們的最後一行，還記得在一個 `do` 中的最後一行代表整個 `do` 的結果。如果我們把 `tell` 擺到最後，則 `do` 的結果則會是 `()`。我們會因此丟掉乘法運算的結果。除此之外，`log` 的結果是不變的。

```
ghci> runWriter multWithLog
(15,["Got number: 3","Got number: 5","Gonna multiply these two"])
```

## Adding logging to programs

歐幾里得算法是找出兩個數的最大公因數。Haskell 已經提供了 `gcd` 的函數，但我們來實作一個具有 `log` 功能的 `gcd`：

```
gcd' :: Int -> Int -> Int
gcd' a b
| b == 0      = a
| otherwise   = gcd' b (a `mod` b)
```

演算法的內容很簡單。首先他檢查第二個數字是否為零。如果是零，那就回傳第一個數字。如果不是，那結果就是第二個數字跟將第一個數字除以第二個數字的餘數兩個數字的最大公因數。舉例來說，如果我們想知道 8 跟 3 的最大公因數，首先可以注意到 3 不是 0。所以我們要求的是 3 跟 2 的最大公因數(8 除以 3 餘二)。接下去我可以看到 2 不是 0，所以我們要再找 2 跟 1 的最大公因數。同樣的，第二個數不是 0，所以我們再找 1 跟 0 的最大公因數。最後第二個數終於是 0 了，所以我們得到最大公因數是 1。

```
ghci> gcd' 8 3
1
```

答案真的是這樣。接著我們想加進 context，context 會是一個 monoid value 並且像是一個 log 一樣。就像之前的範例，我們用一串 string 來當作我們的 monoid。所以 `gcd'` 會長成這樣：

```
gcd' :: Int -> Int -> Writer [String] Int
```

而他的程式碼會像這樣：

```

import Control.Monad.Writer

gcd' :: Int -> Int -> Writer [String] Int
gcd' a b
| b == 0 = do
    tell ["Finished with " ++ show a]
    return a
| otherwise = do
    tell [show a ++ " mod " ++ show b ++ " = " ++ show (a `mod` b)]
    gcd' b (a `mod` b)

```

這個函數接受兩個 `Int` 並回傳一個 `Writer [String] Int`，也就是說是一個有 `log context` 的 `Int`。當 `b` 等於 `0` 的時候，我們用一個 `do` 來組成一個 `Writer` 的值。我們先用 `tell` 來寫入我們的 `log`，然後用 `return` 來當作 `do` 的結果。當然我們也可以這樣寫：

```
Writer (a, ["Finished with " ++ show a])
```

但我想 `do` 的表達方式是比較容易閱讀的。接下來我們看看當 `b` 不等於 `0` 的時候。我們會把 `mod` 的使用情況寫進 `log`。然後在 `do` 當中的第二行遞迴呼叫 `gcd'`。`gcd'` 現在是回傳一個 `Writer` 的型態，所以 `gcd' b (a `mod` b)` 這樣的寫法是完全沒問題的。

儘管去 `trace` 這個 `gcd'` 對於理解十分有幫助，但我想了解整個大概念，把值視為具有 `context` 是更加有用的。

接著來試試跑我們的 `gcd'`，他的結果會是 `Writer [String] Int`，如果我們把他從 `newtype` 中取出來，我們會拿到一個 `tuple`。`tuple` 的第一個部份就是我們要的結果：

```
ghci> fst $ runWriter (gcd' 8 3)
1
```

至於 `log` 呢，由於 `log` 是一連串 `string`，我們就用 `mapM_ putStrLn` 來把這些 `string` 印出來：

```
ghci> mapM_ putStrLn $ snd $ runWriter (gcd' 8 3)
8 mod 3 = 2
3 mod 2 = 1
2 mod 1 = 0
Finished with 1
```

把普通的演算法轉換成具有 `log` 是很棒的經驗，我們不過是把普通的 `value` 重寫成 `Monadic value`，剩下的就靠 `>=` 跟 `Writer` 來幫我們處理一切。用這樣的方法我們幾乎可以對任何函數加上 `logging` 的功能。我們只要把普通的值換成 `Writer`，然後把一般的函數呼叫換成 `>=`（當然也可以用 `do`）

## Inefficient list construction

當製作 `Writer` Monad 的時候，要特別注意你是使用哪種 monoid。使用 list 的話效能有時候是沒辦法接受的。因為 list 是使用 `++` 來作為 `mappend` 的實現。而 `++` 在 list 很長的時候是非常慢的。

在之前的 `gcd'` 中，`log` 並不會慢是因為 list append 的動作實際上看起來是這樣：

```
a ++ (b ++ (c ++ (d ++ (e ++ f))))
```

list 是建立的方向是從左到右，當我們先建立左邊的部份，而把另一串 list 加到右邊的時候效能會不錯。但如果我們不小心使用，而讓 `Writer` monad 實際在操作 list 的時候變成像這樣的話。

```
((((a ++ b) ++ c) ++ d) ++ e) ++ f
```

這會讓我們的操作是 left associative，而不是 right associative。這非常沒有效率，因為每次都是把右邊的部份加到左邊的部份，而左邊的部份又必須要從頭開始建起。

下面這個函數跟 `gcd'` 差不多，只是 `log` 的順序是相反的。他先紀錄剩下的操作，然後紀錄現在的步驟。

```
import Control.Monad.Writer

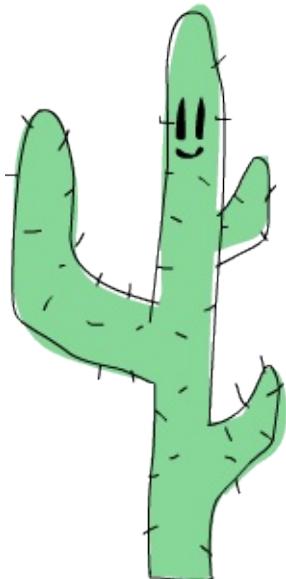
gcdReverse :: Int -> Int -> Writer [String] Int
gcdReverse a b
| b == 0 = do
    tell ["Finished with " ++ show a]
    return a
| otherwise = do
    result <- gcdReverse b (a `mod` b)
    tell [show a ++ " mod " ++ show b ++ " = " ++ show (a `mod` b)]
    return result
```

他先遞迴呼叫，然後把結果綁定到 `result`。然後把目前的動作寫到 `log`，在遞迴的結果之後。最後呈現的就是完整的 `log`。

```
ghci> mapM_ putStrLn $ snd $ runWriter (gcdReverse 8 3)
Finished with 1
2 mod 1 = 0
3 mod 2 = 1
8 mod 3 = 2
```

這沒效率是因為他讓 `++` 成為 left associative 而不是 right associative。

## Difference lists



由於 list 在重複 append 的時候顯得低效，我們最好能使用一種支援高效 appending 的資料結構。其中一種就是 difference list。difference list 很類似 list，只是他是一個函數。他接受一個 list 並 prepend 另一串 list 到他前面。一個等價於 `[1, 2, 3]` 的 difference list 是這樣一個函數 `\xs -> [1, 2, 3] ++ xs`。一個等價於 `[]` 的 difference list 則是 `\xs -> [] ++ xs`。

Difference list 最酷的地方在於他支援高效的 appending。當我們用 `++` 來實現 appending 的時候，他必須要走到左邊的 list 的尾端，然後把右邊的 list 一個個從這邊接上。那 difference list 是怎麼作的呢？appending 兩個 difference list 就像這樣

```
f `append` g = \xs -> f (g xs)
```

`f` 跟 `g` 這邊是兩個函數，他們都接受一個 list 並 prepend 另一串 list。舉例來說，如果 `f` 代表 `("dog"++)`（可以寫成 `\xs -> "dog" ++ xs`）而 `g` 是 `("meat"++)`，那 `f `append` g` 就會做成一個新的函數，等價於：

```
\xs -> "dog" ++ ("meat" ++ xs)
```

append 兩個 difference list 實際上就是用一個函數，這函數先餵一個 list 給第一個 difference list，然後再把結果餵給第二個 difference list。

我們可以用一個 `newtype` 來包起來

```
newtype DiffList a = DiffList { getDiffList :: [a] -> [a] }
```

我們包起來的型態是 `[a] -> [a]`，因為 difference list 不過就是一個轉換一個 list 到另一個 list 的函數。要把普通 list 轉換成 difference list 也很容易。

```

toDiffList :: [a] -> DiffList a
toDiffList xs = DiffList (xs++)

fromDiffList :: DiffList a -> [a]
fromDiffList (DiffList f) = f []

```

要把一個普通 list 轉成 difference list 不過就是照之前定義的，作一個 prepend 另一個 list 的函數。由於 difference list 只是一個 prepend 另一串 list 的一個函數，假如我們要轉回來的話，只要餵給他空的 list 就行了。

這邊我們給一個 difference list 的 `Monoid` 定義

```

instance Monoid (DiffList a) where
    mempty = DiffList (\xs -> [] ++ xs)
    (DiffList f) `mappend` (DiffList g) = DiffList (\xs -> f (g xs))

```

我們可以看到 `mempty` 不過就是 `id`，而 `mappend` 其實是 function composition。

```

ghci> fromDiffList (toDiffList [1,2,3,4] `mappend` toDiffList [1,2,3])
[1,2,3,4,1,2,3]

```

現在我們可以用 difference list 來加速我們的 `gcdReverse`

```

import Control.Monad.Writer

gcd' :: Int -> Int -> Writer (DiffList String) Int
gcd' a b
| b == 0 = do
    tell (toDiffList ["Finished with " ++ show a])
    return a
| otherwise = do
    result <- gcd' b (a `mod` b)
    tell (toDiffList [show a ++ " mod " ++ show b ++ " = " ++ show (a `mod` b)])
    return result

```

我們只要把 monoid 的型態從 `[String]` 改成 `DiffList String`，並在使用 `tell` 的時候把普通的 list 用 `toDiffList` 轉成 difference list 就可以了。

```

ghci> mapM_ putStrLn . fromDiffList . snd . runWriter $ gcdReverse 110 34
Finished with 2
8 mod 2 = 0
34 mod 8 = 2
110 mod 34 = 8

```

我們用 `runWriter` 來取出 `gcdReverse 110 34` 的結果，然後用 `snd` 取出 `log`，並用 `fromDiffList` 轉回普通的 list 印出來。

## Comparing Performance

要體會 Difference List 能如何增進效率，考慮一個從某數數到零的 case。我們紀錄的時候就像 `gcdReverse` 一樣是反過來記的，所以在 `log` 中實際上是從零數到某個數。

```
finalCountDown :: Int -> Writer (DiffList String) ()
finalCountDown 0 = do
    tell (toDiffList ["0"])
finalCountDown x = do
    finalCountDown (x-1)
    tell (toDiffList [show x])
```

如果我們餵 `0`，他就只 `log 0`。如果餵其他正整數，他會先倒數到 `0` 然後 `append` 那些數到 `log` 中，所以如果我們呼叫 `finalCountDown` 並餵給他 `100`，那 `log` 的最後一筆就會是 `"100"`。

如果你把這個函數 `load` 進 `GHCi` 中並餵給他一個比較大的整數 `500000`，你會看到他無停滯地從 `0` 開始數起：

```
ghci> mapM_ putStrLn . fromDiffList . snd . runWriter $ finalCountDown 500000
0
1
2
```

但如果我們用普通的 list 而不用 difference list

```
finalCountDown :: Int -> Writer [String] ()
finalCountDown 0 = do
    tell ["0"]
finalCountDown x = do
    finalCountDown (x-1)
    tell [show x]
```

並下同樣的指令

```
ghci> mapM_ putStrLn . snd . runWriter $ finalCountDown 500000
```

我們會看到整個運算卡卡的。

當然這不是一個嚴謹的測試方法，但足以表顯出 difference list 是比較有效率的寫法。

# Reader Monad



在講 Applicative 的章節中，我們說過了 `(->) r` 的型態只是 `Functor` 的一個 instance。要將一個函數 `f` map over 一個函數 `g`，基本上等價於一個函數，他可以接受原本 `g` 接受的參數，先套用 `g` 然後再把其結果丟給 `f`。

```
ghci> let f = (*5)
ghci> let g = (+3)
ghci> (fmap f g) 8
```

我們已經見識過函數當作 applicative functors 的例子。這樣能讓我們對函數的結果直接進行操作。

```
ghci> let f = (+) <$> (*2) <*> (+10)
ghci> f 3
19
```

`(+) <$> (*2) <*> (+10)` 代表一個函數，他接受一個數值，分別把這數值交給 `(*2)` 跟 `(+10)`。然後把結果加起來。例如說，如果我們餵 `3` 紿這個函數，他會分別對 `3` 做 `(*2)` 跟 `(+10)` 的動作。而得到 `6` 跟 `13`。然後呼叫 `(+)`，而得到 `19`。

其實 `(->) r` 不只是一個 functor 跟一個 applicative functor，他也是一個 monad。就如其他 monadic value 一般，一個函數也可以被想做是包含一個 context 的。這個 context 是說我們期待某個值，他還沒出現，但我們知道我們會把他當作函數的參數，呼叫函數來得到結果。

我們已經見識到函數是怎樣可以看作 functor 或是 applicative functors 了。再來讓我們看看當作 `Monad` 的一個 instance 時會是什麼樣子。你可以在 `Control.Monad.Instances` 裡面找到，他看起來像這樣：

```
instance Monad ((->) r) where
    return x = \_ -> x
    h >>= f = \w -> f (h w) w
```

我們之前已經看過函數的 `pure` 實作了，而 `return` 差不多就是 `pure`。他接受一個值並把他放進一個 minimal context 裡面。而要讓一個函數能夠是某個定值的唯一方法就是讓他完全忽略他的參數。

而 `>=` 的實作看起來有點難以理解，我們可以仔細來看看。當我們使用 `>=` 的時候，餵進去的是一個 monadic value，處理他的是一個函數，而吐出來的也是一個 monadic value。在這個情況下，當我們將一個函數餵進一個函數，吐出來的也是一個函數。這就是為什麼我們在最外層使用了一個 lambda。在我們目前看過的實作中，`>=` 幾乎都是用 lambda 將內部跟外部隔開來，然後在內部來使用 `f`。這邊也是一樣的道理。要從一個函數得到一個結果，我們必須餵給他一些東西，這也是為什麼我們先用 `(h w)` 取得結果，然後將他丟給 `f`。而 `f` 回傳一個 monadic value，在這邊這個 monadic value 也就是一個函數。我們再把 `w` 餵給他。

如果你還不太懂 `>=` 怎麼寫出來的，不要擔心，因為接下來的範例會讓你曉得這真的是一個簡單的 Monad。我們造一個 `do expression` 來使用這個 Monad。

```
import Control.Monad.Instances

addStuff :: Int -> Int
addStuff = do
  a <- (*2)
  b <- (+10)
  return (a+b)
```

這跟我們之前寫的 applicative expression 差不多，只差在他是運作在 monad 上。一個 `do expression` 的結果永遠會是一個 monadic value，這個也不例外。而這個 monadic value 其實是一個函數。只是在這邊他接受一個數字，然後套用 `(*2)`，把結果綁定到 `a` 上面。而 `(+10)` 也同用被套用到同樣的參數。結果被綁定到 `b` 上。`return` 就如其他 monad 一樣，只是製作一個簡單的 monadic value 而不會作多餘的事情。這讓整個函數的結果是 `a+b`。如果我們試著跑跑看，會得到之前的結果。

```
ghci> addStuff 3
19
```

其中 `3` 會被餵給 `(*2)` 跟 `(+10)`。而且他也會被餵給 `return (a+b)`，只是他會忽略掉 `3` 而永遠回傳 `a+b` 正因為如此，function monad 也被稱作 reader monad。所有函數都從一個固定的地方讀取。要寫得更清楚一些，可以把 `addStuff` 改寫如下：

```
addStuff :: Int -> Int
addStuff x = let
  a = (*2) x
  b = (+10) x
  in a+b
```

我們見識了把函數視作具有 context 的值很自然的可以表達成 reader monad。只要我們當作我們知道函數會回傳什麼值就好。他作的就是把所有的函數都黏在一起做成一個大的函數，然後把這個函數的參數都餵給全部組成的函數，這有點取出他們未來的值的意味。實作做完

了然後 `>>=` 就會保證一切都能正常運作。

## State Monad



Haskell 是一個純粹的語言，正因為如此，我們的程式是有一堆沒辦法改變全域狀態或變數的函數所組成，他們只會作些處理並回傳結果。這樣的性質讓我們很容易思考我們的程式在幹嘛，因為我們不需要擔心變數在某一個時間點的值是什麼。然而，有一些領域的問題根本上就是依賴於隨著時間而改變的狀態。雖然我們也可以用 Haskell 寫出這樣的程式，但有時候寫起來蠻痛苦的。這也是為什麼 Haskell 要加進 State Monad 這個特性。這讓我們在 Haskell 中可以容易地處理狀態性的問題，並讓其他部份的程式還是保持純粹性。

當我們處理亂數的時候，我們的函數接受一個 random generator 並回傳一個新的亂數跟一個新的 random generator。如果我們需要很多個亂數，我們可以用前一個函數回傳的 random generator 繼續做下去。當我們要寫一個接受 `StdGen` 的函數並產生丟三個硬幣結果的函數，我們會這樣寫：

```
threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen =
  let (firstCoin, newGen) = random gen
      (secondCoin, newGen') = random newGen
      (thirdCoin, newGen'') = random newGen'
  in (firstCoin, secondCoin, thirdCoin)
```

他接受一個 `gen` 然後用 `random gen` 產生一個 `Bool` 型態的值以及新的 generator。要模擬丟第二個硬幣的話，便使用新的 generator。在其他語言中，多半除了亂數之外不需要多回傳一個 generator。那是因為我們可以對現有的進行修改。但 Haskell 是純粹的語言，我們沒辦法那麼做，所以我們必須要接受一個狀態，產生結果然後回傳一個新的狀態，然後用新的狀態來繼續做下去。

一般來講你應該不會喜歡這麼寫，在程式中有赤裸裸的狀態，但我們又不想放棄 Haskell 的純粹性質。這就是 State Monad 的好處了，他可以幫我們處理這些瑣碎的事情，又讓我們保持 Haskell 的純粹性。

為了深入理解狀態性的計算，我們先來看看應該給他們什麼樣的型態。我們會說一個狀態性的計算是一個函數，他接受一個狀態，回傳一個值跟一個新的狀態。寫起來會像這樣：

```
s -> (a, s)
```

`s` 是狀態的型態，而 `a` 是計算結果的型態。

在其他的語言中，賦值大多是被當作會改變狀態的操作。舉例來說，當我們在命令式語言寫 ```x = 5```，這通常代表的是

如果你用函數語言的角度去思考，你可以把他想做是一個函數，接受一個狀態，並回傳結果跟新的狀態。那新的狀態代表著

這種改變狀態的計算，除了想做是一個接受狀態並回傳結果跟新狀態的函數外，也可以想做是具有 `context` 的值。實際的值是結果。然而要得到結果，我們必須要給一個初始的狀態，才能得到結果跟最後的狀態。

## Stack and Stones

考慮現在我們要對一個堆疊的操作建立模型。你可以把東西推上堆疊頂端，或是把東西從頂端拿下來。如果你要的元素是在堆疊的底層的話，你必須要把他上面的東西都拿下來才能拿到他。

我們用一個 `list` 來代表我們的堆疊。而我們把 `list` 的頭當作堆疊的頂端。為了正確的建立模型，我們要寫兩個函數：`pop` 跟 `push`。`pop` 會接受一個堆疊，取下一個元素並回傳一個新的堆疊，這個新的堆疊不包含取下的元素。`push` 會接受一個元素，把他堆到堆疊中，並回傳一個新的堆疊，其包含這個新的元素。

```
type Stack = [Int]

pop :: Stack -> (Int, Stack)
pop (x:xs) = (x, xs)

push :: Int -> Stack -> (((), Stack)
push a xs = (((), a:xs))
```

我們用 `()` 來當作 `pushing` 的結果，畢竟推上堆疊並不需要什麼回傳值，他的重點是在改變堆疊。注意到 `push` 跟 `pop` 都是改變狀態的計算，可以從他們的型態看出來。

我們來寫一段程式來模擬一個堆疊的操作。我們接受一個堆疊，把 `3` 推上去，然後取出兩個元素。

```
stackManip :: Stack -> (Int, Stack)
stackManip stack = let
    (( ), newStack1) = push 3 stack
    (a , newStack2) = pop newStack1
    in pop newStack2
```

我們拿一個 `stack` 來作 `push 3 stack` 的動作，其結果是一個 tuple。tuple 的第一個部份是 `()`，而第二個部份是新的堆疊，我們把他命名成 `newStack1`。然後我們從 `newStack1` 上 `pop` 出一個數字。其結果是我們之前 `push` 上去的一個數字 `a`，然後把這個更新的堆疊叫做 `newStack2`。然後我們從 `newStack2` 上再 `pop` 出一個數字 `b`，並得到 `newStack3`。我們回傳一個 tuple 跟最終的堆疊。

```
ghci> stackManip [5,8,2,1]
(5,[8,2,1])
```

結果就是 `5` 跟新的堆疊 `[8,2,1]`。注意到 `stackManip` 是一個會改變狀態的操作。我們把一堆會改變狀態的操作綁在一起操作，有沒有覺得很耳熟的感覺。

`stackManip` 的程式有點冗長，因為我們要寫得太詳細，必須把狀態給每個操作，然後把新的狀態再餵給下一個。如果我們可以不要這樣作的話，那程式應該會長得像這樣：

```
stackManip = do
    push 3
    a <- pop
    pop
```

這就是 State Monad 在做的事。有了他，我們便可以免除於要把狀態操作寫得太明白的窘境。

## The State Monad

`Control.Monad.State` 這個模組提供了一個 `newtype` 包起來的型態。

```
newtype State s a = State { runState :: s -> (a,s) }
```

一個 `State s a` 代表的是一個改變狀態的操作，他操縱的狀態為型態 `s`，而產生的結果是 `a`。

我們已經見識過什麼是改變狀態的操作，以及他們是可以被看成具有 `context` 的值。接著來看看他們 `Monad` 的 `instance`：

```

instance Monad (State s) where
    return x = State $ \s -> (x,s)
    (State h) >>= f = State $ \s -> let (a, newState) = h s
                                         (State g) = f a
                                         in g newState
  
```

我們先來看看 `return` 那一行。我們 `return` 要作的事是接受一個值，並做出一個改變狀態的操作，讓他永遠回傳那個值。所以我們才做了一個 `lambda` 函數，`\s -> (x,s)`。我們把 `x` 當成是結果，並且狀態仍然是 `s`。這就是 `return` 要完成的 minimal context。



那 `>>=` 的實作呢？很明顯的把改變狀態的操作餵進 `>>=` 也必須要丟出另一個改變狀態的操作。所以我們用 `State` 這個 newtype wrapper 來把一個 `lambda` 函數包住。這個 `lambda` 會是新的一個改變狀態的操作。但裡面的內容是什麼？首先我們應該要從接受的操作取出結果。由於 `lambda` 是在一個大的操作中，所以我們可以餵給 `h` 我們現在的狀態，也就是 `s`。那會產生 `(a, newState)`。到目前為止每次我們在實作 `>>=` 的時候，我們都會先從 `monadic value` 中取出結果，然後餵給 `f` 來得到新的 `monadic value`。在寫 `writer` 的時候，我們除了這樣作還要確保 `context` 是用 `mappend` 把舊的 `monoid value` 跟新的接起來。在這邊我們則是用 `f a` 得到一個新的操作 `g`。現在我們有了新的操作跟新的狀態（叫做 `newState`），我們就把 `newState` 餵給 `g`。結果便是一個 `tuple`，裡面包含了最後的結果跟最終的狀態。

有了 `>>=`，我們便可以把兩個操作黏在一起，只是第二個被放在一個函數中，專門接受第一個的結果。由於 `pop` 跟 `push` 已經是改變狀態的操作了，我們可以他們包在 `State` 中

```

import Control.Monad.State

pop :: State Stack Int
pop = State $ \(x:xs) -> (x,xs)

push :: Int -> State Stack ()
push a = State $ \xs -> (((),a:xs))
  
```

`pop` 已經滿足我們的條件，而 `push` 要先接受一個 `Int` 才會回傳我們要的操作。所以我們可以改寫先前的範例如下：

```
import Control.Monad.State

stackManip :: State Stack Int
stackManip = do
    push 3
    a <- pop
    pop
```

看到我們是怎麼把一個 `push` 跟兩個 `pop` 黏成一個操作嗎？當我們將他們從一個 `newtype` 取出，其實就是需要一個能餵進初始狀態的函數：

```
ghci> runState stackManip [5,8,2,1]
(5,[8,2,1])
```

我們不須綁定第二個 `pop`，因為我們根本不會用到 `a`，所以可以寫成下面的樣子：

```
stackManip :: State Stack Int
stackManip = do
    push 3
    pop
    pop
```

再來嘗試另外一種方式，先從堆疊上取下一個數字，看看他是不是 `5`，如果是的話就把他放回堆疊上，如果不是的話就堆上 `3` 跟 `8`。

```
stackStuff :: State Stack ()
stackStuff = do
    a <- pop
    if a == 5
        then push 5
    else do
        push 3
        push 8
```

很直覺吧！我們來看看初始的堆疊的樣子。

```
ghci> runState stackStuff [9,0,2,1,0]
((],[8,3,0,2,1,0])
```

還記得我們說過 `do` 的結果會是一個 monadic value，而在 `state` monad 的 case，`do` 也就是一個改變狀態的函數。而由於 `stackManip` 跟 `stackStuff` 都是改變狀態的計算，因此我們可以把他們黏在一起：

```
moreStack :: State Stack ()
moreStack = do
    a <- stackManip
    if a == 100
        then stackStuff
        else return ()
```

如果 `stackManip` 的結果是 `100`，我們就會跑 `stackStuff`，如果不是的話就什麼都不做。`return ()` 不過就是什麼是都不做，全部保持原樣。

`Control.Monad.State` 提供了一個 `MonadState` 的 typeclass，他有兩個有用的函數，分別是 `get` 跟 `put`。對於 `State` 來說，`get` 的實作就像這樣：

```
get = State $ \s -> (s, s)
```

他只是取出現在的狀態除此之外什麼也不做。而 `put` 函數會接受一個狀態並取代掉現有的狀態。

```
put newState = State $ \s -> (((), newState))
```

有了這兩個狀態，我們便可以看到現在堆疊中有什麼，或是把整個堆疊中的元素換掉。

```
stackyStack :: State Stack ()
stackyStack = do
    stackNow <- get
    if stackNow == [1, 2, 3]
        then put [8, 3, 1]
        else put [9, 2, 1]
```

我們可以看看對於 `State` 而言，`>>=` 的型態會是什麼：

```
(>>=) :: State s a -> (a -> State s b) -> State s b
```

我們可以看到狀態的型態都是 `s`，而結果從型態 `a` 變成型態 `b`。這代表我們可以把好幾個改變狀態的計算黏在一起，這些計算的結果可以都不一樣，但狀態的型態會是一樣的。舉例來說，對於 `Maybe` 而言，`>>=` 的型態會是：

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

`Maybe` 不變是有道理的，但如果用 `>>` 來把兩種不同的 monad 接起來是沒道理的。但對於 `state monad` 而言，monad 實際是 `state s`，所以如果 `s` 不一樣，我們就要用 `>>=` 來把兩個 monad 接起來。

## 隨機性與 state monad

在章節的一開始，我們知道了在 Haskell 中要產生亂數的不方便。我們要拿一個產生器，並回傳一個亂數跟一個新的產生器。接下來我們還一定要用新的產生器不可。但 State Monad 讓我們可以方便一些。

`System.Random` 中的 `random` 函數有下列的型態：

```
random :: (RandomGen g, Random a) => g -> (a, g)
```

代表他接受一個亂數產生器，並產生一個亂數跟一個新的產生器。很明顯他是一個會改變狀態的計算，所以我們可以用 `newtype` 把他包在一個 `State` 中，然後把他當作 monadic value 來操作。

```
import System.Random
import Control.Monad.State

randomSt :: (RandomGen g, Random a) => State g a
randomSt = State random
```

這樣我們要丟三個硬幣的結果可以改寫成這樣：

```
import System.Random
import Control.Monad.State

threeCoins :: State StdGen (Bool,Bool,Bool)
threeCoins = do
    a <- randomSt
    b <- randomSt
    c <- randomSt
    return (a,b,c)
```

`threeCoins` 是一個改變狀態的計算，他接受一個初始的亂數產生器，他會把他餵給 `randomSt`，他會產生一個數字跟一個新的產生器，然後會一直傳遞下去。我們用 `return (a,b,c)` 來呈現 `(a,b,c)`，這樣並不會改變最近一個產生器的狀態。

```
ghci> runState threeCoins (mkStdGen 33)
((True,False,True), 680029187 2103410263)
```

要完成像這樣要改變狀態的任務便因此變得輕鬆了很多。

## Error Monad

我們知道 `Maybe` 是拿來賦予一個值具有可能失敗的 context。一個值可能會是 `Just something` 或是一個 `Nothing`。儘管這很有用，但當我們拿到了一個 `Nothing`，我們只知道他失敗了，但我們沒辦法塞進一些有用的資訊，告訴我們究竟是在什麼樣的情況下失敗了。

而 `Either e a` 則能讓我們可以加入一個可能會發生錯誤的 context，還可以增加些有用的訊息，這樣能讓我們知道究竟是什麼東西出錯了。一個 `Either e a` 的值可以是代表正確的 `Right`，或是代表錯誤的 `Left`，例如說：

```
ghci> :t Right 4
Right 4 :: (Num t) => Either a t
ghci> :t Left "out of cheese error"
Left "out of cheese error" :: Either [Char] b
```

這就像是加強版的 `Maybe`，他看起來實在很像一個 monad，畢竟他也可以當作是一個可能會發生錯誤的 context，只是多了些訊息罷了。

在 `Control.Monad.Error` 裡面有他的 `Monad` instance。

```
instance (Error e) => Monad (Either e) where
    return x = Right x
    Right x >>= f = f x
    Left err >>= f = Left err
    fail msg = Left (strMsg msg)
```

`return` 就是建立起一個最小的 context，由於我們用 `Right` 代表正確的結果，所以他把值包在一個 `Right` constructor 裡面。就像實作 `Maybe` 時的 `return` 一樣。

`>>=` 會檢查兩種可能的情況：也就是 `Left` 跟 `Right`。如果進來的是 `Right`，那我們就呼叫 `f`，就像我們在寫 `Just` 的時候一樣，只是呼叫對應的函數。而在錯誤的情況下，`Left` 會被傳出來，而且裡面保有描述失敗的值。

`Either e` 的 `Monad` instance 有一項額外的要求，就是包在 `Left` 中的型態，也就是 `e`，必須是 `Error` typeclass 的 instance。`Error` 這個 typeclass 描述一個可以被當作錯誤訊息的型態。他定義了 `strMsg` 這個函數，他接受一個用字串表達的錯誤。一個明顯的範例就是 `String` 型態，當他是 `String` 的時候，`strMsg` 只不過回傳他接受到的字串。

```
ghci> :t strMsg
strMsg :: (Error a) => String -> a
ghci> strMsg "boom!" :: String
"boom!"
```

但因為我們通常在用 `Either` 來描述錯誤的時候，是用 `String` 來裝錯誤訊息，所以我們也不用擔心這一點。當在 `do` 裡面做 pattern match 失敗的時候，`Left` 的值會拿來代表失敗。

總之來看看一個範例吧：

```
ghci> Left "boom" >>= \x -> return (x+1)
Left "boom"
ghci> Right 100 >>= \x -> Left "no way!"
Left "no way!"
```

當我們用 `>>=` 來把一個 `Left` 餵進一個函數，函數的運算會被忽略而直接回傳丟進去的 `Left` 值。當我們餵 `Right` 值給函數，函數就會被計算而得到結果，但函數還是產生了一個 `Left` 值。

當我們試著餵一個 `Right` 值給函數，而且函數也成功地計算，我們卻碰到了一個奇怪的 type error。

```
ghci> Right 3 >>= \x -> return (x + 100)

<interactive>:1:0:
Ambiguous type variable `a' in the constraints:
  `Error a' arising from a use of `it' at <interactive>:1:0-33
  `Show a' arising from a use of `print' at <interactive>:1:0-33
Probable fix: add a type signature that fixes these type variable(s)
```

Haskell 警告說他不知道要為 `e` 選擇什麼樣的型態，儘管我們是要印出 `Right` 的值。這是因為 `Error e` 被限制成 `Monad`。把 `Either` 當作 `Monad` 使用就會碰到這樣的錯誤，你只要明確寫出 type signature 就行了：

```
ghci> Right 3 >>= \x -> return (x + 100) :: Either String Int
Right 103
```

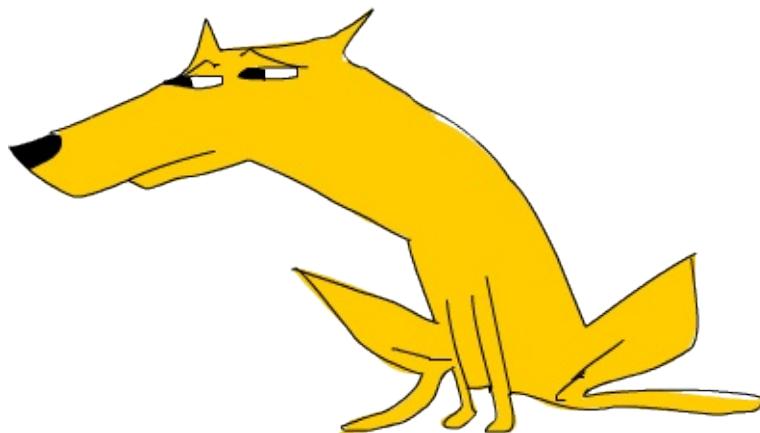
這樣就沒問題了。

撇除這個小毛病，把 `Either` 當 `Monad` 使用就像使用 `Maybe` 一樣。在前一章中，我們展示了 `Maybe` 的使用方式。你可以把前一章的範例用 `Either` 重寫當作練習。

## 一些實用的 Moanic functions

在這個章節，我們會看看一些操作 monadic value 的函數。這樣的函數通常我們稱呼他們為 monadic function。其中有些你是第一次見到，但有些不過是 `filter` 或 `foldl` 的變形。讓我們來看看吧！

### liftM



當我們開始學習 Monad 的時候，我們是先學習 functors，他代表可以被 map over 的事物。接著我們學了 functors 的加強版，也就是 applicative functors，他可以對 applicative values 做函數的套用，也可以把一個一般值放到一個預設的 context 中。最後，我們介紹在 applicative functors 上更進一步的 monad，他讓這些具有 context 的值可以被餵進一般函數中。

也就是說每一個 monad 都是個 applicative functor，而每一個 applicative functor 也都是一個 functor。`Applicative typeclass` 中有加入限制，讓每一個 `Applicative` 都是 `Functor`。但 `Monad` 却沒有這樣的限制，讓每個 `Monad` 都是 `Applicative`。這是因為 `Monad` 這個 typeclass 是在 `Applicative` 引入前就存在的緣故。

但即使每個 monad 都是一個 functor，但我們不需要依賴 `Functor` 的定義。那是因為我們有 `liftM` 這個函數。他會接受一個函數跟一個 monadic value，然後把函數 map over 那些 monadic value。所以他其實就是 `fmap`，以下是他的型態：

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
```

而這是 `fmap` 的型態：

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

如果 `Functor` 跟 `Monad` 的 instance 遵守 functor 跟 monad 的法則（到目前為止我們看過的 monad 都遵守），那這兩個函數其實是等價的。這就像 `pure` 跟 `return` 其實是同一件事，只是一個在 `Applicative` 中，而另外一個在 `Monad` 裡面，我們來試試看 `liftM` 吧：

```

ghci> liftM (*3) (Just 8)
Just 24
ghci> fmap (*3) (Just 8)
Just 24
ghci> runWriter $ liftM not $ Writer (True, "chickpeas")
(False, "chickpeas")
ghci> runWriter $ fmap not $ Writer (True, "chickpeas")
(False, "chickpeas")
ghci> runState (liftM (+100) pop) [1,2,3,4]
(101,[2,3,4])
ghci> runState (fmap (+100) pop) [1,2,3,4]
(101,[2,3,4])

```

我們已經知道 `fmap` 是如何運作在 `Maybe` 上。而 `liftM` 又跟 `fmap` 等價。對於 `Writer` 型態的值而言，函數只有對他的第一個 component 做處理。而對於改變狀態的計算，`fmap` 跟 `liftM` 也都是產生另一個改變狀態的計算。我們也看過了 `(+100)` 當作用在 `pop` 上會產生 `(1, [2,3,4])`。

來看看 `liftM` 是如何被實作的：

```

liftM :: (Monad m) => (a -> b) -> m a -> m b
liftM f m = m >= (\x -> return (f x))

```

或者用 `do` 來表示得清楚些

```

liftM :: (Monad m) => (a -> b) -> m a -> m b
liftM f m = do
  x <- m
  return (f x)

```

我們餵一個 monadic value `m` 給函數，我們套用那個函數然後把結果放進一個預設的 context。由於遵守 monad laws，這保證這操作不會改變 context，只會呈現最後的結果。我們可以看到實作中 `liftM` 也沒有用到 Functor 的性質。這代表我們能只用 monad 提供給我們的就實作完 `fmap`。這特性讓我們可以得到 monad 比 functor 性質要強的結論。

`Applicative` 讓我們可以操作具有 context 的值就像操作一般的值一樣。就像這樣：

```

ghci> (+) <$> Just 3 <*> Just 5
Just 8
ghci> (+) <$> Just 3 <*> Nothing
Nothing

```

使用 applicative 的特性讓事情變得很精簡。`<$>` 不過就是 `fmap`，而 `<*>` 只是一個具有下列型態的函數：

```
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
```

他有點像 `fmap`，只是函數本身有一個 context。我們必須把他從 context 中抽出，對 `f a` 做 map over 的東做，然後再放回 context 中。由於在 Haskell 中函數預設都是 curried，我們便能用 `<$>` 以及 `<*>` 來讓接受多個參數的函數也能接受 applicative 種類的值。

總之 `<*>` 跟 `fmap` 很類似，他也能只用 `Monad` 保證的性質實作出來。`ap` 這個函數基本上就是 `<*>`，只是他是限制在 `Monad` 上而不是 `Applicative` 上。這邊是他的定義：

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
ap mf m = do
  f <- mf
  x <- m
  return (f x)
```

`mf` 是一個 monadic value，他的結果是一個函數。由於函數跟值都是放在 context 中，假設我們從 context 取出的函數叫 `f`，從 context 取出的值叫 `x`，我們把 `x` 餵給 `f` 然後再把結果放回 context。像這樣：

```
ghci> Just (+3) <*> Just 4
Just 7
ghci> Just (+3) `ap` Just 4
Just 7
ghci> [(+1),(+2),(+3)] <*> [10,11]
[11,12,12,13,13,14]
ghci> [(+1),(+2),(+3)] `ap` [10,11]
[11,12,12,13,13,14]
```

由於我們能用 `Monad` 提供的函數實作出 `Applicative` 的函數，因此我們看到 `Monad` 有比 `applicative` 強的性質。事實上，當我們知道一個型態是 `monad` 的時候，大多數會先定義出 `Monad` 的 instance，然後才定義 `Applicative` 的 instance。而且只要把 `pure` 定義成 `return`，`<*>` 定義成 `ap` 就行了。同樣的，如果你已經有了 `Monad` 的 instance，你也可以簡單的定義出 `Functor`，只要把 `fmap` 定義成 `liftM` 就行了。

`liftA2` 是一個方便的函數，他可以把兩個 applicative 的值餵給一個函數。他的定義很簡單：

```
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c
liftA2 f x y = f <$> x <*> y
```

`liftM2` 也是做差不多的事情，只是多了 `Monad` 的限制。在函式庫中其實也有 `liftM3`，`liftM4` 跟 `liftM5`。

我們看到了 monad 相較於 applicative 跟 functor 有比較強的性質。儘管 monad 有 functor 跟 applicative functor 的性質，但他們不見得有 Functor 跟 Applicative 的 instance 定義。所以我們檢視了一些在 monad 中定義，且等價於 functor 或 applicative functor 所具有的函數。

## The join function

如果一個 monadic value 的結果是另一個 monadic value，也就是其中一個 monadic value 被包在另一個裡面，你能夠把他們變成一個普通的 monadic value 嗎？就好像把他們打平一樣。譬如說，我們有 Just (Just 9)，我們能夠把他變成 Just 9 嗎？事實上是可以的，這也是 monad 的一個性質。也就是我要看的 join 函數，他的型態是這樣：

```
join :: (Monad m) => m (m a) -> m a
```

他接受一個包在另一個 monadic value 中的 monadic value，然後會回給我們一個普通的 monadic value。這邊有一些 Maybe 的範例：

```
ghci> join (Just (Just 9))
Just 9
ghci> join (Just Nothing)
Nothing
ghci> join Nothing
Nothing
```

第一行是一個計算成功的結果包在另一個計算成功的結果，他們應該要能結合成為一個比較大的計算成功的結果。第二行則是一個 Nothing 包在一個 Just 中。我們之前在處理 Maybe 型態的值時，會用 `<*>` 或 `>=` 把他們結合起來。輸入必須都是 Just 時結果出來才會是 Just。如果中間有任何的失敗，結果就會是一個失敗的結果。而第三行就是這樣，我們嘗試把失敗的結果接合起來，結果也會是一個失敗。

要 join 一個 list 也是很簡單：

```
ghci> join [[1,2,3],[4,5,6]]
[1,2,3,4,5,6]
```

你可以看到，對於 list 而言 join 不過就是 concat。而要 join 一個包在 writer 中的 Writer，我們必須用 mappend：

```
ghci> runWriter $ join (Writer (Writer (1,"aaa"),"bbb"))
(1,"bbbaaa")
```

"bbb" 先被加到 monoid 中，接著 "aaa" 被附加上去。你想要檢視 writer 中的值的話，必須先把值寫進去才行。

要對 `Either` 做 `join` 跟對 `Maybe` 做 `join` 是很類似的：

```
ghci> join (Right (Right 9)) :: Either String Int
Right 9
ghci> join (Right (Left "error")) :: Either String Int
Left "error"
ghci> join (Left "error") :: Either String Int
Left "error"
```

如果我們對一個包了另外一個改變狀態的計算的進行改變狀態的計算，要作 `join` 的動作會讓外面的先被計算，然後才是計算裡面的：

```
ghci> runState (join (State $ \s -> (push 10,1:2:s))) [0,0,0]
((),[10,1,2,0,0,0])
```

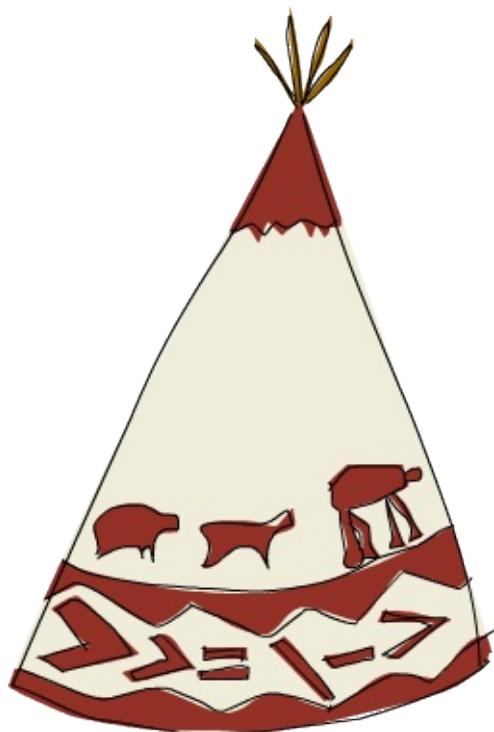
這邊的 `lambda` 函數接受一個狀態，並把 `2` 跟 `1` 放到堆疊中，並把 `push 10` 當作他的結果。當對整個東西做 `join` 的時候，他會先把 `2` 跟 `1` 放到堆疊上，然後進行 `push 10` 的計算，因而把 `10` 放到堆疊的頂端。

`join` 的實作像是這樣：

```
join :: (Monad m) => m (m a) -> m a
join mm = do
  m <- mm
  m
```

因為 `mm` 的結果會是一個 monadic value，我們單獨用 `m <- mm` 拿取他的結果。這也可以說明 `Maybe` 只有當外層跟內層的值都是 `Just` 的時候才會是 `Just`。如果把 `mm` 的值設成 `Just (Just 8)` 的話，他看起來會是這樣：

```
joinedMaybes :: Maybe Int
joinedMaybes = do
  m <- Just (Just 8)
  m
```



最有趣的是對於一個 monadic value 而言，用 `>=` 把他餵進一個函數其實等價於對 monad 做 mapping over 的動作，然後用 `join` 來把值從 nested 的狀態變成扁平的狀態。也就是說 `m >= f` 其實就是 `join (fmap f m)`。如果你仔細想想的話其實很明顯。`>=` 的使用方式是，把一個 monadic value 餵進一個接受普通值的函數，但他卻會回傳 monadic value。如果我們 map over 一個 monadic value，我們會做成一個 monadic value 包了另外一個 monadic value。例如說，我們現在手上有 `Just 9` 跟 `\x -> Just (x+1)`。如果我們把這個函數 map over `Just 9`，我們會得到 `Just (Just 10)`

事實上 `m >= f` 永遠等價於 `join (fmap f m)` 這性質非常有用。如果我們要定義自己的 `Monad` instance，要知道怎麼把 nested monadic value 變成扁平比起要定義 `>=` 是比較容易的一件事。

## filterM

`filter` 函數是 Haskell 中不可或缺的要素。他接受一個斷言(predicate)跟一個 list 來過濾掉斷言為否的部份並回傳一個新的 list。他的型態是這樣：

```
filter :: (a -> Bool) -> [a] -> [a]
```

`predicate` 能接 list 中的一個元素並回傳一個 `Bool` 型態的值。但如果 `Bool` 型態其實是一個 monadic value 呢？也就是他有一個 context。例如說除了 `True` 跟 `False` 之外還伴隨一個 monoid，像是 `["Accepted the number 5"]`，或 `["3 is too small"]`。照前面所學的聽起來是沒問題，而且產出的 list 也會跟隨 context，在這個例子中就是 log。所以如果 `Bool` 會回傳伴隨 context 的布林值，我們會認為最終的結果也會具有 context。要不然這些 context 都會在處理過程中遺失。

在 `Control.Monad` 中的 `filterM` 函數正是我們所需要的，他的型態如下：

```
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
```

`predicate` 會回傳一個 monadic value，他的結果會是 `Bool` 型態，由於他是 monadic value，他的 context 有可能會是任何 context，譬如說可能的失敗，non-determinism，甚至其他的 context。一旦我們能保證 context 也會被保存在最後的結果中，結果也就是一個 monadic value。

我們來寫一個接受 `list` 然後過濾掉小於 4 的函數。先嘗試使用 `filter` 函數：

```
ghci> filter (\x -> x < 4) [9,1,5,2,10,3]
[1,2,3]
```

很簡單吧。接著我們在做個 `predicate`，除了表達 `True` 或 `False` 之外，還提供了一個 `log`。我們會用 `Writer` monad 來表達這件事：

```
keepSmall :: Int -> Writer [String] Bool
keepSmall x
| x < 4 = do
    tell ["Keeping " ++ show x]
    return True
| otherwise = do
    tell [show x ++ " is too large, throwing it away"]
    return False
```

這個函數會回傳 `Writer [String] Bool` 而不是一個單純的 `Bool`。他是一個 monadic predicate。如果掃到的數字小於 4 的話，我們就會回報要保存他，而且回傳 `return True`。

接著，我們把他跟一個 `list` 餵給 `filterM`。由於 `predicate` 會回傳 `Writer`，所以結果仍會是一個 `Writer` 值。

```
ghci> fst $ runWriter $ filterM keepSmall [9,1,5,2,10,3]
[1,2,3]
```

要檢查 `Writer` 的結果，我們想要印出 `log` 看看裡面有什麼東西：

```
ghci> mapM_ putStrLn $ snd $ runWriter $ filterM keepSmall [9,1,5,2,10,3]
9 is too large, throwing it away
Keeping 1
5 is too large, throwing it away
Keeping 2
10 is too large, throwing it away
Keeping 3
```

提供 monadic predicate 給 `filterM`，我們便能夠做 filter 的動作，同時還能保有 monadic context。

一個比較炫的技巧是用 `filterM` 來產生一個 list 的 powerset。一個 powerset 就是一個集合所有子集所形成的集合。如果說我們的 list 是 `[1,2,3]`，那他個 powerset 就會是：

```
[1,2,3]
[1,2]
[1,3]
[1]
[2,3]
[2]
[3]
[]
```

換句話說，要產生一個 powerset 就是要列出所有要丟掉跟保留的組合。`[2,3]` 只不過代表我們把 `1` 純掉而已。

我們要依賴 non-determinism 來寫我們這產生 powerset 的函數。我們接受一個 list `[1,2,3]` 然後查看第一個元素，這個例子中是 `1`，我們會問：我們要保留他呢？還是丟掉他呢？答案是我們都要做。所以我們會用一個 non-deterministic 的 predicate 來過濾我的 list。也就是我們的 `powerset` 函數：

```
powerset :: [a] -> [[a]]
powerset xs = filterM (\x -> [True, False]) xs
```

等等，我們已經寫完了嗎？沒錯，就這麼簡單，我們可以同時丟掉跟保留每個元素。只要我們用 non-deterministic predicate，那結果也就是一個 non-deterministic value，也便是一個 list 的 list。試著跑跑看：

```
ghci> powerset [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

這樣的寫法需要讓你好好想一下，但如果你能接受 list 實際就是 non-deterministic value 的話，那要想通會比較容易一些。

## foldM

`foldl` 的 monadic 的版本叫做 `foldM`。如果你還有印象的話，`foldl` 會接受一個 binary 函數，一個起始累加值跟一串 list，他會從左邊開始用 binary 函數每次帶進一個值來 fold。`foldM` 也是做同樣的事，只是他接受的這個 binary 函數會產生 monadic value。不意外的，他的結果也會是 monadic value。`foldl` 的型態是：

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

而 `foldM` 的型態則是：

```
foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
```

binary 函數的回傳值是 monadic，所以結果也會是 monadic。我們來試著把 list 的值用 fold 全部加起來：

```
ghci> foldl (\acc x -> acc + x) 0 [2,8,3,1]
14
```

這邊起始的累加值是 `0`，首先 `2` 會被加進去，變成 `2`。然後 `8` 被加進去變成 `10`，直到我們沒有值可以再加，那便是最終的結果。

但如果我們想額外加一個條件，也就是當碰到一個數字大於 `9` 時候，整個運算就算失敗呢？一種合理的修改就是用一個 binary 函數，他會檢查現在這個數是否大於 `9`，如果是便引發失敗，如果不是就繼續。由於有失敗的可能性，我們便需要這個 binary 函數回傳一個 `Maybe`，而不是一個普通的值。我們來看看這個函數：

```
binSmalls :: Int -> Int -> Maybe Int
binSmalls acc x
| x > 9      = Nothing
| otherwise = Just (acc + x)
```

由於這邊的 binary 函數是 monadic function，我們不能用普通的 `foldl`，我們必須用

`foldM`：

```
ghci> foldM binSmalls 0 [2,8,3,1]
Just 14
ghci> foldM binSmalls 0 [2,11,3,1]
Nothing
```

由於這串 list 中有一個數值大於 `9`，所以整個結果會是 `Nothing`。另外你也可以嘗試 fold 一個回傳 `Writer` 的 binary 函數，他會在 fold 的過程中紀錄你想紀錄的資訊。

## Making a safe RPN calculator



之前的章節我們實作了一個 RPN 計算機，但我們沒有做錯誤的處理。他只有在輸入是合法的時候才會運算正確。假如有東西出錯了，整個程式便會當掉。我們在這章看到了要怎樣把程式碼轉換成 monadic 的版本，我們先嘗適用 `Maybe` monad 來幫我們的 RPN 計算機加上些錯誤處理。

我們的 RPN 計算機接受一個像 `"1 3 + 2 *"` 這樣的字串，把他斷成 word，變成 `["1", "3", "+", "2", "*"]` 這樣。然後用一個 `binary` 函數，跟一個空的堆疊，從左邊開始或是將數值推進堆疊中，或是操作堆疊最上層的兩個元素。

以下便是程式的核心部份：

```
import Data.List

solveRPN :: String -> Double
solveRPN = head . foldl foldingFunction [] . words
```

我們把輸入變成一個字串的 list，從左邊開始 fold，當堆疊中只剩下一個元素的時候，他便是我們要的答案。以下是我們的 `foldingFunction` 函數：

```
foldingFunction :: [Double] -> String -> [Double]
foldingFunction (x:y:ys) "***" = (x * y):ys
foldingFunction (x:y:ys) "+" = (x + y):ys
foldingFunction (x:y:ys) "-" = (y - x):ys
foldingFunction xs numberString = read numberString:xs
```

這邊我們的累加元素是一個堆疊，我們用一個 `Double` 的 list 來表示他。當我們在做 folding 的過程，如果當前的元素是一個 operator，他會從堆疊上拿下兩個元素，用 operator 施行運算然後把結果放回堆疊。如果當前的元素是一個表示成字串的數字，他會把字串轉換成數字，並回傳一個新的堆疊包含了轉換後的數字。

我們首先把我們的 folding 函數加上處理錯誤的 case，所以他的型態會變成這樣：

```
foldingFunction :: [Double] -> String -> Maybe [Double]
```

他不是回傳一個 `Just` 的堆疊就是回傳 `Nothing`。

`reads` 函數就像 `read` 一樣，差別在於他回傳一個 list。在成功讀取的情況下 list 中只包含讀取的那個元素。如果他失敗了，他會回傳一個空的 list。除了回傳讀取的元素，他也回傳剩下讀取失敗的元素。他必須要看完整串輸入，我們想把他弄成一個 `readMaybe` 的函數，好方便我們進行。

```
readMaybe :: (Read a) => String -> Maybe a
readMaybe st = case reads st of [(x, "")] -> Just x
                     _ -> Nothing
```

測試結果如下：

```
ghci> readMaybe "1" :: Maybe Int
Just 1
ghci> readMaybe "GO TO HELL" :: Maybe Int
Nothing
```

看起來運作正常。我們再把他變成一個可以處理失敗情況的 monadic 函數

```
foldingFunction :: [Double] -> String -> Maybe [Double]
foldingFunction (x:y:ys) "*" = return ((x * y):ys)
foldingFunction (x:y:ys) "+" = return ((x + y):ys)
foldingFunction (x:y:ys) "-" = return ((y - x):ys)
foldingFunction xs numberString = liftM (":xs") (readMaybe numberString)
```

前三種 case 跟前面的很像，只差在堆疊現在是包在 `Just` 裡面（我們常常是用 `return` 來做到這件事，但其實我們也可以用 `Just`）。在最後一種情況，我們用 `readMaybe numberString` 然後我們用 `(:xs)` `map over` 他。所以如果堆疊 `xs` 是 `[1.0, 2.0]` 且 `readMaybe numberString` 產生 `Just 3.0`，那結果便是 `Just [3.0, 1.0, 2.0]`。如果 `readMaybe numberString` 產生 `Nothing` 那結果便是 `Nothing`。我們來試著跑跑看 folding 函數

```

ghci> foldingFunction [3,2] "*"
Just [6.0]
ghci> foldingFunction [3,2] "-"
Just [-1.0]
ghci> foldingFunction [] "*"
Nothing
ghci> foldingFunction [] "1"
Just [1.0]
ghci> foldingFunction [] "1 wawawawa"
Nothing

```

看起來正常運作。我們可以用他來寫一個新的 `solveRPN`。

```

import Data.List

solveRPN :: String -> Maybe Double
solveRPN st = do
  [result] <- foldM foldingFunction [] (words st)
  return result

```

我們仍是接受一個字串把他斷成一串 `word`。然後我們用一個空的堆疊來作 `folding` 的動作，只差在我們用的是 `foldM` 而不是 `foldl`。`foldM` 的結果會是 `Maybe`，`Maybe` 裡面包含了  
一個只有一個元素的 `list`。我們用 `do expression` 來取出值，把他綁定到 `result` 上。當  
`foldM` 回傳 `Nothing` 的時候，整個結果就變成 `Nothing`。也特別注意我們有在 `do` 裡面做  
`pattern match` 的動作，所以如果 `list` 中不是只有一個元素的話，最後結果便會是 `Nothing`。  
最後一行我們用 `return result` 來展示 RPN 計算的結果，把他包在一個 `Maybe` 裡面。

```

ghci> solveRPN "1 2 * 4 +"
Just 6.0
ghci> solveRPN "1 2 * 4 + 5 *"
Just 30.0
ghci> solveRPN "1 2 * 4"
Nothing
ghci> solveRPN "1 8 wharglbllargh"
Nothing

```

第一個例子會失敗是因為 `list` 中不是只有一個元素，所以 `do` 裡面的 `pattern matching` 失敗了。第二個例子會失敗是因為 `readMaybe` 回傳了 `Nothing`。

## Composing monadic functions

當我們介紹 monad law 的時候，我們說過 `<=<` 就像是函數合成一樣，只差在一個是作用在普通函數 `a -> b`。一個是作用在 monadic 函數 `a -> m b`。

```
ghci> let f = (+1) . (*100)
ghci> f 4
401
ghci> let g = (\x -> return (x+1)) <=< (\x -> return (x*100))
ghci> Just 4 >>= g
Just 401
```

在這個例子中我們合成了兩個普通的函數，並餵給給他 `4`。我們也合成了兩個 monadic 函數並用 `>>=` 餵給他 `Just 4`。

如果我們在 `list` 中有一大堆函數，我們可以把他們合成一個巨大的函數。用 `id` 當作累加的起點，`.` 當作 binary 函數，用 `fold` 來作這件事。

```
ghci> let f = foldr (.) id [(+1),(*100),(+1)]
ghci> f 1
201
```

`f` 接受一個數字，然後會幫他加 `1`，乘以 `100`，再加 `1`。我們也可以將 monadic 函數用同樣的方式做合成，只是不用 `.` 而用 `<=<`，不用 `id` 而用 `return`。我們不需要 `foldM`，由於 `<=<` 只用 `foldr` 就足夠了。

當我們在之前的章節介紹 `list monad` 的時候，我們用他來解決一個騎士是否能在三步內走到另一點的問題。那個函數叫做 `moveKnight`，他接受一個座標然後回傳所有可能的下一步。然後產生出所有可能三步的移動。

```
in3 start = return start >>= moveKnight >>= moveKnight >>= moveKnight
```

要檢查我們是否能只用三步從 `start` 走到 `end`，我們用下列函數

```
canReachIn3 :: KnightPos -> KnightPos -> Bool
canReachIn3 start end = end `elem` in3 start
```

如果使用 monadic 版本的合成的話，我們也可以做一個類似的 `in3`，但我們希望他不只有三步的版本，而希望有任意步的版本。如果你仔細觀察 `in3`，他只不過用 `>>=` 跟 `moveKnight` 把之前所有可能結果餵到下一步。把他一般化，就會像下面的樣子：

```
import Data.List

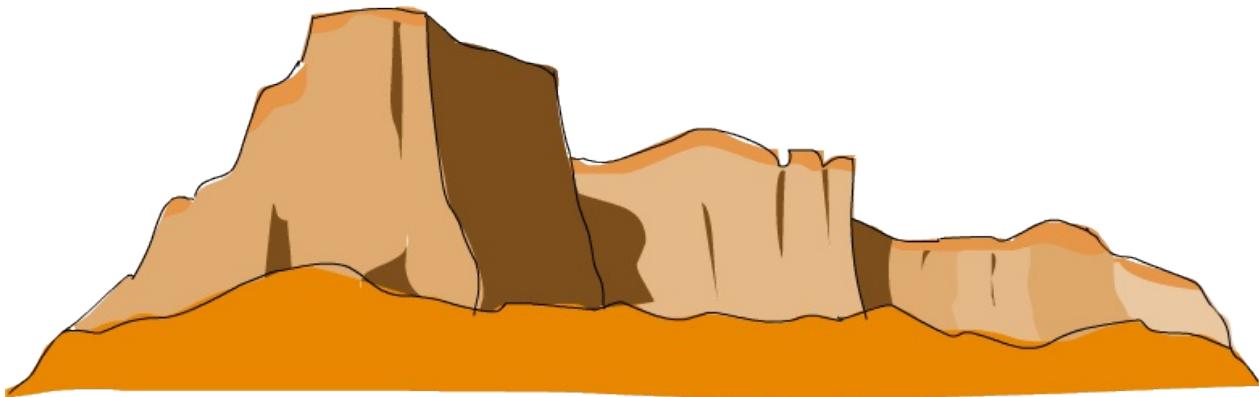
inMany :: Int -> KnightPos -> [KnightPos]
inMany x start = return start >>= foldr (<=<) return (replicate x moveKnight)
```

首先我們用 `replicate` 來做出一個 list，裡面有 `x` 份的 `moveKnight`。然後我們把所有函數都合成起來，就會給我們從起點走 `x` 步內所有可能的的位置。然後我們只需要把起始位置餵給他就好了。

我們也可以一般化我們的 `canReachIn3`：

```
canReachIn :: Int -> KnightPos -> KnightPos -> Bool
canReachIn x start end = end `elem` inMany x start
```

## 定義自己的 Monad



在這一章節，我們會帶你看看究竟一個型態是怎麼被辨認，確認是一個 monad 而且正確定義出 `Monad` 的 instance。我們通常不會為了定義 monad 而定義。比較常發生的是，我們想要針對一個問題建立模型，卻稍後發現我們定義的型態其實是一個 `Monad`，所以就定義一個 `Monad` 的 instance。

正如我們看到的，list 是被拿來當作 non-deterministic values。對於 `[3, 5, 9]`，我們可以看作是一個 non-deterministic value，我們不能知道究竟是哪一個。當我們把一個 list 用 `>>=` 餵給一個函數，他就是把一串可能的選擇都丟給函數，函數一個個去計算在那種情況下的結果，結果也便是一個 list。

如果我們把 `[3, 5, 9]` 看作是 `3, 5, 9` 各出現一次，但這邊沒有每一種數字出現的機率。如果我們把 non-deterministic 的值看作是 `[3, 5, 9]`，但 `3` 出現的機率是 50%，`5` 跟 `9` 出現的機率各是 25% 呢？我們來試著用 Haskell 描述看看。

如果說 list 中的每一個元素都伴隨著他出現的機率。那下面的形式就蠻合理的：

```
[(3, 0.5), (5, 0.25), (9, 0.25)]
```

在數學上，機率通常不是用百分比表示，而是用介於 0 跟 1 的實數表示。0 代表不可能會發生，而 1 代表絕對會發生。但浮點數很有可能很快隨著運算失去精準度，所以 Haskell 有提供有理數。他的型態是擺在 `Data.Ratio` 中，叫做 `Rational`。要創造出一個 `Rational`，我

們會把他寫成一個分數的形式。分子跟分母用 `%` 分隔。這邊有幾個例子：

```
ghci> 1%4
1 % 4
ghci> 1%2 + 1%2
1 % 1
ghci> 1%3 + 5%4
19 % 12
```

第一行代表四分之一，第二行代表兩個二分之一加起來變成一。而第三行我們把三分之一跟四分之五加起來變成十二分之十九。所以我們來用 `Rational` 取代浮點數來當作我們的機率值吧。

```
ghci> [(3,1%2),(5,1%4),(9,1%4)]
[(3,1 % 2),(5,1 % 4),(9,1 % 4)]
```

所以 `3` 有二分之一的機會出現，而 `5` 跟 `9` 有四分之一的機會出現。

可以看到我們幫 `list` 加上了一些額外的 `context`。再我們繼續深入之前，我們用一個 `newtype` 把他包起來，好讓我們幫他寫 `instance`。

```
import Data.Ratio

newtype Prob a = Prob { getProb :: [(a,Rational)] } deriving Show
```

接著我們想問，這是一個 `functor` 嗎？`list` 是一個 `functor`，所以很有可能他也是一個 `functor`，畢竟我們只是在 `list` 上多加一些東西而已。在 `list` 的情況下，我們可以針對每個元素用函數做處理。這邊我們也是用函數針對每個元素做處理，只是我們是輸出機率值。所以我們就來寫個 `functor` 的 `instance` 吧。

```
instance Functor Prob where
  fmap f (Prob xs) = Prob $ map (\(x,p) -> (f x, p)) xs
```

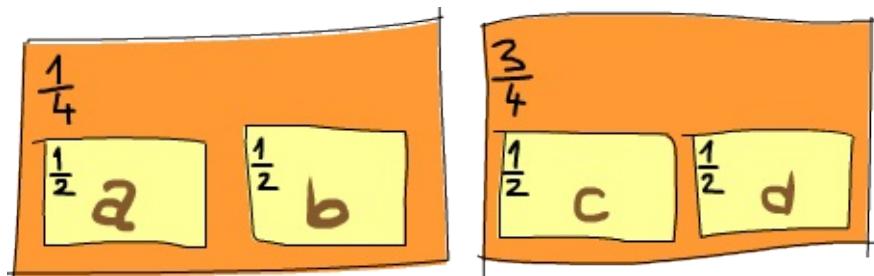
我們可以用 `pattern matching` 的方式把 `newtype` 解開來，套用函數 `f` 之後再包回去。過程中不會動到機率值。

```
ghci> fmap negate (Prob [(3,1%2),(5,1%4),(9,1%4)])
Prob {getProb = [(-3,1 % 2),(-5,1 % 4),(-9,1 % 4)]}
```

要注意機率的和永遠是 `1`。如果我們沒有漏掉某種情形的話，沒有道理他們加起來的值不為 `1`。一個有 75% 機率是正面以及 50% 機率是反面的硬幣根本沒什麼道理。

接著要問一個重要的問題，他是一個 monad 嗎？我們知道 list 是一個 monad，所以他很有可能也是一個 monad。首先來想想 `return`。他在 list 是怎麼運作的？他接受一個普通的值並把他放到一個 list 中變成只有一個元素的 list。那在這邊又如何？由於他是一個最小的 context，他也應該是一個元素的 list。那機率值呢？`return x` 的值永遠都是 `x`，所以機率值不應該是 `0`，而應該是 `1`。

至於 `>=` 呢？看起來有點複雜，所以我們換種方式來思考，我們知道 `m >= f` 會等價於 `join (fmap f m)`，所以我們來想要怎麼把一串包含 probability list 的 list 弄平。舉個例子，考慮一個 list，'a' 跟 'b' 恰出現其中一個的機率為 25%，兩個出現的機率相等。而 'c' 跟 'd' 恰出現其中一個的機率為 75%，兩個出現的機率也是相等。這邊有一個圖將情形畫出來。



每個字母發生的機率有多高呢？如果我們用四個盒子來代表每個字母，那每個盒子的機率為何？每個盒子的機率是他們所裝有的機率值相乘的結果。`'a'` 的機率是八分之一，`'b'` 同樣也是八分之一。八分之一是因為我們把二分之一跟四分之一相乘得到的結果。而 `'c'` 發生的機率是八分之三，是因為二分之一乘上四分之三。`'d'` 同樣也是八分之三。如果把所有的機率加起來，就會得到一，符合機率的規則。

來看看怎麼用一個 list 表達我們要說明的東西：

```
thisSituation :: Prob (Prob Char)
thisSituation = Prob
  [ ( Prob [('a', 1%2), ('b', 1%2)] , 1%4 )
  , ( Prob [('c', 1%2), ('d', 1%2)] , 3%4 )
  ]
```

注意到這邊的型態是 `Prob (Prob Char)`。所以我們要思考的是如何把一串包含機率 list 的 list 打平。如果能成功寫出這樣的邏輯，`>=` 不過就是 `join (fmap f m)`，我們便得到了一個 monad。我們這邊寫了一個 `flatten` 來做這件事。

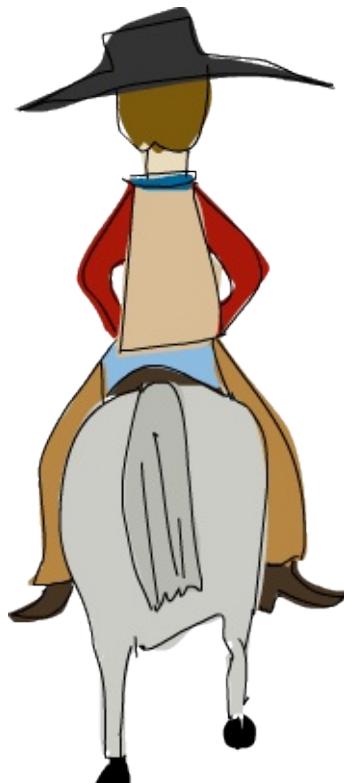
```
flatten :: Prob (Prob a) -> Prob a
flatten (Prob xs) = Prob $ concat $ map multAll xs
  where multAll (Prob innerxs, p) = map (\(x,r) -> (x, p*r)) innerxs
```

`multAll` 接受一個 tuple，裡面包含一個 probability list 跟一個伴隨的機率值 `p`，所以我們要作的事是把 list 裡面的機率值都乘以 `p`，並回傳一個新的 tuple 包含新的 list 跟新的機率值。我們將 `multAll` map over 到我們的 probability list 上，我們就成功地打平了我們的

list。

現在我們就能定義我們的 `Monad` instance。

```
instance Monad Prob where
    return x = Prob [(x, 1%1)]
    m >>= f = flatten (fmap f m)
    fail _ = Prob []
```



由於我們已經把所有苦工的做完了，定義這個 instance 顯得格外輕鬆。我們也定義了 `fail`，我們定義他的方式跟定義 list 一樣。如果在 `do` 中發生了失敗的 pattern match，那就會呼叫 `fail`。

檢查我們定義的 instance 是否遵守 monad law 也是很重要的。monad law 的第一個定律是 `return x >>= f` 應該要等價於 `f x`。要寫出嚴格的證明會很麻煩，但我們可以觀察到下列事實：首先用 `return` 做一個最小的 context，然後用 `fmap` 將一個函數 map over 這個 context，再將他打平。這樣做出來的 probability list，每一個機率值都相當於將我們最初放到 minimal context 中的值乘上 `1%1`。同樣的邏輯，也可以看出 `m >>= return` 是等價於 `m`。第三個定律是 `f <=< (g <=< h)` 應該要等價於 `(f <=< g) <=< h`。我們可以從乘法有結合律的性質，以及 list monad 的特性上推出 probability monad 也符合這個定律。`1%2 * (1%3 * 1%5)` 等於 `(1%2 * 1%3) * 1%5`。

現在我們有了一個 monad，這樣有什麼好處呢？他可以幫助我們計算機率值。我們可以把機率事件看作是具有 context 的 value，而 probability monad 可以保證機率值能正確地被計算成最終的結果。

好比說我們現在有兩個普通的硬幣以及一個灌鉛的硬幣。灌鉛的硬幣十次中有九次會出現正面，只有一次會出現反面。如果我們一次丟擲這三個硬幣，有多大的機會他們都會出現正面呢？讓我們先來表達丟擲硬幣這件事，分別丟的是灌鉛的跟普通的硬幣。

```
data Coin = Heads | Tails deriving (Show, Eq)

coin :: Prob Coin
coin = Prob [(Heads, 1%2), (Tails, 1%2)]

loadedCoin :: Prob Coin
loadedCoin = Prob [(Heads, 1%10), (Tails, 9%10)]
```

最後，來看看擲硬幣的函數：

```
import Data.List (all)

flipThree :: Prob Bool
flipThree = do
  a <- coin
  b <- coin
  c <- loadedCoin
  return (all (==Tails) [a,b,c])
```

試著跑一下的話，我們會看到儘管我們用了不公平的硬幣，三個反面的機率還是不高。

```
ghci> getProb flipThree
[(False, 1 % 40), (False, 9 % 40), (False, 1 % 40), (False, 9 % 40),
 (False, 1 % 40), (False, 9 % 40), (False, 1 % 40), (True, 9 % 40)]
```

同時出現正面的機率是四十分之九，差不多是 25% 的機會。我們的 monad 並沒有辦法 join 所有都是 `False` 的情形，也就是所有硬幣都是出現反面的情況。不過那不是個嚴重的問題，可以寫個函數來將同樣的結果變成一種結果，這就留給讀者當作習題。

在這章節中，我們從提出問題到真的寫出型態，並確認這個型態是一個 monad，寫出他的 instance 並實際操作他。這是個很棒的經驗。現在讀者們應該對於 monad 有不少的了解才是。

# Zippers 資料結構



儘管 Haskell 的純粹性質帶來很多好處，但他讓一些在非純粹語言很容易處理的一些事情變得分別要用另一種方法解決。由於 referential transparency，同樣一件事在 Haskell 中是沒有分別的。所以如果我們有一個裝滿 5 的樹，而我們希望把其中一個換成 6，那我們必須要知道我們究竟是想改變哪個 5。我們也必須知道我們身處在這棵樹的哪裡。但在 Haskell 中，每個 5 都長得一樣，我們並不能因為他們在記憶體中的位址不同就把他們區分開來。我們也不能改變任何狀態，當我們想要改變一棵樹的時候，我們實際上是說我們要一棵新的樹，只是他長得很像舊的。一種解決方式是記住一條從根節點到現在這個節點的路徑。我們可以這樣表達：給定一棵樹，先往左走，再往右走，再往左走，然後改變你走到的元素。雖然這是可行的，但這非常沒有效率。如果我們想接連改變一個在附近的節點，我們必須再從根節點走一次。在這個章節中，我們會看到我們可以集中注意在某個資料結構上，這樣讓改變資料結構跟遍歷的動作非常有效率。

## 來走二元樹吧！

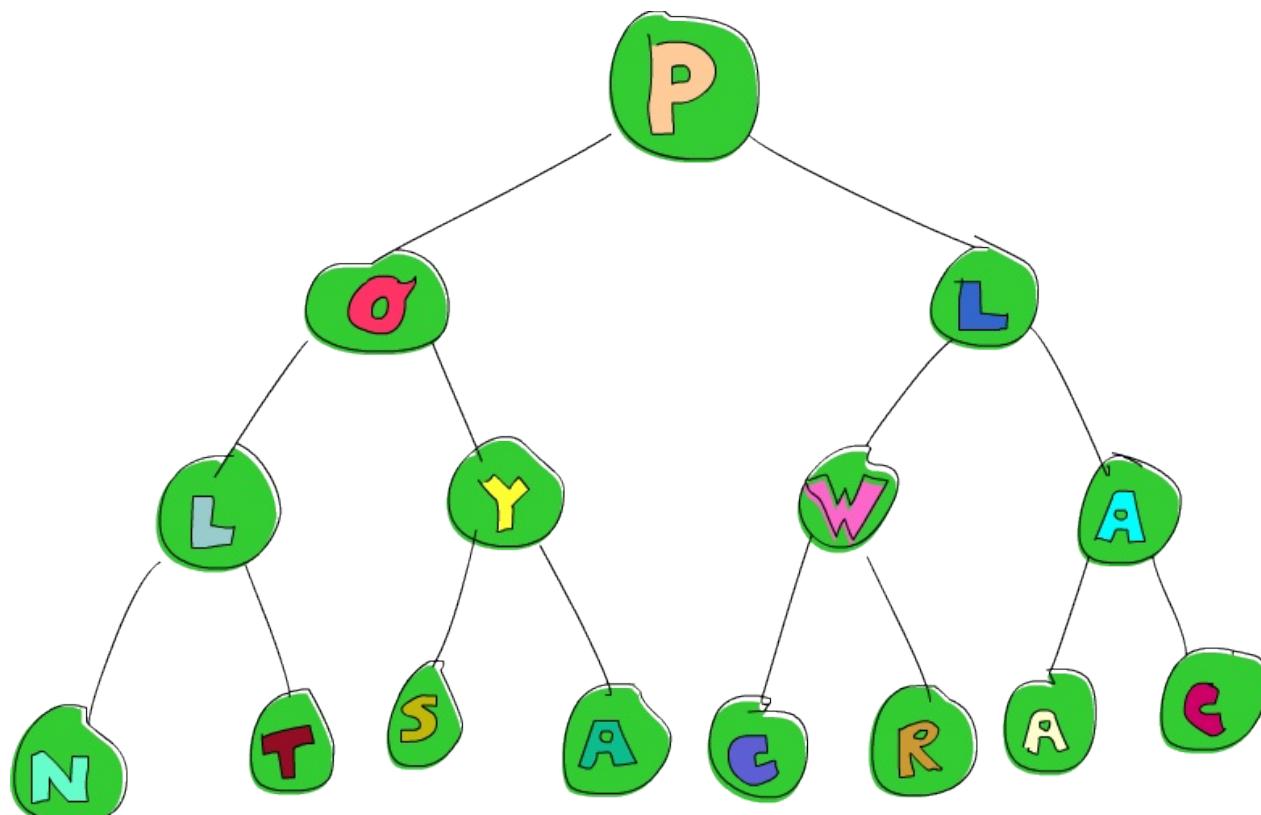
我們在生物課中學過，樹有非常多種。所以我們來自己發明棵樹吧！

```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show)
```

這邊我們的樹不是空的就是有兩棵子樹。來看看一個範例：

```
freeTree :: Tree Char
freeTree =
  Node 'P'
    (Node 'O'
      (Node 'L'
        (Node 'N' Empty Empty)
        (Node 'T' Empty Empty))
      )
    (Node 'Y'
      (Node 'S' Empty Empty)
      (Node 'A' Empty Empty))
    )
  (Node 'L'
    (Node 'W'
      (Node 'C' Empty Empty)
      (Node 'R' Empty Empty))
    )
  (Node 'A'
    (Node 'A' Empty Empty)
    (Node 'C' Empty Empty))
  )
```

畫成圖的話就是像這樣：



注意到 `w` 這個節點了嗎？如果我們想要把他變成 `P`。我們會怎麼做呢？一種方式是用 pattern match 的方式做，直到我們找到那個節點為止。要先往右走再往左走，再改變元素內容，像是這樣：

```
changeToP :: Tree Char -> Tree Char
changeToP (Node x l (Node y (Node _ m n) r)) = Node x l (Node y (Node 'P' m n) r)
```

這不只看起來很醜，而且很不容易閱讀。這到底是怎麼回事？我們使用 pattern match 來拆開我們的樹，我們把 root 繩定成 `x`，把左子樹繩定成 `l`。對於右子樹我們繼續使用 pattern match。直到我們碰到一個子樹他的 root 是 `'w'`。到此為止我們再重建整棵樹，新的樹只差在把 `'w'` 改成了 `'P'`。

有沒有比較好的作法呢？有一種作法是我們寫一個函數，他接受一個樹跟一串 list，裡面包含有行走整個樹時的方向。方向可以是 `L` 或是 `R`，分別代表向左走或向右走。我們只要跟隨指令就可以走達指定的位置：

```
data Direction = L | R deriving (Show)
type Directions = [Direction]

changeToP :: Directions -> Tree Char -> Tree Char
changeToP (L:ds) (Node x l r) = Node x (changeToP ds l) r
changeToP (R:ds) (Node x l r) = Node x l (changeToP ds r)
changeToP [] (Node _ l r) = Node 'P' l r
```

如果在 list 中的第一個元素是 `L`，我們會建構一個左子樹變成 `'P'` 的新樹。當我們遞迴地呼叫 `changeToP`，我們只會傳給他剩下的部份，因為前面的部份已經看過了。對於 `R` 的 case 也一樣。如果 list 已經消耗完了，那表示我們已經走到我們的目的地，所以我們就回傳一個新的樹，他的 root 被修改成 `'P'`。

要避免印出整棵樹，我們要寫一個函數告訴我們目的地究竟是什麼元素。

```
elemAt :: Directions -> Tree a -> a
elemAt (L:ds) (Node _ l _) = elemAt ds l
elemAt (R:ds) (Node _ _ r) = elemAt ds r
elemAt [] (Node x _ _) = x
```

這函數跟 `changeToP` 很像，只是他不會記下沿路上的資訊，他只會記住目的地是什麼。我們把 `'w'` 變成 `'P'`，然後用他來查看。

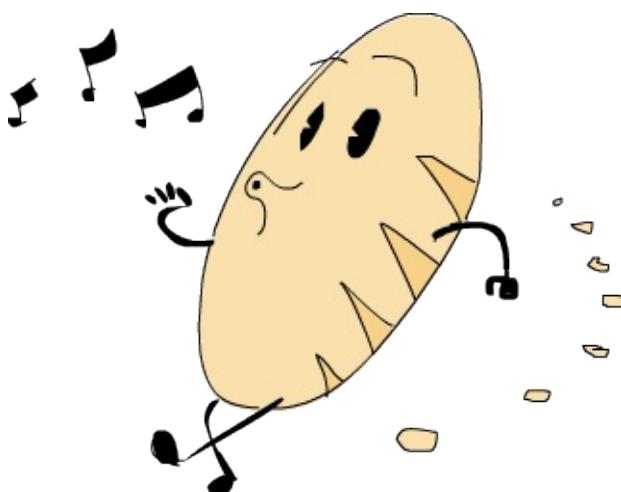
```
ghci> let newTree = changeToP [R,L] freeTree
ghci> elemAt [R,L] newTree
'P'
```

看起來運作正常。在這些函數裡面，包含方向的 list 比較像是一種"focus"，因為他特別指出了棵子樹。一個像 `[R]` 這樣的 list 是聚焦在 root 的右子樹。一個空的 list 代表的是主樹本身。

這個技巧看起來酷炫，但卻不太有效率，特別是在我們想要重複地改變內容的時候。假如我們有一個非常大的樹以及非常長的一串包含方向的 list。我們需要沿著方向從 root 一直走到樹的底部。如果我們想要改變一個鄰近的元素，我們仍需要從 root 開始走到樹的底部。這實在不太令人滿意。

在下一個章節，我們會介紹一個比較好的方法，讓我們可以有效率地改變我們的 focus。

## 凡走過必留下痕跡



我們需要一個比包含一串方向的 list 更好的聚焦的方法。如果我們能夠在從 root 走到指定地點的沿路上撒下些麵包屑，來紀錄我們的足跡呢？當我們往左走，我們便記住我們選擇了左邊，當我們往右走，便記住我們選擇了右邊。

要找個東西來代表我們的麵包屑，就用一串 `Direction` (他可以是 `L` 或者是 `R`)，只是我們叫他 `BreadCrumb` 而不叫 `Direction`。這是因為現在我們把這串 `direction` 反過來看了：

```
type Breadcrumbs = [Direction]
```

這邊有一個函數，他接受一棵樹跟一些麵包屑，並在我們往左走時在 list 的前頭加上 `L`

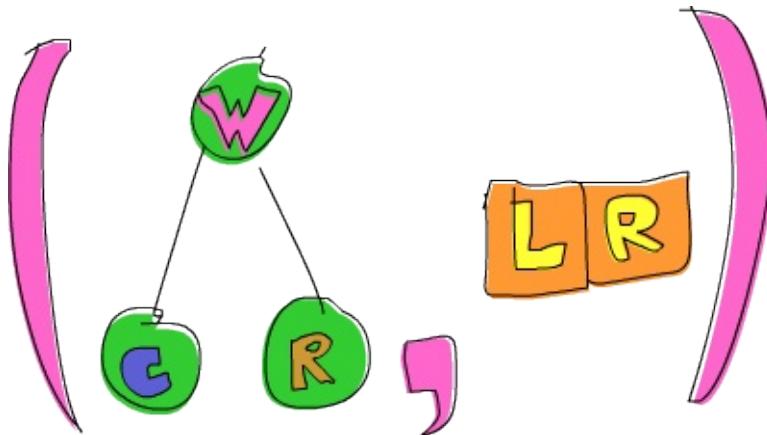
```
goLeft :: (Tree a, Breadcrumbs) -> (Tree a, Breadcrumbs)
goLeft (Node _ l _, bs) = (l, L:bs)
```

我們忽略 root 跟右子樹，直接回傳左子樹以及麵包屑，只是在現有的麵包屑前面加上 `L`。再來看看往右走的函數：

```
goRight :: (Tree a, Breadcrumbs) -> (Tree a, Breadcrumbs)
goRight (Node _ _ r, bs) = (r, R:bs)
```

幾乎是一模一樣。我們再來做一個先往右走再往左走的函數，讓他來走我們的 `freeTree`

```
ghci> goLeft (goRight (freeTree, []))
(Node 'W' (Node 'C' Empty Empty) (Node 'R' Empty Empty), [L, R])
```



現在我們有了一棵樹，他的 root 是 `'W'`，而他的左子樹的 root 是 `'C'`，右子樹的 root 是 `'R'`。而由於我們先往右走再往左走，所以麵包屑是 `[L, R]`。

要再表示得更清楚些，我們能用定義一個 `-:`

```
x -: f = f x
```

他讓我們可以將值餵給函數這件事反過來寫，先寫值，再來是 `-:`，最後是函數。所以我們可以寫成 `(freeTree, []) -: goRight` 而不是 `goRight (freeTree, [])`。我們便可以把上面的例子改寫地更清楚。

```
ghci> (freeTree, []) -: goRight -: goLeft
(Node 'W' (Node 'C' Empty Empty) (Node 'R' Empty Empty), [L, R])
```

## Going back up

如果我們想要往回上走回我們原來的路徑呢？根據留下的麵包屑，我們知道現在的樹是他父親的左子樹，而他的父親是祖父的右子樹。這些資訊並不足夠我們往回走。看起來要達到這件事情，我們除了單純紀錄方向之外，還必須把其他的資料都記錄下來。在這個案例中，也就是他的父親以及他的右子樹。

一般來說，單單一個麵包屑有足夠的資訊讓我們重建父親的節點。所以他應該要包含所有我們沒有選擇的路徑的資訊，並且他應該要紀錄我們沿路走的方向。同時他不應該包含我們現在鎖定的子樹。因為那棵子樹已經在 tuple 的第一個部份中，如果我們也把他紀錄在麵包屑裡，那就會有重複的資訊。

我們來修改一下我們麵包屑的定義，讓他包含我們之前丟掉的資訊。我們定義一個新的型態，而不用 `Direction`：

```
data Crumb a = LeftCrumb a (Tree a) | RightCrumb a (Tree a) deriving (Show)
```

我們用 `LeftCrumb` 來包含我們沒有走的右子樹，而不僅僅只寫個 `L`。我們用 `RightCrumb` 來包含我們沒有走的左子樹，而不僅僅只寫個 `R`。

這些麵包屑包含了所有重建樹所需要的資訊。他們像是軟碟一樣存了許多我們的足跡，而不僅僅只是方向而已。

大致上可以把每個麵包屑想像成一個樹的節點，樹的節點有一個洞。當我們往樹的更深層走，麵包屑攜帶有我們所有走過得所有資訊，只除了目前我們鎖定的子樹。他也必須紀錄洞在哪裡。在 `LeftCrumb` 的案例中，我們知道我們是向左走，所以我們缺少的便是左子樹。

我們也要把 `Breadcrumbs` 的 type synonym 改掉：

```
type Breadcrumbs a = [Crumb a]
```

接著我們修改 `goLeft` 跟 `goRight` 來紀錄一些我們沒走過的路徑的資訊。不像我們之前選擇忽略他。`goLeft` 像是這樣：

```
goLeft :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goLeft (Node x l r, bs) = (l, LeftCrumb x r:bs)
```

你可以看到跟之前版本的 `goLeft` 很像，不只是將 `L` 推到 list 的最前端，我們還加入 `LeftCrumb` 來表示我們選擇向左走。而且我們在 `LeftCrumb` 裡面塞有我們之前走的節點，以及我們選擇不走的右子樹的資訊。

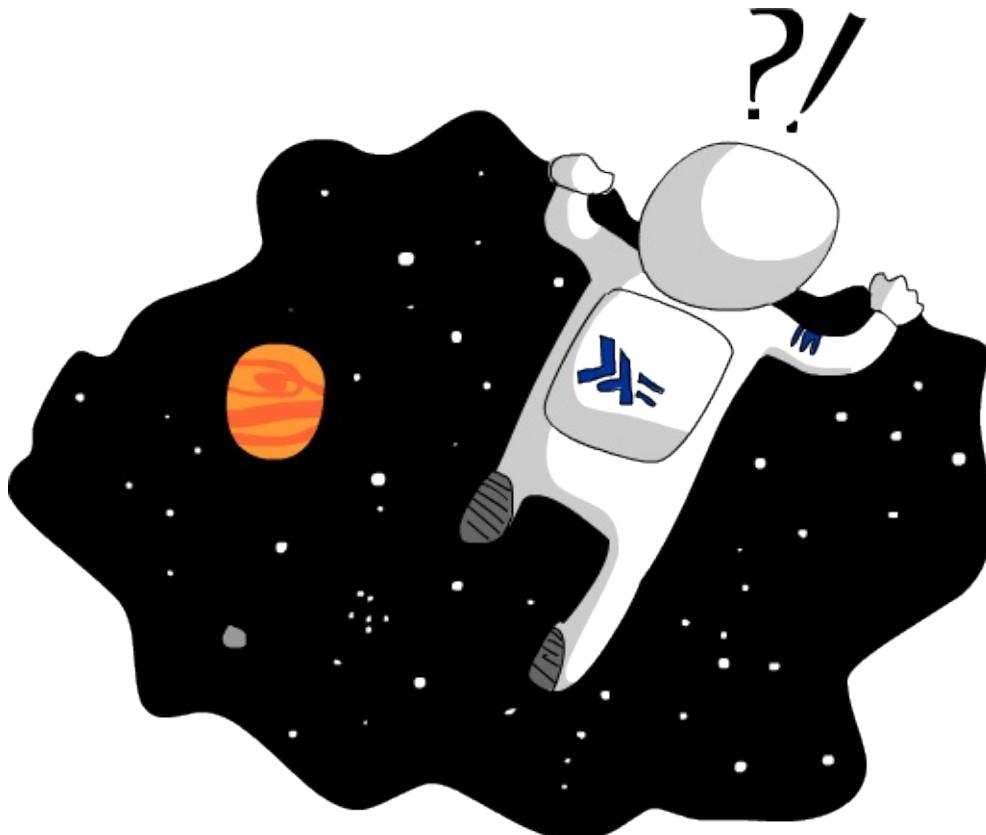
要注意這個函數會假設我們鎖定的子樹並不是 `Empty`。一個空的樹並沒有任何子樹，所以如果我們選擇在一個空的樹中向左走，就會因為我們對 `Node` 做模式匹配而產生錯誤。我們沒有處理 `Empty` 的情況。

`goRight` 也是類似：

```
goRight :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goRight (Node x l r, bs) = (r, RightCrumb x l:bs)
```

在之前我們只能向左或向右走，現在我們由於紀錄了關於父節點的資訊以及我們選擇不走的路的資訊，而獲得向上走的能力。來看看 `goUp` 函數：

```
goUp :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goUp (t, LeftCrumb x r:bs) = (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = (Node x l t, bs)
```



我們鎖定了 `t` 這棵樹並檢查最新的 `Crumb`。如果他是 `LeftCrumb`，那我們就建立一棵新的樹，其中 `t` 是他的左子樹並用關於我們沒走過得右子樹的資訊來填寫其他 `Node` 的資訊。由於我們使用了麵包屑的資訊來建立父子樹，所以新的 list 移除了我們的麵包屑。

如果我們已經在樹的頂端並使用這個函數的話，他會引發錯誤。等一會我們會用 `Maybe` 來表達可能失敗的情況。

有了 `Tree a` 跟 `Breadcrumbs a`，我們就有足夠的資訊來重建整棵樹，並且鎖定其中一棵子樹。這種方式讓我們可以輕鬆的往上，往左，往右走。這樣成對的資料結構我們叫做 Zipper，因為當我們改變鎖定的時候，他表現得很像是拉鍊一樣。所以我們便定義一個 type synonym：

```
type Zipper a = (Tree a, Breadcrumbs a)
```

我個人是比較傾向於命名成 `Focus`，這樣可以清楚強調我們是鎖定在其中一部分，至於 Zipper 被更廣泛地使用，所以這邊仍維持叫他做 zipper。

## Manipulating trees under focus

現在我們具備了移動的能力，我們再來寫一個改變元素的函數，他能改變我們目前鎖定的子樹的 root。

```
modify :: (a -> a) -> Zipper a -> Zipper a
modify f (Node x l r, bs) = (Node (f x) l r, bs)
modify f (Empty, bs) = (Empty, bs)
```

如果我們鎖定一個節點，我們用 `f` 改變他的 root。如果我們鎖定一棵空的樹，那就什麼也不做。我們可以移來移去並走到我們想要改變的節點，改變元素後並鎖定在那個節點，之後我們可以很方便的移上移下。

```
ghci> let newFocus = modify (\_ -> 'P') (goRight (goLeft (freeTree, [])))
```

我們往左走，然後往右走並將 root 取代為 `'P'`，用 `-:` 來表達的話就是：

```
ghci> let newFocus = (freeTree, []) -: goLeft -: goRight -: modify (\_ -> 'P')
```

我們也能往上走並置換節點為 `'X'`：

```
ghci> let newFocus2 = modify (\_ -> 'X') (goUp newFocus)
```

如果我們用 `-:` 表達的話：

```
ghci> let newFocus2 = newFocus -: goUp -: modify (\_ -> 'X')
```

往上走很簡單，畢竟麵包屑中含有我們沒走過的路徑的資訊，只是裡面的資訊是相反的，這有點像是要把襪子反過來才能用一樣。有了這些資訊，我們就不用再從 root 開始走一遍，我們只要把反過來的樹翻過來就好，然後鎖定他。

每個節點有兩棵子樹，即使子樹是空的也是視作有樹。所以如果我們鎖定的是一棵空的子樹我們可以做的事就是把他變成非空的，也就是葉節點。

```
attach :: Tree a -> Zipper a -> Zipper a
attach t (_, bs) = (t, bs)
```

我們接受一棵樹跟一個 zipper，回傳一個新的 zipper，鎖定的目標被換成了提供的樹。我們不只可以用這招把空的樹換成新的樹，我們也能把現有的子樹給換掉。讓我們來用一棵樹換掉我們 `freeTree` 的最左邊：

```
ghci> let farLeft = (freeTree,[]) -: goLeft -: goLeft -: goLeft -: goLeft
ghci> let newFocus = farLeft -: attach (Node 'Z' Empty Empty)
```

`newFocus` 現在鎖定在我們剛剛接上的樹上，剩下部份的資訊都放在麵包屑裡。如果我們用 `goUp` 走到樹的最上層，就會得到跟原來 `freeTree` 很像的樹，只差在最左邊多了 '`z`'。

## I'm going straight to top, oh yeah, up where the air is fresh and clean!

寫一個函數走到樹的最頂端是很簡單的：

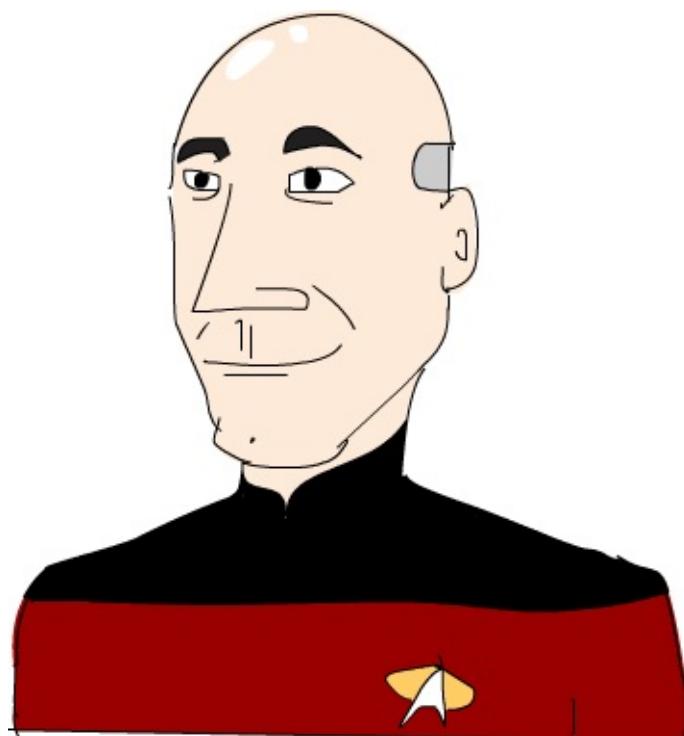
```
topMost :: Zipper a -> Zipper a
topMost (t,[]) = (t,[])
topMost z = topMost (goUp z)
```

如果我們的麵包屑都沒了，就表示我們已經在樹的 `root`，我們便回傳目前的鎖定目標。嘖然，我們便往上走來鎖定到父節點，然後遞迴地呼叫 `topMost`。我們現在可以在我們的樹上四處移動，呼叫 `modify` 或 `attach` 進行我們要的修改。我們用 `topMost` 來鎖定到 `root`，便可以滿意地欣賞我們的成果。

## 來看串列

Zippers 幾乎可以套用在任何資料結構上，所以聽到他可以被套用在 `list` 上可別太驚訝。畢竟，`list` 就是樹，只是節點只有一個兒子，當我們實作我們自己的 `list` 的時候，我們定義了下面的型態：

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
```



跟我們二元樹的定義比較，我們就可以看出我們把 list 看作樹的原則是正確的。

一串 list 像是 `[1, 2, 3]` 可以被寫作 `1:2:3:[]`。他由 list 的 head `1` 以及 list 的 tail `2:3:[]` 組成。而 `2:3:[]` 又由 `2` 跟 `3:[]` 組成。至於 `3:[]`，`3` 是 head 而 tail 是 `[]`。

我們來幫 list 做個 zipper。list 改變鎖定的方式分為往前跟往後（tree 分為往上，往左跟往右）。在樹的情形中，鎖定的部份是一棵子樹跟留下的麵包屑。那究竟對於一個 list 而言一個麵包屑是什麼？當我們處理二元樹的時候，我們說麵包屑必須代表 root 的父節點跟其他未走過的子樹。他也必須記得我們是往左或往右走。所以必須要有除了鎖定的子樹以外的所有資訊。

list 比 tree 要簡單，所以我們不需要記住我們是往左或往右，因為我們只有一種方式可以往 list 的更深層走。我們也不需要哪些路徑我們沒有走過的資訊。似乎我們所需要的資訊只有前一個元素。如果我們的 list 是像 `[3, 4, 5]`，而且我們知道前一個元素是 `2`，我們可以把 `2` 擺回 list 的 head，成為 `[2, 3, 4, 5]`。

由於一個單一的麵包屑只是一個元素，我們不需要把他擺進一個型態裡面，就像我們在做 tree zippers 時一樣擺進 `Crumb`：

```
type ListZipper a = ([a], [a])
```

第一個 list 代表現在鎖定的 list，而第二個代表麵包屑。讓我們寫一下往前跟往後走的函數：

```
goForward :: ListZipper a -> ListZipper a
goForward (x:xs, bs) = (xs, x:bs)

goBack :: ListZipper a -> ListZipper a
goBack (xs, b:bs) = (b:xs, bs)
```

當往前走的時候，我們鎖定了 list 的 tail，而把 head 當作是麵包屑。當我們往回走，我們把最近的麵包屑歛來然後擺到 list 的最前頭。

來看看兩個函數如何運作：

```
ghci> let xs = [1, 2, 3, 4]
ghci> goForward (xs, [])
([2, 3, 4], [1])
ghci> goForward ([2, 3, 4], [1])
([3, 4], [2, 1])
ghci> goForward ([3, 4], [2, 1])
([4], [3, 2, 1])
ghci> goBack ([4], [3, 2, 1])
([3, 4], [2, 1])
```

我們看到在這個案例中麵包屑只不過是一部分反過來的 list。所有我們走過的元素都被丟進麵包屑裡面，所以要往回走很容易，只要把資訊從麵包屑裡面撿回來就好。

這樣的形式也比較容易看出我們為什麼稱呼他為 Zipper，因為他真的就像是拉鍊一般。

如果你正在寫一個文字編輯器，那你可以用一個裝滿字串的 list 來表達每一行文字。你也可以加一個 Zipper 以便知道現在游標移動到那一行。有了 Zipper 你就很容易的可以新增或刪除現有的每一行。

## 陽春的檔案系統

理解了 Zipper 是如何運作之後，我們來用一棵樹來表達一個簡單的檔案系統，然後用一個 Zipper 來增強他的功能。讓我們可以在資料夾間移動，就像我們平常對檔案系統的操作一般。

這邊我們採用一個比較簡化的版本，檔案系統只有檔案跟資料夾。檔案是資料的基本單位，只是他有一個名字。而資料夾就是用來讓這些檔案比較有結構，並且能包含其他資料夾與檔案。所以說檔案系統中的元件不是一個檔案就是一個資料夾，所以我們便用如下的方法定義型態：

```
type Name = String
type Data = String
data FSItem = File Name Data | Folder Name [FSItem] deriving (Show)
```

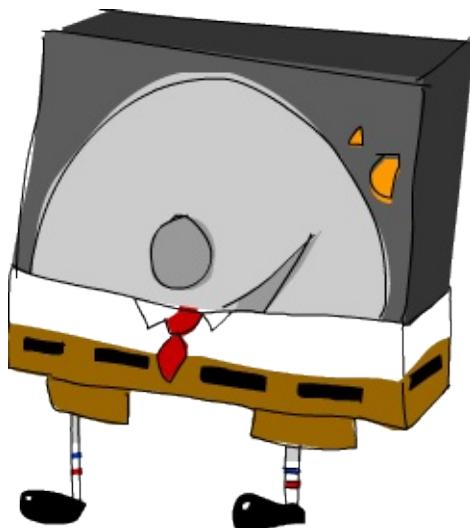
一個檔案是由兩個字串組成，代表他的名字跟他的內容。一個資料夾由一個字串跟一個 list 組成，字串代表名字，而 list 是裝有的元件，如果 list 是空的，就代表他是一個空的資料夾。

這邊是一個裝有些檔案與資料夾的資料夾：

```
myDisk :: FSItem
myDisk =
  Folder "root"
    [ File "goat_yelling_like_man.wmv" "baaaaaaa"
    , File "pope_time.avi" "god bless"
    , Folder "pics"
      [ File "ape_throwing_up.jpg" "bleargh"
      , File "watermelon_smash.gif" "smash!!"
      , File "skull_man(scary).bmp" "Yikes!"
      ]
    , File "dijon_poupon.doc" "best mustard"
    , Folder "programs"
      [ File "fartwizard.exe" "10gotofart"
      , File "owl_bandit.dmg" "mov eax, h00t"
      , File "not_a_virus.exe" "really not a virus"
      , Folder "source code"
        [ File "best_hs_prog.hs" "main = print (fix error)"
        , File "random.hs" "main = print 4"
        ]
      ]
    ]
```

這就是目前我的磁碟的內容。

## A zipper for our file system



我們有了一個檔案系統，我們需要一個 Zipper 來讓我們可以四處走動，並且增加、修改或移除檔案跟資料夾。就像二元樹或 list，我們會用麵包屑留下我們未走過路徑的資訊。正如我們說的，一個麵包屑就像是一個節點，只是他包含所有除了我們現在正鎖定的子樹的資訊。

在這個案例中，一個麵包屑應該要像資料夾一樣，只差在他缺少了我們目前鎖定的資料夾的資訊。為什麼要像資料夾而不是檔案呢？因為如果我們鎖定了一個檔案，我們就沒辦法往下走了，所以要留下資訊說我們是從一個檔案走過來的並沒有道理。一個檔案就像是一棵空的樹一樣。

如果我們鎖定在資料夾 "root"，然後鎖定在檔案 "dijon\_poupon.doc"，那麵包屑裡的資訊會是什麼樣子呢？他應該要包含上一層資料夾的名字，以及在這個檔案前及之後的所有項目。我們要的就是一個 Name 跟兩串 list。藉由兩串 list 來表達之前跟之後的元素，我們就完全可以知道我們目前鎖定在哪。

來看看我們麵包屑的型態：

```
data FSCrumb = FSCrumb Name [FSItem] [FSItem] deriving (Show)
```

這是我們 Zipper 的 type synonym：

```
type FSZipper = (FSItem, [FSCrumb])
```

要往上走是很容易的事。我們只要拿現有的麵包屑來組出現有的鎖定跟麵包屑：

```
fsUp :: FSZipper -> FSZipper
fsUp (item, FSCrumb name ls rs:bs) = (Folder name (ls ++ [item] ++ rs), bs)
```

由於我們的麵包屑有上一層資料夾的名字，跟資料夾中之前跟之後的元素，要往上走不費吹灰之力。

至於要往更深層走呢？如果我們現在在 "root"，而我們希望走到 "dijon\_poupon.doc"，那我們會在麵包屑中留下 "root"，在 "dijon\_poupon.doc" 之前的元素，以及在他之後的元素。

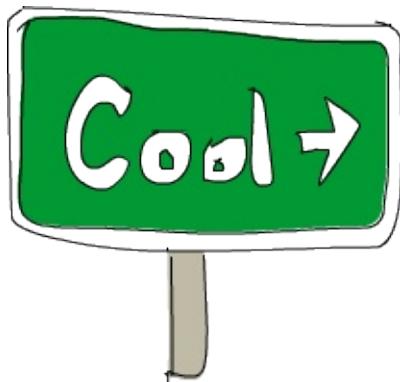
這邊有一個函數，給他一個名字，他會鎖定在在現有資料夾中的一個檔案：

```
import Data.List (break)

fsTo :: Name -> FSZipper -> FSZipper
fsTo name (Folder folderName items, bs) =
  let (ls, item:rs) = break (nameIs name) items
  in (item, FSCrumb folderName ls rs:bs)

nameIs :: Name -> FSItem -> Bool
nameIs name (Folder folderName _) = name == folderName
nameIs name (File fileName _) = name == fileName
```

`fsTo` 接受一個 `Name` 跟 `FSZipper`，回傳一個新的 `FSZipper` 鎖定在某個檔案上。那個檔案必須在現在身處的資料夾才行。這函數不會四處找尋這檔案，他只會看現在的資料夾。



首先我們用 `break` 來把身處資料夾中的檔案們分成在我們要找的檔案前的，跟之後的。如果記性好，`break` 會接受一個 `predicate` 跟一個 `list`，並回傳兩個 `list` 組成的 `pair`。第一個 `list` 裝有 `predicate` 會回傳 `False` 的元素，而一旦碰到一個元素回傳 `True`，他就把剩下的所有元素都放進第二個 `list` 中。我們用了一個輔助函數叫做 `nameIs`，他接受一個名字跟一個檔案系統的元素，如果名字相符的話他就會回傳 `True`。

現在 `ls` 一個包含我們要找的元素之前元素的 `list`。`item` 就是我們要找的元素，而 `rs` 是剩下的部份。有了這些，我們不過就是把 `break` 傳回來的東西當作鎖定的目標，來建造一個麵包屑來包含所有必須的資訊。

如果我們要找的元素不在資料夾中，那 `item:rs` 這個模式會符合到一個空的 `list`，便會造成錯誤。如果我們現在的鎖定不是一個資料夾而是一個檔案，我們也會造成一個錯誤而讓程式當掉。

現在我們有能力在我們的檔案系統中移上移下，我們就來嘗試從 root 走到

`"skull_man(scary).bmp"` 這個檔案吧：

```
ghci> let newFocus = (myDisk,[]) -: fsTo "pics" -: fsTo "skull_man(scary).bmp"
```

`newFocus` 現在是一個鎖定在 `"skull_man(scary).bmp"` 的 `Zipper`。我們把 `zipper` 的第一個部份拿出來看看：

```
ghci> fst newFocus
File "skull_man(scary).bmp" "Yikes!"
```

我們接著往上移動並鎖定在一個鄰近的檔案 `"watermelon_smash.gif"`：

```
ghci> let newFocus2 = newFocus -: fsUp -: fsTo "watermelon_smash.gif"
ghci> fst newFocus2
File "watermelon_smash.gif" "smash!!"
```

## Manipulating our file system

現在我們知道如何遍歷我們的檔案系統，因此操作也並不是難事。這邊便來寫個重新命名目前鎖定檔案或資料夾的函數：

```
fsRename :: Name -> FSZipper -> FSZipper
fsRename newName (Folder name items, bs) = (Folder newName items, bs)
fsRename newName (File name dat, bs) = (File newName dat, bs)
```

我們可以重新命名 "pics" 資料夾為 "cspi"：

```
ghci> let newFocus = (myDisk, []) -: fsTo "pics" -: fsRename "cspi" -: fsUp
```

我們走到 "pics" 這個資料夾，重新命名他然後再往回走。

那寫一個新的元素在我們目前的資料夾呢？

```
fsNewFile :: FSItem -> FSZipper -> FSZipper
fsNewFile item (Folder folderName items, bs) =
  (Folder folderName (item:items), bs)
```

注意這個函數會沒辦法處理當我們在鎖定在一個檔案卻要新增元素的情況。

現在要在 "pics" 資料夾中加一個檔案然後走回 root：

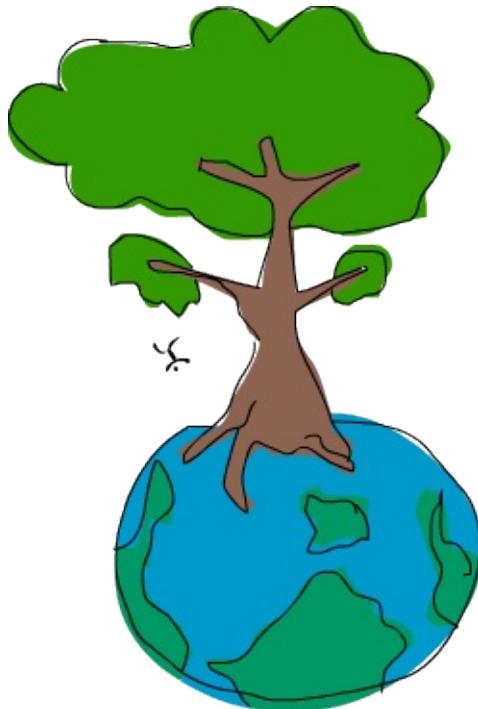
```
ghci> let newFocus = (myDisk, []) -: fsTo "pics" -: fsNewFile (File "heh.jpg" "lol") -: fsUp
```

當我們修改我們的檔案系統，他不會真的修改原本的檔案系統，而是回傳一份新的檔案系統。這樣我們就可以存取我們舊有的系統（也就是 `myDisk`）跟新的系統（`newFocus` 的第一個部份）使用一個 Zippers，我們就能自動獲得版本控制，代表我們能存取到舊的資料結構。這也不僅限於 Zippers，也是由於 Haskell 的資料結構有 `immutable` 的特性。但有了 Zipper，對於操作會變得更容易，我們可以自由地在資料結構中走動。

## 小心每一步

到目前為止，我們並沒有特別留意我們在走動時是否會超出界線。不論資料結構是二元樹，List 或檔案系統。舉例來說，我們的 `goLeft` 函數接受一個二元樹的 Zipper 並鎖定到他的左子樹：

```
goLeft :: Zipper a -> Zipper a
goLeft (Node x l r, bs) = (l, LeftCrumb x r:bs)
```



但如果我們走的樹其實是空的樹呢？也就是說，如果他不是 `Node` 而是 `Empty`？再這情況，我們會因為模式匹配不到東西而造成 runtime error。我們沒有處理空的樹的情形，也就是沒有子樹的情形。到目前為止，我們並沒有試著在左子樹不存在的情形下鎖定左子樹。但要走到一棵空的樹的左子樹並不合理，只是到目前為止我們視而不見而已。

如果我們已經在樹的 root 但仍舊試著往上走呢？這種情形也同樣會造成錯誤。。用了 Zipper 讓我們每一步都好像是我們的最後一步一樣。也就是說每一步都有可能會失敗。這讓你想起什麼嗎？沒錯，就是 Monad。更正確的說是 `Maybe` monad，也就是有可能失敗的 context。

我們用 `Maybe` monad 來加入可能失敗的 context。我們要把原本接受 Zipper 的函數都改成 monadic 的版本。首先，我們來處理 `goLeft` 跟 `goRight`。函數的失敗有可能反應在他們的結果，這個情況也不利外。所以來看下面的版本：

```
goLeft :: Zipper a -> Maybe (Zipper a)
goLeft (Node x l r, bs) = Just (l, LeftCrumb x r:bs)
goLeft (Empty, _) = Nothing

goRight :: Zipper a -> Maybe (Zipper a)
goRight (Node x l r, bs) = Just (r, RightCrumb x l:bs)
goRight (Empty, _) = Nothing
```

然後我們試著在一棵空的樹往左走，我們會得到 `Nothing`：

```
ghci> goLeft (Empty, [])
Nothing
ghci> goLeft (Node 'A' Empty Empty, [])
Just (Empty,[LeftCrumb 'A' Empty])
```

看起來不錯。之前的問題是我們在麵包屑用完的情形下想往上走，那代表我們已經在樹的 root。如果我們不注意的話那 `goUp` 函數就會丟出錯誤。

```
goUp :: Zipper a -> Zipper a
goUp (t, LeftCrumb x r:bs) = (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = (Node x l t, bs)
```

我們改一改讓他可以失敗得好看些：

```
goUp :: Zipper a -> Maybe (Zipper a)
goUp (t, LeftCrumb x r:bs) = Just (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = Just (Node x l t, bs)
goUp (_, []) = Nothing
```

如果我們有麵包屑，那我們就能成功鎖定新的節點，如果沒有，就造成一個失敗。

之前這些函數是接受 Zipper 並回傳 Zipper，這代表我們可以這樣操作：

```
ghci> let newFocus = (freeTree,[]) -: goLeft -: goRight
```

但現在我們不回傳 `zipper a` 而回傳 `Maybe (Zipper a)`。所以沒辦法像上面串起來。我們在之前章節也有類似的問題。他是每次走一步，而他的每一步都有可能失敗。

幸運的是我們可以從之前的經驗中學習，也就是使用 `>>=`，他接受一個有 `context` 的值（也就是 `Maybe (Zipper a)`），會把值餵進函數並保持其他 `context` 的。所以就像之前的例子，我們把 `-:` 換成 `>>=`。

```
ghci> let coolTree = Node 1 Empty (Node 3 Empty Empty)
ghci> return (coolTree,[]) >>= goRight
Just (Node 3 Empty Empty,[RightCrumb 1 Empty])
ghci> return (coolTree,[]) >>= goRight >>= goRight
Just (Empty,[RightCrumb 3 Empty,RightCrumb 1 Empty])
ghci> return (coolTree,[]) >>= goRight >>= goRight >>= goRight
Nothing
```

我們用 `return` 來把 Zipper 放到一個 `Just` 裡面。然後用 `>>=` 來餵到 `goRight` 的函數中。首先我們做了一棵樹他的左子樹是空的，而右邊是有兩顆空子樹的一個節點。當我們嘗試往右走一步，便會得到成功的結果。往右走兩步也還可以，只是會鎖定在一棵空的子樹

上。但往右走三步就沒辦法了，因為我們不能在一棵空子樹上往右走，這也是為什麼結果會是 `Nothing`。

現在我們具備了安全網，能夠在出錯的時候通知我們。

我們的檔案系統仍有許多情況會造成錯誤，例如試著鎖定一個檔案，或是不存在的資料夾。剩下的就留作習題。

# FAQ



## 我能把這份教學放在我的網站嗎？我可以更改裡面的內容嗎？

當然。它受[creative commons attribution noncommercial blah blah blah...](#)許可證的保護，因此你可以分享或修改它。只要你是發自內心想要這麼作，並且只將教學用於非商業目的。

## 推薦其它幾個 Haskell 讀物？

有很多出色的教學，但我得告訴你[Real World Haskell](#)出類拔萃。它太棒了。我覺得它可以和這個教程搭配着一起讀。這個教程提供幾個簡單例子好讓初學者對那點概念有個簡單認識，而 Real World Haskell 真正教給你如何充分地使用它們。

## 我怎麼聯絡到你？

最好的方式是用寄封 email 紿我，email 是 bonus at learnyouahaskell dot com。不過有點耐心，我沒有 24 小時在電腦旁，所以如果我沒有即時地回覆你的 email 的話，請多包含。

## 我想要一些習題！

很快就會有了！很多人不斷來問我能不能加些習題，我正在趕工呢。

## 關於作者

我的名字是 Miran Lipovača，住在斯洛文尼亞的 Ljubljana。大部分時間我都閒閒沒事，不過當我認真的時候我不是在寫程式，繪畫，練格鬥技，或彈 bass。我甚至弄了一個討論 bass 的網站。我甚至蒐集一堆貓頭鷹玩偶，有時候我會對他們自言自語。

## 關於簡體譯者

我的名字是[Fleurer](#)，在淄博的山東理工大學讀書。電子郵件是 me.sword at gmail dot com

## 關於繁體譯者

我是[MnO2](#), Haskell 愛好者,

# Resource

網路上 Haskell 的資源雖不少，但由於目前社群的人力有限。所以比較沒能整理成一套能循序漸進的學習方式。常常會在 Haskell Wiki 上撞到對初學者太過於深入的東西。或是覺得奇怪怎麼不斷有之前沒看過的東西冒出來。造成學習 Haskell 很大的撞牆期。這邊譯者會漸漸補充一些自己覺得有用的資源，嘗試找到一些中階的教材能夠銜接初學跟進階。

## Specification

- [Haskell 98 Report](#): Haskell 的標準，目前 GHC 如果不用任何 Extension，寫出來的程式是符合 Haskell 98 的標準。
- [Haskell 2010 Report](#): 最新的標準，有許多已經實作但要開 Extension 才能用。

## Tools

- [Hoogle](#): Haskell 函數的搜尋引擎，不只可以用函數的名稱搜尋，也可以用函數的型態來搜尋。
- [Hayoo](#): 跟 Hoogle 同樣功能。
- [hdiff](#): 可以方便查詢 package 不同版號之間的差異。
- [packdeps](#): 方便查詢 Hackage 上面 package 之間的相依性。

## Lectures & Articles

- [Wikibook Haskell](#): 豐富的 Wikibook 資源
- [CS240h](#): David Mazières 跟 Bryan O'Sullivan 在 Stanford 開的課。
- [本物のプログラマはHaskellを使う](#): Haskell 專欄
- [Write Yourself a Scheme in 48 Hours](#), Audrey Tang 寫的教學，教你如何用 Haskell 寫出一個 Scheme。
- [德國大學的 Functional Programming 課程](#), 語言是用 FP (英文授課) .HD\_Videoaufzeichnung)
- [Simon Marlow 講解 parallel haskell 的投影片](#)
- [FLOLAC 2012](#)
- [ICFP 2012](#)
- [Explanation of Generalized Algebraic Data Types](#)
- [A Quick Intro to Snap](#)
- [Logic, Languages, Compilation, and Verification 2012](#)

- [Haskell in Halle/Saale](#)
- [Fast Code Nation](#))

## Forum

- [Stackoverflow](#): 著名 stackoverflow 上的\*haskell tag
- [Reddit](#)

## Online Judge

- [H-99: Ninety-Nine Haskell Problems](#)
- [Project Euler](#): 已經算非常著名的 Online Judge, 可惜只有上傳答案。如果問題實在想不出來, Haskell Wiki 上也有參考答案。
- [SPOJ](#): 少數的 Online Judge 系統可以上傳 Haskell 的, 題目非常豐富。也是練 ACM ICPC 常用的網站。

## Books

- [Learn you a Haskell for great good \(Japanese Translation\)](#)
- [Real World Haskell](#)
- [Yesod Book](#), 講解如何使用 Yesod Web Framework

## PL Researchers

- 穆信成老師
- 單中杰老師
- Conal Elliott
- Edward Yang
- Edward Kmett

## Interesting Projects

- [Fay Programming Langauge](#) 用 Haskell 語言的子集, 直接轉譯成 Javascript
- [Leksah](#): Haskell IDE
- [Super Manao Bros](#): 超級瑪利歐

# Taiwan Functional Programming User Group

- [TW-FPUG on Vimeo](#)
- [Haskell 進階運算元件介紹](#)