# Learn Me a Haskell

Jacob Bishop

2023-07-01

# 1 Introduction

PASS

# 2 Starting Out

## 第一个函数

在 `./test/` 文件夹下创建一个 `baby.hs` 的文件，写入：

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

使用 `ghci` 加载该文件（在本项目根目录时使用 `:l tests/baby` ）：

```
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
```

一个带有 `if` 的函数：

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

Haskell 中的 `if` 声明是一个 *表达式*，那么 `else` 是强制性的，因为表达式一定要有所返回。因此加上述函数可以改写为：

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

这里的 `'` 符号是 Haskell 中的有效字符，且在 Haskell 中并没有特殊的意义，因此可以用作于函数名。通常情况下，使用 `'` 代表着一个函数（非懒加载的函数）的严格版本，或是一个有细微变化的函数或者变量。又因为 `'` 是一个有效字符，那么可以创建以下函数：

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

这里又有两点值得注意的地方。首先，函数名不能以大写开头，稍后会进行说明；其次，该函数并没有任何入参。当一个函数没有入参，我们通常称其为一个 *定义 definition*，因为一旦定义了它便不能修改其名称，以及其返回。

## list 的介绍

Haskell 中的 list 是 **同质的 homogenous** 数据结构。

**Note**

在 `GHCI` 中可以使用 `let` 关键字定义一个名称。换言之，`GHCI` 中的 `let a = 1` 等同于脚本中的 `a = 1` 。

通常使用 `++` 操作符将两个数组进行合并：

```
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"
```

可以使用 `:` 操作符将元素直接添加至数组头部：

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

实际上，`[1, 2, 3]` 是 `1:2:3:[]` 的语法糖，其中 `[]` 为一个空数组。如果头部追加 `3` ，`[]` 就变成了 `[3]` ，再次进行头部追加 `2` ，则变为 `[2, 3]` ，以此类推。

如果希望通过索引获取数组中的元素，那么可以使用 `!!` 操作符：

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

超出索引时则会报错。

数组还可以通过操作符 `<` ，`<=` ，`==` ，`>` 以及 `>=` 操作符来进行比较，而比较的方式则是顺序比较。当进行头部比较元素相等时，再进行下一个元素进行比较。

数组的四种基础操作 `head` ，`tail` ，`last` 以及 `init` ：

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

当使用上述四种操作时，需要注意是否应用于空数组，这样的错误在编译期并不能被发现。

其它的操作：

1. `length` 获取数组长度；
2. `null` 检查数组是否为空；
3. `reverse` 翻转数组；
4. `take` 获取数组的头几个元素的数组；

5. `drop` 移除数组的头几个元素，并返回剩余元素的数组；
6. `maximum` 获取最大值；
7. `minimum` 获取最小值；
8. `sum` 求和；
9. `product` 求积；
10. `elem` 元素是否存在于数组中。

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

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

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

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]
[]

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]
[]

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

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

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

## Texas 排列

```
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"
```

带有 step 的排列：

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

而对于浮点数的排列需要注意精度问题：

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

以下是若干用于生产无限长度数组的函数：

`cycle` 循环周期：

```
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` 重复：

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

另外就是 `replicate` 函数可以重复单个元素：

```
ghci> replicate 3 10
[10,10,10]
```

## 列表表达式

数学里的 集合表达式 *set comprehensions* 例如 $S = 2 \cdot x | x \in \mathbb{N}, x \leq 10$；Haskell 中的列表表达式，例如 1 至 10 数组中每个元素乘以 2：

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

为列表表达式添加条件（或称谓语 predicate）：

```
ghci> [x*2 | x <- [1..10], x*2 >= 12]
[12,14,16,18,20]
ghci> [ x | x <- [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]
```

将列表表达式置于一个函数中便于复用：

```
ghci> boomBangs xs = [ if x < 10 then "BOOM!" else "BANG!" | x <- xs, odd x]
ghci> boomBangs [7..13]
["BOOM!","BOOM!","BANG!","BANG!"]
```

多个谓语也是可以的：

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

除此之外，还可以处理若干数组：

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

当然也可以加上谓语：

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

那么对于字符串也可以使用列表表达式：

```
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"]
```

现在让我们编写一个自己的 `length` ，命名 `length'` （这里的 `_` 意为无需使用的变量）：

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

由于字符串是数组，因此我们可以使用列表表达式处理并生产字符串。以下是一个移除所有字符但保留大写字符的函数：

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

## 元组

在某种程度上，元组类似于数组 – 存储若干值至单个变量上。然而有一些基础的差异：数组长度可以无限，元组长度固定；数组中元素类型是同质的，而元组则可以是异质的 heterogenous。

对于对元组（当且仅当包含两个元素）有以下操作：

`fst` 获取对元组的第一个元素：

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

`snd` 获取对元组的第二个元素：

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

另外一个有意思的函数则是 `zip`，它可以将两个数组按对拼接成对元组的数组

```
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")]
```

当两个数组的长度不一时，`zip` 则按最短的那个进行对齐，长的数组剩余部分则被丢弃，这是因为 Haskell 是懒加载的缘故。

```
ghci> zip [5,3,2,6,2,7,2,5,4,6,6] ["im","a","turtle"]
[(5,"im"),(3,"a"),(2,"turtle")]
ghci> zip [1..] ["apple", "orange", "cherry", "mango"]
[(1,"apple"),(2,"orange"),(3,"cherry"),(4,"mango")]
```

# 3 Types and Typeclasses

## 相信类型

通过 `:t` 命令可以得知类型：

```
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
```

函数同样拥有类型：

```
ghci> :t doubleSmallNumber
doubleSmallNumber :: (Ord a, Num a) => a -> a
```

而对于有多个入参的函数而言：

```
ghci> addThree x y z = x + y + z
ghci> :t addThree
addThree :: Num a => a -> a -> a -> a
```

参数由 `->` 符分开，并且入参与返回的类型并无差异，之后我们讨论到为什么是由 `->` 分割而不是 `Int, Int, Int -> Int` 或者其他样式的类型。

接下来是一些常规的类型：

`Int` ：对于 32 位的机器而言最大值大概是 2147483647 而最小值则是 2147483647。

`Integer` ：同样也是整数，只不过范围会大很多，而 `Int` 则更高效率。

`Float` ：单精度。

`Double` ：双精度。

`Bool` ：布尔值。

`Char` ：字符。

## 类型变量

那么 `head` 函数的类型是什么呢?

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

这里的 `a` 则是一个 **类型变量 type variable**，这意味着 `a` 可以是任意类型。这非常像其他语言的泛型，但唯有在 Haskell 中它更为强大，因为它允许我们可以轻易的编写通用的函

数，且不使用任何特定的行为的类型。带有类型变量的函数也被称为 **多态函数 polymorphic functions**。

## Typeclasses 101

typeclass 类似于一个接口用于定义一些行为。如果一个类型是 typeclass 的一部分，这就意味着它支持并且实现了 typeclass 中所描述的行为。

那么 `==` 函数的类型签名是什么呢?

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

**Note**

`==` 操作符是一个函数，`+` ，`*` ，`-` ，`/` 以及其它的操作符也都是。如果一个函数只包含特殊字符，那么默认情况下它被认做是一个中置函数。如果想要检查它的类型，将其传递给另一个函数或作为前缀函数调用它，那么则需要用括号将其包围。

这里有趣的是 `=>` 符号，在该符号之前的被称为一个 **类约束 class constraint**。那么上述的类型声明可以被这么理解：等式函数接受任意两个相同类型的值，并返回一个 `Bool` ，而这两个值必须是 `Eq` 类的成员（即类约束）。

`Eq` typeclass 提供了一个用于测试是否相等的接口。

而 `elem` 函数则拥有 `(Eq a) => a -> [a] -> Bool` 这样的类型，因为其在数组中使用了 `==` 用于检查元素是否为期望的值。

一些基础的 typeclasses：

`Eq` 如上所述。

`Ord` 覆盖了所有标准的比较函数例如 `>` ，`<` ，`>=` 以及 `<=` 。`compare` 函数接受两个同类型的 `Ord` 成员，并返回一个 ordering。`Ordering` 是一个可作为 `GT` ，`LT` 或是 `EQ` 的类型，分别意为 大于，小于以及等于。

`Show` 可以表示为字符串。

`Read` 有点类似于 `Show` 相反的 typeclass，`read` 函数接受一个字符串并返回一个 `Read` 的成员。

```
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]
```

但是如果尝试一下 `read "4"` 呢?

```
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)
```

这里 GHCI 告知它并不知道想要返回什么，通过检查 `read` 的类型签名:

```
ghci> :t read
read :: Read a => String -> a
```

也就是说其返回的是 `Read` 所约束的类型，那么如果在之后没有使用到它，则没有办法知晓其类型。我们可以显式的使用**类型注解**，即在表达式后面加上 `::` 与指定的一个类型:

```
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')
```

大多数表达式可以被编译器推导出其类型，但是有时编译器并不知道返回值的类型，例如 `read "5"` 时该为 `Int` 还是 `Float` 。那么为了知道其类型，Haskell 会解析 `read "5"` 。然而 Haskell 是一个静态类型语言，因此它需要在代码编译前知道所有类型。

`Enum` 成员是序列化的有序类型 – 它们可被枚举。`Enum` typeclass 的主要优势是可以使用在列表区间; 它们也同样定义了 successors 与 predecessors，即可使用 `succ` 以及 `pred` 函数。在这个类中的类型有: `()` , `Bool` `Char` , `Ordering` , `Int` , `Integer` , `Float` 以及 `Double` .

`Bounded` 成员拥有上下界:

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

元组的成员如果为 `Bounded` 的一部分，那么元组也是:

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

`Num` 是一个数值类:

```
ghci> :t 20
20 :: (Num t) => t
ghci> 20 :: Int
20
ghci> 20 :: Integer
20
ghci> 20 :: Float
20.0
ghci> 20 :: Double
20.0
```

这些类型都在 `Num` typeclass 内。如果检查 `*` 的类型，将会看到:

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

`Integral` 同样也是数值 typeclass。

`Floating` 包含 `Float` 与 `Double` 。

`fromIntegral` 是一个处理数值的常用函数,而其类型为 `fromIntegral :: (Num b, Integral a) => a`

。

# 4 Syntax in Functions

## 模式匹配

一个简单案例:

```
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"
```

一个递归案例:

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

模式匹配也可以失败:

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

当输入并不是期望时:

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

即出现了非穷尽的匹配，因此我们总是需要捕获所有模式。

模式匹配也可用作于元组:

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

模式匹配也可作用于列表表达式:

```
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]
```

**Note**

`x:xs` 模式的使用很常见，特别是递归函数。但是包含 `:` 的模式只匹配长度为 1 或更多的数组。

如果希望获取前三个元素以及数组剩余元素，那么可以使用 `x:y:z:zs` ，那么这样仅匹配有三个或以上的元素的数组。

其它案例：

```
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 " ++ show
    y
```

该函数是安全的，因为它考虑到了空数组，以及若干元素数组的情况。

之前通过列表表达式编写了 `length` 函数，现在可以通过模式匹配再加上递归的方式实现一遍：

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

接下来是实现 `sum` ：

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

同样还有一种被称为 *as* 模式的，即在模式前添加名称以及 `@` 符号，例如 `xs@(x:y:ys)` ，该模式将匹配 `x:y:ys` ，同时用户可以轻易的通过 `xs` 来获取整个数组，而无需重复使用 `x:y:ys` 进行表达：

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

最后，用户在模式匹配中不能使用 `++` 符号。

## 守护!

守护是一种检测值的某些属性是否为真，看上去像是 `if` 语句，但是其可读性更强：

```
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
  | bmi <= 18.5 = "You're underweight, you emo, you!"
```

```
    | bmi <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise = "You're a whale, congratulations!"
```

守护是由管道符并接着一个函数名以及函数参数进行定义的。守护本质上就是一个布尔表达式，如果为 `True` ，那么其关联的函数体被执行；如果为 `False` ，那么检查则会移至下一个守护，以此类推。

大多数时候最后一个守护是 `otherwise` ，其被简单的定义为 `otherwise = True` 并捕获所有情况。

当然我们可以使用任意参数的函数来守护：

```
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. Pffft, I bet you're ugly!"
  | weight / height ^ 2 <= 30.0 = "You're fat! Lose some weight, fatty!"
  | otherwise = "You're a whale, congratulations!"
```

另外我们可以实现自己的 `max` 与 `compare` 函数：

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

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

**Note**

我们不仅可以通过引号来调用函数，也可以使用引号来定义他们，有时这样会更加便于阅读。

## Where!?

上一节的 `bmiTell'` 函数中的 `weight / height ^ 2` 被重复了三遍，可以只计算一次并通过名称来绑定计算结果：

```
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. Pffft, 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
```

我们在守护的结尾添加了 `where` 并定义了 `bmi` 这个名称，这里定义的名称对整个守护可见，这样就无需再重复同样代码了。那么我们可以进行更多的定义：

```
bmiTell''' :: (RealFloat a) => a -> a -> String
bmiTell''' weight height
  | bmi <= skinny = "You're underweight, you emo, you!"
  | bmi <= normal = "You're supposedly normal. Pffft, 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
```

当然我们可以通过**模式匹配**来进行变量绑定！上面 `where` 中的代码可以改写为：

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

现在让我们编写另一个相当简单的函数用作获取名字首字母：

```
initials :: String -> String -> String
initials first_name last_name = [f] ++ ". " ++ [l] ++ "."
  where
    (f : _) = first_name
    (l : _) = last_name
```

我们可以直接将模式匹配应用于函数参数。

另外，正如我们可以在 `where` 块中定义约束，我们也可以定义函数：

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

`where` 绑定也可以是嵌套的，这在编写函数中很常见：定义一些辅助函数在函数 `where` 子句，然后这些函数的辅助函数又在其自身的 `where` 子句中。

### Let 的用法

与 `where` 绑定很相似的是 `let` 绑定。前者是一个语法构造器用于在函数的尾部进行变量绑定，这些变量可供整个函数使用，包括守护；而后者则是在任意处绑定一个变量，其自身为

表达式，不过只在作用域生效，因此不能被守护中访问。与 Haskell 其他任意的构造一样，`let` 绑定也可使用模式匹配：

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

这里的结构是 `let <bindings> in <expression>` 。在 *let* 中定义的名称可以在 *in* 之后的表达式中访问。这里同样要注意缩进。现在看来 `let` 仅仅将绑定提前，与 `where` 的作用无异。

不同点在于 `let` 绑定是表达式自身，而 `where` 仅为语法构造。还记得之前提到过的 `if else` 语句是表达式，可以在任意处构造：

```
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
```

那么 `let` 绑定也可以：

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

同样可以在当前作用域引入函数：

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

如果想要绑定若干变量，我们显然不能再列上对齐它们。这就是为什么需要用分号进行分隔：

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

正如之前提到的，可以将模式匹配应用于 *let* 绑定：

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

当然也可以将 *let* 绑定置入列表表达式中：

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

将 `let` 置入列表表达式中类似于一个子句，不过它不会对列表进行筛选，而仅仅绑定名称。该名称可被列表表达式的输出函数可见（即在符号 `|` 前的部分），以及所有的子句，以及绑定后的部分。因此我们可以让函数继续进行筛选：

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

我们不能在 `(w, h) <- xs` 中使用 `bmi` ，因为它在 *let* 绑定之前。

在列表表达式中使用 *let* 绑定可以省略 *in* 的那部分，这是因为名称的可视范围已经被预定义好了。不过我们还是可以在一个子句中使用 *let in* 绑定，该名称仅可在该子句中可见。在 GHCi 中定义函数与常数时，*in* 部分同样也可以省略。如果这么做了，那么该名称可以被整个交互过程中可见。

```
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'
```

## Case 表达式

以下两端代码表达的是同样一件事，它们互为可替换的。

```
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
```

正如所见的那样，case 表达式的语法特别简单：

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

`expression` 与模式匹配。模式匹配的行为正如预期那样：首个匹配上表达式的那个模式将被使用。如果直到最后都没有合适的模式被找到，那么将会抛出运行时错误。

函数参数的模式匹配只能在定义函数时完成，而 case 表达式则可以在任意处使用。例如：

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

case 表达式可以对表达式中间的模型内容进行模式匹配。函数定义中的模式匹配是 case 表达式的语法糖，因此我们也可以这样定义：

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

# 5 Recursion

## 你好递归!

递归对于 Haskell 而言很重要，因为不同于其他命令式语言，Haskell 中的计算是通过声明某物，而不是声明如何获取。这就是为什么 Haskell 中没有 while 循环以及 for 循环，取而代之的则是递归。

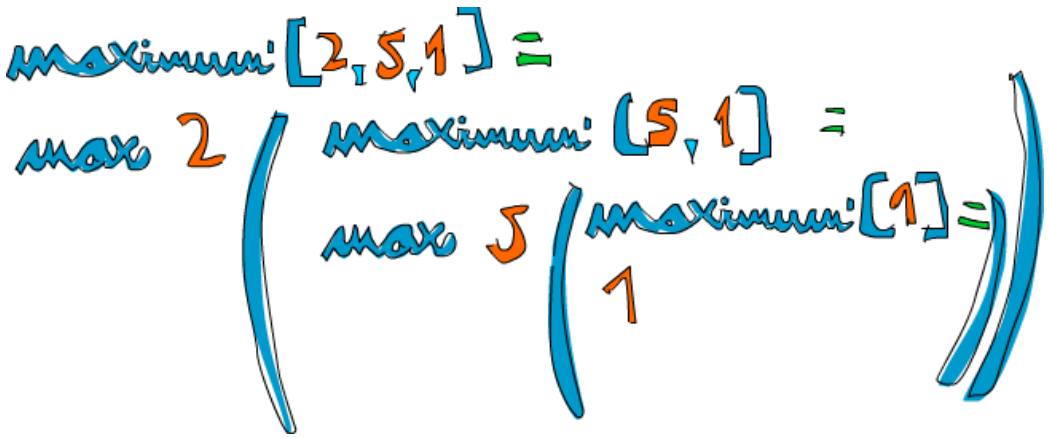
## Maximum

`maximum` 函数接受一组可排序的列表（例如 `Ord` typeclass 的实例），并返回它们之间最大的那个。

现在让我们看一下如何递归的实现这个函数。我们首先可以确立一个边界条件，同时声明该列表的最大值等同于列表中的唯一元素，接着声明如果头大于尾时长列表的头是最大值，如果尾更大那么继续上述过程：

```haskell
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
```

如上所示，模式匹配与递归非常的相配！大多数命令式语言并没有模式匹配，因此需要编写一堆 if else 声明来测试边界条件。而 Haskell 中仅需令它们成为模版。这里使用了 *where* 绑定来定义 `maxTail` 作为列表尾的最大值。



### 更多的递归函数

让我们使用递归再来实现一些函数。首先是 `replicate` ，接受一个 `Int` 以及一些元素，返回一个列表拥有若干重复的元素。

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

这里使用守护而不是模式是因为需要测试一个布尔值条件。

> **Note**
>
> `Num` 并不是 `Ord` 的子类，也就是说一个数值的组成并不依赖于排序。这就是为什么在做加法或减法或比较时，需要同时指定 `Num` 与 `Ord` 的类约束。

接下来是实现 `take` ：

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

注意这里使用了 `_` 来匹配列表，因为我们并不关心列表里面的情况；同时，我们使用了一个守护，但是并没有 `otherwise` 部分，这意味着如果 `n` 大于 0 的情况下，匹配将会失败并跳转到下一个匹配。第二个匹配指明如果尝试从空列表中提取任何元素，返回空列表。第三个模式将一个列表分割成一个头与一个尾，接着将从一个列表中获取 `n` 个元素相等于拥有 `x` 头与一个尾视作一个列表获取 `n-1` 元素。

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

接下来是 `reverse` 函数：

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

由于 Haskell 支持无线列表， `reverse` 并没有一个真正的边界检查，但是如果不这么做，那么则会一直计算下去或者生产出一个无限的数据结构，类似于无限列表。无限列表的好处是我们可以在任意处进行截断。然后是 `repeat` 函数，其返回一个无限列表：

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

接下来是 `zip` 函数:

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

最后一个是 `elem` 函数:

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

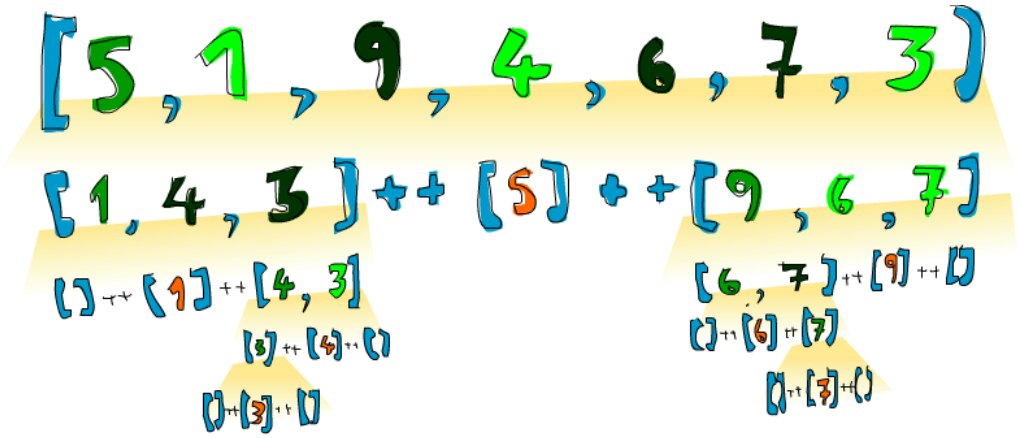
### 快排!

这里是主要算法：**排序列表是这样的一个列表，它包含所有小于（或等于）前面的列表头的值（这些值都是排序过的），然后是中间的列表头然后是所有大于列表头的值（它们也是排序过的）**。注意定义中两次提到了排序，那么我们将进行两次递归！同样也注意我们使用的是动词 *is* 在算法中进行定义，而不是做这个，做那个，再做另一个... 这就是函数式编程的魅力：

```
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
```

测试：

```
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"
"        abcdeeefghhijklmnoooopqrrsttuuvwxyz"
```



# 6 Higher Order Functions

## 柯里化函数

在 Haskell 中每个函数实质上仅接受一个参数。那么迄今为止定义的那么多函数是怎么接受多个参数的呢？这是就是**柯里化函数 curried functions**。

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

两个参数间用空格间隔就是简单的**函数应用 function application**。空格类似于一个操作符，其拥有最高的优先级。例如 `max` ，其签名为 `max :: (Ord a) => a -> a -> a` ，可以被重写为 `max :: (Ord a) => a -> (a -> a)` ，可以这么理解：`max` 接受一个 `a` 并返回（即 `->` ）一个函数，该函数接受一个 `a` 并返回一个 `a` 。这就是为什么返回值类型以及函数的参数都是由箭头符进行分隔的。

那么这样做有什么便利？简单来说如果调用一个仅几个参数的函数，我们得到的是一个**部分应用 partially applied** 的函数，即一个函数接受的参数与留下未填的参数一样多。

来观测一个简单的函数：

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

当使用 `multThree 3 5 9` 或者 `((multThree 3) 5) 9` 时到底发生了什么？首先，`3` 应用至 `multThree` ，因为它们由空格进行了分隔（最高优先级）。这就创建了一个接受一个参数的函数，并返回了一个函数。接下来 `5` 被应用至该函数，以此类推。记住我们的函数类型同样也可以重写成 `multThree :: (Num a) => a -> (a -> (a -> a))` 。接下来观察：

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

调用函数时输入不足的参数，实际上实在创造新的函数。那么如果希望创建一个函数接受一个值并将其与 `100` 进行比较呢？

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

如果带着 `99` 调用它，返回一个 `GT` 。注意 `x` 同时位于等式的右侧。那么调用 `compare 100` 返回的是什么呢？它返回一个接受一个数值参数并将其与 `100` 进行比较的函数。现在将其重写：

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

类型声明仍然相同，因为 `compare 100` 返回一个函数。`compare` 的类型是 `(Ord a) -> a -> (a -> Order`，带着 `100` 调用它返回一个 `(Num a, Ord a) => a -> Ordering` 。这里额外的类约束溜走了，这是因为 `100` 同样也是 `Num` 类的一部分。

中置函数同样可以通过使用分割被部分应用。要分割中置函数，只需将其用圆括号括起来，并只在一侧提供参数：

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

调用 `divideByTen 200` 等同于 `200 / 10` ，等同于 `(/10) 200` 。

那么如果在 GHCI 中尝试 `multThree 3 4` 而不是通过 *let* 将其与名称绑定，或是将其传递至另一个函数呢？

```
ghci> multThree 3 4
<interactive>:1:0:
    No instance for (Show (t -> t))
      arising from a use of `print' at <interactive>: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 会提示我们表达式生成了一个类型为 `a -> a` 的函数，但是并不知道该如何将其打印至屏幕。函数并不是 `Show` typeclass 的实例，因此我们并不会得到一个函数的展示。

## 来点高阶函数

函数可以接受函数作为其参数，也可以返回函数。

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

首先注意的是类型声明。之前我们是不需要圆括号的，因为 `->` 是自然地右结合。然而在这里却是强制性的，它们表明了第一个参数是一个接受某物并返回某物的函数，第二个参数同上所述。我们可以用柯里化函数的方式来进行解读，不过为了避免头疼，我们仅需要说该函数接受两个参数并返回一个值。这里第一个参数是一个函数（即类型 `a -> a` ），而第二个参数则是 `a` 。

函数体非常的简单，仅需要使用参数 `f` 作为一个函数，通过一个空格将 `x` 应用至其，接着再应用一次 `f` 。

```
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]
```

可以看到单个高阶函数可以被用以多种用途。而在命令式编程中，通常使用的是 for 循环、while 循环、将某物设置为一个变量、检查其状态等等，为了达到某些行为，还需要用接口将其封装，类似于函数；而函数式编程则使用高阶函数来抽象出相同的模式。

现在让我们实现一个名为 `flip` 的标准库已经存在的函数，其接受一个函数并返回一个类似于原来函数的函数，仅前两个参数被翻转。简单的实现：

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

观察类型声明，`flip'` 接受一个函数，该函数接受一个 `a` 与 `b`，并返回一个函数，该返回的函数接受一个 `b` 与 `a`。然而默认情况下函数是柯里化的，第二个圆括号是没有必要的，因为 `->` 默认是右结合的。`(a -> b -> c) -> (b -> a -> c)` 等同于 `(a -> b -> c) -> (b -> (a -> c))`，等同于 `(a -> b -> c) -> b -> a -> c`。我们可以用更简单方式来定义该函数：

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

这里我们利用了函数都是柯里化的便利。当不带参数 `y` 与 `x` 时调用 `flip'' f` 时，它将返回一个 `f`，该函数接受两个参数，只不过它们的位置是翻转的。

```
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]
```

## Maps & filters

`map` 接受一个函数以及一个列表，将该函数应用至列表中的每一个元素中，生产一个新的列表。让我们来看一下类型签名：

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

测试：

```
ghci> map (+3) [1,3,5,1,6]
[4,6,8,4,9]
```

```
ghci> map (-1) [1,3,5,1,6]

<interactive>:2:1: error:
    • Could not deduce (Num a0)
        arising from a type ambiguity check for
        the inferred type for ‘it’
      from the context: (Num a, Num (a -> b))
        bound by the inferred type for ‘it’ :
                   forall {a} {b}. (Num a, Num (a -> b)) => [b]
        at <interactive>:2:1-20
      The type variable ‘a0’ is ambiguous
      These potential instances exist:
        instance Num Integer -- Defined in ‘GHC.Num’
        instance Num Double -- Defined in ‘GHC.Float’
        instance Num Float -- Defined in ‘GHC.Float’
        ...plus two others
        ...plus one instance involving out-of-scope types
        (use -fprint-potential-instances to see them all)
    • In the ambiguity check for the inferred type for ‘it’
      To defer the ambiguity check to use sites, enable AllowAmbiguousTypes
      When checking the inferred type
        it :: forall {a} {b}. (Num a, Num (a -> b)) => [b]
ghci> map (subtract 1) [1,3,5,1,6]
[0,2,4,0,5]
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]
```

原书上一章有提到过 `-1` 这样的情况，报错的原因是 Haskell 将 `-1` 识别为负数而不是减法，需要显式调用 `subtract` 才能识别为 partial 函数并应用至列表中的各个元素上。

`filter` 接受一个子句（该子句是一个函数，用于告知某物是否为真），以及一个列表，并返回满足该子句的元素列表。类型签名如下：

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

测试：

```
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"
```

将上一章的 `quicksort` 中的列表表达式替换为 `filter` ：

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

现在尝试一下**找到 100,100 以下最大能被 3829 的值**：

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

接下来尝试一下**找到所有奇数平方在 10,000 以下的和**，不过首先要介绍一下 `takeWhile` 函数。该函数接受一个子句以及一个列表，接着从列表头向后遍历，在子句返回真时返回元素，一旦子句返回假则结束遍历。可以在 GHCI 上用一行来完成任务：

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

当然也可以用列表表达式：

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

接下来一个问题是处理考拉兹猜想 Collatz sequences，其数学表达为：

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \ (\mod 2) \\ 3n+1 & \text{if } n \equiv 1 \ (\mod 2) \end{cases}$$

我们希望知道的是：**对于从 1 至 100 的所有数开始，有多少链的长度是大于 15 的？**首先编写一个函数用于生成链：

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

因为链的最后一位肯定是 1，也就是边界，那么这就是一个简单的递归函数了。测试：

```
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]
```

看起来能正常工作，接下来就是获取长度：

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

我们将 `chain` 函数映射至 `[1..100]` 来获取一个链的列表，接着根据检查长度是否超过 15 的子句来过滤它们、一旦完成过滤，我们就可以看到结果的列表中还剩多少链。

**Note**

该函数的类型是 `numLongChains :: Int` ，因为历史原因 `length` 返回一个 `Int` 而不是一个 `Num a` 。如果我们需要返回一个更通用的 `Num a` ，可以对返回的长度使用 `fromIntegral` 。

使用 `map` ，我们还可以这样做 `map (*) [0..]` ，如果不是因为别的原因要解释柯里化以及偏函数是实值，那么可以将其传递至其它函数，或者置入列表中（仅仅不能将其变为字符串）。迄今为止，我们只映射了单参数的函数至列表，例如 `map (*2) [0..]` 类型是 `(Num a) => [a]` ，我们同样可以这么做 `map (*) [0..]` 。这里将会对列表中的每个数值应用函数 `*` ，即类型为 `(Num a) => a -> a -> a` 。将一个参数应用于需要两个参数的函数将会返回需要一个参数的函数。如果将 `*` 映射至列表 `[0..]` ，得到的则是一个接受单个参数的函数的列表，即 `(Num a) => [a -> a]` 。也就是说 `map (*) [0..]` 生产一个这样的列表 `[(0*),(1*),(2*),(3*),(4*),(5*).` 。

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

## Lambdas

构建 lambda 的方式是写一个 `\` ，接着是由空格分隔的参数，再然后是 `->` 符，最后是函数体。

对于 `numLongChains` 函数而言，不再需要一个 `where` 子句：

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

lambdas 也是表达式，上述代码中的表达式 `\xs -> length xs > 15` 返回的就是一个函数，其用作于告知列表的长度是否超过 15。

对于不熟悉柯里化以及偏函数应用的人而言，通常会在不必要的地方使用 lambdas。例如表达式 `map (+3) [1,6,3,2]` 以及 `map (x-> x + 3) [1,6,3,2]` 是等价的，因为 `(+3)` 以及 `(x-> x + 3)` 皆为接受一个值并加上 3 的函数。

与普通函数一样，lambdas 可以接受任意数量的参数：

```
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]
```

与普通函数一样，也可以在 lambdas 中进行模式匹配。唯一不同点在于不能在一个参数内定义若干模式，例如对同样一个参数做 `[]` 以及 `(x:xs)` 模式。如果一个模式匹配再 lambda 中失败了，一个运行时错误则会出现，所以需要特别注意在 lambdas 中进行模式匹配！

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

lambdas 通常都被圆括号包围，除非我们想让它一直延伸到最右。有趣的来了：函数默认情况下是柯里化的，以下两者相等：

```
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
```

## fold

一个 fold 接受一个二元函数，一个起始值（可以称其为 accumulator）以及一个需要被 fold 的列表。二元函数接受两个参数，第一个是 accumulator，第二个则是列表中第一个（或最后一个）元素，然后再生产一个新的 accumulator。接着二元函数再次被调用，带着新的 accumulator 以及新的列表中第一个（或最后一个）元素，以此类推。一旦遍历完整个列表，仅剩 accumulator 剩余，即最终答案。

首先让我们看一下 `foldl` 函数，也被称为左折叠 left fold。

改写 `sum` 的实现：

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

如果考虑到函数是柯里化的，那么可以简化实现：

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

这是因为 lambda 函数 `\acc x -> acc + x` 等同于 `(+)` ，所以可以省略掉 `xs` 这个参数，因为调用 `foldl (+) 0` 将返回一个接受列表的函数。

接下来通过左折叠来实现 `elem` 这个函数：

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

让我们看一下这里究竟做了些什么。这里的起始值与 accumulator 都是布尔值。在处理 fold 时，accumulator 与返回值的类型总是要一致的。这里的起始值为 `False` ，即假设寻找的值并不在列表中。接着就是检查当前元素是否为需要找到的那个，如果是则将 accumulator 设为 `True` ，不是则不变。

`foldr` 类似于左折叠，只不过 accumulator 是从列表右侧开始。

以下是使用右折叠来实现 `map` ：

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

如果将 `(+3)` 映射至 `[1,2,3]` ，则是从列表右侧开始，获取最后一个元素，即 `3` ，再应用函数得出 `6` ，接着将其放入 accumulator 头部，即将 `[]` 变为 `6:[]` ，这样 `[6]` 现在变成了新的 accumulator（这里的 `:` 就是将元素添加至头部）。

当然了，我们也可以用左折叠来实现：`map' f xs = foldl (\acc x -> acc ++ [f x]) [] xs` ，只不过 `++` 函数比 `:` 而言更加昂贵，因此我们通常从一个列表构建一个新的列表时，会使用右折叠。

**折叠可以用于实现任意一个想要一次性遍历列表中所有元素的函数，并基于此进行返回。任何时候想要遍历一个列表来返回一些东西时，那么这个时候就很可能需要一个 fold。**这也是为什么在函数式编程中，连同 maps 与 filters，fold 是最有用的函数类型。

`foldl1` 与 `foldr1` 类似于 `foldl` 与 `foldr` ，区别在于不需要提供一个显式的初始值。它们假设首个（或最后的）元素为起始值，接着从临近的元素开始进行 fold。由于依赖列表至少有一个元素，空列表的情况下它们会导致运行时错误；而 `foldl` 与 `foldr` 则不会。

现在通过 fold 来实现标准库的函数，看看 fold 的强大之处：

```
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) []

```

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

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

`head` 更好的实现当然是模式匹配，这里只是使用 `fold` 进行展示。这里的 `reverse'` 定义很聪明，从左遍历列表，每次将得到的元素插入至 accumulator 的头部。`acc x -> x : acc` 看起来像是 `:` 函数，只不过翻转了参数，这也是为什么可以将 `reverse` 函数改写为 `foldl (flip (:)) []` 。

`scanl` 与 `scanr` 很像 `foldl` 与 `foldr` ，不同之处在于后者用列表来存储 accumulator 的状态变化；同样 `scanl1` 与 `scanr1` 类似于 `foldl1` 与 `foldr1` 。

```haskell
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]]
```

Scans 可以被认为是一个函数可被 fold 的过程监控。让我们回答这样一个问题：**需要多少个元素才能让所有自然数的根之和超过 1000？** 获取所有自然数平方根只需要 `map sqrt [1..]` ，那么想要和，可以使用 fold，但是又因为我们感兴趣的是求和的这个过程，那么这里就可以使用 scan。一旦完成了 scan，我们就可以看到有多少和是少于 1000 的了。

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

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

这里使用了 `takeWhile` 而不是 `filter` ，因为后者不能作用于无限列表。尽管我们知道列表是递增的，而 `filter` 并不知道，因此这里的 `takeWhile` 在第一个 sum 大于 1000 时将截断 scan 列表。

### 通过 $ 符号进行函数应用

接下来让我们看一下 `$` 函数，也被称为*函数应用 function application*。首先来看一下它的定义：

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

大多数时候，它是一个便捷的函数使得无需再写很多括号。考虑一下表达式 `sum (map sqrt [1..130])`，因为 `$` 拥有最低的优先级，那么可以将该表达式重写为 `sum $ map sqrt [1..130]`，省了很多键盘敲击！当遇到一个 `$`，在其右侧的表达式会作为参数应用于左边的函数。那么 `sqrt 3 + 4 + 9` 呢？这会将 9, 4 以及 3 的平方根。那么如何得到 $3 + 4 + 9$ 的平方根呢，需要 `sqrt (3 + 4 + 9)`，那么如果使用 `$` 可以改为 `sqrt $ 3 + 4 + 9` 因为 `$` 在所有操作符中的优先级最低。这就是为什么可以想象一个 `$` 相当于分别在其右侧以及等式的最右侧编写了一个隐形的圆括号。

那么 `sum (filter (> 10) (map (*2) [2..10]))` 呢？由于 `$` 是右结合的，`f (g (z x))` 就相当于 `f $ g $ z x`。那么用 `$` 重写便是 `sum $ filter (>10) $ map (*2) [2..10]`。

除开省略掉圆括号，`$` 意味着函数应用可以被**视为另一个函数**，这样的话就可以将其映射 map 至一个列表的函数：

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

## 组合函数

数学里的组合函数定义为 $(f \circ g)(x) = f(g(x))$，意为组合两个函数来产生一个新的函数，当一个参数 `x` 输入时，相当于带着 `x` 输入至 `g` 再将结果输入至 `f`。

Haskell 中的组合函数基本上等同于数学定义中的一样。通过 `.` 函数来组合函数，其定义为：

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

注意类型声明。`f` 的参数类型与 `g` 的返回类型一致。

函数组合的用途之一是将函数动态的传递给其他函数。当然了，这点 lambdas 也可以做到，但是很多情况下，组合函数更加简洁精炼。假设我们有一个列表的数值，并希望将它们全部转为复数。其中一种办法就是将它们全部取绝对值后再添加负号：

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

注意到了 lambda 以及组合函数的样式，我们可以将上述重写为：

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

完美！组合函数是右结合的，所以我们可以一次性进行多次组合。表达式 `f (g (z x))` 等同于 `(f . g . z) x`。了解到这些以后，我们可以将：

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

转换为:

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

那么对于接受多个参数的函数而言呢?如果对它们进行组合函数,就需要偏应用它们使得每个函数仅接受一个参数。`sum (replicate 5 (max 6.7 8.9))` 可以被重写为 `(sum . replicate 5 . max 6.7` 或者是 `sum . replicate 5 . max 6.7 $ 8.9` 。那么这里实际上发生的是:一个函数接受了 `max 6.7` 所接受的参数,并将 `replicate 5` 应用至其,接着一个函数接受计算的出的结果并对其求和,最后则是带着 `8.9` 调用该函数。不过正常来讲,人类的读取应为:将 `8.9` 应用至 `max 6.7` ,接着应用 `replicate 5` ,最后则是求和。如果想用组合函数重写一个又很多圆括号的表达式,可以先把最内存函数的最后一个参数放在 `$` 后面,然后在不带最后一个参数的情况下,组合其他的函数调用,即在函数之间加上 `.` 。如果有一个表达式 `replicate 100 (product (map (*3) (zip` ,那么可以重写为 `replicate 100 . product . map (*3) . zipWith max [1,2,3,4,5] [4,5,6,7,8]` 。如果表达式的结尾有三个圆括号,转换为组合函数时,则会有三个组合操作符。

组合函数的另一个常见的用法是以所谓的点自由样式(也称无点样式)来定义函数。拿之前我们写的一个列子:

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

`xs` 在两边都暴露出来。由于有柯里化,我们可以在两边省略 `xs` ,因为调用 `foldl (+) 0` 会创建一个接受一个列表的函数。编写函数 `sum' = foldl (+) 0` 就是被称为无点样式。那么以下函数如何转换成无点样式呢?

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

我们没法将 `x` 从等式两边的右侧移除,函数体中的 `x` 后面有圆括号。`cos (max 50)` 显然没有意义。那么将 `fn` 表达为组合函数即可:

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

漂亮!无点样式可以令用户去思考函数的组合方式,而非数据的传递方式,这样更加的简明。你可以将一组简单的函数组合在一起,使之成为一个负责的函数。不过如果函数过于复杂,再使用无点样式往往会达到反效果。因此构造较长的函数组合链并不好,更好的解决方案则是用 `let` 语句将中间的运算结果绑定一个名字,或者说把问题分解成几个小问题再组合到一起。

本章的 maps 与 filters 中,我们写了:

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

那么更为函数式的写法则是

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

如果需要可读性，则可以这样：

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

# 7 Modules

## 加载模块

Haskell 模块是一系列关联的函数，类型以及 typeclasses 的集合。一个 Haskell 的程序是模块的集合，其中主模块加载若干模块并使用这些模块。

Haskell 的标准库也被分成了若干模块，每个模块所包含的函数与类型服务于某些共同的目的。迄今为止我们所有接触到的函数，类型，typeclasses 都位于 `Prelude` 模块，也是默认加载的模块。

Haskell 加载模块的语法就是 `import <module name>` 。现在尝试一下加载 `Data.List` 这个模块：

```
import Data.List

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

当 `import Data.List` 后，所有在 `Data.List` 中的函数在公共命名空间中变为可用，也就是说可以在脚本中任意处调用这些函数。`nub` 是一个定义在 `Data.List` 中的函数，接受一个列表并去除其中的重复元素。将 `length` 与 `num` 组合起来 `length . nub` 产生的函数等同于 `xs -> length (nub xs)` 。

在使用 GHCI 的时候同样也可以加载模块至全局命名空间内，只需：

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

如果想要加载多个模块，仅需：

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

如果加载的脚本（通过 `:l <xxx script>` ）中已经加载过模块了，那么便不再需要通过 `:m +` 进行加载。

如果只是想要从模块中加载几个函数，那么可以这样做：

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

如果想加载模块，却不包括某些函数，可以这样：

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

为了避免重名，可以使用 `qualified` ，譬如这样：

```
import qualified Data.Map
```

这样的话如果想要调用 `Data.Map` 的 `filter` 函数时，就必须要 `Data.Map.filter` ，这样的话 `filter` 仍然会引用普通的 `filter` 。不过每次都要写 `Data.Map` 就很麻烦，因此可以这么写：

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

现在再要调用 `Data.Map` 的 `filter` 时，仅需 `M.filter` 。

可以在这里找到标准库中有哪些模块。

通过Hoogle可以查看函数的位置，这是一个非常棒的 Haskell 搜索引擎，可以通过名称，模块名称甚至是类型签名来进行搜索。

## Data.List

`Data.List` 模块显然都是关于列表的，它提供了些很有用的函数用于列表处理。我们已经学习到了一些（例如 `map` 与 `filter` ）这是因为 `Prelude` 已经从 `Data.List` 中加载了不少函数。无需再通过 qualified import 导入 `Data.List` ，因为它并不会与任何的 `Prelude` 名称重名（除开那些已经被 `Prelude` 从 `Data.List` 中偷走的）。现在让我们看一下其中一些还没有用到过的函数：

`intersperse` 接受一个元素以及一个列表，将该元素插入至列表中每个元素之间：

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

`intercalate` 接受一个列表以及一个列表的列表，将前者插入至后者之间在打平结果：

```
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` 接受一个列表的列表，如果将其视为一个二维的矩阵，那么就是列变为行，行变位列：

```
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"]
```

假设我们有一些多项式 $3x^2+5x+9, 10x^3+9$ 以及 $8x^3+5x^2+x-1$，我们希望对它们求和，那么可以使用列表 `[0,3,5,9]` ， `[10,0,0,9]` 以及 `[8,5,1,-1]` 来代表它们的系数（分别为 $x^3, x^2, x^1, x^0$ 的系数），那么可以这样：

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

上述做法就是先转置整个列表 `transpose` ，这就代表着每个同幂的系数在一个列表中，再将 `sum` 映射至这个转置后的列表中的每个元素（列表）。

`foldl'` 与 `foldl1'` 相对于它们的懒加载的原版是更严格的版本，当使用原版的懒加载 fold 用于一个很大的列表，很有可能就会得到一个堆栈溢出的错误。罪魁祸首就是因为 fold 的懒加载特性，累加器的值并不会在 folding 时真正的去计算，相反是在需要结果时才去进行计算（这也被称为一个型实转换程序 thunk）。这发生在每个中间累加器上，且所有的这些 thunk 都会导致堆栈溢出。strict 版本由于不是懒加载的，是真实的计算中间值而不是一直在栈上堆叠。因此当使用懒加载的 folds 时遇到了堆栈溢出，可以尝试一下用它们的严格版本。

`concat` 打平一个列表的列表：

```
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]
```

`concatMap` 首先映射一个函数至列表接着将该列表进行 `concat` ：

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

`and` 接受一个布尔值的列表，当所有值皆为 `True` 时返回 `True` ：

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

`or` 与 `and` 类似，只不过是任意元素为 `True` 时返回 `True` ：

```
ghci> or $ map (==4) [2,3,4,5,6,7]
True
ghci> or $ map (>4) [1,2,3]
False
```

`any` 与 `all` 接受一个子句，然后检查列表中的所有元素，通常而言我们会使用这两个函数而不是像上面那样先 `map` 接着 `and` 或是 `or` 。

```
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` 接受一个函数以及一个起始值，将函数应用在这个起始值得到结果后，再将函数应用至该结果，以此类推，返回一个无限列表。

```
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` 接受一个数值与一个列表，将列表从数值作为的索引出切分为两部分，返回一个包含了切分后两个列表的二元元组：

```
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` 是一个非常有用的小函数，它从一个列表中从头开始获取元素，直到元素不再满足子句的条件，返回之前所有满足元素的列表：

```
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"
```

假设我们想要所有三次方都小于 10,000 的值之和，将 `(^3)` 映射至 `[1..]` ，然后应用一个过滤函数再进行求和这是做不到的，因为过滤一个无限长的列表将永远都不会结束。虽然我们知道列表里的元素是递增的，但是 Haskell 并不知道，因此需要这么做：

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

将 `(^3)` 应用至一个无限列表，然后一旦元素超过 10,000 那么列表将被截断，这样才可以求和。

`dropWhile` 也类似，它扔掉所有子句中条件判断为真的元素，一旦子句返回 `False` ，则返回剩余的列表。

```
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]
```

`span` 有点像 `takeWhile` ,不过它返回的是一对列表。第一个列表包含了所有 `takeWhile` 所返回的元素，第二个列表则是剩余的部分：

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

`break` 则是列表中第一个元素令子句条件为真时，切分并返回一对列表。`break p` 等同于 `span (not . p)` 。

```
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])
```

使用 `break` 时，第一个满足条件的元素将会放在第二个列表的头部。

`sort` 则是对一个列表排序。元素的类型必须属于 `Ord` typeclass：

```
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` 接受一个列表，并将相邻的相同的元素组成子列表：

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

如果在 group 一个列表之前进行排序，那么我们就可以知道每个元素在列表中出现了多少次：

```
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` 与 `tails` 类似于 `init` 与 `tail` ，不同之处在于前者们会递归的应用至一个列表直到空：

```
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","")]
```

尝试一下使用 fold 来实现查找子列表：

```
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)
search needle haystack =
  let nlen = length needle
   in foldl (\acc x -> (take nlen x == needle) || acc) False (tails haystack)
```

首先调用 `tails` 来处理需要查找的列表，接着查找每个 tail 直到该 tail 的头是期望找到的。

上述代码的行为实际上就是 `isInfixOf` 函数，该函数在一个列表中查找子列表，在发现了该子列表时返回 `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` 与 `isSuffixOf` 则是从前往后进行查找与从后往前进行查找：

```
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` 与 `notElem` 则是查找元素是否在列表中。

`partition` 接受一个列表以及一个子句，返回一对列表，第一个列表包含了所有符合子句条件的元素，其余的在第二个列表：

```
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])
```

理解 `partition` 有别于 `span` 与 `break` 是非常重要的：

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

因为 `span` 与 `break` 两者是一旦元素满足子句条件便停止，而 `partition` 则是根据子句判断遍历整个列表。

`find` 接受一个子句以及一个列表，返回首个满足子句条件的元素。不过它返回的元素是被包含在 `Maybe` 值之中。我们将会在下一节深入讲解代数数据类型，不过现在我们需要知道的是：一个 `Maybe` 值可以是 `Just something` 或是 `Nothing` 。这就像是一个列表既可以是空列表也可以是带有元素的列表，而 `Maybe` 即可以是无元素又可以是一个元素。

```
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
```

这里需要注意的是 `find` 的类型，它返回的是 `Maybe a` 。

`elemIndex` 有点像 `elem` ，不过它返回的不是布尔值，而是一个由 `Maybe` 包含的索引，如果元素不在列表中，则返回 `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` 类似于 `elemIndex` ，不过它返回的是索引的列表，因为是列表，所以即使找不到元素也可以返回一个空列表，这样就不需要一个 `Maybe` 类型了：

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

`finxIndex` 像 `find` ，不过返回的是索引，而 `findIndices` 返回的是全部匹配元素的索引：

```
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]
```

我们已经尝试过了 `zip` 与 `zipWith` ，那么对于多个列表就可以使用 `zip3` ， `zip4` 等等，以及 `zipWith3` ， `zipWith4` 等等，这里最高可以是 zip 七个列表。不过有更好的办法可以 zip 无穷多个列表，只不过我们现有的知识暂时还没有办法到那儿。

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

跟普通的 `zip` 一样，计算到最短的列表结束为止。

当处理文件或者是其他地方而来的输入， `lines` 则是一个非常有用的函数：

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

`'\n'` 是 unix 的换行符。

`unlines` 则是与 `lines` 相反，将字符串列表变回一个由 `'\n'` 分隔的大字符串：

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

`words` 与 `unwords` 则是分隔与组装一行字符串：

```
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"
```

`nub` 我们已经见识过了，移除列表中的重复元素：

```
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` 接受一个元素以及一个列表，删除列表中第一个出现的元素。

```
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!"
```

`\\` 是一个列表比较函数，类似于集合比较，根据右侧的列表中的元素，移除左边列表中匹配的元素。

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

处理 `[1..10] \\ [2,5,9]` 类似于 `delete 2 . delete 5 . delete 9 $ [1..10]` 。

`union` 类似于集合的并集，它返回的是两个列表的集合，做法是将第二个列表中没有在第一个列表中出现的元素，添加至第一个列表的尾部。注意第二个列表中重复的元素会被移除。

```
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]
```

`intersect` 类似于集合的交集，它返回两个列表同时存在的元素：

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

`insert` 接受一个元素以及一个可以被排序的列表，将该元素插入到最后一个仍然小于或等于下一个元素的位置，`insert` 将会从列表的头开始，直到找到一个元素大于等于它，接着插入到找到的这个元素之前并结束。

```
ghci> insert 4 [3,5,1,2,8,2]
[3,4,5,1,2,8,2]
ghci> insert 4 [1,3,4,4,1]
[1,3,4,4,4,1]
```

如果我们对一个排序后的列表使用 `insert` ，那么返回的列表仍然是排序好的：

```
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]
```

因为历史的原因 `length` ， `take` ， `drop` ， `splitAt` ， `!!` 以及 `replicate` 接受的都是 `Int` 类型，即使它们可以更加的泛用接受 `Integral` 或 `Num` typeclasses（取决于函数本身），但是修改它们则会影响大量已经存在的代码。这就是为什么在 `Data.List` 中引入了 `genericLength` ， `genericTake` ， `genericDrop` ， `genericSplitAt` ， `genericIndex` 以及 `genericReplicate` 作为泛化的版本。例如 `length` 的类型签名是 `length :: [a] -> Int` ，那么如果想要一个列表的均值 `let xs = [1..6] in sum xs / length xs` ，这么做会得到一个类型错误，因为我们不能对 `Int` 使用 `/` 。而 `genericLength` 的类型签名是 `genericLength :: (Num a) =>` 。因为一个 `Num` 可以是一个浮点数，那么 `let xs = [1..6] in sum xs / genericLength xs` 这样做就没有问题了。

对于 `nub` ， `delete` ， `union` ， `intersect` 以及 `group` 而言，它们的泛化版本则是 `nubBy` ， `deleteBy` ， `unionBy` ， `intersectBy` 以及 `groupBy` 。它们的不同之处在于前一批使用的函数是 `==` 用于比较，而后一批则是使用输入的比较函数进行比较。 `group` 等同于 `groupBy (==)` 。

例如，假设我们有一个描述函数美妙值的列表。我们希望基于正负数，将其它们分别分段形成子列表。如果用普通的 `group` 那就只能将相同的相邻元素分组在一起，而使用 `groupBy` 则可以通过 *By* 函数进行同样类型的判断，当认为是同样类型的即返回 `True` ：

```
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]]
```

一个更简单的方法是从 `Data.Function` 中加载 `on` 函数，其定义：

```
ghci> :t on
on :: (b -> b -> c) -> (a -> b) -> a -> a -> c
```

那么使用 `(==) `on` (> 0)` 返回的函数类似于 `\x y -> (x > 0) == (y > 0)` 。 `on` 在 *By* 函数中运用的很广：

```
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]]
```

确实非常有可读性！我们可以大声的念出来：根据元素是否大于零进行相等分组。

`sort` ，`insert` ，`maximum` 以及 `minimum` 也同样拥有更泛用的版本： `sortBy` ，`insertBy` ，`maximumBy` 以及 `minimumBy` 。`sortBy` 的类型签名是 `sortBy :: (a -> a -> Ordering) -> [a] -> [a]` 。如果还记得之前的 `Ordering` 类型可以是 `LT` `EQ` 或 `GT` 。`sort` 等同于 `sortBy compare` ，因为 compare 只接受两个类型为 Ord typeclass，并返回它们的排序关系。

列表可以被比较，一旦可以比较，它们则是根据字典顺序进行比较。那么如果我们有一个列表的列表，并想要不根据内部列表的内容，而是根据长度来进行排序呢？这就可以使用 `sortBy` 函数：

```
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]]
```

这里的 `on` 等同于 `\x y -> length x `compare` length y` 。也就是说处理 *By* 函数时，需要等式时可以使用 `(==) `on` something` ，需要排序时可以 `compare `on` something` 。

## Data.Char

`Data.Char` 正如其名，是一个用于处理字符的模块，同样对过滤或映射字符串有很大的帮助，因为字符串本身就是字符的列表。

`Data.Char` 提供了很多关于字符范围的函数。也就是函数接受一个字符，并告诉我们哪些假设为真或为假：

`isControl` 判断一个字符是否是控制字符

`isSpace` 判断一个字符是否是空格字符，包括空格，tab，换行符等

`isLower` 判断一个字符是否为小写

`isUpper` 判断一个字符是否为大写

`isAlpha` 判断一个字符是否为字母

`isAlphaNum` 判断一个字符是否为字母或数字

`isPrint` 判断一个字符是否可打印

`isDigit` 判断一个字符是否为数字

`isOctDigit` 判断一个字符是否为八进制数字

`isHexDigit` 判断一个字符是否为十六进制数字

`isLetter` 判断一个字符是否为字母

`isMark` 判断一个字符是否为 unicode 注意字符（法语）

`isNumber` 判断一个字符是否为数字

`isPuncuation` 判断一个字符是否为标点符号

`isSymbol` 判断一个字符是否为货币符号

`isSeparator` 判断一个字符是否为 unicode 空格或分隔符

`isAscii` 判断一个字符是否在 unicode 字母表的前 128 位

`isLatin1` 判断一个字符是否为 unicode 字母表的前 256 位

`isAsciiUpper` 判断一个字符是否为大写的 Ascii

`isAsciiLower` 判断一个字符是否为小写的 Ascii

以上所有判断函数的类型声明都是 `Char -> Bool`，大多数时候我们会用它们来过滤字符串或者其它。例如假设我们做一个程序用来获取用户名，而用户名只能由字母与数字构成，我们可以使用 `Data.List` 里的 `all` 函数与 `Data.Char` 里的子句用于判断用户名是否正确：

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

同样可以通过 `isSpace` 来模拟 `Data.List` 中的 `words` 函数：

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

呃这看起来像是 `words` 不过我们留下来空格，那么再用一次 `filter`

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

`Data.Char` 同样提供了一个类似 `Ordering` 的数据类型。`Ordering` 类型可以是 `LT`，`EQ` 或 `GT`，类似于枚举。而 `GeneralCategory` 同样也是枚举，它提供了字符所处的范围 `generalCategory :: Char -> GeneralCategory`，由 31 个种类，这里只列举部分：

```
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]
```

由于 `GeneralCategory` 类型属于 `Eq` typeclass，也就可以这样进行测试 `generalCategory c == Space`。

`toUpper` 转换一个字符成为大写。空格、数字等不变。

`toLower` 转换一个字符成为小写。

`toTitle` 转换一个字符成为 title-case，大多数字符就是大写。

`digitToInt` 将一个字符转为 Int 值，这个字符必须在 `'1'..'9','a'..'f'` 或是 `'A'..'F'` 的范围之内。

```
ghci> map digitToInt "34538"
[3,4,5,3,8]
```

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

`intToDigit` 则与 `digitToInt` 相反。

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

`ord` 与 `chr` 函数将字符转换成相应的数值，反之亦然：

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

下面是 `encode` 与 `decode` ：

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

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

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

```
ghci> encode 3 "Im a little teapot"
"Lp#d#olwwoh#whdsrw"
ghci> decode 3 "Lp#d#olwwoh#whdsrw"
"Im a little teapot"
ghci> decode 5 . encode 5 $ "This is a sentence"
"This is a sentence"
```

## Data.Map

关联列表（通常也被称为字典）是一种用于存储键值对却不保证顺序的列表。

由于 `Data.Map` 中的函数会与 `Prelude` 中的 `Data.List` 冲突，因此需要 qualified import：

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

现在让我们看一下 `Data.Map` 中有什么好东西！

`fromList` 函数接受一个关联列表，并返回一个 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)]
```

如果有重复的键出现，那么之前的值会被丢弃，这里是 `fromList` 的类型签名：

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

意为接受一个键值类型 `k` 与 `v` 对的列表，并返回一个 map。注意这里的键必须属于 `Eq` 以及 `Ord` typeclass，后者是因为需要将键值安排在树结构中。

在需要键值关联的类型时总是使用 `Data.Map` ，除非键不属于 `Ord` typeclass。

`empty` 没有参数，返回一个空 map：

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

`insert` 接受一键，一值，并返回 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)]
```

我们可以通过空 map，`insert` 以及 `foldr` 来实现自己的 `fromList` ：

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

`null` 检查 map 是否为空：

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

`size` 告知 map 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` 创建一个只有一对键值的 map：

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

`lookup` 类似于 `Data.List` 的 `lookup`，不过它作用于 map。如果找到了则返回 `Just something`，反之 `Nothing`

`member` 是一个子句，接受一个键以及一个 map，查找该键是否在 map 中：

```
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` 与 `filter` 于列表的函数相似：

```
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` 与 `fromList` 相反：

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

`keys` 与 `elems` 分别返回键与值的列表。`keys` 等同于 `map fst . Map.toList`，而 `elems` 等同于 `map snd . Map.toList`。

`fromListWith` 这个函数很酷。它作用类似于 `fromList`，不过不会丢弃重复键而是使用函数来决定去留。假设我们有这样一个列表（注意在 ghci 中使用 `:{` 与 `:}` 作为多行输入的开头与结尾）：

```
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")
    ]
```

如果使用 `fromList` 来生成一个 map，我们将会丢弃一些号码！因此这里我们这么做：

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

```
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"
```

如果遇到一个重复的键，传入的函数用于将这些值结合成为其他值。我们还可以首先将所有值映射为单例的列表后在使用 `++` 来结合：

```
phoneBookToMap' :: (Ord k) => [(k, a)] -> Map.Map k [a]
phoneBookToMap' = Map.fromListWith (++) . map (\(k, v) -> (k, [v]))
```

```
ghci> Map.lookup "patsy" $ phoneBookToMap' phoneBook
Just ["827-9162","943-2929","493-2928"]
```

非常的精妙！另一个用例则是如果我们从一个数值关联的列表创建 map，当遇到重复键时，我们希望较大的键的值能保留：

```
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)]
```

或者将重复键的值相加：

```
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` 则是插入一个键值对进入 map，如果键重复则使用传递的函数进行判断：

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

根据官网的介绍现在最好是用 Data.Map.Strict （2023/7/14）。

### Data.Set

由于 `Data.Set` 中的名称会与 `Prelude` 与 `Data.List` 重复，因此需要 qualified import：

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

`fromList` 将列表转换为集合：

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

`intersection` 取交集:

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

`difference` 取在第一个集合却不在第二个集合的元素:

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

`union` 取并集:

```
ghci> Set.union set1 set2
fromList " !.?AIRTabcdefghijlmnorstuvwy"
```

`null` , `size` , `member` , `empty` , `singleton` , `insert` 以及 `delete` 函数与预期一致:

```
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]
```

也可以检查子集等:

```
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
```

`map` 与 `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]
```

`toList` 转为列表，按字典顺序进行排序：

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

## 构建自己的模块

首先我们创建一个名为 `Geometry.hs` 的文件。

当我们提到模块 *exports* 函数，即当导入一个模块时，可以使用该模块导出的函数。

在模块的开头，需要指定模块名称，接着指定想要导出的函数：

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

接下来就是定义函数：

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

import Control.Monad

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

```

```haskell
sphereArea :: Float -> Float
-- sphereArea radius = 4 * pi * (radius ^ 2)
sphereArea = (* (4 * pi)) . (^ 3)

cubeVolume :: Float -> Float
-- cubeVolume side = cuboidVolume side side side
cubeVolume = join (join cuboidVolume)

cubeArea :: Float -> Float
-- cubeArea side = cuboidArea side side side
cubeArea = join (join cuboidArea)

cuboidVolume :: Float -> Float -> Float -> Float
-- cuboidVolume a b c = rectangleArea a b * c
cuboidVolume = ((*) .) . rectangleArea

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
rectangleArea = (*)
```

我们仅需像下面这样就能使用模块:

```haskell
import Geometry
```

需要注意的是 `Geometry.hs` 需要在加载模块的程序同一文件夹下。

模块童谣可以被给予等级结构。每个模块可以拥有若干子模块，子模块里还可以有子模块。

首先创建一个名为 `Geometry` 的文件夹,接着是下面的三个文件夹: `Sphere.hs` , `Cuboid.hs` 以及 `Cube.hs`

`Sphere.hs` :

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

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

area :: Float -> Float
area = (* (4 * pi)) . (^ 3)
```

`Cuboid.hs` :

```haskell
module Geometry.Cuboid
  ( volume,
```

```
    area,
  )
where

volume :: Float -> Float -> Float -> Float
volume = ((*) .) . rectangleArea

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 = (*)
```

以及 `Cube.hs` ：

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

import Control.Monad
import Geometry.Cuboid qualified as Cuboid

volume :: Float -> Float
volume = join (join Cuboid.volume)

area :: Float -> Float
area = join (join Cuboid.area)
```

那么在与文件夹同级的目录下就可以这样加载模块中的子模块：

```
import Geometry.Sphere
```

为了避免命名冲突可以使用 qualified import：

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

# 8 Making Our Own Types and Typeclasses

## 代数数据类型介绍

迄今为止我们接触到了不少的类型：`Bool`，`Int`，`Char`，`Maybe` 等等。但是我们如何构造自己的类型呢？一种方式是使用 **data** 关键字来进行定义。让我们来看一下标准库中的 `Bool` 是怎么定义的：

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

**data** 意味着正在定义一个新的数据类型。在 `=` 之前的部分代表着类型，即 `Bool`；而之后的部分则是 **类型构造函数 value constructors**。它们指定了类型可变的值，这里的 `|` 读作 *或 or*，因此整句代码可以读作：`Bool` 类型可以是 `True` 或 `False` 其中的一个值。

假设这里定义了形状可以是一个圆或是长方形：

```
ghci> data Shape = Circle Float Float Float | Rectangle Float Float Float Float
ghci> :t Circle
Circle :: Float -> Float -> Float -> Shape
ghci> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
```

`Clicle` 的值构造函数有三个字段，均接受浮点数；而 `Rectangle` 的值构造函数有四个字段，均接受浮点数。

值构造函数实际上是最终返回数据类型值的函数。以下是一个接受 shape 并返回 surface 的函数：

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

首先值得注意的是类型声明。我们不能这样 `Circle -> Float` 因为 `Circle` 并非一个类型，`Shape` 才是。这就像我们不能编写一个类型声明为 `True -> Int` 的函数。其次需要注意的是我们不能对构造函数进行模式匹配，之前我们匹配过 `[]`，`False` 或是 `5`，它们是不包含参数的值构造函数。

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

很好成功了！但是如果我们想要打印出 `Circle 10 20 5`，则会得到一个错误。这是因为 Haskell 并不知道该如何将我们的数据类型转换成字符串，因此我们需要让 `Shape` 成为 `Show` typeclass 的一部分：

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

值构造函数是函数，因此我们可以映射它们并偏应用至任何东西：

```
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]
```

我们的数据类型很棒，不过可以更棒。定义一个中间类型增强可读性：

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

修改 `surface` 函数：

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

调用时需要考虑模式：

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

接下来是 `nudge` 函数：

```
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))
```

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

如果我们不想直接处理点，那么可以辅助函数用于创建初始在零点的形状并将其移动至正确点位：

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

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)
```

如果希望将所有的函数与类型导出（注意类型中使用 `..` ，下文有解释），那么可以这么做：

```
module Shapes
  ( Point (..),
    Shape (..),
    surface,
    nudge,
```

```
    baseCircle,
    baseRect,
  )
where
```

这里的 `Shape (..)` 导出了所有 `Shape` 的值构造函数，因此任何加载了该模块的都可以通过 `Rectangle` 与 `Circle` 的值构造函数来创建形状。

当然也可以选择行的不到处任何 `Shape` 的值构造函数，仅需在导出声明中这样写 `Shape` 。这样的话加载该模块的仅能使用辅助函数 `baseCircle` 与 `baseRect` 来创建形状。`Data.Map` 使用了这个技巧。

不到处数据类型的值构造函数会使得它们更为抽象，因为我们隐层了它们的实现；除此之外，使用该模块的将不再能对其使用模式匹配。

## Record Syntax

现在让我们创造一个用于描述人的数据类型，其中信息包括：名，姓，年龄，身高，体重，电话以及喜爱的冰淇淋类型。

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

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

不错，但是不易读。现在让我们创建一个函数来获取信息：

```
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
```

```
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"
```

现在让我们使用 record syntax：

```
data Person = Person
{ firstName :: String,
  lastName :: String,
  age :: Int,
  height :: Float,
  phoneNumber :: String,
  flavor :: String
}
deriving (Show)
```

我们通过两个冒号 `::` 来指定类型。

```
ghci> :t flavor
flavor :: Person -> String
ghci> :t firstName
firstName :: Person -> String
```

使用 record syntax 的另一个好处就是为类型派生 `Show` 时，展示的样子会更方便辨认：

```
data Car = Car String String Int deriving (Show)
```

```
ghci> Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967
```

使用 record syntax：

```
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}
```

## 类型参数

一个值构造函数可以接受一些参数然后生产出一个新值。例如 `Car` 构造函数接受三个值并生产出一个 car 值。同样的方式，**类型构造函数 type constructor** 接受值作为参数并生成出一个新类型。首先让我们看一下 `Maybe` ：

```
data Maybe a = Nothing | Just a
```

这里的 `a` 就是一个类型参数，因为类型参数的介入，我们称 `Maybe` 为一个类型构造函数。取决于我们想要它在不是 `Nothing` 时存储的数据类型，该类型构造函数可以生产 `Maybe Int` ， `Maybe Car` 或者是 `Maybe String` 等等。

你可能不知道的是，在使用 `Maybe` 之前，我们使用了一个由类型参数的类型。

```
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
```

使用类型参数很方便，不过也得合理的使用。

另一个我们已经遇到过的参数化类型例子就是 `Data.Map` 中的 `Map k v` 。如果我们要定义一个 mapping 类型，我们可以添加一个 typeclass 约束在数据声明中：

```
data (Ord k) => Map k v = ...
```

然而在 Haskell 中，**永远不要在数据声明中添加 typeclass 约束**，这是一个非常强大的约定。为什么呢？因为我们并不能从中得到多大的好处，最终还写了更多的类约束，即使我们不需要它们。`Map k v` 要是有 `Ord k` 的约束，那就相当于假定每个 map 的相关函数都认为 k 是可排序的。如果不给数据类型加约束，那么就不用给不关心键是否可排序的函数另加约束了。这类函数的一个例子就是 `toList` ，它只是将 map 转换为关联列表而已，类型声明为 `toList :: Map k v -> [(k, v)]` 如果加上类型约束,那就得 `toList :: (Ord k) => Map k a -> [(k, v)]` ，很明显没有必要这么做。

让我们实现一个 3D 向量类型，并为其添加一些操作。这里使用一个参数化的类型，虽然通常而言包含的是数值类型，不过这样支持了多种数值类型：

```
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)

vMult :: (Num t) => Vector t -> t -> Vector t
(Vector i j k) `vMult` 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
```

这三个函数可以作用于 `Vector Int` ， `Vector Integer` 以及 `Vector Float` 类型上，或者是任何满足 `Num` typeclass 的 `a` 。

再次强调，分辨出类型构造函数还是值构造函数是非常重要的。当定义一个数据类型，`=` 之前的部分就是类型构造函数，而之后的（有可能通过 `|` 来分隔）则是值构造函数。给这样一

个函数类型 `Vector t t t -> Vector t t t -> t` 是错误的，因为我们必须将类型放置在类型声明中，且向量的**类型**构造函数仅接受一个参数，而值构造函数接受三个。

```
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
```

## 派生实例

在 Typeclasses 101 章节中，我们解释了 Typeclasses 的基础，即一种用于定义某些行为的接口。一个类型可以做该 typeclass 的 **instance**，如果该类型支持这些行为。

我们也提到了 typeclasses 有别于 Java，Python，C++ 的类；在这些语言中，类是一个蓝图供我们创建包含了状态与一些行动的对象，而 Typeclasses 更类似于接口。我们不从 typeclasses 中创造数据，而是先构建数据类型，接着思考其可行的行动。如果它可以像等式那样行动，那么我们为其构建一个 `Eq` typeclass 的实例；如果它可以进行排序，那么我们为其构建一个 `Ord` typeclass 的实例。

下一节中，我们将尝试如何通过实现定义在 typeclasses 里的函数，手动创建我们 typeclasses 的类型实例。不过现在让我们看看 Haskell 是如何自动的将我们的类型创建出以下任何 typeclasses 的实例：`Eq` ，`Ord` ，`Enum` ，`Bounded` ，`Show` ，`Read` 。当我们使用 *deriving* 关键字时，Haskell 可以为我们的类型派生出这些行为。

```
data Person = Person { firstName :: String
, lastName :: String
, age :: Int
} deriving (Eq)
```

```
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
```

当然了，由于 `Person` 已经在 `Eq` 了，我们就可以使用那些有类约束 `Eq a` 的函数了，例如 `elem` ：

```
ghci> let beastieBoys = [mca, adRock, mikeD]
ghci> mikeD `elem` beastieBoys
True
```

```
data Person = Person
  { firstName :: String,
    lastName :: String,
    age :: Int
  }
  deriving (Eq, Show, Read)
```

在终端上打印：

```
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}"
```

`Read` 则与 `Show` 相反：

```
ghci> read "Person {firstName =\"Michael\", lastName =\"Diamond\", age = 43}" :: Person
Person {firstName = "Michael", lastName = "Diamond", age = 43}
```

我们可以使用代数数据类型来创建枚举，其中 `Enum` 以及 `Bounded` typeclasses 帮了大忙。`Enum` typeclass 适用于有前置子和后继子的情况，而 `Bounded` typeclass 则代表有最大和最小值。例如：

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
  deriving (Eq, Ord, Show, Read, Bounded, Enum)
```

因为有 `Show` 与 `Read` typeclasses，我们可以将其与字符串互相转换：

```
ghci> Wednesday
Wednesday
ghci> show Wednesday
"Wednesday"
ghci> read "Saturday" :: Day
Saturday
```

因为有 `Eq` 与 `Ord` typeclasses，我们可以进行比较：

```
ghci> Saturday == Sunday
False
ghci> Saturday == Saturday
True
ghci> Saturday > Friday
True
ghci> Monday `compare` Wednesday
LT
```

又因为有 `Bounded` ，我们可以得到最大与最小天：

```
ghci> minBound :: Day
Monday
ghci> maxBound :: Day
Sunday
```

最后是 `Enum` ，我们可以使用前置子与后继子：

```
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]
```

非常棒。

## 类型同义词

之前我们在讨论类型的时候提到过，`[Char]` 和 `String` 类型是相同的也是可互换的。这是因为实现了 **类型同义词 type synonyms**。标准库中的定义：

```
type String = [Char]
```

这里引入了 *type* 关键字。由于我们并没有创建新的东西（如 *data* 关键字），*type* 仅关联已存在的类型的同义词。

```
type PhoneNumber = String
type Name = String
type PhoneBook = [(Name,PhoneNumber)]
```

现在可以实现一个函数用于接受名字，号码，并检查名字与号码的组合是否存在于号码簿中：

```
inPhoneBook :: Name -> PhoneNumber -> PhoneBook -> Bool
inPhoneBook name pNumber pBook = (name, pNumber) `elem` pBook
```

这里若是没有使用类型同义词，那么函数的签名则会是 `String -> String -> [(String,String)] -> Boo`

。

类型同义词同样也可以参数化。

```
type AssocList k v = [(k,v)]
```

那么通过关联列表中的键获取值的函数类型可以是 `(Eq k) => k -> AssocList k v -> Maybe v`

。

正如我们可以偏应用函数来获取一个新的函数，我们还可以偏应用类型参数来获取一个新的类型构造函数。正如我们在调用函数时缺少一些参数会返回一个新的函数，我们可以指定一个类型构造函数部分参数并返回一个偏应用的类型构造函数。如果我们想要一个整数为键的 map，我们可以这么做：

```
type IntMap v = Map Int v
```

或是这样：

```
type IntMap = Map Int
```

另外一个酷炫的数据类型是 `Either a b` 类型，它接受两个类型参数：

```
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)
```

它有两个值构造函数。如果使用了 `Left` ，那么其内容则是类型 `a` ，反之则是 `b` 。因此我们可以封装一个拥有两种类型的值，然后使用模式匹配进行取值。

```
ghci> Right 20
Right 20
ghci> Left "w00t"
Left "w00t"
ghci> :t Right 'a'
Right 'a' :: Either a Char
ghci> :t Left True
Left True :: Either Bool b
```

一个例子：一个壁橱拥有代码组合，每次申请一个壁橱的代码如果已经存在了，那么需要告知重新选择。这里使用 `Data.Map` 来代表壁橱：

```
import Data.Map qualified as Map

data LockerState = Taken | Free deriving (Show, Eq)

type Code = String

type LockerMap = Map.Map Int (LockerState, Code)
```

这里引用了一个新的数据类型来代表一个壁橱是否被占用，同时也为壁橱代码设置了一个类型同义词。现在让我们使用 `Either String Code` 类型来做为查找函数的返回类型，因为查找可能会以两种原因失败 – 橱柜已被占用或者是没有该橱柜。

```
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 taken!"
```

```
lockers :: LockerMap
lockers =
  Map.fromList
    [ (100, (Taken, "ZD39I")),
      (101, (Free, "JAH3I")),
      (103, (Free, "IQSA9")),
      (105, (Free, "QOTSA")),
      (109, (Taken, "893JJ")),
      (110, (Taken, "99292"))
    ]
```

测试一下：

```
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"
```

我们当然可以使用 `Maybe a` 来做结果，不过那样的话就不知道为什么不能拿到代码的原因了，而现在这么做，在拿不到代码的时候是可以知道是什么原因造成的。

## 递归数据结构

正如我们所见，代数数据类型中的构造函数可以有多个（或零）字段，每个字段必须是某些实际类型。有了这个概念，我们构建的类型可以以自身类型为字段！

试想一下列表：`[5]` 。这是一个关于 `5:[]` 的语法糖。`:` 的左侧是一个值，而右侧则是一个列表，且这个列表是空的。那么 `[4,5]` 呢？去掉语法糖后就是 `4:(5:[])` 。观察第一个 `:` ，其左侧为一个元素，而右侧是一个列表 `5:[]` 。以此类推，`3:(4:(5:6:[]))` ，可以写作 `3:4:5:6:[]` （因为 `:` 是右结合的）或者是 `[3,5,6,7]` 。

我们可以说一个列表既能是一个空列表或者是一个与其它列表（无论是否为空列表）用 `:` 所关联的元素。

现在让我们用代数数据类型来实现我们自己的列表！

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
```

这读起来就像是列表的定义，要么是一个空列表要么是一个头与别的列表的组合。如果感到困惑，那么可以尝试用 record syntax 来理解：

```
data List a = Empty | Cons { listHead :: a, listTail :: List a} deriving (Show, Read, Eq, Ord
  )
```

这里可能也会对 `Cons` 构造函数感到困惑。*cons* 就是 `:` ，在列表中，`:` 实际上就是一个构造函数，其接受一个值与另一个列表，并返回一个列表。尝试一下：

```
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))
```

我们可以只用特殊字符来定义函数，这样它们就会自动获得中置的性质。同样可以使用在构造函数上，因为它们也是返回一个数据类型的函数。看看这个：

```
infixr 5 :-:
data List a = Empty | a :-: (List a) deriving (Show, Read, Eq, Ord)
```

首先我们注意到一个新的语法结构，fixity 声明。当我们定义函数为操作符时，我们可以使用 fixity 来指定（但是不是必须的）。一个 fixity 指定了是左结合还是右结合的，同时还有优先级。比如 `*` 的 fixity 是 `infixl 7 *` ，而 `+` 的 fixity 是 `infixl 6` ，说明它们都是左结合的。

现在可以这样 `a :-: (List a)` 而不用 `Cons a (List a)` 。那么我们的列表可以这样写：

```
ghci> :{
ghci| infixr 5 :-:
ghci|
ghci| data List a = Empty | a :-: (List a) deriving (Show, Read, Eq, Ord)
ghci| :}
ghci> 3 :-: 4 :-: 5 :-: Empty
3 :-: (4 :-: (5 :-: Empty))
ghci> let a = 3 :-: 4 :-: 5 :-: Empty
ghci> 100 :-: a
100 :-: (3 :-: (4 :-: (5 :-: Empty)))
```

现在来定义一个两列表加法的函数。这里是 `++` 在普通列表中的定义：

```
infixr 5
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

我们偷过来使用，这里命名为 `.++` ：

```
infixr 5 .++

(.++) :: List a -> List a -> List a
Empty .++ ys = ys
(x :-: xs) .++ ys = x :-: (xs .++ ys)
```

测试：

```
ghci> let a = 3 :-: 4 :-: 5 :-: Empty
ghci> let b = 6 :-: 7 :-: Empty
ghci> a .++ b
(:-:) 3 ((:-:) 4 ((:-:) 5 ((:-:) 6 ((:-:) 7 Empty))))
```

注意这里是如何在 `(x :-: xs)` 上模式匹配的，这是因为模式匹配本身是去匹配构造函数。这里可能匹配 `:-:` 是因为它是我们自设的列表类型的构造函数，我们也可以匹配 `:` ，因为它是 Haskell 列表内置的构造函数。

现在我们开始实现一个**二叉搜索树 binary search tree**。

从 `Data.Set` 与 `Data.Map` 而来的 sets 与 maps 就是用了树来实现的，不过不是普通的二叉搜索树，而是平衡二叉树。

一个树可以是空树或者是一个元素，其中包含了一些信息以及两个树。这听起来很适合代数数据类型！

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a) deriving (Show, Read, Eq)
```

接下来是两个构建树的函数：

```
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)
```

接下来是查看元素是否存在于树中的函数：

```
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
```

测试：

```
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
  EmptyTree EmptyTree) (Node 8 EmptyTree EmptyTree))
```

在这个 `foldr` 中，`treeInsert` 是被接受的函数，`EmptyTree` 是初始的累加器，而 `nums` 则是需要遍历的列表。

再来查看元素：

```
ghci> 8 `treeElem` numsTree
True
ghci> 100 `treeElem` numsTree
False
ghci> 1 `treeElem` numsTree
True
ghci> 10 `treeElem` numsTree
False
```

## Typeclasses 102

这里是标准库中 `Eq` 的定义：

```
class Eq a where
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
x == y = not (x /= y)
x /= y = not (x == y)
```

哇！新的语法和关键字！不要紧，马上会弄清楚。首先当写下 `class Eq a where` 时，意味着正在定义一个新的名为 `Eq` 的 typeclass。这里的 `a` 是类型变量，意为 `a` 是任何定义实例时的类型，并不一定要叫做 `a` ，只需要是小写字母。接着定义了几个函数，并不一定要实现函数体，只需要指定这些函数的类型声明。

一旦有了一个类，就可以实现该类的类型实例，首先是一个类型：

```
data TrafficLight = Red | Yellow | Green
```

接着来实现 `Eq` 的实例：

```
instance Eq TrafficLight where
  Red == Red = True
  Green == Green = True
  Yellow == Yellow = True
  _ == _ = False
```

这里使用了 *instance* 关键字。与定义类时不同的是，我们用 `TrafficLight` 这个时机类型替换了参数 `a` 。

由于 `==` 是用 `/=` 来定义的，同样的 `/=` 也是用 `==` 来定义的。因此我们只需要在实例的定义中复写其中一个就好了（我们复写了 `==` ）。这样叫做定义了一个最小完整定义，即是能让类型符合类行为所需的最小实例化函数数量。而如果 `Eq` 的定义像是这样：

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

那么我们在定义实例的时候就必须将两个函数都实例化，因为 Haskell 并不知道这两个函数是怎么关联在一起的。所以这里的最小完整定义是 `==` 与 `/=` 。

接下来是 `Show` 的实例。

```
instance Show TrafficLight where
  show Red = "Red light"
  show Yellow = "Yellow light"
  show Green = "Green light"
```

测试：

```
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]
```

当我们使用 `deriving` 关键字来自动生产 `Eq` ，那么效果是一样的；但是生产 `Show` 的话，Haskell 会将值构造函数转换成字符串。我们需要的类似于 `"Red light"` 这样的字符串，因此才需要手动进行实现。

我们也可以把 typeclass 定义成其它 typeclass 的子类。`Num` 的类声明很冗长，我们先看一个雏形：

```
class (Eq a) => Num a where
...
```

正如我们之前提到过的，我们可以在很多地方加上类约束。这里是在 `class Num a where` 中的 `a` 上，加上就必须满足 `Eq` 实例的限制。也就是说我们在定义一个类型为 `Num` 之前，必须先为其定义 `Eq` 的实例。在某个类型被视为 `Number` 之前，必须先被定义可以比较，这其实很合理。这就是子类在做的：帮助类声明加上限制。也就是说，在定义 typeclass 中的函数体是，我们可以默认 `a` 是属于 `Eq` 的，因此能使用 `==` 。

那么 `Maybe` 或者列表类型是如何创建 typeclasses 的实例的呢？那么 `Maybe` 为何与比如说 `TrafficLight` 不同，前者自身并不是一个具体类型而是一个接受一个类型参数（如 `char` ）的类型构造函数用于生产具体类型（如 `Maybe Char` ）。让我们再看一下 `Eq` typeclass：

```
class Eq a where
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
x == y = not (x /= y)
x /= y = not (x == y)
```

从类型声明中我们可以看到 `a` 用作于一个具体类型，因为所有在函数中的类型都必须是具体的（记住，你不可能让一个函数的类型是 `a -> Maybe` ，但是可以是 `a -> Maybe a` 或

是 `Maybe Int -> Maybe String` )。这也就是为什么我们不能这样做:

```
instance Eq Maybe where
  ...
```

正如我们所见，`a` 必须是一个具体类型，但是 `Maybe` 并不是。后者是一个类型构造函数接受一个参数并生产具体类型。如果是每个类型都实现一番如 `instance Eq (Maybe Int) where` 以及 `instance Eq (Maybe Char) where` 等等，就会很冗长。因此我们这样做:

```
instance Eq (Maybe m) where
  Just x == Just y = x == y
  Nothing == Nothing = True
  _ == _ = False
```

这就像是在说我们希望让 `Maybe something` 所有类型都实现 `Eq` 实例。虽然 `Maybe` 不是具体类型，`Maybe m` 却是。通过指定类型参数（`m` ，小写字母），我们可以让所有以 `Maybe m` 形式的类型（`m` 是任意类型）称为 `Eq` 的实例。

不过还有一个问题。我们使用了 `==` 在 `Maybe` 的内容上，但是并没有确保 `Maybe` 所包含的可以使用 `Eq` ！这就是为什么我们必须像这样修改我们的实例声明:

```
instance (Eq m) => Eq (Maybe m) where
  Just x == Just y = x == y
  Nothing == Nothing = True
  _ == _ = False
```

我们必须添加一个类约束!

最后一件事，如果想要看 typeclass 的实例，仅需在 GHCI 中输入 `:info YourTypeClass` 。

## 一个 yes-no typeclass

尝试一下 JavaScript 那样的一切皆可布尔值的行为，首先是一个类声明。

```
class YesNo a where
  yesno :: a -> Bool
```

很简单。`YesNo` typeclass 定义一个函数，该函数接受一个类型作为值，其可以被认作是存储了真假的信息。注意这里在函数使用的 `a` 必须是一个具体类型。

接着来定义一些实例，首先是数值:

```
instance YesNo Int where
  yesno 0 = False
  yesno _ = True
```

其次是列表:

```
instance YesNo [a] where
  yesno [] = False
  yesno _ = True
```

再是布尔值：

```
instance YesNo Bool where
  yesno = id
```

等等，什么是 `id` ？它是一个标准库的函数，接受一个参数并返回相同的东西，这里正是我们所要的。

接下来是 `Maybe a` 实例：

```
instance YesNo (Maybe a) where
  yesno (Just _) = True
  yesno Nothing = False
```

我们不需要一个类约束，因为我们不需要对 `Maybe` 中内容的做任何假设。我们只需要在 `Just` 值时为真，而在 `Nothing` 值时为假。

对于用上一章定义过的 `Tree` 类型：

```
instance YesNo (Tree a) where
  yesno EmptyTree = False
  yesno _ = True
```

以及 `TrafficLight` ：

```
instance YesNo TrafficLight where
  yesno Red = False
  yesno _ = True
```

测试：

```
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
```

现在创建一个函数来模拟 if 声明：

```
yesnoIf :: (YesNo y) => y -> a -> a -> a
yesnoIf yesnoVal yesResult noResult = if yesno yesnoVal then yesResult else noResult
```

非常的直接，测试一下

```
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!"
```

## 函子 typeclass

现在让我们学习 `Functor` typeclass，即代表可以被映射的事物。

首先来看一下 `Functor` typeclass 的实现：

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

我们可以看到定义了一个函数，`fmap`，同时改函数不提供任何默认实现。它的类型比较有趣，到现在为止的所有 typeclasses 的定义中，类型变量都是具体变量，例如 `(==) :: (Eq a) => a -> a -> Bool` 中的 `a`。但是现在，`f` 并不是一个具体类型（一个可以存储值的类型，例如 `Int` `Bool` 或 `Maybe String`），而是一个接受一个类型参数的类型构造函数。

回想一下 `map` 的类型签名：`map :: (a -> b) -> [a] -> [b]`。

也就是说接受一个函数，将某种类型的列表转换成另一种类型的列表。实际上 `map` 就是一个 `fmap` 不过仅能作用于列表上。下面是列表的 `Functor` typeclass 的实例：

```
instance Functor [] where
  fmap = map
```

仅仅如此！注意这里为什么没有写 `instance Functor [a] where` 是因为从 `fmap :: (a -> b) -> f a -` 可知，`f` 必须是一个接受一个类型的类型构造函数，而 `[a]` 已经是一个具体类型了（一个拥有任意值的列表），而 `[]` 是一个类型构造函数，接受一个类型并生产出类型入 `[Int]`，`[String]` 或 `[[String]]`。

由于对于列表 `fmap` 就是 `map`，我们可以得到相同的结果：

```
map :: (a -> b) -> [a] -> [b]
ghci> fmap (*2) [1..3]
[2,4,6]
```

```
ghci> map (*2) [1..3]
[2,4,6]
```

那么如果将 `fmap` 或 `map` 应用在空列表上呢？当然是返回一个空列表，仅仅是将一个类型为 `[a]` 的空列表转换成了 `[b]` 的空列表而已。

行为像是盒子的类型都可以是函子。下面是 `Maybe` 的函子实例：

```
instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap f Nothing = Nothing
```

再次注意这里并没有写 `instance Functor (Maybe m) where` 而是 `instance Functor Maybe where` 。`Functor` 想要的是接受一个类型的类型构造函数而不是一个具体类型。

测试：

```
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
```

另一个可以被映射的类型是 `Tree a` 类型：

```
instance Functor Tree where
  fmap f EmptyTree = EmptyTree
  fmap f (Node x leftsub rightsub) = Node (f x) (fmap f leftsub) (fmap f rightsub)
```

测试：

```
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 EmptyTree))
  )) EmptyTree
```

棒！那么如果是 `Either a b` 呢？它可以是个函子吗？`Functor` typeclass 想要的是一个仅接受一个类型参数的类型构造函数，而 `Either` 有两个！让我们看看标准库的实现：

```
instance Functor (Either a) where
  fmap f (Right x) = Right (f x)
  fmap f (Left x) = Left x
```

也就是说仅处理 `Right` 值。为什么呢？回想一下 `Either a b` 类型是如何定义的：

```
data Either a b = Left a | Right b
```

从观察 `fmap` 的类型可以知道，当它运作在 `Either` 上的时候，第一个类型参数必须固定，而第二个可以改变，而其中第一个参数正好就是 `Left` 用的。

继续用盒子比喻，可以把 `Left` 想做是空的盒子，在它旁边写上错误消息，说明为什么它是空的。

`Data.Map` 中的 Map 也可以被定义成函子，像是 `Map k v` 的情况下，`fmap` 可以用 `v -> v'` 这样一个函数来映射 `Map k v` ，得到 `Map k v'` 。

## 类型的类型以及一些 type-foo

类型构造函数接受另一个类型作为参数，最终生产出具体类型。这就像是函数，接受参数并生产值。我们看到了类型构造函数可以被偏应用（`Either String` 是一个类型，接受一个类型并生产一个具体类型，例如 `Either String Int` ），正如函数那样。本节我们将正式的看到类型是如何被应用到别的类型构造函数上。

像是 `3` ， `"YEAH"` 或是 `takeWhile` 的值它们都有自己的类型（函数也是值的一种，我们可以将其传来传去），类型就像是一个标签，值会带着它，这样我们就可以推测出其性质。但是类型也有它们的标签，叫做 **kind**。kind 是类型的类型。

那么 kind 可以用来做什么？让我们现在 GHCI 上使用 `:k` 命令来测试一下：

```
ghci> :k Int
Int :: *
```

一个星号？一个 `*` 代表的这个类型是具体类型，一个具体类型是没有任何类型参数的，而值只能属于具体类型。 `*` 读作 star 或是 type。

在看看 `Maybe` 的 kind：

```
ghci> :k Maybe
Maybe :: * -> *
```

`Maybe` 的类型构造函数接受一个具体类型(如 `Int` )然后返回一个具体类型,如 `Maybe Int` 。这就是 kind 告诉我们的信息。就像 `Int -> Int` 代表这个函数接受一个 `Int` 并返回一个 `Int` 。 `* -> *` 代表这个类型构造函数接受一个具体类型并返回一个具体类型。

```
ghci> :k Maybe Int
Maybe Int :: *
```

正如预计那样，将 `Maybe` 应用至类型参数后会得到一个具体类型（这就是 `* -> *` 的意思）。

再看看别的：

```
ghci> :k Either
Either :: * -> * -> *
```

这告诉我们 `Either` 接受两个具体类型作为参数，并构造出一个具体类型。它看起来也像是一个接受两个参数并返回值的函数类型。类型构造函数是可以柯里化的，所以我们也能将其进行偏应用。

```
ghci> :k Either String
Either String :: * -> *
ghci> :k Either String Int
Either String Int :: *
```

在 `Either` 实现 `Functor` typeclass 的实例时，我们必须偏应用它因为 `Functor` 希望接受的类型只有一个，而 `Either` 却有俩。也就是说，`Functor` 希望类型的类型是 `* -> *`，因此我们必须偏应用 `Either` 来获取一个 kind 为 `* -> *` 而不是 `* -> * -> *`。

现在来定义一个新的 typeclass：

```
class Tofu t where
  tofu :: j a -> t a j
```

这看起来很奇怪. 让我们看一下它的 kind，因为 `j a` 看做是一个值的类型被 `tofu` 函数作为入参，`j a` 就必须要有 `*` 作为 kind。我们假设 `a` 是 `*`，这样可以推导出 `j` 的 kind 是 `* -> *`。我们知道 `t` 必须生产一个具体类型，且其接受两个类型。也知道 `a` 的 kind 是 `*` 同时 `j` 的 kind 是 `* -> *`，可推导 `t` 的 kind 为 `* -> (* -> *) -> *`。因此该函数接受一个具体类型（`a`），一个接受一个具体类型的类型构造函数（`j`），并生成一个具体类型。

好，现在让我们创建一个 kind 为 `* -> (* -> *) -> *` 的类型：

```
data Jige a b = Jige {jigeField :: b a} deriving (Show)
```

那么我们是怎么知道这个类型的 kind 是 `* -> (* -> *) -> *` 的呢？ADT 中的字段用于存储值，因此他们必须是 `*` kind。假设 `a` 是 `*`，意味着 `b` 接受一个类型参数，因此其 kind 为 `* -> *`。现在我们知道了 `a` 与 `b` 的 kind，同时因为它们都是 `Frank` 的参数，而我们知道 `Frank` 的 kind 是 `* -> (* -> *) -> *`，第一个 `*` 代表着 `a`，`(* -> *)` 代表着 `b`。检查一下：

```
ghci> :t Frank {frankField = Just "HAHA"}
Frank {frankField = Just "HAHA"} :: Frank String 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 []
```

由于 `frankField` 拥有 `a b` 样式的类型，其至必须也遵循这样的样式。因此它们可以是 `Just "HAHA"`，即类型为 `Maybe [Char]`，或是 `["Y","E","S"]`，即类型为 `[Char]`。我们可以看到 `Frank` 值的类型匹配 `Frank` 的 kind。`[Char]` 的 kind 为 `*`，`Maybe` 的 kind 为 `* -> *`。因为必须要有一个值，它必须是具体类型，这样才能被全应用，也就是说每个 `Frank blah blah` 值的 kind 都是 `*`。

将 `Frank` 做 `Tofu` 的实例很简单。我们知道 `tofu` 接受一个 `j a`（一个例子就是 `Maybe Int`）并返回一个 `t a j`。因此用 `Frank` 替换 `j`，返回类型便是 `Frank Int Maybe`。

```
instance Tofu Frank where
tofu = Frank
```

```
ghci> tofu (Just 'a') :: Frank Char Maybe
Frank {frankField = Just 'a'}
ghci> tofu ["Hello"] :: Frank [Char] []
Frank {frankField = ["Hello"]}
```

这并不好用，不过至少我们做了练习。现在让我们看看下面的类型：

```
data Barry t k p = Barry {yabba :: p, dabba :: t k}
```

现在想使其称为 `Functor` 的实例。`Functor` 希望的是 `* -> *` ，但是 `Barry` 的 kind 并不是这样，它接受三个参数 `something -> something -> something -> *` 。可以说 `p` 是一个具体类型，因此它的 kind 是 `*` ，而对于 `k` ，我们假设是 `*` ，所以 `t` 的 kind 就会是 `* -> *` 。现在我们把这些代入 something，所以 kind 就变成 `(* -> *) -> * -> * -> *` 。在 GHCI 上检查一下：

```
ghci> :k Barry
Barry :: (* -> *) -> * -> * -> *
```

看起来没问题。现在将这个类型成为 `Functor` ，我们需要偏应用前两个类型参数，剩下的就是 `* -> *` 。这就意味着该实例的声明为：`instance Functor (Barry a b) where` ，如果我们看 `fmap` 针对 `Barry` 的类型，那么将会是 `fmap :: (a -> b) -> Barry c d a -> Barry c d b` ，因为这里替换了 `Functor` 的 `f` 为 `Barry c d` 。而 `Barry` 的第三个类型参数是对于任意类型的，所以不需要涉及它：

```
instance Functor (Barry a b) where
  fmap f (Barry {yabba = x, dabba = y}) = Barry {yabba = f x, dabba = y}
```

很好，我们将 `f` 应用到了第一个字段。

# 9 Input and Output

## Hello, world!

在文件编辑器中敲下：

```
main = putStrLn "hello, world"
```

我们定义了一个 `main` ,其调用了一个名为 `putStrLn` 的函数,其参数为 `"hello, world"` 。将其保存为 `helloworld.hs` 。打开终端并 cd 到该文件处，输入命令：

```
$ ghc --make helloworld
[1 of 1] Compiling Main             ( helloworld.hs, helloworld.o )
Linking helloworld ...
```

执行：

```
$ ./helloworld
hello, world
```

我们第一个编译的程序好了！

测试一下我们所写的。首先是查看 `putStrLn` 的类型：

```
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t putStrLn "hello, world"
putStrLn "hello, world" :: IO ()
```

我们可以这样解读 `putStrLn` ： `putStrLn` 接受一个字符串并返回一个 **I/O action**，其返回值的类型是 `()` （也就是空元组，或是 unit 形态）。一个 I/O action 就是在执行时会造成副作用的动作，常常指读取输入或输出到屏幕，同时也代表返回某些值。

那么 I/O action 会在什么时候触发呢？这就是 `main` 的作用。一个 I/O action 会在我们把它绑定到 `main` 这个名字并执行程序的时候触发。

把整个程序限制在只能有一个 I/O action 看起来有限制，这就是为什么需要 *do* 语法将所有 I/O action 粘合成一个。例如：

```
main = do
  putStrLn "Hello, what's your name?"
  name <- getLine
  putStrLn ("Hey " ++ name ++ ", you rock!")
```

有趣，新的语法！这读起来非常像是命令式编程。注意我们写了 *do* 之后接着一连串指令，就像是在写命令式编程那样。

正因如此，`main` 的类型签名总会是 `main :: IO something` ，这里的 `something` 就是写具体类型。

我们之前没有遇到过的情况是第三行，即 `name <- getLine` 。它看起来像是从输入中读取一行，并存储至变量 `name` 中。让我们测试一下 `getLine` 的类型：

```
ghci> :t getLine
getLine :: IO String
```

好的，`getLine` 是一个 I/O action，其包含了 `String` 返回值类型。这很合理，因为它将等待用户在终端的输入，然后将输入转换为一个字符串。那 `name <- getLine` 是什么呢？你可以将这段代码读作：**执行一个 I/O action，`getLine` 将其结果绑定到 `name` 这个名字**。`getLine` 的类型是 `IO String` ，因此 `name` 的类型也就是 `String` 。当我们从 I/O action 中拿取数据时，就一定同时要在另一个 I/O action 中。这就是 Haskell 如何漂亮的区分 pure 和 impure 程序的方法。`getLine` 在这样的意义下是 impure 的，因为执行两次的时候它没法保证会返回一样的值，这就是为什么它需要在一个 `IO` 的类型构建函数中，这样才能在 I/O action 中取出数据。任何一段程序一旦依赖 I/O 数据，那么则会被视为 I/O 代码（原文用 tainted）。

当提到污染，并不是说 I/O action 所提供的返回不能在 pure 代码中使用。只要我们绑定它到一个名字，我们便可以暂时使用它。也就是说 `name <- getLine` 的 `name` 就是一个普通的字符串。

让我们看以下代码是否合法：

```
nameTag = "Hello, my name is " ++ getLine
```

该代码不合法是因为 `++` 需要两个参数都是同样参数类型的列表，左参数为 `String` （或 `[Char]` ），而 `getLine` 则是 `IO String` 。显然不能将两者合并。要从 `IO String` 中获取值只能在 I/O action 内部做 `name <- getLine` 。换言之，如果要处理 impure 数据，那么就需要再 impure 环境中做。

每个 I/O action 执行都有一个封装后的返回。那么之前的代码同样也可以这么写：

```
main = do
  foo <- putStrLn "Hello, what's your name?"
  name <- getLine
  putStrLn ("Hey " ++ name ++ ", you rock!")
```

然而 `foo` 只会得到一个 `()` 值，因此这么做并无实际意义。注意最后的 `putStrLn` 并没有绑定至任何事物。这是因为在一个 *do* 代码块中，`最后的action 不能绑定至一个名称` ，如前面两行那样。我们在之后讲解 Monad 的时候会说明为什么。

除了最后一行，*do* 代码块中的每一行皆可进行绑定。因此 `putStrLn "BLAH"` 可以被写成 `_ <- putStrLn "BLAH"` ，不过这并没大用，因此不如不写。

初学者可能会这么想：

```
name = getLine
```

记住，从 I/O action 中取值，必须将其置于其它 I/O action 内，并使用 `<-` 将其结果绑定至一个名称。

我们能够在 *do* 代码块中使用 *let* 绑定，就如在列表表达式中那样：

```haskell
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?"
```

注意在 `do` 代码块中的 I/O actions，同时注意 `let` 与其名字，缩进在 Haskell 中不会被无视。

接下来是一个持续读取输入行，并在同一行打印翻转后单词的程序。程序会在输入空行后结束：

```haskell
main = do
line <- getLine
if null line
  then return ()
  else do
    putStrLn $ reverseWords line
    main

reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

首先我们来看一下 `reverseWords` ,这个普通的函数假如接受了个字符串 `"hey there man"` ,那么先调用 `words` 来参数一个字的列表,比如 `["hey", "there", "man"]` 。接着将 `reverse` 映射至列表，获得 `["yeh", "ereht", "nam"]` ，接着再通过 `unwords` 将列表转换为一个字符串 `"yeh ereht nam"` 。这里用到了函数组合，如果没有函数组合的话，我们需要这样做 `reverseWords st = unwords (map reverse (words st))` 。

我们来看一下在 *else* 中发生了什么事。由于我们必须要一个 I/O action 在 *else* 之后，我们使用一个 *do* 代码块将两个 I/O action 整合成一个。我们也可以这么写：

```haskell
else (do
  putStrLn $ reverseWords line
  main)
```

这样可以更显式的将 *do* 代码块认做是一个 I/O action，不过就是更丑了点。在 *do* 代码块中，调用 `reverseWords` 从 `getLine` 而来的行，接着输出至终端。之后就是递归，因为 `main` 自身就是一个 I/O action。也就意味着又回到了程序一开头。

那么当 `null line` 为真时呢？我们看到 `then return ()` 。**在 Haskell 中的 `return` 与其他语言的 `return` 完全不同！**它们只是有相同的名字，这会迷惑很多人，但是实际上大相径

庭。在命令式语言中，`return` 通常结束一个方法的执行，并将结果返回给调用者。在 Haskell（特别是 I/O action 中），则是利用某个 pure 值制造出 I/O action。所以在 I/O 的情况下，`return "haha"` 的类型是 `IO String` 。将 pure 值包成 I/O action 有什么意义呢？这是因为一定要 I/O action 来承载空输入行的情况。因此使用 `return ()` 做了一个没什么用的 I/O action。

在 I/O *do* 代码块中放一个 `return` 并不会结束执行。下面这个程序就会执行到底：

```
main = do
  return ()
  return "HAHAHA"
  line <- getLine
  return "BLAH BLAH BLAH"
  return 4
  putStrLn line
```

所有的这些 `return` 都用于创造 I/O actions，而其并不做任何事情仅仅是封装一个结构，且该结果被抛弃，因为它们并没有绑定到一个名字上。`return` 可以与 `<-` 组合在一起用于绑定名称：

```
main = do
  a <- return "hell"
  b <- return "yeah!"
  putStrLn $ a ++ " " ++ b
```

可以看到，`return` 像是相反的 `<-` ，前者接受一个值，将其放入盒中，后者接受一个盒（并执行它）然后将值取出来，绑定至一个名称。不过这么做有点多余，特别是在 *do* 代码块中，可以使用 *let* 绑定名称：

```
main = do
  let a = "hell"
      b = "yeah"
  putStrLn $ a ++ " " ++ b
```

在 I/O *do* 代码块中需要 `return` 的原因有两个：一个是需要一个什么事都不做的 I/O action，或者是不希望这个 *do* 代码块形成的 I/O action 的结果值是这个代码块中的最后一个 I/O action。我们希望有一个不同的结果值，所以用 `return` 来做一个 I/O action 包装想要的结果放在 *do* 代码块的最后。

在讲下一节的文件之前，让我们看看有哪些实用的函数可以处理 I/O。

`putStr` 类似于 `putStrLn` ，前者不会换行：

```
main = do   putStr "Hey, "
            putStr "I'm "
            putStrLn "Andy!"
```

```
$ runhaskell putstr_test.hs
Hey, I'm Andy!
```

`putChar` 接受一个字符：

```
main = do   putChar 't'
            putChar 'e'
            putChar 'h'
```

```
$ runhaskell putchar_test.hs
teh
```

`print` 接受任意实现了 `Show` 的值（意味着知道如何将该值转换为一个字符串）：

```
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]
```

`getChar` 从输入中读取一个字符：

```
main = do
  c <- getChar
  if c /= ' '
      then do
          putChar c
          main
      else return ()
```

```
$ runhaskell getchar_test.hs
hello sir
hello
```

`when` 函数可以在 `Control.Monad` 中找到（通过 `import Control.Monad` ）。它在 *do* 代码块中很有意思，它像是一个流控声明，但其实它是一个普通函数。其接受一个布尔值以及一个 I/O action，如果改布尔值为真，返回同样的 I/O action；如果为假，则返回 `return ()` ，即什么都不做的 I/O action。下面是使用 `when` 来改写之前的程序：

```
import Control.Monad

main = do
    c <- getChar
    when (c /= ' ') $ do
        putChar c
        main
```

如你所见，它将 `if something then do some I/O action else return ()` 这样的模式封装了起来。

`sequence` 接受一个列表的 I/O action 并返回一个 I/O action。其类型签名 `sequence :: [IO a] -> IO [a` 。例：

```
main = do
  a <- getLine
  b <- getLine
  c <- getLine
  print [a,b,c]
```

等同于：

```
main = do
  rs <- sequence [getLine, getLine, getLine]
  print rs
```

由于对一个队列映射一个返回 I/O action 的函数，再 `sequence` 这个动作太常用了。所以 `mapM` 与 `mapM_` 被引入了。前者接受一个函数与一个列表，映射函数至列表接着再 sequences；后者一样，只不过丢弃返回值。在不关心 sequenced I/O action 的返回值时，我们使用后者。

```
ghci> mapM print [1,2,3]
1
2
3
[(),(),()]
ghci> mapM_ print [1,2,3]
1
2
3
```

`forever` 接受一个 I/O action 返回一个 I/O action，无限循环：

```
import Control.Monad
import Data.Char

main = forever $ do
    putStr "Give me some input: "
    l <- getLine
    putStrLn $ map toUpper l
```

`forM` 类似于 `mapM` ，不同之处在于参数调转了。为什么有用呢？一些创造性的 lambdas 与 *do* 的使用，让我们可以这样做：

```
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
```

```
$ runhaskell form_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
```

## 文件与流

现在让我们来看一下 `getContents` ，这是一个从标准输入中读取所有东西直到遇到结束符的 I/O action。其类型为 `getContents :: IO String` 。`getContents` 是一个 lazy I/O。当使用 `foo <- getContents` ，它不会一次性将所有输入读取存储至内存中然后绑定至 `foo` 。

在我们将一个程序中的输出导向另一个程序时，`getContents` 尤其的有用。我们现在来造一个文件：

```
I'm a lil' teapot
What's with that airplane food, huh?
It's so small, tasteless
```

现在让我们回忆一下之前介绍的 `forever` 函数。它提示用户输入一行，将输入改为大写，然后再次回到第一步：

```
import Control.Monad
import Data.Char

main = forever $ do
    putStr "Give me some input: "
    l <- getLine
    putStrLn $ map toUpper l
```

将其保存为 `capslocker.sh` 并进行编译。通过 unix 的 pipe 将 txt 文件喂给我们的程序：

```
$ 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
```

正如所见，将一个程序（上述案例为 *cat*）的输出 pipe 至另一个程序（*capslocker*）作为输入，是由 `|` 字符来完成的。

而这里的 `forever` 就是接受输入并转换后输出。这就是为什么使用 `getContents` 可以使得程序变得更简洁简短：

```
import Data.Char

main = do
  contents <- getContents
  putStr $ map toUpper contents
```

这里执行 `getContents` I/O action 并将其生产的字符串命名为 `contents` 。接着映射 `toUpper` 至字符串，再打印至终端。注意字符串的本质是列表，也是 lazy 的，而 `getContents` 是 I/O lazy 的，所有在打印出 capslocked 之前，它不会一次性读取所有内容并存储至内存。实际上会一行一行读取并输出 capslocked，这是因为输出才是真的需要输入数据的时候。

```
$ cat haiku.txt | ./capslocker
I'M A LIL' TEAPOT
WHAT'S WITH THAT AIRPLANE FOOD, HUH?
IT'S SO SMALL, TASTELESS
```

如果仅运行 *capslockder* 并尝试输入呢？

```
$ ./capslocker
hey ho
HEY HO
lets go
LETS GO
```

通过 Ctrl-D 退出。正如你所见，它将我们的输入一行一行的打印出来。当 `getContents` 的结果绑定至 `contents` ，它在内存中并不代表着一个真实字符串，更像是一个 promise，承诺它将最终输出字符串。当映射 `toUpper` 至 `contents` ，则同样是一个承诺，承诺将映射该函数至最终的内容。最后就是当 `putStr` 发生时，它告知之前的承诺：我需要一个大写的行！它还未有任何的行，而是告知 `contents` ：该从命令行中获取一行了。这才是 `getContents` 实际上从终端中读取并将这一行交给程序的时候，程序便将这一行用 `toUpper` 处理并交给 `putStr` ，`putStr` 则打印出它，之后 `putStr` 再说：我需要下一行，循环往复，直到读到结束符为止。

下面是一个只打印出少于十个字符行的程序：

```
main = do
  contents <- getContents
  putStr $ shortLinesOnly contents

shortLinesOnly :: String -> String
shortLinesOnly input =
  let allLines = lines input
      shortLines = filter (\line -> length line < 10) allLines
      result = unlines shortLines
  in result
```

我们尽量让程序的 I/O 部分简短，因为我们的程序本就该接受一些输入并根据输入打印出一些东西。

`shortLinesOnly` 函数工作如下：接受一个字符串 `"short\nlooooooooooooooong\nshort again"`，该字符串有三行，两行短，中间那行长。运行 `lines` 函数，转换成 `["short", "looooooooooooooong", "short again"]`，接着绑定至名称 `allLines`。该字符串列表被过滤，只留下少于 10 个字符的字符串，即 `["short", "short again"]`。最后，`unlines` 将列表转换成一个通过换行符分隔的字符串。

```
i'm short
so am i
i am a looooooooong line!!!
yeah i'm long so what hahahaha!!!!!!
short line
loooooooooooooooooooooooooong
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
```

我们将内容放至 *shortlines.txt* 中并 pipe 至编译好的 *shortlinesonly*，这样就得到短行了。

从输入中获取字符串，通过一个函数进行转换，然后再输出的这个模式太常见了，因此存在一个名为 `interact` 的函数专门用作此模式。

```
main = interact shortLinesOnly

shortLinesOnly :: String -> String
shortLinesOnly input =
  let allLines = lines input
      shortLines = filter (\line -> length line < 10) allLines
      result = unlines shortLines
   in result
```

当然可以用更少的代码来表达：

```
main = interact $ unlines . filter ((< 10) . length) . lines
```

能应用 `interact` 的情况有几种，一种是从输入 pipe 读取一些内容接着返回一些结果的程序；另一种是用户一行一行的输入，返回那一行运算后的结果，再接着读取玲一行。

接下来的程序是持续读取一行，接着告诉我们这一行是否为回文。我们可以用 `getLine` 来读取一行，告诉用户是否为回文，接着再执行 `main` 。不过使用 `interact` 的情况会更加的简单，只需要考虑转换函数。我们的需求中，替换的是每一行输出是否为 `"palindrome"` 或者 `"not a palindrome"` ，也就是说这个函数会转换 `"elephant\nABCBA\nwhatever"` 成 `"not a palindrome\npalindrome\nnot a palindrome"` 。

```
main = interact respondPalindromes

respondPalindromes contents =
  unlines
    ( map
        ( \xs ->
            if isPalindrome xs then "palindrome" else "not a palindrome"
        )
        (lines contents)
    )
  where
    isPalindrome xs = xs == reverse xs
```

用 point-free 进行改造：

```
respondPalindromes =
unlines
  . map
    (\xs -> if isPalindrome xs then "palindrome" else "not a palindrome")
  . lines
where
  isPalindrome xs = xs == reverse xs
```

非常的直观。首先它转换 `"elephant\nABCBA\nwhatever"` 为 `["elephant", "ABCBA", "whatever"]` ,接着映射一个 lambda 函数,得到 `["not a palindrome", "palindrome", "not a palindrome"]` 接着 `unlines` 转为一个字符串。测试：

```
$ runhaskell palindromes.hs
hehe
not a palindrome
ABCBA
palindrome
cookie
not a palindrome
```

尽管我们编写了一个转换输入的大字符串成为另一个大字符串的程序，但其表现的好像是一行一行做的。这是因为 Haskell 是 lazy 的，程序想要打印出第一行结果，它必须要现有第一行输入，一旦有了第一行输入，那么便打印出第一行结果。这里使用文件终止符来结束程序。

假设我们有这样一个文件：

```
dogaroo
radar
rotor
madam
```

将其保存至 `words.txt` 。再将其 pipe 至我们的程序：

```
$ cat words.txt | runhaskell palindromes.hs
not a palindrome
palindrome
palindrome
palindrome
```

到现在为止我们遇到的 I/O 都是读取或输出至终端，那么读取和写文件呢？其实我们已经做过这样的事儿了，我们可以想象终端就是一个文件，而写入和读取它们分别是 `stdout` 与 `stdin` 。

首先是一个文件：

```
Hey! Hey! You! You!
I don't like your girlfriend!
No way! No way!
I think you need a new one!
```

接着是程序：

```
import System.IO

main = do
  handle <- openFile "girlfriend.txt" ReadMode
  contents <- hGetContents handle
  putStr contents
  hClose handle
```

用程序读取文件：

```
$ runhaskell girlfriend.hs
Hey! Hey! You! You!
I don't like your girlfriend!
No way! No way!
I think you need a new one!
```

我们的程序拥有若干个 I/O actions 被整合在一个 *do* 代码块中。首先是 `openFile` 这个函数，其类型签名为：`openFile :: FilePath -> IOMode -> IO Handle` ，即 `openFile` 接受一个文件路径，以及一个 `IOMode` ，返回一个将会打开文件并将文件关联到句柄的 I/O action。

`FilePath` 是 `String` 的类型别名：

```
type FilePath = String
```

而 `IOMode` 的定义如下：

```
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

`hGetContents` 接受一个 `Handle`（这样才能知道从哪里获取内容）并返回一个 `IO String` – 一个包含了文件内容的 I/O action。该函数类似于 `getContents` ，不同之处在于 `getContents` 会自动的从标准输入中读取（也就是终端），而 `hGetContents` 接受一个文件句柄，该句柄则是告诉它去哪里读取文件。同样的，`hGetContents` 不会一次性读取文件并存储于内存中，而是按需读取。这样就很酷，因为我们可以将 `contents` 视作文件的所有内容，且并没有全部加载至内存中。因此如果有一个很大的文件，使用 `hGetContents` 并不会打爆内存，而是按需读取。

注意这里用于识别文件的句柄与文件内容的概念上的区别。前者是用于区分文件的依据，如果把整个文件系统想象成一本厚厚的书，每个文件分别是其中的章节，handle 就像是书签标记了正在阅读（或写入）的章节，而内容则是章节本身。

`putStr contents` 将内容打印至标准输出，接着是 `hClose` ，其接受一个句柄返回一个 I/O action 关闭文件。

函数 `withFile` ,其类型签名为 `withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a` ，是另外一种方式，它接受一个文件路径，一个 `IOMode` ，以及一个接受句柄并返回 I/O action 的函数。这个返回的 I/O action 函数将会打开文件，处理并关闭文件。

```
main = do
withFile
  "girlfriend.txt"
  ReadMode
  ( \handle -> do
      contents <- hGetContents handle
      putStr contents
  )
```

下面是自己实现的 `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
```

我们知道返回值会是一个 I/O action，因此我们可以由 *do* 开始。首先打开一个文件，得到一个句柄，接着将 `handle` 应用至我们的函数并重新拿到一个处理事情的 I/O action。将此 I/O action 绑定至 `result` ，关闭句柄接着 `return result` 。`return` 封装 `f` 的结果进 I/O

action，这样 I/O action 中就包含了 `f handle` 得到的结果。如果 `f handle` 返回的是一个从标准输入读取行，并写到文件然后返回读取的行数的 I/O action，在 `withFile'` 的情况中，最后的 I/O action 就会包含读入的行数。

`hGetContents` 类似 `getContents` ,还有 `hGetLine` ,`hPutStr` ,`hPutStrLn` ,`hGetChar` 等等。这些 *h* 开头的函数都是接受一个句柄作为参数，并操作某指定的文件（而不是标准输出）。

加载文件然后将它们的内容视为字符串这件事非常的常见，因此有三个函数能给我们很大的帮助：

`readFile` 的类型签名是 `readFile :: FilePath -> IO String` ，它接受一个文件路径，返回一个 I/O action，该 I/O action 将会读取文件（当然是 lazy 的），并将文件内容作为字符串绑定至某名称：

```
main = do
  contents <- readFile "girlfriend.txt"
  putStr contents
```

由于我们拿不到句柄，所以我们无法关闭文件，而 Haskell 的 `readFile` 在背后帮我们做了这件事。

`writeFile` 的类型签名是 `writeFile :: FilePath -> String -> IO ()` ，它接受一个文件路径以及需要写入的字符串，返回一个真实写入文件的 I/O action。如果文件存在，则会清空内容再写入：

```
import System.IO
import Data.Char

main = do
  contents <- readFile "girlfriend.txt"
  writeFile "girlfriendcaps.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` 的类型签名与 `writeFile` 一样，只不过前者在文件存在时会直接在已有内容尾部写入。

```
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
```

之前提到过的 `contents <- hGetContents handle` 并不会将文件一次性读入内存，所以这么写

```
main = do
withFile
  "something.txt"
  ReadMode
  ( \handle -> do
      contents <- hGetContents handle
      putStr contents
  )
```

实际上像是将文件 pipe 到标准输出。就像是可以把列表想象成流那样，文件也可以是流，即每次读一行再打印到终端上。对于文本文件而言，默认的 buffer 通常以行划分，而对于二进制文件而言，buffer 则跟操作系统有关，即以 chunk 进行划分。

使用 `hSetBuffering` 来控制 buffer 的行为，该函数接受一个句柄以及一个 `BufferMode`，返回一个设置 buffer 行为的 I/O action。`BufferMode` 是一个枚举，有值：`NoBuffering`，`LineBuffering` 或 `BlockBuffering (Maybe Int)`（`Maybe Int` 代表一个 chunk 的字节，如果是 `Nothing` 则有系统决定）。`NoBuffering` 表示一次读一个字符（频繁访问磁盘，性能很差）。

使用 2048 字节的 chunk 读取：

```
main = do
withFile
  "something.txt"
  ReadMode
  ( \handle -> do
      hSetBuffering handle $ BlockBuffering (Just 2048)
      contents <- hGetContents handle
      putStr contents
  )
```

使用更大的 chunk 读取数据可以减少访问磁盘的次数，特别是通过网络来进行文件访问的时候。

我们也可以使用 `hFlush`，该函数接受一个句柄并返回一个 I/O action，该 I/O action flush 句柄关联文件的缓存。在使用行缓存的时候，缓存会在每次换行时 flush；而在使用块缓存

时，会在读取块完成后 flush；同样也会在关闭句柄时 flush。而 `hFlush` 则可以强制进行 flush，之后数据就能被其他程序看见。

以下是一个将 item 加入 todo list 的程序，现在加入移除 item 的功能，这里会用到新函数 `System.Directory` 以及 `System.IO` ：

```
import Data.List
import System.Directory
import System.IO

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:"
  putStr $ 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"
```

首先是将 `ReadMode` 下 `openFile` 返回的句柄绑定至 `handle` 。

其次是一个从 `System.IO` 而来的新函数 `openTempFile` ，它接受一个临时路径以及一个临时文件名，打开一个临时文件。这里使用 `"."` 作为临时路径，即当前地址；使用 `"temp"` 作为临时文件名。该函数返回一个 I/O action，用于创建临时文件，并在 I/O action 中返回一对值：临时文件的名称，以及一个句柄。我们其实可以打开一个名为 *todo2.txt* 的普通文件再进行处理，不过使用 `openTempFile` 是一个更好的实践，这样就不会覆盖任何文件了。

没有使用 `getCurrentDirectory` 获取当前路径后，再传递给 `openTempFile` ，这么做是因为 `.` 在类 unix 系统与 Windows 都代表着当前路径。

接下来是将 `todo.txt` 的内容绑定至 `contents` 。再是将字符串转为字符串列表，这时的 `todoTasks` 就变为 `["Iron the dishes", "Dust the dog", "Take salad out of the oven"]` 。使用 `zipWith` 将零至无穷的列表与字符串列表结合，返回 `["0 - Iron the dishes", "1 - Dust the dog"` 。再通过 `unlines` 将字符串列表转为一个大字符串，打印至终端。这里我们也可以用 `mapM putStrLn numberedTa` 。

接下来询问用户想要删除哪个 item，假设用户输入的是 1，那么 `numberString` 则被绑定上 `"1"` 这个字符串，需要用 `read` 将字符串转换成数字。

接下来就是 `Data.List` 所提供的 `delete` 与 `!!` 函数。`!!` 从列表中通过某索引，返回一个元素，而 `delete` 删除列表中第一个出现的元素，并返回删除后的列表。这里的 `(todoTasks !! number)` 返回的是 `"Dust the dog"` 。将删除元素后的新列表绑定至 `newTodoItems` ，接着通过 `unlines` 合并成一个字符串，再写入临时文件。这时旧文件并没有改变，而临时文件包含了旧文件中所有函数，除了已删除的那行。

最后就是关闭两个文件的句柄，通过 `removeFile` 移除旧文件，再通过 `renameFile` 将临时文件命名为 `todo.txt` 。

测试：

```
$ 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
```

## 命令行参数

如果想要写一个在终端运行的程序，处理命令行参数是不可避免的。而 Haskell 的标准库能让我们有效处理参数。

`System.Environment` 模块有两个很酷的 I/O actions。第一个是 `getArgs` ，其类型是 `getArgs :: IO [String]` ，它是一个拿去命令行参数的 I/O actions，并把结果放在字符串列表中；第二个是 `getProgName` ，其类型是 `getProgName :: IO String` ，它是一个包含了程序名称的 I/O action。

现在来看一下它们是如何工作的：

```
import Data.List
import System.Environment

```

```haskell
main = do
  args <- getArgs
  progName <- getProgName
  putStrLn "The arguments are:"
  mapM_ putStrLn args
  putStrLn "The program name is:"
  putStrLn progName
```

测试：

```
$ ./arg-test first second w00t "multi word arg"
The arguments are:
first
second
w00t
multi word arg
The program name is:
arg-test
```

现在来改造一下 *todo*，我们需要：

1. 查看任务
2. 新增任务
3. 删除任务

首先来构建一个分发关联列表。这些函数的类型将会是 `[String] -> IO ()` ，它们接受关联列表作为参数，并返回一个 I/O action，其中包含查看、新增以及删除：

```haskell
import Data.List
import System.Directory
import System.Environment
import System.IO

dispatch :: [(String, [String] -> IO ())]
dispatch = [("add", add), ("view", view), ("remove", remove)]
```

暂未定义 `main` ， `add` ， `view` 以及 `remove` ，那么首先是 `main`

```haskell
main = do
  (commands : args) <- getArgs
  let (Just action) = lookup command dispatch
  action args
```

首先我们获取参数并将它们绑定至 `(command : args)` ，这里的模式匹配意为第一个参数绑定至 `command` ，其余的参数绑定至 `args` 。

下一行是在分发列表中查找命令。由于 `"add"` 指向了 `add` ，那么得到 `Just add` 作为返回值。这里再次使用模式匹配从 `Maybe` 中提取出我们的函数。那么如果命令不在关联列表

中呢？返回的值是 `Nothing` ，而这里我们不太需要考虑失败的情况，因此模式匹配失败后程序便抛出异常并退出。

最后是调用 `action` 函数，并将剩余参数列表作为参数传递进去，并返回一个 I/O action。

棒极了！现在还剩下 `add` ， `view` 以及 `remove` 需要被实现。首先是 `add` ：

```
add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName $ todoItem ++ "\n"
```

接着是 `view` ：

```
view :: [String] -> IO ()
view [fileName] = do
  contents <- readFile fileName
  let todoTasks = lines contents
      numberedTasks = zipWith (\n line -> show n ++ " - " ++ line) [0 ..] todoTasks
  putStr $ unlines numberedTasks
```

最后来实现 `remove` ：

```
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
```

## 随机

在 `System.Random` 模块中，包含了所有我们所需的随机函数。首先是 `random` ，它的类型是 `random :: (RandomGen g, Random a) => g -> (a, g)` 。一些新的 typeclasses 出现了！ `RandomGen` typeclass 即可以当做随机的源，而 `Random` typeclass 则是一个接受随机值的类型。

使用 `random` 函数首先需要知道随机数生成器。 `System.Random` 模块提懂了一个很酷的名为 `StdGen` 的类型，其本身为一个 `RandomGen` typeclass 的实例。我们既可以手动创建一个 `StdGen` ，也可以告诉系统基于一些随机事物向我们提供一个。

使用 `mkStdGen` 函数可以手动创建一个随机数生成器，其类型为 `mkStdGen :: Int -> StdGen` 。它接受一个整数，并根据这个整数提供一个随机数生成器。

```
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)
```

为什么会这样？因为 `random` 函数可以返回一个任意属于 `Random` typeclass 类型的值，因此我们需要显式的告诉 Haskell 我们需要的是哪种类型，同样还要告诉随机数生成器返回的随机值类型。

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
```

```
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)
```

现在让我们模拟一下丢三个硬币(通过执行 `cabal install --lib random` 安装 `System.Random` 模块)：

```
import System.Random

threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen =
  let (firstCoin, newGen) = random gen :: (Bool, StdGen)
      (secondCoin, newGen') = random newGen :: (Bool, StdGen)
      (thirdCoin, newGen'') = random newGen' :: (Bool, StdGen)
   in (firstCoin, secondCoin, thirdCoin)
```

注意这里与原文不同的地方在于 `random` 之后显式声明了类型 `:: (Bool, StdGen)` ，这是为了粘贴去 GHCI 时避免 *ambiguous type* 错误（因为没有类型推导）；仅在文件中则不需要显式声明，因为函数签名可供 Haskell 进行类型推导。

如果想要丢更多的硬币，可以使用 `randoms` 函数，其接受一个生成器，返回一个无限列表：

```
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]
```

为什么 `randoms` 不另外多返回一个新的生成器呢？我们自己来实现一下 `randoms` 函数：

```
randoms' :: (RandomGen g, Random a) => g -> [a]
randoms' gen = let (value, newGen) = random gen in value : randoms' newGen
```

这是一个递归的定义。从现在的生成器中获取到一个随机数以及一个新的生成器，接着构建一个列表，该随机数位于列表头部，而列表其余部分则是新的生成器所生成的随机数。因为我们可能产生出无限的随机数，因此不能将一个新的生成器返回。

我们可以编写一个函数生成有限流以及新的生成器，像是这样：

```
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)
```

又是一个递归定义(与原文不同的是 `n` 加上了 `Eq` 约束,不然 `finiteRandoms 0 gen = ([], gen)` 将会无法对其进行类型推导)。对于 0 而言，我们仅返回一个空列表以及传入的生成器；对于其他的数而言，返回一个随机数以及一个新的生成器。首先是头，接着是由新的生成器所生成的 *n-1* 作为尾，最后返回整个列表以及生成完 *n-1* 个随机数后的生成器。测试：

```
% runhaskell ownRandom.hs 1 2
([-2108421029521926081],StdGen {unStdGen = SMGen 15138674219130149558 10905525725756348111})
```

`randomR` 则在给定范围内生成随机数，`randomR :: (RandomGen g, Random a) :: (a, a) -> g -> (a,`

```
ghci> randomR (1,6) (mkStdGen 359353)
(6,1494289578 40692)
ghci> randomR (1,6) (mkStdGen 35935335)
(3,1250031057 40692)
```

函数 `randomRs` 生成随机值的流，无需定义范围：

```
ghci> take 10 $ randomRs ('a','z') (mkStdGen 3) :: [Char]
"ndkxbvmomg"
```

你可能会问这一小节跟 I/O 有什么关系？我们尚未考虑到 I/O，现在的问题就是如果在真实程序中，我们总会返回相同的随机数，这并不好。这就是为什么 `System.Random` 提供了 `getStdGen` I/O action，其类型为 `IO StdGen` 。当程序启动后，程序会向系统要一个随机数生成器，并存储其为全局生成器。当将 `getStdGen` 绑定至某变量时，它会去获取这个全局随机数生成器。

以下是一个简单的生成随机字符串的程序：

```
import System.Random

main = do
  gen <- getStdGen
  putStr $ take 20 $ randomRs ('a', 'z') gen
```

```
% runhaskell randomString.hs
rfbhrwkgvbqedgvaxvxz
% runhaskell randomString.hs
fgibjkxmivziojyyrcoa
% runhaskell randomString.hs
puohzegfvvswyrqrsdbe
```

不过要小心，执行多次 `getStdGen` 会向系统请求同一个全局生成器，例如：

```
import System.Random

main = do
  gen <- getStdGen
  putStrLn $ take 20 (randomRs ('a','z') gen)
  gen2 <- getStdGen
  putStr $ take 20 (randomRs ('a','z') gen2)
```

会打印出同样的字符串两次！这里的解决方案是设置一个无限流，每次获取 20 个字符：

```
main = do
  gen <- getStdGen
  let randomChars = randomRs ('a', 'z') gen
      (first20, rest) = splitAt 20 randomChars
      (second20, _) = splitAt 20 rest
  putStrLn first20
  putStr second20
```

另一个方法就是使用 `newStdGen` action：

```
main = do
  gen <- getStdGen
  putStrLn $ take 20 $ randomRs ('a', 'z') gen
  gen' <- newStdGen
  putStrLn $ take 20 $ randomRs ('a', 'z') gen'
```

下面是一个猜数字的小程序：

```
-- import Control.Monad (when)
import Control.Monad (unless)
import System.Random

main = do
  gen <- getStdGen
  askForNumber gen

askForNumber :: StdGen -> IO ()
askForNumber gen = do
  let (randNumber, newGen) = randomR (1, 10) gen :: (Int, StdGen)
  putStr "Which number in the range from 1 to 10 am I thinking of? "
  numberString <- getLine
  -- when (not $ null numberString) $ do
```

```
15    unless (null numberString) $ do
16      let number = read numberString
17      if randNumber == number
18        then putStrLn "You are correct!"
19        else putStrLn $ "Sorry, it was" ++ show randNumber
20      askForNumber newGen
```

**Note**

如果用户输入了 `read` 读不了的东西(例如 `"haha"` ),程序会立刻崩溃。如果使用 `reads` 遇到错误输入是则会返回一个空列表；如果是正确输入则会返回一个包含了一个二元元组的单元素列表，

这里 `when` 用于查看用户输入的字符串是否为空,如果是则一个空 I/O action 的 `return ()` 被执行，程序结束。

另一种做法，用 `main` 做递归（与原文不同，使用了上述建议的 `reads` ，并默认 0 避免错误输入导致的崩溃）:

```
1  main = do
2  gen <- getStdGen
3  let (randNumber, _) = randomR (1, 10) gen :: (Int, StdGen)
4  putStr "Which number in the range from 1 to 10 am I thinking of? "
5  numberString <- getLine
6  unless (null numberString) $ do
7    let number = case reads numberString of
8          [(num, _)] -> num
9          _ -> 0
10   if randNumber == number
11     then putStrLn "You are correct!"
12     else putStrLn $ "Sorry, it was " ++ show randNumber
13   newStdGen
14   main
```

## 字节串

将文件处理成字符串有一个缺陷:它很慢。如我们所知，`String` 其实就是 `[Char]` ，`Char` 并没有一个固定的大小，因为需要若干字节才能表示一个字符。另外就是列表是 lazy 的，如果有一个 `[1,2,3,4]` 列表，它只会在有必要时才会进行计算，也就是说整个列表其实是一个 promise 的列表。当列表的第一个元素被强制计算（假设是打印），列表的剩余部分 `2:3:4:[]` 仍然是一个 promise 列表。

大多数时候这种负担并没什么，但是读取大文件并进行操作时就不一样了。这就是为什么 Haskell 有 **bytestrings**，其元素是一个字节（8 bits），而且与字符串相比，它们的 lazy 性质也不一样。

Bytestrings 有两种模式：strict 以及 lazy。Strict 模式位于 `Data.ByteString` 模块，没有涉及 promise；该模式的好处就是更少的费用开支，而坏处就是一次性加载至内存会导致内存占用过高。Lazy 模式 `Data.ByteString.Lazy` 则是另一种做法，它们被存储于 chunks 中（不是 Thunk），每个 chunk 的大小都是 64k。

使用它们需要 qualified 导入：

```
import Data.ByteString.Lazy qualified as B
import Data.ByteString qualified as S
```

`B` 拥有 lazy bytestring 的类型与函数，而 `S` 则是 strict。大多数时候我们用的是 lazy 的版本。

`pack` 函数的类型签名是 `pack :: [Word8] -> ByteString` ，意为接受一个 `Word8` 类型的字节列表，并返回一个 `ByteString` 。可以想象成接受一个 lazy 的列表，使其变成没那么 lazy，也就是对于 64K lazy。

那么 `Word8` 类型又是什么？它类似于 `Int` ，只是范围较小，介于 0-255 之间。它代表一个 8-bit 的数字，就像 `Int` 一样，它是属于 `Num` 这个 typeclass。例如我们知道 `5` 是多态的，它能够表现成任何数值类型，`Word8` 也能这样表现。

```
ghci> B.pack [99,97,110]
Chunk "can" Empty
ghci> B.pack [98..120]
Chunk "bcdefghijklmnopqrstuvwx" Empty
```

如你所见，通常无需担心 `Word8` ，因为类型系统可以使数值选择类型。如果使用一个大的数值，例如 `336` 作为 `Word8` ，它会包装成 `80` 。

我们把一部分数值打包成 `ByteString` ，使它们可以塞进一个 chunk 内，而 `Empty` 类似于 `[]` 那样的空列表用作于空的 `ByteString` 。

`unpack` 则是 `pack` 的反函数，即接受一个 bytestring 将其转换成字节列表。

`fromChunks` 接受一个 strict 模式的 bytestrings，将其转换成 lazy 模式；`toChunks` 则相反。

```
ghci> B.fromChunks [S.pack [40,41,42], S.pack [43,44,45], S.pack [46,47,48]]
Chunk "()*" (Chunk "+,-" (Chunk "./0" Empty))
```

这对于很多小的 strict bytestring 且不想将它们 join 起来（占用内存），是个不错的做法。

`cons` 是 `:` 的 bytestring 版本，它接受一个字节以及一个 bytestring，将前者放置在后者头部。它是 lazy 的，因此即使 bytestring 的第一个 chunk 不是满的，它也会添加一个 chunk。这也是为什么在插入很多字节的时候最好用 strict 的 `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" (Chunk
    ":" (Chunk ";" (Chunk "<" Empty))))))))))
ghci> foldr B.cons' B.empty [50..60]
Chunk "23456789:;<" Empty
```

如你所见 `empty` 创建一个空的 bytestring。可以看出来 `cons` 与 `cons'` 之间的区别了吗？通过 `foldr` 我们从一个空的 bytestring 开始，接着从右开始遍历数值列表，将每个数值放置在 bytestring 的头部。使用 `cons` 时，则是每个字节都作为一个 chunk，这显然不符合初衷。

bytestring 有一堆与 `Data.List` 中类似的函数，其中包括 `head` ，`tail` ，`init` ，`null` ，`length` ，`map` ，`reverse` ，`foldl` ，`foldr` ，`concat` ，`taskWhile` ，`filter` 等等。

同样也有表现的跟 `System.IO` 中一样的函数，只不过是 `Strings` 被替换成了 `ByteString` 。比如 `System.IO` 中的 `readFile` ，其类型为 `readFile :: FilePath -> IO String` ，而 bytestring 模块中的 `readFile` 其类型为 `readFile :: FilePath -> IO ByteString` 。注意这里使用的是 strict bytestring，读取文件时会一次性读取至内存！如果是 lazy 模式，则会读取成 chunks。

现在让我们写一个简单的程序，从命令行中接受两个文件名，将第一个文件中的内容拷贝至第二个。注意 `System.Directory` 有一个名为 `copyFile` 的函数，不过这里我们还是来实现一下：

```
import Data.ByteString.Lazy qualified as B
import System.Environment

main = do
  (fileName1 : fileName2 : _) <- getArgs
  copyFile fileName1 fileName2

copyFile :: FilePath -> FilePath -> IO ()
copyFile source dest = do
  contents <- B.readFile source
  B.writeFile dest contents
```

```
runhaskell byteStringCopy.hs todo.txt ../../todo.txt
```

当需要更好的性能来读取数据时，可以尝试用 bytestring。

## 异常

尽管有表达力强的类型来帮助失败的情景（`Maybe` ，`Either` 等），Haskell 仍然支持 exception，因为在 I/O 的场景下 exception 是比较合理的。在处理 I/O 的时候会有很多奇怪的事情发生，环境并不能被信赖。比如说打开文件，文件有可能被锁，也有可能被移除了，甚至是硬盘都没了，因此直接跳到处理错误的代码是很合理的。

我们知道 I/O 代码抛出异常是合理的，那么 pure 代码呢？当然是也可以抛出异常。想一想 `div` 与 `head` 函数，它们的类型分别是 `(Integral a) => a -> a -> a` 与 `[a] -> a` 。`Maybe` 或 `Either` 并没有在它们的返回类型中。

```
ghci> 4 `div` 0
*** Exception: divide by zero
ghci> head []
*** Exception: Prelude.head: empty list
```

Pure 代码可以抛出异常，但是它们只能在我们代码的 I/O 部分（也就是 `main` 里的 `do` 代码块）中被捕获。这是因为 pure 代码中你不知道什么东西会在什么时候被计算。因为 lazy 特性的原因，程序没有一个特定的执行顺序，但是 I/O 代码却有。

I/O Exception 是我们在 `main` 中与外界沟通失败而抛出的异常。下面是一个接受命令行参数，打开所指定的文件名，并计算有多少行的代码：

```
import System.Environment
import System.IO

main = do
  (fileName : _) <- getArgs
  contents <- readFile fileName
  let l = show . length . lines
  putStrLn $ "This file has " ++ contents ++ " lines!"
```

当读取一个不存在的文件时：

```
% runhaskell linecount.hs xx.txt
linecount.hs: xx.txt: openFile: does not exist (No such file or directory)
```

改造一下，使用 `System.Directory` 中的 `doesFileExist` 函数来确保文件是否存在：

```
import System.Directory
import System.Environment
import System.IO

main = do
  (fileName : _) <- getArgs
  fileExists <- doesFileExist fileName
  if fileExists
    then do
      contents <- readFile fileName
      let l = show . length . lines
      putStrLn $ "The file has " ++ l contents ++ " lines!"
    else do
      putStrLn "The file doesn't exists!"
```

代码中的 `fileExists <- doesFileExist fileName` 是因为 `doesFileExist` 的类型是 `doesFileExist :: FilePath -> IO Bool` ，意味着返回一个 I/O action 其内容为布尔值，因此不能直接用于 *if* 表达式中。

`System.IO.Error` 模块中的 `catch` 函数可以用于处理异常。其类型是 `catch :: IO a -> (IOError -> IO a) -> IO a`，接受两个参数：第一个参数是一个 I/O action，例如尝试打开文件的 I/O action；第二个参数是一个句柄，如果 I/O action 抛出了 I/O 异常，那么该异常会被传递至该句柄，然后再决定如何处理。最终的返回值也是一个 I/O action，也就是说整个过程中要么如预期那样完成第一个参数的 I/O action，要么就是句柄处理后的结果。

句柄接受一个 `IOError` 类型的值，它代表的是一个 I/O 异常已经发生了，它也带有一些异常信息。至于该类型在语言中如何被实现则是要看编译器。这就意味着我们没法用模式匹配的方式来查看 `IOError` ，类似于不能用模式匹配来查看 `IO something` 的内容那样。但是我们能用一些子句来查看它们。

以下是一个使用了 `catch` 的程序：

```
import Control.Exception
import System.Environment

main = toTry `catch` handler

toTry :: IO ()
toTry = do
  (fileName : _) <- getArgs
  contents <- readFile fileName
  let l = show . length . lines
  putStrLn $ "The file has " ++ l contents ++ " lines!"

handler :: IOError -> IO ()
handler e = putStrLn "Whoops, had some trouble!"
```

与原文不同的是 `catch` 位于 `Control.Exception` 模块中。测试

```
% runhaskell linecount2.hs xx.txt
Whoops, had some trouble!
% runhaskell linecount2.hs todo.txt
The file has 3 lines!
```

下面代码细分了异常的处理：

```
import Control.Exception
import System.Environment
import System.IO.Error (isDoesNotExistError)

main = toTry `catch` handler

toTry :: IO ()
toTry = do
  (fileName : _) <- getArgs
  contents <- readFile fileName
  let l = show . length . lines
  putStrLn $ "The file has " ++ l contents ++ " lines!"
```

```
handler :: IOError -> IO ()
handler e
  | isDoesNotExistError e = putStrLn "The file doesn't exist!"
  | otherwise = ioError e
```

这里使用了 `System.IO.Error` 模块中的 `isDoesNotExistError` 以及 `ioError` 。前者是一个运行在 `ioError` 之上的子句，其类型为 `isDoesNotExistError :: IOError -> Bool` 。这里使用了守护，但实际上也可以用 *if else*。如果异常不是文件不存在而导致的，那么就用 `ioError` 将异常重新抛出。

以下是一些常用的子句：

1. `isAlreadyExistsError`
2. `isDoesNotExistError`
3. `isFullError`
4. `isEOFError`
5. `isIllegalOperation`
6. `isPermissionError`
7. `isUserError`

最后一个 `isUserError` 时在使用 `userError` 函数时的返回，意为我们的代码中抛出的异常。例如 `ioError $ userError "remote computer unplugged!"` ，尽管用 `Either` 或 `Maybe` 来表示可能的错误会比自己抛出异常更好。

所以可以编写这样的一个句柄：

```
handler :: IOError -> IO ()
handler e
  | isDoesNotExistError e = putStrLn "The file doesn't exist!"
  | isFullError e = freeSomeSpace
  | isIllegalOperation e = notifyCops
  | otherwise = ioError e
```

其中 `freeSomeSpace` 与 `notifyCops` 是用户自定义的 I/O action。如果异常不匹配，记得要将异常重新抛出，不然程序还是会崩溃。

`System.IO.Error` 还提供了一些能够询问异常性质的函数，比如说是哪些句柄导致的错误，或者是哪些文件名造成的错误。这些函数都是 `ioe` 开头，详见 此处. 这里使用 `ioeGetFileName` 函数可以知道文件路径，其类型为 `ioeGetFileName :: IOError -> Maybe FilePath` 。修改一下程序：

```haskell
import Control.Exception
import System.Environment
import System.IO.Error

main = toTry `catch` handler

toTry :: IO ()
toTry = do
  (fileName : _) <- getArgs
  contents <- readFile fileName
  let l = show . length . lines
  putStrLn $ "The file has " ++ l contents ++ " lines!"

handler :: IOError -> IO ()
handler e
  | isDoesNotExistError e =
      case ioeGetFileName e of
        Just path -> putStrLn $ "Whoops! File does not exist at: " ++ path
        Nothing -> putStrLn "Whoops! File does not exist at unknown location!"
  | otherwise = ioError e
```

如果不想只用一个 `catch` 来捕获 I/O 部分中的所有异常，那么可以在特定的地方用 `catch` 来进行捕获，或者也可以使用不同的句柄：

```haskell
main = do toTry `catch` handler1
  thenTryThis `catch` handler2
  launchRockets
```

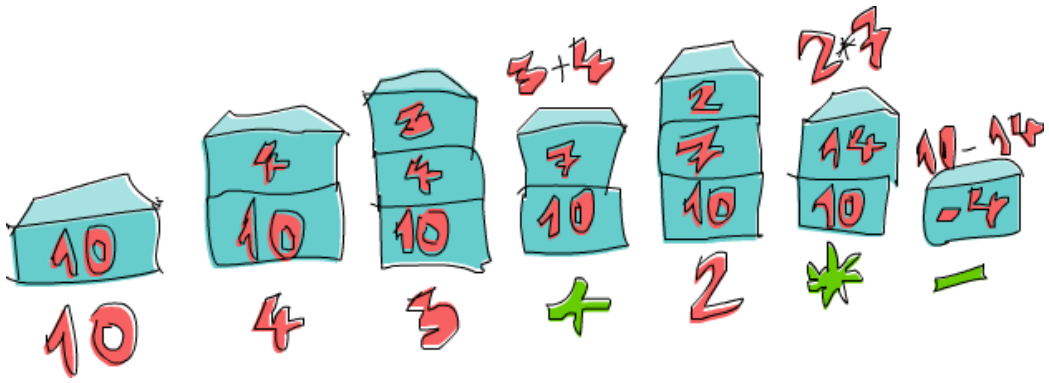
# 10 Functionally Solving Problems

本章我们将解决几个有趣的问题，同时尝试用函数式的方式来解决这些问题。

## 运算逆波兰表示法

学校里学习的数学表达式都是中置（infix）表示法，比如 $10-(4+3)*2$，其中 $+$，$*$，$-$ 就是中置算子。在 Haskell 中就像是 `+` 或者 `elem` 那样。这种写法对于人类来说便于阅读和理解，但是缺点就是必须用括号来描述运算的优先级。

逆波兰法（Reverse Polish notation form）是另一种数学的表示法，上述表达式用 RPN 表达就是 10 4 3 + 2 * −。可以想象成堆叠，从左往右阅读算式，每当碰到一个数值就把它堆上堆叠。当碰到一个算子，就把两个数值从堆叠上拿下来，运算完成后再将结果堆上堆叠。



**Note**

在实现一个函数之前，更重要的是思考其类型声明。在 Haskell 中，一个函数的类型声明能告诉我们很多函数的信息，感谢强壮的类型系统。

现在来写一个 Haskell 函数用于处理 RPN 运算式：

```haskell
import Data.List

solveRPN :: (Num a) => String -> a
solveRPN expression = head (foldl foldingFunction [] (words expression))
    where   foldingFunction stack item = ...
```

首先是接受一个表达式，通过 `words` 将其转换为各个列表中的项。再用 `foldl` 函数接受的 `foldingFunction` 处理堆叠，且 `[]` 作为累加器，初始状态即为空的堆叠，最后用 `head` 获取最终堆叠的单个值。通过 point-free 风格取出括号，再用 `where` 声明函数 `foldingFunction` 。

```haskell
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
```

`foldingFunction` 展开成四个模式匹配，从第一个开始尝试匹配，三种算符情况下取头部两个元素直接进行计算，非三种算符时则进入最后一个匹配，将输入视为可转换的数值字符串（如果 `read` 失败则抛出异常）进行堆叠。测试：

```
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
```

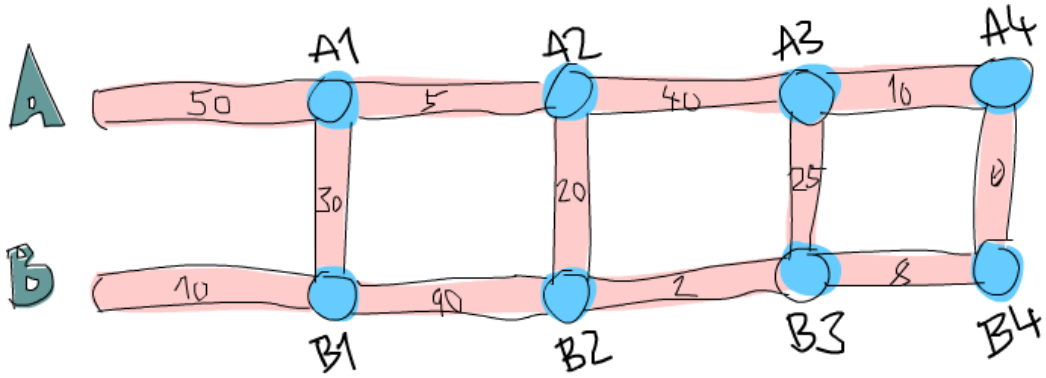
现在修改一下函数使其可以接受更多的算符，为了简化使其返回类型为 `Float` ：

```haskell
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
```

```
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
ghci> solveRPN "43.2425 0.5 ^"
6.575903
```

最后就是注意该函数并没有错误容忍的能力，假设输入的表达式并不合理，那么程序便会崩溃。因此将函数类型改为 `solveRPN :: String -> Maybe Float` 则更为合理，我们会在学习 monads 的时候再来进行实现。当然我们现在也可以实现，只不过会有一大堆检查 `Nothing` 的动作（可以使用 `reads` 来看看一次 read 是否成功）。

## 路径规划




使用 Haskell 数据类型来描述问题。一种做法是起始点与交叉点作为图的节点，想象起点也有一条长度为 1 的虚拟道路连接，每个交叉点都连接对面节点，同时也连到下一个交叉点，除了最后一个节点。数据类型如下：

```
data Node = Node Road Road | EndNode Road
data Road = Road Int Node
```

一个节点要么是普通节点，其包含了去往另一个主路的信息以及去往下一个节点（相同主路）的信息；要么是一个终点，其仅包含了去往另一个主路的信息。一个路则是包含了长度信息，以及其指向的节点。

另一种做法就是用 `Maybe` 来代表往下一个节点（相同主路）的信息。每个节点都有指到另一条主路节点的路径，但只有不是终点的节点才有指向下一个节点（相同主路）的路。

```
data Node = Node Road (Maybe Road)
data Road = Road Int Node
```

实际上，每次检查只会检查三条路径的长度：道路 A 的部分，道路 B 的部分，以及它们相连的部分。当观察 A1 与 B1 的最短路径时，只需考虑第一组的三个部分，即花费 50,10 以及 30。因此道路系统可以由四组数据来表示：$50, 10, 30$，$5, 90, 20$，$40, 2, 25$ 以及 $10, 8, 0$。

让我们的数据类型越简单越好，不过这样已经是极限了：

```
data Section = Section {getA :: Int, getB :: Int, getC :: Int} deriving (Show)
type RoadSystem = [Section]
```

那么 Heathrow 去 London 的道路系统便可这样表示：

```
heathrowToLondon :: RoadSystem
heathrowToLondon = [Section 50 10 30, Section 5 90 20, Section 40 2 25, Section 10 8 0]
```

剩下需要做的就是实现方案。首先引入 `Label` 类型，作为一个枚举包含了 `A` ，`B` 或 `C` ，另外就是一个类型同义词 `Path` ：

```
data Label = A | B | C deriving (Show)
type Path = [(Label, Int)]
```

我们的函数 `optimalPath` 其类型声明就应该是 `optimalPath :: RoadSystem -> Path` 。那么在调用 `heathrowToLondon` 之后，其应该返回如下路径:

```
[(B,10),(C,30),(A,5),(C,20),(B,2),(B,8)]
```

在手动解答时，有一个步骤我们在不断的重复，那就是检查 A 与 B 的最佳路径以及当前的 section，产生新的 A 与 B 的最佳路径。例如，最开始的最佳路径是 `[]` 与 `[]` ，看到 `Section 50 10 30` 后得到新的到 A1 最佳路径为 `[(B,10),(C,30)]` ，而到 B1 的最佳路径是 `[(B,10)]` 。如果吧这个步骤看做一个函数，那么便是接受一对路径以及一个 Section，并产生出新的一对路径。因此函数类型为 `(Path, Path) -> Section -> (Path, Path)` 。

**Note**

类型为 `(Path, Path) -> Section -> (Path, Path)` 非常的有用，因为可以被二元函数 left fold 所使用，其需要的类型就是 `a -> b -> a` 。

```
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)
```

这里发生了什么?首先是基于先前 A 的最佳解计算道路 A 的最佳解,B 同理。使用 `sum $ map snd pathA` ，如果 `pathA` 是 `[(A,100),(C,20)]` ，那么 `priceA` 变为 `120` 。如果直接从之前的 A 去

往下一个 A 节点，那么 `forwardPriceToA` 则是要付的成本；`crossPriceToA` 则是先前在 B 上前往 A 所付出的成本。

记住，路径是相反的，因此要从右向左读。

**Note**

**优化小技巧**：当 `priceA = sum $ map snd pathA` 时，我们在计算每个步骤的成本。如果我们实现的 `roadStep` 类型是 `(Path, Path, Int, Int) -> Section -> (Path, Path, Int, Int)`，那么就不必这么做。这里的整数类型代表 A 与 B 上的最小成本。

现在有了接受一对路径以及一个 section 并返回最佳路径的函数，我们可以用 left fold 来使用它。将 `([],[])` 以及第一个 section 代入至 `roadStep` 中就可以得到一对最佳路径，然后再将这个最佳路径以及新的 section 代入至 `roadStep` 就可以得到下一个最佳路径。以此类推，我们可以用以实现 `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
```

测试：

```
ghci> optimalPath heathrowToLondon
[(B,10),(C,30),(A,5),(C,20),(B,2),(B,8),(C,0)]
```

符合预期！现在要做的是从标准输入读取文本形式的道路系统，并将其转换成 `RoadSystem`，然后再用 `optimalPath` 来运行一遍即可。

首先是一个接受列表并将其切成同样大小的 group 的函数，例如当参数是 `[1..10]` 时，`groupsOf 3` 则应该返回 `[[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)
```

接下来是 main 函数：

```
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 = concatMap (show . fst) path
    pathPrice = sum $ map snd path
putStrLn $ "The best path to take is: " ++ pathString
putStrLn $ "The price is: " ++ show pathPrice
```

创建一个 `paths.txt` 文件：

```
50
10
30
5
90
20
40
2
25
10
8
0
```

执行：

```
$ cat paths.txt | runhaskell heathrow.hs
The best path to take is: BCACBBC
The price is: 75
```

执行成功！我们还可以用 `Data.Random` 来产生一个比较大的路径配置，然后把生产的随机数据输入至程序。如果碰到堆栈溢出，试试使用 `foldl'` ，即 strict 模式。

# 11 Functors, Applicative Functors and Monoids

由纯粹性，高阶函数，参数化代数数据类型，以及 typeclasses 所构成的 Haskell，相比于其他语言，允许我们实现更高等级的多态。我们无需考虑类型是如何属于一个类型阶级系统的，而是由合理的 typeclasses 如何连接它们的。

Typesclasses 是开放的，这就意味着我们可以定义自己的数据类型。

## 函子 Functors 复习

回忆：Functors 可以用于映射至比如列表，`Maybe`，树，等等。在 Haskell 中它们被 `Functor` typeclass 所描述，其仅有一个名为 `fmap` 的 typeclass 方法，该方法的类型是 `fmap :: (a -> b) -> f a -> f b`。它表示：给我一个接受一个 `a` 并返回一个 `b` 的函数，以及给我一个（或多个）`a` 的容器，那么我将给你一个（或多个）`b` 的容器。有点像是将函数应用至容器中的元素。

如果一个类型构造函数是 `Functor` 的实例，那么它的 kind 就必须是 `* -> *`，这代表它必须刚好接受一个类型作为类型参数。像是 `Maybe` 是 Functor 的实例，它接受一个类型参数，如 `Maybe Int` 或 `Maybe String`。如果一个类型构造函数接受两个参数，比如 `Either`，我们不能这样 `instance Functor Either where`，不过可以这样 `instance Functor (Either a) where`，那么 `fmap` 作为 `Either a` 的类型前面就可以这样 `fmap :: (b -> c) -> Either a b -> Either a c`。

现在看看 `IO` 是怎么样的一个 `Functor` 实例。当我们 `fmap` 一个函数在一个 I/O action 上，我们希望拿回的 I/O action 是已经被映射过的：

```
instance Functor IO where
  fmap f action = do
      result <- action
      return (f result)
```

对一个 I/O action 映射的结果仍然是一个 I/O action，因此需要用 *do* 语法来把两个 I/O action 粘合成一个。

```
main = do
  line <- getLine
  let line' = reverse line
  putStrLn $ "You said " ++ line' ++ " backwards!"
  putStrLn $ "Yes, you really said " ++ line' ++ " backwards!"
```

用 `fmap` 改写：

```
main = do
  line <- fmap reverse getLine
  putStrLn $ "You said " ++ line ++ " backwards!"
  putStrLn $ "Yes, you really said " ++ line ++ " backwards!"
```

如果将 `fmap` 应用至 `IO` ，其类型便为 `fmap :: (a -> b) -> IO a -> IO b` 。

如果将一个 I/O action 绑定到一个名称上时，仅仅想要应用一个函数，那么可以考虑使用 `fmap` ，因为这看起来更简洁。如果想要应用多个函数，那么可以声明自己的函数，将其变为 lambda 或者使用函数组合：

```
import Data.Char
import Data.List

main = do
  line <- fmap (intersperse '-' . reverse . map toUpper) getLine
  putStrLn line
```

```
$ runhaskell fmappingIO.hs
hello there
E-R-E-H-T- -O-L-L-E-H
```

`intersperse '-' . reverse . map toUpper` 这个函数接受一个字符串，映射 `toUpper` 后的返回值再应用 `reverse` ，最后应用 `intersperes '-'` 。

另一个我们已经接触过的 `Functor` 实例就是 `(->) r` 。函数类型 `r -> a` 可以被重写为 `(->) r a` ，就像是 `2 + 3` 可以表达为 `(+) 2 3` 。从另一个角度来看 `(->) r a` ，它其实就是一个接受两个类型参数的构造函数，例如 `Either` 。但是记住 `Functor` 只能接受一个参数类型，这也是为什么 `(->)` 不是 `Functor` 的一个实例，但是 `(->) r` 则是。在 `Control.Monad.Instances` 有更多的细节：

```
instance Functor ((->) r) where
  fmap f g = (\x -> f (g x))
```

首先来看看 `fmap` 的类型 `fmap :: (a -> b) -> f a -> f b` 。我们把所有的 `f` 替换成 `(->) r` ，那么就是 `fmap :: (a -> b) -> ((->) r a) -> ((->) r b)` ，然后将 `(->) r a` 与 `(->) r b` 换成 `r -> a` 与 `r -> b` ，则得到 `fmap :: (a -> b) -> (r -> a) -> (r -> b)` 。

将一个函数映射至另一个函数可以生产出一个函数，就像是映射一个函数在 `Maybe` 上生产出一个 `Maybe` ，映射一个函数在一个列表上能生成出一个列表。那么上述的类型实例说的是什么呢？它接受一个 `a` 至 `b` 的函数，以及一个 `r` 至 `a` 的函数，返回一个 `r` 至 `b` 的函数。这就是一个函数组合！另一种实例的写法：

```
instance Functor ((->) r) where
  fmap = (.)
```

很明显就是将 `fmap` 当函数组合在用。在 GHCI 中使用 `:m + Control.Monad.Instances` 加载模块，测试：

```
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"
```

`fmap` 等同于函数组合这件事儿对我们而言并不是很实用，但是这至少是一个有趣的观点。Functors 其实比较像是 computation，函数被映射到另一个计算会经由那个函数映射过后的计算。

在看一下 `fmap` 的类型 `fmap :: (a -> b) -> f a -> f b` ,那么为了简洁不写 `(Functor f) =>` 部分。在学柯里化的时候提到过 Haskell 的函数实际上只接受一个参数，即 `a -> b -> c` ，如果调用时少写一个参数，那么返回的函数就需要接受剩下的参数。所以 `a -> b -> c` 可以写成 `a -> (b -> c)` ，这样柯里化更加明显。

同样的如果写成 `fmap :: (a -> b) -> (f a -> f b)` ，我们可以认为 `fmap` 不是一个接受函数与函子并返回一个函子的函数，而是一个接受函数并返回一个新函数的函数，而这个返回的新函数接受一个函子作为参数，并返回一个函子。接受一个 `a -> b` 函数并返回一个 `f a -> f b` 函数，这被称为*提升 lifting* 一个函数。我们用 GHCI 的 `:t` 命令检查：

```
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]
```

表达式 `fmap (*2)` 是一个接受基于数值的函子 `f` 并返回一个数值函子的函数。函子可以是一个列表一个 `Maybe` ，一个 `Either String` ，等等。表达式 `fmap (replicate 3)` 则接受任意类型的函子并返回一个该类型列表函子。

**Note**

当我们说*一个基于数值的函子 a functor over numbers* 时，你可以认为是*一个拥有数值的函子 a functor that has numbers in it*。前者的描述更加精准，后者则更容易理解。

当使用部分应用绑定至一个名称时，比如说 `fmap (++"!")` ，在 GHCI 更加的显而易见。

```
ghci> :t fmap (++"!")
fmap (++"!") :: Functor f => f [Char] -> f [Char]
```

可以把 `fmap` 想像成一个函数，接受另一个函数与一个函子，然后将接受的函数对函子中每个元素做映射；也可以想象成是一个函数，接受一个函数并将其提升 lift 到可以在函子上操作。这两种想法都是正确的，且在 Haskell 中是等价的。

接下来我们开始学习**函子定律 functor laws**。为了让某物成为一个函子，它必须满足某些条件。所有的函子都要求具有某些性质，它们必须是能被映射的，对它们调用 `fmap` 应该要用一个函数映射至每一个元素，且没有额外的操作。这些行为都被函子定律所描述。对于 `Functor` 的实例而言，总共有两条定律该被遵守。不过它们不会在 Haskell 中自动被检查，需要用户自己确认这些条件。

**函子第一定律是如果将 `id` 函数映射至该函子，得到的函子必须是原来的函子**。用代码表示即 `fmap id = id` ，换言之如果 `fmap id` 至一个函子，返回的结果跟直接 `id` 至该函子是一样的。`id` 是一个 identity 函数，可以写作是 `\x -> x` 。

```
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
```

看一下 `Maybe` 的 `fmap` 的实现，可以知道第一定律是如何遵守的：

```
instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap f Nothing = Nothing
```

**函子第二定律是将两个函数合成并将结果映射至一个函子的结果，与现将第一个函数映射函子再将第二个函子映射至该函子的结果是一样的**。即 `fmap (f . g) = fmap f . fmap g` ，或者另一种写法，对于任何一个函子 `F` ，满足 `fmap (f . g) F = fmap f (fmap g F)` 。

## 高级函子 Applicative functors

本节开始学习高级函子，在 Haskell 中由 `Applicative` typeclass 描述，可在 `Control.Applicative` 模块中找到。

函数在 Haskell 中默认是柯里化的，这就意味着一个函数看起来接受若干参数实际上是接受一个参数并返回一个函数再接受一个参数，以此类推。如果一个函数的类型是 `a -> b -> c` ，那么通常会说它接受两个参数并返回一个 `c` 。这就是为什么可以将 `f x y` 表达为 `(f x) y` 。这个机制允许我们部分应用函数，仅需调用少一些的参数，返回的函数可以被传递至另一个函数。

迄今为止我们映射函数至函子，所映射的函数都是仅一个参数的。但是当映射一个接受两个参数的函数像是 `*` 至一个函子，该怎么办呢？首先让我们来观测几个实际的例子。如果有

`Just 3`，我们做 `fmap (*) (Just 3)`，会得到什么？从 `Maybe` 实现 `Functor` 的实例中我们知道如果是一个 `Just something` 值，会将函数应用至 `Just` 中的 `something`。因此 `fmap (*) (Just 3)` 的结果是 `Just ((*) 3)`，即 `Just (* 3)`。有意思！我们得到了一个包裹在 `Just` 中的函数！

```
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]
```

如果映射 `compare`，其类型是 `(Ord a) => a -> a -> Ordering` 至一个字符列表，得到 `Char -> Ordering` 的列表，因为函数 `compare` 是部分应用在列表中的每个字符上的。不是 `(Ord a) => a -> Ordering` 函数的列表，因为第一个 `a` 被应用的是一个 `Char`，因此第二个 `a` 就必须是 `Char` 类型了。

我们见识到了如何映射“多参数”函数至函子，得到的是一个包含了该函数的函子。那现在我们能对这个包含了函数的函子做什么呢？我们能用一个消费这些函数的函数来映射至这个函子，这些函子中的函数都会被当做参数传给消费函数。

```
ghci> let a = fmap (*) [1,2,3,4]
ghci> :t a
a :: [Integer -> Integer]
ghci> fmap (\f -> f 9) a
[9,18,27,36]
```

但如果一个函子值是 `Just (3 *)`，一个函子值是 `Just 5`，我们希望从 `Just (3 *)` 取出函数并映射至 `Just 5` 呢？普通的函子是没法做的，因为他们仅支持映射普通函数至已存在的函子。即使当我们映射 `\f -> f 9` 至一个包含了函数的函子上，我们也只是映射了一个普通函数。也就是说我们没法用 `fmap` 将一个包在函子里的函数映射至一个函子。我们可以用模式匹配将 `Just` 中的函数抽出来再映射至 `Just 5`，不过我们希望有一个通用方法，对于任何函子都有效。

现在来看看 `Applicative` 这个 typeclass，可以在 `Control.Applicative` 中找到它，其定义了两个函数 `pure` 以及 `<*>`。没有任何的默认实现，因此如果希望某物成为高级函子，那么我们需要定义这两个函数。其定义如下：

```
class (Functor f) => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

这简单的三行告诉了我们很多东西！首先是第一行，定义 `Applicative` 类，并引入了类约束，即 `Applicative` typeclass 的构造函数中的类型首先必须的是一个 `Functor`，因为这

样才能使用 `fmap` 。

第一个定义的方法是 `pure` 。其类型声明是 `pure :: a -> f a` 。`f` 在这里就是高级函子的实例。其接受一个任意类型，返回一个包含了该任意类型的高级函子。

一个更好理解 `pure` 的方式就是它接受一个值，将该值放入某些默认（或纯）context 中 – 这个最小的 context 仍然包含了这个值。

`<*>` 函数非常有趣，其类型声明是 `f (a -> b) -> f a -> f b` ，这跟 `fmap :: (a -> b) -> f a -> f b` 很像。也就是一个增强版的 `fmap` 。`fmap` 接受一个函数和函子，并将函数应用至函子内的值，而 `<*>` 接受一个包含了函数的函子，以及另一个函子，并类似于提取第一个函子中的函数应用至第二个函子中。

下面是 `Maybe` 实现 `Applicative` 的实例：

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  (Just f) <*> something = fmap f something
```

首先是 `pure` ，这里的 `pure = Just` 是因为 `Just` 的值构造函数是普通函数，等同于 `pure x = Just x` 。

其次是 `<*>` ，我们不能从 `Nothing` 中提取函数，因此其返回值就是 `Nothing` ；如果是一个包含函数的 `Just` ，那么将该函数映射至第二个参数，这里第二个参数也有可能是 `Nothing` ，不过 `fmap` 任何函数至 `Nothing` 也都会返回 `Nothin` ，因此无需多虑。

测试一下：

```
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
```

普通的函子能将函数映射至函子本身，但是可能没法拿到结果；而高级函子则可以让你用单一一个函数操作好几个函子。比如：

```
ghci> pure (+) <*> Just 3 <*> Just 5
Just 8
ghci> pure (+) <*> Just 3 <*> Nothing
Nothing
ghci> pure (+) <*> Nothing <*> Just 5
Nothing
```

这里发生了什么?让我们一步一步来看。`<*>` 是左关联的,意味着 `pure (+) <*> Just 3 <*> Just 5` 等同于 `(pure (+) <*> Just 3) <*> Just 5` 。首先 `+` 函数被放进了一个函子中,这个例子中就是 `Maybe` 包含了该函数,因此 `pure (+)` 即是 `Just (+)` 。其次 `Just (+) <*> Just 3` 发生了,其结果为 `Just (3+)` ,这是因为部分应用,仅将 `3` 应用至 `+` 函数返回的是一个接受一个参数的函数。最后 `Just (3+) <*> Just 5` 被运算,结果就是 `Just 8` 。

这很棒吧!用 applicative 风格来使用高级函子,如 `pure f <*> x <*> y <*> ...` 就让我们可以哪一个接受多个参数的函数,而且这些参数不一定是包在函子内的。这样套用多个在函子 context 的值,该函数可以消费任意多的参数,毕竟 `<*>` 只是在做部分应用而已。

如果考虑到 `pure f <*> x` 等同于 `fmap f x` 的话,这样的用法就更方便了。这是一条 applicative 定律,之后会进行详细讲解。如果我们将一个函数置入默认 context 中,接着提取并应用至另一个一个高级函子中的值,那么这就跟映射一个函数至高级函子一样。与其 `pure f <*> x <*> y <*> ...` ,可以写作 `fmap f x <*> y <*> ...` 。这就是为什么 `Control.Applicative` 导出了一个名为 `<$>` 的函数,它实际上就是一个中置版本的 `fmap` ,其定义如下:

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

**Note**

快速提示:类型变量跟参数的名字还有值绑定的名称不冲突。`f` 在函数的类型声明中是类型变量,说明 `f` 应该要满足 `Functor` typeclass 的条件。而在函数本体中的 `f` 则表示一个函数,我们将它映射至 `x` 。我们同样用 `f` 来表示它们并代表它们是相同的东西。

通过使用 `<$>` ,applicative 风格的好处就很显著了。如果将 `f` 应用至三个高级函子,就可以这样 `f <$> x <*> y <*> z` 。如果参数不是高级函子而是普通值,则是 `f x y z` 。

让我们再看看它是如何工作的。我们有一个值 `Just "johntra"` 以及一个值 `Just "volta"` ,我们想要合并他们成为一个在 `Maybe` 函子内部的 `String` 。我们可以这么做:

```
ghci> (++) <$> Just "johntra" <*> Just "volta"
Just "johntravolta"
```

在正式讲解发生了什么之前,先比较一下:

```
ghci> (++) "johntra" "volta"
"johntravolta"
```

可以将一个普通的函数应用在高级函子上真棒。只用写一些 `<$>` 以及 `<*>` 就可以把函数变为 applicative 风格,操作这些 applicatives 并返回一个 applicative。太酷了!

无论如何,当运算 `(++) <$> Just "johntra" <*> Just "volta"` 时,首先 `(++)` 其类型是 `(++) :: [a] -> [a] -> [a]` ,映射至 `Just "johntra"` ,返回值为 `Just ("johntra"++)`

，类型为 `Maybe ([Char] -> [Char])` 。注意 `(++)` 的第一个参数被吃掉了，即 `a` 转换为 `Char` 。接着是 `Just ("johntra"++) <*> Just "volta"` ，从 `Just` 提取出的函数并映射至 `Just "volta"` ，得到 `Just "johntravolta"` 。这里只要任意一个值是 `Nothing` ，那么得到的结果也是 `Nothing` 。

列表（实际上说的是列表构造函数， `[]` ）也是高级函子。惊喜吧！下面是 `[]` 如何作为 `Applicative` 实例的：

```
instance Applicative [] where
  pure x = [x]
  fs <*> xs = [f x | f <- fs, x <- xs]
```

之前我们说 `pure` 接受一个值并将其放入一个默认 context 中，或者说，一个最小 context 仍然包含该值。这里的最小 context 就是空列表， `[]` ，不过空列表并不包含一个值，所以我们没法将其当做 `pure` 。这也是为什么 `pure` 实际上是接受一个值，然后返回一个带有单元素的列表。同样的， `Maybe` 的最小 context 是 `Nothing` ，但它其实表示的是没有值，所以 `pure` 其实是被实现成了 `Just` 。

```
ghci> pure "Hey" :: [String]
["Hey"]
ghci> pure "Hey" :: Maybe String
Just "Hey"
```

那么 `<*>` 呢?如果我们假定 `<*>` 的类型是限制在列表上的话,我们会得到 `(<*>) :: [a -> b]-> [a] ->` 。这是用列表表达式来实现的。 `<*>` 必须以某种方式从左侧参数中提取函数并映射至右侧参数上。不过这里左侧列表可以是没有函数，一个函数，或者多个函数；右侧列表也可以存储若干值。这就是为什么我们使用列表表达式从两方列表中取值。我们要将左侧列表中所有可能的函数映射至右侧列表中所有可能的值。

```
ghci> [(*0),(+100),(^2)] <*> [1,2,3]
[0,0,0,101,102,103,1,4,9]
```

左侧列表包含三个函数，而右侧列表有三个值，那么结果就是有九个元素的列表。左边列表的每个函数都应用在右侧列表的每个值。如果列表中的函数接受两个参数，那么也可以应用到两个列表上：

```
ghci> [(+),(*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]
```

这是因为 `<*>` 是左关联的,即 `[(+),(*)] <*> [1,2]` 会先运行,得到 `[(1+),(2+),(1*),(2*)]` ,又因为左侧列表中的每个函数都会应用至右边列表中的每个值,即 `[(1+),(2+),(1*),(2*)] <*> [3,4]` ，最终结果如上所述。

列表的 applicative 风格是非常有趣的：

```
ghci> (++) <$> ["ha","heh","hmm"] <*> ["?","!","."]
["ha?","ha!","ha.","heh?","heh!","heh.","hmm?","hmm!","hmm."]
```

你可以认为列表是一个非确定性 non-deterministic 的计算。对于像 `100` 或 `"what"` 则是确定性 deterministic 计算，即只有一个结果；而 `[1,2,3]` 则可以看作是没有确定究竟是哪一种结果，也就是说它代表的是所有可能的结果。在运行 `(+) <$> [1,2,3] <*> [4,5,6]` 时，可以视为两个非确定性的计算通过 `+` 进行运算，只不过会产生另一个非确定性的计算，而结果更加不确定。

Applicative 风格对于列表而言是一个取代列表表达式的好方法。第二章里计算 `[2,5,10]` 与 `[8,10,11]` 相乘的结果，需要：

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

通过 applicative 风格改写：

```
ghci> (*) <$> [2,5,10] <*> [8,10,11]
[16,20,22,40,50,55,80,100,110]
```

如果我们希望将两个列表中的所有乘积大于 50 的数找出来，仅需：

```
ghci> filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]
[55,80,100,110]
```

可以看到列表的运算中，`pure f <*> xs` 等同于 `fmap f xs` 。`pure f` 就是 `[f]` ，以及 `[f] <*> xs` 将应用左侧列表中的每个函数至右侧列表中的每个值，但左边其实只有一个函数，所以看上去就像是 mapping。

另一个我们已经遇到的 `Applicative` 实例就是 `IO` ，其实现：

```
instance Applicative IO where
  pure = return
  a <*> b = do
      f <- a
      x <- b
      return (f x)
```

`pure` 将值放入最小 context 内，这里即 `return` ，意味着一个 I/O action 不做任何事情，返回其包含的值，但实际上并没有做任何 I/O 操作比如打印至终端或者文件读取。

而 `<*>` 被限制在 `IO` 上的话，它的类型就是 `<*> :: IO (a -> b) -> IO a -> IO b` ，即接受一个生产函数的 I/O action，以及另一个 I/O action，由它们创建一个新的 I/O action，也就是把第二个参数传给第一个参数，得到的返回值放进新的 I/O action 中。注意 *do* 语法就是将若干 I/O action 粘合成一个，这正是我们现在所做的。

通过 `Maybe` 已经 `[]` ，我们可以认为 `<*>` 就是从其左参数中提取一个函数，再将该函数应用至右参数上。而对于 `IO` ，仍然是提取，不过带上了*序列 sequencing* 这个概念，因为接受两个 I/O actions 顺序执行，或者说是粘合成为一个 I/O action。从第一个 I/O action 中取值，但要取出 I/O action 的结果，因此它必须先被执行。

例如：

```
myAction :: IO String
myAction = do
    a <- getLine
    b <- getLine
    return $ a ++ b
```

这个 I/O action 提示用户输入两行在返回合并的两行。达到这个目的就是将两个 `getLine` I/O actions 粘合在一起后使用 `return` 。而使用 applicative 可以这样写：

```
myAction :: IO String
myAction = (++) <$> getLine <*> getLine
```

表达式 `(++) <$> getLine <*> getLine` 的类型是 `IO String` ，意味着该表达式跟别的 I/O action 完全一样，因此可以这么做：

```
main = do
  a <- (++) <$> getLine <*> getLine
  putStrLn $ "The two lines concatenated turn out to be: " ++ a
```

另一个 `Applicative` 实例就是 `(->) r` ：

```
instance Applicative ((->) r) where
  pure x = (\_ -> x)
  f <*> g = \x -> f x (g x)
```

`pure` 接受值，创建一个忽视入参且将该值作为返回的函数，即 `pure :: a -> (r -> a)` 。

```
ghci> (pure 3) "blah"
3
```

因为有柯里化，函数应用时左关联的，所以可以省略括号：

```
ghci> pure 3 "blah"
3
```

而 `<*>` 的实现则有点神秘，让我们看看作为高级函子如何使用该函数：

```
ghci> :t (+) <$> (+3) <*> (*100)
(+) <$> (+3) <*> (*100) :: (Num a) => a -> a
ghci> (+) <$> (+3) <*> (*100) $ 5
508
```

两个高级函子输入 `<*>` 返回一个高级函子，所以如果输入两个函数，则得到一个新的函数。这里发生了什么？当使用 `(+) <$> (+3) <*> (*100)` 时，我们在创造一个函数，该函数会将 `(+3)` 以及 `(*100)` 的结果，应用 `+` ，再进行返回，即 `5` 给到 `(+3)` 以及 `(*100)` ，得到 `8` 与 `500` 后，将 `+` 应用至 `8` 与 `500` ，得 `508` 。再比如：

```
ghci> (\x y z -> [x,y,z]) <$> (+3) <*> (*2) <*> (/2) $ 5
[8.0,10.0,2.5]
```

这里也一样。创建一个将会调用 `\x y z -> [x,y,z]` 函数的函数，输入的参数是 `(+3)` ，`(*2)` ，以及 `(/2)` 。`5` 则被分别输入进这三个函数中，接着调用 `\x y z -> [x,y,z]` 得到结果。

我们可以认为函数作为盒子包含着它们最终的结果，因此 `k <$> f <*> g` 创建了一个将最终会对 `f` 与 `g` 的结果而调用 `k` 的函数。

另一个我们还没遇到过的 `Applicative` 实例就是 `ZipList` ，它位于 `Control.Applicative` 模块中。

实际上列表要作为一个高级函子可以有很多种方式。已经介绍过的一种就是应用 `<*>` ，左参是若干函数，右参则是若干值，结果这是函数应用于所有值后的所有组合。例如 `[(+3),(*2)] <*> [1,2]` ，`(+3)` 会先应用至 `1` 与 `2` ，接着再是 `(*2)` 应用至 `1` 与 `2` ，最终得 `[4,5,2,4]` 。

而 `[(+3),(*2)] <*> [1,2]` 理论上也可以将左参第一个函数应用至右参第一个值，再将左参第二个函数应用至右参的第二个值，以此类推。

由于一个类型不能对同个 typeclass 定义两个实例，因此才有 `ZipList a` ，它只有一个构造函数 `ZipList` ，一个字段，它的类型是列表：

```
instance Applicative ZipList where
  pure x = ZipList (repeat x)
  ZipList fs <*> ZipList xs = ZipList (zipWith (\f x -> f x) fs xs)
```

`<*>` 就是上面说的顺序应用，这是通过 `zipWith (\f x -> f x) fs xs` 实现的。又因为 `zipWith` 的原理，返回的列表长度由输入的列表中最短的那个决定。

`pure` 在这里同样有趣。它接受一个值，将其置入一个无限循环该值的列表中，例如 `pure "haha"` 在这里返回 `ZipList (["haha", "haha", "haha" ...)` 。这里可能会令人迷惑，之前我们说过 `pure` 是把一个值放入最小的 context 中，而无限长的列表不可能会是一个最小的 context。然而这对 zip 列表来说很合理，因为它必须在列表的每个位置都有值。这也遵守了 `pure f <*> xs` 必须要等价于 `fmap f xs` 的特性。

那么 zip 列表是如何用 applicative 风格运作的呢？ `ZipList a` 类型并没有 `Show` 实例，因此我们需要使用 `getZipList` 函数从一个 zip 列表中提取出一个原始列表。

```
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')]
```

**Note**
`(,,)` 函数等同于 `\x y z -> (x,y,z)` 。同样的 `(,)` 函数等同于 `\x y -> (x,y)` 。

除了 `zipWith` ，标准库还提供了 `zipWith3` ， `zipWith4` ，一直到 7。
`Control.Applicative` 定义了一个名为 `liftA2` 的函数，其定义如下：

```
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b
```

没什么特别的，它就是将一个函数应用在两个高级函子上，并没有用我们才熟悉的 applicative 风格。提及它是因为为什么高级函子要比普通的函子强。如果只是普通的函子，我们只能将一个函数映射在一个函子上，而对于高级函子而言，可以将一个函数映射在若干高级函子上。另一件有趣的事情就是 `liftA2` 的类型，其接受一个普通的二元函数，将其提升为可运作在两个函子上的函数。

这里有个有趣的概念：我们可以将两个高级函子合成为一个高级函子，而新的这个高级函子则将两个函子装在列表中。例如我们有 `Just 3` 与 `Just 4` ，假设后者是一个单例列表：

```
ghci> fmap (\x -> [x]) (Just 4)
Just [4]
```

那么我们有 `Just 3` 与 `Just [4]` ，我们如何得到 `Just [3, 4]` 呢？很简单：

```
ghci> liftA2 (:) (Just 3) (Just [4])
Just [3,4]
ghci> (:) <$> Just 3 <*> Just [4]
Just [3,4]
```

`:` 是一个函数，它接受一个元素与一个列表，返回一个新的列表，而接受的元素位于新列表的头部。现在有了 `Just [3,4]` ，那么再将它跟 `Just 2` 绑定在一起就能变为 `Just [2,3,4]` 了。我们可以将任意数量的高级函子绑在一起成为一个新的高级函子，其中包含了装有结果的列表。让我们尝试着实现一个函数，接受一个列表的高级函子，并返回一个高级函子，其中包含了结果的列表：

```
sequenceA' :: (Applicative f) => [f a] -> f [a]
sequenceA' [] = pure []
sequenceA' (x : xs) = (:) <$> x <*> sequenceA' xs
```

哈，递归！首先将一个列表的高级函子转换成一个装有列表的高级函子。以此我们可以推测出边界调节。如果将一个空的列表变成一个装有列表的高级函子，只需要将该空列表放进一个默认的 context。现在看一下如何递归。如果我们有一个有头尾的列表（这里的 `x` 是一个高级函子，而 `xs` 是一个列表的高级函子），只需再次对尾部调用 `sequenceA'` ，即得到一个装有列表的高级函子。接着将高级函子 `x` 中的值添加至装有高级函子的列表头部即可，就这么简单！

例如 `sequenceA' [Just 1, Just 2]` 则是 `(:) <$> Just 1 <*> sequenceA [Just 2]`，等同于 `(:) <$> Just 1 <*> ((:) <$> sequenceA' [])`，这里的 `sequenceA' []` 就是 `Just []`，那么表达式现在就是 `(:) <$> Just 1 <*> ((:) <$> Just 2 <*> Just [])`，即 `(:) <$> Just 1 <*> Just [2]`，得 `Just [1,2]`！

另外就是通过 fold 来实现 `sequenceA'` 了：

```
sequenceA' :: (Applicative f) => [f a] -> f [a]
sequenceA' = foldr (\x -> (<*>) ((:) <$> x)) (pure [])
```

测试：

```
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],[]]
[]
```

太酷了。当应用在 `Maybe` 上时，`sequenceA'` 创造一个新的 `Maybe`，其中包含了一个装有所有结果的列表。如果其中一个值是 `Nothing`，那整个结果就会是 `Nothing`。

当应用在函数时 `sequenceA'` 接受一堆函数的列表，并返回一个返回为列表的函数。

执行 `(+) <$> (+3) <*> (*2)` 将创建一个接受单一参数的函数，将 `(+3)` 与 `(*2)` 应用至该单一参数，最后调用 `+` 将两者相加。同理，`sequenceA' [(+3),(*2)]` 则创建一个接受单一参数的函数，将列表中的每个函数都应用至该单一参数，最后用 `:` 将它们装进一个列表中。

当我们有一个列表的函数时，想将相同的参数都输入给这些函数时，`sequenceA'` 就非常好用。例如，有一个数值，我们不知道其是否满足一系列的判断，那么就可以这样：

```
ghci> map (\f -> f 7) [(>4),(<10),odd]
[True,True,True]
ghci> and $ map (\f -> f 7) [(>4),(<10),odd]
True
```

下面代码展示了在列表表达式中使用 `sequenceA'`：

```
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]]
```

可能会有点难以理解，让我们首先来看 `sequenceA [[1,2],[3,4]]` ，根据定义 `sequenceA (x : xs) = (:)` 以及边界条件 `sequenceA [] = pure []` 来进行拆解：

1. 首先是 `sequenceA [[1,2],[3,4]]` ；
2. 计算得 `(:) <$> [1,2] <*> sequenceA [[3,4]]` ；
3. 接着计算内部的 `sequenceA` ，得 `(:) <$> [1,2] <*> ((:) <$> [3,4] <*> sequenceA [])` ；
4. 这里碰到了边界条件，即 `(:) <$> [1,2] <*> ((:) <$> [3,4] <*> [[]])`
5. 首先计算 `(:) <$> [3,4] <*> [[]]` 部分，将左列表中每个元素应用 `:` （即 `3` 与 `4` ）以及右列表的每个元素（即 `[]` ），得到 `3:[], 4:[]` ，即 `[3],[4]` 。代入上条表达式即 `(:) <$> [1,2] <*> [[3],[4]]` ；
6. 接下来同理， `:` 应用在左列表的每个元素（ `1` 与 `2` ）以及右列表的每个元素（ `[3]` 与 `[4]` ），得 `[1:[3],1:[4],2:[3],2:[4]]` ，即 `[[1,3],[1,4],[2,3],[2,4]]` 。

计算 `(+) <$> [1,2] <*> [4,5,6]` 会得到一个非确定性的结果 `x + y` ，其中 `x` 代表 `[1,2]` 中的每个值，而 `y` 代表 `[4,5,6]` 中的每个值。我们用列表表示每一种可能的情况。同样的在计算 `sequence [[1,2],[3,4],[5,6],[7,8]]` 时，它的结构会是非确定性的 `[x,y,z,w]` ，其中 `x` 代表 `[1,2]` 中的每个值，而 `y` 代表 `[3,4]` 中的每个值，以此类推。我们使用列表来代表非确定性的计算，每个元素都是一个可能的情况，这也是为什么这里使用了列表的列表。

处理 I/O actions 时， `sequenceA` 的效果与 `sequence` 一样！

```
ghci> sequenceA [getLine, getLine, getLine]
heyh
ho
woo
["heyh","ho","woo]
```

与普通函子一样，高级函子也有一些定理。最重要的定理我们已经提到了，即满足 `pure f <*> x = fmap f x` ，其他的还有：

1. `pure id <*> v = v`

2. `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
3. `pure f <*> pure x = pure (f x)`
4. `u <*> pure y = pure ($ y) <*> u`

总结一下，高级函子不仅有趣还非常的实用，它允许我们结合不同种类的计算，例如 I/O 计算，非确定性结果的计算，有可能失败的计算等等。使用 `<$>` 与 `<*>` ，可以将普通的函数来运作在任意数量的高级函子上。

### newtype 关键字

目前为止我们知道了如何使用 `data` 关键字来定义属于自己的代数数据类型，我们也学习了如何用 `type` 来定义类型同义词。本节我们将会学习如何使用 `newtype` 从一个现有的类型定义出新的类型，并说明为什么这么做。

上一节中，我们见识到了多种方式使得列表类型成为高级函子。一种方式是用 `<*>` 从左列表中获取每个函数并应用至右列表中的每个元素，返回所有可能的组合：

```
ghci> [(+1),(*100),(*5)] <*> [1,2,3]
[2,3,4,100,200,300,5,10,15]
```

另一种方式是左右列表的顺序应用，最终就像是将两个列表 zip 起来。而列表已经实现了 `Applicative` 实例，因此我们的做法是引入 `ZipList a` 类型，其包含一个值构造函数，`ZipList` ：

```
ghci> getZipList $ ZipList [(+1),(*100),(*5)] <*> ZipList [1,2,3]
[2,200,15]
```

那么这个 *newtype* 关键字有关系吗？我们可能会将 `ZipList a` 类型这样定义：

```
data ZipList a = ZipList [a]
```

这样一个类型仅有一个值构造函数，且该构造函数只有一个字段，即列表的某值。同样我们可能会使用 record 语法，这样就可以自动获得一个从 `ZipList` 中提取列表的函数出来：

```
data ZipList a = ZipList { getZipList :: [a] }
```

这看起来不错，实际上也能很好的工作。使用 *data* 关键字包装改类型成为另一个类型，然后将新的这个类型成为 typeclass 的实例，这样便是第二种做法。

Haskell 中的 *newtype* 实际上就是用作于此。在实际的库中，`ZipList a` 的定义如下：

```
newtype ZipList a = ZipList { getZipList :: [a] }
```

没有使用 *data* 关键字，而是使用了 `newtype` 。为什么这么做呢？首先 *newtype* 更快，如果使用的是 *data* 关键字来包装一个类型，那么程序运行时进包与出包都会带来性能损失。但是

使用 *newtype* 时，Haskell 知道你使用的是现有类型包装后的新类型（正如其名），因为你想要的是同样的内核，而又不同的类型。

那么为什么不总是使用 *newtype* 而还有 *data* 呢？因为 *data* 可以有若干值构造函数，而每个构造函数又可以拥有零个或多个字段：

```
data Profession = Fighter | Archer | Accountant

data Race = Human | Elf | Orc | Goblin

data PlayerCharacter = PlayerCharacter Race Profession
```

而 *newtype* 就被限制在了一个字段的构造函数。

就像是 *data* 那样，对于 *newtype* 我们也可以使用 *deriving* 关键字。

```
newtype CharList = CharList { getCharList :: [Char] } deriving (Eq, Show)
```

```
ghci> CharList "this will be shown!"
CharList {getCharList = "this will be shown!"}
ghci> CharList "benny" == CharList "benny"
True
ghci> CharList "benny" == CharList "oisters"
False
```

这个特定的 *newtype*，值构造函数的类型是：

```
CharList :: [Char] -> CharList
```

相反的，被生成出来的 `getCharList` 函数其类型为：

```
getCharList :: CharList -> [Char]
```

它接受一个 `CharList` 值并转换为一个 `[Char]` 值。可以把这个想象成包装和解包，也可以认为将值从一个类型转换为另一个类型。

### 使用 newtype 来实现 typeclass 实例

很多时候会想要将我们的类型属于某个 typeclass，但是类型变量并没有符合我们想要的。把 `Maybe` 定义成 `Functor` 很容易，因为 `Functor` 的 typeclass 定义：

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

我们开始：

```
instance Functor Maybe where
```

接着实现 `fmap` 。所有的类型参数被填上，由于 `Maybe` 取代了 `Functor` 中 `f` 的位置，所以如果我们看看 `fmap` 应用在 `Maybe` 上时是什么样子时，它会是这样：

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

现在想让元组成为 `Functor` 的一个实例时，用 `fmap` 应用在这个元组，则会应用到该元组的第一个元素。例如 `fmap (+3) (1,1)` 会得到 `4,1` 。对于 `Maybe` ，只需要 `instance Functor Maybe where` ，这是因为对于只消费一个参数的类型构造函数很容易定义成 `Functor` 的实例，但是对于 `(a,b)` 这样的有若干个参数类型的就没办法。我们可以使用 `newtype` 来重新定义元组，使得第二个类型参数代表了元组中的第一个元素部分。

```
newtype Pair b a = Pair { getPair :: (a,b)}
```

接着定义 `Functor` 的实例，函数被映射到元组的第一个部分：

```
instance Functor (Pair c) where
  fmap f (Pair (x,y)) = Pair (f x, y)
```

如你所见，可以对 *newtype* 进行模式匹配，接着将函数 `f` 应用至元组的第一部分，再使用 `Pair` 值构造函数将元组转回 `Pair b a` 。 `fmap` 的类型则会是：

```
fmap :: (a -> b) -> Pair c a -> Pair c b
```

我们说 `instance Functor (Pair c) where` 以及 `Pair c` 取代了 `Functor` typeclass `f` 的位置：

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

现在如果转换一个元组为 `Pair b a` ，则可以使用 `fmap` ，同时函数将映射到第一部分：

```
ghci> getPair $ fmap (*100) (Pair (2,3))
(200,3)
ghci> getPair $ fmap reverse (Pair ("london calling", 3))
("gnillac nodnol",3)
```

### newtype 的惰性

我们提到过 *newtype* 通常比 *data* 要快，而 *newtype* 唯一能做的事情就是将现有的类型转换成一个新的类型，也就是说 Haskell 将通过 *newtype* 所定义类型视为原始类型。这也意味着 *newtype* 不仅更快，它还是惰性的。

之前提到过 Haskell 默认是惰性的，意味着计算只有要在打印函数的结果时才会发生，或者说只有当真正需要结果的时候才会进行计算。在 Haskell 中 `undefined` 代表会造成错误的计算，如果试着计算它，也就是将其打印至终端，Haskell 会抛出异常：

```
ghci> undefined
*** Exception: Prelude.undefined
```

而我们构建一个列表，其中包含一些 `undefined` 的值，且第一个不为 `undefined` 时使用 `head` ，那一切都会被顺利的计算，因为 Haskell 不需要列表中其他元素来得出结果：

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

考虑以下类型：

```
data CoolBool = CoolBool { getCoolBool :: Bool }
```

这是一个使用 `data` 关键字定义的代数数据类型，其包含一个值构造函数且只有一个类型为 `Bool` 的字段。编写一个函数用于模式匹配 `CoolBool` 并返回 `"hello"` ，无论里面的 `Bool` 是否为真：

```
helloMe :: CoolBool -> String
helloMe (CoolBool _) = "hello"
```

将其应用至 `undefined` ：

```
ghci> helloMe undefined
"*** Exception: Prelude.undefined
```

这里为什么又要抛出异常呢？通过 *data* 关键字所定义的类型可以拥有若干值构造函数，为了检查给与的函数是否匹配 `CoolBool _` 模式，在构造值的时候 Haskell 必须计算值才知道值构造函数是否被使用。当计算 `undefined` 值时，立刻抛出异常。

不使用 `data` 来定义 `CoolBool` 而是 `newtype` 时：

```
newtype CoolBool = CoolBool {getCoolBool :: Bool}
```

```
ghci> helloMe undefined
"hello"
```

能正常工作！这是为什么呢？正如之前所说的，使用 *newtype* 时 Haskell 内部可以将新的类型用旧的类型来表示。无需再套一层包装，只需要注意它们是不同的类型即可。又因为 Haskell 知道通过 *newtype* 关键字所创建的类型仅可以拥有一个构造函数，它无需计算传入至构造函数的值用于确保 `(CoolBool _)` 的模式，因为 *newtype* 类型仅允许值构造函数有一个字段。

### type vs newtype vs data

快速复习一下 `type` ， `data` 以及 `newtype` 。

`type` 关键字用于类型同义词，代表给予现有类型另一个名字：

```
type IntList = [Int]
```

这样可以允许我们使用 `IntList` 的名称来代指 `[Int]` 。两者可交换使用，但并不会因此有一个 `IntList` 的值构造函数，因为它们只是两种称呼同一个类型的方式，并不在乎用那个名称：

```
ghci> ([1,2,3] :: IntList) ++ ([1,2,3] :: [Int])
[1,2,3,1,2,3]
```

当我们想让类型签名更加清晰，给予我们跟了解函数的 context 时，则可以定义类型同义词。

`newtype` 关键字则是将现有的类型包装成一个新的类型，大部分是为了要让它们可以是特定的 typeclass 的实例。当使用 `newtype` 来包装一个现有的类型是，这个类型与原有类型是分开的。

```
newtype CharList = CharList { getCharList :: [Char] }
```

我们不能用 `++` 将 `CharList` 与 `[Char]` 拼接在一起，也不能用 `++` 来将两个 `CharList` 拼接在一起，这是因为 `++` 只能应用在列表上，而 `CharList` 并不是列表。

当我们在 `newtype` 声明中使用 record 语法的时候，我们会得到将新的类型转换成旧的类型的函数。

最后使用 `data` 关键字是为了定义自己的类型，它们可以在代数数据结构中放任意数量的构造函数。

## 幺半群 Monoids

Haskell 中的 typeclass 用于表示类型某些公共行为的接口。

`*` 是一个函数接受两个数值并使它们相乘。如果某数相乘的是 `1` ，那么结果还是某数其本身，无论是 `1 * x` 还是 `x * 1` 。同样的，`++` 是一个接受两个列表并结合它们的函数。与 `*` 一样，一个列表在结合空列表 `[]` 的情况下并不会改变列表自身。

看起来 `*` 与 `1` 以及 `++` 与 `[]` 都有共同的属性：

1. 函数接受两个参数。
2. 入参与返回值的类型相同。
3. 在使用二元函数时，存在一个值不会改变另一个值。

这两个操作还有共性的地方就是：当拥有若干值时，使用二元函数将这些值变为一个，计算顺序并不影响结果：

```
ghci> (3 * 2) * (8 * 5)
240
ghci> 3 * (2 * (8 * 5))
240
ghci> "la" ++ ("di" ++ "da")
"ladida"
ghci> ("la" ++ "di") ++ "da"
"ladida"
```

我们称这个属性为结合律 *associativity*。`*` 满足结合律，`++` 也是。而 `-` 不是，例如 `(5 - 3) - 4` 以及 `5 - (3 - 4)` 的结果并不一致。

这些属性就是**幺半群** *monoids*！一个幺半群就是有一个遵守结合律的二元函数，以及有一个可以相对那个函数作为 identity 的值。一个相对于一个函数的 identity 意为函数在调用它与另一个值时，结果永远是另一个值。看看幺半群是如何定义的：

```
class Monoid m where
  mempty :: m
  mappend :: m -> m -> m
  mconcat :: [m] -> m
  mconcat = foldr mappend mempty
```

`Monoid` typeclass 定义于 `import Data.Monoid` 。现在让我们花点时间来熟悉它。

首先可以看到只有具体类型才可以作为 `Monoid` 的实例，因为 `m` 并不接受任何类型参数。这与 `Functor` 以及 `Applicative` 不同，后两者的实例需要的是接受一个参数的类型构造函数。

第一个函数 `mempty` 。其实它并不是一个函数，因为它不接受参数，因此它是一个多态的常数，类似于 `Bounded` 的 `minBound` ，`mempty` 代表着特定幺半群的 identity 值。

接下来就是 `mappend` ，一个接受两个同类型值并返回同类型结果的二元函数。值得注意的是它的名字不太符合它真正的意思，它的名字隐含了要将两个东西连接在一起。

最后一个函数 `mconcat` 。它接受一个列表的幺半群值，并将它们用 `mappend` 简化成单一的值。它有一个默认的实现，就是从 `mempty` 作为起始值，然后用 `mappend` 进行 fold。对于大部分的实例而言默认实现没什么问题，我们也不会想要实现自己的 `mconcat` 。当我们定义一个类型属于 `Monoid` 时，大多数时候实现 `mempty` 与 `mappend` 就够了，而 `mconcat` 对于某些实例，有可能会有更高效的方式才需要自己去实现。

继续下去之前我们来看一下幺半群的定理。我们提到过必须要有一个值可作为对应二元函数的 identity 值，且该二元函数必须是满足结合律的。构造一个 `Monoid` 实例却不遵守这些规则，这样的实例对任何人都没有用，因为使用 `Monoid` typeclass，我们依靠的就是幺半群的定理。不然的话，这么做的意义就没有了。所以我们必须确保它们遵循：

1. `mempty `mappend` x = x`
2. `x `mappend` mempty = x`
3. `(x `mappend` y) `mappend` z = x `mappend` (y `mappend` z)`

### 列表是幺半群

列表的幺半群实例非常的简单：

```
instance Monoid [a] where
  mempty = []
  mappend = (++)
```

列表是 `Monoid` typeclass 的一个实例，无论其内部的元素类型。注意写的时候是 `instance Monoid [a]` 而不是 `instance Monoid []` ，因为 `Monoid` 需要一个具体类型作为实例。

测试：

```
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]
[]
```

注意最后一行，我们需要显式的类型注解，如果仅仅是 `mempty` ，GHCI 并不会知道使用哪个实例，因此我们需要宣称希望的是列表实例。

注意幺半群不需要 `a `mappend` b` 等同于 `b `mappend` a` 。列表显然不满足：

```
ghci> "one" `mappend` "two"
"onetwo"
ghci> "two" `mappend` "one"
"twoone"
```

### `Product` 与 `Sum`

我们已经验证了数值被视为幺半群，只需要二元函数是 `*` 以及 identity 值是 `1` 。实际上这并不是唯一的办法将数值视为幺半群。另一个方法就是 `+` 以及 identity 值 `0` ：

```
ghci> 0 + 4
4
ghci> 5 + 0
5
ghci> (1 + 3) + 5
9
ghci> 1 + (3 + 5)
9
```

幺半群定理成立。

`Data.Monoid` 模块为此提供了两种类型，即 `Product` 以及 `Sum` 。`Product` 定义如下：

```
newtype Product a =  Product { getProduct :: a }
  deriving (Eq, Ord, Read, Show, Bounded)
```

简单，就是一个 *newtype* 包装了一个类型参数同时带有一些衍生实例。其 `Monoid` 实例有点像这样：

```
instance Num a => Monoid (Product a) where
  mempty = Product 1
  Product x `mappend` Product y = Product (x * y)
```

`mempty` 是一个包装在 `Product` 构造函数中的 `1` 。`mappend` 模式匹配 `Product` 构造函数，将两值相乘后再包装回 `Product` 。如你所见，还有 `Num a` 类约束，意味着 `Product a` 要在所有 `a` 都是 `Num` 实例的情况下才能是 `Monoid` 的实例。那么作为一个幺半群来使用 `Product a` ，我们就需要做一下 *newtype* 的包装和解包：

```
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
```

这是一个很好的 `Monoid` typeclass 的用例，不过没有人真的会这样将两值相乘。稍后我们会说明尽管这样显而易见的定义还是有它方便的地方。

`Sum` 跟 `Product` 定义的方式相似，也可以：

```
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` 与 `All`

另一个有两种行为像是幺半群的类型就是 `Bool` 。第一个方式是 *or* 函数 `||` 作为二元函数以及 `False` 作为 identity 值。这样如果另一个值是 `True` 那么返回的就是 `True` ，如果是 `False` 则返回 `False` 。`Any` 这个 *newtype* 是 `Monoid` 的一个实例，其定义如下：

```
newtype Any = Any { getAny :: Bool }
  deriving (Eq, Ord, Read, Show, Bounded)
```

其实例像是这样：

```
instance Monoid Any where
  mempty = Any False
  Any x `mappend` Any y = Any (x || y)
```

示例：

```
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
```

另一个 `Bool` 表现为 `Monoid` 的方式是 `&&` 作为二元函数以及 `True` 作为 identity 值。其 *newtype* 定义如下：

```
newtype All = All { getAll :: Bool }
  deriving (Eq, Ord, Read, Show, Bounded)
```

其实例：

```
instance Monoid All where
  mempty = All True
  All x `mappend` All y = All (x && y)
```

示例：

```
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
```

### `Ordering` 幺半群

还记得 `Ordering` 类型么？它是比较运算后得到的结果，其包含三个值：`LT` ，`EQ` 以及 `GT` ：

```
ghci> 1 `compare` 2
LT
ghci> 2 `compare` 2
EQ
ghci> 3 `compare` 2
GT
```

对于列表，数值以及布尔值而言，找到幺半群很容易，因为它们都是一些常见的函数以及一些幺半群行为。而对于 `Ordering` 而言，我们需要花点力气才能辨别一个幺半群，不过实际上它的 `Monoid` 实例是非常符合直觉的：

```
instance Monoid Ordering where
  mempty = EQ
  LT `mappend` _ = LT
  EQ `mappend` y = y
  GT `mappend` _ = GT
```

这个实例这么运作：当 `mappend` 两个 `Ordering` 值时，左值被保留，除非左值是 `EQ` 那么则会保留右值作为结果。

例如如果我们逐字母比较单词 `"ox"` 与 `"on"` ，首先比较的是两个单词的首字母，看看它们是否相等然后再是第二个字母，`'x'` 在字母表是大于 `'n'` 的，因此前者比后者大。而 `EQ` 是 identity 这个直觉，可以在插入相同字母在两个单词的相同位置时，并不会改变字母表顺序得知。`"oix"` 仍然大于 `"oin"` 。

值得注意的是作为 `Monoid` 实例的 `Ordering` ，`x `mappend` y` 不等于 `y `mappend` x` 。因为第一个参数是保留着的，除非是 `EQ` ，`LT `mappend` GT` 得 `LT` ，而 `GT `mappend` LT` 得 `GT` ：

```
ghci> LT `mappend` GT
LT
ghci> GT `mappend` LT
GT
ghci> mempty `mappend` LT
LT
ghci> mempty `mappend` GT
GT
```

好了，那么幺半群如何有用的呢？假设你在编写一个函数接受两个字符串，比较它们长度并返回一个 `Ordering` 。如果两个字符串的长度相等，那么我们希望按字母进行比较。一种方式：

```
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
```

先比较长度的结果为 `a` ，再比较字典顺序的结果为 `b` ，长度一样时，则返回字典顺序。

如果善用 `Ordering` 是一种幺半群，那么可以简化函数：

```
lengthCompare :: String -> String -> Ordering
lengthCompare x y = (length x `compare` length y) `mappend` (x `compare` y)
```

与原文不一样之处在于无须 `import Data.Monoid` 。测试：

```
ghci> lengthCompare "zen" "ants"
LT
ghci> lengthCompare "zen" "ant"
GT
```

记住在使用 `mappend` 时，左参总是被保留的，除非是 `EQ` ，这时右参被保留。这也是为什么我们要将比较重要的顺序放在左参。如果要延展这个函数，要让它们比较元音的顺序，并把这个设为第二重要的比较条件，可以这样改写：

```
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")
```

这里写了一个帮助函数，接受一个字符串并告知有多少个元音拥有筛选，再应用 `length` 。测试：

```
ghci> lengthCompare "zen" "anna"
LT
ghci> lengthCompare "zen" "ana"
LT
ghci> lengthCompare "zen" "ann"
GT
```

第一个例子，`"zen"` 长度短于 `"anna"` 返回 `LT` 。第二个例子，长度一样，但第二个字符串元音多，返回 `LT` 。第三个例子，长度相等，元音个数相等，最后经过字典顺序得 `"zen"` 更大。

`Ordering` 的幺半群允许我们用不同方式比较事务，并根据重要度的不同而定义比较的顺序。

### `Maybe` 幺半群

让我们看看 `Maybe a` 构造 `Monoid` 实例的若干方式，以及这些实例的用途。

一种让 `Maybe a` 视作幺半群的方式是其类型参数 `a` 是幺半群，接着实现 `mappend` 将包在 `Just` 里的至对应 `mappend` ，同时让 `Nothing` 作为 identity。使用 `Nothing` 作为 identity 这样如果 `mappend` 的两个值其中一个是 `Nothing` 时，保留另一个。以下是实例声明：

```
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)
```

注意这里的类约束，即只有当 `a` 是 `Monoid` 时，`Maybe a` 才可作为 `Monoid` 实例。接下来就是 `mappend` 任意一方是 `Nothing` 时，结果则是另一方。如果 `mappend` 两个 `Just` 值，那么就 `mappend` 两个 `Just` 的内容，用 `Just` 包装后再返回值。

```
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})
```

那么如果 `Maybe` 的内容并非 `Monoid` 实例呢？在我们不知道内容是否为幺半群的情况下，是不能 `mappend` 两个值的，那么该怎么做？一种方式是丢弃第二个值并保留第一个。为此，`First a` 类型出现了，其定义：

```
newtype First a = First { getFirst :: Maybe a }
  deriving (Eq, Ord, Read, Show)
```

接受一个 `Maybe a` 将其用 *newtype* 包装。`Monoid` 实例：

```
instance Monoid (First a) where
  mempty = First Nothing
  First (Just x) `mappend` _ = First (Just x)
  First Nothing `mappend` x = x
```

`mempty` 是包在 `First` 中的 `Nothing` 。如果 `mappend` 的第一个参数是 `Just` ，那么直接忽略第二个参数；如果第一个参数是 `Nothing` ，则直接返回第二个参数，无论是 `Just` 或 `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` 在我们有用若干 `Maybe` 值，并想要知道它们中任意一个是否为 `Just` 时，非常的有用。利用 `mconcat` ：

```
ghci> getFirst . mconcat . map First $ [Nothing, Just 9, Just 10]
Just 9
```

如果我们希望 `Maybe a` 保留的是第二个参数的幺半群实现时，`Data.Monoid` 提供了一个 `Last a` 类型：

```
ghci> getLast . mconcat . map Last $ [Nothing, Just 9, Just 10]
Just 10
ghci> getLast $ Last (Just "one") `mappend` Last (Just "two")
Just "two"
```

### 使用幺半群来 fold 数据结构

幺半群的另一个有趣的使用方式就是让它来帮助我们 fold 一些数据结构。目前为止我们只 fold 过列表，不过列表并不是唯一一种可以 fold 的数据结构。我们几乎可以 fold 任意一种数据结构，例如树。

正因为太多的数据结构可以被 fold，`Foldable` typeclass 就被引入了。很像 `Functor` 用作于映射，`Foldable` 用于任何可以被 fold 起来的东西！可以在 `Data.Foldable` 中找到它，又因为与 `Prelude` 中多数名称重名的缘故，我们需要：

```
import Foldable qualified as F
```

让我们来看看与 `Prelude` 中的 `foldr` ， `foldl` ， `foldr1` 以及 `foldl1` 有什么区别。

```
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
```

`foldr` 接受一个列表并进行 fold，而 `Data.Foldable` 中的 `foldr` 则是接受任意可被 fold 的类型。它们作用于列表的行为一致：

```
ghci> foldr (*) 1 [1,2,3]
6
ghci> F.foldr (*) 1 [1,2,3]
6
```

那么其它类型呢？比如说 `Maybe` ：

```
ghci> F.foldl (+) 2 (Just 9)
11
ghci> F.foldr (||) False (Just True)
True
```

现在来看一下之前章节定义的树结构：

```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show, Read, Eq)
```

一棵树要么是一颗空树，要么是一个包含值的节点且还指向另外两棵树。定义它之后我们还将其定义成 `Functor` 的实例，现在将它定义成 `Foldable` 的实例。一种方式是实现 `foldr` ，另一种比较简单的方式是实现 `foldMap` ，它也属于 `Foldable` typeclass。`foldMap` 的类型如下：

```
foldMap :: (Monoid m, Foldable t) => (a -> m) -> t a -> m
```

第一个参数是一个函数，该函数接受一个 foldable 结构包含的类型（这里就是 `a` ）并返回一个 monoid 值，第二个参数就是包含类型 `a` 的 foldable 结构。它将函数映射至 foldable 结构上，因此生产一个包含了 monoid 值的 foldable 结构。接着用 `mappend` 在这两个 monoids 值上，并返回单个 monoid 值。比较酷的是只要实现了这个函数，那么就可以让我们的函数成为

`Foldable` 。所以只要实现某个类型的 `foldMap` ，就可以得到那个类型的 `foldr` 与 `foldl` 。

接下来就是定义 `Tree` 成为 `Foldable` 实例：

```
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
```

我们这样思考：如果提供一个接受树元素并返回一个 monoid 值的函数，我们如何减少我们的树变为单个 monoid 值呢？我们在做 `fmap` 树的时候是将函数应用在节点上，接着递归的应用函数在做子树与右子树上。这里我们不仅仅是要映射一个函数，还要通过使用 `mappend` 合并值返回一个 monoid 值。首先是考虑空树，它不包含任何值，因此它变为 `mempty` 。

非空树比较有趣。它包含了两个子树同时还有值。这种情况下使用 `foldMap` 递归同一个函数 `f` 在左右子树上。记住，`foldMap` 返回一个 monoid 值。函数 `f` 也应用在节点的值上。这时就有了三个 monoid 值（两个从子树而来，一个从节点而来），我们希望将它们合并成为一个值。为此，使用 `mappend` ，自然而然的从左子树开始，接着是节点，再然后就是右子树。

注意我们并不一定要提供一个将普通值转成 monoid 的函数。我们只是把它当做 `foldMap` 的参数，要决定的是如何应用该函数，将多个 monoids 转换成一个 monoid。

考虑下面这棵树：

```
testTree =
  Node
    5
    ( Node
        3
        (Node 1 Empty Empty)
        (Node 6 Empty Empty)
    )
    ( Node
        9
        (Node 8 Empty Empty)
        (Node 10 Empty Empty)
    )
```

对其进行 fold：

```
ghci> F.foldl (+) 0 testTree
42
ghci> F.foldl (*) 1 testTree
64800
```

`foldMap` 不仅对创建新的 `Foldable` 实例有用；它对简化我们的结构称为单一的 monoid 值也有用。例如我们想要知道树中有没有 `3` ，我们可以这么做：

```
ghci> getAny $ F.foldMap (\x -> Any $ x == 3) testTree
True
```

此处的 `x-> Any $ x == 3` 接受一个数值并返回一个 monoid，即一个包在 `Any` 中的 `Bool` 。`foldMap` 将这个函数应用至树的每个节点，并把结果用 `mappend` 简化成单一的 monoid。如果这么做：

```
ghci> getAny $ F.foldMap (\x -> Any $ x > 15) testTree
False
```

我们也能将 `foldMap` 配合 `\x -> [x]` 使用来将树转成列表：

```
ghci> F.foldMap (\x -> [x]) testTree
[1,3,6,5,8,9,10]
```

# 12 A Fistful of Monads

当我们第一次谈及函子时，得知这是一个用于映射的抽象概念。接着就是高级函子，让我们将某些类型视为在某些 contexts 中，保留 contexts 的同时还能应用普通函数在 contexts 中的值上。

本章我们开始学习单子 monads，它是一个增强版的高级函子，正如是一个高级函子是函子的增强版那样。

单子是高级函子的自然演进，可以这样考虑：如果有一个带 context 的值，`m a` ，该如何将其应用至一个接受普通 `a` 并返回一个 context 的函数？也就是说，如何应用一个类型为 `a -> m b` 的函数至一个类型为 `m a` 的值？概括一下就是想要一个这样的函数：

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

**当我们拥有一个漂亮的值以及一个接受普通值但返回漂亮值的函数，我们如何才能将漂亮值喂给这个函数?**这就是单子处理的主要问题。这里用 `m a` 而不是 `f a` 是因为 `m` 意为 `Monad` ，而单子就是支持 `>>=` 操作的高级函子，此处的 `>>=` 函数读作绑定 *bind*。

## 从处理 `Maybe` 开始

不出意外，`Maybe` 就是一个单子，让我看看它是如何以单子运作的。

`Maybe a` 代表一个类型 `a` 的值处于有可能失败的 context 中。`Just "dharma"` 意为字符串 `"dharma"` 同时 `Nothing` 代表着缺值，或者说如果将字符串作为计算的结果，那么 `Nothing` 则代表计算失败了。

将 `Maybe` 视为一个函子时，想要的是 `fmap` 一个函数在其值，也就是 `Just` 中的元素，否则保留 `Nothing` 状态，即内部没有任何元素。

```
ghci> fmap (++"!") (Just "wisdom")
Just "wisdom!"
ghci> fmap (++"!") Nothing
Nothing
```

将 `Maybe` 视为一个高级函子时，applicatives 同样包装了函数。使用 `<*>` 应用一个函数在 `Maybe` 内的值，它们都必须是包在 `Just` 来代表值存在，否则返回 `Nothing` 。

```
ghci> Just (+3) <*> Just 3
Just 6
ghci> Nothing <*> Just "greed"
Nothing
ghci> Just ord <*> Nothing
Nothing
```

当我们使用 applicative 风格让普通函数作用在 `Maybe` 值时，所有的值都需要时 `Just` 值，否则返回 `Nothing` ！

```
ghci> max <$> Just 3 <*> Just 6
Just 6
ghci> max <$> Just 3 <*> Nothing
Nothing
```

现在让我们想想该如何使用 `>>=` 在 `Maybe` 上。正如我们所说的，`>>=` 接受一个 monadic 值以及一个接受普通值并返回一个 monadic 值的函数。

`>>=` 将接受一个 `Maybe a` 值以及一个类型 `a -> Maybe a` 的函数，以某种方式应用该函数至 `Maybe a` 。假设我们有一个函数 `\x -> Just (x+1)` 。它接受一个值，加 `1` 并将其包装在一个 `Just` 内。

```
ghci> (\x -> Just (x+1)) 1
Just 2
ghci> (\x -> Just (x+1)) 100
Just 101
```

现在的问题是如何将一个 `Maybe` 值喂给该函数？如果思考了 `Maybe` 是如何作为一个高级函子的，那么该问题的答案就很简单了。如果拿到一个 `Just` ，就把包在 `Just` 内的值喂给函数；如果拿到一个 `Nothing` 则返回 `Nothing` 。

现在我们不调用 `>>=` 而是调用 `applyMaybe` ，那么它接受一个 `Maybe a` 以及一个返回 `Maybe b` 的函数，并将该函数应用至 `Maybe a` ：

```
applyMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
applyMaybe Nothing f  = Nothing
applyMaybe (Just x) f = f x
```

现在让我们试试：

```
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
```

上述例子中，可以看到当使用 `applyMaybe` 于一个 `Just` 值以及一个函数，该函数很容易的就被应用到了 `Just` 内的值；而但作用于一个 `Nothing` ，那么结果就是 `Nothing` 。那么如果函数返回的是 `Nothing` ？

```
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
```

符合预期。如果左侧的 monadic 值是 `Nothing`，那么整个结果就是 `Nothing`；如果右侧函数返回的是 `Nothing`，结果还是 `Nothing`。这非常像是使用 `Maybe` 作为高级函子时，过程中有任何一个 `Nothing` 时，整个结果就会是 `Nothing`。

你或许会问，这样有用么？看起来高级函子比单子更强，因为高级函子允许我们用一个普通函数应用至 contexts 中的值上。而单子可以这么做是因为它们是升级版的高级函子，那肯定是可以做高级函子不能做的事情。

稍后我们再来讨论 `Maybe`，现在让我们看看属于单子的那些 typeclass。

## 单子 typeclass

正如函子拥有 `Functor` typeclass，高级函子拥有 `Applicative` typeclass，单子也有自己的 typeclass：`Monad`！

```
1  class Monad m where
2    return :: a -> m a
3
4    (>>=) :: m a -> (a -> m b) -> m b
5
6    (>>) :: m a -> m b -> m b
7    x >> y = x >>= \_ -> y
8
9    fail :: String -> m a
10   fail msg = error msg
```

首先从第一行开始，`class Monad m where`。等等之前不是提到过单子是高级函子的演进吗？那么不应该有一个类约束像是 `class (Applicative m) => Monad m where`，也就是一个类型必须先是高级函子才能是单子么？的确要有，但是 Haskell 被创造的早期，人们没有想到高级函子适合被放进语言中，所以最后并没有这个限制。但的确每个单子都是高级函子，即使 `Monad` 并没有这么宣告。

`Monad` typeclass 的第一个函数定义就是 `return`。它等同于 `pure`，只不过名字不同，其类型为 `(Monad m) => a -> m a`。接受一个值，将其放入最小默认 context 中。对于 `Maybe` 而言，就是将值放入 `Just` 中。

接下来的是函数 `>>=`，或绑定。它像是函数应用那样，只不过它接受的不是普通值而是一个 monadic 值（即具有 context 的值）并把该值喂给一个接受普通值的函数，最后返回一个 monadic 值。

接下来就是函数 `>>`，现在无需太多的关注因为它拥有一个默认实现，同时在构造 `Monad` 实例时我们几乎永远不用考虑去实现它。

最后的函数就是 `Monad` typeclass 的 `fail`。我们永远不会显式的在代码中用到，而是会被 Haskell 用在处理语法错误。目前不需要太在意 `fail`。

现在来看一下 `Maybe` 的 `Monad` 实例。

```
instance Monad Maybe where
  return x = Just x
  Nothing >>= f = Nothing
  Just x >>= f  = f x
  fail _ = Nothing
```

`return` 与 `pure` 等价。

`>>=` 与 `applyMaybe` 是一样的。当 `Maybe a` 喂给我们函数时，我们保留 context，并在左值为 `Nothing` 时返回 `Nothing` ，左值为 `Just` 时将 `f` 应用至其内部值。

```
ghci> return "WHAT" :: Maybe String
Just "WHAT"
ghci> Just 9 >>= \x -> return (x*10)
Just 90
ghci> Nothing >>= \x -> return (x*10)
Nothing
```

我们已经在 `Maybe` 使用过 `pure` 了，这里的 `return` 就是 `pure` 。

注意是如何把 `Just 9` 喂给 `\x -> return (x*10)` 的。在函数中 `x` 绑定到 `9` 。它看起来不用模式匹配就能从 `Maybe` 中抽取值，且并没有丢失 `Maybe` 的 context。

### 走钢丝

现在我们知道了如何将一个 `Maybe a` 值喂给 `a -> Maybe b` 类型的函数。现在看看我们如何重复使用 `>>=` 来处理多个 `Maybe a` 值。

... 省略原文故事。

我们用一对整数来代表平衡杆的状态。第一个位置代表左侧的鸟的数量，第二个位置代表右侧的鸟的数量。

```
type Birds = Int
type Pole = (Birds,Birds)
```

接下来定义两个函数，它们接受一个代表鸟的数量的数值以及分别放在杆子的左右侧。

```
landLeft :: Birds -> Pole -> Pole
landLeft n (left,right) = (left + n,right)

landRight :: Birds -> Pole -> Pole
landRight n (left,right) = (left,right + n)
```

测试：

```
ghci> landLeft 2 (0,0)
(2,0)
ghci> landRight 1 (1,2)
(1,3)
ghci> landRight (-1) (1,2)
(1,1)
```

鸟儿飞走只需要用负值表达即可。由于函数的输入与返回都是 `Pole` 的缘故，我们可以串联 `landLeft` 与 `landRight` ：

```
ghci> landLeft 2 (landRight 1 (landLeft 1 (0,0)))
(3,1)
```

如果编写这样的一个函数：

```
x -: f = f x
```

那么就可以先写参数，然后再是函数：

```
ghci> 100 -: (*3)
300
ghci> True -: not
False
ghci> (0,0) -: landLeft 2
(2,0)
```

使用这样的方法我们可以用更加可读的方式来重复之前的表达：

```
ghci> (0,0) -: landLeft 1 -: landRight 1 -: landLeft 2
(3,1)
```

那么如果一次性飞入 10 只鸟儿呢？

```
ghci> landLeft 10 (0,3)
(10,3)
```

平衡杆会失去平衡（超过 4 只鸟）！这是显而易见的，但是如果在一系列的操作当中：

```
ghci> (0,0) -: landLeft 1 -: landRight 4 -: landLeft (-1) -: landRight (-2)
(0,2)
```

这看起来没问题，但是实际上中间有一刻是右侧有 4 只鸟，而左侧没有鸟！修复这个问题，我们需重审 `landLeft` 与 `landRight` 函数。这些函数是需要返回失败的。这就是使用 `Maybe` 的绝佳时刻！

```
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
```

测试：

```
ghci> landLeft 2 (0,0)
Just (2,0)
ghci> landLeft 10 (0,3)
Nothing
```

现在我们需要一种方法,接受一个 `Maybe Pole` ,将其喂给接受 `Pole` 并返回 `Maybe Pole` 的函数。幸运的是，我们拥有 `>>=` ：

```
ghci> landRight 1 (0,0) >>= landLeft 2
Just (2,1)
```

注意，`landLeft 2` 的类型是 `Pole -> Maybe Pole` 。我们无法将 `landRight 1 (0,0)` 的 `Maybe Pole` 结果喂给它，因此使用了 `>>=` 来将带有 context 的值给到了 `landLeft 2` 。`>>=` 确实允许我们将 `Maybe` 值视为一个带有 context 的值，因为如果将一个 `Nothing` 喂给 `landLeft 2` ，那么结果就是 `Nothing` 且失败被传递下去：

```
ghci> Nothing >>= landLeft 2
Nothing
```

测试一系列的操作：

```
ghci> return (0,0) >>= landRight 2 >>= landLeft 2 >>= landRight 2
Just (2,4)
```

开始时通过 `return` 将一个 pole 包装进一个 `Just` 。

稍早之前的一系列操作：

```
ghci> return (0,0) >>= landLeft 1 >>= landRight 4 >>= landLeft (-1) >>= landRight (-2)
Nothing
```

符合预期，最后的情形代表了失败的情况。

如果只把 `Maybe` 当做高级函子使用的话是没有办法达到我们想要的效果。因为高级函子并不允许 applicative 值之间有弹性的交互。它们最多就是让我们可以用 applicative 风格来传递参数至函数。applicative 操作符拿到它们的结果并用 applicative 的方式喂给另一个函数，最后将最终的 applicative 值放在一起。这里面每一步之间并没有多少操作空间。而我们的这个例子需要的是每一步都依赖前一步的结果。

我们也可以写一个 `banana` 函数必定返回失败：

```
banana :: Pole -> Maybe Pole
banana _ = Nothing
```

将该函数置于整个过程中，不管前面的状态如何，都会产生失败：

```
ghci> return (0,0) >>= landLeft 1 >>= banana >>= landRight 1
Nothing
```

`Just (1,0)` 被喂给 `banana` 产生 `Nothing` 之后所有的结果便是 `Nothing` 了。

除开构建一个忽略输入并返回一个预先设定好的 monadic 值，还可以使用 `>>` 函数，其默认实现如下：

```
(>>) :: (Monad m) => m a -> m b -> m b
m >> n = m >>= \_ -> n
```

一般而言，碰到一个完全忽略前面状态的函数，它就只会返回它想返回的值。然而碰到单子时，它们的 context 还是必须要被考虑到的。看一下 `>>` 串联 `Maybe` 的情况。

```
ghci> Nothing >> Just 3
Nothing
ghci> Just 3 >> Just 4
Just 4
ghci> Just 3 >> Nothing
Nothing
```

如果把 `>>` 换成 `>>= \_ ->` ，就很容易理解了。

将 `banana` 改用 `>>` 与 `Nothing` 来表达：

```
ghci> return (0,0) >>= landLeft 1 >> Nothing >>= landRight 1
Nothing
```

注意 `Maybe` 对 `>>=` 的实现，它其实就是在遇到 `Nothing` 时返回 `Nothing` ，遇到 `Just` 值时继续用 `Just` 传值。

## do 表示法

Haskell 中的单子非常的有用，它们得到了属于自己的特殊语法，即 `do` 表示法。我们已经学习到了 `do` 标记法用作 I/O，同时将若干 I/O actions 粘合成为一个。实际上 `do` 表示法不仅仅作用于 `IO` ，而是可以用于任意单子。其原则仍然不变：顺序的粘合其他的 monadic 值。现在让我们看看 `do` 表示法是如何工作的，以及为什么这么有用。

考虑以下 monadic 应用的例子：

```
ghci> Just 3 >>= (\x -> Just (show x ++ "!"))
Just "3!"
```

那么如果还有另一个 `>>=` 在函数内呢？

```
ghci> Just 3 >>= (\x -> Just "!" >>= (\y -> Just (show x ++ y)))
Just "3!"
```

一个嵌套的 `>>=` !在最外层的 lambda 函数，将 `Just "!"` 喂给 lambda `\y -> Just (show x ++ y)` 。在内部的 lambda， `y` 变成 `"!"` 。 `x` 仍然是 `3` 因为是从外层的 lambda 取值的。这些行为让我们想到了下列式子：

```
ghci> let x = 3; y = "!" in show x ++ y
"3!"
```

差别在于前者的值是 monadic，带有可能失败的 context。我们可以把其中任何一步替换成失败的状态：

```
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
```

为了解释的更清楚，我们来编写一个自己的 `Maybe` 脚本：

```
foo :: Maybe String
foo = Just 3 >>= (\x -> Just "!" >>= (\y -> Just (show x ++ y)))
```

为了避免这些麻烦的 lambda，Haskell 允许我们使用 `do` 表示法：

```
foo :: Maybe String
foo = do
  x <- Just 3
  y <- Just "!"
  Just (show x ++ y)
```

看起来好像是不必每一步都去检查 `Maybe` 值是 `Just` 或 `Nothing` 。如果任意步骤取出了 `Nothing` ，那么整个 `do` 的结果就会是 `Nothing` 。我们把所有责任都交给 `>>=` ，它来处理所有的 `context` 问题。这里的 `do` 表示法就是另一种语法的形式来串联所有的 monadic 值。

在 `do` 表达式中，每一行都是一个 monadic 值。想要获取结果，需要 `<-` 。如果有一个 `Maybe String` ，同时我们通过 `<-` 绑定它到一个变量上，那么该变量将变为一个 `String` ，就像是我们使用 `>>=` 将 monadic 值喂给 lambdas 那样。`do` 表达式中，最后一个 monadic 值，例如这里的 `Just (show x ++ y)` ，不可以将它绑定至一个结果，因为这样的写法转换成 `>>=` 的结果时不合理。它必须是所有 monadic 值粘合后最终的结果，因此需要考虑前面所有可能失败的情景。

例如以下：

```
ghci> Just 9 >>= (\x -> Just (x > 8))
Just True
```

因为 `>>=` 的左参是一个 `Just` 值，lambda 被应用至 `9` ，同时返回是一个 `Just True` 。如果将上述重写为 `do` 表示法，可得：

```
marySue :: Maybe Bool
marySue = do
    x <- Just 9
    Just (x > 8)
```

比较这两种写法，很容易看出来为什么整个 monadic 值的结果会是在 `do` 表示法中最后一个，因为它串联了前面所有的结果。

用 `do` 表示法来改写 routine 函数：

```
routine :: Maybe Pole
routine = do
```

```
    start <- return (0,0)
    first <- landLeft 2 start
    second <- landRight 2 first
    landLeft 1 second
```

测试：

```
ghci> routine
Just (3,2)
```

中间插入香蕉皮：

```
routine :: Maybe Pole
routine = do
    start <- return (0,0)
    first <- landLeft 2 start
    Nothing
    second <- landRight 2 first
    landLeft 1 second
```

在 `do` 表示法中没有通过 `<-` 绑定 monadic 值时，就像是将 `>>` 放在 monadic 值之后，即忽略计算的结果。我们只是要让它们有序，而不是要它们的结果，而且这比写 `_ <- Nothing` 要好看。

何时使用 `do` 表示法或是 `>>=` 取决于你。

在 `do` 表示法中，我们还可以用模式匹配来绑定 monadic 值，就好像我们在 `let` 表达式以及函数参数那样：

```
justH :: Maybe Char
justH = do
  (x : xs) <- Just "hello"
  return x
```

如果这个模式匹配失败了呢？函数中的模式匹配失败后会去匹配下一个模式，如果所有的模式都匹配不上，那么错误将被抛出，然后程序就挂掉了。另一方面，如果在 `let` 中进行模式匹配失败会直接抛出错误。毕竟在 `let` 表达式的情况下没有失败就跳至下一个选项的设计。至于在 `do` 表示法中模式匹配失败时，就会调用 `fail` 函数。它是 `Monad` typeclass 中的一部分，它运行在现在的 context 下失败时不会挂掉程序。它的默认实现如下：

```
fail :: (Monad m) => String -> m a
fail msg = error msg
```

默认情况下确实是让程序崩溃，不过一些单子（例如 `Maybe` ）表示可能失败的 context 时，通常会实现自己的失败函数。例如 `Maybe` 的实现如下：

```
fail _ = Nothing
```

它忽视所有错误并创造一个 `Nothing` 。

```
justH :: Maybe Char
justH = do
  (x : _) <- Just "hello";
  return x
```

这里模式匹配失败，产生的影响等同于一个 `Nothing` 。测试：

```
ghci> wopwop
Nothing
```

这样模式匹配的失败只会限制在单子的 context 中，而不会让程序崩溃，这样的处理方式好很多。

## 列表单子

目前为止我们学习了 `Maybe` 是如何被视为可能失败的 context，也学习了如何用 `>>=` 来把这些可能失败的值传给函数。这一小节中，我们开始学习如何用列表的 monadic 性质来写非确定性 non-deterministic 的程序。

把列表当做高级函子有如下特性：

```
ghci> (*) <$> [1,2,3] <*> [10,100,1000]
[10,100,1000,20,200,2000,30,300,3000]
```

所有可能的组合相乘。

这个非确定性的 context 可以很漂亮的转换为单子。列表的 `Monad` 实例如下：

```
instance Monad [] where
return x = [x]
xs >>= f = concat (map f xs)
fail _ = []
```

`return` 即 `pure` ，不多赘述。

`>>=` 意为接受一个带有 context 的值（一个 monadic 值），将其喂给一个接受普通值的函数并返回一个带有 context 的值。尝试一下将一个非确定性值喂给一个函数：

```
ghci> [3,4,5] >>= \x -> [x,-x]
[3,-3,4,-4,5,-5]
```

在对 `Maybe` 使用 `>>=` 时，monadic 值喂给函数时考虑到了可能失败的 context。而列表的 `>>=` 则考虑到了非确定性。`[3,4,5]` 是一个非确定性的值，将它喂给一个返回非确定性值的函数，那么结果也会是非确定性的。该函数接受一个数值并生产两个结果：一个负数，一个正数。因此当使用 `>>=` 将这个列表喂给函数，每个列表中的数值保留了原有的符号，又新增了相反符号的值。

为了了解这是如何做到的，我们仅需跟着实现。首先起始的是列表 `[3,4,5]` ，接着将 lambda 应用在它们上，得到：

```
[[3,-3],[4,-4],[5,-5]]
```

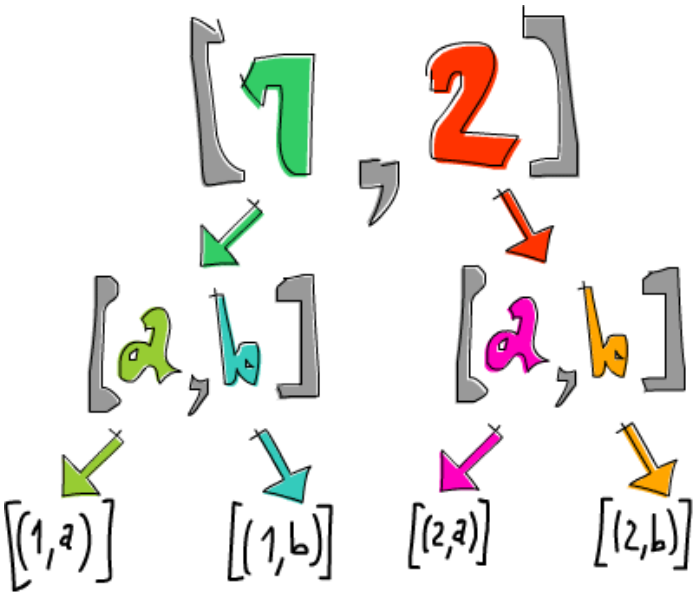
lambda 应用在了列表中的每个元素上，最后打平列表，这样就应用了一个非确定性的函数至一个非确定性的值了！

非确定性也有考虑到失败的可能。`[]` 其实就等价于 `Nothing` ，因为它没有结果。所以失败等同于返回一个空列表。

```
ghci> [] >>= \x -> ["bad","mad","rad"]
[]
ghci> [1,2,3] >>= \x -> []
[]
```

和 `Maybe` 一样，我们可以用 `>>=` 把它们串起来：

```
ghci> [1,2] >>= \n -> ['a','b'] >>= \ch -> return (n,ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```




列表 `[1,2]` 绑定至 `n` ， `['a','b']` 绑定至 `ch` 。接着使用 `return (n,ch)` （或是 `[(n,ch)]` ），意为将一对 `n,ch` 并将其放入默认最小 context 中。可以这么理解：对于 `[1,2]` 中的每个元素而言，经历 `['a','b']` 中每个元素，并生产一对元组。

一般而言， `return` 接受一个值并将其放入最小默认 context 中，它是不会多做额外的事情，仅仅只是用于展出结果。

**Note**

在处理非确定性值的时候，可以把列表中的每个元素想象成计算路线中的一个分支。

将之前的表达式用 `do` 重写：

```
listOfTuples :: [(Int, Char)]
listOfTuples = do
  n <- [1, 2]
  ch <- ['a', 'b']
  return (n, ch)
```

这样写可以更清楚的知道 `n` 走过 `[1,2]` 中的每个值，而 `ch` 则走过 `['a','b']` 中的每个值。正如 `Maybe` 那样，从 monadic 值中取出普通值再喂给函数。`>>=` 会帮我们处理好一切 context 相关的问题，不同之处在于列表的 context 指的是非确定性。

使用 `do` 表示法很想我们之前做的：

```
ghci> [ (n,ch) | n <- [1,2], ch <- ['a','b'] ]
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

正是列表表达式！在之前的 `do` 表达式的例子中，`n` 走过 `[1,2]` 的每个元素，`ch` 走过 `['a','b']` 的每个元素，接着把 `(n,ch)` 放进一个 context 中。这跟列表表达式的目的一样，只是列表表达式不需要在最后用 `return` 来得到 `(n,ch)` 结果。

实际上，列表表达式是一个语法糖。无论是列表表达式还是 `do` 表示法，都会转换成使用 `>>=` 来做非确定性特征的计算。

列表表达式允许我们过滤输出。例如仅收集含有数字 `7` 的数字：

```
ghci> [ x | x <- [1..50], '7' `elem` show x ]
[7,17,27,37,47]
```

我们将 `show` 应用至 `x` 上使数值变为字符串再查看字符 `'7'` 是否存在在字符串中。为了知道筛选列表表达式是如何转换成列表单子的，我们需要检查 `guard` 函数以及 `MonadPlus` typeclass。该 typeclass 用于表达同时可以表现为幺半群的单子，以下是其定义：

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

`mzero` 是 `Monoid` typeclass 中 `mempty` 的同义词，`mplus` 则关联 `maapend` 。列表既是幺半群又是单子，因此可以实现 `MonadPlus` 这个 typeclass：

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

对于列表而言，`mzero` 代表一个非确定性计算且没有任何结果 – 一个失败的计算，`mplus` 将两个非确定性的值结合成一个。`guard` 函数定义如下：

```
guard :: (MonadPlus m) => Bool -> m ()
guard True = return ()
guard False = mzero
```

它接受一个布尔值，如果为 `True` ，则接受一个 `()` 并将其置入最小默认 context 中代表成功；否则创建一个失败的 monadic 值。

```
ghci> guard (5 > 2) :: Maybe ()
Just ()
ghci> guard (1 > 2) :: Maybe ()
Nothing
ghci> guard (5 > 2) :: [()]
[()]
ghci> guard (1 > 2) :: [()]
[]
```

看起来很有趣，不过如何有用呢？在列表单子中，我们用它来过滤非确定性计算。

```
ghci> [1..50] >>= (\x -> guard ('7' `elem` show x) >> return x)
[7,17,27,37,47]
```

这里的结果与之前的列表表达式是一致的。那么 `guard` 是如何工作的呢？让我们看看 `guard` 函数是如何与 `>>` 关联的：

```
ghci> guard (5 > 2) >> return "cool" :: [String]
["cool"]
ghci> guard (1 > 2) >> return "cool" :: [String]
[]
```

如果 `guard` 成功，得到一个空的元组，接着用 `>>` 来忽略空的元组，得到不同的结果；反之， `return` 失败。这是因为用 `>>=` 把空的列表喂给函数总是返回空的列表。 `guard` 大致的意思就是：如果一个布尔值为 `False` 那么久产生一个失败的状态，反之返回一个 `()` 。如此计算可以继续进行。

用 `do` 改写之前的例子：

```
sevensOnly :: [Int]
sevensOnly = do
  x <- [1 .. 50]
  guard ('7' `elem` show x)
  return x
```

这里不写最后一行的 `return x` ，整个列表就会是包含一堆空元组的列表。

上述例子写成列表表达式就是这样：

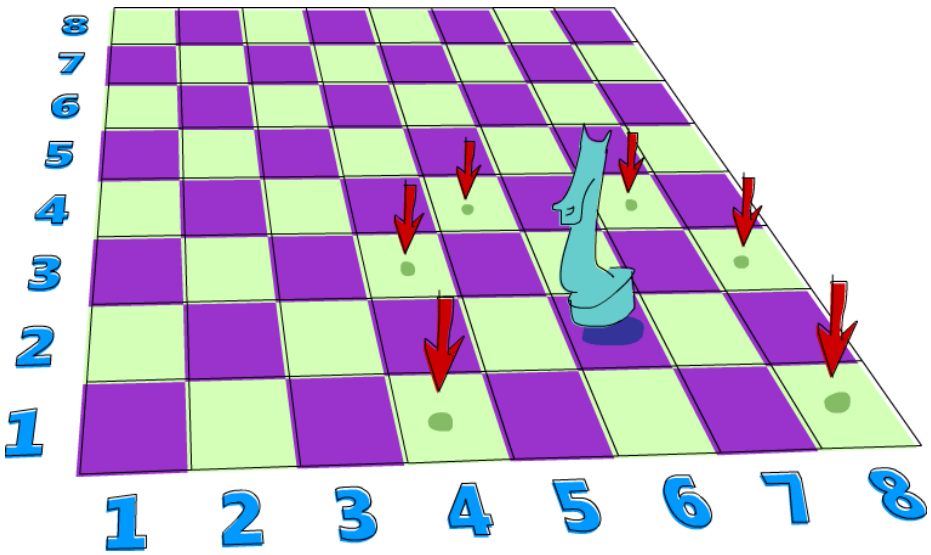
```
ghci> [ x | x <- [1..50], '7' `elem` show x ]
[7,17,27,37,47]
```

因此列表表达式的过滤基本上跟 `guard` 是一致的。

### 一个骑士任务

这里有一个用非确定性解决的问题。假设有一个棋盘以及一个骑士棋子，我们想要知道骑士是否可以在三步之内移动到想要的位置。我们只需要一对数值来表示骑士在棋盘上的位置。

第一个数值代表棋盘行数，第二个代表列数。




用一个类型来模拟骑士所处棋盘的位置：

```
type KnightPos = (Int, Int)
```

假设起始点 `(6,2)` ，那么能用三步到 `(6,1)` 么？下一步的最优解是什么？不如全部一起考虑！要好好利用所谓的非确定性。所有我们不是只选择一步，而是选择全部。首先写一个函数返回所有可能的下一步：

```
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')
```

骑士的移动为日字形（参考象棋中的马）即横一竖二或者横二竖一。`(c',r')` 走过列表中的每个元素，`guard` 用于保证移动后停留在棋盘上，否则产生一个空的列表代表失败，这样 `return (c',r')` 就不会执行。

如果不用列表单子写，那么也可以用 `filter` 来实现：

```
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]
```

测试：

```
ghci> moveKnight (6,2)
[(8,1),(8,3),(4,1),(4,3),(7,4),(5,4)]
ghci> moveKnight (8,1)
[(6,2),(7,3)]
```

接下来就是 `in3` 函数，接受一个起始位置，每一步都是一个非确定性的位置，然后用 `>>=` 喂给 `moveKnight` ：

```
in3 :: KnightPos -> [KnightPos]
in3 start = do
  first <- moveKnight start
  second <- moveKnight first
  moveKnight second
```

也可以不使用 `do` ：

```
in3 start = return start >>= moveKnight >>= moveKnight >>= moveKnight
```

接下来就是接受两个位置并测试是否可以三步内移动到指定位置的函数：

```
canReachIn3 :: KnightPos -> KnightPos -> Bool
canReachIn3 start end = end `elem` in3 start
```

测试：

```
ghci> (6,2) `canReachIn3` (6,1)
True
ghci> (6,2) `canReachIn3` (7,3)
False
```

## 单子定律

与高级函子以及函子一样，单子也有一些定律是所有实例都必须遵守的。这也是因为某些实现 `Monad` typeclass 的实例并不能称其为单子那样。对于一个类型想要成为真正的单子，单子定律就必须被满足。

### 左同一性 left identity

第一个单子定律就是如果接受一个值，将其置入默认 context 并 `return`，接着通过 `>>=` 将其喂给一个函数，这个过程与接受一个值并喂给函数的结果是一致的。正式的表示：

1. `return x >>= f` 与 `f x` 是一致的

对于 `Maybe` 单子 `return` 被定义为 `Just`。`Maybe` 单子描述的是失败的可能性，如果右普通值放入 context，那么把这个动作视为计算成功是很合理的。例如：

```
ghci> return 3 >>= (\x -> Just (x+100000))
Just 100003
ghci> (\x -> Just (x+100000)) 3
Just 100003
```

对于列表单子而言，`return` 是把值放入一个列表中，变为单例列表。`>>=` 则是走过列表中所有元素，并将它们喂给函数做计算，而又因为单例列表中只有一个值，因此这与直接对元素做运算是等价的：

```
ghci> return "WoM" >>= (\x -> [x,x,x])
["WoM","WoM","WoM"]
ghci> (\x -> [x,x,x]) "WoM"
["WoM","WoM","WoM"]
```

对于 `IO` 而言，我们已经知道了 `return` 不会产生副作用，而是在结果中展示原有的值，因此该定律对 `IO` 也是有效的。

### 右同一性 left identity

第二个单子定律就是如果有一个 monadic 值，将它通过 `>>=` 喂给 `return`，结果还是原有的 monadic 值。正式的表示：

1. `m >>= return` 与 `m` 是一致的

这一条定律可能没有像第一条那样明显，让我们看看它为何成立。通过 `>>=` 将 monadic 值喂给的这些函数都是接受普通值并返回 monadic 值的。`return` 将一个值置入最小 context 且仍然将该值作为其结果。这就意味着，例如 `Maybe` 它并没有处于失败的状态，而对于列表也没有变成非确定性的。

```
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!
```

仔细看一下列表对 `>>=` 实现：

```
xs >>= f = concat (map f xs)
```

将 `[1,2,3,4]` 喂给 `return` ，那么 `return` 将 `[1,2,3,4]` 映射成 `[[1],[2],[3],[4]]` 然后再将这些小的列表结合起来成为原有的列表。

左右同一性描述的是 `return` 的行为，它之所以重要是因为它将普通值转为具有 context 的值。

### 结合律

最后一条单子定律就是当我们通过 `>>=` 串联起所有的 monadic 函数，其嵌套顺序是无关紧要的。正式的表示：

1. `(m >>= f) >>= g` 等同于 `m >>= (\x -> f x >>= g)`

这里有一个 monadic 值 `m` 以及两个 monadic 函数 `f` 与 `g` 。处理 `(m >>= f) >>= g` 时，是将 `m` 喂给 `f` ，得到一个 monadic 值，再将其喂给 `g` ；表达式 `m >>= (\x -> f x >>= g)` 则接受一个 monadic 值，将其喂给一个函数，该函数会把 `f x` 的结果喂给 `g` 。我们不太容易看出两者相同，用一个例子便于理解。

之前的走钢丝的例子中，我们曾把几个函数串在一起：

```
ghci> return (0,0) >>= landRight 2 >>= landLeft 2 >>= landRight 2
Just (2,4)
```

从 `Just (0,0)` 开始，将值传给 `landRight 2` ，结果在传给下一个 monadic 函数，以此类推。用括号表示出优先级的话如下：

```
ghci> ((return (0,0) >>= landRight 2) >>= landLeft 2) >>= landRight 2
Just (2,4)
```

可以改写成这样：

```
return (0,0) >>= (\x ->
landRight 2 x >>= (\y ->
landLeft 2 y >>= (\z ->
landRight 2 z)))
```

`return (0,0)` 等价于 `Just (0,0)` ，喂给 lambda 时，里面的 `x` 就等于 `(0,0)` 。`landRight` 接受一个数值与 `pole` ，得 `Just (0,2)` ，再喂给另一个 lambda，以此类推，最后得到 `Just (2,4)` 。

也就是说将值喂给 monadic 函数的顺序并不重要，重要的是它的意义。这里是另一个视角来观察这个定律：考虑组合两个函数 `f` 与 `g` ，组合的实现如下：

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

`g` 的类型是 `a -> b` 而 `f` 的类型是 `b -> c`，我们将其变为了类型 `a -> c`，因此它的参数都是在这些函数中传递。那么如果这两个函数是 monadic 的，它们返回的是 monadic 值呢？如果一个函数的类型是 `a -> m b`，我们没法将它的返回直接传给 `b -> m c` 的函数，因为该函数接受的是一个普通的 `b`，并不是一个 monadic 值。不过我们可以通过 `>>=` 来实现：

```
(<=<) :: (Monad m) => (b -> m c) -> (a -> m b) -> (a -> m c)
f <=< g = (\x -> g x >>= f)
```

这样就可以组合两个 monadic 函数了：

```
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]
```

那么这与结合律有什么关系呢？当我们将该定律视为组合，就意味着 `f <=< (g <=< h)` 与 `(f <=< g) <=< h` 应该是等价的。这就是另一种方式说明对单子而言，嵌套顺序对计算并无影响。

如果将头两个定律转换为 `<=<`，那么左同一性定律意味着对于所有 monadic 函数 `f` 而言，`f <=< return` 等同于 `f`，那么右同一性定律即 `return <=< f` 等同于 `f`。

这与普通函数 `f` 很像，即 `(f . g) . h` 等同于 `f . (g . h)`，`f . id` 等同于 `f`，以及 `id . f` 等同于 `f`。

# 13 For a Few Monads More

## Writer 单子

我们已经见识过了 `Maybe` ，列表以及 `IO` 单子，现在让我们看看 `Writer` 单子。

相较于 `Maybe` 用作于将值置入可能失败的 context 中，以及列表用作于非确定性的 context，`Writer` 单子则是用作于将值置入一个附加值的 context 中，类似于日志。`Writer` 可以让我们在计算的同时收集所有日志记录，并将它们合并成一个日志并附加在结果上。

例如我们想要将我们的值附加在字符串上用于解释事情的经过，亦或许是调试的目的。考虑一个函数接受一个值并返回一个布尔值：

```
isBigGang :: Int -> Bool
isBigGang x = x > 9
```

如果想要不止是 `True` 或 `False` 值，还想要一段解释应该怎么办呢？

```
isBigGang :: Int -> (Bool, String)
isBigGang x = (x > 9, "Compared gang size to 9.")
```

相较于返回一个 `Bool` ，我们返回的是一个元组，其中第二个值为解释信息。

```
ghci> isBigGang 3
(False,"Compared gang size to 9.")
ghci> isBigGang 30
(True,"Compared gang size to 9.")
```

那么如果我们有一个函数接受的是普通值，同时返回的是代用 context 的值，那么我们应该如何接受一个带有 context 的值，并将其喂给这个普通函数呢？

当我们在探索 `Maybe` 单子时，我们曾编写了 `applyMaybe` 函数，即接受一个 `Maybe a` 值，以及一个类型为 `a -> Maybe b` 的函数，并将 `Maybe a` 值喂给该函数，即使该函数接受的是一个普通的 `a` 而不是一个 `Maybe a` 。`applyMaybe` 考虑到了 context 的处理，也就是会注意可能失败的场景，而在 `a -> Maybe b` 中，只需处理普通的数即可。因为 `applyMaybe`（之后变成了 `>>=` ）会帮忙检查 `Nothing` 或 `Just` 的情况。

以同样的方式，再编写一个接受值以及附加日志的函数，也就是一个 `(a, String)` 值，以及一个 `a -> (b,String)` 类型的函数，最后将值喂给函数。该函数有的 context 是附加日志值，而不是一个可能失败的 context，因此原有的日志会被保留，并附上从函数产生的新的日志：

```
applyLog :: (a,String) -> (a -> (b,String)) -> (b,String)
applyLog (x,log) f = let (y,newLog) = f x in (y,log ++ newLog)
```

当我们拥有一个带有 context 的值并想将其喂给一个函数，我们通常会试着将值从 context 中剥离，然后再将其应用至函数并检查该 context 注重的是什么。在 `Maybe` 单子中，我们检查是否为一个 `Just x` ，如果是则将函数应用至 `x` 。而在日志的情况，我们知道元组的其中一部分是值另一部分是日志，因此我们先取出值 `x` ，将 `f` 应用至 `x` ，得到 `(y,newLog)` ，

其中 `y` 是新的值而 `newLog` 则是新的日志。不过如果返回的是 `newLog` ，那么并没有包含旧的日志，因此需要返回的是 `(y, log ++ newLog)` ：

```
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")
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")
```

看一下 lambda 里是什么情况，`x` 是一个普通字符串而不是一个元组，`applyLog` 用于追加日志。

### 幺半群来拯救我们

现在的 `applyLog` 从 `(a, String)` 类型中获取值，但是日志就必须是一个 `String` 吗？它使用 `++` 来追加日志，那么这不应该适用于所有列表而不单单只是字符列表么？当然是，现在改变一下类型：

```
applyLog :: (a, [c]) -> (a -> (b, [c])) -> (b, [c])
```

那么这对 bytestring 有效吗？当然，不过现在生效的只能是列表。看起来我们需要构建为 bytestring 另一个 `applyLog` 。不过等等！列表和 bytestring 都是幺半群。同样的，它们都是 `Monoid` typeclass 的实例，这就意味着它们都实现了 `mappend` 函数。那么无论是对于列表还是 bytestring 而言，`mappend` 正是用作于追加值。

```
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)
```

棒！那么现在我们的 `applyLog` 就能为任意幺半群工作了。修改一下实现，将 `++` 替换为 `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)
```

由于包含值现在可以是任意幺半群，我们不再需要把一个元组想成一个值以及一个日志，而是一个值与一个幺半群的值。例如我们可以有一个元组包含了一个物品名称以及其作为幺半群的价格。我们只需要使用 `Sum` newtype 来确保价格可以被求和。

```
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)
```

我们用字符串来代表事务，以及 `Sum` 的 `Int` 作为 `newtype` 用于追踪总花销。提醒一下，对 `Sum` 进行 `mappend` 可以将它们加总在一起：

```
ghci> Sum 3 `mappend` Sum 9
Sum {getSum = 12}
```

通过 `applyLog` 将价格进行求和：

```
ghci> ("beans", Sum 10) `applyLog` addDrink
("milk",Sum {getSum = 35})
ghci> ("jerky", Sum 25) `applyLog` addDrink
("whiskey",Sum {getSum = 124})
ghci> ("meat", Sum 5) `applyLog` addDrink
("beer",Sum {getSum = 35})
```

由于 `addDrink` 返回的是类型为 `(Food,Price)` 的元组，那么可以继续应用 `addDrink` 在返回值上：

```
ghci> ("meat", Sum 5) `applyLog` addDrink `applyLog` addDrink
("beer",Sum {getSum = 65})
```

### Writer 类型

我们已经看到了一个值与一个幺半群可以像一个幺半群值那样运作，再测试一下 `Monad` 实例。`Control.Monad.Writer` 模块中提供了 `Writer w a` 类型，连同其 `Monad` 的实例，以及一些处理这个类型的函数。

首先是这个类型本身：

```
newtype Writer w a = Writer { runWriter :: (a, w) }
```

通过 `newtype` 包裹使其成为 `Monad` 的一个实例，同时其类型又有别于普通的元组。其中 `a` 类型参数代表着值，而 `w` 类型参数代表附加的幺半群值。

其 `Monad` 实例定义如下：

```
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')
```

首先解释一下 `>>=` ，它的实现基本上等同于 `applyLog` ，只不过现在的元组是包裹在 `Writer` 的 `newtype` 中，我们需要模式匹配将其解包。首先将函数 `f` 应用在 `x` 上，得到

一个 `Writer w a` 值，接着用一个 `let` 表达式进行模式匹配。再将 `y` 作为新结果，并使用 `mappend` 将旧的幺半群值与新值结合。最后的返回值则用 `Writer` 构造函数打包起来的元组。

那 `return` 呢？回忆一下 `return` 的作用是接受一个值，并返回一个最小默认的 context 来包装我们的值。那么究竟是什么样的 context 能代表 `Writer` 呢？如果我们希望幺半群值所造成的影响越小越好，那么 `mempty` 是个合理的选择，其被当做 identity 幺半群值，例如 `""` ，`Sum 0` ，或是空的 bytestring。当我们对 `mempty` 使用 `mappend` 与其它幺半群结合，那么结果便是其它的幺半群值。因此当用 `return` 来做一个 `Writer` ，再用 `>>=` 喂给其它函数，那函数返回的便是计算后的幺半群。

```
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})
```

由于 `Writer` 没有定义 `Show` 的实例，那么就必须要 `runWriter` 来讲 `Writer` 转成正常的元组。

这里的 `Writer` 实例并未定义 `fail` ，因此模式匹配失败时便会调用 `error` 。

### 使用 do 表示法与 Writer

我们现在有了一个 `Monad` 实例，那么就可以用 `do` 表示法对 `Writer` 值进行串联。下面例中所有的幺半群值都会用 `mappend` 连接起来并得到最后结果：

```
import Control.Monad.Trans.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)
```

与原文不同在于 `import Control.Monad.Trans.Writer` 。`logNumber` 接受一个数值将其变为 `Writer` 值。对于幺半群而言，我们使用字符串列表将数值附加为字符串后置入这个单例列表中。`multWithLog` 则是一个 `Writer` 值，将 `3` 与 `5` 相乘并将它们附带的日志包含进最后的日志中。这里使用了 `return` 来展示作为返回的 `a*b` 。又因为 `return` 仅仅是将某物置入一个最小 context 中，因此我们可以确保它不会添加任何东西进日志中。

```
ghci> runWriter multWithLog
(15,["Got number: 3","Got number: 5"])
```

有时我们仅仅只是想将一些幺半群值在某些特定位置被包含。为此，`tell` 函数就很有用了。它是 `MonadWriter` typeclass 中的一部分，它当做 `Writer` 使用时也能接受一个幺半群值，例如 `["This is going on"]` 。我们能用它把幺半群值接到任何一个 dummy 值 `()` 上来构成一个新的 `Writer` 。当拿到的结果是 `()` 时，不会将其绑定至一个变量上。例如：

```
multWithLog :: Writer [String] Int
multWithLog = do
  a <- logNumber 3
  b <- logNumber 5
  tell ["Gonna multiply these two"]
  return (a * b)
```

可得：

```
ghci> runWriter multWithLog
(15,["Got number: 3","Got number: 5","Gonna multiply these two"])
```

### 添加日志至程序中

欧几里德算法是计算两个数值的最大公约数，即可以被两数相除的最大值。Haskell 有一个 `gcd` 函数，不过让我们实现一个自己的版本并提供日志功能。下面是普通的算法：

```
gcd' :: Int -> Int -> Int
gcd' a b
  | b == 0 = a
  | otherwise = gcd' b (a `mod` b)
```

首先检查第二个值是否为 0，如果是则返回第一个值；如果不是，那么第一个数除以第二个数的余数，再次与第二个数进行计算，以此类推。测试：

```
ghci> gcd' 8 3
1
```

很好，那么与之前一样使用字符串列表作为我们的幺半群，用于存储日志。那么新版的 `gcd'` 函数的类型则会是：

```
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)
```

新函数接受两个普通的 `Int` 值并返回一个 `Writer [String] Int` ，即一个包含了长日志 context 的 `Int` 。除开使用 `do` 表达式，我们也可以这样写：

```
Writer (a, ["Finished with " ++ show a])
```

不过 `do` 表达式更方便阅读。接下来测试一下 `gcd'`，其结果是 `Writer [String] Int`，如果从 `newtype` 提取出来则会是一个元组，而元组的第一部分则是结果：

```
ghci> fst $ runWriter (gcd' 8 3)
1
```

至于日志，由于日志是一连串的字符串，那么就可以使用 `mapM_ putStrLn` 将它们打印出来：

```
ghci> mapM_ putStrLn $ snd $ runWriter (gcd' 8 3)
8 mod 3 = 2
3 mod 2 = 1
2 mod 1 = 0
Finished with 1
```

把普通的算法转成拥有日志是很棒的经验，仅仅是把普通的值重写成 monadic 值，剩下的就是靠 `>>=` 与 `Writer` 来帮忙处理。用这种方法我们几乎可以对任何函数加上日志的功能，只需要把普通的值转换成 `Writer`，然后把普通函数调用换成 `>>=` 即可（当然也可以使用 `do`）。

### 效率低的列表构造函数

在使用 `Writer` 单子的时候，你必须注意使用的幺半群，因为列表有时会变得非常的慢。这是因为列表的 `mappend` 使用的是 `++`，而 `++` 在很长的列表尾部添加元素是非常慢的。

在我们的 `gcd'` 函数中，日志很快是因为列表的追加是这样的：

```
a ++ (b ++ (c ++ (d ++ (e ++ f))))
```

列表是一种从左至右而构建的数据结构，它的高效是因为我们首先构建的是左边的部分，而把另一串列表加到右边的时候性能也不错。但是如果不小心像是下面这样操作列表：

```
((((a ++ b) ++ c) ++ d) ++ e) ++ f
```

这变成了左结合，而非右结合，这样效率非常低，因为每次都把右边的部分加到左边的部分，而左边的部分又必须要从头开始构建。

以下函数类似于 `gcd'`，只是日志的顺序是相反的，它先记录剩下的操作然后再记录当前的操作：

```
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
```

它先递归调用，然后把结果绑定到 `result` ，然后把当前操作记录到日志，在递归的结果之后，最后展示的是完整的日志：

```
ghci> mapM_ putStrLn $ snd $ runWriter (gcdReverse 8 3)
Finished with 1
2 mod 1 = 0
3 mod 2 = 1
8 mod 3 = 2
```

这样没有效率是因为它让 `++` 成了左结合而不是右结合。

### 差异列表

由于列表在重复追加的情况下效率低下，那么最好要用一种高效追加的数据结构。其中一种就是差异列表 difference lists，与普通列表类似，而它接受一个列表并前置追加另一串列表。一个等价于 `[1,2,3]` 的差异列表是这样一个函数 `\xs -> [1,2,3] ++ xs` 。一个等价于 `[]` 的差异列表则是 `\xs -> [] ++ xs` 。

差异列表最酷的地方在于它支持高效的尾部追加。普通列表下，当我们通过 `++` 向列表追加另一个列表时，它必须要走到左边列表尾部，然后把右边列表中的元素一个一个的接上。那么差异列表是怎么实现的呢？两个差异列表的追加像是这样：

```
f `append` g = \xs -> f (g xs)
```

`f` 与 `g` 函数接受列表并前置追加。如果 `f` 函数是 `("dog"++)` （另一种 `\x -> "dog" ++ xs` 的写法）以及 `g` 函数是 `("meat"++)` ，那么 `f `append` g` 构造的函数就等同于：

```
\xs -> "dog" ++ ("meat" ++ xs)
```

我们通过构建一个新函数先将一个列表喂给第一个差异列表，然后再把结果喂给第二个差异列表，这样的方式将两个差异列表进行了追加操作。

让我们创建一个 `newtype` 包装差异列表，这样我们可以让它实现幺半群实例：

```
newtype DiffList a = DiffList { getDiffList :: [a] -> [a] }
```

包装中的类型是 `[a] -> [a]` ，因为差异列表实际上就是个接受一个列表并返回另一个列表的函数。将普通列表转换成差异列表很简单，反之亦然：

```
toDiffList :: [a] -> DiffList a
toDiffList xs = DiffList (xs ++)

fromDiffList :: DiffList a -> [a]
fromDiffList (DiffList f) = f []
```

把一个普通列表转换成差异列表就是按照之前定义那样，实现一个前置追加至另一个列表的函数。由于差异列表只是一个前置追加的函数，那么将差异列表转回普通列表只需要喂给它空列表即可。

以下是差异列表的 `Monoid` 定义：

```
instance Semigroup (DiffList a) where
  DiffList f <> DiffList g = DiffList (f . g)

instance Monoid (DiffList a) where
  mempty = DiffList ([] ++)
```

与原文不同在于现在需要先定义 `Semigroup` 的实例（`<>` 函数的本质与 `Monoid` 实例中的 `mappend` 一致），这样在 `Monoid` 实例中便无需再次定义 `mappend` 了。对于列表而言 `mempty` 就是 `id` 函数，而 `mappend` 则是函数组合。测试

```
ghci> fromDiffList (toDiffList [1,2,3,4] `mappend` toDiffList [1,2,3])
[1,2,3,4,1,2,3]
```

顶级！现在可以提高我们 `gcdReverse` 函数的效率了：

```
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
```

我们只需把幺半群的类型从 `[String]` 改为 `DiffList String` ，并且在 `tell` 时把普通列表用 `toDiffList` 转成差异列表即可。

```
ghci> mapM_ putStrLn . fromDiffList . snd . runWriter $ gcd' 110 34
Finished with 2
8 mod 2 = 0
34 mod 8 = 2
110 mod 34 = 8
```

用 `runWriter` 取出 `gcdReverse 110 34` 结果，用 `snd` 取出日志，并用 `fromDiffList` 转成普通列表再打印出来。

### 性能比较

为了感受差异列表是如何提 cdReverse 高效率的，考虑一个从某数数到零的案例。我们像 `gcdReverse` 那样反过来进行记录，这样日志实际上是从零到某个数。

```
finalCountDown :: Int -> Writer (DiffList String) ()
finalCountDown 0 = do
  tell (toDiffList ["0"])
finalCountDown x = do
  finalCountDown (x - 1)
  tell (toDiffList [show x])
```

喂给它 `0` 时只记录 0；喂给它其他正整数，则会先倒数到 `0` 然后追加那些数至日志中。如果调用 `finalCountDown` 并喂给它 `100` ，那么日志最后的记录就会是 `"100"` 。

如果把这个函数加载至 GHCI 中并喂给它一个比较大的整数 `500000` ，那么则会看到它无停滞的从 `0` 开始数起：

```
ghci> mapM_ putStrLn . fromDiffList . snd . runWriter $ finalCountDown 500000
0
1
2
```

如果我们用的是普通列表而不是差异列表：

```
finalCountDown :: Int -> Writer [String] ()
finalCountDown 0 = do
    tell ["0"]
finalCountDown x = do
    finalCountDown (x-1)
    tell [show x]
```

并给出同样指令：

```
ghci> mapM_ putStrLn . snd . runWriter $ finalCountDown 500000
```

我们会看到整个运算异常的卡顿。

当然这并不是一个严谨的测试方法，不过这足够表明差异化列表是更有效率的写法。

## Reader 单子

在高级函子的章节中我们学习到了函数类型 `(->) r` 是 `Functor` 的实例。将函数 `f` 应用在函数 `g` 将会得到与参数输入 `g` 后的结果再输入 `f` 的一致的结果。因此本质上我们创造了一个类似 `g` 的函数，只不过在返回结果之前又用 `f` 应用在了该结果上。例如：

```
ghci> let f = (*5)
ghci> let g = (+3)
ghci> (fmap f g) 8
55
```

我们同样也见到了函数时高级函子。它们允许我们对函数的结果直接进行操作：

```
ghci> let f = (+) <$> (*2) <*> (+10)
ghci> f 3
19
```

表达式 `(+) <$> (*2) <*> (+10)` 创造了个函数，它接受一个值，并将该值分别进行 `*2` 与 `+10` 最后将两者加总起来。

函数类型 `(->) r` 不仅仅是一个函子，一个高级函子，同时还是一个单子。正如我们遇到过的其他单子值那样，函数同样可以被视为一个带有 context 的值。函数的 context 就是期待的某个还未出现的值，但我们知道会把它当做函数的参数，调用函数来得到结果。

函数的 `Monad` 实例位于 `Control.Monad.Instances` ，类似于：

```
instance Monad ((->) r) where
  return x = \_ -> x
  h >>= f = \w -> f (h w) w
```

`return` 即 `pure` 不多赘述。

`>>=` 看起来有点神秘。当使用 `>>=` 将一个单子值喂给一个函数时，结果总是一个单子值，那么在这里当我们将一个函数喂给另一个函数，返回的仍然是一个函数。这就是为什么返回的是一个 lambda。迄今为止，所有的 `>>=` 实现总是以某种方式将内部与外部隔离开，然后在内部使用 `f` 。这里也一样，要从一个函数得到一个结果，我们必须喂给它一些东西，这就是为什么用 `(h w)` 得到结果后再对其应用 `f` 。`f` 返回一个单子值，这里是一个函数，再将该函数应用至 `w` 上。

下面是使用该单子的一个 `do` 表达式：

```
addStuff :: Int -> Int
addStuff = do
  a <- (* 2)
  b <- (+ 10)
  return (a + b)
```

这与早前我们写的高级单子表达式是同一个东西，只不过现在依赖的是函数的单子特性。一个 `do` 表达式总是返回一个单子值，这里也不例外，这里的单子值是一个函数。测试：

```
ghci> addStuff 3
19
```

`(*2)` 与 `(+10)` 都应用在了 `3` 上。`return (a+b)` 亦是如此，只不过它会忽略 `3` 并永远返回 `a+b` 。正因如此，函数的单子性童谣被称为 reader 单子。所有的函数读取一个共同的源。为了更好的演示，我们来重写一下 `addStuff` ：

```
addStuff :: Int -> Int
addStuff x =
  let a = (* 2) x
      b = (+ 10) x
   in a + b
```

我们看到 reader 单子允许我们将函数视为带 context 的值。我们可以装作已经知道函数将要返回什么。它通过粘合若干函数成为一个函数的方式，并把这个函数的参数喂给所有被粘

合函数。所以如果我们有很多函数都缺一个参数且它们最终将应用至同一值时，我们可以使用 reader 单子来提取它们未来的返回，并通过 `>>=` 来确保它们都能执行。

## 美味的状态性计算

Haskell 是一个由函数构成的纯语言，函数不能修改任何全局状态或变量，仅能计算并返回值。然而有些问题依赖于时间变化的状态。而这些问题在 Haskell 中并不是问题，只不过模型的实现可能会很痛苦。这就是为什么 Haskell 提供了状态单子，处理起状态性的问题简单而不失纯性。

在处理随机数的问题中，我们使用了接受随机生成器作为参数的函数，并返回一个随机值与一个新的随机生成器。如果想要生成若干随机数，我们必须使用前一个函数所返回的随机生成器。当创建一个接受 `StdGen` 的函数，根据生成器抛三次硬币，我们需要这么做：

```
threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen =
  let (firstCoin, newGen) = random gen
      (secondCoin, newGen') = random newGen
      (thirdCoin, newGen'') = random newGen'
   in (firstCoin, secondCoin, thirdCoin)
```

它接受生成器 `gen` 接着 `random gen` 生成一个 `Bool` 值以及一个新的生成器。

你可能认为为了避免手动处理状态性的计算，我们会放弃 Haskell 的纯性。然而实际上并不需要，因为存在一个特殊的状态单子用于处理所有状态而不失 Haskell 的纯性。

那么为了更好的理解状态性计算，让我们给其一个类型。我们说状态性计算是一个函数，其接受某些状态并返回某些新状态。该函数类型如下：

```
s -> (a, s)
```

`s` 是状态的类型，而 `a` 是状态计算的结果。

这个状态性计算，即一个函数接受一个状态并返回一个结果与一个新的状态，可以被视为一个带有 context 的值。实际值是结果，而 context 则是我们提供的某些初始状态用于获取结果，同时得到结果时获得一个新的状态。

### 堆

假设我们希望对堆进行建模。我们可以在堆的顶部放置新项，或者移除已有项。

这里用列表来代表这个堆，列表的头部则是堆的顶部。为此我们需要两个函数：`pop` 与 `push` 。`pop` 接受一个堆，移除一项后，返回移除的这个项以及一个新的没有此项的堆。`push` 接受一个项以及一个堆，将该项置入堆头部，返回 `()` 作为结果，以及一个新的堆：

```
type Stack = [Int]

```

```
pop :: Stack -> (Int, Stack)
pop (x : xs) = (x, xs)

push :: Int -> Stack -> ((), Stack)
push a xs = ((), a : xs)
```

注意是如何应用 `push` 的第一个参数的，我们得到一个状态性计算。`pop` 由于其类型已经是一个状态性计算了。

现在用这些函数测试：

```
stackManip :: Stack -> (Int, Stack)
stackManip stack =
  let ((), newStack1) = push 3 stack
      (a, newStack2) = pop newStack1
   in pop newStack2
```

测试：

```
ghci> stackManip [5,8,2,1]
(5,[8,2,1])
```

注意 `stackManip` 它自身如何是一个状态性计算的。我们接受了一些状态性的计算，并将它们粘合在一起。Hmm，听起来很熟悉。

上述代码看起来不轻松，因为我们手动的给每个状态性计算提供了状态，存储它们并将它们置入下一个计算中。如果无需为每个函数手动提供状态，而像是下面这样，会不会很酷：

```
stackManip = do
  push 3
  a <- pop
  pop
```

使用状态单子将允许我们做这样的动作。通过状态单子我们具备了无需管理状态的状态性计算。

### 状态单子

`Control.Monad.State` 模块提供了一个 `newtype` ，其包装了状态性计算。以下是其定义：

```
newtype State s a = State { runState :: s -> (a,s) }
```

一个 `State s a` 是一个状态性计算，操作类型 `s` 的状态，并返回带有 `a` 类型的结果。

现在我们见识到了状态性计算，以及它们最终是如何被视为带有 context 的值，那么让我们看看它的 `Monad` 实例：

```
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
```

让我们先来看看 `return` ，它的目的是接受一个值进行一个状态性计算，做出一个改变状态的操作并总是返回该值。因此需要一个完整的 lambda 函数 `\s -> (x,s)` 。我们总是展示 `x` 作为状态性计算的结果，且不改变状态，因为 `return` 总是将值置入最小 context 中。因此 `return` 将做出一个状态性计算并展示确定值，并保持状态不变。

那么 `>>=` 呢？状态性计算喂给 `>>=` 函数后也必须要返回另一个改变状态的计算，对吗？首先来看一下 `State` 这个 `newtype` 包装，内部是一个 lambda。该 lambda 则是我们新的状态性计算。那么里面发生了什么？我们以某种方式从第一个状态性计算中提取了值。因为我们正处于一个状态性计算中，我们可以给状态性计算 `h` 当前的状态 `s` ，其返回结果与新的状态：`(a, newState)` 。迄今为止我们每次实现 `>>=` 时，一旦从单子值中提取出了结果，我们会将函数 `f` 应用在该结果上来得到一个新的单子值。在 `Writer` 中，在得到新的单子值后，我们仍然需要确保 context 通过 `mappend` 连接了旧的单子值与新的单子值。这里我们使用 `f a` 并得到一个新的状态性计算 `g` 。现在我们有了一个新的状态性计算以及一个新的状态（即 `newState` ），我们把 `newState` 喂给 `g` ，得到一个元组，其包含了最后的结果与最终的状态。

有了 `>>=` 就类似于将两个状态性计算粘合在了一起，只是第二个隐藏在一个函数中用于接受第一个的结果。由于 `pop` 与 `push` 已经是改变状态的计算了，因此可以把它们包进 `State` 中：

```
pop :: State Stack Int
pop = state $ \(x : xs) -> (x, xs)

push :: Int -> State Stack ()
push a = state $ \xs -> ((), a : xs)
```

现在重写之前的 `stackManip` ：

```
stackManip :: State Stack Int
stackManip = do
  push 3
  a <- pop
  pop
```

看到我们是如何将一个 push 与两个 pops 粘合至一个状态性计算的了么？从 `newtype` 解包后就得到了一个函数这样就可以供我们去提供一些初始化状态：

```
ghci> runState stackManip [5,8,2,1]
(5,[8,2,1])
```

实际上，我们并不需要将 `pop` 绑定至 `a` ，因为我们完全没有用上 `a` ，因此可以这样：

```
stackManip :: State Stack Int
```

```
stackManip = do
    push 3
    pop
    pop
```

而利用绑定值：

```
stackStuff :: State Stack ()
stackStuff = do
    a <- pop
    if a == 5
        then push 5
        else do
            push 3
            push 8
```

很直接，测试：

```
ghci> runState stackStuff [9,0,2,1,0]
((),[8,3,0,2,1,0])
```

记住，`do` 表达式返回的是一个单子值，而在 `State` 单子的情况下，`do` 也就是一个改变状态的函数。而由于 `stackManip` 与 `stackStuff` 都是改变状态的计算，因此才可以把它们粘合在一起：

```
import Control.Monad

moreStack :: State Stack ()
moreStack = do
  a <- stackManip
  when (a == 100) stackStuff
```

与原文不同之处在于使用了 `Control.Monad.when` 函数。

`Control.Monad.State` 模块提供了一个称为 `MonadState` 的 typeclass，它有两个非常有用的功能，名为 `get` 以及 `put` 。对于 `State` 而言，`get` 函数的实现像是这样：

```
get = State $ \s -> (s,s)
```

即取出当前状态并什么都不做。而 `put` 则会接受一个状态并取代现有的状态：

```
put newState = State $ \s -> ((),newState)
```

有了这两个状态就可以看到现在堆叠中有什么，或者把整个堆叠中的元素替换掉：

```
stackyStack :: State Stack ()
stackyStack = do
  stackNow <- get
  if stackNow == [1, 2, 3]
    then put [8, 3, 1]
    else put [9, 2, 1]
```

以下是作用于 `State` 的 `>>=` 类型：

```
(>>=) :: State s a -> (a -> State s b) -> State s b
```

状态 `s` 的类型是保持不变的，而结果的类型可以从 `a` 变为 `b` 。这就意味着可以将若干状态性计算粘合在一起，它们的结果可以是不同类型而状态的类型却维持不变。那么这是为什么呢？例如 `Maybe` 的 `>>=` 类型为：

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

单子本身 `Maybe` 是不变的。在两个不同的单子间使用 `>>=` 是无意义的。而对于状态单子而言，实际上是 `State s` ，如果 `s` 不一致，那么就变成了两个不同的单子间使用 `>>=` 。

### 随机性与状态单子

本节的开头我们看到了生成随机数有时很笨拙，因为每个随机函数都接受一个生成器并返回一个带着新生成器的随机数，也就是说如果想要生成另一个随机数时就不能用旧的生成器（不然生成的随机数是一样的）。有了状态单子处理起来就简单多了。

`System.Random` 模块中的 `random` 函数拥有以下类型：

```
random :: (RandomGen g, Random a) => g -> (a, g)
```

意为接受一个随机生成器并生产一个带着新的生成器的随机数。可以看到这就是一个状态性计算，因此可以将其包装进 `State` 的 `newtype` 中，接着作为单子值来使用它：

```
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)
```

现在的 `threeCoins` 是一个状态性计算，在接受一个初始化的随机数生成器后，传递给第一个 `randomSt` ，生产出一个至以及一个新的生成器，新的生成器传递下去，以此类推。使用 `return (a,b,c)` 来展示 `(a,b,c)` 同时不改变最近一个生成器的状态：

```
ghci> runState threeCoins (mkStdGen 33)
((True,False,True),680029187 2103410263)
```

### Error 单子

我们了解了 `Maybe` ，对于 `Either e a` 而言，在 `Control.Monad.Error` 模块中有它的 `Monad` 实例：

```
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` 无需赘述，使用 `Right` 构造函数作为最小默认 context，因为 `Right` 代表正确的计算。

`>>=` 检验了两种可能性，即 `Left` 与 `Right` 。`Right` 情况下，函数 `f` 应用在了内部的值，这类似于 `Just` 的情况；而 `Left` 情况下意为错误情况，`Left` 值被保留，即描述错误的信息。

`Either e` 的 `Monad` 实例有一个额外的需求，那就是包含在 `Left` 中的类型，即 `e` ，必须是 `Error` typeclass 的实例。`Error` 这个 typeclass 描述了一个可被当做错误消息的类型，它定义了 `strMsg` 这个函数，其接受一个形态为字符串的错误。一个 `Error` 的实例就是 `String` 类型！

```
ghci> :t strMsg
strMsg :: (Error a) => String -> a
ghci> strMsg "boom!" :: String
"boom!"
```

由于在使用 `Either` 时，我们通常会用 `String` 来描述错误，所以无需过多担心。在 `do` 表示法中模式匹配失败后，一个 `Left` 值用于提示错误。

```
ghci> Left "boom" >>= \x -> return (x+1)
Left "boom"
ghci> Right 100 >>= \x -> Left "no way!"
Left "no way!"
```

当我们使用 `>>=` 将一个 `Left` 值喂给一个函数时，函数被忽略接着返回一个幺元 `Left` 值；但我们将一个 `Right` 值喂给函数时，函数会应用在其内部值，但是这种情况下函数仍然会生产一个 `Left` 值！

```
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 说它不知道我们 `Either e a` 中 `e` 的类型，即使我们提供的是 `Right` 值。这是因为 `Error e` 约束在了 `Monad` 实例。因此我们需要显式的提供类型签名：

```
ghci> Right 3 >>= \x -> return (x + 100) :: Either String Int
Right 103
```

## 一些有用的 monadic 函数

本节中，我们将探索一些函数，它们要么是操作单子值的，要么是返回单子值的（或者两者兼具!）。这些函数通常被认为是 monadic 函数。

### liftM

每个单子都是一个高级函子，每个高级函子都是一个函子。`Applicative` typeclass 拥有一个类约束，即类型必须是 `Functor` 的实例。

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
```

而 `fmap` ：

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

测试：

```
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])
```

新版的 `Monad` typeclass 已经将 `Applicative` 作为类约束了，因此略过本段关于 `liftM` 的描述。

### join 函数

一个思考：如果一个单子值的结果是另一个单子值，也就是单子值嵌套时，如何打平它们成为单个单子值呢？例如 `Just (Just 9)` ，我们能将其变为 `Just 9` 么？事实是任何嵌套的单子值都可以被打平，而这一特性是单子特有的。为此 `join` 函数存在了。它的类型：

```
join :: (Monad m) => m (m a) -> m a
```

它接受一个嵌套的单子值并返回一个单子值，类似于打平了它。测试 `Maybe` ：

```
ghci> join (Just (Just 9))
Just 9
ghci> join (Just Nothing)
Nothing
```

```
ghci> join Nothing
Nothing
```

打平列表非常符合直觉：

```
ghci> join [[1,2,3],[4,5,6]]
[1,2,3,4,5,6]
```

正如所见，对于列表而言 `join` 即 `concat` 。而对于嵌套的 `Writer` ，我们需要 `mappend` 么半群的值：

```
ghci> runWriter $ join (Writer (Writer (1,"aaa"),"bbb"))
(1,"bbbaaa")
```

外层的么半群值 `"bbb"` 先，然后再是追加 `"aaa"` 。直觉而言，如果希望测试 `Writer` 的值，必须先将其么半群值输出至日志，这样才能检查到其中的内容。

打平 `Either` 与 `Maybe` 类似：

```
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"
```

如果应用 `join` 至一个结果也为状态性计算的状态性计算，那么先计算外层再计算内层：

```
ghci> runState (join (State $ \s -> (push 10,1:2:s))) [0,0,0]
((),[10,1,2,0,0,0])
```

这里的 lambda 接受一个状态并将 `2` 与 `1` 置入堆中并将 `push 10` 作为结果。那么当整个被 `join` 打平时，它首先将 `2` 与 `1` 置入堆中，接着计算 `push 10` ，因此是将 `10` 放置堆的顶端。

`join` 的实现：

```
join :: (Monad m) => m (m a) -> m a
join mm = do
    m <- mm
    m
```

因为 `mm` 的结果是一个单子值，单独用 `m <- mm` 来取得其结果。这也说明了 `Maybe` 只有当内外层的值都是 `Just` 时，才会是 `Just` 。如果把 `mm` 设置成 `Just (Just 8)` ，那么它看起来像是这样：

```
joinedMaybes :: Maybe Int
joinedMaybes = do
    m <- Just (Just 8)
    m
```

对于每个单子而言可能最有趣的地方就在于 `join` 了，将一个单子值通过 `>>=` 喂给一个函数等同于映射该函数再使用 `join` 来打平嵌套的单子值！换言之，`m >>= f` 总是等同于 `join (fmap f m)` ！假设我们有 `Just 9` 以及函数 `\x -> Just (x+1)` ，如果映射该函数至 `Just 9` ，将会得到 `Just (Just 10)` 。

如果我们在构建自己的 `Monad` 实例时，`m >>= f` 等同于 `join (fmap f m)` 的这个事实是非常有用的，因为打平一个嵌套单子值总是比思考实现 `>>=` 要来的轻松。

### filterM

`filter` 是 Haskell 中非常重要的一个函数。它接受一个子句以及一个用于筛选的列表，返回一个满足条件的元素所构成的新列表。其类型：

```
filter :: (a -> Bool) -> [a] -> [a]
```

那么如果函数返回的是一个单子值呢？假设每个 `True` 或 `False` 都携带一个幺半群值，如 `["Accepted the number 5"]` 或 `["3 is too small"]` ，我们希望最后的结果也携带着 context。

`Control.Monad` 模块中的 `filterM` 函数正是我们想要的！其类型如下：

```
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
```

子句返回一个单子值，其结果为 `Bool` 类型，但由于是单子值，它的 context 有可能会是任何 context，例如可能会失败，非确定性，等等。一旦我们能保证 context 也会被保存在最后的结果中，其结果也会是一个单子值。

以下是一个接受列表并过滤小于 4 的函数，先使用 `filter` 函数：

```
ghci> filter (\x -> x < 4) [9,1,5,2,10,3]
[1,2,3]
```

很简单，现在让我们做一个子句，除开展示 `True` 或 `False` 结果，同样提供一个日志。当然了，用的还是 `Writer` 单子：

```
keepSmall :: Int -> Writer [String] Bool
keepSmall x
  | x < 4 = do
      tell ["Keeping " ++ show x]
      return True
  | otherwise = do
      tell [show x ++ " is tool large, throwing it away"]
      return False
```

没有直接返回一个 `Bool` ，函数返回的是一个 `Writer [Stirng] Bool` ，即一个 monadic 子句。

现在将其传给 `filterM` ，由于子句返回的是 `Writer` 值，那么作为结果的列表同样也是一个 `Writer` 值。

```
ghci> fst $ runWriter $ filterM keepSmall [9,1,5,2,10,3]
[1,2,3]
```

测试 `Writer` 的结果，可以看到有序的日志：

```
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
```

一个炫酷的技巧使用 `filterM` 来产生一个列表的超集，即包含了所有子集所形成的集合。如果说列表是 `[1,2,3]` ，那么它的超集如下：

```
[1,2,3]
[1,2]
[1,3]
[1]
[2,3]
[2]
[3]
[]
```

让一个函数返回列表的超集，我们需要依赖非确定性。那么筛选列表，并使用一个子句能够同时保留以及丢弃列表中的每个元素。

```
powerset :: [a] -> [[a]]
powerset = filterM (const [True, False])
```

与原文不同之处在于 lambda 被替换成了 `const` 。`const x` 是一个单元函数，用于将所有输入应用至 `x` 上。

```
ghci> powerset [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

这样的写法需要好好的思考一番，但如果能接受列表其实就是非确定性值的话，那么要想通就会容易一些。

### foldM

与 `foldl` 所 monadic 对应的是 `foldM` 。前者接受一个二元函数，一个起始累加器以及一个用于 fold 的列表，接着将列表通过二元函数从左向右折叠成一个单值；后者做的事情一样，只不过它所接受的二元函数生产的是一个单子值，不出意外的返回值也是单子。`foldl` 类型：

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

而 `foldM` 类型为：

```
foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
```

假设我们现在想要对一个列表求和，不过加上一个条件就是如果在列表中遇到任何值是大于 `9` 的，那么就直接失败。以下是一个返回 `Maybe` 累加器的二元函数：

```
binSmalls :: Int -> Int -> Maybe Int
binSmalls acc x
    | x > 9     = Nothing
    | otherwise = Just (acc + x)
```

正因为二元函数现在是一个 monadic 函数，那么便不可使用普通的 `foldl` 函数了，不过我们有 `foldM` ：

```
ghci> foldM binSmalls 0 [2,8,3,1]
Just 14
ghci> foldM binSmalls 0 [2,11,3,1]
Nothing
```

### 构建一个安全的 RPN 计算器

早期的章节中我们实现了一个 RPN 计算器，我们也注意到它在遇到错误输入时并不安全，即直接导致程序崩溃。那么现在让我们用 `Maybe` 单子来重新实现一个安全的版本。

原来的代码：

```
solveRPN :: String -> Double
solveRPN = head . foldl foldingFunction [] . words

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
```

现在将 `foldingFunction` 改为返回 `Maybe` 值：

```
foldingFunction :: [Double] -> String -> Maybe [Double]
```

所以它将返回 `Just` 一个新的堆，或是失败 `Nothing` 。

`reads` 函数类似于 `read` ，前者成功时会返回一个单例列表，失败时返回一个空列表。除开它读到的值，还会返回一个读取失败的字符串。我们假设它总是会消费所有的输入，在此创建一个 `readMaybe` 函数：

```
readMaybe :: (Read a) => String -> Maybe a
readMaybe st = case reads st of
  [(x, "")] -> Just x
  _ -> Nothing
```

测试：

```
ghci> readMaybe "1" :: Maybe Int
Just 1
ghci> readMaybe "GO TO HELL" :: Maybe Int
Nothing
```

看起来没问题，那么现在让我们改造函数成为可失败的 monadic 函数：

```
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)
```

前三个例子与旧代码类似，除了在最后用 `Just` 来包装新的堆（这里使用了 `return` 来实现，当然了也可以使用 `Just` ）。最后一个例子使用了 `readMaybe numberString` 并作用于 `(:xs)` 。例如 `xs` 是 `[1.0,2.0]` 并且 `readMaybe numberString` 返回一个 `Just 3.0` ，那么结果就会是 `Just [3.0,1.0,2.0]` ；如果 `readMaybe numberString` 返回的是 `Nothing` ，那么结果也会是 `Nothing` 。测试：

```
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
```

看起来生效了！那么现在就是新改进的 `solveRPN` 了。

```
solveRPN :: String -> Maybe Double
solveRPN st = do
  [result] <- foldM foldingFunction [] (words st)
  return result
```

跟之前一样，接受字符串并将其转为单词列表。接着使用折叠，空列表起始，不同于之前的一个普通的 `foldl` ，这里使用 `foldM` 。而 `foldM` 的结果就是一个包含列表的 `Maybe` 值（即最终的堆）以及一个单例列表。使用 `do` 表达式获取值，并称其 `result` 。测试：

```
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
```

### 组合 monadic 函数

当我们在学习单子定律时，我们说 `<=<` 函数就像组合那样，只不过它不作用于普通函数如 `a -> b` ，而是作用于 monadic 函数如 `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
```

上述例子中我们先组合了两个普通函数，将组合后的函数应用至 `4` ；接着我们组合了两个 monadic 函数，并通过 `>>=` 将 `Just 4` 喂给它。

如果我们有一个列表的函数，我们可以使用 `id` 作为初始累加器以及 `.` 函数作为二元函数，将它们组合成一个大函数。例如：

```
ghci> let f = foldr (.) id [(+1),(*100),(+1)]
ghci> f 1
201
```

函数 `f` 接受一个数值，加 `1` 后乘以 `100` 再加 `1` 。我们可以以同样的方式来组合 monadic 函数，不同之处在于使用 `<=<` 而不是 `.` 以及 `return` 而不是 `id` 。这里无需使用 `foldM` 因为 `<=<` 函数能确保组合是以 monadic 的方式进行的。

上一章中提到的列表单子，我们用了棋盘中的骑士走三步来举例。`moveKnight` 函数接受骑士在棋盘上的起始位置，并返回下一步所有可能的移动。接着就是生成三步后所有可能的位置。

```
in3 start = return start >>= moveKnight >>= moveKnight >>= moveKnight
```

然后检查从 `start` 至 `end` 是否在三步中：

```
canReachIn3 :: KnightPos -> KnightPos -> Bool
canReachIn3 start end = end `elem` in3 start
```

使用 monadic 函数组合我们可以创建一个 `in3` 函数，但我们希望它不只是三步的版本，而是任意步的版本。如果仔细观察 `in3` ，它只不过用了 `>>=` 以及 `moveKnight` 把之前所有可能结果喂给了下一步。将其泛用化后如：

```
import Data.List

inMany :: Int -> KnightPos -> [KnightPos]
inMany x start = return start >>= foldr (<=<) return (replicate x moveKnight)
```

首先用 `replicate` 构建一个列表，里面有 `x` 份的 `moveKnight` ，接着把所有函数合成后，得到从起点走 `x` 步内所有可能的位置。然后我们只需将初始值喂给它即可。

泛用化 `canReachIn3` ：

```
canReachIn :: Int -> KnightPos -> KnightPos -> Bool
canReachIn x start end = end `elem` inMany x start
```

## 创造自己的单子

本节中我们将学习一个类型是如何构建的，定义成单子的并给与其合适的 `Monad` 实例。我们通常不会为了构建一个单子而去创造一个单子，而经常发生的是在对一个问题建立模型时，发现定义的类型其实是一个单子，这才定义一个 `Monad` 的实例。

如我们见到的，列表用于表示非确定性的值。一个 `[3,5,9]` 列表可被视为单个非确定的值，我们不能知道究竟是哪一个。当我们通过 `>>=` 将列表喂给一个函数时，它就是把一串可能的选择都丢给函数，函数一个个去计算在那种情况下的结果，这个结果也是一个列表。

如果我们把 `[3,5,9]` 看作是 `3` ，`5` ，`9` 各出现一次，我们注意到这并没有每一种数字出现的概率。那么如果把非确定的值视为 `[3,5,9]` 的时候，需要 `3` 出现的概率是 50%，剩下的各 25% 该怎么做呢?

如果说列表中每个元素都带有其出现的概率，那么下面形式非常合理:

```
[(3,0.5),(5,0.25),(9,0.25)]
```

Haskell 提供了名为 `Rational` 的有理数，其位于 `Data.Ratio` 模块中。`Rational` 需要写成一个分数的形式。分子与分母用 `%` 分隔:

```
ghci> 1%4
1 % 4
ghci> 1%2 + 1%2
1 % 1
ghci> 1%3 + 5%4
19 % 12
```

现在通过 `Rational` 来表达概率:

```
ghci> [(3,1%2),(5,1%4),(9,1%4)]
[(3,1 % 2),(5,1 % 4),(9,1 % 4)]
```

接下来是将上面提到的过得带有概率的列表用 `newtype` 包裹:

```
import Data.Ratio

newtype Prob a = Prob {getProb :: [(a, Rational)]} deriving (Show)
```

那么它是函子吗? 列表是函子，那么 `Prob` 也应该是一个函子。

```
import Data.Bifunctor qualified as B

instance Functor Prob where
  fmap f (Prob xs) = Prob $ map (B.first f) xs
```

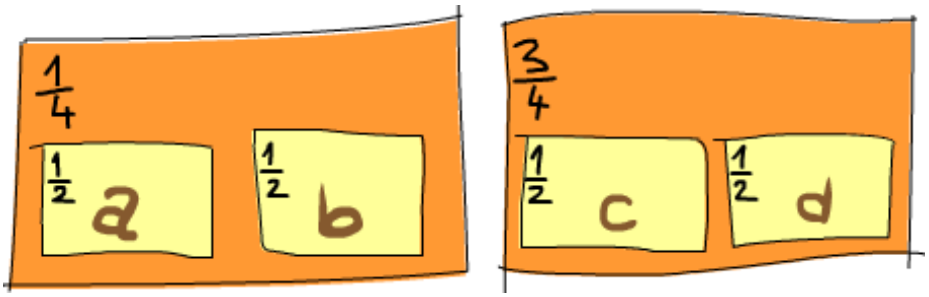
与原文不同之处在于 `import Data.Bifunctor qualified as B` 后使用了 `B.first f` 来取代 lambda 函数 `\(x, p) -> (f x, p)` 。从 `newtype` 中解包，将函数 `f` 应用至值，保留概率不变。测试：

```
ghci> fmap negate (Prob [(3,1%2),(5,1%4),(9,1%4)])
Prob {getProb = [(-3,1 % 2),(-5,1 % 4),(-9,1 % 4)]}
```

还有额外一点需要注意的是概率之和应该永远是 `1` 。

现在有一个大问题，那就是它是否为单子？鉴于列表是一个单子，它也应该是一个单子才对。首先让我们思考一下 `return` ，它是如何作用于列表的？它接受一个值，并将其置入一个单例列表中。那么这里如何做呢？由于它必须是一个默认最小 context，那么也该创造一个单例列表。那么概率该怎么办？ `return x` 理应构建一个单子值，并总是展示 `x` 为其结果，那么概率为 `0` 没有道理。如果总是将其展示为结果，那么概率应该是 `1` ！

那么 `>>=` 该如何？似乎有点棘手，那么可以使用单子的 `m >>= f` 等于 `join (fmap f m)` 这个事实来思考该如何打平一个概率列表。例如，假设一个列表的 `'a'` 与 `'b'` 其中之一出现的概率是 25%，另外一个列表的 `'c'` 与 `'d'` 其中之一出现的概率是 75%。如图所示：




这种情况的概率列表的表达式：

```
thisSituation :: Prob (Prob Char)
thisSituation =
  Prob
    [ (Prob [('a', 1 % 2), ('b', 1 % 2)], 1 % 4),
      (Prob [('c', 1 % 2), ('d', 1 % 2)], 3 % 4)
    ]
```

注意它的类型是 `Prob (Prob Char)` 。那么现在要考虑该如何打平一个嵌套的概率列表了，一旦逻辑成立，`>>=` 也就是 `join (fmap f m)` ，即得到了一个单子。以下是一个 `flatten` 来完成这件事：

```
flatten :: Prob (Prob a) -> Prob a
flatten (Prob xs) = Prob $ concatMap multAll xs
  where
    multAll (Prob innerxs, p) = map (B.second (p *)) innerxs
```

与原文不同之处在于 `concat $ map` 被 `concatMap` 代替，且 `\(x, r) -> (x, p * r)` 被简化成 `(Data.Bifunctor.second (p *))` 。函数 `multAll` 接受一个概率列表，需要将列表里面的概率都乘以 `p` ，返回一个新的概率列表。将 `multAll` 映射到了概率列表时，即成功的打平了列表。

现在来定义 `Monad` 实例：

```
import Data.Bifunctor qualified as B
import Data.Ratio

instance Functor Prob where
fmap f (Prob xs) = Prob $ map (B.first f) xs

flatten :: Prob (Prob a) -> Prob a
flatten (Prob xs) = Prob $ concatMap multAll xs
  where
    multAll (Prob innerxs, p) = map (B.second (p *)) innerxs

instance Applicative Prob where
  pure x = Prob [(x, 1 % 1)]
  Prob fs <*> Prob xs = Prob $ [(f x, p * q) | (f, p) <- fs, (x, q) <- xs]

instance Monad Prob where
  m >>= f = flatten (fmap f m)

instance MonadFail Prob where
  fail _ = Prob []
```

与原文不同之处在于，GHC 7.10 之后 `Monad` 的 superclass 是 `Applicative` ，因此需要实现 `instance Applicative Prob` ，详见 Haskell Wiki。对于实现 `Applicative` 的 `<*>` 可以参考十一章的 `instance Applicative []` ，即 `fs <*> xs = [f x | f <- fs, x <- xs]` 。具体来讲 `fs` 是一个函数列表，`xs` 为值列表，通过列表表达式将每个函数 `f` 应用至每个值 `x` 。那么在这里同理，`Prob fs` 是一个函数的概率列表，`Prob xs` 是一个值的概率列表，`(f, p) <- fs` 则是解构每个函数概率列表中的元素为 `f` 函数与 `p` 概率，而 `(x, q) <- xs` 则是解构每个值概率列表中的元素为 `x` 值与 `q` 概率，然后将解构出来的函数 `f` 应用至值 `x` ，两个概率 `p` 与 `q` 相乘，最后得到新的元组所构成的概率列表。还有一点不同的在于 `fail` 现在要单独实现在 `MonadFail` 上。

那么我们现在有了一个单子，那么我们可以用它做什么？好吧，它可以帮助我们计算概率，我们将概率事件的值视为带有 context，而概率单子可以确保这些概率能在最终的计算中反映出来。

假设我们有两个普通硬币以及一个灌铅的硬币，后者在十次抛掷中有九次会出现正面，那么一次性丢这三个硬币，有多大的概率会出现三个正面呢？

```
data Coin = Heads | Tails deriving (Show, Eq)
```

```
2
3  coin :: Prob Coin
4  coin = Prob [(Heads, 1 % 2), (Tails, 1 % 2)]
5
6  loadedCoin :: Prob Coin
7  loadedCoin = Prob [(Heads, 1 % 10), (Tails, 9 % 10)]
```

接下来是投掷三个硬币：

```
1  import Data.List (all)
2
3  flipThree :: Prob Bool
4  flipThree = do
5    a <- coin
6    b <- coin
7    c <- loadedCoin
8    return (all (== Tails) [a, b, c])
```

测试：

```
1  ghci> getProb flipThree
2  [(False,1 % 40),(False,9 % 40),(False,1 % 40),(False,9 % 40),
3   (False,1 % 40),(False,9 % 40),(False,1 % 40),(True,9 % 40)]
```

同时出现正面的机率是四十分之九，差不多是 25% 的机会。单子并没有办法 join 所有都是 False 的情形，即所有硬币都是出现反面的情况。不过那不是个严重的问题，可以写个函数来将同样的结果变成一种结果。以下是用于聚合 `Prob Bool` 的函数，返回值 `(Rational, Rational)` 是一个分别代表正确概率与错误概率的元组：

```
1  getJoinedProb :: Prob Bool -> (Rational, Rational)
2  getJoinedProb = foldr (\(x, p) (accT, accF) -> if x then (accT + p, accF) else (accT, accF +
     p)) (0, 0) . getProb
```

测试：

```
1  ghci> getJoinedProb flipThree
2  (9 % 40,31 % 40)
```

# 14 Zippers

虽然 Haskell 的纯性带来了非常多的便利，但是在处理某些问题时，我们需要不同与命令式语言那样的方式来解决。由于引用透明，Haskell 中相同的值时没有区别的。

如果我们有一颗树的节点都是五，当我们想要将它们其中一个变为六时，我们必须要知道到底究竟是哪个五想要变成六。也就是说必须知道其位置。在不纯的语言中，仅需知道其内存地址，并改变它。然而在 Haskell 中，一个五跟其它的五并无差异，因此我们不能通过内存地址来进行辨别。同样的，我们也不能真的修改任何东西；当我们说改变一颗树，真正的意思其实是接受一棵树，返回一个与原始树相近的树，且只有细微的不同。

一种方法就是从树的根部记住通向节点的路径，但是这样效率低下。如果想要修改一个临近之前修改过的节点，我们又要从头来过!

本章我们将学习如何集中注意在某个数据结构上，使得改变数据结构与遍历的动作变得高效。

## 走二元树

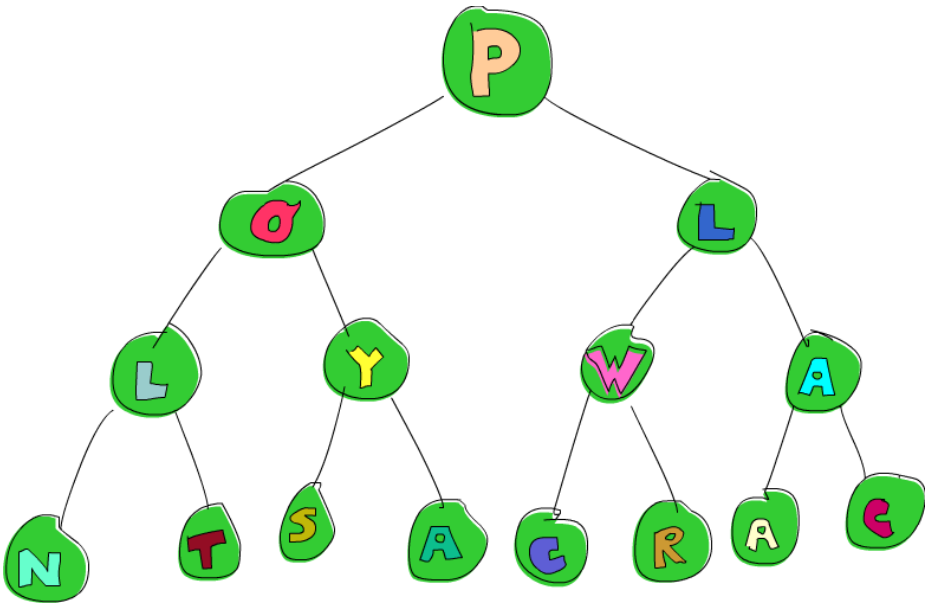
首先是之前章节介绍过的树的定义:

```haskell
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show)
```

下面是一个数的例子:

```haskell
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))
    )
```

其图形如下：




注意到 `W` 在树中的位置了吗？我们想要将其变为 `P` 。那么该怎么做呢？其中一种方式就是通过对树进行模式匹配，直到第一次找到该元素的位置，然后再进行修改：

```
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)
```

呃这不仅丑陋而且还令人迷惑。那么发生了什么？首先是模式匹配根元素 `x` （即 `'P'` ）以及左子树 `l` ，而对于右子树再次进行模式匹配，以此类推，最终替换 `'W'` 成为 `'P'` 。

那么有没有更好的办法呢？不如构建一个函数，接受一棵树以及一个方向列表。方向即 `L` 或 `R` ：

```
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
```

如果方向列表的第一个元素是 `L` ，我们将构建一颗与旧树一样的树，只不过左子树会有元素变为 `'P'` 。当递归调用 `changeToP` ，则传入列表尾部，因为头部已经被消费了。 `R` 同理。如果列表为空，意味着到达了目的地，返回一个新的树，其根被改为 `'P'` 。

为了避免打印出整棵树，让我们构建一个函数，接受一个方向的列表，返回目的地的元素：

```
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
```

这个函数跟 `changeToP` 很像，只不过不是记住路径并重构树，而是忽略所有经过只记住目的地。这里修改了 `'W'` 至 `'P'` ，检查是否修改成功：

```
ghci> let newTree = changeToP [R,L] freeTree
ghci> elemAt [R,L] newTree
'P'
```

虽然这个技巧可能看起来很酷，但是它还是低效，特别是当我们需要频繁的修改元素时。假设我们有一个很大的树，一个指向了树底部元素的很长的方向列表。那么当我们修改一个元素后，又要修改另一个临近的元素，我们又得重头开始一直走到树的底部！

## 留下面包屑的路径

那么专注在一个子树，我们希望一个比方向列表更好的东西，避免总是从树的根部开始。那么如果从树根部开始，每次移动一步并留下印记，类似于留下面包屑会有帮助吗？换言之，向左时记住向左，向右时记住向右。

为了表示面包屑，可以使用一个 `Direction` 列表，由于我们留下的方向现在是相反的，为了区分 `Directions` ，我们称其为 `Breadcrumbs` ，

```
type Breadcrumbs = [Direction]
```

下面是一个接受一棵树以及面包屑的函数，当移动至左子树时，添加 `L` 在列表头部：

```
goLeft :: (Tree a, Breadcrumbs) -> (Tree a, Breadcrumbs)
goLeft (Node _ l _, bs) = (l, L : bs)
```

忽略根以及右子树，返回左子树以及在旧的面包屑头部添加了 `L` 的新面包屑。下面是往右的函数：

```
goRight :: (Tree a, Breadcrumbs) -> (Tree a, Breadcrumbs)
goRight (Node _ _ r, bs) = (r, R : bs)
```

现在使用这些函数来接受我们的 `freeTree` ，先是向右再向左：

```
ghci> goLeft (goRight (freeTree, []))
(Node 'W' (Node 'C' Empty Empty) (Node 'R' Empty Empty),[L,R])
```

现在有了一个以 `'W'` 为根的树，`'C'` 为左子树的根而 `'R'` 为右子树的根，面包屑则为 `[L, R]` ，因为我们是先右后左。

我们可以定义一个 `-:` 函数来将代码变得更可读：

```
x -: f = f x
```

它可以将函数的应用变为先写值，然后是 `-:` 接着函数。那么改写以后就成了：

```
ghci> (freeTree, []) -: goRight -: goLeft
(Node 'W' (Node 'C' Empty Empty) (Node 'R' Empty Empty),[L,R])
```

### 从后至前

那么如果想要反向走树呢？根据面包屑我们可以知道当前树是其父树的左子树，然后更上一层则是右子树。仅仅如此，它并没有告知当前子树其父足够的信息，让我们能够继续往树上走。除了单纯记录方向之外，还必须把其它的数据记录下来。这个案例中就是子树的父信息以及其右子树。

一般而言，一个面包屑有足够的信息用于重建父节点。所以它应该包含所有没有选择的路径信息，记录我们走过的方向；同时它不应该包含现在关注的子树，因为它已经在元组的第一部分了，如果也记录下来就会有重复的信息。

现在来修改一下面包屑的定义，让它包含我们之前丢掉的信息：

```
data Crumb a = LeftCrumb a (Tree a) | RightCrumb a (Tree a) deriving (Show)
```

用 `LeftCrumb` 来包含我们走过的右子树，以及没走过的右子树，而不是仅仅写个 `L` ，`RightCrumb` 同理。

这些面包屑现在包含了重构树所需的所有信息。它们像是软盘一样存储了走过的路径，而不仅仅只有方向。

基本而言可以将每个面包屑视为一个树的节点，树的节点有一个洞，当向树的更深处走，面包屑携带所有走过的信息，除了当前关注的子树被记录在洞那里。

我们也要把 `Breadcrumbs` 的类型别名改成：

```
type Breadcrumbs' a = [Crumb a]
```

接着修改 `goLeft` 以及 `goRight` 来记录没有走过的路径信息，而不是像之前那样忽略掉：

```
goLeft' :: (Tree a, Breadcrumbs' a) -> (Tree a, Breadcrumbs' a)
goLeft' (Node x l r, bs) = (l, LeftCrumb x r : bs)
```

类似于之前的 `goLeft` ，不同于仅仅添加一个 `L` 在面包屑列表头部，而是添加了一个 `LeftCrumb` 记录从左走过，以及未走过的右子树。

注意该函数假设了当前注意的树并不为 `Empty` 。一颗空树是不会有任何子树的，所以如果在一颗空树上向左走，异常将会出现，因为对 `Node` 的匹配不成功，且没有额外去匹配 `Empty` 。

`goRight` 也类似：

```
goRight' :: (Tree a, Breadcrumbs' a) -> (Tree a, Breadcrumbs' a)
goRight' (Node x l r, bs) = (r, RightCrumb x l : bs)
```

现在有了向左和向右，我们就能具备了返回树根的能力了，以下是 `goUp` 函数：

```
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)
```

该函数接受一个正在关注的树 `t` ，以及一个 `Breadcrumbs` 用于检查最近的 `Crumb` ，如果是一个 `LeftCrumb` ，那么构建一颗新树，而 `t` 作为该树的左子树。这里通过两次模式匹配，首先是 `Breadcrumbs` 的头，接着是 `LeftCrumb` ，拿到 `x` 右子树 `r` 用来构建 `Node` 剩余部分，并将剩下的 `Breadcrumbs` 即 `bs` 一并返回。

注意该函数在树的顶部会导致错误，稍后我们将用 `Maybe` 单子来代表失败来进行改进。

有了一对元组 `Tree a` 与 `Breadcrumbs a` ，我们便具备了所有重构的信息。这个模版可以让向上，向左以及向右变得简单很多。这样一对包含了关注的部分以及其它信息的元组被称为 zipper。因此我们可以对其进行类型别名：

```
type Zipper a = (Tree a, Breadcrumbs' a)
```

### 操作当前注意的树

现在我们有了向上与向下的移动，那么就可以构建一个函数用于修改 zipper 所关注的元素了：

```
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)
```

如果关注在一个节点，通过函数 `f` 修改其根元素；如果关注在一颗空树，则不做改动。测试：

```
ghci> let newFocus = modify (\_ -> 'P') (goRight (goLeft (freeTree,[])))
```

用可读性更好的 `-:` 函数：

```
ghci> let newFocus = (freeTree,[]) -: goLeft -: goRight -: modify (\_ -> 'P')
```

接着是向上移动，替换字符为 `'X'` ：

```
ghci> let newFocus2 = modify (\_ -> 'X') (goUp newFocus)
```

使用 `-:` ：

```
ghci> let newFocus2 = newFocus -: goUp -: modify (\_ -> 'X')
```

向上移动很简单，因为面包屑的另一部分数据结构是没有关注的，不过它是倒转过来的，有点像是要把袜子反过来才能用那样。有了这些信息，我们便不用再从根走一遍，而是从反过来的树顶部开始，将其部分翻转后再将其置为关注。

每个节点都有两个子树，即使这些子树是空树。所以当关注了一颗空的子树，可以将其替换为一颗非空子树，即叶节点：

```
attach :: Tree a -> Zipper a -> Zipper a
attach t (_, bs) = (t, bs)
```

接受一颗树以及一个 zipper 并返回一个新的 zipper，其关注的被替换为接受的树。我们不仅能用这个方法来延展一棵树，还可以替换整个子树。通过 `attach` 来替换 `freeTree` 最左的树：

```
ghci> let farLeft = (freeTree,[]) -: goLeft -: goLeft -: goLeft -: goLeft
ghci> let newFocus = farLeft -: attach (Node 'Z' Empty Empty)
```

### 直接走到树顶端

直接走到树顶端很简单：

```
topMost :: Zipper a -> Zipper a
topMost (t,[]) = (t,[])
topMost z = topMost (goUp z)
```

如果面包屑没了，那么就表示已经在树的根处了，并返回当前关注的树。

## 专注于列表

Zippers 几乎可以用于任何数据结构，不出意外它可以被用于列表的列表。毕竟列表很像树，只不过在树中的节点有若干子树，而列表中的节点只包含一个子列表。我们在定义自己的列表类型那一章中，定义了如下数据类型：

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
```

现在让我们为列表做一个 zipper。切换子列表关注，我们只需要向前或者向后移动（不同于树的向上，向左或向右）。在树的情景下，关注的部分是一颗子树，另外还携带就是移动后的面包屑。那么对于列表而言，单个面包屑应该由什么构成呢？在处理二元树时，面包屑必须包含父节点的根以及其它没有走过的子树，同时还要记住左右方向。因此它必须包含一个节点的所有信息，除了正在关注子树（存于 zipper 的第一部分）。

列表比数简单多了，无需记住左右方向，因为只有一个方向即深入列表。又因为每个节点仅有一颗子树，也无需记住没有走过的路径。这么看上去只需要记住上一个元素即可。假设有一个列表 `[3,4,5]` 且已知上一个元素是 `2` ，那么可以将其放置在列表头部，得 `[2,3,4,5]` 。

由于单个面包屑在这里仅为一个元素，我们无需将其置入一个数据类型，比如为树所创建的 `Crumb` 数据类型那样，只需要类型别名即可：

```
type ListZipper a = ([a], [a])
```

这里第一个列表代表正在关注着的列表，而第二个列表则是面包屑列表。下面是向前与向后的函数：

```
goForward :: ListZipper a -> ListZipper a
goForward (x : xs, bs) = (xs, x : bs)

goBack :: ListZipper a -> ListZipper a
goBack (xs, b : bs) = (b : xs, bs)
```

当向前时，我们关注当前列表的尾部并留下头部元素作为一个面包屑；当向后时，我们获取最近的面包屑并将其置入关注列表的头部。

测试：

```
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])
```

可以看到面包屑在列表的情况下就是一个倒转的列表。移出的元素进入面包屑的头部，因此移动回去只需将面包屑的头部元素移动进关注列表。

这也是为什么称其为一个拉链 zipper，因为这看上去就像是拉链上下移动。

## 一个简单的文件系统

理解了 zipper 是如何运作的，现在让我们用树来表达一个简单的文件一同，接着为其构建一个 zipper，使得我们可以在文件夹之间移动，就像是我们通常使用文件系统进行跳转那样。

简单的来看一个文件系统，基本上就是由文件与文件夹构成。以下是一些类型别名以及一个数据类型：

```
type Name = String
type Data = String
data FSItem = File Name Data | Folder Name [FSItem] deriving (Show)
```

文件包含了两个字符串，前者为名称，后者为数据。文件夹包含了一个字符串以及一个列表的文件系统，如果列表为空则意为空文件夹。

以下是一个包含了文件和子文件夹的文件夹：

```
myDisk :: FSItem
myDisk =
  Folder
    "root"
    [ File "goat_yelling_like_man.wmv" "baaaaaa",
      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"
            ]
        ]
    ]
```

### 文件系统的 zipper

我们有了一个文件系统，我们需要一个 zipper 是我们可以走动，增加，修改，或者移除文件或文件夹。像是二元树或列表那样，用面包屑来保存走过路径的信息。一个面包屑就是一个节点，它包含了除了正在关注的子树之外的所有信息。

在此，一个面包屑应该像是一个文件夹，它无需包含当前关注的文件夹。你可能会问为什么不是一个文件呢？因为如果关注在一个文件，我们就无法深入文件系统，因此留下一个从文件而来的面包屑没有意义。一个文件就类似于一颗空树。

如果关注在 `"root"` 文件夹，接着将关注转到文件 `"dijon_poupon.doc"` ，那么这个时候的面包屑应该长什么样？它应该包含了父文件夹的名称，以及这个文件之前以及之后的所有项。通过两个独立的列表分别保存之前的项与之后的项，我们可以精确的知道回去时的地址。

下面是面包屑的类型：

```
data FSCrumb = FSCrumb Name [FSItem] [FSItem] deriving (Show)
```

以及 zipper 的类型别名：

```
type FSZipper = (FSItem, [FSCrumb])
```

返回父节点非常的容易，仅需将最近的面包屑与当前所关注的进行组合：

```
fsUp :: FSZipper -> FSZipper
fsUp (item, FSCrumb name ls rs : bs) = (Folder name (ls ++ [item] ++ rs), bs)
```

因为面包屑知道父文件夹的名称，以及所关注项之前（ `ls` ）与之后（ `rs` ）的所有项，所以移动就很简单了。

那么更深入文件系统呢？假设从 `"root"` 到 `"dijon_poupon.doc"` ，那么面包屑就包含了 `"root"` ，以及在 `"dijon_poupon.doc"` 之前的所有项以及之后的所有项。

下面是一个函数，接受一个名称，并关注在当前文件夹下的一个文件或文件夹：

```
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` 接受一个 `Name` 与一个 `FSZipper` 并返回一个新的 `FSZipper` ，其关注移至给定名称。该文件必须处于当前所关注的文件夹下。

这里首先用了 `break` 来将列表分成前后两个列表。`break` 函数接受一个子句以及一个列表，返回两个列表，前者为不满足子句的所有项所组成的列表，而后者为第一个满足子句的项以及该项之后的所有项所组成的列表。

那么在此，`ls` 就是一个在查询项之前的所有项，`item` 就是查询的项，而 `rs` 则是 `item` 后的所有项。有了这些，从 `break` 所得到的项作为关注项，并构建一个面包屑。

注意如果所查询的项并非一个文件夹，模式 `item:rs` 则会尝试匹配一个空列表，继而报错。同理，如果当前关注的并非一个文件夹而是一个文件时，我们也会得到报错使得程序崩溃。

现在可以对文件系统进行上下移动了。首先从根开始，再走到 `"skull_man(scary).bmp"` 文件那儿。

```
let newFocus = (myDisk, []) -: fsTo "pics" -: fsTo "skull_man(scary).bmp"
```

`newFocus` 现在是一个 zipper，其关注在 `"skull_man(scary).bmp"` 文件上。现在让我们看看它是否正确：

```
ghci> fst newFocus
File "skull_man(scary).bmp" "Yikes!"
```

接着向上移动接着关注到相邻文件 `"watermelon_smash.gif"` ：

```
ghci> let newFocus2 = newFocus -: fsUp -: fsTo "watermelon_smash.gif"
ghci> fst newFocus2
File "watermelon_smash.gif" "smash!!"
```

### 操作文件系统

我们现在知道了该如何在文件系统中移动，操作它们也很简单。以下函数用于对当前关注的文件或文件夹进行改名：

```
fsRename :: Name -> FSZipper -> FSZipper
fsRename newName (Folder name items, bs) = (Folder newName items, bs)
fsRename newName (File name dat, bs) = (File newName dat, bs)
```

将 `"pics"` 文件夹改名为 `"cspi"` ：

```
ghci> let newFocus = (myDisk,[]) -: fsTo "pics" -: fsRename "cspi" -: fsUp
```

接下来是创建新文件：

```
fsNewFile :: FSItem -> FSZipper -> FSZipper
fsNewFile item (Folder folderName items, bs) = (Folder folderName (item : items), bs)
```

让我们在 `"pics"` 文件夹下创建一个新文件，然后再移动回跟目录：

```
ghci> let newFocus = (myDisk,[]) -: fsTo "pics" -: fsNewFile (File "heh.jpg" "lol") -: fsUp
```

这里很酷的一点就是当我们在修改文件系统时，它并没有真正的被修改，而是返回了一个新的文件系统。这样我们就可以访问我们旧的文件系统（即 `myDisk` ），同时可以通过 zipper 访问新的系统（ `newFocus` 的第一个部分），这样就自动获得了版本控制。这并不仅限于 zippers，而是 Haskell 的数据结构是不可变的特性。有了 zippers 我们可以更加轻松且有效率的处理数据结构，也因此 Haskell 数据结构的持久性才真正开始闪耀。

## 注意你的脚步

迄今如此，当走过数据结构时，无论是二元树，列表还是文件系统，我们并没有关心走了几步。例如 `goLeft` 函数接受一个二元树的 zipper，接着将关注移动至子树：

```
goLeft :: Zipper a -> Zipper a
goLeft (Node x l r, bs) = (l, LeftCrumb x r:bs)
```

如果我们走到了一个空树会如何呢？也就是说不是一个 `Node` ，而是一个 `Empty` 。这种情况下我们会得到一个运行时错误，因为模式匹配将失败，同时我们没有为空树设计匹配。

让我们使用 `Maybe` 单子来为我们的移动添加一个可能失败的 context。

```
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
```

测试：

```
ghci> goLeft (Empty, [])
Nothing
ghci> goLeft (Node 'A' Empty Empty, [])
Just (Empty,[LeftCrumb 'A' Empty])
```

那么向上呢？

```
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
```

测试：

```
gchi> let newFocus = (freeTree,[]) -: goLeft -: goRight
```

由于现在返回的是 `Maybe (Zipper a)` 而不是 `Zipper a` ，那么链式函数的方法就不管用了。不过我们可以使用 `>>=` 来解决这个问题。

```
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
```