Learn Me a Haskell

Jacob Bishop

2023-07-01

1 Introduction

WIP

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缀函数。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 -> b` 。

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。

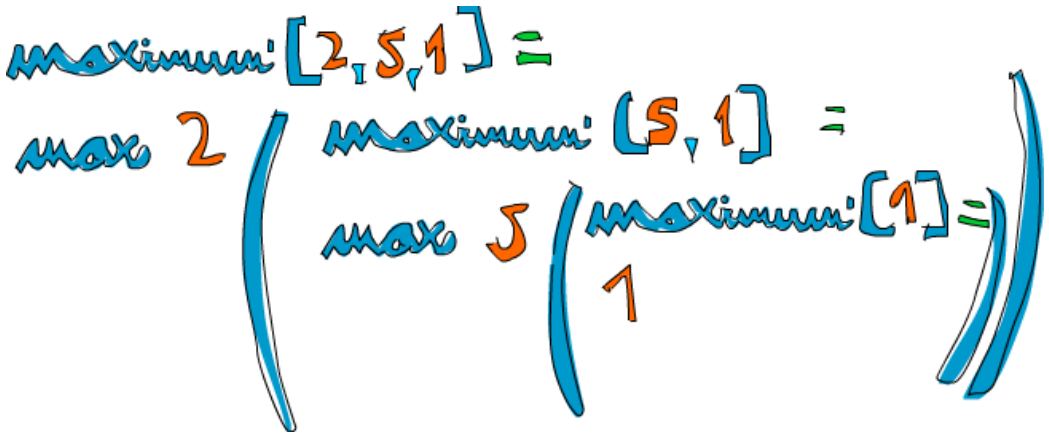
Maximum

`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
```

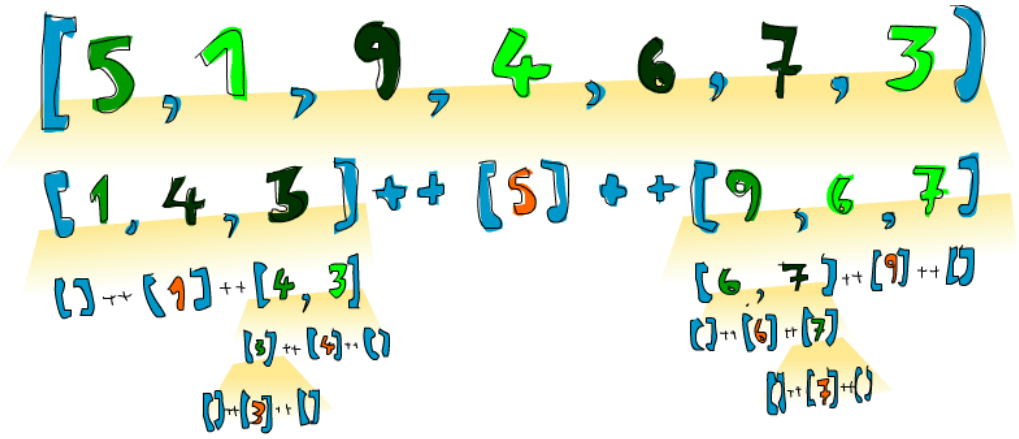
快排!

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 -> Ordering)`，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。

中缀函数同样可以通过使用分割被部分应用。要分割中缀函数，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：

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

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) 8.9` 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) (zipWith max [1,2,3,4,5] [4,5,6,7,8])))` ，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

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

如果需要可读性，则可以这样：

```
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：

```
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) => [b] -> 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)

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

```

```
36  rectangleArea :: Float -> Float -> Float
37  -- rectangleArea a b = a * b
38  rectangleArea = (*)
```

我们仅需像下面这样就能使用模块:

```
1 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 ：

```
1  module Geometry.Sphere
2  ( volume,
3    area,
4  )
5  where
6
7  volume :: Float -> Float
8  volume = (* (4.0 / 3.0 * pi)) . (^ 3)
9
10 area :: Float -> Float
11 area = (* (4 * pi)) . (^ 3)
```

Cuboid.hs ：

```
1  module Geometry.Cuboid
2    ( volume,
3      area,
4    )
5  where
6
7  volume :: Float -> Float -> Float -> Float
8  volume = ((*) .) . rectangleArea
9
10 area :: Float -> Float -> Float -> Float
11 area a b c = rectangleArea a b * 2 + rectangleArea a c * 2 + rectangleArea c b * 2
12
13 rectangleArea :: Float -> Float -> Float
14 rectangleArea = (*)
```

以及 Cube.hs ：

```
1  module Geometry.Cube
2    ( volume,
3      area,
4    )
5  where
```

```
6
7   import Control.Monad
8   import Geometry.Cuboid qualified as Cuboid
9
10  volume :: Float -> Float
11  volume = join (join Cuboid.volume)
12
13  area :: Float -> Float
14  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：

```
1 import Geometry.Sphere
```

为了避免命名冲突可以使用 qualified import：

```
1   import qualified Geometry.Sphere as Sphere
2   import qualified Geometry.Cuboid as Cuboid
3   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)] -> Bool 。

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缀的性质。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 -> f b` 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：

```haskell
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）：

```haskell
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：

```haskell
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：

```haskell
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
looooooooooooooooooooooooooong
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`：

```haskell
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 numberedTasks` 。

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

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：

```
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`

```
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, g)`

```
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
  unless (null numberString) $ do
    let number = read numberString
    if randNumber == number
      then putStrLn "You are correct!"
      else putStrLn $ "Sorry, it was" ++ show randNumber
    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）：

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

```
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：

```
main = do toTry `catch` handler1
  thenTryThis `catch` handler2
  launchRockets
```

10 Functionally Solving Problems

WIP

11 Functors, Applicative Functors and Monoids

WIP

12 A Fistful of Monads

WIP

13 For a Few Monads More

WIP

14 Zippers

WIP