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。

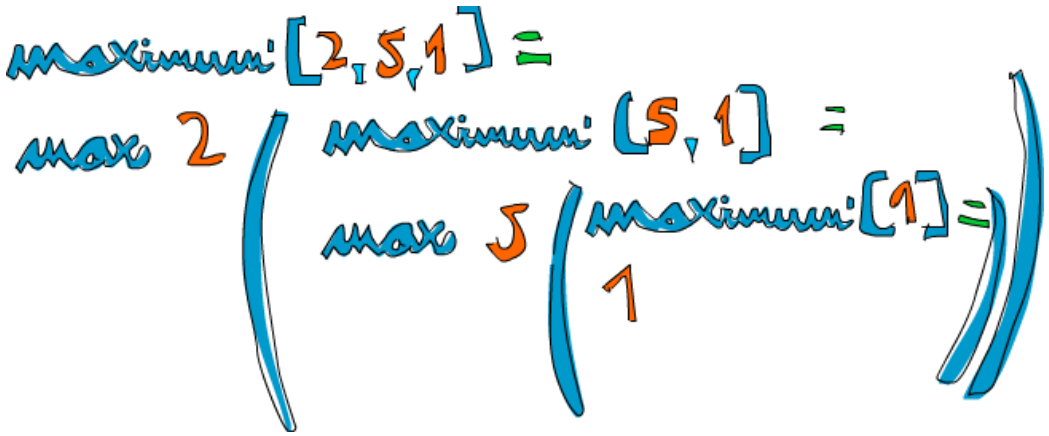
Maximum

`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
```

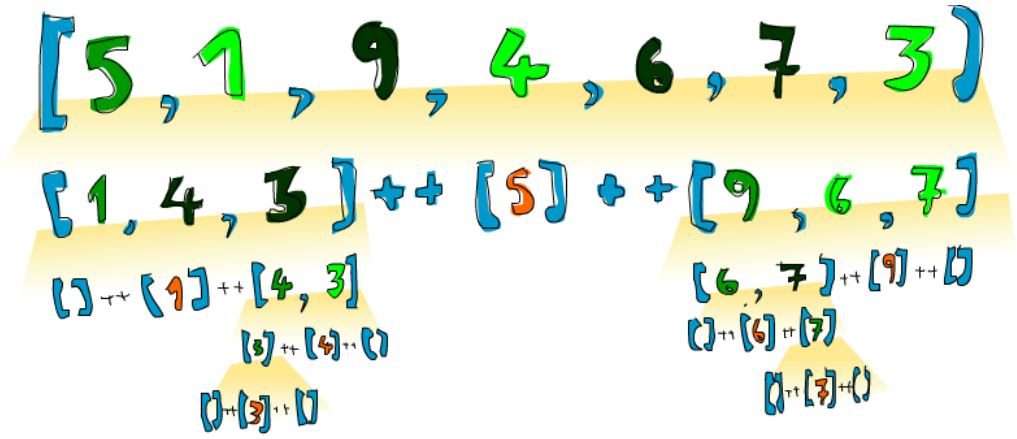
快排!

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，这样的话就可以映射函数应用至一个列表的函数了。

```
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

WIP

8 Making Our Own Types and Typeclasses

WIP

9 Input and Output

WIP

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