

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：

```
1 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`）：

```
1 ghci> :l baby
2 [1 of 1] Compiling Main           ( baby.hs, interpreted )
3 Ok, modules loaded: Main.
4 ghci> doubleMe 9
5 18
6 ghci> doubleMe 8.3
7 16.6
```

一个带有 `if` 的函数：

```
1 doubleSmallNumber x =
2   if x > 100
3   then x
4   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：

```
1 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：

```
1 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：

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

可以使用 `:` 操作符将元素直接添加至数组头部：

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

```
1 ghci> "Steve Buscemi" !! 6
2 'B'
3 ghci> [9.4,33.2,96.2,11.2,23.25] !! 1
4 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`：

```
1 ghci> head [5,4,3,2,1]
2 5
3 ghci> tail [5,4,3,2,1]
4 [4,3,2,1]
5 ghci> last [5,4,3,2,1]
6 1
7 ghci> init [5,4,3,2,1]
8 [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。

```
1 ghci> length [5,4,3,2,1]
2 5
3
4 ghci> null [1,2,3]
5 False
6 ghci> null []
7 True
8
9 ghci> reverse [5,4,3,2,1]
10 [1,2,3,4,5]
11
12 ghci> take 3 [5,4,3,2,1]
13 [5,4,3]
14 ghci> take 1 [3,9,3]
15 [3]
16 ghci> take 5 [1,2]
17 [1,2]
18 ghci> take 0 [6,6,6]
19 []
20
21 ghci> drop 3 [8,4,2,1,5,6]
22 [1,5,6]
23 ghci> drop 0 [1,2,3,4]
24 [1,2,3,4]
25 ghci> drop 100 [1,2,3,4]
26 []
27
28 ghci> minimum [8,4,2,1,5,6]
29 1
30 ghci> maximum [1,9,2,3,4]
31 9
32
33 ghci> sum [5,2,1,6,3,2,5,7]
34 31
35 ghci> product [6,2,1,2]
36 24
37 ghci> product [1,2,5,6,7,9,2,0]
```

```

38 0
39
40 ghci> 4 `elem` [3,4,5,6]
41 True
42 ghci> 10 `elem` [3,4,5,6]
43 False

```

Texas 排列

```

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

```

带有 step 的排列:

```

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

```

而对于浮点数的排列需要注意精度问题:

```

1 ghci> [0.1, 0.3 .. 1]
2 [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:

```

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

```

repeat 重复:

```

1 ghci> take 10 (repeat 5)
2 [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:

```

1 ghci> replicate 3 10
2 [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:

```
1 ghci> [x*2 | x <- [1..10]]
2 [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）：

```
1 ghci> [x*2 | x <- [1..10], x*2 >= 12]
2 [12,14,16,18,20]
3 ghci> [ x | x <- [50..100], x `mod` 7 == 3]
4 [52,59,66,73,80,87,94]
```

将列表表达式置于一个函数中便于复用：

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

多个谓词也是可以的：

```
1 ghci> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
2 [10,11,12,14,16,17,18,20]
```

除此之外，还可以处理若干数组：

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

当然也可以加上谓词：

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

那么对于字符串也可以使用列表表达式：

```
1 ghci> let nouns = ["hobo", "frog", "pope"]
2 ghci> let adjectives = ["lazy", "grouchy", "scheming"]
3 ghci> [adjective ++ " " ++ noun | adjective <- adjectives, noun <- nouns]
4 ["lazy hobo", "lazy frog", "lazy pope", "grouchy hobo", "grouchy frog",
5 "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）：

```
1 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：

```
1 removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
2 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：

```
1 ghci> fst (8,11)
2 8
3 ghci> fst ("Wow", False)
4 "Wow"
```

snd 获取对元组的第二个元素：

```
1 ghci> snd (8,11)
2 11
3 ghci> snd ("Wow", False)
4 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

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

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

```
1 ghci> :t 'a'
2 'a' :: Char
3 ghci> :t True
4 True :: Bool
5 ghci> :t "HELLO!"
6 "HELLO!" :: [Char]
7 ghci> :t (True, 'a')
8 (True, 'a') :: (Bool, Char)
9 ghci> :t 4 == 5
10 4 == 5 :: Bool
```

函数同样拥有类型：

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

而对于有多个入参的函数而言：

```
1 ghci> addThree x y z = x + y + z
2 ghci> :t addThree
3 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。

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？

```
1 ghci> :t head
2 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？

```
1 ghci> :t (==)
2 (==) :: 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。

```
1 ghci> read "True" || False
2 True
3 ghci> read "8.2" + 3.8
4 12.0
5 ghci> read "5" - 2
6 3
7 ghci> read "[1,2,3,4]" ++ [3]
8 [1,2,3,4,3]
```

但是如果尝试一下 `read "4"` 呢?

```
1 ghci> read "4"
2 <interactive>:1:0:
3   Ambiguous type variable `a' in the constraint:
4     `Read a' arising from a use of `read' at <interactive>:1:0-7
5   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:

```
1 ghci> :t read
2 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:

```
1 ghci> read "5" :: Int
2 5
3 ghci> read "5" :: Float
4 5.0
5 ghci> (read "5" :: Float) * 4
6 20.0
7 ghci> read "[1,2,3,4]" :: [Int]
8 [1,2,3,4]
9 ghci> read "(3, 'a')" :: (Int, Char)
10 (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:

```
1 ghci> minBound :: Int
2 -2147483648
3 ghci> maxBound :: Char
4 '\1114111'
5 ghci> maxBound :: Bool
6 True
7 ghci> minBound :: Bool
8 False
```

元组的成员如果为 `Bounded` 的一部分, 那么元组也是:

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

`Num` 是一个数值类:

```
1 ghci> :t 20
2 20 :: (Num t) => t
3 ghci> 20 :: Int
4 20
5 ghci> 20 :: Integer
6 20
7 ghci> 20 :: Float
8 20.0
9 ghci> 20 :: Double
10 20.0
```

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

```
1 ghci> :t (*)
2 (*) :: (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：

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

一个递归案例：

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

模式匹配也可以失败：

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

当输入并不是期望时：

```
1 ghci> charName 'a'
2 "Albert"
3 ghci> charName 'b'
4 "Broseph"
5 ghci> charName 'h'
6 "*** 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。

模式匹配也可作用于元组：

```
1 addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
2 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：

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

```
1 tell :: (Show a) => [a] -> String
2 tell [] = "The list is empty"
3 tell (x:[]) = "The list has one element: " ++ show x
4 tell (x:y:[]) = "The list has two elements: " ++ show x ++ " and " ++ show y
5 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：

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

接下来是实现 `sum`：

```
1 sum' :: (Num a) => [a] -> a
2 sum' [] = 0
3 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：

```
1 capital :: String -> String
2 capital "" = "Empty string, whoops!"
3 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：

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

```

4 | bmi <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
5 | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
6 | 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：

```

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

```

另外我们可以实现自己的 `max` 与 `compare` 函数：

```

1 max' :: (Ord a) => a -> a -> a
2 max' a b
3   | a > b = a
4   | otherwise = b

1 compare' :: (Ord a) => a -> a -> Ordering
2 a `compare'` b
3   | a > b = GT
4   | a == b = EQ
5   | 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：

```

1 bmiTell'' :: (RealFloat a) => a -> a -> String
2 bmiTell'' weight height
3   | bmi <= 18.5 = "You're underweight, you emo, you!"
4   | bmi <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
5   | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"

```

```

6 | otherwise = "You're a whale, congratulations!"
7 where
8   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：

```

1 bmiTell'' :: (RealFloat a) => a -> a -> String
2 bmiTell'' weight height
3   | bmi <= skinny = "You're underweight, you emo, you!"
4   | bmi <= normal = "You're supposedly normal. Pffft, I bet you're ugly!"
5   | bmi <= fat    = "You're fat! Lose some weight, fatty!"
6   | otherwise    = "You're a whale, congratulations!"
7 where
8   bmi = weight / height ^ 2
9   skinny = 18.5
10  normal = 25.0
11  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：

```

1 ...
2 where
3   bmi = weight / height ^ 2
4   (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：

```

1 initials :: String -> String -> String
2 initials first_name last_name = [f] ++ ". " ++ [l] ++ "."
3 where
4   (f : _) = first_name
5   (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：

```

1 calcBmis :: (RealFloat a) => [(a, a)] -> [a]
2 calcBmis xs = [bmi w h | (w, h) <- xs]
3 where
4   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：

```
1 cylinder :: (RealFloat a) => a -> a -> a
2 cylinder r h =
3   let sideArea = 2 * pi * r * h
4       topArea = pi * r ^ 2
5   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：

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

那么 `let` 绑定也可以：

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

同样可以在当前作用域引入函数：

```
1 ghci> [let square x = x * x in (square 5, square 3, square 2)]
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：

```
1 ghci> (let a = 100; b = 200; c = 300 in a*b*c, let foo="Hey "; bar = "there!" in foo ++ bar)
2 (6000000,"Hey there!")
```

正如之前提到的，可以将模式匹配应用于 `let` 绑定：

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

当然也可以将 `let` 绑定置入列表表达式中：

```
1 calcBmis' :: (RealFloat a) => [(a, a)] -> [a]
2 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：

```
1 calcBmis'' :: (RealFloat a) => [(a, a)] -> [a]
2 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。

```
1 ghci> let zoot x y z = x * y + z
2 ghci> zoot 3 9 2
3 29
4 ghci> let boot x y z = x * y + z in boot 3 4 2
5 14
6 ghci> boot
7 <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。

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

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

正如所见的那样，`case` 表达式的语法特别简单：

```
1 case expression of pattern -> result
2 pattern -> result
3 pattern -> result
4 ...
```

`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：

```
1 describeList :: [a] -> String
2 describeList xs =
3   "The list is " ++ case xs of
4     [] -> "empty."
5     [x] -> "a singleton list."
6     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：

```
1 describeList' :: [a] -> String
2 describeList' xs = "The list is " ++ what xs
3 where
4     what [] = "empty."
5     what [x] = "a singleton list."
6     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:

```
1 maximum' :: (Ord a) => [a] -> a
2 maximum' [] = error "maximum of empty list"
3 maximum' [x] = x
4 maximum' (x : xs)
5   | x > maxTail = x
6   | otherwise = maxTail
7 where
8   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。

$$\begin{aligned} \text{maximum}' [2, 5, 1] &= \\ \max 2 \left(\begin{aligned} &\text{maximum}' [5, 1] = \\ &\max 5 \left(\begin{aligned} &\text{maximum}' [1] = \\ &1 \end{aligned} \right) \end{aligned} \right) \end{aligned}$$

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。

```
1 replicate' :: (Num i, Ord i) => i -> a -> [a]
2 replicate' n x
3   | n <= 0 = []
4   | 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`：

```
1 take' :: (Num i, Ord i) => i -> [a] -> [a]
2 take' n _
3   | n <= 0 = []
4 take' _ [] = []
5 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。

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

接下来是 `reverse` 函数：

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

由于 Haskell 支持无限列表，`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任意处进行截断。然后是 `repeat` 函数，其返回一个无限列表：

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

接下来是 `zip` 函数：

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

最后一个是 `elem` 函数：

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

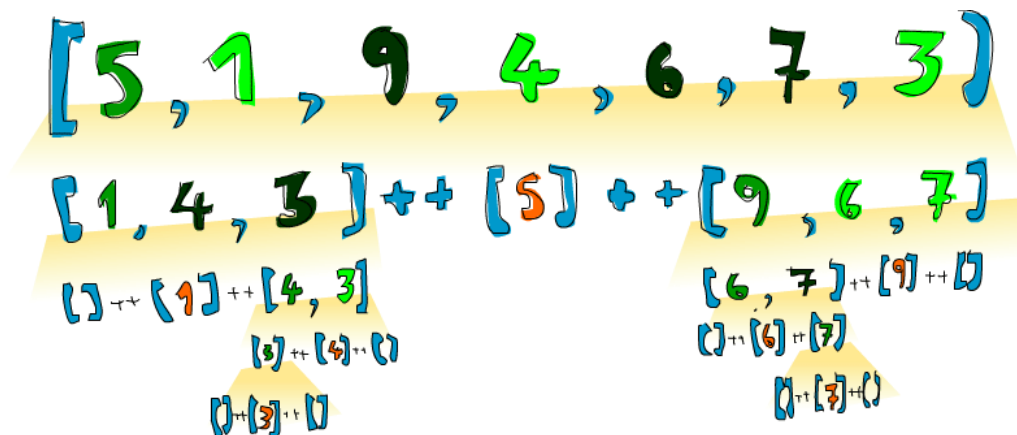
快排!

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：

```
1 quicksort :: (Ord a) => [a] -> [a]
2 quicksort [] = []
3 quicksort (x:xs) =
4   let
5     smallerSorted = quicksort [a | a <- xs, a <= x]
6     biggerSorted = quicksort [a | a <- xs, a > x]
7   in
8     smallerSorted ++ [x] ++ biggerSorted
```

测试：

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



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？这就是柯里化函数 **curried functions**。

```
1 ghci> max 4 5
2 5
3 ghci> (max 4) 5
4 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：

```
1 multThree :: (Num a) => a -> a -> a -> a
2 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：

```
1 ghci> let multTwoWithNine = multThree 9
2 ghci> multTwoWithNine 2 3
3 54
4 ghci> let multWithEighteen = multTwoWithNine 2
5 ghci> multWithEighteen 10
6 180
```

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？

```
1 compareWithHundred :: (Num a, Ord a) => a -> Ordering
2 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：

```
1 compareWithHundred :: (Num a, Ord a) => a -> Ordering
2 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：

```
1 divideByTen :: (Floating a) => a -> a
2 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？

```
1 ghci> multThree 3 4
2 <interactive>:1:0:
3   No instance for (Show (t -> t))
4     arising from a use of `print' at <interactive>:1:0-12
5   Possible fix: add an instance declaration for (Show (t -> t))
6   In the expression: print it
7   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。

一些高级是在于顺序

函数可以接受函数作为其参数，也可以返回函数。

```
1 applyTwice :: (a -> a) -> a -> a
2 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`。

```
1 ghci> applyTwice (+3) 10
2 16
3 ghci> applyTwice (++ " HAHA") "HEY"
4 "HEY HAHA HAHA"
5 ghci> applyTwice ("HAHA " ++) "HEY"
6 "HAHA HAHA HEY"
```

```

7 ghci> applyTwice (multThree 2 2) 9
8 144
9 ghci> applyTwice (3:) [1]
10 [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：

```

1 flip' :: (a -> b -> c) -> (b -> a -> c)
2 flip' f = g
3   where
4     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：

```

1 flip'' :: (a -> b -> c) -> b -> a -> c
2 flip'' 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。

```

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

```

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