# Real World Haskell中文版

## 第一章：入门

Prelude> :set prompt "ghci>"

ghci>

ghci> :module + Data.Ratio

ghci> True && 1

<interactive>:1:8:

No instance for (Num Bool)

arising from the literal `1' at <interactive>:1:8

Possible fix: add an instance declaration for (Num Bool)

In the second argument of `(&&)', namely `1'

In the expression: True && 1

In the definition of `it': it = True && 1

* “No instance for (Num Bool)” 告诉我们**ghci**尝试解析数字1为Bool类型但是失败。
* “arising from the literal `1’” 表示是由于使用了数字1而引发了问题。
* “In the definition of `it’” 引用了一个**ghci**的快捷方式。我们会在后面提到。
* Prelude> :set +t
* Prelude> 'c' *-- 输入表达式*
* 'c' *-- 输出值*
* it **::** Char *-- 输出值的类型*
* Prelude> "foo"
* "foo"
* it **::** [Char]

Prelude Data.Ratio> 3 + 2

5

Prelude Data.Ratio> :**type** it

it **::** Integer

Prelude Data.Ratio> :**type** 3 + 2

3 + 2 **::** Num a **=>** a

## 第二章：类型和函数

Haskell中每个函数和表达式都有各自的类型T。一个值的类型标识了它和该类型的其他值所共有的一簇属性（property）。

Haskell 中的类型有三个有趣的方面：首先，它们是强（strong）类型的；其次，它们是静态（static）的；第三，它们可以通过自动推导（automatically inferred）得出。

:i Integral Integer

函数应用的优先级比操作符要高

Haskell 有一个特殊的类型 () ，这种类型只有一个值 () ，它的作用相当于包含零个元素的元组，类似于 C 语言中的 void

Haskell 的函数应用是左关联的。

Prelude> lines "the quick**\n**brown fox**\n**jumps"

["the quick","brown fox","jumps"]

Side effect 副作用指的是，函数的行为受系统的全局状态所影响。

Tip：x:xs，这个xs是指复数个x

Haskell 是一门以表达式为主导（expression-oriented）的语言。在命令式语言中，代码由陈述（statement）而不是表达式组成，因此在省略 if 语句的 else 分支的情况下，程序仍是有意义的。但是，当代码由表达式组成时，一个缺少 else分支的 if 语句，在条件部分为 False 时，是没有办法给出一个结果的，当然这个 else 分支也不会有任何类型，因此，省略 else 分支对于 Haskell 是无意义的，编译器也不会允许这么做。

在使用严格求值的语言里，函数的参数总是在应用函数之前被求值。以 isOdd 为例子：子表达式 (1 + 2) 会首先被求值，得出结果 3 。接着，将 3 绑定到变量 n ，应用到函数 isOdd 。最后， mod 3 2 返回 1 ，而 1 == 1 返回 True 。

Haskell 使用了另外一种求值方式 —— 非严格求值。在这种情况下，求值 isOdd (1 + 2) 并不会即刻使得子表达式 1 + 2 被求值为 3 ，相反，编译器做出了一个“承诺”，说，“当真正有需要的时候，我有办法计算出 isOdd (1 + 2) 的值”。 非严格求值通常也被称为惰性求值。

用于追踪未求值表达式的记录被称为块（thunk）。这就是事情发生的经过：编译器通过创建块来延迟表达式的求值，直到这个表达式的值真正被需要为止。如果某个表达式的值不被需要，那么从始至终，这个表达式都不会被求值。

可以通过代换（substitution）和重写（rewriting）去了解 Haskell 求值表达式的方式。

惰性求值可以延迟计算直到真正需要一个值为止，并且在求值时，也只执行可以确立出（establish）值的那部分表达式。[译注：比如之前提到的， (||) 的左操作对象的值为 True 时，就无需对右操作对象估值的情况。]

函数的返回值可能是一个块（一个被延迟计算的表达式）。

这种类型的多态被称为参数多态。可以用一个类比来帮助理解这个名字：就像函数的参数可以被绑定到一个实际的值一样，Haskell 的类型也可以带有参数，并且在稍后可以将这些参数绑定到其它实际的类型上。

参数多态是 Haskell 支持的多态中最明显的一个。Haskell 的参数多态直接影响了 Java 和 C# 等语言的泛型（generic）功能的设计。Java 泛型中的类型变量和 Haskell 的参数化类型非常相似。而 C++ 的模板也和参数多态相去不远。

很少有语言像 Haskell 那样，默认使用纯函数。这个选择不仅意义深远，而且至关重要。

因为纯函数的值只取决于输入的参数，所以通常只要看看函数的名字，还有它的类型签名，就能大概知道函数是干什么用的。纯度减轻了理解一个函数所需的工作量。一个纯函数的行为并不取决于全局变量、数据库的内容或者网络连接状态。纯代码（pure code）从一开始就是模块化的：每个函数都是自包容的，并且都带有定义良好的接口。

将纯函数作为默认的另一个不太明显的好处是，它使得与不纯代码之间的交互变得简单。一种常见的 Haskell 风格就是，将带有副作用的代码和不带副作用的代码分开处理。在这种情况下，不纯函数需要尽可能地简单，而复杂的任务则交给纯函数去做。

## 第三章：定义类型并简化函数

*-- file: ch03/BookStore.hs*

**data** BookInfo **=** Book Int String [String]

**deriving** (Show)

当一个类型拥有一个以上的值构造器时，这些值构造器通常被称为“备选”（alternatives）或“分支”（case）。同一类型的所有备选，创建出的的值的类型都是相同的。

*-- file: ch03/BookStore.hs*

**type** CardHolder **=** String

**type** CardNumber **=** String

**type** Address **=** [String]

**data** BillingInfo **=** CreditCard CardNumber CardHolder Address

| CashOnDelivery

| Invoice CustomerID

**deriving** (Show)

这个程序提供了三种付款的方式。如果使用信用卡付款，就要使用 CreditCard 作为值构造器，并输入信用卡卡号、信用卡持有人和地址作为参数。如果即时支付现金，就不用接受任何参数。最后，可以通过货到付款的方式来收款，在这种情况下，只需要填写客户的 ID 就可以了。

**Prelude>** :t (1.0,2.0)

(1.0,2.0) :: (Fractional t, Fractional t1) => (t1, t)

当对 Book 构造器进行模式匹配时，我们逆转（reverse）它的构造过程：首先，检查这个值是否由 Book 构造器生成 —— 如果是的话，那么就对这个值进行探查（inspect），并取出创建这个值时，提供给构造器的各个值。

*-- file: ch03/BookStore.hs*

nicerID (Book id **\_** **\_** ) **=** id

nicerTitle (Book **\_** title **\_** ) **=** title

nicerAuthors (Book **\_** **\_** authors) **=** authors

我们把这种代码叫做“样板代码（boilerplate code）”：尽管是必需的，但是又长又烦。Haskell 程序员不喜欢样板代码。幸运的是，语言的设计者提供了避免这个问题的方法：我们在定义一种数据类型的同时，就可以定义好每个成分的访问器。（逗号的位置是一个风格问题，如果你喜欢的话，也可以把它放在每行的最后。）

*-- file: ch03/BookStore.hs*

**data** Customer **=** Customer {

customerID **::** CustomerID

, customerName **::** String

, customerAddress **::** Address

} **deriving** (Show)

*-- file: ch03/NestedLets.hs*

quux a **=** **let** a **=** "foo"

**in** a ++ "eek!"

在函数的内部，由于 let-绑定的变量名 a 屏蔽了函数的参数，使得参数 a 没有起到任何作用，因此该参数可以是任何类型的。

titysecond :: [a]->Maybe a

titysecond []= Nothing

titysecond (\_:x:xs) = Just x

*-- file: ch03/letwhere.hs*

foo **=** x

**where** x **=** y

**where** y **=** 2

ghci>foo

2

## 第四章：函数式编程

*-- file: ch04/SplitLines.hs*

splitLines [] **=** []

splitLines cs **=**

**let** (pre, suf) **=** break isLineTerminator cs

**in** pre : **case** suf **of**

('**\r**':'**\n**':rest) **->** splitLines rest

('**\r**':rest) **->** splitLines rest

('**\n**':rest) **->** splitLines rest

**\_** **->** []

isLineTerminator c **=** c == '**\r**' || c == '**\n**'

如果一个函数只为合法输入的一个子集定义了返回值（函数返回的调用过程中产生的错误不属于返回值），这样的函数称作 部分函数（partial function）。类似的，对于整个输入域都能返回一个合法结果的函数，我们称之为 全函数（total function）。[译注：全函数是部分函数的特殊形式]。

ghci> :**type** span

span **::** (a **->** Bool) **->** [a] **->** ([a], [a])

ghci> span even [2,4,6,7,9,10,11]

([2,4,6],[7,9,10,11])

ghci> :**type** break

break **::** (a **->** Bool) **->** [a] **->** ([a], [a])

ghci> break even [1,3,5,6,8,9,10]

([1,3,5],[6,8,9,10])

takeWhile和dropWhile函数带着谓词：takeWhile从开头遍历一个列表，抽取使谓词返回True的元素组成一个新列表；dropWhile则是把使谓词返回True的元素丢掉。（译注：这里的表述容易引起歧义，实际上两个函数都是走到第一个使谓词返回False的元素处就停止操作了，即使这个元素后面还有使谓词返回True的元素，两个函数也不再take或drop了）

ghci> :**type** takeWhile

takeWhile **::** (a **->** Bool) **->** [a] **->** [a]

ghci> takeWhile odd [1,3,5,6,8,9,11]

[1,3,5]

ghci> :**type** dropWhile

dropWhile **::** (a **->** Bool) **->** [a] **->** [a]

ghci> dropWhile even [2,4,6,7,9,10,12]

[7,9,10,12]

ghci> :**type** zip

zip **::** [a] **->** [b] **->** [(a, b)]

ghci> zip [12,72,93] "zippity"

[(12,'z'),(72,'i'),(93,'p')]

ghci> :**type** zipWith

zipWith **::** (a **->** b **->** c) **->** [a] **->** [b] **->** [c]

ghci> zipWith (+) [1,2,3] [4,5,6]

[5,7,9]

ghci> lines "foo**\n**bar"

["foo","bar"]

ghci> unlines ["foo", "bar"]

"foo**\n**bar**\n**"

ghci> words "the **\r** quick **\t** brown**\n\n\n**fox"

["the","quick","brown","fox"]

ghci> unwords ["jumps", "over", "the", "lazy", "dog"]

"jumps over the lazy dog"

以下是 Adler-32 算法的 Haskell 实现：

*-- file: ch04/Adler32.hs*

**import** **Data.Char** (ord)

**import** **Data.Bits** (shiftL, (.&.), (.|.))

base **=** 65521

adler32 xs **=** helper 1 0 xs

**where** helper a b (x:xs) **=** **let** a' **=** (a + (ord x .&. 0xff)) `mod` base

b' **=** (a' + b) `mod` base

**in** helper a' b' xs

helper a b [] **=** (b `shiftL` 16) .|. a

*-- file: ch04/append.hs*

append **::** [a] **->** [a] **->** [a]

append xs ys **=** foldr (:) ys xs

**可以通过观察括号的包围方式，以及累积器初始化值摆放的位置，来区分 foldl 和 foldr ：foldl 将处累积器初始化值放在左边，括号也是从左边开始包围。另一方面，foldr 将累积器初始化值放在右边，而括号也是从右边开始包围。**

综上所述，最好不要在实际代码中使用 foldl ：即使计算不失败，它的效率也好不到那里去。更好的办法是，使用 Data.List里面的 foldl' 来代替。

显示递归都可以用foldl或是foldr来做

loop 函数是尾递归函数的一个例子：如果输入非空，这个函数做的最后一件事，就是递归地调用自身。

所有可以用 foldr 定义的函数，统称为主递归（primitive recursive）。很大一部分列表处理函数都是主递归函数。比如说， map 就可以用 foldr 定义：

asIntfold :: String->Int

-- asIntfoldr xs = foldr ((+).(10\*).digitToInt) xs [] //这个方法是不行的

asIntfold xs = foldl (\acc x ->acc\*10 + digitToInt(x) ) 0 xs

 foldl' 就是左折叠的严格版本，它使用特殊的 seq 函数来绕过 Haskell 默认的非严格求值。

## 第五章：编写JSON库

JSON （JavaScript 对象符号）是一种小型、表示简单、便于存储和发送的语言。它通常用于从 web 服务向基于浏览器的 JavaScript 程序传送数据。JSON 的格式由 www.json.org 描述，而细节由 RFC 4627 补充。

JSON 支持四种基本类型值：字符串、数字、布尔值和一个特殊值， null 。

"a string"

12345

**true**

**null**

*-- file: ch05/SimpleJSON.hs*

**data** JValue **=** JString String

| JNumber Double

| JBool Bool

| JNull

| JObject [(String, JValue)]

| JArray [JValue]

**deriving** (Eq, Ord, Show)

[译注：这里的 JObject [(String, JValue)] 不能改为 JObject [(JString, JValue)] ，因为值构造器里面声明的是类构造器，不能是值构造器。

**module** **Main** (main) **where**

**import** **SimpleJSON**

main **=** print (JObject [("foo", JNumber 1), ("bar", JBool False)])

Main.hs 的名字和 main 函数的命名是有特别含义的，要创建一个可执行文件， ghc 需要一个命名为 Main 的模块，并且这个模块里面还要有一个 main 函数，而 main 函数在程序执行时会被调用。