Preface

Have We Got a Deal for You!

Haskell 是一门很有深度的语言；我们认为学习 **Haskell** 是一种极有价值的体验。我们主要关注三个方面。**第一是新颖：我们鼓励（invite）你从一种不同的而且有价值的观点思考编程。**第二是性能：我们将向你展示如何编写简短，快速并且安全的软件。最后，我们给你准备了许多乐趣：应用漂亮的编程技术去解决真实问题。

Novelty

Haskell 或许与你以前使用过的语言有相当大的不同。相对于通常的概念，函数编程给我们提供了一种极为不同的思考软件的方法。

Haskell 中，我们将那些会修改数据的代码的关注减到最小。相反地，我们将焦点转移到这样的函数上来：它们以不可修改的值作为输入，并且产生新的值作为输出。相同的输入总是产生相同的输出，这是函数式编程的核心思想。

Haskell 函数除了取得输入与返回输出外不与外部世界交互；我们称这样的函数为纯函数。纯代码和那些读写文件、进行网络通信，或移动机器手臂的代码之间有巨大的不同。这使得组织，理解，测试我们的程序变得容易。

我们将抛弃那些可能看起来很基础的 ideas，例如语言内置的 for 循环。我们有其它更灵活的方法来执行重复性的任务。

甚至在 Haskell 中表达式赋值的方式也是不同的。我们**推迟(defer)所有计算**，直到确确实实真的需要结果时我们才去计算表达式的值——**Haskell 是惰性语言，这一点将深刻地影响我们如何写程序。**

Power

贯穿全书地，我们将向你展示 Haskell 中那些对应于传统语言特性的替代品是如何产生(lead to)高效，灵活，可靠代码。**Haskell 充满了如何创建卓越软件的顶尖 ideas。**

因为纯代码除了取得输入与返回输出外不与外部世界交互，并且永远不会修改自己处理的数据，**所以那种以不可见的方式损坏数据的惊人代码是非常罕见的。**在我们使用纯函数的上下文中，函数的行为是始终如一的。

纯代码比那些与外界交互的代码更容易测试。

Lazy evaluation 有其“怪异”之处。假设我们要从一个列表中找出k 个最小元素，传统语言中显而易见的做法是将列表整个排序，然后选出前K 个元素，但是这样做代价较高。高效的做法是，我们写一个特殊的函数，在一次传参操作中这个函数获得全部这些值，然后此函数会执行一些稍复杂的bookkeeping 操作。Haskell 中用排序然后取元素的方法实际上工作得很好：**laziness 保证仅将列表排序到足够找出K 个最小元素的程度，不做多余的排序。**

从一个 List 中找出 k 个最小元素的小巧高效方法

```
-- file: ch00/KMinima.hs
-- lines beginning with "--" are comments.
minima k xs = take k (sort xs)
```

这段代码使用了标准库。

Enjoyment

因为 Haskell 与传统语言有极大的不同，所以要掌握这门语言本身和函数编程技术需要大量思考和练习。

What to Expect from This Book

使用 **QuickCheck** 库生成自动测试用例

What to Expect from Haskell

鼓励使用 **pure，lazy** 风格，但不是唯一选择。Haskell 也支持传统语言的一些特性。

Compared to Traditional Static Languages

有时，Haskell 程序比类似的 C/C++ 程序要慢，Haskell 在生产力和可靠性上有大量优点，这些优点比起不大的性能损失重要得多。

多核处理器已经普及，但是其基于传统编程语言仍然难以编程。

Haskell 拥有独一无二的技术使得多核程序易于驾驭，它支持并行编程，提供了 software transactional memory 支持灵活并发。

Compared to Modern Dynamic Languages

过去十年，动态类型，解释性语言变得越来越流行，使开发者在生产力上获得了实质上的利益。尽管这种获益常伴随着巨大的性

能开销，但是在许多任务中性能因素的影响是不成问题的。

Hakell 和动态类型语言的相似之处是简短：解决样的问题我们可以写比传统语言更短的代码。**通常 haskell 和动态语言的 size 是一样的**。

Haskell 几乎总是拥有巨大的运行时性能上的优势。

GHC(Glasgow Haskell Compiler) 也有一个解释器，可以不用编译的运行脚本。

当然，我们，作者们，我们喜欢动态语言，仍然每天使用动态语言，但是我们更喜欢 **Haskell**。

Chapter 1. Getting Started

Your Haskell Environment

GHC 有三个主要 components:

ghc

产生 native code 的编译器

ghci

交互式解释器及调试器

runghc

将 Haskell 程序作为脚本运行而无需编译

Getting Started with ghci, the Interpreter

“Prelude”提示符表示标准库已经加载并可以使用了。

打印帮助的命令“:?”

改变提示符

:set prompt "ghci> "

加载模块

:module + Data.Ratio

现在可以使用比例数了（或称有理数）

Basic Interaction: Using ghci as a Calculator

ghci 可以做为一个桌面计算器

Simple Arithmetic

infix form

```
ghci> 2 + 2
4
ghci> 31337 * 101
3165037
ghci> 7.0 / 2.0
3.5
```

Prefix form 必须确保操作符在括号里面

```
ghci> 2 + 2
4
ghci> (+) 2 2
4
```

Haskell 支持整数和浮点数，整数可以任意大

```
ghci> 313 ^ 15
27112218957718876716220410905036741257
```

An Arithmetic Quirk: Writing Negative Numbers

负数通常要用括号“()”括起来

ghci> -3
-3

这个例子中我们写的并不是“-3”这个数，而是一个 operator – 操作符作用于数字 3。“-”操作符是 Haskell 中唯一的一个 **unary operator**,不可以将它与 **infix operators** 混在一起：

ghci> 2 + -3 --> 发生错误

正确的写法：

ghci> **2 + (-3)**
-1
ghci> **3 + (-(13 * 37))**
-478

Haskell 将“*-”视为单个操作符

ghci> 2*-3 -->出错

应该这样写：ghci> 2*(-3)

对负数的这些处理似乎是让人讨厌的，但这是因为 **Haskell** 允许在任何时侯创建新的操作符。

Boolean Logic, Operators, and Value Comparisons

Haskell 的布尔值是True 和 False。 “&&”是“and”， “||”是“or”

```
ghci> True && False
False
ghci> False || True
True
```

Haskell 中零不是 false，非零不是 true

等于：“==”
小于“<”
大于等于“>=”

不等于运算符“/=”

```
ghci> 2 /= 3
True
```

逻辑非运算符“not”

```
ghci> not True
False
```

Operator Precedence and Associativity

查看运算符的优先级

:info (+) -->infixl 6 +

infixl 6 + 表示优级级是 6，优先级中 1 最小，9 最大。

Infixl 表示运行符是左结合的，infixr 表示运算符是右结合的

:info (^) -->infixr 8 ^

Undefined Values, and Introducing Variables

预加载的标准库只定了 Pi 这一个常量值。

let 语法只是 ghci 支持的，不是普通 Haskell 程序支持的

let e = exp 1 --> 定义欧位常数

“^”和“**”两个乘方运算符的区别

“^”只能以整数作为指数，而“”，能以浮点数作为指数。**

(e ** pi) – pi -->19.99909997918947

Dealing with Precedence and Associativity Rules

表达式中尽量多用括号。

Command-Line Editing in ghci

略

Lists

[1,2] 是 (1:(2:[])) 的简略写法

list 列表的写法

[1, 2, 3] -- 注意了，未尾不能有多余的逗号

列表可以任意长

[] –空列表
["foo", "bar", "baz", "quux", "fnord", "xyzzy"]

列表的元素必须有相同的类型

列表的补齐运算符“..”(enumeration notation)

[1..10] -->[1,2,3,4,5,6,7,8,9,10]

这个符号对文本类型无效

先写前两项 + “..” + 最后一项会按前面的“增量”补齐列表

[1.0,1.25..2.0] --> [1.0,1.25,1.5,1.75,2.0]
[1,4..15] --> [1,4,7,10,13]
[10,9..1] --> [10,9,8,7,6,5,4,3,2,1]

Operators on Lists

连接两个列表的运算符“++”

[3,1,3] ++ [3,7] --> [3,1,3,3,7]
[] ++ [False,True] ++ [True] --> [False,True,True]

在表的前端添加一个元素的运算符“:”

1 : [2,3] --> [1,2,3]
1 : [] --> [1]

“:”的前面必须是一个元素，后面必须是一个列表, [1,2]:3 这种写法是错误的。

Strings and Characters

Haskell 中，**string** 是字符的列表。

"This is a string."

打印字符串的函数 **putStrLn**

putStrLn "Here's a newline -->\n<-- See?"
--> Here's a newline -->
--> <-- See?

‘a’ 是一个 char，”a” 是一个 list

string 只是字符的列表

let a = [‘h’,’i’]
a == “hi” --> true

“””” 是“[]”的另一种写法

"" == [] -->True

list 的运算符全都可以用于 string

'a':"bc" --> "abc"
"foo" ++ "bar" --> "foobar"

注意：”a” 是列表，’a’ 只是一个 char。

First Steps with Types

整数(Integer) 是可以任意大的，只和系统的内存大小有关

类型名以大写字母开头，变量名以小写字母开头

:set 命令可以改变 ghci 的默入行为

用“type”命令打印类型信息

:type 'a'

让 ghci 显示类型信息

:set +t -- 这是逆命令:unset +t
'c'
--> 'c'

--> **it :: Char -- 其中的 it 就代表前面的变量，而且**可以在后面使用。**(只对 ghci 有效)**

“xx :: type”表示 xx 拥有类型 type

整数(Integer) 是可以任意大的，只和系统的内存大小有关。

比例数的写法“a%b”

:m +Data.Ratio -- **:m 是:module 的缩写**
11 % 29

A Simple Program

打印文本文件的行数

```
-- a.hs
main = interact wordCount
       where wordCount input = show (length (lines input)) ++ "\n"

-- a.txt
a small program
that counts the number of lines
in its input.

runghc a <a.txt
--> 3
```

注意：代码中的缩进是必须的。

Chapter 2. Types and Functions

Why Care About Types?

略

Haskell’s Type System

Haskell 的类型是**静态，强类型，能够自动推断**。

Static Types

静态类型的意思是编译器在编译时知道所有值和表达式的类型。

静态强类型保证 **Haskell 在运行时不可能出现类型错误**。

What to Expect from the Type System

略

Some Common Basic Types

Char 值
表示 Unicode 字符

Bool 值
True, False

Int 类型

用于有符号，固定宽度值。宽度决定于机器字长，如 32 位，64 位等。**Haskell 保证 Int 不小于 28 bits**。

Integer 值

Integer 不如 Int 常用，因为性能和空间开销大。**Integer 不会不声不响的溢出**。

Double

double 通常是 64 位，**不推荐使用 float，因为 float 性能不高**。

定义变量类型的运算符“::”

```
'a' :: Char
:type 'a' --> 'a' :: Char
```

格式是：变量+“::”+ 类型签名

Function Application

调用函数的方法

odd 3 --> True
odd 6 --> False

compare 2 3 --> LT
compare 3 3 --> EQ
compare 3 2 --> GT

(compare 2 3) == LT --> True
compare (sqrt 3) (sqrt 6) --> LT

Useful Composite Data Types: Lists and Tuples

tuple 是固定大小，可以装不同类型的元素。

tuple 一般只用来存不多的几个元素。

tuple 通常作为返回多个值的函数的返回值。

tuple 的写法

(1964, "Labyrinths")

使用 **head** 函数获取 **list** 的第一个元素

head [1,2,3,4] --> [2,3,4]

使用 **tail** 函数获取 **list** 的第一个元素后面的所有元素

tail [True,False] --> [False]

“[a]”表示类型为 a 的列表

list 是多态的，可以存放任意类型的值，**[a]** 中的 **a** 是一个占位符

:type [[True],[False,False]] --> [[True],[False,False]] :: [[Bool]]

Functions over Lists and Tuples

take 函数获取 list 的前 n 个元素

take 2 [1,2,3,4,5] --> [1,2]

drop 函数移除 list 的前 n 个元素

drop 3 [1,2,3,4,5] --> [4,5]

获取 tuple(pair tuple) 的第一或第二个元素

fst (1,'a')
snd (1,'a')

注意，这两函数只对含两个元素的 tuple 有效

Function Types and Purity

使用 lines 函数分割串

lines "the quick\nbrown fox\njumps"
--> ["the quick","brown fox","jumps"]

没有 side effects 的函数是纯函数，有 side effects 的是非纯函数。

有 **side effects** 的函数的签名带有 **IO** 标记

```
:type readFile
--> readFile :: FilePath -> IO String
```

Haskell 的类型系统会阻止混用纯和非纯函数。

Haskell Source Files, and Writing Simple Functions

ghci 的操作环境称为 IO monad

定义函数的方法

```
-- file: ch03/add.hs
add a b = a + b
```

```
:load c:\add.hs
add 1 2 --> 3
```

用 cd 命令改变 ghci 的工作目录

```
:cd c:\
:load add.hs
add 1 2 --> 3
```

Just What Is a Variable, Anyway?

变量一旦赋值就不能再改变

```
x = 10
x = 11    --  错误
```

变量是表达式的替代品

Conditional Evaluation

使用函数 **null** 来测试 **List** 列表是否为空

条件语句

```
-- myDrop.hs
myDrop n xs = if n <= 0 || null xs
                then xs
                else myDrop (n - 1) (tail xs)

:load myDrop.hs
myDrop 2 "foobar"
"obar"
```

“xs”中的“s”表示复数字尾。参数 n 决定了递归调用的次数，既**将一个列表 tail 多少次**。

逻辑或运算符“||”不是语言内置的，只是一个普通函数

Understanding Evaluation by Example

Lazy Evaluation

测试一个数是不是偶数

```
isOdd n = mod n 2 == 1
isOdd (1 + 2)
```

传参的时侯 (1+2) 不会先计算出结果 3 然后再传给 isOdd 函数。

发生的事情是：创建一个 **thunk (表达式的引用)** 传给函数，并延迟计算表达式的结果。如果一直用不到那个结果就永运不会计算它。

Polymorphism in Haskell

返回列表的最后一个元素

```
last [1,2,3,4,5] --> 5
```

The Type of a Function of More Than One Argument

“->” 的读法

```
:type take
take :: Int -> [a] -> [a]
```

“take :: Int -> [a] -> [a]”是右结合的，相当于**“take :: Int -> ([a] -> [a])”**，接受一个参数 **int**，返回另一个函数。

“->”的前面是函数的参数，后面是函数的返回值。

定义函数的类型签名

```
-- add.hs
add a = a + 1
add :: Int -> Int
```

Chapter 3. Defining Types, Streamlining Functions

使用“data”自定义新类型

```
-- book.hs
data BookInfo = Book Int String [String]
              deriving (Show)

mybook = Book 9780135072455 "Algebra of Programming"
               ["Richard Bird", "Oege de Moor"]
---------------
:load book
mybook
--> Book 9780135072455 "Algebra of Programming" ["Richard Bird","Oege de Moor"]
:type mybook
--> mybook :: BookInfo
```

“BookInfo”是**类型构造器(type constructor)**，类型名必须以大写字母开头。

“Book”是**值构造器(value constructor)**，用于创建类型为BookInfo 的值。值构造器也必须以大写字母开头。

“Int”，“String”，“[String]”是 **slot，值就装在这些 slot 里面。**

在 ghci 中构造 Book

```
ghci> Book 0 "The Book of Imaginary Beings" ["Jorge Luis Borges"]
ghci> let cities = Book 173 "Use of Weapons" ["Iain M. Banks"]
```

“:info”命令显示详细类型信息

:info BookInfo

Naming Types and Values

类型名和类型值是彼此独立的。

类型构造器只用于类型声明或类型签名。值构造器只用于真实代码。

类型构造器和值构造器可以有相同的名字

data **BookReview** = **BookReview** BookInfo CustomerID String

Type Synonyms

定义类型别名的方法

type B = Int
data A = A BookInfo Int B
type C = (A, B)

这里 C 的类型是 tupe(A,B)，这些类型别名没有值构造器

Algebraic Data Types

所有使用 **Data** 关键字定义的类型都是代数类型

Bool 是最简单的代数数据类型

代数数据类型可以有多个值构造器

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

Bool 类型有两个值构造器，Ture 和 False。

代数类型的值构造器可以接受零个或多个参数。

```
type CardHolder = String
type CardNumber = String
type Address = [String]

data BillingInfo = CreditCard CardNumber CardHolder Address
                 | CashOnDelivery
                 | Invoice CustomerID
                 deriving (Show)
```

BillingInfo 有三个值构造器，而且各自接受的参数也不同。

Tuples, Algebraic Data Types, and When to Use Each

有相同类型签名的 **pairs tuple** 具有相同的类型

```
a = ("Porpoise", "Grey")
b = ("Table", "Oak")
```

“==”操作符要求两边有相同的类型

Analogues to Algebraic Data Types in Other Languages

The enumeration

用代数类型实现枚举

```
data Roygbiv = Red
             | Orange
             | Yellow
             | Green
             | Blue
             | Indigo
             | Violet
             deriving (Eq, Show)

Red == Yellow --> False
Green == Green --> True
```

The discriminated union

用代数类型实现联合(union)

```
type Vector = (Double, Double)
data Shape = Circle Vector Double
           | Poly [Vector]
```

如果使用 Circle 构造器，我们实际上创建了并存储了一个 Circle 值，相反用 Poly ，就存 Poly 值。

Pattern Matching

假设我们得到某类型的一些值：

1. 如果此类型有多个值构造器，我们应能够知道这个值是由哪个构造器构造出来的。

2. 如果值构造器拥有数据域**(slot)**，我们应能够取出这些数据值。

Haskell 有一个简单但极有用的模式匹配机制，可以完成这两个任务。

实现 **Not** 函数

```
myNot True = False
myNot False = True
```

myNot True -->Fale
myNot True -->Ture

看起来像是定义了两个名为 myNot 的函数，但 **Haskell** 允许将函数定义为一系列的等式。这样同一个函数对于不同的输入模式具有不同的行为。

排在前面的函数等式优先匹配。

列表求和（后面有改进版）

```
sumList (x:xs) = x + sumList xs
sumList [] = 0

sumList [1,2]   --> 3
```

（运算符“:” 在表的前端添加一个元素，前面必须是一个元素，后面必须是一个列表）**[1,2] 是 (1:(2:[])) 的简略写法**。但函数参数列表**(x:xs)** 中的“：”是表示匹配，结果是 **1 匹配 x，2:[] 匹配 xs。函数的参数匹配就把 list 拆成两部分，part1 + 递归调用 part2。**注意函数等式的排列顺序，排在前面的优先匹配。

最后，**标准函数 sum** 与上面我们定义的函数有同样的功能。

Construction and Deconstruction

不要被 deconstruction 迷惑，模式匹配不解构任何东西，只是让我们“look inside”某个东西。

Further Adventures

返回 3-tuple 的第三个值的函数

```
third (a, b, c) = c
```

一个稍复杂的参数匹配例子

```
complicated (True, a, x:xs, 5) = (a, xs)

complicated (True, 1, [1,2,3], 5)    -- > (1,[2,3])
complicated (False, 1, [1,2,3], 5)    --  参数匹配失败


data BookInfo = Book Int String [String]
            deriving (Show)
bookID (Book id title authors) = id

bookID (Book 1 “tt” [“dsa”]) --> 1


m3list (3:xs) = 3 + m3list xs    --  只匹配全 3 为元素的 list，不是全 3 会引起运行时错误
m3list [] = 0
m3list [3,3,3]    --> 9
```

The Wild Card Pattern

“_”称为 wild card

模式中的任意值“_”

```
data BookInfo = Book Int String [String]
              deriving (Show)
bookID (Book id _    _) = id
```

bookID (Book 1 “tt” [“dsa”]) --> 1

Exhaustive Patterns and Wild Cards

默认匹配

```
goodExample (x:xs) = x + goodExample xs
```

goodExample _ = 0

goodExample [1,2] -- > 3

Record Syntax

nicerID (Book id _ _) = id 这种代码称为 **boilerplate**，既 bulky 又 irksome。

取出记录里的字段值

```
type CustomerID = Int
type Address = [String]

data Customer = Customer {
      customerID :: CustomerID,    -- customerID 是一个函数，输入Customer返回CustomerID
      customerName :: String,
      customerAddress :: Address
} deriving (Show)

let aa = Customer 1 “dfs” [“dfsa”]
customerID aa --> 1
```

记录函数(accessor functions)的另一种写法

```
data Customer = Customer Int String [String]
                    deriving (Show)
customerID :: Customer -> Int
customerID (Customer id _ _) = id
```

创建记录类型值的 **Detail** 风格

```
customer2 = Customer {
                customerID = 271828,
                customerAddress = ["1048576 Disk Drive",
                                   "Milpitas, CA 95134",
                                   "USA"],
                customerName = "Jane Q. Citizen"
}

data CalendarTime = CalendarTime {
                            ctYear :: Int,
                            ctDay, ctHour, ctMin, ctSec :: Int
}

ctime = CalendarTime {
          ctYear = 1,
          ctDay=3, ctHour=4, ctMin=5, ctSec=6
}
```

使用记录的 **detail** 创建风格可以改变记录的 **order**。这里 customerName 就和 customerAddress 的顺序就对调了。

使用记录的 **detail** 创建风格打印的时侯也会显示更多信息。

如果不使用记录语法，要从一个类型提取某个字段的值将是一件痛若的事。

Parameterized Types

Maybe a 类型

```
data Maybe a = Something a
             | Nonething
             deriving (Eq, Show)
Something 2 --> Something 2
:type Something  “string"
--> Something  “string" :: Main.Maybe [char]
Something  (Something  2)  -->  嵌套定义要加括号
```

Recursive Types

list 类型是递归定义的。

List a 类型

```
data List a = Cons a (List a)
            | Nil
              deriving (Show)
fromList (x:xs) = Cons x (fromList xs)   --  用 list  来构造 List a
fromList [] = Nil

Nil -> Nil
Cons 0 Nil -- > Cons 0 Nil
Cons 1 it -- > Cons 1 (Cons 0 Nil)
fromList "durian"
--> Cons 'd' (Cons 'u' (Cons 'r' (Cons 'i' (Cons 'a' (Cons 'n' Nil)))))
```

Cons 构造器需要两个参数，一个 a，一个 list a

二叉树

```
data Tree a = Node a (Tree a) (Tree a)
            | Empty
            deriving (Show)
Node 1 (Node 2 Empty Empty) (Node 3 Empty Empty)
simpleTree = Node "parent" (Node "left child" Empty Empty)
(Node "right child" Empty Empty)
```

Haskell 没有 Null 类型，可以用 **Maybe** 类型获得 **Null** 类型的效果，但是这样做需要模式匹配；替代方法是用无参的值构造器（如 **Nil**，**Empty**）。

Reporting Errors

error 函数输出错误消息，并立既终止程序。

error :: String -> a

因为返回类型是 a ，所以可以在任何地方调用，并返回正确的类型。

返回 **List** 列表的第二个元素，不够两个元素就报错（有问题版）

```
mySecond :: [a] -> a    -- 不写好像也可以自动推断出参数类型因为 tail 接受 list
mySecond xs = if null (tail xs)
                then error "list too short"
                else head (tail xs)

mySecond "ab"               --> 'b'
mySecond "a"                --> Exception: list too short
head (mySecond [[9]])       --> Exception: list too short
mySecond []   --> 参数匹配失败，要求[a]，这里是[]
```

错误处理函数 **error** 的缺点是不能区分对待可恢复错误和能终止程序的至命错误。

A More Controlled Approach

返回 **List** 列表的第二个元素，不够两个元素就报错（待改进）

```
safeSecond :: [a] -> Maybe a

safeSecond [] = Nothing
safeSecond xs = if null (tail xs)
                then Nothing
                else Just (head (tail xs))

safeSecond []             --> Nothing
safeSecond [1]            --> Nothing
safeSecond [1,2]     -->Just 2
safeSecond [1,2,3]  -->Just 2
```

返回 **List** 列表的第二个元素，不够两个元素就报错

```
tidySecond :: [a] -> Maybe a
tidySecond (_:x:_) = Just x    -- 亮点!
tidySecond _ = Nothing

tidySecond [1,2]     -->Just 2
```

模式“**_:x:_**”只匹配有两个元素以上的 **List** 列表。

Introducing Local Variables

let 关键字开始一个变量声明块，并以关键字 **in** 作为结束。

函数体内部使用 **let** 定义局部变量。

```
lend amount balance = let reserve = 100    -- 如果 ghci 报错就在行后加个分号“;”
                          newBalance = balance - amount
                      in if balance < reserve
                         then Nothing
                         else Just newBalance
```

let 的左边被捆绑到右边的表达式，注意 **let** 是与表达式而不是与值捆绑

如果我们能使用一个名字，它就是“**in scope**”，否则就是“**out of scope**”；如果一个名字在整个源文件可见，我们就说它位于“**top level**”

Shadowing

let 可以互相嵌套但不是聪明的做法

```
foo = let a = 1
          in let b = 2
              in a + b
print foo    -- >3
```

shadowing 变量(影子变量)

```
bar = let x = 1
          in ((let x = "foo" in x), x)
-- >("foo",1)
```

里面的 x，是外面 x 的 shadowing ，**shadowing** 变量是一个同名变量，类型和值都可以不同。

shadow 一个函数的参数

```
quux a = let a = "foo";
              in    a ++ "eek!"

type quux    -- t -> [Char]
```

因为函数的参数永远不会被用到，所以参数可以是任意类型 **t**

使用 **GHC** 的选项“**-fwarn-name-shadowing**”可以开启 **shadow** 警告，以避免发生莫明其妙的问题。

The where Clause

我们可以使用另一种机制**引入局部变量：where 子句**。

使用 where 子句推迟局部变量的定义

```
lend2 amount balance = if amount < reserve * 0.5
                       then Just newBalance
                       else Nothing
    where reserve = 100
          newBalance = balance – amount    -- 注意和上面对齐，不然会出错
```

where 子句有助于将读者的注意力集中在重要的位置，局部变量的值留到后面用 where 来定义。

Local Functions, Global Variables

定义局部函数和定义局部变量一样简单。

定义局部函数

```
pluralise :: String -> [Int] -> [String]
pluralise word counts = map plural counts
      where plural 0 = "no " ++ word ++ "s"
            plural 1 = "one " ++ word
            plural n = show n ++ " " ++ word ++ "s"
```

局部函数 plural 含有多个等式，并使用了外部函数 pluralise 的变量 word。

定义全局变量

```
itemName = "Weighted Companion Cube"
```

在源文件的 top level 定义既可

The Offside Rule and Whitespace in an Expression

Haskell 这种使用缩排的规则称为“**offside rule**”。

第一个使用 top-level 的声明或定义可以从任何列开始，然后 Haskell 编译器或解释器就会记住那个缩排层级数；**所有后面的 top-level 声明都必须和第一个 top-level 具有同样的缩进**。

```
bar = let b = 2
                c = True
          in let a = b
               in (a, c)
```

a 只对里面的 let 可见，外面的 let 不可见。

```
foo = x
      where x = y
                  where y = 2    -- 注意这个 where 的缩进
```

A Note About Tabs Versus Spaces

把编辑器调成用空格代替 Tab

The Offside Rule Is Not Mandatory

“offside rule”不是强制的

可以用大括号“{}”将 **equations** 括起来，并用分号“;”分隔里面的每项。

用大括号代替缩排

```
bar = let a = 1
          b = 2
          c = 3
      in a + b + c

foo = let { a = 1; b = 2;
          c = 3 }
      in a + b + c
```

The case Expression

函数定义不是唯一可以使用模式匹配的地方。“**case**”语句也**会进行模式匹配**。

case 语句

```
fromMaybe defval wrapped =
    case wrapped of
        Nothing ->      defval
        Just value ->   value
```

“-> ” 左边是模式，如被匹配右边就得以 evaluate

case 关键字后接一个任意的表达式，并且用这个表达式的值去匹配“**of**”后面的表达式。匹配的须序是自上往下。

“**of**”后面的所有表达式必须具有相同的类型。

wild card 表达式“**_**”可以放在 **case** 语句的最后用作默认匹配。

Common Beginner Mistakes with Patterns

以下是初学者对模式的常见误用

Incorrectly Matching Against a Variable

“of”后面应该写 case 对应的表达式的值

错误的版本

```
data Fruit = Apple | Orange
apple = "apple"
orange = "orange"

whichFruit :: String -> Fruit
whichFruit f = case f of
                    apple -> Apple
                    orange -> Orange
```

正确的版本

```
data Fruit = Apple | Orange    -- 亮点
betterFruit f = case f of
                      "apple"  →    Apple
                      "orange"  →    Orange
```

函数 betterFruit 的类型签名可以自动推断出来，因为输入与 String 型的"apple" 匹配，输出与 Fruit 类型的"Apple" 匹配。

Incorrectly Trying to Compare for Equality

模式里的名字只能出现一次

错误的例子

```
bad_nodesAreSame (Node a _ _) (Node a _ _) = Just a
bad_nodesAreSame _ _ = Nothing
```

解决这个问题要用到 **guards**

Conditional Evaluation with Guards

对函数的参数进行条件测试

```
-- Node 是 Tree 的值构造器；结点本身也是树，两者是一回事
data Tree a = Node a (Tree a) (Tree a)
            | Empty
            deriving (Show)

nodesAreSame (Node a _ _) (Node b _ _)
    | a == b = Just a
nodesAreSame _ _ = Nothing
```

“| a == b” 是参数模式的“guards”。一个模式可以有零个或多个**“guards”**。

“guards” 就是类型为 **bool** 的一个表达式。

“guards”使用符号“|” 引入

使用 **guards** 让代码变得更清晰

```
myDrop n xs = if n <= 0 || null xs
              then xs
              else myDrop (n - 1) (tail xs)

niceDrop n xs | n <= 0 = xs
niceDrop _ [] = []
niceDrop n (_:xs) = niceDrop (n - 1) xs
```

Chapter 4. Functional Programming

A Simple Command-Line Framework

Haskell 中所有函数都仅接受一个参数。如果一个函数需要多个参数，则每给它一个参数，它就返回一个 **partial** 函数。

组合使用库函数，如 **map**，**take**，**filter** 来代替尾部递归和匿名函数，可以使得代码更可读，加快编码速度，**bug** 更少。

能使用库函数组合（用“.”操作符），就不要使用 **fold**。用 **fold** 代替 **hand-rolled** 尾递归循环。

好的函数名就是一个 **tiny** 文档。

使用 **ghc** 在命令行中编译**.hs** 源文件

```
{-
-- in.txt
hello,world!
-}
```

```
-- InteractWith.hs
module Main where

import System.Environment (getArgs)
interactWith function inputFile outputFile = do
      input <- readFile inputFile
      writeFile outputFile (function input)

main = mainWith myFunction
      where mainWith function = do
                  args <- getArgs
                  case args of
                        [input,output] -> interactWith function input output
                        _ -> putStrLn "error: exactly two arguments needed"

            -- replace "id" with the name of our function below
            myFunction = id
```

这是一个完整的文件处理程序，

在命令行中运行
ghc --make InteractWith
InteractWith in.txt out.txt

运行程序后生成 out.txt，内容和 in.txt 相同。

程序中出现了一些新的符号，**“do” 关键字引入一个“action 块”，action 块会对现实世界造成某些影响**，如读或写一个文件。

“<-”在 do 里面是赋值运算符。

Warming Up: Portably Splitting Lines of Text

Haskell 内置的 **lines** 函数用来分割字串。

:type lines
lines :: String -> [String]
lines "line 1\nline 2" -- > ["line 1","line 2"]
lines "foo**n****n**bar\n" -- >["foo","","bar"]
lines "a**r****n**b" -- >["a**r**","b"]

Windows 中，**当以文本模式读一个文件，I/O 库会把"\r\n" 转成"\n"；如果写以一个文件行为正好相反**。Unix 系统文本模式不作任何转换。

实现可移值的行分割函数

```
splitLines :: String -> [String]
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'

splitLines "foo"    -- >["foo"]
break isLineTerminator "foo"    -- >("foo","")
splitLines "foo\r\nbar"    -- >["foo","bar"]
break isLineTerminator "foo\r\nbar"    -- > ("foo","\r\nbar")
"foo" : ["bar"]    -- > ["foo","bar"]
```

标准函数 **break** 用来将 **list** 分成两部分，接受一个返回 **bool** 值的函数作为第一个参数，一旦函数返回 **true** 就在那一点将 **list** 一分为二。

break 函数返回一个 **pair**

遇到奇数就将 **list** 一分为二

```
break odd [2,4,5,6,8]      -- >([2,4],[5,6,8])
```

遇到大写字母就将 **list** 一分为二

```
:module +Data.Char
break isUpper "isUpper" -- >("is","Upper")
```

A Line-Ending Conversion Program

文件转换 - Unix 文本转 Windows 文本

```
-- FixLines.hs

-- 编译运行 FixLines.hs 的方法
-- ghc --make FixLines
-- FixLines gpl-3.0.txt fixed-gpl-3.0.txt

-- gpl-3.0.txt
-- this file created by Unix system
-- http://www.gnu.org/licenses/gpl-3.0.txt

module Main where

import System.Environment (getArgs)

-- 文件处理函数
interactWith function inputFile outputFile = do
    input <- readFile inputFile
    writeFile outputFile (function input)

-- 可移值的行分割函数
splitLines :: String -> [String]
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'

fixLines :: String -> String
fixLines input = unlines (splitLines input)



main = mainWith myFunction
```

```
where mainWith function = do
        args <- getArgs
        case args of
            [input,output] -> interactWith function input output
            _ -> putStrLn "error: exactly two arguments needed"

    -- replace "id" with the name of our function below
    myFunction = fixLines
```

Infix Functions

中缀有助于提高代码可读性

中缀语法糖“``”

```
a `plus` b = a + b
data a `Pair` b = a `Pair` b
                    deriving (Show)
```

前缀和中缀构造都可以

```
True `Pair` "something"    -- >True `Pair` "something"
Pair True "something"        -- >True `Pair` "something"
```

前缀和中缀调用都可以

```
1 `plus` 2    -- >3
plus 1 2       -- >3
```

测试一个值是否是列表的一个元素

```
elem 'a' "camogie"    -- >True
3 `elem` [1,2,4,8]    -- >False
```

测试一个字串是否是另一个字串的前缀

```
"foo" `isPrefixOf` "foobar"    -- >True
```

使用 isPrefixOf 函数前用 import 命令导入 Data.List 模块

测试一个字串是否是另一个字串的中缀

```
"needle" `isInfixOf` "haystack full of needle thingies"    -- >True
```

测试一个字串是否是另一个字串的后缀

```
"end" `isSuffixOf` "the end"    -- >True
```

Working with Lists

列表作为函数编程的最佳搭档理应享受特殊待遇。标准库为处理 list 定义了一打函数，其中一些是实际编程中不可或缺的。

以下将会介绍数量众多的 list 函数。为什么要一次呈现如些之多的函数？因为它们易于学习，并且无处不在。如果我们不熟悉它们就会把时间浪费在编写标准库已经提供的函数上。

Data.List 是标准list 函数真实逻辑义意上的home，**Prelude** 仅仅导出了 **Data.List** 的一个小子集，而且其中一些很有用的函数并不包括在内。

ghci 中加载 Data.List 模块

```
:module +Data.List
```

Basic List Manipulation

head 和 last，tail 和 init 是两对效果相反的函数

length 函数取列表元素个数

```
length []    -- >0
length "strings are lists, too"    -- >22
```

null 函数检查一个列表是否为空

```
null []    -- >True
null "plugh"    -- >False
```

head 函数取列表的第一个元素

head [1,2,3] -- >1

last 函数取列表的最后一个元素

last "bar" -- >'r'

tail 函数取除掉 **head** 后剩余的元素列表

tail "foo" -- >"oo"

init 函数取除掉 **last** 后剩余的元素列表

init "bar" -- >"ba"

(注意，上面这些函数遇到空列表通常会出错，要小心处理。)

Safely and Sanely Working with Crashy Functions

使用 **lenght** 函数来判断列表是否为空不是好办法

```
myDumbExample xs = if length xs > 0
                   then head xs
                   else 'Z'
```

list 不存储自身的长度信息，所以要获取列表的长度唯一方法是遍历整个列表。因此当我们想知道一个列表是否为空时使用 lenght 函数进行判断不是好办法。**lenght** 遇到无限列表时就出现死循环了。

用 **null** 函数来判断一个列表是否为空是较好的选择，此函数会在常量时间返回。

应该使用 **null** 函数来判断一个列表是否为空

```
mySmartExample xs = if not (null xs)
                    then head xs
                    else 'Z'
-- 亮点
myOtherExample (x:_) = x
myOtherExample [] = 'Z'
```

Partial and Total Functions

形如 **head []** 这种“[]”输入合法却会引发错误的函数就称为“**partial functions**”，是不安全的函数。

Prelude 定义了一些不安全的“partial functions”，例如head 函数。

More Simple List Manipulations

“++” 运算符连接两个列表

```
"foo" ++ "bar"    -- >"foobar"
[True] ++ []    -- >[True]
```

concat 函数将两个列表的列表合并成单个列表

```
concat [[1,2,3], [4,5,6]]    -- >[1,2,3,4,5,6]
concat [[[1,2],[3]], [[4],[5],[6]]]    -- >[[1,2],[3],[4],[5],[6]]
concat (concat [[[1,2],[3]], [[4],[5],[6]]])    -- >[1,2,3,4,5,6]
```

调用一次 **concat** 就拆掉一层括号

reverse 函数将列表逆序

```
reverse "foo"    -- >"oof"
```

and, or 函数对一 list 的 bool 值进行逻辑测试

and [] -- >True
or [] -- >False
and [True,False,True] -- >False
or [False,False,False,True,False] -- >True

all, any 函数是 and, or 的非常有用的弟妹

用 all 函数判断 list 中元素是否全为奇数

all odd [] -- >True
all odd [1,3,5] -- >True

用 any 函数判断 list 中是否存在偶数

any even [] -- >False
any even [3,1,4,1,5,9,2,6,5] -- >True

空列表[] 中，全部是偶数(all)，并且不存在偶数(any)

Working with Sublists

take 函数取前 **n** 个元素的列表

take 2 [1] -- >[1]

drop 函数取除掉前 **n** 个元素后的列表

drop 3 "xyzzy" -- >"zy"

splitAt 函数在指定位置分割列表

splitAt 3 "foobar" -- >("foo","bar")

splitAt 组合了 **take** 和 **drop**，先 **take** 得到前面的部分，再 **drop** 得到后面的部分，然后组合成一个 **pair** 返回。

takeWhile 接受一个布尔函数和一个列表，返回那些使得布尔函数为真的元素列表

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

注意，一旦布尔函数返回 false 就不往下取了

dropWhile 接受一个布尔函数和一个列表，返回除去那些使得布尔函数为真的元素列表

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

span 函数在第一次使得布尔函数为假的位置分割列表

span even [2,4,6,7,9,10,12] -- >([2,4,6],[7,9,10,12])

span 组合了 **takeWhile** 和 **dropWhile**，先 **takeWhile** 得到前面的部分，再 **dropWhile** 得到

后面的部分，然后组合成一个 **pair** 返回。

break 函数在第一次使得布尔函数为真的位置分割列表

break even [1,3,5,6,8,9,10] -- >([1,3,5],[6,8,9,10])

break 组合了 takeWhile 和 dropWhile，先 takeWhile 得到前面的部分，再 dropWhile 得到后面的部分，然后组合成一个 pair 返回。

Searching Lists

elem 函数判断列表中是否存在某个元素

2 `elem` [5,3,2,1,1] -- >True

notElem 函数判断列表中是否不存在某个元素

2 `notElem` [5,3,2,1,1] -- >False

filter 函数返回所有使得布尔函数为真的元素列表

filter odd [2,4,1,3,6,8,5,7] -- >[1,3,5,7]

isPrefixOf, isInfixOf 和 **isSuffixOf** 函数分别用来判断一个列表是否是另一个列表的前缀中缀或后缀

"foo" `isPrefixOf` "foobar" -- >True

[2,6] `isInfixOf` [3,1,4,1,5,9,2,6,5,3,5,8,9,7,9] -- >True

".c" `isSuffixOf` "crashme.c" -- >True -- **亮点**

Working with Several Lists at Once

zip 函数将两个列表的元素一一对应并配成 **pair** 的列表其长度与最短的列表一至

```
zip [12,72,93] "zippity"    -- >[(12,'z'),(72,'i'),(93,'p')]
```

zipWith 函数接受一个操作函数并接受两个列表作为输入，其生成的列表长度与最短的列表一至

```
zipWith (+) [1,2,3] [4,5,6]    -- >[5,7,9]
```

zip3,zipWith3 到 **zip7,zipWith7** 函数是上面函数的参数加长版，能将更多的列表 **zip** 起来

```
zip3 [12,72,93] "zippity" [1, 2, 3] -- >[(12,'z',1),(72,'i',2),(93,'p',3)]
```

Special String-Handling Functions

lines 函数将含有换行符的字串处理成列表

lines "foo\nbar" -- >["foo","bar"]

unlines 函数将字串列表处理成含有换行符的列表

unlines ["foo", "bar"] -- >"foo\nbarλn"

words 函数提取字串的所有单词并生成列表

words "the \r quick \t brown\n\n\nfox" -- >["the","quick","brown","fox"]

unwords 函数将单词列表连成字串

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

How to Think About Loops

遍历处理 List 的关键是拆分(用模式匹配)，连接，终止。

和传统语言不同，Haskell 既没有 for 循环也没有 while 循环。

用什么来替代循环呢？这个问题有许多可能的答案。

Haskell 中用尾部递当来代替循环。

在命令式语言中，循环的执行花费**常量空间开销**。Haskell 的尾部递归中每次调用自已都要分配一些空间，让函数知道应该在哪里返回。所以要花费**线性空间开销**。但是，函数语言会检测尾部递归，并应用一种称为**“尾部调用优化”技术使得尾部递归变成常量空间开销**

Explicit Recursion

C 语言字符串转数字

```c
#include "stdio.h"

int as_int(char *str)
{
    int acc; /* accumulate the partial result */
    for (acc = 0; isdigit(*str); str++) {
        acc = acc * 10 + (*str - '0');
    }
    return acc;
}

int main()
{
    char *strdigit = "678413";
```

```
    int digit;

    digit = as_int(strdigit);
    printf("%d", digit);
}
```

如何用 Haskell 直观的实现字串转整数呢？

数字字串转整数

```
import Data.Char (digitToInt) -- we'll need ord shortly
asInt :: String -> Int
loop :: Int -> String -> Int
asInt xs = loop 0 xs
loop acc [] = acc
loop acc (x:xs) = let acc' = acc * 10 + digitToInt x
                  in loop acc' xs

asInt "33"    -- >33
asInt ""    -- >0
```

函数签名可以不写，但是它有助于提醒我们正在做的事情

acc 变量用于“**accumulate** the partial result”，不断的用列表值构器“:” 进行模式匹配将字串列表拆成两部分来控制循环。

acc' 在变量名后加“'” 表示与原变量差别不大的变量（“'” 发音为“prime”）。

字串 String 就是字符列表[Char]，可以这样考虑列表结构：它要不是空列表[] ，要不就是单个元素后接剩余的列表。

修复上面这段代码缺陷留作 97 页的练习。

因为 loop 函数最后做的一件事情是简单的调用自己，所以是一个尾部递归函数。

对列表分为空和非空两种种况进行处理这是一种称为结构递归(structural recursion)的方法。

structural recursion 作为一种有用的技术不限于只用在 list 上次，也可以用于其它代数数据类型上。

Transforming Every Piece of Input

计算列表的乘方

```
square :: [Double] -> [Double]
square (x:xs) = x*x : square xs
square [] = []

square [2,3,4] -- >[4.0,9.0,16.0]
```

小写字串转大写字串

```
import Data.Char (toUpper)
upperCase :: String -> String
upperCase (x:xs) = toUpper x : upperCase xs
upperCase [] = []

upperCase "hello,world!"    -- >"HELLO,WORLD!"
```

Mapping over a List

map 函数接受一个处理函数，并用它处理列表中的每一个元素

```
square2 xs = map squareOne xs
      where squareOne x = x * x

upperCase2 xs = map toUpper xs

map negate [1,2,3] -- >[-1,-2,-3]
```

因为 map **以另一个函数作为参数**，所以我们将其称为**“higher-order”函数**

map 函数的实现

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

myMap toUpper "hello,world!"    -- >"HELLO,WORLD!"
```

Selecting Pieces of Input

过滤掉列表中不是奇数的元素

```
oddList :: [Int] -> [Int]
oddList (x:xs) | odd x = x : oddList xs
               | otherwise = oddList xs
oddList _ = []

oddList [1,1,2,3,5,8,13,21,34]    -- >[1,1,3,5,13,21]
```

其中“|” 是“guard expresses”

用 **filter** 函数过滤掉列表中不是奇数的元素

```
filter odd [1,1,2,3,5,8,13,21,34]    -- >[1,1,3,5,13,21]
```

Computing One Answer over a Collection

Data.List 模块定义了一个名为 **foldl'** 的函数，与 **foldl** 不同的是它不会创建 **thunks**。

由于 **foldl** 的 **thunking** 行为，在实际编程中应避免使用这个函数，而应该用 **foldl'** 替代。让 Foldl 仅用于测试。

用 **foldl** 代劳尾部递归

Haskell 中 **fold** 函数极为常见，而且拥有规则，可预测的行为。相比使用显示递归，使用 **fold** 函数可以让代码更容易理解。

Foldr 是 **list** 编程工具箱的重要成员。

Foldr 可以增量式的消费和生产一个 **list** ，这对编写 **lazy data-processing** 非常有用。

对 **foldl** 我们只需要简单的考虑两件事：第一，**accumulator** 的初始值是什么。第二，如何更新 **accumulator(**“+” 函数**)**。这样我们的代码更短了，也易于理解。

另一种考虑 **foldr** 的方法：第一，它的前两个参数是“对列表的每一个 **head/tail** 做什么？”。第二，“要把列表的 **end** 替换成什

么？”

列表求和（后面有改进版）

```
mySum xs = helper 0 xs
      where helper acc (x:xs) = helper (acc + x) xs
                helper acc _ = acc
```

helper 是**尾部递归函数，因为它做的最后一做事情就是调用自已**

Adler-32 checksum 算法（后面有改进版）

```
--import Data.Char (ord)
--import Data.Bits (shiftL, (.&.), (.|.))
base = 65521
adler32_try2 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
```

shiftL 是逻辑左移函数，“**.&.**” 是按位与运算符，“**.|.**” 是按位或运算符。

The Left Fold

foldl 函数的实现

```
myfoldl :: (a -> b -> a) -> a -> [b] -> a
myfoldl step zero (x:xs) = myfoldl step (step zero x) xs
myfoldl _ zero [] = zero

{-
foldl (+) 0 (1:2:3:[])
== foldl (+) (0 + 1) (2:3:[])
== foldl (+) ((0 + 1) + 2) (3:[])
== foldl (+) (((0 + 1) + 2) + 3) []
== (((0 + 1) + 2) + 3)
-}
```

列表求和（后面有改进版）

```
foldlSum xs = foldl step 0 xs
      where step acc x = acc + x
```

列表求和（后面有改进版）

```
niceSum :: [Integer] -> Integer
niceSum xs = foldl (+) 0 xs
```

这里不再使用尾部递归，因为 **foldl** 代劳了。对 **foldl** 我们只需要简单的考虑两件事：第一，**accumulator** 的初始值是什么。第二，如何更新 **accumulator(“+” 函数)**。这样我们的代码更短了，也易于理解。

另一种考虑 **foldr** 的方法：第一，它的前两个参数是“对列表的每一个 **head/tail** 做什么？”。第二，“要把列表的 **end** 替换成什么？”

Adler-32 checksum 算法（后面有改进版）

```
adler32_foldl xs = let (a, b) = foldl step (1, 0) xs
                       in (b `shiftL` 16) .|. a
    where step (a, b) x = let a' = a + (ord x .&. 0xff)
                              in (a' `mod` base, (a' + b) `mod` base)
```

这里 accumulator 是一个 pair，所以 **foldl** 也返回 **pair**

Why Use Folds, Maps, and Filters?

前面的例子使用 fold 函数没让代码变短多少，为什么要用 fold？因为在 **Haskell** 中 **fold** 函数极为常见，而且拥有规则，可预测的行为。

相比使用显示递归，使用 **fold** 函数可以让代码更容易理解。

Folding from the Right

foldr 的实现

```
myfoldr :: (a → b → b) → b → [a] → b
myfoldr step zero (x:xs) = step x (myfoldr step zero xs)
myfoldr _ zero [] = zero

{-
foldr (+) 0 (1:2:3:[])
== 1 + foldr (+) 0 (2:3:[])
== 1 + (2 + foldr (+) 0 (3:[])
== 1 + (2 + (3 + foldr (+) 0 [])) ????
== 1 + (2 + (3 + 0))
-}
```

看起来似乎 foldr 比 foldl 的用处小，但如果用显示递归实现 filter 看起来像这样:

filter 实现（后面有改进版）

```
myfilter :: (a → Bool) → [a] → [a]
myfilter p [] = []
myfilter p (x:xs)
    | p x = x : myfilter p xs
    | otherwise = myfilter p xs
```

filter 实现

```
myFilter p xs = foldr step [] xs
    where step x ys | p x = x : ys
                    | otherwise = ys

myFilter odd [1,2,3,4]  -- >[1,3]

{-
step 1  (foldr step [] 2:3:4:[])
step 1  (step 2  (foldr step [] 3:4:[]))
step 1  (step 2  (step 3  (foldr step [] 4:[])))
step 1  (step 2  (step 3  (step 4  (foldr step [] [])))) ????
```

-}

这种定义初看起来让人头痛。

第一，accumulator 的初始值是[]。

第二，由 step 函数来更新 accumulator，如果一个元素让布尔函数为真就连入列表

map 的实现

```
myMap' :: (a → b) → [a] → [b]
myMap' f xs = foldr step [] xs
    where step x ys = f x : ys
```

实际上 foldl 可以用 foldr 实现

```
myFoldl :: (a → b → a) → a → [b] → a
myFoldl f z xs = foldr step id xs z
    where step x g a = g (f a x)
```

（这样看起来 foldr 比 foldl 更有用？）

复制一个列表

```
identity :: [a] -> [a]
identity xs = foldr (:) [] xs

identity [1,2,3]    -- >[1,2,3]

{-
1:(2:(3:[]))????
-}
```

identity 返回列表的一个拷贝

另一种考虑 **foldr** 的方法：第一，它的前两个参数是“对列表的每一个 **head/tail** 做什么？”。第二，“要把列表的 **end** 替换成什么？”

如果 **foldr** 将列表的 **end** 替换成其它的值，这提示我们对于 **Haskell** 的“++” 运算符的另一种看法。

连接两个列表就是把前一个的 **end** 替换成另一个

```
append :: [a] -> [a] -> [a]
append xs ys = foldr (:) ys xs

append [1,3,5] [2,4,6]    -- >[1,3,5,2,4,6]
```

Foldr 是 list 编程工具箱的重要成员。

Left Folds, Laziness, and Space Leaks

习惯上将 **foldl** 作为测试用，而且我们在实践中从不使用 **foldl**。

在需要结果前，foldl 会存储一个 thunk，而 thunk 比一个单值的开销要大。

ghc 为 thunk 准备的 stack 大小有一个最大值限制，超过这个大小就会引发错误。要感谢这种限制使得我们可以尝试非常大的 thunk 而不用但心消耗掉所有内存。

```
foldl (+) 0 [1..1000]    -- >500500
```

这里 **foldl** 将创建一个 **thunk**，这个 **thunk** 含有 **1000** 个整数，并且应用了 **999** 次“+” 函数。为了表示一个单值花费了太多内存和 effort。

```
foldl (+) 0 [1..100000]    -- >Exception: stack overflow（书上是这样说，实际上好像是没反应）
```

在小的 expressions 中，foldl 工作正常，但速度慢。我们这种把不可见的 **thunking** 称

为“**space leak**”，因为我们的代码操作正常，但是内存使用不在正常范围。

使用 foldl 带来的“space leak” 是 Haskell 新手的一个典型路障。幸运的是，这很容易避免。

Data.List 模块定义了一个名为 foldl' 的函数，与 foldl 不同的是它不会创建 thunks。

由于 foldl 的 thunking 行为，在实际编程中应避免使用这个函数，而应该用 foldl' 替代。

Further Reading

要进一步了解 fold ，请参见这编极好的文章：“A tutorial on the universality and expressiveness of fold” (http://www.cs.nott.ac.uk/~gmh/fold.pdf)

Anonymous (lambda) Functions

lambda 函数准则—— 保证 lambda 定义的模板不会失败。

匿名函数也称 lambda 函数

匿名函数以一个“\” 符号开始(backslash character，发音为 lambda)，后接参数(可以包括模板)，然后接“→ ”，最后是函数体。

匿名函数的定义

```
isInAny2 needle haystack = any (\s →  needle `isInfixOf` s) haystack
```

这里 lambda 函数包围在括号里面了，所以 Haskell 知道函数体在哪里结束。

lambda 函数有几个非常重要的限制 —— 普通函数可以使用多个子句定义多个不同的模板和 guards ，而 **lambda 函数只能定义单个语句**。

lambda 函数只能定义单个语句

普通函数

```
safeHead (x:_) = Just x
safeHead _ = Nothing
```

lambda 函数

```
unsafeHead = \(x:_) →  x

unsafeHead [] -- 引发运行时错误
```

lambda 函数准则—— 保证 **lambda** 定义的模板不会失败。

Partial Function Application and Currying

“->” 只有一个意思，左边标记参数的类型，右边标记返回值的类型。

Haskell 中所有函数都仅接受一个参数。

dropWhile 函数看起来像是接受两个参数，实际上 **dropWhile** 只接受一个参数，并返回一个函数，这个函数接受一个参数。

dropWhile 接受 **isSpace** 为参数，返回另一个**([Char] → [Char])**函数

复合函数 **dropWhile isSpace**

```
{-
ghci> :module +Data.Char
ghci> :type dropWhile isSpace
dropWhile isSpace :: [Char] → [Char]
-}
```

给列表中的字串去空格

```
import Data.Char (isSpace)
map (dropWhile isSpace) ["  a  ","f","  e"] -- >["a  ","f","e"]  -- 后面的空格去不掉，因为一旦布尔函数返回 false 就不再往后 drop 了。
```

每给函数一个参数，就会“**chop**” 掉类型签名前端的一个元素。

zip3 接受三个参数，返回一个 **three-tuples**

你取一个，我取一个，他取一个元素组成一个 **three-tuples**，重复这个过程生成并返回一个列表。

```
:type zip3
-- >zip3 :: [a] → [b] → [c] → [(a, b, c)]

:type zip3 "abc"
-- >zip3 "abc" :: [b] → [c] → [(char, b, c)]
```

-- 从“**b**” 开始说明已经有一个参数了，而且看得出那个参数的类型是 **char**

```
let zip3foo = zip3 "foo"
  (zip3 "foo") "aaa" "bbb"
let zip3foobar = zip3 "foo" "bar"
```

当我们传递的参数少于一个函数可接受的参数数量时，就称它为函数的“**partial application**”

partial 函数可以帮助我们避免去编写“**tiresome throwaway**”函数。就这个目标来说，**partial** 函数通常比匿名函数更有用。

partial 函数

比较这四个不同版本的函数，这些函数完成同样的事情

```
-- 使用辅助函数实现
isInAny needle haystack = any inSequence haystack
      where inSequence s = needle `isInfixOf` s

"needle" `isInfixOf` "haystack full of needle thingies"    -- >True

-- 使用匿名函数实现
isInAny2 needle haystack = any ( λ s → needle `isInfixOf` s) haystack

-- 使用 partial 函数实现
isInAny3 needle haystack = any (isInfixOf needle) haystack

-- 使用 section 函数实现(后面有解释)

isInAny4 needle haystack = any (needle `isInfixOf`) haystack
```

```
isInAny4 "llo" ["hello,world!"]    -- >True
```

这里的 isInfixOf needle 是 partial 函数，已经拥有部分参数了，所需的剩余参数由 any 给出。

any 函数接受一个布尔函数(isInfixOf needle)和一个列表["hello,world!"]，any 函数会从列表中一个个的把元素提出来("hello,world!")，并传给布尔函数。

Partial function application 被命名为 **currying**，是逻辑学家 Haskell Curry 命名的。

作为 currying 的另一个例子，下面是更好的列表求和函数

列表求和

```
nicerSum :: [Integer] -> Integer
nicerSum = foldl (+) 0

nicerSum [1,2,3] -- >6
```

Sections

中缀风格的 partial 函数

将中缀操作符用括号括起来，然后将参数放在左边或右边就得到一个 partial 函数。这种 partial 函数也称为 section。

sections 函数

```
 (1+) 2    -- >3
map (*3) [24,36]    -- >[72,108]
map (2^) [3,5,7,9]    -- >[8,32,128,512]
 (`elem` ['a'..'z']) 'f'    -- >True
```

测试一个字串是否是全小写

```
all (`elem` ['a'..'z']) "Frobozz"    -- >False
```

section 函数实现(前面有各种实现的比较)

```
isInAny4 needle haystack = any (needle `isInfixOf`) haystack
```

As-patterns

tails 函数生成字串的所有后缀串的列表，最后再附加一个空串列表

```
tail (tail "foobar")    -- >"obar"
tails []    -- >[[]]
tails "foobar"    -- >["foobar","oobar","obar","bar","ar","r",""]
```

如果我们需要一个类似 **tails** 的行为但是仅返回那些非空后缀的函数呢？可以使用“**@**” 符号

生成字串的所有后缀串的列表(后面有改进版)

```
suffixes :: [a] -> [[a]]
suffixes xs@(_:xs') = xs : suffixes xs'
suffixes _ = []

nn = suffixes "foo"    -- >["foo","oo","o"]
{-
"foo":"oo":"o":[]
-}
```

模板 **xs@(_:xs')** 称为“**as-pattern**”，意思为“将 **xs** 绑定为@右边成功匹配的值”。

xs@(_:xs')如果匹配成功则 **xs** 等于整个被匹配的列表。

不使用“**as-pattern**” 的版本

```
noAsPattern :: [a] -> [[a]]
noAsPattern (x:xs) = (x:xs) : noAsPattern xs
noAsPattern _ = []
```

(x:xs) 会在运行时分配一个新的 **list node**，开销便宜，但不是免费的。

xs@(_:xs') 中的 **xs** 由于使用了一个已存在的值(一个匹配)，所以避免了开销。

Code Reuse Through Composition

生成字串的所有后缀串的列表

```
suffixes2 xs = init (tails xs)

suffixes2 "foo"    -- >["foo","oo","o"]

{-
tails "foo"    -- >["foo","oo","o",""]
init ["foo","oo","o",""]    -- >["foo","oo","o"]
-}
```

init 函数返回整个列表，除了最后一个元素。

复合函数

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

suffixes3 xs = compose init tails xs
```

使用 compose 函数(复合函数)加 partial 函数定义

```
suffixes4 = compose init tails
```

幸运的是，我们不需要自已编自 compose 函数。

使用“.” 操作符构造 compose 函数(复合函数)

```
suffixes5 = init . tails

suffixes5 "foo"    -- >["foo","oo","o"]
```

“.” 不是特殊的语法，而是一个普通的操作符

```
:type (.) --  括号不可少，否则会出错
```

-- >(.) :: (b -> c) -> (a -> b) -> a -> c -- 这里应该这样读：**a** 传给**(a -> b)** 作参数，**(a -> b)** 传给**(b -> c)** 作参数，**c** 是**(b -> c)** 的返回值。

复合函数中，右边的函数是左边函数的参数。

计算一个字串中大写字母开头的单词数

```
capCount = length . filter (isUpper . head) . words

capCount "Hello there, Mom!" -- 2

{-
words "Hello there, Mom!"    -->["Hello","there,","Mom!"]
head "Hello" -- >'H'
isUpper 'H'    -- >true
["Hello",,"Mom!"]
-}
```

注意，words 函数是根据空白来分单词的，所以"there," 里面的逗号也算单词的一部分了。

filter 将列表**["Hello","there,","Mom!"]** 中的元素一个个提取出来，并传给**(isUpper . head)**，如果这个布尔函数返回 **false** 就将那个元素过滤掉。

解析一个 C 头文件，它是 libpcap 库的一部分。这个头文件具有以下形式：

```
{-
#define DLT_EN10MB 1 /* Ethernet (10Mb) */
#define DLT_EN3MB 2 /* Experimental Ethernet (3Mb) */
#define DLT_AX25 3 /* Amateur Radio AX.25 */
-}
```

我们的目标是提取类似这样的名字“DLT_EN10MB” 和 “DLT_AX25”

提取以特定字符开头的单词

```
import Data.List (isPrefixOf)
dlts :: String -> [String]
dlts = foldr step [] . lines
       where step l ds
```

```
      | "#define DLT_" `isPrefixOf` l = secondWord l : ds
      | otherwise = ds
secondWord = head . tail . words
```

```
dlts "#define DLT_CHAOS 5\n#define DLT_AX25 3\n#define DLT_AX25 3" -- >["DLT_CHAOS","DLT_AX25","DLT_AX25"]
```

Tips for Writing Readable Code

尽量组合使用库函数，如 **map**，**take**，**filter** 来代替尾部递归和匿名函数，可以使得代码更可读，加快编码速度，**bug** 更少。

能使用库函数组合，就不要使用 **fold**。用 **fold** 代替 **hand-rolled** 尾递归循环。

好的函数名就是一个 **tiny** 文档。

Space Leaks and Strict Evaluation

用 **foldr** 和 **foldl'** 代替 **foldl** 可以避免 **Space Leaks**

Avoiding Space Leaks with seq

Seq 函数存大的唯一目的就是强制立该计算出表达式的值。

seq 将第一个参数强制计算出结果，并返回它的第二个参数。

foldl' 的实现

```
myfoldl' _ zero [] = zero
myfoldl' step zero (x:xs) =
      let new = step zero x
      in new `seq` myfoldl' step new xs

myfoldl' (+) 1 (2:[])    -- >3
```

上面的调用将展开成：

```
{-
let new = 1 + 2
in new `seq` foldl' (+) new []
-}
```

seq 将 new 这个 thunk 强制计算出结果 3，将返回它的第二个参数，既 foldl' (+) 3 []。

多亏 seq ，这里 thunk 消失了。

Learning to Use seq

seq 必须出现在表达式的最前面。

错误：seq 的 hiddenInside 误用

```
hiddenInside x y = someFunc (x `seq` y)
```

错误：seq 的 hiddenByLet 误用

```
hiddenByLet x y z = let a = x `seq` someFunc y
                    in anotherFunc a z
```

正确：seq 将被首先 evaluated，它强制算出 x 的值

```
onTheOutside x y = x `seq` someFunc y
```

seq 的链式应用

```
chained x y z = x `seq` y `seq` someFunc z    -- 亮点!
```

seq 不能用于两个不相关的表达式

```
badExpression step zero (x:xs) =
    seq (step zero x)    -- 错误：这个表达式的值不会对第二个表达式造成任何影响
        (badExpression step (step zero x) xs)
```

seq 一遇到构造器就会停下来

将 seq 用于(1+2):(3+4):[] 这个表达式，只有(1+2) 这个 thunk 被算出来，因为它一遇到“:”这个构造器就停上了。

对 tuple 也是这样，如((1+2),(3+4)) 它立刻就遇到了构造器。

strict 的 pair 和 strict 的 list

```
strictPair (a,b) = a `seq` b `seq` (a,b)    -- 亮点

strictList (x:xs) = x `seq` x : strictList xs -- 亮点
strictList [] = []
```

节约的使用 **seq**，它不是免费的，**seq** 会在运行时去检查一个表达式是否已被计算。

不正确的使用 **seq** 会增大 **space leak**，或造成新的 **space leak**。

如何知道是否需要使用 **seq**，或它是否能很好的工作？最好的办法是进行性能测量和分析。

Chapter 5. Writing a Library: Working with JSON Data

A Whirlwind Tour of JSON

本章将开发一个小的，但完整的 Haskell 库。这个库以 JSON 格式(JavaScript Object Notation)来操作和序列化数据。

Representing JSON Data in Haskell

用 algebraic 数据类型来表示 JSON 数据类型

```
data JValue = JString String
            | JNumber Double
            | JBool Bool
            | JNull
            | JObject [(String, JValue)]
            | JArray [JValue]
            deriving (Eq, Ord, Show)
```

对每一个 JSON 类型，我们提供一个不同的值构造器。其中一些构造器拥有参数：如果我们想创建一个 JSON string，就必须提供一个 String 值作为 JString 构造器的参数。

```
getString :: JValue -> Maybe String
getString (JString s) = Just s
getString _ = Nothing

getInt (JNumber n) = Just (truncate n)
getInt _ = Nothing
getDouble (JNumber n) = Just n

getDouble _ = Nothing
getBool (JBool b) = Just b
```

```
getBool _ = Nothing
getObject (JObject o) = Just o
getObject _ = Nothing
getArray (JArray a) = Just a
getArray _ = Nothing
isNull v = v == JNull

{-
ghci> JString "foo"
JString "foo"
ghci> JNumber 2.7
JNumber 2.7
ghci> :type JBool True
JBool True :: JValue

ghci> getString (JString "hello")
Just "hello"
ghci> getString (JNumber 3)
Nothing

-}
```

truncate 函数通过丢弃小数点将浮点数或比例数转成整数

```
-- import Data.Ratio
truncate 5.8    -- >5
truncate (22 % 7)    -- >3
```

The Anatomy of a Haskell Module

定义模块的方法

```
-- JSON.hs

module JSON
      (
        JValue(..)
        , getString
        , getInt
        , getDouble
        , getBool
        , getObject
        , getArray
        , isNull
      ) where
```

module 后接模块名，模块名必须以大写字母开头。源文件名和模块名必须相同。

模块名后接导出列表，**where** 关键字指出后面 **follow** 模块体。

导出列表指出此模块中的哪些名字在其它模块中可见。

JValue 后面的特殊 notation **“(..)”** 表示导出 **type** 和所有构造器。

如果忽略导出列表及包围它的括号“()”，则会导出模块中的所有名字

```
-- module ExportEverything where
```

如果想要不导出任何名字（这样几乎没什么用），可以让导出列表为空。

```
-- module ExportNothing () where

data JValue = JString String
```

```
    | JNumber Double
    | JBool Bool
    | JNull
    | JObject [(String, JValue)]
    | JArray [JValue]
    deriving (Eq, Ord, Show)

getString :: JValue -> Maybe String
getString (JString s) = Just s
getString _ = Nothing

getInt (JNumber n) = Just (truncate n)
getInt _ = Nothing
getDouble (JNumber n) = Just n

getDouble _ = Nothing
getBool (JBool b) = Just b
getBool _ = Nothing
getObject (JObject o) = Just o
getObject _ = Nothing
getArray (JArray a) = Just a
getArray _ = Nothing
isNull v = v == JNull
```

Compiling Haskell Source

ghc 生成 native 代码

将源文件编译成目标文件

ghc -c JSON.hs

“-c” 选项表示仅生成目标代码，如果忽略“-c” 则 ghc 会尝试生成完整的可执行程序，这会导致失败，因为我们没有写 main 函数

上面的命令执行后会生成 JSON.hi 和 SimpleJSON.o 。前一个是 **interface file**，含有导出的名字。后一个是 **object file**，含有机器码。

Generating a Haskell Program and Importing Modules

链接库并生成可执行文件

-- 双击运行批处理

-- JSON.bat
ghc -c JSON.hs -- 生成库
ghc -o JSON Main.hs JSON.o -- 生成可执行文件
JSON -- 运行可执行文件

cmd.exe

-- JSON.hs

```
module JSON
      (
        JValue(..)
        , getString
        , getInt
        , getDouble
        , getBool
        , getObject
        , getArray
        , isNull
      ) where

data JValue = JString String
                  | JNumber Double
                  | JBool Bool
                  | JNull
                  | JObject [(String, JValue)]
                  | JArray [JValue]
                  deriving (Eq, Ord, Show)

getString :: JValue -> Maybe String
getString (JString s) = Just s
getString _ = Nothing
```

```
getInt (JNumber n) = Just (truncate n)
getInt _ = Nothing
getDouble (JNumber n) = Just n

getDouble _ = Nothing
getBool (JBool b) = Just b
getBool _ = Nothing
getObject (JObject o) = Just o
getObject _ = Nothing
getArray (JArray a) = Just a
getArray _ = Nothing
isNull v = v == JNull

------------------------------------------------------
-- Main.hs
module Main (main) where
import JSON
main = print (JObject [("foo", JNumber 1), ("bar", JBool False)])
```

Printing JSON Data

```
-- PutJSON.hs
module PutJSON where
import Data.List (intercalate)
import SimpleJSON

renderJValue :: JValue -> String
renderJValue (JString s) = show s
renderJValue (JNumber n) = show n
renderJValue (JBool True) = "true"
renderJValue (JBool False) = "false"
renderJValue JNull = "null"

renderJValue (JObject o) = "{" ++ pairs o ++ "}"
      where pairs [] = ""
            pairs ps = intercalate ", " (map renderPair ps)
            renderPair (k,v) = show k ++ ": " ++ renderJValue v

renderJValue (JArray a) = "[" ++ values a ++ "]"
      where values [] = ""
            values vs = intercalate ", " (map renderJValue vs)
```

Type Inference Is a Double-Edged Sword

类型推断是把双刃剑，**top level** 的代码尽量自己写函数签名。

将字串格式化为字首大写

```
-- 不写函数签名的后果
upcaseFirst (c:cs) = (toUpper c):cs -- 如果忘记 ":cs" 就会发生隐式错误

camelCase :: String -> String
camelCase xs = concat (map upcaseFirst (words xs))

camelCase "hello,world!"    -- >"Hello,world!"
```

A More General Look at Rendering

Haskell 有一些 **pretty-printing libraries**。

```
data Doc = Empty
         | Char Char
         | Text String
         | Line
         | Concat Doc Doc
         | Union Doc Doc
         deriving (Show,Eq)

string :: String -> Doc
string str = undefined

text :: String -> Doc
text str = undefined

double :: Double -> Doc
double num = undefined

renderJValue :: JValue -> Doc
renderJValue (JBool True) = text "true"
renderJValue (JBool False) = text "false"
renderJValue JNull = text "null"
renderJValue (JNumber num) = double num
renderJValue (JString str) = string str
```

Developing Haskell Code Without Going Nuts

一种开发程序 **skeleton** 的有用技术是编写“**placeholder**” 或“**stub**” 版的类型和函数。

我们只是提到由 Prettify 模块提供 string，text，和 double 函数，但是其定义不做任何事情，函数只是简单的返回 **undefined**。

```
-- PrettyStub.hs
module PrettyStub
       (
         Doc(..)
         , string
         , text
         , double
       ) where

import JSON
data Doc = ToBeDefined
                 deriving (Show)
string :: String -> Doc
string str = undefined
text :: String -> Doc
text str = undefined
double :: Double -> Doc
double num = undefined
```

特殊值 undefined 类型是 a，所以总是 typechecks，不论我们在哪里用它。

如果试图对 undefined 进行 evaluate 我们的程序就会 crach。

```
{-
ghci> :type double
double :: Double -> Doc
ghci> double 3.14
*** Exception: Prelude.undefined
-}
```

Pretty Printing a String

```
string :: String -> Doc
string = enclose "" "" . hcat . map oneChar
```

这里“.” 的写法称为“Point-free style” ，它和函数复合操作符“.” 没有关系。

作为对比，下面是“pointy” 版本，它使用变量 s 来引用要操作的变量

```
pointyString :: String -> Doc
pointyString s = enclose "" "" (hcat (map oneChar s))

enclose :: Char -> Char -> Doc -> Doc
enclose left right x = char left <> x <> char right
```

enclose 函数只是简单的将一个 Doc 值用一对 opening 和 closing 符号包围起来。

```
(<>) :: Doc -> Doc -> Doc
a <> b = undefined
```

“<>” 函数是 Doc 的“++” 运算符，它 appends 两个 Doc 。

```
hcat :: [Doc] -> Doc
hcat xs = undefined
```

hcat 将多个 Doc (装在 list 里面)连接成一个 Doc。

我们的 string 函数应用 oneChar 函数到一个字串中的每一个字符，全部连接，并用 quotes 将 result 包围起来。

oneChar 函数 escapes 一个字符，或简单的把它 renders。

simpleEscapes 值是 pairs 的列表。我们将 pairs 的列表称为**关联列表(association list)**，或简称 alist。我们的每一个 alist 元素都将一个字符与一个 escaped representation 表示相关联。（**例如，换行这个字符其相关联的 escaped 表示是“\n”**）

```
take 4 simpleEscapes    -- >[('\b',"\\b"),('\n',"\\n"),('\f',"\\f"),('\r',"\\r")]

------------------------------------------------------
-- PrettyJSON.hs
module PrettyJSON where

import JSON
import Numeric
import Data.Char (ord)
import Data.Bits (shiftR, (.&.), (.|.))
data Doc = ToBeDefined
                deriving (Show)

string :: String -> Doc
string = enclose "" "" . hcat . map oneChar

text :: String -> Doc
text str = undefined

pointyString :: String -> Doc
pointyString s = enclose "" "" (hcat (map oneChar s))

enclose :: Char -> Char -> Doc -> Doc
enclose left right x = char left <> x <> char right

(<>) :: Doc -> Doc -> Doc
a <> b = undefined

char :: Char -> Doc
char c = undefined

hcat :: [Doc] -> Doc
hcat xs = undefined

oneChar :: Char -> Doc
oneChar c = case lookup c simpleEscapes of
                        Just r -> text r
                        Nothing | mustEscape c -> hexEscape c
                                | otherwise -> char c
    where mustEscape c = c < ' ' || c == '\x7f' || c > '\xff'
```

-- **case** 表达式检查一个字符是否匹配 **alist** 中的元素。如果成功匹配就 **emit** 它，否则就需要将它 **escape** 成更复杂的形式。

```
simpleEscapes :: [(Char, String)]
simpleEscapes = zipWith ch "\b\n\f\r\t\\\"/" "bnfrt\\\"/"
      where ch a b = (a, ['\\',b])    -- 亮点

smallHex :: Int -> Doc
smallHex x = text "\\u"
               <> text (replicate (4 - length h) '0')
               <> text h
      where h = showHex x ""
```

-- 更复杂的 escaping 涉及将一个字符转成以"\u" 开头，后接四个十六进制数字，以此来表示 Unicode 字符。

-- showHex 函数返回一个数的十六进制表示(需要引入 Numeric 库)。

```
{-
ghci> showHex 114111 ""
"1bdbf"
-}
```

-- replicate 函数将输入参数复制指定次数并装在 list 中返回

```
{-
ghci> replicate 5 "foo"
["foo","foo","foo","foo","foo"]
-}

astral :: Int -> Doc
astral n = smallHex (a + 0xd800) <> smallHex (b + 0xdc00)
      where a = (n `shiftR` 10) .&. 0x3ff
            b = n .&. 0x3ff
```

-- smallHex 函数有个缺陷，它表示的字符冲顶只到 0xffff，而 Unicode 的最大有效值是 0x10ffff。

-- 为了正确表示大于 0xffff 的 JOSON 字符，我们用一些复杂的规则将超过 0xffff 的字符分割成两部分。这给了我们一个在比特级上操作 Haskel 数字的机会。

```
-- shiftR 和“.&.” 在 Data.Bits 模块
{-
ghci> 0x10000 `shiftR` 4 :: Int
4096
ghci> 7 .&. 2 :: Int
2
-}

hexEscape :: Char -> Doc
hexEscape c | d < 0x10000 = smallHex d
                  | otherwise = astral (d - 0x10000)
      where d = ord c
```

Arrays and Objects, and the Module Header

```
series :: Char -> Char -> (a -> Doc) -> [a] -> Doc
series open close item = enclose open close
                                  . fsep . punctuate (char ',') . map item
```

-- 注意到虽然我们的类型签名要求的参数是四个，但是我们在函数定义处只列出三个。这种规则可以让我们简化函数定义，例如 myLength xs = length xs 可以简化为 myLength = length

```
fsep :: [Doc] -> Doc
fsep xs = undefined
```

-- fsep 函数将一个 Doc 列表 combines 成一个 Doc。可能会对 lines 做些包装，如果输出不适合单行。

```
punctuate :: Doc -> [Doc] -> [Doc]
punctuate p [] = []
punctuate p [d] = [d]
punctuate p (d:ds) = (d <> p) : punctuate p ds

renderJValue (JArray ary) = series '[' ']' renderJValue ary

renderJValue (JObject obj) = series '{' '}' field obj
      where field (name,val) = string name
                                  <> text ": "
```

<> renderJValue val

Writing a Module Header

```
module PrettyJSON
      (
         renderJValue
      ) where

--import JSON
import Numeric (showHex)
import Data.Char (ord)
import Data.Bits (shiftR, (.&.))
import JSON (JValue(..))
import Prettify (Doc, (<>), char, double, fsep, hcat, punctuate, text,
                         compact, pretty)

module Prettify
      (
      -- * Constructors
         Doc
      -- * Basic combinators
      , (<>)
      , empty
      , char
      , text
      , line
      -- * Derived combinators
      , double
      , fsep
      , hcat
      , punctuate
      -- * Renderers
      , compact
      , pretty
      ) where
```

Fleshing Out the Pretty-Printing Library

```
-- Prettify.hs
data Doc = Empty
         | Char Char
         | Text String
         | Line
         | Concat Doc Doc
         | Union Doc Doc
         deriving (Show,Eq)
```

观察一下 Doc 这个类型，实际上 Doc 是一个 tree。

Concat 和 Union 从其它 Doc 构造出新的 tree。Empty，Text，Line 等是叶子。

我们模块的 header **导出了 Doc 这个类型名，但并不导出它的构造器，而是提供了函数来构造(Empty,Text,Char 等)。**

将 Prettify 中的 stubbed 函数替换成真实定义的函数

```
{-
empty :: Doc
empty = Empty

char :: Char -> Doc
char c = Char c

text :: String -> Doc
text "" = Empty
text s = Text s

double :: Double -> Doc
double d = text (show d)

line :: Doc
line = Line
```

```
(<>) :: Doc -> Doc -> Doc
Empty <> y = y
x <> Empty = x
x <> y = x `Concat` y    -- 注意了，Concat 是 DOC 的值构造器

-}
```

Line 构造器表示一个 line break。这是一个**硬 line break**，它总是会被打印。有时，我们想要**软 Line break**，它只在一个 line 宽过窗体或 page 时被打印(稍后会引入 softline)。

用模式匹配对 Empty 做了特别处理，使得在一个 Doc 的左边或右边接一个 Empty 没有任何效果，以防止 tree 因无用的值而膨胀。

可以说 Empty 是 concatenation 的单位元，类似于 0 是加法的单位元，1 是乘法的单位元。

```
{-
ghci> text "foo" <> text "bar"
Concat (Text "foo") (Text "bar")
ghci> text "foo" <> empty
Text "foo"
ghci> empty <> text "bar"
Text "bar"
-}
```

hcat 和 fsep 函数将一列表的 Doc (子树)连接成一个 Doc。

回想一下我们是如何定义 list 的 concatenation 的。

```
{-
concat :: [[a]] -> [a]
concat = foldr (++) []
-}
```

因为(<>) 类似于(++)，而 Empty 类似于([])，这提示我们如何用 folds 来实现 hcat 和 fsep。

```
{-
```

```
hcat :: [Doc] -> Doc
hcat = fold (<>)

fold :: (Doc -> Doc -> Doc) -> [Doc] -> Doc
fold f = foldr f empty    --  亮点
-}
```

fsep 的定义依赖于其它一些函数

```
{-
fsep :: [Doc] -> Doc
fsep = fold (</>)

(</>) :: Doc -> Doc -> Doc
x </> y = x <> softline <> y

softline :: Doc
softline = group line
-}
```

softline 函数会插入一个新行，如果当前行变得过宽，否则就插入一个空格。如何做到这一点？如果 Doc 类型不包含任何关于 rendering 的信息。答案是，**每一次遇到软换行，我们使用 Union 构造器维护文档的两个可选表示。**

```
{-
group :: Doc -> Doc
group x = flatten x `Union` x
-}
```

flatten 函数将一个 line 替换成一个空格，将两行转成一个更长的行。

```
{-
flatten :: Doc -> Doc
flatten (x `Concat` y) = flatten x `Concat` flatten y
flatten Line = Char ' '
flatten (x `Union` _) = flatten x
flatten other = other
-}
```

注意到我们总是调用 flatten 在 Union 的左元素：每个 Union 的左元素总是有相同的宽度(in characters)，或比右边宽。

Compact Rendering

```
-- Prettify.hs
{-
compact :: Doc -> String
compact x = transform [x]   -- 亮点 -- 注意这里是怎样将参数装进 list 里面的
      where transform [] = ""
                transform (d:ds) =
                  case d of
                        Empty -> transform ds
                        Char c -> c : transform ds
                        Text s -> s ++ transform ds
                        Line -> '\n' : transform ds
                        a `Concat` b -> transform (a:b:ds)
                        _ `Union` b -> transform (b:ds)
-}
```

compact 将其参数包装进一个 list 里面，并对它应用辅助函数 transform。transform 函数将其参数看成一个 items 的 stack 来处理，list 中的第一个元素就是 stack 的 top。

```
{-
ghci> let value = renderJValue (JObject [("f", JNumber 1), ("q", JBool True)])
ghci> :type value
value :: Doc
ghci> putStrLn (compact value)
{"f": 1.0,
"q": true
}
-}

renderJValue (JObject [("f", JNumber 1), ("q", JBool True)])
compact value   -- 用 print 打印不知道为什么出错
compact (char 'f' <> text "oo")   -- "foo"
```

True Pretty Printing

compact 对机器到机器的通信有用，但是其结果不总是易读的：每一行的信息非常少。为生成更可读的输出，我们将编写另一个函数，pretty 函数。相比 compact ，pretty 接受一个额外的参数：一行的最大宽度，列数（假定 typeface 是固定宽度）：

```
{-
pretty :: Int -> Doc -> String
pretty width x = best 0 [x]
      where best col (d:ds) =
                  case d of
                        Empty -> best col ds
                        Char c -> c : best (col + 1) ds
                        Text s -> s ++ best (col + length s) ds
                        Line -> '\n' : best 0 ds
                        a `Concat` b -> best col (a:b:ds)
                        a `Union` b -> nicest col (best col (a:ds))
                                                  (best col (b:ds))
            best _ _ = ""
            nicest col a b | (width - least) `fits` a = a
                           | otherwise = b
              where least = min width col
-}
```

pretty 的 Int 参数控制其遇到软件换行 softline 时的行为。只有在 softline 这里 pretty 有可选的行为，它或继续当前行，或开始新的一行。

```
{-
fits :: Int -> String -> Bool
w `fits` _ | w < 0 = False
w `fits` "" = True
w `fits` ('\n':_) = True
w `fits` (c:cs) = (w - 1) `fits` cs
-}
```

Following the Pretty Printer

```
ghci> empty </> char 'a'
Concat (Union (Char ' ') Line) (Char 'a')

ghci> 2 `fits` " a"
True

ghci> putStrLn (pretty 10 value)
{"f": 1.0,
"q": true
}
ghci> putStrLn (pretty 20 value)
{"f": 1.0, "q": true
}
ghci> putStrLn (pretty 30 value)
{"f": 1.0, "q": true }
```

Creating a Package

Haskell 社区创建了一个名为 **Cabal 的标准工具集**，可以帮助我们创建，安装，和分发软件。Cabal 将软件组织为包的形式。一个包含一个库，并且可能有几个可执行程序。

Writing a Package Description

要使用包，Cabal 需要对它的一个描述，就是一个后缀为.cabal 的文本文件。这个文件位于 project 的顶级目录中。它具有简单的格式。

```
{-
Cabal-Version: >= 1.2   -- 要求 Cabal 的版本不低于 1.2

library
    Exposed-Modules: Prettify
                     PrettyJSON
                     JSON
    Build-Depends:   base >= 2.0   -- base 包含很多 Haskell 核心模块，如 Prelude，所以几乎总是需要的。
-}
```

Exposed-Modules 包含要导出让用户可见的模块。有一个可选域，Other-Modules，表示内部模块需要用到，但用户不可见的模块。

GHC’s Package Manager

GHC 包含一个简单的包管理器，它 tracks 一个包的安装，包的版本是什么。命令行工具 ghc-pkg 让我们可以操作其包数据库。

ghc-pkg list 命令会列出已安装的包。命令 ghc-pkg unregister 用于卸载包(已安装的文件需要手工删除)。

Setting Up, Building, and Installing

一个包除了有.cabal 文件，还必须要有一个安装文件。

典型的安装文件像这样：

```
{-
Setup.hs
#!/usr/bin/env runhaskell
import Distribution.Simple
main = defaultMain
-}
```

一旦写好了.cabal 和 Steup.hs 文件，这儿还有三件事要做：

1. 指导 Cabal 如何创建和在包要安装在哪里，远行一个简单命令：
$ runghc Setup configure
这个保证我们需要的包是可用的，还有其存储的 settings 对稍后其它 Cabal 命令也可用。
如果不给 configure 提供任何参数，Cabal 将我们的包安装在系统范围包数据库。

2. build 这个包
$ runghc Setup build

3. 如果这一步成功，我们可以安装这个包。我们不需要去指出安装在哪里——Cabal 将使用我们在 configure 那一步中提供的 settings。它将安装在我们自已的目录，并更新 GHC 的 per-user 包数据库。

Practical Pointers and Further Reading

GHC 已经提供了一个名为 Text.PrettyPrint.HughesPJ 的 pretty-printing 库。建议使用它，而不是自己去写。

pretty-printing 库的设计文档可以在这里找到(http://citeseer.ist.psu.edu/hughes95design.html)

```
-----------------------------------------------------------------------------------------------
-- 本章的代码

--Setup.bat
Setup.hs
import Distribution.Simple
main = defaultMain

------------------------------------------------
-- Setup.hs
import Distribution.Simple
main = defaultMain

------------------------------------------------
-- mypretty.cabal
Name: mypretty
Version: 0.1

Synopsis: My pretty printing library, with JSON support
Description:
    A simple pretty-printing library that illustrates how to
    develop a Haskell library.
Author: Real World Haskell
Maintainer: nobody@realworldhaskell.org

Cabal-Version: >= 1.2
library
    Exposed-Modules: Prettify
                     PrettyJSON
                     JSON
    Build-Depends:   base >= 2.0
-------------------------------------------------------
```

```
-- JSON.hs

--  定义模块的方法

module JSON
      (
        JValue(..)
        , getString
        , getInt
        , getDouble
        , getBool
        , getObject
        , getArray
        , isNull
      ) where

-- module  后接模块名，模块名必须以大写字母开头。源文件名和模块名必须相同。

--  模块名后接导出列表，where  关键字指出后面 follow  模块体。

--  导出列表指出此模块中的哪些名字在其它模块中可见。

-- JValue  后面的特殊 notation  “(..)”  表示导出 type  和所有构造器。

--  如果忽略导出列表及包围它的括号“()”，则会导出模块中的所有名字

-- module ExportEverything where

--  如果想要不导出任何名字（这样几乎没什么用），可以让导出列表为空。

-- module ExportNothing () where

data JValue = JString String
                    | JNumber Double
                    | JBool Bool
                    | JNull
                    | JObject [(String, JValue)]
                    | JArray [JValue]
                    deriving (Eq, Ord, Show)

getString :: JValue -> Maybe String
getString (JString s) = Just s
getString _ = Nothing
```

```
getInt (JNumber n) = Just (truncate n)
getInt _ = Nothing
getDouble (JNumber n) = Just n

getDouble _ = Nothing
getBool (JBool b) = Just b
getBool _ = Nothing
getObject (JObject o) = Just o
getObject _ = Nothing
getArray (JArray a) = Just a
getArray _ = Nothing
isNull v = v == JNull

--------------------------------------------------------------------------------
-- Prettify.hs
module Prettify
    (
    -- * Constructors
      Doc
    -- * Basic combinators
    , (<>)
    , empty
    , char
    , text
    , line
    -- * Derived combinators
    , double
    , fsep
    , hcat
    , punctuate
    -- * Renderers
    , compact
    , pretty
    ) where

punctuate :: Doc -> [Doc] -> [Doc]
punctuate p [] = []
punctuate p [d] = [d]
punctuate p (d:ds) = (d <> p) : punctuate p ds

data Doc = Empty
         | Char Char
         | Text String
         | Line
```

```
            | Concat Doc Doc
            | Union Doc Doc
            deriving (Show,Eq)

empty :: Doc
empty = Empty
char :: Char -> Doc
char c = Char c
text :: String -> Doc
text "" = Empty
text s = Text s
double :: Double -> Doc
double d = text (show d)

line :: Doc
line = Line

(<>) :: Doc -> Doc -> Doc
Empty <> y = y
x <> Empty = x
x <> y = x `Concat` y

hcat :: [Doc] -> Doc
hcat = fold (<>)
fold :: (Doc -> Doc -> Doc) -> [Doc] -> Doc
fold f = foldr f empty

fsep :: [Doc] -> Doc
fsep = fold (</>)
(</>) :: Doc -> Doc -> Doc
x </> y = x <> softline <> y
softline :: Doc
softline = group line

group :: Doc -> Doc
group x = flatten x `Union` x

flatten :: Doc -> Doc
flatten (x `Concat` y) = flatten x `Concat` flatten y
flatten Line = Char ' '
flatten (x `Union` _) = flatten x
flatten other = other

compact :: Doc -> String
```

```
compact x = transform [x]
      where transform [] = ""
            transform (d:ds) =
              case d of
                   Empty -> transform ds
                   Char c -> c : transform ds
                   Text s -> s ++ transform ds
                   Line -> '\n' : transform ds
                   a `Concat` b -> transform (a:b:ds)
                   _ `Union` b -> transform (b:ds)

pretty :: Int -> Doc -> String
pretty width x = best 0 [x]
      where best col (d:ds) =
              case d of
                   Empty -> best col ds
                   Char c -> c : best (col + 1) ds
                   Text s -> s ++ best (col + length s) ds
                   Line -> '\n' : best 0 ds
                   a `Concat` b -> best col (a:b:ds)
                   a `Union` b -> nicest col (best col (a:ds))
                                             (best col (b:ds))
            best _ _ = ""
            nicest col a b | (width - least) `fits` a = a
                           | otherwise = b
              where least = min width col

fits :: Int -> String -> Bool
w `fits` _ | w < 0 = False
w `fits` "" = True
w `fits` ('\n':_) = True
w `fits` (c:cs) = (w - 1) `fits` cs

--------------------------------------------------------------------------------
-- PrettyJSON.hs


module PrettyJSON
      (
        renderJValue
       ,simpleEscapes
      ) where

--import JSON
```

```
import Numeric (showHex)
import Data.Char (ord)
import Data.Bits (shiftR, (.&.))
import JSON (JValue(..))
import Prettify (Doc, (<>), char, double, fsep, hcat, punctuate, text,
                 compact, pretty)

string :: String -> Doc
string = enclose "" "" . hcat . map oneChar

enclose :: Char -> Char -> Doc -> Doc
enclose left right x = char left <> x <> char right

oneChar :: Char -> Doc
oneChar c = case lookup c simpleEscapes of
              Just r -> text r
              Nothing | mustEscape c -> hexEscape c
                      | otherwise -> char c
    where mustEscape c = c < ' ' || c == '\x7f' || c > '\xff'

-- case 表达式检查一个字符是否匹配 alist 中的元素。如果成功匹配就 emit 它，否则就需要将它 escape 成更复杂的形式。

simpleEscapes :: [(Char, String)]
simpleEscapes = zipWith ch "\b\n\f\r\t\\\"/" "bnfrt\\\"/"
    where ch a b = (a, ['\\',b])    -- 亮点

smallHex :: Int -> Doc
smallHex x = text "\\u"
          <> text (replicate (4 - length h) '0')
          <> text h
    where h = showHex x ""

-- 更复杂的 escaping 涉及将一个字符转成以"\u" 开头，后接四个十六进制数字，以此来表示 Unicode 字符。

-- showHex 函数返回一个数的十六进制表示(需要引入 Numeric 库)。

{-
ghci> showHex 114111 ""
"1bdbf"
-}

-- replicate 函数将输入参数复制指定次数并装在 list 中返回
```

```
{-
ghci> replicate 5 "foo"
["foo","foo","foo","foo","foo"]
-}

astral :: Int -> Doc
astral n = smallHex (a + 0xd800) <> smallHex (b + 0xdc00)
      where a = (n `shiftR` 10) .&. 0x3ff
            b = n .&. 0x3ff

-- smallHex 函数有个缺陷，它表示的字符冲顶只到 0xffff，而 Unicode 的最大有效值是 0x10ffff。

-- 为了正确表示大于 0xffff 的 JOSON 字符，我们用一些复杂的规则将超过 0xffff 的字符分割成两部分。这给了我们一个在比特级上操作 Haskel 数字的机会。

-- shiftR 和“.&.” 在 Data.Bits 模块
{-
ghci> 0x10000 `shiftR` 4 :: Int
4096
ghci> 7 .&. 2 :: Int
2
-}

hexEscape :: Char -> Doc
hexEscape c | d < 0x10000 = smallHex d
            | otherwise = astral (d - 0x10000)
      where d = ord c


-- Arrays and Objects, and the Module Header

series :: Char -> Char -> (a -> Doc) -> [a] -> Doc
series open close item = enclose open close
                         . fsep . punctuate (char ',') . map item

-- 注意到虽然我们的类型签名要求的参数是四个，但是我们在函数定义处只列出三个。这种规则可以让我们简化函数定义，例如 myLength xs = length xs 可以简化为 myLength = length

-- PrettyStub.hs
{-
fsep :: [Doc] -> Doc
```

```
fsep xs = undefined
-}

-- fsep 函数将一个 Doc 列表 combines 成一个 Doc。可能到 lines 做些包装，如果输出不适合单行。

--punctuate :: Doc -> [Doc] -> [Doc]
--punctuate p [] = []
--punctuate p [d] = [d]
--punctuate p (d:ds) = (d <> p) : punctuate p ds

renderJValue (JArray ary) = series '[' ']' renderJValue ary

renderJValue (JObject obj) = series '{' '}' field obj
      where field (name,val) = string name
                            <> text ": "
                            <> renderJValue val
```

Chapter 6. Using Typeclasses

用 **class** 声明 **Typeclass** 就是声明一组函数。

The Need for Typeclasses

Color 相等测试(后面有改进版)

```
data Color = Red | Green | Blue

colorEq :: Color -> Color -> Bool
colorEq Red Red = True
colorEq Green Green = True
colorEq Blue Blue = True
colorEq _ _ = False

cc = colorEq Red Red    -- >True
dd = colorEq Red Green    -- >False
```

String 相等测试

```
stringEq :: [Char] -> [Char] -> Bool

stringEq [] [] = True
stringEq (x:xs) (y:ys) = x == y && stringEq xs ys
stringEq _ _ = False
```

现在你看到了，我们要写不同的函数来对不同的类型作相等测试，不仅效率低而且讨人厌。

如果可以用“==” 操作符来比较任何东西就会方便得多。

What Are Typeclasses?

Typeclasses 定义了一组函数，这些函数对于给定的不同类据类型有不同的实现。Typeclasses 可能看起来像 OO 中的对象，但实际上它们相当不同。

让我们用 Typeclasses 来解决前面的困境。作为开始，我们必须定义 Typeclasses 本身。

我们需要一个函数，它不在乎其参数类型是什么。**这是我们的第一个 Typeclasses 定义：**

```
class BasicEq a where
    isEqual :: a -> a -> Bool
```

这里我们声明了一个名为 BasicEq 的 typeclass，而 a 是 BasicEq 的实例。

class 是定义 typeclass 的关键字。注意了，它和 OO 的类没有任何关系。

实例化 typeclass 的类型

```
instance BasicEq Bool where
    isEqual True True = True
    isEqual False False = True
    isEqual _ _ = False
```

定义不相等函数也可能是有用的

```
class BasicEq2 a where
    isEqual2 :: a -> a -> Bool
    isNotEqual2 :: a -> a -> Bool
```

这里两个函数的定义循环依赖，必须且只须定义其中一个

```
class BasicEq3 a where
    isEqual3 :: a -> a -> Bool
    isEqual3 x y = not (isNotEqual3 x y)
```

```
    isNotEqual3 :: a -> a -> Bool
    isNotEqual3 x y = not (isEqual3 x y)
```

如不给出定义，这两函数会引发无限循环。

内置 typeclass Eq 的定义

```
{-
class Eq a where
    (==), (/=) :: a -> a -> Bool    -- 亮点，有相同签名的两函数可以一起定义

        -- Minimal complete definition:
        -- (==) or (/=)
    x /= y = not (x == y)
    x == y = not (x /= y)
-}
```

Declaring Typeclass Instances

Color 相等测试

```
instance BasicEq3 Color where
    isEqual3 Red Red = True
    isEqual3 Green Green = True
    isEqual3 Blue Blue = True
    isEqual3 _ _ = False
```

这里 isNotEqual3 这里没有给出定义，所以编译器自动使用 Typeclass 声明 isNotEqual3 时的默入定义。

Important Built-in Typeclasses

Haskell 的 Prelude 包定义了一些标准 typeclass 。

typeclass 是 Haskell 语言的重要核心。

Show

Show 是一个 typeclass ，用于将 values 转成 Strings。

类型类 Show 最重要的函数是 show。

```
{-
ghci> :type show
show :: (Show a) => a -> String

ghci> show 1
"1"
ghci> show [1, 2, 3]
"[1,2,3]"
ghci> show (1, 2)
"(1,2)"
-}

{-
ghci> putStrLn (show 1)
1
ghci> putStrLn (show [1,2,3])
[1,2,3]

ghci> show "Hello!"
"\"Hello!\""
ghci> putStrLn (show "Hello!")
"Hello!"
ghci> show ['H', 'i']
"\"Hi\""
ghci> putStrLn (show "Hi")
```

```
"Hi"
ghci> show "Hi, \"Jane\""
"\"Hi, \\\"Jane\\\"\""
ghci> putStrLn (show "Hi, \"Jane\"")
"Hi, \"Jane\""
-}
```

为自己的类型定义 **Show** 实例

```
instance Show Color where
        show Red = "Red"
        show Green = "Green"
        show Blue = "Blue"

show Red    -- >"Red"
```

Read

Read 这个 typeclass 与 Show 本质上相反。它定义了一些函数，这些函数接受一个 String，解析它，并返回任意类型的数据(就是你用来实例化的那个类型)。Read 最有用的函数是 read。

```
{-
main = do
        putStrLn "Please enter a Double:"
        inpStr <- getLine    -- 读入一行
        let inpDouble = (read inpStr)::Double
        putStrLn ("Twice " ++ show inpDouble ++ " is " ++ show (inpDouble * 2))
-}
```

(read inpStr)::Double 类似于“显示类型转换”，因为 read 的返回值是 a ，read String => a 。

ghci 在内部使用 show 来显示结果。

```
instance Read Color where
    readsPrec _ value =
        tryParse [("Red", Red), ("Green", Green), ("Blue", Blue)]
        where tryParse [] = []
              tryParse ((attempt, result):xs) =
                  if (take (length attempt) value) == attempt
                      then [(result, drop (length attempt) value)]
                      else tryParse xs
```

```
{-
ghci> (read "Red")::Color
Red
ghci> (read "Green")::Color
Green
ghci> (read "Blue")::Color
Blue
ghci> (read "[Red]")::[Color]
[Red]
ghci> (read "[Red,Red,Blue]")::[Color]
[Red,Red,Blue]
ghci> (read "[Red, Red, Blue]")::[Color]
```

```
*** Exception: Prelude.read: no parse
-}
```

Parsec 比 Read 更易用，Read 通常只用于简单任务。（参见第 16 章）

Serialization with read and show

read 和 **show** 是进行序列化的极好工具。**show** 生成人和机器都可读的输出。

String 处理通常是 **lazy** 的，所以 **read** 和 **show** 可以用来处理相当大的数据结构。

先写文件，文件名是“**test**”，内容是“**[Just 5,Nothing,Nothing,Just 8,Just 9]**”

```
{-
ghci> let d1 = [Just 5, Nothing, Nothing, Just 8, Just 9]::[Maybe Int]
ghci> putStrLn (show d1)
[Just 5,Nothing,Nothing,Just 8,Just 9]
ghci> writeFile "test" (show d1)    -- 亮点，写文件的方法
-}
```

然后再读回来

```
{-
ghci> input <- readFile "test"
"[Just 5,Nothing,Nothing,Just 8,Just 9]"
ghci> let d2 = read input    -- 亮点，读文件的方法
<interactive>:1:9:
Ambiguous type variable `a' in the constraint:
`Read a' arising from a use of `read' at <interactive>:1:9-18
Probable fix: add a type signature that fixes these type variable(s)
ghci> let d2 = (read input)::[Maybe Int]
ghci> print d1
[Just 5,Nothing,Nothing,Just 8,Just 9]
ghci> print d2
[Just 5,Nothing,Nothing,Just 8,Just 9]
ghci> d1 == d2
```

```
True
-}

{-
ghci> putStrLn $ show [("hi", 1), ("there", 3)]   -- 亮点，用“$” 操作符当括号
[("hi",1),("there",3)]
ghci> putStrLn $ show [[1, 2, 3], [], [4, 0, 1], [], [503]]
[[1,2,3],[],[4,0,1],[],[503]]
ghci> putStrLn $ show [Left 5, Right "three", Left 0, Right "nine"]
[Left 5,Right "three",Left 0,Right "nine"]
ghci> putStrLn $ show [Left 0, Right [1, 2, 3], Left 5, Right []]
[Left 0,Right [1,2,3],Left 5,Right []]
-}
```

Numeric Types

Haskell 的数值类型非常强大。你可以使用从 32 位到 64 位的整数，还有任意精度的比例数。“+” 可以适用于所有这些数值。这是使用 typeclass 实现的。你也可以定义自己的数值类型，并使它们成为 Haskell 中的 fist-class citizens。

Equality, Ordering, and Comparisons

“==” 和“/=” 定义于 Eq class 。

“>=” 和“<=” 定义于 Ord class 。

Ord 的任何实例都可以用 Data.List.sort 排序。

几乎所有 Haskell 的类型都是 Eq 的实例，Ord 的实例也接近这么多。

Automatic Derivation

对一些简单的数据类型，**Haskell** 编译器可以自动为我们继承实例，包括 **Read**， **Show**， **Bounded**， **Enum**， **Eq**， 和 **Ord**

使用 **deriving** 关键字自动继承 **typeclass** 实例

```
data Color = Red | Green | Blue
        deriving (Read, Show, Eq, Ord)

{-
ghci> show Red
"Red"
ghci> (read "Red")::Color
Red
ghci> (read "[Red,Red,Blue]")::[Color]
[Red,Red,Blue]
ghci> (read "[Red, Red, Blue]")::[Color]
[Red,Red,Blue]
ghci> Red == Red
True
ghci> Red == Blue
False
ghci> Data.List.sort [Blue,Green,Blue,Red]    -- 大小和定义顺序一致
[Red,Green,Blue,Blue]
ghci> Red < Blue
True
-}
```

注意到 Color 的 order 接照我们定义构造器的顺序来定义的。

类似这样的 **data MyType = MyType (Int -> Bool)** 不能自动继承，因为 **show** 不知道怎样 **render** 一个函数。

```
data CannotShow = CannotShow
                        deriving (Show)
```

```
-- 不能编译
data CannotDeriveShow = CannotDeriveShow CannotShow
                                  deriving (Show)

-- 正确
data OK = OK

instance Show OK where
      show _ = "OK"

data ThisWorks = ThisWorks OK
                        deriving (Show)
```

Typeclasses at Work: Making JSON Easier to Use

```
--import JSON

result :: JValue

result = JObject [
    ("query", JString "awkward squad haskell"),
    ("estimatedCount", JNumber 3920),
    ("moreResults", JBool True),
    ("results", JArray [
        JObject [
        ("title", JString "Simon Peyton Jones: papers"),
        ("snippet", JString "Tackling the awkward ..."),
        ("url", JString "http://.../marktoberdorf/")
        ]])
    ]

type JSONError = String
class JSON a where
    toJValue :: a -> JValue
    fromJValue :: JValue -> Either JSONError a    -- Either 类似于 Mabe，我们用它表示这里的操作可能会失败。

instance JSON JValue where
    toJValue = id
    fromJValue = Right
```

More Helpful Errors

fromJValue 函数使用了 Either 类型，和 **Mabe** 一样，**Either** 这个类型也是也是预定义了的。我们用它表示那里的操作可能会失败。

Either 的定义

```
{-
data Either a b = Left a
```

```
                | Right b
                deriving (Eq, Ord, Read, Show)
-}
```

Either 的结构类似于 Nothing ，但是 Either 有一个“something bad happened”的 Left 构造器。

```
instance JSON Bool where
    toJValue = JBool
    fromJValue (JBool b) = Right b
    fromJValue _ = Left "not a JSON boolean"    -- 有不好的事情发生了!
```

Making an Instance with a Type Synonym

Haskell 98 标准不允许写这样的代码，虽然看起来很 perfectly：

```
instance JSON String where
    toJValue = JString
    fromJValue (JString s) = Right s
    fromJValue _ = Left "not a JSON string"
```

使用这个可以使得上面的代码有效：

{-# LANGUAGE TypeSynonymInstances #-} -- 把这个放到源码的最顶端(不是 top 是不行的!)

Living in an Open World

```
-- JSONClass.hs
{-# LANGUAGE TypeSynonymInstances #-}

module JSONClass () where

import JSON

type JSONError = String

class JSON a where
        toJValue :: a -> JValue
        fromJValue :: JValue -> Either JSONError a

instance JSON JValue where
        toJValue = id
        fromJValue = Right

instance JSON Bool where
        toJValue = JBool
        fromJValue (JBool b) = Right b
        fromJValue _ = Left "not a JSON boolean"

instance JSON String where
        toJValue = JString
        fromJValue (JString s) = Right s
        fromJValue _ = Left "not a JSON string"

doubleToJValue :: (Double -> a) -> JValue -> Either JSONError a
doubleToJValue f (JNumber v) = Right (f v)
doubleToJValue _ _ = Left "not a JSON number"

instance JSON Int where
        toJValue = JNumber . realToFrac
        fromJValue = doubleToJValue round

instance JSON Integer where
        toJValue = JNumber . realToFrac
        fromJValue = doubleToJValue round

instance JSON Double where
```

```
    toJValue = JNumber
    fromJValue = doubleToJValue id

------------------------------------------------------------
-- BrokenClass.hs
{-# LANGUAGE FlexibleInstances, TypeSynonymInstances #-}

module BrokenClass () where

import JSON

type JSONError = String

class JSON a where
    toJValue :: a -> JValue
    fromJValue :: JValue -> Either JSONError a

instance (JSON a) => JSON [a] where
    toJValue = undefined
    fromJValue = undefined

instance (JSON a) => JSON [(String, a)] where
    toJValue = undefined
    fromJValue = undefined
```

aa = toJValue [("foo","bar")] -- 出错！解释见后面。-- Error：“Overlapping instances for JSON [([Char], [Char])]”

When Do Overlapping Instances Cause Problems?

toJValue [("foo","bar")] -- Error：“Overlapping instances for JSON [([Char], [Char])]”

要了解 Overlapping Instances 错误是怎么回事看下面的代码：

```
-- 要正常编译加这个：{-# LANGUAGE FlexibleInstances, TypeSynonymInstances #-}
class Borked a where
    bork :: a -> String

instance Borked Int where
    bork = show

instance Borked (Int, Int) where
```

```
    bork (a, b) = bork a ++ ", " ++ bork b

instance (Borked a, Borked b) => Borked (a, b) where
    bork (a, b) = ">>" ++ bork a ++ " " ++ bork b ++ "<<"
```

我们有两个 pairs 实例：一个是 Int pairs，另一个是任意 Borked pairs。(不知道应该匹配哪一个？)

Relaxing Some Restrictions on Typeclasses

TypeSynonymInstances 扩展使得我们可以使用[a] 这种泛类型的特化版本[Char]

OverlappingInstances 扩展使得当发生多匹配时使得编译器选择最特殊的那一个。

通常联合使用 TypeSynonymInstances，OverlappingInstances

```
{-
{-# LANGUAGE TypeSynonymInstances, OverlappingInstances #-}
import Data.List
class Foo a where
    foo :: a -> String

instance Foo a => Foo [a] where
    foo = concat . intersperse ", " . map foo

instance Foo Char where
    foo c = [c]

instance Foo String where
    foo = id
-}
```

How Does Show Work for Strings?

略

How to Give a Type a New Identity

除了 data 关键字，Haskell 还提供了一个 newtype 关键字用来定义新类型

```
data DataInt = D Int
                    deriving (Eq, Ord, Show)

newtype NewtypeInt = N Int
                         deriving (Eq, Ord, Show)
```

newtype 声明的目的是重命名一个已存在的类型，给它一个不同的标识。注意到 newtype 和 data 声明是很像的。

使用 newtype 声明的是一种真正的新类型，而用 type 声明的只是一种昵称

```
newtype UniqueID = UniqueID Int
                       deriving (Eq)
```

UniqueID 和 Int 是两种不同的类型，作为 UniqueID 的用户我们不知道它是用 Int 实现的。

```
{-
ghci> N 1 < N 2
True
-}
```

因为没有暴露 Int 的 Num 或 Integral 实例，所以 NewtypeInt 的值不是数字：

```
{-
ghci> N 313 + N 37    -- 出错
-}
```

Differences Between Data and Newtype Declarations

newtype 的使用比 **Date** 有更多的限制。

newtype 只能有一个值构造器，构造器的参数也只能有一个。

```
{-
data TwoFields = TwoFields Int Int

-- ok: exactly one field
newtype Okay = ExactlyOne Int

-- ok: type parameters are no problem
newtype Param a b = Param (Either a b)

-- ok: record syntax is fine
newtype Record = Record {
        getInt :: Int
      }

-- bad: no fields
newtype TooFew = TooFew

-- bad: more than one field
newtype TooManyFields = Fields Int Int

-- bad: more than one constructor
newtype TooManyCtors = Bad Int
                     | Worse Int
-}
```

用 **data** 创建类型在运行时有一个 **bookkeeping** 开销，而 **newtype** 没有。

下面的代码不会 **crash**

```
{-
```

```
ghci> case undefined of N _ -> 1
1
-}
```

因为在运行时没有构造器要 **present** 。匹配的时侯 **N _** 简单的等效于 **wild card** “**_**” ，因为 **wild card** 总是匹配的，所以就不需要去 **evaluated** 那个 **undefined** 。

JSON Typeclasses Without Overlapping Instances

```
-- JSONClass
{-# LANGUAGE TypeSynonymInstances #-}

module JSONClass ( JAry(fromJAry), jary ) where

--import JSON

import Control.Arrow (second)

data JValue = JString String
            | JNumber Double
            | JBool Bool
            | JNull
            | JObject (JObj JValue) -- was [(String, JValue)]
            | JArray (JAry JValue) -- was [JValue]
            deriving (Eq, Ord, Show)


type JSONError = String

class JSON a where
    toJValue :: a -> JValue
    fromJValue :: JValue -> Either JSONError a

instance JSON JValue where
    toJValue = id
    fromJValue = Right

instance JSON Bool where
    toJValue = JBool
    fromJValue (JBool b) = Right b
    fromJValue _ = Left "not a JSON boolean"

instance JSON String where
    toJValue = JString
    fromJValue (JString s) = Right s
    fromJValue _ = Left "not a JSON string"
```

```
doubleToJValue :: (Double -> a) -> JValue -> Either JSONError a
doubleToJValue f (JNumber v) = Right (f v)
doubleToJValue _ _ = Left "not a JSON number"

instance JSON Int where
        toJValue = JNumber . realToFrac
        fromJValue = doubleToJValue round

instance JSON Integer where
        toJValue = JNumber . realToFrac
        fromJValue = doubleToJValue round

instance JSON Double where
        toJValue = JNumber
        fromJValue = doubleToJValue id


newtype JAry a = JAry {
        fromJAry :: [a]
    } deriving (Eq, Ord, Show)
```

-- 通常导出一个 newtype 时我们不导出它的构造器，而是提供一个函数来构造

```
jary :: [a] -> JAry a
jary = JAry

newtype JObj a = JObj {
        fromJObj :: [(String, a)]
    } deriving (Eq, Ord, Show)

jaryFromJValue :: (JSON a) => JValue -> Either JSONError (JAry a)
jaryToJValue :: (JSON a) => JAry a -> JValue

instance (JSON a) => JSON (JAry a) where
        toJValue = jaryToJValue
        fromJValue = jaryFromJValue


listToJValues :: (JSON a) => [a] -> [JValue]
listToJValues = map toJValue
```

```
jvaluesToJAry :: [JValue] -> JAry JValue
jvaluesToJAry = JAry

jaryOfJValuesToJValue :: JAry JValue -> JValue
jaryOfJValuesToJValue = JArray

jaryToJValue = JArray . JAry . map toJValue . fromJAry

jaryFromJValue (JArray (JAry a)) =
      whenRight JAry (mapEithers fromJValue a)
jaryFromJValue _ = Left "not a JSON array"

whenRight :: (b -> c) -> Either a b -> Either a c
whenRight _ (Left err) = Left err
whenRight f (Right a) = Right (f a)

mapEithers :: (a -> Either b c) -> [a] -> Either b [c]
mapEithers f (x:xs) = case mapEithers f xs of
                                    Left err -> Left err
                                    Right ys -> case f x of
                                                          Left err -> Left err
                                                          Right y -> Right (y:ys)
mapEithers _ _ = Right []


instance (JSON a) => JSON (JObj a) where
      toJValue = JObject . JObj . map (second toJValue) . fromJObj
      fromJValue (JObject (JObj o)) = whenRight JObj (mapEithers unwrap o)
            where unwrap (k,v) = whenRight ((,) k) (fromJValue v)
      fromJValue _ = Left "not a JSON object"
```

The Dreaded Monomorphism Restriction

```
myShow = show    --  出错？

-- ok  ？
myShow2 value = show value
myShow3 :: (Show a) => a -> String
myShow3 = show
```

Chapter 7. I/O

使用<- 从 **IO** 获取输入，使用 **let** 从 **pure code** 获取输入。

pure code 就是相同的输入返回相同的输出，并且没有 **side effects** 的代码。在 **Haskell** 中只有 **I/O actions** 不遵循这一规则。

严格分隔 **pure code** 和非 **pure cod** 有利于编译器自动优化和并行化。

Classic I/O in Haskell

```
-- runghc.bat
{-
@echo off
ghci main
-}

{-

main = do
    putStrLn "Greetings! What is your name?"
    inpStr <- getLine
    putStrLn $ "Welcome to Haskell, " ++ inpStr ++ "!"
-}

{-
$ runghc basicio.hs
Greetings! What is your name?
John
Welcome to Haskell, John!
-}
```

putStrLn 会在输出一个 **String** 后再输出一个换行。

```
{-
ghci> let writefoo = putStrLn "foo"
ghci> writefoo
foo
-}
```

foo 不是 witefoo 的返回值，而是 putStrLn 的一个 side effect(这就是不纯的后果！)

main 函数本身就是一个 IO ()，关键字 do 表示后面的代码有 side effect 。

```
name2reply :: String -> String
name2reply name =
    "Pleased to meet you, " ++ name ++ ".\n" ++
    "Your name contains " ++ charcount ++ " characters."
        where charcount = show (length name)

main :: IO ()
main = do
    putStrLn "Greetings once again. What is your name?"
    inpStr <- getLine
    let outStr = name2reply inpStr
    putStrLn outStr
```

使用<- 从 IO 获取输入，使用 let 从 pure code 获取输入。

Pure Versus I/O

pure code 就是相同的输入返回相同的输出，并且没有 side effects 的代码。在 Haskell 中只有 I/O actions 不遵循这一规则。

Why Purity Matters

略

Working with Files and Handles

openFile 函数(需要引入 System.IO)会返回一个文件句柄。配套的 hPutStrLn 用来往文件输出。用完后要用 hClose 关闭句柄。

作为开始，让我们以命令式的方式来读和写文件。这看起来像你在其它语言看到过的 while 循环。这不是的 Haskell 的最佳方式，后面我们会改进它。

读写文件(循环的方式)

```
import System.IO
import Data.Char(toUpper)

main :: IO ()
main = do
          inh <- openFile "input.txt" ReadMode
          outh <- openFile "output.txt" WriteMode
          mainloop inh outh
          hClose inh
          hClose outh

mainloop :: Handle -> Handle -> IO ()
mainloop inh outh =
     do ineof <- hIsEOF inh
        if ineof
              then return ()
              else do inpStr <- hGetLine inh
                          hPutStrLn outh (map toUpper inpStr)
                          mainloop inh outh
```

return 的含义与 C 等语言的是不同的，return 与“<-” 是反义。return 接受一个 pure value 并 wraps 到 IO 里。

所有 I/O action 都必须返回 IO type，如果你的结果是从 pure computation 来的就必须 return to wrap it in IO。

More on openFile

使用 **openBinaryFile** 处理二进制文件。一些操作系统，如 **Windows** 在处理二进制和文本文件上的行为是相当不同的。

Closing Handles

Haskell 会为文件在内部维护一个缓存，直到用 **hClose** 关闭文件才会进行数据写入。

Seek and Tell

hTell 函数报告 **current position** 前面有多少个字节。刚开始是 **0**，读了 **5** 字节以后就是 **5**，等等。

hSeek 函数设置 **current position**

hIsSeekable 函数用来检查一个 **Handle** 是否可以 **seek** 。

Standard Input, Output, and Error

getLine, print 的实现

```
{-
getLine = hGetLine stdin
putStrLn = hPutStrLn stdout
print = hPrint stdout
-}
```

使和 **echo** 命令给程序输入参数

echo John|runghc callingpure.hs

Temporary Files

openTempFile, openBinaryTempFile,System.Directory.getTemporaryDirectory

Extended Example: Functional I/O and Temporary Files

如果 1 的输出正好可以作为 2 的输入，就可以将两函数组合成复合函数。

在 Leksah IDE 中添加依赖包

Leksah ->Edit Package ->Dependencies ->Enter -> 输入“directory” ->Add ->Save

读写临时文件

```
import System.IO
import System.Directory(getTemporaryDirectory, removeFile)
import System.IO.Error(catch)
import Control.Exception(finally)

main :: IO ()
main = withTempFile "mytemp.txt" myAction

myAction :: FilePath -> Handle -> IO ()
myAction tempname temph =
    do
        putStrLn "Welcome to tempfile.hs"
        putStrLn $ "I have a temporary file at " ++ tempname

        pos <- hTell temph
        putStrLn $ "My initial position is " ++ show pos

        let tempdata = show [1..10]
        putStrLn $ "Writing one line containing " ++
                   show (length tempdata) ++ " bytes: " ++
                   tempdata
        hPutStrLn temph tempdata
```

```
        pos <- hTell temph
        putStrLn $ "After writing, my new position is " ++ show pos

        putStrLn $ "The file content is: "
        hSeek temph AbsoluteSeek 0

        c <- hGetContents temph

        putStrLn c

        putStrLn $ "Which could be expressed as this Haskell literal:"
        print c

withTempFile :: String -> (FilePath -> Handle -> IO a) -> IO a
withTempFile pattern func =
    do
        tempdir <- catch (getTemporaryDirectory) (\_ -> return ".")
        (tempfile, temph) <- openTempFile tempdir pattern   -- pattern 就是"mytemp.txt"，
mytemp976.txt 里的数字是系统加的，而且每次都不同。

        finally (func tempfile temph)
                (do hClose temph
                    removeFile tempfile)
```

hPutStrLn 与 hPutStr 的区别是 hPutStrLn 带换行。

Lazy I/O

Haskell 是 lazy 语言，I/O 数据也仅在其值必须被 known 时才 evaluated 出来。

hGetContents

由于 lazy 特性，一次整个读 500 GB 大小的文件是可能的。

hGetContents 在你调用的时侯，实际上没有任何数据被读取。

用 lazy I/O 来处理 500G 的数据文件(后面有改进版)

```
{-
import System.IO
import Data.Char(toUpper)

main :: IO ()
main = do
        inh <- openFile "input.txt" ReadMode
        outh <- openFile "output.txt" WriteMode
        inpStr <- hGetContents inh
        let result = processData inpStr
        hPutStr outh result
        hClose inh
        hClose outh

processData :: String -> String
processData = map toUpper
-}
```

用 lazy I/O 来处理 500G 的数据文件

```
import System.IO
import Data.Char(toUpper)
main = do
        inh <- openFile "input.txt" ReadMode
        outh <- openFile "output.txt" WriteMode
        inpStr <- hGetContents inh
        hPutStr outh (map toUpper inpStr)
        hClose inh
```

hClose outh

readFile and writeFile

Haskell 程序员常用 hGetContents 作为过滤器。他们读一个文件，过滤某些内容，然后再将结果写到别的什么地方。实际上**使用 readFile 和 writeFile 是实现过滤器的更简洁的方法。**

readFile 在内部使用 hGetContents

用 lazy I/O 来处理 500G 的数据文件

```
-- ab.bat
runhaskell main
cmd
------------------------------
-- input.txt
hello,world!
------------------------------
-- main.hs
import Data.Char(toUpper)
main = do
    inpStr <- readFile "input.txt"
    writeFile "output.txt" (map toUpper inpStr)
```

运行 ab.bat，生成 output.txt，内容是：

HELLO,WORLD!

A Word on Lazy Output

略

interact

使用 interact 与用户交互

```
-- main.hs
import Data.Char(toUpper)

main = interact (map toUpper)

--  先运行批处理
-- runghc.bat
{-
@echo off
ghci main
-}
{-
*Main> main      --  然后，运行 main  函数
hello,world!    --  接着，输入字串
HELLO,WORLD!    --  最后，程序输出字串
-}

--加输入提示版

module Main where
import Data.Char(toUpper)
main = interact (map toUpper . (++) "Your data, in uppercase, is:\n\n")

-- 1. (++) "Your data, in uppercase, is:\n\n" :: [Char] -> [Char]
-- 2. map toUpper :: [Char] -> [Char]

-- 1  的输出正好可以 2  作为输入。

以上代码有个小问题，输入提示的部分也变成大写了。

module Main where
import Data.Char(toUpper)
main = interact ((++) "Your data, in uppercase, is:\n\n" .
                  map toUpper)

这里把提示字串移出 map  外了。
```

Filters with interact

```
main = interact (unlines . filter (elem 'a') . lines)

-- runghc filter.hs < input.txt
{-
I like Haskell
Haskell is great
-}
```

The IO Monad

如果要从键盘读一行，I/O 函数不可能每次都返回同样的结果对不对？**可以认为 I/O 就是改变世界的状态。**

Actions

action 类似函数。定义 **action** 的时侯它们什么也不做，在被 **invoked** 的时侯就会执行一些任务。

IO () 是一个 action

```
ghci> :type putStrLn
putStrLn :: String -> IO ()
ghci> :type getLine
getLine :: IO String
```

action 可以存储或传递到纯代码里。

```
runall :: [IO ()] -> IO ()
runall [] = return ()
runall (firstelem:remainingelems) =
    do firstelem
       runall remainingelems
abc = putStrLn "abc"    --  一个 action
main = do print "Start of the program"
          do abc
          print $ map show [1..10]    -- > ["1","2","3","4","5","6","7","8","9","10"]
          runall $ map (\s -> putStrLn ("Data: " ++ s) )
["1","2","3","4","5","6","7","8","9","10"]
          print "Done!"
```

“$” 的意思是给后面的所有东西加个括号。

可以这样认为：**do** 块中除了 **let** 外每一条语句都必须产生一个

I/O action 。

```
str2action :: String -> IO ()
str2action input = putStrLn ("Data: " ++ input)

list2actions :: [String] -> [IO ()]
list2actions = map str2action

numbers :: [Int]
numbers = [1..10]

strings :: [String]
strings = map show numbers

actions :: [IO ()]
actions = list2actions strings

printitall :: IO ()
printitall = runall actions

-- Take a list of actions, and execute each of them in turn.
runall :: [IO ()] -> IO ()
runall [] = return ()
runall (firstelem:remainingelems) =
    do firstelem
       runall remainingelems

main = do str2action "Start of the program"
          runall $ map (\s -> putStrLn ("Data: " ++ s) )
["1","2","3","4","5","6","7","8","9","10"]
          --printitall
          str2action "Done!"
```

加复数“s” 字尾用于提示“这是一个是列表”

这里的代码完成的功能是：数字 ->字串 ->action

使用 mapM_ 函数产生 I/O 输入

```
str2message :: String -> String
str2message input = "Data: " ++ input
```

```
str2action :: String -> IO ()
str2action = putStrLn . str2message

numbers :: [Int]
numbers = [1..10]

main = do str2action "Start of the program"
          mapM_ (str2action . show) numbers
          str2action "Done!"
```

mapM_ 类似 **map** ，接受一个 **I/O action** 函数作第一个参数，一个列表作为第二个参数。**mapM_** 会抛出那个 **I/O** 函数的结果。

```
ghci> :type mapM
mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]
ghci> :type mapM_
mapM_ :: (Monad m) => (a -> m b) -> [a] -> m ()
```

mapM ，**mapM** 这些函数实际上工作于任何 **Monad** 上。

带下划线的函数通常会丢弃它们的结果。

map 和 **mapM** 的区别是 **mapM** 会执行 **action** 。

Sequencing

do 块实际上就是将许多 **actions** “**joining together**” 的简便做法。有两个操作符可以用来代替 **do** 块。

用来代替 **do** 语句的运算符“>>”，“>>=”

```
ghci> :type (>>)
(>>) :: (Monad m) => m a -> m b -> m b
ghci> :type (>>=)
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

“>>” 运算符将两个 **action** 连起来。第一个 **action** 先执行，然后第二个 **action** 再执行。整个表达式的结果就是第二个 **action** 的结果，并且会丢弃第一个结果。

```
putStrLn "line 1" >> putStrLn "line 2"
```

“>>=” 运算符会返回一个 **action** ，并将这个 **action** 传给右边的表达式。

```
getLine >>= putStrLn
```

getLine 从 I/O 读一行，然后传给 putStrLn 打印出来。

```
main =
    putStrLn "Greetings! What is your name?" >>
    getLine >>=
        (\inpStr -> putStrLn $ "Welcome to Haskell, " ++ inpStr ++ "!")
```

The True Nature of Return

return 用于将数据封装进 Monad 。对 I/O ， return 会先获取纯数据，然后传给 I/O Monad 。

```
main =
    putStrLn "Greetings! What is your name?" >>
    return "guys" >>=
        (\inpStr -> putStrLn $ "Welcome to Haskell, " ++ inpStr ++ "!")
```

询问用户“yes” or "no"

```
{-

isGreen :: IO Bool
isGreen =
    do putStrLn "Is green your favorite color?"
        inpStr <- return "yes"    -- getLine
        return ((toUpper . head $ inpStr) == 'Y')
-}
```

将纯与不纯代码分开

```
isYes :: String -> Bool
isYes inpStr = (toUpper . head $ inpStr) == 'Y'

isGreen :: IO Bool
isGreen =
    do putStrLn "Is green your favorite color?"
        inpStr <- getLine
        return (isYes inpStr)
```

“<-” 操作符用于从 Monad 中取出纯数据

```
returnTest :: IO ()
returnTest =
    do one <- return 1
        let two = 2
```

```
        putStrLn $ show (one + two)

main = returnTest    -- >3
```

“let” 操作符用于在 do 块中定义纯代码(不带 action)

Is Haskell Really Imperative?

略

Side Effects with Lazy I/O

当我们说 Haskell 没有 side effects ，究竟意味着什么？恶劣的循环就算是纯代码也可能导致系统内存耗尽然后 crash。

纯函数不会修改全局变量，不会请求 I/O 。

hGetContents 不适合这些场合：与用户交互以获取数据，或从管道获取其它程序的数据。

Buffering

I/O 子系统是现代计算机最慢的部分。

操作系统会将最常用的数据放到内存中。

程序语言通常进行 buffering ，会从 OS 申请一大块内存，这意味着就算代码中只操作一个字节的数据也会占用那个大内存。

Buffering Modes

三种 **BufferMode** 类型： **NoBuffering, LineBuffering,** 和 **BlockBuffering** 。

NoBuffering 从 **OS** 一次读一个字符，一次写入一个字符(立即写入的)。性能很低，不适用于 **general-purpose use**。

LineBuffering 遇到换行符，或是数据量太大时就执行写操作。读入换行符之前的所有数据。当从终端读取时一遇到回车就立即返回数据。通常作为默放设置。

BlockBuffering 引起 **Haskell** 在可能的时侯读或写固定大小的数据。它有最佳性能，在处理成组的大数据时。接受一个 **Mabe** 参数，用 **Just 4096** 设置 **buffer** 为 **4096**，用 **Nothing** 设置默认 **buffer** 大小。

默认 **buffering** 模式依赖于操作系统和 **Haskell** 实现。

用 **hGetBuffering** 获取当前模式，用 **hSetBuffering** 设置 **buffer** 模式。便如 **hSetBuffering stdin (BlockBuffering Nothing)**

Flushing The Buffer

hFlush ，**hClose** 都会将 **buffer** 中的数据立即写入。

Reading Command-Line Arguments

System.Environment.getArgs 返回 **IO [String]**，列表里的每个元素对应命令行传过来的一个参数。

System.Environment.getProgName 用于获取程序名。

System.Console.GetOpt 提供了一些解析命令行选项的工具，对使用复杂选项的程序很有用。

Environment Variables

System.Environment: getEnv 或 **getEnvironment. getEnv** 这两个函数用于查找指定的变量，如果不存在就引发错误。

getEnvironment 返回整个环境**[(String, String)]**。**lookup** 函数用于查找特定的环境变量。

linux 中可以用 **System.Posix.Env** 模块中的 **putEnv** 或 **setEnv** 来设置环境变量。对 **Windows** ，**Haskell** 中不存在这种函数。

Chapter 8. Efficient File Processing, Regular Expressions, and Filename Matching

Efficient File Processing

用 **String** 来进行 **I/O** 操作性能很糟糕

```
-- in.txt
{-
10
11
12
-}
main = do
        contents <- readFile "c:/in.txt" -- getContents
        print (sumFile contents)
            where sumFile = sum . map read . words

{-
let numbers = ["10","11","12"]
let rr = (map read numbers)::[Int]   -- read 返因的类型是 a ，最后 map 返回的是[a]，所以要显示类型转换。
print rr
-}
```

注意：read 返回的类型是 **a** ，可以显示转成 **Int** ，**Dobule** 等，但转成 **String** 是会出错的。因为 **read** 就是用于解析的，读一个串，再转成一个串没有意义。

虽然 **String** 类型是读写文件的默认设置，但是效率不高。上面这段代码的性能非常糟糕。

[Char] 中的元素都是是单独分配，而且有一些 **bookkeeping overhead** 。

对于 **I/O** ，**bytestring** 库是一种比 **String** 更快，开销更小的选择。

Data.ByteString 库中的 **ByteString** 是真正的二进制或文本串**(**想像一下 **C** 语言的串**)**

Data.ByteString.Lazy 库中的 **ByteString** 是一个 **chunk** 的 **list(**注意这里 **list** 的含义，不知道是不是指列表的意思**)**，最大 **64 KB** 。

如果数据有数百 **MB** 或数百 **TB** ，则通常 **Data.ByteString.Lazy** 的性能最好。它的 **chunk size** 对现代 **CPU** 的 **L1 cache** 比较友好。垃圾收集器能很快地丢弃不再使用的 **chunk** 流数据。

Binary I/O and Qualified Imports

测试一个文件是否是 **linux** 可执行文件

```
-- 需要在 IDE 中加入 bytestring 包，保存后重启 IDE 才会生效。
module Main where

import qualified Data.ByteString.Lazy as L   -- 注意 as L 的写法
import Data.Word
```

```
hasElfMagic :: L.ByteString -> Bool
hasElfMagic content = L.take 4 content == elfMagic
      where elfMagic = L.pack [0x7f, 0x45, 0x4c, 0x46]

isElfFile :: FilePath -> IO Bool
isElfFile path = do
      content <- L.readFile path
      return (hasElfMagic content)

main = do
      let aa = 0x7f::Word8
      isElf <- isElfFile "c:/echo"
      print isElf
```

echo 是 linux 中的可执行文件。

Hakell 中用 Word8 来表示 byte 。

ByteString 一次最多读 64 K 的 chunk 。

lazy 的 ByteString 就是为 binary I/O 准备的。

Text I/O

prices.csv 文件中的数据记录了每月股票价格，如何找出最高收盘价？

计算最高股票收盘价

```
-- prices.csv
{-
Date,Open,High,Low,Close,Volume,Adj Close
2008-08-01,20.09,20.12,19.53,19.80,19777000,19.80
2008-06-30,21.12,21.20,20.60,20.66,17173500,20.66
2008-05-30,27.07,27.10,26.63,26.76,17754100,26.76
2008-04-30,27.17,27.78,26.76,27.41,30597400,27.41
-}

import qualified Data.ByteString.Lazy.Char8 as L

closing = readPrice . (!!4) . L.split ','    -- point-free style

readPrice :: L.ByteString -> Maybe Int
readPrice str =
    case L.readInt str of
        Nothing -> Nothing
        Just (dollars,rest) ->
            case L.readInt (L.tail rest) of
                Nothing -> Nothing
                Just (cents,more) ->
                    Just (dollars * 100 + cents)


highestClose = maximum . (Nothing:) . map closing . L.lines   -- 亮点，注意(Nothing:)
```

的用法，“:” 是一个列表构造器，左边已经有一个元素参数了，右边还需要一个列表参数。

```
highestCloseFrom path = do
    contents <- L.readFile path
    print (highestClose contents)
```

```
main = do
    contents <- L.readFile "c:/prices.csv"
    let byteStrs = L.lines contents
    print byteStrs -- >[Chunk "Date,Open,High,Low,Close,Volume,Adj Close\r" Empty,Chunk
"2008-08-01,20.09,20.12,19.53,19.80,19777000,19.80\r" Empty,Chunk
"2008-06-30,21.12,21.20,20.60,20.66,17173500,20.66\r" Empty,Chunk
"2008-05-30,27.07,27.10,26.63,26.76,17754100,26.76\r" Empty,Chunk
"2008-04-30,27.17,27.78,26.76,27.41,30597400,27.41" Empty]
    let byteStr = (!!1) byteStrs
    print byteStr   -- >Chunk "2008-08-01,20.09,20.12,19.53,19.80,19777000,19.80\r" Empty
    let byteStrs' = L.split ',' byteStr
    print byteStrs'   -- >[Chunk "2008-08-01" Empty,Chunk "20.09" Empty,Chunk "20.12"
Empty,Chunk "19.53" Empty,Chunk "19.80" Empty,Chunk "19777000" Empty,Chunk "19.80\r"
Empty]
    let byteStr' = (!!4) byteStrs'
    print byteStr'   -- >Chunk "19.80" Empty   -- 类型是 L.ByteString
    let price = readPrice byteStr'
    print price -- >Just 1980

    let prices =   map (readPrice.(!!4) . L.split ',') (L.lines contents)
    print prices   -- >[Nothing,Just 1980,Just 2066,Just 2676,Just 2741]   -- Nothing 的出现是
因为文本开头的那段英文

    let prices' = Nothing: prices   -- 亮点，加一个 Nothing 元素保证列表
不空，因为有些函数一遇上空列表就出错。
    print prices' -- >[Nothing,Nothing,Just 1980,Just 2066,Just 2676,Just 2741]

    let maxprice = maximum prices   -- maximum 接受一个有序集
    print maxprice -- > Just 2741


    highestCloseFrom "c:/prices.csv"
    print $ highestClose L.empty   -- 测试没有数据文件的情形   -- L.readFile L.empty 返
回 Nothing，maximum [Nothing] 返回 Nothing
    putStr =<< readFile "c:/prices.csv"   -- readFile 返回一个 monad ，“=<<” 运行符取出
monad 中的 String，然后传给 putStr 。
```

(!!4) 运算符从 list 中取第 4 个元素。

L.readInt 用于解析一个整数。返回一个整数和剩余 String 的

pair ， 并封装在 **Mabe** 中。

我们用了一个技巧来绕过这个事实，既不能将 **maximum** 应用于空列表。因为如果不存在股票数据时，我们不希望程序抛出异常。

“**(Nothing:)**” 表达式保证 **Maybe Int** 的列表不空。

因为已经将 I/O 和逻辑分开，所以我们能够测试没有数据文件的情形：print $ highestCloseL.empty

Filename Matching

三种模式：**glob patterns**，**wild card patterns** 和 **shell-style patterns**

“*”表示匹配任意字串

“?”表示匹配任意一个字符

“[]”表示匹配括号里的任意内容，可以在括号里加“!”表示“不是”。

“a–z”表示匹配 a 到 z

一个例子，pic[0-9].[pP][nN][gG]

Regular Expressions in Haskell

Haskell 的正则表达式库比其它语言要丰富。

“=~” 操作符大量用于多态，结果是造成这个操作符的签名很难理解，所以这里暂时不做介绍。

“=~” 操作符的两个参数都是 **typeclass**，而且返回值也是 **typeclass** 。左边的参数是要匹配的 **text** ，右边的参数是一个模式(正则表达式)。

“=~” 操作符的参数既可以是 **String** 也可以是 **ByteString** 。

The Many Types of Result

出错：“unable to load package `regex-posix-0.94.2” 的解决

{-
:module +Text.Regex.Posix

"your right hand" =~ "(hand|foot)" :: Bool

出错： : unable to load package `regex-posix-0.94.2'

cmd 下运行以下两个命令：

cabal update

cabal install regex-posix
-}

RegexContext 是一个 **typeclass** ，它描述了 **target** 类型应具有的行为。

Bool 类型是 **RegexContext** 的实例，**Int** 也是 **RegexContext** 的实例。

“**=~**” 是个多态模式匹配运算符，根据你所指定的返回值类型不同它完成的功能也不同。**Bool** 类型的返回值，它的功能是验证存在性；**Int** 类型的返回值，它的功能是计算发生成功匹配的次数。

Bool 返回值，“=~” 的功能是验证是否能够成功匹配

在整个表达式的后面加“:: Bool” 表示表达式的结果类型是 Bool ； 当 ghci 不能自动推断

结果类型时可以这样做。

使用“=~” 运算符测试模式是否可以成功匹配

"your right hand" =~ "(hand|foot)" :: Bool -- >True -- 可以成功匹配

Int 返回值，“=~” 的功能是计算发生成功匹配的次数

使用“=~” 运算符测试模式可以成功匹配多少次

"honorificabilitudinitatibus" =~ "[aeiou]" :: Int -- >13 -- **计算一个字串含有多少个元音**

String 返回值，“=~” 的功能是捕获第一个成功匹配的串，如匹配不能功就返回空串。

使用“=~” 运算符捕获第一个成功匹配的串

"I, B. Ionsonii, uurit a lift'd batch" =~ "(uu|ii)" :: String -- >"ii" -- 捕获第一个匹配

[String] 返回值，“=~” 的功能是捕获所有匹配，然后封装进 list 并返回

使用“=~” 运算符捕获所有成功匹配的串

"I, B. Ionsonii, uurit a lift'd batch" =~ "(uu|ii)" :: [String] -- >["ii","uu"]

Bool，Int，String，[String] 都是简单“result types”，还没完呢。

(String,String,String) 返回值，“=~” 的功能是返回第一次成功匹配的位置之前的串，成功匹配的串，成功匹配的位置之后的串。如果匹配失败，就只返回之前的串，后接两空串。

使用“=~” 运算符，以模式为准将字串分成三部分

```
pat = "(foo[a-z]*bar|quux)"
"before foodiebar after" =~ pat :: (String,String,String)
-- >("before ","foodiebar"," after")
```

使用“=~” 运算符，以模式为准将字串分成三部分，最后在加上[“成功匹配的串”]

```
pat = "(foo[a-z]*bar|quux)"
"before foodiebar after" =~ pat :: (String,String,String,[String])
-- >("before ","foodiebar"," after",["foodiebar"])
```

使用“=~” 运算符，计算第一次成功匹配串的位置和长度

```
-- pat = "(foo[a-z]*bar|quux)"
-- "i foobarbar a quux" =~ pat :: (Int,Int)
-- >(2,9)
```

使用“=~” 运算符，计算全部成功匹配串的位置和长度

```
-- pat = "(foo[a-z]*bar|quux)"
-- "i foobarbar a quux" =~ pat :: [(Int,Int)]    --  出错，待解决
-- >[(2,9),(14,4)]
```

还有更多 **RegexContext classtype** 的实例，请参见：**Text.Regex.Base.Context**

More About Regular Expressions

Mixing and Matching String Types

“=~” 是以 typeclass 作为其参数和返回值的类型。

```
-- :type pack "foo"
-- >pack "foo" :: ByteString
```

是否匹配？

```
-- pack "foo" =~ "bar" :: Bool   -- 左边用“ByteString” 参数，右边用“String” 参数(模式)
-- > false   -- 不匹配
```

匹配多少个？

```
"foo" =~ pack "bar" :: Int   -- 右边用“ByteString” 参数(模式)，左边用“String” 参数
-- >0   -- 零个
```

获得所有匹配的位置和长度

```
pack "foo" =~ pack "o" :: [(Int, Int)]   -- 错误，待解决
-- >[(1,1),(2,1)]
```

注意这种错误

```
pack "good food" =~ ".ood" :: [ByteString]   -- ok
"good food" =~ ".ood" :: [ByteString]   -- Error
"good food" =~ ".ood" :: [String]   -- OK
```

Other Things You Should Know

当你看 **Haskell** 库文档时会发现有许多正则表达式相关的模块。这些模块在 **Text.Regex.Base** 下定义了通用 **API** 接口。可以有多种实现同时存在，在本书写作时，**GHC** 以 **Text.Regex.Posix** 作为默认的正则表达式实现，这个包提供了 **POSIX** 正则语义。

其它的库可以从 **Hackage** 下载，其中一些比 **POSIX engine** 的性能要好，如 **regex-tdfa**。它们都有相同的接口。

Translating a glob Pattern into a Regular Expression

```
module GlobRegex (globToRegex, fnmatch) where

import Text.Regex.Posix ((=~))

globToRegex :: String -> String
globToRegex cs = '^' : globToRegex' cs ++ "$"

globToRegex' :: String -> String
globToRegex' "" = ""
globToRegex' ('*':cs) = ".*" ++ globToRegex' cs
globToRegex' ('?':cs) = '.' : globToRegex' cs
globToRegex' ('[':'!':c:cs) = "[^" ++ c : charClass cs
globToRegex' ('[':c:cs) = '[' : c : charClass cs
globToRegex' ('[':_) = error "unterminated character class"
globToRegex' (c:cs) = escape c ++ globToRegex' cs

escape :: Char -> String
escape c | c `elem` regexChars = '\\' : [c]
         | otherwise = [c]
  where regexChars = "\\+()^$.{}]|"

charClass :: String -> String
charClass (']':cs) = ']' : globToRegex' cs
charClass (c:cs) = c : charClass cs
charClass [] = error "unterminated character class"

matchesGlob:: FilePath -> String -> Bool
name `matchesGlob` pat = name =~ globToRegex pat


-- globToRegex "f??.c"
-- >"^f..\\.c$"
```

```
-- "foo.c" =~ globToRegex "f??.c" :: Bool
-- >True

-- "^f..\\.c$"  风格的模式
-- "foo.c" =~ "^f..\\.c$" :: Bool
-- >True
```

文件名匹配

```
fnmatch :: FilePath -> String -> Bool
name `fnmatch` pat = name =~ globToRegex pat

fnmatch "*.cvs" "abc.cvs"    -- > True
fnmatch "*.cvs" "abc.c"    -- > False
```

An important Aside: Writing Lazy Functions

我们看到 **globToRegex'** 是递归的，但不是尾递归的。

Making Use of Our Pattern Matcher

错误：“ **Ambiguous type variable `e' in the constraint:**” 的解决

```
{-

《Real World Haskell》 使用的是旧接口的 handle 函数。

需要将 import Control.Exception (handle) ，
改成：import Control.OldException (handle)
-}


System.FilePath 模块抽象了操作系统处理路径名的细节。
```

使用“</>” 函数将两个路径 **components** 连接起来

```
-- import System.FilePath
```

```
"foo" </> "bar"    -- >"foo\\bar"
```

使用 **dropTrailingPathSeparator** 去掉路径最后面的“/”

```
-- import System.FilePath
dropTrailingPathSeparator "foo/"    -- >"foo"
```

使用 **splitFileName** 函数把路径和文件名分开

```
-- import System.FilePath
splitFileName "foo/bar/Quux.hs"    -- >("foo/bar/","Quux.hs")
splitFileName "zippity"    -- >("","zippity")
```

forM 函数将它的第二个参数**(**一个 **action)**，作用于第一个参数**(**一个列表**)**，并返回 **result** 的列表。

const 函数总是返回第一个参数，不管第二个参数是什么。

Handling Errors Through API Design

略

Putting Our Code to Work

flip 函数用来改变另一个函数的参数顺序。

将 C 盘下的“*.cc” 文件全部改名成“*.cpp”

```
-- running.bat
@echo off
runghc main.hs && Pause

------------------------------------------------------------
-- main.hs
module Main where

import Glob
import RenameFiles


main = do
    -- 获得 C 盘下所有“.cc” 文件
    strs <- namesMatching "c:/*.cc"    -- ["c:/2.cc","c:/1.cc"]
    print strs

---------------------------------------------------------------------

-- GlobRegex.hs
module GlobRegex (globToRegex, matchesGlob) where

import Text.Regex.Posix ((=~))

globToRegex :: String -> String
globToRegex cs = '^' : globToRegex' cs ++ "$"

globToRegex' :: String -> String
globToRegex' "" = ""
globToRegex' ('*':cs) = ".*" ++ globToRegex' cs
globToRegex' ('?':cs) = '.' : globToRegex' cs
globToRegex' ('[':'!':c:cs) = "[^" ++ c : charClass cs
```

```
globToRegex' ('[':c:cs) = '[' : c : charClass cs
globToRegex' ('[':_) = error "unterminated character class"
globToRegex' (c:cs) = escape c ++ globToRegex' cs

escape :: Char -> String
escape c | c `elem` regexChars = '\\' : [c]
         | otherwise = [c]
  where regexChars = "\\+()^$.{}]|"

charClass :: String -> String
charClass (']':cs) = ']' : globToRegex' cs
charClass (c:cs) = c : charClass cs
charClass [] = error "unterminated character class"

matchesGlob :: FilePath -> String -> Bool
name `matchesGlob` pat = name =~ globToRegex pat


-- globToRegex "f??.c"
-- >"^f..\\.c$"

-- "foo.c" =~ globToRegex "f??.c" :: Bool
-- >True
```

"^f..\\.c$" 风格的模式

```
-- "foo.c" =~ "^f..\\.c$" :: Bool
-- >True
```

文件名匹配

```
-- matchesGlob "*.cvs" "abc.cvs"    -- > True
-- matchesGlob "*.cvs" "abc.c"    -- > False

--------------------------------------------------------------------------
-- Glob.hs

module Glob (
                    namesMatching
                ) where

import System.FilePath (dropTrailingPathSeparator, splitFileName, (</>))
```

```
import System.Directory (doesDirectoryExist, doesFileExist,
                                    getCurrentDirectory, getDirectoryContents)

import Control.OldException (handle)
import Control.Monad (forM)
import GlobRegex (matchesGlob)



isPattern :: String -> Bool
isPattern = any (`elem` "[*?")

namesMatching pat
    | not (isPattern pat) = do    -- 如果我们传的 String 不含 pattern ，就简单的检查给定的名字是否在文件系统中。
        exists <- doesNameExist pat
        return (if exists then [pat] else [])

    | otherwise = do        -- 处理 glob pattern
        case splitFileName pat of
          ("", baseName) -> do
              curDir <- getCurrentDirectory
              listMatches curDir baseName
          (dirName, baseName) -> do
              dirs <- if isPattern dirName
                          then namesMatching (dropTrailingPathSeparator dirName)
                          else return [dirName]
              let listDir = if isPattern baseName
                                then listMatches
                                else listPlain
              pathNames <- forM dirs $ \dir -> do
                                baseNames <- listDir dir baseName
                                return (map (dir </>) baseNames)
              return (concat pathNames)

doesNameExist :: FilePath -> IO Bool
doesNameExist name = do
    fileExists <- doesFileExist name
    if fileExists
        then return True
        else doesDirectoryExist name

listMatches :: FilePath -> String -> IO [String]
listMatches dirName pat = do
    dirName' <- if null dirName
```

```
                    then getCurrentDirectory
                    else return dirName
    handle (const (return [])) $ do
        names <- getDirectoryContents dirName'
        let names' = if isHidden pat
                        then filter isHidden names
                        else filter (not . isHidden) names
        return (filter (`matchesGlob` pat) names')

isHidden ('.':_) = True
isHidden _ = False

listPlain :: FilePath -> String -> IO [String]
listPlain dirName baseName = do
    exists <- if null baseName
                then doesDirectoryExist dirName
                else doesNameExist (dirName </> baseName)
    return (if exists then [baseName] else [])

--------------------------------------------------------------------------
-- RenameFiles.hs

module RenameFiles (
    --rename
   cc2cpp
) where

import System.FilePath (replaceExtension)
import System.Directory (doesFileExist, renameDirectory, renameFile)
import Glob (namesMatching)


renameWith :: (FilePath -> FilePath)
            -> FilePath
            -> IO FilePath

renameWith f path = do
    let path' = f path
    rename path path'
    return path'

rename :: FilePath -> FilePath -> IO ()
rename old new = do
    isFile <- doesFileExist old
```

```
    let f = if isFile then renameFile else renameDirectory
    f old new

cc2cpp =
    mapM (renameWith (flip replaceExtension ".cpp")) =<< namesMatching "*.cc"
```

flip 函数用来改变另一个函数的参数顺序。

获得 C 盘下所有“.cc” 文件

```
-- namesMatching "c:/*.cc"

-- strs <- namesMatching "c:/*.cc"    -- >["c:/2.cc","c:/1.cc"]
-- replaceExtension :: FilePath -> String -> FilePath
-- replaceExtension "c:/1.cc" ".cpp"    -- >"c:/1.cpp"
-- flip replaceExtension ".cpp" "c:/1.cc"      -- >"c:/1.cpp"
-- (flip replaceExtension ".cpp") "c:/1.cc"    -- >"c:/1.cpp"
-- rename "c:/1.cc" "c:/1.cpp"    -- 1.cc 被改名成 1.cpp
```

Chapter 9. I/O Case Study: A Library for Searching the Filesystem

The find Command

Haskell 中参数位置非常重要，如果把参数放在了错误的位置上就会失去 **partial application** 的 **gives** 了。

给定一个目录的列表，**find** 命令递归的搜索每一个目录，并打印所有匹配的文件名。

搜索条件可以是名字与“**glob pattern**”匹配的，**entry** 是普通文件，最后修改日期等。利用“**and**”，“**or**”操作符，这些条件还可以组合使用，形成更复杂的条件。

Starting Simple: Recursively Listing a Directory

递归列出一个目录及其子目录的内容

filter 语句确保一个目录的 listing 不含特殊的目录名“.” 或“..”，如果我们忘记过滤这些就会造成无限递归。

```
{-
ghci> :m +Control.Monad
ghci> :type mapM
mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]
ghci> :type forM
forM :: (Monad m) => [a] -> (a -> m b) -> m [b]
-}
```

循环体检查当前项是否是一个目录，如果是就递归地调用 getRecursiveContents 去 list 那个目录。否则，它 return 当前 entry 的名字。(不要忘记 return 的含义，它将东西封装进 Monad)

递归打印目录及其子目录的内容

```
module RecursiveContents (getRecursiveContents) where

import Control.Monad (forM)
import System.Directory (doesDirectoryExist, getDirectoryContents)
import System.FilePath ((</>))

getRecursiveContents :: FilePath -> IO [FilePath]
getRecursiveContents topdir = do
    names <- getDirectoryContents topdir
    let properNames = filter (`notElem` [".", ".."]) names
    paths <- forM properNames $ \name -> do
        let path = topdir </> name
        isDirectory <- doesDirectoryExist path
        if isDirectory
            then getRecursiveContents path
            else return [path]
    return (concat paths)

-- contents <- getRecursiveContents "c:/haskell"
-- >["c:/haskell\\hello.lkshw","c:/haskell\\hello.lkshs",...]
```

Revisiting Anonymous and Named Functions

尽管在前面的章节中我们列出了一些不要使用匿名函数的理由，但在这里我们在循环体中使用了匿名函数。

在循环体中使用匿名函数是 **Haskell** 中的匿名函最常见的应用。

forM 和 mapM 接受一个函数作为参数，多数循环体中的代码块在程序中只出现一次。既然只在一个地方使用就没必要给它命名。

Why Provide Both mapM and forM?

如果循环很短，但是 **data** 很长，就用 **mapM**。

如果 **data** 很短，但是循环很长，就用 **forM**。

如果都很长，就是 **let** 和 **where** 将其中一个变短。

A Naive Finding Function

takeExtension 函数用于从文件名中提取扩展名

文件查找器

```
-- SimpleFinder.hs
module SimpleFinder (simpleFind) where

import RecursiveContents (getRecursiveContents)

simpleFind :: (FilePath -> Bool) -> FilePath -> IO [FilePath]
simpleFind p path = do
      names <- getRecursiveContents path
      return (filter p names)

-------------------------------------------------
-- main.hs
import SimpleFinder
import System.FilePath

files <- simpleFind (\p -> takeExtension p == ".hs") "c:/haskell"
-- > ["c:/haskell\\hello\\src\\SimpleFinder.hs","c:/haskell\\hello\\src\\Setup.hs",...]
```

这个程序存在很多问题，比如不能区分以".hs" 结尾的目录和文件，文件系统的遍历方式不受控制等等，这些问题我们将会在后面的章节中解决。

Predicates: From Poverty to Riches, While Remaining Pure

使用 **doesFileExist** 和 **doesDirectoryExist** 函数来判断一个 **entry** 是文件还是目录。

使用 **getPermissions** 函数来检测一个对文或目录操作是否合法

使用 **getModificationTime** 函数来获取一个 **entry** 的最后修改时间。

System.Posix 和 **System.Win32** 模块提供了特定操作系统的更多功能。**Hackage** 上还存在 **unix-compat** 这个库，为 **Windows** 提供了类 **Unix API** 的功能。

```
type Predicate = FilePath -- path to directory entry
                 -> Permissions -- permissions
                 -> Maybe Integer -- file size (Nothing if not file)
                 -> ClockTime -- last modified
                 -> Bool
```

Predicate 就是一个接受四参数的函数，这样写是为了节约击键数和空间。

filterM 函数与 **filter** 类似，不同的是 **filterM** 允许其布尔函数执行 **I/O** 。

Sizing a File Safely

获取文件大小(后面有改进版)

```
{-
simpleFileSize :: FilePath -> IO Integer
simpleFileSize path = do
    h <- openFile path ReadMode
    size <- hFileSize h
    hClose h
    return size
-}
```

获取文件大小(后面有改进版)

```
{-
saferFileSize :: FilePath -> IO (Maybe Integer)
saferFileSize path = handle (\_ -> return Nothing) $ do
    h <- openFile path ReadMode
    size <- hFileSize h
    hClose h
    return (Just size)
-}
```

The Acquire-Use-Release Cycle

获取文件大小

```
getFileSize :: FilePath -> IO (Maybe Integer)
getFileSize path = handle (\_ -> return Nothing) $
    bracket (openFile path ReadMode) hClose $ \h -> do
        size <- hFileSize h
        return (Just size)
```

Control.Exception 模块提供了一个 **bracket** 函数，这个函数以三个 **action** 作为参数。

bracket 函数的第一个参数会产生一个 **source** ，第二个参数会

释放那个 **source**，第三个参数会使用那个 **source** 。

A Domain-Specific Language for Predicates

```
-- 待改进
{-
myTest path _ (Just size) _ =
      takeExtension path == ".cpp" && size > 131072
myTest _ _ _ _ = False
-}

type InfoP a = FilePath -- path to directory entry
                -> Permissions -- permissions
                -> Maybe Integer -- file size (Nothing if not file)
                -> ClockTime -- last modified
                -> a

pathP :: InfoP FilePath
pathP path _ _ _ = path

sizeP :: InfoP Integer
sizeP _ _ (Just size) _ = size
sizeP _ _ Nothing _ = -1

equalP :: (Eq a) => InfoP a -> a -> InfoP Bool
equalP f k = \w x y z -> f w x y z == k

equalP' :: (Eq a) => InfoP a -> a -> InfoP Bool
equalP' f k w x y z = f w x y z == k
```

Avoiding Boilerplate with Lifting

```
liftP :: (a -> b -> c) -> InfoP a -> b -> InfoP c
liftP q f k w x y z = f w x y z `q` k

greaterP, lesserP :: (Ord a) => InfoP a -> a -> InfoP Bool
greaterP = liftP (>)
lesserP = liftP (<)
```

Haskell 中参数位置非常重要，如果把参数放在了错误的位置上

就会失去 **partial application** 的 **gives** 了。

Gluing Predicates Together

Haskell 中我们将那些以其它函数作为参数，并返回新函数的函数称为 **combinators** 。

```
simpleAndP :: InfoP Bool -> InfoP Bool -> InfoP Bool
simpleAndP f g w x y z = f w x y z && g w x y z

liftP2 :: (a -> b -> c) -> InfoP a -> InfoP b -> InfoP c
liftP2 q f g w x y z = f w x y z `q` g w x y z

andP = liftP2 (&&)
orP = liftP2 (||)

constP :: a -> InfoP a
constP k _ _ _ _ = k

liftP' q f k w x y z = f w x y z `q` constP k w x y z

myTest path _ (Just size) _ =
    takeExtension path == ".cpp" && size > 131072
myTest _ _ _ _ = False

liftPath :: (FilePath -> a) -> InfoP a
liftPath f w _ _ _ = f w

myTest2 = (liftPath takeExtension `equalP` ".cpp") `andP`
    (sizeP `greaterP` 131072)
```

Defining and Using New Operators

```
(==?) = equalP
(&&?) = andP
(>?) = greaterP

myTest3 = (liftPath takeExtension ==? ".cpp") &&? (sizeP >? 131072)
```

```
infix 4 ==?
infixr 3 &&?
infix 4 >?
myTest4 = liftPath takeExtension ==? ".cpp" &&? sizeP >? 131072
```

Controlling Traversal

```
data Info = Info {
      infoPath :: FilePath
    , infoPerms :: Maybe Permissions
    , infoSize :: Maybe Integer
    , infoModTime :: Maybe ClockTime
    } deriving (Eq, Ord, Show)

traverse :: ([Info] -> [Info]) -> FilePath -> IO [Info]
traverse order path = do
      names <- getUsefulContents path
      contents <- mapM getInfo (path : map (path </>) names)
      liftM concat $ forM (order contents) $ \info -> do
      if isDirectory info && infoPath info /= path
          then traverse order (infoPath info)
          else return [info]

getUsefulContents :: FilePath -> IO [String]
getUsefulContents path = do
      names <- getDirectoryContents path
      return (filter (`notElem` [".", ".."]) names)

isDirectory :: Info -> Bool
isDirectory = maybe False searchable . infoPerms

maybeIO :: IO a -> IO (Maybe a)
maybeIO act = handle (\_ -> return Nothing) (Just `liftM` act)

getInfo :: FilePath -> IO Info
getInfo path = do
      perms <- maybeIO (getPermissions path)
      size <- maybeIO (bracket (openFile path ReadMode) hClose hFileSize)
      modified <- maybeIO (getModificationTime path)
      return (Info path perms size modified)
```

infos <- getInfo "c:/haskell\\hello.lkshw"

-- >Info {infoPath = "c:/haskell\\hello.lkshw", infoPerms = Just (Permissions {readable = True, writable = True, executable = False, searchable = False}), infoSize = Just 364, infoModTime = Just Sun Feb 27 14:23:54 ÖÐ¹ú±ê×¼Ê±¼ä 2011}

Density, Readability, and the Learning Process

```
traverseVerbose order path = do
      names <- getDirectoryContents path
      let usefulNames = filter (`notElem` [".", ".."]) names
      contents <- mapM getEntryName ("" : usefulNames)
      recursiveContents <- mapM recurse (order contents)
      return (concat recursiveContents)
   where getEntryName name = getInfo (path </> name)
            isDirectory info = case infoPerms info of
                                    Nothing -> False
                                    Just perms -> searchable perms
            recurse info = do
                  if isDirectory info && infoPath info /= path
                     then traverseVerbose order (infoPath info)
                     else return [info]
```

Another Way of Looking at Traversal

```
data Iterate seed = Done { unwrap :: seed }
                  | Skip { unwrap :: seed }
                  | Continue { unwrap :: seed }
                    deriving (Show)

type Iterator seed = seed -> Info -> Iterate seed

foldTree :: Iterator a -> a -> FilePath -> IO a
foldTree iter initSeed path = do
      endSeed <- fold initSeed path
      return (unwrap endSeed)
   where
      fold seed subpath = getUsefulContents subpath >>= walk seed

      walk seed (name:names) = do
            let path' = path </> name
            info <- getInfo path'
            case iter seed info of
                  done@(Done _) -> return done
                  Skip seed' -> walk seed' names
```

```
                Continue seed'
                    | isDirectory info -> do
                        next <- fold seed' path'
                        case next of
                            done@(Done _) -> return done
                            seed'' -> walk (unwrap seed'') names
                    | otherwise -> walk seed' names
        walk seed _ = return (Continue seed)


atMostThreePictures :: Iterator [FilePath]
atMostThreePictures paths info
        | length paths == 3
          = Done paths
        | isDirectory info && takeFileName path == ".svn"
          = Skip paths
        | extension `elem` [".jpg", ".png"]
          = Continue (path : paths)
        | otherwise
          = Continue paths
    where extension = map toLower (takeExtension path)
          path = infoPath info

countDirectories count info =
        Continue (if isDirectory info
                  then count + 1
                  else count)
```

这里传给 foldTree 的初始 seed 应是 0 。

Useful Coding Guidelines

略

Common Layout Styles

略

检测给定的目录下有多少个目录(非递归)

-- foldTree countDirectories 0 "c:/haskell" >>= print -- >1 -- 如果 haskell 目录下有两个目录就会输出 2
-- getUsefulContents "c:/haskell" >>= print -- ["hello.lkshw","hello.lkshs","hello"]
-- walk 0 ("hello.lkshw":["hello.lkshs","hello"])
-- let path' = "c:/haskell" </> "hello.lkshw" -- "c:/haskell\\hello.lkshw"
-- info <- getInfo path' -- Info {infoPath = "c:/haskell\\hello.lkshw", infoPerms = Just (Permissions {readable = True, writable = True, executable = False, searchable = False}), infoSize = Just 364, infoModTime = Just Sun Feb 27 14:23:54 ÖÐ¹ú±ê×¼Ê±¼ä 2011}
-- countDirectories 0 info -- Continue {unwrap = 0}
-- isDirectory info -- False
-- walk 0 ["hello.lkshs","hello"]

获取给定目录下所有以“.bat” 结尾的文件(递归?)

-- foldTree atMostThreePictures [] "C:/haskell\\hello\\src" >>= print -- ["C:/haskell\\hello\\src\\RenameFiles.hs","C:/haskell\\hello\\src\\Setup.hs","C:/haskell\\hello\\src\\SimpleFinder.hs"]

-- BetterPredicate
---
module BetterPredicate (betterFind, getInfo, Info(..), countDirectories, isDirectory, getInfo, foldTree, atMostThreePictures, getUsefulContents) where

import Control.Monad (filterM, forM, liftM)
import System.Directory (Permissions(..), getModificationTime, getPermissions, getDirectoryContents)
import System.Time (ClockTime(..))
import System.FilePath --(takeExtension)
import Control.OldException (bracket, handle)
import System.IO (IOMode(..), hClose, hFileSize, openFile)
import Data.Char

-- the function we wrote earlier
import RecursiveContents (getRecursiveContents)

type Predicate = FilePath -- path to directory entry
  -> Permissions -- permissions
  -> Maybe Integer -- file size (Nothing if not file)
  -> ClockTime -- last modified

```
                    -> Bool

getFileSize :: FilePath -> IO (Maybe Integer)
getFileSize path = handle (\_ -> return Nothing) $
    bracket (openFile path ReadMode) hClose $ \h -> do
        size <- hFileSize h
        return (Just size)

betterFind :: Predicate -> FilePath -> IO [FilePath]
betterFind p path = getRecursiveContents path >>= filterM check
    where check name = do
                perms <- getPermissions name
                size <- getFileSize name
                modified <- getModificationTime name
                return (p name perms size modified)

type InfoP a = FilePath -- path to directory entry
                -> Permissions -- permissions
                -> Maybe Integer -- file size (Nothing if not file)
                -> ClockTime -- last modified
                -> a

pathP :: InfoP FilePath
pathP path _ _ _ = path

sizeP :: InfoP Integer
sizeP _ _ (Just size) _ = size
sizeP _ _ Nothing _ = -1

equalP :: (Eq a) => InfoP a -> a -> InfoP Bool
equalP f k = \w x y z -> f w x y z == k

equalP' :: (Eq a) => InfoP a -> a -> InfoP Bool
equalP' f k w x y z = f w x y z == k



liftP :: (a -> b -> c) -> InfoP a -> b -> InfoP c
liftP q f k w x y z = f w x y z `q` k

greaterP, lesserP :: (Ord a) => InfoP a -> a -> InfoP Bool
greaterP = liftP (>)
lesserP = liftP (<)
```

```
simpleAndP :: InfoP Bool -> InfoP Bool -> InfoP Bool
simpleAndP f g w x y z = f w x y z && g w x y z

liftP2 :: (a -> b -> c) -> InfoP a -> InfoP b -> InfoP c
liftP2 q f g w x y z = f w x y z `q` g w x y z

andP = liftP2 (&&)
orP = liftP2 (||)

constP :: a -> InfoP a
constP k _ _ _ _ = k

liftP' q f k w x y z = f w x y z `q` constP k w x y z

myTest path _ (Just size) _ =
      takeExtension path == ".cpp" && size > 131072
myTest _ _ _ _ = False

liftPath :: (FilePath -> a) -> InfoP a
liftPath f w _ _ _ = f w

myTest2 = (liftPath takeExtension `equalP` ".cpp") `andP`
      (sizeP `greaterP` 131072)



(==?) = equalP
(&&?) = andP
(>?) = greaterP

myTest3 = (liftPath takeExtension ==? ".cpp") &&? (sizeP >? 131072)

infix 4 ==?
infixr 3 &&?
infix 4 >?
myTest4 = liftPath takeExtension ==? ".cpp" &&? sizeP >? 131072

data Info = Info {
      infoPath :: FilePath
   , infoPerms :: Maybe Permissions
   , infoSize :: Maybe Integer
   , infoModTime :: Maybe ClockTime
   } deriving (Eq, Ord, Show)

traverse :: ([Info] -> [Info]) -> FilePath -> IO [Info]
```

```
traverse order path = do
      names <- getUsefulContents path
      contents <- mapM getInfo (path : map (path </>) names)
      liftM concat $ forM (order contents) $ \info -> do
      if isDirectory info && infoPath info /= path
          then traverse order (infoPath info)
          else return [info]

getUsefulContents :: FilePath -> IO [String]
getUsefulContents path = do
      names <- getDirectoryContents path
      return (filter (`notElem` [".", ".."]) names)

isDirectory :: Info -> Bool
isDirectory = maybe False searchable . infoPerms

maybeIO :: IO a -> IO (Maybe a)
maybeIO act = handle (\_ -> return Nothing) (Just `liftM` act)

getInfo :: FilePath -> IO Info
getInfo path = do
      perms <- maybeIO (getPermissions path)
      size <- maybeIO (bracket (openFile path ReadMode) hClose hFileSize)
      modified <- maybeIO (getModificationTime path)
      return (Info path perms size modified)

traverseVerbose order path = do
      names <- getDirectoryContents path
      let usefulNames = filter (`notElem` [".", ".."]) names
      contents <- mapM getEntryName ("" : usefulNames)
      recursiveContents <- mapM recurse (order contents)
      return (concat recursiveContents)
   where getEntryName name = getInfo (path </> name)
           isDirectory info = case infoPerms info of
                                   Nothing -> False
                                   Just perms -> searchable perms
           recurse info = do
                if isDirectory info && infoPath info /= path
                    then traverseVerbose order (infoPath info)
                    else return [info]

data Iterate seed = Done { unwrap :: seed }
                  | Skip { unwrap :: seed }
                  | Continue { unwrap :: seed }
```

```
                    deriving (Show)

type Iterator seed = seed -> Info -> Iterate seed

foldTree :: Iterator a -> a -> FilePath -> IO a
foldTree iter initSeed path = do
    endSeed <- fold initSeed path
    return (unwrap endSeed)
  where
    fold seed subpath = getUsefulContents subpath >>= walk seed
```

-- 亮点，**fold** 的作用是帮其它两函数 **holding** 参数

```
    walk seed (name:names) = do
        let path' = path </> name
        info <- getInfo path'
        case iter seed info of
            done@(Done _) -> return done
            Skip seed' -> walk seed' names
            Continue seed'
                | isDirectory info -> do
                    next <- fold seed' path'
                    case next of
                        done@(Done _) -> return done
                        seed'' -> walk (unwrap seed'') names
                | otherwise -> walk seed' names
    walk seed _ = return (Continue seed)


atMostThreePictures :: Iterator [FilePath]
atMostThreePictures paths info
    | length paths == 3
```

-- 控制深度?

```
      = Done paths
    | isDirectory info && takeFileName path == ".svn"
      = Skip paths
    | extension `elem` [".bat", ".png"]
```

-- 搜索条件

```
      = Continue (path : paths)
    | otherwise
      = Continue paths
  where extension = map toLower (takeExtension path)
        path = infoPath info
```

```
countDirectories count info =
        Continue (if isDirectory info
                        then count + 1
                        else count)
```

Chapter 10. Code Case Study: Parsing a Binary Data Format

Grayscale Files

(==>) 函数会创建一个闭包，如“**(+5)**”就是一个闭包。

functor 可以使得我们的代码更 **tidy**，更 **expressive** 。

functor 可以避免去复制代码。

functor 有助于避免代码冗长。

PMG 图像有两种格式，**p2(ASCII** 编码**)** 和 **p5(**多为二进制编码**)** 。

文件头是这样的：一个 **string("p2"** 或 **"p5") +** 一个 **whitespace +** 三个数字**(**分别表示宽、高，和最大灰度值**)** ，并且中间用 **whitespace** 分隔。

文件头后是 **image data**，可能是二进制数据**(raw file)**，也可能是 **ASCII** 数字**(plain file)**，并以单个空格分开。

raw file 在单个文件中可以包含多个图像，每一个都有自己的文件头。**plain file** 只包含单个图像。

Parsing a Raw PGM File

```
module PNM () where

import qualified Data.ByteString.Lazy.Char8 as L8
import qualified Data.ByteString.Lazy as L
import Data.Char (isSpace)

data Greymap = Greymap {
             greyWidth :: Int
           , greyHeight :: Int
           , greyMax :: Int
           , greyData :: L.ByteString
           } deriving (Eq)
```

阻止编译器自动继承 **Show** 实例，因为图像数据非常巨大。

```
instance Show Greymap where   -- 亮点，自己实现 Show 的实例，不打印过大的字段
      show (Greymap w h m _) = "Greymap " ++ show w ++ "x" ++ show h ++
                                          " " ++ show m

matchHeader :: L.ByteString -> L.ByteString -> Maybe L.ByteString
matchHeader prefix str
      | prefix `L8.isPrefixOf` str
            = Just (L8.dropWhile isSpace (L.drop (L.length prefix) str))
      | otherwise
            = Nothing

getNat :: L.ByteString -> Maybe (Int, L.ByteString)
getNat s = case L8.readInt s of
                  Nothing -> Nothing
                  Just (num,rest)
                        | num <= 0 -> Nothing
                        | otherwise -> Just (fromIntegral num, rest)

getBytes :: Int -> L.ByteString -> Maybe (L.ByteString, L.ByteString)
getBytes n str = let count = fromIntegral n
                                 both@(prefix,_) = L.splitAt count str
```

```
                    in if L.length prefix < count
                        then Nothing
                        else Just both


parseP5 :: L.ByteString -> Maybe (Greymap, L.ByteString)
parseP5 s =
    case matchHeader (L8.pack "P5") s of
        Nothing -> Nothing
        Just s1 ->
            case getNat s1 of
                Nothing -> Nothing
                Just (width, s2) ->
                    case getNat (L8.dropWhile isSpace s2) of
                        Nothing -> Nothing
                        Just (height, s3) ->
                            case getNat (L8.dropWhile isSpace s3) of
                                Nothing -> Nothing
                                Just (maxGrey, s4)
                                    | maxGrey > 255 -> Nothing
                                    | otherwise ->
                                        case getBytes 1 s4 of
                                            Nothing -> Nothing
                                            Just (_, s5) ->
                                                case getBytes (width * height) s5 of
                                                    Nothing -> Nothing
                                                    Just (bitmap, s6) ->
                                                        Just (Greymap width height
maxGrey bitmap, s6)
```

Getting Rid of Boilerplate Code

如果**(>>?)** 函数左边得到的不是 **Nothing** 就将它作为参数传给右边的函数，否则就什么也不做。使用这个操作符可以将函数链起来。

```
(>>?) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>? _ = Nothing
Just v >>? f = f v
```

解析函数第二次尝试

```
parseP5_take2 :: L.ByteString -> Maybe (Greymap, L.ByteString)
parseP5_take2 s =
      matchHeader (L8.pack "P5") s >>?
      \s -> skipSpace ((), s) >>?
      (getNat . snd) >>?
      skipSpace >>?
      \(width, s) -> getNat s >>?
      skipSpace >>?
      \(height, s) -> getNat s >>?
      \(maxGrey, s) -> getBytes 1 s >>?
      (getBytes (width * height) . snd) >>?
      \(bitmap, s) -> Just (Greymap width height maxGrey bitmap, s)

skipSpace :: (a, L.ByteString) -> Maybe (a, L.ByteString)
skipSpace (a, s) = Just (a, L8.dropWhile isSpace s)
```

Implicit State

```
data ParseState = ParseState {
      string :: L.ByteString
    , offset :: Int64 -- imported from Data.Int
    } deriving (Show)

simpleParse :: ParseState -> (a, ParseState)
simpleParse = undefined
```

这个解析器能够报告错误

```
betterParse :: ParseState -> Either String (a, ParseState)
betterParse = undefined
```

为了不把内部实现暴露给用户所以用 **newtype** 包装一下

```
newtype Parse a = Parse {
            runParse :: ParseState -> Either String (a, ParseState)
      }
```

newtype 只是编译时封装一个函数，所以没有运行时 **overhead** 。当我们想要使用那个函数，就用 **runParser** 这个访问器。

不暴露 **Parse** 的值构造器可以确保没有人会意外的创建一个解析器，也不能通过模式匹配检查其内部。

The Identity Parser

```
identity :: a -> Parse a
identity a = Parse (\s -> Right (a, s))

parse :: Parse a -> L.ByteString -> Either String a
parse parser initState
      = case runParse parser (ParseState initState 0) of
            Left err -> Left err
            Right (result, _) -> Right result
```

Record Syntax, Updates, and Pattern Matching

记录语法比访问器更有用，我们可以用它拷贝或部份改变已存在的值

```
modifyOffset :: ParseState -> Int64 -> ParseState
modifyOffset initState newOffset =
      initState { offset = newOffset }

{-
ghci> let before = ParseState (L8.pack "foo") 0
ghci> let after = modifyOffset before 3
ghci> before   -- 亮点，查看前一个结果
ParseState {string = Chunk "foo" Empty, offset = 0}
ghci> after    -- 亮点，查看后一个结果
```

```
ParseState {string = Chunk "foo" Empty, offset = 3}
-}
```

A More Interesting Parser

```
parseByte :: Parse Word8
parseByte =
      getState ==> \initState ->
      case L.uncons (string initState) of
            Nothing ->
                  bail "no more input"
            Just (byte,remainder) ->
                  putState newState ==> \_ ->
                        identity byte
             where newState = initState { string = remainder,
                                          offset = newOffset }
                   newOffset = offset initState + 1

getState :: Parse ParseState
getState = Parse (\s -> Right (s, s))

putState :: ParseState -> Parse ()
putState s = Parse (\_ -> Right ((), s))

bail :: String -> Parse a
bail err = Parse $ \s -> Left $
                "byte offset " ++ show (offset s) ++ ": " ++ err

(==>) :: Parse a -> (a -> Parse b) -> Parse b
firstParser ==> secondParser = Parse chainedParser
      where chainedParser initState =
                  case runParse firstParser initState of
                        Left errMessage ->
                              Left errMessage
                        Right (firstResult, newState) ->
                              runParse (secondParser firstResult) newState
```

L8.uncons 从 ByteString 中取一个元素

```
{-
ghci> L8.uncons (L8.pack "foo")
```

```
Just ('f',Chunk "oo" Empty)
ghci> L8.uncons L8.empty
Nothing
-}
```

Obtaining and Modifying the Parse State

```
{-
bail :: String -> Parse a
bail err = Parse $ \s -> Left $
      "byte offset " ++ show (offset s) ++ ": " ++ err
-}
```

Chaining Parsers Together

```
{-
(==>) :: Parse a -> (a -> Parse b) -> Parse b
firstParser ==> secondParser = Parse chainedParser
      where chainedParser initState =
                  case runParse firstParser initState of
                        Left errMessage ->
                              Left errMessage
                        Right (firstResult, newState) ->
                              runParse (secondParser firstResult) newState
-}
```

(==>) 函数会创建一个闭包，如“(+5)”就是一个闭包。

Introducing Functors

fmap 是泛型 map ，针对不同的类型实现不同的 fmap 。

```
{-
map (1+) [1,2,3]    -- >[2,3,4]
map (+2) [1,2,3]    -- >[3,4,5]
-}
```

map-like activity 在其他例子中也很有用。例如考虑一个二叉树：

```
data Tree a = Node (Tree a) (Tree a)
                | Leaf a
                  deriving (Show)
```

如果我们希望将一颗 strings 树映射成一颗 strings length 树(既原来每个结点 string 用它的长度替换)，我们可以写这样的函数：

```
treeLengths (Leaf s) = Leaf (length s)
treeLengths (Node l r) = Node (treeLengths l) (treeLengths r)
```

改写一下，使其更具通用性

```
treeMap :: (a -> b) -> Tree a -> Tree b
treeMap f (Leaf a) = Leaf (f a)
treeMap f (Node l r) = Node (treeMap f l) (treeMap f r)
```

参数是作用于结点的函数和一颗树。

两个函数输出是一样的

```
{-
let tree = Node (Leaf "foo") (Node (Leaf "x") (Leaf "quux"))
print $ treeLengths tree    -- >Node (Leaf 3) (Node (Leaf 1) (Leaf 4))
print $ treeMap length tree    -- >Node (Leaf 3) (Node (Leaf 1) (Leaf 4))
print $ treeMap (odd . length) tree    -- >Node (Leaf True) (Node (Leaf True) (Leaf False))    -- 亮点，复合函数的妙用
-}
```

Haskell 提供了一个名为 Functor 的著名 typeclass ，完成类似 treeMap 的功能。

Functor 这个 typeclass 提供了一个 fmap 函数(位于 GHC.Base 库)

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

树的 fmap 实现

```
instance Functor Tree where
	fmap = treeMap
```

这里一定义就暴露给外部了，不需要导出。

```
-- fmap length (Node (Leaf "Livingstone") (Leaf "I presume"))    -- >Node (Leaf 11) (Leaf 9)
```

列表的 fmap 实现(位于标准库)

```
{-
instance Functor [] where
	fmap = map
-}
```

Mabe 的 fmap 实现(位于标准库)

```
{-
instance Functor Maybe where
	fmap _ Nothing = Nothing
	fmap f (Just x) = Just (f x)
-}

{-
fmap (+1) [1,2,3]    -- >[2,3,4]
map (+1) [1,2,3]    -- >[2,3,4]

-- fmap 相比 map 有什么特别的地方？
```

```
-}
```

Functor 的实例只能是那些恰好只有一个类型参数的类型，**Either a b** 或 **(a, b)** 有两个类型参数所以不能用来实例化 **Functor** ，**Bool** 或 **Int** 也不行，因为它们没有类型参数。

```
data Foo a = Foo a
instance Functor Foo where
    fmap f (Foo a) = Foo (f a)
```

出错：类型参数不能存在某种约束，例如这里的 **a** 明确指定为 **Eq** 的实例

```
{-
data Eq a => Bar a = Bar a
instance Functor Bar where
    fmap f (Bar a) = Bar (f a)
-}
```

Constraints on Type Definitions Are Bad

准则：不要在类型的参数上加限制，如果有需要就在函数的参数上加。

为一个类型定义加以限制永远不是一个好主意。它会强制你在所有操作那个类型的函数上加入同样的限制。

假设我们需要一个栈，我们需要查看其元素是否符合特定的 **ordering** 。

数据结构：栈

```
-- a 被限制为有序集的元素
data (Ord a) => OrdStack a = Bottom
                           | Item a (OrdStack a)
```

deriving (Show)

我们写了一个函数，此函数检查这个栈是否为递增的(上面的元素总比下面的元素大)：

```
isIncreasing :: (Ord a) => OrdStack a -> Bool
isIncreasing (Item a rest@(Item b _))
    | a < b = isIncreasing rest
    | otherwise = False
isIncreasing _ = True
```

push 函数并不需要 **a** 的类型限制，但是如果去掉这个限制就会导至编译错误

```
push :: (Ord a) => a -> OrdStack a -> OrdStack a
push a s = Item a s
```

这个例子中应保持 **isIncreasing** 函数的参数 **a** 为 **ord** 这个限制(失去它就不能用“<” 运算符)，其它限制全部去掉。

准则：不要在类型的参数上加限制，如果有需要就在函数的参数上加。

多数 **Haskell** 容器类型遵循这一准则。

Infix Use of fmap

“<$>” 操作符是 **fmap** 的别名。

将 **fmap** 作为操作符来用(为了少写括号？)，而 **map** 函数几乎从来不这么用的

```
{-
ghci> (1+) `fmap` [1,2,3] ++ [4,5,6]
[2,3,4,4,5,6]
-}

{-
ghci> fmap (1+) ([1,2,3] ++ [4,5,6])
[2,3,4,5,6,7]
-}
```

如果你真的想将 **fmap** 函数用作操作符，**Control.Applicative** 模块提供了一个“**<$>**” 操作符，“**<$>**” 操作符是 **fmap** 的别名。

Flexible Instances

你可能希望写一个 **Functor** 的实例，这个实例以 **Either Int b** 为实例参数(注意了，其类型参数只有 **b** 一个)

```
-- {-# LANGUAGE FlexibleInstances #-} --  必须加入这个编译选项
{-
instance Functor (Either Int) where
      fmap _ (Left n) = Left n
      fmap f (Right r) = Right (f r)
-}

fmap (== "cheeseburger") (Left 1 :: Either Int String)    -- >Left 1
fmap (== "cheeseburger") (Right "fries" :: Either Int String)    -- >Right False
```

注意了，因为已存在类似定义会与 **Control.Monad.Instances** 发生 **overlap** 冲突。加入**{-# LANGUAGE OverlappingInstances #-}** 编译选项可以使得当发生多匹配时使得编译器选择最特殊的那一个。

```
-- instance Functor (Either a) -- Defined in Control.Monad.Instances
```

Thinking More About Functors

准则一：将 **fmap id** 应用于一个值，就必须能够返回这个值的 **identical**

fmap id (Node (Leaf "a") (Leaf "b")) -- >Node (Leaf "a") (Leaf "b") -- 亮点，**id** 函数

准则二：**functors** 必须能够复合

(fmap even . fmap length) (Just "twelve") -- >Just True
fmap (even . length) (Just "twelve") -- >Just True

fmap odd (Just 1) -- >Just True
fmap odd Nothing -- >Nothing

Writing a Functor Instance for Parse

```
instance Functor Parse where
      fmap f parser = parser ==> \result ->
                              identity (f result)
```

```
-- <$>  是 famp
{-
ghci> parse parseByte L.empty
Left "byte offset 0: no more input"
ghci> parse (id <$> parseByte) L.empty
Left "byte offset 0: no more input"
-}

{-
ghci> let input = L8.pack "foo"
ghci> L.head input
102
ghci> parse parseByte input
Right 102
ghci> parse (id <$> parseByte) input
Right 102
-}

{-
ghci> parse ((chr . fromIntegral) <$> parseByte) input
Right 'f'
ghci> parse (chr <$> fromIntegral <$> parseByte) input
Right 'f'
-}
```

Using Functors for Parsing

functors 可以使得我们的代码更 tidy，更 expressive 。

回想一下前面的 parseByte ，现在我们想要解析 ASCII 而不只是 Word8 值。

虽然我们可以写 parseChar ，但它的结构和 parseByte 是很相似的。现在我们可以利用 functor 来避免去复制代码。

使用 chr 函数将 Int 转成 Char

```
-- chr :: Int -> Char        -- Defined in GHC.Base

--import Data.Char (chr)
w2c :: Word8 -> Char
w2c = chr . fromIntegral

-- import Control.Applicative
peekByte :: Parse (Maybe Word8)
peekByte = (fmap fst . L.uncons . string) <$> getState
```

-- peekByte 返回 Nothing 如果已是 input string 的 end，否则返回下一个字符，但是没有 consuming 它。

```
peekChar :: Parse (Maybe Char)
peekChar = fmap w2c <$> peekByte
```

-- 类似于 takeWhile 函数

```
parseWhile :: (Word8 -> Bool) -> Parse [Word8]
parseWhile p = (fmap p <$> peekByte) ==> \mp ->
                    if mp == Just True
                    then parseByte ==> \b ->
                            (b:) <$> parseWhile p
                    else identity []
```

-- 这是不使用 functor 的版本，较冗长

```
parseWhileVerbose p =
      peekByte ==> \mc ->
      case mc of
```

```
Nothing -> identity []
Just c | p c ->
                parseByte ==> \b ->
                parseWhileVerbose p ==> \bs ->
                identity (b:bs)
          | otherwise ->
                identity []
```

functor 有助于避免代码冗长

Rewriting Our PGM Parser

```
parseRawPGM =
      parseWhileWith w2c notWhite ==> \header -> skipSpaces ==>&
      assert (header == "P5") "invalid raw header" ==>&
      parseNat ==> \width -> skipSpaces ==>&
      parseNat ==> \height -> skipSpaces ==>&
      parseNat ==> \maxGrey ->
      parseByte ==>&
      parseBytes (width * height) ==> \bitmap ->
      identity (Greymap width height maxGrey bitmap)
  where notWhite = (`notElem` " \r\n\t")


parseWhileWith :: (Word8 -> a) -> (a -> Bool) -> Parse [a]
parseWhileWith f p = fmap f <$> parseWhile (p . f)


parseNat :: Parse Int
parseNat = parseWhileWith w2c isDigit ==> \digits ->
                if null digits
                then bail "no more input"
                else let n = read digits
                         in if n < 0
                             then bail "integer overflow"
                             else identity n

(==>&) :: Parse a -> Parse b -> Parse b
p ==>& f = p ==> \_ -> f

skipSpaces :: Parse ()
skipSpaces = parseWhileWith w2c isSpace ==>& identity ()

assert :: Bool -> String -> Parse ()
assert True _ = identity ()
assert False err = bail err

parseBytes :: Int -> Parse L.ByteString
parseBytes n =
      getState ==> \st ->
      let n' = fromIntegral n
          (h, t) = L.splitAt n' (string st)
```

```
        st' = st { offset = offset st + L.length h, string = t }
    in putState st' ==>&
        assert (L.length h == n') "end of input" ==>&
        identity h
```

Future Directions

在第 **14** 章，我们会看到使用 **monads** 可以大大简化本章的代码。