Learn Real World Haskell

Jacob Bishop

2023-08-20

1 Getting started

略

2 Types and functions

略

3 Defining types, streamlining functions

略

4 Functional programming

略

5 Writing a library: working with JSON data

在 Haskell 中表示 JSON

首先是在 Haskell 中定义 JSON 这个数据，这里使用代数数据类型来表达 JSON 类型的范围。

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

对于每种 JSON 类型，我们都提供了独立的值构造函数。测试：

```
ghci> :l SimpleJSON
[1 of 1] Compiling Main             ( SimpleJSON.hs, interpreted )
Ok, one module loaded.
ghci> JString "foo"
JString "foo"
ghci> JNumber 2.7
JNumber 2.7
ghci> :type JBool True
JBool True :: JValue
```

构造一个从 `JValue` 获取字符串的函数：

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

测试：

```
ghci> :reload
[1 of 1] Compiling Main             ( SimpleJSON.hs, interpreted )
Ok, one module loaded.
ghci> getString (JString "hello")
Just "hello"
ghci> getString (JNumber 3)
Nothing
```

接下来是其它类型的访问函数：

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

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

```
6
7  getBool (JBool n) = Just n
8  getBool _ = Nothing
9
10 getObject (JObject o) = Just o
11 getObject _ = Nothing
12
13 getArray (JArray a) = Just a
14 getArray _ = Nothing
15
16 isNull v = v == JNull
```

`truncate` 函数可以让浮点类型或者有理数去掉小数点后变为整数：

```
1 ghci> truncate 5.8
2 5
3 ghci> :module +Data.Ratio
4 ghci> truncate (22 % 7)
5 3
```

Haskell 模块详解

一个 Haskell 源文件包含了单个模块的定义。模块允许我们在它其内部进行定义，并由其它模块访问：

```
1  module SimpleJSON
2    ( JValue (..),
3      getString,
4      getInt,
5      getDouble,
6      getBool,
7      getObject,
8      getArray,
9      isNull,
10   )
11 where
```

如果省略了导出（即圆括号以及其所包含的名称），那么该模块中的所有名称都会被导出。

编译 Haskell 源

编译一个源文件：

```
1 ghc -c SimpleJSON.hs
```

`-c` 选项告诉 **ghc** 仅生成对象代码。如果省略了该选项，那么编译器则会尝试生成一整个可执行文件。这会导致失败，因为我们并没有一个 `main` 函数，即 GHC 所认为的一个独立程序的执行入口。

编译后会得到两个新文件：`SimpleJSON.hi` 与 `SimpleJSON.o` 。前者是一个接口 *interface* 文件，即 **ghc** 以机器码的形式存储模块导出的名称信息；后者是一个对象 *object* 文件，其包含了生产的机器码。

生成一个 Haskell 程序，导入模块

添加一个 `Main.hs` 文件，其内容如下：

```
module Main where

import SimpleJSON

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

与原文 `module Main () where` 的不同之处在于，现在的 `Main` 后不再需要一个 `()` 。

接下来是编译 `main` 函数：

```
ghc -o simple Main.hs
```

与原文 `ghc -o simple Main.hs SimpleJSON.o` 不同，现在没了 `SimpleJSON.o` 这个文件，加上后会报错重复 symbol 的编译错误（因为在 `Main.hs` 中已经做了 `import SimpleJSON` 导入了）。

这次省略掉了 `-c` 选项，因此编译器尝试生成一个可执行。生成可执行的过程被称为链接 *linking*（与 C++ 一样），即在一次编译中链接源文件与可执行文件。

这里给了 **ghc** 一个新选项 `-o` ，其接受一个参数：可执行文件的名称，这里是 `simple` ，执行它：

```
./simple
JObject [("foo",JNumber 1.0),("bar",JBool False)]
```

打印 JSON 数据

现在我们希望将 Haskell 的值渲染成 JSON 数据，创建一个 `PutJSON.hs` 文件：

```
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)
```

好的 Haskell 风格需要分隔纯代码与 I/O 代码。我们的 `renderJValue` 函数不会与外界交互，但是仍然需要一个打印的函数：

```
putJValue :: JValue -> IO ()
putJValue = putStrLn . renderJValue
```

类型推导是把双刃剑

假设我们编写了一个自认为返回 `String` 的函数，但是并不为其写类型签名：

```
upcaseFirst (c:cs) = toUpper c -- forgot ":cs" here
```

这里希望单词首字母大写，但是忘记了将剩余的字符放进结果中。我们认为函数的类型是 `String -> String` ，但是编译器则会将其视为 `String -> Char` 。假设我们尝试在其他地方调用该函数：

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

那么当我们尝试编译该代码或者是加载进 **ghci**，我们并不会得到明显的错误信息：

```
ghci> :load Trouble
[1 of 1] Compiling Main             ( Trouble.hs, interpreted )

Trouble.hs:9:27:
    Couldn't match expected type `[Char]' against inferred type `Char'
      Expected type: [Char] -> [Char]
      Inferred type: [Char] -> Char
    In the first argument of `map', namely `upcaseFirst'
    In the first argument of `concat', namely
        `(map upcaseFirst (words xs))'
Failed, modules loaded: none.
```

注意这里的报错是在 `upcaseFirst` 函数处，那么假设我们认为 `upcaseFirst` 的定义与类型是正确的，那么查找到真正的错误可能会花掉一些时间。

更加泛用的渲染

我们将更为泛用的打印模块称为 `Prettify` ，那么其源文件即 `Prettify.hs` 。

为了使 `Prettify` 满足实际需求，我们还要一个新的 JSON 渲染器来使用 `Prettify` 的 API。在 `Prettify` 模块中将使用一个抽象类型 `Doc` 。基于建立在抽象类型的泛用渲染库，我们可以选择灵活高效的实现。

`PrettyJSON.hs` 示例：

```
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
```

这里的 `text` ， `double` 以及 `string` 都会在 `Prettify` 模块中提供。

开发 Haskell 代码而不发疯

一个用于快速开发程序框架的技巧就是编写占位符，或者类型与函数的根 *stub* 版本。例如上述 `string` ， `text` 以及 `double` 函数将被 `Prettify` 模块提供。如果我们没有提供这些函数或者 `Doc` 类型，那么“早点编译，经常编译”这个尝试就会失败。为了避免这个问题现在让我们编写一个不做任何事情的根程序。

```
import SimpleJSON

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` 有一个 `a` 类型，无论在哪里使用它，总是会有类型检查。如果尝试计算，则会使程序崩溃：

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

尽管还不能运行根代码，但是编译器的类型检查器可以确保我们的程序类型正确。

漂亮的打印字符串

当需要打印一个漂亮的字符串时，我们必须遵循 JSON 的转义规则。字符串就是一系列被包裹在引号中的字符们。`PrettyJSON.hs` ：

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

以及 `enclose` 函数将一个 `Doc` 值简单的包裹在一个开始与结束字符之间：

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

这里提供的 `<>` 函数定义在 `Prettify` 库中，它需要两个 `Doc` 值，即 `Doc` 版本的 `(++)` 。还是先在根文件中定义：

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

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

我们的库还需要提供 `hcat` ，将若干 `Doc` 值合成成为一个（类似于列表的 `concat` ）：

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

我们的 `string` 函数应用 `oneChar` 函数到字符串中的每个字符，将它们连接，然后用引号包装。而 `oneChar` 函数则是用来转义或包装一个独立的字符。在 `PrettyJSON.hs` 中：

```
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'

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

这里的 `simpleEscapes` 是一个列表的二元组，我们称其为关联 *association* 列表，或者简称 `alist` 。每个 `alist` 的元素都将字符关联了它的转义表达，测试：

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

我们的 `case` 表达式尝试查看字符是否匹配关联列表。如果匹配则返回匹配值，如若不然则需要以更复杂的方式来转义该字符。只有当两种转义都不需要时，才会返回普通字符。保守起见，我们输出的唯一未转义字符是可打印的 ASCII 字符。

更复杂的转义包含了将一个字符转为字符串 `"\u"` 并跟随四个十六进制的字符用于表达 Unicode 字符的数值。仍然是 `PrettyJSON.hs` ：

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

这里的 `showHex` 函数需要从 `Numeric` 库中加载，其用于返回一个值的十六进制：

```
ghci> showHex 114111 ""
"1bdbf"
```

`replicate` 函数则是由 Prelude 提供：

```
ghci> replicate 5 "foo"
["foo","foo","foo","foo","foo"]
```

`smallHex` 提供的四位编码只能表示最大 `0xffff` 的 Unicode 字符，而有效的 Unicode 字符的范围可以达到 `0x10ffff` 。为了正确的表达一个超出了 `0xffff` 的 JSON 字符串，我们遵循一些复杂的规则将其分为两部分。这使我们有机会对 Haskell 的数执行一些位级操作。还是 `PrettyJSON.hs` ：

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

这里 `shiftR` 函数和 `(.&.)` 函数都源自 `Data.Bits` 模块，前者将一个数移动右一位，后者则是执行一个字节层面的两值 *and* 操作。

```
ghci> 0x10000 `shiftR` 4   :: Int
4096
ghci> 7 .&. 2   :: Int
2
```

现在有了 `smallHex` 与 `astral` ，我们可以提供 `hexEscape` 的定义了：

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

```
    where
      d = ord c
```

其中 `ord` 由 `Data.Char` 模块提供。

数组与对象，以及模块头

相比于字符串的漂亮打印，数组和对象就是小菜一碟了。我们已经知道了它们两者其实很相似：都是由起始字符开始，接着是一系列由逗号分隔的值，最后接上结束字符。让我们编写一个函数捕获数组与对象的共同结构：

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

让我们首先从函数类型开始。它接受起始与结束字符，一个打印某些未知类型 `a` 值的函数，以及一个类型为 `a` 的列表，返回一个类型为 `Doc` 的值。

注意尽管我们的类型签名提及了四个参数，在函数定义中仅列出了三个。这遵循了简化定义的规则，如 `myLength xs = length xs` 等同于 `myLength = length` 。

我们已经有了之前编写过的 `enclose` ，即包装一个 `Doc` 值进起始与结束字符之间。那么 `fsep` 函数则位于 `Prettify` 模块中，其结合一个 `Doc` 值列表成为一个 `Doc` ，在输出不适合单行的情况下还需要换行。

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

那么现在，遵循上述提供的例子，你应该能够定义你自己的 `Prettify.hs` 的根文件了。这里不再显式的定义更多的根了。

`puctuate` 函数同样位于 `Prettify` 模块中：

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

通过 `series` 的定义，漂亮打印一个数组就非常直接了：

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

对于对象而言，还需要额外的一些工作：每个元素同时需要处理名称与值：

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

编写一个模块头

现在已经有了 `PrettyJSON.hs` 文件，我们需要回到其顶部添加模块声明：

```
module PrettyJSON (renderJValue) where
```

这里导出了一个名称：`renderJValue` ，也就是我们的 JSON 渲染函数。模块中其它的定义都是用于支持 `renderJValue` 的，因此没有必要对其它模块可见。

充实我们的漂亮打印库

在 `Prettify` 模块中，提供了 `Doc` 类型作为一个代数数据类型：

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

观察可知 `Doc` 类型实际上是一颗树。`Concat` 与 `Union` 构造函数根据其它两个 `Doc` 值创建一个内部节点，而 `Empty` 以及其它简单的构造函数用于构建叶子。

在模块的头部，我们导出该类型的名称，而不是它们的构造函数：这样可以防止使用 Doc 的构造函数被用于创建与模式匹配 `Doc` 值。

相反的，要创建一个 `Doc` ，用户需要调用我们在 `Prettify` 模块中所提供的函数：

```
empty :: Doc
empty = Empty

char :: Char -> Doc
char = Char

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

double :: Double -> Doc
double = text . show
```

`Line` 构造函数代表一个换行，其创建的是一个 *hard* 换行，即总是会在漂亮的打印中出现。有时我们想要一个 *soft* 换行，即只会在窗口或者页面上过长显示时才会换行。稍后将会介绍 `softline` 函数。

```
line :: Doc
line = Line
```

另外就是用于连接两个 `Doc` 值的 `(<>)` 函数（这里使用 `(<+>)`，因为 `(<>)` 在 Prelude 中已经有定义了）：

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

测试：

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

接下来是用于连接 `Doc` 列表的 `hcat` 函数：

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

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

以及 `fsep` 函数，它还依赖若干其它函数：

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

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

softline :: Doc
softline = group line
```

这里需要解释一下，`softline` 函数应该在当前行特别宽的时候进行换行，否则插入空格。如果我们的 `Doc` 类型不包含任何关于渲染的信息，那么该如何呢？答案就是每次遇到一个软换行，通过 `Union` 构造函数来维护两个可选项：

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

`flatten` 函数将一个 `Line` 替换为空格，将两行转换为一个更长的行。

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

注意总是调用 `flatten` 在 `Union` 的左元素上：每个 `Union` 的左侧总是大于等于右侧宽度（字符距离）。

紧密渲染

我们需要频繁的使用包含尽可能少字符的数据。例如通过网络连接发送 JSON 数据就没有必要美观：软件并不在乎数据的美观与否，添加很多空格只会带来性能下降。

因此我们提供了一个紧密渲染的函数：

```
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)
```

`compact` 函数将其参数包裹成一个列表，然后将帮助函数 `transform` 应用至该列表。`transform` 函数视其参数为堆叠的项用于处理，列表的第一个元素即堆的顶部。

`transform` 函数的 `(d:ds)` 模式将堆顶部的元素取出 `d` 并留下 `ds` 。在 `case` 表达式中，前面几个分支都在 `ds` 上递归，每次递归消费堆顶部的元素；最后两个分支则是在 `ds` 之前添加项：`Concat` 添加两个元素至堆，而 `Union` 分支则忽略它左侧元素，调用的是 `flatten` ，再将其右侧元素至堆。

测试 `compact` ：

```
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
}
```

为了更好的理解代码是如何运作的，让我们用一个更简单的例子来展示细节：

```
ghci> char 'f' <> text "oo"
Concat (Char 'f') (Text "oo")
ghci> compact (char 'f' <> text "oo")
"foo"
```

当我们应用 `compact` 时，它会将它的参数转为一个列表后应用函数 `transform` 。

- 接下来 `transform` 函数接受了一个单例列表，然后进行模式匹配 `(d:ds)` ，这里的 `d` 是 `Concat (Char 'f') (Text "oo")` ，而 `ds` 则是一个空列表。

  由于 `d` 的构造函数是 `Concat` ，其模式匹配就在 `case` 表达式中。那么在右侧，添加 `Char 'f'` 与 `Text "oo"` 值堆，然后递归的应用 `transform` 。

- – `transform` 函数接受了一个包含两项的列表，继续匹配 `d:ds` 模式。此时变量 `d` 绑定到了 `Char 'f'` ，而 `ds` 则是 `[Text "oo"]` 。

  `case` 表达式匹配到了 `Char` 分支。那么在右侧，使用 `(:)` 来构建一个列表，其头部为 `'f'` ，其余部分则是递归应用 `transform` 后的结果。

  – * 递归的调用接受到一个单例列表，其变量 `d` 绑定至 `Text "oo"` ， `ds` 绑定至 `[]` 。

    `case` 表达式匹配 `Text` 分支。那么在右侧，使用 `(++)` 来连接 `"oo"` 与递归调用 `transform` 后的结果。

    * · 最后的调用，`transform` 得到一个空列表，返回一个空字符串。

    * 结果是 `"oo" ++ ""`

  – 结果是 `'f' : "oo" ++ ""`

真实的漂亮打印

我们的 `compact` 函数对于机器之间的交流是有帮助的，但是它的结果对人类而言并不易读：每行的信息很少。为了生成一个更可读的输出，我们将要编写另一个函数 `pretty` 。相比于 `compact` ， `pretty` 接受一个额外的参数：一行的最大宽度。

```
pretty :: Int -> Doc -> String
```

确切来说，`Int` 参数控制了 `pretty` 在遇到一个 `softline` 时的行为。在一个 `softline` 时， `pretty` 会选择继续留在当前行还是另起一行。其余情况下，必须严格遵守漂亮打印函数所设定的指令。

以下是实现的代码：

```
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
```

`best` 帮助函数接受两个参数：当前行使用了的列数，以及剩余的仍需处理的 `Doc` 列表。

在简单的情况下，随着消费输入 `best` 直接更新了 `col` 变量。其中 `Concat` 情况也很明显：将两个连接过的部分推至堆叠，且不触碰 `col` 。

有趣的情况在 `Union` 构造函数。回想一下之前将 `flatten` 应用至左侧元素，且不对右侧做任何操作。还有就是 `flatten` 将新行替换成空格。因此我们则需要检查这两种布局，`flatten` 后的还是原始的，更适合我们的宽度限制。

为此需要编写一个小的帮助函数来决定 `Doc` 值的一行是否适合给定的长度：

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

遵循漂亮打印

为了理解代码如何工作的，首先考虑一个简单的 `Doc` 值：

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

我们将应用 `pretty 2` 在该值。当我们首先应用 `best` ， `col` 值为零。它匹配 `Concat` 模式，接着将 `Union (Char ' ') Line` 与 `Char 'a'` 推至堆，接着是递归应用自身，它匹配了 `Union (Char ' ') Line` 。

现在忽略 Haskell 通常的计算顺序，两个子表达式，`best 0 [Char ' ', char 'a']` 与 `best 0 [Line, Char 'a']` ，前者计算得到 `" a"` ，而后者得到 `"na "` 。接着将它们替换到外层的表达式中，得到 `nicest 0 " a" "\na"` 。

为了明白 `nicest` 的结果，我们做一个小小的替换。`width` 以及 `col` 分别是 0 与 2，那么 `least` 就是 0，`width - least` 就是 2。这里用 **ghci** 来计算一下 `2 `fits` " a"` ：

```
ghci> 2 `fits` " a"
True
```

计算得到 `True` ，那么这里的 `nicest` 结果就是 `" a"` 。

如果将 `pretty` 函数应用到之前同样的 JSON 数据上，可以看到根据提供的最大长度，它会得到不同的结果：

```
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 }
```

创建一个库

Haskell 社区构建了一个标准工具库，名为 Cabal，其用于构建，安装，以及分发软件。Cabal 以 `包package` 的方式管理软件，一个包包含了一个库，以及若干可执行程序。

编写一个包的描述

要使用包，Cabal 需要一些描述。这些描述保存在一个文本文件中，以 `.cabal` 后缀命名。该文件应位于项目的根目录。

包描述由一系列的全局属性开始，其应用于包中所有的库以及可执行。

```
name:               pretty-json
version:            0.1.0.0
```

包名称必须是唯一的。如果你创建并安装了一个同名包在你的系统上，GHC 会感到迷惑。

```
synopsis:           My pretty printing library, with JSON support
description:
  A simple pretty printing library that illustrates how to
  develop a Haskell library.
author:             jacob xie
maintainer:         jacobbishopxy@gmail.com
```

这里还要有 license 信息，大多数 Haskell 包都使用 BSD license，Cabal 称为 BSD3。

另外就是 Cabal 的版本：

```
cabal-version:      2.4
```

在一个包中要描述一个独立的库，需要 *library* 这个部分。注意缩进在这里很重要。

```
library
  default-language: Haskell2010
  build-depends:    base
  exposed-modules:
    Prettify
    PrettyJSON
    SimpleJSON
```

`exposed-modules` 字段包含了一个模块列表，用于暴露给使用该包的用户导入。另一个可选字段 `other-modules` 包含了一个 内部 *internal* 模块的列表，它们提供给 `exposed-modules` 内的模块使用，而对用户不可见。

`build-depends` 字段包含了一个逗号分隔的包列表，它们是我们库所需的依赖。`base` 包中包含了 Haskell 很多核心模块，例如 Prelude，所以它总是必须的。

GHC 的包管理器

GHC 包含了一个简单的包管理器用于追踪安装了哪些包，以及这些包的版本号。一个名为 **ghc-pkg** 的命令行工具提供了包数据库的管理。

这里说*数据库 database* 是因为 GHC 区分了*系统级别*的包，即对所有用户可用；以及用户可见的包，即仅对当前用户可用。后者可以避免管理员权限来安装包。

ghc-pkg 提供了不同的子命令，多数时候我们仅需两个命令：`ghc-pkg list` 用于查看已安装的包；当需要删除包时则使用 `ghc-pkg unregister` 。

设置，构建与安装

除了一个 `.cabal` 文件，一个包还必须包含一个 *setup* 文件。在包需要的情况下，它允许 Cabal 的构建过程中包含大量的定制。

`Setup.hs` 示例：

```
#!/usr/bin/env runhaskell

import Distribution.Simple

main = defaultMain
```

一旦有了 `.cable` 与 `Setup.hs` 文件，那么就剩下三步了。

指导 Cabal 如何构建以及在哪里安装包，只需一个简单命令：

```
runghc Setup configure
```

这可以确保我们所需要的包都是可用的，并且保存设定为了之后的 Cabal 命令。

如果没有为 `configure` 提供任何参数，Cabal 则会将包安装在系统层的包数据库。要在 home 路径安装则需要提供一些额外的信息：

```
runghc Setup configure --prefix=$HOME --user
```

接下来就是包的构建：

```
runghc Setup build
```

如果成功了，我们就可以安装这个包了。我们无需指定其安装的位置：Cabal 会使用我们在 `configure` 步骤中所提供的配置。即安装在我们自己的路径，并更新 GHC 的用于层包数据库。

```
runghc Setup install
```

6 Using typeclasses

Typeclasses 是 Haskell 中最强大的特性。它们允许我们定义通用性的接口，为各种类型提供公共特性集。Typeclasses 是一些语言特性的核心，例如相等性测试和数字运算符。

对 Typeclasses 的需求

假设在没有相等性测试 `==` 的情况下需要构建一个简单的 `color` 类型，那么相等测试就应该如下：

```
data Color = Red | Green | Blue

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

现在假设我们想为 `StringS` 增加一个相等性测试。由于 Haskell 的 `String` 是字符列表，我们可以编写一个简单的函数用于测试。这里为了简化我们使用 `==` 操作符用于说明。

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

现在已经发现问题了：我们必须为每个不同的类型使用不同名称的比较函数，这是很低效且令人讨厌的。因此需要一个通用的函数可用于比较任何东西。此外，当新的数据类型之后被创建时，已经存在的代码不能被改变。

Haskell 的 typeclasses 就是设计用来解决上述问题的。

什么是 typeclasses

Typeclasses 定义了一系列的函数，它们可以根据给定的数据类型有不同的实现。

首先我们必须定义 typeclass 本身。我们希望一个函数接受同样类型的两个参数，返回一个 `Bool` 来表示它们是否相等。我们无需在意类型是什么，但需要两个参数的类型相同。下面是 typeclass 的第一个定义：

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

通过 **ghci** 的类型检查 `:type` 可以得知 `isEqual` 的类型：

```
ghci> :type isEqual
isEqual :: (BasicEq a) => a -> a -> Bool
```

现在可以为特定类型定义 `isEqual` ：

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

测试：

```
ghci> isEqual False False
True
ghci> isEqual False True
False
ghci> isEqual "Hi" "Hi"

<interactive>:1:0:
    No instance for (BasicEq [Char])
      arising from a use of `isEqual' at <interactive>:1:0-16
    Possible fix: add an instance declaration for (BasicEq [Char])
    In the expression: isEqual "Hi" "Hi"
    In the definition of `it': it = isEqual "Hi" "Hi"
```

注意在尝试比较两个字符串时，**ghci** 发现我们并未给 `String` 提供 `BasicEq` 的实例。因此 `ghci` 并不知道该如何对 `String` 进行比较，同时提议我们可以通过为 `[Char]` 定义 `BasicEq` 实例来解决这个问题。

下面是定义一个包含了两个函数的 typeclass：

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

虽然 `BasicEq2` 的定义没有问题，但是它让我们做了额外的事情。就逻辑而言，如果我们知道了 `isEqual` 或 `isNotEqual` 中的一个，我们便知道了另一个。那么与其让用户定义 typeclass 中两个函数，我们可以提供一个默认的实践。

```
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)
```

声明 typeclass 实例

现在知道了如何定义 typeclasses，接下来就是直到如何定义 typeclasses 的实例。回忆一下，类型是由一个个特定 typeclass 所构成的实例所赋予了意义，这些 typeclasses 又实现了必要的函数。

之前为我们的 `Color` 类型创建了相等性测试，现在让我们试试让其成为 `BasicEq3` 的实例：

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

注意这里提供了与之前定义的一样的函数，实际上实现是完全相同的。不过在这种情况下，我们可以使用 `isEqual3` 在任何定义了 `BasicEq3` 实例的类型上。

另外注意 `BasicEq3` 定义了 `isEqual3` 与 `isNotEqual3` ，而我们仅实现了它们其中一个。这是因为 `BasicEq3` 包含了默认实现，因此没有显式定义 `isNotEqual3` 时，编译器自动的使用了 `BasicEq3` 中的默认实现。

重要的内建 Typeclasses

Show

`Show` typeclass 用于将值转换为 `String` 。可能最常用于将数字转换为字符串，很多类型都有它的实例，因此还可以用于转换更多的类型。如果自定义类型实现了 `Show` 的实例，那么就可以在 **ghci** 上展示或者在程序中打印出来。

`Show` 中最重要的函数就是 `show` ，它接受一个参数：用于转换的数据，返回一个 `String` 来表示该数据。

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

一些其它的例子：

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

ghci 展示的结果与输入到 Haskell 程序中的结果相同。表达式 `show 1` 返回的是单个字符的字符串，其包含数字 `1` ，也就是说引号并不是字符串本身。通过 `putStrLn` 可以更清楚的看到：

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

我们也可以使用 `show` 在字符串上：

```
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\""
```

现在为我们自己的类型实现 `Show` 的实例:

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

Read

`Read` typeclass 基本就是相反的 `Show` : 它定义了函数接受一个 `String` , 分析它, 并返回属于 `Read` 成员的任何类型的数据。

```
ghci> :type read
read :: (Read a) => String -> a
```

以下是一个 `read` 与 `show` 的例子:

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

`read` 的类型: `(Read a) => String -> a` 。这里的 `a` 是每个 `Read` 实例的类型。也就是说特定的解析函数是根据预期的 `read` 返回类型所决定的。

```
ghci> (read "5.0")::Double
5.0
ghci> (read "5.0")::Integer
*** Exception: Prelude.read: no parse
```

在尝试解析 `5.0` 为 `Integer` 时异常。当预期返回值的类型是 `Integer` 时, `Integer` 的解析函数并不接受小数, 因此异常被抛出。

`Read` 提供了一些相当复杂的解析器。你可以通过实现 `readsPrec` 函数定义一个简单的解析。该实现在解析成功时, 返回只包含一个元组的列表, 如果解析失败则返回空列表。下面是一个实现 `Color` 的 `Read` 实例的例子:

```haskell
instance Read Color where
-- readsPrec is the main function for parsing input
readsPrec _ value =
  -- We pass tryParse a list of pairs. Each pair has a string
  -- and the desired return value. tryParse will try to match
  -- the input to one of these strings
  tryParse [("Red", Red), ("Green", Green), ("Blue", Blue)]
  where
    -- If there is nothing left to try, fail
    tryParse [] = []
    tryParse ((attempt, result) : xs) =
      -- Compare the start of the string to be parsed to the
      -- text we are looking for.
      if take (length attempt) value == attempt
        then -- If we have a match, return the result and the remaining input
          [(result, drop (length attempt) value)]
        else -- If we don't have a match, try the next pair in the list of attempts.
          tryParse xs
```

测试：

```
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
```

注意最后一个例子的异常。这是因为我们的解析器并没有聪明到能处理空格。

Tip

Read 并没有大范围的被使用

虽然可以使用 `Read` typeclass 来构建复杂的解析器，但许多人发现使用 Parsec 会更容易，且仅依赖 `Read` 来处理简单的任务。在第 16 章会详细介绍 Parsec。

通过 Read 和 Show 进行序列化

我们经常需要存储一个内存中的数据结构至硬盘供未来使用或者通过网络发送出去，那么将内存中数据转换为一个平坦的字节序列用于存储的这个过程就被称为*序列化 serialization*。

Tip

解析大型字符串

在 Haskell 中字符串的处理通常都是惰性的，因此 `read` 与 `show` 可以被用于处理很大的数据结构而不发生异常。Haskell 内建的 `read` 与 `show` 实例是高效的，并且都是纯 Haskell。如何处理解析异常的更多细节将会在第 19 章中详细介绍。

测试：

```
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)
```

再是反序列：

```
ghci> input <- readFile "test"
"[Just 5,Nothing,Nothing,Just 8,Just 9]"
ghci> let d2 = read input

<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
```

这里解释器并不知道 `d2` 的类型，因此抛出了异常。

以下是一些稍微复杂点的数据结构：

```
ghci> putStrLn $ show [("hi", 1), ("there", 3)]
[("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 []]
```

数值类型

Haskell 拥有强大的数值类型。

选定的数值类型	
类型	描述
Double	双精度浮点数。浮点数的通常选择。
Float	单精度浮点数。通常用于与 C 交互。
Int	带方向的确定精度整数；最小范围 $[-2^29..2^29-1]$。
Int8	8-bit 带方向的整数
Int16	16-bit 带方向的整数
Int32	32-bit 带方向的整数
Int64	64-bit 带方向的整数
Integer	带方向的任意精度整数；范围仅受机器限制。很常用。
Rational	带方向的任意精度的有理数。以两个 Integers 进行存储。
Word	无方向的确定精度整数；存储大小与 Int 一致
Word8	8-bit 无方向的整数
Word16	16-bit 无方向的整数
Word32	32-bit 无方向的整数
Word64	64-bit 无方向的整数

有很多不同的数值类型。有些运算，比如加法对所有类型都适用；还有其它类型的计算例如 `asin` ，仅适用于浮点类型。

选定的数值函数与常量			
项	类型	模块	描述
(*)	Num a => a -> a -> a	Prelude	加法
(-)	Num a => a -> a -> a	Prelude	减法
(*)	Num a => a -> a -> a	Prelude	乘法
(/)	Fractional a => a -> a -> a	Prelude	除法
(**)	Floating a => a -> a -> a	Prelude	幂
(^)	(Num a, Integral b) => a -> b -> a	Prelude	非负数的幂
(i	(Fractional a, Integral b) => a -> b -> a	Prelude	分数的幂
(%)	Integral a => a -> a -> Ratio a	Data.Ratio	比例
(.&.)	Bits a => a -> a -> a	Data.Bits	Bitwise 和
(.|.)	Bits a => a -> a -> a	Data.Bits	Bitwise 或
abs	Num a => a -> a	Prelude	绝对值
approxRational	RealFrac a => a -> a -> Rational	Data.Ratio	基于分数分子与分母的近似有理组合
cos	Floating a => a -> a	Prelude	Cosine，还有 acos，cosh，与 acosh
div	Integral a => a -> a -> a	Prelude	整数除法，总是向下取整，quot 同理
fromInteger	Num a => Integer -> a	Prelude	从 Integer 类型转换为任意数值类型
fromIntegral	(Integral a, Num b) => a -> b	Prelude	比上述更泛化的转换
fromRational	Fractional a => Rational -> a	Prelude	从 Rational 转换，可能有损
log	Floating a => a -> a	Prelude	自然 log
logBase	Floating a => a -> a -> a	Prelude	显式底的 log
maxBound	Bounded a => a	Prelude	bound 类型的最大值
minBound	Bounded a => a	Prelude	bound 类型的最小值
mod	Integral a => a -> a -> a	Prelude	整数模
pi	Floating a => a	Prelude	数学常数的派
quot	Integral a => a -> a -> a	Prelude	整数除法；商的小数部分向零截断

接上表

项	类型	模块	描述
recip	Fractional a => a -> a	Prelude	倒数
rem	Integral a => a -> a -> a	Prelude	整数除法的余数
round	(RealFrac a, Integral b) => a -> b	Prelude	向最接近的整数方向取整
shift	Bits a => a -> Int -> a	Bits	左移指定 bits，右移可能为负
sin	Floating a => a -> a	Prelude	Sine，另 asin，sinh，与 asinh
sqrt	Floating a => a -> a	Prelude	平方根
tan	Floating a => a -> a	Prelude	Tangent，令 atan，tanh，与 atanh
toInteger	Integral a => a -> Integer	Prelude	转换任意 Integral 至一个 Integer
toRational	Real a => a -> Rational	Prelude	转换 Real 至 Rational
truncate	(RealFrac a, Integral b) => a -> b	Prelude	向零截断数值
xor	Bits a => a -> a -> a	Data.Bits	Bitwise 的异或

数值类型的 Typeclass 实例								
项	Bits	Bounded	Floating	Fractional	Integral	Num	Rea	RealFrac
Double			X	X		X	X	X
Float	X	X			X	X	X	
Int	X	X			X	X	X	
Int16	X	X			X	X	X	
Int32	X	X			X	X	X	
Int64	X	X			X	X	X	
Integer	X				X	X	X	
Rational or any Ratio				X		X	X	X
Word	X	X			X	X	X	
Word16	X	X			X	X	X	
Word32	X	X			X	X	X	
Word64	X	X			X	X	X	

在数值类型间转换是另一个常用的需求。

数值类型之间的转换				
原始类型	目标类型			
	Double, Float	Int, Word	Integer	Rational
Double, Float	fromRational . toRational	truncate *	truncate *	toRational
Int, Word	fromIntegral	fromIntegral	fromIntegral	fromIntegral
Integer	fromIntegral	fromIntegral	N/A	fromIntegral
Rational	fromRational	truncate *	truncate *	N/A

自动化派生

对于很多简单的数据类型，Haskell 编译器可以自动的为我们派生出 `Read` ，`Show` ，`Bounded` ， `Enum` ， `Eq` 以及 `Ord` 的实例。这将大大的节省用户编写代码的时间。

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

测试：

```
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]
[Red,Green,Blue,Blue]
ghci> Red < Blue
True
```

自动派生也不总是能成功。例如如果定义了一个类型 `data MyType = MyType (Int -> Bool)` ，编译器将不能为其派生 `show` 的实例，因为它不知道如何渲染一个函数。这种情况下我们会得到一个编译错误。

当我们自动派生某些 typeclass 的实例时，在数据声明中引用的类型也必须是该 typeclass 的实例（手动或自动的）。

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

```haskell

-- will not compile, since CannotShow is not an instance of Show
data CannotDeriveShow = CannotDeriveShow CannotShow
          deriving (Show)

data OK = OK

instance Show OK where
show _ = "OK"

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

注意 `CannotShow` 的 `deriving (Show)` 是被注释掉的，因此 `CannotDeriveShow` 才无法派生 `Show` 。

Typeclasses 实战：令 JSON 使用起来更方便

上一章讲述的 `JValue` 处理 JSON 并不容易。例如，下面是一个实际 JSON 数据的删减整理的片段：

```
{
  "query": "awkward squad haskell",
  "estimatedCount": 3920,
  "moreResults": true,
  "results":
  [{
    "title": "Simon Peyton Jones: papers",
    "snippet": "Tackling the awkward squad: monadic input/output ...",
    "url": "http://research.microsoft.com/~simonpj/papers/marktoberdorf/",
   },
   {
    "title": "Haskell for C Programmers | Lambda the Ultimate",
    "snippet": "... the best job of all the tutorials I've read ...",
    "url": "http://lambda-the-ultimate.org/node/724",
   }]
}
```

以及在 Haskell 中的表达：

```haskell
import SimpleJSON

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/")
       ]])
    ]
```

因为 Haskell 并不支持列表中包含不同类型的值，我们不能直接表示一个包含了多个类型的 JSON 对象。我们必须将每个值通过 `JValue` 构造函数来进行包装。这限制了我们的灵活性：如果想要更换数值 `3920` 成为一个字符串 `"3,920"` ，我们必须更换构造函数，即 `JNumber` 变为 `JString` 。

Haskell 的 typeclasses 为此类问题提供了一个诱人的解决方案：

```
type JSONError = String

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

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

现在无需再将类似 `JNumber` 这样的构造函数应用在值上将值包裹，直接应用 `toJValue` 函数在该值上即可。

我们同样提供了一个 `fromJValue` 函数，用于将一个 `JValue` 转换成一个我们所期望的类型。

更多有帮助的错误

让我们构造一下自己的 `Maybe` 与 `Either` ：

```
data Maybe a
  = Nothing
  | Just a
  deriving (Eq, Ord, Read, Show)

data Either a b
  = Left a
  | Right b
  deriving (Eq, Ord, Read, Show)
```

在 `Bool` 值实例中尝试一下：

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

通过类型同义词来创建一个实例

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

活在开放世界

Haskell 的 typeclasses 允许我们在任何合适的时候创建新的 typeclass 实例。

```
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
```

我们还希望转换一个列表成为 JSON，暂时用 `undefined` 作为实例的方法。

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

对象亦是如此：

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

合适重叠实例会导致问题

如果我们将这些定义放入一个原文件中，并加载至 `ghci` ，每个初始化看起来都没问题：

```
ghci> :load BrokenClass
[1 of 2] Compiling SimpleJSON       ( ../ch05/SimpleJSON.hs, interpreted )
[2 of 2] Compiling BrokenClass      ( BrokenClass.hs, interpreted )
Ok, modules loaded: SimpleJSON, BrokenClass.
```

然而当我们尝试使用元组的列表时，错误便发生了。

```
ghci> toJValue [("foo","bar")]

<interactive>:1:0:
    Overlapping instances for JSON [([Char], [Char])]
      arising from a use of `toJValue' at <interactive>:1:0-23
    Matching instances:
      instance (JSON a) => JSON [a]
        -- Defined at BrokenClass.hs:(44,0)-(46,25)
      instance (JSON a) => JSON [(String, a)]
        -- Defined at BrokenClass.hs:(50,0)-(52,25)
    In the expression: toJValue [("foo", "bar")]
    In the definition of `it': it = toJValue [("foo", "bar")]
```

重叠实例的问题是 Haskell 的开放世界假设的结果。下面用一个更简单的例子来说明到底发生了什么：

```
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 typeclass `Borked` 的 pairs 实例：一对是 `Int`，另一对是其它任意值。

假设我们希望 `bork` 一对 `Int` 值，那么编译器必须选择一个实例来使用。由于这些实例相邻，那么看起来就能够简单的选择更加明确的实例。

然而 GHC 默认是保守的，它会坚持必须只有一个可能的实例。因此最终在使用 `bork` 时会抛出异常。

Tip

什么时候重叠的实例会有影响?

正如我们之前提到的那样，我们可以将一个 typeclass 的实例散落在若干模块中。GHC 并不会抱怨重叠实例的存在。而真正抱怨的时候就是当我们尝试使用这个受影响 typeclass 的方法时，也就是当强制要求选择哪一个实例需要使用的时候。

放宽 typeclasses 的某些限制

通常而言，我们不可以编写一个特定多态类型的 typeclass 实例。例如 `[Char]` 类型就是 `[a]` 指定 `Char` 类型的多态。因此禁止将 `[Char]` 声明为 typeclass 的实例。这个非常的不

方便，因为字符串在真实代码中处处存在。

`TypeSynonymInstances` 语言扩展移除了这个限制，允许我们编写上述的实例。

GHC 还支持另一个有用的语言扩展，`OverlappingInstances` ，专门用于处理重叠实例。当存在若干重叠实例需要选择时，该扩展使编译器选择最明确的那个。

我们通常将该扩展与 `TypeSynoymInstances` 一起使用。例如：

```
import Data.List

class Foo a where
  foo :: a -> String

instance {-# OVERLAPS #-} Foo a => Foo [a] where
  -- foo = concat . intersperse ", " . map foo
  foo = intercalate ", " . map foo

instance {-# OVERLAPS #-} Foo Char where
  foo c = [c]

instance {-# OVERLAPS #-} Foo String where
  foo = id

main :: IO ()
main = do
  putStrLn $ "foo: " ++ foo "SimpleClass"
  putStrLn $ "foo: " ++ foo ["a", "b", "c"]
```

与原文不同之处在于，注解 `{-# LANGUAGE OverlappingInstances #-}` 在 6.8.1 后被弃用，现在则是在 `instance` 后使用 `{-# OVERLAPS #-}` 来表示重叠的实例，详见文档。

如果我们将 `foo` 应用至一个 `String` ，编译器将使用 `String` 指定的实现。尽管存在 `[a]` 与 `Char` 的 `Foo` 实例，但是 `String` 的实例更加的明确，因此 GHC 会选择它。

如何给类型一个新的身份

略。

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

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

略。

总结：三总命名类型的方法：

- `data` 关键字引入了一个真实的新的代数数据类型。

- `type` 关键字给予了已存在的类型一个同义词。我们可以替换的使用类型与其同义词。
- `newtype` 关键字给予已存在类型一个新的身份。原始类型和新的类型是不可替换的。

JSON typeclasses 没有重叠的实例

现在需要帮助编译器区分 JSON 数组的 `[a]`，以及 JSON 对象的 `[(String,[a])]`。它们是造成重叠问题的原因。我们将列表类型包裹起来，这样编译器则不会视其为列表：

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

当需要将其从模块中导出时，我们需要导出该类型的所有细节。我们模块头看起来像是这样：

```
module JSONClass (JAry (..)) where
```

接下来是另一个包装类型用于隐藏 JSON 对象：

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

有了这些类型定义，那么就可以对 `JValue` 类型做点小修改：

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

这个改动并不会影响已经写过的 `JSON` typeclass 实例，不过还是需要为 `JAry` 与 `JObj` 类型编写 `JSON` typeclass 实例。

```
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
```

让我们慢慢的看一下将 `JAry a` 转换成 `JValue` 的每个步骤。给定一个列表，我们知道其所有元素都是 `JSON` 实例，将其转换成一个列表的 `JValue` 很简单。

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

有了上述代码后，将其变为一个 `JAry JValue` 就仅仅是应用 `newtype` 类型构造函数即可：

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

（记住这并没有性能损耗，仅仅只是告诉编译器隐藏我们在使用一个列表的事实。）将其转换成一个 `JValue` ，我们应用赢一个类型构造函数：

```
jaryOfJValuesToJValue :: JAry JValue -> JValue
jaryOfJValuesToJValue = JArray
```

将这些部分用函数组合的方式集成起来，就获得了一个简洁的一行代码转换成一个 `JValue` ：

```
jaryToJValue = JArray . JAry . map toJValue . fromJAry
```

从 `JValue` 转换至一个 `JAry a` 则需要更多的工作，我们还是将其分解成可复用的部分。基础函数很直接：

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

`whenRight` 函数检查它的参数：如果是由 `Right` 构造函数构建的它则调用一个函数，如果是由 `Left` 构造则保留值不变：

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

更复杂的是 `mapEithers` ，它的行为类似于普通的 `map` 函数，但是如果遇到一个 `Left` 值，它会立刻返回，而不是继续累积一个 `Right` 值的列表。

```
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 []
```

由于隐藏在 `JObj` 类型的列表中的元素有少许结构，因此它与 `JValue` 之间的转换会变得更复杂。幸运的是我们可以复用刚刚定义好的函数：

```
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"
```

7 Input and output

Haskell 中的经典 I/O

略

处理文件以及句柄

Haskell 为 I/O 定义了相当多的基本函数，其中许多函数与其它编程语言中的函数相似。`System.IO` 库中提供了所有的基础 I/O 函数。

使用 `openFile` 会返回一个文件的 `Handle` ，它用于对文件执行指定的操作。Haskell 提供了例如 `hPutStrLn` 这样的函数，其类似于 `putStrLn` ，不同在于接受一个额外的参数 – 一个 `Handle` – 指定哪个文件被操作。当我们结束时，需要用 `hClose` 来结束 `Handle` 。这些函数都定义在 `System.IO` 中，“h”开头的函数对应了几乎所有的非“h”开头的函数；例如 `print` 打印在屏幕上，而 `hPrint` 打印至一个文件。

一个例子：

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

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

  putStrLn "whatever"

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
```

`mainloop` 首先检查输入是否结束（EOF），如果没有则读取一行，将该行转为大写后写入输出文件中，再递归的调用 `mainloop` 。

略

可能的 IOMode 值				
IOMode	**可读?**	**可写?**	**起始位置**	**说明**
ReadMode	Yes	No	文件起始	文件必须存在
WriteMode	No	Yes	文件起始	文件存在时会被清空
ReadWriteMode	Yes	Yes	文件起始	文件不存在时创建；否则现有数据保留
AppendMode	No	Yes	文件尾部	文件不存在时创建；否则现有数据保留

关闭句柄

上述例子中我们已经见到了 `hClose` 用于关闭文件的句柄。在之后的小节中我们将了解缓存 Buffering 这个概念，即 Haskell 在内部为文件维护了缓存。这样做能显著的提升性能，然而这就意味着在对打开的文件调用 `hClose` 之前，数据可能不会被刷新到操作系统中。

另一个原因是打开着的文件消耗系统的资源。如果程序运行的时间很长，开开了很多文件但是并没有关闭它们，那么程序很可能会因为资源枯竭而导致崩溃。

Seek 与 Tell

从一个关联了磁盘文件的 `Handle` 读写时，操作系统会维护一个关于当前位置的内部记录。每当进行下一层读取的时，操作系统返回下一个数据块，其起始点为当前位置，且增加的位置反应了读取了多少数据。

你可以使用 `hTell` 来找到文件中当前的位置。当文件被创建时，它是空的且你的位置将会是 0。在编写 5 个字节后，你的位置则变为了 5，以此类推。`hTell` 接受一个 `Handle` 并返回一个 `IO Integer` 来表示你的位置。

`hTell` 的同伴是 `hSeek` ,它允许你更改文件的位置,其三个参数为: `Handle` , `SeekMode` 以及一个位置。

`SeekMode` 有三个不同类型，用于指定如何解析给定的位置。`AbsoluteSeek` 为文件中的精确位置，等同于 `hTell` ； `RelativeSeek` 意为当前位置开始的多少位置，正值向后，负值向前； `SeekFromEnd` 则是从文件末尾往前多少的位置。

并不是所有的 `Handle` 都是可以 seek 的。一个 `Handle` 通常关联一个文件，但是它还可以关联其它的东西，比如网络连接，磁带机，或者终端。可以使用 `hIsSeekable` 来查看给定的 `Handle` 是否可以 seek。

标准输入，输出以及错误

较早之前我们指出每个非“h”函数通常都会有与其关联的“h”函数用作于任何 `Handle` 上。实际上非“h”函数只不过是“h”函数的一种缩写。

在 `System.IO` 中有三个预定义的 `Handle` ：`stdin` 标准输入，通常是键盘；`stdout` 标准输出，通常是显示器；`stderr` 标准错误，通常也是显示器。

像是 `getLine` 类似的函数可以简单的定义成：

```
getLine = hGetLine stdin
putStrLn = hPutStrLn stdout
print = hPrint stdout
```

删除与文件重命名

`System.Directory` 模块提供了两个比较有用的函数：`removeFile` 接受单个参数，即文件名，然后删除该文件；`renameFile` 接受两个文件名，一个旧名称以及一个新的名称，如果新的文件名是一个不同的路径，那么可以认为这是一个移动。旧的文件名必须在调用 `renameFile` 之前就存在，另外如果新文件已经存在，则重命名后将其删除。

跟很多其它接受一个文件名的函数一样，如果“旧”名称不存在，`renameFile` 则会抛出异常。

`System.Directory` 模块中还有很多其他的函数，像是创建或移除文件夹，在路径中查找文件列表，测试文件是否存在，等等。

临时文件

程序员频繁的需要临时文件。这些文件被用于存储大量等待计算的数据，可被其它程序所用的数据，等等。

通过名为 `openTempFile` 的函数（以及相关联的 `openBinaryTempFile` ）可以帮助我们解决不少问题。

`openTempFile` 接受两个参数：创建文件的路径，以及一个“template”用于命名文件。路径可以是 `.` 来代表当前路径，或者也可以使用 `System.Directory.getTemporaryDirectory` 来找到操作系统所给出的最佳放置临时文件的位置。template 则将一些随机字符添加至文件，用以确保结果是真正唯一的。实际上它保证了可以在一个唯一的文件名上工作。

`openTempFile` 的返回类型是 `IO (FilePath, Handle)` 。元组的第一部分是被创建文件的名称，第二部分则是一个模式为 `ReadWriteMode` 的打开了文件的 `Handle` 。当我们使用完文件，我们希望用 `hClose` 来操作 `Handle` 用于关闭文件，接着调用 `removeFile` 来删除它。接下来我们会看到一个例子用于展示这些函数的用法。

扩展案例：函数式 I/O 以及临时文件

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

```haskell

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

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

  -- Let's see what the initial position is
  pos <- hTell temph
  putStrLn $ "My initial position is " ++ show pos

  -- Now, write some data to the temporary file
  let tempdata = show [1 .. 10]
  putStrLn $ "Writing one line containing " ++ show (length tempdata) ++ " bytes: " ++
   tempdata
  hPutStrLn temph tempdata

  -- Get our new position. This doesn't actually modify pos in memory,
  -- but makes the name "pos" correspond to a different value for
  -- the remainder of the "do" block.
  pos <- hTell temph
  putStrLn $ "After writing, my new position is " ++ show pos

  -- Seek to the beginning of the file and display it
  putStrLn "The file content is: "
  hSeek temph AbsoluteSeek 0

  -- hGetContents performs a lazy read of the entire file
  c <- hGetContents temph

  -- Copy the file byte-for-byte to stdout, followed by \n
  putStrLn $ "c: " ++ c

  -- Let's also display it as a Haskell literal
  putStrLn "Which could be expressed as this Haskell literal:"
  print c

{-
  This function takes two parameters: a filename pattern and another function.
  It will create a temporary file, and pass the name and Handle of that file to
  the given function.

  The temporary file is created with openTempFile. The directory is the one
  indicated by getTemporaryDirectory, or, if the system has no notion of a temporary
  directory, "." is used. The given pattern is passed to openTempFile.

```

```
52    After the given function terminates, even if it terminates due to an exception,
53    the Handle is closed and the file is deleted.
54  -}
55  withTempFile :: String -> (FilePath -> Handle -> IO a) -> IO a
56  withTempFile pattern func = do
57    -- The library ref says that getTemporaryDirectory may raise on exception on
58    -- systems that have no notion of a temporary directory. So, we run
59    -- getTemporaryDirectory under catch. catch takes two functions: one to run,
60    -- and a different one to run if the first raised an exception.
61    -- If getTemporaryDirectory raised an exception, just use "." (the current
62    -- working directory).
63    tempdir <-
64      catch
65        getTemporaryDirectory
66        (\(_ :: IOException) -> return ".") -- explicit annotates exception type
67    (tempfile, temph) <- openTempFile tempdir pattern
68
69    -- Call (func tempfile temph) to perform the action on the temporary file.
70    -- finally takes two actions. The first is the action to run, the second is an action
71    -- to run after the first, regardless of the temporary file is always deleted.
72    -- The return value from finally is the first action's return value.
73    finally
74      (func tempfile temph)
75      ( do
76          hClose temph
77          removeFile tempfile
78      )
```

首先 `withTempFile` 函数证明了 Haskell 在 I/O 时也没有忘记其函数式的天性。该函数接受一个 `String` 与另一个函数。传递给 `withTempFile` 的函数带着临时文件的 `Handle` 被调用，当该函数退出时，临时文件关闭并被删除。

异常处理可以让程序变得更加健壮。通常我们希望在程序结束后删除临时文件，即使程序的执行过程中出现了错误。更多的错误处理会在第十九章讲述。

现在回到程序，`main` 做的事情很简单，调用函数 `withTempFile` 并传入临时文件名以及 `myAction` 函数。

`myAction` 展示了一些信息到终端上，写入了一些数据到文件中，seek 到文件的起始位置，通过 `hGetContents` 进行数据读取，接着按照字节展示文件内容，同时通过 `print c` 进行打印。这等同于 `putStrLn (show c)` 。

现在看看输出：

```
1  % runhaskell tempfiles.hs
2  Welcome to tempfile.hs
3  I have a temporary file at /var/folders/nb/w1q3ztlj139_vz9pqglmbks80000gn/T/mytemp3709-0.txt
4  My initial position is 0
5  Writing one line containing 22 bytes: [1,2,3,4,5,6,7,8,9,10]
6  After writing, my new position is 23
```

```
The file content is:
c: [1,2,3,4,5,6,7,8,9,10]

Which could be expressed as this Haskell literal:
"[1,2,3,4,5,6,7,8,9,10]\n"
```

Lazy I/O

Haskell 还提供了另一种方式处理 I/O。由于 Haskell 是 lazy 语言，意味着任何数据只会在其值必须被知道时才会被计算，那么就有了一些新颖的 I/O 处理方法。

hGetContents

一个新颖的 I/O 方式就是 `hGetContents` 函数，其类型 `Handle -> IO String` ，这里的 `String` 代表着由文件的 `Handle` 所返回的所有数据。

在严格求值的语言中，使用这样的函数通常是一个坏主意。假设读一个 500GB 的文件，那么就有可能因为内存不足而导致程序崩溃。在这些语言中，传统的做法就是使用循环来处理文件的所有数据。

但是 `hGetContents` 不一样，它返回的 `String` 是惰性的，即在调用 `hGetContents` 时，并没有读取任何数据。只有在处理列表的元素（字符）时才会从 `Handle` 中读取数据。当 `String` 的元素不再使用时，Haskell 的垃圾收集器则会自动释放内存。

来看一个例子：

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

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
```

测试：

```
ghci> :load toupper-lazy1.hs
[1 of 1] Compiling Main             ( toupper-lazy1.hs, interpreted )
Ok, modules loaded: Main.
ghci> processData "Hello, there!  How are you?"
```

```
"HELLO, THERE!  HOW ARE YOU?"
ghci> :type processData
processData :: String -> String
ghci> :type processData "Hello!"
processData "Hello!" :: String
```

Warning

如果我们在上述例子中尝试在使用 `inpStr` 的地方（对 `processData` 的调用）之后继续使用 `inpStr` ，那么程序则会失去其内存效率。这是因为编译器将会强制保留 `inpStr` 值在内存中供未来使用。在这里编译器知道 `inpStr` 不会再被使用，那么就在其使用结束后尽快的释放内存。这里需要记住：内存只有在最后一次使用后才会被释放。

这段代码有点啰嗦，我们可以让它变得更简洁一些：

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

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

在使用 `hGetContents` 时，我们甚至不需要从输入文件中消费所有的数据。每当 Haskell 系统确定 `hGetContents` 所返回的整个字符可以被垃圾回收时 – 意味着它不会再被使用 – 文件将自动被关闭。同样的原则也适用于从文件中读取的数据。每当给定的数据不再需要时，Haskell 环境就会释放存储该数据的内存。严格来说我们完全都不需要调用 `hClose` 。然而这是一个好的习惯，因为之后对程序的改动会使得 `hClose` 变得很重要。

Warning

在使用 `hGetContents` 时，必须记住即使在之后的程序可能不再显式的引用 `Handle` ，但还是要保持 `Handle` 不被关闭，直到 `hGetContents` 的结果被消费掉了。不这么做的话就可能会导致丢失部分或者全部文件数据。这是因为 Haskell 是惰性的，所以通常可以假设只有在输出了涉及输入的计算结果之后才消费了输入。

readFile 与 writeFile

Haskell 程序员经常使用 `hGetContents` 作为筛选。从一个文件读取数据，处理数据，接着将数据写到别的地方。这非常的通用所以有了一些快捷的方式：`readFile` 以及 `writeFile` 就是这样的函数，以字符串的方式处理文件。它们处理了所有关于打开文件，关闭文件，读取数据，以及写入数据的细节。`readFile` 在内部使用了 `hGetContents` 。

通过 **ghci** 查看这些函数的类型：

```
ghci> :type readFile
readFile :: FilePath -> IO String
ghci> :type writeFile
writeFile :: FilePath -> String -> IO ()
```

使用 `readFile` 与 `writeFile` 改造之前的代码：

```
import Data.Char (toUpper)

main :: IO ()
main = do
  inpStr <- readFile "input.txt"
  writeFile "output.txt" $ map toUpper inpStr
```

关于 Lazy Output

现在我们能理解惰性输入是如何在 Haskell 中工作的。那么惰性输出又是怎么样呢?

我们知道 Haskell 中所有计算都是在其值被需要时才会进行。由于函数例如 `writeFile` 以及 `putStr` 会输出所有传给它们的 `String` ，那么整个 `String` 必须被计算。因此可以保证 `putStr` 的参数将被完整求值。

那么输入惰性的意义呢? 上述例子中，对 `putStr` 或 `writeFile` 的调用是否会强制将整个输入字符串立刻加载到内存中，仅仅只是为了输出?

回答是不。`putStr` （以及其它类似的所有输出函数）在数据可同时，输出数据。它们同样也不需要保留已经输出了的数据，因此只要程序中没有其它东西需要它，内存就可以立刻释放。在某种意义上，可以将 `readFile` 与 `writeFile` 之间的 `String` 视为连接两者的管道。数据进入一端，以某种方式转换后，从另一端流出。

interact

通过 `interact` 函数可以进一步简化我们的程序：

```
import Data.Char (toUpper)

main :: IO ()
main = interact $ map toUpper
```

仅用了一行！测试：

```
$ runghc toupper-lazy4.hs < input.txt > output.txt
```

或者想要打印到屏幕上：

```
$ runghc toupper-lazy4.hs < input.txt
```

如果想要看到 Haskell 的输出确实是在接收到数据块后立刻输出到数据块，可以不加任何参数运行 `runghc toupper-lazy4.hs` ，这样就可以看到在敲击输入回车后，所有字符都会被立刻以大写的形式输出。

我们同样还可以通过 `interact` 编写在输出前添加新行的程序：

```
import Data.Char (toUpper)

main = interact $ map toUpper . (++) "Your data, in uppercase, is:\n\n"
```

由于我们在 `(++)` 后调用的 `map` ，那么新行也会变为大写，修改一下：

```
import Data.Char (toUpper)

main = interact $ (++) "Your data, in uppercase, is:\n\n" . map toUpper
```

这就将新行移到 `map` 外去了。

interact 加上过滤

`interact` 的另一个常用方法就是过滤。比如打印带有字符“a”的每行：

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

通过 **ghci** 查看上述的三个新函数：

```
ghci> :type lines
lines :: String -> [String]
ghci> :type unlines
unlines :: [String] -> String
ghci> :type elem
elem :: (Eq a) => a -> [a] -> Bool
```

测试：

```
$ runghc filter.hs < input.txt
I like Haskell
Haskell is great
```

IO 单子

Actions

大多数语言不会区分纯函数和非纯函数。Haskell 的函数有其数学意义：它们是存粹的计算，且不会被任何外界的东西改变。另外，计算可以在任何时刻进行。

那么就需要另一些工具来处理 I/O 了。这个工具在 Haskell 中叫做 *actions*。Actions 类似于函数，在定义的时候不会做任何事情，而是在执行任务时才会被唤醒。I/O actions 被定义在 `IO` 单子中。我们会在第十四章中讲解单子。看看一些类型：

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

`putStrLn` 的类型跟其它函数无异。函数接受一个参数并返回一个 `IO ()` 。这个 `IO ()` 就是 action。我们可以存储以及传递 actions 至纯代码中，尽管这并不常用。一个 action 不会做任何事情直到它被唤起。看看例子：

```
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 (fst : remaining) = do
  fst
  runall remaining

main :: IO ()
main = do
  str2action "Start of the program"
  printitall
```

```
30    str2action "Done!"
```

`str2action` 这个函数接受一个参数并返回一个 `IO ()` 。在 `main` 的结尾，可以在另一个 action 中直接使用它，并打印出一行。或者可以存储 – 而不是执行 – 这个 action 到纯代码中。在 `list2actions` 中 – 使用了 `map` 应用在 `str2action` 上并返回一个列表的 actions，正如我们在纯代码中的那样。可以看到通过 `printitall` 的所有内容都是使用纯的工具构建的。

尽管定义了 `printitall` ，它不会被执行直到它的 action 在某处被计算。注意在 `main` 中是如何将 `str2action` 作为一个 I/O action 被执行的，而之前我们在 I/O 单子外使用它，并将结果转为一个列表。

可以这么理解：每个声明，除了 `let` ，在一个 `do` 代码块中必须产生一个将要执行的 I/O action。

对 `printitall` 的调用最终执行所有这些操作。实际上由于 Haskell 是惰性的，所以直到这里才会生成 actions。

测试：

```
% runhaskell actions.hs
Data: Start of the program
Data: 1
Data: 2
Data: 3
Data: 4
Data: 5
Data: 6
Data: 7
Data: 8
Data: 9
Data: 10
Data: Done!
```

实际上我们可以将其写成更紧凑的形式。以下是重构：

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

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

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

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

注意 `str2action` 中使用了标准函数组合操作符。在 `main` 中，调用了 `mapM_` ，该函数类似于 `map` 接受一个函数与一个列表。而提供给 `mapM_` 的函数则是一个应用至列表中每个元素的 I/O action 。`mapM_` 会抛出函数的结果，不过如果有需要，也可以使用 `mapM` 返回 I/O 结果的列表。看一下它们的类型：

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

> **Tip**
> 实际上这些函数并不仅仅作用于 I/O；它们可用于任何 `Monad` 。

为什么有了 `map` 还需要 `mapM` 呢？因为 `map` 是一个返回列表的纯函数，它不能直接执行 actions，而 `mapM` 则是 `IO` 单子中的工具，可用于实际执行 actions。

回到 `main` ，`mapM_` 应用了 `(str2action . show)` 在每个 `numbers` 中的元素上。`show` 将每个数值转为 `String` ，而 `str2action` 转换每个 `String` 为一个 action。`mapM_` 结合这些独立的 actions 成为一个大的 action 并打印出来。

Sequencing

`do` 代码块其实是合并所有 actions 的简便注解。在不使用 `do` 时还有两个操作符可以使用：`>>` 与 `>>=` 。

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

`>>` 操作符将两个 actions 串联起来：先执行第一个 action，再执行第二个 action。整个计算的结果则是第二个 action 的结果，而第一个 action 的结果则被丢弃。这类似于 `do` 代码块中的一行。可以用 `putStrLn "line 1" >> putStrLn "line 2"` 来测试一下。它将会打印出两行，将第一个 `putStrLn` 的结果丢弃，将第二个的结果作为返回。

`>>=` 操作符运行一个 action，然后将其结果传给一个返回 action 的函数。第二个 action 也将运行，整个表达式的结果还是第二个 action 的结果。例如，可以用 `getLine >>= putStrLn` 测试，它从键盘读取一行然后再将其输出。

现在不用 `do` 代码块来重写一下之前的例子：

```
main :: IO ()
main = do
  putStrLn "Greetings! What is your name?"
```

```
    inpStr <- getLine
    putStrLn $ "Welcome to Haskell, " ++ inpStr ++ "!"
```

不使用 `do` ：

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

在定义 `do` 代码块时，Haskell 编译器内部会执行类似的转换。

Return 的本质

在本章开头我们提到了 `return` 可能并不像是它长得那样。很多语言的关键字 `return` 是立刻终结函数的执行并返回一个值给到调用者。

Haskell 中的 `return` 函数则大相径庭。Haskell 中的 `return` 将数据包装进单子中。在谈及 I/O 时，`return` 用于将纯数据放入 IO 单子中。

为什么要这么做呢？记住，任何结果依赖于 I/O 的东西都必须在 IO 单子内。因此，如果我们正在编写一个执行 I/O 的函数，那么一个纯计算，我们需要使用 `return` 来使这个纯计算成为函数的正确返回值。例如：

```
import Data.Char (toUpper)

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

我们有一个返回 `Bool` 的纯计算。计算传递给 `return` ，将其放入 `IO` 单子内。由于它是 `do` 代码块中的最后一个值，它变成了 `isGreen` 的返回值，但这并不因为我们使用了 `return` 函数。

下面是同样一个程序，只不过将纯计算剥离成为一个独立的函数。这样可以隔离纯函数，同样使得意图更加的清晰。

```
import Data.Char (toUpper)

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

isGreen :: IO Bool
isGreen = do
  putStrLn "Is green your favorite color?"
  isYes <$> getLine
```

与原文不同之处在于使用了 `<$>` ，即 `fmap` 函数，替换掉了之前的 `inpStr <- getLine` 以及 `return` 。

最后是一个做作的例子用于展示 `return` 并不需要位于 `do` 代码块的结尾。

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

Haskell 真的是命令式的吗?

这些 `do` 代码块看起来像是命令式的语言。毕竟大多数情况下我们都是在按顺序运行命令。

但是 Haskell 本质上仍然是一种惰性语言。虽然有时有必要对 I/O 操作进行排序，但这是使用了 Haskell 中的工具来完成的。Haskell 还通过 IO 单子实现了 I/O 与语言其他部分的良好分离。

Lazy I/O 的副作用

本章提到的 `hGetContents` ，其返回的 `String` 可以用在纯代码中。

我们需要更具体的了解副作用是什么。当我们说 Haskell 没有副作用时，这到底是什么意思?

某种层度上，副作用总是可能的。一个没有优化的循环，即使是用纯代码写的，也有可能导致系统的内存耗尽然后导致程序崩溃。或者它可能导致数据被交换到磁盘。

当我们提到没有副作用，意思其实是 Haskell 的纯代码不会运行能触发副作用的命令。纯函数不会修改一个全局变量，I/O 请求，或者运行一个命令关闭系统。

当你有一个 `hGetContents` 而来的字符串被传递至一个纯函数时，函数不会知道 `String` 是从磁盘文件而来的。它只会表现得跟通常那样，但处理该 `String` 可能会导致环境发出 I/O 命令。纯函数不会这么做；它们是纯函数处理的结果，就像将内存交换到磁盘的例子一样。

在某些情况下，你可能需要更多的控制 I/O 发生的确切时间。也许是正在交互式的从用户那里读取数据，或者通过管道从另一个程序读取数据，并需要直接与用户通信。这些情况下，`hGetContents` 可能不合适。

缓存

缓存模式

Haskell 有三种不同的缓存模式：`BufferMode` 类型：`NoBuffering` ，`LineBuffering` 以及 `BlockBuffering` 。

`NoBuffering` 正如其名 – 无缓存。通过例如 `hGetLine` 等函数读取的数据将每次从操作系统重读取一个字符。写入的数据将立刻写入，也经常一次写入一个字符。通常 `NoBuffering` 的性能很差，不适合通用用途。

`LineBuffering` 使输出缓冲区在输出换行符或缓存过大时被写入。在输入时，它通常会尝试读取任何可用的数据块，直到它第一次看到换行符。当从终端读取数据时，每次按下回车键后应该立刻返回数据。这通常是合理的默认值。

`BlockBuffering` 使 Haskell 在可能的情况下以固定大小块来读写数据。在批量处理大量数据时，这是性能最好的方法，即使这些数据是以行记录的。然而它不能用于交互式程序，因为它会阻塞输入，直到读取到完整的块。`BlockBuffering` 接受 `Maybe` 类型的参数：如果 `Nothing` ，它将使用定义好的缓存大小；或者你可以使用 `Just 4096` 之类的设置将缓存设置为 4096 字节。

默认和缓存模式取决于操作系统和 Haskell 的实现。可以通过调用 `hGetBuffering` 向系统询问当前的缓存模式。当前的缓存模式可以通过 `hSetBuffering` 设置，它接受 `Handle` 和 `BufferMode` 。例如 `hSetBuffering stdin (BlockBuffering Nothing)` 。

刷新缓存

对于任何类型的缓存，有时我们会希望强制 Haskell 写出保存在缓存中的任何数据。有些时候这会自动发生：比如说调用 `hClose` 时。而调用 `hFlush` 将强制立刻写入任何挂起的数据。当 `Handle` 是一个网络 socket 且想要立刻传输数据时，或者是想要将磁盘上的数据提供给可能并发读取它的其它程序时，这会很有用。

读取命令行参数

`System.Environment.getArgs` 以 `IO [String]` 返回所有参数。类似 C 里的 `argv` ，始于 `argv[1]` 。程序名（C 中的 `argv[0]` ）则可以通过 `System.Environment.getProgName` 得到。

`System.Console.GetOpt` 模块提供了一些解析命令行选项的工具。如果我们的程序有一些复杂的选项，该模块则很有帮助。

环境变量

读取环境变量可以用 `System.Environment` 的两个函数：`getEnv` 或 `getEnvironment`。前者查找特定变量，变量不存在时抛出异常；后者将所有环境变量以 `[(String, String)]` 输出，可以使用例如 `lookup` 函数来找到所需的变量。

8 Efficient file processing, regular expressions, and file name matching

高效文件处理

下面是一个简单的微基准测试，读取一个全是数字的文本文件，打印它们的总和：

```haskell
main :: IO ()
main = do
  contents <- getContents
  print $ sumFile contents
  where
    sumFile = sum . map read . words
```

尽管 `String` 是用于读写文件的默认类型，它并不高效，因此像这样一个简单的程序性能是很差的。

`String` 代表着 `Char` 值的列表；列表中每个元素的内存分配都是独立的，需要额外的开销。这些因素会影响必须读写文本或二进制数据程序的内存消耗和性能。像这样的简单基准测试，即使是用解释性语言（如 Python）编写的程序，性能也比使用 `String` 的 Haskell 代码高出一个数量级。

`bytestring` 库提供了一个快速并且又低开销的 `String` 类型替代品。通过 `bytestring` 编写的代码通常可以媲美或超过 C 语言的性能和内存消耗，同时还保持了 Haskell 的表现力和简洁性。

该库提供了两个模块。每个定义的函数几乎都是 `String` 对应函数的直接替代品：

- `Data.ByteString` 模块定义了一个严格 *strict* 类型的 `ByteString` 。其以二进制或文本数据的数组形式来表示字符串。
- `Data.ByteString.Lazy` 模块提供了惰性 *lazy* 类型的 `ByteString` 。其以块 *chunks* 数组来表示字符串，最大大小为 64KB。

两个 `ByteString` 类型在特定环境都有更好的表现。对于流式处理一个大数据（几百 MB 至 TB），惰性 `ByteString` 类型表现的最好。它的块大小被调整为适合现代 CPU 的 L1 缓存，且垃圾回收可以快速丢弃不再使用的流数据块。

而严格的 `ByteString` 类型在不太关心内存占用或需要随机访问数据的环境下性能更好。

二进制 I/O 与 qualified imports

现在来开发一个函数来说明 `ByteString` 的一些 API。我们将决定一个文件是否为 ELF 对象：这种格式多用于现代 Unix 类型系统的可执行文件。

查看文件的首四个字节，检查它们是否匹配一个字节列表：

```haskell
import Data.ByteString.Lazy qualified as L

hasElfMagic :: L.ByteString -> Bool
hasElfMagic content = L.take 4 content == elfMagic
  where
    elfMagic = L.pack [0x7f, 0x45, 0x4c, 0x46]
```

`ByteString` 的惰性模块以及严格模块意图解决二进制 I/O。Haskell 用 `Word8` 来展示字节的数据类型；可以通过 `Data.Word` 来引入它。

上述例子的 `L.pack` 函数接受一个 `Word8` 列表，打包它们成为一个惰性的 `ByteString`。（`L.unpack` 函数作用相反。）而 `hasElfMagic` 函数则是比较 `ByteString` 的前四个字符。

以下是将其运用在一个文件上的例子：

```haskell
isElfFile :: FilePath -> IO Bool
isElfFile path = do
  content <- L.readFile path
  return $ hasElfMagic content
```

`L.readFile` 函数是 `readFile` 的惰性 `ByteString` 版本，即按需读取。它效率很高，每次读取最大 64KB。惰性 `ByteString` 很适合我们的任务：因为我们至多读取文件的前四个字节，因此我们可以安全的使用该函数在任何大小的文件上。

Text I/O

`bytestring` 库还提供了两个模块用于 text I/O 功能，`Data.ByteString.Char8` 以及 `Data.ByteString.Lazy.Char8`。它们将单独的字符串元素导出成 `Char` 而不是 `Word8`。

Warning

上述模块中的函数进作用于字节大小的 `Char` 值，也就是仅适用于 ASCII 以及某些欧洲字符集。超过 255 的字符会被截断。

面向字符的 `bytestring` 模块为文本处理提供了一些有用的函数。以下是包含了月度的股票价格的文件：

```
ghci> putStr =<< readFile "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
```

如何找到最高价呢？收盘价在第四列，用都好分隔。下面是得到收盘价的函数：

```haskell
closing = readPrice . (!! 4) . L.split ','
```

该函数是 point-free 风格，需要从右往左阅读。`L.split` 将一个惰性的 `ByteString` 分隔开来，分隔发生在每一次找到匹配值时。`(!!)` 操作符则是获取列表中第 k 个元素。`readPrice` 函数将一个代表分数的字符串转换为数值：

```
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)
```

上面使用了 `L.readInt` 函数，即解析整数，返回整数以及字符串剩余部分。

找到最高收盘价的函数：

```
highestClose = maximum . (Nothing :) . map closing . L.lines

highestCloseFrom path = do
  contents <- L.readFile path
  print $ highestClose contents
```

这里使用了一个小技巧。当我们提供一个空列表给 `maximum` 函数会抛出异常：

```
ghci> maximum [3,6,2,9]
9
ghci> maximum []
*** Exception: Prelude.maximum: empty list
```

测试：

```
ghci> :load HighestClose
[1 of 1] Compiling Main             ( HighestClose.hs, interpreted )
Ok, modules loaded: Main.
ghci> highestCloseFrom "prices.csv"
Loading package array-0.1.0.0 ... linking ... done.
Loading package bytestring-0.9.0.1 ... linking ... done.
Just 2741
```

因为从逻辑中分离了 I/O，测试空数据时不需要创建一个空文件：

```
ghci> highestClose L.empty
Nothing
```

文件名匹配

很多面向系统的编程语言都提供了根据模式来匹配文件的功能。尽管 Haskell 的标准库有不错的系统编程工具，它并没有提供这类的匹配函数。

这类模式通常被称为 `glob 模式` ，通配符模式或者 shell 样式模式。它们有一些简单的规则：

- 将字符串与模式进行匹配，从字符串的开头开始，到末尾结束。
- 大多字面字符都匹配自己。例如，模式中的文本 foo 将匹配输入字符串中的 foo，且仅匹配 foo。
- `*` （星号）字符表示“匹配任何内容”；它将匹配任何文本，包括空字符串。
- `?` （问号）字符匹配任何单个字符。模式 `pic??.jpg` 将匹配 `picaa.jpg` 或 `pic01.jpg` 等名称。
- `[` （开方括号）字符开始一个字符类，以 `]` 结束。它的意思是“匹配这个类中的任何字符”。字符类可以在 `[` 后面加一个 `!` 来表示“匹配不属于这个类的任何字符”。

  作为一种简写，一个字符后面跟着一个 `-` （破折号），后面跟着另一个字符，表示一个范围：”匹配此集合中的任何字符“

虽然 Haskell 的标准库中没有提供 glob 模式的匹配，但是它提供了一个非常好的正则表达式库。

Haskell 中的正则表达式

首先让我们加载 `Text.Regex.Posix` 库

```
ghci> :module +Text.Regex.Posix
```

根据官方 Wiki提到的 `regex-posix` 速度非常的慢，在生产代码中需要换成别的库。

我们这里只需要正则表达式匹配的函数，一个中缀操作符 `(=~)` 即可（从 Perl 中借用）。要克服的第一个障碍是 Haskell 的 regexp 库大量使用了多态。因此 `(=~)` 操作符的类型签名很难理解，因此这里暂时不做解释。

`=~` 对它的两个参数和返回类型使用了 typeclasses。第一个参数（即 `=~` 左侧）是要匹配的文本；第二个参数（右侧）则是要匹配的正则表达式。我们可以传 `String` 或 `ByteString` 作为参数。

结果的多种类型

`=~` 操作符的结果类型是多态的，因此 Haskell 编译器需要知道其类型。在真实代码中，它有可能根据使用的场景被推导出正确的类型。但是 **ghci** 没法这么做，它没有足够的信息来进行推导。

当 **ghci** 不能推导 `target` 类型时，我们需要进行显式指定：

```
ghci> "my left foot" =~ "foo" :: Bool
Loading package array-0.1.0.0 ... linking ... done.
Loading package containers-0.1.0.1 ... linking ... done.
Loading package bytestring-0.9.0.1 ... linking ... done.
Loading package mtl-1.1.0.0 ... linking ... done.
Loading package regex-base-0.93.1 ... linking ... done.
Loading package regex-posix-0.93.1 ... linking ... done.
True
ghci> "your right hand" =~ "bar" :: Bool
False
ghci> "your right hand" =~ "(hand|foot)" :: Bool
True
```

正则表达式库的核心是一个名为 `RegexContext` 的 typeclass，用于描述 `target` 类型的行为；基础库为我们定义了很多实例。`Bool` 类型就定义了该 typeclass 的实例，因此我们能得到可用的结果。另一个实例就是 `Int` ：

```
ghci> "a star called henry" =~ "planet" :: Int
0
ghci> "honorificabilitudinitatibus" =~ "[aeiou]" :: Int
13
```

如果寻求的是 `String` 结果，我们将得到第一个匹配的子字符串，或是不匹配时得到一个空字符串。

```
ghci> "I, B. Ionsonii, uurit a lift'd batch" =~ "(uu|ii)" :: String
"ii"
ghci> "hi ludi, F. Baconis nati, tuiti orbi" =~ "Shakespeare" :: String
""
```

另外一个可作为结果的类型是 `[String]` ，其返回所有匹配字符串的列表：

```
ghci> "I, B. Ionsonii, uurit a lift'd batch" =~ "(uu|ii)" :: [String]

<interactive>:1:0:
    No instance for (RegexContext Regex [Char] [String])
      arising from a use of `=~' at <interactive>:1:0-50
    Possible fix:
      add an instance declaration for
      (RegexContext Regex [Char] [String])
    In the expression:
          "I, B. Ionsonii, uurit a lift'd batch" =~ "(uu|ii)" :: [String]
    In the definition of `it':
        it = "I, B. Ionsonii, uurit a lift'd batch" =~ "(uu|ii)" ::
             [String]
ghci> "hi ludi, F. Baconis nati, tuiti orbi" =~ "Shakespeare" :: [String]

<interactive>:1:0:
    No instance for (RegexContext Regex [Char] [String])
      arising from a use of `=~' at <interactive>:1:0-54
```

```
    Possible fix:
      add an instance declaration for
      (RegexContext Regex [Char] [String])
    In the expression:
            "hi ludi, F. Baconis nati, tuiti orbi" =~ "Shakespeare" :: [String]
    In the definition of `it':
        it = "hi ludi, F. Baconis nati, tuiti orbi" =~ "Shakespeare" ::
             [String]
```

Tip

注意字符串返回

如果期望结果是一个 `String` ，那么就要小心了。因为 `(=~)` 返回一个空字符传来表示”没有匹配“，如果空字符串也可能是 regexp 的有效匹配，那么就会难以识别。在这种情况下就应该是有一个不同的返回类型，例如 `[String]` 。

上述都是”简单的“返回类型，这远没有结束。继续往下推进前，让我们定义一个在之后例子中都会用到的匹配：

```
ghci> let pat = "(foo[a-z]*bar|quux)"
```

我们可以获得关于匹配发生的上下文的大量信息。如果期望是 `(String,String,String)` 元组，则会在第一个匹配之前获得文本，匹配成功的文本，以及紧随其后的文本。

```
ghci> "before foodiebar after" =~ pat :: (String,String,String)
("before ","foodiebar"," after")
```

如果匹配失败，那么整个文本则会在元组中以”之前“元素作为返回，而其它两个元素为空：

```
ghci> "no match here" =~ pat :: (String,String,String)
("no match here","","")
```

请求一个四元素元组则会给我们第四个元素，它是匹配模式中所有组的列表：

```
ghci> "before foodiebar after" =~ pat :: (String,String,String,[String])
("before ","foodiebar"," after",["foodiebar"])
```

我们还可以获取匹配的数值信息。一对 `Int` 给出的第一个匹配的起始偏移量以及长度。如果我们请求这些配对的列表，我们将得到所有匹配的信息。

```
ghci> "before foodiebar after" =~ pat :: (Int,Int)
(7,9)
ghci> "i foobarbar a quux" =~ pat :: [(Int,Int)]

<interactive>:1:0:
    No instance for (RegexContext Regex [Char] [(Int, Int)])
      arising from a use of `=~' at <interactive>:1:0-26
    Possible fix:
```

```
        add an instance declaration for
        (RegexContext Regex [Char] [(Int, Int)])
    In the expression: "i foobarbar a quux" =~ pat :: [(Int, Int)]
    In the definition of `it':
        it = "i foobarbar a quux" =~ pat :: [(Int, Int)]
```

如果请求单个元组，则用值 `-1` 作为元组的第一个元素（匹配偏移量）表示匹配失败；如果请求元组列表，则用空列表表示匹配失败：

```
ghci> "eleemosynary" =~ pat :: (Int,Int)
(-1,0)
ghci> "mondegreen" =~ pat :: [(Int,Int)]

<interactive>:1:0:
    No instance for (RegexContext Regex [Char] [(Int, Int)])
      arising from a use of `=~' at <interactive>:1:0-18
    Possible fix:
      add an instance declaration for
      (RegexContext Regex [Char] [(Int, Int)])
    In the expression: "mondegreen" =~ pat :: [(Int, Int)]
    In the definition of `it': it = "mondegreen" =~ pat :: [(Int, Int)]
```

更多的关于正则表达式

混合与匹配字符串类型

之前提到 `=~` 操作符使用 typeclasses 作为其参数类型以及返回类型。可以使用 `String` 或严格 `ByteString` 的值来进行正则表达式计算。

```
ghci> :module +Data.ByteString.Char8
ghci> :type pack "foo"
pack "foo" :: ByteString
```

可以尝试不同组合的 `String` 与 `ByteString`：

```
ghci> pack "foo" =~ "bar" :: Bool
False
ghci> "foo" =~ pack "bar" :: Int
0
ghci> pack "foo" =~ pack "o" :: [(Int, Int)]

<interactive>:1:0:
    No instance for (RegexContext Regex ByteString [(Int, Int)])
      arising from a use of `=~' at <interactive>:1:0-21
    Possible fix:
      add an instance declaration for
      (RegexContext Regex ByteString [(Int, Int)])
    In the expression: pack "foo" =~ pack "o" :: [(Int, Int)]
```

```
    In the definition of `it':
        it = pack "foo" =~ pack "o" :: [(Int, Int)]
```

不过需要注意的是，如果期望匹配结果中包含字符串值，那么所匹配的文本必须是同一类型的字符串：

```
ghci> pack "good food" =~ ".ood" :: [ByteString]

<interactive>:1:0:
    No instance for (RegexContext Regex ByteString [ByteString])
      arising from a use of `=~' at <interactive>:1:0-25
    Possible fix:
      add an instance declaration for
      (RegexContext Regex ByteString [ByteString])
    In the expression: pack "good food" =~ ".ood" :: [ByteString]
    In the definition of `it':
        it = pack "good food" =~ ".ood" :: [ByteString]
```

上述例子中，使用 `pack` 将 `String` 转为了 `ByteString` 。类型检查器可以接受是因为 `ByteString` 在结果类型中展示出来了。但是如果尝试获取一个 `String` ，则不会成功：

```
ghci> "good food" =~ ".ood" :: [ByteString]

<interactive>:1:0:
    No instance for (RegexContext Regex [Char] [ByteString])
      arising from a use of `=~' at <interactive>:1:0-20
    Possible fix:
      add an instance declaration for
      (RegexContext Regex [Char] [ByteString])
    In the expression: "good food" =~ ".ood" :: [ByteString]
    In the definition of `it':
        it = "good food" =~ ".ood" :: [ByteString]
```

我们可以通过将左侧的字符串类型与结果再次匹配来简单的修复这个问题：

```
ghci> "good food" =~ ".ood" :: [String]

<interactive>:1:0:
    No instance for (RegexContext Regex [Char] [String])
      arising from a use of `=~' at <interactive>:1:0-20
    Possible fix:
      add an instance declaration for
      (RegexContext Regex [Char] [String])
    In the expression: "good food" =~ ".ood" :: [String]
    In the definition of `it': it = "good food" =~ ".ood" :: [String]
```

其它需要注意的事项

查看 Haskell 库文档时，可以看到几个与 regexp 相关的模块。`Text.Regex.Base` 模块定义了所有其他 regexp 模块所遵循的通用 API。可以同时安装 regexp API 的多个实现。在编写本文时，GHC 与一个实现 `Text.Regex.Posix` 捆绑在一起。顾名思义，这个包提供 POSIX regexp 语义。

Tip

Perl 与 POSIX 正则表达式

如果是从 Perl、Python 或 Java 等语言来学习 Haskell 的，并且在其中一种语言中使用过正则表达式，那么应该意识到由 `Text.Regex.Posix` 模块处理的 POSIX regexp 与 Perl 风格的 `regexp` 在某些重要方面是不同的。以下是一些比较明显的区别。

Perl regexp 引擎在匹配备选项时执行左偏匹配，而 POSIX 引擎选择最贪婪的匹配。这意味着给定一个正则表达式 `(foo|fo*)` 和一个文本字符串 `fooooo`，perl 风格的引擎将给出 `foo` 的匹配（最左边的匹配），而 POSIX 引擎将匹配整个字符串（最贪婪的匹配）。

POSIX regexp 的语法不如 perl 风格的 regexp 统一。它们还缺乏 perl 风格的 regexp 提供的许多功能，例如零宽度断言和对贪婪匹配的控制。

将一个 glob 模式转为一个正则表达式

我们已经见识到了很多种将文本域正则表达式匹配的方法，让我们将注意力转回到 glob 模式。我们希望编写一个接受一个 glob 模式的函数，将其表示形式返回为正则表达式。glob 模式和正则表达式两者都是 text 字符串，因此函数的类型就很明确了：

```
module GlobRegex
  ( globToRegex,
  )
where

import Text.Regex.Posix ((=~))

globToRegex :: String -> String
```

我们生成的正则表达式必须是**固定的**，使得它可以从同到尾的匹配字符串。

```
globToRegex cs = '^' : globToRegex' cs ++ "$"
```

由于 `String` 就是 `[Char]` 的别名，这里 `:` 操作符将一个值（这里是 `^` 字符）放到一个列表的头部，而这个列表则是还未出现的 `globToRegex` 函数所返回的。

Tip

在定义一个值之前使用它

Haskell 使用一个值或者函数时，并不需要它们被声明或者在一个源文件中定义。一个定义出现在第一个被使用的地方之后是很正常的。Haskell 编译器并不关心这一层面上的顺序。这使我们可以灵活的按照逻辑来构建代码，而不是遵循编译器编写者的顺序。

Haskell 模块的编写者通常会使用这个灵活性来让“更加重要”的代码前置于源文件中。这正式我们编写 `globToRegex` 函数以及其帮助函数的方式。

有了正则表达式以后 `globToRegex'` 函数则用作于大部分的翻译工作。以下使用 Haskell 的模式匹配来进行编码：

```
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
```

第一个子句规定了如果达到了 glob 模式的末尾（到那时会看到空字符串），返回 `$` ，即正则表达式中的“匹配到了末尾”。接下来的是的子句则是一系列从 glob 语义切换到正则表达语义的模式。最后一个子句处理其它的每个字符，转义优先。

`escape` 函数确保正则引擎不会解释某些特定字符：

```
escape :: Char -> String
escape c
  | c `elem` regexChars = '\\' : [c]
  | otherwise = [c]
  where
    regexChars = "\\+()^$.{}]|"
```

`charClass` 帮助函数用于检测字符是否正确的终止，不修改输入直到遇到 `]` ：

```
charClass :: String -> String
charClass (']' : cs) = ']' : globToRegex' cs
charClass (c : cs) = c : charClass cs
charClass [] = error "unterminated character class"
```

我们已经定义好了 `globToRegex` 与其帮助函数们，现在加载进 **ghci** 中测试一下：

```
ghci> :set -package regex-posix
package flags have changed, resetting and loading new packages...
ghci> :l GlobRegex.hs
[1 of 1] Compiling GlobRegex        ( GlobRegex.hs, interpreted )
```

```
Ok, one module loaded.
ghci> globToRegex "f??.c"
"^f..\\.c$"
```

看起来像是一个合理的正则，那么用它来匹配字符串呢?

```
ghci> "foo.c" =~ globToRegex "f??.c" :: Bool
True
ghci> "test.c" =~ globToRegex "t[ea]s*" :: Bool
True
ghci> "teste.txt" =~ globToRegex "t[ea]s*" :: Bool
True
```

正常工作！我们还可以为 `fnmatch` 创建一个临时定义：

```
ghci> let fnmatch pat name = name =~ globToRegex pat :: Bool
ghci> :type fnmatch
fnmatch
  :: rgx-bs-0.94.0.2-01560e7d:Text.Regex.Base.RegexLike.RegexLike
       Text.Regex.Posix.Wrap.Regex source1 =>
     String -> source1 -> Bool
ghci> fnmatch "d*" "myname"
False
```

`fnmatch` 并不符合“Haskell 本性”，通常 Haskell 风格的函数都需要描述性的“驼峰”名称，例如“file name matches”为 `fileNameMatches` ，那么我们的库可以这样命名：

```
matchesGlob :: FilePath -> String -> Bool
name `matchesGlob` pat = name =~ globToRegex pat
```

一个重要的旁白：编写惰性函数

在一个命令式的语言中，`globToRegex'` 函数通常会表述为一个循环。例如 Python 的标准 fnmatch 模块中包含了一个名为 `translate` 的函数，其功能与我们的 `globToRegex` 函数一致。它的实现就是循环。

如果你有别的函数式编程的经验例如 Scheme 或 ML，那么肯定有“模仿循环的方法就是通过尾递归”这个概念。

再来看一下 `globToRegex'` 函数，并不是一个尾递归函数。可以测试一下最后的子句（其它的子句结构都类似）。

```
globToRegex' (c : cs) = escape c ++ globToRegex' cs
```

它递归的应用了自身，且递归的结果作为一个参数在 `(++)` 函数上。由于递归的应用并非作为函数的最后处理部分，那么 `globToRegex'` 并不是一个尾递归。

那么为什么这个函数并非尾递归呢? 答案就在 Haskell 的非严格计算策略。在提及这个话题之前，让我们快速的了解为什么在传统语言中，我们需要避免这样的尾递归定义。以下是 `(++)` 操作符的一个简单定义。它属于递归，但不是尾递归。

```
(++) :: [a] -> [a] -> [a]

(x:xs) ++ ys = x : (xs ++ ys)
[]     ++ ys = ys
```

在严格执行语言中，如果我们计算 `"foo" ++ "bar"` ，整个列表会被构建出来，接着再返回。非严格计算则会延迟直到真正需要被用到的时候。

如果需要表达式 `"foo" + "bar"` 的一个元素时，函数定义的第一个模式将被匹配，接着返回表达式 `x : (xs ++ ys)` 。因为 `(:)` 构造函数是非严格的，那么 `xs ++ ys` 的计算就能被递延：按需生产更多的元素。当我们生产更多的结果时，我们将不再使用 `x` ，因此垃圾回收器将会进行回收。由于是按需生产元素，且不持有已经完成的部分，编译器就可以以常量的空间来计算我们的代码。

使用我们的模式匹配器

通过一个函数用于匹配 glob 模式是很好的，但是我们希望将其用于实际应用。在 Unix 系的系统中 `glob` 函数返回所有匹配到的文件名称以及路径。让我们在 Haskell 中也构建一个类似的函数：

```
module Glob (namesMatching) where
```

`System.Directory` 模块提供了用于处理路径以及内容的标准函数。

```
import System.Directory (doesDirectoryExist, doesFileExist, getCurrentDirectory,
  getDirectoryContents)
```

而 `System.FilePath` 模块抽象了操作系统路径名称转换的细节， `(</>)` 函数将两个路径连接在一起：

```
ghci> :m +System.FilePath
ghci> "foo" </> "bar"
"foo/bar"
```

`dropTrailingPathSeparator` 函数正如其名，去除末尾的路径分隔：

```
ghci> dropTrailingPathSeparator "foo/"
"foo"
```

`splitFileName` 则分离了路径与文件：

```
ghci> splitFileName "foo/bar/Quux.hs"
("foo/bar/","Quux.hs")
```

`System.Directory` 模块结合 `System.FilePath` 的使用，我们可以编写一个同时作用于 Unix 系与 Windows 的 `namesMatching` 函数：

```
import System.FilePath (dropTrailingPathSeparator, splitFileName, (</>))
```

在这个模块中，我们将模仿一个“for”循环；首次在 Haskell 中尝试错误处理；并使用写好的 `matchesGlob` 函数。

```
import Control.Exception (handle)
import Control.Monad (forM)
import GlobRegex (matchesGlob)
```

由于路径与文件存在于拥有副作用的“真实世界”，我们的 globbing 函数的返回值则必须带上 `IO` 。

如果传入的字符串没有包含模式字符，那么就简单的检查给定的名称是否存在于文件系统重。（注意，我们在这里使用 Haskell 的函数守护语法来编写一个漂亮整洁的定义。使用“if”也是可以的，不过就没有这么美观了）

```
isPattern :: String -> Bool
isPattern = any (`elem` "[*?")

namesMatching pat
  | not (isPattern pat) = do
      exists <- doesNameExist pat
      return ([pat | exists])
```

函数 `doesNameExist` 会在稍后进行定义。

那么如果字符串是一个 glob 模式呢？继续我们的函数定义：

```
| otherwise = do
    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)
```

我们使用了 `splitFileName` 将字符串分离成了一对“所有东西除了最终名称”以及“最终名称”。如果首个元素为空，则在当前路径寻找一个模式；否则检查路径名称是否包含模式，没有包含则创建一个包含路径名称的单例列表，包含则列举所有匹配的路径。

Tip

需要注意的事情

`System.FilePath` 模块可以变得有点棘手。上述的例子中 `splitFileName` 函数留下了末尾的斜杠

```
ghci> :module +System.FilePath
ghci> splitFileName "foo/bar"
("foo/","bar")

```

如果我们忘了（或者不了解）要移除斜杠，我们则会无限在 `namesMatching` 中递归，因为下列行为

```
ghci> splitFileName "foo/"
("foo/","")

```

你可以猜想一下之后将会发生什么！

最后，收集了所有匹配的路径，得到了一个列表的列表，再将它们打平成一个单独的列表。

这里陌生的 `forM` 函数其行为类似于一个“for”循环：它映射它的第二个参数（一个 action）在第一个参数（一个列表）上，并返回一个列表的结果。

还剩下一些需要处理的问题。首先是 `doesNameExist` 函数。`System.Directory` 模块并不能检查一个名称是否存在与文件系统中，它强制要求我们决定检查的是文件还是路径。这个 API 很丑陋，因此我们需要将两者合二为一。首先检查是否为文件，因为文件相较于路径更为普遍。

```
doesNameExist :: FilePath -> IO Bool
doesNameExist name = do
  fileExists <- doesFileExist name
  if fileExists
    then return True
    else doesDirectoryExist name
```

还有另外两个函数需要定义，它们的返回都是一个路径中的所有名称。`listMatches` 函数返回的是一个路径中匹配了给定 glob 模式的所有文件。

```
listMatches :: FilePath -> String -> IO [String]
listMatches dirName pat = do
  dirName' <-
    if null dirName
      then getCurrentDirectory
      else return dirName
  handle errorHandler $ do
    names <- getDirectoryContents dirName'
```

```
    let names' =
          if isHidden pat
            then filter isHidden names
            else filter (not . isHidden) names
    return (filter (`matchesGlob` pat) names')
  where
    errorHandler :: SomeException -> IO [String]
    errorHandler _ = return []

isHidden :: [Char] -> Bool
isHidden ('.' : _) = True
isHidden _ = False
```

与原文不同之处在于 `handle errorHandler` 这里需要显式的为 `errorHandler` 标注类型，否则原文的 `handle (const (return []))` 将会在编译时抛出类型不明确的异常，详见StackOverflow。

`listPlain` 函数的返回要么是一个空的列表要么是一个单例列表，这取决于传递的名称是否存在：

```
listPlain :: FilePath -> String -> IO [String]
listPlain dirName baseName = do
  exists <-
    if null baseName
      then doesDirectoryExist dirName
      else doesNameExist (dirName </> baseName)
  return ([baseName | exists])
```

通过 API 设计来处理错误

如果传递给我们的 `globToRegex` 的是一个错误的模式，并不会是一场灾难。这有可能使用户输入了错误的模式，这种情况下我们需要报告有意义的错误信息。

```
type GlobError = String

globToRegex :: String -> Either GlobError String
```

运行我们的代码

`namesMatching` 函数并不能很好地单独工作，我们需要将其与其它函数结合。

首先定义一个 `renameWith` 函数，并非只是简单的为一个文件重命名，而是将一个函数应用至文件名称，用函数返回的值来重命名文件。

```
import System.FilePath (replaceExtensions)
import System.Directory (doesFileExist, renameDirectory, renameFile)
import Glob (namesMatching)
```

```
4
5  renameWith :: (FilePath -> FilePath) -> FilePath -> IO FilePath
6  renameWith f path = do
7    let path' = f path
8    rename path path'
9    return path'
```

这里还是需要一个帮助函数来处理文件与路径：

```
1  rename :: FilePath -> FilePath -> IO ()
2  rename old new = do
3    isFile <- doesFileExist old
4    let f = if isFile then renameFile else renameDirectory
5    f old new
```

`System.FilePath` 模块提供了很多操作文件名的函数，它们可以很好地与 `renameWith` 与 `namesMatching` 函数结合，使得我们可以快速的使用它们来创造复杂行为的函数。例如修改 C++ 源文件的后缀。

```
1  cc2cpp :: IO [FilePath]
2  cc2cpp = mapM (renameWith (flip replaceExtension ".cpp")) =<< namesMatching "*.cc"
```

`flip` 函数接受另一个函数作为参数，交换该函数的入参。`=<<` 函数将其右侧 action 的结果喂给左侧的 action。

9 I/O case study: a library for searching the filesystem

略

简单开始：递归展示一个路径

在设计我们的库之前，让我们先解决几个小问题。我们的第一个问题就是递归的列出一个路径的所有文件与子路径。

```haskell
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)
```

`filter` 表达式确保一个路径列表中不会包含特殊的路径名称，例如 `.` 或者 `..` 。如果没有过滤它们则会有无限的递归。

这里又遇到了上一章的 `forM` ；它等同于 `mapM` 只不过参数调转了：

```
ghci> :m +Control.Monad
ghci> :t mapM
mapM :: (Traversable t, Monad m) => (a -> m b) -> t a -> m (t b)
ghci> :t forM
forM :: (Traversable t, Monad m) => t a -> (a -> m b) -> m (t b)
```

与原文（如下）不同：

```
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]
```

函数体中会检查当前输入是否为一个路径：如果是，则递归调用 `getRecursiveContents` 来列举该路径；否则，它返回一个单例列表，即当前输入的名称。（别忘了 `return` 函数在 Haskell 有独特的意义：它通过单子类型构造函数来包装一个值。）

另一个值得指出的点在于变量 `isDirectory` 的使用。一个命令式的语言例如 Python，通常会表述为 `if os.path.isdir(path)` ，然而 `doesDirectoryExist` 函数是一个 *action*；它的返回类型是 `IO Bool` 而不是 `Bool` 。又因为一个 `if` 表达式需要的是 `Bool` 类型，我们需要使用 `<-` 来获取被 `IO` 所包裹着的 action 的 `Bool` 返回，这样才能在 `if` 中使用解包过后的 `Bool` 。

每个 loop 的便利都会返回一个名称列表，因此这里 `forM` 的结果是 `IO [[FilePath]]` 。我们使用 `concat` 将其打平称为一个列表。

重访匿名与命名函数

在之前的匿名（lambda）函数的章节中，我们列举了一些不要使用匿名函数的原因，然而在这里我们使用了一个匿名函数作为 loop 的本体。这就是 Haskell 中最常见的匿名函数使用方法。

我们已经从 `forM` 与 `mapM` 的类型中得知它们接受函数作为参数。大多数的 loop 本体会是一个在程序中出现一次的代码块。因为我们只会在一处使用这个 loop 本体，那么为什么要给它一个名称呢?

当然了，有时我们需要在若干地方部署同样的代码。这种情况下与其复制黏贴一个匿名函数，不如给已存在的函数一个名称。

为什么同时提供了 `mapM` 与 `forM` ?

同时存在两个除了参数顺序不同功能却一样的函数看起来有点奇怪，实际上它们适用于不同的场景。

考虑上述例子，使用匿名函数作为函数体。如果我们使用 `mapM` 而不是 `forM` ，那么则需要将变量 `properNames` 放置在函数体之后。为了让代码能被正确的解析，我们还需要将整个匿名函数体用圆括号包裹起来，或者是用一个命名函数来代替。

相反，如果 loop 本体就是一个命名函数，且需要遍历的列表是由一个复杂的表达式计算而得的时候，那么就有更好的理由使用 `mapM` 了。

一个简单的查询函数

我们可以是用 `getRecursiveContents` 函数作为一个简单的文件查询的基础:

```
1 import RecursiveContents (getRecursiveContents)
2
3 simpleFind :: (FilePath -> Bool) -> FilePath -> IO [FilePath]
4 simpleFind p path = do
5   names <- getRecursiveContents path
6   return $ filter p names
```

这个函数接受一个谓词，用于筛选 `getRecursiveContents` 所返回的名称。每个传递到谓词的名称都是一个完整的路径，那么我们该如何执行一个类似“找到所有后缀为 `.c` 的文件”这样的常规操作呢?

`System.FilePath` 模块包含了大量的宝贵的函数可以帮助我们操作文件名。上述案例可以用 `takeExtension` 。

```
ghci> :m +System.FilePath
ghci> :t takeExtension
takeExtension :: FilePath -> String
ghci> takeExtension "foo/bar.c"
".c"
ghci> takeExtension "quux"
""
```

这样我们就可以编写一个接受路径并提取后缀再与 `.c` 进行比较的函数了:

```
ghci> :l SimpleFinder
[1 of 2] Compiling RecursiveContents ( RecursiveContents.hs, interpreted )
[2 of 2] Compiling Main              ( SimpleFinder.hs, interpreted )
Ok, two modules loaded.
ghci> :t simpleFind (\p -> takeExtension p == ".c")
simpleFind (\p -> takeExtension p == ".c")
  :: FilePath -> IO [FilePath]
```

虽然 `simpleFind` 可以工作，不过还有一些瑕疵。首先是谓词并没有很强的表达力。它只能查看一个路径名称，而不能知晓输入的是一个文件还是路径。这就意味着尝试使用 `simpleFind` 将列出 `.c` 结尾的目录已经具有相同扩展名的文件。

其次就是 `simpleFind` 没有提供该如何遍历文件系统的控制。为什么这点很重要，可以考虑一下在拥有子版本管理的树形文件系统下，进行源文件的搜索。子版本在每个它管理的路径下都会维护一个私有的 `.svn` 路径；它们每一个包含了若干我们不需要考虑的子文件夹与文件。虽然我们可以简单的过滤掉任何包含 `.svn` 的情况，不过更有效的方式则是在第一时间就避免这些文件夹的遍历。

最后就是 `simpleFind` 是严格执行的，因为它包含了一系列在 `IO` 单子中所执行的 actions。如果我们有上百万的文件需要遍历，我们则需要等待很久并且还会得到一个巨大的包含了百万名称的结果。这对于资源管理与响应速度而言都非常的不好。我们更加倾向于一个惰性流式的结果。

谓词：从贫穷到富裕，同时保持 pure

我们的谓词只能查看文件名称。这就包含了大量有趣的行为：例如当我们想要展示的文件大于给定的大小时。

一个比较简单的做法就是修改 `IO` ：谓词的类型不再是 `FilePath -> Bool` ，而改为 `FilePath -> IO Bool` 。这使得我们可以用任意 I/O 作为谓词函数的一部分。尽管这看起来

很吸引人，但它也有潜在的问题：这样的谓词可能会有任意的副作用，因为返回类型为 I/O a 的函数可以拥有任意的副作用。

让我们利用类型系统来编写更加可以预测，更是 bug 的代码：我们将通过避免“IO”污染来保持谓词的纯粹性。这样可以确保它们不会产生任何讨厌的副作用。同时我们还会喂给它们更多的信息，这样它们就能获得我们想要的表达能力，而不会成为潜在的危险。

Haskell 的 `System.Directory` 模块提供了一组有用的文件元数据。

```
ghci> :m +System.Director
```

- 我们可以使用 `doesFileExist` 与 `doesDirectoryExist` 来判断一个输入是否是文件还是路径。近些年暂时还没有便捷的方法来查看其它的文件类型，例如命名通道，软硬链接等。

```
        ghci> :t doesFileExist
        doesFileExist :: FilePath -> IO Bool
        ghci> doesFileExist "."
        False
        ghci> :t doesDirectoryExist
        doesDirectoryExist :: FilePath -> IO Bool
        ghci> doesDirectoryExist "."
        True

```

- `getPermissions` 函数允许我们得知对于一个文件或者路径的操作是否合法

```
        ghci> :t getPermissions
        getPermissions :: FilePath -> IO Permissions
        ghci> :i Permissions
        type Permissions :: *
        data Permissions
          = directory-1.3.6.2:System.Directory.Internal.Common.Permissions {readable ::
    Bool,
                                                                           writable ::
    Bool,
                                                                           executable
    :: Bool,
                                                                           searchable
    :: Bool}
                -- Defined in ‘directory-1.3.6.2:System.Directory.Internal.Common’
        instance [safe] Eq Permissions
          -- Defined in ‘directory-1.3.6.2:System.Directory.Internal.Common’
        instance [safe] Ord Permissions
          -- Defined in ‘directory-1.3.6.2:System.Directory.Internal.Common’
        instance [safe] Show Permissions
          -- Defined in ‘directory-1.3.6.2:System.Directory.Internal.Common’
        instance [safe] Read Permissions
```

```
          -- Defined in ‘directory-1.3.6.2:System.Directory.Internal.Common’
ghci> getPermissions "."
Permissions {readable = True, writable = True, executable = False, searchable =
True}
ghci> :t searchable
searchable :: Permissions -> Bool
ghci> searchable it

```

- 最后则是 `getModificationTime` 可以告诉我们输入是何时被修改的：

```
ghci> :t getModificationTime
getModificationTime
  :: FilePath
     -> IO time-1.11.1.1:Data.Time.Clock.Internal.UTCTime.UTCTime
ghci> getModificationTime "."
2023-10-14 13:58:10.895659105 UTC

```

那么对于新的谓词而言有多少数据是需要知道的呢？由于可以通过 `Permissions` 来查看一个输入是否为文件还是路径，因此我们不再需要 `doesFileExist` 与 `doesDirectoryExist` 。这样我们有了四个需要查看的数据：

```
import Control.Exception (bracket, handle)
import Control.Monad (filterM)
import Data.Time.Clock (UTCTime)
import RecursiveContents (getRecursiveContents)
import System.Directory (Permissions (..), getModificationTime, getPermissions)
import System.FilePath (takeExtensions)
import System.IO (IOMode (..), hClose, hFileSize, openFile)

type Predicate =
  FilePath ->
  Permissions ->
  Maybe Integer ->
  UTCTime ->
  Bool
```

与原文不同之处在于导入部分，由于 `import System.Time (ClockTime(..))` 已经废弃了，根据 `getModificationTime` 可知需要使用 `import Data.Time.Clock (UTCTime)` 。

我们的 `Predicate` 类型就是一个有着四个参数函数的别名。

注意谓词的返回类型是 `Bool` 而不是 `IO Bool` ：谓词是纯的，并不运行 I/O。有了这个类型，编写一个更具表达性的查询函数就好办了。

```
getFileSize :: FilePath -> IO (Maybe Integer)
getFileSize = undefined

```

```haskell
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
```

我们暂时先跳过 `getFileSize` 函数，稍后会进行讲解。

我们不能使用 `filter` 来调用谓词 `p` ，因为 `p` 的纯性意味着它不能执行收集所需元数据所需的 I/O。

这时就需要一个我们并不熟悉的 `filterM` 函数了，它的行为类似于普通的 `filter` 函数，不过是在 `IO` 单子中对谓词进行计算，即允许谓词执行 I/O。

```haskell
ghci> :m +Control.Monad
ghci> :t filterM
filterM :: Applicative m => (a -> m Bool) -> [a] -> m [a]
```

安全的调整文件大小

虽然 `System.Directory` 没有提供文件大小的查询，不过我们可以用 `System.IO` 来做。它包含了一个名为 `hFileSize` 的函数，可以返回一个文件的字节大小。

```haskell
simpleFileSize :: FilePath -> IO Integer
simpleFileSize path = do
  h <- openFile path ReadMode
  size <- hFileSize h
  hClose h
  return size
```

虽然这个函数可以工作，但是它还不能为我们所用。在 `betterFind` 中，我们可以无条件的对任何地址输入调用 `getFileSize` ；如果输入的不是一个文件它会返回 `Nothing` ，否则返回 `Just` 。实际上该函数的输入不是一个文件或者不能打开文件（权限不够）时会抛出异常。

以下是一个安全版本：

```haskell
saferFileSize :: FilePath -> IO (Maybe Integer)
saferFileSize path = handle errorHandler $ do
  h <- openFile path ReadMode
  size <- hFileSize h
  hClose h
  return $ Just size
  where
    errorHandler :: SomeException -> IO (Maybe Integer)
    errorHandler _ = return Nothing
```

这里与原文不同在于匿名函数 `handle (_ -> return Nothing)` 改成了 `errorHandler`，并标注了类型 `errorHandler :: SomeException -> IO (Maybe Integer)` 。（与第八章的处理方式一致）

函数体内部几乎是一致的，除了 `handle` 子句。

这里的异常处理是只要有异常发生就返回 `Nothing` ，其余的改动就是返回值被包裹了 `Just` 。

`saferFileSize` 函数现在拥有了正确的类型签名，也不会再抛出任何异常了，不过它仍然不完整。这里只处理了 `openFile` 的异常，但是 `hFileSize` 仍然会抛出异常，比如说命名管道。这样的异常会被 `handle` 捕获，但是我们的 `hClose` 永远不会被执行。

Haskell 的实现会在文件句柄不再使用时自动关闭，但是这要在垃圾回收时才会发生，而接下来垃圾回收的时间却不能保证。

文件句柄是稀有资源，这份稀有性是由操作系统决定的，比如 Linux 系统中一个进程默认同时打开文件的最大数量是 1024 个。

这不难想象一个场景，在调用了使用了 `saferFileSize` 的 `betterFind` 函数因为穷尽了最大文件开启数，而垃圾回收还没开始时，程序崩溃了。

这是一种特别危险的 bug：它会在若干条件组合在一起时变得难以追踪。在 `betterFind` 访问了足够多数量的非文件时句柄没有被关闭而达到了进程最大文件开启数的上限，然后在垃圾回收还未发生之前又打开了另外的文件。

更糟糕的是，任何后续错误都是由程序中无法访问数据而引起，并且还没有垃圾回收。出现这样的 bug 依赖于程序结构，文件系统内容，以及当前程序的运行时还未触发垃圾回收。

这样的问题在开发的过程中很容易被忽视，然后后续发生时会让诊断变得异常困难。

获取-使用-释放

我们需要 `openFile` 成功时 `hClose` 能总是被调用。`Control.Exception` 模块为此提供了 `bracket` 函数。

```
ghci> :m +Control.Exception
ghci> :type bracket
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
```

`bracket` 函数接受三个 actions 参数。第一个 action 用于获取一个资源，第二个则是释放资源，第三个则是介于前面两者之间，让我们称其为“使用”action。如果“获取”action 成功了，“释放”action *总是*会被调用。这确保了资源总是能被释放。“使用”和“释放”actions 分别传给了“获取”action。

当“使用”action 在执行时发生了异常，`bracket` 会调用“释放”action 然后再重新抛出异常。如果“使用”action 成功了，`bracket` 会调用“释放”action 然后再返回”使用“action 所返回的值。

我们现在可以编写一个完全安全的函数了：它不会抛出异常；也不会产生句柄垃圾而导致程序崩溃。

```
getFileSize :: FilePath -> IO (Maybe Integer)
getFileSize path = handle errorHandler $
  bracket (openFile path ReadMode) hClose $ \h -> do
    size <- hFileSize h
    return (Just size)
  where
    errorHandler :: SomeException -> IO (Maybe Integer)
    errorHandler _ = return Nothing
```

让我们看一下这里的 `bracket` ，首先是打开文件，返回打开文件的句柄，接着是关闭句柄，最后是对句柄调用 `hFileSize` 并将结果包装进 `Just` 。

我们需要同时使用 `bracket` 与 `handle` 才能达成目的。前者能保证垃圾文件句柄不会堆积，后者确保异常被处理。

用于谓词的领域特定语言

让我们尝试着写一个谓词，我们的谓词将会检查一个 C++ 源文件是否超过了 128KB。

```
myTest path _ (Just size) _ = takeExtension path == ".cpp" && size > 131072
myTest _ _ _ _ = False
```

这看起来并不怎么样。谓词有四个参数，总是忽略其中两个，定义另外两个。我们当然可以做的更好，让我们编写一些让谓词变得更简洁的代码。

有时这类库被认为是*内嵌领域特定语言*：我们使用编程语言的工具（因此是*内嵌的*）来编写代码用以优雅的解决特类问题（因此是*领域特定的*）。

第一步首先是编写一个返回某个参数的函数。下面是从传入至 `Predicate` 的参数中提取 path：

```
pathP path _ _ _ = path
```

如果不提供一个类型签名，Haskell 则会为该函数推导出一个非常通用的类型。这在未来可能会带来错误而难以解析，因此我们给 `pathP` 一个类型。

```
type InfoP a =
  FilePath -> -- path to directory entry
  Permissions -> -- permissions
  Maybe Integer -> -- file size (Nothing if not file)
  UTCTime -> -- last modified
  a

pathP :: InfoP FilePath
pathP path _ _ _ = path
```

我们创建了一个类型同义词使得可以将其用于其它类似结构函数。该类型同义词接受一个类型参数，这样我们就可以指定不同的结果类型。

```
sizeP :: InfoP Integer
sizeP _ _ (Just size) _ = size
sizeP _ _ Nothing _ = -1
```

实际上，本章较早之前所定义的 `Predicate` 类型与 `InfoP Bool` 是一致的。（因此我们可以去除 `Predicate` 类型。）

那么 `pathP` 与 `sizeP` 怎么用呢？通过一点点粘合，可以将它们放入谓词中使用（`P` 后缀即意味着“predicate”）。

```
equalP :: (Eq a) => InfoP a -> a -> InfoP Bool
-- equalP f k = \w x y z -> f w x y z == k
equalP f k w x y z = f w x y z == k
```

`equalP` 的类型签名值得注意。它接受一个 `InfoP a` ，同时兼容 `pathP` 与 `sizeP` 。它接受一个 `a` ，返回一个 `InfoP Bool` ，即 `Predicate` 的同义词。换言之，`equalP` 构建一个谓语。

`equalP` 函数通过返回一个匿名函数来工作。该匿名函数接受一个谓词作为参数，将其传递给 `f` ，接着将其返回与 `k` 作比较。

由于 Haskell 的所有函数都是柯里化的，这么编写 `equalP` 实际上并不必要。我们可以省略匿名函数接着依赖柯里化来完成剩下的部分，现在让我们来编写一个相同效果的函数。

```
equalP' :: (Eq a) => InfoP a -> a -> InfoP Bool
equalP' f k w x y z = f w x y z == k
```

在继续探索之前，让我们将我们的模块加载至 **ghci**。

```
ghci> :l BetterPredicate.hs
[1 of 2] Compiling RecursiveContents ( RecursiveContents.hs, interpreted )
[2 of 2] Compiling Main              ( BetterPredicate.hs, interpreted )
Ok, two modules loaded.
```

让我们看看由这些函数所构建的谓词是否生效。

```
ghci> :t betterFind (sizeP `equalP` 1024)
betterFind (sizeP `equalP` 1024) :: FilePath -> IO [FilePath]
```

注意我们并没有真正的调用 `betterFind` ，不过却能确认表达式的类型。我们现在拥有了一个更具表现力的方式来展示所有相同大小的文件。

通过 lifting 避免重复

除开 `equalP` ，我们还可以编写其它的二元函数。我们倾向于不为每个定义都编写一个完整的定义，因为这看起来冗余而没有必要。

为此让我们使用 Haskell 的抽象能力。我们将 `equalP` 的定义而不是直接调用 `(==)` ，将其作为参数传递给希望调用的二元函数中。

```
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 (<)
```

这里接受如 `(>)` 这样的函数，转换它为另一个作用于不同上下文的函数，如 `greaterP` ，这意味着将其*提升 lifting* 至该上下文。这解释了函数名中带有 `lift` 。Lifting 让我们复用代码并且减少重复的模式。在本书的后续章节中，我们将大量的使用它。

我们在提升一个函数时，通常会分别将其原始版本以及新版本称为 *unlifted* 与 *lifted*。

通过这种方式，我们的 `q` （需要提升的函数）作为 `liftP` 的第一个参数是经过深思熟虑的。这使得可以编写如 `greaterP` 以及 `lesserP` 这样简洁的定义。相较于其它的语言，偏应用使得寻找“最佳”的参数顺序成为了 Haskell 中 API 设计中更为重要的环节。在没有偏应用的语言中，参数的顺序在于使用习惯与方便性。而在 Haskell 中将参数设计在错误的位置，则会丢失偏应用所带来的简洁性。

我们可以通过组合子来复现一下这样的简洁性。例如，`forM` 在 2007 年之前并没有被加入至 `Control.Monad` 模块中，更早之前人们则会使用 `flip mapM` 来替代。

```
ghci> :m +Control.Monad
ghci> :t mapM
mapM :: (Traversable t, Monad m) => (a -> m b) -> t a -> m (t b)
ghci> :t forM
forM :: (Traversable t, Monad m) => t a -> (a -> m b) -> m (t b)
ghci> :t flip mapM
flip mapM
  :: (Traversable t, Monad m) => t a -> (a -> m b) -> m (t b)
```

与原文不同之处在于上述函数变得更加泛化了，从之前的只对列表生效变为了对所有满足 `Traversable` typeclass 的类型生效。

将谓词们粘合在一起

如果我们希望组合谓词们，那么就可以遵循之前所做的那样：

```
simpleAndP :: InfoP Bool -> InfoP Bool -> InfoP Bool
simpleAndP f g w x y z = f w x y z && g w x y z
```

现在我们了解了 lifting，那么减少之前必须为布尔值操作符所编写的代码量就变得很自然了。

```
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

andP2 = liftP (&&)
orP2 = liftP (||)
```

注意 `liftP2` 跟之前的 `liftP` 很像。实际上前者更通用，因为我们可以根据 `liftP2` 来编写 `liftP` ：

```
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
```

Tip

组合子 Combinators

Haskell 中，我们将接受函数作为参数并返回新函数的函数成为组合子。

现在我们有了些帮助函数，可以回到较早之前定义的 `myTest` 函数了。

```
myTest :: (Ord a, Num a) => FilePath -> p1 -> Maybe a -> p2 -> Bool
myTest path _ (Just size) _ = takeExtension path == ".cpp" && size > 131072
myTest _ _ _ _ = False
```

那么该函数使用新的组合子会长什么样呢?

```
liftPath :: (FilePath -> a) -> InfoP a
liftPath f w _ _ _ = f w

myTest2 = (liftPath takeExtension `equalP` ".cpp") `andP` (sizeP `greaterP` 131072)
```

我们添加了一个最终的组合子，`liftPath` ，因为操作文件名是一个常用的操作。

定义与使用新的操作符

我们还可以通过定义新的中缀操作符来让我们的领域特定语言更进一步。

```
-- explicit annotation
(==?) :: InfoP String -> String -> InfoP Bool
(==?) = equalP

(&&?) :: InfoP Bool -> InfoP Bool -> InfoP Bool
(&&?) = andP

-- explicit annotation
(>?) :: InfoP Integer -> Integer -> InfoP Bool
(>?) = greaterP

myTest3 = (liftPath takeExtension ==? ".cpp") &&? (sizeP >? 131072)
```

注意与原文不同之处在于 `(==?)` 以及 `(>?)` 加上了显式类型签名，因为在没有 `myTest3` 时，编译器无法正确的进行类型推导而导致编译错误。

我们选择如 `(==?)` 这样的名称作为提升函数，这样可以在视觉上与原始函数对应。

上面定义中的圆括号是必要的，因为我们没有告诉 Haskell 新操作符的优先级或结合性。语言规定没有固定声明的操作符应被视为 `infixl 9` ，即在最高优先级从左到右求值。

我们可以通过为新操作符编写固定声明，第一步是找出未提升的操作符使得可以模拟它们。

```
ghci> :i ==
type Eq :: * -> Constraint
class Eq a where
  (==) :: a -> a -> Bool
  ...
        -- Defined in ‘GHC.Classes’
infix 4 ==
ghci> :i &&
(&&) :: Bool -> Bool -> Bool    -- Defined in ‘GHC.Classes’
infixr 3 &&
ghci> :i >
type Ord :: * -> Constraint
class Eq a => Ord a where
  ...
  (>) :: a -> a -> Bool
  ...
        -- Defined in ‘GHC.Classes’
infix 4 >
```

有了它们以后，我们可以编写一个不带圆括号的表达式，解析后与 `myTest3` 一致：

```
infix 4 ==?
infixr 3 &&?
infix 4 >?

myTest4 = liftPath takeExtension ==? ".cpp" &&? sizeP >? 131072
```

控制遍历

在遍历文件系统时，我们希望得到更多在方向上的控制。一个简单的方法便是允许传递给定路径的子路径列表至函数，并返回另一个列表。该列表允许元素被移除，或者可以以其他方式排序，或是两者皆可。最简单的控制函数是 `id` ，返回的值与输入值保持一致。

为了多样性，我们将改变一些展示的方式。相较于使用一个解释性函数类型 `InfoP a` ，我们将使用一个普通的代数数据类型来展示同样的信息。

```
import Data.Time (UTCTime)
import System.Directory (Permissions)

data Info = Info
```

```
    { infoPath :: FilePath,
      infoPerms :: Maybe Permissions,
      infoSize :: Maybe Integer,
      infoModTime :: Maybe UTCTime
    }
    deriving (Eq, Ord, Show)

getInfo :: FilePath -> IO info
```

我们使用 record 语义来“自由的”访问函数，例如 `infoPath` 。`traverse` 函数的类型很简单，正如之前推测的那样。为了获取文件或者路径的 `Info` ，我们可以调用 `getInfo` action。

```
traverse' :: ([Info] -> [Info]) -> FilePath -> IO [Info]
```

与原文不同之处在于 `traverse` 已经在 `Prelude` 中定义了，因此需要另一个名称。`traverse` 的定义如下：

```
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
```

首先是变量 `contents` 的赋值，从右至左来阅读代码。我们已经知道了 `names` 是一个路径列表，需要确保当前路径被前置放置在列表中的每个元素中，同时还要包含当前路径其本身。接着使用 `mapM` 应用 `getInfo` 函数在上述的返回值（路径列表）上。

再次从右往左阅读代码，我们可以看到该行的最后一个元素是一个匿名函数的定义。给定该匿名函数一个 `Info` 值，该函数要么递归的访问一个路径（有额外的检查可以确保我们不会再次访问 `path` ），要么返回一个单利列表（用于匹配 `traverse'` 的返回类型）。

用户提供的遍历控制函数 `order` 处理 `contents` 的结果为一个 `Info` 值列表，然后再被 `forM` 将上述匿名函数应用至这个列表的每个元素上。

最后是 `liftM` 函数，接受一个普通函数 `concat` ，提升该函数至 `IO` 单子。换言之，`liftM` 将 `forM` 的结果（类型为 `IO [[Info]]` ）从 `IO` 单子中提取出来，将 `concat` 应用在提取的结果上（返回值类型为 `[Info]` ，即我们所需要的），然后再将返回结果放入 `IO` 单子中。

最后，不要忘记定义我们的 `getInfo` 函数：

```
getInfo :: FilePath -> IO Info
getInfo path = do
  perms <- maybeIO (getPermissions path)
  size <- maybeIO (withFile path ReadMode hFileSize)
  modified <- maybeIO (getModificationTime path)
  return (Info path perms size modified)

maybeIO :: IO a -> IO (Maybe a)
maybeIO act = handle (\(SomeException _) -> return Nothing) $ Just `liftM` act
```

`size <- maybeIO (bracket (openFile path ReadMode) hClose hFileSize)` 可以被 `withFile` 简化成 `size <- maybeIO (withFile path ReadMode hFileSize)` ，其次就是 `maybeIO` 的匿名函数的入参显式声明类型 `(\(SomeException _) -> return Nothing)` 。

这里唯一需要注意的是一个有用的组合子 `maybeIO` ，其将一个可能会抛出异常的 `IO` action，以 `Maybe` 包装结果。

密集，可读性与学习过程

Haskell 中像 `traverse'` 这样密集的代码是不常见的。代码的表现力是很重要的，它需要相关的少量代码可以被流畅的阅读以及编写。

对比一下下面相同作用的密集代码，这更像是一个经验较少的 Haskell 程序员所写的：

```
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 = maybe False searchable (infoPerms info)
  recurse info = do
    if isDirectory info && infoPath info /= path
      then traverseVerbose order (infoPath info)
      else return [info]
```

与原文不同之处在于 `isDirectory` 的 `case ... of` 被 `maybe` 替换了。

编写可维护的 Haskell 代码的关键是在密集与可读性中寻找平衡。

另一个角度看待遍历

尽管 `traverse'` 函数相较于 `betterFind` 函数，给与了我们更多的控制，但它还有一个重要的不足：我们可以在路径上进行递归，却不能过滤其它的名称，直到我们构建了整个列表的

名称树。如果遍历的路径包含了 100,000 个文件，而我们只关心其中三个，我们将分配 100,000 个元素的列表，然后才有机会修剪改列表保留三个所需的元素。

一种解决方案就是提供一个筛选函数作为 `traverse'` 的参数，这样可以将其应用在我们生成的名称列表上。这使得分配的列表只包含所需的元素。

然而这种方案同样有一个弱点：假设我们知道只需三个元素，而这三个元素在 100,000 个元素的前排，这种情况下我们无需再访问剩余的 99,997 个元素。这绝不是一个人为的例子：例如，邮箱的文件夹里的邮件，一个路径代表了邮箱所包含的千分之十的文件是非常常见的。

我们可以从另一个角度来解决上述两种方案的弱点：如果我们将文件系统的遍历视为一个路径层次的折叠 *fold* 呢?

我们熟知的 fold 有 `foldr` 以及 `foldl'` ，非常简洁的阐述了遍历一个列表并累积一个结果。延展折叠路径列表的概念并不难，不过我们所希望的是添加一个控制到折叠上。我们将这个控制表述为一个代数数据类型。

```
import ControlledVisit (Info)

data Iterate seed
  = Done {unwrap :: seed}
  | Skip {unwrap :: seed}
  | Continue {unwrap :: seed}
  deriving (Show)

type Iterator seed = seed -> Info -> Iterate seed
```

`Iterator` 类型给与我们所想要折叠的函数了一个便捷的别名。它接受一个 `seed` 以及一个用于表示路径的 `Info` 值，返回一个新的 `seed` 以及一个折叠函数的指示，该指示代表着 `Iterate` 类型的构造函数。

- 如果指示为 `Done` ，那么遍历应该立即停止。由 `Done` 所包装的值应该作为结果被返回。
- 如果指示为 `Skip` ，同时当前 `Info` 代表着一个路径，遍历将不会在该路径上递归。
- 其余情况下的遍历为 `Continue` ，使用包装的值作为 fold 函数的下一个调用。

我们的折叠逻辑上是一个左折叠，因为折叠第一个输入，同时每一步的 seed 是上一步所返回的结果。

```
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)
```

上述代码中有几个有趣的点。首先作用域的使用避免了额外的参数传递。顶层的 `foldTree` 函数仅是 `fold` 的一个包装，从 `fold` 的构造函数中提取最终结果。

由于 `fold` 是一个本地函数，我们无需传递 `foldTree` 的 `iter` 变量给它；它本身就可以访问外层的作用域。同样的，`walk` 也可以看到外层作用域的 `path` 。

另一个值得关注的点就是 `walk` 是一个尾递归循环，而非早前定义的函数中被 `forM` 所调用的匿名函数。这使得可以在需要的时候停止循环，也就是当迭代器返回 `Done` 时退出。

尽管 `fold` 调用 `walk` ，`walk` 调用 `fold` 递归的遍历子路径。每个函数返回一个被包装在 `Iterate` 的 seed：当 `fold` 被 `walk` 调用再返回，`walk` 检测其结果并决定是否继续还是退出（`Done` ）。这种方式下，调用者提供的迭代器返回 `Done` 时会立刻终止两个函数之间的递归调用。

那么一个迭代器在实际使用上是什么样的呢？下面是一个复杂的例子，用于查看至多三个 bitmap 图像，且不会递归到 Subversion 元数据目录中。

```
atMostThreePictures :: Iterator [FilePath]
atMostThreePictures paths info
  | length paths == 3 =
      Done paths
  | isDirectory info && takeFileName path == ".svn" =
      Skip paths
  | otherwise =
      Continue paths
  where
    extension = map toLower $ takeExtension path
    path = infoPath info
```

调用 `foldTree atMostThreePictures []` ，返回值类型为 `IO [FilePath]` 。

当然了，迭代器也不需要很复杂。下面是一个统计路径数的例子：

```
countDirectories count info =
Continue (if isDirectory info then count + 1 else count)
```

这两门传递给 `foldTree` 的初始 seed 需要是数字零。

有用的编码指导

略

10 Code case study: parsing a binary data format

本章我们将要探讨一个通用任务：解析一个二进制文件。使用该任务有两个目的，第一是借助解析来探讨程序的规划，重构，以及“模板化代码移除”。我们将展示如何简化重复的代码，并为我们在第 14 章 Monads 中做准备。

我们将使用的文件格式源自 netpbm 套件，这是一个古老而可敬的用于处理位图图像的程序和文件格式集合。这些文件格式具有广泛的时候以及容易解析的双重有点。更重要的是为了方便起见，netpbm 文件没有被压缩。

灰度文件

netpbm 的灰度文件格式为 PGM (“portable grey map”)。它有两种格式构；“plain”（或“P2”）格式是由 ASCII 编码的，而更常用的“raw”（“P5”）则是二进制格式。

两种格式的文件都有 header，即一个用于描述格式的“魔力”字符串。对于一个 plain 文件而言，字符串为 `P2` ；raw 文件则是 `P5` 。紧随字符串后的则是一个空格然后再是三个数字：图片的宽度，长度，以及最大灰度。这些数字皆为 ASCII 小数，由空格分隔。

接下来的就是图片数据。raw 文件中是二进制的字符串，plain 文件中则是由单空格字符分隔的 ASCII 小数构成。

raw 文件可以包含一系列的图片，一张接着一张，每张由 header 开头；plain 文件仅包含一张图片。

解析一个原始 PGM 文件

我们的第一个解析函数仅需考虑 raw PGM 文件，同时该解析函数是一个纯函数。该函数并不对数据获取负责，仅仅用作于解析。这在 Haskell 中是一个常规操作。通过分离数据读取，我们获得了更加灵活的操作空间。

我们将使用 `ByteString` 类型来存储 greymap 数据。由于 PGM 文件的头部是 ASCII 文本，而本体是二进制的，我们需要同时引用文本以及二进制的 `ByteString` 模块。

```
import qualified Data.ByteString.Lazy as L
import qualified Data.ByteString.Lazy.Char8 as L8
import Data.Char (isSpace)
```

我们将使用一个直接的数据类型来表示 PGM 图片。

```
data Greymap = Greymap
{ greyWidth :: Int,
  greyHeight :: Int,
  greyMax :: Int,
  greyData :: L.ByteString
}
```

```
  deriving (Eq)
```

通常而言，Haskell 的 `Show` 实例应该生成一个字符串作为展示，同时我们可以通过 `read` 将其从字符串中转回。然而对于 bitmap 图像文件而言，这会潜在的生产出巨大的文本字符串，例如对一个图片执行 `show` 。因此我们不会让编译器自动派生一个 `Show` 实例：我们编写自己的实现，并将其简化。

```
instance Show Greymap where
  show (Greymap w h m _) = "Greymap " ++ show w ++ "x" ++ show h ++ " " ++ show m
```

因为我们的 `Show` 实例是特意避开了打印 bitmap 数据，因此我们无法编写一个 `Read` 实例，毕竟我们不能通过 `show` 的结果来重新构建一个合法的 `Greymap` 。

接下来是我们解析函数的类型：

```
parseP5 :: L.ByteString -> Maybe (Greymap, L.ByteString)
```

该函数接受一个 `ByteString` ，如果解析成功则返回一个单独解析好的 `Greymap` ，以及剩下还需要解析的字符串。

我们的解析函数需要需要花费一点时间。首先是确保输入的文件是 raw PGM；接着解析文件头剩下的数值；然后再是解析 bitmap 数据。下面是一个浅显的表达方式：

```
matchHeader :: L.ByteString -> L.ByteString -> Maybe L.ByteString
matchHeader = undefined

getNat :: L.ByteString -> Maybe (Int, L.ByteString)
getNat = undefined

getBytes :: Int -> L.ByteString -> Maybe (L.ByteString, L.ByteString)
getBytes = undefined

-- parse function
parseP5 :: L.ByteString -> Maybe (Greymap, L.ByteString)
parseP5 s =
  case matchHeader (L8.pack "PS") 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)
```

这非常字面化的代码将所有的解析放在了一个长的阶梯型的 `case` 表达式上。每个函数在消费完其所需的字符串后，都会返回剩下的 `ByteString` 。接着解构每个结果，如果解析失败则返回 `Nothing` 。下面是解析过程中所用到的函数：

```
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
```

移除样板代码

虽然 `parseP5` 函数能工作，其形态并不令人满意。我们不停地重复 `Maybe` 值的构建与解构，且仅在匹配 `Just` 时继续。所有的这些相似的 `case` 表达式就像是“模板代码”那样。

我们可以看到有两个模式。首先是很多应用的函数都有相同的类型，接受 `ByteString` 作为其最后一个参数，并返回 `Maybe` 。其次 `parseP5` 函数中的每一步“阶梯”都会解构一个 `Maybe` 值，要么失败或是传递未解包的结果给一个函数。

我们可以轻易地用一个函数捕获第二个模式。

```
(>>?) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>? _ = Nothing
Just v >>? f = f v
```

`>>?` 函数的作用非常简单：它接受一个值作为其右参，以及一个函数作为其左参。如果值非 `Nothing` ，那么便将函数应用在 `Just` 值上。我们定义了作为一个操作符的函数，以此可

以链接所有函数在一起。最后我们并未提供给 `(>>?)` 一个优先级声明，那么默认为 `infixl 9`（左结合，最强的操作符优先级）。换言之，`a >>? b >>? c` 将会从左至右进行计算，类似于 `(a >>? b) >>? c` 。

有了这个链接函数，我们可以再尝试一下编写我们的解析函数。

```
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)
```

理解这个函数的关键在于链条的思考。每个 `(>>?)` 的左侧总是一个 `Maybe` 值；右侧则是一个返回 `Maybe` 值的函数。那么表达式的两侧类型都是 `Maybe` ，即满足下一个 `(>>?)` 表达式。

另一个增加可读性的工作就是添加了 `skipSpace` 函数。

隐式状态

我们的代码仍然在显式的传递一对一对的值，第一个元素作为解析过程中的结果，第二个元素则是当前剩余的 `ByteString` 。当我们想要扩展代码时，例如追踪消费了多少字节数量以便报告解析异常位置的时候，我们需要修改八个不同的地方，仅仅是为了传递一个三元组。

即使在少量代码的情况下，这样的需求都会使得代码难以修改。这个问题存在于使用模式匹配从每个元组中提取值时：我们默认了在代码中直接使用元组。

让我们讲解一下新代码为何是不灵活的。首先修改解析器使用的状态类型。

```
data ParseState = ParseState
{ string :: L.ByteString,
  offset :: Int64
}
deriving (Show)
```

在我们新的代数数据类型中，我们同时拥有了追踪剩余字符串以及当前偏移量的能力。最重要的变化还是使用了 record 语义：现在可以避免对状态的模式匹配了，而是使用访问函数 `string` 以及 `offset` 。

我们给与需要解析的状态一个名称。当我们为某物命名时，它可以变得更有意义。例如，我们可以将解析视为一种函数：其消费一个解析状态，并同时生产一个新的解析状态，以及一些额外的信息。我们可以直接将其视为一个 Haskell 类型。

```
simpleParse :: ParseState -> (a, ParseState)
simpleParse = undefined
```

为了更好的帮助用户，我们可以在解析失败时报告异常信息。这仅仅需要我们解析器一点小小的改动。

```
betterParse :: ParseState -> Either String (a, ParseState)
betterParse = undefined
```

之前显式的使用状态元组时，要扩展解析器的时候我们很快就发现问题了。为了避免重复，我们将使用 `newtype` 声明来隐去解析类型的细节。

```
newtype Parse a = Parse
{ runParse :: ParseState -> Either String (a, ParseState)
}
```

记住 `newtype` 定义就是一个编译时的包装函数，因此它没有任何的运行时负荷。当我们使用该函数时，我们将应用 `runParser` 访问函数。

如果我们不从模块中导出 `Parse` 值，我们可以确保他人不会意外的创建一个解析器，也避免了通过模式匹配来查看内部实现。

唯一性解析器

让我们尝试定义一个简单的解析器，*identity* 解析器。它的功能便是将任何传递进其的参数转换为解析器的结果。这种情况下，它更像是 `id` 函数。

```
identity :: a -> Parse a
identity a = Parse (\s -> Right (a, s))
```

该函数不会改变解析状态，并使用其参数作为解析的结果。我们将函数体包装成 `Parse` 类型来满足类型检查器。那么我们该如何使用该包装后的函数来进行解析呢?

首先我们通过 `runParse` 函数去除 `Parse` 的包装，以此获取其中的函数。接着构造一个 `ParseState` ，并以此运行我们的解析函数。最后则是将解析的结果从最终的 `ParseState` 中分离。

```
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
```

由于 `identity` 解析器与 `parse` 函数都不会测试状态，我们甚至不需要创建一个字符串输入来进行测试。

```
ghci> :l Parse.hs
[1 of 1] Compiling Main             ( Parse.hs, interpreted )
Ok, one module loaded.
ghci> :t parse (identity 1) undefined
parse (identity 1) undefined :: Num a => Either String a
ghci> parse (identity 1) undefined
Right 1
ghci> parse (identity "foo") undefined
Right "foo"
```

解析器甚至不会检查其不感兴趣的输入，之后我们将见识到其有用之处。

Record 语义，更新，以及模式匹配

Record 语义不仅仅只有访问函数有用：我们用它来拷贝以及修改现有值的一部分。使用中，概念如下：

```
modifyOffset :: ParseState -> Int64 -> ParseState
modifyOffset initState newOffset = initState {offset = newOffset}
```

这会创造一个新的与 `initState` 一样的 `ParseState` 值，不过其 `offset` 字段被设置成了指定的 `newOffset` 。

```
ghci> :l Parse.hs
[1 of 1] Compiling Main             ( Parse.hs, interpreted )
Ok, one module loaded.
ghci> let before = ParseState (L8.pack "foo") 0
ghci> let after = modifyOffset before 3
ghci> before
ParseState {string = "foo", offset = 0}
ghci> after
ParseState {string = "foo", offset = 3}
```

我们可以在花括号内部设置任意数量的字段，通过逗号进行分隔。

一个更有趣的解析器

现在让我们聚焦到编写一个更有意义的解析器上。我们暂时没有太大的野心：我们需要的是解析一个单字节。

```
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
```

定义中出现了若干新函数。

`L8.uncons` 函数从一个 `ByteString` 中接受首个元素。

```
ghci> L8.uncons (L8.pack "foo")
Just ('f',"oo")
ghci> L8.uncons L8.empty
Nothing
```

`getState` 函数获取当前的解析状态，而 `putState` 则是用于替换状态；`bail` 函数则是终结解析并报告异常；`(==>)` 函数将解析器链接起来。我们将稍后对每个函数进行解析。

Tip

Hanging lambdas

`parseByte` 的定义拥有一个视觉上的风格是我们之前没有讨论过的。它包含的匿名函数为参数以及 `->` 作为一行的结尾，而函数体起始于下一行。

这种方式的匿名函数并没有一个官方的名称，所以让我们称其“hanging lambda”。它主要的作用是为函数体留下更多的空间，同样可以将函数及其后续部分的关系在视觉上进行区分。例如通常而言，首个函数的结果会被当做参数传递给下一个函数。

获取与修改解析状态

`parseByte` 函数并不将解析状态作为一个参数，而是调用 `getState` 来获取一个状态的拷贝，`putState` 则是将当前状态替换。

```
getState :: Parse ParseState
getState = Parse (\s -> Right (s, s))

putState :: ParseState -> Parse ()
putState s = Parse (\_ -> Right ((), s))
```

当读取这些函数，记住元组的左元素是 `Parse` 的结果，而右元素则是当前解析状态。这使得接下来的函数变得更加方便使用。

`getState` 函数提取当前解析中的状态，使得调用者可以访问字符串。`putState` 函数替换当前解析中的状态。该状态通过 `==>` 链在下一个函数中可见。

这些函数让我们显式的移动状态在仅需要其所需的函数中。很多函数并不需要知道当前的状态，因此它们永远也不会调用 `getState` 或者 `putState` 。这相较于之前手动使用元组的解析器，这让我们可以编写出更简洁的代码，

我们将解析状态的细节打包进 `ParseState` 类型中，通过访问函数而不是模式匹配来进行工作。现在的解析状态是被隐式的传递，这为我们带来了极大的便利。如果我们希望添加更多的信息在解析状态中，我们仅需要修改 `ParseState` 的定义，以及任何需要这些新的信息的函数。相较于早期编写的解析器，所有的状态都被模式匹配所暴露，现在的代码则更加模块化：所有被影响的代码为需要新信息的代码。

报告解析异常

我们小心的定义 `Parse` 类型来兼容所有可能的异常。`==>` 组合子检查一个解析异常，并在出现异常时停止。然而我们仍未介绍 `bail` 函数，其用于报告解析异常。

```
bail :: String -> Parse a
bail err = Parse $ \s ->
  Left $
    "byte offset " ++ show (offset s) ++ ": " ++ err
```

在我们调用 `bail` 之后，`(==>)` 将会成功的在 `Left` 构造子上进行模式匹配包装错误信息，接着唤起链条中下一个解析函数。这将导致错误信息通过链条反向传递至上一级调用者。

链起所有解析器

`(==>)` 函数类似之前的 `(>>?)` 函数：它用于“粘合”使得所有函数链接在一起。

```
(==>) :: 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
```

`(==>)` 的函数体很有意思。回忆一下 `Parse` 类型是一个被包装装的函数。因为 `(==>)` 将两个 `Parse` 值链接生产出第三个值，它必须返回一个包装后的函数。

该函数不需要真正的“做”什么：它仅仅是创建一个闭包 *closure* 来记住 `firstParser` 与 `secondParser` 的值。

Tip

闭包其实就是一个带有环境的函数，其边界变量为其可见区域。闭包在 Haskell 中很常见。例如 `(+5)` 就是一个闭包。该闭包的实现必须将值 `5` 作为操作符 `(+)` 的第二个参数，因此无论函数传递的值是什么，其返回将会加五。

闭包不会被解包并执行，直到应用了 `parse` 函数。此时它将被应用至一个 `ParseState` 上，首先是应用 `firstParser` 并检验其结果。如果解析失败，闭包同样也会失败；否则它将传递解析的结果，以及新的 `ParseState` 至 `secondParser` 。

这真的是一个美妙的东西：我们高效的将 `ParseState` 隐式的传递至 `Parse` 链。（我们在稍后的章节中继续学习。）

函子简介

我们现在已经很熟悉 `map` 函数了，它将一个函数应用在一个列表中的每个元素上，返回很可能是另一种类型的列表。

```
ghci> map (+1) [1,2,3]
[2,3,4]
ghci> map show [1,2,3]
["1","2","3"]
ghci> :t map show
map show :: Show a => [a] -> [String]
```

`map` -类似的行为在其它实例中也很有用。例如，假设有一个二元树。

```
data Tree a
= Node (Tree a) (Tree a)
| Leaf a
deriving (Show)
```

如果我们想要将一个字符串树转换为一个包含了这些字符串长度的树，我们可以这样做：

```
treeLengths (Leaf s) = Leaf (length s)
treeLengths (Node l r) = Node (treeLengths l) (treeLengths r)
```

现在可以将目光转向更为通用的函数：

```
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)
```

正如我们希望的那样，`treeLengths` 与 `treeMap length` 可以得到同样的结果。

```
ghci> :l TreeMap.hs
[1 of 1] Compiling Main             ( TreeMap.hs, interpreted )
Ok, one module loaded.
```

```
ghci> let tree = Node (Leaf "foo") (Node (Leaf "x") (Leaf "quux"))
ghci> treeLengths tree
Node (Leaf 3) (Node (Leaf 1) (Leaf 4))
ghci> treeMap length tree
Node (Leaf 3) (Node (Leaf 1) (Leaf 4))
ghci> treeMap (odd . length) tree
Node (Leaf True) (Node (Leaf True) (Leaf False))
```

Haskell 提供了一个著名的 typeclass 来泛化 `treeMap` 。该 typeclass 就是 `Functor` ，其定义为一个函数， `fmap` 。

我们可以认为 `fmap` 类似于之前小节中所提到的*提升 lifting* 函数。它接受类型为 `a -> b` 的函数，将其提升至应用在容器上的 `f a -> f b` 函数，这里的 `f` 即容器类型。

如果我们将 `f` 替换为 `Tree` 类型，那么 `fmap` 的类型应与 `treeMap` 一致，那么实际上我们可以用 `treeMap` 作为 `Tree` 的 `fmap` 实现。

```
instance Functor Tree where
fmap = treeMap
```

`Functor` 的定义强制了一些明显的限制在 `fmap` 上。例如我们仅能实现 `Functor` 实例在只有一个类型参数的类型上。

例如我们不可以将 `fmap` 实现在 `Either a b` 或 `(a, b)` 上，因为它们有两个类型参数。同样的，我们不能实现在 `Bool` 或 `Int` 上，因为它们没有类型参数。

另外，我们不能在类型约束上添加任何约束。这是什么意思呢？为了更好的进行解释，首先看一下普通的 `data` 定义以及其 `Functor` 实例。

```
data Foo a = Foo a

instance Functor Foo where
  fmap f (Foo a) = Foo (f a)
```

当我们定义一个新的类型，我们可以添加一个类型约束在 `data` 关键字之后。

```
data Eq a => Bar a = Bar a

instance Functor Bar where
  fmap f (Bar a) = Bar (f a)
```

报错如下：

```
• No instance for (Eq a) arising from a use of ‘Bar’
  Possible fix:
    add (Eq a) to the context of
      the type signature for:
        fmap :: forall a b. (a -> b) -> Bar a -> Bar b
• In the pattern: Bar a
  In an equation for ‘fmap’: fmap f (Bar a) = Bar (f a)
  In the instance declaration for ‘Functor Bar’ typecheck(-Wdeferred-type-errors)
```

类型定义上的约束并不好

在一个类型定义上添加一个约束这种行为基本上永远不会是一个好主意。它造成的影响便是强迫用户在每个函数上也都添加类型约束。假设我们需要一个栈数据结构，使得我们可以查询其元素是否遵从某种排序。下面是该数据类型的定义：

```
data (Ord a) => OrdStack a
  = Bottom
  | Item a (OrdStack a)
  deriving (Show)
```

如果我们希望编写一个函数用于检查栈结构是否增加，我们显然需要一个 `Ord` 约束来执行一对元素的比较。

```
isIncreasing :: (Ord a) => OrdStack a -> Bool
isIncreasing (Item a rest@(Item b _))
  | a < b = isIncreasing rest
  | otherwise = False
isIncreasing _ = True
```

然而因为我们在类型定义上加上了类型约束，该约束实际上影响了完全不需要的地方：我们需要添加 `Ord` 约束至 `push` ，其本身并不关心堆上的顺序。

```
push :: (Ord a) => a -> OrdStack a -> OrdStack a
push a s = Item a s
```

这个时候尝试移除上述函数的 `Ord` 约束，`push` 则会在类型检查时失败。

这也是为什么较早之前我们尝试着为 `Bar` 编写一个 `Functor` 实例时失败了：它需要 `Eq` 约束在 `fmap` 上。

现在我们有了 Haskell 中在类型定义上添加类型约束是不好的认知，那么更合理的做法呢？答案就是简单的移除在类型定义上的类型约束，且放置在真正需要约束的函数上。

fmap 的中缀使用

很多时候我们会看到 `fmap` 作为操作符一样调用：

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

如果希望作为操作符那样使用 `fmap` ，那么 `Control.Applicative` 模块中有一个操作符 `(<$>)` ，它是 `fmap` 的别名。名称中 `$` 展现了一种相似性，即将一个函数应用至其参数（通过 `($)` 操作符）以及将一个函数提升至一个函子。

灵活的实例

我们可能希望能为 `Either Int b` 实现一个 `Functor` 实例，因为它仅有一个类型参数。

```
instance Functor (Either Int) where
  fmap _ (Left n) = Left n
  fmap f (Right r) = Right (f r)
```

然而 Haskell 98 的类型系统不能保证，这样的一个实例约束的检查能终止。非终止的约束检查可能会让编译器陷入无限循环，因此禁用了这样形式的实例。

```
• Illegal instance declaration for ‘Functor (Either Int)’
  (All instance types must be of the form (T a1 ... an)
  where a1 ... an are *distinct type variables*,
  and each type variable appears at most once in the instance head.
  Use FlexibleInstances if you want to disable this.)
• In the instance declaration for ‘Functor (Either Int)’ typecheck
```

原书使用一下代码解决编译出错的问题：

```
{-# LANGUAGE FlexibleInstances #-}

instance Functor (Either Int) where
  fmap _ (Left n) = Left n
  fmap f (Right r) = Right (f r)
```

然而实际上新的 Haskell 仍然不能通过编译，报错如下：

```
• Overlapping instances for Functor (Either Int)
    arising from a use of ‘GHC.Base.$dm<$’
  Matching instances:
    instance Functor (Either a) -- Defined in ‘Data.Either’
    instance Functor (Either Int)
      -- Defined at <...>/EitherIntFlexible.hs:7:10
• In the expression: GHC.Base.$dm<$ @(Either Int)
  In an equation for ‘<$’ : (<$) = GHC.Base.$dm<$ @(Either Int)
  In the instance declaration for ‘Functor (Either Int)’ typecheck(-Wdeferred-type-errors)
```

实际上，Haskell 的标准库早已更新，详见上一本书的笔记《Learn Me a Haskell》中第八章的“函子 typeclass”部分。Haskell 的标准实现如下：

```
instance Functor (Either a) where
  fmap f (Right x) = Right (f x)
  fmap f (Left x) = Left x
```

因为有了 Haskell 的标准实现，我们可以直接进行以下操作：

```
ghci> fmap (== "cheeseburger") (Left 1 :: Either Int String)
Left 1
ghci> fmap (== "cheeseburger") (Right "fries" :: Either Int String)
Right False
```

更多有关函子的思考

我们已经做了一些有关函子该如何工作的隐式假设。那么将它们显式的展示出来并思考它们的规则就更有帮助了，因为这让我们可以将函子视为一致的，良好行为的对象。我们仅需记住两个简单的规则。

第一条规则就是函子必须是保持恒等 *identity* 的，也就是说将 `fmap id` 应用至一值时，返回的总是相同的值。

```
ghci> fmap id (Node (Leaf "a") (Leaf "b"))
Node (Leaf "a") (Leaf "b")
```

第二条规则则是函子必须是可组合的 *composable*，也就是说将两个 `fmap` 组合起来使用与分别两次 `fmap` 使用的结果是一致的。

```
ghci> (fmap even . fmap length) (Just "twelve")
Just True
ghci> fmap (even . length) (Just "twelve")
Just True
```

另一种看待这两天规则的方式是，函子必须保持形状 *shape*。集合的结构不应该受到函子的影响；只有它所包含的值应该改变。

```
ghci> fmap odd (Just 1)
Just True
ghci> fmap odd Nothing
Nothing
```

如果你在编写一个 `Functor` 实例，那么牢记这些规则并且进行了测试是很有帮助的，因为编译器并不能上述列举的规则。另一方面，如果仅仅使用函子，那么规则是“自然而然的”无需记住它们。

为解析编写函子实例

对于已经调研过得这些类型，我们对 `fmap` 所预期的行为就显而易见了。相较于 `Parse` 因为复杂度的缘故会较难理解。一个合理的猜想就是我们所 `fmap` 的函数应该被应用在一个解析器的当前结果上，同时不对解析状态做任何修改。

```
instance Functor Parse where
fmap f parser =
  parser ==> \result ->
    identity (f result)
```

这个定义很容易阅读，让我们用几个快速的测试来检查我们是否遵循了函子的规则。

首先检查的是 identity 是否被遵循。首先是一个应该失败的解析：从一个空的字符串中解析一个字节（别忘了 `(<$>)` 就是 `fmap` ）。

```
ghci> :l Parse.hs
[1 of 2] Compiling Main             ( Parse.hs, interpreted )
Ok, one module loaded.
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"
```

现在测试一下成功的案例：

```
ghci> let input = L8.pack "foo"
ghci> L.head input
102
ghci> parse parseByte input
Right 102
ghci> parse (id <$> parseByte) input
Right 102
```

通过上述结果同样可知我们的函子实例遵循了第二条规则，即保持形状。失败保持失败，成功保持成功。

最后就是组合起来也是保存原有形状的。

```
ghci> :m +Data.Char
ghci> parse ((chr . fromIntegral) <$> parseByte) input
Right 'f'
ghci> parse (chr <$> fromIntegral <$> parseByte) input
Right 'f'
```

为解析使用函子

所有关于函子的讲解都是为了一个目的：它们可以让我们编写简洁有力的代码。回忆一下之前介绍过的 `parseByte` 函数。我们总是想要处理 ASCII 字符而不是 `Word8` 值。

我们可以模仿 `parseByte` 那样编写一个 `parseChar` 函数，不过现在可以通过 `Parse` 的函子特性来避免这些重复的代码。函子接受一个解析的结果并为该结果应用一个函数，因此我们需要一个函数可以将 `Word8` 转换成一个 `Char` 。

```
w2c :: Word8 -> Char
w2c = chr . fromIntegral

parseChar :: Parse Char
parseChar = w2c <$> parseByte
```

可以使用函子来编写一个简洁的“peek”函数。它将在输入的字符串结束时返回 `Nothing` ；否则返回下一个字符，同时不消耗它（即检查时不会影响当前的解析状态）。

```
peekByte :: Parse (Maybe Word8)
peekByte = fmap fst . L.uncons . string <$> getState
```

同样的方法可以为 `parseChar` 定义一个更加简洁的 `peekChar` 版本。

```
peekChar :: Parse (Maybe Char)
peekChar = fmap w2c <$> peekByte
```

注意 `peekByte` 与 `peekChar` 都调用了 `fmap`，即 `(<$>)`。这是因为 `Parse (Maybe a)` 类型是一个函子中的函子。因此我们需要将一个函数提升两次来“走进”里面的函子。

最后让我们编写另一个泛用的组合子，也就是类似于 `takeWhile` 版本的 `Parse`：消费输入直到谓词返回 `True`。

```
parseWhile :: (Word8 -> Bool) -> Parse [Word8]
parseWhile p =
  (fmap p <$> peekByte) ==> \mp ->
    if mp == Just True
      then
        parseByte ==> \b ->
          (b :) <$> parseWhile p
      else identity []
```

下面是一个不使用函子的啰嗦版本，不过更加直接：

```
parseWhileVerbose p =
peekByte ==> \mc ->
  case mc of
    Nothing -> identity []
    Just c
      | p c ->
          parseByte ==> \b ->
            parseWhileVerbose p ==> \bs ->
              identity (b : bs)
      | otherwise ->
          identity []
```

这个啰嗦的版本在不熟悉函子时虽然看似易读，但是在 Haskell 中使用函子更为常见，它的表达更加简洁，因此应该同时成为读与写的本能。

重写 PGM 解析器

没有新的解析代码时，原始的 PGM 解析函数会是什么样子呢？

```
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")
```

该定义又使用了以下的帮助函数：

```
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 ==> const f

skipSpaces :: Parse ()
skipSpaces = parseWhileWith w2c isSpace ==>& identity ()

assert :: Bool -> String -> Parse ()
assert True _ = identity ()
assert False err = bail err
```

`(==>&)` 组合子像 `(==>)` 那样将解析器链接起来，不过右侧的解析器会无视左侧所返回的结果。`assert` 函数允许我们检查一个属性，当该属性为 `False` 时会终止解析并返回一个有用的错误信息。

最后就是检查与修改解析状态。

```
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"
```

```
9             =>& identity h
```

未来的方向

略

11 Testing and quality assurance

12 Barcode recognition

WIP

13 Data structures

WIP

14 Monads

WIP

15 Programming with monads

WIP

16 The Parsec parsing library

WIP

17 The foreign function interface

WIP

18 Monad transformers

WIP

19 Error handling

WIP

20 Systems programming

WIP

21 Working with databases

WIP

22 Web client programming

WIP

23 GUI programming

WIP

24 Basic concurrent and parallel programming

WIP

25 Profiling and tuning for performance

WIP

26 Advanced library design: building a Bloom filter

WIP

27 Network programming

WIP

28 Software transactional memory

WIP