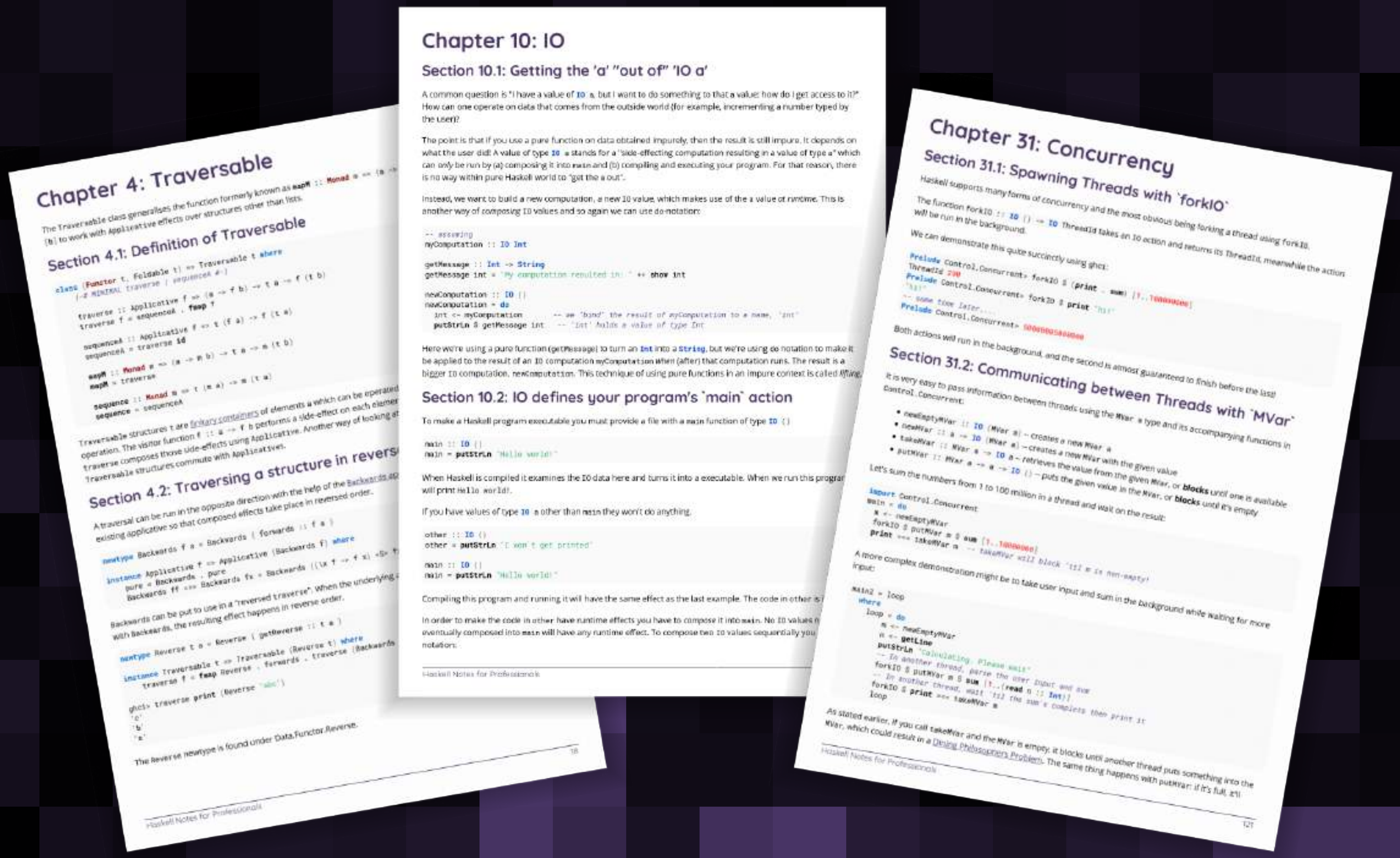
专业人员笔记

Haskell
专业人员笔记


Chapter 4: Traversable

Section 4.1: Definition of Traversable

Section 4.2: Traversing a structure in reverse

Chapter 10: IO

Section 10.1: Getting the 'a' "out of" 'IO a'

Section 10.2: IO defines your program's `main` action

Chapter 31: Concurrency

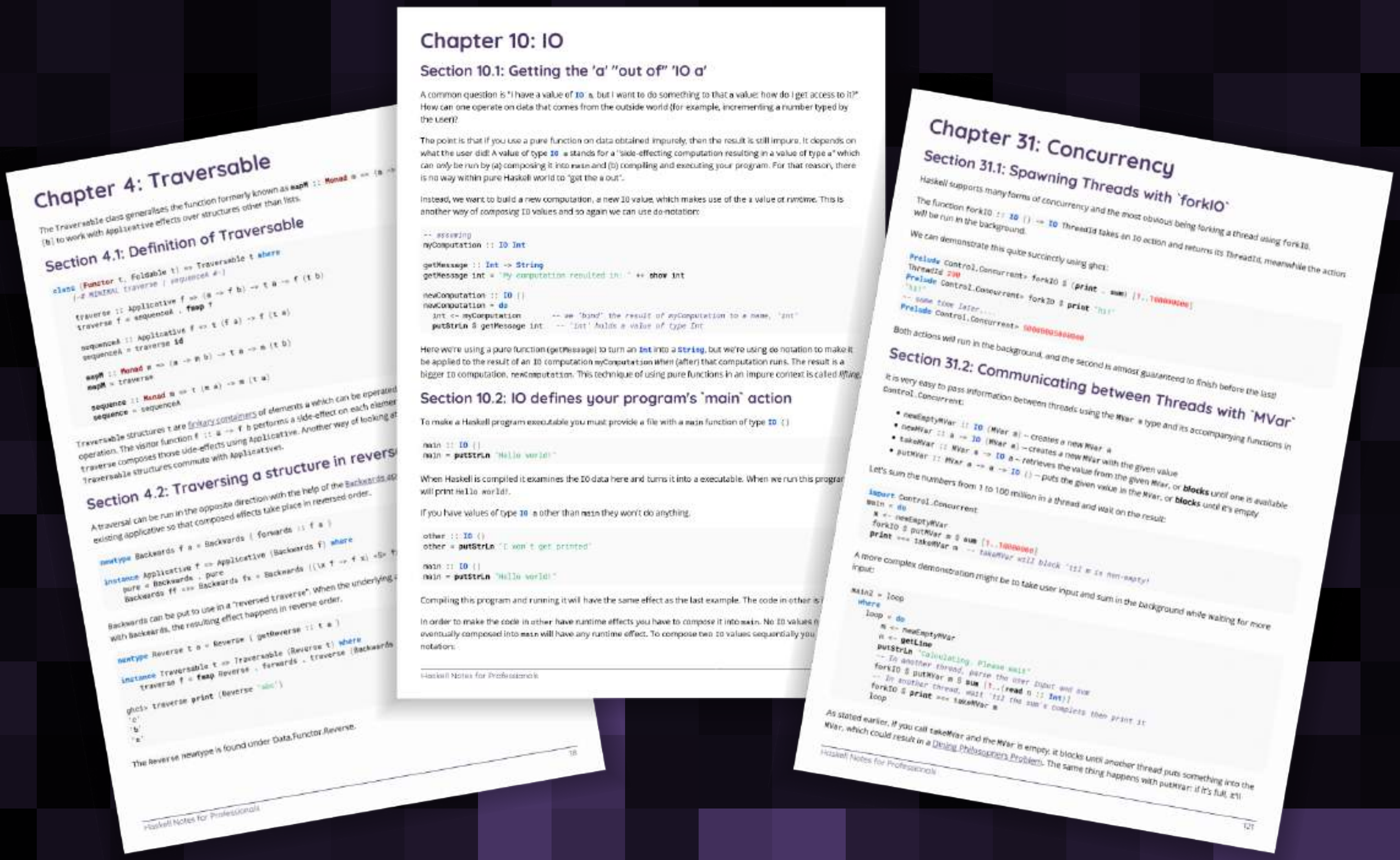
Section 31.1: Spawning Threads with `forkIO`

Section 31.2: Communicating between Threads with `MVar`

200多页
专业提示和技巧

GoalKicker.com
免费编程书籍

Haskell
Notes for Professionals


Chapter 4: Traversable

Section 4.1: Definition of Traversable

Section 4.2: Traversing a structure in reverse

Chapter 10: IO

Section 10.1: Getting the 'a' "out of" 'IO a'

Section 10.2: IO defines your program's `main` action

Chapter 31: Concurrency

Section 31.1: Spawning Threads with `forkIO`

Section 31.2: Communicating between Threads with `MVar`

200+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

目录

Contents

Chapter 40: Cabal 137
Section 40.1: Working with sandboxes 137
Section 40.2: Install packages 137
Chapter 41: Type algebra 138
Section 41.1: Addition and multiplication 138
Section 41.2: Functions 139
Section 41.3: Natural numbers in type algebra 139
Section 41.4: Recursive types 140
Section 41.5: Derivatives 141
Chapter 42: Arrows 142
Section 42.1: Function compositions with multiple channels 142
Chapter 43: Typed holes 143
Section 43.1: Syntax of typed holes 143
Section 43.2: Semantics of typed holes 143
Section 43.3: Using typed holes to define a class instance 143
Chapter 44: Rewrite rules (GHC) 146
Section 44.1: Using rewrite rules on overloaded functions 146
Chapter 45: Date and Time 147
Section 45.1: Finding Today's Date 147
Section 45.2: Adding, Subtracting and Comparing Days 147
Chapter 46: List Comprehensions 148
Section 46.1: Basic List Comprehensions 148
Section 46.2: Do Notation 148
Section 46.3: Patterns in Generator Expressions 148
Section 46.4: Guards 149
Section 46.5: Parallel Comprehensions 149
Section 46.6: Local Bindings 149
Section 46.7: Nested Generators 150
Chapter 47: Streaming IO 151
Section 47.1: Streaming IO 151
Chapter 48: Google Protocol Buffers 152
Section 48.1: Creating, building and using a simple .proto file 152
Chapter 49: Template Haskell & QuasiQuotes 154
Section 49.1: Syntax of Template Haskell and Quasiquotes 154
Section 49.2: The Q type 155
Section 49.3: An n-arity curry 156
Chapter 50: Phantom types 158
Section 50.1: Use Case for Phantom Types: Currencies 158
Chapter 51: Modules 159
Section 51.1: Defining Your Own Module 159
Section 51.2: Exporting Constructors 159
Section 51.3: Importing Specific Members of a Module 159
Section 51.4: Hiding Imports 160
Section 51.5: Qualifying Imports 160
Section 51.6: Hierarchical module names 160
Chapter 52: Tuples (Pairs, Triples, ...) 162
Section 52.1: Extract tuple components 162
Section 52.2: Strictness of matching a tuple 162

关于

About

第1章：Haskell语言入门

版本	发布日期
Haskell 2010	2012-07-10
Haskell 98	2002-12-01

第1.1节：入门

在线REPL

开始编写Haskell最简单的方法可能是访问Haskell网站或Try Haskell，并使用主页上的在线REPL（读-评估-打印循环）。在线REPL支持大多数基本功能，甚至包括一些IO。还有一个基本教程，可以通过输入命令help来启动。是学习Haskell基础知识并尝试一些内容的理想工具。

GHC(i)

对于准备更深入学习的程序员，有GHCi，这是一个随Glorious/Glasgow Haskell Compiler一起提供的交互式环境。可以单独安装GHC，但那只是一个编译器。为了能够安装新库，还必须安装像Cabal和Stack这样的工具。如果你使用类Unix操作系统，最简单的安装方法是使用以下命令安装Stack：

```
curl -sSL https://get.haskellstack.org/ | sh
```

这会将GHC安装在系统的隔离环境中，因此易于移除。所有命令都必须以
stack 开头。不过，另一种简单的方法是安装Haskell平台。该平台有两种版本：

1. minimal版本仅包含GHC（用于编译）和Cabal/Stack（用于安装和构建包）
2. full版本则额外包含项目开发、性能分析和覆盖率分析工具。此外包含了一组额外的广泛使用的软件包。

这些平台可以通过下载安装程序并按照说明操作，或者使用您的发行版的软件包管理器安装（注意此版本不保证是最新的）：

- Ubuntu、Debian、Mint：

  ```
  sudo apt-get install haskell-platform
  ```

- Fedora：

  ```
  sudo dnf install haskell-platform
  ```

- Redhat：

  ```
  sudo yum install haskell-platform
  ```

- Arch Linux：

  ```
  sudo pacman -S ghc cabal-install haskell-haddock-api \
                 haskell-haddock-library happy alex
  ```

Chapter 1: Getting started with Haskell Language

Version	Release Date
Haskell 2010	2012-07-10
Haskell 98	2002-12-01

Section 1.1: Getting started

Online REPL

The easiest way to get started writing Haskell is probably by going to the Haskell website or Try Haskell and use the online REPL (read-eval-print-loop) on the home page. The online REPL supports most basic functionality and even some IO. There is also a basic tutorial available which can be started by typing the command `help`. An ideal tool to start learning the basics of Haskell and try out some stuff.

GHC(i)

For programmers that are ready to engage a little bit more, there is *GHCi*, an interactive environment that comes with the *Glorious/Glasgow Haskell Compiler*. The *GHC* can be installed separately, but that is only a compiler. In order to be able to install new libraries, tools like *Cabal* and *Stack* must be installed as well. If you are running a Unix-like operating system, the easiest installation is to install *Stack* using:

```
curl -sSL https://get.haskellstack.org/ | sh
```

This installs GHC isolated from the rest of your system, so it is easy to remove. All commands must be preceded by `stack` though. Another simple approach is to install a Haskell Platform. The platform exists in two flavours:

1. The **minimal** distribution contains only *GHC* (to compile) and *Cabal/Stack* (to install and build packages)
2. The **full** distribution additionally contains tools for project development, profiling and coverage analysis. Also an additional set of widely-used packages is included.

These platforms can be installed by downloading an installer and following the instructions or by using your distribution's package manager (note that this version is not guaranteed to be up-to-date):

- Ubuntu, Debian, Mint:

  ```
  sudo apt-get install haskell-platform
  ```

- Fedora:

  ```
  sudo dnf install haskell-platform
  ```

- Redhat:

  ```
  sudo yum install haskell-platform
  ```

- Arch Linux:

  ```
  sudo pacman -S ghc cabal-install haskell-haddock-api \
                 haskell-haddock-library happy alex
  ```

- Gentoo：

```
sudo layman -a haskell
sudo emerge haskell-platform
```

- OSX 使用 Homebrew：

```
brew cask install haskell-platform
```

- OSX 使用 MacPorts：

```
sudo port install haskell-platform
```

安装完成后，应当可以通过在终端任何位置输入ghci命令来启动GHCi。如果安装顺利，控制台应显示类似如下内容

```
me@notebook:~$ ghci
GHCi，版本 6.12.1: http://www.haskell.org/ghc/  :? 获取帮助
Prelude>
```

可能还会有一些关于在Prelude>之前加载了哪些库的更多信息。现在，控制台已经变成了一个Haskell交互式环境（REPL），你可以像使用在线REPL一样执行Haskell代码。要退出这个
交互式环境，可以输入:q或:quit。有关GHCi中可用命令的更多信息，请按照启动屏幕上的提示输入:?。

因为在一行中反复写相同的内容并不总是很实用，写Haskell代码到文件中可能是个好主意。这些文件通常以.hs为扩展名，可以通过使用:l或:load命令加载到REPL中。

如前所述，GHCi是GHC的一部分，后者实际上是一个编译器。该编译器可以将包含Haskell代码的.hs文件转换成可运行的程序。由于.hs文件中可以包含许多函数，必须在文件中定义一个main函数。这个函数将作为程序的起点。文件test.hs可以通过以下命令编译

```
ghc test.hs
```

如果没有错误且main函数定义正确，这将创建目标文件和可执行文件。

更高级的工具

1. 之前已经提到过作为包管理器，stack也可以成为Haskell开发中非常有用的工具
   以完全不同的方式。一旦安装，它能够

   - 安装（多个版本的）*GHC*
   - 项目创建和脚手架
   - 依赖管理
   - 构建和测试项目
   - 基准测试

2. IHaskell 是一个IPython 的 Haskell 内核，允许将（可运行的）代码与 Markdown 结合使用
   数学符号。

- Gentoo:

```
sudo layman -a haskell
sudo emerge haskell-platform
```

- OSX with Homebrew:

```
brew cask install haskell-platform
```

- OSX with MacPorts:

```
sudo port install haskell-platform
```

Once installed, it should be possible to start *GHCi* by invoking the `ghci` command anywhere in the terminal. If the installation went well, the console should look something like

```
me@notebook:~$ ghci
GHCi, version 6.12.1: http://www.haskell.org/ghc/  :? for help
Prelude>
```

possibly with some more information on what libraries have been loaded before the `Prelude>`. Now, the console has become a Haskell REPL and you can execute Haskell code as with the online REPL. In order to quit this interactive environment, one can type `:q`or `:quit`. For more information on what commands are available in *GHCi*, type `:?` as indicated in the starting screen.

Because writing the same things again and again on a single line is not always that practically, it might be a good idea to write the Haskell code in files. These files normally have `.hs` for an extension and can be loaded into the REPL by using `:l` or `:load`.

As mentioned earlier, *GHCi* is a part of the *GHC*, which is actually a compiler. This compiler can be used to transform a `.hs` file with Haskell code into a running program. Because a `.hs` file can contain a lot of functions, a `main` function must be defined in the file. This will be the starting point for the program. The file `test.hs` can be compiled with the command

```
ghc test.hs
```

this will create object files and an executable if there were no errors and the `main` function was defined correctly.

More advanced tools

1. It has already been mentioned earlier as package manager, but *stack* can be a useful tool for Haskell development in completely different ways. Once installed, it is capable of

   - installing (multiple versions of) *GHC*
   - project creation and scaffolding
   - dependency management
   - building and testing projects
   - benchmarking

2. IHaskell is a haskell kernel for IPython and allows to combine (runnable) code with markdown and mathematical notation.

第1.2节：你好，世界！

一个基本的"你好，世界！"程序在Haskell中可以简洁地用一两行代码表示：

```
main :: IO ()
main = putStrLn "Hello, World!"
```

第一行是可选的类型注解，表示main是类型为IO ()的值，代表一个I/O操作该操作"计算"一个类型为()（读作"unit"；空元组表示无信息）的值，同时在外部世界执行一些副作用（这里是在终端打印字符串）。这个类型注解通常会被省略，因为main只有这一种可能的类型。

将此代码保存到一个helloworld.hs文件中，并使用Haskell编译器（如GHC）进行编译：

```
ghc helloworld.hs
```

执行编译后的文件将会在屏幕上打印输出"Hello, World!"：

```
./helloworld
你好，世界！
```

或者，runhaskell 或 runghc 可以在不编译的情况下以解释模式运行程序：

```
runhaskell helloworld.hs
```

交互式REPL也可以用来代替编译。它随大多数Haskell环境一起提供，例如带有GHC编译器的ghci：

```
ghci> putStrLn "Hello World!"
Hello, World!
ghci>
```

或者，使用load（或:l）从文件加载脚本到ghci中：

```
ghci> :load helloworld
```

:reload（或:r）会重新加载ghci中的所有内容：

```
Prelude> :l helloworld.hs
[1 of 1] 正在编译 Main             ( helloworld.hs，解释模式 )

<稍后经过一些编辑>

*Main> :r
好的，模块已加载: Main。
```

解释：

第一行是类型签名，声明了main的类型：

```
main :: IO ()
```

类型为IO ()的值描述了可以与外部世界交互的动作。

Section 1.2: Hello, World!

A basic "Hello, World!" program in Haskell can be expressed concisely in just one or two lines:

```
main :: IO ()
main = putStrLn "Hello, World!"
```

The first line is an optional type annotation, indicating that `main` is a value of type `IO ()`, representing an I/O action which "computes" a value of type `()` (read "unit"; the empty tuple conveying no information) besides performing some side effects on the outside world (here, printing a string at the terminal). This type annotation is usually omitted for `main` because it is its *only* possible type.

Put this into a `helloworld.hs` file and compile it using a Haskell compiler, such as GHC:

```
ghc helloworld.hs
```

Executing the compiled file will result in the output `"Hello, World!"` being printed to the screen:

```
./helloworld
Hello, World!
```

Alternatively, `runhaskell` or `runghc` make it possible to run the program in interpreted mode without having to compile it:

```
runhaskell helloworld.hs
```

The interactive REPL can also be used instead of compiling. It comes shipped with most Haskell environments, such as `ghci` which comes with the GHC compiler:

```
ghci> putStrLn "Hello World!"
Hello, World!
ghci>
```

Alternatively, load scripts into ghci from a file using `load` (or `:l`):

```
ghci> :load helloworld
```

`:reload` (or `:r`) reloads everything in ghci:

```
Prelude> :l helloworld.hs
[1 of 1] Compiling Main             ( helloworld.hs, interpreted )

<some time later after some edits>

*Main> :r
Ok, modules loaded: Main.
```

Explanation:

This first line is a type signature, declaring the type of `main`:

```
main :: IO ()
```

Values of type `IO ()` describe actions which can interact with the outside world.

由于Haskell拥有一个完善的Hindley-Milner类型系统，支持自动类型推断，类型签名在技术上是可选的：如果你省略了main :: IO ()，编译器将通过分析main的定义自动推断类型。然而，不为顶层定义写类型签名被认为是很不好的风格。原因包括：

- Haskell中的类型签名是非常有用的文档，因为类型系统非常表达力强，你通常可以仅通过查看函数的类型就知道它适合做什么。这
“文档”可以通过像GHCi这样的工具方便地访问。且与普通文档不同，编译器的类型检查器会确保它实际上与函数定义相匹配！
- 类型签名将错误局限于本地。如果你在定义中犯错但没有提供类型签名，编译器可能不会立即报错，而是推断出一个无意义的类型，并用它进行类型检查。你可能会在使用该值时收到晦涩难懂的错误信息。有了签名，编译器能非常准确地在错误发生的地方发现问题。

第二行执行实际的工作：

```
main = putStrLn "Hello, World!"
```

如果你来自命令式语言，注意这个定义也可以写成这样可能会有所帮助：

```
main = do {
   putStrLn "Hello, World!" ;
   return ()
   }
```

或者等价地（Haskell 使用基于布局的解析；但*注意不要混用制表符和空格*，否则会干扰该机制）：

```
main = do
    putStrLn "Hello, World!"
    return ()
```

do 块中的每一行表示某个单子（这里是 I/O）计算，因此整个 do 块表示由这些子步骤组成的整体操作，通过以特定于给定单子的方式组合它们（对于 I/O这意味着按顺序执行它们）。

do 语法本身是单子的语法糖，比如这里的 IO，**return** 是一个无操作动作，产生其参数而不执行任何副作用或可能属于特定
单子定义的额外计算。

上述等同于定义main= putStrLn"Hello, World!"，因为值putStrLn"Hello,World!"已经具有类型IO()。作为一个“语句”来看，putStrLn"Hello, World!"可以被视为一个完整的程序，而你只是简单地定义main来引用这个程序。

你可以在线查询putStrLn的签名：

```
putStrLn :: String -> IO ()

putStrLn (v :: String) :: IO ()
```

putStrLn是一个函数，它以字符串作为参数并输出一个I/O动作（即表示运行时可以执行的程序的值）。运行时总是执行名为main的动作，所以我们只需将其定义为等于putStrLn "Hello, World!"。

Because Haskell has a fully-fledged Hindley-Milner type system which allows for automatic type inference, type signatures are technically optional: if you simply omit the `main :: IO ()`, the compiler will be able to infer the type on its own by analyzing the *definition* of `main`. However, it is very much considered bad style not to write type signatures for top-level definitions. The reasons include:

- Type signatures in Haskell are a very helpful piece of documentation because the type system is so expressive that you often can see what sort of thing a function is good for simply by looking at its type. This “documentation” can be conveniently accessed with tools like GHCi. And unlike normal documentation, the compiler's type checker will make sure it actually matches the function definition!
- Type signatures *keep bugs local*. If you make a mistake in a definition without providing its type signature, the compiler may not immediately report an error but instead simply infer a nonsensical type for it, with which it actually typechecks. You may then get a cryptic error message when *using* that value. With a signature, the compiler is very good at spotting bugs right where they happen.

This second line does the actual work:

```
main = putStrLn "Hello, World!"
```

If you come from an imperative language, it may be helpful to note that this definition can also be written as:

```
main = do {
   putStrLn "Hello, World!" ;
   return ()
   }
```

Or equivalently (Haskell has layout-based parsing; but *beware mixing tabs and spaces inconsistently* which will confuse this mechanism):

```
main = do
    putStrLn "Hello, World!"
    return ()
```

Each line in a do block represents some monadic (here, I/O) *computation*, so that the whole do block represents the overall action comprised of these sub-steps by combining them in a manner specific to the given monad (for I/O this means just executing them one after another).

The do syntax is itself a syntactic sugar for monads, like `IO` here, and **`return`** is a no-op action producing its argument without performing any side effects or additional computations which might be part of a particular monad definition.

The above is the same as defining `main = putStrLn "Hello, World!"`, because the value `putStrLn "Hello, World!"` already has the type `IO ()`. Viewed as a “statement”, `putStrLn "Hello, World!"` can be seen as a complete program, and you simply define `main` to refer to this program.

You can look up the signature of **`putStrLn`** online:

```
putStrLn :: String -> IO ()
-- thus,
putStrLn (v :: String) :: IO ()
```

`putStrLn` is a function that takes a string as its argument and outputs an I/O-action (i.e. a value representing a program that the runtime can execute). The runtime always executes the action named `main`, so we simply need to define it as equal to `putStrLn "Hello, World!"`.

第1.3节：阶乘

阶乘函数是Haskell的“Hello World！”（以及函数式编程的一般意义上的“Hello World！”），因为它简洁地展示了语言的基本原理。

变体1

```haskell
fac :: (Integral a) => a -> a
fac n = product [1..n]
```

实时演示

- Integral是整数类型的类。示例包括Int和Integer。
- `(Integral a) =>` 对类型 a 施加约束，使其属于该类
- fac :: a -> a 表示 fac 是一个接受 a 并返回 a 的函数
- `product` 是一个函数，通过将列表中的所有数字相乘来累积结果。
- [1..n] 是一种特殊表示法，等价于 enumFromTo 1 n，表示数字范围 1 ≤ x ≤ n。

变体 2

```haskell
fac :: (Integral a) => a -> a
fac 0 = 1
fac n = n * fac (n - 1)
```

实时演示

该变体使用模式匹配将函数定义拆分为不同的情况。第一个定义在参数为 0 时调用（有时称为终止条件），否则调用第二个定义（定义的顺序很重要）。它也展示了递归，因为 fac 调用了自身。

值得注意的是，由于重写规则，当使用启用优化的 GHC 编译时，两个版本的 fac 将编译成相同的机器代码。因此，在效率方面，两者是等价的。

第1.4节：斐波那契数列，使用惰性求值

惰性求值意味着 Haskell 只会计算那些值被需要的列表项。

基本的递归定义是：

```
f (0)  <-  0
f (1)  <-  1
f (n)  <-  f (n-1) + f (n-2)
```

如果直接计算，会非常慢。但，想象一下我们有一个列表记录所有结果，

```
fibs !! n  <-  f (n)
```

然后

```
                  ┌──────┐   ┌──────┐   ┌──────┐
                  │ f(0) │   │ f(1) │   │ f(2) │
fibs  ->  0 : 1 : │  +   │ : │  +   │ : │  +   │ :  .....
                  │ f(1) │   │ f(2) │   │ f(3) │
                  └──────┘   └──────┘   └──────┘

                  ┌----------------------------------------┐
```

Section 1.3: Factorial

The factorial function is a Haskell "Hello World!" (and for functional programming generally) in the sense that it succinctly demonstrates basic principles of the language.

Variation 1

```haskell
fac :: (Integral a) => a -> a
fac n = product [1..n]
```

Live demo

- `Integral` is the class of integral number types. Examples include `Int` and `Integer`.
- `(Integral a) =>` places a constraint on the type `a` to be in said class
- `fac :: a -> a` says that `fac` is a function that takes an `a` and returns an `a`
- `product` is a function that accumulates all numbers in a list by multiplying them together.
- `[1..n]` is special notation which desugars to `enumFromTo 1 n`, and is the range of numbers `1 ≤ x ≤ n`.

Variation 2

```haskell
fac :: (Integral a) => a -> a
fac 0 = 1
fac n = n * fac (n - 1)
```

Live demo

This variation uses pattern matching to split the function definition into separate cases. The first definition is invoked if the argument is `0` (sometimes called the stop condition) and the second definition otherwise (the order of definitions is significant). It also exemplifies recursion as `fac` refers to itself.

It is worth noting that, due to rewrite rules, both versions of `fac` will compile to identical machine code when using GHC with optimizations activated. So, in terms of efficiency, the two would be equivalent.

Section 1.4: Fibonacci, Using Lazy Evaluation

Lazy evaluation means Haskell will evaluate only list items whose values are needed.

The basic recursive definition is:

```
f (0)  <-  0
f (1)  <-  1
f (n)  <-  f (n-1) + f (n-2)
```

If evaluated directly, it will be *very* slow. But, imagine we have a list that records all the results,

```
fibs !! n  <-  f (n)
```

Then

```
                  ┌──────┐   ┌──────┐   ┌──────┐
                  │ f(0) │   │ f(1) │   │ f(2) │
fibs  ->  0 : 1 : │  +   │ : │  +   │ : │  +   │ :  .....
                  │ f(1) │   │ f(2) │   │ f(3) │
                  └──────┘   └──────┘   └──────┘

                  ┌────────────────────────────────────────┐
```

```
                    | f(0)   :   f(1)   :   f(2)   :  ..... |
                    ------------------------------------------
      -> 0 : 1 :                 +
                    ------------------------------------------
                    | f(1)   :   f(2)   :   f(3)   :  ..... |
                    ------------------------------------------
```

这是编码为：

```
fibn n = fibs !! n
    其中
fibs = 0 : 1 : map f [2..]
    f n = fibs !! (n-1) + fibs !! (n-2)
```

或者甚至写成

```
GHCi> let fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
GHCi> take 10 fibs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

zipWith 通过对传入的两个列表对应元素应用给定的二元函数来生成一个列表，因此 zipWith (+) [x1, x2, ...] [y1, y2, ...] 等同于 [x1 + y1, x2 + y2, ...]。

另一种写法是用 scanl 函数来定义 fibs：

```
GHCi> let fibs = 0 : scanl (+) 1 fibs
GHCi> take 10 fibs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

scanl 构建了部分结果的列表，这些结果是 foldl 从左到右沿输入列表产生的。也就是说，scanl f z0 [x1, x2, ...] 等于 [z0, z1, z2, ...]，其中 z1 = f z0 x1；z2 = f z1 x2；....。

得益于惰性求值，这两个函数定义了无限列表而无需完全计算它们。也就是说，我们可以编写一个fib函数，获取无限斐波那契数列的第n个元素：

```
GHCi> let fib n = fibs !! n  -- (!!) 是列表下标操作符
-- 或者用点自由风格：
GHCi> let fib = (fibs !!)
GHCi> fib 9
34
```

第1.5节：素数

一些最显著的变体：

小于100

```
import Data.List ( (\\) )

ps100 = ((([2..100] \\ [4,6..100]) \\ [6,9..100]) \\ [10,15..100]) \\ [14,21..100]

   -- = (((2:[3,5..100]) \\ [9,15..100]) \\ [25,35..100]) \\ [49,63..100]

   -- = (2:[3,5..100]) \\ ([9,15..100] ++ [25,35..100] ++ [49,63..100])
```

无限制

埃拉托斯特尼筛法，使用data-ordlist包：

```
                    | f(0)   :   f(1)   :   f(2)   :  ..... |
                    ------------------------------------------
      -> 0 : 1 :                 +
                    ------------------------------------------
                    | f(1)   :   f(2)   :   f(3)   :  ..... |
                    ------------------------------------------
```

This is coded as:

```
fibn n = fibs !! n
    where
    fibs = 0 : 1 : map f [2..]
    f n = fibs !! (n-1) + fibs !! (n-2)
```

Or even as

```
GHCi> let fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
GHCi> take 10 fibs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

`zipWith` makes a list by applying a given binary function to corresponding elements of the two lists given to it, so `zipWith (+) [x1, x2, ...] [y1, y2, ...]` is equal to `[x1 + y1, x2 + y2, ...]`.

Another way of writing `fibs` is with the `scanl` function:

```
GHCi> let fibs = 0 : scanl (+) 1 fibs
GHCi> take 10 fibs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

`scanl` builds the list of partial results that `foldl` would produce, working from left to right along the input list. That is, `scanl f z0 [x1, x2, ...]` is equal to `[z0, z1, z2, ...] where z1 = f z0 x1; z2 = f z1 x2; ...`.

Thanks to lazy evaluation, both functions define infinite lists without computing them out entirely. That is, we can write a `fib` function, retrieving the nth element of the unbounded Fibonacci sequence:

```
GHCi> let fib n = fibs !! n  -- (!!) being the list subscript operator
-- or in point-free style:
GHCi> let fib = (fibs !!)
GHCi> fib 9
34
```

Section 1.5: Primes

A few *most salient* variants:

Below 100

```
import Data.List ( (\\) )

ps100 = ((([2..100] \\ [4,6..100]) \\ [6,9..100]) \\ [10,15..100]) \\ [14,21..100]

   -- = (((2:[3,5..100]) \\ [9,15..100]) \\ [25,35..100]) \\ [49,63..100]

   -- = (2:[3,5..100]) \\ ([9,15..100] ++ [25,35..100] ++ [49,63..100])
```

Unlimited

Sieve of Eratosthenes, using data-ordlist package:

```
import qualified Data.List.Ordered

ps   = 2 : _Y ((3:) . minus [5,7..] . unionAll . map (\p -> [p*p, p*p+2*p..]))

_Y g = g (_Y g)   -- = g (g (_Y g)) = g (g (g (g (...)))) = g . g . g . g . ...
```

传统

（一种次优的试除筛法）

```
ps = 筛法 [2..]
     其中
筛法 (x:xs) = [x] ++ 筛法 [y | y <- xs, rem y x > 0]

-- = map head ( iterate (\(x:xs) -> filter ((> 0).(`rem` x)) xs) [2..] )
```

最优试除法

```
ps = 2 : [n | n <- [3..], all ((> 0).rem n) $ takeWhile ((<= n).(^2)) ps]

-- = 2 : [n | n <- [3..], foldr (\p r-> p*p > n || (rem n p > 0 && r)) True ps]
```

过渡

从试除法到埃拉托斯特尼筛法：

```
[n | n <- [2..], []==[i | i <- [2..n-1], j <- [0,i..n], j==n]]
```

最短代码

```
nubBy (((>1).).gcd) [2..]          -- 即 nubBy (\a b -> gcd a b > 1) [2..]
```

nubBy 也来自 Data.List，类似于 (\\)。

第1.6节：声明值

我们可以在REPL中这样声明一系列表达式：

```
Prelude> let x = 5
Prelude> let y = 2 * 5 + x
Prelude> let result = y * 10
Prelude> x
5
Prelude> y
15
Prelude> 结果
150
```

要在文件中声明相同的值，我们写如下内容：

```
-- demo.hs

module Demo where
-- 我们声明模块的名称，
-- 以便在项目中可以按名称导入。

x = 5

y = 2 * 5 + x

result = y * 10
```

```
import qualified Data.List.Ordered

ps   = 2 : _Y ((3:) . minus [5,7..] . unionAll . map (\p -> [p*p, p*p+2*p..]))

_Y g = g (_Y g)   -- = g (g (_Y g)) = g (g (g (g (...)))) = g . g . g . g . ...
```

Traditional

(a sub-optimal trial division sieve)

```
ps = sieve [2..]
     where
     sieve (x:xs) = [x] ++ sieve [y | y <- xs, rem y x > 0]

-- = map head ( iterate (\(x:xs) -> filter ((> 0).(`rem` x)) xs) [2..] )
```

Optimal trial division

```
ps = 2 : [n | n <- [3..], all ((> 0).rem n) $ takeWhile ((<= n).(^2)) ps]

-- = 2 : [n | n <- [3..], foldr (\p r-> p*p > n || (rem n p > 0 && r)) True ps]
```

Transitional

From trial division to sieve of Eratosthenes:

```
[n | n <- [2..], []==[i | i <- [2..n-1], j <- [0,i..n], j==n]]
```

The Shortest Code

```
nubBy (((>1).).gcd) [2..]          -- i.e., nubBy (\a b -> gcd a b > 1) [2..]
```

nubBy is also from `Data.List`, like `(\\)`.

Section 1.6: Declaring Values

We can declare a series of expressions in the REPL like this:

```
Prelude> let x = 5
Prelude> let y = 2 * 5 + x
Prelude> let result = y * 10
Prelude> x
5
Prelude> y
15
Prelude> result
150
```

To declare the same values in a file we write the following:

```
-- demo.hs

module Demo where
-- We declare the name of our module so
-- it can be imported by name in a project.

x = 5

y = 2 * 5 + x

result = y * 10
```

模块名称首字母大写，变量名称则不然。

Module names are capitalized, unlike variable names.

第二章：超载字面量

第2.1节：字符串

字面量的类型

在没有任何扩展的情况下，字符串字面量的类型–即双引号之间的内容–仅仅是字符串，也就是字符列表：

```
Prelude> :t "foo"
"foo" :: [Char]
```

然而，当启用OverloadedStrings扩展时，字符串字面量变得多态，类似于数字字面量：

```
Prelude> :set -XOverloadedStrings
Prelude> :t "foo"
"foo" :: Data.String.IsString t => t
```

这使我们能够定义字符串类类型的值，而无需任何显式转换。实质上，OverloadedStrings扩展只是将每个字符串字面量包装在通用的fromString转换函数中，因此如果上下文需要例如更高效的Text而不是String，你无需自己担心。

使用字符串字面量

```
{-# LANGUAGE OverloadedStrings #-}

import Data.Text (Text, pack)
import Data.ByteString (ByteString, pack)


withString :: String
withString = "Hello String"

-- 以下两个示例仅在启用 OverloadedStrings 时允许

withText :: Text
withText = "Hello Text"      -- 代替：withText = Data.Text.pack "Hello Text"

withBS :: ByteString
withBS = "Hello ByteString"  -- 代替：withBS = Data.ByteString.pack "Hello ByteString"
```

注意我们如何能够以构造普通 String
（或 [Char]）值的相同方式构造 Text 和 ByteString 的值，而不是使用各自类型的 pack 函数来显式编码字符串。

有关 OverloadedStrings 语言扩展的更多信息，请参阅该扩展的文档。

第2.2节：浮点数字

字面量的类型

```
Prelude> :t 1.0
1.0 :: Fractional a => a
```

选择带注解的具体类型

您可以使用类型注解来指定类型。唯一的要求是该类型必须具有Fractional

Chapter 2: Overloaded Literals

Section 2.1: Strings

The type of the literal

Without any extensions, the type of a string literal – i.e., something between double quotes – is just a string, aka list of characters:

```
Prelude> :t "foo"
"foo" :: [Char]
```

However, when the `OverloadedStrings` extension is enabled, string literals become polymorphic, similar to number literals:

```
Prelude> :set -XOverloadedStrings
Prelude> :t "foo"
"foo" :: Data.String.IsString t => t
```

This allows us to define values of string-like types without the need for any explicit conversions. In essence, the `OverloadedStrings` extension just wraps every string literal in the generic `fromString` conversion function, so if the context demands e.g. the more efficient `Text` instead of `String`, you don't need to worry about that yourself.

Using string literals

```
{-# LANGUAGE OverloadedStrings #-}

import Data.Text (Text, pack)
import Data.ByteString (ByteString, pack)


withString :: String
withString = "Hello String"

-- The following two examples are only allowed with OverloadedStrings

withText :: Text
withText = "Hello Text"      -- instead of: withText = Data.Text.pack "Hello Text"

withBS :: ByteString
withBS = "Hello ByteString"  -- instead of: withBS = Data.ByteString.pack "Hello ByteString"
```

Notice how we were able to construct values of `Text` and `ByteString` in the same way we construct ordinary `String` (or `[Char]`) Values, rather than using each types `pack` function to encode the string explicitly.

For more information on the `OverloadedStrings` language extension, see the extension documentation.

Section 2.2: Floating Numeral

The type of the literal

```
Prelude> :t 1.0
1.0 :: Fractional a => a
```

Choosing a concrete type with annotations

You can specify the type with a *type annotation*. The only requirement is that the type must have a `Fractional`

实例。

```
Prelude> 1.0 :: Double
1.0
it :: Double
Prelude> 1.0 :: Data.Ratio.Ratio Int
1 % 1
it :: GHC.Real.Ratio Int
```

否则编译器会报错

```
Prelude> 1.0 :: Int
<interactive>:
无法为字面量 `1.0' 生成 (Fractional Int) 的实例
    表达式中: 1.0 :: Int
在 `it' 的等式中: it = 1.0 :: Int
```

第2.3节：整数数字

字面量的类型

```
Prelude> :t 1
1 :: Num a => a
```

选择带注解的具体类型

只要目标类型是带有注解的Num类型，就可以指定类型：

```
Prelude> 1 :: Int
1
it :: Int
Prelude> 1 :: Double
1.0
它 :: 双精度浮点数
Prelude> 1 :: 无符号整数
1
it :: 无符号整数
```

如果不这样做，编译器会报错

Prelude> 1 :: 字符串

```
<interactive>:
没有为 (Num String) 提供实例，导致字面量`1`出错
    在表达式：1 :: 字符串
在`it`的等式中：it = 1 :: 字符串
```

第2.4节：列表字面量

GHC的OverloadedLists扩展允许你使用列表字面量语法构造类似列表的数据结构。

这允许你像这样使用Data.Map：

```
> :set -XOverloadedLists
> import qualified Data.Map as M
> M.lookup "foo" [("foo", 1), ("bar", 2)]
Just 1
```

instance.

```
Prelude> 1.0 :: Double
1.0
it :: Double
Prelude> 1.0 :: Data.Ratio.Ratio Int
1 % 1
it :: GHC.Real.Ratio Int
```

if not the compiler will complain

```
Prelude> 1.0 :: Int
<interactive>:
    No instance for (Fractional Int) arising from the literal `1.0'
    In the expression: 1.0 :: Int
    In an equation for `it': it = 1.0 :: Int
```

Section 2.3: Integer Numeral

The type of the literal

```
Prelude> :t 1
1 :: Num a => a
```

choosing a concrete type with annotations

You can specify the type as long as the target type is `Num` with an *annotation*:

```
Prelude> 1 :: Int
1
it :: Int
Prelude> 1 :: Double
1.0
it :: Double
Prelude> 1 :: Word
1
it :: Word
```

if not the compiler will complain

Prelude> 1 :: String

```
<interactive>:
    No instance for (Num String) arising from the literal `1'
    In the expression: 1 :: String
    In an equation for `it': it = 1 :: String
```

Section 2.4: List Literals

GHC's OverloadedLists extension allows you to construct list-like data structures with the list literal syntax.

This allows you to Data.Map like this:

```
> :set -XOverloadedLists
> import qualified Data.Map as M
> M.lookup "foo" [("foo", 1), ("bar", 2)]
Just 1
```

而不是这样（注意额外使用了M.fromList）：

```
> import Data.Map as M
> M.lookup "foo" (M.fromList [("foo", 1), ("bar", 2)])
Just 1
```

Instead of this (note the use of the extra M.fromList):

```
> import Data.Map as M
> M.lookup "foo" (M.fromList [("foo", 1), ("bar", 2)])
Just 1
```

第3章：Foldable（可折叠）

Foldable 是类型类 t :: * -> * 的集合，这些类型支持folding操作。折叠操作使用组合函数以确定的顺序聚合结构中的元素。

第3.1节：Foldable的定义

```
class Foldable t where
    {-# MINIMAL foldMap | foldr #-}

    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap f = foldr (mappend . f) mempty

    foldr :: (a -> b -> b) -> b -> t a -> b
    foldr f z t = appEndo (foldMap (Endo #. f) t) z

    -- 以及若干可选方法
```

直观上（虽然不是技术上的），Foldable 结构是元素的容器，a 允许以明确定义的顺序访问其元素。foldMap 操作将容器中的每个元素映射到一个Monoid，并使用Monoid 结构将它们合并。

第3.2节：二叉树的Foldable实例

要实例化Foldable，您需要至少为foldMap或foldr提供定义。

```
数据 Tree a = 叶子 (Leaf)
            | 节点 (Node)  (Tree a) a (Tree a)

实例 Foldable Tree 其中
foldMap f 叶子 (Leaf)  = mempty
foldMap f (节点 (Node)  l x r) = foldMap f l `mappend` f x `mappend` foldMap f r

    foldr f 初始值 (acc)  叶子 (Leaf)  = 初始值 (acc)
    foldr f 初始值 (acc)  (节点 (Node)  l x r) = foldr f (f x (foldr f 初始值 (acc)  r)) l
```

该实现执行树的中序遍历。

```
ghci> let myTree = Node (Node Leaf 'a' Leaf) 'b' (Node Leaf 'c' Leaf)

--    +--'b'--+
--    |       |
-- +-'a'-+ +-'c'-+
-- |     | |     |
-- *     * *     *

ghci> toList myTree
"abc"
```

DeriveFoldable 扩展允许 GHC 根据类型的结构生成 Foldable 实例。我们可以通过调整 Node 构造函数的布局来改变机器生成的遍历顺序。

```
data Inorder a = ILeaf
               | INode (Inorder a) a (Inorder a)  -- 如前所述
               deriving Foldable
```

Chapter 3: Foldable

Foldable is the class of types t :: * -> * which admit a *folding* operation. A fold aggregates the elements of a structure in a well-defined order, using a combining function.

Section 3.1: Definition of Foldable

```
class Foldable t where
    {-# MINIMAL foldMap | foldr #-}

    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap f = foldr (mappend . f) mempty

    foldr :: (a -> b -> b) -> b -> t a -> b
    foldr f z t = appEndo (foldMap (Endo #. f) t) z

    -- and a number of optional methods
```

Intuitively (though not technically), Foldable structures are containers of elements a which allow access to their elements in a well-defined order. The foldMap operation maps each element of the container to a Monoid and collapses them using the Monoid structure.

Section 3.2: An instance of Foldable for a binary tree

To instantiate Foldable you need to provide a definition for at least foldMap or **foldr**.

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

instance Foldable Tree where
    foldMap f Leaf = mempty
    foldMap f (Node l x r) = foldMap f l `mappend` f x `mappend` foldMap f r

    foldr f acc Leaf = acc
    foldr f acc (Node l x r) = foldr f (f x (foldr f acc r)) l
```

This implementation performs an in-order traversal of the tree.

```
ghci> let myTree = Node (Node Leaf 'a' Leaf) 'b' (Node Leaf 'c' Leaf)

--    +--'b'--+
--    |       |
-- +-'a'-+ +-'c'-+
-- |     | |     |
-- *     * *     *

ghci> toList myTree
"abc"
```

The DeriveFoldable extension allows GHC to generate Foldable instances based on the structure of the type. We can vary the order of the machine-written traversal by adjusting the layout of the Node constructor.

```
data Inorder a = ILeaf
               | INode (Inorder a) a (Inorder a)  -- as before
               deriving Foldable
```

```
数据 先序 a = 先序叶子
                   | 先序节点 a (先序 a) (先序 a)派生 可折叠


数据 后序 a = 后序叶子
                    | 后序节点 (后序 a) (后序 a) a
                    派生 可折叠

-- 来自早期树类型的注入
中序 :: 树 a -> 中序 a
中序 叶子 = 中序叶子
中序 (节点 左 x 右) = 中序节点 (中序 左) x (中序 右)

先序 :: 树 a -> 先序 a
先序 叶子 = 先序叶子
先序 (节点 左 x 右) = 先序节点 x (先序 左) (先序 右)

后序 :: 树 a -> 后序 a
后序叶子 = PoLeaf
后序 (节点 l x r) = PoNode (后序 l) (后序 r) x

ghci> 转换为列表 (中序 myTree)
"abc"
ghci> 转换为列表 (先序 myTree)
"bac"
ghci> 转换为列表 (后序 myTree)
"acb"
```

第3.3节：计算可折叠结构中的元素数量

length 计算元素 a 在可折叠结构 t a 中出现的次数。

```
ghci> length [7, 2, 9]  -- t ~ []
3
ghci> length (Right 'a')  -- t ~ Either e
1  -- 'Either e a' 可能包含零个或一个 'a'
ghci> length (Left "foo")  -- t ~ Either String
0
ghci> length (3, True)  -- t ~ (,) Int
1  -- '(c, a)' 总是恰好包含一个 'a'
```

length 定义为等价于：

```
class Foldable t where
    -- ...
    length :: t a -> Int
    length = foldl' (\c _ -> c+1) 0
```

注意，这个返回类型 `Int` 限制了对通过调用
`length` 函数获得的值可以执行的操作。`fromIntegral`是一个有用的函数，可以帮助我们解决这个问题。

第3.4节：反向折叠结构

任何折叠都可以借助 the Dual monoid 反向运行，该结构将现有的 monoid 翻转，使聚合过程反向进行。

```
newtype Dual a = Dual { getDual :: a }
```

```
data Preorder a = PrLeaf
                | PrNode a (Preorder a) (Preorder a)
                deriving Foldable

data Postorder a = PoLeaf
                 | PoNode (Postorder a) (Postorder a) a
                 deriving Foldable

-- injections from the earlier Tree type
inorder :: Tree a -> Inorder a
inorder Leaf = ILeaf
inorder (Node l x r) = INode (inorder l) x (inorder r)

preorder :: Tree a -> Preorder a
preorder Leaf = PrLeaf
preorder (Node l x r) = PrNode x (preorder l) (preorder r)

postorder :: Tree a -> Postorder a
postorder Leaf = PoLeaf
postorder (Node l x r) = PoNode (postorder l) (postorder r) x

ghci> toList (inorder myTree)
"abc"
ghci> toList (preorder myTree)
"bac"
ghci> toList (postorder myTree)
"acb"
```

Section 3.3: Counting the elements of a Foldable structure

`length` counts the occurrences of elements `a` in a foldable structure `t a`.

```
ghci> length [7, 2, 9]  -- t ~ []
3
ghci> length (Right 'a')  -- t ~ Either e
1  -- 'Either e a' may contain zero or one 'a'
ghci> length (Left "foo")  -- t ~ Either String
0
ghci> length (3, True)  -- t ~ (,) Int
1  -- '(c, a)' always contains exactly one 'a'
```

`length` is defined as being equivalent to:

```
class Foldable t where
    -- ...
    length :: t a -> Int
    length = foldl' (\c _ -> c+1) 0
```

Note that this return type `Int` restricts the operations that can be performed on values obtained by calls to the `length` function. `fromIntegral`is a useful function that allows us to deal with this problem.

Section 3.4: Folding a structure in reverse

Any fold can be run in the opposite direction with the help of the `Dual` monoid, which flips an existing monoid so that aggregation goes backwards.

```
newtype Dual a = Dual { getDual :: a }
```

```haskell
instance Monoid m => Monoid (Dual m) where
    mempty = Dual mempty
    (Dual x) `mappend` (Dual y) = Dual (y `mappend` x)
```

当 foldMap 调用的底层幺半群被 Dual 翻转时，fold 会反向运行；以下 Reverse 类型定义在 Data.Functor.Reverse 中：

```haskell
newtype Reverse t a = Reverse { getReverse :: t a }

instance Foldable t => Foldable (Reverse t) where
    foldMap f = getDual . foldMap (Dual . f) . getReverse
```

我们可以利用这个机制为列表编写一个简洁的 reverse 函数：

```haskell
reverse :: [a] -> [a]
reverse = toList . Reverse
```

第3.5节：将 Foldable 结构展平成列表

toList 将一个 Foldable 结构 t a 展平成一个 a 的列表。

```haskell
ghci> toList [7, 2, 9]  -- t ~ []
[7, 2, 9]
ghci> toList (Right 'a')  -- t ~ Either e
"a"
ghci> toList (Left "foo")  -- t ~ Either String
[]
ghci> toList (3, True)  -- t ~ (,) Int
[True]
```

toList 定义为等价于：

```haskell
class Foldable t where
    -- ...
toList :: t a -> [a]
    toList = foldr (:) []
```

第3.6节：对Foldable结构的每个元素执行副作用

traverse_ 对Foldable结构中的每个元素执行一个Applicative动作。它忽略动作的结果，仅保留副作用。（如果需要不丢弃结果的版本，请使用Traversable。）

```haskell
-- 使用Writer applicative函子（以及Sum幺半群）
ghci> runWriter $ traverse_ (\x -> tell (Sum x)) [1,2,3]
((),Sum {getSum = 6})
-- 使用IO applicative函子
ghci> traverse_ putStrLn (Right "traversing")
traversing
ghci> traverse_ putStrLn (Left False)
-- 无输出
```

for_ 是 traverse_ 参数顺序翻转后的版本。它类似于命令式语言中的 foreach 循环。

```haskell
ghci> let greetings = ["Hello", "Bonjour", "Hola"]
ghci> :{
```

```haskell
instance Monoid m => Monoid (Dual m) where
    mempty = Dual mempty
    (Dual x) `mappend` (Dual y) = Dual (y `mappend` x)
```

When the underlying monoid of a foldMap call is flipped with Dual, the fold runs backwards; the following Reverse type is defined in Data.Functor.Reverse:

```haskell
newtype Reverse t a = Reverse { getReverse :: t a }

instance Foldable t => Foldable (Reverse t) where
    foldMap f = getDual . foldMap (Dual . f) . getReverse
```

We can use this machinery to write a terse **reverse** for lists:

```haskell
reverse :: [a] -> [a]
reverse = toList . Reverse
```

Section 3.5: Flattening a Foldable structure into a list

toList flattens a Foldable structure t a into a list of as.

```haskell
ghci> toList [7, 2, 9]  -- t ~ []
[7, 2, 9]
ghci> toList (Right 'a')  -- t ~ Either e
"a"
ghci> toList (Left "foo")  -- t ~ Either String
[]
ghci> toList (3, True)  -- t ~ (,) Int
[True]
```

toList is defined as being equivalent to:

```haskell
class Foldable t where
    -- ...
    toList :: t a -> [a]
    toList = foldr (:) []
```

Section 3.6: Performing a side-effect for each element of a Foldable structure

traverse_ executes an Applicative action for every element in a Foldable structure. It ignores the action's result, keeping only the side-effects. (For a version which doesn't discard results, use Traversable.)

```haskell
-- using the Writer applicative functor (and the Sum monoid)
ghci> runWriter $ traverse_ (\x -> tell (Sum x)) [1,2,3]
((),Sum {getSum = 6})
-- using the IO applicative functor
ghci> traverse_ putStrLn (Right "traversing")
traversing
ghci> traverse_ putStrLn (Left False)
-- nothing printed
```

for_ is traverse_ with the arguments flipped. It resembles a foreach loop in an imperative language.

```haskell
ghci> let greetings = ["Hello", "Bonjour", "Hola"]
ghci> :{
```

```
ghci|     for_ greetings $ \greeting -> do
ghci|         print (greeting ++ " Stack Overflow!")
ghci| :}
"Hello Stack Overflow!"
"Bonjour Stack Overflow!"
"Hola Stack Overflow!"
```

sequenceA_ 将一个包含 Applicative 操作的 Foldable 集合折叠成一个单一操作，忽略结果。

```
ghci> let actions = [putStrLn "one", putStLn "two"]
ghci> sequenceA_ actions
one
two
```

traverse_ 定义为等价于：

```
traverse_ :: (Foldable t, Applicative f) => (a -> f b) -> t a -> f ()
traverse_ f = foldr (\x action -> f x *> action) (pure ())
```

sequenceA_ 定义为：

```
sequenceA_ :: (Foldable t, Applicative f) -> t (f a) -> f ()
sequenceA_ = traverse_ id
```

此外，当Foldable也是一个Functor时，traverse_和sequenceA_具有以下关系：

```
traverse_ f = sequenceA_ . fmap f
```

第3.7节：将Foldable结构展平成一个Monoid

foldMap将Foldable结构的每个元素映射为一个Monoid，然后将它们组合成一个单一的值。

foldMap和foldr可以相互定义，这意味着Foldable的实例只需为其中一个提供定义。

```
class Foldable t where
foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap f = foldr (mappend . f) mempty
```

使用Product monoid的示例用法：

```
product :: (Num n, Foldable t) => t n -> n
product = getProduct . foldMap Product
```

第3.8节：检查Foldable结构是否为空

null如果foldable结构 t a中没有元素 a，则返回True；如果有一个或多个元素，则返回False。对于结构，其null为True时，长度length为0。

```
ghci> null []
True
ghci> null [14, 29]
False
ghci> null Nothing
True
ghci> null (Right 'a')
```

```
ghci|     for_ greetings $ \greeting -> do
ghci|         print (greeting ++ " Stack Overflow!")
ghci| :}
"Hello Stack Overflow!"
"Bonjour Stack Overflow!"
"Hola Stack Overflow!"
```

sequenceA_ collapses a Foldable full of Applicative actions into a single action, ignoring the result.

```
ghci> let actions = [putStrLn "one", putStLn "two"]
ghci> sequenceA_ actions
one
two
```

traverse_ is defined as being equivalent to:

```
traverse_ :: (Foldable t, Applicative f) => (a -> f b) -> t a -> f ()
traverse_ f = foldr (\x action -> f x *> action) (pure ())
```

sequenceA_ is defined as:

```
sequenceA_ :: (Foldable t, Applicative f) -> t (f a) -> f ()
sequenceA_ = traverse_ id
```

Moreover, when the Foldable is also a **Functor**, traverse_ and sequenceA_ have the following relationship:

```
traverse_ f = sequenceA_ . fmap f
```

Section 3.7: Flattening a Foldable structure into a Monoid

foldMap maps each element of the Foldable structure to a Monoid, and then combines them into a single value.

foldMap and **foldr** can be defined in terms of one another, which means that instances of Foldable need only give a definition for one of them.

```
class Foldable t where
    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap f = foldr (mappend . f) mempty
```

Example usage with the Product monoid:

```
product :: (Num n, Foldable t) => t n -> n
product = getProduct . foldMap Product
```

Section 3.8: Checking if a Foldable structure is empty

null returns True if there are no elements a in a foldable structure t a, and False if there is one or more. Structures for which **null** is True have a **length** of 0.

```
ghci> null []
True
ghci> null [14, 29]
False
ghci> null Nothing
True
ghci> null (Right 'a')
```

```
False
ghci> null ('x', 3)
False
```

null定义为等价于：

```
class Foldable t where
    -- …
    null :: t a -> Bool
    null = foldr (\_ _ -> False) True
```

```
False
ghci> null ('x', 3)
False
```

null is defined as being equivalent to:

```
class Foldable t where
    -- ...
    null :: t a -> Bool
    null = foldr (\_ _ -> False) True
```

第四章：可遍历

Traversable类推广了以前称为mapM的函数::Monad m =>(a -> m b) ->[a] -> m [b]，使其能够在除列表之外的结构上使用Applicative效果。

第4.1节：Traversable的定义

```haskell
class (Functor t, Foldable t) => Traversable t where
    {-# MINIMAL traverse | sequenceA #-}

traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
    traverse f = sequenceA . fmap f

sequenceA :: Applicative f => t (f a) -> f (t a)
    sequenceA = traverse id

    mapM :: Monad m => (a -> m b) -> t a -> m (t b)
    mapM = traverse

    sequence :: Monad m => t (m a) -> m (t a)
    sequence = sequenceA
```

Traversable结构 t是元素 a的有限容器，可以通过带有副作用的“访问者”操作进行操作。访问者函数 f :: a -> f b对结构中的每个元素执行副作用，traverse使用Applicative组合这些副作用。另一种看法是，sequenceA表示Traversable结构与Applicative可交换。

第4.2节：反向遍历结构

借助Backwards applicative functor，可以反向运行遍历，该函数将现有的applicative翻转，使组合的效果按相反顺序发生。

```haskell
newtype Backwards f a = Backwards { forwards :: f a }

instance Applicative f => Applicative (Backwards f) where
    pure = Backwards . pure
Backwards ff <*> Backwards fx = Backwards ((\x f -> f x) <$> fx <*> ff)
```

Backwards 可以用于“反向traverse”。当traverse调用的底层应用函子被Backwards翻转时，产生的效果将以相反的顺序发生。

```haskell
newtype Reverse t a = Reverse { getReverse :: t a }

instance Traversable t => Traversable (Reverse t) where
    traverse f = fmap Reverse . forwards . traverse (Backwards . f) . getReverse

ghci> traverse print (Reverse "abc")
'c'
'b'
'a'
```

Reverse newtype 定义在 Data.Functor.Reverse 模块下。

Chapter 4: Traversable

The Traversable class generalises the function formerly known as mapM :: Monad m => (a -> m b) -> [a] -> m [b] to work with Applicative effects over structures other than lists.

Section 4.1: Definition of Traversable

```haskell
class (Functor t, Foldable t) => Traversable t where
    {-# MINIMAL traverse | sequenceA #-}

    traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
    traverse f = sequenceA . fmap f

    sequenceA :: Applicative f => t (f a) -> f (t a)
    sequenceA = traverse id

    mapM :: Monad m => (a -> m b) -> t a -> m (t b)
    mapM = traverse

    sequence :: Monad m => t (m a) -> m (t a)
    sequence = sequenceA
```

Traversable structures t are finitary containers of elements a which can be operated on with an effectful "visitor" operation. The visitor function f :: a -> f b performs a side-effect on each element of the structure and traverse composes those side-effects using Applicative. Another way of looking at it is that sequenceA says Traversable structures commute with Applicatives.

Section 4.2: Traversing a structure in reverse

A traversal can be run in the opposite direction with the help of the Backwards applicative functor, which flips an existing applicative so that composed effects take place in reversed order.

```haskell
newtype Backwards f a = Backwards { forwards :: f a }

instance Applicative f => Applicative (Backwards f) where
    pure = Backwards . pure
    Backwards ff <*> Backwards fx = Backwards ((\x f -> f x) <$> fx <*> ff)
```

Backwards can be put to use in a "reversed traverse". When the underlying applicative of a traverse call is flipped with Backwards, the resulting effect happens in reverse order.

```haskell
newtype Reverse t a = Reverse { getReverse :: t a }

instance Traversable t => Traversable (Reverse t) where
    traverse f = fmap Reverse . forwards . traverse (Backwards . f) . getReverse

ghci> traverse print (Reverse "abc")
'c'
'b'
'a'
```

The Reverse newtype is found under Data.Functor.Reverse.

第4.3节：二叉树的 Traversable 实例

traverse 的实现通常看起来像是在 fmap 的基础上提升到 Applicative 上下文中的实现。

```
数据 Tree a = 叶子 (Leaf)
              | 节点 (Node)  (Tree a) a (Tree a)

instance Traversable Tree where
traverse f Leaf = pure Leaf
traverse f (Node l x r) = Node <$> traverse f l <*> f x <*> traverse f r
```

该实现执行树的中序遍历。

```
ghci> let myTree = Node (Node Leaf 'a' Leaf) 'b' (Node Leaf 'c' Leaf)

--    +--'b'--+
--    |       |
-- +-'a'-+ +-'c'-+
-- |     | |     |
-- *     * *     *

ghci> traverse print myTree
'a'
'b'
'c'
```

DeriveTraversable扩展允许GHC基于类型的结构生成Traversable实例。我们可以通过调整Node构造函数的布局来改变机器生成遍历的顺序。

```
data Inorder a = ILeaf
               | INode (Inorder a) a (Inorder a)  -- 如前所述
               deriving (Functor, Foldable, Traversable)  -- 也使用了DeriveFunctor和
DeriveFoldable

数据 先序 a = 先序叶子
                | PrNode a (Preorder a) (Preorder a)
                deriving (Functor, Foldable, Traversable)

数据 后序 a = 后序叶子
                 | 后序节点 (后序 a) (后序 a) a
                 deriving (Functor, Foldable, Traversable)

-- 来自早期树类型的注入
inorder :: Tree a -> Inorder a
inorder Leaf = ILeaf
中序 (节点 左 x 右) = 中序节点 (中序 左) x (中序 右)

先序 :: 树 a -> 先序 a
先序 叶子 = 先序叶子
先序 (节点 左 x 右) = 先序节点 x (先序 左) (先序 右)

后序 :: 树 a -> 后序 a
后序叶子 = PoLeaf
postorder (Node l x r) = PoNode (postorder l) (postorder r) x

ghci> traverse print (inorder myTree)
'a'
'b'
'c'
ghci> 遍历 打印 (先序 myTree)
```

Section 4.3: An instance of Traversable for a binary tree

Implementations of `traverse` usually look like an implementation of **`fmap`** lifted into an `Applicative` context.

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

instance Traversable Tree where
    traverse f Leaf = pure Leaf
    traverse f (Node l x r) = Node <$> traverse f l <*> f x <*> traverse f r
```

This implementation performs an in-order traversal of the tree.

```
ghci> let myTree = Node (Node Leaf 'a' Leaf) 'b' (Node Leaf 'c' Leaf)

--    +--'b'--+
--    |       |
-- +-'a'-+ +-'c'-+
-- |     | |     |
-- *     * *     *

ghci> traverse print myTree
'a'
'b'
'c'
```

The `DeriveTraversable` extension allows GHC to generate `Traversable` instances based on the structure of the type. We can vary the order of the machine-written traversal by adjusting the layout of the `Node` constructor.

```
data Inorder a = ILeaf
               | INode (Inorder a) a (Inorder a)  -- as before
               deriving (Functor, Foldable, Traversable)  -- also using DeriveFunctor and
DeriveFoldable

data Preorder a = PrLeaf
                | PrNode a (Preorder a) (Preorder a)
                deriving (Functor, Foldable, Traversable)

data Postorder a = PoLeaf
                 | PoNode (Postorder a) (Postorder a) a
                 deriving (Functor, Foldable, Traversable)

-- injections from the earlier Tree type
inorder :: Tree a -> Inorder a
inorder Leaf = ILeaf
inorder (Node l x r) = INode (inorder l) x (inorder r)

preorder :: Tree a -> Preorder a
preorder Leaf = PrLeaf
preorder (Node l x r) = PrNode x (preorder l) (preorder r)

postorder :: Tree a -> Postorder a
postorder Leaf = PoLeaf
postorder (Node l x r) = PoNode (postorder l) (postorder r) x

ghci> traverse print (inorder myTree)
'a'
'b'
'c'
ghci> traverse print (preorder myTree)
```

```
'b'
'a'
'c'
ghci> 遍历 打印 (后序 myTree)
'a'
'c'
'b'
```

第4.4节：作为带内容的形状的可遍历结构

如果一个类型 t 是 Traversable，那么 t a 的值可以分成两部分：“形状”和“内容”：

```
数据 Traversed t a = Traversed { 形状 :: t (), 内容 :: [a] }
```

其中“内容”与使用 Foldable 实例“访问”的内容相同。

从 t a 到 Traversed t a 的转换只需要 Functor 和 Foldable

```
break :: (Functor t, Foldable t) => t a -> Traversed t a
break ta = Traversed (fmap (const ()) ta) (toList ta)
```

但回溯时关键使用了traverse函数

```
import Control.Monad.State

-- 不变式：状态非空
pop :: State [a] a
pop = state $ \(a:as) -> (a, as)

recombine :: Traversable t => Traversed t a -> t a
recombine (Traversed s c) = evalState (traverse (const pop) s) c
```

Traversable定律要求break . recombine 和 recombine . break 都是恒等函数。值得注意的是，这意味着contents中恰好有足够的元素完全填充shape，且没有剩余。

Traversed t本身也是Traversable。traverse的实现通过使用列表的Traversable实例访问元素，然后将惰性形状重新附加到结果上。

```
instance Traversable (Traversed t) where
    traverse f (Traversed s c) = fmap (Traversed s) (traverse f c)
```

第4.5节：为Traversable结构实例化Functor和Foldable

```
import Data.Traversable as Traversable

data MyType a =  -- ...
instance Traversable MyType where
    traverse = -- ...
```

每个Traversable结构都可以使用Data.Traversable中的fmapDefault和foldMapDefault函数被构造为Foldable和Functor。

```
实例 函子 MyType 的定义
    fmap = Traversable.fmapDefault
```

```
'b'
'a'
'c'
ghci> traverse print (postorder myTree)
'a'
'c'
'b'
```

Section 4.4: Traversable structures as shapes with contents

If a type t is Traversable then values of t a can be split into two pieces: their "shape" and their "contents":

```
data Traversed t a = Traversed { shape :: t (), contents :: [a] }
```

where the "contents" are the same as what you'd "visit" using a Foldable instance.

Going one direction, from t a to Traversed t a doesn't require anything but Functor and Foldable

```
break :: (Functor t, Foldable t) => t a -> Traversed t a
break ta = Traversed (fmap (const ()) ta) (toList ta)
```

but going back uses the traverse function crucially

```
import Control.Monad.State

-- invariant: state is non-empty
pop :: State [a] a
pop = state $ \(a:as) -> (a, as)

recombine :: Traversable t => Traversed t a -> t a
recombine (Traversed s c) = evalState (traverse (const pop) s) c
```

The Traversable laws require that break . recombine and recombine . break are both identity. Notably, this means that there are exactly the right number elements in contents to fill shape completely with no left-overs.

Traversed t is Traversable itself. The implementation of traverse works by visiting the elements using the list's instance of Traversable and then reattaching the inert shape to the result.

```
instance Traversable (Traversed t) where
    traverse f (Traversed s c) = fmap (Traversed s) (traverse f c)
```

Section 4.5: Instantiating Functor and Foldable for a Traversable structure

```
import Data.Traversable as Traversable

data MyType a =  -- ...
instance Traversable MyType where
    traverse = -- ...
```

Every Traversable structure can be made a Foldable Functor using the fmapDefault and foldMapDefault functions found in Data.Traversable.

```
instance Functor MyType where
    fmap = Traversable.fmapDefault
```

```
实例 可折叠 MyType 的定义
    foldMap = Traversable.foldMapDefault
```

fmapDefault 是通过在 Identity 应用函子中运行 traverse 来定义的。

```
新类型 Identity a = Identity { runIdentity :: a }

实例 应用函子 Identity 的定义
pure = Identity
Identity f <*> Identity x = Identity (f x)

fmapDefault :: Traversable t => (a -> b) -> t a -> t b
fmapDefault f = runIdentity . traverse (Identity . f)
```

foldMapDefault 是使用 Const 应用函子定义的，该函子在累积一个

幺半群值时忽略其参数。

```
新类型 Const c a = Const { getConst :: c }

实例 幺半群 m => 应用函子 (Const m) 的定义
    pure _ = Const mempty
Const x <*> Const y = Const (x `mappend` y)

foldMapDefault :: (Traversable t, Monoid m) => (a -> m) -> t a -> m
foldMapDefault f = getConst . traverse (Const . f)
```

第4.6节：借助累积参数转换 Traversable 结构

这两个mapAccum函数结合了折叠和映射的操作。

```
--                                              可遍历结构
--                                                                      |
--                                            一个种子值   |
--                                                               |      |
--                                                              |-|   |---|
mapAccumL, mapAccumR :: Traversable t => (a -> b -> (a, c)) -> a -> t b -> (a, t c)--
                         |------------------|               |--------|--
          |                           |--                          一个折叠函数，产生新的映射元素
 'c' 和新的累积器值 'a'          |--
                |--                                                   最终累积器值--
                                                 和映射结构
```

这些函数推广了 fmap，因为它们允许映射的值依赖于折叠过程中之前发生的情况。它们也推广了 foldl/foldr，因为它们不仅将结构映射，还将其归约为一个值。

例如，tails 可以使用 mapAccumR 实现，其对应的 inits 可以使用 mapAccumL 实现。

```
tails, inits :: [a] -> [[a]]
tails = uncurry (:) . mapAccumR (\xs x -> (x:xs, xs)) []
inits = uncurry snoc . mapAccumL (\xs x -> (x `snoc` xs, xs)) []
    where snoc x xs = xs ++ [x]

ghci> tails "abc"
["abc", "bc", "c", ""]
ghci> inits "abc"
```

```
instance Foldable MyType where
    foldMap = Traversable.foldMapDefault
```

fmapDefault is defined by running traverse in the Identity applicative functor.

```
newtype Identity a = Identity { runIdentity :: a }

instance Applicative Identity where
    pure = Identity
    Identity f <*> Identity x = Identity (f x)

fmapDefault :: Traversable t => (a -> b) -> t a -> t b
fmapDefault f = runIdentity . traverse (Identity . f)
```

foldMapDefault is defined using the Const applicative functor, which ignores its parameter while accumulating a monoidal value.

```
newtype Const c a = Const { getConst :: c }

instance Monoid m => Applicative (Const m) where
    pure _ = Const mempty
    Const x <*> Const y = Const (x `mappend` y)

foldMapDefault :: (Traversable t, Monoid m) => (a -> m) -> t a -> m
foldMapDefault f = getConst . traverse (Const . f)
```

Section 4.6: Transforming a Traversable structure with the aid of an accumulating parameter

The two mapAccum functions combine the operations of folding and mapping.

```
--                                                       A Traversable structure
--                                                                   |
--                                                     A seed value  |
--                                                             |     |
--                                                            |-|  |---|
mapAccumL, mapAccumR :: Traversable t => (a -> b -> (a, c)) -> a -> t b -> (a, t c)
--                                       |------------------|              |--------|
--                                                |                                 |
--                          A folding function which produces a new mapped          |
--                            element 'c' and a new accumulator value 'a'           |
--                                                                                  |
--                                                               Final accumulator value
--                                                                and mapped structure
```

These functions generalise **fmap** in that they allow the mapped values to depend on what has happened earlier in the fold. They generalise **foldl**/**foldr** in that they map the structure in place as well as reducing it to a value.

For example, tails can be implemented using mapAccumR and its sister inits can be implemented using mapAccumL.

```
tails, inits :: [a] -> [[a]]
tails = uncurry (:) . mapAccumR (\xs x -> (x:xs, xs)) []
inits = uncurry snoc . mapAccumL (\xs x -> (x `snoc` xs, xs)) []
    where snoc x xs = xs ++ [x]

ghci> tails "abc"
["abc", "bc", "c", ""]
ghci> inits "abc"
```

```
["", "a", "ab", "abc"]
```

mapAccumL 是通过遍历 State 应用函子来实现的。

```
{-# LANGUAGE DeriveFunctor #-}

newtype State s a = State { runState :: s -> (s, a) } deriving Functor
instance Applicative (State s) where
pure x = State $ \s -> (s, x)
    State ff <*> State fx = State $ \s -> let (t, f) = ff s
                                              (u, x) = fx t
                                          in (u, f x)

mapAccumL f z t = runState (traverse (State . flip f) t) z
```

mapAccumR 是通过反向运行 mapAccumL 实现的。

```
mapAccumR f z = fmap getReverse . mapAccumL f z . Reverse
```

第4.7节：转置列表的列表

注意到zip将列表元组转置为元组列表，

```
ghci> uncurry zip ([1,2],[3,4])
[(1,3), (2,4)]
```

以及transpose和sequenceA类型之间的相似性，

```
-- transpose 交换内层列表和外层列表
--           +---+-->--+-+
--           |   |     | |
transpose :: [[a]] -> [[a]]
--            | |     |   |
--            +-+-->--+---+

-- sequenceA 交换内层的 Applicative 和外层的 Traversable
--                                             +------>------+
--                                             |             |
sequenceA :: (Traversable t, Applicative f) => t (f a) -> f (t a)
--                                                |         |
--                                                +--->---+
```

这个想法是利用 [] 的 Traversable 和 Applicative 结构，将 sequenceA 作为一种 n 元的 zip，逐点地将所有内部列表一起压缩。

[] 的默认“优先选择” Applicative 实例不适合我们的用途——我们需要一个“拉链式”的 Applicative。为此我们使用 Control.Applicative 中的 ZipList 新类型。

```
newtype ZipList a = ZipList { getZipList :: [a] }

instance Applicative ZipList where
pure x = ZipList (repeat x)
    ZipList fs <*> ZipList xs = ZipList (zipWith ($) fs xs)
```

现在我们通过在 ZipList Applicative 中遍历，免费获得了 transpose 函数。

```
transpose :: [[a]] -> [[a]]
```

```
["", "a", "ab", "abc"]
```

`mapAccumL` is implemented by traversing in the `State` applicative functor.

```
{-# LANGUAGE DeriveFunctor #-}

newtype State s a = State { runState :: s -> (s, a) } deriving Functor
instance Applicative (State s) where
    pure x = State $ \s -> (s, x)
    State ff <*> State fx = State $ \s -> let (t, f) = ff s
                                              (u, x) = fx t
                                          in (u, f x)

mapAccumL f z t = runState (traverse (State . flip f) t) z
```

`mapAccumR` works by running `mapAccumL` in reverse.

```
mapAccumR f z = fmap getReverse . mapAccumL f z . Reverse
```

Section 4.7: Transposing a list of lists

Noting that **zip** transposes a tuple of lists into a list of tuples,

```
ghci> uncurry zip ([1,2],[3,4])
[(1,3), (2,4)]
```

and the similarity between the types of `transpose` and `sequenceA`,

```
-- transpose exchanges the inner list with the outer list
--           +---+-->--+-+
--           |   |     | |
transpose :: [[a]] -> [[a]]
--            | |     |   |
--            +-+-->--+---+

-- sequenceA exchanges the inner Applicative with the outer Traversable
--                                             +------>------+
--                                             |             |
sequenceA :: (Traversable t, Applicative f) => t (f a) -> f (t a)
--                                                |         |
--                                                +--->---+
```

the idea is to use `[]`'s `Traversable` and `Applicative` structure to deploy `sequenceA` as a sort of *n-ary* ***zip***, zipping together all the inner lists together pointwise.

`[]`'s default "prioritised choice" `Applicative` instance is not appropriate for our use - we need a "zippy" `Applicative`. For this we use the `ZipList` newtype, found in `Control.Applicative`.

```
newtype ZipList a = ZipList { getZipList :: [a] }

instance Applicative ZipList where
    pure x = ZipList (repeat x)
    ZipList fs <*> ZipList xs = ZipList (zipWith ($) fs xs)
```

Now we get `transpose` for free, by traversing in the `ZipList` `Applicative`.

```
transpose :: [[a]] -> [[a]]
transpose = getZipList . traverse ZipList

ghci> let myMatrix = [[1,2,3],[4,5,6],[7,8,9]]
ghci> transpose myMatrix
[[1,4,7],[2,5,8],[3,6,9]]
```

```
transpose = getZipList . traverse ZipList

ghci> let myMatrix = [[1,2,3],[4,5,6],[7,8,9]]
ghci> transpose myMatrix
[[1,4,7],[2,5,8],[3,6,9]]
```

第5章：镜头

Lens 是一个为 Haskell 提供镜头（lenses）、同构（isomorphisms）、折叠（folds）、遍历（traversals）、获取器（getters）和设置器（setters）的库，它暴露了一个统一的接口，用于查询和操作任意结构，类似于 Java 的访问器和修改器概念。

第5.1节：记录的镜头

简单记录

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Lens

data Point = Point {
    _x :: Float,
    _y :: Float
}
makeLenses ''Point
```

创建了镜头 x 和 y 。

```
let p = Point 5.0 6.0
p ^. x     -- 返回 5.0
set x 10 p -- 返回 Point { _x = 10.0, _y = 6.0 }
p & x +~ 1 -- 返回 Point { _x = 6.0, _y = 6.0 }
```

管理具有重复字段名的记录

```
data Person = Person { _personName :: String }
makeFields "Person
```

为Person创建类型类HasName、透镜name，并使Person成为HasName的实例。后续的记录也将被添加到该类中：

```
data Entity = Entity { _entityName :: String }
makeFields "Entity
```

为了使makeFields生效，需要启用 Template Haskell 扩展。从技术上讲，通过其他方式（例如手动）创建这种方式生成的透镜也是完全可能的。

第5.2节：使用透镜操作元组

获取

```
("a", 1) ^. _1 -- 返回 "a"
("a", 1) ^. _2 -- 返回 1
```

设置

```
("a", 1) & _1 .~ "b" -- 返回 ("b", 1)
```

修改

```
("a", 1) & _2 %~ (+1) -- 返回 ("a", 2)
```

两者 遍历

Chapter 5: Lens

Lens is a library for Haskell that provides lenses, isomorphisms, folds, traversals, getters and setters, which exposes a uniform interface for querying and manipulating arbitrary structures, not unlike Java's accessor and mutator concepts.

Section 5.1: Lenses for records

Simple record

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Lens

data Point = Point {
    _x :: Float,
    _y :: Float
}
makeLenses ''Point
```

Lenses x and y are created.

```
let p = Point 5.0 6.0
p ^. x     -- returns 5.0
set x 10 p -- returns Point { _x = 10.0, _y = 6.0 }
p & x +~ 1 -- returns Point { _x = 6.0, _y = 6.0 }
```

Managing records with repeating fields names

```
data Person = Person { _personName :: String }
makeFields ''Person
```

Creates a type class `HasName`, lens `name` for `Person`, and makes `Person` an instance of `HasName`. Subsequent records will be added to the class as well:

```
data Entity = Entity { _entityName :: String }
makeFields ''Entity
```

The Template Haskell extension is required for `makeFields` to work. Technically, it's entirely possible to create the lenses made this way via other means, e.g. by hand.

Section 5.2: Manipulating tuples with Lens

Getting

```
("a", 1) ^. _1 -- returns "a"
("a", 1) ^. _2 -- returns 1
```

Setting

```
("a", 1) & _1 .~ "b" -- returns ("b", 1)
```

Modifying

```
("a", 1) & _2 %~ (+1) -- returns ("a", 2)
```

`both` Traversal

```
(1, 2) & both *~ 2 -- 返回 (2, 4)
```

第5.3节：透镜和棱镜

一个透镜的 a意味着你可以总是在任何 s中找到一个 a。一个棱镜的 a意味着你有时会发现
那个 s实际上就是一个 a，但有时它是别的东西。

更清楚地说，我们有_1 :: 透镜(a, b) a，因为任何元组总是有第一个元素。我们有_Just ::
棱镜(Maybe a) a，因为有时Maybe a实际上是一个包裹在Just中的 a值，但有时是Nothing。

基于这种直觉，一些标准组合子可以相互平行地解释

- view :: 透镜 s a -> (s -> a) “获取” s中的 a
- set :: 透镜 s a -> (a -> s -> s) “设置” s中的 a
- review :: 棱镜 s a -> (a -> s) “实现” 一个 a可能是一个 s
- preview :: 棱镜 s a -> (s -> Maybe a) “尝试” 将一个 s转换成一个 a。

另一种理解方式是，类型为Lens' s a的值表明 s具有与(r,
a)相同的结构，其中的 r未知。另一方面，Prism' s a表明 s具有与Either r a
相同的结构，其中的 r未知。基于此，我们可以用这些知识来编写上面提到的四个函数：

```
-- 不再提供`Lens' s a`，而是我们*知道* `s ~ (r, a)`

view :: (r, a) -> a
view (r, a) = a

set :: a -> (r, a) -> (r, a)
set a (r, _) = (r, a)

-- 不再提供`Prism' s a`，而是我们*知道* `s ~ Either r a`

review :: a -> Either r a
review a = Right a

preview :: Either r a -> Maybe a
preview (Left _) = Nothing
preview (Right a) = Just a
```

第5.4节：有状态的透镜

透镜操作符有一些在有状态上下文中操作的有用变体。它们是通过将

操作符名称中的~替换为=获得的。

```
(+~) :: Num a => ASetter s t a a -> a -> s -> t
(+=) :: (MonadState s m, Num a) => ASetter' s a -> a -> m ()
```

> 注意：有状态的变体不应改变类型，因此它们具有Lens'或Simple Lens'签名。

去除&链式操作

如果需要将镜头操作链式连接，通常看起来像这样：

```
change :: A -> A
```

```
(1, 2) & both *~ 2 -- returns (2, 4)
```

Section 5.3: Lens and Prism

A Lens' s a means that you can *always* find an a within any s. A Prism' s a means that you can *sometimes* find that s actually just *is* a but sometimes it's something else.

To be more clear, we have _1 :: Lens' (a, b) a because any tuple *always* has a first element. We have _Just :: Prism' (Maybe a) a because *sometimes* Maybe a is actually an a value wrapped in Just but *sometimes* it's Nothing.

With this intuition, some standard combinators can be interpreted parallel to one another

- view :: Lens' s a -> (s -> a) "gets" the a out of the s
- set :: Lens' s a -> (a -> s -> s) "sets" the a slot in s
- review :: Prism' s a -> (a -> s) "realizes" that an a could be an s
- preview :: Prism' s a -> (s -> Maybe a) "attempts" to turn an s into an a.

Another way to think about it is that a value of type Lens' s a demonstrates that s has the same structure as (r, a) for some unknown r. On the other hand, Prism' s a demonstrates that s has the same structure as Either r a for some r. We can write those four functions above with this knowledge:

```
-- `Lens' s a` is no longer supplied, instead we just *know* that `s ~ (r, a)`

view :: (r, a) -> a
view (r, a) = a

set :: a -> (r, a) -> (r, a)
set a (r, _) = (r, a)

-- `Prism' s a` is no longer supplied, instead we just *know* that `s ~ Either r a`

review :: a -> Either r a
review a = Right a

preview :: Either r a -> Maybe a
preview (Left _) = Nothing
preview (Right a) = Just a
```

Section 5.4: Stateful Lenses

Lens operators have useful variants that operate in stateful contexts. They are obtained by replacing ~ with = in the operator name.

```
(+~) :: Num a => ASetter s t a a -> a -> s -> t
(+=) :: (MonadState s m, Num a) => ASetter' s a -> a -> m ()
```

> Note: The stateful variants aren't expected to change the type, so they have the Lens' or Simple Lens' signatures.

Getting rid of & chains

If lens-ful operations need to be chained, it often looks like this:

```
change :: A -> A
change a = a & lensA %~ operationA
             & lensB %~ operationB
             & lensC %~ operationC
```

这得益于&的结合律。不过，有状态版本更清晰。

```
change a = flip execState a $ do
    lensA %= operationA
lensB %= operationB
    lensC %= operationC
```

如果lensX实际上是id，整个操作当然可以直接通过modify提升来执行。

带有结构化状态的命令式代码

假设这个示例状态：

```
data Point = Point { _x :: Float, _y :: Float }
data Entity = Entity { _position :: Point, _direction :: Float }
data World = World { _entities :: [Entity] }

makeLenses "Point
makeLenses "Entity
makeLenses "World
```

我们可以编写类似经典命令式语言的代码，同时仍然能够利用Haskell的优势：

```
updateWorld :: MonadState World m => m ()
updateWorld = do
    -- 移动第一个实体
entities . ix 0 . position . x += 1

    -- 对所有实体执行某些操作
entities . traversed . position %= \p -> p `pointAdd` ...

    -- 或者只对部分实体执行操作
entities . traversed . filtered (\e -> e ^. position.x > 100) %= ...
```

第5.5节：镜头组装

如果你有一个 f :: Lens' a b 和一个 g :: Lens' b c，那么 f . g 是一个 Lens' a c，通过先应用 f 然后应用

g 得到。值得注意的是：

- 透镜（Lenses）组合如同函数（实际上它们就是函数）如果你
- 把 Lens 的 view 功能看作数据"从左到右"流动，这可能与你对函数组合的常规直觉相反。但如果你把 . 符号看作面向对象语言中的方法调用，这种感觉就很自然了。

不仅仅是将 Lens 与 Lens 组合，(.) 还可以用来组合几乎任何"Lens-like"类型。结果的类型有时不容易看清，因为类型变得更复杂，但你可以使用 the lens chart 来弄清楚。组合 x . y 的类型是该图中 x 和 y 类型的最小上界。

第5.6节：不使用Template Haskell编写透镜

为了揭开Template Haskell的神秘面纱，假设你有

```
change a = a & lensA %~ operationA
             & lensB %~ operationB
             & lensC %~ operationC
```

This works thanks to the associativity of &. The stateful version is clearer, though.

```
change a = flip execState a $ do
    lensA %= operationA
    lensB %= operationB
    lensC %= operationC
```

If lensX is actually id, the whole operation can of course be executed directly by just lifting it with modify.

Imperative code with structured state

Assuming this example state:

```
data Point = Point { _x :: Float, _y :: Float }
data Entity = Entity { _position :: Point, _direction :: Float }
data World = World { _entities :: [Entity] }

makeLenses ''Point
makeLenses ''Entity
makeLenses ''World
```

We can write code that resembles classic imperative languages, while still allowing us to use benefits of Haskell:

```
updateWorld :: MonadState World m => m ()
updateWorld = do
    -- move the first entity
    entities . ix 0 . position . x += 1

    -- do some operation on all of them
    entities . traversed . position %= \p -> p `pointAdd` ...

    -- or only on a subset
    entities . traversed . filtered (\e -> e ^. position.x > 100) %= ...
```

Section 5.5: Lenses compose

If you have a f :: Lens' a b and a g :: Lens' b c then f . g is a Lens' a c gotten by following f first and then g. Notably:

- Lenses compose as functions (really they just *are* functions)
- If you think of the view functionality of Lens, it seems like data flows "left to right"—this might feel backwards to your normal intuition for function composition. On the other hand, it ought to feel natural if you think of .-notation like how it happens in OO languages.

More than just composing Lens with Lens, (.) can be used to compose nearly any "Lens-like" type together. It's not always easy to see what the result is since the type becomes tougher to follow, but you can use the lens chart to figure it out. The composition x . y has the type of the least-upper-bound of the types of both x and y in that chart.

Section 5.6: Writing a lens without Template Haskell

To demystify Template Haskell, suppose you have

```
数据类型 Example a = Example { _foo :: Int, _bar :: a }
```

那么

```
makeLenses 'Example
```

会生成（或多或少）

```
foo :: Lens' (Example a) Int
bar :: Lens (Example a) (Example b) a b
```

不过，这里没有什么特别神奇的东西。你可以自己写这些：

```
foo :: Lens' (Example a) Int
-- :: Functor f => (Int -> f Int) -> (Example a -> f (Example a))    ;; 展开别名
foo wrap (Example foo bar) = fmap (\newFoo -> Example newFoo bar) (wrap foo)

bar :: Lens (Example a) (Example b) a b
-- :: Functor f => (a -> f b) -> (Example a -> f (Example b))    ;; 展开别名
bar wrap (Example foo bar) = fmap (\newBar -> Example foo newBar) (wrap bar)
```

本质上，你想用wrap函数“访问”你的镜头的“焦点”，然后重建“整个”类型。

第5.7节：使用makeFields的字段

（此示例摘自这个StackOverflow回答）

假设你有许多不同的数据类型，它们都应该有一个同名的镜头，在本例中容量。 makeFields 切片将创建一个类来实现这一点，且不会产生命名空间冲突。

```
{-# LANGUAGE FunctionalDependencies
           , MultiParamTypeClasses
           , TemplateHaskell
  #-}

模块 Foo
所在位置

导入 Control.Lens

data Foo
  = Foo { fooCapacity :: Int }派生 (
  Eq, Show)
$(makeFields ''Foo)

数据 Bar
  = Bar { barCapacity :: Double }派生 (
  Eq, Show)
$(makeFields ''Bar)
```

然后在 ghci 中：

```
*Foo
λ 令 f = Foo 3
|     b = Bar 7
|
b :: Bar
```

```
data Example a = Example { _foo :: Int, _bar :: a }
```

then

```
makeLenses 'Example
```

produces (more or less)

```
foo :: Lens' (Example a) Int
bar :: Lens (Example a) (Example b) a b
```

There's nothing particularly magical going on, though. You can write these yourself:

```
foo :: Lens' (Example a) Int
-- :: Functor f => (Int -> f Int) -> (Example a -> f (Example a))    ;; expand the alias
foo wrap (Example foo bar) = fmap (\newFoo -> Example newFoo bar) (wrap foo)

bar :: Lens (Example a) (Example b) a b
-- :: Functor f => (a -> f b) -> (Example a -> f (Example b))    ;; expand the alias
bar wrap (Example foo bar) = fmap (\newBar -> Example foo newBar) (wrap bar)
```

Essentially, you want to "visit" your lens' "focus" with the `wrap` function and then rebuild the "entire" type.

Section 5.7: Fields with makeFields

(This example copied from this StackOverflow answer)

Let's say you have a number of different data types that all ought to have a lens with the same name, in this case `capacity`. The `makeFields` slice will create a class that accomplish this without namespace conflicts.

```
{-# LANGUAGE FunctionalDependencies
           , MultiParamTypeClasses
           , TemplateHaskell
  #-}

module Foo
where

import Control.Lens

data Foo
  = Foo { fooCapacity :: Int }
  deriving (Eq, Show)
$(makeFields ''Foo)

data Bar
  = Bar { barCapacity :: Double }
  deriving (Eq, Show)
$(makeFields ''Bar)
```

Then in ghci:

```
*Foo
λ let f = Foo 3
|     b = Bar 7
|
b :: Bar
```

```
f :: Foo

*Foo
λ fooCapacity f
3
it :: Int

*Foo
λ barCapacity b
7.0
it  :: 双精度浮点数

*Foo
λ f ^. 容量
3
it :: Int

*Foo
λ b ^. 容量
7.0
it  :: 双精度浮点数

λ :信息 HasCapacity
类 HasCapacity s a | s -> a 其中
  capacity :: Lens' s a
    -- 定义于 Foo.hs:14:3
实例 HasCapacity Foo Int -- 定义于 Foo.hs:14:3
实例 HasCapacity Bar Double -- 定义于 Foo.hs:19:3
```

所以它实际上做的是声明一个类 HasCapacity s a，其中 capacity 是一个从 s 到 a 的 Lens'（a 一旦知道 s 就固定了）。它通过去掉字段名中的（小写化的）数据类型名称来确定“capacity”这个名字；我觉得不必在字段名或镜头名上使用下划线很令人愉快，因为有时记录语法实际上就是你想要的。你可以使用 makeFieldsWith 和各种 lensRules 来有不同的选项计算镜头名称。

如果有帮助的话，使用 ghci -ddump-splices Foo.hs：

```
[1 of 1] 编译 Foo              ( Foo.hs，解释执行 )
Foo.hs:14:3-18: 拼接声明
    makeFields ''Foo
  ======>
    类 HasCapacity s a | s -> a 其中
      capacity :: Lens' s a
    实例 HasCapacity Foo Int 其中
      {-# INLINE capacity #-}
capacity = iso (\ (Foo x_a7fG) -> x_a7fG) Foo
Foo.hs:19:3-18: 拼接声明
    makeFields ''Bar
  ======>
    instance HasCapacity Bar Double where
      {-# INLINE capacity #-}
capacity = iso (\ (Bar x_a7ne) -> x_a7ne) Bar
好的，模块已加载: Foo。
```

所以第一个拼接生成了类 HasCapacity 并为 Foo 添加了实例；第二个使用了已有的类并为 Bar 创建了实例。

如果你从另一个模块导入了 HasCapacity 类，这也同样有效；makeFields 可以为已有类添加更多实例，并将你的类型分布在多个模块中。但如果你在另一个模块中再次使用它

```
f :: Foo

*Foo
λ fooCapacity f
3
it :: Int

*Foo
λ barCapacity b
7.0
it :: Double

*Foo
λ f ^. capacity
3
it :: Int

*Foo
λ b ^. capacity
7.0
it :: Double

λ :info HasCapacity
class HasCapacity s a | s -> a where
  capacity :: Lens' s a
    -- Defined at Foo.hs:14:3
instance HasCapacity Foo Int -- Defined at Foo.hs:14:3
instance HasCapacity Bar Double -- Defined at Foo.hs:19:3
```

So what it's actually done is declared a class `HasCapacity s a`, where capacity is a `Lens'` from s to a (a is fixed once s is known). It figured out the name "capacity" by stripping off the (lowercased) name of the data type from the field; I find it pleasant not to have to use an underscore on either the field name or the lens name, since sometimes record syntax is actually what you want. You can use makeFieldsWith and the various lensRules to have some different options for calculating the lens names.

In case it helps, using ghci -ddump-splices Foo.hs:

```
[1 of 1] Compiling Foo              ( Foo.hs, interpreted )
Foo.hs:14:3-18: Splicing declarations
    makeFields ''Foo
  ======>
    class HasCapacity s a | s -> a where
      capacity :: Lens' s a
    instance HasCapacity Foo Int where
      {-# INLINE capacity #-}
      capacity = iso (\ (Foo x_a7fG) -> x_a7fG) Foo
Foo.hs:19:3-18: Splicing declarations
    makeFields ''Bar
  ======>
    instance HasCapacity Bar Double where
      {-# INLINE capacity #-}
      capacity = iso (\ (Bar x_a7ne) -> x_a7ne) Bar
Ok, modules loaded: Foo.
```

So the first splice made the class `HasCapcity` and added an instance for Foo; the second used the existing class and made an instance for Bar.

This also works if you import the `HasCapcity` class from another module; `makeFields` can add more instances to the existing class and spread your types out across multiple modules. But if you use it again in another module

而你没有导入该类，它将创建一个新的类（同名），你将拥有两个不兼容的重载 capacity 透镜。

第5.8节：Classy 透镜

除了用于生成 Lens 的标准函数 makeLenses 外，Control.Lens.TH 还提供了 makeClassy函数。 makeClassy 的类型相同，工作方式基本与 makeLenses 一致，但有一个关键区别。除了生成标准的透镜和遍历器外，如果类型没有参数，它还会创建一个类，描述所有拥有该类型作为字段的数据类型。例如

```haskell
data Foo = Foo { _fooX, _fooY :: Int }
  makeClassy ''Foo
```

将创建

```haskell
class HasFoo t where
foo :: Simple Lens t Foo

instance HasFoo Foo where foo = id

fooX, fooY :: HasFoo t => Simple Lens t Int
```

第5.9节：遍历（Traversals）

遍历（Traversal）表示 s 中有0个或多个 a。

```
toListOf :: Traversal' s a -> (s -> [a])
```

任何类型 t，只要是 Traversable，自动拥有 traverse :: Traversal (t a) a。

我们可以使用 Traversal 来设置或映射所有这些 a值

```
> set traverse 1 [1..10]
[1,1,1,1,1,1,1,1,1,1]

> over traverse (+1) [1..10]
[2,3,4,5,6,7,8,9,10,11]
```

一个 f :: Lens' s a 表示在 s 中恰好有一个 a。一个 g :: Prism' a b 表示在 a 中有0个或1个 b。组合 f . g 得到一个 Traversal's b，因为先执行 f 然后执行 g 展示了在

where you haven't imported the class, it'll make a new class (with the same name), and you'll have two separate overloaded capacity lenses that are not compatible.

Section 5.8: Classy Lenses

In addition to the standard `makeLenses` function for generating `Lenses`, `Control.Lens.TH` also offers the `makeClassy` function. `makeClassy` has the same type and works in essentially the same way as `makeLenses`, with one key difference. In addition to generating the standard lenses and traversals, if the type has no arguments, it will also create a class describing all the datatypes which possess the type as a field. For example

```haskell
data Foo = Foo { _fooX, _fooY :: Int }
  makeClassy ''Foo
```

will create

```haskell
class HasFoo t where
   foo :: Simple Lens t Foo

instance HasFoo Foo where foo = id

fooX, fooY :: HasFoo t => Simple Lens t Int
```

Section 5.9: Traversals

A `Traversal' s a` shows that s has 0-to-many as inside of it.

```
toListOf :: Traversal' s a -> (s -> [a])
```

Any type t which is `Traversable` automatically has that `traverse :: Traversal (t a) a`.

We can use a `Traversal` to set or map over all of these a values

```
> set traverse 1 [1..10]
[1,1,1,1,1,1,1,1,1,1]

> over traverse (+1) [1..10]
[2,3,4,5,6,7,8,9,10,11]
```

A `f :: Lens' s a` says there's exactly one a inside of s. A `g :: Prism' a b` says there are either 0 or 1 bs in a. Composing `f . g` gives us a `Traversal' s b` because following f and then g shows how there there are 0-to-1 bs in s.

第6章：QuickCheck

第6.1节：声明属性

最简单的情况下，property 是一个返回 Bool 的函数。

```
prop_reverseDoesNotChangeLength xs = length (reverse xs) == length xs
```

属性声明了程序的高级不变量。QuickCheck 测试运行器将使用
100 个随机输入来评估该函数，并检查结果是否始终为 True。

按照惯例，作为属性的函数名称以 prop_ 开头。

第6.2节：为自定义类型随机生成数据

Arbitrary 类用于可以被 QuickCheck 随机生成的类型。

Arbitrary 的最小实现是 arbitrary 方法，它在 Gen 单子中运行以生成一个
随机值。

以下是非空列表数据类型的一个任意实例。

```
import Test.QuickCheck.Arbitrary (Arbitrary(..))
import Test.QuickCheck.Gen (oneof)
import Control.Applicative ((<$>), (<*>))

data NonEmpty a = End a | Cons a (NonEmpty a)

instance Arbitrary a => Arbitrary (NonEmpty a) where
    arbitrary = oneof [  -- 从列表中随机选择一个情况
        End <$> arbitrary,  -- 调用 a 类型的 Arbitrary 实例
Cons <$>
arbitrary <*>  -- 调用 a 类型的 Arbitrary 实例
            arbitrary  -- 递归调用 NonEmpty 的 Arbitrary 实例
        ]
```

第6.3节：使用蕴含（==>）来检查带有前提条件的性质

```
prop_evenNumberPlusOneIsOdd :: Integer -> Property
prop_evenNumberPlusOneIsOdd x = even x ==> odd (x + 1)
```

如果你想检查在前提条件成立的情况下性质是否成立，可以使用该

==>

运算符。注意，如果任意输入满足前提条件的概率非常低，QuickCheck 可能会提前放弃。

```
prop_overlySpecific x y = x == 0 ==> x * y == 0

ghci> quickCheck prop_overlySpecific
*** 放弃了！ 仅通过了 31 个测试。
```

第6.4节：检查单个属性

quickCheck 函数在100个随机输入上测试一个属性。

Chapter 6: QuickCheck

Section 6.1: Declaring a property

At its simplest, a *property* is a function which returns a **Bool**.

```
prop_reverseDoesNotChangeLength xs = length (reverse xs) == length xs
```

A property declares a high-level invariant of a program. The QuickCheck test runner will evaluate the function with 100 random inputs and check that the result is always True.

By convention, functions that are properties have names which start with prop_.

Section 6.2: Randomly generating data for custom types

The Arbitrary class is for types that can be randomly generated by QuickCheck.

The minimal implementation of Arbitrary is the arbitrary method, which runs in the Gen monad to produce a random value.

Here is an instance of Arbitrary for the following datatype of non-empty lists.

```
import Test.QuickCheck.Arbitrary (Arbitrary(..))
import Test.QuickCheck.Gen (oneof)
import Control.Applicative ((<$>), (<*>))

data NonEmpty a = End a | Cons a (NonEmpty a)

instance Arbitrary a => Arbitrary (NonEmpty a) where
    arbitrary = oneof [  -- randomly select one of the cases from the list
        End <$> arbitrary,  -- call a's instance of Arbitrary
        Cons <$>
            arbitrary <*>  -- call a's instance of Arbitrary
            arbitrary  -- recursively call NonEmpty's instance of Arbitrary
        ]
```

Section 6.3: Using implication (==>) to check properties with preconditions

```
prop_evenNumberPlusOneIsOdd :: Integer -> Property
prop_evenNumberPlusOneIsOdd x = even x ==> odd (x + 1)
```

If you want to check that a property holds given that a precondition holds, you can use the

==>

operator. Note that if it's very unlikely for arbitrary inputs to match the precondition, QuickCheck can give up early.

```
prop_overlySpecific x y = x == 0 ==> x * y == 0

ghci> quickCheck prop_overlySpecific
*** Gave up! Passed only 31 tests.
```

Section 6.4: Checking a single property

The quickCheck function tests a property on 100 random inputs.

```
ghci> quickCheck prop_reverseDoesNotChangeLength
+++ OK，通过了 100 个测试。
```

如果某个输入导致属性失败，quickCheck 会打印出反例。

```
prop_reverseIsAlwaysEmpty xs = reverse xs == []  -- 对所有 xs 显然不成立

ghci> quickCheck prop_reverseIsAlwaysEmpty
*** 失败！ 可反驳的 (经过 2 次测试):
[()]
```

第6.5节：检查文件中的所有属性

quickCheckAll 是一个 Template Haskell 辅助工具，它会查找当前文件中所有以prop_开头的定义并对其进行测试。

```
{-# LANGUAGE TemplateHaskell #-}

import Test.QuickCheck (quickCheckAll)
import Data.List (sort)

幂等 :: Eq a => (a -> a) -> a -> Bool
幂等 f x = f (f x) == f x

prop_sortIdempotent = 幂等 sort

-- 不以 prop_ 开头，不会被测试运行器识别
排序不改变长度 xs = 长度 (排序 xs) == 长度 xs


返回 []
主函数 = $quickCheckAll
```

注意 return [] 这一行是必须的。它使得该行以上的定义对 Template Haskell 可见。

```
$ runhaskell QuickCheckAllExample.hs
=== prop_sortIdempotent 来自 tree.hs:7 ===
+++ OK，100 次测试通过。
```

第6.6节：限制测试数据的大小

使用quickcheck测试具有较差渐近复杂度的函数可能很困难，因为随机输入通常没有大小限制。通过对输入大小添加上限，我们仍然可以测试这些耗时的函数。

```
import Data.List(permutations)
import Test.QuickCheck

longRunningFunction :: [a] -> Int
longRunningFunction xs = length (permutations xs)

factorial :: Integral a => a -> a
factorial n = product [1..n]

prop_numberOfPermutations xs =
longRunningFunction xs == factorial (length xs)

ghci> quickCheckWith (stdArgs { maxSize = 10}) prop_numberOfPermutations
```

```
ghci> quickCheck prop_reverseDoesNotChangeLength
+++ OK, passed 100 tests.
```

If a property fails for some input, quickCheck prints out a counterexample.

```
prop_reverseIsAlwaysEmpty xs = reverse xs == []  -- plainly not true for all xs

ghci> quickCheck prop_reverseIsAlwaysEmpty
*** Failed! Falsifiable (after 2 tests):
[()]
```

Section 6.5: Checking all the properties in a file

quickCheckAll is a Template Haskell helper which finds all the definitions in the current file whose name begins with prop_ and tests them.

```
{-# LANGUAGE TemplateHaskell #-}

import Test.QuickCheck (quickCheckAll)
import Data.List (sort)

idempotent :: Eq a => (a -> a) -> a -> Bool
idempotent f x = f (f x) == f x

prop_sortIdempotent = idempotent sort

-- does not begin with prop_, will not be picked up by the test runner
sortDoesNotChangeLength xs = length (sort xs) == length xs


return []
main = $quickCheckAll
```

Note that the return [] line is required. It makes definitions textually above that line visible to Template Haskell.

```
$ runhaskell QuickCheckAllExample.hs
=== prop_sortIdempotent from tree.hs:7 ===
+++ OK, passed 100 tests.
```

Section 6.6: Limiting the size of test data

It can be difficult to test functions with poor asymptotic complexity using quickcheck as the random inputs are not usually size bounded. By adding an upper bound on the size of the input we can still test these expensive functions.

```
import Data.List(permutations)
import Test.QuickCheck

longRunningFunction :: [a] -> Int
longRunningFunction xs = length (permutations xs)

factorial :: Integral a => a -> a
factorial n = product [1..n]

prop_numberOfPermutations xs =
    longRunningFunction xs == factorial (length xs)

ghci> quickCheckWith (stdArgs { maxSize = 10}) prop_numberOfPermutations
```

通过使用带有修改版stdArgs的quickCheckWith，我们可以将输入的大小限制为最多10。在这种情况下，由于我们生成的是列表，这意味着我们生成的列表最大长度为10。对于这些较短的列表，我们的permutations函数运行时间不会太长，但我们仍然可以相当有信心我们的定义是正确的。

By using `quickCheckWith` with a modified version of `stdArgs` we can limit the size of the inputs to be at most 10. In this case, as we are generating lists, this means we generate lists of up to size 10. Our permutations function doesn't take too long to run for these short lists but we can still be reasonably confident that our definition is correct.

第7章：常用GHC语言扩展

第7.1节：RankNTypes

想象以下情况：

```
foo :: Show a => (a -> String) -> String -> Int -> IO ()
foo show' string int = do
   putStrLn (show' string)
   putStrLn (show' int)
```

这里，我们想传入一个将值转换为字符串的函数，将该函数分别应用于字符串参数和整数参数，并打印它们。
在我看来，这没有理由失败！我们有一个函数可以作用于我们传入的两种类型的参数。

不幸的是，这样无法通过类型检查！GHC 会根据函数体中该类型的第一次出现推断 a 类型。
也就是说，一旦我们遇到：

```
putStrLn (show' string)
```

GHC 会推断 `show' :: String -> String`，因为 `string` 是一个 `String`。它在尝试
`show' int` 时会出错。

RankNTypes 允许你将类型签名写成如下形式，对所有满足 show' 类型的函数进行量化：

```
foo :: (forall a. Show a => (a -> String)) -> String -> Int -> IO ()
```

这是二级多态性：我们断言show'函数必须适用于所有 a类型在我们的函数中，之前的实现现在可以工作了。

RankNTypes扩展允许在类型签名中任意嵌套forall ...块。换句话说，它允许N级多态性。

第7.2节：OverloadedStrings

通常，Haskell中的字符串字面量类型为String（这是[Char]的类型别名）。虽然这对较小的教学程序没有问题，但实际应用通常需要更高效的存储方式，如Text或
ByteString。

OverloadedStrings只是将字面量的类型更改为

```
"test" :: Data.String.IsString a => a
```

允许它们直接传递给期望此类类型的函数。许多库为它们的类字符串类型实现了此接口，包括Data.Text和Data.ByteString，这两者都在时间和空间上相较于[Char]提供了某些优势。

还有一些OverloadedStrings的独特用法，比如Postgresql-simple库，它允许用双引号像普通字符串一样编写SQL查询，但提供了防止不当拼接的保护，这种拼接是SQL注入攻击的臭名昭著来源。

Chapter 7: Common GHC Language Extensions

Section 7.1: RankNTypes

Imagine the following situation:

```
foo :: Show a => (a -> String) -> String -> Int -> IO ()
foo show' string int = do
   putStrLn (show' string)
   putStrLn (show' int)
```

Here, we want to pass in a function that converts a value into a String, apply that function to both a string parameter and and int parameter and print them both. In my mind, there is no reason this should fail! We have a function that works on both types of the parameters we're passing in.

Unfortunately, this won't type check! GHC infers the a type based off of its first occurrence in the function body. That is, as soon as we hit:

```
putStrLn (show' string)
```

GHC will infer that `show' :: String -> String`, since `string` is a `String`. It will proceed to blow up while trying to `show' int`.

`RankNTypes` lets you instead write the type signature as follows, quantifying over all functions that satisfy the `show'` type:

```
foo :: (forall a. Show a => (a -> String)) -> String -> Int -> IO ()
```

This is rank 2 polymorphism: We are asserting that the `show'` function must work for all as *within* our function, and the previous implementation now works.

The `RankNTypes` extension allows arbitrary nesting of `forall ...` blocks in type signatures. In other words, it allows rank N polymorphism.

Section 7.2: OverloadedStrings

Normally, string literals in Haskell have a type of `String` (which is a type alias for `[Char]`). While this isn't a problem for smaller, educational programs, real-world applications often require more efficient storage such as `Text` or `ByteString`.

`OverloadedStrings` simply changes the type of literals to

```
"test" :: Data.String.IsString a => a
```

Allowing them to be directly passed to functions expecting such a type. Many libraries implement this interface for their string-like types including Data.Text and Data.ByteString which both provide certain time and space advantages over `[Char]`.

There are also some unique uses of `OverloadedStrings` like those from the Postgresql-simple library which allows SQL queries to be written in double quotes like a normal string, but provides protections against improper concatenation, a notorious source of SQL injection attacks.

要创建IsString类的实例，需要实现fromString函数。示例†：

```
data Foo = A | B | 其他 字符串 派生 Show

instance IsString Foo where
fromString "A" = A
fromString "B" = B
fromString xs = Other xs

tests :: [ Foo ]
tests = [ "A", "B", "Testing" ]
```

† 本示例由 Lyndon Maydwell（GitHub 上的 sordina）提供，见 here。

第7.3节：二进制字面量

标准 Haskell 允许你以十进制（无前缀）、十六进制（以 0x 或 0X 开头）和八进制（以 0o 或 0O 开头）书写整数字面量。 BinaryLiterals 扩展增加了二进制（以 0b 或 0B 开头）的选项。

```
0b1111 == 15     -- 计算结果为：True
```

第7.4节：存在量化

这是一种类型系统扩展，允许存在量化的类型，换句话说，就是具有仅在运行时实例化的类型变量†。

存在类型的值类似于面向对象语言中的抽象基类引用：你不知道它包含的确切类型，但你可以限制类型的class。

```
data S = forall a. Show a => S a
```

或者等价地，使用 GADT 语法：

```
{-# LANGUAGE GADTs #-}
data S where
   S :: Show a => a -> S
```

存在类型为类似几乎异构容器打开了大门：如上所述，实际上在一个 S 值中可以有各种类型，但它们都可以被 show，因此你也可以这样做

```
instance Show S where
     show (S a) = show a    -- 我们依赖于上面 (Show a)
```

现在我们可以创建这样一组对象：

```
ss = [S 5, S "test", S 3.0]
```

这也允许我们使用多态行为：

```
mapM_ print ss
```

存在量词（Existentials）可以非常强大，但请注意，它们在Haskell中实际上并不经常需要。在示例中

To create a instance of the IsString class you need to impliment the fromString function. Example†:

```
data Foo = A | B | Other String deriving Show

instance IsString Foo where
  fromString "A" = A
  fromString "B" = B
  fromString xs  = Other xs

tests :: [ Foo ]
tests = [ "A", "B", "Testing" ]
```

† This example courtesy of Lyndon Maydwell (sordina on GitHub) found here.

Section 7.3: BinaryLiterals

Standard Haskell allows you to write integer literals in decimal (without any prefix), hexadecimal (preceded by 0x or 0X), and octal (preceded by 0o or 0O). The BinaryLiterals extension adds the option of binary (preceded by 0b or 0B).

```
0b1111 == 15     -- evaluates to: True
```

Section 7.4: ExistentialQuantification

This is a type system extension that allows types that are existentially quantified, or, in other words, have type variables that only get instantiated at runtime†.

A value of existential type is similar to an abstract-base-class reference in OO languages: you don't know the exact type in contains, but you can constrain the *class* of types.

```
data S = forall a. Show a => S a
```

or equivalently, with GADT syntax:

```
{-# LANGUAGE GADTs #-}
data S where
   S :: Show a => a -> S
```

Existential types open the door to things like almost-heterogenous containers: as said above, there can actually be various types in an S value, but all of them can be **show**n, hence you can also do

```
instance Show S where
    show (S a) = show a    -- we rely on (Show a) from the above
```

Now we can create a collection of such objects:

```
ss = [S 5, S "test", S 3.0]
```

Which also allows us to use the polymorphic behaviour:

```
mapM_ print ss
```

Existentials can be very powerful, but note that they are actually not necessary very often in Haskell. In the example

如上所述，您实际上能用Show实例做的就是显示（显然！）这些值，即创建一个字符串表示。因此，整个S类型包含的信息量正好等同于显示它时得到的字符串。因此，通常最好直接存储该字符串，尤其是因为Haskell是惰性的，因此该字符串最初也只是一个未求值的延迟计算（thunk）。

另一方面，存在类型会引发一些独特的问题。例如，类型信息在存在类型中被“隐藏”的方式。如果你对一个S值进行模式匹配，你将能够在作用域内使用其包含的类型（更准确地说，是其
显示实例），但这些信息永远无法超出其作用域，因此其作用域变成了一个有点像“秘密社团”的存在：编译器不允许任何内容逃出作用域，除非其类型已经从外部已知的值。
这可能导致奇怪的错误比如无法匹配类型‘a0’与‘()’ ‘a0’是不可触及的。

†与普通的参数多态性相比，后者通常在编译时解决（允许完全的类型擦除）。

存在类型不同于Rank-N类型–这些扩展大致上是彼此的对偶：要实际使用存在类型的值，你需要一个（可能受限的）多态函数，比如示例中的show。多态函数是普遍量化的，即它适用于给定类别中的任何类型，而存在量化意味着它适用于某个事先未知的特定类型。如果你有一个多态函数，那就足够了，但要将多态函数作为参数传递，则需要{-#

语言 Rank2Types #-}:

```
genShowSs :: (∀ x . Show x => x -> String) -> [S] -> [String]
genShowSs f = map (\(S a) -> f a)
```

第7.5节：LambdaCase

一种语法扩展，允许你用 \case 代替 \arg -> case arg of。

考虑以下函数定义：

```
dayOfTheWeek :: Int -> String
dayOfTheWeek 0 = "Sunday"
dayOfTheWeek 1 = "Monday"
dayOfTheWeek 2 = "Tuesday"
dayOfTheWeek 3 = "Wednesday"
dayOfTheWeek 4 = "Thursday"
dayOfTheWeek 5 = "Friday"
dayOfTheWeek 6 = "Saturday"
```

如果你想避免重复函数名，你可能会写成这样：

```
dayOfTheWeek :: Int -> String
dayOfTheWeek i = case i of
    0 -> "Sunday"
    1 -> "Monday"
    2 -> "Tuesday"
    3 -> “星期三”
    4 -> "星期四"
    5 -> "星期五"
    6 -> "星期六"
```

above, all you can actually do with the Show instance is show (duh!) the values, i.e. create a string representation. The entire S type therefore contains exactly as much information as the string you get when showing it. Therefore, it is usually better to simply store that string right away, especially since Haskell is lazy and therefore the string will at first only be an unevaluated thunk anyway.

On the other hand, existentials cause some unique problems. For instance, the way the type information is “hidden” in an existential. If you pattern-match on an S value, you will have the contained type in scope (more precisely, its Show instance), but this information can never escape its scope, which therefore becomes a bit of a “secret society”: the compiler doesn't let *anything* escape the scope except values whose type is already known from the outside. This can lead to strange errors like `Couldn't match type ‘a0’ with ‘()’ ‘a0’ is untouchable`.

†Contrast this with ordinary parametric polymorphism, which is generally resolved at compile time (allowing full type erasure).

Existential types are different from Rank-N types – these extensions are, roughly speaking, dual to each other: to actually use values of an existential type, you need a (possibly constrained-) polymorphic function, like **show** in the example. A polymorphic function is *universally* quantified, i.e. it works *for any* type in a given class, whereas existential quantification means it works *for some* particular type which is a priori unknown. If you have a polymorphic function, that's sufficient, however to pass polymorphic functions as such as arguments, you need `{-# LANGUAGE Rank2Types #-}`:

```
genShowSs :: (∀ x . Show x => x -> String) -> [S] -> [String]
genShowSs f = map (\(S a) -> f a)
```

Section 7.5: LambdaCase

A syntactic extension that lets you write `\case` in place of `\arg -> case arg of`.

Consider the following function definition:

```
dayOfTheWeek :: Int -> String
dayOfTheWeek 0 = "Sunday"
dayOfTheWeek 1 = "Monday"
dayOfTheWeek 2 = "Tuesday"
dayOfTheWeek 3 = "Wednesday"
dayOfTheWeek 4 = "Thursday"
dayOfTheWeek 5 = "Friday"
dayOfTheWeek 6 = "Saturday"
```

If you want to avoid repeating the function name, you might write something like:

```
dayOfTheWeek :: Int -> String
dayOfTheWeek i = case i of
    0 -> "Sunday"
    1 -> "Monday"
    2 -> "Tuesday"
    3 -> "Wednesday"
    4 -> "Thursday"
    5 -> "Friday"
    6 -> "Saturday"
```

使用 LambdaCase 扩展，你可以将其写成一个函数表达式，而无需为参数命名：

```
{-# LANGUAGE LambdaCase #-}

dayOfTheWeek :: Int -> String
dayOfTheWeek = \case
    0 -> "Sunday"
    1 -> "Monday"
    2 -> "Tuesday"
    3 -> “星期三”
    4 -> "星期四"
    5 -> "星期五"
    6 -> "星期六"
```

第7.6节：函数依赖（FunctionalDependencies）

如果你有一个多参数类型类，参数为 a、b、c 和 x，该扩展允许你表达类型 x 可以由 a、b 和 c 唯一确定：

```
class SomeClass a b c x | a b c -> x where ...
```

在声明该类的实例时，会检查其与所有其他实例以确保函数依赖成立，也就是说，不存在具有相同 a b c 但不同 x 的其他实例。

您可以在逗号分隔的列表中指定多个依赖项：

```
class OtherClass a b c d | a b -> c d, a d -> b where ...
```

例如，在MTL中我们可以看到：

```
class MonadReader r m| m -> r where ...
instance MonadReader r ((->) r) where ...
```

现在，如果你有一个类型为 MonadReader a ((->) Foo) => a 的值，编译器可以推断出 a ~ Foo，因为第二个参数完全决定第一个参数，并将相应地简化类型。

SomeClass 类可以被看作是参数 a b c 的函数，结果是 x。这样的类可以用于在类型系统中进行计算。

第7.7节：FlexibleInstances

常规实例要求：

> 所有实例类型必须是形式为 (T a1 ... an)
> 其中 a1 ... an 是*不同的类型变量*，
> 且每个类型变量在实例头中最多出现一次。

这意味着，例如，虽然你可以为[a]创建一个实例，但你不能专门为其创建实例 [Int].; FlexibleInstances 放宽了这一点：

```
class C a where

-- 开箱即用
instance C [a] where
```

Using the LambdaCase extension, you can write that as a function expression, without having to name the argument:

```
{-# LANGUAGE LambdaCase #-}

dayOfTheWeek :: Int -> String
dayOfTheWeek = \case
    0 -> "Sunday"
    1 -> "Monday"
    2 -> "Tuesday"
    3 -> "Wednesday"
    4 -> "Thursday"
    5 -> "Friday"
    6 -> "Saturday"
```

Section 7.6: FunctionalDependencies

If you have a multi-parameter type-class with arguments a, b, c, and x, this extension lets you express that the type x can be uniquely identified from a, b, and c:

```
class SomeClass a b c x | a b c -> x where ...
```

When declaring an instance of such class, it will be checked against all other instances to make sure that the functional dependency holds, that is, no other instance with same `a b c` but different `x` exists.

You can specify multiple dependencies in a comma-separated list:

```
class OtherClass a b c d | a b -> c d, a d -> b where ...
```

For example in MTL we can see:

```
class MonadReader r m| m -> r where ...
instance MonadReader r ((->) r) where ...
```

Now, if you have a value of type `MonadReader a ((->) Foo) => a`, the compiler can infer that `a ~ Foo`, since the second argument completely determines the first, and will simplify the type accordingly.

The `SomeClass` class can be thought of as a function of the arguments `a b c` that results in `x`. Such classes can be used to do computations in the typesystem.

Section 7.7: FlexibleInstances

Regular instances require:

> 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.

That means that, for example, while you can create an instance for `[a]` you can't create an instance for specifically `[Int]`.; `FlexibleInstances` relaxes that:

```
class C a where

-- works out of the box
instance C [a] where
```

```
-- 需要 FlexibleInstances
instance C [Int] where
```

第7.8节：GADT（广义代数数据类型）

传统的代数数据类型在其类型变量上是参数化的。例如，如果我们定义一个ADT如下

```
data Expr a = IntLit Int
            | BoolLit Bool
            | If (Expr Bool) (Expr a) (Expr a)
```

希望这能静态地排除类型不正确的条件表达式，但这不会按预期工作，因为
type of `IntLit :: Int -> Expr a` 是全称量化的：对于 *任何* 选择的 `a`，它都会产生类型为 `Expr a` 的值。
特别地，对于 `a ~ Bool`，我们有 `IntLit :: Int -> Expr Bool`，这允许我们构造类似 `If (IntLit 1) e1 e2` 的表达式，而这正是 `If` 构造函数的类型试图排除的情况。

广义代数数据类型（Generalised Algebraic Data Types，GADT）允许我们控制数据构造函数的结果类型，使其不只是参数化的。我们可以将我们的Expr类型重写为如下的GADT：

```
data Expr a where
IntLit :: Int -> Expr Int
  BoolLit :: Bool -> Expr Bool
  If :: Expr Bool -> Expr a -> Expr a -> Expr a
```

这里，构造函数IntLit的类型是Int -> Expr Int，因此IntLit 1 :: Expr Bool将无法通过类型检查。

对GADT值进行模式匹配会导致返回项的类型被细化。例如，可以这样为Expr a编写一个求值器：

```
crazyEval :: Expr a -> a
crazyEval (IntLit x) =
   -- 这里我们可以使用`(+)`，因为x :: Int
   x + 1
crazyEval (BoolLit b) =
   -- 这里我们可以使用`not`，因为b :: Bool
   not b
crazyEval (If b thn els) =
  -- 因为 b :: Expr Bool，我们可以使用 `crazyEval b :: Bool`。
  -- 同时，因为 thn :: Expr a 和 els :: Expr a，我们可以将任一传递给
  -- 递归调用 `crazyEval` 并得到一个 a 类型的返回值
crazyEval $ if crazyEval b then thn else els
```

注意，我们能够在上述定义中使用 `(+)`，是因为当例如 `IntLit x` 被模式匹配时，我们也
得知 `a ~ Int`（对于 **not** 和 `if_then_else_` 当 `a ~ Bool` 时同理）。

第7.9节：元组切片（TupleSections）

一种语法扩展，允许以切片方式应用元组构造函数（它是一个操作符）：

```
(a,b) == (,) a b

-- 使用元组切片
(a,b) == (,) a b == (a,) b == (,b) a
```

N元组

它同样适用于元组元数大于二的情况

```
-- requires FlexibleInstances
instance C [Int] where
```

Section 7.8: GADTs

Conventional algebraic datatypes are parametric in their type variables. For example, if we define an ADT like

```
data Expr a = IntLit Int
            | BoolLit Bool
            | If (Expr Bool) (Expr a) (Expr a)
```

with the hope that this will statically rule out non-well-typed conditionals, this will not behave as expected since the type of `IntLit :: Int -> Expr a` is universially quantified: for *any* choice of `a`, it produces a value of type `Expr a`. In particular, for `a ~ Bool`, we have `IntLit :: Int -> Expr Bool`, allowing us to construct something like `If (IntLit 1) e1 e2` which is what the type of the `If` constructor was trying to rule out.

Generalised Algebraic Data Types allows us to control the resulting type of a data constructor so that they are not merely parametric. We can rewrite our `Expr` type as a GADT like this:

```
data Expr a where
  IntLit :: Int -> Expr Int
  BoolLit :: Bool -> Expr Bool
  If :: Expr Bool -> Expr a -> Expr a -> Expr a
```

Here, the type of the constructor `IntLit` is `Int -> Expr Int`, and so `IntLit 1 :: Expr Bool` will not typecheck.

Pattern matching on a GADT value causes refinement of the type of the term returned. For example, it is possible to write an evaluator for `Expr a` like this:

```
crazyEval :: Expr a -> a
crazyEval (IntLit x) =
   -- Here we can use `(+)` because x :: Int
   x + 1
crazyEval (BoolLit b) =
   -- Here we can use `not` because b :: Bool
   not b
crazyEval (If b thn els) =
  -- Because b :: Expr Bool, we can use `crazyEval b :: Bool`.
  -- Also, because thn :: Expr a and els :: Expr a, we can pass either to
  -- the recursive call to `crazyEval` and get an a back
  crazyEval $ if crazyEval b then thn else els
```

Note that we are able to use `(+)` in the above definitions because when e.g. `IntLit x` is pattern matched, we also learn that `a ~ Int` (and likewise for **not** and `if_then_else_` when `a ~ Bool`).

Section 7.9: TupleSections

A syntactic extension that allows applying the tuple constructor (which is an operator) in a section way:

```
(a,b) == (,) a b

-- With TupleSections
(a,b) == (,) a b == (a,) b == (,b) a
```

N-tuples

It also works for tuples with arity greater than two

```
(,2,) 1 3 == (1,2,3)
```

映射

这在其他使用区段的地方也很有用：

```
map (,"tag") [1,2,3] == [(1,"tag"), (2, "tag"), (3, "tag")]
```

没有此扩展的上述示例看起来是这样的：

```
map (\a -> (a, "tag")) [1,2,3]
```

第7.10节：重载列表

在GHC 7.8中添加。

重载列表（OverloadedLists）类似于重载字符串（OverloadedStrings），允许列表字面量被如下方式解糖：

```
[]          -- fromListN 0 []
[x]         -- fromListN 1 (x : [])
[x .. ]     -- fromList (enumFrom x)
```

这在处理诸如Set、Vector和Map等类型时非常方便。

```
['0' .. '9']             :: 集合 字符
[1 .. 10]                :: 向量 整数
[("default",0), (k1,v1)] :: 映射 字符串 整数
['a' .. 'z']             :: 文本
```

GHC.Exts 中的 IsList 类旨在与此扩展一起使用。

IsList 配备了一个类型函数 Item 和三个函数，fromList :: [Item l] -> l，toList :: l -> [Item l] 和 fromListN :: Int -> [Item l] -> l，其中 fromListN 是可选的。典型实现如下：

```
实例 IsList [a] 其中
  type Item [a] = a
fromList = id
toList   = id

实例 (Ord a) => IsList (Set a) 其中
  type Item (Set a) = a
fromList = Set.fromList
  toList   = Set.toList
```

示例取自 OverloadedLists – GHC。

第7.11节：多参数类型类

这是一个非常常见的扩展，允许具有多个类型参数的类型类。你可以将多参数类型类（MPTC）视为类型之间的关系。

```
{-# LANGUAGE MultiParamTypeClasses #-}

class Convertable a b where
    convert :: a -> b

instance Convertable Int Float where
```

```
(,2,) 1 3 == (1,2,3)
```

Mapping

This can be useful in other places where sections are used:

```
map (,"tag") [1,2,3] == [(1,"tag"), (2, "tag"), (3, "tag")]
```

The above example without this extension would look like this:

```
map (\a -> (a, "tag")) [1,2,3]
```

Section 7.10: OverloadedLists

added in GHC 7.8.

OverloadedLists, similar to OverloadedStrings, allows list literals to be desugared as follows:

```
[]          -- fromListN 0 []
[x]         -- fromListN 1 (x : [])
[x .. ]     -- fromList (enumFrom x)
```

This comes handy when dealing with types such as `Set`, `Vector` and `Map`s.

```
['0' .. '9']             :: Set Char
[1 .. 10]                :: Vector Int
[("default",0), (k1,v1)] :: Map String Int
['a' .. 'z']             :: Text
```

`IsList` class in `GHC.Exts` is intended to be used with this extension.

`IsList` is equipped with one type function, `Item`, and three functions, `fromList :: [Item l] -> l`, `toList :: l -> [Item l]` and `fromListN :: Int -> [Item l] -> l` where `fromListN` is optional. Typical implementations are:

```
instance IsList [a] where
  type Item [a] = a
  fromList = id
  toList   = id

instance (Ord a) => IsList (Set a) where
  type Item (Set a) = a
  fromList = Set.fromList
  toList   = Set.toList
```

Examples taken from OverloadedLists – GHC.

Section 7.11: MultiParamTypeClasses

It's a very common extension that allows type classes with multiple type parameters. You can think of MPTC as a relation between types.

```
{-# LANGUAGE MultiParamTypeClasses #-}

class Convertable a b where
    convert :: a -> b

instance Convertable Int Float where
```

```
convert i = fromIntegral i
```

参数的顺序很重要。

多参数类型类有时可以被类型族替代。

第7.12节：Unicode语法

一个扩展，允许你使用Unicode字符代替某些内置的操作符和名称。

ASCII	Unicode	用途
::	∷	具有类型
->	→	函数类型，lambda，case 分支等。
=>	⇒	类约束
forall	∀	显式多态
<-	←	do 表示法
*	★	类型（或种类）的类型（例如，Int :: ★）
>-		proc 表示法 用于 箭头
-<		proc 表示法 用于 箭头
>>-		proc 表示法 用于 箭头
-<<		proc 表示法 用于 箭头

例如：

```
runST :: (forall s. ST s a) -> a
```

将变为

```
runST ∷ (∀ s. ST s a) → a
```

注意，* 与 ★ 的例子略有不同：由于 * 不是保留符号，★ 也可以像 * 一样用于乘法，或者任何名为 (*) 的函数，反之亦然。例如：

```
ghci> 2 ★ 3
6
ghci> let (*) = (+) in 2 ★ 3
5
ghci> let (★) = (-) in 2 * 3
-1
```

第7.13节：PatternSynonyms（模式同义词）

模式同义词是模式的抽象，类似于函数是表达式的抽象。

在这个例子中，我们来看一下接口Data.Sequence所暴露的内容，并看看如何通过模式同义词来改进它。Seq类型是一个数据类型，内部使用了复杂的表示法，以实现各种操作的良好渐近复杂度，尤其是O(1)的（解）头部和（解）尾部操作。

但这种表示法笨重且其某些不变量无法用Haskell的类型系统表达。
因此，Seq类型作为抽象类型暴露给用户，同时提供保持不变量的访问器和构造函数，其中包括：

```
convert i = fromIntegral i
```

The order of parameters matters.

MPTCs can sometimes be replaced with type families.

Section 7.12: UnicodeSyntax

An extension that allows you to use Unicode characters in lieu of certain built-in operators and names.

ASCII	Unicode	Use(s)
::	∷	has type
->	→	function types, lambdas, case branches, etc.
=>	⇒	class constraints
forall	∀	explicit polymorphism
<-	←	do notation
*	★	the type (or kind) of types (e.g., Int :: ★)
>-	☐	proc notation for Arrows
-<	☐	proc notation for Arrows
>>-	☐	proc notation for Arrows
-<<	☐	proc notation for Arrows

For example:

```
runST :: (forall s. ST s a) -> a
```

would become

```
runST ∷ (∀ s. ST s a) → a
```

Note that the * vs. ★ example is slightly different: since * isn't reserved, ★ also works the same way as * for multiplication, or any other function named (*), and vice-versa. For example:

```
ghci> 2 ★ 3
6
ghci> let (*) = (+) in 2 ★ 3
5
ghci> let (★) = (-) in 2 * 3
-1
```

Section 7.13: PatternSynonyms

Pattern synonyms are abstractions of patterns similar to how functions are abstractions of expressions.

For this example, let's look at the interface Data.Sequence exposes, and let's see how it can be improved with pattern synonyms. The Seq type is a data type that, internally, uses a complicated representation to achieve good asymptotic complexity for various operations, most notably both O(1) (un)consing and (un)snocing.

But this representation is unwieldy and some of its invariants cannot be expressed in Haskell's type system. Because of this, the Seq type is exposed to users as an abstract type, along with invariant-preserving accessor and constructor functions, among them:

```haskell
empty :: Seq a

(<|) :: a -> Seq a -> Seq a
data ViewL a = EmptyL | a :< (Seq a)
viewl :: Seq a -> ViewL a

(|>) :: Seq a -> a -> Seq a
data ViewR a = EmptyR | (Seq a) :> a
viewr :: Seq a -> ViewR a
```

但使用这个接口有点麻烦：

```haskell
uncons :: Seq a -> Maybe (a, Seq a)
uncons xs = case viewl xs of
x :< xs' -> Just (x, xs')
    EmptyL -> Nothing
```

我们可以使用视图模式来稍微整理一下：

```haskell
{-# LANGUAGE ViewPatterns #-}

uncons :: Seq a -> Maybe (a, Seq a)
uncons (viewl -> x :< xs) = Just (x, xs)
uncons _ = Nothing
```

使用PatternSynonyms语言扩展，我们可以通过允许模式匹配来伪装成拥有一个cons列表或snoc列表，从而提供一个更好的接口：

```haskell
{-# LANGUAGE PatternSynonyms #-}
import Data.Sequence (Seq)
import qualified Data.Sequence as Seq

pattern Empty :: Seq a
pattern Empty <- (Seq.viewl -> Seq.EmptyL)

pattern (:<) :: a -> Seq a -> Seq a
pattern x :< xs <- (Seq.viewl -> x Seq.:< xs)

pattern (:>) :: Seq a -> a -> Seq a
pattern xs :> x <- (Seq.viewr -> xs Seq.:> x)
```

这使我们能够以非常自然的风格编写uncons函数：

```haskell
uncons :: Seq a -> Maybe (a, Seq a)
uncons (x :< xs) = Just (x, xs)
uncons _ = Nothing
```

第7.14节：ScopedTypeVariables（作用域类型变量）

ScopedTypeVariables 允许你在声明内部引用全称量化类型。更明确地说：

```haskell
import Data.Monoid

foo :: forall a b c. (Monoid b, Monoid c) => (a, b, c) -> (b, c) -> (a, b, c)
foo (a, b, c) (b', c') = (a :: a, b'', c'')
    where (b'', c'') = (b <> b', c <> c') :: (b, c)
```

```haskell
empty :: Seq a

(<|) :: a -> Seq a -> Seq a
data ViewL a = EmptyL | a :< (Seq a)
viewl :: Seq a -> ViewL a

(|>) :: Seq a -> a -> Seq a
data ViewR a = EmptyR | (Seq a) :> a
viewr :: Seq a -> ViewR a
```

But using this interface can be a bit cumbersome:

```haskell
uncons :: Seq a -> Maybe (a, Seq a)
uncons xs = case viewl xs of
    x :< xs' -> Just (x, xs')
    EmptyL -> Nothing
```

We can use view patterns to clean it up somewhat:

```haskell
{-# LANGUAGE ViewPatterns #-}

uncons :: Seq a -> Maybe (a, Seq a)
uncons (viewl -> x :< xs) = Just (x, xs)
uncons _ = Nothing
```

Using the `PatternSynonyms` language extension, we can give an even nicer interface by allowing pattern matching to pretend that we have a cons- or a snoc-list:

```haskell
{-# LANGUAGE PatternSynonyms #-}
import Data.Sequence (Seq)
import qualified Data.Sequence as Seq

pattern Empty :: Seq a
pattern Empty <- (Seq.viewl -> Seq.EmptyL)

pattern (:<) :: a -> Seq a -> Seq a
pattern x :< xs <- (Seq.viewl -> x Seq.:< xs)

pattern (:>) :: Seq a -> a -> Seq a
pattern xs :> x <- (Seq.viewr -> xs Seq.:> x)
```

This allows us to write uncons in a very natural style:

```haskell
uncons :: Seq a -> Maybe (a, Seq a)
uncons (x :< xs) = Just (x, xs)
uncons _ = Nothing
```

Section 7.14: ScopedTypeVariables

`ScopedTypeVariables` let you refer to universally quantified types inside of a declaration. To be more explicit:

```haskell
import Data.Monoid

foo :: forall a b c. (Monoid b, Monoid c) => (a, b, c) -> (b, c) -> (a, b, c)
foo (a, b, c) (b', c') = (a :: a, b'', c'')
    where (b'', c'') = (b <> b', c <> c') :: (b, c)
```

重要的是，我们可以使用 a、b 和 c 来指导编译器处理声明中的子表达式（如 where 子句中的元组以及最终结果中的第一个 a）。实际上，ScopedTypeVariables 有助于将复杂函数拆分为多个部分，使程序员能够为没有具体类型的中间值添加类型签名。

第7.15节：RecordWildCards（记录通配符）

参见 RecordWildCards

The important thing is that we can use `a`, `b` and `c` to instruct the compiler in subexpressions of the declaration (the tuple in the `where` clause and the first `a` in the final result). In practice, `ScopedTypeVariables` assist in writing complex functions as a sum of parts, allowing the programmer to add type signatures to intermediate values that don't have concrete types.

Section 7.15: RecordWildCards

See RecordWildCards

第8章：自由单子

第8.1节：自由单子将单子计算拆分为数据结构和解释器

例如，涉及从提示符读取和写入命令的计算：

首先我们将计算的“命令”描述为一个函子（Functor）数据类型

```haskell
{-# LANGUAGE DeriveFunctor #-}

data TeletypeF next
    = PrintLine String next
    | ReadLine (String -> next)
    deriving Functor
```

然后我们使用Free来创建“基于TeletypeF的自由单子（Free Monad）”，并构建一些基本操作。

```haskell
import Control.Monad.Free (Free, liftF, iterM)

type Teletype = Free TeletypeF

printLine :: String -> Teletype ()
printLine str = liftF (PrintLine str ())

readLine :: Teletype String
readLine = liftF (ReadLine id)
```

由于Free f在f是函子（Functor）时是一个单子（Monad），我们可以使用标准的单子组合子（包括do notation）来构建Teletype计算。

```haskell
import Control.Monad -- 我们可以使用标准组合子

echo :: Teletype ()
echo = readLine >>= printLine

mockingbird :: Teletype a
mockingbird = forever echo
```

最后，我们编写一个“解释器”，将Teletype a值转换成我们能处理的类型，比如IO a

```haskell
interpretTeletype :: Teletype a -> IO a
interpretTeletype = foldFree run where
  run :: TeletypeF a -> IO a
run (PrintLine str x) = putStrLn *> return x
  run (ReadLine f) = fmap f getLine
```

我们可以用它来在 IO 中“运行” Teletype a 计算

```
> interpretTeletype mockingbird
hello
hello
goodbye
goodbye
这将永远继续
这将永远继续
```

Chapter 8: Free Monads

Section 8.1: Free monads split monadic computations into data structures and interpreters

For instance, a computation involving commands to read and write from the prompt:

First we describe the "commands" of our computation as a Functor data type

```haskell
{-# LANGUAGE DeriveFunctor #-}

data TeletypeF next
    = PrintLine String next
    | ReadLine (String -> next)
    deriving Functor
```

Then we use `Free` to create the "Free Monad over `TeletypeF`" and build some basic operations.

```haskell
import Control.Monad.Free (Free, liftF, iterM)

type Teletype = Free TeletypeF

printLine :: String -> Teletype ()
printLine str = liftF (PrintLine str ())

readLine :: Teletype String
readLine = liftF (ReadLine id)
```

Since `Free f` is a `Monad` whenever `f` is a `Functor`, we can use the standard `Monad` combinators (including `do` notation) to build `Teletype` computations.

```haskell
import Control.Monad -- we can use the standard combinators

echo :: Teletype ()
echo = readLine >>= printLine

mockingbird :: Teletype a
mockingbird = forever echo
```

Finally, we write an "interpreter" turning `Teletype a` values into something we know how to work with like `IO a`

```haskell
interpretTeletype :: Teletype a -> IO a
interpretTeletype = foldFree run where
  run :: TeletypeF a -> IO a
  run (PrintLine str x) = putStrLn *> return x
  run (ReadLine f) = fmap f getLine
```

Which we can use to "run" the `Teletype a` computation in `IO`

```
> interpretTeletype mockingbird
hello
hello
goodbye
goodbye
this will go on forever
this will go on forever
```

第8.2节：Freer单子

自由单子有一种替代的表达方式，称为Freer（或Prompt，或Operational）单子。Freer单子不要求其底层指令集具有Functor实例，并且它比标准自由单子具有更明显的类似列表的结构。

Freer单子将程序表示为属于指令集 i :: * -> * 的一系列原子指令。每个指令使用其参数来声明其返回类型。例如，基本指令集的集合为状态单子如下：

```
data StateI 是一个 where
Get :: StateI s s  -- Get 指令返回类型为 's' 的值
    Put :: s -> StateI s ()  -- Put 指令包含一个 's' 类型的参数并返回 ()
```

这些指令的顺序执行是通过 :>>= 构造函数完成的。 :>>= 接受一个返回 a 的单个指令，并将其添加到程序的其余部分，将其返回值传递给后续操作。换句话说，给定一个返回 a 的指令，以及一个将 a 转换为返回 b 的程序的函数，:>>= 将生成一个返回 b 的程序。

```
data Freer i a where
Return :: a -> Freer i a
    (:>>=) :: i a -> (a -> Freer i b) -> Freer i b
```

注意 a 在 :>>= 构造函数中是存在量化的。解释器唯一能知道 a 是什么的方法是通过对 GADT i 进行模式匹配。

> **附注：共Yoneda引理告诉我们 Freer 与 Free 同构。回顾一下**
> CoYoneda 函子的定义：
>
> ```
> data CoYoneda i b where
>   CoYoneda :: i a -> (a -> b) -> CoYoneda i b
> ```
>
> Freer i 等价于 Free (CoYoneda i)。如果你取 Free 的构造函数并设定 f ~ CoYoneda i，得到：
>
> ```
> Pure :: a -> Free (CoYoneda i) a
> Free :: CoYoneda i (Free (CoYoneda i) b) -> Free (CoYoneda i) b ~
>        i a -> (a -> Free (CoYoneda i) b) -> Free (CoYoneda i) b
> ```
>
> 我们可以通过设置 Freer i ~ Free (CoYoneda i) 来恢复 Freer i 的构造函数。

因为对于任何 i，CoYoneda i 都是一个 Functor，Freer 对于任何 i 都是一个 Monad，即使 i 不是 Functor。

```
实例 Monad (Freer i) 其中
    return = Return
Return x >>= f = f x
    (i :>>= g) >>= f = i :>>= fmap (>>= f) g  -- 使用 `(->) r` 的 Functor 实例，因此 fmap = (.)
```

可以通过将指令映射到某个处理器 monad 来为 Freer 构建解释器。

```
foldFreer :: Monad m => (forall x. i x -> m x) -> Freer i a -> m a
foldFreer eta (Return x) = return x
foldFreer eta (i :>>= f) = eta i >>= (foldFreer eta . f)
```

Section 8.2: The Freer monad

There's an alternative formulation of the free monad called the Freer (or Prompt, or Operational) monad. The Freer monad doesn't require a Functor instance for its underlying instruction set, and it has a more recognisably list-like structure than the standard free monad.

The Freer monad represents programs as a sequence of atomic *instructions* belonging to the instruction set `i :: * -> *`. Each instruction uses its parameter to declare its return type. For example, the set of base instructions for the `State` monad are as follows:

```
data StateI s a where
    Get :: StateI s s  -- the Get instruction returns a value of type 's'
    Put :: s -> StateI s ()  -- the Put instruction contains an 's' as an argument and returns ()
```

Sequencing these instructions takes place with the `:>>=` constructor. `:>>=` takes a single instruction returning an `a` and prepends it to the rest of the program, piping its return value into the continuation. In other words, given an instruction returning an `a`, and a function to turn an `a` into a program returning a `b`, `:>>=` will produce a program returning a `b`.

```
data Freer i a where
    Return :: a -> Freer i a
    (:>>=) :: i a -> (a -> Freer i b) -> Freer i b
```

Note that `a` is existentially quantified in the `:>>=` constructor. The only way for an interpreter to learn what `a` is is by pattern matching on the GADT `i`.

> **Aside**: The co-Yoneda lemma tells us that `Freer` is isomorphic to `Free`. Recall the definition of the `CoYoneda` functor:
>
> ```
> data CoYoneda i b where
>   CoYoneda :: i a -> (a -> b) -> CoYoneda i b
> ```
>
> `Freer i` is equivalent to `Free (CoYoneda i)`. If you take `Free`'s constructors and set `f ~ CoYoneda i`, you get:
>
> ```
> Pure :: a -> Free (CoYoneda i) a
> Free :: CoYoneda i (Free (CoYonda i) b) -> Free (CoYonda i) b ~
>         i a -> (a -> Free (CoYoneda i) b) -> Free (CoYoneda i) b
> ```
>
> from which we can recover `Freer i`'s constructors by just setting `Freer i ~ Free (CoYoneda i)`.

Because `CoYoneda i` is a `Functor` for any `i`, `Freer` is a `Monad` for any `i`, even if `i` isn't a `Functor`.

```
instance Monad (Freer i) where
    return = Return
    Return x >>= f = f x
    (i :>>= g) >>= f = i :>>= fmap (>>= f) g  -- using `(->) r`'s instance of Functor, so fmap = (.)
```

Interpreters can be built for `Freer` by mapping instructions to some handler monad.

```
foldFreer :: Monad m => (forall x. i x -> m x) -> Freer i a -> m a
foldFreer eta (Return x) = return x
foldFreer eta (i :>>= f) = eta i >>= (foldFreer eta . f)
```

例如，我们可以使用常规的 State s monad 作为处理器来解释 Freer (StateI s) monad：

```
runFreerState :: Freer (StateI s) a -> s -> (a, s)
runFreerState = State.runState . foldFreer toState
    where toState :: StateI 是一个 -> State 是一个
        toState Get = State.get
        toState (Put x) = State.put x
```

第8.3节：foldFree和iterM是如何工作的？

有一些函数可以通过将Free计算解释为另一个单子m来帮助拆解它们：iterM :: (Functor f, Monad m) => (f (m a) -> m a) -> (Free f a -> m a) 和 foldFree :: Monad m => (forall x. f x -> m x) -> (Free f a -> m a)。它们是做什么的？

首先让我们看看手动拆解并解释一个Teletype a函数到IO需要做什么。我们可以将Free f a视为定义为

```
data Free f a
    = Pure a
    | Free (f (Free f a))
```

Pure情况很简单：

```
interpretTeletype :: Teletype a -> IO a
interpretTeletype (Pure x) = return x
interpretTeletype (Free teletypeF) = _
```

现在，如何解释一个用Free构造器构建的Teletype计算？我们希望通过检查teletypeF :: TeletypeF (Teletype a)，得到一个类型为IO a的值。首先，我们将编写一个函数runIO:: TeletypeF a -> IO a，它将自由单子的单层映射到一个IO动作：

```
runIO :: TeletypeF a -> IO a
runIO (PrintLine msg x) = putStrLn msg *> return x
runIO (ReadLine k) = fmap k getLine
```

现在我们可以使用 runIO 来填充 interpretTeletype 的其余部分。回想一下，teletypeF :: TeletypeF (Teletype a)是 TeletypeF 函子的一层，包含了剩余的 Free 计算。我们将使用 runIO 来解释最外层（因此我们有 runIO teletypeF :: IO (Teletype a)），然后使用 IO 单子中的 >>= 组合子来解释返回的 Teletype a。

```
interpretTeletype :: Teletype a -> IO a
interpretTeletype (Pure x) = return x
interpretTeletype (Free teletypeF) = runIO teletypeF >>= interpretTeletype
```

foldFree 的定义与 interpretTeletype 相同，只是将 runIO 函数提取了出来。因此，foldFree 独立于任何特定的基础函子和目标单子。

```
foldFree :: Monad m => (forall x. f x -> m x) -> Free f a -> m a
foldFree eta (Pure x) = return x
foldFree eta (Free fa) = eta fa >>= foldFree eta
```

foldFree 具有 rank-2 类型：eta 是一个自然变换。我们本可以给 foldFree 一个类型为 Monad m => (f (Free f a) -> m (Free f a)) -> Free f a -> m a，但那样会赋予 eta 检查 Free 计算在 f 层内部的自由度。给 foldFree 这个更严格的类型确保 eta 一次只能处理一层。

For example, we can interpret the Freer (StateI s) monad using the regular State s monad as a handler:

```
runFreerState :: Freer (StateI s) a -> s -> (a, s)
runFreerState = State.runState . foldFreer toState
    where toState :: StateI s a -> State s a
          toState Get = State.get
          toState (Put x) = State.put x
```

Section 8.3: How do foldFree and iterM work?

There are some functions to help tear down Free computations by interpreting them into another monad m: iterM :: (Functor f, Monad m) => (f (m a) -> m a) -> (Free f a -> m a) and foldFree :: Monad m => (forall x. f x -> m x) -> (Free f a -> m a). What are they doing?

First let's see what it would take to tear down an interpret a Teletype a function into IO manually. We can see Free f a as being defined

```
data Free f a
    = Pure a
    | Free (f (Free f a))
```

The Pure case is easy:

```
interpretTeletype :: Teletype a -> IO a
interpretTeletype (Pure x) = return x
interpretTeletype (Free teletypeF) = _
```

Now, how to interpret a Teletype computation that was built with the Free constructor? We'd like to arrive at a value of type IO a by examining teletypeF :: TeletypeF (Teletype a). To start with, we'll write a function runIO :: TeletypeF a -> IO a which maps a single layer of the free monad to an IO action:

```
runIO :: TeletypeF a -> IO a
runIO (PrintLine msg x) = putStrLn msg *> return x
runIO (ReadLine k) = fmap k getLine
```

Now we can use runIO to fill in the rest of interpretTeletype. Recall that teletypeF :: TeletypeF (Teletype a) is a layer of the TeletypeF functor which contains the rest of the Free computation. We'll use runIO to interpret the outermost layer (so we have runIO teletypeF :: IO (Teletype a)) and then use the IO monad's >>= combinator to interpret the returned Teletype a.

```
interpretTeletype :: Teletype a -> IO a
interpretTeletype (Pure x) = return x
interpretTeletype (Free teletypeF) = runIO teletypeF >>= interpretTeletype
```

The definition of foldFree is just that of interpretTeletype, except that the runIO function has been factored out. As a result, foldFree works independently of any particular base functor and of any target monad.

```
foldFree :: Monad m => (forall x. f x -> m x) -> Free f a -> m a
foldFree eta (Pure x) = return x
foldFree eta (Free fa) = eta fa >>= foldFree eta
```

foldFree has a rank-2 type: eta is a natural transformation. We could have given foldFree a type of Monad m => (f (Free f a) -> m (Free f a)) -> Free f a -> m a, but that gives eta the liberty of inspecting the Free computation inside the f layer. Giving foldFree this more restrictive type ensures that eta can only process a single layer at a time.

iterM 确实赋予折叠函数检查子计算的能力。前一次迭代的（单子）结果可用于下一次迭代，作为 f 参数的一部分。 iterM 类似于一个 paramorphism，而foldFree 则类似于一个 catamorphism。

```
iterM :: (Monad m, Functor f) => (f (m a) -> m a) -> Free f a -> m a
iterM phi (Pure x) = return x
iterM phi (Free fa) = phi (fmap (iterM phi) fa)
```

第8.4节：自由单子类似于不动点

比较Free的定义与Fix的定义：

```
数据类型 Free f a = Return a
                | Free (f (Free f a))

新类型 Fix f = Fix { unFix :: f (Fix f) }
```

特别地，比较Free构造函数的类型与Fix构造函数的类型。 Free像Fix一样分层一个函子，区别在于Free多了一个Return a的情况。

iterM does give the folding function the ability to examine the subcomputation. The (monadic) result of the previous iteration is available to the next, inside f's parameter. iterM is analogous to a *paramorphism* whereas foldFree is like a *catamorphism*.

```
iterM :: (Monad m, Functor f) => (f (m a) -> m a) -> Free f a -> m a
iterM phi (Pure x) = return x
iterM phi (Free fa) = phi (fmap (iterM phi) fa)
```

Section 8.4: Free Monads are like fixed points

Compare the definition of Free to that of Fix:

```
data Free f a = Return a
              | Free (f (Free f a))

newtype Fix f = Fix { unFix :: f (Fix f) }
```

In particular, compare the type of the Free constructor with the type of the Fix constructor. Free layers up a functor just like Fix, except that Free has an additional Return a case.

第9章：类型类

Haskell中的类型类是一种将与类型相关的行为与该类型的定义分离的手段。相比之下，比如在Java中，你会将行为定义为类型定义的一部分——即在接口、抽象类或具体类中——而Haskell将这两者分开。

Haskell的base包中已经定义了许多类型类。它们之间的关系在下面的备注部分有所说明。

第9.1节：Eq

除了函数和IO之外，Prelude中的所有基本数据类型（如Int、String、Eq a => [a]）都有Eq的实例。如果一个类型实例化了Eq，意味着我们知道如何比较两个值的值或结构相等性。

```
> 3 == 2
False
> 3 == 3
True
```

必需的方法

- (==) :: Eq a => a -> a -> Boolean 或 (/=) :: Eq a => a -> a -> Boolean（如果只实现其中一个，另一个默认取已定义方法的否定）

定义

- `(==) :: Eq a => a -> a -> Boolean`
- `(/=) :: Eq a => a -> a -> Boolean`

直接超类

无

显著子类

- `Ord`

第9.2节：幺半群（Monoid）

实例化Monoid的类型包括列表、数字以及返回值为Monoid的函数等。要实例化Monoid，类型必须支持一个结合性的二元操作（mappend或(<>)），用于组合其值，并且有一个特殊的“零”值（mempty），使得与该值组合不会改变原值：

```
mempty  <>  x == x
x <>  mempty  == x

x <> (y <> z) == (x <> y) <> z
```

直观地，Monoid 类型类似于“列表”，因为它们支持将值连接在一起。或者，Monoid 类型可以被看作是我们关心顺序但不关心分组的值序列。例如，二叉树是一个Monoid，但使用Monoid操作我们无法观察其分支结构，只能遍历其值（参见Foldable和Traversable）。

必需的方法

- mempty :: Monoid m => m

Chapter 9: Type Classes

Typeclasses in Haskell are a means of defining the behaviour associated with a type separately from that type's definition. Whereas, say, in Java, you'd define the behaviour as part of the type's definition -- i.e. in an interface, abstract class or concrete class -- Haskell keeps these two things separate.

There are a number of typeclasses already defined in Haskell's `base` package. The relationship between these is illustrated in the Remarks section below.

Section 9.1: Eq

All basic datatypes (like `Int`, `String`, `Eq a => [a]`) from Prelude except for functions and `IO` have instances of `Eq`. If a type instantiates `Eq` it means that we know how to compare two values for *value* or *structural* equality.

```
> 3 == 2
False
> 3 == 3
True
```

Required methods

- `(==) :: Eq a => a -> a -> Boolean` or `(/=) :: Eq a => a -> a -> Boolean` (if only one is implemented, the other defaults to the negation of the defined one)

Defines

- `(==) :: Eq a => a -> a -> Boolean`
- `(/=) :: Eq a => a -> a -> Boolean`

Direct superclasses

None

Notable subclasses

- `Ord`

Section 9.2: Monoid

Types instantiating `Monoid` include lists, numbers, and functions with `Monoid` return values, among others. To instantiate `Monoid` a type must support an associative binary operation (`mappend` or `(<>)`) which combines its values, and have a special "zero" value (`mempty`) such that combining a value with it does not change that value:

```
mempty  <>  x == x
x <>  mempty  == x

x <> (y <> z) == (x <> y) <> z
```

Intuitively, `Monoid` types are "list-like" in that they support appending values together. Alternatively, `Monoid` types can be thought of as sequences of values for which we care about the order but not the grouping. For instance, a binary tree is a `Monoid`, but using the `Monoid` operations we cannot witness its branching structure, only a traversal of its values (see `Foldable` and `Traversable`).

Required methods

- `mempty :: Monoid m => m`

- mappend :: Monoid m => m -> m -> m

直接超类

无

第9.3节：Ord

实例化了Ord的类型包括例如Int、String和[a]（对于存在Ord a实例的类型a）。如果一个类型实例化了Ord，意味着我们知道该类型值的“自然”排序。注意，通常一个类型有多种可能的“自然”排序，Ord强制我们选择其中一种。

Ord提供了标准的(<=)、(<)、(>)、(>=)操作符，但有趣的是，它们都是用自定义的代数数据类型定义的

```
数据类型Ordering = LT | EQ | GT

compare :: Ord a => a -> a -> Ordering
```

必需的方法

- compare :: Ord a => a -> a -> Ordering 或 (<=) :: Ord a => a -> a -> Boolean（标准的默认compare方法在实现中使用(<=)）

定义

- `compare :: Ord a => a -> a -> Ordering`
- `(<=) :: Ord a => a -> a -> Boolean`
- `(<) :: Ord a => a -> a -> Boolean`
- `(>=) :: Ord a => a -> a -> Boolean`
- `(>) :: Ord a => a -> a -> Boolean`
- `min :: Ord a => a -> a -> a`
- **max :: Ord a => a -> a -> a**

直接超类

- 等式

第9.4节：数字

数字类型中最通用的类别，更准确地说是“Num”类，即可以进行通常意义上的加法、减法和乘法，但不一定能进行除法的数字。

该类别包含整数类型（如Int、Integer、Word32等）和分数类型（如Double、Rational，以及复数等）。对于有限类型，其语义通常被理解为“模运算”，即
存在溢出和下溢†。

请注意，数值类的规则远不如单子（Monad）或幺半群（Monoid）定律，或等值比较的规则严格。特别是，浮点数通常只在近似意义上遵守这些定律。

方法

- fromInteger :: Num a => Integer -> a。将整数转换为通用数字类型（如有必要，进行范围内的环绕）。Haskell中的数字字面量可以理解为单态的Integer字面量，外加通用转换，因此你可以在Int上下文和Complex Double

- `mappend :: Monoid m => m -> m -> m`

Direct superclasses

None

Section 9.3: Ord

Types instantiating `Ord` include, e.g., `Int`, `String`, and `[a]` (for types a where there's an `Ord` a instance). If a type instantiates `Ord` it means that we know a “natural” ordering of values of that type. Note, there are often many possible choices of the “natural” ordering of a type and `Ord` forces us to favor one.

`Ord` provides the standard `(<=)`, `(<)`, `(>)`, `(>=)` operators but interestingly defines them all using a custom algebraic data type

```
data Ordering = LT | EQ | GT

compare :: Ord a => a -> a -> Ordering
```

Required methods

- `compare :: Ord a => a -> a -> Ordering` or `(<=) :: Ord a => a -> a -> Boolean` (the standard’s default `compare` method uses `(<=)` in its implementation)

Defines

- `compare :: Ord a => a -> a -> Ordering`
- `(<=) :: Ord a => a -> a -> Boolean`
- `(<) :: Ord a => a -> a -> Boolean`
- `(>=) :: Ord a => a -> a -> Boolean`
- `(>) :: Ord a => a -> a -> Boolean`
- `min :: Ord a => a -> a -> a`
- `max :: Ord a => a -> a -> a`

Direct superclasses

- Eq

Section 9.4: Num

The most general class for number types, more precisely for rings, i.e. numbers that can be added and subtracted and multiplied in the usual sense, but not necessarily divided.

This class contains both integral types (`Int`, `Integer`, `Word32` etc.) and fractional types (`Double`, `Rational`, also complex numbers etc.). In case of finite types, the semantics are generally understood as *modular arithmetic*, i.e. with over- and underflow†.

Note that the rules for the numerical classes are much less strictly obeyed than the monad or monoid laws, or those for equality comparison. In particular, floating-point numbers generally obey laws only in a approximate sense.

The methods

- `fromInteger :: Num a => Integer -> a`. convert an integer to the general number type (wrapping around the range, if necessary). Haskell number literals can be understood as a monomorphic `Integer` literal with the general conversion around it, so you can use the literal 5 in both an `Int` context and a `Complex Double`

环境中使用字面量5。

- (+) :: Num a => a -> a -> a。标准加法，通常被理解为结合律和交换律，
即，

```
a + (b + c) ≡ (a + b) + c
  a + b ≡ b + a
```

- (-) :: Num a => a -> a -> a. 减法，是加法的逆运算：

```
  (a - b) + b ≡ (a + b) - b ≡ a
```

- (*) :: Num a => a -> a -> a. 乘法，是结合律运算且对加法具有分配律：

```
a * (b * c) ≡ (a * b) * c
  a * (b + c) ≡ a * b + a * c
```

对于最常见的情况，乘法也是交换律的，但这绝不是必须的条件。

- negate :: Num a => a -> a. 一元取反运算符的全称。 -1 是 negate 的语法糖
1.

```
  -a ≡ negate a ≡ 0 - a
```

- abs :: Num a => a -> a. 绝对值函数总是返回一个非负且大小相同的结果

```
  abs (-a) ≡ abs a
  abs (abs a) ≡ abs a
```

abs a ≡ 0 仅当 a ≡ 0 时才成立。

对于 real 类型，非负的含义很明确：你总是有 abs a >= 0。复数等类型没有明确的顺序，然而 abs 的结果应始终位于实数子集‡（即给出一个也可以写成单个数字字面量且无负号的数字）。

- signum :: Num a => a -> a。符号函数，根据名称，只返回 -1 或 1，取决于参数的符号。实际上，这只对非零实数成立；一般来说，signum 更好地被
理解为 归一化 函数：

```
  abs (signum a) ≡ 1   -- 除非 a≡0
  signum a * abs a ≡ a -- 这对所有 Num 实例都必须成立
```

注意，Haskell 2010 报告的第 6.4.4 节 明确要求此最后的等式对任何有效的 Num 实例都成立。

一些库，尤其是 linear 和 hmatrix，对 Num 类的理解要宽松得多：它们仅将其视为 重载算术运算符的一种方式。虽然这对 + 和 - 来说相当直接，但对于 * 就已经有些麻烦，其他方法更是如此。例如，* 应该表示矩阵乘法还是元素乘法？

setting.

- (+) :: Num a => a -> a -> a. Standard addition, generally understood as associative and commutative, i.e.,

```
  a + (b + c) ≡ (a + b) + c
  a + b ≡ b + a
```

- (-) :: Num a => a -> a -> a. Subtraction, which is the inverse of addition:

```
  (a - b) + b ≡ (a + b) - b ≡ a
```

- (*) :: Num a => a -> a -> a. Multiplication, an associative operation that's distributive over addition:

```
  a * (b * c) ≡ (a * b) * c
  a * (b + c) ≡ a * b + a * c
```

for the most common instances, multiplication is also commutative, but this is definitely not a requirement.

- **negate** :: **Num** a => a -> a. The full name of the unary negation operator. -1 is syntactic sugar for **negate** 1.

```
  -a ≡ negate a ≡ 0 - a
```

- **abs** :: **Num** a => a -> a. The absolute-value function always gives a non-negative result of the same magnitude

```
  abs (-a) ≡ abs a
  abs (abs a) ≡ abs a
```

abs a ≡ 0 should only happen if a ≡ 0.

For real types it's clear what non-negative means: you always have **abs** a >= 0. Complex etc. types don't have a well-defined ordering, however the result of **abs** should always lie in the real subset‡ (i.e. give a number that could also be written as a single number literal without negation).

- **signum** :: **Num** a => a -> a. The sign function, according to the name, yields only -1 or 1, depending on the sign of the argument. Actually, that's only true for nonzero real numbers; in general **signum** is better understood as the *normalising* function:

```
  abs (signum a) ≡ 1   -- unless a≡0
  signum a * abs a ≡ a -- This is required to be true for all Num instances
```

Note that section 6.4.4 of the Haskell 2010 Report explicitly requires this last equality to hold for any valid **Num** instance.

Some libraries, notably linear and hmatrix, have a much laxer understanding of what the **Num** class is for: they treat it just as *a way to overload the arithmetic operators*. While this is pretty straightforward for + and -, it already becomes troublesome with * and more so with the other methods. For instance, *should * mean matrix multiplication or element-wise multiplication?*

定义此类非数字实例可能是个坏主意；请考虑使用专门的类，例如
向量空间.

† 特别地，无符号类型的“负数”会绕回到很大的正数，例如 (-4 :: Word32) ==
4294967292。

‡ 这一点通常不成立：向量类型没有真正的子集。对于这类类型的有争议的 Num 实例，通常是逐元素定义 abs 和 signum，从数学角度来说这并不合理。

第9.5节：Maybe 和函子类

在 Haskell 中，数据类型可以像函数一样带有参数。以 Maybe 类型为例。

Maybe 是一个非常有用的类型，它允许我们表示失败的概念，或者失败的可能性。换句话说，如果计算有失败的可能性，我们就使用 Maybe 类型。 Maybe 有点像其他类型的包装器，赋予它们额外的功能。

它的实际声明相当简单。

```
Maybe a = Just a | Nothing
```

这说明 Maybe 有两种形式，一种是 Just，表示成功，另一种是 Nothing，表示失败。 Just 接受一个参数，决定 Maybe 的类型，而 Nothing 不接受参数。例如，值 Just "foo" 的类型是 Maybe String，即带有额外 Maybe 功能的字符串类型。值 Nothing 的类型是 Maybe a，其中 a 可以是任意类型。

这种通过包装类型赋予其额外功能的思想非常有用，且不仅限于 Maybe。其他例子包括 Either、IO 和列表类型，它们各自提供不同的功能。然而，这些包装类型共有一些操作和能力，其中最显著的是修改封装值的能力。

通常人们会将这类类型看作可以放入值的盒子。不同的盒子装有不同的值并执行不同的操作，但如果不能访问其中的内容，它们就毫无用处。

为了封装这个概念，Haskell 提供了一个标准的类型类，名为Functor。它的定义如下。

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

如图所示，该类只有一个函数fmap，接受两个参数。第一个参数是一个从类型a到另一种类型b的函数。第二个参数是一个包含类型a值的函子（包装类型）。它返回一个包含类型b值的函子（包装类型）。

简单来说，fmap接受一个函数并将其应用于函子内部的值。它是一个类型成为Functor类成员所必需的唯一函数，但它非常有用。针对函子进行更具体操作的函数可以在Applicative和Monad类型类中找到。

第9.6节：类型类继承：Ord类型类

Haskell 支持类的扩展概念。例如，类Ord继承了Eq中的所有操作，但另外还包含一个返回值之间Ordering的compare函数。类Ord还可能包含常用的顺序比较运算符，以及min方法和max方法。

符号=>的含义与函数签名中的相同，要求类型a必须实现Eq，才能实现Ord。

It is arguably a bad idea to define such non-number instances; please consider dedicated classes such as VectorSpace.

† In particular, the “negatives” of unsigned types are wrapped around to large positive, e.g. (-4 :: Word32) == 4294967292.

‡ This is widely *not* fulfilled: vector types do not have a real subset. The controversial Num-instances for such types generally define **abs** and **signum** element-wise, which mathematically speaking doesn't really make sense.

Section 9.5: Maybe and the Functor Class

In Haskell, data types can have arguments just like functions. Take the Maybe type for example.

Maybe is a very useful type which allows us to represent the idea of failure, or the possiblity thereof. In other words, if there is a possibility that a computation will fail, we use the Maybe type there. Maybe acts kind of like a wrapper for other types, giving them additional functionality.

Its actual declaration is fairly simple.

```
Maybe a = Just a | Nothing
```

What this tells is that a Maybe comes in two forms, a Just, which represents success, and a Nothing, which represents failure. Just takes one argument which determines the type of the Maybe, and Nothing takes none. For example, the value Just "foo" will have type Maybe String, which is a string type wrapped with the additional Maybe functionality. The value Nothing has type Maybe a where a can be any type.

This idea of wrapping types to give them additional functionality is a very useful one, and is applicable to more than just Maybe. Other examples include the Either, IO and list types, each providing different functionality. However, there are some actions and abilities which are common to all of these wrapper types. The most notable of these is the ability to modify the encapsulated value.

It is common to think of these kinds of types as boxes which can have values placed in them. Different boxes hold different values and do different things, but none are useful without being able to access the contents within.

To encapsulate this idea, Haskell comes with a standard typeclass, named Functor. It is defined as follows.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

As can be seen, the class has a single function, **fmap**, of two arguments. The first argument is a function from one type, a, to another, b. The second argument is a functor (wrapper type) containing a value of type a. It returns a functor (wrapper type) containing a value of type b.

In simple terms, **fmap** takes a function and applies to the value inside of a functor. It is the only function necessary for a type to be a member of the Functor class, but it is extremely useful. Functions operating on functors that have more specific applications can be found in the Applicative and Monad typeclasses.

Section 9.6: Type class inheritance: Ord type class

Haskell supports a notion of class extension. For example, the class Ord inherits all of the operations in Eq, but in addition has a **compare** function that returns an Ordering between values. Ord may also contain the common order comparison operators, as well as a **min** method and a **max** method.

The => notation has the same meaning as it does in a function signature and requires type a to implement Eq, in

以实现Ord。

```
data Ordering = EQ | LT | GT

class Eq a => Ord a where
    compare :: Ord a => a -> a -> Ordering
    (<)     :: Ord a => a -> a -> Bool
    (<=)    :: Ord a => a -> a -> Bool
    (>)     :: Ord a => a -> a -> Bool
    (>=)    :: Ord a => a -> a -> Bool
    min     :: Ord a => a -> a -> a
    max     :: Ord a => a -> a -> a
```

所有跟随compare的方法都可以通过多种方式从它派生出来：

```
x < y   = compare x y == LT
x <= y  = x < y || x == y -- 注意这里使用了继承自 Eq 的 (==)
x > y   = not (x <= y)
x >= y  = not (x < y)

min x y = case compare x y of
               EQ -> x
LT -> x
               GT -> y

max x y = case compare x y of
               EQ -> x
LT -> y
               GT -> x
```

自身扩展Ord的类型类必须至少实现compare方法或(<=)方法中的一个，这样构建了有向继承结构。

order to implement **Ord**.

```
data Ordering = EQ | LT | GT

class Eq a => Ord a where
    compare :: Ord a => a -> a -> Ordering
    (<)     :: Ord a => a -> a -> Bool
    (<=)    :: Ord a => a -> a -> Bool
    (>)     :: Ord a => a -> a -> Bool
    (>=)    :: Ord a => a -> a -> Bool
    min     :: Ord a => a -> a -> a
    max     :: Ord a => a -> a -> a
```

All of the methods following `compare` can be derived from it in a number of ways:

```
x < y   = compare x y == LT
x <= y  = x < y || x == y -- Note the use of (==) inherited from Eq
x > y   = not (x <= y)
x >= y  = not (x < y)

min x y = case compare x y of
               EQ -> x
               LT -> x
               GT -> y

max x y = case compare x y of
               EQ -> x
               LT -> y
               GT -> x
```

Type classes that themselves extend **Ord** must implement at least either the `compare` method or the `(<=)` method themselves, which builds up the directed inheritance lattice.

第10章：输入输出（IO）

第10.1节：从 'IO a' 中获取 'a'

一个常见的问题是“我有一个值类型为IO a，但我想对这个 a值做点什么：我如何访问它？”如何操作来自外部世界的数据（例如，增加用户输入的数字）？

关键是，如果你对以非纯方式获得的数据使用纯函数，那么结果仍然是非纯的。这取决于用户做了什么！类型为IO a的值代表“一个产生类型为 a的值的副作用计算”，它只能通过（a）将其组合到main中，以及（b）编译并执行你的程序来运行。因此，在纯Haskell世界中，没有办法“取出 a”。

相反，我们想构建一个新的计算，一个新的IO值，它在运行时利用 a值。这是另一种组合IO值的方法，因此我们可以再次使用do记法：

```
-- 假设
myComputation :: IO Int

getMessage :: Int -> String
getMessage int = "我的计算结果是： " ++ show int

newComputation :: IO ()
newComputation = do
int <- myComputation        -- 我们将myComputation的结果“绑定”到名称'int'
  putStrLn $ getMessage int    -- 'int'保存了一个Int类型的值
```

这里我们使用纯函数（getMessage）将一个Int转换为String，但我们使用do记法使其应用于IO计算myComputation的结果（在该计算运行之后）。结果是一个更大的IO计算，newComputation。将纯函数用于非纯上下文的这种技术称为lifting。

第10.2节：IO定义了程序的`main`操作

要使 Haskell 程序可执行，必须提供一个类型为 IO () 的 main 函数文件

```
main :: IO ()
main = putStrLn "Hello world!"
```

当 Haskell 被编译时，它会检查这里的 `IO` 数据并将其转换为可执行文件。当我们运行此程序时，它将打印 `Hello world!`。

如果你有类型为 IO a 的值，且不是 main，它们不会有任何作用。

```
other :: IO ()
other = putStrLn "我不会被打印"

main :: IO ()
main = putStrLn "Hello world!"
```

编译并运行此程序将与上一个示例效果相同。 other 中的代码会被忽略。

为了使 other 中的代码在运行时生效，必须将其 compose（组合）到 main 中。没有最终组合到 main 的 IO 值不会有任何运行时效果。要顺序组合两个 IO 值，可以使用 do 记法：

Chapter 10: IO

Section 10.1: Getting the 'a' "out of" 'IO a'

A common question is "I have a value of `IO a`, but I want to do something to that a value: how do I get access to it?" How can one operate on data that comes from the outside world (for example, incrementing a number typed by the user)?

The point is that if you use a pure function on data obtained impurely, then the result is still impure. It depends on what the user did! A value of type `IO a` stands for a "side-effecting computation resulting in a value of type a" which can *only* be run by (a) composing it into `main` and (b) compiling and executing your program. For that reason, there is no way within pure Haskell world to "get the a out".

Instead, we want to build a new computation, a new `IO` value, which makes use of the a value *at runtime*. This is another way of *composing* `IO` values and so again we can use `do`-notation:

```
-- assuming
myComputation :: IO Int

getMessage :: Int -> String
getMessage int = "My computation resulted in: " ++ show int

newComputation :: IO ()
newComputation = do
  int <- myComputation       -- we "bind" the result of myComputation to a name, 'int'
  putStrLn $ getMessage int   -- 'int' holds a value of type Int
```

Here we're using a pure function (`getMessage`) to turn an `Int` into a `String`, but we're using `do` notation to make it be applied to the result of an `IO` computation `myComputation` *when* (after) that computation runs. The result is a bigger `IO` computation, `newComputation`. This technique of using pure functions in an impure context is called *lifting*.

Section 10.2: IO defines your program's `main` action

To make a Haskell program executable you must provide a file with a `main` function of type `IO ()`

```
main :: IO ()
main = putStrLn "Hello world!"
```

When Haskell is compiled it examines the `IO` data here and turns it into a executable. When we run this program it will print `Hello world!`.

If you have values of type `IO a` other than `main` they won't do anything.

```
other :: IO ()
other = putStrLn "I won't get printed"

main :: IO ()
main = putStrLn "Hello world!"
```

Compiling this program and running it will have the same effect as the last example. The code in `other` is ignored.

In order to make the code in `other` have runtime effects you have to *compose* it into `main`. No `IO` values not eventually composed into `main` will have any runtime effect. To compose two `IO` values sequentially you can use `do`-notation:

```
other :: IO ()
other = putStrLn "我会被打印......但仅在我被组合到 main 的时候"

main :: IO ()
main = do
  putStrLn "Hello world!"
  other
```

当你编译并运行这个程序时，它会输出

```
你好，世界！
我将被打印......但只有在我被组合进main时
```

注意，操作顺序是由其他如何被组合进main决定的，而不是定义顺序。

第10.3节：检查文件结束条件

与大多数其他语言的标准输入输出库的做法有些不同，Haskell的isEOF不需要你在检查EOF条件之前执行读取操作；运行时会为你完成这一步。

```
import System.IO( isEOF )


eofTest :: Int -> IO ()
eofTest line = do
end <- isEOF
    if end then
        putStrLn $ "文件末尾在第 " ++ show 行 ++ " 行到达。"
    否则执行
        getLine
eofTest $ line + 1


main :: IO ()
main =
eofTest 1
```

输入：

```
第 #1 行。
第 #2 行。
第 #3 行。
```

输出：

```
文件结束-在第 4 行达到。
```

第10.4节：将标准输入的所有内容读取到字符串中

```
main = do
input <- getContents
    putStr input
```

输入：

```
other :: IO ()
other = putStrLn "I will get printed... but only at the point where I'm composed into main"

main :: IO ()
main = do
  putStrLn "Hello world!"
  other
```

When you compile and run *this* program it outputs

```
Hello world!
I will get printed... but only at the point where I'm composed into main
```

Note that the order of operations is described by how `other` was composed into `main` and not the definition order.

Section 10.3: Checking for end-of-file conditions

A bit counter-intuitive to the way most other languages' standard I/O libraries do it, Haskell's `isEOF` does not require you to perform a read operation before checking for an EOF condition; the runtime will do it for you.

```
import System.IO( isEOF )


eofTest :: Int -> IO ()
eofTest line = do
    end <- isEOF
    if end then
        putStrLn $ "End-of-file reached at line " ++ show line ++ "."
    else do
        getLine
        eofTest $ line + 1


main :: IO ()
main =
    eofTest 1
```

Input:

```
Line #1.
Line #2.
Line #3.
```

Output:

```
End-of-file reached at line 4.
```

Section 10.4: Reading all contents of standard input into a string

```
main = do
    input <- getContents
    putStr input
```

Input:

```
这是一个示例句子。
这也是一个示例句子！
```

输出：

```
这是一个示例句子。
这也是一个示例句子！
```

注意：该程序实际上会在所有输入完全读取之前打印部分输出。这意味着，例如，如果你对一个50MiB的文件使用getContents，Haskell的惰性求值和垃圾回收机制将确保只将当前需要的文件部分（即对后续执行不可或缺的部分）加载到内存中。因此，50MiB的文件不会一次性全部加载到内存中。

第10.5节：IO的角色和目的

Haskell是一种纯语言，意味着表达式不能有副作用。副作用是指表达式或函数除了产生值之外所做的任何事情，例如修改全局计数器或打印到标准输出。

在Haskell中，有副作用的计算（特别是那些可能对现实世界产生影响的计算）使用IO来建模。严格来说，IO是一个类型构造器，接受一个类型并生成一个类型。例如，IO Int是产生Int值的I/O计算的类型。IO类型是抽象的，IO提供的接口确保某些非法值（即类型不合理的函数）不存在，通过确保所有执行IO的内置函数的返回类型都被包含在IO中。

当运行Haskell程序时，表示为名为main的Haskell值的计算（其类型可以是 IO x，x为任意类型）将被执行。

操作输入输出值

标准库中有许多函数提供了通用编程语言应执行的典型IO操作，例如对文件句柄的读写。通用的IO操作主要通过两个函数创建和组合：

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

该函数（通常称为bind）接受一个IO操作和一个返回IO操作的函数，并产生通过将该函数应用于第一个IO操作产生的值而得到的IO操作结果。

```
return :: a -> IO a
```

该函数接受任意值（即纯值），返回一个不执行任何IO但产生该值的IO计算。换句话说，它是一个无操作的I/O动作。

还有其他常用的一般函数，但都可以用上述两个函数来表示。例如，(>>) :: IO a -> IO b -> IO b 类似于(>>=)，但第一个操作的结果被忽略。

使用这些函数向用户问候的简单程序：

```
main :: IO ()
 main =
   putStrLn "你叫什么名字？" >>
```

```
This is an example sentence.
And this one is, too!
```

Output:

```
This is an example sentence.
And this one is, too!
```

Note: This program will actually print parts of the output before all of the input has been fully read in. This means that, if, for example, you use **getContents** over a 50MiB file, Haskell's lazy evaluation and garbage collector will ensure that only the parts of the file that are currently needed (read: indispensable for further execution) will be loaded into memory. Thus, the 50MiB file won't be loaded into memory at once.

Section 10.5: Role and Purpose of IO

Haskell is a pure language, meaning that expressions cannot have side effects. A side effect is anything that the expression or function does other than produce a value, for example, modify a global counter or print to standard output.

In Haskell, side-effectful computations (specifically, those which can have an effect on the real world) are modelled using IO. Strictly speaking, IO is a type constructor, taking a type and producing a type. For example, IO Int is the type of an I/O computation producing an Int value. The IO type is *abstract*, and the interface provided for IO ensures that certain illegal values (that is, functions with non-sensical types) cannot exist, by ensuring that all built-in functions which perform IO have a return type enclosed in IO.

When a Haskell program is run, the computation represented by the Haskell value named main, whose type can be IO x for any type x, is executed.

Manipulating IO values

There are many functions in the standard library providing typical IO actions that a general purpose programming language should perform, such as reading and writing to file handles. General IO actions are created and combined primarily with two functions:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

This function (typically called *bind*) takes an IO action and a function which returns an IO action, and produces the IO action which is the result of applying the function to the value produced by the first IO action.

```
return :: a -> IO a
```

This function takes any value (i.e., a pure value) and returns the IO computation which does no IO and produces the given value. In other words, it is a no-op I/O action.

There are additional general functions which are often used, but all can be written in terms of the two above. For example, (>>) :: IO a -> IO b -> IO b is similar to (>>=) but the result of the first action is ignored.

A simple program greeting the user using these functions:

```
main :: IO ()
main =
  putStrLn "What is your name?" >>
```

```
    getLine >>= name ->
    putStrLn ("Hello " ++ name ++ "!")
```

该程序还使用了 putStrLn :: String -> IO () 和 getLine :: IO String。

注意：上述某些函数的类型实际上比给出的类型更通用（即 `>>=`、`>>` 和 `return`）。

IO 语义

Haskell 中的 IO 类型的语义与命令式编程语言非常相似。例如，在命令式语言中写 s1 ; s2 表示先执行语句 s1，然后执行语句 s2，在 Haskell 中可以写成s1 >> s2 来模拟相同的操作。

然而，IO 的语义与命令式背景下的预期略有不同。
return 函数 不会 中断控制流——如果后面还有其他 IO 操作顺序执行，它对程序没有影响。例如，return () >> putStrLn "boom" 会正确地将 "boom" 打印到标准输出。

IO 的形式语义可以用上一节中函数的简单等式来表示：

```
return x >>= f ≡ f x, ∀ f x
y >>= return ≡ return y, ∀ y
(m >>= f) >>= g ≡ m >>= (\x -> (f x >>= g)), ∀ m f g
```

这些定律通常分别称为左恒等、右恒等和合成。它们可以用函数的方式更自然地表述

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

如下：

```
 return >=> f ≡ f, ∀ f
f >=> return ≡ f, ∀ f
 (f >=> g) >=> h ≡ f >=> (g >=> h), ∀ f g h
```

惰性 IO

执行 I/O 计算的函数通常是严格的，意味着在一系列操作中，所有前面的操作必须完成后，才能开始下一个操作。
通常这是有用且预期的行为——
putStrLn "X" >> putStrLn "Y" 应该打印 "XY"。然而，某些库函数执行惰性 I/O，意味着只有当值被实际使用时，才会执行产生该值所需的 I/O 操作。
此类函数的例子有 getContents 和 readFile。惰性 I/O 可能会大幅降低 Haskell 程序的性能，因此在使用库函数时，应注意哪些函数是惰性的。

IO 和 do 语法

Haskell 提供了一种更简单的方法，将不同的 IO 值组合成更大的 IO 值。这种特殊语法称为 do 语法*，它只是 >>=、>> 和 return 函数用法的语法糖。

上一节中的程序可以用两种不同的方式使用do表示法编写，第一种是对布局敏感的，第二种是对布局不敏感的：

```
main = do
```

```
    getLine >>= \name ->
    putStrLn ("Hello " ++ name ++ "!")
```

This program also uses `putStrLn :: String -> IO ()` and `getLine :: IO String`.

Note: the types of certain functions above are actually more general than those types given (namely `>>=`, `>>` and `return`).

IO semantics

The `IO` type in Haskell has very similar semantics to that of imperative programming languages. For example, when one writes `s1 ; s2` in an imperative language to indicate executing statement `s1`, then statement `s2`, one can write `s1 >> s2` to model the same thing in Haskell.

However, the semantics of `IO` diverge slightly of what would be expected coming from an imperative background. The `return` function *does not* interrupt control flow - it has no effect on the program if another `IO` action is run in sequence. For example, `return () >> putStrLn "boom"` correctly prints "boom" to standard output.

The formal semantics of `IO` can given in terms of simple equalities involving the functions in the previous section:

```
return x >>= f ≡ f x, ∀ f x
y >>= return ≡ return y, ∀ y
(m >>= f) >>= g ≡ m >>= (\x -> (f x >>= g)), ∀ m f g
```

These laws are typically referred to as left identity, right identity, and composition, respectively. They can be stated more naturally in terms of the function

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

as follows:

```
return >=> f ≡ f, ∀ f
f >=> return ≡ f, ∀ f
(f >=> g) >=> h ≡ f >=> (g >=> h), ∀ f g h
```

Lazy IO

Functions performing I/O computations are typically strict, meaning that all preceding actions in a sequence of actions must be completed before the next action is begun. Typically this is useful and expected behaviour - `putStrLn "X" >> putStrLn "Y"` should print "XY". However, certain library functions perform I/O lazily, meaning that the I/O actions required to produce the value are only performed when the value is actually consumed. Examples of such functions are `getContents` and `readFile`. Lazy I/O can drastically reduce the performance of a Haskell program, so when using library functions, care should be taken to note which functions are lazy.

IO and `do` notation

Haskell provides a simpler method of combining different IO values into larger IO values. This special syntax is known as `do` notation* and is simply syntactic sugar for usages of the `>>=`, `>>` and `return` functions.

The program in the previous section can be written in two different ways using `do` notation, the first being layout-sensitive and the second being layout insensitive:

```
main = do
```

```
    putStrLn "你叫什么名字？"
   name <- getLine
    putStrLn ("你好 " ++ name ++ "！")


main = do {
    putStrLn "你叫什么名字？" ;
   name <- getLine ;
    putStrLn ("你好 " ++ name ++ "！")
    }
```

这三个程序完全等价。

*注意do表示法也适用于一类更广泛的类型构造器，称为monads（单子）。

第10.6节：写入标准输出

根据Haskell 2010语言规范，以下是Prelude中可用的标准IO函数，因此使用它们无需导入任何模块。

putChar :: Char -> IO () - 向 stdout 写入一个 char

```
Prelude> putChar 'a'
aPrelude>  -- 注意，没有换行
```

putStr :: String -> IO () - 向 stdout 写入一个 String

```
Prelude> putStr "This is a string!"
This is a string!Prelude>  -- 注意，没有换行
```

putStrLn :: String -> IO () - 向 stdout 写入一个 String 并添加换行

```
Prelude> putStrLn "Hi there, this is another String!"
Hi there, this is another String!
```

print :: Show a => a -> IO () - 向 stdout 写入 a 的 Show 实例

```
Prelude> print "hi"
"hi"
Prelude> print 1
1
Prelude> print 'a'
'a'
Prelude> print (Just 'a')  -- Maybe 是 `Show` 类型类的实例
Just 'a'
Prelude> print Nothing
Nothing
```

回想一下，你可以使用deriving为你自己的类型实例化Show：

```
-- 在 ex.hs 中
data Person = Person { name :: String } deriving Show
main = print (Person "Alex")  -- 由于使用了`deriving`，Person 是 `Show` 的一个实例
```

在 GHCi 中加载并运行：

```
Prelude> :load ex.hs
[1 of 1] 正在编译 ex             ( ex.hs，解释执行 )
好的，模块已加载：ex。
*Main> main  -- 来自 ex.hs
Person {name = "Alex"}
*Main>
```

```
    putStrLn "What is your name?"
    name <- getLine
    putStrLn ("Hello " ++ name ++ "!")


main = do {
    putStrLn "What is your name?" ;
    name <- getLine ;
    putStrLn ("Hello " ++ name ++ "!")
    }
```

All three programs are exactly equivalent.

*Note that do notation is also applicable to a broader class of type constructors called *monads*.

Section 10.6: Writing to stdout

Per the Haskell 2010 Language Specification the following are standard IO functions available in Prelude, so no imports are required to use them.

putChar :: Char -> IO () - writes a char to stdout

```
Prelude> putChar 'a'
aPrelude>  -- Note, no new line
```

putStr :: String -> IO () - writes a String to stdout

```
Prelude> putStr "This is a string!"
This is a string!Prelude>  -- Note, no new line
```

putStrLn :: String -> IO () - writes a String to stdout and adds a new line

```
Prelude> putStrLn "Hi there, this is another String!"
Hi there, this is another String!
```

print :: Show a => a -> IO () - writes a an instance of Show to stdout

```
Prelude> print "hi"
"hi"
Prelude> print 1
1
Prelude> print 'a'
'a'
Prelude> print (Just 'a')  -- Maybe is an instance of the `Show` type class
Just 'a'
Prelude> print Nothing
Nothing
```

Recall that you can instantiate Show for your own types using deriving:

```
-- In ex.hs
data Person = Person { name :: String } deriving Show
main = print (Person "Alex")  -- Person is an instance of `Show`, thanks to `deriving`
```

Loading & running in GHCi:

```
Prelude> :load ex.hs
[1 of 1] Compiling ex             ( ex.hs, interpreted )
Ok, modules loaded: ex.
*Main> main  -- from ex.hs
Person {name = "Alex"}
*Main>
```

第10.7节：从整个文件中读取单词

在 Haskell 中，通常不必费心处理文件句柄，而是直接从磁盘读取或写入整个文件到内存†，并使用纯字符串数据结构对文本进行所有的分割/处理。这避免了 IO 和程序逻辑的混合，有助于大大减少错误。

```
-- | 程序中实际处理数据的有趣部分
-- 但不进行任何输入输出操作！
reverseWords :: String -> [String]
reverseWords = reverse . words

-- | 一个简单的包装器，只从磁盘获取数据，让
--   'reverseWords' 执行其功能，并将结果输出到标准输出。
main :: IO ()
main = do
content <- readFile "loremipsum.txt"
   mapM_ putStrLn $ reverseWords content
```

如果 loremipsum.txt 包含

```
Lorem ipsum dolor sit amet,
consectetur adipiscing elit
```

那么程序将输出

```
elit
adipiscing
consectetur
amet,
sit
dolor
ipsum
Lorem
```

这里，mapM_ 遍历了文件中所有单词的列表，并用 putStrLn 将每个单词打印到单独的一行。

†如果你认为这在内存上很浪费，你的观点是有道理的。实际上，Haskell 的惰性求值通常可以避免整个文件必须同时驻留在内存中......但请注意，这种惰性 IO 会带来一系列自己的问题。对于性能关键的应用，通常更合理的是严格地一次性读取整个文件；你可以使用Data.Text版本的readFile来实现这一点。

第10.8节：从标准输入读取一行

```
main = do
line <- getLine
    putStrLn line
```

输入：

```
这是一个示例。
```

输出：

```
这是一个示例。
```

Section 10.7: Reading words from an entire file

In Haskell, it often makes sense *not to bother with file handles* at all, but simply read or write an entire file straight from disk to memory†, and do all the partitioning/processing of the text with the pure string data structure. This avoids mixing IO and program logic, which can greatly help avoiding bugs.

```
-- | The interesting part of the program, which actually processes data
--   but doesn't do any IO!
reverseWords :: String -> [String]
reverseWords = reverse . words

-- | A simple wrapper that only fetches the data from disk, lets
--   'reverseWords' do its job, and puts the result to stdout.
main :: IO ()
main = do
   content <- readFile "loremipsum.txt"
   mapM_ putStrLn $ reverseWords content
```

If loremipsum.txt contains

```
Lorem ipsum dolor sit amet,
consectetur adipiscing elit
```

then the program will output

```
elit
adipiscing
consectetur
amet,
sit
dolor
ipsum
Lorem
```

Here, **mapM_** went through the list of all words in the file, and printed each of them to a separate line with **putStrLn**.

†If you think this is wasteful on memory, you have a point. Actually, Haskell's laziness can often avoid that the entire file needs to reside in memory simultaneously... but beware, this kind of lazy IO causes its own set of problems. For performance-critical applications, it often makes sense to enforce the entire file to be read at once, strictly; you can do this with the Data.Text version of **readFile**.

Section 10.8: Reading a line from standard input

```
main = do
    line <- getLine
    putStrLn line
```

Input:

```
This is an example.
```

Output:

```
This is an example.
```

第10.9节：从`stdin`读取

根据Haskell 2010语言规范，以下是Prelude中可用的标准IO函数，因此使用它们不需要导入任何模块。

getChar :: IO Char - 从stdin读取一个Char

```
-- MyChar.hs
main = do
myChar <- getChar
  print myChar

-- 在你的shell中

runhaskell MyChar.hs
a -- 你输入 a 并按回车
'a'  -- 程序打印 'a'
```

getLine :: IO String - 从标准输入读取一个String，不包含换行符

```
Prelude> getLine
Hello there!  -- 用户输入一些文本并按回车
"Hello there!"
```

read :: Read a => String -> a - 将字符串转换为一个值

```
Prelude> read "1" :: Int
1
Prelude> read "1" :: Float
1.0
Prelude> read "True" :: Bool
True
```

其他较少使用的函数有：

- getContents :: IO String - 返回所有用户输入作为一个字符串，按需惰性读取interact :: (String -> Strin
- g) -> IO () - 接受一个类型为 String->String 的函数作为参数。标准输入设备的全部输入作为参数传递给该函数

第10.10节：从标准输入解析并构造对象

```
readFloat :: IO 浮点数
readFloat =
    fmap read getLine


main :: IO ()
main = do
    putStr "请输入第一个数字："
   first <- readFloat

    putStr "请输入第二个数字："
   second <- readFloat

    putStrLn $ show first ++ " + " ++ show second ++ " = " ++ show ( first + second )
```

输入：

```
请输入第一个数字: 9.5
请输入第二个数字: -2.02
```

Section 10.9: Reading from `stdin`

As-per the Haskell 2010 Language Specification, the following are standard IO functions available in Prelude, so no imports are required to use them.

`getChar :: IO Char` - read a `Char` from `stdin`

```
-- MyChar.hs
main = do
  myChar <- getChar
  print myChar

-- In your shell

runhaskell MyChar.hs
a -- you enter a and press enter
'a'  -- the program prints 'a'
```

`getLine :: IO String` - read a `String` from `stdin`, sans new line character

```
Prelude> getLine
Hello there!  -- user enters some text and presses enter
"Hello there!"
```

`read :: Read a => String -> a` - convert a String to a value

```
Prelude> read "1" :: Int
1
Prelude> read "1" :: Float
1.0
Prelude> read "True" :: Bool
True
```

Other, less common functions are:

- `getContents :: IO String` - returns all user input as a single string, which is read lazily as it is needed
- `interact :: (String -> String) -> IO ()` - takes a function of type String->String as its argument. The entire input from the standard input device is passed to this function as its argument

Section 10.10: Parsing and constructing an object from standard input

```
readFloat :: IO Float
readFloat =
    fmap read getLine


main :: IO ()
main = do
    putStr "Type the first number: "
    first <- readFloat

    putStr "Type the second number: "
    second <- readFloat

    putStrLn $ show first ++ " + " ++ show second ++ " = " ++ show ( first + second )
```

Input:

```
Type the first number: 9.5
Type the second number: -2.02
```

输出：

```
9.5 + -2.02 = 7.48
```

第10.11节：从文件句柄读取

就像在I/O库的其他几个部分一样，隐式使用标准流的函数都有对应的
System.IO 执行相同任务，但在左侧多了一个类型为 Handle 的额外参数，表示正在处理的流。例如，getLine :: IO S
tring 有一个对应的 hGetLine :: Handle -> IO String。

```
import System.IO( Handle, FilePath, IOMode( ReadMode ),
                  openFile, hGetLine, hPutStr, hClose, hIsEOF, stderr )

import Control.Monad( when )


dumpFile :: Handle -> FilePath -> Integer -> IO ()

dumpFile handle filename lineNumber = do      -- 按行显示文件内容
    end <- hIsEOF handle
when ( not end ) $ do
line <- hGetLine handle
        putStrLn $ filename ++ ":" ++ show lineNumber ++ ": " ++ line
        dumpFile handle filename $ lineNumber + 1


main :: IO ()

main = do
hPutStr stderr "请输入文件名: "
    filename <- getLine
handle <- openFile filename ReadMode
    dumpFile handle filename 1
    hClose handle
```

文件内容 example.txt:

```
这是一个示例。
你好，世界！
这是另一个例子。
```

输入：

```
输入文件名: example.txt
```

输出：

```
example.txt:1: 这是一个例子。
example.txt:2: 你好，世界！
example.txt:3: 这是另一个例子
```

Output:

```
9.5 + -2.02 = 7.48
```

Section 10.11: Reading from file handles

Like in several other parts of the I/O library, functions that implicitly use a standard stream have a counterpart in `System.IO` that performs the same job, but with an extra parameter at the left, of type `Handle`, that represents the stream being handled. For instance, `getLine :: IO String` has a counterpart `hGetLine :: Handle -> IO String`.

```
import System.IO( Handle, FilePath, IOMode( ReadMode ),
                  openFile, hGetLine, hPutStr, hClose, hIsEOF, stderr )

import Control.Monad( when )


dumpFile :: Handle -> FilePath -> Integer -> IO ()

dumpFile handle filename lineNumber = do      -- show file contents line by line
    end <- hIsEOF handle
    when ( not end ) $ do
        line <- hGetLine handle
        putStrLn $ filename ++ ":" ++ show lineNumber ++ ": " ++ line
        dumpFile handle filename $ lineNumber + 1


main :: IO ()

main = do
    hPutStr stderr "Type a filename: "
    filename <- getLine
    handle <- openFile filename ReadMode
    dumpFile handle filename 1
    hClose handle
```

Contents of file `example.txt`:

```
This is an example.
Hello, world!
This is another example.
```

Input:

```
Type a filename: example.txt
```

Output:

```
example.txt:1: This is an example.
example.txt:2: Hello, world!
example.txt:3: This is another example
```

第11章：记录语法

第11.1节：基本语法

记录是代数和的扩展数据类型，允许字段被命名：

```
data StandardType = StandardType String Int Bool --创建和类型的标准方式

data RecordType = RecordType { -- 使用记录语法的相同和类型
aString :: String
  , aNumber :: Int
  , isTrue  :: Bool
  }
```

然后可以使用字段名从记录中获取命名字段

```
> let r = RecordType {aString = "Foobar", aNumber= 42, isTrue = True}
> :t r
r :: RecordType
> :t aString
aString :: RecordType -> String
> aString r
  "Foobar"
```

记录可以进行模式匹配

```
case r of
RecordType{aNumber = x, aString=str} -> ... -- x = 42, str = "Foobar"
```

注意并非所有字段都需要命名

记录是通过命名字段创建的，但也可以作为普通的和类型创建（当字段数量较少且不太可能变化时，这通常很有用）

```
r  = RecordType {aString = "Foobar", aNumber= 42, isTrue = True}
r' = RecordType  "Foobar" 42 True
```

如果创建记录时没有指定字段名，编译器会发出警告，生成的值将会是
未定义。

```
> let r = RecordType {aString = "Foobar", aNumber= 42}
  <interactive>:1:9: Warning:
RecordType的字段未初始化: isTrue
> isTrue r
错误 'undefined'
```

可以通过设置字段的值来更新记录的字段。未提及的字段不会改变。

```
> let r = RecordType {aString = "Foobar", aNumber= 42, isTrue = True}
> let r' = r{aNumber=117}
    -- r'{aString = "Foobar", aNumber= 117, isTrue = True}
```

为复杂的记录类型创建透镜通常很有用。

Chapter 11: Record Syntax

Section 11.1: Basic Syntax

Records are an extension of sum algebraic `data` type that allow fields to be named:

```
data StandardType = StandardType String Int Bool --standard way to create a sum type

data RecordType = RecordType { -- the same sum type with record syntax
    aString :: String
  , aNumber :: Int
  , isTrue  :: Bool
  }
```

The field names can then be used to get the named field out of the record

```
> let r = RecordType {aString = "Foobar", aNumber= 42, isTrue = True}
> :t r
  r :: RecordType
> :t aString
  aString :: RecordType -> String
> aString r
  "Foobar"
```

Records can be pattern matched against

```
case r of
  RecordType{aNumber = x, aString=str} -> ... -- x = 42, str = "Foobar"
```

Notice that not all fields need be named

Records are created by naming their fields, but can also be created as ordinary sum types (often useful when the number of fields is small and not likely to change)

```
r  = RecordType {aString = "Foobar", aNumber= 42, isTrue = True}
r' = RecordType  "Foobar" 42 True
```

If a record is created without a named field, the compiler will issue a warning, and the resulting value will be **`undefined`**.

```
> let r = RecordType {aString = "Foobar", aNumber= 42}
  <interactive>:1:9: Warning:
     Fields of RecordType not initialized: isTrue
> isTrue r
  Error 'undefined'
```

A field of a record can be updated by setting its value. Unmentioned fields do not change.

```
> let r = RecordType {aString = "Foobar", aNumber= 42, isTrue = True}
> let r' = r{aNumber=117}
    -- r'{aString = "Foobar", aNumber= 117, isTrue = True}
```

It is often useful to create lenses for complicated record types.

第11.2节：使用字段标签定义数据类型

可以定义带有字段标签的数据类型。

```
data Person = Person { age :: Int, name :: String }
```

这个定义不同于普通的记录定义，因为它还定义了*记录访问器*，可以用来
访问数据类型的部分内容。

在这个例子中，定义了两个记录访问器，age 和 name，分别允许我们访问 age 和 name 字段。

```
age :: Person -> Int
name :: Person -> String
```

记录访问器只是由编译器自动生成的 Haskell 函数。因此，它们
像普通的 Haskell 函数一样使用。

通过命名字段，我们还可以在许多其他上下文中使用字段标签，从而使代码更
易读。

模式匹配

```
lowerCaseName :: Person -> String
lowerCaseName (Person { name = x }) = map toLower x
```

我们可以在模式匹配时将相关字段标签位置的值绑定到一个新值（在
本例中为 x），该值可以在定义的右侧使用。

使用NamedFieldPuns进行模式匹配

```
lowerCaseName :: Person -> String
lowerCaseName (Person { name }) = map toLower name
```

NamedFieldPuns扩展允许我们只指定想要匹配的字段标签，这个name在定义的右侧被遮蔽，因此引用name指的是该值而非记录访问器。

使用RecordWildcards进行模式匹配

```
lowerCaseName :: Person -> String
lowerCaseName (Person { .. }) = map toLower name
```

使用RecordWildcards匹配时，所有字段标签都会被引入作用域。（在此具体示例中为name和
age）

该扩展稍有争议，因为如果你不确定Person的定义，值是如何被引入作用域的并不清楚。

记录更新

```
setName :: String -> Person -> Person
setName newName person = person { name = newName }
```

也有用于带字段标签的数据类型更新的特殊语法。

第11.3节：RecordWildCards

```
{-# LANGUAGE RecordWildCards #-}
```

Section 11.2: Defining a data type with field labels

It is possible to define a data type with field labels.

```
data Person = Person { age :: Int, name :: String }
```

This definition differs from a normal record definition as it also defines *record accessors* which can be used to access parts of a data type.

In this example, two record accessors are defined, `age` and `name`, which allow us to access the `age` and `name` fields respectively.

```
age :: Person -> Int
name :: Person -> String
```

Record accessors are just Haskell functions which are automatically generated by the compiler. As such, they are used like ordinary Haskell functions.

By naming fields, we can also use the field labels in a number of other contexts in order to make our code more readable.

Pattern Matching

```
lowerCaseName :: Person -> String
lowerCaseName (Person { name = x }) = map toLower x
```

We can bind the value located at the position of the relevant field label whilst pattern matching to a new value (in this case x) which can be used on the RHS of a definition.

Pattern Matching with `NamedFieldPuns`

```
lowerCaseName :: Person -> String
lowerCaseName (Person { name }) = map toLower name
```

The `NamedFieldPuns` extension instead allows us to just specify the field label we want to match upon, this name is then shadowed on the RHS of a definition so referring to `name` refers to the value rather than the record accessor.

Pattern Matching with `RecordWildcards`

```
lowerCaseName :: Person -> String
lowerCaseName (Person { .. }) = map toLower name
```

When matching using `RecordWildCards`, all field labels are brought into scope. (In this specific example, `name` and `age`)

This extension is slightly controversial as it is not clear how values are brought into scope if you are not sure of the definition of `Person`.

Record Updates

```
setName :: String -> Person -> Person
setName newName person = person { name = newName }
```

There is also special syntax for updating data types with field labels.

Section 11.3: RecordWildCards

```
{-# LANGUAGE RecordWildCards #-}
```

```
数据 Client = Client { firstName     :: 字符串
                     , lastName      :: 字符串
                     , clientID      :: 字符串
                     } 派生 (显示)

打印客户姓名 :: Client -> IO ()
打印客户姓名 Client{..} = 执行
    输出行 firstName
    输出行 lastName
    输出行 clientID
```

模式 Client{..} 引入构造函数 Client 的所有字段作用域，等同于模式

```
Client{ firstName = firstName, lastName = lastName, clientID = clientID }
```

它也可以与其他字段匹配器组合使用，如下所示：

```
Client { firstName = "Joe", .. }
```

这等同于

```
Client{ firstName = "Joe", lastName = lastName, clientID = clientID }
```

第11.4节：在更改字段值的同时复制记录

假设你有以下类型：

```
data Person = Person { name :: String, age:: Int } deriving (Show, Eq)
```

以及两个值：

```
alex = Person { name = "Alex", age = 21 }
jenny = Person { name = "Jenny", age = 36 }
```

可以通过从alex复制并指定要更改的值来创建一个新的Person类型的值：

```
anotherAlex = alex { age = 31 }
```

现在alex和anotherAlex的值将是：

```
Person {name = "Alex", age = 21}

Person {name = "Alex", age = 31}
```

第11.5节：使用newtype的记录

记录语法可以与newtype一起使用，前提是恰好有一个构造函数且该构造函数只有一个字段。这里的好处是自动创建一个函数来解包newtype。这些字段通常以运行（run）用于单子，获取（get）用于幺半群，和解包（un）用于其他类型开头命名。

```
newtype 状态 s a = 状态 { 运行状态 :: s -> (s, a) }

newtype 乘积 a = 乘积 { 获取乘积 :: a }

newtype Fancy = Fancy { unfancy :: String }
```

```
data Client = Client { firstName     :: String
                     , lastName      :: String
                     , clientID      :: String
                     } deriving (Show)

printClientName :: Client -> IO ()
printClientName Client{..} = do
    putStrLn firstName
    putStrLn lastName
    putStrLn clientID
```

The pattern `Client{..}` brings in scope all the fields of the constructor `Client`, and is equivalent to the pattern

```
Client{ firstName = firstName, lastName = lastName, clientID = clientID }
```

It can also be combined with other field matchers like so:

```
Client { firstName = "Joe", .. }
```

This is equivalent to

```
Client{ firstName = "Joe", lastName = lastName, clientID = clientID }
```

Section 11.4: Copying Records while Changing Field Values

Suppose you have this type:

```
data Person = Person { name :: String, age:: Int } deriving (Show, Eq)
```

and two values:

```
alex = Person { name = "Alex", age = 21 }
jenny = Person { name = "Jenny", age = 36 }
```

a new value of type `Person` can be created by copying from `alex`, specifying which values to change:

```
anotherAlex = alex { age = 31 }
```

The values of `alex` and `anotherAlex` will now be:

```
Person {name = "Alex", age = 21}

Person {name = "Alex", age = 31}
```

Section 11.5: Records with newtype

Record syntax can be used with `newtype` with the restriction that there is exactly one constructor with exactly one field. The benefit here is the automatic creation of a function to unwrap the newtype. These fields are often named starting with `run` for monads, `get` for monoids, and `un` for other types.

```
newtype State s a = State { runState :: s -> (s, a) }

newtype Product a = Product { getProduct :: a }

newtype Fancy = Fancy { unfancy :: String }
```

```
  -- 一个想要避免与普通字符串连接的花哨字符串
```

需要注意的是，记录语法通常从不用于构造值，字段名仅用于解包

```
getProduct $ mconcat [Product 7, Product 9, Product 12]
-- > 756
```

```
  -- a fancy string that wants to avoid concatenation with ordinary strings
```

It is important to note that the record syntax is typically never used to form values and the field name is used strictly for unwrapping

```
getProduct $ mconcat [Product 7, Product 9, Product 12]
-- > 756
```

第12章：部分应用

第12.1节：章节

分段是一种简洁的方式，用于部分应用中缀运算符的参数。

例如，如果我们想写一个函数，在单词末尾添加“ing”，我们可以使用区间来简洁地定义一个函数。

```
> (++ "ing") "laugh"
"laughing"
```

注意我们部分应用了第二个参数。通常，我们只能按照指定的顺序部分应用参数。

我们也可以使用左区间来部分应用第一个参数。

```
> ("re" ++) "do"
"redo"
```

我们也可以用普通的前缀部分应用来等价地写出这个表达式：

```
> ((++) "re") "do"
"redo"
```

关于减法的说明

初学者经常错误地对取负进行区间操作。

```
> map (-1) [1,2,3]
***error: Could not deduce...
```

这不起作用，因为-1被解析为字面量-1，而不是作为运算符-作用于1。为了解决这个问题，存在**subtract**函数。

```
> map (subtract 1) [1,2,3]
[0,1,2]
```

第12.2节：部分应用的加法函数

我们可以使用*部分应用*来“锁定”第一个参数。应用一个参数后，剩下一个函数
该函数期望再接收一个参数后返回结果。

```
(+) :: Int -> Int -> Int

addOne :: Int -> Int
addOne = (+) 1
```

然后我们可以使用addOne对一个Int加一。

```
> addOne 5
6
>
m
a  p  a           d n  1 2 3
      d           O e [ , , ]
[2,3,4]
```

Chapter 12: Partial Application

Section 12.1: Sections

Sectioning is a concise way to partially apply arguments to infix operators.

For example, if we want to write a function which adds "ing" to the end of a word we can use a section to succinctly define a function.

```
> (++ "ing") "laugh"
"laughing"
```

Notice how we have partially applied the second argument. Normally, we can only partially apply the arguments in the specified order.

We can also use left sectioning to partially apply the first argument.

```
> ("re" ++) "do"
"redo"
```

We could equivalently write this using normal prefix partial application:

```
> ((++) "re") "do"
"redo"
```

A Note on Subtraction

Beginners often incorrectly section negation.

```
> map (-1) [1,2,3]
***error: Could not deduce...
```

This does not work as `-1` is parsed as the literal `-1` rather than the sectioned operator `-` applied to `1`. The **subtract** function exists to circumvent this issue.

```
> map (subtract 1) [1,2,3]
[0,1,2]
```

Section 12.2: Partially Applied Adding Function

We can use *partial application* to "lock" the first argument. After applying one argument we are left with a function which expects one more argument before returning the result.

```
(+) :: Int -> Int -> Int

addOne :: Int -> Int
addOne = (+) 1
```

We can then use `addOne` in order to add one to an **Int**.

```
> addOne 5
6
> map addOne [1,2,3]
[2,3,4]
```

第12.3节：返回部分应用的函数

返回部分应用的函数是一种编写简洁代码的技巧。

```
add :: Int -> Int -> Int
add x = (+x)

add 5 2
```

在这个例子中，(+x) 是一个部分应用函数。注意，add 函数的第二个参数不需要在函数定义中指定。

调用 add 5 2 的结果是七。

Section 12.3: Returning a Partially Applied Function

Returning partially applied functions is one technique to write concise code.

```
add :: Int -> Int -> Int
add x = (+x)

add 5 2
```

In this example (+x) is a partially applied function. Notice that the second parameter to the add function does not need to be specified in the function definition.

The result of calling `add 5 2` is seven.

第13章：幺半群（Monoid）

第13.1节：列表的 Monoid 实例

```
instance Monoid [a] where
    mempty  = []
mappend = (++)
```

检查该实例的 Monoid 定律：

```
mempty `mappend` x = x   <->   [] ++ xs = xs  -- 在前面添加空列表无效

x `mappend` mempty = x   <->   xs ++ [] = xs  -- 在后面添加空列表无效

x `mappend` (y `mappend` z) = (x `mappend` y) `mappend` z
    <->
xs ++ (ys ++ zs) = (xs ++ ys) ++ zs          -- 连接列表是结合的
```

第13.2节：将一组单子（Monoids）折叠成单个值

mconcat :: [a] -> a 是 Monoid 类型类的另一种方法:

```
ghci> mconcat [Sum 1, Sum 2, Sum 3]
Sum {getSum = 6}
ghci> mconcat ["concat", "enate"]
"concatenate"
```

它的默认定义是 mconcat = foldr mappend mempty。

第13.3节：数值单子

数字在两种方式下是单子的：加法以0为单位元，乘法以1为单位元。两者在不同情况下都同样有效且有用。因此，与其为数字选择一个偏好的实例，不如使用两个newtypes，Sum和Product来标记它们以实现不同的功能。

```
newtype Sum n = Sum { getSum :: n }

instance Num n => Monoid (Sum n) where
    mempty = Sum 0
Sum x `mappend` Sum y = Sum (x + y)

newtype Product n = Product { getProduct :: n }

instance Num n => Monoid (Product n) where
    mempty = Product 1
Product x `mappend` Product y = Product (x * y)
```

这实际上允许开发者通过将值包装在适当的newtype中来选择使用哪种功能。

```
Sum 3     <> Sum 5     == Sum 8
Product 3 <> Product 5 == Product 15
```

Chapter 13: Monoid

Section 13.1: An instance of Monoid for lists

```
instance Monoid [a] where
    mempty  = []
    mappend = (++)
```

Checking the Monoid laws for this instance:

```
mempty `mappend` x = x   <->   [] ++ xs = xs  -- prepending an empty list is a no-op

x `mappend` mempty = x   <->   xs ++ [] = xs  -- appending an empty list is a no-op

x `mappend` (y `mappend` z) = (x `mappend` y) `mappend` z
    <->
xs ++ (ys ++ zs) = (xs ++ ys) ++ zs           -- appending lists is associative
```

Section 13.2: Collapsing a list of Monoids into a single value

`mconcat :: [a] -> a` is another method of the `Monoid` typeclass:

```
ghci> mconcat [Sum 1, Sum 2, Sum 3]
Sum {getSum = 6}
ghci> mconcat ["concat", "enate"]
"concatenate"
```

Its default definition is `mconcat = foldr mappend mempty`.

Section 13.3: Numeric Monoids

Numbers are monoidal in two ways: *addition* with 0 as the unit, and *multiplication* with 1 as the unit. Both are equally valid and useful in different circumstances. So rather than choose a preferred instance for numbers, there are two `newtypes`, `Sum` and `Product` to tag them for the different functionality.

```
newtype Sum n = Sum { getSum :: n }

instance Num n => Monoid (Sum n) where
    mempty = Sum 0
    Sum x `mappend` Sum y = Sum (x + y)

newtype Product n = Product { getProduct :: n }

instance Num n => Monoid (Product n) where
    mempty = Product 1
    Product x `mappend` Product y = Product (x * y)
```

This effectively allows for the developer to choose which functionality to use by wrapping the value in the appropriate `newtype`.

```
Sum 3     <> Sum 5     == Sum 8
Product 3 <> Product 5 == Product 15
```

第13.4节：()的Monoid实例

() 是一个 Monoid。由于类型 () 只有一个值，mempty 和 mappend 只能做一件事：

```
instance Monoid () where
    mempty = ()
    () `mappend` () = ()
```

Section 13.4: An instance of Monoid for ()

() is a Monoid. Since there is only one value of type (), there's only one thing mempty and mappend could do:

```
instance Monoid () where
    mempty = ()
    () `mappend` () = ()
```

第14章：范畴论

第14.1节：范畴论作为组织抽象的系统

范畴论是一种现代数学理论，是抽象代数的一个分支，专注于连通性和关系的本质。它对于为许多高度可重用的编程抽象提供坚实的基础和通用语言非常有用。Haskell 以范畴论为灵感，设计了标准库和多个流行第三方库中可用的一些核心类型类。

一个例子

Functor类型类表示如果一个类型F实例化了Functor（我们写作Functor F），那么我们就有一个通用操作

```
fmap :: (a -> b) -> (F a -> F b)
```

它允许我们对F进行"映射"。标准（但不完美）的直觉是F a是一个包含类型为a的值的容器，fmap让我们可以对这些包含的元素逐个应用转换。一个例子是Maybe

```
实例 Functor Maybe 定义如下
  fmap f Nothing = Nothing     -- 如果没有包含任何值，则不做任何操作
  fmap f (Just a) = Just (f a) -- 否则，应用我们的转换
```

基于这个直觉，一个常见的问题是"为什么不把Functor叫做像Mappable这样更明显的名字？"

范畴论的一个提示

原因是Functor属于范畴论中的一组常见结构，因此通过称其为Functor，我们可以看到它如何与这一更深层次的知识体系相连接。

特别地，范畴论高度关注从一个地方到另一个地方的箭头的概念。在Haskell中，最重要的一组箭头是函数箭头a -> b。范畴论中常研究的一件事是如何将一组箭头与另一组箭头关联起来。特别地，对于任何类型构造器F，形如F a

-> F b 也很有趣。

所以函子（Functor）是任何一个 F，使得普通的 Haskell 箭头 a -> b 与 F 特定的箭头 F a -> F b 之间存在联系。这个联系由 fmap 定义，我们也认识到必须满足的一些定律

```
forall (x :: F a) . fmap id x == x

forall (f :: a -> b) (g :: b -> c) . fmap g . fmap f = fmap (g . f)
```

所有这些定律自然来源于函子的范畴理论解释，如果我们仅仅把函子看作"映射元素"，这些定律就不会显得那么明显必要。

第14.2节：Haskell 类型作为一个范畴

范畴的定义

Haskell 类型及其之间的函数构成了（几乎是†）一个范畴。对于每个对象（类型）a，我们有一个恒等态射（函数）(id :: a -> a)；以及态射的复合 ((.) :: (b -> c) -> (a -> b)-> a -> c)，它们遵守范畴定律：

Chapter 14: Category Theory

Section 14.1: Category theory as a system for organizing abstraction

Category theory is a modern mathematical theory and a branch of abstract algebra focused on the nature of connectedness and relation. It is useful for giving solid foundations and common language to many highly reusable programming abstractions. Haskell uses Category theory as inspiration for some of the core typeclasses available in both the standard library and several popular third-party libraries.

An example

The Functor typeclass says that if a type F instantiates Functor (for which we write Functor F) then we have a generic operation

```
fmap :: (a -> b) -> (F a -> F b)
```

which lets us "map" over F. The standard (but imperfect) intuition is that F a is a container full of values of type a and fmap lets us apply a transformation to each of these contained elements. An example is Maybe

```
instance Functor Maybe where
  fmap f Nothing = Nothing     -- if there are no values contained, do nothing
  fmap f (Just a) = Just (f a) -- else, apply our transformation
```

Given this intuition, a common question is "why not call Functor something obvious like Mappable?".

A hint of Category Theory

The reason is that Functor fits into a set of common structures in Category theory and therefore by calling Functor "Functor" we can see how it connects to this deeper body of knowledge.

In particular, Category Theory is highly concerned with the idea of arrows from one place to another. In Haskell, the most important set of arrows are the function arrows a -> b. A common thing to study in Category Theory is how one set of arrows relates to another set. In particular, for any type constructor F, the set of arrows of the shape F a -> F b are also interesting.

So a Functor is any F such that there is a connection between normal Haskell arrows a -> b and the F-specific arrows F a -> F b. The connection is defined by fmap and we also recognize a few laws which must hold

```
forall (x :: F a) . fmap id x == x

forall (f :: a -> b) (g :: b -> c) . fmap g . fmap f = fmap (g . f)
```

All of these laws arise naturally from the Category Theoretic interpretation of Functor and would not be as obviously necessary if we only thought of Functor as relating to "mapping over elements".

Section 14.2: Haskell types as a category

Definition of the category

The Haskell types along with functions between types form (almost†) a category. We have an identity morphism (function) (id :: a -> a) for every object (type) a; and composition of morphisms ((.) :: (b -> c) -> (a -> b) -> a -> c), which obey category laws:

```
f . id = f = id . f
h . (g . f) = (h . g) . f
```

我们通常称这个范畴为 Hask。

同构

在范畴论中，当存在一个具有逆态射的态射时，我们称之为同构，换句话说，存在一个态射可以与之组合以产生恒等态射。在Hask中，这相当于有一对态射f和g，使得：

```
f . g == id == g . f
```

如果我们在两个类型之间找到这样一对态射，我们称它们彼此同构。

两个同构类型的例子是((),a)和 a，其中a是某个类型。我们可以构造两个态射：

```
f :: ((),a) -> a
f ((),x) = x

g :: a -> ((),a)
g x = ((),x)
```

我们可以验证f . g == id == g . f。

函子

在范畴论中，函子是从一个范畴映射到另一个范畴，映射对象和态射。我们这里只研究一个范畴，即Haskell类型的范畴Hask，因此我们只会看到从Hask到Hask的函子，这种起点和终点范畴相同的函子称为端函子。我们的端函子将是多态类型，接受一个类型并返回另一个类型：

```
F :: * -> *
```

遵守范畴函子定律（保持恒等和组合）等价于遵守Haskell函子定律：

```
fmap (f . g) = (fmap f) . (fmap g)
fmap id = id
```

因此，例如，[]、Maybe 或 (-> r) 在 Hask 中是函子。

单子

范畴论中的单子是endofunctors范畴上的一个幺半群。该范畴的对象是endofunctors F :: * -> *，态射是自然变换（它们之间的变换 forall a . F a -> G a）。

可以在一个幺半群范畴上定义一个幺半群对象，它是一个具有两个态射的类型：

```
zero :: () -> M
mappend :: (M,M) -> M
```

我们可以将其大致翻译为Hask endofunctors范畴中的形式：

```
f . id = f = id . f
h . (g . f) = (h . g) . f
```

We usually call this category **Hask**.

Isomorphisms

In category theory, we have an isomorphism when we have a morphism which has an inverse, in other words, there is a morphism which can be composed with it in order to create the identity. In **Hask** this amounts to have a pair of morphisms f,g such that:

```
 f . g == id == g . f
```

If we find a pair of such morphisms between two types, we call them *isomorphic to one another*.

An example of two isomorphic types would be ((),a) and a for some a. We can construct the two morphisms:

```
f :: ((),a) -> a
f ((),x) = x

g :: a -> ((),a)
g x = ((),x)
```

And we can check that f . g == id == g . f.

Functors

A functor, in category theory, goes from a category to another, mapping objects and morphisms. We are working only on one category, the category **Hask** of Haskell types, so we are going to see only functors from **Hask** to **Hask**, those functors, whose origin and destination category are the same, are called **endofunctors**. Our endofunctors will be the polymorphic types taking a type and returning another:

```
F :: * -> *
```

To obey the categorical functor laws (preserve identities and composition) is equivalent to obey the Haskell functor laws:

```
fmap (f . g) = (fmap f) . (fmap g)
fmap id = id
```

So, we have, for example, that [], Maybe or (-> r) are functors in **Hask**.

Monads

A monad in category theory is a monoid on the **category of endofunctors**. This category has endofunctors as objects F :: * -> * and natural transformations (transformations between them forall a . F a -> G a) as morphisms.

A monoid object can be defined on a monoidal category, and is a type having two morphisms:

```
zero :: () -> M
mappend :: (M,M) -> M
```

We can translate this roughly to the category of Hask endofunctors as:

```
return :: a -> m a
join :: m (m a) -> m a
```

并且，遵守单子定律等价于遵守范畴幺半群对象定律。

†事实上，由于存在 undefined，所有类型的类及类型之间的函数类在Haskell中并不严格构成一个范畴。通常通过简单地将 Hask 范畴的对象定义为不含底值的类型来解决这个问题，这排除了非终止函数和无限值（协数据）。关于该主题的详细讨论，请参见 here。

第14.3节：类别的定义

一个范畴 C 包含：

- 一组称为 Obj(C) 的对象；
- 一组（称为 Hom(C)）的这些对象之间的态射。如果 a 和 b 属于 Obj(C)，则态射 f 属于 Hom(C) 通常表示为 f : a -> b，所有从 a 到 b 的态射集合记为 hom(a,b)；
- 一个特殊的态射称为 *identity* 态射——对于每个 a : Obj(C)，存在一个态射 **id** : a -> a；
- 一个复合运算符（.），将两个态射 f : a -> b，g : b -> c 组合成一个态射 a -> c

满足以下定律：

```
对于 所有 f : a -> x, g : x -> b, 有 id . f = f 且 g . id = g
对于 所有 f : a -> b, g : b -> c 和 h : c -> d, 有 h . (g . f) = (h . g) . f
```

换句话说，与恒等态射的复合（无论左侧还是右侧）不会改变另一个态射，且复合运算满足结合律。

在 Haskell 中，Category 被定义为 Control.Category 中的一个类型类：

```
-- | 一个范畴的类。
-- id 和 (.) 必须构成一个幺半群。
class Category cat where
    -- | 恒等态射
    id :: cat a a

    -- | 态射的复合
    (.) :: cat b c -> cat a b -> cat a c
```

在这种情况下，cat :: k -> k -> * 体现了态射关系——当且仅当cat a b 被占据（即有值）时，存在态射 cat a b。 a 、b 和 c 都属于 Obj(C)。 Obj(C) 本身由 kind k 表示——例如，当 k ~ *，通常情况下，对象就是类型。

Haskell 中 Category 的典型例子是函数范畴：

```
instance Category (->) where
  id = Prelude.id
  (.) = Prelude..
```

```
return :: a -> m a
join :: m (m a) -> m a
```

And, to obey the monad laws is equivalent to obey the categorical monoid object laws.

†In fact, the class of all types along with the class of functions between types do *not* strictly form a category in Haskell, due to the existance of **undefined**. Typically this is remedied by simply defining the objects of the **Hask** category as types without bottom values, which excludes non-terminating functions and infinite values (codata). For a detailed discussion of this topic, see here.

Section 14.3: Definition of a Category

A category C consists of:

- A collection of objects called Obj(C) ;
- A collection (called Hom(C)) of morphisms between those objects. If a and b are in Obj(C), then a morphism f in Hom(C) is typically denoted f : a -> b, and the collection of all morphism between a and b is denoted hom(a,b) ;
- A special morphism called the *identity* morphism - for every a : Obj(C) there exists a morphism **id** : a -> a ;
- A composition operator (.), taking two morphisms f : a -> b, g : b -> c and producing a morphism a -> c

which obey the following laws:

```
For all f : a -> x, g : x -> b, then id . f = f and g . id = g
For all f : a -> b, g : b -> c and h : c -> d, then h . (g . f) = (h . g) . f
```

In other words, composition with the identity morphism (on either the left or right) does not change the other morphism, and composition is associative.

In Haskell, the Category is defined as a typeclass in Control.Category:

```
-- | A class for categories.
--   id and (.) must form a monoid.
class Category cat where
    -- | the identity morphism
    id :: cat a a

    -- | morphism composition
    (.) :: cat b c -> cat a b -> cat a c
```

In this case, cat :: k -> k -> * objectifies the morphism relation - there exists a morphism cat a b if and only if cat a b is inhabited (i.e. has a value). a, b and c are all in Obj(C). Obj(C) itself is represented by the *kind* k - for example, when k ~ *, as is typically the case, objects are types.

The canonical example of a Category in Haskell is the function category:

```
instance Category (->) where
  id = Prelude.id
  (.) = Prelude..
```

另一个常见的例子是 Monad 的 Kleisli 箭头的 Category：

```
newtype Kleisli m a b = Kleisli (a -> m b)

class Monad m => Category (Kleisli m) where
  id = Kleisli return
Kleisli f . Kleisli g = Kleisli (f >=> g)
```

第14.4节：Hask中的类型余积

直觉

两个类型A和B的范畴积应包含包含类型A或类型B实例所需的最小信息。我们现在可以看到，两个类型的直观余积应该是Either a b。
其他候选类型，如Either a (b,Bool)，会包含部分不必要的信息，因此它们不是最小的。

正式定义来源于范畴余积的范畴定义。

范畴余积

范畴余积是范畴积的对偶概念。它是通过将积的定义中的所有箭头反转直接得到的。两个对象X、Y的余积是另一个对象Z，带有两个包含映射：i_1: X → Z和i_2: Y → Z；使得从X和Y到另一个对象的任意两个态射都能唯一地通过这些包含映射分解。换句话说，如果存在两个态射f_1 : X → W和f_2 : Y → W，则存在唯一的态射g : Z → W，使得g ○ i_1 = f_1 且g ○ i_2 = f_2
Hask中的余积

转换到Hask范畴类似于积的转换：

```
-- 如果有两个函数
f1 :: A -> W
f2 :: B -> W
-- 我们有一个带有两个包含映射的余积
i1 :: A -> Z
i2 :: B -> Z
-- 我们可以构造一个唯一的函数
g  :: Z -> W
-- 使得另外两个函数可以通过 g 分解
g . i1 == f1
g . i2 == f2
```

在 Hask 中，两个类型 A 和 B 的余积类型是 Either a b 或任何与之同构的类型：

```
-- 余积
-- 两个包含映射是 Left 和 Right
data Either a b = Left a | Right b

-- 如果我们有这些函数，我们可以通过余积将它们分解
decompose :: (A -> W) -> (B -> W) -> (Either A B -> W)
decompose f1 f2 (Left x)  = f1 x
decompose f1 f2 (Right y) = f2 y
```

Another common example is the `Category` of `Kleisli` arrows for a `Monad`:

```
newtype Kleisli m a b = Kleisli (a -> m b)

class Monad m => Category (Kleisli m) where
  id = Kleisli return
  Kleisli f . Kleisli g = Kleisli (f >=> g)
```

Section 14.4: Coproduct of types in Hask

Intuition

The categorical product of two types **A** and **B** should contain the minimal information necessary to contain inside an instance of type **A** or type **B**. We can see now that the intuitive coproduct of two types should be `Either a b`. Other candidates, such as `Either a (b,Bool)`, would contain a part of unnecessary information, and they wouldn't be minimal.

The formal definition is derived from the categorical definition of coproduct.

Categorical coproducts

A categorical coproduct is the dual notion of a categorical product. It is obtained directly by reversing all the arrows in the definition of the product. The coproduct of two objects **X**,**Y** is another object **Z** with two inclusions: **i_1: X → Z** and **i_2: Y → Z**; such that any other two morphisms from **X** and **Y** to another object decompose uniquely through those inclusions. In other words, if there are two morphisms **f_1 : X → W** and **f_2 : Y → W**, exists a unique morphism **g : Z → W** such that **g ○ i_1 = f_1** and **g ○ i_2 = f_2**

Coproducts in Hask

The translation into the **Hask** category is similar to the translation of the product:

```
-- if there are two functions
f1 :: A -> W
f2 :: B -> W
-- and we have a coproduct with two inclusions
i1 :: A -> Z
i2 :: B -> Z
-- we can construct a unique function
g  :: Z -> W
-- such that the other two functions decompose using g
g . i1 == f1
g . i2 == f2
```

The coproduct type of two types A and B in **Hask** is `Either a b` or any other type isomorphic to it:

```
-- Coproduct
-- The two inclusions are Left and Right
data Either a b = Left a | Right b

-- If we have those functions, we can decompose them through the coproduct
decompose :: (A -> W) -> (B -> W) -> (Either A B -> W)
decompose f1 f2 (Left x)  = f1 x
decompose f1 f2 (Right y) = f2 y
```

第14.5节：Hask中的类型积

范畴积

在范畴论中，两个对象X和Y的积是另一个对象Z，带有两个投影：π_1 ：Z → X 和 π_2 ：Z → Y；使得来自另一个对象的任意两个态射都能唯一地通过这些投影分解。换句话说，如果存在 f_1 ：W → X 和 f_2 ：W → Y，则存在唯一的态射 g ：W → Z 使得 π_1 ○ g = f_1 且 π_2 ○ g =f_2。

Hask中的积

这在Haskell类型的Hask范畴中转化为如下，Z是A和B的积，当且仅当：

```
-- 如果有两个函数
f1 :: W -> A
f2 :: W -> B
-- 我们可以构造一个唯一的函数
g  :: W -> Z
-- 我们有两个投影
p1 :: Z -> A
p2 :: Z -> B
-- 使得另外两个函数可以通过 g 分解
p1 . g == f1
p2 . g == f2
```

两个类型 A 和 B 的积类型，遵循上述定律，是这两个类型的元组 (A,B)，两个投影分别是 **fst** 和 **snd**。如果我们有两个函数 `f1 :: W -> A` 和 `f2 :: W -> B`，可以唯一地将它们分解如下，我们可以验证它符合上述规则：

```
decompose :: (W -> A) -> (W -> B) -> (W -> (A,B))
decompose f1 f2 = (\x -> (f1 x, f2 x))
```

我们可以验证该分解是正确的：

```
fst . (decompose f1 f2) = f1
snd . (decompose f1 f2) = f2
```

同构意义下的唯一性

选择(A,B)作为A和B的乘积并不是唯一的。另一个合乎逻辑且等价的选择是：

```
data Pair a b = Pair a b
```

此外，我们也可以选择(B,A)作为乘积，甚至(B,A,())，并且我们也可以按照规则找到类似上述的分解函数：

```
decompose2 :: (W -> A) -> (W -> B) -> (W -> (B,A,()))
decompose2 f1 f2 = (\x -> (f2 x, f1 x, ()))
```

这是因为乘积不是唯一的，而是同构唯一的。任意两个A和B的乘积不必相等，但它们应该是同构的。举例来说，我们刚定义的两个不同的乘积，(A,B)和(B,A,())，是同构的：

```
iso1 :: (A,B) -> (B,A,())
iso1 (x,y) = (y,x,())
```

Section 14.5: Product of types in Hask

Categorical products

In category theory, the product of two objects **X**, **Y** is another object **Z** with two projections: π_1 **: Z → X** and π_2 **: Z → Y**; such that any other two morphisms from another object decompose uniquely through those projections. In other words, if there exist **f_1 : W → X** and **f_2 : W → Y**, exists a unique morphism **g : W → Z** such that π_1 ○ **g = f_1** and π_2 ○ **g = f_2**.

Products in Hask

This translates into the **Hask** category of Haskell types as follows, `Z` is product of `A`, `B` when:

```
-- if there are two functions
f1 :: W -> A
f2 :: W -> B
-- we can construct a unique function
g  :: W -> Z
-- and we have two projections
p1 :: Z -> A
p2 :: Z -> B
-- such that the other two functions decompose using g
p1 . g == f1
p2 . g == f2
```

The **product type of two types** `A`, `B`, which follows the law stated above, **is the tuple** of the two types `(A,B)`, and the two projections are **`fst`** and **`snd`**. We can check that it follows the above rule, if we have two functions `f1 :: W -> A` and `f2 :: W -> B` we can decompose them uniquely as follow:

```
decompose :: (W -> A) -> (W -> B) -> (W -> (A,B))
decompose f1 f2 = (\x -> (f1 x, f2 x))
```

And we can check that the decomposition is correct:

```
fst . (decompose f1 f2) = f1
snd . (decompose f1 f2) = f2
```

Uniqueness up to isomorphism

The choice of `(A,B)` as the product of `A` and `B` is not unique. Another logical and equivalent choice would have been:

```
data Pair a b = Pair a b
```

Moreover, we could have also chosen `(B,A)` as the product, or even `(B,A,())`, and we could find a decomposition function like the above also following the rules:

```
decompose2 :: (W -> A) -> (W -> B) -> (W -> (B,A,()))
decompose2 f1 f2 = (\x -> (f2 x, f1 x, ()))
```

This is because the product is not unique but *unique up to isomorphism*. Every two products of `A` and `B` do not have to be equal, but they should be isomorphic. As an example, the two different products we have just defined, `(A,B)` and `(B,A,())`, are isomorphic:

```
iso1 :: (A,B) -> (B,A,())
iso1 (x,y) = (y,x,())
```

```
iso2 :: (B,A,()) -> (A,B)
iso2 (y,x,()) = (x,y)
```

分解的唯一性

需要强调的是，分解函数也必须是唯一的。有些类型满足作为乘积所需的所有规则，但分解并不唯一。举例来说，我们可以尝试使用(A,(B,Bool))及投影函数fst fst和 snd作为A和B的乘积：

```
decompose3 :: (W -> A) -> (W -> B) -> (W -> (A,(B,Bool)))
decompose3 f1 f2 = (\x -> (f1 x, (f2 x, True)))
```

我们可以验证它确实有效：

```
fst         . (decompose3 f1 f2) = f1 x
(fst . snd) . (decompose3 f1 f2) = f2 x
```

但这里的问题是我们可以写出另一种分解，即：

```
decompose3' :: (W -> A) -> (W -> B) -> (W -> (A,(B,Bool)))
decompose3' f1 f2 = (\x -> (f1 x, (f2 x, False)))
```

并且，由于分解是不唯一的，(A,(B,Bool))并不是A和B在Hask中的乘积

第14.6节：用范畴论描述Haskell的Applicative

Haskell的Functor允许将任意类型a（Hask的一个对象）映射到类型F a，同时也将函数a -> b（Hask的一个态射）映射为类型为F a -> F b的函数。这在某种意义上对应于范畴论中的定义，即函子保持基本的范畴结构。

monoidal category（幺半范畴）是具有一些额外结构的范畴：

- 一个张量积（参见Hask中的类型乘积）
- 一个张量单位（单位对象）

以对偶作为我们的乘积，这一定义可以用以下方式翻译成Haskell：

```
class Functor f => Monoidal f where
    mcat :: f a -> f b -> f (a,b)
    munit :: f ()
```

Applicative类等价于这个Monoidal类，因此可以基于它来实现：

```
instance Monoidal f => Applicative f where
    pure x = fmap (const x) munit
f <*> fa = (\\(f, a) -> f a) <$> (mcat f fa)
```

```
iso2 :: (B,A,()) -> (A,B)
iso2 (y,x,()) = (x,y)
```

Uniqueness of the decomposition

It is important to remark that also the decomposition function must be unique. There are types which follow all the rules required to be product, but the decomposition is not unique. As an example, we can try to use `(A,(B,Bool))` with projections **`fst fst . snd`** as a product of `A` and `B`:

```
decompose3 :: (W -> A) -> (W -> B) -> (W -> (A,(B,Bool)))
decompose3 f1 f2 = (\x -> (f1 x, (f2 x, True)))
```

We can check that it does work:

```
fst         . (decompose3 f1 f2) = f1 x
(fst . snd) . (decompose3 f1 f2) = f2 x
```

But the problem here is that we could have written another decomposition, namely:

```
decompose3' :: (W -> A) -> (W -> B) -> (W -> (A,(B,Bool)))
decompose3' f1 f2 = (\x -> (f1 x, (f2 x, False)))
```

And, as the decomposition is **not unique**, `(A,(B,Bool))` is **not** the product of `A` and `B` in **Hask**

Section 14.6: Haskell Applicative in terms of Category Theory

A Haskell's `Functor` allows one to map any type `a` (an object of **Hask**) to a type `F a` and also map a function `a -> b` (a morphism of **Hask**) to a function with type `F a -> F b`. This corresponds to a Category Theory definition in a sense that functor preserves basic category structure.

A **monoidal category** is a category that has some *additional* structure:

- A tensor product (see Product of types in Hask)
- A tensor unit (unit object)

Taking a pair as our product, this definition can be translated to Haskell in the following way:

```
class Functor f => Monoidal f where
    mcat :: f a -> f b -> f (a,b)
    munit :: f ()
```

The `Applicative` class is equivalent to this `Monoidal` one and thus can be implemented in terms of it:

```
instance Monoidal f => Applicative f where
    pure x = fmap (const x) munit
    f <*> fa = (\(f, a) -> f a) <$> (mcat f fa)
```

第15章：列表

第15.1节：列表基础

Haskell Prelude中列表的类型构造器是[]。存放类型为Int的列表的类型声明写法如下：

```
xs :: [Int]    -- 或者等价但不太方便的写法，
xs :: [] Int
```

Haskell中的列表是同质的序列，意思是所有元素必须是相同类型。与元组不同，列表类型不受长度影响：

```
[1,2,3]   :: [Int]
[1,2,3,4] :: [Int]
```

列表是用两个构造器构造的：

- [] 构造一个空列表。
- (:)，发音为“cons”，用于将元素添加到列表前面。将 x（类型为 a 的值）添加到 xs（同类型 a 的值的列表）上，会创建一个新列表，其 head（第一个元素）是 x，tail（其余元素）是 xs。

我们可以如下定义简单列表：

```
ys :: [a]
ys = []

xs :: [Int]
xs = 12 : (99 : (37 : []))
-- 或者  = 12 : 99 : 37 : []      -- ((:) 是右结合的)
-- 或者  = [12, 99, 37]           -- （列表的语法糖）
```

注意 (++)，可以用来构建列表，是递归地基于 (:) 和 [] 定义的。

第15.2节：处理列表

要处理列表，我们可以简单地对列表类型的构造函数进行模式匹配：

```
listSum :: [Int] -> Int
listSum []          = 0
listSum (x:xs) = x + listSum xs
```

我们可以通过指定更复杂的模式来匹配更多的值：

```
sumTwoPer :: [Int] -> Int
sumTwoPer [] = 0
sumTwoPer (x1:x2:xs) = x1 + x2 + sumTwoPer xs
sumTwoPer (x:xs) = x + sumTwoPer xs
```

注意，在上述示例中，我们必须提供更全面的模式匹配，以处理作为参数传入的奇数长度列表的情况。

Chapter 15: Lists

Section 15.1: List basics

The type constructor for lists in the Haskell Prelude is []. The type declaration for a list holding values of type Int is written as follows:

```
xs :: [Int]    -- or equivalently, but less conveniently,
xs :: [] Int
```

Lists in Haskell are *homogeneous* sequences, which is to say that all elements must be of the same type. Unlike tuples, list type is not affected by length:

```
[1,2,3]   :: [Int]
[1,2,3,4] :: [Int]
```

Lists are constructed using two constructors:

- [] constructs an empty list.
- (:), pronounced "cons", prepends elements to a list. Consing x (a value of type a) onto xs (a list of values of the same type a) creates a new list, whose *head* (the first element) is x, and *tail* (the rest of the elements) is xs.

We can define simple lists as follows:

```
ys :: [a]
ys = []

xs :: [Int]
xs = 12 : (99 : (37 : []))
-- or  = 12 : 99 : 37 : []      -- ((:) is right-associative)
-- or  = [12, 99, 37]           -- (syntactic sugar for lists)
```

Note that (++), which can be used to build lists is defined recursively in terms of (:) and [].

Section 15.2: Processing lists

To process lists, we can simply pattern match on the constructors of the list type:

```
listSum :: [Int] -> Int
listSum []          = 0
listSum (x:xs) = x + listSum xs
```

We can match more values by specifying a more elaborate pattern:

```
sumTwoPer :: [Int] -> Int
sumTwoPer [] = 0
sumTwoPer (x1:x2:xs) = x1 + x2 + sumTwoPer xs
sumTwoPer (x:xs) = x + sumTwoPer xs
```

Note that in the above example, we had to provide a more exhaustive pattern match to handle cases where an odd length list is given as an argument.

Haskell Prelude 定义了许多用于处理列表的内置函数，如map、filter等。尽可能应使用这些函数，而不是自己编写递归函数。

第15.3节：范围

使用范围表示法创建从1到10的列表很简单：

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

要指定步长，在起始元素后添加逗号和下一个元素：

```
[1,3..10]  -- [1,3,5,7,9]
```

请注意，Haskell 总是将步长视为项之间的算术差值，并且你不能指定超过前两个元素和上限：

```
[1,3,5..10] -- 错误
[1,3,9..20] -- 错误
```

要生成降序范围，必须始终指定负步长：

```
[5..1]     -- []

[5,4..1]   -- [5,4,3,2,1]
```

由于 Haskell 是非严格求值，列表元素只有在需要时才会被计算，这使我们可以使用无限列表。 [1..] 是一个从 1 开始的无限列表。该列表可以绑定到变量或作为函数参数传递：

```
take 5 [1..]   -- 返回 [1,2,3,4,5]，尽管 [1..] 是无限的
```

使用浮点数范围时要小心，因为它接受最多半个步长的溢出，以避免舍入问题：

```
[1.0,1.5..2.4]    -- [1.0,1.5,2.0,2.5]，尽管 2.5 > 2.4

[1.0,1.1..1.2]    -- [1.0,1.1,1.2000000000000002]，尽管 1.2000000000000002 > 1.2
```

范围不仅适用于数字，还适用于实现了 Enum 类型类的任何类型。给定一些可枚举变量 a、b、c，范围语法等同于调用这些 Enum 方法：

```
[a..]    == enumFrom a
[a..c]   == enumFromTo a c
[a,b..]  == enumFromThen a b
[a,b..c] == enumFromThenTo a b c
```

例如，使用Bool时是

```
 [False ..]      -- [False,True]
```

注意False后面的空格，以防止被解析为模块名限定（即False..会被解析为来自模块False的.）。

The Haskell Prelude defines many built-ins for handling lists, like **map**, **filter**, etc.. Where possible, you should use these instead of writing your own recursive functions.

Section 15.3: Ranges

Creating a list from 1 to 10 is simple using range notation:

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

To specify a step, add a comma and the next element after the start element:

```
[1,3..10]  -- [1,3,5,7,9]
```

Note that Haskell always takes the step as the arithmetic difference between terms, and that you cannot specify more than the first two elements and the upper bound:

```
[1,3,5..10] -- error
[1,3,9..20] -- error
```

To generate a range in descending order, always specify the negative step:

```
[5..1]     -- []

[5,4..1]   -- [5,4,3,2,1]
```

Because Haskell is non-strict, the elements of the list are evaluated only if they are needed, which allows us to use infinite lists. [1..] is an infinite list starting from 1. This list can be bound to a variable or passed as a function argument:

```
take 5 [1..]   -- returns [1,2,3,4,5] even though [1..] is infinite
```

Be careful when using ranges with floating-point values, because it accepts spill-overs up to half-delta, to fend off rounding issues:

```
[1.0,1.5..2.4]    -- [1.0,1.5,2.0,2.5] , though 2.5 > 2.4

[1.0,1.1..1.2]    -- [1.0,1.1,1.2000000000000002] , though 1.2000000000000002 > 1.2
```

Ranges work not just with numbers but with any type that implements **Enum** typeclass. Given some enumerable variables a, b, c, the range syntax is equivalent to calling these **Enum** methods:

```
[a..]    == enumFrom a
[a..c]   == enumFromTo a c
[a,b..]  == enumFromThen a b
[a,b..c] == enumFromThenTo a b c
```

For example, with **Bool** it's

```
 [False ..]      -- [False,True]
```

Notice the space after False, to prevent this to be parsed as a module name qualification (i.e. False.. would be parsed as . from a module False).

第15.4节：列表字面量

```
emptyList     = []

singletonList = [0]               -- = 0 : []

listOfNums    = [1, 2, 3]         -- = 1 : 2 : [3]

listOfStrings = ["A", "B", "C"]
```

第15.5节：列表连接

```
listA      = [1, 2, 3]

listB      = [4, 5, 6]

listAThenB = listA ++ listB       -- [1, 2, 3, 4, 5, 6]

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

第15.6节：访问列表中的元素

访问列表中的第n个元素（从零开始计数）：

```
list = [1 .. 10]

firstElement = list !! 0          -- 1
```

注意 !! 是一个部分函数，因此某些输入会产生错误：

```
list !! (-1)     -- *** 异常：Prelude.!!: 负索引

list !! 1000     -- *** 异常：Prelude.!!: 索引过大
```

还有 Data.List.genericIndex，这是 !! 的重载版本，接受任何 Integral 类型的值作为索引。

```
import Data.List (genericIndex)

list `genericIndex` 4             -- 5
```

当以单链表实现时，这些操作的时间复杂度为 O(n)。如果你经常通过索引访问元素，可能更适合使用 Data.Vector（来自 vector 包）或其他数据结构。

第15.7节：列表的基本功能

```
head [1..10]       --    1

last [1..20]       --    20

tail [1..5]        --    [2, 3, 4, 5]

init [1..5]        --    [1, 2, 3, 4]

length [1 .. 10]   --    10
```

Section 15.4: List Literals

```
emptyList     = []

singletonList = [0]               -- = 0 : []

listOfNums    = [1, 2, 3]         -- = 1 : 2 : [3]

listOfStrings = ["A", "B", "C"]
```

Section 15.5: List Concatenation

```
listA      = [1, 2, 3]

listB      = [4, 5, 6]

listAThenB = listA ++ listB       -- [1, 2, 3, 4, 5, 6]

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

Section 15.6: Accessing elements in lists

Access the *n*th element of a list (zero-based):

```
list = [1 .. 10]

firstElement = list !! 0          -- 1
```

Note that `!!` is a partial function, so certain inputs produce errors:

```
list !! (-1)     -- *** Exception: Prelude.!!: negative index

list !! 1000     -- *** Exception: Prelude.!!: index too large
```

There's also `Data.List.genericIndex`, an overloaded version of `!!`, which accepts any **`Integral`** value as the index.

```
import Data.List (genericIndex)

list `genericIndex` 4             -- 5
```

When implemented as singly-linked lists, these operations take *O(n)* time. If you frequently access elements by index, it's probably better to use `Data.Vector` (from the vector package) or other data structures.

Section 15.7: Basic Functions on Lists

```
head [1..10]       --    1

last [1..20]       --    20

tail [1..5]        --    [2, 3, 4, 5]

init [1..5]        --    [1, 2, 3, 4]

length [1 .. 10]   --    10
```

```
reverse [1 .. 10]  --    [10, 9 .. 1]

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

drop 5 [1 .. 10]   --    [6, 7, 8, 9, 10]

concat [[1,2], [], [4]]   --    [1,2,4]
```

第15.8节：使用 `map` 进行转换

我们经常希望转换或变换集合（列表或可遍历的结构）中的内容。在Haskell中，我们使用 map：

```
-- 简单地加1
map (+ 1) [1,2,3]
[2,3,4]

map odd [1,2,3]
[True,False,True]

data 性别 = 男 | 女 deriving Show
data 人 = 人 String 性别 Int deriving Show

-- 从人员列表中提取年龄
map (\(人 n g a) -> a) [(人 "Alex" 男 31),(人 "Ellie" 女 29)]
[31,29]
```

第15.9节：使用 `filter` 进行过滤

给定一个列表：

```
li = [1,2,3,4,5]
```

我们可以使用谓词通过 filter :: (a -> Bool) -> [a] -> [a] 来过滤列表：

```
filter (== 1) li       -- [1]

filter (even) li       -- [2,4]

filter (odd) li        -- [1,3,5]

-- 稍微复杂一点的例子
comfy i = notTooLarge && isEven
  其中
notTooLarge = (i + 1) < 5
      isEven = even i

filter comfy li        -- [2]
```

当然，这不仅仅是关于数字：

```
data Gender = Male | Female deriving Show
data Person = Person String Gender Int deriving Show

onlyLadies :: [Person] -> Person
onlyLadies x = filter isFemale x
  where
isFemale (Person _ Female _) = True
```

```
reverse [1 .. 10]  --    [10, 9 .. 1]

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

drop 5 [1 .. 10]   --    [6, 7, 8, 9, 10]

concat [[1,2], [], [4]]   --    [1,2,4]
```

Section 15.8: Transforming with `map`

Often we wish to convert, or transform the contents of a collection (a list, or something traversable). In Haskell we use **map**:

```
-- Simple add 1
map (+ 1) [1,2,3]
[2,3,4]

map odd [1,2,3]
[True,False,True]

data Gender = Male | Female deriving Show
data Person = Person String Gender Int deriving Show

-- Extract just the age from a list of people
map (\(Person n g a) -> a) [(Person "Alex" Male 31),(Person "Ellie" Female 29)]
[31,29]
```

Section 15.9: Filtering with `filter`

Given a list:

```
li = [1,2,3,4,5]
```

we can filter a list with a predicate using **filter** :: (a -> Bool) -> [a] -> [a]:

```
filter (== 1) li       -- [1]

filter (even) li       -- [2,4]

filter (odd) li        -- [1,3,5]

-- Something slightly more complicated
comfy i = notTooLarge && isEven
  where
     notTooLarge = (i + 1) < 5
     isEven = even i

filter comfy li        -- [2]
```

Of course it's not just about numbers:

```
data Gender = Male | Female deriving Show
data Person = Person String Gender Int deriving Show

onlyLadies :: [Person] -> Person
onlyLadies x = filter isFemale x
  where
    isFemale (Person _ Female _) = True
isFemale _ = False

onlyLadies [(Person "Alex" 男性 31),(Person "Ellie" 女性 29)]
 -- [Person "Ellie" 女性 29]
```

第15.10节：foldr

右折叠的实现方式如下：

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)              -- = x `f` foldr f z xs
```

右折叠foldr是向右结合的。也就是说：

```
foldr (+) 0 [1, 2, 3]      -- 等价于 1 + (2 + (3 + 0))
```

原因是foldr的计算方式如下（参见foldr的归纳步骤）：

```
foldr (+) 0 [1, 2, 3]                        --          foldr (+) 0  [1,2,3]
(+) 1 (foldr (+) 0 [2, 3])                   -- 1 +        foldr (+) 0  [2,3]
(+) 1 ((+) 2 (foldr (+) 0 [3]))              -- 1 + (2 +     foldr (+) 0  [3])
(+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [])))       -- 1 + (2 + (3 +  foldr (+) 0 []))
(+) 1 ((+) 2 ((+) 3 0))                      -- 1 + (2 + (3 +             0   ))
```

最后一行等价于 1 + (2 + (3 + 0))，因为 ((+) 3 0) 与 (3 + 0) 是相同的。

第15.11节：列表的压缩与解压

zip 接受两个列表并返回对应元素对的列表：

```
zip []     _      = []
zip _      []     = []
zip (a:as) (b:bs) = (a,b) : zip as bs

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

用函数压缩两个列表：

```
zipWith f  []     _      = []
zipWith f  _      []     = []
zipWith f  (a:as) (b:bs) = f a b : zipWith f as bs

> zipWith (+) [1,3,5] [2,4,6]
> [3,7,11]
```

解压列表：

```
unzip = foldr (\(a,b) ~(as,bs) -> (a:as,b:bs)) ([],[])

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

```
    isFemale _ = False

onlyLadies [(Person "Alex" Male 31),(Person "Ellie" Female 29)]
-- [Person "Ellie" Female 29]
```

Section 15.10: foldr

This is how the right fold is implemented:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)              -- = x `f` foldr f z xs
```

The right fold, **foldr**, associates to the right. That is:

```
foldr (+) 0 [1, 2, 3]      -- is equivalent to 1 + (2 + (3 + 0))
```

The reason is that **foldr** is evaluated like this (look at the inductive step of **foldr**):

```
foldr (+) 0 [1, 2, 3]                        --          foldr (+) 0  [1,2,3]
(+) 1 (foldr (+) 0 [2, 3])                   -- 1 +        foldr (+) 0  [2,3]
(+) 1 ((+) 2 (foldr (+) 0 [3]))              -- 1 + (2 +     foldr (+) 0  [3])
(+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [])))       -- 1 + (2 + (3 +  foldr (+) 0 []))
(+) 1 ((+) 2 ((+) 3 0))                      -- 1 + (2 + (3 +             0   ))
```

The last line is equivalent to `1 + (2 + (3 + 0))`, because `((+) 3 0)` is the same as `(3 + 0)`.

Section 15.11: Zipping and Unzipping Lists

zip takes two lists and returns a list of corresponding pairs:

```
zip []     _      = []
zip _      []     = []
zip (a:as) (b:bs) = (a,b) : zip as bs

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

Zipping two lists with a function:

```
zipWith f  []     _      = []
zipWith f  _      []     = []
zipWith f  (a:as) (b:bs) = f a b : zipWith f as bs

> zipWith (+) [1,3,5] [2,4,6]
> [3,7,11]
```

Unzipping a list:

```
unzip = foldr (\(a,b) ~(as,bs) -> (a:as,b:bs)) ([],[])

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

第15.12节：foldl

这是左折叠的实现方式。注意step函数中参数的顺序与foldr（右折叠）相比是颠倒的：

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f acc []     =  acc
foldl f acc (x:xs) =  foldl f (f acc x) xs          -- = foldl f (acc `f` x) xs
```

左折叠foldl是向左结合的。也就是说：

```
foldl (+) 0 [1, 2, 3]     -- 等价于 ((0 + 1) + 2) + 3
```

原因是foldl的计算方式如下（看foldl的归纳步骤）：

```
foldl (+) 0 [1, 2, 3]                        --  foldl (+)    0   [ 1,   2,   3 ]
foldl (+) ((+) 0 1) [2, 3]                   --  foldl (+)   (0 + 1)   [ 2,   3 ]
foldl (+) ((+) ((+) 0 1) 2) [3]              --  foldl (+)  ((0 + 1) + 2)   [ 3 ]
foldl (+) ((+) ((+) ((+) 0 1) 2) 3) []       --  foldl (+) (((0 + 1) + 2) + 3) []
((+) ((+) ((+) 0 1) 2) 3)                    --            (((0 + 1) + 2) + 3)
```

最后一行等价于`((0 + 1) + 2) + 3`。因为一般来说`(f a b)`与`(a `f` b)`是相同的，所以`((+) 0 1)`特别等同于`(0 + 1)`。

Section 15.12: foldl

This is how the left fold is implemented. Notice how the order of the arguments in the step function is flipped compared to **foldr** (the right fold):

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f acc []     =  acc
foldl f acc (x:xs) =  foldl f (f acc x) xs          -- = foldl f (acc `f` x) xs
```

The left fold, **foldl**, associates to the left. That is:

```
foldl (+) 0 [1, 2, 3]     -- is equivalent to ((0 + 1) + 2) + 3
```

The reason is that **foldl** is evaluated like this (look at **foldl**'s inductive step):

```
foldl (+) 0 [1, 2, 3]                        --  foldl (+)    0   [ 1,   2,   3 ]
foldl (+) ((+) 0 1) [2, 3]                   --  foldl (+)   (0 + 1)   [ 2,   3 ]
foldl (+) ((+) ((+) 0 1) 2) [3]              --  foldl (+)  ((0 + 1) + 2)   [ 3 ]
foldl (+) ((+) ((+) ((+) 0 1) 2) 3) []       --  foldl (+) (((0 + 1) + 2) + 3) []
((+) ((+) ((+) 0 1) 2) 3)                    --            (((0 + 1) + 2) + 3)
```

The last line is equivalent to `((0 + 1) + 2) + 3`. This is because `(f a b)` is the same as `(a `f` b)` in general, and so `((+) 0 1)` is the same as `(0 + 1)` in particular.

第16章：排序算法

第16.1节：插入排序

```
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) | x < y     = x:y:ys
                | otherwise = y:(insert x ys)

isort :: Ord a => [a] -> [a]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

示例用法：

```
> isort [5,4,3,2,1]
```

结果：

```
[1,2,3,4,5]
```

第16.2节：排列排序

也称为bogosort。

```
import Data.List (permutations)

sorted :: Ord a => [a] -> Bool
sorted (x:y:xs) = x <= y && sorted (y:xs)
sorted _        = True

psort :: Ord a => [a] -> [a]
psort = head . filter sorted . permutations
```

极其低效（在当今的计算机上）。

第16.3节：归并排序

两个有序列表的有序合并

保留重复元素：

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) | x <= y    = x:merge xs (y:ys)
                    | otherwise = y:merge (x:xs) ys
```

自顶向下版本：

```
msort :: Ord a => [a] -> [a]
msort [] = []
msort [a] = [a]
msort xs = merge (msort (firstHalf xs)) (msort (secondHalf xs))
```

Chapter 16: Sorting Algorithms

Section 16.1: Insertion Sort

```
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) | x < y     = x:y:ys
                | otherwise = y:(insert x ys)

isort :: Ord a => [a] -> [a]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

Example use:

```
> isort [5,4,3,2,1]
```

Result:

```
[1,2,3,4,5]
```

Section 16.2: Permutation Sort

Also known as bogosort.

```
import Data.List (permutations)

sorted :: Ord a => [a] -> Bool
sorted (x:y:xs) = x <= y && sorted (y:xs)
sorted _        = True

psort :: Ord a => [a] -> [a]
psort = head . filter sorted . permutations
```

Extremely inefficient (on today's computers).

Section 16.3: Merge Sort

Ordered merging of two ordered lists

Preserving the duplicates:

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) | x <= y    = x:merge xs (y:ys)
                    | otherwise = y:merge (x:xs) ys
```

Top-down version:

```
msort :: Ord a => [a] -> [a]
msort [] = []
msort [a] = [a]
msort xs = merge (msort (firstHalf xs)) (msort (secondHalf xs))
```

```
firstHalf  xs = let { n = length xs } in take (div n 2) xs
secondHalf xs = let { n = length xs } in drop (div n 2) xs
```

这样定义是为了清晰，而不是为了效率。

示例用法：

```
> msort [3,1,4,5,2]
```

结果：

```
[1,2,3,4,5]
```

自底向上版本：

```
msort [] = []
msort xs = go [[x] | x <- xs]
    where
go [a] = a
go xs = go (pairs xs)
    pairs (a:b:t) = merge a b : pairs t
    pairs t = t
```

第16.4节：快速排序

```
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort [a | a <- xs, a < x]
                      ++ [x] ++
qsort [b | b <- xs, b >= x]
```

第16.5节：冒泡排序

```
bsort :: Ord a => [a] -> [a]
bsort s = case bsort' s of
t | t == s     -> t
                  | otherwise -> bsort t
  where bsort' (x:x2:xs) | x > x2     = x2:(bsort' (x:xs))
                         | otherwise = x:(bsort' (x2:xs))
        bsort' s = s
```

第16.6节：选择排序

选择排序 反复选择最小元素，直到列表为空。

```
import Data.List (minimum, delete)

ssort :: Ord t => [t] -> [t]
ssort [] = []
ssort xs = let { x = minimum xs }
           in  x : ssort (delete x xs)
```

```
firstHalf  xs = let { n = length xs } in take (div n 2) xs
secondHalf xs = let { n = length xs } in drop (div n 2) xs
```

It is defined this way for clarity, not for efficiency.

Example use:

```
> msort [3,1,4,5,2]
```

Result:

```
[1,2,3,4,5]
```

Bottom-up version:

```
msort [] = []
msort xs = go [[x] | x <- xs]
    where
    go [a] = a
    go xs = go (pairs xs)
    pairs (a:b:t) = merge a b : pairs t
    pairs t = t
```

Section 16.4: Quicksort

```
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort [a | a <- xs, a < x]
                      ++ [x] ++
               qsort [b | b <- xs, b >= x]
```

Section 16.5: Bubble sort

```
bsort :: Ord a => [a] -> [a]
bsort s = case bsort' s of
               t | t == s     -> t
                 | otherwise -> bsort t
  where bsort' (x:x2:xs) | x > x2     = x2:(bsort' (x:xs))
                         | otherwise = x:(bsort' (x2:xs))
        bsort' s = s
```

Section 16.6: Selection sort

Selection sort selects the minimum element, repeatedly, until the list is empty.

```
import Data.List (minimum, delete)

ssort :: Ord t => [t] -> [t]
ssort [] = []
ssort xs = let { x = minimum xs }
           in  x : ssort (delete x xs)
```

第17章：类型族

第17.1节：数据类型族

数据族可用于构建基于其类型参数具有不同实现的数据类型。

独立数据族

```
{-# LANGUAGE TypeFamilies #-}
data family List a
数据实例 列表 字符 = 空 | 连接 字符 (列表 字符)
数据实例 列表 () = 单元列表 整数
```

在上述声明中，Nil :: List Char，和 UnitList :: Int -> List ()

数据族也可以与类型类关联。这对于具有“辅助对象”的类型通常很有用，这些辅助对象是泛型类型类方法所必需的，但根据具体实例需要包含不同的信息。例如，列表中的索引位置只需要一个数字，而在树中则需要一个数字来指示每个节点的路径：

```
类 Container f 其中
  data Location f
get :: Location f -> f a -> Maybe a

实例 Container [] 其中
  数据 Location [] = ListLoc Int
  get (ListLoc i) xs
    | i < length xs  = Just $ xs!!i
    | 否则       = Nothing

实例 Container Tree 其中
  数据 Location Tree = ThisNode | NodePath Int (Location Tree)
  get ThisNode (Node x _) = Just x
get (NodePath i path) (Node _ sfo) = get path =<< get i sfo
```

第17.2节：类型同义词族

类型同义词族只是类型级函数：它们将参数类型与结果类型关联起来。这些有三种不同的形式。

封闭类型同义词族

它们的工作方式很像普通的值级Haskell函数：你指定一些子句，将某些类型映射到其他类型：

```
{-# LANGUAGE TypeFamilies #-}
type family Vanquisher a where
    Vanquisher Rock = Paper
Vanquisher Paper = Scissors
    Vanquisher Scissors = Rock

data Rock=Rock; data Paper=Paper; data Scissors=Scissors
```

开放类型同义词族

Chapter 17: Type Families

Section 17.1: Datatype Families

Data families can be used to build datatypes that have different implementations based on their type arguments.

Standalone data families

```
{-# LANGUAGE TypeFamilies #-}
data family List a
data instance List Char = Nil | Cons Char (List Char)
data instance List () = UnitList Int
```

In the above declaration, `Nil :: List Char`, and `UnitList :: Int -> List ()`

Associated data families

Data families can also be associated with typeclasses. This is often useful for types with “helper objects”, which are required for generic typeclass methods but need to contain different information depending on the concrete instance. For instance, indexing locations in a list just requires a single number, whereas in a tree you need a number to indicate the path at each node:

```
class Container f where
  data Location f
  get :: Location f -> f a -> Maybe a

instance Container [] where
  data Location [] = ListLoc Int
  get (ListLoc i) xs
    | i < length xs  = Just $ xs!!i
    | otherwise      = Nothing

instance Container Tree where
  data Location Tree = ThisNode | NodePath Int (Location Tree)
  get ThisNode (Node x _) = Just x
  get (NodePath i path) (Node _ sfo) = get path =<< get i sfo
```

Section 17.2: Type Synonym Families

Type synonym families are just type-level functions: they associate parameter types with result types. These come in three different varieties.

Closed type-synonym families

These work much like ordinary value-level Haskell functions: you specify some clauses, mapping certain types to others:

```
{-# LANGUAGE TypeFamilies #-}
type family Vanquisher a where
    Vanquisher Rock = Paper
    Vanquisher Paper = Scissors
    Vanquisher Scissors = Rock

data Rock=Rock; data Paper=Paper; data Scissors=Scissors
```

Open type-synonym families

它们更像类型类实例：任何人都可以在其他模块中添加更多子句。

```
type family DoubledSize w

type instance DoubledSize Word16 = Word32
type instance DoubledSize Word32 = Word64
-- 其他实例可能出现在其他模块中，但两个实例不能重叠
-- 以产生不同的结果。
```

类相关类型同义词

开放类型族也可以与实际类结合使用。通常在类似关联数据族的情况下，当某些类方法需要额外的辅助对象时会这样做，而这些辅助对象可以针对不同实例不同，但也可能被共享。一个很好的例子是VectorSpace类：

```
class VectorSpace v where
  type Scalar v :: *
  (*^) :: Scalar v -> v -> v

instance VectorSpace Double where
  type Scalar Double = Double
  μ *^ n = μ * n

instance VectorSpace (Double,Double) where
  type Scalar (Double,Double) = Double
  μ *^ (n,m) = (μ*n, μ*m)

instance VectorSpace (Complex Double) where
  type Scalar (Complex Double) = Complex Double
  μ *^ n = μ*n
```

注意在前两个实例中，Scalar的实现是相同的。使用关联数据族是不可能做到的：数据族是单射的，而类型同义词族不是。

虽然非单射性带来了一些可能性，如上所示，但它也使类型推断更加困难。例如，以下代码将无法通过类型检查：

```
class Foo a where
  type Bar a :: *
  bar :: a -> Bar a
instance Foo Int where
  type Bar Int = String
  bar = show
instance Foo Double where
  type Bar Double = Bool
  bar = (>0)

main = putStrLn (bar 1)
```

在这种情况下，编译器无法确定使用哪个实例，因为传给bar的参数本身就是多态的数字字面量。并且类型函数Bar无法在“逆方向”中解析，正是因为它不是单射†，因此不可逆（可能存在多个类型满足Bar a = String）。

†仅凭这两个实例，它实际上是单射的，但编译器无法知道以后是否有人会添加更多实例，从而破坏该行为。

These work more like typeclass instances: anybody can add more clauses in other modules.

```
type family DoubledSize w

type instance DoubledSize Word16 = Word32
type instance DoubledSize Word32 = Word64
-- Other instances might appear in other modules, but two instances cannot overlap
-- in a way that would produce different results.
```

Class-associated type synonyms

An open type family can also be combined with an actual class. This is usually done when, like with associated data families, some class method needs additional helper objects, and these helper objects *can* be different for different instances but may possibly also shared. A good example is VectorSpace class:

```
class VectorSpace v where
  type Scalar v :: *
  (*^) :: Scalar v -> v -> v

instance VectorSpace Double where
  type Scalar Double = Double
  μ *^ n = μ * n

instance VectorSpace (Double,Double) where
  type Scalar (Double,Double) = Double
  μ *^ (n,m) = (μ*n, μ*m)

instance VectorSpace (Complex Double) where
  type Scalar (Complex Double) = Complex Double
  μ *^ n = μ*n
```

Note how in the first two instances, the implementation of `Scalar` is the same. This would not be possible with an associated data family: data families are injective, type-synonym families aren't.

While non-injectivity opens up some possibilities like the above, it also makes type inference more difficult. For instance, the following will not typecheck:

```
class Foo a where
  type Bar a :: *
  bar :: a -> Bar a
instance Foo Int where
  type Bar Int = String
  bar = show
instance Foo Double where
  type Bar Double = Bool
  bar = (>0)

main = putStrLn (bar 1)
```

In this case, the compiler can't know what instance to use, because the argument to `bar` is itself just a polymorphic `Num` literal. And the type function `Bar` can't be resolved in “inverse direction”, precisely because it's not injective† and hence not invertible (there could be more than one type with `Bar a = String`).

†With only these two instances, it *is* actually injective, but the compiler can't know somebody won't add more instances later on and thereby break the behaviour.

第17.3节：单射性

类型族不一定是单射的。因此，我们无法从应用中推断参数。例如，在servant中，给定类型Server a，我们无法推断类型a。为了解决这个问题，我们可以使用Proxy。例如，在servant中，serve函数的类型是... Proxy a -> Server a -> ...。我们可以从Proxy a推断出a，因为Proxy是由data定义的，且是单射的。

Section 17.3: Injectivity

Type Families are not necessarily injective. Therefore, we cannot infer the parameter from an application. For example, in `servant`, given a type `Server a` we cannot infer the type `a`. To solve this problem, we can use `Proxy`. For example, in `servant`, the `serve` function has type `... Proxy a -> Server a -> ...`. We can infer `a` from `Proxy a` because `Proxy` is defined by **`data`** which is injective.

第18章：单子（Monad）

单子是一种可组合动作的数据类型。单子类（Monad）是类型构造器的类，其值表示此类动作。也许IO是最为人熟知的：一个IO a的值是“从真实世界中获取一个a值的配方”。

我们说类型构造器m（例如[]或Maybe）形成单子，如果存在一个满足某些关于动作组合定律的instance Monad m。然后我们可以将m a视为“结果类型为a的动作”。

第18.1节：单子的定义

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

处理单子最重要的函数是bind操作符>>=：

```
(>>=) :: m a -> (a -> m b) -> m b
```

- *把 m a 想象成 “一种带有 a 结果的动作”。*
- *把 a -> m b 想象成 “一种（依赖于 a 参数的）动作，带有 b 结果。”。*

>>= 通过将第一个动作的结果传递给第二个动作，将两个动作连接起来。

由 Monad 定义的另一个函数是：

```
return :: a -> m a
```

它的名字不太恰当：这个 return 与命令式编程语言中的 return 关键字毫无关系。

return x 是产生 x 作为结果的平凡动作。（它的平凡性体现在以下方面：）

```
return x >>= f       ≡  f x      -- “左单位元”单子定律
       x >>= return  ≡  x        -- “右单位元”单子定律
```

第18.2节：没有通用方法可以从单子计算中提取值

你可以将值包装成动作，并将一个计算的结果传递给另一个计算：

```
return :: Monad m => a -> m a
(>>=)  :: Monad m => m a -> (a -> m b) -> m b
```

然而，Monad 的定义并不保证存在类型为 Monad m => m a -> a 的函数。

这意味着通常情况下，无法从计算中提取值（即“解包”它）。许多实例都是这种情况：

```
extract :: Maybe a -> a
extract (Just x) = x          -- 当然，这个可行，但...
extract Nothing  = undefined  -- 我们无法从失败中提取值。
```

Chapter 18: Monads

A monad is a data type of composable actions. Monad is the class of type constructors whose values represent such actions. Perhaps IO is the most recognizable one: a value of IO a is a "recipe for retrieving an a value from the real world".

We say a type constructor m (such as [] or Maybe) *forms a monad* if there is an instance Monad m satisfying certain laws about composition of actions. We can then reason about m a as an "action whose result has type a".

Section 18.1: Definition of Monad

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

The most important function for dealing with monads is the **bind operator** >>=:

```
(>>=) :: m a -> (a -> m b) -> m b
```

- Think of m a as *"an action with an a result"*.
- Think of a -> m b as *“an action (depending on an a parameter) with a b result.”*.

>>= **sequences two actions together by piping the result from the first action to the second.**

The other function defined by Monad is:

```
return :: a -> m a
```

Its name is unfortunate: this return has nothing to do with the return keyword found in imperative programming languages.

return x **is the trivial action yielding x as its result.** (It is trivial in the following sense:)

```
return x >>= f       ≡  f x      -- “left identity” monad law
       x >>= return  ≡  x        -- “right identity” monad law
```

Section 18.2: No general way to extract value from a monadic computation

You can wrap values into actions and pipe the result of one computation into another:

```
return :: Monad m => a -> m a
(>>=)  :: Monad m => m a -> (a -> m b) -> m b
```

However, the definition of a Monad doesn’t guarantee the existence of a function of type Monad m => m a -> a.

That means there is, in general, **no way to extract a value from a computation** (i.e. “unwrap” it). This is the case for many instances:

```
extract :: Maybe a -> a
extract (Just x) = x          -- Sure, this works, but...
extract Nothing  = undefined  -- We can't extract a value from failure.
```

具体来说，不存在函数 IO a -> a，这常常让初学者感到困惑；参见此示例。

第18.3节：Monad 作为 Applicative 的子类

从 GHC 7.10 开始，`Applicative` 是 `Monad` 的超类（即，每个 `Monad` 类型也必须是一个 Applicative）。所有 Applicative 的方法（pure, <*>）都可以用 Monad 的方法（`return`, `>>=`）来实现。

显然，pure 和 return 具有相同的作用，因此 pure = return。对于 <*> 的定义也相对明确：

```
mf <*> mx = do { f <- mf; x <- mx; return (f x) }
        -- = mf >>= (|f -> mx >>= (|x -> return (f x)))
        -- = [r    | f <- mf, x <- mx, r <- return (f x)]   -- 使用 MonadComprehensions
        -- = [f x | f <- mf, x <- mx]
```

此函数在标准库中定义为ap。

因此，如果你已经为某个类型定义了`Monad`的实例，那么通过定义，你实际上可以"免费"获得该类型的`Applicative`实例

```
instance Applicative < type > where
    pure  = return
    (<*>) = ap
```

与单子定律一样，这些等价关系并非强制执行，但开发者应确保它们始终得到遵守。

第18.4节：Maybe单子

`Maybe` 用于表示可能为空的值——类似于其他语言中的 `null`。通常它用作可能以某种方式失败的函数的输出类型。

考虑以下函数：

```
halve :: Int -> Maybe Int
halve x
  | even x = Just (x `div` 2)
  | odd x  = Nothing
```

将 halve 看作一个动作，依赖于一个 Int，尝试将整数减半，如果是奇数则失败。

我们如何对一个整数进行三次 halve 操作？

```
takeOneEighth :: Int -> Maybe Int            -- （阅读"do"子部分后：）
takeOneEighth x =
  case halve x of                               --  do {
    Nothing -> Nothing
Just oneHalf ->                             --      oneHalf     <- halve x
      case halve oneHalf of
Nothing -> Nothing
Just oneQuarter ->                        --     oneQuarter <- halve oneHalf
          case halve oneQuarter of
Nothing -> Nothing                    --     oneEighth   <- halve oneQuarter
            Just oneEighth ->
仅为八分之一                  --     返回八分之一 }
```

Specifically, there is no function `IO a -> a`, which often confuses beginners; see this example.

Section 18.3: Monad as a Subclass of Applicative

As of GHC 7.10, `Applicative` is a superclass of `Monad` (i.e., every type which is a `Monad` must also be an `Applicative`). All the methods of `Applicative` (`pure`, `<*>`) can be implemented in terms of methods of `Monad` (`return`, `>>=`).

It is obvious that `pure` and `return` serve equivalent purposes, so `pure = return`. The definition for `<*>` is too relatively clear:

```
mf <*> mx = do { f <- mf; x <- mx; return (f x) }
       -- = mf >>= (\f -> mx >>= (\x -> return (f x)))
       -- = [r   | f <- mf, x <- mx, r <- return (f x)]   -- with MonadComprehensions
       -- = [f x | f <- mf, x <- mx]
```

This function is defined as `ap` in the standard libraries.

Thus if you have already defined an instance of `Monad` for a type, you effectively can get an instance of `Applicative` for it "for free" by defining

```
instance Applicative < type > where
    pure  = return
    (<*>) = ap
```

As with the monad laws, these equivalencies are not enforced, but developers should ensure that they are always upheld.

Section 18.4: The Maybe monad

`Maybe` is used to represent possibly empty values - similar to `null` in other languages. Usually it is used as the output type of functions that can fail in some way.

Consider the following function:

```
halve :: Int -> Maybe Int
halve x
  | even x = Just (x `div` 2)
  | odd x  = Nothing
```

Think of `halve` as an action, depending on an `Int`, that tries to halve the integer, failing if it is odd.

How do we `halve` an integer three times?

```
takeOneEighth :: Int -> Maybe Int                -- (after you read the 'do' sub-section:)
takeOneEighth x =
  case halve x of                                  --  do {
    Nothing -> Nothing
    Just oneHalf ->                                --      oneHalf    <- halve x
      case halve oneHalf of
        Nothing -> Nothing
        Just oneQuarter ->                         --      oneQuarter <- halve oneHalf
          case halve oneQuarter of
            Nothing -> Nothing                     --      oneEighth  <- halve oneQuarter
            Just oneEighth ->
              Just oneEighth                       --      return oneEighth }
```

- takeOneEighth 是一个由三个 halve 步骤串联而成的 sequence。
- 如果一个 halve 步骤失败，我们希望整个组合 takeOneEighth 也失败。
- 如果一个 halve 步骤成功，我们希望将其结果传递下去。

```
实例 Monad Maybe 定义如下
  -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  Nothing >>= f  = Nothing                            -- infixl 1 >>=
  Just x  >>= f  = Just (f x)                      -- 另外，f =<< m = m >>= f

  -- return :: a -> Maybe a
  return x       = Just x
```

现在我们可以写：

```
takeOneEighth :: Int -> Maybe Int
takeOneEighth x = halve x >>= halve >>= halve              -- 或者，
    -- 返回 x >>= halve >>= halve >>= halve             -- 其解析为
    -- (((return x) >>= halve) >>= halve) >>= halve       -- 也可以写成
    -- (halve =<<) . (halve =<<) . (halve =<<) $ return x    -- 或者等价地写成
    --   halve <=<      halve <=<      halve       $          x
```

Kleisli 组合 <=< 定义为 (g <=< f) x = g =<< f x，或者等价地为 (f >=> g) x = f x >>= g。利用它，上述定义变为

```
takeOneEighth :: Int -> Maybe Int
takeOneEighth = halve <=< halve <=< halve                   -- 右结合优先级 1 <=<
        -- 或者等价地,
        --      halve >=> halve >=> halve                -- 右结合优先级 1 >=>
```

每个单子（即每个 Monad 类型类的实例）都应遵守三个单子定律，分别是
Monad 类型类：

```
1.   return x >>= f  =  f x
2.     m >>= return  =  m
3. (m >>= g) >>= h  =  m >>= (\y -> g y >>= h)
```

其中 m 是一个单子，f 的类型是 a -> m b，g 的类型是 b -> m c。

或者等价地，使用上述定义的>=> Kleisli 组合运算符：

```
1.    return >=> g   =  g                    -- do { y <- return x ; g y } == g x
2.    f >=> return   =  f                   -- do { y <- f x ; return y } == f x
3. (f >=> g) >=> h   =  f >=> (g >=> h)       -- do { z <- do { y <- f x; g y } ; h z }
                                              --  == do { y <- f x ; do { z <- g y; h z } }
```

遵守这些定律使得推理单子变得更加容易，因为它保证了使用单子函数及其组合的行为是合理的，类似于其他单子。

让我们检查一下Maybe单子是否遵守这三个单子定律。

1. **左单位定律 - return x >>= f = f x**

```
return z >>= f
= (Just z) >>= f
= f z
```

- `takeOneEighth` is a *sequence* of three `halve` steps chained together.
- If a `halve` step fails, we want the whole composition `takeOneEighth` to fail.
- If a `halve` step succeeds, we want to pipe its result forward.

```
instance Monad Maybe where
  -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  Nothing >>= f  = Nothing                            -- infixl 1 >>=
  Just x  >>= f  = Just (f x)                         -- also, f =<< m = m >>= f

  -- return :: a -> Maybe a
  return x       = Just x
```

and now we can write:

```
takeOneEighth :: Int -> Maybe Int
takeOneEighth x = halve x >>= halve >>= halve             -- or,
    -- return x >>= halve >>= halve >>= halve            -- which is parsed as
    -- (((return x) >>= halve) >>= halve) >>= halve      -- which can also be written as
    -- (halve =<<) . (halve =<<) . (halve =<<) $ return x    -- or, equivalently, as
    --  halve <=<     halve <=<     halve      $        x
```

Kleisli composition `<=<` is defined as `(g <=< f) x = g =<< f x`, or equivalently as `(f >=> g) x = f x >>= g`. With it the above definition becomes just

```
takeOneEighth :: Int -> Maybe Int
takeOneEighth = halve <=< halve <=< halve                 -- infixr 1 <=<
        -- or, equivalently,
        --      halve >=> halve >=> halve                -- infixr 1 >=>
```

There are three monad laws that should be obeyed by every monad, that is every type which is an instance of the `Monad` typeclass:

```
1.  return x >>= f  =  f x
2.    m >>= return  =  m
3. (m >>= g) >>= h  =  m >>= (\y -> g y >>= h)
```

where `m` is a monad, `f` has type `a -> m b` and `g` has type `b -> m c`.

Or equivalently, using the `>=>` Kleisli composition operator defined above:

```
1.    return >=> g  =  g                   -- do { y <- return x ; g y } == g x
2.    f >=> return  =  f                   -- do { y <- f x ; return y } == f x
3. (f >=> g) >=> h  =  f >=> (g >=> h)     -- do { z <- do { y <- f x; g y } ; h z }
                                           --  == do { y <- f x ; do { z <- g y; h z } }
```

Obeying these laws makes it a lot easier to reason about the monad, because it guarantees that using monadic functions and composing them behaves in a reasonable way, similar to other monads.

Let's check if the `Maybe` monad obeys the three monad laws.

1. **The left identity law** - `return x >>= f = f x`

```
return z >>= f
= (Just z) >>= f
= f z
```

2. **右单位定律 - m >>= return = m**

- Just 数据构造器

```
Just z >>= return
= return z
= 仅是 z
```

- 无 数据构造器

```
无 >>= 返回
= 无
```

3. 结合律 - (m >>= f) >>= g = m >>= (\x -> f x >>= g)

- Just 数据构造器

```
-- 左侧
((仅是 z) >>= f) >>= g
= f z >>= g

-- 右侧
(仅是 z) >>= (\x -> f x >>= g)
(\x -> f x >>= g) z
= f z >>= g
```

- 无 数据构造器

```
-- 左侧
(无 >>= f) >>= g
= 无 >>= g
= 无

-- 右侧
无 >>= (\x -> f x >>= g)
= 无
```

第18.5节：IO单子

无法从类型为 a 的表达式中获取类型为 IO a 的值，而且也不应该有这种方式。这实际上是单子被用来模拟 IO 的一个重要原因。

类型为 IO a 的表达式可以被看作表示一个可以与真实世界交互的动作，如果执行，将产生类型为 a 的结果。例如，前导库中的函数 getLine :: IO String 并不意味着 getLine 底层有某个具体的字符串可以提取——它意味着 getLine 表示从标准输入获取一行的动作。

不足为奇的是，main :: IO ()，因为Haskell程序确实表示一个与真实世界交互的计算/动作。

由于 IO 是一个单子，你 可以 对类型为 IO a 的表达式做的事情：

- 使用 (>>) 将两个动作顺序连接，产生一个新的动作，该动作先执行第一个动作，丢弃其产生的任何值，然后执行第二个动作。

```
-- 将字符串 "Hello" 和 "World" 依次打印到标准输出
putStrLn "Hello" >> putStrLn "World"
```

2. **The right identity law** - m >>= return = m

- Just data constructor

```
Just z >>= return
= return z
= Just z
```

- Nothing data constructor

```
Nothing >>= return
= Nothing
```

3. **The associativity law** - (m >>= f) >>= g = m >>= (\x -> f x >>= g)

- Just data constructor

```
-- Left-hand side
((Just z) >>= f) >>= g
= f z >>= g

-- Right-hand side
(Just z) >>= (\x -> f x >>= g)
(\x -> f x >>= g) z
= f z >>= g
```

- Nothing data constructor

```
-- Left-hand side
(Nothing >>= f) >>= g
= Nothing >>= g
= Nothing

-- Right-hand side
Nothing >>= (\x -> f x >>= g)
= Nothing
```

Section 18.5: IO monad

There is no way to get a value of type a out of an expression of type IO a and there shouldn't be. This is actually a large part of why monads are used to model IO.

An expression of type IO a can be thought of as representing an action that can interact with the real world and, if executed, would result in something of type a. For example, the function getLine :: IO String from the prelude doesn't mean that underneath getLine there is some specific string that I can extract - it means that getLine represents the action of getting a line from standard input.

Not surprisingly, main :: IO () since a Haskell program does represent a computation/action that interacts with the real world.

The things you *can* do to expressions of type IO a because IO is a monad:

- Sequence two actions using (>>) to produce a new action that executes the first action, discards whatever value it produced, and then executes the second action.

```
-- print the lines "Hello" then "World" to stdout
putStrLn "Hello" >> putStrLn "World"
```

- 有时你不想丢弃第一个动作产生的值——你实际上希望它被传递到第二个动作中。为此，我们有>>=。对于IO，它的类型是(>>=) :: IO a -> (a -> IO b) -> IOb。

```haskell
-- 从标准输入读取一行并打印出来
getLine >>= putStrLn
```

- 将一个普通值转换成一个动作，该动作立即返回你给它的值。这个函数在你开始使用do记法之前不太明显有用。

```haskell
-- 创建一个只返回5的动作
return 5
```

更多关于IO单子的内容请参见Haskell Wiki here。

第18.6节：列表单子

列表构成一个单子。它们有一个等价的单子实例定义如下：

instance Monad [] where return x = [x] xs >>= f = concat (map f xs)

我们可以用它们来模拟计算中的非确定性。当我们使用 xs >>= f时，函数 f :: a -> [b] 映射到列表 xs 上，得到一个由 f 对 xs 中每个元素应用结果组成的列表的列表，然后将所有结果列表连接成一个包含所有结果的列表。举例来说，我们使用 do-notation 计算两个非确定性数字的和，和由所有成对数字的和的列表表示。
来自两个列表的整数，每个列表表示一个非确定性数字的所有可能值：

```haskell
sumnd xs ys = do
  x <- xs
y <- ys
  return (x + y)
```

或者等价地，使用 liftM2 在 Control.Monad 中：

```haskell
sumnd = liftM2 (+)
```

我们得到：

```
> sumnd [1,2,3] [0,10]
[1,11,2,12,3,13]
```

第18.7节：do-notation

do-notation 是单子（monads）的语法糖。规则如下：

do x <- mx do x <- mx y <- my 等价于 do y <- my ... ... do let a = b let a = b in ... 等价于 do ... do m m >> (e 等价于 e) do x <- m m >>= (\x -> e 等价于 e) do m 等价于 m

例如，这些定义是等价的：

```haskell
example :: IO Integer
example =
  putStrLn "你叫什么名字？" >> (
    getLine >>= (\name ->
```

- Sometimes you don't want to discard the value that was produced in the first action - you'd actually like it to be fed into a second action. For that, we have >>=. For IO, it has type (>>=) :: IO a -> (a -> IO b) -> IO b.

```haskell
-- get a line from stdin and print it back out
getLine >>= putStrLn
```

- Take a normal value and convert it into an action which just immediately returns the value you gave it. This function is less obviously useful until you start using do notation.

```haskell
-- make an action that just returns 5
return 5
```

More from the Haskell Wiki on the IO monad here.

Section 18.6: List Monad

The lists form a monad. They have a monad instantiation equivalent to this one:

instance Monad [] where return x = [x] xs >>= f = concat (map f xs)

We can use them to emulate non-determinism in our computations. When we use xs >>= f, the function f :: a -> [b] is mapped over the list xs, obtaining a list of lists of results of each application of f over each element of xs, and all the lists of results are then concatenated into one list of all the results. As an example, we compute a sum of two non-deterministic numbers using **do-notation**, the sum being represented by list of sums of all pairs of integers from two lists, each list representing all possible values of a non-deterministic number:

```haskell
sumnd xs ys = do
  x <- xs
  y <- ys
  return (x + y)
```

Or equivalently, using liftM2 in Control.Monad:

```haskell
sumnd = liftM2 (+)
```

we obtain:

```
> sumnd [1,2,3] [0,10]
[1,11,2,12,3,13]
```

Section 18.7: do-notation

do-notation is syntactic sugar for monads. Here are the rules:

do x <- mx do x <- mx y <- my is equivalent to do y <- my ... ... do let a = b let a = b in ... is equivalent to do ... do m m >> (e is equivalent to e) do x <- m m >>= (\x -> e is equivalent to e) do m is equivalent to m

For example, these definitions are equivalent:

```haskell
example :: IO Integer
example =
  putStrLn "What's your name?" >> (
    getLine >>= (\name ->
      putStrLn ("你好, " ++ name ++ "。" ) >> (
        putStrLn "我们应该返回什么？" >> (
          getLine >>= (\line ->
            let n = (read line :: Integer) in
              return (n + n))))))
```

```
example :: IO Integer
example = do
  putStrLn "你叫什么名字？"
  name <- getLine
  putStrLn ("你好, " ++ name ++ "。" )
  putStrLn "我们应该返回什么？"
  line <- getLine
  let n = (read line :: Integer)
  return (n + n)
```

```
      putStrLn ("Hello, " ++ name ++ ".") >> (
        putStrLn "What should we return?" >> (
          getLine >>= (\line ->
            let n = (read line :: Integer) in
              return (n + n))))))
```

```
example :: IO Integer
example = do
  putStrLn "What's your name?"
  name <- getLine
  putStrLn ("Hello, " ++ name ++ ".")
  putStrLn "What should we return?"
  line <- getLine
  let n = (read line :: Integer)
  return (n + n)
```

第19章：堆栈

第19.1节：使用Stack进行性能分析

通过stack为项目配置性能分析。首先使用--profile标志构建项目：

```
stack build --profile
```

为了使其生效，cabal文件中不需要GHC标志（如-prof）。stack会自动为项目中的库和可执行文件启用性能分析。下次运行项目中的可执行文件时，可以使用常用的+RTS标志：

```
stack exec -- my-bin +RTS -p
```

第19.2节：结构

文件结构

一个简单的项目包含以下文件：

```
➜  helloworld ls
LICENSE          Setup.hs         helloworld.cabal src                 stack.yaml
```

在文件夹 `src` 中有一个名为 `Main.hs` 的文件。这是 `helloworld` 项目的"起点"。默认情况下 `Main.hs` 包含一个简单的"Hello，World!"程序。

Main.hs

```
模块 Main 所在

main :: IO ()
main = do
   putStrLn "hello world"
```

运行程序

确保你在目录 helloworld 中，然后运行：

```
stack build # 编译程序
stack exec helloworld # 运行程序
# 输出 "hello world"
```

第19.3节：构建并运行Stack项目

在这个例子中，我们的项目名称是"helloworld"，它是通过 `stack new helloworld simple`

创建的。首先我们必须用 `stack build` 构建项目，然后可以用它来运行

```
stack exec helloworld-exe
```

第19.4节：查看依赖关系

要查找项目直接依赖的包，可以简单地使用以下命令：

Chapter 19: Stack

Section 19.1: Profiling with Stack

Configure profiling for a project via `stack`. First build the project with the `--profile` flag:

```
stack build --profile
```

GHC flags are not required in the cabal file for this to work (like `-prof`). `stack` will automatically turn on profiling for both the library and executables in the project. The next time an executable runs in the project, the usual `+RTS` flags can be used:

```
stack exec -- my-bin +RTS -p
```

Section 19.2: Structure

File structure

A simple project has the following files included in it:

```
➜  helloworld ls
LICENSE          Setup.hs         helloworld.cabal src                 stack.yaml
```

In the folder `src` there is a file named `Main.hs`. This is the "starting point" of the `helloworld` project. By default `Main.hs` contains a simple "Hello, World!" program.

Main.hs

```
module Main where

main :: IO ()
main = do
  putStrLn "hello world"
```

Running the program

Make sure you are in the directory `helloworld` and run:

```
stack build # Compile the program
stack exec helloworld # Run the program
# prints "hello world"
```

Section 19.3: Build and Run a Stack Project

In this example our project name is "helloworld" which was created with `stack new helloworld simple`

First we have to build the project with `stack build` and then we can run it with

```
stack exec helloworld-exe
```

Section 19.4: Viewing dependencies

To find out what packages your project directly depends on, you can simply use this command:

```
stack list-dependencies
```

通过这种方式，你可以知道stack实际拉取了哪些版本的依赖。

Haskell项目经常会间接引入许多库，有时这些外部依赖会引发需要追踪的问题。如果你发现有一个不受欢迎的外部依赖想要定位，可以通过整个依赖图进行grep，找出究竟是哪个依赖最终引入了该不需要的包：

```
stack dot --external | grep template-haskell
```

stack dot 会以文本形式打印出依赖图，便于搜索。它也可以被可视化查看：

```
stack dot --external | dot -Tpng -o my-project.png
```

如果需要，你还可以设置依赖图的深度：

```
stack dot --external --depth 3 | dot -Tpng -o my-project.png
```

第19.5节：安装Stack

通过运行命令

```
stack install
```

Stack会将可执行文件复制到文件夹

```
/Users/<yourusername>/.local/bin/
```

第19.6节：安装Stack

Mac OSX

使用Homebrew：

```
brew install haskell-stack
```

第19.7节：创建一个简单项目

要创建一个名为 helloworld 的项目，请运行：

```
stack new helloworld simple
```

这将创建一个名为 helloworld 的目录，里面包含 Stack 项目所需的文件。

第19.8节：Stackage 包和更改 LTS（解析器）版本

Stackage 是 Haskell 包的仓库。我们可以将这些包添加到 stack 项目中。

向项目中添加 lens。

在 stack 项目中，有一个名为 stack.yaml 的文件。在 stack.yaml 中有一段内容如下：

```
stack list-dependencies
```

This way you can find out what version of your dependencies where actually pulled down by stack.

Haskell projects frequently find themselves pulling in a lot of libraries indirectly, and sometimes these external dependencies cause problems that you need to track down. If you find yourself with a rogue external dependency that you'd like to identify, you can grep through the entire dependency graph and identify which of your dependencies is ultimately pulling in the undesired package:

```
stack dot --external | grep template-haskell
```

`stack dot` prints out a dependency graph in text form that can be searched. It can also be viewed:

```
stack dot --external | dot -Tpng -o my-project.png
```

You can also set the depth of the dependency graph if you want:

```
stack dot --external --depth 3 | dot -Tpng -o my-project.png
```

Section 19.5: Stack install

By running the command

```
stack install
```

Stack will copy a executable file to the folder

```
/Users/<yourusername>/.local/bin/
```

Section 19.6: Installing Stack

Mac OSX

Using Homebrew:

```
brew install haskell-stack
```

Section 19.7: Creating a simple project

To create a project called **helloworld** run:

```
stack new helloworld simple
```

This will create a directory called `helloworld` with the files necessary for a Stack project.

Section 19.8: Stackage Packages and changing the LTS (resolver) version

Stackage is a repository for Haskell packages. We can add these packages to a stack project.

Adding lens to a project.

In a stack project, there is a file called `stack.yaml`. In `stack.yaml` there is a segment that looks like:

```
resolver: lts-6.8
```

Stackage 为每个 lts 版本维护一个包列表。在我们的例子中，我们需要 lts-6.8 的包列表。要查找这些包，请访问：

```
https://www.stackage.org/lts-6.8 # 如果使用不同版本，请将 6.8 更改为正确的
resolver 版本号。
```

查看软件包时，有一个Lens-4.13。

我们现在可以通过修改helloworld.cabal的部分来添加语言包：

```
build-depends: base >= 4.7 && < 5
```

改为：

```
build-depends: base >= 4.7 && 5,
                 lens == 4.13
```

显然，如果我们想更改为更新的LTS（发布后），只需更改解析器版本号，例如：

```
resolver: lts-6.9
```

下一次执行stack build时，Stack将使用LTS 6.9版本，因此会下载一些新的依赖项。

```
resolver: lts-6.8
```

Stackage keeps a list of packages for every revision of `lts`. In our case we want the list of packages for `lts-6.8` To find these packages visit:

```
https://www.stackage.org/lts-6.8 # if a different version is used, change 6.8 to the correct
resolver number.
```

Looking through the packages, there is a Lens-4.13.

We can now add the language package by modifying the section of `helloworld.cabal`:

```
  build-depends: base >= 4.7 && < 5
```

to:

```
  build-depends: base >= 4.7 && 5,
                 lens == 4.13
```

Obviously, if we want to change a newer LTS (after it's released), we just change the resolver number, eg.:

```
resolver: lts-6.9
```

With the next `stack build` Stack will use the LTS 6.9 version and hence download some new dependencies.

第20章：广义代数数据类型

第20.1节：基本用法

当启用GADTs扩展时，除了常规的数据声明外，你还可以按如下方式声明广义代数数据类型：

```
data DataType a 其中
Constr1 :: Int -> a -> Foo a -> DataType a
    Constr2 :: Show a => a -> DataType a
Constr3 :: DataType Int
```

GADT 声明明确列出了数据类型所有构造函数的类型。与常规数据类型声明不同，构造函数的类型可以是任何 N 元（包括零元）函数，最终结果是应用于某些参数的数据类型。

在本例中，我们声明类型 DataType 有三个构造函数：Constr1、Constr2 和 Constr3。

Constr1 构造函数与使用常规数据声明声明的构造函数没有区别：data DataType a = Constr1 Int a (Foo a) | ...

然而，Constr2 要求 a 必须是 Show 的实例，因此在使用该构造函数时必须存在该实例。另一方面，在对其进行模式匹配时，a 是 Show 实例的事实会进入作用域，因此你可以写：

```
foo :: DataType a -> String
foo val = case val of
Constr2 x -> show x
   ...
```

注意，Show a 约束不会出现在函数的类型中，只在 -> 右侧的代码中可见。

Constr3的类型是DataType Int，这意味着每当类型为DataType a的值是一个Constr3时，就知道a ~ Int。这个信息也可以通过模式匹配恢复。

Chapter 20: Generalized Algebraic Data Types

Section 20.1: Basic Usage

When the GADTs extension is enabled, besides regular data declarations, you can also declare generalized algebraic datatypes as follows:

```
data DataType a where
    Constr1 :: Int -> a -> Foo a -> DataType a
    Constr2 :: Show a => a -> DataType a
    Constr3 :: DataType Int
```

A GADT declaration lists the types of all constructors a datatype has, explicitly. Unlike regular datatype declarations, the type of a constructor can be any N-ary (including nullary) function that ultimately results in the datatype applied to some arguments.

In this case we've declared that the type DataType has three constructors: Constr1, Constr2 and Constr3.

The Constr1 constructor is no different from one declared using a regular data declaration: data DataType a = Constr1 Int a (Foo a) | ...

Constr2 however requires that a has an instance of Show, and so when using the constructor the instance would need to exist. On the other hand, when pattern-matching on it, the fact that a is an instance of Show comes into scope, so you can write:

```
foo :: DataType a -> String
foo val = case val of
    Constr2 x -> show x
    ...
```

Note that the Show a constraint doesn't appear in the type of the function, and is only visible in the code to the right of ->.

Constr3 has type DataType Int, which means that whenever a value of type DataType a is a Constr3, it is known that a ~ Int. This information, too, can be recovered with a pattern match.

第21章：递归模式

第21.1节：不动点

Fix接受一个“模板”类型并绑定递归结点，将模板像千层面一样层叠起来。

```
newtype Fix f = Fix { unFix :: f (Fix f) }
```

在一个Fix f内部，我们找到模板f的一层。为了填充f的参数，Fix f插入了它自身。所以当你查看模板f时，会发现Fix f的递归出现。

下面是一个典型递归数据类型如何被转换到我们的模板和不动点框架中的方法。我们去除类型的递归出现，并用 r 参数标记它们的位置。

```
{-# LANGUAGE DeriveFunctor #-}

-- 自然数
-- data Nat = Zero | Suc Nat
data NatF r = Zero_ | Suc_ r deriving Functor
type Nat = Fix NatF

zero :: 自然数
zero = 固定 零_
suc :: 自然数 -> 自然数
suc n = 固定 (后继_ n)


-- 列表：注意额外的类型参数 a
-- 数据类型 List a = Nil | Cons a (List a)
data ListF a r = Nil_ | Cons_ a r deriving Functor
type List a = Fix (ListF a)

nil :: List a
nil = Fix Nil_
cons :: a -> List a -> List a
cons x xs = Fix (Cons_ x xs)


-- 二叉树：注意两个递归出现
-- 数据类型 Tree a = Leaf | Node (Tree a) a (Tree a)
data TreeF a r = Leaf_ | Node_ r a r deriving Functor
type Tree a = Fix (TreeF a)

leaf :: Tree a
leaf = Fix Leaf_
node :: Tree a -> a -> Tree a -> Tree a
node l x r = Fix (Node_ l x r)
```

第21.2节：原始递归

参数形态（Paramorphisms）模拟原始递归。在折叠的每次迭代中，折叠函数接收子树以进行进一步处理。

```
para :: Functor f => (f (Fix f, a) -> a) -> Fix f -> a
para f = f . fmap (\x -> (x, para f x)) . unFix
```

Prelude中的ails可以被建模为参数形态。

Chapter 21: Recursion Schemes

Section 21.1: Fixed points

Fix takes a "template" type and ties the recursive knot, layering the template like a lasagne.

```
newtype Fix f = Fix { unFix :: f (Fix f) }
```

Inside a Fix f we find a layer of the template f. To fill in f's parameter, Fix f plugs in *itself*. So when you look inside the template f you find a recursive occurrence of Fix f.

Here is how a typical recursive datatype can be translated into our framework of templates and fixed points. We remove recursive occurrences of the type and mark their positions using the r parameter.

```
{-# LANGUAGE DeriveFunctor #-}

-- natural numbers
-- data Nat = Zero | Suc Nat
data NatF r = Zero_ | Suc_ r deriving Functor
type Nat = Fix NatF

zero :: Nat
zero = Fix Zero_
suc :: Nat -> Nat
suc n = Fix (Suc_ n)


-- lists: note the additional type parameter a
-- data List a = Nil | Cons a (List a)
data ListF a r = Nil_ | Cons_ a r deriving Functor
type List a = Fix (ListF a)

nil :: List a
nil = Fix Nil_
cons :: a -> List a -> List a
cons x xs = Fix (Cons_ x xs)


-- binary trees: note two recursive occurrences
-- data Tree a = Leaf | Node (Tree a) a (Tree a)
data TreeF a r = Leaf_ | Node_ r a r deriving Functor
type Tree a = Fix (TreeF a)

leaf :: Tree a
leaf = Fix Leaf_
node :: Tree a -> a -> Tree a -> Tree a
node l x r = Fix (Node_ l x r)
```

Section 21.2: Primitive recursion

Paramorphisms model primitive recursion. At each iteration of the fold, the folding function receives the subtree for further processing.

```
para :: Functor f => (f (Fix f, a) -> a) -> Fix f -> a
para f = f . fmap (\x -> (x, para f x)) . unFix
```

The Prelude's tails can be modelled as a paramorphism.

```
tails :: List a -> List (List a)
tails = para alg
    其中alg Nil_ = cons nil nil  -- [[]]
        alg (Cons_ x (xs, xss)) = cons (cons x xs) xss  -- (x:xs):xss
```

第21.3节：原始核心递归

自形态（Apomorphisms）模拟原始核心递归。在展开的每次迭代中，展开函数可以返回一个新的种子或整个子树。

```
apo :: Functor f => (a -> f (Either (Fix f) a)) -> a -> Fix f
apo f = Fix . fmap (either id (apo f)) . f
```

注意apo和para是对偶的。类型中的箭头是反转的；para中的元组是apo中的Either的对偶，且它们的实现是彼此的镜像。

第21.4节：一次折叠结构的一层

Catamorphisms，或称为*folds*，模拟原始递归。 cata逐层拆解一个不动点，使用一个*代数*函数（或*折叠函数*）来处理每一层。 cata需要模板类型f的Functor实例。

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata f = f . fmap (cata f) . unFix

-- 列表示例
foldr :: (a -> b -> b) -> b -> List a -> b
foldr f z = cata alg
    其中 alg Nil_ = z
alg (Cons_ x acc) = f x acc
```

第21.5节：一次展开结构的一层

Anamorphisms，或称为*unfolds*，模拟原始核心递归。 ana逐层构建一个不动点，使用一个*余代数*函数（或*展开函数*）来生成每一新层。 ana需要模板类型f的Functor实例。

```
ana :: 函子 f => (a -> f a) -> a -> Fix f
ana f = Fix . fmap (ana f) . f

-- 列表示例
unfoldr :: (b -> Maybe (a, b)) -> b -> List a
unfoldr f = ana coalg
    where coalg x = case f x of
                        Nothing -> Nil_
                        Just (x, y) -> Cons_ x y
```

注意ana和cata是对偶的。它们的类型和实现是彼此的镜像。

第21.6节：展开然后折叠，融合

通常程序结构是先构建一个数据结构，然后将其折叠成单一值。这被称为hylomorphism或refold。为了提高效率，可以完全省略中间结构。

```
hylo :: 函子 f => (a -> f a) -> (f b -> b) -> a -> b
hylo f g = g . fmap (hylo f g) . f  -- 没有提及 Fix！
```

```
tails :: List a -> List (List a)
tails = para alg
    where alg Nil_ = cons nil nil  -- [[]]
          alg (Cons_ x (xs, xss)) = cons (cons x xs) xss  -- (x:xs):xss
```

Section 21.3: Primitive corecursion

Apomorphisms model primitive corecursion. At each iteration of the unfold, the unfolding function may return either a new seed or a whole subtree.

```
apo :: Functor f => (a -> f (Either (Fix f) a)) -> a -> Fix f
apo f = Fix . fmap (either id (apo f)) . f
```

Note that apo and para are *dual*. The arrows in the type are flipped; the tuple in para is dual to the Either in apo, and the implementations are mirror images of each other.

Section 21.4: Folding up a structure one layer at a time

Catamorphisms, or *folds*, model primitive recursion. cata tears down a fixpoint layer by layer, using an *algebra* function (or *folding function*) to process each layer. cata requires a Functor instance for the template type f.

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata f = f . fmap (cata f) . unFix

-- list example
foldr :: (a -> b -> b) -> b -> List a -> b
foldr f z = cata alg
    where alg Nil_ = z
          alg (Cons_ x acc) = f x acc
```

Section 21.5: Unfolding a structure one layer at a time

Anamorphisms, or *unfolds*, model primitive corecursion. ana builds up a fixpoint layer by layer, using a *coalgebra* function (or *unfolding function*) to produce each new layer. ana requires a Functor instance for the template type f.

```
ana :: Functor f => (a -> f a) -> a -> Fix f
ana f = Fix . fmap (ana f) . f

-- list example
unfoldr :: (b -> Maybe (a, b)) -> b -> List a
unfoldr f = ana coalg
    where coalg x = case f x of
                        Nothing -> Nil_
                        Just (x, y) -> Cons_ x y
```

Note that ana and cata are *dual*. The types and implementations are mirror images of one another.

Section 21.6: Unfolding and then folding, fused

It's common to structure a program as building up a data structure and then collapsing it to a single value. This is called a *hylomorphism* or *refold*. It's possible to elide the intermediate structure altogether for improved efficiency.

```
hylo :: Functor f => (a -> f a) -> (f b -> b) -> a -> b
hylo f g = g . fmap (hylo f g) . f  -- no mention of Fix!
```

推导：

```
hylo f g = cata g . ana f
          = g . fmap (cata g) . unFix . Fix . fmap (ana f) . f  -- cata 和 ana 的定义
          = g . fmap (cata g) . fmap (ana f) . f  -- unfix . Fix = id
          = g . fmap (cata g . ana f) . f  -- 函子定律
          = g . fmap (hylo f g) . f  -- hylo 的定义
```

Derivation:

```
hylo f g = cata g . ana f
         = g . fmap (cata g) . unFix . Fix . fmap (ana f) . f  -- definition of cata and ana
         = g . fmap (cata g) . fmap (ana f) . f  -- unfix . Fix = id
         = g . fmap (cata g . ana f) . f  -- Functor law
         = g . fmap (hylo f g) . f  -- definition of hylo
```

第22章：Data.Text

第22.1节：文本字面量

OverloadedStrings 语言扩展允许使用普通字符串字面量来表示Text值。

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text
myText = "overloaded"
```

第22.2节：检查一个Text是否是另一个Text的子串

```
ghci> :set -XOverloadedStrings
ghci> import Data.Text as T
```

isInfixOf :: Text -> Text -> Bool 用于检查一个Text是否包含在另一个Text中。

```
ghci> "rum" `T.isInfixOf` "crumble"
True
```

isPrefixOf :: 文本 -> 文本 -> 布尔值 检查一个文本是否出现在另一个文本的开头。

```
ghci> "crumb" `T.isPrefixOf` "crumble"
True
```

isSuffixOf :: 文本 -> 文本 -> 布尔值 检查一个文本是否出现在另一个文本的结尾。

```
ghci> "rumble" `T.isSuffixOf` "crumble"
True
```

第22.3节：去除空白字符

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text
myText = "\        空白字符    \r"
```

strip 从文本值的开始和结尾移除空白字符。

```
ghci> T.strip myText
"前后空白字符"
```

stripStart 仅从开始处移除空白字符。

```
ghci> T.stripStart myText"前后
空白字符    \r"
```

stripEnd 仅从末尾移除空白字符。

Chapter 22: Data.Text

Section 22.1: Text Literals

The OverloadedStrings language extension allows the use of normal string literals to stand for Text values.

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text
myText = "overloaded"
```

Section 22.2: Checking if a Text is a substring of another Text

```
ghci> :set -XOverloadedStrings
ghci> import Data.Text as T
```

isInfixOf :: Text -> Text -> Bool checks whether a Text is contained anywhere within another Text.

```
ghci> "rum" `T.isInfixOf` "crumble"
True
```

isPrefixOf :: Text -> Text -> Bool checks whether a Text appears at the beginning of another Text.

```
ghci> "crumb" `T.isPrefixOf` "crumble"
True
```

isSuffixOf :: Text -> Text -> Bool checks whether a Text appears at the end of another Text.

```
ghci> "rumble" `T.isSuffixOf` "crumble"
True
```

Section 22.3: Stripping whitespace

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text
myText = "       leading and trailing whitespace       "
```

strip removes whitespace from the start and end of a Text value.

```
ghci> T.strip myText
"leading and trailing whitespace"
```

stripStart removes whitespace only from the start.

```
ghci> T.stripStart myText
"leading and trailing whitespace       "
```

stripEnd removes whitespace only from the end.

```
ghci> T.stripEnd myText"\r
[redacted]白字符"
```

filter 可用于移除中间的空白字符或其他字符。

```
ghci> T.filter /=' ' "文本字符串中间的空格"
"spacesinthemiddleofatextstring"
```

第22.4节：文本索引

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text

myText = "密西西比"
```

可以通过index函数返回特定索引处的字符。

```
ghci> T.index myText 2
's'
```

findIndex函数接受一个类型为(Char -> Bool)的函数和Text，返回给定字符首次出现的索引，若未出现则返回Nothing。

```
ghci> T.findIndex ('s'==) myText
Just 2
ghci> T.findIndex ('c'==) myText
Nothing
```

count函数返回查询文本Text在另一个文本Text中出现的次数。

```
ghci> count ("miss"::T.Text) myText
1
```

第22.5节：拆分文本值

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text
myText = "密西西比"
```

splitOn根据子字符串的出现，将一个文本Text拆分成文本列表Texts。

```
ghci> T.splitOn "ss" myText
["mi","i","ippi"]
```

splitOn是intercalate的逆操作。

```
ghci> intercalate "ss" (splitOn "ss" "mississippi")
"mississippi"
```

```
ghci> T.stripEnd myText
"[redacted]    leading and trailing whitespace"
```

filter can be used to remove whitespace, or other characters, from the middle.

```
ghci> T.filter /=' ' "spaces in the middle of a text string"
"spacesinthemiddleofatextstring"
```

Section 22.4: Indexing Text

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text

myText = "mississippi"
```

Characters at specific indices can be returned by the `index` function.

```
ghci> T.index myText 2
's'
```

The `findIndex` function takes a function of type `(Char -> Bool)` and Text and returns the index of the first occurrence of a given string or Nothing if it doesn't occur.

```
ghci> T.findIndex ('s'==) myText
Just 2
ghci> T.findIndex ('c'==) myText
Nothing
```

The `count` function returns the number of times a query `Text` occurs within another `Text`.

```
ghci> count ("miss"::T.Text) myText
1
```

Section 22.5: Splitting Text Values

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text
myText = "mississippi"
```

`splitOn` breaks a `Text` up into a list of `Texts` on occurrences of a substring.

```
ghci> T.splitOn "ss" myText
["mi","i","ippi"]
```

`splitOn` is the inverse of `intercalate`.

```
ghci> intercalate "ss" (splitOn "ss" "mississippi")
"mississippi"
```

split根据满足布尔谓词的字符，将文本Text值拆分成多个块。

```
ghci> T.split (== 'i') myText
["m","ss","ss","pp",""]
```

第22.6节：文本的编码与解码

各种Unicode编码的编码和解码函数可以在`Data.Text.Encoding`

模块中找到。

```
ghci> import Data.Text.Encoding
ghci> decodeUtf8 (encodeUtf8 "my text")
"my text"
```

请注意，`decodeUtf8`在遇到无效输入时会抛出异常。如果您想自己处理无效的UTF-8，请使用`decodeUtf8With`。

```
ghci> decodeUtf8With (\errorDescription input -> Nothing) messyOutsideData
```

`split` breaks a `Text` value into chunks on characters that satisfy a Boolean predicate.

```
ghci> T.split (== 'i') myText
["m","ss","ss","pp",""]
```

Section 22.6: Encoding and Decoding Text

Encoding and decoding functions for a variety of Unicode encodings can be found in the `Data.Text.Encoding` module.

```
ghci> import Data.Text.Encoding
ghci> decodeUtf8 (encodeUtf8 "my text")
"my text"
```

Note that `decodeUtf8` will throw an exception on invalid input. If you want to handle invalid UTF-8 yourself, use `decodeUtf8With`.

```
ghci> decodeUtf8With (\errorDescription input -> Nothing) messyOutsideData
```

第23章：使用GHCi

第23.1节：GHCi中的断点

GHCi开箱即用地支持命令式风格的断点，适用于解释执行的代码（已通过:加载的代码）。

使用以下程序：

```
-- mySum.hs
doSum n = do
  putStrLn ("Counting to " ++ (show n))
  let v = sum [1..n]
  putStrLn ("sum to " ++ (show n) ++ " = " ++ (show v))
```

加载到 GHCi 中：

```
Prelude> :load mySum.hs
[1 of 1] 正在编译 Main            ( mySum.hs, 解释执行 )
好的，模块已加载：Main。
*Main>
```

我们现在可以使用行号设置断点：

```
*Main> :break 2
断点 0 已在 mySum.hs:2:3-39 处激活
```

当我们运行函数时，GHCi 会在相关行停止：

```
*Main> doSum 12
已在 mySum.hs:2:3-39 停止
_result :: IO () = _
n :: 整数 = 12
[mySum.hs:2:3-39] *Main>
```

程序中我们可能会对当前所在位置感到困惑，因此我们可以使用:list来澄清：

```
[mySum.hs:2:3-39] *Main> :list
1doSum n = do
2    putStrLn ("Counting to " ++ (show n))    -- GHCi 会强调这一行，因为我们就在这里
停止了
3    let v = sum [1..n]
```

我们可以打印变量，并继续执行：

```
[mySum.hs:2:3-39] *Main> n



sum to 12 = 78
*Main>
```

第23.2节：退出GHCi

你可以简单地使用:q或:quit退出GHCi

Chapter 23: Using GHCi

Section 23.1: Breakpoints with GHCi

GHCi supports imperative-style breakpoints out of the box with interpreted code (code that's been `:loaded`).

With the following program:

```
-- mySum.hs
doSum n = do
  putStrLn ("Counting to " ++ (show n))
  let v = sum [1..n]
  putStrLn ("sum to " ++ (show n) ++ " = " ++ (show v))
```

loaded into GHCi:

```
Prelude> :load mySum.hs
[1 of 1] Compiling Main             ( mySum.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

We can now set breakpoints using line numbers:

```
*Main> :break 2
Breakpoint 0 activated at mySum.hs:2:3-39
```

and GHCi will stop at the relevant line when we run the function:

```
*Main> doSum 12
Stopped at mySum.hs:2:3-39
_result :: IO () = _
n :: Integer = 12
[mySum.hs:2:3-39] *Main>
```

It might be confusing where we are in the program, so we can use `:list` to clarify:

```
[mySum.hs:2:3-39] *Main> :list
1  doSum n = do
2    putStrLn ("Counting to " ++ (show n))    -- GHCi will emphasise this line, as that's where we've
stopped
3    let v = sum [1..n]
```

We can print variables, and continue execution too:

```
[mySum.hs:2:3-39] *Main> n
12
:continue
Counting to 12
sum to 12 = 78
*Main>
```

Section 23.2: Quitting GHCi

You can quit GHCi simply with `:q` or `:quit`

```
ghci> :q
正在退出GHCi。

ghci> :quit
正在退出GHCi。
```

另外，快捷键 CTRL+D（Cmd+D（OSX系统）与:q具有相同的效果。

第23.3节：重新加载已加载的文件

如果你已经在GHCi中加载了一个文件（例如使用:l filename.hs），并且你在GHCi外的编辑器中修改了该文件，你必须使用:r或:reload重新加载该文件以应用更改，因此无需再次输入文件名。

```
ghci> :r
好的，模块已加载: Main。

ghci> :reload
好的，模块已加载: Main。
```

第23.4节：启动GHCi

在命令行提示符下输入ghci以启动GHCi。

```
$ ghci
GHCi，版本8.0.1: http://www.haskell.org/ghc/  :? 获取帮助
Prelude>
```

第23.5节：更改GHCi默认提示符

默认情况下，GHCi的提示符会显示你在交互会话中加载的所有模块。如果你加载了许多模块，提示符可能会很长：

```
Prelude Data.List Control.Monad> -- 等等
```

:set prompt 命令用于更改此交互会话的提示符。

```
Prelude Data.List Control.Monad> :set prompt "foo> "
foo>
```

要永久更改提示符，请将 :set prompt "foo> " 添加到 GHCi 配置文件中。

第23.6节：GHCi 配置文件

GHCi 使用位于 `~/.ghci` 的配置文件。配置文件由一系列命令组成，GHCi 启动时会执行这些命令。

```
$ echo ":set prompt 🟩foo> 🟩" > ~/.ghci
$ ghci
GHCi，版本 8.0.1: http://www.haskell.org/ghc/  :? 获取帮助
已从 ~/.ghci 加载 GHCi 配置
foo>
```

```
ghci> :q
Leaving GHCi.

ghci> :quit
Leaving GHCi.
```

Alternatively, the shortcut CTRL+D (Cmd+D for OSX) has the same effect as `:q`.

Section 23.3: Reloading a already loaded file

If you have loaded a file into GHCi (e.g. using `:l filename.hs`) and you have changed the file in an editor outside of GHCi you must reload the file with `:r` or `:reload` in order to make use of the changes, hence you don't need to type again the filename.

```
ghci> :r
OK, modules loaded: Main.

ghci> :reload
OK, modules loaded: Main.
```

Section 23.4: Starting GHCi

Type `ghci` at a shell prompt to start GHCI.

```
$ ghci
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
Prelude>
```

Section 23.5: Changing the GHCi default prompt

By default, GHCI's prompt shows all the modules you have loaded into your interactive session. If you have many modules loaded this can get long:

```
Prelude Data.List Control.Monad> -- etc
```

The `:set prompt` command changes the prompt for this interactive session.

```
Prelude Data.List Control.Monad> :set prompt "foo> "
foo>
```

To change the prompt permanently, add `:set prompt "foo> "` to the GHCi config file.

Section 23.6: The GHCi configuration file

GHCi uses a configuration file in `~/.ghci`. A configuration file consists of a sequence of commands which GHCi will execute on startup.

```
$ echo ":set prompt 🟩foo> 🟩" > ~/.ghci
$ ghci
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from ~/.ghci
foo>
```

第23.7节：加载文件

:l 或 :load 命令会进行类型检查并加载文件。

```
$ echo "f = putStrLn \"示 \"" > exa ple.hs
$ ghci
GHCi，版本 8.0.1: http://www.haskell.org/ghc/  :? 获取帮助
ghci> :l example.hs
[1 of 1] 正在编译 Main             ( example.hs，解释执行 )
完成，模块已加载：Main。
ghci> f
示例
```

第 23.8 节：多行语句

:{ 指令开始 *多行模式*，:} 结束它。在多行模式下，GHCi 会将换行符解释为分号，而不是指令的结束。

```
ghci> :{
ghci| myFoldr f z [] = z
ghci| myFoldr f z (y:ys) = f y (myFoldr f z ys)
ghci| :}
ghci> :t myFoldr
myFoldr :: (a -> b -> b) -> b -> [a] -> b
```

Section 23.7: Loading a file

The `:l` or `:load` command type-checks and loads a file.

```
$ echo "f = putStrLn  example " > example.hs
$ ghci
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
ghci> :l example.hs
[1 of 1] Compiling Main             ( example.hs, interpreted )
Ok, modules loaded: Main.
ghci> f
example
```

Section 23.8: Multi-line statements

The `:{` instruction begins *multi-line mode* and `:}` ends it. In multi-line mode GHCi will interpret newlines as semicolons, not as the end of an instruction.

```
ghci> :{
ghci| myFoldr f z [] = z
ghci| myFoldr f z (y:ys) = f y (myFoldr f z ys)
ghci| :}
ghci> :t myFoldr
myFoldr :: (a -> b -> b) -> b -> [a] -> b
```

第24章：严格性

第24.1节：感叹号模式

带有感叹号（!）注释的模式是严格求值的，而不是惰性求值。

```
foo (!x, y) !z = [x, y, z]
```

在此示例中，x 和 z 都将在返回列表之前被求值到弱头正常形式。这等价于：

```
foo (x, y) z = x `seq` z `seq` [x, y, z]
```

感叹号模式通过 Haskell 2010 的 BangPatterns 语言扩展启用。

第24.2节：惰性模式

惰性模式，或称为 irrefutable 模式（用语法 ~pat 表示），是总是匹配的模式，甚至不查看被匹配的值。这意味着惰性模式即使面对底值也会匹配。然而，对不可拒绝模式子模式中绑定变量的后续使用将强制进行模式匹配，除非匹配成功，否则会求值到底值。

以下函数的参数是惰性的：

```
f1 :: Either e Int -> Int
f1 ~(Right 1) = 42
```

于是我们得到

```
λ» f1 (Right 1)
[illegible]
*** 类型不匹配 ***
```

以下函数使用了惰性模式编写，但实际上使用了该模式的变量，这会强制匹配，因此对于Left参数会失败：

```
f2 :: Either e Int -> Int
f2 ~(Right x) = x + 1

λ» f2 (Right 1)
2
λ» f2 (Right 2)
[illegible]
*** 异常: oops!
λ» f2 (Left "foo")
*** 异常: lazypat.hs:5:1-21: 不可反驳的模式匹配失败，模式为 (Right x)
```

Chapter 24: Strictness

Section 24.1: Bang Patterns

Patterns annotated with a bang (`!`) are evaluated strictly instead of lazily.

```
foo (!x, y) !z = [x, y, z]
```

In this example, `x` and `z` will both be evaluated to weak head normal form before returning the list. It's equivalent to:

```
foo (x, y) z = x `seq` z `seq` [x, y, z]
```

Bang patterns are enabled using the Haskell 2010 `BangPatterns` language extension.

Section 24.2: Lazy patterns

Lazy, or *irrefutable*, patterns (denoted with the syntax `~pat`) are patterns that always match, without even looking at the matched value. This means lazy patterns will match even bottom values. However, subsequent uses of variables bound in sub-patterns of an irrefutable pattern will force the pattern matching to occur, evaluating to bottom unless the match succeeds.

The following function is lazy in its argument:

```
f1 :: Either e Int -> Int
f1 ~(Right 1) = 42
```

and so we get

```
λ» f1 (Right 1)
42
λ» f1 (Right 2)
42
λ» f1 (Left "foo")
42
λ» f1 (error "oops!")
42
λ» f1 "oops!"
*** type mismatch ***
```

The following function is written with a lazy pattern but is in fact using the pattern's variable which forces the match, so will fail for `Left` arguments:

```
f2 :: Either e Int -> Int
f2 ~(Right x) = x + 1

λ» f2 (Right 1)
2
λ» f2 (Right 2)
3
λ» f2 (Right (error "oops!"))
*** Exception: oops!
λ» f2 (Left "foo")
*** Exception: lazypat.hs:5:1-21: Irrefutable pattern failed for pattern (Right x)
```

```
λ» f2 (error "oops!")
*** 异常: oops!
```

let 绑定是惰性的，表现为不可反驳的模式：

```
act1 :: IO ()
act1 = do
ss <- readLn
    let [s1, s2] = ss :: [String]
    putStrLn "Done"

act2 :: IO ()
act2 = do
ss <- readLn
    let [s1, s2] = ss
    putStrLn s1
```

这里 act1 适用于解析为任意字符串列表的输入，而在 act2 中，putStrLn s1 需要 s1 的值，
这就强制要求对 [s1, s2] 进行模式匹配，因此它只适用于恰好包含两个字符串的列表：

```
λ» act1
> ["foo"]
完成
λ» act2
> ["foo"]
*** readIO: 无法解析 ***
```

第24.3节：范式

此示例提供了一个简要概述——有关 *normal forms* 的更深入解释和示例，请参见 this question。

简约范式

简约范式（或在上下文明确时简称范式）是对表达式中所有可约子表达式进行求值后的结果。由于Haskell的非严格语义（通常称为惰性求值），如果子表达式位于绑定器下（即lambda抽象——\x -> ..），则该子表达式不可约。表达式的范式具有唯一性，如果存在的话。

换句话说，在指称语义方面，子表达式的约简顺序并不重要。
然而，编写高性能 Haskell 程序的关键通常是确保在正确的时间计算正确的表达式，即理解操作语义。

一个其正规形式就是其自身的表达式被称为处于正规形式。

有些表达式，例如let x = 1:x in x，没有规范形式，但仍然是有生产力的。该示例表达式仍然有一个*值*，如果允许无限值，这里是列表[1,1, ...]。其他表达式，例如let y = 1+y in y，没有值，或者它们的值是未定义的。

弱头归一形态

RNF对应于对表达式的完全求值——同样，弱头正规形（WHNF）对应于对表达式的头部进行求值。表达式的头部 e 被完全求值当且仅当 e 是一个应用 Cone1 e2 .. en，且 Con 是一个构造器；或者是一个抽象 \x -> e1；或者是一个部分应用 f e1 e2 .. en，其中部分应用意味着 f 接受的参数超过 n 个（或等价地， e 的类型是函数类型）。在任何情况下，子表达式 e1..en 可以被求值也可以不被求值，表达式仍处于WHNF——它们甚至可以

```
λ» f2 (error "oops!")
*** Exception: oops!
```

`let` bindings are lazy, behave as irrefutable patterns:

```
act1 :: IO ()
act1 = do
    ss <- readLn
    let [s1, s2] = ss :: [String]
    putStrLn "Done"

act2 :: IO ()
act2 = do
    ss <- readLn
    let [s1, s2] = ss
    putStrLn s1
```

Here `act1` works on inputs that parse to any list of strings, whereas in `act2` the `putStrLn s1` needs the value of `s1` which forces the pattern matching for `[s1, s2]`, so it works only for lists of exactly two strings:

```
λ» act1
> ["foo"]
Done
λ» act2
> ["foo"]
*** readIO: no parse ***
```

Section 24.3: Normal forms

This example provides a brief overview - for a more in-depth explanation of *normal forms* and examples, see this question.

Reduced normal form

The reduced normal form (or just normal form, when the context is clear) of an expression is the result of evaluating all reducible subexpressions in the given expression. Due to the non-strict semantics of Haskell (typically called *laziness*), a subexpression is not reducible if it is under a binder (i.e. a lambda abstraction - `\x -> ..`). The normal form of an expression has the property that if it exists, it is unique.

In other words, it does not matter (in terms of denotational semantics) in which order you reduce subexpressions. However, the key to writing performant Haskell programs is often ensuring that the right expression is evaluated at the right time, i.e, the understanding the operational semantics.

An expression whose normal form is itself is said to be *in normal form*.

Some expressions, e.g. `let x = 1:x in x`, have no normal form, but are still productive. The example expression still has a *value*, if one admits infinite values, which here is the list `[1,1, ...]`. Other expressions, such as `let y = 1+y in y`, have no value, or their value is `undefined`.

Weak head normal form

The RNF corresponds to fully evaluating an expression - likewise, the weak head normal form (WHNF) corresponds to evaluating to the *head* of the expression. The head of an expression `e` is fully evaluated if `e` is an application `Con e1 e2 .. en` and `Con` is a constructor; or an abstraction `\x -> e1`; or a partial application `f e1 e2 .. en`, where partial application means `f` takes more than `n` arguments (or equivalently, the type of `e` is a function type). In any case, the subexpressions `e1..en` can be evaluated or unevaluated for the expression to be in WHNF - they can even

未定义。

Haskell 的求值语义可以用 WHNF（弱头正范式）来描述——要对表达式 e 求值，首先将其求值到 WHNF，然后从左到右递归地求值其所有子表达式。

原始的seq函数用于将表达式求值到弱头归约形式（WHNF）。seq x y在指称上等同于y（seq x y的值正是y）；此外，当y被求值到WHNF时，x也会被求值到WHNF。表达式也可以通过叹号模式（由-XBangPatterns扩展启用）求值到WHNF，其语法如下：

```
f !x y = ...
```

在其中，x 会在 f 被求值时被求值为弱头规范形式（WHNF），而 y 不一定会被求值。一个强制模式（bang pattern）也可以出现在构造函数中，例如：

```
data X = Con A !B C .. N
```

在这种情况下，构造函数 Con 被称为在 B 字段上是严格的，这意味着当构造函数被应用于足够（这里是两个）参数时，B 字段会被求值为弱头规范形式（WHNF）。

第24.4节：严格字段

在一个 data 声明中，给类型前加上感叹号（!）会使该字段成为一个 严格字段。当数据构造函数被应用时，这些字段会被求值为弱头规范形式，因此字段中的数据保证始终处于弱头规范形式。

严格字段可以用于记录类型和非记录类型：

```
data User = User
    { identifier :: !Int
    , firstName :: !Text
    , lastName :: !Text
    }

data T = MkT !Int !Int
```

be **undefined**.

The evaluation semantics of Haskell can be described in terms of the WHNF - to evaluate an expression e, first evaluate it to WHNF, then recursively evaluate all of its subexpressions from left to right.

The primitive **seq** function is used to evaluate an expression to WHNF. **seq** x y is denotationally equal to y (the value of **seq** x y is precisely y); furthermore x is evaluated to WHNF when y is evaluated to WHNF. An expression can also be evaluated to WHNF with a bang pattern (enabled by the -XBangPatterns extension), whose syntax is as follows:

```
f !x y = ...
```

In which x will be evaluated to WHNF when f is evaluated, while y is not (necessarily) evaluated. A bang pattern can also appear in a constructor, e.g.

```
data X = Con A !B C .. N
```

in which case the constructor Con is said to be strict in the B field, which means the B field is evaluated to WHNF when the constructor is applied to sufficient (here, two) arguments.

Section 24.4: Strict fields

In a **data** declaration, prefixing a type with a bang (!) makes the field a *strict field*. When the data constructor is applied, those fields will be evaluated to weak head normal form, so the data in the fields is guaranteed to always be in weak head normal form.

Strict fields can be used in both record and non-record types:

```
data User = User
    { identifier :: !Int
    , firstName :: !Text
    , lastName :: !Text
    }

data T = MkT !Int !Int
```

第25章：函数中的语法

第25.1节：模式匹配

Haskell 支持在函数定义中以及通过case语句进行模式匹配表达式。

case 语句类似于其他语言中的 switch，但它支持 Haskell 的所有类型。

让我们从简单的开始：

```
longName :: String -> String
longName name = case name of
                   "Alex"   -> "Alexander"
                   "Jenny" -> "Jennifer"
_       -> "Unknown"  -- 如果你愿意，可以把它看作“默认”情况
```

或者，我们可以像写方程一样定义函数，这也是模式匹配，只是不使用case
语句：

```
longName "Alex"  = "Alexander"
longName "Jenny" = "Jennifer"
longName _       = "Unknown"
```

一个更常见的例子是使用Maybe类型：

```
data Person = Person { name :: String, petName :: (Maybe String) }

hasPet :: Person -> Bool
hasPet (Person _ Nothing) = False
hasPet _ = True  -- Maybe 只能取 `Just a` 或 `Nothing`，所以这个通配符足够了
```

模式匹配也可以用于列表：

```
isEmptyList :: [a] -> Bool
isEmptyList [] = True
isEmptyList _  = False

addFirstTwoItems :: [Int] -> [Int]
addFirstTwoItems []        = []
addFirstTwoItems (x:[])    = [x]
addFirstTwoItems (x:y:ys)  = (x + y) : ys
```

实际上，模式匹配可以用于任何类型类的任何构造函数。例如，列表的构造函数是 :，元组的构造函数是 ,

第25.2节：使用 where 和 guards

给定以下函数：

```
annualSalaryCalc :: (RealFloat a) => a -> a -> String
annualSalaryCalc hourlyRate weekHoursOfWork
   | hourlyRate * (weekHoursOfWork * 52) <= 40000  = "可怜的孩子，试着找份别的工作"
   | hourlyRate * (weekHoursOfWork * 52) <= 120000 = "钱，钱，钱！"
   | hourlyRate * (weekHoursOfWork * 52) <= 200000 = "Ri¢hie Ri¢h"
   | otherwise = "你好，埃隆·马斯克！"
```

Chapter 25: Syntax in Functions

Section 25.1: Pattern Matching

Haskell supports pattern matching expressions in both function definition and through case statements.

A case statement is much like a switch in other languages, except it supports all of Haskell's types.

Let's start simple:

```
longName :: String -> String
longName name = case name of
                   "Alex"   -> "Alexander"
                   "Jenny"  -> "Jennifer"
                   _        -> "Unknown"   -- the "default" case, if you like
```

Or, we could define our function like an equation which would be pattern matching, just without using a case statement:

```
longName "Alex"  = "Alexander"
longName "Jenny" = "Jennifer"
longName _       = "Unknown"
```

A more common example is with the Maybe type:

```
data Person = Person { name :: String, petName :: (Maybe String) }

hasPet :: Person -> Bool
hasPet (Person _ Nothing) = False
hasPet _ = True  -- Maybe can only take `Just a` or `Nothing`, so this wildcard suffices
```

Pattern matching can also be used on lists:

```
isEmptyList :: [a] -> Bool
isEmptyList [] = True
isEmptyList _  = False

addFirstTwoItems :: [Int] -> [Int]
addFirstTwoItems []        = []
addFirstTwoItems (x:[])    = [x]
addFirstTwoItems (x:y:ys)  = (x + y) : ys
```

Actually, Pattern Matching can be used on any constructor for any type class. E.g. the constructor for lists is : and for tuples ,

Section 25.2: Using where and guards

Given this function:

```
annualSalaryCalc :: (RealFloat a) => a -> a -> String
annualSalaryCalc hourlyRate weekHoursOfWork
  | hourlyRate * (weekHoursOfWork * 52) <= 40000  = "Poor child, try to get another job"
  | hourlyRate * (weekHoursOfWork * 52) <= 120000 = "Money, Money, Money!"
  | hourlyRate * (weekHoursOfWork * 52) <= 200000 = "Ri¢hie Ri¢h"
  | otherwise = "Hello Elon Musk!"
```

我们可以使用 where 来避免重复，使代码更易读。请看下面使用 where 的替代函数：

```
annualSalaryCalc' :: (RealFloat a) => a -> a -> String
annualSalaryCalc' hourlyRate weekHoursOfWork
  | annualSalary <= smallSalary  = "可怜的孩子，试着找份别的工作"
  | annualSalary <= mediumSalary = "钱，钱，钱！"
  | annualSalary <= highSalary   = "Ri¢hie Ri¢h"
  | otherwise = "你好，埃隆·马斯克！"
  其中
annualSalary = hourlyRate * (weekHoursOfWork * 52)
      (smallSalary, mediumSalary, highSalary)  = (40000, 120000, 200000)
```

如所见，我们在函数体末尾使用了 `where`，消除了计算 (`hourlyRate * (weekHoursOfWork * 52)`) 的重复，同时也用 `where` 来组织薪资范围。

公共子表达式的命名也可以通过 `let` 表达式实现，但只有 `where` 语法才允许 *guards* 引用这些命名的子表达式。

第25.3节：保护装置

函数可以使用守卫来定义，守卫可以被视为根据输入对行为进行分类。

以下是函数定义示例：

```
absolute :: Int -> Int  -- 为简化起见，定义限制为 Int 类型
absolute n = if (n < 0) then (-n) else n
```

我们可以使用守卫重新排列它：

```
absolute :: Int -> Int
absolute n
  | n < 0 = -n
  | otherwise = n
```

在此上下文中，otherwise 是 True 的有意义别名，因此它应始终是最后一个守卫。

We can use **where** to avoid the repetition and make our code more readable. See the alternative function below, using **where**:

```
annualSalaryCalc' :: (RealFloat a) => a -> a -> String
annualSalaryCalc' hourlyRate weekHoursOfWork
  | annualSalary <= smallSalary  = "Poor child, try to get another job"
  | annualSalary <= mediumSalary = "Money, Money, Money!"
  | annualSalary <= highSalary   = "Ri¢hie Ri¢h"
  | otherwise = "Hello Elon Musk!"
  where
      annualSalary = hourlyRate * (weekHoursOfWork * 52)
      (smallSalary, mediumSalary, highSalary)  = (40000, 120000, 200000)
```

As observed, we used the **where** in the end of the function body eliminating the repetition of the calculation (`hourlyRate * (weekHoursOfWork * 52)`) and we also used **where** to organize the salary range.

The naming of common sub-expressions can also be achieved with **let** expressions, but only the **where** syntax makes it possible for *guards* to refer to those named sub-expressions.

Section 25.3: Guards

A function can be defined using guards, which can be thought of classifying behaviour according to input.

Take the following function definition:

```
absolute :: Int -> Int  -- definition restricted to Ints for simplicity
absolute n = if (n < 0) then (-n) else n
```

We can rearrange it using guards:

```
absolute :: Int -> Int
absolute n
  | n < 0 = -n
  | otherwise = n
```

In this context **otherwise** is a meaningful alias for `True`, so it should always be the last guard.

第26章：函子

函子（Functor）是类型类 f :: * -> * 的集合，这些类型可以进行协变映射。对数据结构映射一个函数意味着将该函数应用于结构中的所有元素，而不改变结构本身。

第26.1节：函子的类定义及定律

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

一种看法是，fmap将一个值的函数提升为一个带有上下文f的值的函数。

一个正确的Functor实例应满足functor定律，尽管编译器不会强制执行这些定律：

```
fmap id = id                    -- 恒等律
fmap f . fmap g = fmap (f . g)  -- 结合律
```

有一个常用的中缀别名叫做<$>，代表fmap。

```
infixl 4 <$>
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

第26.2节：用单一值替换Functor中的所有元素

Data.Functor模块包含两个组合子，<$和$>，它们忽略Functor中包含的所有值，全部替换为单一的常量值。

```
infixl 4 <$, $>

<$ :: Functor f => a -> f b -> f a
(<$) = fmap . const

$> :: 函子 f => f a -> b -> f b
($>) = flip (<$)
```

void 忽略计算的返回值。

```
void :: Functor f => f a -> f ()
void = (() <$)
```

第26.3节：Functor的常见实例

Maybe

Maybe 是一个包含可能缺失值的Functor：

```
instance Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

`Maybe`的`Functor`实例将函数应用于包装在`Just`中的值。如果计算之前失败了（因此`Maybe`值是`Nothing`），则没有值可供函数应用，所以**`fmap`**是无操作。

Chapter 26: Functor

`Functor` is the class of types `f :: * -> *` which can be covariantly *mapped* over. Mapping a function over a data structure applies the function to all the elements of the structure without changing the structure itself.

Section 26.1: Class Definition of Functor and Laws

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

One way of looking at it is that **`fmap`** *lifts* a function of values into a function of values in a context `f`.

A correct instance of `Functor` should satisfy the *functor laws*, though these are not enforced by the compiler:

```
fmap id = id                    -- identity
fmap f . fmap g = fmap (f . g)  -- composition
```

There's a commonly-used infix alias for **`fmap`** called `<$>`.

```
infixl 4 <$>
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

Section 26.2: Replacing all elements of a Functor with a single value

The `Data.Functor` module contains two combinators, `<$` and `$>`, which ignore all of the values contained in a functor, replacing them all with a single constant value.

```
infixl 4 <$, $>

<$ :: Functor f => a -> f b -> f a
(<$) = fmap . const

$> :: Functor f => f a -> b -> f b
($>) = flip (<$)
```

`void` ignores the return value of a computation.

```
void :: Functor f => f a -> f ()
void = (() <$)
```

Section 26.3: Common instances of Functor

Maybe

`Maybe` is a `Functor` containing a possibly-absent value:

```
instance Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

`Maybe`'s instance of `Functor` applies a function to a value wrapped in a `Just`. If the computation has previously failed (so the `Maybe` value is a `Nothing`), then there's no value to apply the function to, so **`fmap`** is a no-op.

```
> fmap (+ 3) (Just 3)
Just 6
> fmap length (Just "mousetrap")
Just 9
> fmap sqrt Nothing
Nothing
```

我们可以使用等式推理来验证该实例的函子定律。对于恒等律，

```
fmap id Nothing
Nothing  -- fmap 的定义
id Nothing  -- id 的定义

fmap id (Just x)
Just (id x)  -- fmap 的定义
Just x  -- id 的定义
id (Just x)  -- id 的定义
```

对于复合律，

```
(fmap f . fmap g) Nothing
fmap f (fmap g Nothing)  -- (.) 的定义
fmap f Nothing  -- fmap 的定义
Nothing  -- fmap 的定义
fmap (f . g) Nothing  -- 因为对于所有 f，Nothing = fmap f Nothing

(fmap f . fmap g) (Just x)
fmap f (fmap g (Just x))  -- (.) 的定义
fmap f (Just (g x))  -- fmap 的定义
Just (f (g x))  -- fmap 的定义
Just ((f . g) x)  -- (.) 的定义
fmap (f . g) (Just x)  -- fmap 的定义
```

列表

列表的Functor实例将函数应用于列表中的每个值。

```
实例 Functor [] 其中
    fmap f [] = []
    fmap f (x:xs) = f x : fmap f xs
```

这也可以写成列表推导式： fmap f xs = [f x | x <- xs]。

这个例子表明 `fmap` 泛化了 `map`。 `map` 只作用于列表，而 `fmap` 作用于任意的 `Functor`。

可以通过归纳法证明恒等律成立：

```
-- 基础情况
fmap id []
[]  -- fmap 的定义
id []  -- id 的定义

-- 归纳步骤
fmap id (x:xs)
id x : fmap id xs  -- fmap 的定义
x : fmap id xs  -- id 的定义
x : id xs  -- 归纳假设
x : xs  -- id 的定义
```

```
> fmap (+ 3) (Just 3)
Just 6
> fmap length (Just "mousetrap")
Just 9
> fmap sqrt Nothing
Nothing
```

We can check the functor laws for this instance using equational reasoning. For the identity law,

```
fmap id Nothing
Nothing  -- definition of fmap
id Nothing  -- definition of id

fmap id (Just x)
Just (id x)  -- definition of fmap
Just x  -- definition of id
id (Just x)  -- definition of id
```

For the composition law,

```
(fmap f . fmap g) Nothing
fmap f (fmap g Nothing)  -- definition of (.)
fmap f Nothing  -- definition of fmap
Nothing  -- definition of fmap
fmap (f . g) Nothing  -- because Nothing = fmap f Nothing, for all f

(fmap f . fmap g) (Just x)
fmap f (fmap g (Just x))  -- definition of (.)
fmap f (Just (g x))  -- definition of fmap
Just (f (g x))  -- definition of fmap
Just ((f . g) x)  -- definition of (.)
fmap (f . g) (Just x)  -- definition of fmap
```

Lists

Lists' instance of `Functor` applies the function to every value in the list in place.

```
instance Functor [] where
    fmap f [] = []
    fmap f (x:xs) = f x : fmap f xs
```

This could alternatively be written as a list comprehension: `fmap f xs = [f x | x <- xs]`.

This example shows that `fmap` generalises `map`. `map` only operates on lists, whereas `fmap` works on an arbitrary `Functor`.

The identity law can be shown to hold by induction:

```
-- base case
fmap id []
[]  -- definition of fmap
id []  -- definition of id

-- inductive step
fmap id (x:xs)
id x : fmap id xs  -- definition of fmap
x : fmap id xs  -- definition of id
x : id xs  -- by the inductive hypothesis
x : xs  -- definition of id
```

```
id (x : xs) -- id 的定义
```

类似地，组合律：

```
-- 基础情况
(fmap f . fmap g) []
fmap f (fmap g [])  -- (.) 的定义
fmap f []  -- fmap 的定义
[]  -- fmap 的定义
fmap (f . g) []  -- 因为 [] = fmap f [], 对所有 f 成立

-- 归纳步骤
(fmap f . fmap g) (x:xs)
fmap f (fmap g (x:xs))  -- (.) 的定义
fmap f (g x : fmap g xs)  -- fmap 的定义
f (g x) : fmap f (fmap g xs)  -- fmap 的定义
(f . g) x : fmap f (fmap g xs)  -- (.) 的定义
(f . g) x : fmap (f . g) xs  -- 归纳假设
fmap (f . g) xs  -- fmap 的定义
```

函数

并非每个函子（Functor）都像容器。函数作为函子（Functor）的实例，是将一个函数应用到另一个函数的返回值上
。

```
实例 函子 ((->) r) 其中
    fmap f g = \x -> f (g x)
```

注意这个定义等价于 fmap = (.)。因此 fmap 泛化了函数组合。

再次验证恒等律：

```
fmap id g
\x -> id (g x)  -- fmap 的定义
\x -> g x  -- id 的定义
g  -- eta-约简
id g  -- id 的定义
```

以及组合律：

```
(fmap f . fmap g) h
fmap f (fmap g h)  -- (.) 的定义
fmap f (\x -> g (h x))  -- fmap 的定义
\y -> f ((\x -> g (h x)) y)  -- fmap 的定义
\y -> f (g (h y))  -- beta-约简
\y -> (f . g) (h y)  -- (.) 的定义
fmap (f . g) h  -- fmap 的定义
```

第26.4节：派生Functor

DeriveFunctor 语言扩展允许GHC自动生成 Functor 的实例。

```
{-# LANGUAGE DeriveFunctor #-}

数据类型 List a = Nil | Cons a (List a) 派生 Functor

-- Functor List的实例自动定义
--    fmap f Nil = Nil
--    fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

```
id (x : xs)  -- definition of id
```

and similarly, the composition law:

```
-- base case
(fmap f . fmap g) []
fmap f (fmap g [])  -- definition of (.)
fmap f []  -- definition of fmap
[]  -- definition of fmap
fmap (f . g) []  -- because [] = fmap f [], for all f

-- inductive step
(fmap f . fmap g) (x:xs)
fmap f (fmap g (x:xs))  -- definition of (.)
fmap f (g x : fmap g xs)  -- definition of fmap
f (g x) : fmap f (fmap g xs)  -- definition of fmap
(f . g) x : fmap f (fmap g xs)  -- definition of (.)
(f . g) x : fmap (f . g) xs  -- by the inductive hypothesis
fmap (f . g) xs  -- definition of fmap
```

Functions

Not every Functor looks like a container. Functions' instance of Functor applies a function to the return value of another function.

```
instance Functor ((->) r) where
    fmap f g = \x -> f (g x)
```

Note that this definition is equivalent to fmap = (.). So fmap generalises function composition.

Once more checking the identity law:

```
fmap id g
\x -> id (g x)  -- definition of fmap
\x -> g x  -- definition of id
g  -- eta-reduction
id g  -- definition of id
```

and the composition law:

```
(fmap f . fmap g) h
fmap f (fmap g h)  -- definition of (.)
fmap f (\x -> g (h x))  -- definition of fmap
\y -> f ((\x -> g (h x)) y)  -- definition of fmap
\y -> f (g (h y))  -- beta-reduction
\y -> (f . g) (h y)  -- definition of (.)
fmap (f . g) h  -- definition of fmap
```

Section 26.4: Deriving Functor

The DeriveFunctor language extension allows GHC to generate instances of Functor automatically.

```
{-# LANGUAGE DeriveFunctor #-}

data List a = Nil | Cons a (List a) deriving Functor

-- instance Functor List where            -- automatically defined
--    fmap f Nil = Nil
--    fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

```
map :: (a -> b) -> List a -> List b
map = fmap
```

第26.5节：多项式函子

有一组有用的类型组合器，可以用较小的 Functor 构建出较大的 Functor。这些作为 Functor 的示例实例具有启发性，同时它们也是泛型编程中的一种技术，因为它们可以用来表示一大类常见的函子。

恒等函子

恒等函子简单地封装其参数。它是SKI演算中 I 组合子的类型级实现。

```
newtype I a = I a

instance Functor I where
    fmap f (I x) = I (f x)
```

我可以在数据.函子.身份模块中以身份的名称找到。

常量函子

常量函子忽略其第二个参数，仅包含一个常量值。它是类型级别的
const，SKI演算中的K组合子。

```
newtype K c a = K c
```

注意K c a不包含任何 a值；K ()与Proxy同构。这意味着K对
fmap的实现根本不进行任何映射！

```
instance Functor (K c) where
    fmap _ (K c) = K c
```

K也被称为Const，来自Data.Functor.Const。

本例中剩余的函子将较小的函子组合成更大的函子。

函子积

函子积接受一对函子并将它们打包。它类似于元组，尽管(,) ::
* -> * -> * 作用于 类型 *，(:*:) :: (* -> *) -> (* -> *) -> (* -> *) 作用于 函子 * -> *。

```
infixl 7 :*:
数据 (f :*: g) a = f a :*: g a

实例 (函子 f, 函子 g) => 函子 (f :*: g) 其中
    fmap f (fx :*: gy) = fmap f fx :*: fmap f gy
```

该类型可以在 Data.Functor.Product 模块中以 Product 名称找到。

函子余积

正如 :*: 类似于 (,)，:+: 是函子级别的 Either 类比。

```
infixl 6 :+:
```

```
map :: (a -> b) -> List a -> List b
map = fmap
```

Section 26.5: Polynomial functors

There's a useful set of type combinators for building big **Functor**s out of smaller ones. These are instructive as example instances of **Functor**, and they're also useful as a technique for generic programming, because they can be used to represent a large class of common functors.

The identity functor

The identity functor simply wraps up its argument. It's a type-level implementation of the `I` combinator from SKI calculus.

```
newtype I a = I a

instance Functor I where
    fmap f (I x) = I (f x)
```

I can be found, under the name of `Identity`, in the `Data.Functor.Identity` module.

The constant functor

The constant functor ignores its second argument, containing only a constant value. It's a type-level analogue of **const**, the K combinator from SKI calculus.

```
newtype K c a = K c
```

Note that `K c a` doesn't contain any `a`-values; `K ()` is isomorphic to `Proxy`. This means that `K`'s implementation of **fmap** doesn't do any mapping at all!

```
instance Functor (K c) where
    fmap _ (K c) = K c
```

`K` is otherwise known as `Const`, from `Data.Functor.Const`.

The remaining functors in this example combine smaller functors into bigger ones.

Functor products

The functor product takes a pair of functors and packs them up. It's analogous to a tuple, except that while `(,) :: * -> * -> *` operates on `types *`, `(:*:) :: (* -> *) -> (* -> *) -> (* -> *)` operates on `functors * -> *`.

```
infixl 7 :*:
data (f :*: g) a = f a :*: g a

instance (Functor f, Functor g) => Functor (f :*: g) where
    fmap f (fx :*: gy) = fmap f fx :*: fmap f gy
```

This type can be found, under the name `Product`, in the `Data.Functor.Product` module.

Functor coproducts

Just like `:*:` is analogous to `(,)`, `:+:` is the functor-level analogue of **Either**.

```
infixl 6 :+:
```

```
数据 (f :+: g) a = InL (f a) | InR (g a)

实例 (函子 f, 函子 g) => 函子 (f :+: g) 其中
    fmap f (InL fx) = InL (fmap f fx)
    fmap f (InR gy) = InR (fmap f gy)
```

:+: 可以在 Data.Functor.Sum 模块中以 Sum 名称找到。

函子组合

最后，:.: 就像一个类型级的 (.)，将一个函子（functor）的输出连接到另一个函子的输入。

```
infixr 9 :.:
newtype (f :.: g) a = Cmp (f (g a))

instance (Functor f, Functor g) => Functor (f :.: g) where
    fmap f (Cmp fgx) = Cmp (fmap (fmap f) fgx)
```

Compose 类型可以在 Data.Functor.Compose 中找到

用于泛型编程的多项式函子

I、K、:*:、:+: 和 :.: 可以被看作是一套用于某类简单数据类型的构建模块。这套工具包在与固定点（fixed points）结合时尤其强大，因为用这些组合子构建的数据类型会自动成为 Functor 的实例。你使用这套工具包来构建一个模板类型，使用 I 标记递归点，然后将其插入 Fix，得到一个可以与标准递归方案库一起使用的类型。

名称	作为数据类型	使用函子工具包
值对	`data Pair a = Pair a a`	`type Pair = I :*: I`
Two-by-two grids	`type Grid a = Pair (Pair a)`	`type Grid = Pair :.: Pair`
Natural numbers	`data Nat = Zero | Succ Nat`	`type Nat = Fix (K () :+: I)`
列表	`data List a = Nil | Cons a (List a)`	`type List a = Fix (K () :+: K a :*: I)`
二叉树	`data Tree a = Leaf | Node (Tree a) a (Tree a)`	`type Tree a = Fix (K () :+: I :*: K a :*: I)`
玫瑰树	`data 玫瑰 a = 玫瑰 a (列表 (玫瑰 a))`	`类型 玫瑰 a = 固定 (K a :*: 列表 :.: I)`

这种用于设计数据类型的"工具包"方法是泛型编程库（如generics-sop）的核心思想。其思路是使用如上所示的工具包编写泛型操作，然后使用类型类将任意数据类型转换为其泛型表示形式，反之亦然：

```
类 泛型 a 其中
    类型 表示 a -- 使用工具包构建的泛型表示
    到 :: a -> 表示 a
从 :: 表示 a -> a
```

第26.6节：范畴论中的函子

在范畴论中，函子被定义为范畴之间保持结构的映射（即"同态"）。具体来说，（所有）对象映射到对象，（所有）箭头映射到箭头，从而保持范畴定律。

对象是 Haskell 类型且态射是 Haskell 函数的范畴称为Hask。因此，从Hask到Hask的函子将由类型到类型的映射和函数到函数的映射组成。

这一范畴理论概念与 Haskell 编程构造Functor的关系相当

```
data (f :+: g) a = InL (f a) | InR (g a)

instance (Functor f, Functor g) => Functor (f :+: g) where
    fmap f (InL fx) = InL (fmap f fx)
    fmap f (InR gy) = InR (fmap f gy)
```

:+: can be found under the name Sum, in the Data.Functor.Sum module.

Functor composition

Finally, :.: works like a type-level (.), taking the output of one functor and plumbing it into the input of another.

```
infixr 9 :.:
newtype (f :.: g) a = Cmp (f (g a))

instance (Functor f, Functor g) => Functor (f :.: g) where
    fmap f (Cmp fgx) = Cmp (fmap (fmap f) fgx)
```

The Compose type can be found in Data.Functor.Compose

Polynomial functors for generic programming

I, K, :*:, :+: and :.: can be thought of as a kit of building blocks for a certain class of simple datatypes. The kit becomes especially powerful when you combine it with fixed points because datatypes built with these combinators are automatically instances of Functor. You use the kit to build a template type, marking recursive points using I, and then plug it into Fix to get a type that can be used with the standard zoo of recursion schemes.

Name	As a datatype	Using the functor kit
Pairs of values	`data Pair a = Pair a a`	`type Pair = I :*: I`
Two-by-two grids	`type Grid a = Pair (Pair a)`	`type Grid = Pair :.: Pair`
Natural numbers	`data Nat = Zero | Succ Nat`	`type Nat = Fix (K () :+: I)`
Lists	`data List a = Nil | Cons a (List a)`	`type List a = Fix (K () :+: K a :*: I)`
Binary trees	`data Tree a = Leaf | Node (Tree a) a (Tree a)`	`type Tree a = Fix (K () :+: I :*: K a :*: I)`
Rose trees	`data Rose a = Rose a (List (Rose a))`	`type Rose a = Fix (K a :*: List :.: I)`

This "kit" approach to designing datatypes is the idea behind *generic programming* libraries such as generics-sop. The idea is to write generic operations using a kit like the one presented above, and then use a type class to convert arbitrary datatypes to and from their generic representation:

```
class Generic a where
    type Rep a  -- a generic representation built using a kit
    to :: a -> Rep a
    from :: Rep a -> a
```

Section 26.6: Functors in Category Theory

A Functor is defined in category theory as a structure-preserving map (a 'homomorphism') between categories. Specifically, (all) objects are mapped to objects, and (all) arrows are mapped to arrows, such that the category laws are preserved.

The category in which objects are Haskell types and morphisms are Haskell functions is called **Hask**. So a functor from **Hask** to **Hask** would consist of a mapping of types to types and a mapping from functions to functions.

The relationship that this category theoretic concept bears to the Haskell programming construct Functor is rather

直接。类型到类型的映射采用类型f:: * -> *的形式，函数到函数的映射采用函数fmap:: (a -> b) -> (f a -> f b)的形式。将它们组合在一个类中，

```
class Functor (f :: * -> *) where
    fmap :: (a -> b) -> f a -> f b
```

fmap是一个操作，它接受一个函数（态射类型），:: a -> b，并将其映射为另一个函数，:: f a -> f b。假设（但由程序员保证）Functor的实例确实是数学上的函子，保持Hask的范畴结构：

```
fmap (id {- :: a -> a -})  ==  id {- :: f a -> f a -}
fmap (h . g)               ==  fmap h . fmap g
```

fmap将函数:: a -> b提升到Hask的一个子范畴中，保持任何恒等箭头的存在以及复合的结合性。

Functor类仅编码了Hask上的endo函子。但在数学中，函子可以映射任意范畴之间。对此概念更忠实的编码如下所示：

```
class Category c where
    id  :: c i i
    (.) :: c j k -> c i j -> c i k

class (Category c1, Category c2) => CFunctor c1 c2 f where
    cfmap :: c1 a b -> c2 (f a) (f b)
```

标准的 Functor 类是该类的一个特例，其中源类别和目标类别均为 Hask。
例如，

```
instance Category (->) where        -- Hask
    id     = \x -> x
f . g = \x -> f (g x)

instance CFunctor (->) (->) [] where
    cfmap = fmap
```

direct. The mapping from types to types takes the form of a type `f :: * -> *`, and the mapping from functions to functions takes the form of a function `fmap :: (a -> b) -> (f a -> f b)`. Putting those together in a class,

```
class Functor (f :: * -> *) where
    fmap :: (a -> b) -> f a -> f b
```

`fmap` is an operation that takes a function (a type of morphism), `:: a -> b`, and maps it to another function, `:: f a -> f b`. It is assumed (but left to the programmer to ensure) that instances of `Functor` are indeed mathematical functors, preserving **Hask**'s categorical structure:

```
fmap (id {- :: a -> a -})  ==  id {- :: f a -> f a -}
fmap (h . g)               ==  fmap h . fmap g
```

`fmap` lifts a function `:: a -> b` into a subcategory of **Hask** in a way that preserves both the existence of any identity arrows, and the associativity of composition.

The `Functor` class only encodes *endo*functors on **Hask**. But in mathematics, functors can map between arbitrary categories. A more faithful encoding of this concept would look like this:

```
class Category c where
    id  :: c i i
    (.) :: c j k -> c i j -> c i k

class (Category c1, Category c2) => CFunctor c1 c2 f where
    cfmap :: c1 a b -> c2 (f a) (f b)
```

The standard Functor class is a special case of this class in which the source and target categories are both **Hask**.
For example,

```
instance Category (->) where        -- Hask
    id     = \x -> x
    f . g = \x -> f (g x)

instance CFunctor (->) (->) [] where
    cfmap = fmap
```

第27章：使用 Tasty 进行测试

第27.1节：SmallCheck、QuickCheck 和 HUnit

```
import Test.Tasty
import Test.Tasty.SmallCheck as SC
import Test.Tasty.QuickCheck as QC
import Test.Tasty.HUnit


main :: IO ()
main = defaultMain tests

tests :: TestTree
tests = testGroup "Tests" [smallCheckTests, quickCheckTests, unitTests]

smallCheckTests :: TestTree
smallCheckTests = testGroup "SmallCheck Tests"
  [ SC.testProperty "字符串长度 <= 3" $
      \s -> length (take 3 (s :: String)) <= 3
  , SC.testProperty "字符串长度 <= 2" $  -- 应该失败
      \s -> length (take 3 (s :: String)) <= 2
  ]

quickCheckTests :: TestTree
quickCheckTests = testGroup "QuickCheck Tests"
  [ QC.testProperty "字符串长度 <= 5" $
      \s -> length (take 5 (s :: String)) <= 5
  , QC.testProperty "字符串长度 <= 4" $  -- 应该失败
      \s -> length (take 5 (s :: String)) <= 4
  ]

unitTests :: TestTree
unitTests = testGroup "单元测试"
  [ testCase "字符串比较 1" $
      assertEqual "描述" "OK" "OK"

  , testCase "字符串比较 2" $  -- 应该失败
      assertEqual "描述" "fail" "fail!"
  ]
```

安装软件包：

```
cabal install tasty-smallcheck tasty-quickcheck tasty-hunit
```

使用 cabal 运行：

```
cabal exec runhaskell test.hs
```

Chapter 27: Testing with Tasty

Section 27.1: SmallCheck, QuickCheck and HUnit

```
import Test.Tasty
import Test.Tasty.SmallCheck as SC
import Test.Tasty.QuickCheck as QC
import Test.Tasty.HUnit


main :: IO ()
main = defaultMain tests

tests :: TestTree
tests = testGroup "Tests" [smallCheckTests, quickCheckTests, unitTests]

smallCheckTests :: TestTree
smallCheckTests = testGroup "SmallCheck Tests"
  [ SC.testProperty "String length <= 3" $
      \s -> length (take 3 (s :: String)) <= 3
  , SC.testProperty "String length <= 2" $  -- should fail
      \s -> length (take 3 (s :: String)) <= 2
  ]

quickCheckTests :: TestTree
quickCheckTests = testGroup "QuickCheck Tests"
  [ QC.testProperty "String length <= 5" $
      \s -> length (take 5 (s :: String)) <= 5
  , QC.testProperty "String length <= 4" $  -- should fail
      \s -> length (take 5 (s :: String)) <= 4
  ]

unitTests :: TestTree
unitTests = testGroup "Unit Tests"
  [ testCase "String comparison 1" $
      assertEqual "description" "OK" "OK"

  , testCase "String comparison 2" $  -- should fail
      assertEqual "description" "fail" "fail!"
  ]
```

Install packages:

```
cabal install tasty-smallcheck tasty-quickcheck tasty-hunit
```

Run with cabal:

```
cabal exec runhaskell test.hs
```

第28章：创建自定义数据类型

第28.1节：创建带有值构造函数参数的数据类型

值构造函数是返回某个数据类型值的函数。正因为如此，就像其他函数一样，它们可以接受一个或多个参数：

```
data Foo = Bar String Int | Biz String
```

让我们检查一下Bar值构造函数的类型。

```
:t Bar
```

打印结果

```
Bar :: String -> Int -> Foo
```

这证明了Bar确实是一个函数。

创建我们自定义类型的变量

```
let x = Bar "Hello" 10
let y = Biz "Goodbye"
```

第28.2节：创建带类型参数的数据类型

类型构造器可以接受一个或多个类型参数：

```
data Foo a b = Bar a b | Biz a b
```

Haskell中的类型参数必须以小写字母开头。我们的自定义数据类型还不是真正的类型。为了创建该类型的值，我们必须用实际类型替换所有类型参数。因为 a和 b可以是任何类型，所以我们的值构造器是多态函数。

创建我们自定义类型的变量

```
let x = Bar "Hello" 10      -- x :: Foo [Char] Integer
let y = Biz "Goodbye" 6.0   -- y :: Fractional b => Foo [Char] b
let z = Biz True False      -- z :: Foo Bool Bool
```

第28.3节：创建简单数据类型

在Haskell中创建自定义数据类型最简单的方法是使用data关键字：

```
data Foo = Bar | Biz
```

类型的名称在data和=之间指定，称为类型构造器(type constructor)。在=之后，我们指定数据类型的所有值构造器(value constructors)，使用|符号分隔。Haskell中有一条规则，所有类型和值构造器必须以大写字母开头。上述声明可以这样理解：

> 定义一个名为Foo的类型，它有两个可能的值：Bar和Biz。

Chapter 28: Creating Custom Data Types

Section 28.1: Creating a data type with value constructor parameters

Value constructors are functions that return a value of a data type. Because of this, just like any other function, they can take one or more parameters:

```
data Foo = Bar String Int | Biz String
```

Let's check the type of the Bar value constructor.

```
:t Bar
```

prints

```
Bar :: String -> Int -> Foo
```

which proves that Bar is indeed a function.

Creating variables of our custom type

```
let x = Bar "Hello" 10
let y = Biz "Goodbye"
```

Section 28.2: Creating a data type with type parameters

Type constructors can take one or more type parameters:

```
data Foo a b = Bar a b | Biz a b
```

Type parameters in Haskell must begin with a lowercase letter. Our custom data type is not a real type yet. In order to create values of our type, we must substitute all type parameters with actual types. Because a and b can be of any type, our value constructors are polymorphic functions.

Creating variables of our custom type

```
let x = Bar "Hello" 10      -- x :: Foo [Char] Integer
let y = Biz "Goodbye" 6.0   -- y :: Fractional b => Foo [Char] b
let z = Biz True False      -- z :: Foo Bool Bool
```

Section 28.3: Creating a simple data type

The easiest way to create a custom data type in Haskell is to use the data keyword:

```
data Foo = Bar | Biz
```

The name of the type is specified between data and =, and is called a **type constructor**. After = we specify all **value constructors** of our data type, delimited by the | sign. There is a rule in Haskell that all type and value constructors must begin with a capital letter. The above declaration can be read as follows:

> Define a type called Foo, which has two possible values: Bar and Biz.

创建我们自定义类型的变量

```
let x = Bar
```

上述语句创建了一个名为x的变量，类型为Foo。我们通过检查它的类型来验证这一点。

```
:t x
```

打印结果

```
x :: Foo
```

第28.4节：带有记录参数的自定义数据类型

假设我们想创建一个数据类型Person，它包含名字和姓氏、年龄、电话号码、街道、邮政编码和城镇。

我们可以这样写

```
data Person = Person String String Int Int String String String
```

如果我们现在想要获取电话号码，我们需要编写一个函数

```
getPhone :: Person -> Int
getPhone (Person _ _ _ phone _ _ _) = phone
```

嗯，这样没什么意思。我们可以用参数做得更好：

```
data Person' = Person' { firstName     :: String
                       , lastName      :: String
                       , age           :: Int
                       , phone         :: Int
                       , street        :: String
                       , code          :: String
                       , town          :: String }
```

现在我们得到了函数 phone 其中

```
:t phone
phone :: Person' -> Int
```

我们现在可以做任何想做的事情，例如：

```
printPhone :: Person' -> IO ()
printPhone = putStrLn . show . phone
```

我们也可以通过模式匹配绑定电话号码：

```
getPhone' :: Person' -> Int
getPhone' (Person {phone = p}) = p
```

有关参数的简便使用，请参见 RecordWildCards

Creating variables of our custom type

```
let x = Bar
```

The above statement creates a variable named `x` of type `Foo`. Let's verify this by checking its type.

```
:t x
```

prints

```
x :: Foo
```

Section 28.4: Custom data type with record parameters

Assume we want to create a data type Person, which has a first and last name, an age, a phone number, a street, a zip code and a town.

We could write

```
data Person = Person String String Int Int String String String
```

If we want now to get the phone number, we need to make a function

```
getPhone :: Person -> Int
getPhone (Person _ _ _ phone _ _ _) = phone
```

Well, this is no fun. We can do better using parameters:

```
data Person' = Person' { firstName     :: String
                       , lastName      :: String
                       , age           :: Int
                       , phone         :: Int
                       , street        :: String
                       , code          :: String
                       , town          :: String }
```

Now we get the function phone where

```
:t phone
phone :: Person' -> Int
```

We can now do whatever we want, eg:

```
printPhone :: Person' -> IO ()
printPhone = putStrLn . show . phone
```

We can also bind the phone number by Pattern Matching:

```
getPhone' :: Person' -> Int
getPhone' (Person {phone = p}) = p
```

For easy use of the parameters see RecordWildCards

第29章：Reactive-banana

第29.1节：向库中注入外部事件

这个示例不像 reactive-banana-wx 那样绑定到任何具体的GUI工具包。相反，它展示了如何将任意的IO操作注入到FRP机制中。

Control.Event.Handler模块提供了一个addHandler函数，该函数创建一对AddHandler a和a-> IO ()值。前者被 reactive-banana 本身用来获取一个Event a值，而后者是一个普通函数，用于触发相应的事件。

import Data.Char (toUpper) import Control.Event.Handler import Reactive.Banana main = do (inputHandler, inputFire) <- newAddHandler

在我们的例子中，处理器的a参数类型是String，但让编译器推断该类型的代码稍后会编写。

现在我们定义描述我们的FRP驱动系统的EventNetwork。这是通过compile函数完成的：

main = do (inputHandler, inputFire) <- newAddHandler compile $ do inputEvent <- fromAddHandler inputHandler

函数fromAddHandler将AddHandler a值转换为Event a，相关内容将在下一个示例中介绍。

最后，我们启动"事件循环"，该循环将在用户输入时触发事件：

main = do (inputHandler, inputFire) <- newAddHandler compile $ do ... forever $ do input <- getLine inputFire input

第29.2节：事件类型

在reactive-banana中，Event类型表示一段时间内的一系列事件流。一个Event类似于模拟脉冲信号，因为它在时间上不是连续的。因此，Event仅是Functor类型类的一个实例。你不能将两个Event合并，因为它们可能在不同时间触发。你可以对Event的[当前]值进行操作，并用某些IO动作对其做出反应。

对Event值的转换使用fmap完成：

main = do (inputHandler, inputFire) <- newAddHandler compile $ do inputEvent <- fromAddHandler inputHandler -- 将信号中的所有字符转换为大写 let inputEvent' = fmap (map toUpper) inputEvent

对Event的响应方式相同。首先用类型为a -> IO ()的动作对其进行fmap，然后传递给reactimate函数：

main = do (inputHandler, inputFire) <- newAddHandler compile $ do inputEvent <- fromAddHandler inputHandler -- 将信号中的所有字符转换为大写 let inputEvent' = fmap (map toUpper) inputEvent let inputEventReaction = fmap putStrLn inputEvent' -- 其类型为`Event (IO ()) reactimate inputEventReaction

现在每当调用inputFire "something"时，"SOMETHING"将被打印。

第29.3节：激活EventNetworks

由compile返回的EventNetwork必须被激活，reactimated事件才会生效。

```
main = do
    (inputHandler, inputFire) <- newAddHandler
```

Chapter 29: Reactive-banana

Section 29.1: Injecting external events into the library

This example is not tied to any concrete GUI toolkit, like reactive-banana-wx does, for instance. Instead it shows how to inject arbitrary IO actions into FRP machinery.

The Control.Event.Handler module provides an addHandler function which creates a pair of AddHandler a and a -> IO () values. The former is used by reactive-banana itself to obtain an Event a value, while the latter is a plain function that is used to trigger the corresponding event.

import Data.Char (toUpper) import Control.Event.Handler import Reactive.Banana main = do (inputHandler, inputFire) <- newAddHandler

In our case the a parameter of the handler is of type String, but the code that lets compiler infer that will be written later.

Now we define the EventNetwork that describes our FRP-driven system. This is done using compile function:

main = do (inputHandler, inputFire) <- newAddHandler compile $ do inputEvent <- fromAddHandler inputHandler

The fromAddHandler function transforms AddHandler a value into a Event a, which is covered in the next example.

Finally, we launch our "event loop", that would fire events on user input:

main = do (inputHandler, inputFire) <- newAddHandler compile $ do ... forever $ do input <- getLine inputFire input

Section 29.2: Event type

In reactive-banana the Event type represents a stream of some events in time. An Event is similar to an analog impulse signal in the sense that it is not continuous in time. As a result, Event is an instance of the Functor typeclass only. You can't combine two Events together because they may fire at different times. You can do something with an Event's [current] value and react to it with some IO action.

Transformations on Events value are done using **fmap**:

main = do (inputHandler, inputFire) <- newAddHandler compile $ do inputEvent <- fromAddHandler inputHandler -- turn all characters in the signal to upper case let inputEvent' = fmap (map toUpper) inputEvent

Reacting to an Event is done the same way. First you **fmap** it with an action of type a -> IO () and then pass it to reactimate function:

main = do (inputHandler, inputFire) <- newAddHandler compile $ do inputEvent <- fromAddHandler inputHandler -- turn all characters in the signal to upper case let inputEvent' = fmap (map toUpper) inputEvent let inputEventReaction = fmap putStrLn inputEvent' -- this has type `Event (IO ()) reactimate inputEventReaction

Now whenever inputFire "something" is called, "SOMETHING" would be printed.

Section 29.3: Actuating EventNetworks

EventNetworks returned by compile must be actuated before reactimated events have an effect.

```
main = do
    (inputHandler, inputFire) <- newAddHandler
```

```
eventNetwork <- compile $ do
        inputEvent <- fromAddHandler inputHandler
        let inputEventReaction = fmap putStrLn inputEvent
        reactimate inputEventReaction

    inputFire "这不会被打印到控制台！"
    actuate eventNetwork
inputFire "这将被打印到控制台！"
```

第29.4节：行为类型

为了表示连续信号，reactive-banana 提供了Behavior a类型。与Event不同，Behavior是一个Applicative，这使你可以使用n元纯函数（使用<$>和<*>）组合n个Behavior。

要从Event a获得Behavior a，可以使用accumE函数：

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
inputBehavior <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
```

accumE接受Behavior的初始值和一个Event，该事件包含一个将其设置为新值的函数。

与事件（Event）类似，你可以使用fmap来处理当前的行为（Behavior）值，但你也可以将它们与(<*>)结合使用。

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
inputBehavior  <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        inputBehavior' <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        let constantTrueBehavior = (==) <$> inputBehavior <*> inputBehavior'
```

要对行为（Behavior）的变化做出反应，可以使用changes函数：

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
inputBehavior  <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        inputBehavior' <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        let constantTrueBehavior = (==) <$> inputBehavior <*> inputBehavior'
inputChanged <- changes inputBehavior
```

唯一需要注意的是changes返回的是事件（Event）(Future a)而不是事件（Event） a。因此，应使用reactimate'而不是reactimate。其背后的原理可以从文档中获得。

```
    eventNetwork <- compile $ do
        inputEvent <- fromAddHandler inputHandler
        let inputEventReaction = fmap putStrLn inputEvent
        reactimate inputEventReaction

    inputFire "This will NOT be printed to the console!"
    actuate eventNetwork
    inputFire "This WILL be printed to the console!"
```

Section 29.4: Behavior type

To represent continious signals, reactive-banana features Behavior a type. Unlike Event, a Behavior is an Applicative, which lets you combine n Behaviors using an n-ary pure function (using <$> and <*>).

To obtain a Behavior a from the Event a there is accumE function:

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
        inputBehavior <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
```

accumE takes Behavior's initial value and an Event, containing a function that would set it to the new value.

As with Events, you can use **fmap** to work with current Behaviors value, but you can also combine them with (<*>).

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
        inputBehavior  <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        inputBehavior' <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        let constantTrueBehavior = (==) <$> inputBehavior <*> inputBehavior'
```

To react on Behavior changes there is a changes function:

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
        inputBehavior  <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        inputBehavior' <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        let constantTrueBehavior = (==) <$> inputBehavior <*> inputBehavior'
        inputChanged <- changes inputBehavior
```

The only thing that should be noted is that changes return Event (Future a) instead of Event a. Because of this, reactimate' should be used instead of reactimate. The rationale behind this can be obtained from the documentation.

第30章：优化

第30.1节：为性能分析编译程序

GHC编译器对带有性能分析注释的编译提供了成熟的支持。

在编译时使用-prof和-fprof-auto标志，将为你的二进制文件添加运行时使用的性能分析支持。

假设我们有如下程序：

```
main = print (fib 30)
fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)
```

像这样编译：

```
ghc -prof -fprof-auto -rtsopts Main.hs
```

然后使用运行时系统选项进行性能分析运行：

```
./Main +RTS -p
```

程序执行结束后（程序退出时）会生成一个main.prof文件，里面包含各种性能分析信息，比如成本中心，显示运行代码各部分的成本分解：

```
2011年10月12日 星期三 16:14  时间和内存分配性能分析报告  (最终)

         Main +RTS -p -RTS

总时间  =         0.68 秒   (34 个时钟周期 @ 20 毫秒)
      总分配内存 = 204,677,844 字节  (不包括性能分析开销)

成本中心 模块  %时间 %内存分配

fib          Main      100.0   100.0


单独      继承
成本中心 模块                   编号      入口次数  %时间 %内存分配    %时间 %内存分配

MAIN          MAIN                    102              0     0.0      0.0     100.0   100.0
 CAF          GHC.IO.Handle.FD        128              0     0.0      0.0       0.0     0.0
 CAF          GHC.IO.Encoding.Iconv   120              0     0.0      0.0       0.0     0.0
 CAF          GHC.Conc.Signal         110              0     0.0      0.0       0.0     0.0
 CAF          Main                    108              0     0.0      0.0     100.0   100.0
  main        Main                    204              1     0.0      0.0     100.0   100.0
   fib        Main                    205        2692537   100.0    100.0     100.0   100.0
```

第30.2节：成本中心

成本中心是对Haskell程序的注释，可以由GHC编译器自动添加——使用-fprot-auto——或者由程序员使用{-# SCC "name" #-}<expression>添加，其中"name"是你想要的任何名称，而**<expression>**是任何有效的Haskell表达式：

Chapter 30: Optimization

Section 30.1: Compiling your Program for Profiling

The GHC compiler has mature support for compiling with profiling annotations.

Using the `-prof` and `-fprof-auto` flags when compiling will add support to your binary for profiling flags for use at runtime.

Suppose we have this program:

```
main = print (fib 30)
fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)
```

Compiled it like so:

```
ghc -prof -fprof-auto -rtsopts Main.hs
```

Then ran it with runtime system options for profiling:

```
./Main +RTS -p
```

We will see a `main.prof` file created post execution (once the program has exited), and this will give us all sorts of profiling information such as cost centers which gives us a breakdown of the cost associated with running the various parts of the code:

```
    Wed Oct 12 16:14 2011 Time and Allocation Profiling Report  (Final)

           Main +RTS -p -RTS

        total time  =        0.68 secs   (34 ticks @ 20 ms)
        total alloc = 204,677,844 bytes  (excludes profiling overheads)

COST CENTRE MODULE  %time %alloc

fib         Main    100.0  100.0


                                                      individual     inherited
COST CENTRE MODULE                  no.     entries  %time %alloc   %time %alloc

MAIN        MAIN                    102           0    0.0    0.0   100.0  100.0
 CAF        GHC.IO.Handle.FD        128           0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Encoding.Iconv   120           0    0.0    0.0     0.0    0.0
 CAF        GHC.Conc.Signal         110           0    0.0    0.0     0.0    0.0
 CAF        Main                    108           0    0.0    0.0   100.0  100.0
  main      Main                    204           1    0.0    0.0   100.0  100.0
   fib      Main                    205     2692537  100.0  100.0   100.0  100.0
```

Section 30.2: Cost Centers

Cost centers are annotations on a Haskell program which can be added automatically by the GHC compiler -- using `-fprot-auto` -- or by a programmer using `{-# SCC "name" #-} <expression>`, where "name" is any name you wish and **`<expression>`** is any valid Haskell expression:

```haskell
-- Main.hs
main :: IO ()
main = do let l = [1..9999999]
          print $ {-# SCC "print_list" #-} (length l)
```

使用-fprof编译并用+RTS -p运行，例如ghc -prof -rtsopts Main.hs && ./Main.hs +RTS -p，程序退出后会生成Main.prof文件。

```haskell
-- Main.hs
main :: IO ()
main = do let l = [1..9999999]
          print $ {-# SCC "print_list" #-} (length l)
```

Compiling with `-fprof` and running with `+RTS -p` e.g. `ghc -prof -rtsopts Main.hs && ./Main.hs +RTS -p` would produce `Main.prof` once the program's exited.

第31章：并发

第31.1节：使用`forkIO`创建线程

Haskell支持多种形式的并发，最明显的是使用forkIO创建线程。

函数 forkIO :: IO () -> IO ThreadId 接受一个 IO 操作并返回其 ThreadId，同时该操作将在后台运行。

我们可以用ghci非常简洁地演示这一点：

```
Prelude Control.Concurrent> forkIO $ (print . sum) [1..000000000]
线程ID 290
Prelude Control.Concurrent> forkIO $ print "hi!"
"hi!"
-- 一段时间后....
Prelude Control.Concurrent> 50000005000000
```

两个操作都会在后台运行，第二个几乎可以保证在第一个完成之前结束！

第31.2节：使用`MVar`在线程间通信

使用MVar a类型及其配套函数在

Control.Concurrent中，在线程间传递信息非常简单：

- newEmptyMVar :: IO (MVar a) -- 创建一个新的MVar a
- newMVar :: a -> IO (MVar a) -- 使用给定值创建一个新的MVar
- takeMVar :: MVar a -> IO a -- 从给定的MVar中取出值，若无值则**阻塞**直到有值可用
- putMVar :: MVar a -> a -> IO () -- 将给定值放入MVar，若MVar非空则阻塞直到其为空

让我们在一个线程中对1到一亿的数字求和，并等待结果：

```
import Control.Concurrent
main = do
m <- newEmptyMVar
forkIO $ putMVar m $ sum [1..10000000]
  print =<< takeMVar m  -- takeMVar 会阻塞直到 m 非空！
```

一个更复杂的示例可能是接收用户输入并在后台求和，同时等待更多输入：

```
main2 = loop
  其中
loop = do
m <- newEmptyMVar
n <- getLine
        putStrLn "正在计算，请稍候"
        -- 在另一个线程中，解析用户输入并求和
        forkIO $ putMVar m $ 求和 [1..(读取 n :: Int)]
        -- 在另一个线程中，等待求和完成然后打印结果
        forkIO $ 打印 =<< 取MVar m
       循环
```

如前所述，如果调用akeMVar且MVar为空，它会阻塞直到另一个线程向MVar中放入内容，这可能导致哲学家就餐问题。对于putMVar也是同样的情况：如果它已满，它会

Chapter 31: Concurrency

Section 31.1: Spawning Threads with `forkIO`

Haskell supports many forms of concurrency and the most obvious being forking a thread using forkIO.

The function forkIO :: IO () -> IO ThreadId takes an IO action and returns its ThreadId, meanwhile the action will be run in the background.

We can demonstrate this quite succinctly using ghci:

```
Prelude Control.Concurrent> forkIO $ (print . sum) [1..100000000]
ThreadId 290
Prelude Control.Concurrent> forkIO $ print "hi!"
"hi!"
-- some time later....
Prelude Control.Concurrent> 50000005000000
```

Both actions will run in the background, and the second is almost guaranteed to finish before the last!

Section 31.2: Communicating between Threads with `MVar`

It is very easy to pass information between threads using the MVar a type and its accompanying functions in Control.Concurrent:

- newEmptyMVar :: IO (MVar a) -- creates a new MVar a
- newMVar :: a -> IO (MVar a) -- creates a new MVar with the given value
- takeMVar :: MVar a -> IO a -- retrieves the value from the given MVar, or **blocks** until one is available
- putMVar :: MVar a -> a -> IO () -- puts the given value in the MVar, or **blocks** until it's empty

Let's sum the numbers from 1 to 100 million in a thread and wait on the result:

```
import Control.Concurrent
main = do
  m <- newEmptyMVar
  forkIO $ putMVar m $ sum [1..10000000]
  print =<< takeMVar m  -- takeMVar will block 'til m is non-empty!
```

A more complex demonstration might be to take user input and sum in the background while waiting for more input:

```
main2 = loop
  where
    loop = do
        m <- newEmptyMVar
        n <- getLine
        putStrLn "Calculating. Please wait"
        -- In another thread, parse the user input and sum
        forkIO $ putMVar m $ sum [1..(read n :: Int)]
        -- In another thread, wait 'til the sum's complete then print it
        forkIO $ print =<< takeMVar m
        loop
```

As stated earlier, if you call takeMVar and the MVar is empty, it blocks until another thread puts something into the MVar, which could result in a Dining Philosophers Problem. The same thing happens with putMVar: if it's full, it'll

阻塞直到它变为空！

看下面这个函数：

```
concurrent ma mb = do
  a <- takeMVar ma
  b <- takeMVar mb
  putMVar ma a
  putMVar mb b
```

我们用一些MVar运行这两个函数

```
concurrent ma mb     -- 新线程 1
concurrent mb ma     -- 新线程 2
```

可能发生的情况是：

1. 线程 1 读取ma并阻塞ma
2. 线程 2 读取mb并阻塞mb

现在线程 1 无法读取mb，因为线程 2 阻塞了它，线程 2 也无法读取ma，因为线程 1 阻塞了它。一个经典的死锁！

第31.3节：带有软件事务内存的原子块

Haskell中另一个强大且成熟的并发工具是软件事务内存（Software Transactional Memory），它允许多个线程以原子方式写入类型为TVar a的单个变量。

TVar a是与STM单子相关的主要类型，代表事务变量。它们的使用方式类似于MVar，但在STM单子中通过以下函数使用：

atomically :: STM a -> IO a

以原子方式执行一系列STM操作。

readTVar :: TVar a -> STM a

读取TVar的值，例如：

```
value <- readTVar t
```

writeTVar :: TVar a -> a -> STM ()

向指定的TVar写入一个值。

```
t <- newTVar Nothing
writeTVar t (Just "Hello")
```

此示例取自Haskell维基：

```
import Control.Monad
import Control.Concurrent
import Control.Concurrent.STM

main = do
  -- 初始化一个新的 TVar
shared <- atomically $ newTVar 0
```

block 'til it's empty!

Take the following function:

```
concurrent ma mb = do
  a <- takeMVar ma
  b <- takeMVar mb
  putMVar ma a
  putMVar mb b
```

We run the the two functions with some MVars

```
concurrent ma mb     -- new thread 1
concurrent mb ma     -- new thread 2
```

What could happen is that:

1. Thread 1 reads ma and blocks ma
2. Thread 2 reads mb and thus blocks mb

Now Thread 1 cannot read mb as Thread 2 has blocked it, and Thread 2 cannot read ma as Thread 1 has blocked it. A classic deadlock!

Section 31.3: Atomic Blocks with Software Transactional Memory

Another powerful & mature concurrency tool in Haskell is Software Transactional Memory, which allows for multiple threads to write to a single variable of type TVar a in an atomic manner.

TVar a is the main type associated with the STM monad and stands for transactional variable. They're used much like MVar but within the STM monad through the following functions:

atomically :: STM a -> IO a

Perform a series of STM actions atomically.

readTVar :: TVar a -> STM a

Read the TVar's value, e.g.:

```
value <- readTVar t
```

writeTVar :: TVar a -> a -> STM ()

Write a value to the given TVar.

```
t <- newTVar Nothing
writeTVar t (Just "Hello")
```

This example is taken from the Haskell Wiki:

```
import Control.Monad
import Control.Concurrent
import Control.Concurrent.STM

main = do
  -- Initialise a new TVar
  shared <- atomically $ newTVar 0
```

```haskell
  -- 读取值
before <- atomRead shared
  putStrLn $ "之前: " ++ show before
  forkIO $ 25 `timesDo` (dispVar shared >> milliSleep 20)
  forkIO $ 10 `timesDo` (appV ((+) 2) shared >> milliSleep 50)
  forkIO $ 20 `timesDo` (appV pred shared >> milliSleep 25)
  milliSleep 800
after <- atomRead shared
  putStrLn $ "之后: " ++ show after
  where timesDo = replicateM_
milliSleep = threadDelay . (*) 1000

atomRead = atomically . readTVar
dispVar x = atomRead x >>= print
appV fn x = atomically $ readTVar x >>= writeTVar x . fn
```

```haskell
  -- Read the value
  before <- atomRead shared
  putStrLn $ "Before: " ++ show before
  forkIO $ 25 `timesDo` (dispVar shared >> milliSleep 20)
  forkIO $ 10 `timesDo` (appV ((+) 2) shared >> milliSleep 50)
  forkIO $ 20 `timesDo` (appV pred shared >> milliSleep 25)
  milliSleep 800
  after <- atomRead shared
  putStrLn $ "After: " ++ show after
  where timesDo = replicateM_
       milliSleep = threadDelay . (*) 1000

atomRead = atomically . readTVar
dispVar x = atomRead x >>= print
appV fn x = atomically $ readTVar x >>= writeTVar x . fn
```

第32章：函数组合

第32.1节：从右到左的组合

(.) 允许我们组合两个函数，将一个函数的输出作为另一个函数的输入：

```
(f . g) x = f (g x)
```

例如，如果我们想对输入数字的后继进行平方，可以写成

```
((^2) . succ) 1          --    4
```

还有 (<<<)，它是 (.) 的别名。所以，

```
(+ 1) <<< sqrt $ 25     --     6
```

第32.2节：与二元函数的组合

常规组合适用于一元函数。在二元函数的情况下，我们可以定义

```
(f .: g) x y = f (g x y)            -- 这也等于
                 = f ((g x) y)
                 = (f . g x) y          -- 根据 (.) 的定义
                 = (f .) (g x) y
                 = ((f .) . g) x y
```

因此，(f .: g) = ((f .) . g) 通过 eta-缩减，且进一步，

```
(.:) f g     = ((f .) . g)
                = (.) (f .) g
                = (.) ((.) f) g
                = ((.) . (.)) f g
```

所以 (.:) = ((.) . (.))，这是一个半著名的定义。

示例：

```
(map (+1) .: filter) even [1..5]      --  [3,5]
(length    .: filter) even [1..5]      --  2
```

第32.3节：从左到右的组合

Control.Category 定义了 (>>>)，当专门用于函数时，是

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

示例：

```
sqrt >>> (+ 1) $ 25     --     6.0
```

Chapter 32: Function composition

Section 32.1: Right-to-left composition

(.) lets us compose two functions, feeding output of one as an input to the other:

```
(f . g) x = f (g x)
```

For example, if we want to square the successor of an input number, we can write

```
((^2) . succ) 1          --    4
```

There is also (<<<) which is an alias to (.). So,

```
(+ 1) <<< sqrt $ 25     --     6
```

Section 32.2: Composition with binary function

The regular composition works for unary functions. In the case of binary, we can define

```
(f .: g) x y = f (g x y)          -- which is also
             = f ((g x) y)
             = (f . g x) y        -- by definition of (.)
             = (f .) (g x) y
             = ((f .) . g) x y
```

Thus, (f .: g) = ((f .) . g) by eta-contraction, and furthermore,

```
(.:) f g    = ((f .) . g)
            = (.) (f .) g
            = (.) ((.) f) g
            = ((.) . (.)) f g
```

so (.:) = ((.) . (.)), a semi-famous definition.

Examples:

```
(map (+1) .: filter) even [1..5]      --  [3,5]
(length    .: filter) even [1..5]      --  2
```

Section 32.3: Left-to-right composition

Control.Category defines (>>>), which, when specialized to functions, is

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

Example:

```
sqrt >>> (+ 1) $ 25     --     6.0
```

第33章：数据库

第33.1节：Postgres

Postgresql-simple 是一个用于与 PostgreSQL 后端数据库通信的中级 Haskell 库。它非常简单易用，并提供了一个类型安全的 API 用于读写数据库。

运行一个简单的查询就像这样简单：

```
{-# LANGUAGE OverloadedStrings #-}

import Database.PostgreSQL.Simple

main :: IO ()
main = do
  -- 使用 libpq 字符串连接
conn <- connectPostgreSQL "host='my.dbhost' port=5432 user=bob pass=bob"
  [Only i] <- query_ conn "select 2 + 2"  -- 无参数替换执行
  print i
```

参数替换

PostreSQL-Simple 支持使用 query 进行安全的参数化查询的参数替换：

```
main :: IO ()
main = do
  -- 使用 libpq 字符串连接
conn <- connectPostgreSQL "host='my.dbhost' port=5432 user=bob pass=bob"
  [Only i] <- query conn "select ? + ?" [1, 1]
  print i
```

执行插入或更新

你可以使用 execute 来运行插入/更新 SQL 查询：

```
main :: IO ()
main = do
  -- 使用 libpq 字符串连接
conn <- connectPostgreSQL "host='my.dbhost' port=5432 user=bob pass=bob"
  execute conn "insert into people (name, age) values (?, ?)" ["Alex", 31]
```

Chapter 33: Databases

Section 33.1: Postgres

Postgresql-simple is a mid-level Haskell library for communicating with a PostgreSQL backend database. It is very simple to use and provides a type-safe API for reading/writing to a DB.

Running a simple query is as easy as:

```
{-# LANGUAGE OverloadedStrings #-}

import Database.PostgreSQL.Simple

main :: IO ()
main = do
  -- Connect using libpq strings
  conn <- connectPostgreSQL "host='my.dbhost' port=5432 user=bob pass=bob"
  [Only i] <- query_ conn "select 2 + 2"  -- execute with no parameter substitution
  print i
```

Parameter substitution

PostreSQL-Simple supports parameter substitution for safe parameterised queries using query:

```
main :: IO ()
main = do
  -- Connect using libpq strings
  conn <- connectPostgreSQL "host='my.dbhost' port=5432 user=bob pass=bob"
  [Only i] <- query conn "select ? + ?" [1, 1]
  print i
```

Executing inserts or updates

You can run inserts/update SQL queries using execute:

```
main :: IO ()
main = do
  -- Connect using libpq strings
  conn <- connectPostgreSQL "host='my.dbhost' port=5432 user=bob pass=bob"
  execute conn "insert into people (name, age) values (?, ?)" ["Alex", 31]
```

第34章：Data.Aeson - Haskell中的JSON

第34.1节：使用泛型的智能编码和解码

使用Aeson将Haskell数据类型编码为JSON最简单快捷的方法是使用泛型。

```
{-# LANGUAGE DeriveGeneric #-}

import GHC.Generics
import Data.Text
import Data.Aeson
import Data.ByteString.Lazy
```

首先让我们创建一个数据类型Person：

```
data Person = Person { firstName :: Text
                     , lastName  :: Text
                     , age       :: Int
                     } deriving (Show, Generic)
```

为了使用Data.Aeson包中的encode和decode函数，我们需要使Person成为ToJSON和FromJSON的实例。由于我们为Person派生了Generic，可以为这些类创建空实例。这些方法的默认定义是基于Generic类型类提供的方法。

```
实例 ToJSON Person
实例 FromJSON Person
```

完成！为了提高编码速度，我们可以稍微修改一下ToJSON实例：

```
instance ToJSON Person where
    toEncoding = genericToEncoding defaultOptions
```

现在我们可以使用encode函数将Person转换为（惰性）字节串：

```
encodeNewPerson :: Text -> Text -> Int -> ByteString
encodeNewPerson first last age = encode $ Person first last age
```

解码时我们只需使用decode：

```
> encodeNewPerson "Hans" "Wurst" 30
"{🟩lastName🟩:🟩Wurst🟩,🟩age🟩:30,🟩firstName🟩:🟩Hans🟩}"


> decode $ encodeNewPerson "Hans" "Wurst" 30
Just (Person {firstName = "Hans", lastName = "Wurst", age = 30})
```

第34.2节：快速生成Data.Aeson.Value的方法

```
{-# LANGUAGE OverloadedStrings #-}
module Main where

import Data.Aeson

main :: IO ()
main = do
```

Chapter 34: Data.Aeson - JSON in Haskell

Section 34.1: Smart Encoding and Decoding using Generics

The easiest and quickest way to encode a Haskell data type to JSON with Aeson is using generics.

```
{-# LANGUAGE DeriveGeneric #-}

import GHC.Generics
import Data.Text
import Data.Aeson
import Data.ByteString.Lazy
```

First let us create a data type Person:

```
data Person = Person { firstName :: Text
                     , lastName  :: Text
                     , age       :: Int
                     } deriving (Show, Generic)
```

In order to use the `encode` and `decode` function from the `Data.Aeson` package we need to make `Person` an instance of `ToJSON` and `FromJSON`. Since we derive `Generic` for `Person`, we can create empty instances for these classes. The default definitions of the methods are defined in terms of the methods provided by the `Generic` type class.

```
instance ToJSON Person
instance FromJSON Person
```

Done! In order to improve the encoding speed we can slightly change the `ToJSON` instance:

```
instance ToJSON Person where
    toEncoding = genericToEncoding defaultOptions
```

Now we can use the `encode` function to convert `Person` to a (lazy) Bytestring:

```
encodeNewPerson :: Text -> Text -> Int -> ByteString
encodeNewPerson first last age = encode $ Person first last age
```

And to decode we can just use `decode`:

```
> encodeNewPerson "Hans" "Wurst" 30
"{🟩lastName🟩:🟩Wurst🟩,🟩age🟩:30,🟩firstName🟩:🟩Hans🟩}"


> decode $ encodeNewPerson "Hans" "Wurst" 30
Just (Person {firstName = "Hans", lastName = "Wurst", age = 30})
```

Section 34.2: A quick way to generate a Data.Aeson.Value

```
{-# LANGUAGE OverloadedStrings #-}
module Main where

import Data.Aeson

main :: IO ()
main = do
```

```
  let example = Data.Aeson.object [ "key" .= (5 :: Integer), "somethingElse" .= (2 :: Integer) ] ::
值
  print . encode $ 示例
```

第34.3节：可选字段

有时，我们希望JSON字符串中的某些字段是可选的。例如，

```
data Person = Person { firstName :: Text
                     , lastName  :: Text
                     , age       :: Maybe Int
                     }
```

这可以通过以下方式实现

```
import Data.Aeson.TH

$(deriveJSON defaultOptions{omitNothingFields = True} ''Person)
```

```
  let example = Data.Aeson.object [ "key" .= (5 :: Integer), "somethingElse" .= (2 :: Integer) ] ::
Value
  print . encode $ example
```

Section 34.3: Optional Fields

Sometimes, we want some fields in the JSON string to be optional. For example,

```
data Person = Person { firstName :: Text
                     , lastName  :: Text
                     , age       :: Maybe Int
                     }
```

This can be achieved by

```
import Data.Aeson.TH

$(deriveJSON defaultOptions{omitNothingFields = True} ''Person)
```

第35章：高阶函数

第35.1节：高阶函数基础

在继续之前，请审查部分申请。

在 Haskell 中，可以将其他函数作为参数传入或返回函数的函数称为高阶函数。

以下都是高阶函数：

```
map       :: (a -> b) -> [a] -> [b]
filter    :: (a -> Bool) -> [a] -> [a]
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
iterate   :: (a -> a) -> a -> [a]
zipWith   :: (a -> b -> c) -> [a] -> [b] -> [c]
scanr     :: (a -> b -> b) -> b -> [a] -> [b]
scanl     :: (b -> a -> b) -> b -> [a] -> [b]
```

这些函数特别有用，因为它们允许我们在已有函数的基础上创建新函数，方法是将函数作为参数传递给其他函数。因此得名*高阶函数*。

考虑：

```
Prelude> :t (map (+3))
(map (+3)) :: Num b => [b] -> [b]

Prelude> :t (map (=='c'))
(map (=='c')) :: [Char] -> [Bool]

Prelude> :t (map zipWith)
(map zipWith) :: [a -> b -> c] -> [[a] -> [b] -> [c]]
```

这种轻松创建函数的能力（例如这里使用的部分应用）是函数式编程特别强大的特性之一，它使我们能够推导出简短优雅的解决方案，而这些解决方案在其他语言中可能需要几十行代码。例如，以下函数给出了两个列表中对齐元素的数量。

```
aligned :: [a] ->  [a] -> Int
aligned xs ys = length (filter id (zipWith (==) xs ys))
```

第35.2节：Lambda表达式

Lambda表达式类似于其他语言中的匿名函数。

Lambda表达式是开放公式，同时指定了要绑定的变量。求值（找到函数调用的值）是通过将lambda表达式主体中的绑定变量替换为用户提供的参数来实现的。简单来说，lambda表达式允许我们通过变量绑定和替换来表达函数。

Lambda表达式看起来像

```
\x -> let {y = ...x...} in y
```

在lambda表达式中，箭头左侧的变量被视为在右侧

Chapter 35: Higher-order functions

Section 35.1: Basics of Higher Order Functions

Review Partial Application before proceeding.

In Haskell, a function that can take other functions as arguments or return functions is called a *higher-order function*.

The following are all *higher-order functions*:

```
map       :: (a -> b) -> [a] -> [b]
filter    :: (a -> Bool) -> [a] -> [a]
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
iterate   :: (a -> a) -> a -> [a]
zipWith   :: (a -> b -> c) -> [a] -> [b] -> [c]
scanr     :: (a -> b -> b) -> b -> [a] -> [b]
scanl     :: (b -> a -> b) -> b -> [a] -> [b]
```

These are particularly useful in that they allow us to create new functions on top of the ones we already have, by passing functions as arguments to other functions. Hence the name, *higher-order functions*.

Consider:

```
Prelude> :t (map (+3))
(map (+3)) :: Num b => [b] -> [b]

Prelude> :t (map (=='c'))
(map (=='c')) :: [Char] -> [Bool]

Prelude> :t (map zipWith)
(map zipWith) :: [a -> b -> c] -> [[a] -> [b] -> [c]]
```

This ability to easily create functions (like e.g. by partial application as used here) is one of the features that makes functional programming particularly powerful and allows us to derive short, elegant solutions that would otherwise take dozens of lines in other languages. For example, the following function gives us the number of aligned elements in two lists.

```
aligned :: [a] ->  [a] -> Int
aligned xs ys = length (filter id (zipWith (==) xs ys))
```

Section 35.2: Lambda Expressions

Lambda expressions are similar to *anonymous functions* in other languages.

Lambda expressions are open formulas which also specify variables which are to be bound. Evaluation (finding the value of a function call) is then achieved by substituting the bound variables in the lambda expression's body, with the user supplied arguments. Put simply, lambda expressions allow us to express functions by way of variable binding and substitution.

Lambda expressions look like

```
\x -> let {y = ...x...} in y
```

Within a lambda expression, the variables on the left-hand side of the arrow are considered bound in the right-

即函数体中被绑定的变量。

考虑数学函数

```
f(x) = x^2
```

作为Haskell定义，它是

```
f     x =  x^2

f = \x -> x^2
```

这意味着函数f等价于lambda表达式\x -> x^2。

考虑高阶函数map的参数，它是类型为a -> b的函数。如果它在对map的调用中只使用了一次，并且在程序的其他地方都没有使用，那么将其指定为lambda表达式而不是命名这样一个一次性函数会更方便。用lambda表达式写出来，

```
\x -> let {y = ...x...} in y
```

x持有类型为a的值，...x...是一个指代变量x的Haskell表达式，y持有类型为b的值。例如，我们可以写成如下形式

```
map (\x -> x + 3)

map (\(x,y) -> x * y)

map (\xs -> 'c':xs) ["apples", "oranges", "mangos"]

map (\f -> zipWith f [1..5] [1..5]) [(+), (*), (-)]
```

第35.3节：柯里化

在Haskell中，所有函数都被视为柯里化函数：也就是说，Haskell中的所有函数都只接受一个参数。

我们来看函数div：

div :: Int -> Int -> Int

如果我们用6和2调用这个函数，结果不出所料是3：

Prelude> div 6 2 3

然而，这个行为并不像我们想象的那样。首先，div6被求值，返回一个函数，类型为Int -> Int。然后将这个结果函数应用于值2，得到3。

当我们查看函数的类型签名时，可以将思维从"接受两个Int类型的参数"转变为"接受一个Int并返回一个接受Int的函数"。如果考虑到类型符号中的箭头是向右结合的，这一点就更加明确，因此div实际上可以这样理解：

div :: Int -> (Int -> Int)

一般来说，大多数程序员在学习语言时可以忽略这种行为。从理论角度看，"当所有函数都被统一视为（一个参数输入，一个结果输出）时，形式证明会更简单。"

hand side, i.e. the function's body.

Consider the mathematical function

```
f(x) = x^2
```

As a Haskell definition it is

```
f     x =  x^2

f = \x -> x^2
```

which means that the function f is equivalent to the lambda expression \x -> x^2.

Consider the parameter of the higher-order function **map**, that is a function of type a -> b. In case it is used only once in a call to **map** and nowhere else in the program, it is convenient to specify it as a lambda expression instead of naming such a throwaway function. Written as a lambda expression,

```
\x -> let {y = ...x...} in y
```

x holds a value of type a, ...x... is a Haskell expression that refers to the variable x, and y holds a value of type b. So, for example, we could write the following

```
map (\x -> x + 3)

map (\(x,y) -> x * y)

map (\xs -> 'c':xs) ["apples", "oranges", "mangos"]

map (\f -> zipWith f [1..5] [1..5]) [(+), (*), (-)]
```

Section 35.3: Currying

In Haskell, all functions are considered curried: that is, all functions in Haskell take just *one* argument.

Let's take the function **div**:

div :: Int -> Int -> Int

If we call this function with 6 and 2 we unsurprisingly get 3:

Prelude> div 6 2 3

However, this doesn't quite behave in the way we might think. First **div 6** is evaluated and **returns a function** of type Int -> Int. This resulting function is then applied to the value 2 which yields 3.

When we look at the type signature of a function, we can shift our thinking from "takes two arguments of type Int" to "takes one Int and returns a function that takes an Int". This is reaffirmed if we consider that arrows in the type notation associate *to the right*, so **div** can in fact be read thus:

div :: Int -> (Int -> Int)

In general, most programmers can ignore this behaviour at least while they're learning the language. From a theoretical point of view, "formal proofs are easier when all functions are treated uniformly (one argument in, one result out)."

第36章：容器 - Data.Map

第36.1节：导入模块

containers包中的Data.Map模块提供了一个Map结构，既有严格实现也有惰性实现。

使用Data.Map时，通常会以限定方式导入，以避免与Prelude中已定义的函数冲突：

import qualified Data.Map as Map

因此我们需要在Map函数调用前加上Map.，例如：

```
Map.empty -- 给我一个空的 Map
```

第36.2节：Monoid 实例

Map k v 提供了一个 Monoid 实例，语义如下：

- mempty 是空的 Map，即与 Map.empty 相同
- m1 <> m2 是 m1 和 m2 的左偏联合，即如果某个键同时存在于 m1 和 m2 中，则 m1 中的值会被选取作为 m1 <> m2 的值。该操作在 Monoid 实例之外也可用，名称为 Map.union。

第36.3节：构造

我们可以通过元组列表这样创建一个 Map：

Map.fromList [("Alex", 31), ("Bob", 22)]

Map 也可以通过单个值构造：

> Map.singleton "Alex" 31 fromList [("Alex",31)]

还有一个 empty 函数。

empty :: Map k a

Data.Map 还支持典型的集合操作，如union、difference和intersection。

第36.4节：检查是否为空

我们使用null函数来检查给定的Map是否为空：

> Map.null $ Map.fromList [("Alex", 31), ("Bob", 22)] False > Map.null $ Map.empty True

第36.5节：查找值

对映射有许多查询操作。

member :: Ord k => k -> Map k a -> Bool 如果类型为k的键在Map k a中，则返回True：

> Map.member "Alex" $ Map.singleton "Alex" 31 True > Map.member "Jenny" $ Map.empty False

notMember 类似：

Chapter 36: Containers - Data.Map

Section 36.1: Importing the Module

The `Data.Map` module in the containers package provides a `Map` structure that has both strict and lazy implementations.

When using `Data.Map`, one usually imports it qualified to avoid clashes with functions already defined in Prelude:

import qualified Data.Map as Map

So we'd then prepend `Map` function calls with `Map.`, e.g.

```
Map.empty -- give me an empty Map
```

Section 36.2: Monoid instance

`Map k v` provides a Monoid instance with the following semantics:

- `mempty` is the empty `Map`, i.e. the same as `Map.empty`
- `m1 <> m2` is the left-biased union of `m1` and `m2`, i.e. if any key is present both in `m1` and `m2`, then the value from `m1` is picked for `m1 <> m2`. This operation is also available outside the `Monoid` instance as `Map.union`.

Section 36.3: Constructing

We can create a Map from a list of tuples like this:

Map.fromList [("Alex", 31), ("Bob", 22)]

A Map can also be constructed with a single value:

> Map.singleton "Alex" 31 fromList [("Alex",31)]

There is also the `empty` function.

empty :: Map k a

Data.Map also supports typical set operations such as `union`, `difference` and `intersection`.

Section 36.4: Checking If Empty

We use the **`null`** function to check if a given Map is empty:

> Map.null $ Map.fromList [("Alex", 31), ("Bob", 22)] False > Map.null $ Map.empty True

Section 36.5: Finding Values

There are many querying operations on maps.

`member :: Ord k => k -> Map k a -> Bool` yields `True` if the key of type `k` is in `Map k a`:

> Map.member "Alex" $ Map.singleton "Alex" 31 True > Map.member "Jenny" $ Map.empty False

`notMember` is similar:

> Map.notMember "Alex" $ Map.singleton "Alex" 31 False > Map.notMember "Jenny" $ Map.empty True

你也可以使用 findWithDefault :: Ord k => a -> k -> Map k a -> a 来在键不存在时返回默认值：

Map.findWithDefault 'x' 1 (fromList [(5,'a'), (3,'b')]) == 'x' Map.findWithDefault 'x' 5 (fromList [(5,'a'), (3,'b')]) == 'a'

第36.6节：插入元素

插入元素很简单：

> let m = Map.singleton "Alex" 31 fromList [("Alex",31)] > Map.insert "Bob" 99 m fromList [("Alex",31),("Bob",99)]

第36.7节：删除元素

> let m = Map.fromList [("Alex", 31), ("Bob", 99)] fromList [("Alex",31),("Bob",99)] > Map.delete "Bob" m fromList [("Alex",31)]

> Map.notMember "Alex" $ Map.singleton "Alex" 31 False > Map.notMember "Jenny" $ Map.empty True

You can also use `findWithDefault :: Ord k => a -> k -> Map k a -> a` to yield a default value if the key isn't present:

Map.findWithDefault 'x' 1 (fromList [(5,'a'), (3,'b')]) == 'x' Map.findWithDefault 'x' 5 (fromList [(5,'a'), (3,'b')]) == 'a'

Section 36.6: Inserting Elements

Inserting elements is simple:

> let m = Map.singleton "Alex" 31 fromList [("Alex",31)] > Map.insert "Bob" 99 m fromList [("Alex",31),("Bob",99)]

Section 36.7: Deleting Elements

> let m = Map.fromList [("Alex", 31), ("Bob", 99)] fromList [("Alex",31),("Bob",99)] > Map.delete "Bob" m fromList [("Alex",31)]

第37章：结合性声明

声明组件	含义
`infixr`	该运算符是右结合的
`infixl`	该运算符是左结合的
`infix`	该运算符是非结合的
可选数字	运算符的绑定优先级（范围0...9，默认9）
`op1，...，opn`	运算符

第37.1节：结合性

`infixl` 与 `infixr` 与 `infix` 描述了括号将被分组的侧面。例如，考虑以下 fixity 声明（在 base 中）

```
infixl 6 -
infixr 5 :
infix  4 ==
```

infixl 告诉我们 - 是左结合的，这意味着 1 - 2 - 3 - 4 被解析为

```
((1 - 2) - 3) - 4
```

infixr 告诉我们 : 是右结合的，这意味着 1 : 2 : 3 : [] 被解析为

```
1 : (2 : (3 : []))
```

infix 告诉我们 == 不能在没有括号的情况下使用，这意味着 True == False == True 是语法错误。另一方面，`True == (False == True)` 或 `(True == False) == True` 是正确的。

没有显式优先级声明的运算符默认为 infixl 9。

第37.2节：绑定优先级

紧跟结合性信息的数字描述了运算符应用的顺序。它必须始终在 `0` 到 `9` 之间（含）。这通常被称为运算符的结合强度。例如，考虑以下fixity声明（在base中）

```
infixl 6 +
infixl 7 *
```

由于*的结合优先级高于+，我们将1 * 2 + 3 读作

```
(1 * 2) + 3
```

简而言之，数字越大，操作符就越会“拉近”其两侧的括号。

备注

- 函数应用*总是*比运算符绑定得更紧，因此 f x \`op\` g y必须被解释为`(f x)op(g y)`无论运算符\`op\`及其结合性声明如何。
- 如果在结合优先级声明中省略了绑定优先级（例如我们有infixl*!?），默认值是9。

Chapter 37: Fixity declarations

Declaration component	Meaning
`infixr`	the operator is right-associative
`infixl`	the operator is left-associative
`infix`	the operator is non-associative
optional digit	binding precedence of the operator (range 0...9, default 9)
`op1, ... , opn`	operators

Section 37.1: Associativity

`infixl` vs `infixr` vs `infix` describe on which sides the parens will be grouped. For example, consider the following fixity declarations (in base)

```
infixl 6 -
infixr 5 :
infix  4 ==
```

The `infixl` tells us that `-` has left associativity, which means that `1 - 2 - 3 - 4` gets parsed as

```
((1 - 2) - 3) - 4
```

The `infixr` tells us that `:` has right associativity, which means that `1 : 2 : 3 : []` gets parsed as

```
1 : (2 : (3 : []))
```

The `infix` tells us that `==` cannot be used without us including parenthesis, which means that `True == False == True` is a syntax error. On the other hand, `True == (False == True)` or `(True == False) == True` are fine.

Operators without an explicit fixity declaration are `infixl 9`.

Section 37.2: Binding precedence

The number that follows the associativity information describes in what order the operators are applied. It must always be between `0` and `9` inclusive. This is commonly referred to as how tightly the operator binds. For example, consider the following fixity declarations (in base)

```
infixl 6 +
infixl 7 *
```

Since `*` has a higher binding precedence than `+` we read `1 * 2 + 3` as

```
(1 * 2) + 3
```

In short, the higher the number, the closer the operator will "pull" the parens on either side of it.

Remarks

- Function application *always* binds higher than operators, so `f x `op` g y` must be interpreted as `(f x)op(g y)` no matter what the operator \`op\` and its fixity declaration are.
- If the binding precedence is omitted in a fixity declaration (for example we have `infixl *!?`) the default is `9`.

第37.3节：示例声明

- `infixr 5 ++`
- `infixl 4 <*>, <*, *>, <**>`
- `中缀左结合 8 `shift`, `rotate`, `shiftL`, `shiftR`, `rotateL`, `rotateR``
- `中缀 4 ==, /=, <, <=, >=, >`
- `infix ??`

Section 37.3: Example declarations

- `infixr 5 ++`
- `infixl 4 <*>, <*, *>, <**>`
- `infixl 8 `shift`, `rotate`, `shiftL`, `shiftR`, `rotateL`, `rotateR``
- `infix 4 ==, /=, <, <=, >=, >`
- `infix ??`

第38章：网页开发

第38.1节：Servant

Servant 是一个用于在类型层声明API的库，然后：

> - 编写服务器（Servant的这部分可以视为一个网页框架），
> - 获取客户端函数（在Haskell中），
> - 为其他编程语言生成客户端函数，
> - 为你的网页应用生成文档
> - 以及更多...

Servant拥有简洁而强大的API。一个简单的API可以用很少的代码行数编写：

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}

import Data.Text
import Data.Aeson.Types
import GHC.Generics
import Servant.API

data SortBy = Age | Name

data User = User {
name :: 字符串,
age :: 整数
} 派生 (Eq, Show, Generic)

实例 ToJSON 用户  -- 自动将用户转换为 JSON
```

现在我们可以声明我们的 API：

```
类型 用户API = "users" :> 查询参数 "sortby" 排序方式 :> 获取 '[JSON] [用户]
```

这表示我们希望对 /users 进行 GET 请求，带有类型为 排序方式 的查询参数 sortby，并在响应中返回类型为 用户 的 JSON。

现在我们可以定义我们的处理器：

```
-- 这里我们将返回用户数据，或者例如进行数据库查询
服务器 :: 服务器 用户API
服务器 = 返回 [用户 "Alex" 31]

用户API :: 代理 用户API
用户API = 代理

应用1 :: 应用程序
应用1 = 服务 用户API 服务器
```

主方法监听端口8081并提供我们的用户API：

```
main :: IO ()
main = run 8081 app1
```

Chapter 38: Web Development

Section 38.1: Servant

Servant is a library for declaring APIs at the type-level and then:

> - write servers (this part of servant can be considered a web framework),
> - obtain client functions (in haskell),
> - generate client functions for other programming languages,
> - generate documentation for your web applications
> - and more...

Servant has a concise yet powerful API. A simple API can be written in very few lines of code:

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}

import Data.Text
import Data.Aeson.Types
import GHC.Generics
import Servant.API

data SortBy = Age | Name

data User = User {
  name :: String,
  age :: Int
} deriving (Eq, Show, Generic)

instance ToJSON User  -- automatically convert User to JSON
```

Now we can declare our API:

```
type UserAPI = "users" :> QueryParam "sortby" SortBy :> Get '[JSON] [User]
```

which states that we wish to expose `/users` to `GET` requests with a query param `sortby` of type `SortBy` and return JSON of type `User` in the response.

Now we can define our handler:

```
-- This is where we'd return our user data, or e.g. do a database lookup
server :: Server UserAPI
server = return [User "Alex" 31]

userAPI :: Proxy UserAPI
userAPI = Proxy

app1 :: Application
app1 = serve userAPI server
```

And the main method which listens on port `8081` and serves our user API:

```
main :: IO ()
main = run 8081 app1
```

注意，Stack 有一个用于在 Servant 中生成基本 API 的模板，这对于快速启动和运行非常有用。

第38.2节：Yesod

Yesod 项目可以使用 stack new 通过以下模板创建：

- yesod-最小化。最简单的 Yesod 脚手架。
- yesod-mongo。使用MongoDB作为数据库引擎。
- yesod-mysql。使用 MySQL 作为数据库引擎。
- yesod-postgres。使用PostgreSQL作为数据库引擎。
- yesod-postgres-fay。使用PostgreSQL作为数据库引擎。前端使用Fay语言。
- yesod-简单。推荐使用的模板，如果你不需要数据库。
- yesod-sqlite。使用 SQLite 作为数据库引擎。

`yesod-bin` 包提供了 `yesod` 可执行文件，可用于运行开发服务器。注意你也可以直接运行你的应用程序，因此 `yesod` 工具是可选的。

Application.hs 包含在处理程序之间分发请求的代码。如果你使用了数据库和日志功能，也会设置相关配置。

`Foundation.hs` 定义了 `App` 类型，可以看作是所有处理程序的环境。处于 `HandlerT` 单子中，你可以使用 `getYesod` 函数获取该值。

Import.hs 是一个仅重新导出常用内容的模块。

Model.hs 包含生成用于数据库交互的代码和数据类型的 Template Haskell。仅在使用数据库时存在。

config/models 是你定义数据库模式的地方。由 Model.hs 使用。

config/routes 定义了 Web 应用的 URI。对于路由的每个 HTTP 方法，你需要创建一个名为 {method}{RouteR} 的处理程序。

`static/` 目录包含网站的静态资源。这些资源会被 `Settings/StaticFiles.hs`

模块编译进二进制文件中。

templates/ 目录包含用于响应请求的 Shakespeare 模板。

最后，Handler/目录包含定义路由处理程序的模块。

每个处理程序都是基于 IO 的 HandlerT 单子操作。你可以检查请求参数、请求体及其他信息，使用 runDB 进行数据库查询，执行任意 IO 操作，并向用户返回各种类型的内容。要提供 HTML 服务，使用 defaultLayout 函数，该函数允许整洁地组合莎士比亚模板。

Note, Stack has a template for generating basic APIs in Servant, which is useful for getting up and running very quick.

Section 38.2: Yesod

Yesod project can be created with `stack new` using following templates:

- `yesod-minimal`. Simplest Yesod scaffold possible.
- `yesod-mongo`. Uses MongoDB as DB engine.
- `yesod-mysql`. Uses MySQL as DB engine.
- `yesod-postgres`. Uses PostgreSQL as DB engine.
- `yesod-postgres-fay`. Uses PostgreSQL as DB engine. Uses Fay language for front-end.
- `yesod-simple`. Recommended template to use, if you don't need database.
- `yesod-sqlite`. Uses SQlite as DB engine.

`yesod-bin` package provides `yesod` executable, which can be used to run development server. Note that you also can run your application directly, so `yesod` tool is optional.

`Application.hs` contains code that dispatches requests between handlers. It also sets up database and logging settings, if you used them.

`Foundation.hs` defines `App` type, that can be seen as an environment for all handlers. Being in `HandlerT` monad, you can get this value using `getYesod` function.

`Import.hs` is a module that just re-exports commonly used stuff.

`Model.hs` contains Template Haskell that generates code and data types used for DB interaction. Present only if you are using DB.

`config/models` is where you define your DB schema. Used by `Model.hs`.

`config/routes` defines URI's of the Web application. For each HTTP method of the route, you'd need to create a handler named `{method}{RouteR}`.

`static/` directory contains site's static resources. These get compiled into binary by `Settings/StaticFiles.hs` module.

`templates/` directory contains Shakespeare templates that are used when serving requests.

Finally, `Handler/` directory contains modules that define handlers for routes.

Each handler is a `HandlerT` monad action based on IO. You can inspect request parameters, its body and other information, make queries to the DB with `runDB`, perform arbitrary IO and return various types of content to the user. To serve HTML, `defaultLayout` function is used that allows neat composition of shakespearian templates.

第39章：向量

第39.1节：Data.Vector 模块

Data.Vector 模块由 vector 提供，是一个用于处理数组的高性能库。

一旦导入了 Data.Vector，就可以轻松开始使用 Vector：

```
Prelude> import Data.Vector
Prelude Data.Vector> let a = fromList [2,3,4]

Prelude Data.Vector> a
fromList [2,3,4] :: Data.Vector.Vector

Prelude Data.Vector> :t a
a :: Vector Integer
```

你甚至可以拥有多维数组：

```
Prelude Data.Vector> let x = fromList [ fromList [1 .. x] | x <- [1..10] ]

Prelude Data.Vector> :t x
x :: 向量 (向量 整数)
```

第39.2节：过滤向量

过滤奇数元素：

```
Prelude Data.Vector> Data.Vector.filter odd y
fromList [1,3,5,7,9,11] :: Data.Vector.Vector
```

第39.3节：映射（`map`）和归约（`fold`）向量

向量可以被map映射和fold归约，filter过滤以及zip配对：

```
Prelude Data.Vector> Data.Vector.map (^2) y
fromList [0,1,4,9,16,25,36,49,64,81,100,121] :: Data.Vector.Vector
```

归约为单个值：

```
Prelude Data.Vector> Data.Vector.foldl (+) 0 y
66
```

第39.4节：处理多个向量

将两个数组压缩成一个成对数组：

```
Prelude Data.Vector> Data.Vector.zip y y
fromList [(0,0),(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9),(10,10),(11,11)] ::
Data.Vector.Vector
```

Chapter 39: Vectors

Section 39.1: The Data.Vector Module

The Data.Vector module provided by the vector is a high performance library for working with arrays.

Once you've imported Data.Vector, it's easy to start using a Vector:

```
Prelude> import Data.Vector
Prelude Data.Vector> let a = fromList [2,3,4]

Prelude Data.Vector> a
fromList [2,3,4] :: Data.Vector.Vector

Prelude Data.Vector> :t a
a :: Vector Integer
```

You can even have a multi-dimensional array:

```
Prelude Data.Vector> let x = fromList [ fromList [1 .. x] | x <- [1..10] ]

Prelude Data.Vector> :t x
x :: Vector (Vector Integer)
```

Section 39.2: Filtering a Vector

Filter odd elements:

```
Prelude Data.Vector> Data.Vector.filter odd y
fromList [1,3,5,7,9,11] :: Data.Vector.Vector
```

Section 39.3: Mapping (`map`) and Reducing (`fold`) a Vector

Vectors can be **map**'d and fold'd, filter'd **and**zip`'d:

```
Prelude Data.Vector> Data.Vector.map (^2) y
fromList [0,1,4,9,16,25,36,49,64,81,100,121] :: Data.Vector.Vector
```

Reduce to a single value:

```
Prelude Data.Vector> Data.Vector.foldl (+) 0 y
66
```

Section 39.4: Working on Multiple Vectors

Zip two arrays into an array of pairs:

```
Prelude Data.Vector> Data.Vector.zip y y
fromList [(0,0),(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9),(10,10),(11,11)] ::
Data.Vector.Vector
```

第40章：Cabal

第40.1节：使用沙箱

Haskell项目可以使用系统范围内的包，也可以使用沙箱。沙箱是一个隔离的包数据库，可以防止依赖冲突，例如当多个Haskell项目使用同一包的不同版本时。

要为Haskell包初始化沙箱，请进入其目录并运行：

```
cabal sandbox init
```

现在可以通过简单运行cabal install来安装包。

列出沙箱中的包：

```
cabal sandbox hc-pkg 列表
```

删除沙箱：

```
cabal sandbox delete
```

添加本地依赖：

```
cabal sandbox add-source /path/to/dependency
```

第40.2节：安装包

安装新包，例如 aeson：

```
cabal install aeson
```

Chapter 40: Cabal

Section 40.1: Working with sandboxes

A Haskell project can either use the system wide packages or use a sandbox. A sandbox is an isolated package database and can prevent dependency conflicts, e. g. if multiple Haskell projects use different versions of a package.

To initialize a sandbox for a Haskell package go to its directory and run:

```
cabal sandbox init
```

Now packages can be installed by simply running `cabal install`.

Listing packages in a sandbox:

```
cabal sandbox hc-pkg list
```

Deleting a sandbox:

```
cabal sandbox delete
```

Add local dependency:

```
cabal sandbox add-source /path/to/dependency
```

Section 40.2: Install packages

To install a new package, e.g. aeson:

```
cabal install aeson
```

第41章：类型代数

第41.1节：加法和乘法

加法和乘法在这种类型代数中有对应物。它们分别对应于**标记联合**和**乘积类型**。

```haskell
data Sum a b = A a | B b
data Prod a b = Prod a b
```

我们可以看到每种类型的居民数量如何对应于代数的运算。

同样地，我们可以使用Either和(,)作为加法和乘法的类型构造器。它们与我们之前定义的类型是同构的：

```haskell
type Sum' a b = Either a b
type Prod' a b = (a,b)
```

加法和乘法的预期结果遵循类型代数的同构关系。例如，我们可以看到1 + 2、2 + 1和3之间的同构关系；因为1 + 2 = 3 = 2 + 1。

```haskell
data Color = Red | Green | Blue

f :: Sum () Bool -> Color
f (Left ())      = Red
f (Right True)  = Green
f (Right False) = Blue

g :: Color -> Sum () Bool
g Red   = Left ()
g Green = Right True
g Blue  = Right False

f' :: Sum Bool () -> Color
f' (Right ())    = Red
f' (Left True)  = Green
f' (Left False) = Blue

g' :: Color -> Sum Bool ()
g' Red   = Right ()
g' Green = Left True
g' Blue  = Left False
```

加法和乘法的规则

交换律、结合律和分配律的通用规则是有效的，因为以下类型之间存在平凡的同构关系：

```haskell
-- 交换律
Sum a b          <=> Sum b a
Prod a b         <=> Prod b a
-- 结合律
Sum (Sum a b) c   <=> Sum a (Sum b c)
Prod (Prod a b) c <=> Prod a (Prod b c)
-- 分配律
Prod a (Sum b c)  <=> Sum (Prod a b) (Prod a c)
```

Chapter 41: Type algebra

Section 41.1: Addition and multiplication

The addition and multiplication have equivalents in this type algebra. They correspond to the **tagged unions** and **product types**.

```haskell
data Sum a b = A a | B b
data Prod a b = Prod a b
```

We can see how the number of inhabitants of every type corresponds to the operations of the algebra.

Equivalently, we can use `Either` and `(,)` as type constructors for the addition and the multiplication. They are isomorphic to our previously defined types:

```haskell
type Sum' a b = Either a b
type Prod' a b = (a,b)
```

The expected results of addition and multiplication are followed by the type algebra up to isomorphism. For example, we can see an isomorphism between 1 + 2, 2 + 1 and 3; as 1 + 2 = 3 = 2 + 1.

```haskell
data Color = Red | Green | Blue

f :: Sum () Bool -> Color
f (Left ())      = Red
f (Right True)  = Green
f (Right False) = Blue

g :: Color -> Sum () Bool
g Red   = Left ()
g Green = Right True
g Blue  = Right False

f' :: Sum Bool () -> Color
f' (Right ())    = Red
f' (Left True)  = Green
f' (Left False) = Blue

g' :: Color -> Sum Bool ()
g' Red   = Right ()
g' Green = Left True
g' Blue  = Left False
```

Rules of addition and multiplication

The common rules of commutativity, associativity and distributivity are valid because there are trivial isomorphisms between the following types:

```haskell
-- Commutativity
Sum a b           <=> Sum b a
Prod a b          <=> Prod b a
-- Associativity
Sum (Sum a b) c   <=> Sum a (Sum b c)
Prod (Prod a b) c <=> Prod a (Prod b c)
-- Distributivity
Prod a (Sum b c)  <=> Sum (Prod a b) (Prod a c)
```

第41.2节：函数

函数可以看作是我们代数中的指数。如我们所见，如果取一个有n个实例的类型a和一个有m个实例的类型b，则类型a -> b将有m的n次方个实例。

例如，Bool -> Bool同构于(Bool,Bool)，因为2*2 = 2^2。

```
iso1 :: (Bool -> Bool) -> (Bool,Bool)
iso1 f = (f True,f False)

iso2 :: (Bool,Bool) -> (Bool -> Bool)
iso2 (x,y) = (\p -> if p then x else y)
```

第41.3节：类型代数中的自然数

我们可以将Haskell类型与自然数联系起来。这个联系可以通过为每个类型分配其拥有的元素数量来建立。

有限联合类型

对于有限类型，只需看到我们可以根据构造函数的数量为每个数字分配一个自然类型。例如：

```
type Color = Red | Yellow | Green
```

将是3。Bool类型将是2。

```
type Bool = True | False
```

同构意义下的唯一性

我们已经看到多个类型会对应同一个数字，但在这种情况下，它们是同构的。也就是说，会有一对态射f和g，其复合是恒等映射，连接这两种类型。

```
f :: a -> b
g :: b -> a

f . g == id == g . f
```

在这种情况下，我们会说这些类型是同构的。只要它们是同构的，我们将在代数中认为两个类型是相等的。

例如，数字二的两种不同表示显然是同构的：

```
类型 Bit  = I    | O
类型 Bool = True | False

bitValue :: Bit -> Bool
bitValue I = True
bitValue O = False

booleanBit :: Bool -> Bit
booleanBit True  = I
booleanBit False = O
```

Section 41.2: Functions

Functions can be seen as exponentials in our algebra. As we can see, if we take a type a with n instances and a type b with m instances, the type a -> b will have m to the power of n instances.

As an example, Bool -> Bool is isomorphic to (Bool,Bool), as 2*2 = 2^2.

```
iso1 :: (Bool -> Bool) -> (Bool,Bool)
iso1 f = (f True,f False)

iso2 :: (Bool,Bool) -> (Bool -> Bool)
iso2 (x,y) = (\p -> if p then x else y)
```

Section 41.3: Natural numbers in type algebra

We can draw a connection between the Haskell types and the natural numbers. This connection can be made assigning to every type the number of inhabitants it has.

Finite union types

For finite types, it suffices to see that we can assign a natural type to every number, based in the number of constructors. For example:

```
type Color = Red | Yellow | Green
```

would be **3**. And the Bool type would be **2**.

```
type Bool = True | False
```

Uniqueness up to isomorphism

We have seen that multiple types would correspond to a single number, but in this case, they would be isomorphic. This is to say that there would be a pair of morphisms f and g, whose composition would be the identity, connecting the two types.

```
f :: a -> b
g :: b -> a

f . g == id == g . f
```

In this case, we would say that the types are **isomorphic**. We will consider two types equal in our algebra as long as they are isomorphic.

For example, two different representations of the number two are trivally isomorphic:

```
type Bit  = I    | O
type Bool = True | False

bitValue :: Bit -> Bool
bitValue I = True
bitValue O = False

booleanBit :: Bool -> Bit
booleanBit True  = I
booleanBit False = O
```

因为我们可以看到 `bitValue . booleanBit == id == booleanBit . bitValue`

一和零

数字1的表示显然是只有一个构造函数的类型。在Haskell中，这个类型规范地是()类型，称为Unit。每个只有一个构造函数的其他类型都与()同构。

而我们对0的表示将是一个没有构造函数的类型。这就是Haskell中的Void类型，如Data.Void中定义的。这相当于一个无居民类型，没有数据构造函数：

```
数据 Void
```

第41.4节：递归类型

列表

列表可以定义为：

```
data List a = Nil | Cons a (List a)
```

如果我们将其翻译成我们的类型代数，我们得到

> List(a) = 1 + a * List(a)

但我们现在可以在这个表达式中多次替换List(a)，以得到：

> List(a) = 1 + a + a*a + a*a*a + a*a*a*a + ...

如果我们将列表视为一种类型，该类型可以只包含一个值，如[]；或者包含类型 a 的每个值，如[x]；或者包含两个类型 a 的值，如[x,y]；依此类推，这样理解是合理的。从那里我们应该得到的List的理论定义是：

```
-- 无法运行的Haskell代码！
data List a = Nil
            | One a
            | Two a a
            | Three a a a
          ...
```

树

例如，我们也可以对二叉树做同样的事情。如果我们将它们定义为：

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

我们得到表达式：

> Tree(a) = 1 + a * Tree(a) * Tree(a)

如果我们一次又一次地进行相同的替换，就会得到以下序列：

Because we can see `bitValue . booleanBit == id == booleanBit . bitValue`

One and Zero

The representation of the number **1** is obviously a type with only one constructor. In Haskell, this type is canonically the type `()`, called Unit. Every other type with only one constructor is isomorphic to `()`.

And our representation of **0** will be a type without constructors. This is the **Void** type in Haskell, as defined in `Data.Void`. This would be equivalent to a unhabited type, wihtout data constructors:

```
data Void
```

Section 41.4: Recursive types

Lists

Lists can be defined as:

```
data List a = Nil | Cons a (List a)
```

If we translate this into our type algebra, we get

> List(a) = 1 + a * List(a)

But we can now substitute *List(a)* again in this expression multiple times, in order to get:

> List(a) = 1 + a + a*a + a*a*a + a*a*a*a + ...

This makes sense if we see a list as a type that can contain only one value, as in [ ]; or every value of type a, as in [x]; or two values of type a, as in [x,y]; and so on. The theoretical definition of List that we should get from there would be:

```
-- Not working Haskell code!
data List a = Nil
            | One a
            | Two a a
            | Three a a a
            ...
```

Trees

We can do the same thing with binary trees, for example. If we define them as:

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

We get the expression:

> Tree(a) = 1 + a * Tree(a) * Tree(a)

And if we make the same substitutions again and again, we would obtain the following sequence:

> Tree(a) = 1 + a + 2 (a*a) + 5 (a*a*a) + 14 (a*a*a*a) + ...

这里得到的系数对应于卡特兰数序列，第 n 个卡特兰数正好是具有 n 个节点的可能二叉树的数量。

第41.5节：导数

类型的导数是其”单孔上下文”的类型。这是如果我们在每个可能的位置让一个类型变量消失并将结果相加时得到的类型
。

例如，我们可以取三元组类型(a,a,a)，并对其求导，得到

```
data OneHoleContextsOfTriple = (a,a,()) | (a,(),a) | ((),a,a)
```

这与我们通常对导数的定义是一致的，即：

> d/da (a*a*a) = 3*a*a

关于这个主题的更多内容可以阅读这篇文章。

> Tree(a) = 1 + a + 2 (a*a) + 5 (a*a*a) + 14 (a*a*a*a) + ...

The coefficients we get here correspond to the Catalan numbers sequence, and the n-th catalan number is precisely the number of possible binary trees with n nodes.

Section 41.5: Derivatives

The derivative of a type is the type of its type of one-hole contexts. This is the type that we would get if we make a type variable disappear in every possible point and sum the results.

As an example, we can take the triple type `(a,a,a)`, and derive it, obtaining

```
data OneHoleContextsOfTriple = (a,a,()) | (a,(),a) | ((),a,a)
```

This is coherent with our usual definition of derivation, as:

> d/da (a*a*a) = 3*a*a

More on this topic can be read on this article.

第42章：箭头

第42.1节：多通道的函数组合

箭头是，模糊地说，像函数一样组合的态射类，既有串行组合也有“并行组合”。虽然它作为函数的*推广*最为有趣，箭头(->)实例本身已经相当有用。例如，以下函数：

```
spaceAround :: Double -> [Double] -> Double
spaceAround x ys = minimum greater - maximum smaller
 其中 (greater, smaller) = partition (>x) ys
```

也可以用箭头组合子来编写：

```
spaceAround x = partition (>x) >>> minimum *** maximum >>> uncurry (-)
```

这种组合方式可以通过图示来最好地理解：

```
                       ──── minimum ────
                  ╱          *          ╲
──── partition (>x) >>>       *          >>>  uncurry (-) ───
                  ╲          *          ╱
                       ──── maximum ────
```

这里，

- 操作符 >>> 只是普通 . 组合操作符的反转版本（还有一个 <<< 版本是从右向左组合）。它将数据从一个处理步骤传递到下一个。
- 输出的 ╱ ╲ 表示数据流被分成两个“通道”。在 Haskell 类型中，这通过元组实现：

  ```
  partition (>x) :: [Double] -> ([Double], [Double])
  ```

  将流量分成两个[双重]通道，而

  ```
  uncurry (-) :: (Double,Double) -> Double
  ```

  合并两个 Double 通道。

- *** 是并行† 组合运算符。它允许 **maximum** 和 **minimum** 在不同通道的数据上独立操作。对于函数，该运算符的签名是

  ```
  (***) :: (b->c) -> (β->γ) -> (b,β)->(c,γ)
  ```

†至少在 **Hask** 范畴中（即在 Arrow (->) 实例中），f***g 并不真正并行计算 f 和 g，即不在不同线程上执行。不过理论上这是可能的。

Chapter 42: Arrows

Section 42.1: Function compositions with multiple channels

Arrow is, vaguely speaking, the class of morphisms that compose like functions, with both serial composition and “parallel composition”. While it is most interesting as a *generalisation* of functions, the Arrow (->) instance itself is already quite useful. For instance, the following function:

```
spaceAround :: Double -> [Double] -> Double
spaceAround x ys = minimum greater - maximum smaller
 where (greater, smaller) = partition (>x) ys
```

can also be written with arrow combinators:

```
spaceAround x = partition (>x) >>> minimum *** maximum >>> uncurry (-)
```

This kind of composition can best be visualised with a diagram:

```
                       ──── minimum ────
                  ╱          *          ╲
──── partition (>x) >>>       *          >>>  uncurry (-) ───
                  ╲          *          ╱
                       ──── maximum ────
```

Here,

- The >>> operator is just a flipped version of the ordinary . composition operator (there's also a <<< version that composes right-to-left). It pipes the data from one processing step to the next.
- the out-going ╱ ╲ indicate the data flow is split up in two “channels”. In terms of Haskell types, this is realised with tuples:

  ```
  partition (>x) :: [Double] -> ([Double], [Double])
  ```

  splits up the flow in two [Double] channels, whereas

  ```
  uncurry (-) :: (Double,Double) -> Double
  ```

  merges two Double channels.

- *** is the parallel† composition operator. It lets **maximum** and **minimum** operate independently on different channels of the data. For functions, the signature of this operator is

  ```
  (***) :: (b->c) -> (β->γ) -> (b,β)->(c,γ)
  ```

†At least in the **Hask** category (i.e. in the Arrow (->) instance), f***g does not actually compute f and g in parallel as in, on different threads. This would theoretically be possible, though.

第43章：类型洞

第43.1节：类型洞的语法

类型洞是在表达式上下文中，单个下划线（_）或一个未在作用域内的有效Haskell标识符。
在类型洞出现之前，这两种情况都会触发错误，因此新语法不会
干扰任何旧语法。

控制类型洞的行为

带类型的空洞的默认行为是在遇到带类型的空洞时产生编译时错误。然而，有几个标志可以微调它们的行为。这些标志总结如下（GHC trac）：

> 默认情况下，GHC 启用了带类型的空洞，并在遇到带类型的空洞时产生编译错误。
>
> 当启用 -fdefer-type-errors 或 -fdefer-typed-holes 时，空洞错误会被转换为警告，并在运行时求值时导致运行时错误。
>
> 警告标志 -fwarn-typed-holes 默认开启。在没有 -fdefer-type-errors 或 -fdefer-typed-holes 的情况下，该标志无效，因为带类型的空洞在这些条件下是错误。如果启用了任一延迟标志（将带类型空洞错误转换为警告），则 -fno-warn-typed-holes 标志会禁用这些警告。这意味着编译会静默成功，求值空洞时会产生运行时错误。

第43.2节：带类型空洞的语义

类型空洞的值可以简单地说是 undefined，尽管带类型的空洞会触发编译时错误，因此严格来说不必为其赋值。然而，带类型的空洞（当启用时）会产生编译时错误（或带延迟类型错误时的警告），其中会说明带类型空洞的名称、其推断出的最一般类型以及任何局部绑定的类型。例如：

```
Prelude> \x -> _var + length (drop 1 x)

<interactive>:19:7: 警告:
发现空洞 `_var', 类型为: Int
    相关绑定包括
x :: [a] (绑定于 <interactive>:19:2)
      it :: [a] -> Int (绑定于 <interactive>:19:1)
    在 `(+)' 的第一个参数中, 即 `_var'
    在表达式中: _var + length (drop 1 x)
    在表达式中: \ x -> _var + length (drop 1 x)
```

注意，在 GHCi 交互式环境中输入的表达式中出现带类型空洞的情况下（如上所示），还会报告输入表达式的类型，作为 it（此处类型为 [a] -> Int）。

第43.3节：使用类型占位符定义类实例

带类型的空洞可以通过交互式过程使定义函数变得更容易。

假设你想定义一个类实例Foo Bar（针对你的自定义Bar类型，以便将其用于某些需要Foo实例的多态库函数）。传统上你会去查阅Foo的文档，弄清楚需要定义哪些方法，仔细研究它们的类型等等。–但使用带类型的空洞，你实际上可以跳过这些步骤！

Chapter 43: Typed holes

Section 43.1: Syntax of typed holes

A typed hole is a single underscore (_) or a valid Haskell identifier which is not in scope, in an expression context. Before the existance of typed holes, both of these things would trigger an error, so the new syntax does not interfere with any old syntax.

Controlling behaviour of typed holes

The default behaviour of typed holes is to produce a compile-time error when encountering a typed hole. However, there are several flags to fine-tune their behaviour. These flags are summarized as follows (GHC trac):

> By default GHC has typed holes enabled and produces a compile error when it encounters a typed hole.
>
> When `-fdefer-type-errors` **or** `-fdefer-typed-holes` is enabled, hole errors are converted to warnings and result in runtime errors when evaluated.
>
> The warning flag `-fwarn-typed-holes` is on by default. Without `-fdefer-type-errors` or `-fdefer-typed-holes` this flag is a no-op, since typed holes are an error under these conditions. If either of the defer flags are enabled (converting typed hole errors into warnings) the `-fno-warn-typed-holes` flag disables the warnings. This means compilation silently succeeds and evaluating a hole will produce a runtime error.

Section 43.2: Semantics of typed holes

The value of a type hole can simply said to be **`undefined`**, although a typed hole triggers a compile-time error, so it is not strictly necessary to assign it a value. However, a typed hole (when they are enabled) produces a compile time error (or warning with deferred type errors) which states the name of the typed hole, its inferred *most general* type, and the types of any local bindings. For example:

```
Prelude> \x -> _var + length (drop 1 x)

<interactive>:19:7: Warning:
    Found hole `_var' with type: Int
    Relevant bindings include
      x :: [a] (bound at <interactive>:19:2)
      it :: [a] -> Int (bound at <interactive>:19:1)
    In the first argument of `(+)', namely `_var'
    In the expression: _var + length (drop 1 x)
    In the expression: \ x -> _var + length (drop 1 x)
```

Note that in the case of typed holes in expressions entered into the GHCi repl (as above), the type of the expression entered also reported, as `it` (here of type `[a] -> Int`).

Section 43.3: Using typed holes to define a class instance

Typed holes can make it easier to define functions, through an interactive process.

Say you want to define a class instance `Foo Bar` (for your custom `Bar` type, in order to use it with some polymorphic library function that requires a `Foo` instance). You would now traditionally look up the documentation of `Foo`, figure out which methods you need to define, scrutinise their types etc. – but with typed holes, you can actually skip that!

首先定义一个虚拟实例：

```
instance Foo Bar where
```

编译器现在会报错

```
Bar.hs:13:10: 警告:
没有明确实现
   ‘foom’ 和 ‘quun’
在‘Foo Bar’的instance声明中
```

好的，所以我们需要为Bar定义foom。但那到底is什么呢？我们又懒得去查阅文档，直接问编译器：

```
instance Foo Bar where
  foom = _
```

这里我们使用了一个带类型的空洞作为一个简单的“文档查询”。编译器输出了

```
Bar.hs:14:10:
发现空洞‘_’，类型为: Bar -> Gronk Bar
    相关绑定包括
foom :: Bar -> Gronk Bar (绑定于 Foo.hs:4:28)
    表达式中: _
在‘foom’的等式中: foom = _
    在‘Foo Bar’的instance声明中
```

注意编译器已经用我们想要实例化的具体类型Bar填充了类的类型变量。这使得签名比类文档中找到的多态签名更容易理解，尤其是当你处理更复杂的方法时，比如多参数类型类的方法。

但是Gronk到底是什么？此时，向Hayoo求助可能是个好主意。不过我们或许还能不借助它：盲猜一下，我们假设这不仅是一个类型构造器，还是唯一的值构造器，也就是说它可以作为一个函数使用，某种方式产生一个Gronk a值。所以我们尝试

```
instance Foo Bar where
  foom bar = _ Gronk
```

如果我们幸运的话，Gronk 实际上是一个值，编译器现在会说

```
发现空洞‘_’
类型为: (Int -> [(Int, b0)] -> Gronk b0) -> Gronk Bar
其中: ‘b0’是一个模糊的类型变量
```

好的，这很难看–首先注意Gronk有两个参数，所以我们可以细化我们的尝试：

```
实例 Foo Bar 其中
  foom bar = Gronk _ _
```

现在这很清楚了：

```
发现空洞‘_’类型为: [(Int, Bar)]
相关绑定包括
bar :: Bar  (绑定于 Bar.hs:14:29)
```

First just define a dummy instance:

```
instance Foo Bar where
```

The compiler will now complain

```
Bar.hs:13:10: Warning:
No explicit implementation for
  ‘foom’ and ‘quun’
In the instance declaration for ‘Foo Bar’
```

Ok, so we need to define foom for Bar. But what *is* that even supposed to be? Again we're too lazy to look in the documentation, and just ask the compiler:

```
instance Foo Bar where
  foom = _
```

Here we've used a typed hole as a simple “documentation query”. The compiler outputs

```
Bar.hs:14:10:
    Found hole ‘_’ with type: Bar -> Gronk Bar
    Relevant bindings include
      foom :: Bar -> Gronk Bar (bound at Foo.hs:4:28)
    In the expression: _
    In an equation for ‘foom’: foom = _
    In the instance declaration for ‘Foo Bar’
```

Note how the compiler has already filled the class type variable with the concrete type Bar that we want to instantiate it for. This can make the signature a lot easier to understand than the polymorphic one found in the class documentation, especially if you're dealing with a more complicated method of e.g. a multi-parameter type class.

But what the hell is Gronk? At this point, it is probably a good idea to ask Hayoo. However we may still get away without that: as a blind guess, we assume that this is not only a type constructor but also the single value constructor, i.e. it can be used as a function that will somehow produce a Gronk a value. So we try

```
instance Foo Bar where
  foom bar = _ Gronk
```

If we're lucky, Gronk is actually a value, and the compiler will now say

```
    Found hole ‘_’
      with type: (Int -> [(Int, b0)] -> Gronk b0) -> Gronk Bar
    Where: ‘b0’ is an ambiguous type variable
```

Ok, that's ugly – at first just note that Gronk has two arguments, so we can refine our attempt:

```
instance Foo Bar where
  foom bar = Gronk _ _
```

And this now is pretty clear:

```
    Found hole ‘_’ with type: [(Int, Bar)]
    Relevant bindings include
      bar :: Bar (bound at Bar.hs:14:29)
```

```
foom :: Bar -> Gronk Bar （绑定于 Foo.hs:15:24）
在‘Gronk’的第二个参数中，即‘_’
表达式中：Gronk _ _
在‘foom’的等式中: foom bar = Gronk _ _
```

你现在可以进一步推进，例如分解bar值（组件及其类型将显示在相关绑定部分）。通常，在某个时刻，正确的定义是完全显而易见的，因为你看到了所有可用的参数，且类型像拼图一样匹配。或者，你可能会发现定义是不可能的，并且知道原因。

所有这些在带有交互式编译的编辑器中效果最佳，例如带有 haskell-mode 的 Emacs。你可以像在解释型动态命令式语言的 IDE 中使用鼠标悬停查询值一样使用类型空洞，但没有那些限制。

```
      foom :: Bar -> Gronk Bar (bound at Foo.hs:15:24)
    In the second argument of ‘Gronk’, namely ‘_’
    In the expression: Gronk _ _
    In an equation for ‘foom’: foom bar = Gronk _ _
```

You can now further progress by e.g. deconstructing the `bar` value (the components will then show up, with types, in the `Relevant bindings` section). Often, it is at some point completely obvious what the correct definition will be, because you you see all avaliable arguments and the types fit together like a jigsaw puzzle. Or alternatively, you may see that the definition is *impossible* and why.

All of this works best in an editor with interactive compilation, e.g. Emacs with haskell-mode. You can then use typed holes much like mouse-over value queries in an IDE for an interpreted dynamic imperative language, but without all the limitations.

第44章：重写规则（GHC）

第44.1节：在重载函数上使用重写规则

在这个问题中，@Viclib 询问了如何使用重写规则利用类型类定律来消除一些重载函数调用：

> 请注意以下类：
>
> ```
> class ListIsomorphic l where
>     toList    :: l a -> [a]
>     fromList  :: [a] -> l a
> ```
>
> 我还要求toList . fromList == id。如何编写重写规则告诉 GHC 进行该替换？

这是 GHC 重写规则机制的一个有些棘手的用例，因为重载函数会被重写为它们的具体实例方法，这些规则由 GHC 在幕后隐式创建（所以类似
fromList :: Seq a -> [a]会被重写为Seq$fromList等）。

但是，通过首先将toList和fromList重写为非内联的非类型类方法，我们可以防止它们被过早重写，并保持它们直到组合规则可以触发：

```
{-# RULES
"protect toList"    toList = toList';
  "protect fromList" fromList = fromList';
  "fromList/toList"   forall x . fromList' (toList' x) = x; #-}

{-# NOINLINE [0] fromList' #-}
fromList' :: (ListIsomorphic l) => [a] -> l a
fromList' = fromList

{-# NOINLINE [0] toList' #-}
toList' :: (ListIsomorphic l) => l a -> [a]
toList' = toList
```

Chapter 44: Rewrite rules (GHC)

Section 44.1: Using rewrite rules on overloaded functions

In this question, @Viclib asked about using rewrite rules to exploit typeclass laws to eliminate some overloaded function calls:

> Mind the following class:
>
> ```
> class ListIsomorphic l where
>     toList    :: l a -> [a]
>     fromList  :: [a] -> l a
> ```
>
> I also demand that `toList . fromList == id`. How do I write rewrite rules to tell GHC to make that substitution?

This is a somewhat tricky use case for GHC's rewrite rules mechanism, because overloaded functions are rewritten into their specific instance methods by rules that are implicitly created behind the scenes by GHC (so something like `fromList :: Seq a -> [a]` would be rewritten into `Seq$fromList` etc.).

However, by first rewriting `toList` and `fromList` into non-inlined non-typeclass methods, we can protect them from premature rewriting, and preserve them until the rule for the composition can fire:

```
{-# RULES
  "protect toList"   toList = toList';
  "protect fromList" fromList = fromList';
  "fromList/toList"  forall x . fromList' (toList' x) = x; #-}

{-# NOINLINE [0] fromList' #-}
fromList' :: (ListIsomorphic l) => [a] -> l a
fromList' = fromList

{-# NOINLINE [0] toList' #-}
toList' :: (ListIsomorphic l) => l a -> [a]
toList' = toList
```

第45章：日期和时间

第45.1节：查找今天的日期

当前日期和时间可以通过getCurrentTime获取：

```
import Data.Time

print =<< getCurrentTime
-- 2016-08-02 12:05:08.937169 UTC
```

或者，仅返回日期，使用fromGregorian：

```
fromGregorian 1984 11 17  -- 返回一个日期（Day）
```

第45.2节：日期的加减与比较

给定一个日期（Day），我们可以进行简单的算术运算和比较，例如加法：

```
import Data.Time

addDays 1 (fromGregorian 2000 1 1)
-- 2000-01-02
addDays 1 (fromGregorian 2000 12 31)
-- 2001-01-01
```

减法：

```
addDays (-1) (fromGregorian 2000 1 1)
-- 1999-12-31

addDays (-1) (fromGregorian 0 1 1)
-- -0001-12-31
-- wat
```

甚至找出差异：

```
diffDays (fromGregorian 2000 12 31) (fromGregorian 2000 1 1)
365
```

注意顺序很重要：

```
diffDays (fromGregorian 2000 1 1) (fromGregorian 2000 12 31)
-365
```

Chapter 45: Date and Time

Section 45.1: Finding Today's Date

Current date and time can be found with `getCurrentTime`:

```
import Data.Time

print =<< getCurrentTime
-- 2016-08-02 12:05:08.937169 UTC
```

Alternatively, just the date is returned by `fromGregorian`:

```
fromGregorian 1984 11 17  -- yields a Day
```

Section 45.2: Adding, Subtracting and Comparing Days

Given a `Day`, we can perform simple arithmetic and comparisons, such as adding:

```
import Data.Time

addDays 1 (fromGregorian 2000 1 1)
-- 2000-01-02
addDays 1 (fromGregorian 2000 12 31)
-- 2001-01-01
```

Subtract:

```
addDays (-1) (fromGregorian 2000 1 1)
-- 1999-12-31

addDays (-1) (fromGregorian 0 1 1)
-- -0001-12-31
-- wat
```

and even find the difference:

```
diffDays (fromGregorian 2000 12 31) (fromGregorian 2000 1 1)
365
```

note that the order matters:

```
diffDays (fromGregorian 2000 1 1) (fromGregorian 2000 12 31)
-365
```

第46章：列表推导式

第46.1节：基本列表推导式

Haskell 有列表推导式，这与数学中的集合推导式以及命令式语言如 Python 和 JavaScript 中的类似实现非常相似。最基本的列表推导式形式如下。

```
[ x | x <- someList ]
```

例如

```
[ x | x <- [1..4] ]    -- [1,2,3,4]
```

函数也可以直接应用于 x：

```
[ f x | x <- someList ]
```

这等价于：

```
map f someList
```

示例：

```
[ x+1 | x <- [1..4]]    -- [2,3,4,5]
```

第46.2节：Do 语法

任何列表推导式都可以用列表单子（list monad）的do表示法相应地编码。

```
[f x | x <- xs]                 f  <$> xs         do { x <- xs ; return (f x) }

[f x | f <- fs, x <- xs]        fs <*> xs         do { f <- fs ; x <- xs ; return (f x) }

[y   | x <- xs, y <- f x]       f  =<< xs         do { x <- xs ; y <- f x ; return y }
```

守卫可以使用Control.Monad.guard来处理：

```
[x   | x <- xs, even x]                         do { x <- xs ; guard (even x) ; return x }
```

第46.3节：生成器表达式中的模式

然而，生成器表达式中的 x 不仅仅是变量，也可以是任意模式。在模式不匹配的情况下，生成的元素会被跳过，列表的处理继续进行下一个元素，因此起到了过滤器的作用：

```
[x | Just x <- [Just 1, Nothing, Just 3]]     -- [1, 3]
```

带有变量 x 的生成器模式会创建一个新的作用域，包含其右侧的所有表达式，其中 x 被定义为生成的元素。

这意味着守卫可以编码为

Chapter 46: List Comprehensions

Section 46.1: Basic List Comprehensions

Haskell has list comprehensions, which are a lot like set comprehensions in math and similar implementations in imperative languages such as Python and JavaScript. At their most basic, list comprehensions take the following form.

```
[ x | x <- someList ]
```

For example

```
[ x | x <- [1..4] ]    -- [1,2,3,4]
```

Functions can be directly applied to x as well:

```
[ f x | x <- someList ]
```

This is equivalent to:

```
map f someList
```

Example:

```
[ x+1 | x <- [1..4]]    -- [2,3,4,5]
```

Section 46.2: Do Notation

Any list comprehension can be correspondingly coded with list monad's do notation.

```
[f x | x <- xs]                 f  <$> xs         do { x <- xs ; return (f x) }

[f x | f <- fs, x <- xs]        fs <*> xs         do { f <- fs ; x <- xs ; return (f x) }

[y   | x <- xs, y <- f x]       f  =<< xs         do { x <- xs ; y <- f x ; return y }
```

The guards can be handled using `Control.Monad.guard`:

```
[x   | x <- xs, even x]                         do { x <- xs ; guard (even x) ; return x }
```

Section 46.3: Patterns in Generator Expressions

However, x in the generator expression is not just variable, but can be any pattern. In cases of pattern mismatch the generated element is skipped over, and processing of the list continues with the next element, thus acting like a filter:

```
[x | Just x <- [Just 1, Nothing, Just 3]]     -- [1, 3]
```

A generator with a variable x in its pattern creates new scope containing all the expressions on its right, where x is defined to be the generated element.

This means that guards can be coded as

```
[ x | x <- [1..4], even x] ==
[ x | x <- [1..4], () <- [() | even x]] ==
[ x | x <- [1..4], () <- if even x then [()] else []]
```

第46.4节：守卫

列表推导式的另一个特性是守卫（guards），它们也充当过滤器。守卫是布尔表达式，
出现在列表推导式中竖线的右侧。

它们最基本的用法是

```
[x    | p x]   ===   if p x then [x] else []
```

守卫中使用的任何变量必须出现在推导式左侧，或者在作用域内。因此，

```
[ f x | x <- list, pred1 x y, pred2 x]     -- `y` 必须在外部作用域中定义
```

这等价于

```
map f (filter pred2 (filter (\x -> pred1 x y) list))          -- 或者，

-- ($ list) (filter (`pred1` y) >>> filter pred2 >>> map f)

-- list >>= (\x-> [x | pred1 x y]) >>= (\x-> [x | pred2 x]) >>= (\x -> [f x])
```

(操作符 >>= 是 infixl 1，即它向左结合（左括号优先）。示例：

```
[ x       | x <- [1..4], even x]           -- [2,4]

[ x^2 + 1 | x <- [1..100], even x ]        -- map (\x -> x^2 + 1) (filter even [1..100])
```

第46.5节：并行推导式

使用并行列表推导式语言扩展时，

```
[(x,y) | x <- xs | y <- ys]
```

等价于

```
zip xs ys
```

示例：

```
[(x,y) | x <- [1,2,3] | y <- [10,20]]

-- [(1,10),(2,20)]
```

第46.6节：局部绑定

列表推导式可以引入局部绑定变量以保存一些中间值：

```
[(x,y) | x <- [1..4], let y=x*x+1, even y]    -- [(1,2),(3,10)]
```

同样的效果也可以用一个技巧实现，

```
[ x | x <- [1..4], even x] ==
[ x | x <- [1..4], () <- [() | even x]] ==
[ x | x <- [1..4], () <- if even x then [()] else []]
```

Section 46.4: Guards

Another feature of list comprehensions is guards, which also act as filters. Guards are Boolean expressions and appear on the right side of the bar in a list comprehension.

Their most basic use is

```
[x    | p x]   ===   if p x then [x] else []
```

Any variable used in a guard must appear on its left in the comprehension, or otherwise be in scope. So,

```
[ f x | x <- list, pred1 x y, pred2 x]     -- `y` must be defined in outer scope
```

which is equivalent to

```
map f (filter pred2 (filter (\x -> pred1 x y) list))          -- or,

-- ($ list) (filter (`pred1` y) >>> filter pred2 >>> map f)

-- list >>= (\x-> [x | pred1 x y]) >>= (\x-> [x | pred2 x]) >>= (\x -> [f x])
```

(the >>= operator is infixl 1, i.e. it associates (is parenthesized) to the left). Examples:

```
[ x       | x <- [1..4], even x]           -- [2,4]

[ x^2 + 1 | x <- [1..100], even x ]        -- map (\x -> x^2 + 1) (filter even [1..100])
```

Section 46.5: Parallel Comprehensions

With Parallel List Comprehensions language extension,

```
[(x,y) | x <- xs | y <- ys]
```

is equivalent to

```
zip xs ys
```

Example:

```
[(x,y) | x <- [1,2,3] | y <- [10,20]]

-- [(1,10),(2,20)]
```

Section 46.6: Local Bindings

List comprehensions can introduce local bindings for variables to hold some interim values:

```
[(x,y) | x <- [1..4], let y=x*x+1, even y]    -- [(1,2),(3,10)]
```

Same effect can be achieved with a trick,

```
[(x,y) | x <- [1..4], y <- [x*x+1], even y]    -- [(1,2),(3,10)]
```

列表推导中的let是递归的，和往常一样。但生成器绑定不是递归的，这使得变量遮蔽成为可能：

```
[x | x <- [1..4], x <- [x*x+1], even x]        -- [2,10]
```

第46.7节：嵌套生成器

列表推导也可以从多个列表中提取元素，在这种情况下，结果将是两个元素的所有可能组合的列表，就好像这两个列表是以嵌套的方式处理一样。例如，

```
[ (a,b) | a <- [1,2,3], b <- ['a','b'] ]

-- [(1,'a'), (1,'b'), (2,'a'), (2,'b'), (3,'a'), (3,'b')]
```

```
[(x,y) | x <- [1..4], y <- [x*x+1], even y]   -- [(1,2),(3,10)]
```

The `let` in list comprehensions is recursive, as usual. But generator bindings are not, which enables *shadowing*:

```
[x | x <- [1..4], x <- [x*x+1], even x]       -- [2,10]
```

Section 46.7: Nested Generators

List comprehensions can also draw elements from multiple lists, in which case the result will be the list of every possible combination of the two elements, as if the two lists were processed in the *nested* fashion. For example,

```
[ (a,b) | a <- [1,2,3], b <- ['a','b'] ]

-- [(1,'a'), (1,'b'), (2,'a'), (2,'b'), (3,'a'), (3,'b')]
```

第47章：流式输入输出

第47.1节：流式输入输出

io-streams是一个基于流的库，专注于IO的流抽象。它暴露了两种类型：

- InputStream：只读智能句柄
- OutputStream：只写智能句柄

我们可以使用makeInputStream :: IO (Maybe a) -> IO (InputStream a)来创建一个流。从流中读取使用read :: InputStream a -> IO (Maybe a)，其中Nothing表示文件结束符（EOF）：

```
import Control.Monad (forever)
import qualified System.IO.Streams as S
import System.Random (randomRIO)

main :: IO ()
main = do
是 <- S.makeInputStream $ randomInt  -- 创建一个 InputStream
  forever $ printStream =<< S.read 是  -- 永远从该流读取
  return ()

randomInt :: IO (Maybe Int)
randomInt = do
r <- randomRIO (1, 100)
  return $ Just r

printStream :: Maybe Int -> IO ()
printStream Nothing  = print "没东西!"
printStream (Just a) = putStrLn $ show a
```

Chapter 47: Streaming IO

Section 47.1: Streaming IO

io-streams is Stream-based library that focuses on the Stream abstraction but for IO. It exposes two types:

- InputStream: a read-only smart handle
- OutputStream: a write-only smart handle

We can create a stream with makeInputStream :: IO (Maybe a) -> IO (InputStream a). Reading from a stream is performed using read :: InputStream a -> IO (Maybe a), where Nothing denotes an EOF:

```
import Control.Monad (forever)
import qualified System.IO.Streams as S
import System.Random (randomRIO)

main :: IO ()
main = do
  is <- S.makeInputStream $ randomInt  -- create an InputStream
  forever $ printStream =<< S.read is  -- forever read from that stream
  return ()

randomInt :: IO (Maybe Int)
randomInt = do
  r <- randomRIO (1, 100)
  return $ Just r

printStream :: Maybe Int -> IO ()
printStream Nothing  = print "Nada!"
printStream (Just a) = putStrLn $ show a
```

第48章：谷歌协议缓冲区（Google Protocol Buffers）

第48.1节：创建、构建和使用一个简单的 .proto 文件

让我们先创建一个简单的 .proto 文件 person.proto

```
包 Protocol;

消息 Person {
required string firstName = 1;
    required string lastName  = 2;
    optional int32  age       = 3;
}
```

保存后，我们现在可以通过运行以下命令来创建可用于我们项目的 Haskell 文件

```
$HOME/.local/bin/hprotoc --proto_path=. --haskell_out=. person.proto
```

我们应该会得到类似如下的输出：

```
加载文件路径: "/<path-to-project>/person.proto"
所有 proto 文件已加载
Haskell 名称重整完成
递归模块已解析
./Protocol/Person.hs
./Protocol.hs
处理完成，祝您有美好的一天。
```

`hprotoc` 会在当前目录下创建一个新文件夹 `Protocol`，里面包含 `Person.hs`，我们可以直接将其导入到我们的 Haskell 项目中：

```
import Protocol (Person)
```

下一步，如果使用堆栈，请添加

```
protocol-缓冲区
 , protocol-缓冲区-描述符
```

构建-依赖: 和

```
协议
```

在你的 .cabal 文件中暴露-模块。

如果我们现在从流中收到一个传入消息，该消息的类型将是 ByteString。

为了将 `ByteString`（显然应包含编码的"Person"数据）转换为我们的 Haskell 数据类型，我们需要调用函数 `messageGet`，该函数通过以下方式导入

```
import Text.ProtocolBuffers (messageGet)
```

该函数使得可以使用以下方式创建类型为 Person 的值：

```
transformRawPerson :: ByteString -> Maybe Person
transformRawPerson raw = case messageGet raw of
```

Chapter 48: Google Protocol Buffers

Section 48.1: Creating, building and using a simple .proto file

Let us first create a simple `.proto` file `person.proto`

```
package Protocol;

message Person {
    required string firstName = 1;
    required string lastName  = 2;
    optional int32  age       = 3;
}
```

After saving we can now create the Haskell files which we can use in our project by running

```
$HOME/.local/bin/hprotoc --proto_path=. --haskell_out=. person.proto
```

We should get an output similar to this:

```
Loading filepath: "/<path-to-project>/person.proto"
All proto files loaded
Haskell name mangling done
Recursive modules resolved
./Protocol/Person.hs
./Protocol.hs
Processing complete, have a nice day.
```

`hprotoc` will create a new folder `Protocol` in the current directory with `Person.hs` which we can simply import into our haskell project:

```
import Protocol (Person)
```

As a next step, if using Stack add

```
   protocol-buffers
 , protocol-buffers-descriptor
```

to `build-depends:` and

```
Protocol
```

to `exposed-modules` in your `.cabal` file.

If we get now a incoming message from a stream, the message will have the type `ByteString`.

In order to transform the `ByteString` (which obviously should contain encoded "Person" data) into our Haskell data type, we need to call the function `messageGet` which we import by

```
import Text.ProtocolBuffers (messageGet)
```

which enables to create a value of type `Person` using:

```
transformRawPerson :: ByteString -> Maybe Person
transformRawPerson raw = case messageGet raw of
```

```
    左侧  _          -> 无
    右侧（person, _） -> 仅 person
```

```
    Left   _            -> Nothing
    Right (person, _)  -> Just person
```

第49章：模板Haskell与准引号

第49.1节：模板Haskell和准引号的语法

Template Haskell 通过 -XTemplateHaskell GHC 扩展启用。该扩展启用了本节进一步详细介绍的所有语法特性。有关 Template Haskell 的完整详细信息，请参阅 user guide。

接头

- 拼接是由模板Haskell支持的一种新的语法实体，写作$(...)，其中(...)是某个表达式。
- 在$和表达式的第一个字符之间不得有空格；并且模板Haskell会覆盖$操作符的解析——例如，f$g通常被解析为($) f g，而启用模板Haskell时，它被解析为一个拼接。
- 当拼接出现在顶层时，$ 可以省略。在这种情况下，拼接的表达式是整行内容。
- 拼接表示在编译时运行的代码，用于生成 Haskell 抽象语法树（AST），该 AST 会被编译为 Haskell 代码并插入到程序中
- 拼接可以出现在以下位置：表达式、模式、类型和顶层声明。每种情况下拼接表达式的类型分别是Q Exp、Q Pat、Q Type、Q [Decl]。请注意，声明拼接只能出现在顶层，而其他拼接则可以分别出现在其他表达式、模式或类型内部。

表达式引用（注：不是准引用）

- 表达式引用是一种新的语法实体，写作以下形式之一：
  - `[e|..|]` 或 `[|..|]` - `..` 是一个表达式，引用的类型为 `Q Exp`；
  - `[p|..|]` - `..` 是一个模式，引用的类型为 `Q Pat`；
  - `[t|..|]` - `..` 是一个类型，引用的类型为 `Q Type`；
  - `[d|..|]` - `..` 是一组声明，引用的类型为 `Q [Dec]`。
- 表达式引用接受一个编译时程序并生成该程序所表示的抽象语法树（AST）。
- 在引用中使用一个值（例如 `\x -> [| x |]`）而不带拼接，对应于语法糖 `\x -> [| $(lift x) |]`，其中 `lift :: Lift t => t -> Q Exp` 来自于类

class Lift t where lift :: t -> Q Exp default lift :: Data t => t -> Q Exp 类型化拼接和引用类

- 型化拼接类似于之前提到的（未类型化）拼接，写作 $$(..)，其中 (..) 是一个表达式。
- 如果 e 的类型是 Q (TExp a)，则 $$e 的类型是 a。
- 类型化引用的形式为 [||..||]，其中 .. 是类型为 a 的表达式；生成的引用类型为 Q (TExp a)。
- 类型化表达式可以转换为未类型化表达式：unType :: TExp a -> Exp。

准引号（QuasiQuotes）

Chapter 49: Template Haskell & QuasiQuotes

Section 49.1: Syntax of Template Haskell and Quasiquotes

Template Haskell is enabled by the `-XTemplateHaskell` GHC extension. This extension enables all the syntactic features further detailed in this section. The full details on Template Haskell are given by the user guide.

Splices

- A splice is a new syntactic entity enabled by Template Haskell, written as `$(...)`, where `(...)` is some expression.
- There must not be a space between $ and the first character of the expression; and Template Haskell overrides the parsing of the $ operator - e.g. `f$g` is normally parsed as `($) f g` whereas with Template Haskell enabled, it is parsed as a splice.
- When a splice appears at the top level, the $ may be omitted. In this case, the spliced expression is the entire line.
- A splice represents code which is run at compile time to produce a Haskell AST, and that AST is compiled as Haskell code and inserted into the program
- Splices can appear in place of: expressions, patterns, types, and top-level declarations. The type of the spliced expression, in each case respectively, is `Q Exp`, `Q Pat`, `Q Type`, `Q [Decl]`. Note that declaration splices may *only* appear at the top level, whereas the others may be inside other expressions, patterns, or types, respectively.

Expression quotations (note: *not* a QuasiQuotation)

- An expression quotation is a new syntactic entity written as one of:
  - `[e|..|]` or `[|..|]` - `..` is an expression and the quotation has type `Q Exp`;
  - `[p|..|]` - `..` is a pattern and the quotation has type `Q Pat`;
  - `[t|..|]` - `..` is a type and the quotation has type `Q Type`;
  - `[d|..|]` - `..` is a list of declarations and the quotation has type `Q [Dec]`.
- An expression quotation takes a compile time program and produces the AST represented by that program.
- The use of a value in a quotation (e.g. `\x -> [| x |]`) without a splice corresponds to syntactic sugar for `\x -> [| $(lift x) |]`, where `lift :: Lift t => t -> Q Exp` comes from the class

class Lift t where lift :: t -> Q Exp default lift :: Data t => t -> Q Exp **Typed splices and quotations**

- Typed splices are similair to previously mentioned (untyped) splices, and are written as `$$(..)` where `(..)` is an expression.
- If `e` has type `Q (TExp a)` then `$$e` has type `a`.
- Typed quotations take the form `[||..||]` where `..` is an expression of type `a`; the resulting quotation has type `Q (TExp a)`.
- Typed expression can be converted to untyped ones: `unType :: TExp a -> Exp`.

QuasiQuotes

- QuasiQuotes 泛化了表达式引用——之前，表达式引用所使用的解析器是固定集合中的一个（e,p,t,d），但 QuasiQuotes 允许定义并使用自定义解析器，以在编译时生成代码。准引用可以出现在与普通引用相同的所有上下文中。
- 准引用的写法为 [iden|...|]，其中 iden 是类型为 Language.Haskell.TH.Quote.QuasiQuoter 的标识符。
- 一个 QuasiQuoter 仅由四个解析器组成，分别对应引用可能出现的四种不同上下文：

data QuasiQuoter = QuasiQuoter { quoteExp :: String -> Q Exp, quotePat :: String -> Q Pat, quoteType :: String -> Q Type, quoteDec :: String -> Q [Dec] } **Names**

- Haskell 标识符由类型 Language.Haskell.TH.Syntax.Name 表示。名称构成了 Template Haskell 中表示 Haskell 程序的抽象语法树的叶子节点。
- 当前作用域中的标识符可以通过 'e 或 'T 转换为名称。在第一种情况下，e 被解释为表达式作用域，而第二种情况下 T 是类型作用域（回想一下，类型和数值构造器在 Haskell 中可以共享名称而不会产生歧义）。

第 49.2 节：Q 类型

在 Language.Haskell.TH.Syntax 中定义的 Q :: * -> * 类型构造器是一个抽象类型，表示可以访问运行该计算的模块的编译时环境的计算。
Q 类型还处理变量替换，TH 称之为 *name capture*（并在 here 中讨论）。所有拼接的类型均为 Q X，X 为某个类型。

编译时环境包括：

- 范围内的标识符及其相关信息，
  - 函数类型
  - 构造函数的类型和源数据类型
  - 类型声明的完整规范（类，类型族）
- 拼接发生的源代码位置（行，列，模块，包）
- 函数的优先级（GHC 7.10）
- 启用的 GHC 扩展（GHC 8.0）

类型 Q 还具有生成新名称的能力，使用函数 newName :: String -> Q Name。注意名称不会隐式绑定到任何地方，因此用户必须自行绑定，并且确保名称的最终使用范围正确是用户的责任。

Q 具有 Functor、Monad、Applicative 的实例，这是操作 Q 值的主要接口，配合 Language.Haskell.TH.Lib 中提供的组合子使用，该库为 TH 抽象语法树的每个构造函数定义了辅助函数，形式如下：

```
LitE :: Lit -> Exp
litE :: Lit -> ExpQ

AppE :: Exp -> Exp -> Exp
appE :: ExpQ -> ExpQ -> ExpQ
```

注意 ExpQ、TypeQ、DecsQ 和 PatQ 是通常存储在 Q 类型中的 AST 类型的同义词。

TH 库提供了函数 runQ :: Quasi m => Q a -> m a，并且存在实例 Quasi IO，因此看起来

- QuasiQuotes generalize expression quotations - previously, the parser used by the expression quotation is one of a fixed set (e,p,t,d), but QuasiQuotes allow a custom parser to be defined and used to produce code at compile time. Quasi-quotations can appear in all the same contexts as regular quotations.
- A quasi-quotation is written as [iden|...|], where iden is an identifier of type Language.Haskell.TH.Quote.QuasiQuoter.
- A QuasiQuoter is simply composed of four parsers, one for each of the different contexts in which quotations can appear:

data QuasiQuoter = QuasiQuoter { quoteExp :: String -> Q Exp, quotePat :: String -> Q Pat, quoteType :: String -> Q Type, quoteDec :: String -> Q [Dec] } **Names**

- Haskell identifiers are represented by the type Language.Haskell.TH.Syntax.Name. Names form the leaves of abstract syntax trees representing Haskell programs in Template Haskell.
- An identifier which is currently in scope may be turned into a name with either: 'e or 'T. In the first case, e is interpreted in the expression scope, while in the second case T is in the type scope (recalling that types and value constructors may share the name without amiguity in Haskell).

Section 49.2: The Q type

The Q :: * -> * type constructor defined in Language.Haskell.TH.Syntax is an abstract type representing computations which have access to the compile-time environment of the module in which the computation is run. The Q type also handles variable substituion, called *name capture* by TH (and discussed here.) All splices have type Q X for some X.

The compile-time environment includes:

- in-scope identifiers and information about said identifiers,
  - types of functions
  - types and source data types of constructors
  - full specification of type declarations (classes, type families)
- the location in the source code (line, column, module, package) where the splice occurs
- fixities of functions (GHC 7.10)
- enabled GHC extensions (GHC 8.0)

The Q type also has the ability to generate fresh names, with the function newName :: String -> Q Name. Note that the name is not bound anywhere implicitly, so the user must bind it themselves, and so making sure the resulting use of the name is well-scoped is the responsibility of the user.

Q has instances for Functor, Monad, Applicative and this is the main interface for manipulating Q values, along with the combinators provided in Language.Haskell.TH.Lib, which define a helper function for every constructor of the TH ast of the form:

```
LitE :: Lit -> Exp
litE :: Lit -> ExpQ

AppE :: Exp -> Exp -> Exp
appE :: ExpQ -> ExpQ -> ExpQ
```

Note that ExpQ, TypeQ, DecsQ and PatQ are synonyms for the AST types which are typically stored inside the Q type.

The TH library provides a function runQ :: Quasi m => Q a -> m a, and there is an instance Quasi IO, so it would

Q 类型似乎只是一个高级的 IO。然而，使用 runQ :: Q a -> IO a 会产生一个 IO 操作，该操作无法访问任何编译时环境——这仅在实际的 Q 类型中可用。如果尝试访问该环境，这些 IO 操作将在运行时失败。

第 49.3 节：n 元柯里化

熟悉的

```
curry :: ((a,b) -> c) -> a -> b -> c
curry = \f a b -> f (a,b)
```

该函数可以推广到任意元组的元数，例如：

```
curry3 :: ((a, b, c) -> d) -> a -> b -> c -> d
curry4 :: ((a, b, c, d) -> e) -> a -> b -> c -> d -> e
```

然而，手动为元数从2到（例如）20的元组编写此类函数将非常繁琐（且忽略了程序中出现20元组几乎肯定表明设计存在问题，应通过记录类型来修正）。

我们可以使用Template Haskell为任意的 n 生成这样的 curryN 函数：

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Monad (replicateM)
import Language.Haskell.TH (ExpQ, newName, Exp(..), Pat(..))
import Numeric.Natural (Natural)

curryN :: Natural -> Q Exp
```

函数 curryN 接受一个自然数，生成该元数的curry函数，作为Haskell抽象语法树（AST）。

```
curryN n = do
f  <- newName "f"
xs <- replicateM (fromIntegral n) (newName "x")
```

首先，我们为函数的每个参数生成*fresh*类型变量——一个用于输入函数，
一个用于该函数的每个参数。

```
  let args = map VarP (f:xs)
```

表达式args表示模式f x1 x2 .. xn。注意，模式是一个独立的语法实体——我们可以将相同的模式放入lambda表达式、函数绑定，甚至let绑定的左侧（这将是错误的）。

```
ntup = TupE (map VarE xs)
```

函数必须从参数序列构建参数元组，这正是我们在这里所做的。注意模式变量（VarP）和表达式变量（VarE）之间的区别。

```
  return $ LamE args (AppE (VarE f) ntup)
```

最后，我们生成的值是抽象语法树（AST）\f x1 x2 .. xn -> f (x1, x2, .. , xn)。

我们也可以使用引用和“提升”的构造函数来编写此函数：

```
...
```

seem that the Q type is just a fancy IO. However, the use of runQ :: Q a -> IO a produces an IO action which does *not* have access to any compile-time environment - this is only available in the actual Q type. Such IO actions will fail at runtime if trying to access said environment.

Section 49.3: An n-arity curry

The familiar

```
curry :: ((a,b) -> c) -> a -> b -> c
curry = \f a b -> f (a,b)
```

function can be generalized to tuples of arbitrary arity, for example:

```
curry3 :: ((a, b, c) -> d) -> a -> b -> c -> d
curry4 :: ((a, b, c, d) -> e) -> a -> b -> c -> d -> e
```

However, writing such functions for tuples of arity 2 to (e.g.) 20 by hand would be tedious (and ignoring the fact that the presence of 20 tuples in your program almost certainly signal design issues which should be fixed with records).

We can use Template Haskell to produce such curryN functions for arbitrary n:

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Monad (replicateM)
import Language.Haskell.TH (ExpQ, newName, Exp(..), Pat(..))
import Numeric.Natural (Natural)

curryN :: Natural -> Q Exp
```

The curryN function takes a natural number, and produces the curry function of that arity, as a Haskell AST.

```
curryN n = do
  f  <- newName "f"
  xs <- replicateM (fromIntegral n) (newName "x")
```

First we produces *fresh* type variables for each of the arguments of the function - one for the input function, and one for each of the arguments to said function.

```
  let args = map VarP (f:xs)
```

The expression args represents the pattern f x1 x2 .. xn. Note that a pattern is separate syntactic entity - we could take this same pattern and place it in a lambda, or a function binding, or even the LHS of a let binding (which would be an error).

```
      ntup = TupE (map VarE xs)
```

The function must build the argument tuple from the sequence of arguments, which is what we've done here. Note the distinction between pattern variables (VarP) and expression variables (VarE).

```
  return $ LamE args (AppE (VarE f) ntup)
```

Finally, the value which we produce is the AST \f x1 x2 .. xn -> f (x1, x2, .. , xn).

We could have also written this function using quotations and 'lifted' constructors:

```
...
```

```
import Language.Haskell.TH.Lib

curryN' :: Natural -> ExpQ
curryN' n = do
f  <- newName "f"
xs <- replicateM (fromIntegral n) (newName "x")
  lamE (map varP (f:xs))
        [| $(varE f) $(tupE (map varE xs)) |]
```

注意，引用必须在语法上有效，因此 [| \ $(map varP (f:xs)) -> .. |] 是无效的，因为在常规 Haskell 中没有办法声明 “模式列表”——上述表达式被解释为 \ var -> ..，且拼接的表达式预期类型为 PatQ，即单个模式，而非模式列表。

最后，我们可以在 GHCi 中加载这个 TH 函数：

```
>:set -XTemplateHaskell
>:t $(curryN 5)
$(curryN 5)
  :: ((t1, t2, t3, t4, t5) -> t) -> t1 -> t2 -> t3 -> t4 -> t5 -> t
>$(curryN 5) (\(a,b,c,d,e) -> a+b+c+d+e) 1 2 3 4 5
15
```

此示例主要改编自 here。

```
import Language.Haskell.TH.Lib

curryN' :: Natural -> ExpQ
curryN' n = do
  f  <- newName "f"
  xs <- replicateM (fromIntegral n) (newName "x")
  lamE (map varP (f:xs))
        [| $(varE f) $(tupE (map varE xs)) |]
```

Note that quotations must be syntactically valid, so `[| \ $(map varP (f:xs)) -> .. |]` is invalid, because there is no way in regular Haskell to declare a 'list' of patterns - the above is interpreted as `\ var -> ..` and the spliced expression is expected to have type `PatQ`, i.e. a single pattern, not a list of patterns.

Finally, we can load this TH function in GHCi:

```
>:set -XTemplateHaskell
>:t $(curryN 5)
$(curryN 5)
  :: ((t1, t2, t3, t4, t5) -> t) -> t1 -> t2 -> t3 -> t4 -> t5 -> t
>$(curryN 5) (\(a,b,c,d,e) -> a+b+c+d+e) 1 2 3 4 5
15
```

This example is adapted primarily from here.

第50章：幻影类型

第50.1节：幻影类型的用例：货币

幻影类型对于处理具有相同表示但逻辑上不是同一类型的数据非常有用。

一个很好的例子是处理货币。如果你处理货币，你绝对不想例如将两种不同货币的金额相加。5.32欧元 + 2.94美元的结果货币是什么？这是未定义的，也没有合理的理由这样做。

一个解决方案可能如下所示：

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

数据 USD
数据 EUR

newtype Amount a = Amount Double
                 deriving (Show, Eq, Ord, Num)
```

GeneralizedNewtypeDeriving 扩展允许我们为 Amount 类型派生 Num。GHC 重用了 Double 的 Num 实例。

现在，如果你用例如 (5.0 :: Amount EUR) 来表示欧元金额，你就解决了在类型层面上将双精度金额区分开的问题，同时没有引入额外开销。像 (1.13 :: Amount EUR) + (5.30 :: Amount USD) 会导致类型错误，需要你适当处理货币转换。

更全面的文档可以在 haskell 维基文章中找到

Chapter 50: Phantom types

Section 50.1: Use Case for Phantom Types: Currencies

Phantom types are useful for dealing with data, that has identical representations but isn't logically of the same type.

A good example is dealing with currencies. If you work with currencies you absolutely never want to e.g. add two amounts of different currencies. What would the result currency of `5.32€ + 2.94$` be? It's not defined and there is no good reason to do this.

A solution to this could look something like this:

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

data USD
data EUR

newtype Amount a = Amount Double
                 deriving (Show, Eq, Ord, Num)
```

The `GeneralisedNewtypeDeriving` extension allows us to derive `Num` for the `Amount` type. GHC reuses `Double`'s `Num` instance.

Now if you represent Euro amounts with e.g. `(5.0 :: Amount EUR)` you have solved the problem of keeping double amounts separate at the type level without introducing overhead. Stuff like `(1.13 :: Amount EUR) + (5.30 :: Amount USD)` will result in a type error and require you to deal with currency conversion appropriately.

More comprehensive documentation can be found in the haskell wiki article

第51章：模块

第51.1节：定义你自己的模块

如果我们有一个名为 Business.hs 的文件，我们可以定义一个 Business 模块，可以被 import，如下所示：

```
module Business (
Person (..), -- ^ 导出 Person 类型及其所有构造函数和字段名
    employees    -- ^ 导出 employees 函数
) where
-- 开始类型、函数定义等
```

当然，更深层次的层级结构也是可能的；参见分层模块名称示例。

第51.2节：导出构造函数

要导出类型及其所有构造函数，必须使用以下语法：

```
module X (Person (..)) where
```

因此，对于名为People.hs的文件中的以下顶层定义：

```
data Person = Friend String | Foe deriving (Show, Eq, Ord)

isFoe Foe = True
isFoe _   = False
```

该模块顶部的声明：

```
module People (Person (..)) where
```

将只导出Person及其构造函数Friend和Foe。

如果省略module关键字后面的导出列表，模块中顶层绑定的所有名称
都将被导出：

```
module People where
```

将导出Person、其构造函数以及isFoe函数。

第51.3节：导入模块的特定成员

Haskell支持从模块中导入部分项目。

```
import qualified Data.Stream (map) as D
```

只会从Data.Stream导入map，调用该函数时需要使用D.前缀：

```
D.map odd [1..]
```

否则编译器会尝试使用Prelude中的map函数。

Chapter 51: Modules

Section 51.1: Defining Your Own Module

If we have a file called `Business.hs`, we can define a `Business` module that can be `import`-ed, like so:

```
module Business (
    Person (..), -- ^ Export the Person type and all its constructors and field names
    employees    -- ^ Export the employees function
) where
-- begin types, function definitions, etc
```

A deeper hierarchy is of course possible; see the Hierarchical module names example.

Section 51.2: Exporting Constructors

To export the type and all its constructors, one must use the following syntax:

```
module X (Person (..)) where
```

So, for the following top-level definitions in a file called `People.hs`:

```
data Person = Friend String | Foe deriving (Show, Eq, Ord)

isFoe Foe = True
isFoe _   = False
```

This module declaration at the top:

```
module People (Person (..)) where
```

would only export `Person` and its constructors `Friend` and `Foe`.

If the export list following the module keyword is omitted, all of the names bound at the top level of the module would be exported:

```
module People where
```

would export `Person`, its constructors, and the `isFoe` function.

Section 51.3: Importing Specific Members of a Module

Haskell supports importing a subset of items from a module.

```
import qualified Data.Stream (map) as D
```

would only import **map** from `Data.Stream`, and calls to this function would require `D.`:

```
D.map odd [1..]
```

otherwise the compiler will try to use `Prelude`'s **map** function.

第51.4节：隐藏导入

Prelude通常定义了一些名称在其他地方也会使用的函数。不隐藏这些导入（或在发生冲突时使用限定导入）会导致编译错误。

Data.Stream定义了名为map、head和tail的函数，这些函数通常与Prelude中定义的冲突。我们可以使用hiding来隐藏Prelude中的这些导入：

```
import Data.Stream -- 来自 Data.Stream 的所有内容
import Prelude 隐藏 (map, head, tail, scan, foldl, foldr, filter, dropWhile, take) -- 等等
```

实际上，隐藏 Prelude 冲突需要编写过多代码，因此实际上你会使用对 Data.Stream 的**限定**导入。

第51.5节：限定导入

当多个模块定义了同名函数时，编译器会报错。在这种情况下（或为了提高可读性），我们可以使用限定导入：

```
import qualified Data.Stream as D
```

现在，当我们使用在 Prelude 和 Data.Stream 中都定义的map时，可以避免歧义的编译错误：

```
map (== 1) [1,2,3] -- 将使用 Prelude.map
D.map (odd) (fromList [1..]) -- 将使用 Data.Stream.map
```

也可以只对冲突的名称进行限定导入，例如`import Data.Text as T`，这样就可以使用Text而不是T.Text等。

第51.6节：分层模块名称

模块名称遵循文件系统的分层结构。文件布局如下：

```
Foo/
├── Baz/
│   └── Quux.hs
└── Bar.hs
Foo.hs
Bar.hs
```

模块头部看起来如下：

```
-- 文件 Foo.hs
module Foo where

-- 文件 Bar.hs
module Bar where

-- 文件 Foo/Bar.hs
module Foo.Bar where

-- 文件 Foo/Baz/Quux.hs
module Foo.Baz.Quux where
```

注意：

Section 51.4: Hiding Imports

Prelude often defines functions whose names are used elsewhere. Not hiding such imports (or using qualified imports where clashes occur) will cause compilation errors.

`Data.Stream` defines functions named **`map`**, **`head`** and **`tail`** which normally clashes with those defined in Prelude. We can hide those imports from Prelude using **`hiding`**:

```
import Data.Stream -- everything from Data.Stream
import Prelude hiding (map, head, tail, scan, foldl, foldr, filter, dropWhile, take) -- etc
```

In reality, it would require too much code to hide Prelude clashes like this, so you would in fact use a **`qualified`** import of `Data.Stream` instead.

Section 51.5: Qualifying Imports

When multiple modules define the same functions by name, the compiler will complain. In such cases (or to improve readability), we can use a **`qualified`** import:

```
import qualified Data.Stream as D
```

Now we can prevent ambiguity compiler errors when we use **`map`**, which is defined in **`Prelude`** and `Data.Stream`:

```
map (== 1) [1,2,3] -- will use Prelude.map
D.map (odd) (fromList [1..]) -- will use Data.Stream.map
```

It is also possible to import a module with only the clashing names being qualified via `import Data.Text as T`, which allows one to have `Text` instead of `T.Text` etc.

Section 51.6: Hierarchical module names

The names of modules follow the filesystem's hierarchical structure. With the following file layout:

```
Foo/
├── Baz/
│   └── Quux.hs
└── Bar.hs
Foo.hs
Bar.hs
```

the module headers would look like this:

```
-- file Foo.hs
module Foo where

-- file Bar.hs
module Bar where

-- file Foo/Bar.hs
module Foo.Bar where

-- file Foo/Baz/Quux.hs
module Foo.Baz.Quux where
```

Note that:

- 模块名基于声明该模块的文件路径
- 文件夹可能与模块同名，这为模块提供了自然的层级命名结构

- the module name is based on the path of the file declaring the module
- Folders may share a name with a module, which gives a naturally hierarchical naming structure to modules

第52章：元组（对、三元组等）

第52.1节：提取元组组件

使用fst和snd函数（来自Prelude或Data.Tuple）来提取对偶的第一个和第二个分量。

```
fst (1, 2) -- 计算结果为1

snd (1, 2) -- 计算结果为2
```

或者使用模式匹配。

```
case (1, 2) of (result, _) => result -- 计算结果为1

case (1, 2) of (_, result) => result -- 计算结果为2
```

模式匹配同样适用于包含两个以上分量的元组。

```
case (1, 2, 3) of (result, _, _) => result -- 计算结果为1

case (1, 2, 3) of (_, result, _) => result -- 计算结果为2

case (1, 2, 3) of (_, _, result) => result -- 计算结果为3
```

Haskell 没有为包含两个以上分量的元组提供像fst或snd这样的标准函数。Hackage上的tuple库在Data.Tuple.Select模块中提供了此类函数。

第52.2节：匹配元组的严格性

模式(p1, p2)在最外层的元组构造器上是严格的，这可能导致意外的严格性行为。例如，以下表达式会发散（使用Data.Function.fix）：

```
fix $ \(x, y) -> (1, 2)
```

由于在元组构造函数中对(x, y)的匹配是严格的。然而，使用不可拒绝模式的以下表达式，按预期计算为(1, 2)：

```
fix $ \ ~(x, y) -> (1, 2)
```

第52.3节：构造元组值

使用括号和逗号来创建元组。使用一个逗号来创建一对元素。

```
(1, 2)
```

使用更多逗号来创建包含更多元素的元组。

```
(1, 2, 3)

(1, 2, 3, 4)
```

注意，也可以使用未简化的形式声明元组。

Chapter 52: Tuples (Pairs, Triples, ...)

Section 52.1: Extract tuple components

Use the **fst** and **snd** functions (from **Prelude** or Data.Tuple) to extract the first and second component of pairs.

```
fst (1, 2) -- evaluates to 1

snd (1, 2) -- evaluates to 2
```

Or use pattern matching.

```
case (1, 2) of (result, _) => result -- evaluates to 1

case (1, 2) of (_, result) => result -- evaluates to 2
```

Pattern matching also works for tuples with more than two components.

```
case (1, 2, 3) of (result, _, _) => result -- evaluates to 1

case (1, 2, 3) of (_, result, _) => result -- evaluates to 2

case (1, 2, 3) of (_, _, result) => result -- evaluates to 3
```

Haskell does not provide standard functions like **fst** or **snd** for tuples with more than two components. The tuple library on Hackage provides such functions in the Data.Tuple.Select module.

Section 52.2: Strictness of matching a tuple

The pattern (p1, p2) is strict in the outermost tuple constructor, which can lead to unexpected strictness behaviour. For example, the following expression diverges (using Data.Function.fix):

```
fix $ \(x, y) -> (1, 2)
```

since the match on (x, y) is strict in the tuple constructor. However, the following expression, using an irrefutable pattern, evaluates to (1, 2) as expected:

```
fix $ \ ~(x, y) -> (1, 2)
```

Section 52.3: Construct tuple values

Use parentheses and commas to create tuples. Use one comma to create a pair.

```
(1, 2)
```

Use more commas to create tuples with more components.

```
(1, 2, 3)

(1, 2, 3, 4)
```

Note that it is also possible to declare tuples using in their unsugared form.

```
(,) 1 2     -- 等同于 (1,2)
(,,) 1 2 3  -- 等同于 (1,2,3)
```

元组可以包含不同类型的值。

```
("answer", 42, '?')
```

元组可以包含复杂的值，例如列表或更多的元组。

```
([1, 2, 3], "hello", ('A', 65))

(1, (2, (3, 4), 5), 6)
```

第52.4节：编写元组类型

使用括号和逗号来编写元组类型。使用一个逗号来编写一对类型。

```
(Int, Int)
```

使用更多逗号来编写包含更多组件的元组类型。

```
(Int, Int, Int)

(Int, Int, Int, Int)
```

元组可以包含不同类型的值。

```
(String, Int, Char)
```

元组可以包含复杂的值，例如列表或更多的元组。

```
([Int], String, (Char, Int))

(Int, (Int, (Int, Int), Int), Int)
```

第52.5节：元组的模式匹配

元组的模式匹配使用元组构造器。例如，要匹配一个对，我们使用(,)构造器：

```
myFunction1 (a, b) = ...
```

我们使用更多的逗号来匹配包含更多元素的元组：

```
myFunction2 (a, b, c) = ...

myFunction3 (a, b, c, d) = ...
```

元组模式可以包含复杂的模式，例如列表模式或更多的元组模式。

```
myFunction4 ([a, b, c], d, e) = ...

myFunction5 (a, (b, (c, d), e), f) = ...
```

```
(,) 1 2     -- equivalent to (1,2)
(,,) 1 2 3  -- equivalent to (1,2,3)
```

Tuples can contain values of different types.

```
("answer", 42, '?')
```

Tuples can contain complex values such as lists or more tuples.

```
([1, 2, 3], "hello", ('A', 65))

(1, (2, (3, 4), 5), 6)
```

Section 52.4: Write tuple types

Use parentheses and commas to write tuple types. Use one comma to write a pair type.

```
(Int, Int)
```

Use more commas to write tuple types with more components.

```
(Int, Int, Int)

(Int, Int, Int, Int)
```

Tuples can contain values of different types.

```
(String, Int, Char)
```

Tuples can contain complex values such as lists or more tuples.

```
([Int], String, (Char, Int))

(Int, (Int, (Int, Int), Int), Int)
```

Section 52.5: Pattern Match on Tuples

Pattern matching on tuples uses the tuple constructors. To match a pair for example, we'd use the `(,)` constructor:

```
myFunction1 (a, b) = ...
```

We use more commas to match tuples with more components:

```
myFunction2 (a, b, c) = ...

myFunction3 (a, b, c, d) = ...
```

Tuple patterns can contain complex patterns such as list patterns or more tuple patterns.

```
myFunction4 ([a, b, c], d, e) = ...

myFunction5 (a, (b, (c, d), e), f) = ...
```

第52.6节：对元组应用二元函数（取消柯里化）

使用uncurry函数（来自Prelude或Data.Tuple）将二元函数转换为作用于元组的函数。

```
uncurry (+) (1, 2) -- 计算 3

uncurry map (negate, [1, 2, 3]) -- 计算 [-1, -2, -3]

uncurry uncurry ((+), (1, 2)) -- 计算 3

map (uncurry (+)) [(1, 2), (3, 4), (5, 6)] -- 计算结果为 [3, 7, 11]

uncurry (curry f) -- 计算结果与 f 相同
```

第52.7节：将元组函数应用于两个参数（柯里化）

使用 curry 函数（来自 Prelude 或 Data.Tuple）将接受元组的函数转换为接受两个参数的函数。

```
curry fst 1 2 -- 计算结果为 1

curry snd 1 2 -- 计算结果为 2

curry (uncurry f) -- 计算结果与 f 相同

import Data.Tuple (swap)
curry swap 1 2 -- 计算结果为 (2, 1)
```

第52.8节：交换元组的两个元素

使用 swap（来自 Data.Tuple）交换元组的两个元素。

```
import Data.Tuple (swap)
swap (1, 2) -- 计算结果为 (2, 1)
```

或者使用模式匹配。

```
case (1, 2) of (x, y) => (y, x) -- 计算结果为 (2, 1)
```

Section 52.6: Apply a binary function to a tuple (uncurrying)

Use the **uncurry** function (from **Prelude** or Data.Tuple) to convert a binary function to a function on tuples.

```
uncurry (+) (1, 2) -- computes 3

uncurry map (negate, [1, 2, 3]) -- computes [-1, -2, -3]

uncurry uncurry ((+), (1, 2)) -- computes 3

map (uncurry (+)) [(1, 2), (3, 4), (5, 6)] -- computes [3, 7, 11]

uncurry (curry f) -- computes the same as f
```

Section 52.7: Apply a tuple function to two arguments (currying)

Use the **curry** function (from **Prelude** or Data.Tuple) to convert a function that takes tuples to a function that takes two arguments.

```
curry fst 1 2 -- computes 1

curry snd 1 2 -- computes 2

curry (uncurry f) -- computes the same as f

import Data.Tuple (swap)
curry swap 1 2 -- computes (2, 1)
```

Section 52.8: Swap pair components

Use swap (from Data.Tuple) to swap the components of a pair.

```
import Data.Tuple (swap)
swap (1, 2) -- evaluates to (2, 1)
```

Or use pattern matching.

```
case (1, 2) of (x, y) => (y, x) -- evaluates to (2, 1)
```

第53章：使用 Gloss 进行图形绘制

第53.1节：安装Gloss

使用Cabal工具可以轻松安装Gloss。安装了Cabal后，可以运行cabal install gloss来安装Gloss。

或者也可以从源代码构建该包，通过从Hackage或GitHub下载源代码，然后执行以下操作：

1. 进入gloss/gloss-rendering/目录，执行cabal install
2. 进入gloss/gloss/目录，再次执行cabal install

第53.2节：在屏幕上显示内容

在Gloss中，可以使用display函数创建非常简单的静态图形。

要使用它，首先需要import Graphics.Gloss。然后代码中应包含以下内容：

```
main :: IO ()
main = display window background drawing
```

window的类型是Display，可以通过两种方式构造：

```
-- 将窗口定义为具有指定名称和大小的实际窗口
window = InWindow 名称 (宽度, 高度) (0,0)

-- 定义窗口为全屏窗口
window = FullScreen
```

这里 InWindow 中的最后一个参数 (0,0) 标记了左上角的位置。

对于早于1.11版本： 在较早版本的Gloss中，FullScreen 需要另一个参数，该参数表示绘制的框架大小，框架随后被拉伸到全屏大小，例如： FullScreen (1024,768)

background 的类型是 Color。它定义背景颜色，所以非常简单：

```
background = white
```

接下来是绘图本身。绘图可以非常复杂。如何指定这些将在其他地方介绍（[目前可以参考这里][1]），但它可以像下面这个半径为80的圆一样简单：

```
drawing = Circle 80
```

示例总结

正如Hackage文档中或多或少所述，将内容显示到屏幕上非常简单：

```
import Graphics.Gloss

main :: IO ()
main = display window background drawing
    其中
window = InWindow "漂亮的窗口" (200, 200) (0, 0)
```

Chapter 53: Graphics with Gloss

Section 53.1: Installing Gloss

Gloss is easily installed using the Cabal tool. Having installed Cabal, one can run `cabal install gloss` to install Gloss.

Alternatively the package can be built from source, by downloading the source from Hackage or GitHub, and doing the following:

1. Enter the `gloss/gloss-rendering/` directory and do `cabal install`
2. Enter the `gloss/gloss/` directory and once more do `cabal install`

Section 53.2: Getting something on the screen

In Gloss, one can use the `display` function to create very simple static graphics.

To use this one needs to first `import Graphics.Gloss`. Then in the code there should the following:

```
main :: IO ()
main = display window background drawing
```

`window` is of type `Display` which can be constructed in two ways:

```
-- Defines window as an actual window with a given name and size
window = InWindow name (width, height) (0,0)

-- Defines window as a fullscreen window
window = FullScreen
```

Here the last argument `(0,0)` in `InWindow` marks the location of the top left corner.

For versions older than 1.11: In older versions of Gloss `FullScreen` takes another argument which is meant to be the size of the frame that gets drawn on which in turn gets stretched to fullscreen-size, for example: `FullScreen (1024,768)`

`background` is of type `Color`. It defines the background color, so it's as simple as:

```
background = white
```

Then we get to the drawing itself. Drawings can be very complex. How to specify these will be covered elsewhere ([one can refer to this for the moment][1]), but it can be as simple as the following circle with a radius of 80:

```
drawing = Circle 80
```

Summarizing example

As more or less stated in the documentation on Hackage, getting something on the screen is as easy as:

```
import Graphics.Gloss

main :: IO ()
main = display window background drawing
    where
      window = InWindow "Nice Window" (200, 200) (0, 0)
```

```haskell
background = white
      drawing = Circle 80
```

```haskell
      background = white
      drawing = Circle 80
```

第54章：状态单子

状态单子是一种携带状态的单子，该状态可能在单子中的每次计算运行时发生变化。实现通常形式为 State s a，表示一个携带并可能修改类型为 s 的状态并产生类型为 a 结果的计算，但"状态单子"一词通常指任何携带状态的单子。 mtl 和 transformers 包提供了状态单子的一般实现。

第54.1节：用计数器给树的节点编号

我们有如下的树数据类型：

```
data Tree a = Tree a [Tree a] deriving Show
```

我们希望编写一个函数，为树的每个节点分配一个数字，数字来自一个递增的计数器：

```
tag :: Tree a -> Tree (a, Int)
```

长方法

首先我们用长方法来做，因为它很好地展示了State单子的底层机制。

```
import Control.Monad.State

-- 为`Tree`的节点编号的函数。
tag :: Tree a -> Tree (a, Int)
tag tree =
    -- tagStep 是实际执行操作的地方。它只是启动
    -- 过程，初始计数器值为`0`。
evalState (tagStep tree) 0

-- 这是计算的一个单子"步骤"。它假设
-- 它隐式地访问当前计数器的值。
tagStep :: Tree a -> State Int (Tree (a, Int))
tagStep (Tree a 子树) = do
    -- `get :: State s s` 操作访问 State 单子隐式的状态
    -- 参数。在这里我们将该值绑定到
    -- 变量 `counter`。
counter <- get

    -- `put :: s -> State s ()` 设置 State 单子隐式的状态参数
    -- 下一个执行的 `get` 将看到
    -- `counter + 1` 的值（假设中间没有其他的 put 操作）。
put (counter + 1)

    -- 递归进入子树。`mapM` 是一个实用函数
    -- 用于在列表上执行单子操作（如 `tagStep`）
    -- 元素，并生成结果列表。每次执行
    -- `tagStep` 将使用产生的计数器值执行
    -- 来自前一个列表元素的执行。
   子树' <- mapM tagStep 子树

    return $ Tree (a, counter) subtrees'
```

重构

将计数器拆分为一个后增量操作

获取当前计数器然后将计数器加1的部分可以拆分成一个postIncrement

Chapter 54: State Monad

State monads are a kind of monad that carry a state that might change during each computation run in the monad. Implementations are usually of the form `State s a` which represents a computation that carries and potentially modifies a state of type `s` and produces a result of type `a`, but the term "state monad" may generally refer to any monad which carries a state. The `mtl` and `transformers` package provide general implementations of state monads.

Section 54.1: Numbering the nodes of a tree with a counter

We have a tree data type like this:

```
data Tree a = Tree a [Tree a] deriving Show
```

And we wish to write a function that assigns a number to each node of the tree, from an incrementing counter:

```
tag :: Tree a -> Tree (a, Int)
```

The long way

First we'll do it the long way around, since it illustrates the `State` monad's low-level mechanics quite nicely.

```
import Control.Monad.State

-- Function that numbers the nodes of a `Tree`.
tag :: Tree a -> Tree (a, Int)
tag tree =
    -- tagStep is where the action happens.  This just gets the ball
    -- rolling, with `0` as the initial counter value.
    evalState (tagStep tree) 0

-- This is one monadic "step" of the calculation.  It assumes that
-- it has access to the current counter value implicitly.
tagStep :: Tree a -> State Int (Tree (a, Int))
tagStep (Tree a subtrees) = do
    -- The `get :: State s s` action accesses the implicit state
    -- parameter of the State monad.  Here we bind that value to
    -- the variable `counter`.
    counter <- get

    -- The `put :: s -> State s ()` sets the implicit state parameter
    -- of the `State` monad.  The next `get` that we execute will see
    -- the value of `counter + 1` (assuming no other puts in between).
    put (counter + 1)

    -- Recurse into the subtrees.  `mapM` is a utility function
    -- for executing a monadic actions (like `tagStep`) on a list of
    -- elements, and producing the list of results.  Each execution of
    -- `tagStep` will be executed with the counter value that resulted
    -- from the previous list element's execution.
    subtrees' <- mapM tagStep subtrees

    return $ Tree (a, counter) subtrees'
```

Refactoring

Split out the counter into a postIncrement action

The bit where we are `getting` the current counter and then `putting` counter + 1 can be split off into a `postIncrement`

动作，类似于许多C风格语言提供的：

```
postIncrement :: Enum s => State s s
postIncrement = do
result <- get
    modify succ
    return result
```

将树遍历拆分成一个高阶函数

树遍历逻辑可以拆分成它自己的函数，如下所示：

```
mapTreeM :: Monad m => (a -> m b) -> Tree a -> m (Tree b)
mapTreeM action (Tree a subtrees) = do
a' <- action a
subtrees' <- mapM (mapTreeM action) subtrees
    return $ Tree a' subtrees'
```

有了这个和postIncrement函数，我们可以重写tagStep：

```
tagStep :: Tree a -> State Int (Tree (a, Int))
tagStep = mapTreeM step
    where step :: a -> State Int (a, Int)
          step a = do
              counter <- postIncrement
              return (a, counter)
```

使用Traversable类

上面的mapTreeM解决方案可以很容易地重写为Traversable类的一个实例：

```
instance Traversable Tree where
    traverse action (Tree a subtrees) =
        Tree <$> action a <*> traverse action subtrees
```

注意，这要求我们使用Applicative（即<*>操作符），而不是Monad。

有了这个，现在我们可以像专业人士一样编写tag了：

```
tag :: Traversable t => t a -> t (a, Int)
tag init t = evalState (traverse step t) 0
    where step a = do tag <- postIncrement
                      return (a, tag)
```

注意，这适用于任何Traversable类型，而不仅仅是我们的Tree类型！

去除Traversable模板代码

GHC有一个DeriveTraversable扩展，可以省去编写上述实例的需要：

```
{-# LANGUAGE DeriveFunctor, DeriveFoldable, DeriveTraversable #-}

data Tree a = Tree a [Tree a]
            deriving (Show, Functor, Foldable, Traversable)
```

action, similar to what many C-style languages provide:

```
postIncrement :: Enum s => State s s
postIncrement = do
    result <- get
    modify succ
    return result
```

Split out the tree walk into a higher-order function

The tree walk logic can be split out into its own function, like this:

```
mapTreeM :: Monad m => (a -> m b) -> Tree a -> m (Tree b)
mapTreeM action (Tree a subtrees) = do
    a' <- action a
    subtrees' <- mapM (mapTreeM action) subtrees
    return $ Tree a' subtrees'
```

With this and the `postIncrement` function we can rewrite `tagStep`:

```
tagStep :: Tree a -> State Int (Tree (a, Int))
tagStep = mapTreeM step
    where step :: a -> State Int (a, Int)
          step a = do
              counter <- postIncrement
              return (a, counter)
```

Use the `Traversable` class

The `mapTreeM` solution above can be easily rewritten into an instance of the `Traversable` class:

```
instance Traversable Tree where
    traverse action (Tree a subtrees) =
        Tree <$> action a <*> traverse action subtrees
```

Note that this required us to use `Applicative` (the `<*>` operator) instead of `Monad`.

With that, now we can write `tag` like a pro:

```
tag :: Traversable t => t a -> t (a, Int)
tag init t = evalState (traverse step t) 0
    where step a = do tag <- postIncrement
                      return (a, tag)
```

Note that this works for any `Traversable` type, not just our `Tree` type!

Getting rid of the `Traversable` boilerplate

GHC has a `DeriveTraversable` extension that eliminates the need for writing the instance above:

```
{-# LANGUAGE DeriveFunctor, DeriveFoldable, DeriveTraversable #-}

data Tree a = Tree a [Tree a]
            deriving (Show, Functor, Foldable, Traversable)
```

第55章：管道

第55.1节：生产者

生产者（Producer）是一种单子动作，可以产出值供下游消费：

```
type Producer b = Proxy X () () b
yield :: Monad m => a -> Producer a m ()
```

例如：

```
naturals :: Monad m => Producer Int m ()
naturals = each [1..] -- each 是 Pipes 导出的一个实用函数
```

当然，我们也可以有基于其他值的函数形式的生产者：

```
naturalsUntil :: Monad m => Int -> Producer Int m ()
naturalsUntil n = each [1..n]
```

第55.2节：连接管道

使用>->连接生产者（Producer）、消费者（Consumer）和管道（Pipe）以组合更大的管道（Pipe）函数。

```
printNaturals :: MonadIO m => Effect m ()
printNaturals = naturalsUntil 10 >-> intToStr >-> fancyPrint
```

Producer、Consumer、Pipe 和 Effect 类型都是基于通用的 Proxy 类型定义的。因此 >-> 可以用于多种用途。由左侧参数定义的类型必须与右侧参数消费的类型匹配：

```
(>->) :: Monad m => Producer b m r -> Consumer b   m r -> Effect       m r
(>->) :: Monad m => Producer b m r -> Pipe     b c m r -> Producer   c m r
(>->) :: Monad m => Pipe   a b m r -> Consumer b   m r -> Consumer a   m r
(>->) :: Monad m => Pipe   a b m r -> Pipe     b c m r -> Pipe     a c m r
```

第55.3节：管道（Pipes）

管道（Pipes）既可以 await 也可以 yield。

```
类型 Pipe a b = Proxy () a () b
```

此管道等待一个 Int 并将其转换为 String：

```
intToStr :: Monad m => Pipe Int String m ()
intToStr = forever $ await >>= (yield . show)
```

第55.4节：使用 runEffect 运行管道

我们使用 runEffect 来运行我们的 Pipe：

```
main :: IO ()
main = do
runEffect $ naturalsUntil 10 >-> intToStr >-> fancyPrint
```

Chapter 55: Pipes

Section 55.1: Producers

A Producer is some monadic action that can yield values for downstream consumption:

```
type Producer b = Proxy X () () b
yield :: Monad m => a -> Producer a m ()
```

For example:

```
naturals :: Monad m => Producer Int m ()
naturals = each [1..] -- each is a utility function exported by Pipes
```

We can of course have Producers that are functions of other values too:

```
naturalsUntil :: Monad m => Int -> Producer Int m ()
naturalsUntil n = each [1..n]
```

Section 55.2: Connecting Pipes

Use >-> to connect Producers, Consumers and Pipes to compose larger Pipe functions.

```
printNaturals :: MonadIO m => Effect m ()
printNaturals = naturalsUntil 10 >-> intToStr >-> fancyPrint
```

Producer, Consumer, Pipe, and Effect types are all defined in terms of the general Proxy type. Therefore >-> can be used for a variety of purposes. Types defined by the left argument must match the type consumed by the right argument:

```
(>->) :: Monad m => Producer b m r -> Consumer b   m r -> Effect       m r
(>->) :: Monad m => Producer b m r -> Pipe     b c m r -> Producer   c m r
(>->) :: Monad m => Pipe   a b m r -> Consumer b   m r -> Consumer a   m r
(>->) :: Monad m => Pipe   a b m r -> Pipe     b c m r -> Pipe     a c m r
```

Section 55.3: Pipes

Pipes can both await and yield.

```
type Pipe a b = Proxy () a () b
```

This Pipe awaits an Int and converts it to a String:

```
intToStr :: Monad m => Pipe Int String m ()
intToStr = forever $ await >>= (yield . show)
```

Section 55.4: Running Pipes with runEffect

We use runEffect to run our Pipe:

```
main :: IO ()
main = do
  runEffect $ naturalsUntil 10 >-> intToStr >-> fancyPrint
```

请注意，runEffect 需要一个 Effect，它是一个自包含的 Proxy，且没有输入或输出：

```
runEffect :: Monad m => Effect m r -> m r
type Effect = Proxy X () () X
```

（其中X是空类型，也称为Void）。

第55.5节：消费者

消费者只能等待来自上游的值。

```
type Consumer a = Proxy () a () X
await :: Monad m => Consumer a m a
```

例如：

```
fancyPrint :: MonadIO m => Consumer String m ()
fancyPrint = forever $ do
  numStr <- await
liftIO $ putStrLn ("我收到: " ++ numStr)
```

第55.6节：代理单子转换器

pipes的核心数据类型是Proxy单子变换器。Pipe、Producer、Consumer等都是基于

Proxy定义的。

由于Proxy是一个单子变换器，Pipe的定义采用单子脚本的形式，这些脚本会await和yield
值，同时执行来自基础单子m的效果。

第55.7节：结合Pipes和网络通信

Pipes支持客户端和服务器之间的简单二进制通信

在此示例中：

1. 客户端连接并发送一个FirstMessage
2. 服务器接收并回复DoSomething0
3. 客户端接收并回复DoNothing

第4步中的第2步和第3步无限重复

通过网络交换的命令数据类型：

```
-- Command.hs
{-# LANGUAGE DeriveGeneric #-}
module Command where
import Data.Binary
import GHC.Generics (Generic)

data Command = FirstMessage
          | DoNothing
          | DoSomething Int
          deriving (Show,Generic)

instance Binary Command
```

这里，服务器等待客户端连接：

Note that runEffect requires an Effect, which is a self-contained Proxy with no inputs or outputs:

```
runEffect :: Monad m => Effect m r -> m r
type Effect = Proxy X () () X
```

(where X is the empty type, also known as Void).

Section 55.5: Consumers

A Consumer can only await values from upstream.

```
type Consumer a = Proxy () a () X
await :: Monad m => Consumer a m a
```

For example:

```
fancyPrint :: MonadIO m => Consumer String m ()
fancyPrint = forever $ do
  numStr <- await
  liftIO $ putStrLn ("I received: " ++ numStr)
```

Section 55.6: The Proxy monad transformer

pipes's core data type is the Proxy monad transformer. Pipe, Producer, Consumer and so on are defined in terms of Proxy.

Since Proxy is a monad transformer, definitions of Pipes take the form of monadic scripts which await and yield values, additionally performing effects from the base monad m.

Section 55.7: Combining Pipes and Network communication

Pipes supports simple binary communication between a client and a server

In this example:

1. a client connects and sends a FirstMessage
2. the server receives and answers DoSomething 0
3. the client receives and answers DoNothing
4. step 2 and 3 are repeated indefinitely

The command data type exchanged over the network:

```
-- Command.hs
{-# LANGUAGE DeriveGeneric #-}
module Command where
import Data.Binary
import GHC.Generics (Generic)

data Command = FirstMessage
          | DoNothing
          | DoSomething Int
          deriving (Show,Generic)

instance Binary Command
```

Here, the server waits for a client to connect:

```
module Server where

import Pipes
import qualified Pipes.Binary as PipesBinary
import qualified Pipes.Network.TCP as PNT
import qualified Command as C
import qualified Pipes.Parse as PP
import qualified Pipes.Prelude as PipesPrelude

pageSize :: Int
pageSize = 4096

-- 纯处理器，用于 PipesPrelude.map
pureHandler :: C.Command -> C.Command
pureHandler c = c  -- 回答我们收到的相同命令

-- 非纯处理器，用于 PipesPremude.mapM
sideffectHandler :: MonadIO m => C.Command -> m C.Command
sideffectHandler c = do
liftIO $ putStrLn $ "接收到的消息 = " ++ (show c)
  return $ C.DoSomething 0
  -- 无论客户端传入什么命令 `c`，都回复 DoSomething 0

main :: IO ()
main = PNT.serve (PNT.Host "127.0.0.1") "23456" $
  \(connectionSocket, remoteAddress) -> do
                 putStrLn $ "远程连接来自 IP = " ++ (show remoteAddress)
                 _ <- runEffect $ do
                   let bytesReceiver = PNT.fromSocket connectionSocket pageSize
                   let commandDecoder = PP.parsed PipesBinary.decode bytesReceiver
                   commandDecoder >-> PipesPrelude.mapM sideffectHandler >-> for cat
PipesBinary.encode >-> PNT.toSocket connectionSocket
                   -- 如果我们想使用 pureHandler
                   --commandDecoder >-> PipesPrelude.map pureHandler >-> for cat PipesBinary.Encode
>-> PNT.toSocket connectionSocket
                 return ()
```

客户端连接方式如下：

```
模块 Client 位置

导入 Pipes
导入限定 Pipes.Binary 作为 PipesBinary
导入限定 Pipes.Network.TCP 作为 PNT
导入限定 Pipes.Prelude 作为 PipesPrelude
导入限定 Pipes.Parse 作为 PP
导入限定 Command 作为 C

pageSize :: Int
pageSize = 4096

-- 纯处理器，用于 PipesPrelude.amp
pureHandler :: C.Command -> C.Command
pureHandler c = c  -- 回复从服务器接收到的相同命令

-- 非纯处理器，用于 PipesPremude.mapM
sideffectHandler :: MonadIO m => C.Command -> m C.Command
sideffectHandler c = do
liftIO $ putStrLn $ "Received: " ++ (show c)
  return C.DoNothing  -- 无论从服务器接收到什么，回复 DoNothing
```

```
module Server where

import Pipes
import qualified Pipes.Binary as PipesBinary
import qualified Pipes.Network.TCP as PNT
import qualified Command as C
import qualified Pipes.Parse as PP
import qualified Pipes.Prelude as PipesPrelude

pageSize :: Int
pageSize = 4096

-- pure handler, to be used with PipesPrelude.map
pureHandler :: C.Command -> C.Command
pureHandler c = c  -- answers the same command that we have receveid

-- impure handler, to be used with PipesPremude.mapM
sideffectHandler :: MonadIO m => C.Command -> m C.Command
sideffectHandler c = do
  liftIO $ putStrLn $ "received message = " ++ (show c)
  return $ C.DoSomething 0
  -- whatever incoming command `c` from the client, answer DoSomething 0

main :: IO ()
main = PNT.serve (PNT.Host "127.0.0.1") "23456" $
  \(connectionSocket, remoteAddress) -> do
                 putStrLn $ "Remote connection from ip = " ++ (show remoteAddress)
                 _ <- runEffect $ do
                   let bytesReceiver = PNT.fromSocket connectionSocket pageSize
                   let commandDecoder = PP.parsed PipesBinary.decode bytesReceiver
                   commandDecoder >-> PipesPrelude.mapM sideffectHandler >-> for cat
PipesBinary.encode >-> PNT.toSocket connectionSocket
                   -- if we want to use the pureHandler
                   --commandDecoder >-> PipesPrelude.map pureHandler >-> for cat PipesBinary.Encode
>-> PNT.toSocket connectionSocket
                 return ()
```

The client connects thus:

```
module Client where

import Pipes
import qualified Pipes.Binary as PipesBinary
import qualified Pipes.Network.TCP as PNT
import qualified Pipes.Prelude as PipesPrelude
import qualified Pipes.Parse as PP
import qualified Command as C

pageSize :: Int
pageSize = 4096

-- pure handler, to be used with PipesPrelude.amp
pureHandler :: C.Command -> C.Command
pureHandler c = c  -- answer the same command received from the server

-- inpure handler, to be used with PipesPremude.mapM
sideffectHandler :: MonadIO m => C.Command -> m C.Command
sideffectHandler c = do
  liftIO $ putStrLn $ "Received: " ++ (show c)
  return C.DoNothing  -- whatever is received from server, answer DoNothing
```

```
main :: IO ()
main = PNT.connect ("127.0.0.1") "23456" $
  \(connectionSocket, remoteAddress) -> do
    putStrLn $ "已连接到远程服务器，ip = " ++ (show remoteAddress)
    sendFirstMessage connectionSocket
    _ <- runEffect $ do
      let bytesReceiver = PNT.fromSocket connectionSocket pageSize
      let commandDecoder = PP.parsed PipesBinary.decode bytesReceiver
      commandDecoder >-> PipesPrelude.mapM sideffectHandler >-> for cat PipesBinary.encode >->
PNT.toSocket connectionSocket
    return ()

sendFirstMessage :: PNT.Socket -> IO ()
sendFirstMessage s = do
  _ <- runEffect $ do
    let encodedProducer = PipesBinary.encode C.FirstMessage
    encodedProducer >-> PNT.toSocket s
  return ()
```

```
main :: IO ()
main = PNT.connect ("127.0.0.1") "23456" $
  \(connectionSocket, remoteAddress) -> do
    putStrLn $ "Connected to distant server ip = " ++ (show remoteAddress)
    sendFirstMessage connectionSocket
    _ <- runEffect $ do
      let bytesReceiver = PNT.fromSocket connectionSocket pageSize
      let commandDecoder = PP.parsed PipesBinary.decode bytesReceiver
      commandDecoder >-> PipesPrelude.mapM sideffectHandler >-> for cat PipesBinary.encode >->
PNT.toSocket connectionSocket
    return ()

sendFirstMessage :: PNT.Socket -> IO ()
sendFirstMessage s = do
  _ <- runEffect $ do
    let encodedProducer = PipesBinary.encode C.FirstMessage
    encodedProducer >-> PNT.toSocket s
  return ()
```

第56章：中缀运算符

第56.1节：序言

逻辑

&& 是逻辑与，|| 是逻辑或。

== 是等于， /= 是不等于， < / <= 是小于， > / >= 是大于运算符。

算术运算符

数值运算符 +、- 和 / 的行为大致符合预期。（除法仅适用于分数，以避免舍入问题–整数除法必须使用 quot 或 div 进行）。更为特殊的是 Haskell 的三个指数运算符：

- ^ 将任何数值类型的底数提升到非负整数次幂。这通过（快速）迭代乘法简单实现。例如。

```
4^5  ≡  (4*4)*(4*4)*4
```

- ^^ 在正指数情况下作用相同，但也适用于负指数。例如。

```
3^^(-2)  ≡  1 / (2*2)
```

与 ^ 不同，这需要分数底数类型（即 4^^5 :: Int 不适用，仅 4^5 :: Int 或 4^^5 :: Rational）可用。

- ** 实现实数指数运算。它适用于非常通用的参数，但计算开销比 ^ 或 ^^ 大，且通常会产生小的浮点误差。

```
2**pi  ≡  exp (pi * log 2)
```

列表

有两个连接运算符：

- :（读作 cons）将单个参数添加到列表前面。该运算符实际上是一个构造器，因此也可以用于模式匹配（“逆构造”）列表。
- ++ 连接整个列表。

```
[1,2] ++ [3,4]  ≡  1 : 2 : [3,4]  ≡  1 : [2,3,4]  ≡  [1,2,3,4]
```

!! 是一个索引操作符。

```
[0, 10, 20, 30, 40] !! 3  ≡  30
```

注意，对列表进行索引效率较低（复杂度为 O(n)，而数组为 O(1)，映射为 O(log n)）；在 Haskell 中，通常更倾向于通过折叠或模式匹配来解构列表，而不是使用索引。

Chapter 56: Infix operators

Section 56.1: Prelude

Logical

&& is logical AND, || is logical OR.

== is equality, /= non-equality, < / <= lesser and > / >= greater operators.

Arithmetic operators

The numerical operators +, - and / behave largely as you'd expect. (Division works only on fractional numbers to avoid rounding issues – integer division must be done with quot or div). More unusual are Haskell's three exponentiation operators:

- ^ takes a base of any number type to a non-negative, integral power. This works simply by (fast) iterated multiplication. E.g.

```
4^5  ≡  (4*4)*(4*4)*4
```

- ^^ does the same in the positive case, but also works for negative exponents. E.g.

```
3^^(-2)  ≡  1 / (2*2)
```

Unlike ^, this requires a fractional base type (i.e. 4^^5 :: Int will not work, only 4^5 :: Int or 4^^5 :: Rational).

- ** implements real-number exponentiation. This works for very general arguments, but is more computionally expensive than ^ or ^^, and generally incurs small floating-point errors.

```
2**pi  ≡  exp (pi * log 2)
```

Lists

There are two concatenation operators:

- : (pronounced cons) prepends a single argument before a list. This operator is actually a constructor and can thus also be used to *pattern match* (“inverse construct”) a list.
- ++ concatenates entire lists.

```
[1,2] ++ [3,4]  ≡  1 : 2 : [3,4]  ≡  1 : [2,3,4]  ≡  [1,2,3,4]
```

!! is an indexing operator.

```
[0, 10, 20, 30, 40] !! 3  ≡  30
```

Note that indexing lists is inefficient (complexity *O*(*n*) instead of *O*(1) for arrays or *O*(log *n*) for maps); it's generally preferred in Haskell to deconstruct lists by folding ot pattern matching instead of indexing.

控制流

- $ 是函数应用操作符。

```
f $ x  ≡  f x
       ≡  f(x)  -- 不推荐的写法
```

  该操作符主要用于避免括号。它还有一个严格版本 $!，强制在应用函数之前先计算参数。

- . 用于函数组合。

```
(f . g) x  ≡  f (g x)  ≡  f $ g x
```

- >> 序列化单子动作。例如，writeFile "foo.txt" "bla" >> putStrLn "Done." 会先写入文件，随后在屏幕上打印一条消息。
- ___ >= 也执行相同操作，同时接受一个参数，从第一个动作传递给后续动作。readLn >>= \x -> print (x^2) 会等待用户输入一个数字，然后将该数字的平方输出到屏幕上。

第56.2节：查找中缀运算符的信息

由于中缀运算符在Haskell中非常常见，你经常需要查找它们的签名等信息。幸运的是，这和查找其他函数一样简单：

- Haskell搜索引擎Hayoo和Hoogle可以用于查找中缀运算符，就像查找任何其他库中定义的内容一样。
- 在GHCi或IHaskell中，你可以使用:i和:t（info和type）指令来了解运算符的基本属性。例如，

```
Prelude> :i +
class Num a where
  (+) :: a -> a -> a
  ...
      -- 定义于‘GHC.Num’
infixl 6 +
Prelude> :i ^^
(^^) :: (Fractional a, Integral b) => a -> b -> a
      -- 定义于 ‘GHC.Real’
infixr 8 ^^
```

  这告诉我 ^^ 的结合力比 + 强，两者的元素类型都是数值类型，但 ^^ 要求指数为整数，底数为分数类型。

  更简洁的 :t 需要将操作符放在括号中，例如

```
Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool
```

第56.3节：自定义操作符

在Haskell中，你可以定义任意中缀操作符。例如，我可以定义列表包裹操作符为

Control flow

- $ is a function application operator.

```
f $ x  ≡  f x
       ≡  f(x)  -- disapproved style
```

  This operator is mostly used to avoid parentheses. It also has a strict version $!, which forces the argument to be evaluated before applying the function.

- . composes functions.

```
(f . g) x  ≡  f (g x)  ≡  f $ g x
```

- >> sequences monadic actions. E.g. **writeFile** "foo.txt" "bla" >> **putStrLn** "Done." will first write to a file, then print a message to the screen.
- >>= does the same, while also accepting an argument to be passed from the first action to the following. **readLn** >>= \x -> **print** (x^2) will wait for the user to input a number, then output the square of that number to the screen.

Section 56.2: Finding information about infix operators

Because infixes are so common in Haskell, you will regularly need to look up their signature etc.. Fortunately, this is just as easy as for any other function:

- The Haskell search engines Hayoo and Hoogle can be used for infix operators, like for anything else that's defined in some library.
- In GHCi or IHaskell, you can use the :i and :t (**i**nfo and **t**ype) directives to learn the basic properties of an operator. For example,

```
Prelude> :i +
class Num a where
  (+) :: a -> a -> a
  ...
      -- Defined in ‘GHC.Num’
infixl 6 +
Prelude> :i ^^
(^^) :: (Fractional a, Integral b) => a -> b -> a
      -- Defined in ‘GHC.Real’
infixr 8 ^^
```

  This tells me that ^^ binds more tightly than +, both take numerical types as their elements, but ^^ requires the exponent to be integral and the base to be fractional.
  The less verbose :t requires the operator in parentheses, like

```
Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool
```

Section 56.3: Custom operators

In Haskell, you can define any infix operator you like. For example, I could define the list-enveloping operator as

```haskell
(>+<) :: [a] -> [a] -> [a]
env >+< l = env ++ l ++ env

GHCi> "**">+<"emphasis"
"**emphasis**"
```

你应该总是给这样的操作符一个结合性声明，比如

```haskell
infixr 5 >+<
```

(这意味着>+<的结合力和++以及:一样强。)

```haskell
(>+<) :: [a] -> [a] -> [a]
env >+< l = env ++ l ++ env

GHCi> "**">+<"emphasis"
"**emphasis**"
```

You should always give such operators a fixity declaration, like

```haskell
infixr 5 >+<
```

(which would mean `>+<` binds as tightly as `++` and `:` do).

第57章：并行

类型/函数	详细信息
data Eval a	Eval是一个Monad，使定义并行策略更容易
type Strategy a = a -> Eval a	一个体现并行评估策略的函数。该函数遍历其参数的（部分）内容，按并行或顺序方式评估子表达式
rpar :: 策略 a	并行计算其参数（用于并行评估）
rseq :: 策略 a	将其参数计算到弱头正常形式
force :: NFData a => a -> a	在返回参数本身之前，计算其参数的整个结构，将其规约到正常形式。该函数由 Control.DeepSeq 模块提供

第57.1节：Eval Monad

Haskell 中的并行性可以使用 Control.Parallel.Strategies 中的 Eval Monad 来表达，使用 rpar 和 rseq 函数（以及其他函数）。

```
f1 :: [Int]
f1 = [1..100000000]

f2 :: [Int]
f2 = [1..000000000]

main = runEval $ do
a <- rpar (f1) -- 这将花费一些时间...
b <- rpar (f2) -- 这将花费更多时间...
  return (a,b)
```

运行上面的 main 会立即执行并“返回”，而两个值 a 和 b 则通过 rpar 在后台计算。

注意：确保使用 -threaded 编译，以实现并行执行。

第57.2节：rpar

rpar :: Strategy a 并行执行给定的策略（回顾：type Strategy a = a -> Eval a）：

```
import Control.Concurrent
import Control.DeepSeq
import Control.Parallel.Strategies
import Data.List.Ordered

main = loop
  其中
loop = do
        putStrLn "请输入一个数字"
       n <- getLine

        let lim = read n :: Int
hf  = quot lim 2
result = runEval $ do
                -- 我们将计算分成两半，这样可以并行计算素数
                as <- rpar (force (primesBtwn 2 hf))
bs <- rpar (force (primesBtwn (hf + 1) lim))
                return (as ++ bs)

forkIO $ putStrLn ("素数是: " ++🟩(show result) ++ " 对于 " ++ n ++ "")                    🟩
```

Chapter 57: Parallelism

Type/Function	Detail
data Eval a	Eval is a Monad that makes it easier to define parallel strategies
type Strategy a = a -> Eval a	a function that embodies a parallel evaluation strategy. The function traverses (parts of) its argument, evaluating subexpressions in parallel or in sequence
rpar :: Strategy a	sparks its argument (for evaluation in parallel)
rseq :: Strategy a	evaluates its argument to weak head normal form
force :: NFData a => a -> a	evaluates the entire structure of its argument, reducing it to normal form, before returning the argument itself. It is provided by the Control.DeepSeq module

Section 57.1: The Eval Monad

Parallelism in Haskell can be expressed using the `Eval` Monad from `Control.Parallel.Strategies`, using the `rpar` and `rseq` functions (among others).

```
f1 :: [Int]
f1 = [1..100000000]

f2 :: [Int]
f2 = [1..200000000]

main = runEval $ do
  a <- rpar (f1) -- this'll take a while...
  b <- rpar (f2) -- this'll take a while and then some...
  return (a,b)
```

Running `main` above will execute and "return" immediately, while the two values, `a` and `b` are computed in the background through `rpar`.

Note: ensure you compile with `-threaded` for parallel execution to occur.

Section 57.2: rpar

`rpar :: Strategy a` executes the given strategy (recall: `type Strategy a = a -> Eval a`) in parallel:

```
import Control.Concurrent
import Control.DeepSeq
import Control.Parallel.Strategies
import Data.List.Ordered

main = loop
  where
    loop = do
      putStrLn "Enter a number"
      n <- getLine

      let lim = read n :: Int
          hf  = quot lim 2
          result = runEval $ do
            -- we split the computation in half, so we can concurrently calculate primes
            as <- rpar (force (primesBtwn 2 hf))
            bs <- rpar (force (primesBtwn (hf + 1) lim))
            return (as ++ bs)

      forkIO $ putStrLn ("🟩Primes are: " ++ (show result) ++ " for " ++ n ++ "🟩")
      loop

-- 计算两个整数之间的素数
-- 故意低效以便演示用途
primesBtwn n m = eratos [n..m]
   其中
eratos []      = []
eratos (p:xs) = p : eratos (xs `minus` [p, p+p..])
```

运行此程序将演示并发行为：

```
请输入一个数字
12
请输入一个数字

对于12，质数是: [2,3,5,7,8,9,10,11,12]

100
请输入一个数字

质数是        :
              [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,
             70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100] 对于
100

200000000
请输入一个数字
-- 正在等待 200000000
200
请输入一个数字

质数为：
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,102,103,104,105,106,107
,108,109,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,128,129,130,131,13
          2,133,134,135,136,137,138,139,140,141,142,143,144,145,146,147,148,149,150,151,152,153,154,155,156,1
          57,158,159,160,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,176,177,178,179,180,181,
182,183,184,185,186,187,188,189,190,191,192,193,194,195,196,197,198,199,200] 用于 200

-- 仍在等待 200000000
```

第 57.3 节：rseq

我们可以使用 rseq :: 策略 a 来强制一个参数进入弱头正规形态：

```
f1 :: [Int]
f1 = [1..100000000]

f2 :: [Int]
f2 = [1..000000000]

main = runEval $ do
a <- rpar (f1) -- 这将花费一些时间...
b <- rpar (f2) -- 这将花费更多时间...
  rseq a
  return (a,b)
```

这微妙地改变了 rpar 示例的语义；后者会立即返回，同时在后台计算值，而此示例将等待 a 被求值到弱头正范式（WHNF）。

```
      loop

-- Compute primes between two integers
-- Deliberately inefficient for demonstration purposes
primesBtwn n m = eratos [n..m]
  where
    eratos []     = []
    eratos (p:xs) = p : eratos (xs `minus` [p, p+p..])
```

Running this will demonstrate the concurrent behaviour:

```
Enter a number
12
Enter a number

Primes are: [2,3,5,7,8,9,10,11,12] for 12

100
Enter a number

Primes are:
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,
70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100] for
100

200000000
Enter a number
-- waiting for 200000000
200
Enter a number

Primes are:
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,102,103,104,105,106,107
,108,109,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,128,129,130,131,13
2,133,134,135,136,137,138,139,140,141,142,143,144,145,146,147,148,149,150,151,152,153,154,155,156,1
57,158,159,160,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,176,177,178,179,180,181,
182,183,184,185,186,187,188,189,190,191,192,193,194,195,196,197,198,199,200] for 200

-- still waiting for 200000000
```

Section 57.3: rseq

We can use `rseq :: Strategy a` to force an argument to Weak Head Normal Form:

```
f1 :: [Int]
f1 = [1..100000000]

f2 :: [Int]
f2 = [1..200000000]

main = runEval $ do
  a <- rpar (f1) -- this'll take a while...
  b <- rpar (f2) -- this'll take a while and then some...
  rseq a
  return (a,b)
```

This subtly changes the semantics of the `rpar` example; whereas the latter would return immediately whilst computing the values in the background, this example will wait until a can be evaluated to WHNF.

第58章：使用 taggy-lens 和 lens 解析 HTML

第58.1节：从树中筛选元素

查找 id="article" 的 div，并剥离所有内部的 script 标签。

```
#!/usr/bin/env stack
-- stack --resolver lts-7.1 --install-ghc runghc --package text --package lens --package taggy-lens -
-package string-class --package classy-prelude
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}

import ClassyPrelude
import Control.Lens hiding (children, element)
import Data.String.Class (toText, fromText, toString)
import Data.Text (Text)
import Text.Taggy.Lens
import qualified Text.Taggy.Lens as Taggy
import qualified Text.Taggy.Renderer as Renderer

somehtmlSmall :: Text
somehtmlSmall =
    "<!doctype html><html><body>\   █
    █div id=█article█><div>first</div><div>second</div><script>this should be
removed</script><div>third</div></div>█
    █/body></html>"

renderWithoutScriptTag :: Text
renderWithoutScriptTag =
    let mArticle :: Maybe Taggy.Element
        mArticle =
            (fromText somehtmlSmall) ^? html .
allAttributed (ix "id" . only "article")
        mArticleFiltered =
            fmap
                (transform
                    (children %~
                     filter (\n -> n ^? element . name /= Just "script")))
                 mArticle
    in maybe "" (toText . Renderer.render) mArticleFiltered

main :: IO ()
main = print renderWithoutScriptTag
-- 输出：
-- "<div id=\"article\"><div>first</div><div>second</div><div>third</div></div>"
```

基于 @duplode 的 SO 答案的贡献

第 58.2 节：从具有特定 id 的 div 中提取文本内容

Taggy-lens 允许我们使用透镜来解析和检查 HTML 文档。

```
#!/usr/bin/env stack
-- stack --resolver lts-7.0 --install-ghc runghc --package text --package lens --package taggy-lens
```

Chapter 58: Parsing HTML with taggy-lens and lens

Section 58.1: Filtering elements from the tree

Find div with **id**="article" and strip out all the inner script tags.

```
#!/usr/bin/env stack
-- stack --resolver lts-7.1 --install-ghc runghc --package text --package lens --package taggy-lens -
-package string-class --package classy-prelude
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}

import ClassyPrelude
import Control.Lens hiding (children, element)
import Data.String.Class (toText, fromText, toString)
import Data.Text (Text)
import Text.Taggy.Lens
import qualified Text.Taggy.Lens as Taggy
import qualified Text.Taggy.Renderer as Renderer

somehtmlSmall :: Text
somehtmlSmall =
    "<!doctype html><html><body>█
    █div id=█article█><div>first</div><div>second</div><script>this should be
removed</script><div>third</div></div>█
    █/body></html>"

renderWithoutScriptTag :: Text
renderWithoutScriptTag =
    let mArticle :: Maybe Taggy.Element
        mArticle =
            (fromText somehtmlSmall) ^? html .
            allAttributed (ix "id" . only "article")
        mArticleFiltered =
            fmap
                (transform
                    (children %~
                     filter (\n -> n ^? element . name /= Just "script")))
                mArticle
    in maybe "" (toText . Renderer.render) mArticleFiltered

main :: IO ()
main = print renderWithoutScriptTag
-- outputs:
-- "<div id=\"article\"><div>first</div><div>second</div><div>third</div></div>"
```

Contribution based upon @duplode's SO answer

Section 58.2: Extract the text contents from a div with a particular id

Taggy-lens allows us to use lenses to parse and inspect HTML documents.

```
#!/usr/bin/env stack
-- stack --resolver lts-7.0 --install-ghc runghc --package text --package lens --package taggy-lens
```

```haskell
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text.Lazy as TL
import qualified Data.Text.IO as T
import Text.Taggy.Lens
import Control.Lens

someHtml :: TL.Text
someHtml =
    "\█
    ██doctype html><html><body>\   █
    ██div>第一个 div</div>\   █
    ██div id=██thediv██>第二个 div</div>\   █
    ██div id=██not-thediv██>第三个 div</div>"

main :: IO ()
main = do
T.putStrLn
        (someHtml ^. html . allAttributed (ix "id" . only "thediv") . contents)
```

```haskell
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text.Lazy as TL
import qualified Data.Text.IO as T
import Text.Taggy.Lens
import Control.Lens

someHtml :: TL.Text
someHtml =
    "█
    ██!doctype html><html><body>█
    ██div>first div</div>█
    ██div id=██thediv██>second div</div>█
    ██div id=██not-thediv██>third div</div>"

main :: IO ()
main = do
    T.putStrLn
        (someHtml ^. html . allAttributed (ix "id" . only "thediv") . contents)
```

第59章：外部函数接口

第59.1节：从 Haskell 调用 C

出于性能考虑，或者由于存在成熟的C库，你可能希望从Haskell程序中调用C代码。这里有一个简单的示例，展示如何将数据传递给C库并获取返回结果。

foo.c：

```
#include <inttypes.h>

int32_t foo(int32_t a) {
  return a+1;
}
```

Foo.hs：

```
import Data.Int

main :: IO ()
main = print =<< hFoo 41

foreign import ccall unsafe "foo" hFoo :: Int32 -> IO Int32
```

关键字 unsafe 生成比 'safe' 更高效的调用，但要求C代码绝不回调Haskell系统。由于 foo 完全是C代码且永远不会调用Haskell，我们可以使用 unsafe。

我们还需要指示cabal编译并链接C源代码。

foo.cabal：

```
名称:                  foo
版本:                0.0.0.1
构建-类型:           简单
额外-源-文件:  *.c
cabal-版本:          >= 1.10

可执行文件 foo
default-语言: Haskell2010
  main-是:        Foo.hs
C-源文件:      foo.c
  构建-依赖: base
```

然后你可以运行：

```
> cabal configure
> cabal build foo
> ./dist/build/foo/foo
42
```

第59.2节：将Haskell函数作为回调传递给C代码

C函数接受指向其他函数的指针作为参数是非常常见的。最常见的例子是在某些GUI工具库中设置按钮点击时执行的操作。可以将Haskell函数作为C回调传递。

Chapter 59: Foreign Function Interface

Section 59.1: Calling C from Haskell

For performance reasons, or due to the existence of mature C libraries, you may want to call C code from a Haskell program. Here is a simple example of how you can pass data to a C library and get an answer back.

foo.c:

```
#include <inttypes.h>

int32_t foo(int32_t a) {
  return a+1;
}
```

Foo.hs:

```
import Data.Int

main :: IO ()
main = print =<< hFoo 41

foreign import ccall unsafe "foo" hFoo :: Int32 -> IO Int32
```

The `unsafe` keyword generates a more efficient call than 'safe', but requires that the C code never makes a callback to the Haskell system. Since `foo` is completely in C and will never call Haskell, we can use `unsafe`.

We also need to instruct cabal to compile and link in C source.

foo.cabal:

```
name:                 foo
version:              0.0.0.1
build-type:           Simple
extra-source-files:   *.c
cabal-version:        >= 1.10

executable foo
  default-language: Haskell2010
  main-is:       Foo.hs
  C-sources:     foo.c
  build-depends: base
```

Then you can run:

```
> cabal configure
> cabal build foo
> ./dist/build/foo/foo
42
```

Section 59.2: Passing Haskell functions as callbacks to C code

It is very common for C functions to accept pointers to other functions as arguments. Most popular example is setting an action to be executed when a button is clicked in some GUI toolkit library. It is possible to pass Haskell functions as C callbacks.

调用此C函数：

```
void event_callback_add (Object *obj, Object_Event_Cb func, const void *data)
```

我们首先将其导入到 Haskell 代码中：

```
foreign import ccall "header.h event_callback_add"
    callbackAdd :: Ptr () -> FunPtr Callback -> Ptr () -> IO ()
```

现在查看 C 头文件中 Object_Event_Cb 的定义，定义 Haskell 中的 Callback 类型：

```
type Callback = Ptr () -> Ptr () -> IO ()
```

最后，创建一个特殊函数，将类型为 Callback 的 Haskell 函数封装成指针 FunPtr Callback：

```
foreign import ccall "wrapper"
    mkCallback :: Callback -> IO (FunPtr Callback)
```

现在我们可以向 C 代码注册回调函数了：

```
cbPtr <- mkCallback $ \objPtr dataPtr -> do
    -- 回调代码
    return ()
callbackAdd cpPtr
```

注销回调后，释放已分配的FunPtr非常重要：

```
freeHaskellFunPtr cbPtr
```

To call this C function:

```
void event_callback_add (Object *obj, Object_Event_Cb func, const void *data)
```

we first import it to Haskell code:

```
foreign import ccall "header.h event_callback_add"
    callbackAdd :: Ptr () -> FunPtr Callback -> Ptr () -> IO ()
```

Now looking at how `Object_Event_Cb` is defined in C header, define what `Callback` is in Haskell:

```
type Callback = Ptr () -> Ptr () -> IO ()
```

Finally, create a special function that would wrap Haskell function of type `Callback` into a pointer `FunPtr Callback`:

```
foreign import ccall "wrapper"
    mkCallback :: Callback -> IO (FunPtr Callback)
```

Now we can register callback with C code:

```
cbPtr <- mkCallback $ \objPtr dataPtr -> do
    -- callback code
    return ()
callbackAdd cpPtr
```

It is important to free allocated `FunPtr` once you unregister the callback:

```
freeHaskellFunPtr cbPtr
```

第60章：Gtk3

第60.1节：Gtk中的Hello World

本示例展示了如何在Gtk3中创建一个简单的“Hello World”，设置一个窗口和按钮控件。
示例代码还将演示如何设置控件的不同属性和操作。

```
module Main (Main.main) where

import Graphics.UI.Gtk

main :: IO ()
main = do
  initGUI
window <- windowNew
  on window objectDestroy mainQuit
set window [ containerBorderWidth := 10, windowTitle := "Hello World" ]
  button <- buttonNew
设置按钮 [ buttonLabel := "Hello World" ]
 当按钮 buttonActivated $ 执行
    putStrLn "A █clicked█-处理程序，输出 \"d█troy\""█
   销毁窗口 widgetDestroy window
设置窗口 [ containerChild := button ]
 显示所有控件 widgetShowAll window
mainGUI -- 主循环
```

Chapter 60: Gtk3

Section 60.1: Hello World in Gtk

This example show how one may create a simple "Hello World" in Gtk3, setting up a window and button widgets. The sample code will also demonstrate how to set different attributes and actions on the widgets.

```
module Main (Main.main) where

import Graphics.UI.Gtk

main :: IO ()
main = do
  initGUI
  window <- windowNew
  on window objectDestroy mainQuit
  set window [ containerBorderWidth := 10, windowTitle := "Hello World" ]
  button <- buttonNew
  set button [ buttonLabel := "Hello World" ]
  on button buttonActivated $ do
    putStrLn "A █clicked█-handler to say █destroy█"
    widgetDestroy window
  set window [ containerChild := button ]
  widgetShowAll window
  mainGUI -- main loop
```

第61章：Monad变换器

第61.1节：一个单子计数器

一个关于如何使用monad变换器组合reader、writer和state monad的示例。源代码
可以在此仓库中找到

我们想实现一个计数器，其值按给定常数递增。

我们从定义一些类型和函数开始：

```
newtype 计数器 = MkCounter {cValue :: Int}派生 (Sh
   ow)

-- | 'inc c n' 将计数器增加 'n' 单位。
inc :: 计数器 -> 整数 -> 计数器
inc (MkCounter c) n = MkCounter (c + n)
```

假设我们想使用计数器执行以下计算：

- 将计数器设置为0
- 将增量常数设置为3
- 将计数器增加3次
- 将增量常数设置为5
- 将计数器增加2次

状态单子（state monad）提供了传递状态的抽象。我们可以利用状态单子，将我们的增量函数定义为状态转换器。

```
-- | CounterS 是一个单子。
类型 CounterS = 状态 计数器

-- | 将计数器增加'n'单位。
incS :: 整数-> CounterS ()
incS n = 修改 (\c -> inc c n)
```

这已经使我们能够以更清晰简洁的方式表达计算：

```
-- | 我们想运行的计算，使用状态单子。
mComputationS :: CounterS ()
mComputationS = do
incS 3
  incS 3
  incS 3
  incS 5
  incS 5
```

但我们仍然需要在每次调用时传递增量常数。我们希望避免这样做。

添加环境

reader monad（读取器单子）提供了一种方便的方式来传递环境。该单子在函数式编程中用于实现面向对象世界中称为依赖注入的功能。

在其最简单的版本中，读取器单子需要两种类型：

Chapter 61: Monad Transformers

Section 61.1: A monadic counter

An example on how to compose the reader, writer, and state monad using monad transformers. The source code can be found in this repository

We want to implement a counter, that increments its value by a given constant.

We start by defining some types, and functions:

```
newtype Counter = MkCounter {cValue :: Int}
  deriving (Show)

-- | 'inc c n' increments the counter by 'n' units.
inc :: Counter -> Int -> Counter
inc (MkCounter c) n = MkCounter (c + n)
```

Assume we want to carry out the following computation using the counter:

- set the counter to 0
- set the increment constant to 3
- increment the counter 3 times
- set the increment constant to 5
- increment the counter 2 times

The state monad provides abstractions for passing state around. We can make use of the state monad, and define our increment function as a state transformer.

```
-- | CounterS is a monad.
type CounterS = State Counter

-- | Increment the counter by 'n' units.
incS :: Int-> CounterS ()
incS n = modify (\c -> inc c n)
```

This already enables us to express a computation in a more clear and succinct way:

```
-- | The computation we want to run, with the state monad.
mComputationS :: CounterS ()
mComputationS = do
  incS 3
  incS 3
  incS 3
  incS 5
  incS 5
```

But we still have to pass the increment constant at each invocation. We would like to avoid this.

Adding an environment

The reader monad provides a convenient way to pass an environment around. This monad is used in functional programming to perform what in the OO world is known as *dependency injection*.

In its simplest version, the reader monad requires two types:

- 被读取的值的类型（即我们的环境，下面的 r ），
- 读取器单子返回的值（下面的 a ）。

Reader r a

然而，我们还需要使用状态单子。因此，我们需要使用ReaderT转换器：

```
newtype ReaderT r m a :: * -> (* -> *) -> * -> *
```

使用ReaderT，我们可以如下定义带有环境和状态的计数器：

```
type CounterRS = ReaderT Int CounterS
```

我们定义了一个incR函数，该函数从环境中获取增量常数（使用ask），并且为了用我们的CounterS单子定义增量函数，我们使用了属于单子变换器monad transformer类的lift函数。

```
-- | 按照环境指定的单位数增加计数器。
incR :: CounterRS ()
incR = ask >>= lift . incS
```

使用reader单子，我们可以如下定义计算：

```
-- | 我们想要运行的计算，使用reader和state单子。
mComputationRS :: CounterRS ()
mComputationRS = do
local (const 3) $ do
    incR
     incR
     incR
local (const 5) $ do
     incR
      incR
```

需求变了：我们需要日志记录！

现在假设我们想给计算添加日志记录，这样我们就可以看到计数器随时间的变化。

我们还有一个用于执行此任务的单子，writer monad。与reader monad一样，由于我们要组合它们，我们需要使用reader monad变换器：

```
newtype WriterT w m a :: * -> (* -> *) -> * -> *
```

这里w表示要累积的输出类型（必须是一个幺半群monoid，这允许我们累积该值），m是内部单子，a是计算的类型。

然后我们可以如下定义带有日志、环境和状态的计数器：

```
type CounterWRS = WriterT [Int] CounterRS
```

利用lift，我们可以定义一个版本的递增函数，该函数在每次递增后记录计数器的值：

- the type of the value being read (i.e. our environment, r below),
- the value returned by the reader monad (a below).

Reader r a

However, we need to make use of the state monad as well. Thus, we need to use the ReaderT transformer:

```
newtype ReaderT r m a :: * -> (* -> *) -> * -> *
```

Using ReaderT, we can define our counter with environment and state as follows:

```
type CounterRS = ReaderT Int CounterS
```

We define an incR function that takes the increment constant from the environment (using ask), and to define our increment function in terms of our CounterS monad we make use of the lift function (which belongs to the monad transformer class).

```
-- | Increment the counter by the amount of units specified by the environment.
incR :: CounterRS ()
incR = ask >>= lift . incS
```

Using the reader monad we can define our computation as follows:

```
-- | The computation we want to run, using reader and state monads.
mComputationRS :: CounterRS ()
mComputationRS = do
  local (const 3) $ do
    incR
    incR
    incR
    local (const 5) $ do
      incR
      incR
```

The requirements changed: we need logging!

Now assume that we want to add logging to our computation, so that we can see the evolution of our counter in time.

We also have a monad to perform this task, the writer monad. As with the reader monad, since we are composing them, we need to make use of the reader monad transformer:

```
newtype WriterT w m a :: * -> (* -> *) -> * -> *
```

Here w represents the type of the output to accumulate (which has to be a monoid, which allow us to accumulate this value), m is the inner monad, and a the type of the computation.

We can then define our counter with logging, environment, and state as follows:

```
type CounterWRS = WriterT [Int] CounterRS
```

And making use of lift we can define the version of the increment function which logs the value of the counter after each increment:

```
incW :: CounterWRS ()
incW = lift incR >> get >>= tell . (:[]) . cValue
```

现在包含日志记录的计算可以写成如下形式：

```
mComputationWRS :: CounterWRS ()
mComputationWRS = do
local (const 3) $ do
    incW
     incW
     incW
local (const 5) $ do
      incW
        incW
```

一次性完成所有操作

这个例子旨在展示单子变换器的工作原理。然而，我们可以通过在单个递增操作中组合所有方面（环境、状态和日志记录）来实现相同的效果。

为此，我们利用类型约束：

```
inc' :: (MonadReader Int m, MonadState Counter m, MonadWriter [Int] m) => m ()
inc' = ask >>= modify . (flip inc) >> get >>= tell . (:[]) . cValue
```

这里我们得出一个适用于满足上述约束的任何单子的解决方案。计算函数定义如下，类型为：

```
mComputation' :: (MonadReader Int m, MonadState Counter m, MonadWriter [Int] m) => m ()
```

因为在其主体中我们使用了 inc'。

例如，我们可以在ghci REPL中运行此计算，方法如下：

```
runState ( runReaderT ( runWriterT mComputation' ) 15 )  (MkCounter 0)
```

```
incW :: CounterWRS ()
incW = lift incR >> get >>= tell . (:[]) . cValue
```

Now the computation that contains logging can be written as follows:

```
mComputationWRS :: CounterWRS ()
mComputationWRS = do
  local (const 3) $ do
    incW
    incW
    incW
    local (const 5) $ do
      incW
      incW
```

Doing everything in one go

This example intended to show monad transformers at work. However, we can achieve the same effect by composing all the aspects (environment, state, and logging) in a single increment operation.

To do this we make use of type-constraints:

```
inc' :: (MonadReader Int m, MonadState Counter m, MonadWriter [Int] m) => m ()
inc' = ask >>= modify . (flip inc) >> get >>= tell . (:[]) . cValue
```

Here we arrive at a solution that will work for any monad that satisfies the constraints above. The computation function is defined thus with type:

```
mComputation' :: (MonadReader Int m, MonadState Counter m, MonadWriter [Int] m) => m ()
```

since in its body we make use of inc'.

We could run this computation, in the `ghci` REPL for instance, as follows:

```
runState ( runReaderT ( runWriterT mComputation' ) 15 )  (MkCounter 0)
```

第62章：双函子

第62.1节：双函子的定义

双函子 是具有两个类型参数的类型类（f :: * -> * -> *），这两个参数都可以协变地同时映射。

```
类 Bifunctor f 其中
bimap :: (a -> c) -> (b -> d) -> f a b -> f c d
```

bimap 可以被看作是对数据类型应用一对 fmap 操作。

类型 f 的正确 Bifunctor 实例必须满足 bifunctor 定律，这些定律类似于 functor 定律：

```
bimap id id = id  -- 恒等律
bimap (f . g) (h . i) = bimap f h . bimap g i  -- 复合律
```

Bifunctor 类位于 Data.Bifunctor 模块中。对于 GHC 版本 >7.10，该模块随编译器捆绑；对于早期版本，需要安装 bifunctors 包。

第62.2节：Bifunctor 的常见实例

二元组

(,) 是具有 Bifunctor 实例的类型示例。

```
instance Bifunctor (,) where
    bimap f g (x, y) = (f x, g y)
```

bimap 接受一对函数并将它们分别应用于元组的对应分量。

```
bimap (+ 2) (++ "nie") (3, "john") --> (5,"johnnie")
bimap ceiling length (3.5 :: Double, "john" :: String) --> (4,4)
```

Either

Either的 Bifunctor 实例根据值是 Left 还是

Right 来选择应用两个函数中的一个。

```
instance Bifunctor Either where
    bimap f g (Left x) = Left (f x)
    bimap f g (Right y) = Right (g y)
```

第62.3节：first 和 second

如果只想对第一个参数或第二个参数进行协变映射，则应使用 first 或 second（代替 bimap）。

```
first :: Bifunctor f => (a -> c) -> f a b -> f c b
first f = bimap f id

second :: Bifunctor f => (b -> d) -> f a b -> f a d
second g = bimap id g
```

例如，

Chapter 62: Bifunctor

Section 62.1: Definition of Bifunctor

Bifunctor is the class of types with two type parameters (f :: * -> * -> *), both of which can be covariantly mapped over simultaneously.

```
class Bifunctor f where
    bimap :: (a -> c) -> (b -> d) -> f a b -> f c d
```

bimap can be thought of as applying a pair of **fmap** operations to a datatype.

A correct instance of Bifunctor for a type f must satisfy the *bifunctor laws*, which are analogous to the *functor laws*:

```
bimap id id = id  -- identity
bimap (f . g) (h . i) = bimap f h . bimap g i  -- composition
```

The Bifunctor class is found in the Data.Bifunctor module. For GHC versions >7.10, this module is bundled with the compiler; for earlier versions you need to install the bifunctors package.

Section 62.2: Common instances of Bifunctor

Two-element tuples

(,) is an example of a type that has a Bifunctor instance.

```
instance Bifunctor (,) where
    bimap f g (x, y) = (f x, g y)
```

bimap takes a pair of functions and applies them to the tuple's respective components.

```
bimap (+ 2) (++ "nie") (3, "john") --> (5,"johnnie")
bimap ceiling length (3.5 :: Double, "john" :: String) --> (4,4)
```

Either

Either's instance of Bifunctor selects one of the two functions to apply depending on whether the value is Left or Right.

```
instance Bifunctor Either where
    bimap f g (Left x) = Left (f x)
    bimap f g (Right y) = Right (g y)
```

Section 62.3: first and second

If mapping covariantly over only the first argument, or only the second argument, is desired, then first or second ought to be used (in lieu of bimap).

```
first :: Bifunctor f => (a -> c) -> f a b -> f c b
first f = bimap f id

second :: Bifunctor f => (b -> d) -> f a b -> f a d
second g = bimap id g
```

For example,

```
ghci> second (+ 2) (Right 40)
Right 42
ghci> second (+ 2) (Left "uh oh")
Left "uh oh"
```

```
ghci> second (+ 2) (Right 40)
Right 42
ghci> second (+ 2) (Left "uh oh")
Left "uh oh"
```

第63章：代理

第63.1节：使用代理（Proxy）

在 Data.Proxy 中发现的 Proxy :: k -> * 类型，当你需要向编译器提供一些类型信息（例如，选择一个类型类实例），但这些信息在运行时无关紧要时使用。

```
{-# LANGUAGE PolyKinds #-}

数据 Proxy a = Proxy
```

使用 Proxy 的函数通常使用 ScopedTypeVariables 来基于 a 类型选择一个类型类实例。

例如，一个经典的模糊函数示例，

```
showread :: String -> String
showread = show . read
```

由于展开器不知道使用哪个 Show 或 Read 的实例，导致类型错误，可以使用 Proxy 来解决：

```
{-# LANGUAGE ScopedTypeVariables #-}

导入 Data.Proxy

showread :: forall a. (Show a, Read a) => Proxy a -> String -> String
showread _ = (show :: a -> String) . read
```

调用带有Proxy的函数时，需要使用类型注解来声明你所指的 a。

```
ghci> showread (Proxy :: Proxy Int) "3"
"3"
ghci> showread (Proxy :: Proxy Bool) "'m'"  -- 尝试将字符字面量解析为Bool类型
"*** Exception: Prelude.read: no parse
```

第63.2节："多态代理"（polymorphic proxy）惯用法

由于Proxy不包含运行时信息，因此永远不需要对Proxy构造函数进行模式匹配。所以一个常见的惯用法是使用类型变量对Proxy数据类型进行抽象。

```
showread :: forall proxy a. (Show a, Read a) => proxy a -> String -> String
showread _ = (show :: a -> String) . read
```

现在，如果你恰好在作用域中有某个 f a，对于某个 f，调用 f时就不需要写出Proxy :: Proxy a。

```
ghci> let chars = "foo"  -- chars :: [Char]
ghci> showread chars "'a'"
"'a'"
```

第63.3节：代理就像 ()

由于Proxy不包含运行时信息，你总是可以为任意的f编写一个自然变换f a -> Proxy a。

Chapter 63: Proxies

Section 63.1: Using Proxy

The Proxy :: k -> * type, found in Data.Proxy, is used when you need to give the compiler some type information - eg, to pick a type class instance - which is nonetheless irrelevant at runtime.

```
{-# LANGUAGE PolyKinds #-}

data Proxy a = Proxy
```

Functions which use a Proxy typically use ScopedTypeVariables to pick a type class instance based on the a type.

For example, the classic example of an ambiguous function,

```
showread :: String -> String
showread = show . read
```

which results in a type error because the elaborator doesn't know which instance of Show or Read to use, can be resolved using Proxy:

```
{-# LANGUAGE ScopedTypeVariables #-}

import Data.Proxy

showread :: forall a. (Show a, Read a) => Proxy a -> String -> String
showread _ = (show :: a -> String) . read
```

When calling a function with Proxy, you need to use a type annotation to declare which a you meant.

```
ghci> showread (Proxy :: Proxy Int) "3"
"3"
ghci> showread (Proxy :: Proxy Bool) "'m'"  -- attempt to parse a char literal as a Bool
"*** Exception: Prelude.read: no parse
```

Section 63.2: The "polymorphic proxy" idiom

Since Proxy contains no runtime information, there is never a need to pattern-match on the Proxy constructor. So a common idiom is to abstract over the Proxy datatype using a type variable.

```
showread :: forall proxy a. (Show a, Read a) => proxy a -> String -> String
showread _ = (show :: a -> String) . read
```

Now, if you happen to have an f a in scope for some f, you don't need to write out Proxy :: Proxy a when calling f.

```
ghci> let chars = "foo"  -- chars :: [Char]
ghci> showread chars "'a'"
"'a'"
```

Section 63.3: Proxy is like ()

Since Proxy contains no runtime information, you can always write a natural transformation f a -> Proxy a for any f.

```
proxy :: f a -> Proxy a
proxy _ = Proxy
```

这就像任何给定的值总是可以被擦除为()一样：

```
unit :: a -> ()
unit _ = ()
```

从技术上讲，Proxy是函子范畴中的终对象，就像()是值范畴中的终对象一样。

```
proxy :: f a -> Proxy a
proxy _ = Proxy
```

This is just like how any given value can always be erased to `()`:

```
unit :: a -> ()
unit _ = ()
```

Technically, `Proxy` is the terminal object in the category of functors, just like `()` is the terminal object in the category of values.

第64章：应用函子（Applicative Functor）

Applicative是类型类，类型f :: * -> *允许在一个结构上进行提升的函数应用，其中函数本身也嵌入在该结构中。

第64.1节：替代定义

由于每个应用函子都是函子，fmap总是可以用于它；因此Applicative的本质是携带内容的配对，以及创建它的能力：

```
类 Functor f => PairingFunctor f 其中
funit :: f ()                   -- 创建一个上下文，不携带任何重要内容
  fpair :: (f a,f b) -> f (a,b)   -- 将一对上下文合并成携带对的上下文
```

该类与 Applicative 同构。

```
pure a = const a <$> funit = a <$ funit

fa <*> fb = (\(a,b) -> a b) <$> fpair (fa, fb) = uncurry ($) <$> fpair (fa, fb)
```

反过来，

```
funit = pure ()

fpair (fa, fb) = (,) <$> fa <*> fb
```

第64.2节：Applicative的常见实例

Maybe

Maybe 是一个包含可能缺失值的应用函子。

```
实例 Applicative Maybe 其中
    pure = Just

仅仅是 f <*> 仅仅是 x = 仅仅是 $ f x
    _ <*> _ = 无
```

pure通过对给定值应用Just将其提升为Maybe。函数(<*>)将包装在Maybe中的函数应用于Maybe中的值。如果函数和值都存在（由Just构造），则函数将应用于该值并返回包装后的结果。如果任一缺失，计算无法继续，返回Nothing。

列表

使列表符合类型签名<*> :: [a -> b] -> [a] -> [b]的一种方法是取两个列表的笛卡尔积，将第一个列表的每个元素与第二个列表的每个元素配对：

```
fs <*> xs = [f x | f <- fs, x <- xs]
         -- = do { f <- fs; x <- xs; return (f x) }

pure x = [x]
```

这通常被解释为模拟非确定性，其中一个值列表代表一个非确定性值

Chapter 64: Applicative Functor

Applicative is the class of types f :: * -> * which allows lifted function application over a structure where the function is also embedded in that structure.

Section 64.1: Alternative definition

Since every Applicative Functor is a Functor, **fmap** can always be used on it; thus the essence of Applicative is the pairing of carried contents, as well as the ability to create it:

```
class Functor f => PairingFunctor f where
  funit :: f ()                  -- create a context, carrying nothing of import
  fpair :: (f a,f b) -> f (a,b)  -- collapse a pair of contexts into a pair-carrying context
```

This class is isomorphic to Applicative.

```
pure a = const a <$> funit = a <$ funit

fa <*> fb = (\(a,b) -> a b) <$> fpair (fa, fb) = uncurry ($) <$> fpair (fa, fb)
```

Conversely,

```
funit = pure ()

fpair (fa, fb) = (,) <$> fa <*> fb
```

Section 64.2: Common instances of Applicative

Maybe

Maybe is an applicative functor containing a possibly-absent value.

```
instance Applicative Maybe where
    pure = Just

    Just f <*> Just x = Just $ f x
    _ <*> _ = Nothing
```

pure lifts the given value into Maybe by applying Just to it. The (<*>) function applies a function wrapped in a Maybe to a value in a Maybe. If both the function and the value are present (constructed with Just), the function is applied to the value and the wrapped result is returned. If either is missing, the computation can't proceed and Nothing is returned instead.

Lists

One way for lists to fit the type signature <*> :: [a -> b] -> [a] -> [b] is to take the two lists' Cartesian product, pairing up each element of the first list with each element of the second one:

```
fs <*> xs = [f x | f <- fs, x <- xs]
         -- = do { f <- fs; x <- xs; return (f x) }

pure x = [x]
```

This is usually interpreted as emulating nondeterminism, with a list of values standing for a nondeterministic value

其可能的取值范围涵盖该列表；因此，两个非确定性值的组合涵盖两个列表中所有可能值的组合：

```
ghci> [(+1),(+2)] <*> [3,30,300]
[4,31,301,5,32,302]
```

无限流和拉链列表

有一类Applicative可以将它们的两个输入“拉链”在一起。一个简单的例子是无限
流：

```
data Stream a = Stream { headS :: a, tailS :: Stream a }
```

流的 Applicative 实例将函数流逐点应用于参数流，通过位置配对两个流中的值。pure 返回一个常量流——一个单一固定值的无限列表：

```
实例 Applicative 流 其中
    pure x = 让 s = 流 x s 在 s
    流 f fs <*> 流 x xs = 流 (\f x) (fs <*> xs)
```

列表也支持“拉链式”的 Applicative 实例，对应 ZipList 新类型：

```
newtype ZipList a = ZipList { getZipList :: [a] }

instance Applicative ZipList where
ZipList xs <*> ZipList ys = ZipList $ zipWith ($) xs ys
```

由于 zip 会根据最短输入裁剪结果，唯一满足 Applicative 定律的 pure 实现是返回一个无限列表：

```
pure a = ZipList (repeat a)   -- ZipList (fix (a:)) = ZipList [a,a,a,a,...
```

例如：

```
ghci> getZipList $ ZipList [ (+1), (+2) ] <*> ZipList [3, 30, 300]
[4, 32]
```

这两种可能让我们联想到外积和内积，类似于第一种情况中将一个 1 列（n x 1）矩阵与一个 1 行（1 x m）矩阵相乘，得到 n x m 矩阵（但被展平）；第二种情况则是将 1 行和 1 列矩阵相乘（但不求和）。

函数

当专门化为函数 (->) r 时，pure 和 <*> 的类型签名分别对应 K 和 S 组合子：

```
pure :: a -> (r -> a)
<*> :: (r -> (a -> b)) -> (r -> a) -> (r -> b)
```

pure 必须是 **const**，且 <*> 接受一对函数并将它们分别应用于一个固定参数，应用两个
结果：

```
instance Applicative ((->) r) where
    pure = const
f <*> g = \x -> f x (g x)
```

whose possible values range over that list; so a combination of two nondeterministic values ranges over all possible combinations of the values in the two lists:

```
ghci> [(+1),(+2)] <*> [3,30,300]
[4,31,301,5,32,302]
```

Infinite streams and zip-lists

There's a class of Applicatives which "zip" their two inputs together. One simple example is that of infinite streams:

```
data Stream a = Stream { headS :: a, tailS :: Stream a }
```

Stream's Applicative instance applies a stream of functions to a stream of arguments point-wise, pairing up the values in the two streams by position. pure returns a constant stream – an infinite list of a single fixed value:

```
instance Applicative Stream where
    pure x = let s = Stream x s in s
    Stream f fs <*> Stream x xs = Stream (f x) (fs <*> xs)
```

Lists too admit a "zippy" Applicative instance, for which there exists the ZipList newtype:

```
newtype ZipList a = ZipList { getZipList :: [a] }

instance Applicative ZipList where
    ZipList xs <*> ZipList ys = ZipList $ zipWith ($) xs ys
```

Since **zip** trims its result according to the shortest input, the only implementation of pure that satisfies the Applicative laws is one which returns an infinite list:

```
    pure a = ZipList (repeat a)   -- ZipList (fix (a:)) = ZipList [a,a,a,a,...
```

For example:

```
ghci> getZipList $ ZipList [(+1),(+2)] <*> ZipList [3,30,300]
[4,32]
```

The two possibilities remind us of the outer and the inner product, similar to multiplying a 1-column (n x 1) matrix with a 1-row (1 x m) one in the first case, getting the n x m matrix as a result (but flattened); or multiplying a 1-row and a 1-column matrices (but without the summing up) in the second case.

Functions

When specialised to functions (->) r, the type signatures of pure and <*> match those of the K and S combinators, respectively:

```
pure :: a -> (r -> a)
<*> :: (r -> (a -> b)) -> (r -> a) -> (r -> b)
```

pure must be **const**, and <*> takes a pair of functions and applies them each to a fixed argument, applying the two results:

```
instance Applicative ((->) r) where
    pure = const
    f <*> g = \x -> f x (g x)
```

函数是典型的“拉链式”应用子。例如，由于无限流与 (->) 自然数 同构，...

```
-- | 索引流中的元素
to :: Stream a -> (Nat -> a)
to (Stream x xs) Zero = x
to (Stream x xs) (Suc n) = to xs n

-- | 按顺序列出函数的所有返回值
from :: (Nat -> a) -> Stream a
from f = from' Zero
    where from' n = Stream (f n) (from' (Suc n))
```

...以更高阶的方式表示流会自动生成高效的Applicative实例。

Functions are the prototypical "zippy" applicative. For example, since infinite streams are isomorphic to `(->) Nat`, ...

```
-- | Index into a stream
to :: Stream a -> (Nat -> a)
to (Stream x xs) Zero = x
to (Stream x xs) (Suc n) = to xs n

-- | List all the return values of the function in order
from :: (Nat -> a) -> Stream a
from f = from' Zero
    where from' n = Stream (f n) (from' (Suc n))
```

... representing streams in a higher-order way produces the zippy `Applicative` instance automatically.

第65章：作为自由单子的常见单子

第65.1节：自由的 Empty ~~ Identity

给定

```
数据类型 Empty a
```

我们有

```
数据类型 Free Empty a
      = Pure a
-- Free 构造函数是不可能的！
```

它与以下类型同构

```
data 身份 a
      = 身份 a
```

第65.2节：自由恒等式 ~~ (自然数,) ~~ 写入器自然数

给定

```
data 恒等式 a = 恒等式 a
```

我们有

```
data 自由恒等式 a
      = 纯 a
      | 自由 (恒等式 (自由恒等式 a))
```

它与以下类型同构

```
data 延迟 a
      = 现在 a
      | 稍后 (延迟 a)
```

或等价于（如果你承诺先计算第一个元素）(自然数, a)，也称为写入器自然数 a，定义为

```
data 自然数 = 零 | 后继 自然数
data 写入器自然数 a = 写入器自然数 a
```

第65.3节：自由可能性 ~~ 可能性转换器 (写入器自然数)

给定

```
data 可能性 a = 仅 a
            | 无
```

我们有

```
data Free Maybe a
```

Chapter 65: Common monads as free monads

Section 65.1: Free Empty ~~ Identity

Given

```
data Empty a
```

we have

```
data Free Empty a
      = Pure a
-- the Free constructor is impossible!
```

which is isomorphic to

```
data Identity a
      = Identity a
```

Section 65.2: Free Identity ~~ (Nat,) ~~ Writer Nat

Given

```
data Identity a = Identity a
```

we have

```
data Free Identity a
      = Pure a
      | Free (Identity (Free Identity a))
```

which is isomorphic to

```
data Deferred a
      = Now a
      | Later (Deferred a)
```

or equivalently (if you promise to evaluate the fst element first) `(Nat, a)`, aka `Writer Nat a`, with

```
data Nat = Z | S Nat
data Writer Nat a = Writer Nat a
```

Section 65.3: Free Maybe ~~ MaybeT (Writer Nat)

Given

```
data Maybe a = Just a
            | Nothing
```

we have

```
data Free Maybe a
```

```
    = Pure a
    | Free (Just (Free Maybe a))
    | Free Nothing
```

这等价于

```
data Hopes a
    = Confirmed a
    | Possible (Hopes a)
    | Failed
```

或等价地（如果你承诺先评估第一个元素）(Nat, Maybe a)，也称为MaybeT (Writer Nat) a，定义为

```
data Nat = Z | S Nat
data Writer Nat a = Writer Nat a
data MaybeT (Writer Nat) a = MaybeT (Nat, Maybe a)
```

第65.4节：Free (Writer w) ~~ Writer [w]

给定

```
data Writer w a = Writer w a
```

我们有

```
data Free (Writer w) a
    = Pure a
    | Free (Writer w (Free (Writer w) a))
```

它与以下类型同构

```
data ProgLog w a
    = Done a
    | After w (ProgLog w a)
```

或者，等价地，（如果你承诺先计算对数），Writer [w] a。

第65.5节：Free (Const c) ~~ Either c

给定

```
data Const c a = Const c
```

我们有

```
data Free (Const c) a
    = Pure a
    | Free (Const c)
```

它与以下类型同构

```
data Either c a
    = Right a
    | Left c
```

```
    = Pure a
    | Free (Just (Free Maybe a))
    | Free Nothing
```

which is equivalent to

```
data Hopes a
    = Confirmed a
    | Possible (Hopes a)
    | Failed
```

or equivalently (if you promise to evaluate the fst element first) (Nat, Maybe a), aka MaybeT (Writer Nat) a with

```
data Nat = Z | S Nat
data Writer Nat a = Writer Nat a
data MaybeT (Writer Nat) a = MaybeT (Nat, Maybe a)
```

Section 65.4: Free (Writer w) ~~ Writer [w]

Given

```
data Writer w a = Writer w a
```

we have

```
data Free (Writer w) a
    = Pure a
    | Free (Writer w (Free (Writer w) a))
```

which is isomorphic to

```
data ProgLog w a
    = Done a
    | After w (ProgLog w a)
```

or, equivalently, (if you promise to evaluate the log first), Writer [w] a.

Section 65.5: Free (Const c) ~~ Either c

Given

```
data Const c a = Const c
```

we have

```
data Free (Const c) a
    = Pure a
    | Free (Const c)
```

which is isomorphic to

```
data Either c a
    = Right a
    | Left c
```

第65.6节：Free (Reader x) ~~ Reader (Stream x)

给定

```
data Reader x a = Reader (x -> a)
```

我们有

```
data Free (Reader x) a
      = Pure a
      | Free (x -> Free (Reader x) a)
```

它与以下类型同构

```
data Demand x a
      = Satisfied a
      | Hungry (x -> Demand x a)
```

或等价于 `Stream x -> a`，满足

```
data Stream x = Stream x (Stream x)
```

Section 65.6: Free (Reader x) ~~ Reader (Stream x)

Given

```
data Reader x a = Reader (x -> a)
```

we have

```
data Free (Reader x) a
      = Pure a
      | Free (x -> Free (Reader x) a)
```

which is isomorphic to

```
data Demand x a
      = Satisfied a
      | Hungry (x -> Demand x a)
```

or equivalently `Stream x -> a` with

```
data Stream x = Stream x (Stream x)
```

第66章：作为共自由余单子的基础的常见函子

第66.1节：共自由空 ~~ 空

给定

```
数据类型 Empty a
```

我们有

```
data Cofree Empty a
    --  = a :< ...  不可能！
```

第66.2节：余自由函子（常量c）~~ 写入器c

给定

```
data Const c a = Const c
```

我们有

```
data Cofree (Const c) a
     = a :< Const c
```

它与以下类型同构

```
data Writer c a = Writer c a
```

第66.3节：Cofree Identity ~~ Stream

给定

```
data 恒等式 a = 恒等式 a
```

我们有

```
data Cofree Identity a
     = a :< Identity (Cofree Identity a)
```

它与以下类型同构

```
data Stream a = Stream a (Stream a)
```

第66.4节：Cofree Maybe ~~ NonEmpty

给定

```
data 可能性 a = 仅 a
             | 无
```

我们有

Chapter 66: Common functors as the base of cofree comonads

Section 66.1: Cofree Empty ~~ Empty

Given

```
data Empty a
```

we have

```
data Cofree Empty a
   --  = a :< ...  not possible!
```

Section 66.2: Cofree (Const c) ~~ Writer c

Given

```
data Const c a = Const c
```

we have

```
data Cofree (Const c) a
     = a :< Const c
```

which is isomorphic to

```
data Writer c a = Writer c a
```

Section 66.3: Cofree Identity ~~ Stream

Given

```
data Identity a = Identity a
```

we have

```
data Cofree Identity a
     = a :< Identity (Cofree Identity a)
```

which is isomorphic to

```
data Stream a = Stream a (Stream a)
```

Section 66.4: Cofree Maybe ~~ NonEmpty

Given

```
data Maybe a = Just a
             | Nothing
```

we have

```
data Cofree Maybe a
     = a :< Just (Cofree Maybe a)
     | a :< Nothing
```

它与以下类型同构

```
data NonEmpty a
     = NECons a (NonEmpty a)
     | NESingle a
```

第66.5节：Cofree (Writer w) ~~ WriterT w Stream

给定

```
data Writer w a = Writer w a
```

我们有

```
data Cofree (Writer w) a
     = a :< (w, Cofree (Writer w) a)
```

这等价于

```
data Stream (w,a)
     = Stream (w,a) (Stream (w,a))
```

which can properly be written as WriterT w Stream with

```
data WriterT w m a = WriterT (m (w,a))
```

第66.6节：Cofree (Either e) ~~ NonEmptyT (Writer e)

给定

```
data Either e a = Left e
                | Right a
```

我们有

```
data Cofree (Either e) a
     = a :< Left e
     | a :< Right (Cofree (Either e) a)
```

它与以下类型同构

```
data Hospitable e a
     = Sorry_AllIHaveIsThis_Here'sWhy a e
     | EatThis a (Hospitable e a)
```

或者，如果你承诺只在完整结果之后评估对数，NonEmptyT(Writer e) a 与

```
data NonEmptyT (Writer e) a = NonEmptyT (e,a,[a])
```

```
data Cofree Maybe a
     = a :< Just (Cofree Maybe a)
     | a :< Nothing
```

which is isomorphic to

```
data NonEmpty a
     = NECons a (NonEmpty a)
     | NESingle a
```

Section 66.5: Cofree (Writer w) ~~ WriterT w Stream

Given

```
data Writer w a = Writer w a
```

we have

```
data Cofree (Writer w) a
     = a :< (w, Cofree (Writer w) a)
```

which is equivalent to

```
data Stream (w,a)
     = Stream (w,a) (Stream (w,a))
```

which can properly be written as `WriterT w Stream` with

```
data WriterT w m a = WriterT (m (w,a))
```

Section 66.6: Cofree (Either e) ~~ NonEmptyT (Writer e)

Given

```
data Either e a = Left e
                | Right a
```

we have

```
data Cofree (Either e) a
     = a :< Left e
     | a :< Right (Cofree (Either e) a)
```

which is isomorphic to

```
data Hospitable e a
     = Sorry_AllIHaveIsThis_Here'sWhy a e
     | EatThis a (Hospitable e a)
```

or, if you promise to only evaluate the log after the complete result, `NonEmptyT (Writer e) a` with

```
data NonEmptyT (Writer e) a = NonEmptyT (e,a,[a])
```

第66.7节：Cofree (Reader x) ~~ Moore x

给定

```
data Reader x a = Reader (x -> a)
```

我们有

```
data Cofree (Reader x) a
     = a :< (x -> Cofree (Reader x) a)
```

它与以下类型同构

```
data Plant x a
     = Plant a (x -> Plant x a)
```

又称Moore 机。

Section 66.7: Cofree (Reader x) ~~ Moore x

Given

```
data Reader x a = Reader (x -> a)
```

we have

```
data Cofree (Reader x) a
     = a :< (x -> Cofree (Reader x) a)
```

which is isomorphic to

```
data Plant x a
     = Plant a (x -> Plant x a)
```

aka Moore machine.

第67章：算术

数值类型类层次结构

Num 位于数值类型类层次结构的根部。其特征操作和一些常见实例如下所示（默认由 Prelude 加载的以及Data.Complex的实例）：

```
λ> :i Num
类 Num a 其中
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
      -- 定义于 'GHC.Num'
实例 RealFloat a => Num (Complex a) -- 定义于 'Data.Complex'
实例 Num Word -- 定义于 'GHC.Num'
实例 Num 整数 -- 定义于 'GHC.Num'
实例 Num 整型 -- 定义于 'GHC.Num'
实例 Num 浮点数 -- 定义于 'GHC.Float'
实例 Num 双精度浮点数 -- 定义于 'GHC.Float'
```

我们已经见过Fractional类，它要求Num并引入了"除法"(/)和数的倒数的概念：

```
λ> :i Fractional
类 Num a => Fractional a 其中
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
  {-# MINIMAL fromRational, (recip | (/)) #-}
      -- 定义于 'GHC.Real'
实例 RealFloat a => Fractional (Complex a) -- 定义于 'Data.Complex'
实例 Fractional Float -- 定义于 'GHC.Float'
实例 Fractional Double -- 定义于 'GHC.Float'
```

Real 类模拟了......实数。它要求Num和Ord，因此它模拟了一个有序数域。作为反例，复数*不是*有序域（即它们没有自然的排序关系）：

```
λ> :i Real
class (Num a, Ord a) => Real a where
  toRational :: a -> Rational
  {-# MINIMAL toRational #-}
      -- 定义于 'GHC.Real'
instance Real Word -- 定义于 'GHC.Real'
instance Real Integer -- 定义于 'GHC.Real'
instance Real Int -- 定义于 'GHC.Real'
instance Real Float -- 定义于 'GHC.Float'
instance Real Double -- 定义于 'GHC.Float'
```

RealFrac 表示可能被四舍五入的数字

Chapter 67: Arithmetic

The numeric typeclass hierarchy

Num sits at the root of the numeric typeclass hierarchy. Its characteristic operations and some common instances are shown below (the ones loaded by default with Prelude plus those of Data.Complex):

```
λ> :i Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
      -- Defined in 'GHC.Num'
instance RealFloat a => Num (Complex a) -- Defined in 'Data.Complex'
instance Num Word -- Defined in 'GHC.Num'
instance Num Integer -- Defined in 'GHC.Num'
instance Num Int -- Defined in 'GHC.Num'
instance Num Float -- Defined in 'GHC.Float'
instance Num Double -- Defined in 'GHC.Float'
```

We have already seen the Fractional class, which requires Num and introduces the notions of "division" (/) and reciprocal of a number:

```
λ> :i Fractional
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
  {-# MINIMAL fromRational, (recip | (/)) #-}
      -- Defined in 'GHC.Real'
instance RealFloat a => Fractional (Complex a) -- Defined in 'Data.Complex'
instance Fractional Float -- Defined in 'GHC.Float'
instance Fractional Double -- Defined in 'GHC.Float'
```

The Real class models .. the real numbers. It requires Num and Ord, therefore it models an ordered numerical field. As a counterexample, Complex numbers are *not* an ordered field (i.e. they do not possess a natural ordering relationship):

```
λ> :i Real
class (Num a, Ord a) => Real a where
  toRational :: a -> Rational
  {-# MINIMAL toRational #-}
      -- Defined in 'GHC.Real'
instance Real Word -- Defined in 'GHC.Real'
instance Real Integer -- Defined in 'GHC.Real'
instance Real Int -- Defined in 'GHC.Real'
instance Real Float -- Defined in 'GHC.Float'
instance Real Double -- Defined in 'GHC.Float'
```

RealFrac represents numbers that may be rounded

```
λ> :i RealFrac
class (Real a, Fractional a) => RealFrac a where
  properFraction :: Integral b => a -> (b, a)
  truncate :: Integral b => a -> b
  round :: Integral b => a -> b
  ceiling :: Integral b => a -> b
  floor :: Integral b => a -> b
  {-# MINIMAL properFraction #-}
      -- 定义于 ‘GHC.Real’
instance RealFrac Float -- 定义于 ‘GHC.Float’
instance RealFrac Double -- 定义于 ‘GHC.Float’
```

Floating（隐含 Fractional）表示可能没有有限小数展开的常数和运算。

```
λ> :i Floating
class Fractional a => Floating a where
  pi :: a
  exp :: a -> a
  log :: a -> a
  sqrt :: a -> a
  (**) :: a -> a -> a
  logBase :: a -> a -> a
  sin :: a -> a
  cos :: a -> a
  tan :: a -> a
  asin :: a -> a
  acos :: a -> a
  atan :: a -> a
  sinh :: a -> a
  cosh :: a -> a
  tanh :: a -> a
  asinh :: a -> a
  acosh :: a -> a
  atanh :: a -> a
GHC.Float.log1p :: a -> a
GHC.Float.expm1 :: a -> a
GHC.Float.log1pexp :: a -> a
GHC.Float.log1mexp :: a -> a
  {-# MINIMAL pi, exp, log, sin, cos, asin, acos, atan, sinh, cosh,
              asinh, acosh, atanh #-}
      -- 定义于 ‘GHC.Float’
instance RealFloat a => Floating (Complex a) -- 定义于 ‘Data.Complex’
instance Floating Float -- 定义于 ‘GHC.Float’
instance Floating Double -- 定义于 ‘GHC.Float’
```

注意：虽然诸如 `sqrt . negate :: Floating a => a -> a` 这样的表达式完全有效，但它们可能返回 `NaN`（“非数”），这可能不是预期的行为。在这种情况下，我们可能需要在复数域（稍后展示）上进行操作。

在Haskell中，所有表达式（包括数值常量和作用于它们的函数）都有一个可判定的类型。编译时，类型检查器根据组成表达式的基本函数的类型推断表达式的类型。由于数据默认是不可变的，因此没有“类型转换”操作，但存在复制数据并在合理范围内泛化或特化类型的函数。

```
λ> :i RealFrac
class (Real a, Fractional a) => RealFrac a where
  properFraction :: Integral b => a -> (b, a)
  truncate :: Integral b => a -> b
  round :: Integral b => a -> b
  ceiling :: Integral b => a -> b
  floor :: Integral b => a -> b
  {-# MINIMAL properFraction #-}
      -- Defined in ‘GHC.Real’
instance RealFrac Float -- Defined in ‘GHC.Float’
instance RealFrac Double -- Defined in ‘GHC.Float’
```

`Floating` (which implies `Fractional`) represents constants and operations that may not have a finite decimal expansion.

```
λ> :i Floating
class Fractional a => Floating a where
  pi :: a
  exp :: a -> a
  log :: a -> a
  sqrt :: a -> a
  (**) :: a -> a -> a
  logBase :: a -> a -> a
  sin :: a -> a
  cos :: a -> a
  tan :: a -> a
  asin :: a -> a
  acos :: a -> a
  atan :: a -> a
  sinh :: a -> a
  cosh :: a -> a
  tanh :: a -> a
  asinh :: a -> a
  acosh :: a -> a
  atanh :: a -> a
  GHC.Float.log1p :: a -> a
  GHC.Float.expm1 :: a -> a
  GHC.Float.log1pexp :: a -> a
  GHC.Float.log1mexp :: a -> a
  {-# MINIMAL pi, exp, log, sin, cos, asin, acos, atan, sinh, cosh,
              asinh, acosh, atanh #-}
      -- Defined in ‘GHC.Float’
instance RealFloat a => Floating (Complex a) -- Defined in ‘Data.Complex’
instance Floating Float -- Defined in ‘GHC.Float’
instance Floating Double -- Defined in ‘GHC.Float’
```

Caution: while expressions such as `sqrt . negate :: Floating a => a -> a` are perfectly valid, they might return `NaN` ("not-a-number"), which may not be an intended behaviour. In such cases, we might want to work over the Complex field (shown later).

In Haskell, all expressions (which includes numerical constants and functions operating on those) have a decidable type. At compile time, the type-checker infers the type of an expression from the types of the elementary functions that compose it. Since data is immutable by default, there are no "type casting" operations, but there are functions that copy data and generalize or specialize the types within reason.

第67.1节：基本示例

```
λ> :t 1
1 :: Num t => t

λ> :t pi
pi :: Floating a => a
```

在上述示例中，类型检查器推断出的是类型类而非具体类型。Haskell中，Num类是最通用的数值类（因为它包含整数和实数），但pi必须属于更专门的类，因为它有非零的小数部分。

```
list0 :: [Integer]
list0 = [1, 2, 3]

list1 :: [Double]
list1 = [1, 2, pi]
```

上述具体类型是由GHC推断的。更通用的类型如list0 :: Num a => [a]也可以，但由于上述注意事项（例如，如果将一个Double加到一个Num列表中），这类类型更难保持。

第67.2节："无法推导 (Fractional Int) ..."

标题中的错误信息是初学者常见的错误。让我们看看它是如何产生的以及如何修复它。

假设我们需要计算一组数字的平均值；下面的声明看似可以做到这一点，
但它无法编译：

```
averageOfList ll = sum ll / length ll
```

问题出在除法(/)函数上：它的签名是(/) :: Fractional a => a -> a -> a，但在上述情况下，分母（由length :: Foldable t => t a -> Int给出）是Int类型（且Int不属于Fractional类），因此出现错误信息。

我们可以用fromIntegral :: (Num b, Integral a) => a -> b来修正错误信息。可以看到该函数接受任何Integral类型的值，并返回对应的Num类中的值：

```
averageOfList' :: (Foldable t, Fractional a) => t a -> a
averageOfList' ll = sum ll / fromIntegral (length ll)
```

第67.3节：函数示例

(+)的类型是什么？

```
λ> :t (+)
(+) :: Num a => a -> a -> a
```

sqrt的类型是什么？

```
λ> :t sqrt
sqrt :: 浮点数 a => a -> a
```

sqrt . fromIntegral 的类型是什么？

Section 67.1: Basic examples

```
λ> :t 1
1 :: Num t => t

λ> :t pi
pi :: Floating a => a
```

In the examples above, the type-checker infers a type-*class* rather than a concrete type for the two constants. In Haskell, the `Num` class is the most general numerical one (since it encompasses integers and reals), but `pi` must belong to a more specialized class, since it has a nonzero fractional part.

```
list0 :: [Integer]
list0 = [1, 2, 3]

list1 :: [Double]
list1 = [1, 2, pi]
```

The concrete types above were inferred by GHC. More general types like `list0 :: Num a => [a]` would have worked, but would have also been harder to preserve (e.g. if one consed a `Double` onto a list of `Num`s), due to the caveats shown above.

Section 67.2: `Could not deduce (Fractional Int) ...`

The error message in the title is a common beginner mistake. Let's see how it arises and how to fix it.

Suppose we need to compute the average value of a list of numbers; the following declaration would seem to do it, but it wouldn't compile:

```
averageOfList ll = sum ll / length ll
```

The problem is with the division `(/)` function: its signature is `(/) :: Fractional a => a -> a -> a`, but in the case above the denominator (given by `length :: Foldable t => t a -> Int`) is of type `Int` (and `Int` does not belong to the `Fractional` class) hence the error message.

We can fix the error message with `fromIntegral :: (Num b, Integral a) => a -> b`. One can see that this function accepts values of any `Integral` type and returns corresponding ones in the `Num` class:

```
averageOfList' :: (Foldable t, Fractional a) => t a -> a
averageOfList' ll = sum ll / fromIntegral (length ll)
```

Section 67.3: Function examples

What's the type of `(+)` ?

```
λ> :t (+)
(+) :: Num a => a -> a -> a
```

What's the type of `sqrt` ?

```
λ> :t sqrt
sqrt :: Floating a => a -> a
```

What's the type of `sqrt . fromIntegral` ?

```
sqrt . fromIntegral :: (整数 a, 浮点数 c) => a -> c
```

```
sqrt . fromIntegral :: (Integral a, Floating c) => a -> c
```

第68章：角色

TypeFamilies 语言扩展允许程序员定义类型级函数。区别类型函数与非GADT类型构造器的是，类型函数的参数可以是非参数化的，而类型构造器的参数总是参数化的。这个区别对于 GeneralizedNewTypeDeriving 扩展的正确性非常重要。为了解释这个区别，Haskell中引入了角色的概念。

第68.1节：名义角色

Haskell Wiki 有一个类型函数非参数化参数的例子：

```
类型 家族 Inspect x
类型实例 Inspect 年龄 = Int
类型实例 Inspect Int = Bool
```

这里 x 是非参数化的，因为为了确定将 Inspect 应用于类型参数的结果，类型函数必须检查 x。

在这种情况下，x 的角色是名义的。我们可以使用 RoleAnnotations 扩展显式声明该角色：

```
type role Inspect nominal
```

第68.2节：表示角色

类型函数的参数参数示例：

```
data List a = Nil | Cons a (List a)

type family DoNotInspect x
type instance DoNotInspect x = List x
```

这里 x 是参数化的，因为为了确定将 DoNotInspect 应用于类型参数的结果，类型函数不需要检查 x。

在这种情况下，x 的角色是表示性的。我们可以使用 RoleAnnotations 扩展显式声明该角色：

```
type role DoNotInspect representational
```

第68.3节：幻影角色

幻影类型参数具有幻影角色。幻影角色不能被显式声明。

Chapter 68: Role

The TypeFamilies language extension allows the programmer to define type-level functions. What distinguishes type functions from non-GADT type constructors is that parameters of type functions can be non-parametric whereas parameters of type constructors are always parametric. This distinction is important to the correctness of the GeneralizedNewTypeDeriving extension. To explicate this distinction, roles are introduced in Haskell.

Section 68.1: Nominal Role

Haskell Wiki has an example of a non-parametric parameter of a type function:

```
type family Inspect x
type instance Inspect Age = Int
type instance Inspect Int = Bool
```

Here x is non-parametric because to determine the outcome of applying Inspect to a type argument, the type function must inspect x.

In this case, the role of x is nominal. We can declare the role explicitly with the RoleAnnotations extension:

```
type role Inspect nominal
```

Section 68.2: Representational Role

An example of a parametric parameter of a type function:

```
data List a = Nil | Cons a (List a)

type family DoNotInspect x
type instance DoNotInspect x = List x
```

Here x is parametric because to determine the outcome of applying DoNotInspect to a type argument, the type function do not need to inspect x.

In this case, the role of x is representational. We can declare the role explicitly with the RoleAnnotations extension:

```
type role DoNotInspect representational
```

Section 68.3: Phantom Role

A phantom type parameter has a phantom role. Phantom roles cannot be declared explicitly.

第69章：使用RankNTypes的任意阶多态

GHC的类型系统通过使用
Rank2Types和RankNTypes语言扩展支持类型中的任意阶显式全称量化。

第69.1节：RankNTypes

StackOverflow迫使我必须有一个示例。如果该主题被批准，我们应该将此示例移到这里。

Chapter 69: Arbitrary-rank polymorphism with RankNTypes

GHC's type system supports arbitrary-rank explicit universal quantification in types through the use of the `Rank2Types` and `RankNTypes` language extensions.

Section 69.1: RankNTypes

StackOverflow forces me to have one example. If this topic is approved, we should move this example here.

第70章：GHCJS

GHCJS是一个使用GHC API的Haskell到JavaScript的编译器。

第70.1节：使用Node.js运行“Hello World!”

ghcjs可以使用与ghc相同的命令行参数调用。生成的程序可以直接从shell使用Node.js和SpiderMonkey jsshell运行。例如：

```
$ ghcjs -o helloWorld helloWorld.hs
$ node helloWorld.jsexe/all.js
你好，世界!
```

Chapter 70: GHCJS

GHCJS is a Haskell to JavaScript compiler that uses the GHC API.

Section 70.1: Running "Hello World!" with Node.js

`ghcjs` can be invoked with the same command line arguments as `ghc`. The generated programs can be run directly from the shell with Node.js and SpiderMonkey jsshell. for example:

```
$ ghcjs -o helloWorld helloWorld.hs
$ node helloWorld.jsexe/all.js
Hello world!
```

第71章：XML

XML文档的编码和解码。

第71.1节：使用`xml`库编码记录

```
{-# LANGUAGE RecordWildCards #-}
import Text.XML.Light

data Package = Package
  { pOrderNo  :: String
  , pOrderPos :: String
  , pBarcode  :: String
  , pNumber   :: String
  }

-- | 从Package创建XML
instance Node Package where
  node qn Package {..} =
    node qn
      [ unode "package_number" pNumber
      , unode "package_barcode" pBarcode
      , unode "order_number" pOrderNo
      , unode "order_position" pOrderPos
      ]
```

Chapter 71: XML

Encoding and decoding of XML documents.

Section 71.1: Encoding a record using the `xml` library

```
{-# LANGUAGE RecordWildCards #-}
import Text.XML.Light

data Package = Package
  { pOrderNo  :: String
  , pOrderPos :: String
  , pBarcode  :: String
  , pNumber   :: String
  }

-- | Create XML from a Package
instance Node Package where
  node qn Package {..} =
    node qn
      [ unode "package_number" pNumber
      , unode "package_barcode" pBarcode
      , unode "order_number" pOrderNo
      , unode "order_position" pOrderPos
      ]
```

第72章：Reader / ReaderT

Reader提供将一个值传递给每个函数的功能。这里有一份带有一些图示的有用指南：http://adit.io/posts/2013-06-10-three-useful-monads.html

第72.1节：简单演示

Reader单子的一个关键部分是ask (https://hackage.haskell.org/package/mtl-2.2.1/docs/Control-Monad-Reader.html#v:ask)函数，定义如下仅作说明用途：

```
import Control.Monad.Trans.Reader hiding (ask)
import Control.Monad.Trans

ask :: Monad m => ReaderT r m r
ask = reader id

main :: IO ()
main = do
  let f = (runReaderT $ readerExample) :: Integer -> IO String
  x <- f 100
  print x
  --
  let fIO = (runReaderT $ readerExampleIO) :: Integer -> IO String
  y <- fIO 200
  print y

readerExample :: ReaderT Integer IO String
readerExample = do
x <- ask
  return $ "值为: " ++ show x

liftAnnotated :: IO a -> ReaderT Integer IO a
liftAnnotated = lift

readerExampleIO :: ReaderT Integer IO String
readerExampleIO = do
x <- reader id
lift $ print "来自内部的问候"
  liftAnnotated $ print "来自内部的问候..."
  return $ "值为: " ++ show x
```

以上将打印出：

```
"值为: 100"
"来自内部的问候"
"来自内部的问候..."
"值为: 200"
```

Chapter 72: Reader / ReaderT

Reader provides functionality to pass a value along to each function. A helpful guide with some diagrams can be found here: http://adit.io/posts/2013-06-10-three-useful-monads.html

Section 72.1: Simple demonstration

A key part of the Reader monad is the ask (https://hackage.haskell.org/package/mtl-2.2.1/docs/Control-Monad-Reader.html#v:ask) function, which is defined for illustrative purposes:

```
import Control.Monad.Trans.Reader hiding (ask)
import Control.Monad.Trans

ask :: Monad m => ReaderT r m r
ask = reader id

main :: IO ()
main = do
  let f = (runReaderT $ readerExample) :: Integer -> IO String
  x <- f 100
  print x
  --
  let fIO = (runReaderT $ readerExampleIO) :: Integer -> IO String
  y <- fIO 200
  print y

readerExample :: ReaderT Integer IO String
readerExample = do
  x <- ask
  return $ "The value is: " ++ show x

liftAnnotated :: IO a -> ReaderT Integer IO a
liftAnnotated = lift

readerExampleIO :: ReaderT Integer IO String
readerExampleIO = do
  x <- reader id
  lift $ print "Hello from within"
  liftAnnotated $ print "Hello from within..."
  return $ "The value is: " ++ show x
```

The above will print out:

```
"The value is: 100"
"Hello from within"
"Hello from within..."
"The value is: 200"
```

第73章：函数调用语法

Haskell的函数调用语法，结合C风格语言的对比进行解释。此内容面向有C风格语言背景的读者。

第73.1节：部分应用 - 第一部分

在Haskell中，函数可以部分应用；我们可以将所有函数视为接受单一参数，并返回一个修改后的函数，该函数中该参数是固定的。为说明这一点，我们可以如下括号化函数：

```
(((plus) 1) 2)
```

这里，函数(plus)被应用于1，得到函数((plus) 1)，该函数又被应用于2，得到函数(((plus) 1) 2)。因为plus 1 2是一个不接受参数的函数，你可以将其视为一个普通值；然而在Haskell中，函数和值之间几乎没有区别。

更详细地说，函数plus是一个对其参数进行加法运算的函数。
函数plus 1是一个将1加到其参数上的函数。
函数plus 1 2是一个将1加到2上的函数，其结果总是值3。

第73.2节：部分应用 - 第二部分

作为另一个例子，我们有函数map，它接受一个函数和一个值列表，并将该函数应用于列表中的每个值：

```
map :: (a -> b) -> [a] -> [b]
```

假设我们想要对列表中的每个值加一。你可以决定定义自己的函数，该函数对其参数加一，然后用map将该函数映射到你的列表上

```
addOne x = plus 1 x
map addOne [1,2,3]
```

但如果你再看看addOne的定义，加上括号以示强调：

```
(addOne) x = ((plus) 1) x
```

函数addOne，应用于任意值 x，等同于部分应用函数plus 1应用于 x。
这意味着函数addOne和plus 1是相同的，我们可以避免定义新函数，只需用plus 1替换addOne，记得用括号将plus 1作为子表达式隔离开：

```
map (plus 1) [1,2,3]
```

第73.3节：基本函数调用中的括号

对于C风格的函数调用，例如

```
plus(a, b); // 括号只包围参数，参数间用逗号分隔
```

那么等价的Haskell代码是

Chapter 73: Function call syntax

Haskell's function call syntax, explained with comparisons to C-style languages where applicable. This is aimed at people who are coming to Haskell from a background in C-style languages.

Section 73.1: Partial application - Part 1

In Haskell, functions can be partially applied; we can think of all functions as taking a single argument, and returning a modified function for which that argument is constant. To illustrate this, we can bracket functions as follows:

```
(((plus) 1) 2)
```

Here, the function (plus) is applied to 1 yielding the function ((plus) 1), which is applied to 2, yielding the function (((plus) 1) 2). Because plus 1 2 is a function which takes no arguments, you can consider it a plain value; however in Haskell, there is little distinction between functions and values.

To go into more detail, the function plus is a function that adds its arguments.
The function plus 1 is a function that adds 1 to its argument.
The function plus 1 2 is a function that adds 1 to 2, which is always the value 3.

Section 73.2: Partial application - Part 2

As another example, we have the function **map**, which takes a function and a list of values, and applies the function to each value of the list:

```
map :: (a -> b) -> [a] -> [b]
```

Let's say we want to increment each value in a list. You may decide to define your own function, which adds one to its argument, and **map** that function over your list

```
addOne x = plus 1 x
map addOne [1,2,3]
```

but if you have another look at addOne's definition, with parentheses added for emphasis:

```
(addOne) x = ((plus) 1) x
```

The function addOne, when applied to any value x, is the same as the partially applied function plus 1 applied to x. This means the functions addOne and plus 1 are identical, and we can avoid defining a new function by just replacing addOne with plus 1, remembering to use parentheses to isolate plus 1 as a subexpression:

```
map (plus 1) [1,2,3]
```

Section 73.3: Parentheses in a basic function call

For a C-style function call, e.g.

```
plus(a, b); // Parentheses surrounding only the arguments, comma separated
```

Then the equivalent Haskell code will be

```
(plus a b) -- 括号包围函数和参数，无逗号
```

在Haskell中，函数应用不需要显式使用括号，括号仅用于消除表达式的歧义，就像数学中一样；因此在括号包围表达式中所有文本的情况下，实际上不需要括号，以下写法也是等价的：

```
plus a b -- 这里不需要括号！
```

重要的是要记住，虽然在C风格语言中，函数

第73.4节：嵌入函数调用中的括号

在前面的例子中，我们最终没有使用括号，因为它们并不影响语句的含义。然而，在更复杂的表达式中，它们通常是必要的，比如下面的例子。

在C语言中：

```
plus(a, take(b, c));
```

在Haskell中变为：

```
(plus a (take b c))
-- 或者等价地，省略最外层括号
plus a (take b c)
```

注意，这与以下表达式不等价：

```
plus a take b c -- 这不是我们想要的！
```

有人可能会认为，因为编译器知道 **`take`** 是一个函数，它应该能够知道你想要
将它应用于参数 `b` 和 `c`，并将其结果传递给 `plus`。
然而，在 Haskell 中，函数经常将其他函数作为参数，并且函数与其他值之间几乎没有实际区别；因此编译器不
能仅仅因为 take 是一个函数就假设你的意图。

因此，最后一个例子类似于以下的 C 函数调用：

```
plus(a, take, b, c); // 这不是我们想要的！
```

```
(plus a b) -- Parentheses surrounding the function and the arguments, no commas
```

In Haskell, parentheses are not explicitly required for function application, and are only used to disambiguate expressions, like in mathematics; so in cases where the brackets surround all the text in the expression, the parentheses are actually not needed, and the following is also equivalent:

```
plus a b -- no parentheses are needed here!
```

It is important to remember that while in C-style languages, the function

Section 73.4: Parentheses in embedded function calls

In the previous example, we didn't end up needing the parentheses, because they did not affect the meaning of the statement. However, they are often necessary in more complex expression, like the one below.
In C:

```
plus(a, take(b, c));
```

In Haskell this becomes:

```
(plus a (take b c))
-- or equivalently, omitting the outermost parentheses
plus a (take b c)
```

Note, that this is not equivalent to:

```
plus a take b c -- Not what we want!
```

One might think that because the compiler knows that **`take`** is a function, it would be able to know that you want to apply it to the arguments `b` and `c`, and pass its result to `plus`.
However, in Haskell, functions often take other functions as arguments, and little actual distinction is made between functions and other values; and so the compiler cannot assume your intention simply because **`take`** is a function.

And so, the last example is analogous to the following C function call:

```
plus(a, take, b, c); // Not what we want!
```

第74章：日志记录

在 Haskell 中，日志记录通常通过 IO 单子中的函数实现，因此仅限于非纯函数或“IO 操作”。

在 Haskell 程序中有多种记录信息的方法：从 **putStrLn**（或 **print**）到诸如 hslogger 的库，或通过 Debug.Trace。

第74.1节：使用 hslogger 进行日志记录

hslogger 模块提供了类似于 Python 的 logging 框架的 API，支持分层命名的日志记录器、级别以及重定向到 stdout 和 stderr 之外的句柄。

默认情况下，所有级别为WARNING及以上的消息都会发送到标准错误输出（stderr），其他所有日志级别都会被忽略。

```
import           System.Log.Logger (Priority (DEBUG), debugM, infoM, setLevel,
                                    updateGlobalLogger, warningM)

main = do
debugM "MyProgram.main" "这条信息不会被看到"
  infoM "MyProgram.main" "这条信息也不会被看到"
  warningM "MyProgram.main" "这条信息将会被看到"
```

我们可以使用updateGlobalLogger通过名称设置日志记录器的级别：

```
updateGlobalLogger "MyProgram.main" (setLevel DEBUG)

  debugM "MyProgram.main" "这条信息现在将会被看到"
```

每个日志记录器都有一个名称，并且它们是分层排列的，因此MyProgram是MyParent.Module的父级。

Chapter 74: Logging

Logging in Haskell is achieved usually through functions in the IO monad, and so is limited to non-pure functions or "IO actions".

There are several ways to log information in a Haskell program: from **putStrLn** (or **print**), to libraries such as hslogger or through Debug.Trace.

Section 74.1: Logging with hslogger

The hslogger module provides a similar API to Python's logging framework, and supports hierarchically named loggers, levels and redirection to handles outside of stdout and stderr.

By default, all messages of level WARNING and above are sent to stderr and all other log levels are ignored.

```
import           System.Log.Logger (Priority (DEBUG), debugM, infoM, setLevel,
                                    updateGlobalLogger, warningM)

main = do
  debugM "MyProgram.main" "This won't be seen"
  infoM "MyProgram.main" "This won't be seen either"
  warningM "MyProgram.main" "This will be seen"
```

We can set the level of a logger by its name using updateGlobalLogger:

```
  updateGlobalLogger "MyProgram.main" (setLevel DEBUG)

  debugM "MyProgram.main" "This will now be seen"
```

Each Logger has a name, and they are arranged hierarchically, so MyProgram is a parent of MyParent.Module.

第75章：Attoparsec

类型	详细信息
Parser i a表示解析器的核心类型。 i是字符串类型，例如ByteString。	
IResult i r解析结果，构造函数包括Fail i [String] String，Partial (i -> IResult i r)和Done i r。构造函数。	

Attoparsec 是一个解析组合子库，特别针对高效处理网络协议
和复杂的文本/二进制文件格式而设计。

Attoparsec 不仅提供速度和效率，还支持回溯和增量输入。

它的 API 与另一个解析组合子库 Parsec 非常相似。

有用于兼容ByteString、Text和Char8的子模块。建议使用OverloadedStrings语言扩展。

第75.1节：组合子

解析输入最好通过由较小的、单一功能的解析器函数组成的较大解析器函数来实现。

假设我们想解析以下表示工作时间的文本：

> 星期一：0800 1600。

我们可以将其拆分为两个“标记”：日期名称——“星期一”——和时间部分“0800”到“1600”。

要解析日期名称，我们可以编写如下代码：

数据 Day = Day 字符串 day :: 解析器 Day day = do name <- takeWhile1 (/= ':') skipMany1 (char ':') skipSpace return $ Day name

要解析时间部分，我们可以这样写：

数据 TimePortion = TimePortion 字符串 字符串 time = do start <- takeWhile1 isDigit skipSpace end <- takeWhile1 isDigit return $ TimePortion start end

现在我们有了两个用于解析文本各个部分的解析器，可以将它们组合成一个“更大”的解析器来读取整天的工作时间：

数据 WorkPeriod = WorkPeriod 日期 TimePortion work = do d <- day t <- time return $ WorkPeriod d t

然后运行解析器：

parseOnly work "Monday: 0800 1600"

第75.2节：位图 - 解析二进制数据

Attoparsec 使解析二进制数据变得简单。假设有以下定义：

```
import           Data.Attoparsec.ByteString (Parser, eitherResult, parse, take)
import           Data.Binary.Get            (getWord32le, runGet)
import          Data.ByteString            (ByteString, readFile)
```

Chapter 75: Attoparsec

Type	Detail
Parser i a	The core type for representing a parser. i is the string type, e.g. ByteString.
IResult i r	The result of a parse, with Fail i [String] String, Partial (i -> IResult i r) and Done i r as constructors.

Attoparsec is a parsing combinator library that is "aimed particularly at dealing efficiently with network protocols and complicated text/binary file formats".

Attoparsec offers not only speed and efficiency, but backtracking and incremental input.

Its API closely mirrors that of another parser combinator library, Parsec.

There are submodules for compatibility with ByteString, Text and Char8. Use of the OverloadedStrings language extension is recommended.

Section 75.1: Combinators

Parsing input is best achieved through larger parser functions that are composed of smaller, single purpose ones.

Let's say we wished to parse the following text which represents working hours:

> Monday: 0800 1600.

We could split these into two "tokens": the day name -- "Monday" -- and a time portion "0800" to "1600".

To parse a day name, we could write the following:

data Day = Day String day :: Parser Day day = do name <- takeWhile1 (/= ':') skipMany1 (char ':') skipSpace return $ Day name

To parse the time portion we could write:

data TimePortion = TimePortion String String time = do start <- takeWhile1 isDigit skipSpace end <- takeWhile1 isDigit return $ TimePortion start end

Now we have two parsers for our individual parts of the text, we can combine these in a "larger" parser to read an entire day's working hours:

data WorkPeriod = WorkPeriod Day TimePortion work = do d <- day t <- time return $ WorkPeriod d t

and then run the parser:

parseOnly work "Monday: 0800 1600"

Section 75.2: Bitmap - Parsing Binary Data

Attoparsec makes parsing binary data trivial. Assuming these definitions:

```
import           Data.Attoparsec.ByteString (Parser, eitherResult, parse, take)
import           Data.Binary.Get            (getWord32le, runGet)
import           Data.ByteString            (ByteString, readFile)
```

```haskell
import           Data.ByteString.Char8      (unpack)
import           Data.ByteString.Lazy       (fromStrict)
import           Prelude                     hiding (readFile, take)

-- 位图头中的DIB部分
data DIB = BM | BA | CI | CP | IC | PT
           deriving (Show, Read)

type Reserved = ByteString

-- 整个位图头
data Header = Header DIB Int Reserved Reserved Int
              deriving (Show)
```

我们可以轻松地从位图文件中解析头部。这里，我们有4个解析器函数，表示位图文件的头部部分：

首先，DIB部分可以通过读取前两个字节来读取

```haskell
dibP :: Parser DIB
dibP = read . unpack <$> take 2
```

同样，位图的大小、保留部分和像素偏移量也可以轻松读取：

```haskell
sizeP :: Parser Int
sizeP = fromIntegral . runGet getWord32le . fromStrict <$> take 4

reservedP :: Parser Reserved
reservedP = take 2

addressP :: 解析器 整数
addressP = fromIntegral . runGet getWord32le . fromStrict <$> take 4
```

然后可以将其组合成整个头部的更大解析函数：

```haskell
bitmapHeader :: Parser Header
bitmapHeader = do
dib <- dibP
    sz <- sizeP
    reservedP
    reservedP
    offset <- addressP
    return $ Header dib sz "" "" offset
```

```haskell
import           Data.ByteString.Char8      (unpack)
import           Data.ByteString.Lazy       (fromStrict)
import           Prelude                    hiding (readFile, take)

-- The DIB section from a bitmap header
data DIB = BM | BA | CI | CP | IC | PT
           deriving (Show, Read)

type Reserved = ByteString

-- The entire bitmap header
data Header = Header DIB Int Reserved Reserved Int
              deriving (Show)
```

We can parse the header from a bitmap file easily. Here, we have 4 parser functions that represent the header section from a bitmap file:

Firstly, the DIB section can be read by taking the first 2 bytes

```haskell
dibP :: Parser DIB
dibP = read . unpack <$> take 2
```

Similarly, the size of the bitmap, the reserved sections and the pixel offset can be read easily too:

```haskell
sizeP :: Parser Int
sizeP = fromIntegral . runGet getWord32le . fromStrict <$> take 4

reservedP :: Parser Reserved
reservedP = take 2

addressP :: Parser Int
addressP = fromIntegral . runGet getWord32le . fromStrict <$> take 4
```

which can then be combined into a larger parser function for the entire header:

```haskell
bitmapHeader :: Parser Header
bitmapHeader = do
    dib <- dibP
    sz <- sizeP
    reservedP
    reservedP
    offset <- addressP
    return $ Header dib sz "" "" offset
```

第76章：zipWithM

zipWithM 对应于 zipWith ，就像 mapM 对应于 map ：它允许你使用单子函数组合两个列表。

来自模块 Control.Monad

第76.1节：计算销售价格

假设你想查看某组销售价格是否对一家商店合理。

这些商品的原价是5美元，所以如果销售价格低于5美元，你不想接受销售，但你确实想知道新的价格是多少。

计算单个价格很简单：你计算销售价格，如果没有利润则返回Nothing：

```
calculateOne :: Double -> Double -> Maybe Double
calculateOne price percent = let newPrice = price*(percent/100)
                             in if newPrice < 5 then Nothing else Just newPrice
```

要计算整个销售的价格，zipWithM 使这变得非常简单：

```
calculateAllPrices :: [Double] -> [Double] -> Maybe [Double]
calculateAllPrices prices percents = zipWithM calculateOne prices percents
```

如果任何销售价格低于5美元，这将返回Nothing。

Chapter 76: zipWithM

`zipWithM` is to **`zipWith`** as **`mapM`** is to **`map`**: it lets you combine two lists using a monadic function.

From the module `Control.Monad`

Section 76.1: Calculatings sales prices

Suppose you want to see if a certain set of sales prices makes sense for a store.

The items originally cost $5, so you don't want to accept the sale if the sales price is less for any of them, but you do want to know what the new price is otherwise.

Calculating one price is easy: you calculate the sales price, and return `Nothing` if you don't get a profit:

```
calculateOne :: Double -> Double -> Maybe Double
calculateOne price percent = let newPrice = price*(percent/100)
                             in if newPrice < 5 then Nothing else Just newPrice
```

To calculate it for the entire sale, `zipWithM` makes it really simple:

```
calculateAllPrices :: [Double] -> [Double] -> Maybe [Double]
calculateAllPrices prices percents = zipWithM calculateOne prices percents
```

This will return `Nothing` if any of the sales prices are below $5.

第77章：Profunctor

Profunctor 是由profunctors包在Data.Profunctor中提供的一个类型类。

完整解释请参见“备注”部分。

第77.1节：(->) Profunctor

(->) 是一个前函子（profunctor）的简单例子：左边的参数是函数的输入，右边的参数与读取函子（reader functor）实例相同。

```
实例 Profunctor (->) 的定义
    lmap f g = g . f
rmap f g = g . g
```

Chapter 77: Profunctor

Profunctor is a typeclass provided by the profunctors package in Data.Profunctor.

See the "Remarks" section for a full explanation.

Section 77.1: (->) Profunctor

(->) is a simple example of a profunctor: the left argument is the input to a function, and the right argument is the same as the reader functor instance.

```
instance Profunctor (->) where
    lmap f g = g . f
    rmap f g = g . g
```

第78章：类型应用

类型应用（TypeApplications）是在编译器难以推断某个表达式的类型时，替代类型注解的一种方式。

本系列示例将解释TypeApplications扩展的目的及其用法

别忘了在源文件顶部添加{-# LANGUAGE TypeApplications #-}以启用该扩展。

第78.1节：避免类型注解

我们使用类型注解来避免歧义。类型应用也可以达到同样的目的。例如

```
x :: Num a => a
x = 5

main :: IO ()
main = print x
```

这段代码存在歧义错误。我们知道 a有一个Num实例，并且为了打印它，我们知道它需要一个
Show 实例。如果 a 例如是一个 Int，这样可能可行，因此为了解决错误，我们可以添加类型注解

```
main = print (x :: Int)
```

另一种使用类型应用的解决方案如下所示

```
main = print @Int x
```

要理解这意味着什么，我们需要查看 print 的类型签名。

```
print :: Show a => a -> IO ()
```

该函数接受一个类型为 a 的参数，但另一种看法是它实际上接受两个参数。
第一个是一个 type 参数，第二个是一个值，其类型是第一个参数。

值参数和类型参数之间的主要区别在于，后者在调用函数时是隐式提供的。谁提供它们？类型推断算法！什么

TypeApplications 让我们可以显式地给出这些类型参数。当类型推断无法确定正确类型时，这尤其有用。

所以，为了分解上述示例

```
print :: Show a => a -> IO ()
print @Int :: Int -> IO ()
print @Int x :: IO ()
```

第78.2节：其他语言的类型申请

如果你熟悉像Java、C#或C++这样的语言以及泛型/模板的概念，那么这个比较可能对你有用。

假设我们在C#中有一个通用函数

Chapter 78: Type Application

TypeApplications are an alternative to type *annotations* when the compiler struggles to infer types for a given expression.

This series of examples will explain the purpose of the TypeApplications extension and how to use it

Don't forget to enable the extension by placing {-# LANGUAGE TypeApplications #-} at the top of your source file.

Section 78.1: Avoiding type annotations

We use type annotations to avoid ambiguity. Type applications can be used for the same purpose. For example

```
x :: Num a => a
x = 5

main :: IO ()
main = print x
```

This code has an ambiguity error. We know that a has a Num instance, and in order to print it we know it needs a Show instance. This could work if a was, for example, an Int, so to fix the error we can add a type annotation

```
main = print (x :: Int)
```

Another solution using type applications would look like this

```
main = print @Int x
```

To understand what this means we need to look at the type signature of **print**.

```
print :: Show a => a -> IO ()
```

The function takes one parameter of type a, but another way to look at it is that it actually takes two parameters. The first one is a *type* parameter, the second one is a value whose type is the first parameter.

The main difference between value parameters and the type parameters is that the latter ones are implicitly provided to functions when we call them. Who provides them? The type inference algorithm! What TypeApplications let us do is give those type parameters explicitly. This is especially useful when the type inference can't determine the correct type.

So to break down the above example

```
print :: Show a => a -> IO ()
print @Int :: Int -> IO ()
print @Int x :: IO ()
```

Section 78.2: Type applications in other languages

If you're familiar with languages like Java, C# or C++ and the concept of generics/templates then this comparison might be useful for you.

Say we have a generic function in C#

```
public static T DoNothing<T>(T in) { return in; }
```

要用float调用此函数，我们可以写成DoNothing(5.0f)，或者如果想要明确指出类型，可以写成DoNothing<float>(5.0f)。尖括号内的部分是类型应用。

在 Haskell 中也是如此，不同之处在于类型参数不仅在调用处是隐式的，在定义处也是隐式的。

```
doNothing :: a -> a
doNothing x = x
```

这也可以通过使用ScopedTypeVariables、Rank2Types或RankNTypes扩展来明确表示，如下所示。

```
doNothing :: forall a. a -> a
doNothing x = x
```

然后在调用处，我们可以再次写成 doNothing 5.0 或者 doNothing @Float 5.0

第78.3节：参数顺序

当类型参数是隐式时，一旦有多个类型参数，问题就变得明显。它们的顺序是怎样的？

```
const :: a -> b -> a
```

写成 const @Int 是表示 a 等于 Int，还是 b？如果我们使用 forall 明确声明类型参数，比如 const :: forall a b. a -> b -> a，那么顺序就是写的那样：先是 a，然后是 b。

如果不这样做，那么变量的顺序就是从左到右。第一个提到的变量是第一个类型参数，第二个是第二个类型参数，依此类推。

如果我们想指定第二个类型变量，但不指定第一个呢？我们可以对第一个变量使用通配符，像这样

```
const @_ @Int
```

该表达式的类型是

```
const @_ @Int :: a -> Int -> a
```

第78.4节：与模糊类型的交互

假设你正在引入一个表示字节大小的类型类。

```
class SizeOf a where
    sizeOf :: a -> Int
```

问题是，对于该类型的每个值，大小应该是常量。我们实际上不希望 sizeOf 函数依赖于 a，而只依赖于它的类型。

在没有类型应用的情况下，我们最好的解决方案是定义如下的 Proxy 类型

```
数据 Proxy a = Proxy
```

该类型的目的是携带类型信息，但不携带值信息。然后我们的类可以写成这样

```
public static T DoNothing<T>(T in) { return in; }
```

To call this function with a float we can do DoNothing(5.0f) or if we want to be explicit we can say DoNothing<float>(5.0f). That part inside of the angle brackets is the type application.

In Haskell it's the same, except that the type parameters are not only implicit at call sites but also at definition sites.

```
doNothing :: a -> a
doNothing x = x
```

This can also be made explicit using either ScopedTypeVariables, Rank2Types or RankNTypes extensions like this.

```
doNothing :: forall a. a -> a
doNothing x = x
```

Then at the call site we can again either write doNothing 5.0 or doNothing @Float 5.0

Section 78.3: Order of parameters

The problem with type arguments being implicit becomes obvious once we have more than one. Which order do they come in?

```
const :: a -> b -> a
```

Does writing const @Int mean a is equal to Int, or is it b? In case we explicitly state the type parameters using a forall like const :: forall a b. a -> b -> a then the order is as written: a, then b.

If we don't, then the order of variables is from left to right. The first variable to be mentioned is the first type parameter, the second is the second type parameter and so on.

What if we want to specify the second type variable, but not the first? We can use a wildcard for the first variable like this

```
const @_ @Int
```

The type of this expression is

```
const @_ @Int :: a -> Int -> a
```

Section 78.4: Interaction with ambiguous types

Say you're introducing a class of types that have a size in bytes.

```
class SizeOf a where
    sizeOf :: a -> Int
```

The problem is that the size should be constant for every value of that type. We don't actually want the sizeOf function to depend on a, but only on it's type.

Without type applications, the best solution we had was the Proxy type defined like this

```
data Proxy a = Proxy
```

The purpose of this type is to carry type information, but no value information. Then our class could look like this

```haskell
class SizeOf a where
    sizeOf :: Proxy a -> Int
```

现在你可能会想，为什么不干脆去掉第一个参数？那么我们的函数类型就只是sizeOf :: Int，或者更准确地说，因为它是类的方法，应该是 sizeOf :: SizeOf a => Int，甚至更明确地写成 sizeOf :: forall a. SizeOf a => Int。

问题在于类型推断。如果我在某处写了 sizeOf，推断算法只知道我期望一个
Int。它根本不知道我想用什么类型替换 a。正因为如此，除非你启用了{-# LANGUAGE AllowAmbiguousTypes #-}扩展，否则编译器会拒绝该定义。在启用该扩展的情况下，定义可以编译，但在任何地方使用都会出现歧义错误。

幸运的是，类型应用的引入拯救了局面！现在我们可以写成 `sizeOf @Int`，明确表示 a 是
Int。类型应用允许我们提供一个类型参数，即使它没有出现在函数的
实际参数中！

```haskell
class SizeOf a where
    sizeOf :: Proxy a -> Int
```

Now you might be wondering, why not drop the first argument altogether? The type of our function would then just be `sizeOf :: Int` or, to be more precise because it is a method of a class, `sizeOf :: SizeOf a => Int` or to be even more explicit `sizeOf :: forall a. SizeOf a => Int`.

The problem is type inference. If I write `sizeOf` somewhere, the inference algorithm only knows that I expect an `Int`. It has no idea what type I want to substitute for `a`. Because of this, the definition gets rejected by the compiler *unless* you have the `{-# LANGUAGE AllowAmbiguousTypes #-}` extension enabled. In that case the definition compiles,it just can't be used anywhere without an ambiguity error.

Luckily, the introduction of type applications saves the day! Now we can write `sizeOf @Int`, explicitly saying that `a` is `Int`. Type applications allow us to provide a type parameter, even if it doesn't appear in the *actual parameters of the function*!

鸣谢

非常感谢所有来自 Stack Overflow Documentation 的人员帮助提供了这些内容，
更多更改可以发送至 web@petercv.com 以发布或更新新内容

3442	第10章
亚当·瓦格纳	第2章
亚历克	第18章和第37章
alejosocorro	第一章
阿米泰·斯特恩	第一章
安塔尔·斯佩克特	第7章
arjanen	第9章
arrowd	第14、29、38和59章
arseniiv	第1、12、21和32章
巴尔特克·巴纳切维奇	第5章和第7章
baxbaxwalanuksiwe	第一章
本杰明·霍奇森	第3、4、6、8、10、13、15、18、21、22、23、24、26、50、51、55、62、63和64章
本杰明·科瓦奇	第1、7、14和24章
比利·布朗	第9章
bleakgadfly	第59和60章
布赖恩·敏	第16章
布尔克哈德	第一章
仙人掌	第3、7、8、9、11、24、36、44和52章
carpemb	第19章
卡斯滕	第2章
克里斯·斯特里钦斯基	第34和72章
克里斯托夫·施拉姆	第50章
CiscoIPPhone	第12章
crockeea	第59章
Dair	第19章
达米安·纳达莱斯	第61章
丹尼尔·乔尔	第一章
大卫·格雷森	第10章
Delapouite	第25章
dkasak	第22章
多鲁克	第35章
dsign	第5章
erisco	第1章和第15章
fgb	第18章
菲拉斯·莫阿拉	第15章
加利弗雷语	第1章和第15章
格齐亚杰维奇	第1、6和9章
Itbot	第36章
J·阿特金	第一章
J·亚伯拉罕森	第3、4、5、8、9、10和14章
詹姆斯	第23章和第25章
雅诺什·波特茨基	第7、11、15、19、23、25、28、31、34和48章
jkeuhlen	第22章
约翰·F·米勒	第5、7和11章
朱尔斯	第一章
K48	第7章
卡波尔	第18和28章

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

3442	Chapter 10
Adam Wagner	Chapter 2
Alec	Chapters 18 and 37
alejosocorro	Chapter 1
Amitay Stern	Chapter 1
Antal Spector	Chapter 7
arjanen	Chapter 9
arrowd	Chapters 14, 29, 38 and 59
arseniiv	Chapters 1, 12, 21 and 32
Bartek Banachewicz	Chapters 5 and 7
baxbaxwalanuksiwe	Chapter 1
Benjamin Hodgson	Chapters 3, 4, 6, 8, 10, 13, 15, 18, 21, 22, 23, 24, 26, 50, 51, 55, 62, 63 and 64
Benjamin Kovach	Chapters 1, 7, 14 and 24
Billy Brown	Chapter 9
bleakgadfly	Chapters 59 and 60
Brian Min	Chapter 16
Burkhard	Chapter 1
Cactus	Chapters 3, 7, 8, 9, 11, 24, 36, 44 and 52
carpemb	Chapter 19
Carsten	Chapter 2
Chris Stryczynski	Chapters 34 and 72
Christof Schramm	Chapter 50
CiscoIPPhone	Chapter 12
crockeea	Chapter 59
Dair	Chapter 19
Damian Nadales	Chapter 61
Daniel Jour	Chapter 1
David Grayson	Chapter 10
Delapouite	Chapter 25
dkasak	Chapter 22
Doruk	Chapter 35
dsign	Chapter 5
erisco	Chapters 1 and 15
fgb	Chapter 18
Firas Moalla	Chapter 15
Gallifreyan	Chapters 1 and 15
gdziadkiewicz	Chapters 1, 6 and 9
Itbot	Chapter 36
J Atkin	Chapter 1
J. Abrahamson	Chapters 3, 4, 5, 8, 9, 10 and 14
James	Chapters 23 and 25
Janos Potecki	Chapters 7, 11, 15, 19, 23, 25, 28, 31, 34 and 48
jkeuhlen	Chapter 22
John F. Miller	Chapters 5, 7 and 11
Jules	Chapter 1
K48	Chapter 7
Kapol	Chapters 18 and 28

科斯坦丁·雷布尼科夫	第58章
夸尔茨	第1、5、9、18和46章
leftaroundabout	第1、2、7、9、10、17、42、43、56、65和66章
liminalisht	第26和62章
路易斯·卡西利亚斯	第54章
卢卡·霍尔瓦特	第78章
林恩	第15章和第18章
马兹·布赫	第7章
马里奥·罗曼	第14章、第18章和第41章
mathk	第26章
马修·皮克林	第5、6、11、12和34章
马修·沃尔顿	第15章
mb21	第一章
米克尔	第70和71章
米尔赞·伊尔克古洛夫	第15章
mkovacs	第7章
mniip	第7、17和20章
mnoronha	第1、3、15、22、26、46、52、56和64章
特绍尔德先生	第一章
ocharles	第1章和第69章
ocramz	第67章
pdexter	第2章
pouya	第一章
塞巴斯蒂安·格拉夫	第一章
鞋	第16章
斯特凡·罗兰	第55章
史蒂夫·特劳特	第6章
tlo	第27章和第40章
托克萨里斯	第52章
安德伦	第29章
unhammer	第22章
user239558	第一章
user2407038	第10、14、15、24、43和49章
向量路径	第16章
威尔·内斯	第1、10、12、13、15、16、18、24、25、26、32、35、36、37、43、45、46、51章和64
维萨德	第53章
xuh	第17、34和68章
约什	第7章和第15章
zbw	第1章、第76章和第77章
zeronone	第9章
佐伊·休尔	第73章
λlεx	第1章、第7章、第9章、第10章、第11章、第15章、第18章、第23章、第25章、第26章、第30章、第31章、第33章、第35章、第36章、第38章、第39章、第45章、第47章、第48章、第51章、第55章、第57章、第58章、第64章、第74章和第75章

Kostiantyn Rybnikov	Chapter 58
Kwarrtz	Chapters 1, 5, 9, 18 and 46
leftaroundabout	Chapters 1, 2, 7, 9, 10, 17, 42, 43, 56, 65 and 66
liminalisht	Chapters 26 and 62
Luis Casillas	Chapter 54
Luka Horvat	Chapter 78
Lynn	Chapters 15 and 18
Mads Buch	Chapter 7
Mario Román	Chapters 14, 18 and 41
mathk	Chapter 26
Matthew Pickering	Chapters 5, 6, 11, 12 and 34
Matthew Walton	Chapter 15
mb21	Chapter 1
Mikkel	Chapters 70 and 71
Mirzhan Irkegulov	Chapter 15
mkovacs	Chapter 7
mniip	Chapters 7, 17 and 20
mnoronha	Chapters 1, 3, 15, 22, 26, 46, 52, 56 and 64
Mr Tsjolder	Chapter 1
ocharles	Chapters 1 and 69
ocramz	Chapter 67
pdexter	Chapter 2
pouya	Chapter 1
Sebastian Graf	Chapter 1
Shoe	Chapter 16
Stephane Rolland	Chapter 55
Steve Trout	Chapter 6
tlo	Chapters 27 and 40
Toxaris	Chapter 52
Undreren	Chapter 29
unhammer	Chapter 22
user239558	Chapter 1
user2407038	Chapters 10, 14, 15, 24, 43 and 49
Vektorweg	Chapter 16
Will Ness	Chapters 1, 10, 12, 13, 15, 16, 18, 24, 25, 26, 32, 35, 36, 37, 43, 45, 46, 51 and 64
Wysaard	Chapter 53
xuh	Chapters 17, 34 and 68
Yosh	Chapters 7 and 15
zbw	Chapters 1, 76 and 77
zeronone	Chapter 9
Zoey Hewll	Chapter 73
λlεx	Chapters 1, 7, 9, 10, 11, 15, 18, 23, 25, 26, 30, 31, 33, 35, 36, 38, 39, 45, 47, 48, 51, 55, 57, 58, 64, 74 and 75

You may also like

C
Notes for Professionals
300+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

C#
Notes for Professionals
700+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

C++
Notes for Professionals
600+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

Java
Notes for Professionals
900+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

Perl
Notes for Professionals
90+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

PHP
Notes for Professionals
400+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

Python
Notes for Professionals
700+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

R
Notes for Professionals
400+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books

Ruby
Notes for Professionals
200+ pages
of professional hints and tricks
GoalKicker.com
Free Programming Books