



函数式程序设计

尽享程序设计之奥秘：描述 **what**，推理 **why**，演算 **how**！

作者：胡振江

组织：北京大学

时间：2020 年 7 月 20 日

版本：0.0.1

电子邮件：huzj@pku.edu.cn



你所温柔正确的人总是难以生存，因为这世界既不温柔，也不正确。——比企谷八幡

目录

第一部分 函数式语言的基础	1
1 基本概念	2
1.1 为什么要学习函数式程序设计	2
1.2 什么是函数式程序设计	3
1.2.1 函数和类型	3
1.2.2 函数复合	4
1.2.3 表达式	4
1.3 Haskell 程序设计环境	4
2 基本数据类型	6
2.1 布尔类型: Bool	6
3 递归数据类型	7
4 规约模型及程序效率	8
5 抽象数据类型	9
6 副作用的抽象	10
7 应用 1: 并行计算	11
8 应用 2: 数据分析	12
第二部分 函数式程序的演算基础	13
9 基本概念	14
10 序列类型上的程序演算理论	15
11 递归函数的结构化	16
12 演算规则及其开发	17
13 应用 1: 最优化问题	18
14 应用 2: 并行自动化	19

第三部分 函数式算法的设计	20
15 基本概念	21
16 最小自由数问题	22
17 数独游戏	23
第四部分 多范式程序语言中的函数式思维	24
18 简介	25
19 OCaml	26
20 Scala	27
21 Javascript	28
22 Python	29
第五部分 Appendix	30
23 Hudak	31
24 Nots on Bird's Book	32
第六部分 Thompson	33
25 基本类型及定义	34
25.1 布尔类型	34
25.2 整数类型: Integer 和 Int	35
25.3 字符类型: Char	36
25.4 字符串类型: String	37
25.5 浮点数类型: Float	38
26 函数式程序设计与开发	39
26.1 函数式程序设计	39
27 组合类型	40
27.1 组类型	40
27.2 代数类型	41
27.2.1 枚举类型	41

27.2.2 积类型	42
27.2.3 和类型	42
27.3 列表	43

第一部分

函数式语言的基础

第 1 章 基本概念

内容提要

- 为什么要学习函数式程序设计
- Haskell 程序设计环境
- 什么是函数式程序设计

1.1 为什么要学习函数式程序设计

强调用数学思维能力在程序中的作用。而正是这种能力使得函数式程序设计成为有史以来最棒的程序设计方法。这其中涉及到的数学知识并不复杂，就像我们中学时学过的各种等式理论以及根据这些等式理论进行化简。这使得我们可以从一个简单，明显却不太高效的方法入手，通过应用一些熟知的恒等式进行变换，最后等到一个非常高效的解。这种程序设计方法其乐无穷。

另外，函数式程序设计也能培养和提高简单而清晰地表达计算思想和方法的能力。

函数式程序设计不是屠龙之技。在我上大学的时候，国内几乎没有人搞函数式语言，我的恩师上海交通大学的孙永强教授是为数很少的函数式程序语言研究者之一，由于但是的计算机计算能力的低下，尽管写出的程序简洁漂亮，但是运行起来很慢，给人以一种仅仅存在于某些偏门语言里的学究气的概论。然而我们观察当今的主流语言，会发现函数式程序设计近乎成为一种标配，关键的函数式程序设计的特征或深或浅地嵌入到各式语言中去。

1989 年，John Hughes 曾经写了一篇非常有影响的论文，叫 *Why Functional Programming Matters*，阐述了函数式程序设计的特征及其重要性。函数式语言最主要的特征是用纯函数和高阶函数最简洁地描写计算，和有惰性计算方式来进行计算。这迎合了人们表达问题的能力和高效解决大规模开发问题的需要。

学习一种全新的程序设计方式，困难不在于掌握新的语言，真正考验人的是怎么学会用另一种方式去思考。尽管如此，我们还是使用一种函数式语言，从某种意义上而言，来激励大家用函数式的思维方式。比如，用函数式设计语言，你无法用我们平时离不开的 `loop`，而需要用递归结构来描述重复的计算。

计算机科学的进步经常是间歇式的，好想法有时被搁置数十年后才突然间变成主流。举个例子来说，深度学习是 xxxx 提出来的，... 早年 Java 总被认为太慢，内存消耗太高，不适合于高性能的应用，如今硬件的发展使它成为利用最为广泛的一种语言。

函数式语言的发展轨迹也十分类似，诞生于学院，经过几十年的时间渐渐流行，并浸染了几乎所有现代语言。

在计算机科学短短的发展史上，有时候会从技术主流分流一些树杈，有源于实务的，也有源于学术的。例如，在 20 世纪 90 年代个人电脑大发展的时期，第四代程序设计语言（4GL）出现了爆发式的流行，... 这些语言的最大买点之一是比 C 等第三代语言

(3GL) 有更高层次的抽象，4GL 的一行命令，3GL 需要很多行才能写出来。

Working with Legacy Code 的作者 Michael Feathers 总结出函数式抽象与面向对象抽象的区别：面向对象的程序设计通过封装不确定因素来使代码被人理解；函数式程序设计通过尽量减少不确定因素来使代码被人理解。OOP 用封装，作用域，可见性来控制谁能感知状态，谁能控制状态，使得实现和接口分开。而函数式语言采用了另一种做法；与其建立机制来控制可变状态，不如尽可能消灭可变状态这个不确定因素。假如语言不对外暴露那么多有出错可能的特性，那么开发者就不那么容易出错。

OOP 的世界提倡开发者针对具体问题建立专门的数据结构，并设计处理这个数据结构的一套方法。而函数式语言实现重用的思想完全不同。函数式语言提倡在有限的几种关键数据 (list, set, tree) 上运用针对这些数据结构高度优化过的操作，以此构成基本的运转机制。用户根据需求，插入自己的数据结构和高阶函数去调整机构的运转方式。函数式程序设计的结构很方便我们在比较细小的层面上重用代码。

1.2 什么是函数式设计

函数式程序设计是程序的一种构造方法。它强调函数(function)及其函数作用(function application)。函数式程序设计语言是通过函数来描述计算的语言，而函数式程序设计就是利用函数及其组合来进行程序设计。

函数式程序设计使用简单的数学语言，清晰地描述问题和算法。

函数式语言设计的数学基础简答，支持对程序的性质进行推理 (reasoning)

那么，什么是函数呢？根据维基百科，函数是描述两个集合间的一种对应关系：输入值集合中的每项元素皆能对应唯一一项输出值集合中的元素。我们通常叫这个输出为函数的结果，而这个输入为函数的参数。

1.2.1 函数和类型

对于函数，我们用下面的记号

$$f :: X \rightarrow Y$$

来表示 f 是一个函数，其参数类型为 X ，返回值的类型为 Y 。在这里，类型可以简单地理解某类数据的集合。例如：

$$\sin :: \text{Float} \rightarrow \text{Float}$$

表示 \sin 是一个从浮点数类型到浮点数类型的一个函数。

数学上，我用 $f(x)$ 表示函数 f 作用于其参数 x ，但是在 Haskell 中我们可以省略括号，而使用 $f\ x$ 来表示这个函数作用。例如， $\sin\ 3.14$ 和 $\sin(3.14)$ 都可以表示将函数 \sin 作用于 3.14 。这里需要注意的有两点。一是，函数作用是左结合的，因此 $\log\ \sin\ x$ 表示的是 $\log(\sin\ x)$ ，而不是 $\log(\sin\ x)$ 。二是，函数作用结合最紧密，因此， $\sin\ x + 3$ 表示的是 $(\sin\ x) + 3$ ，而不是 $\sin(x+3)$ 。

1.2.2 函数复合

给定两个函数 $f :: X \rightarrow Y$ 和 $g :: Y \rightarrow Z$ ，我们可以将他们复合成一个新的函数：

$$g \circ f :: X \rightarrow Z$$

该函数将 f 作用于类型为 X 的参数，得到类型为 Y 的结果，然后再将 g 作用于这个结果，最后得到类型 Z 为的结果。

$$(g \circ f) x = f (g x)$$

1.2.3 表达式

表达式 (expression) 是用来表示计算的。每个表达式都有某个类型的值 (value)。

1.3 Haskell 程序设计环境

格拉斯哥 Haskell 编译器 (GHC: Glasgow Haskell Compiler) 是标准的 Haskell 编译器。它将 Haskell 编译成本地代码，支持并行执行，并带有更好的性能分析工具和调试工具。由于这些因素，在本书中我们将采用 GHC。GHC 主要有三个部分组成。

- ghc 是生成本地原生代码的优化编译器。
- ghci 是一个交互解释器和调试器。
- runghc 是一个以脚本形式 (并不要首先编译) 运行 Haskell 代码的程序，

如果你用的是 Windows 或 Mac，强烈推荐你下载 Haskell Platform。在 Linux 上，很多发行版在官方仓库里包含了 Haskell Platform。比如在基于 Debian 的系统中，你可以通过运行 `sudo apt-get install haskell-platform` 来安装。如果你用的发行版没有包含 Haskell Platform，你可以按照 Haskell Platform 网页上的介绍手动安装。

alex 这个重要的包你需要手动更新。Haskell Platform 包含的 alex 是版本 2，而 Yesod 使用的 Javascript 最小化 (minifier) 工具 hjsmin，需要版本 3。一定要在 Haskell Platform 搭建好后运行 `cabal install alex`，否则会有关于 language-javascript 包的报错信息。

Glasgow Haskell Compiler (GHC) 是一个学术 GHCi 是一个广泛使用的用于开发和运行 Haskell 程序的环境。

在 Haskell 语言的众多实现中，有两个被广泛应用，Hugs 和 GHC。其中 Hugs 是一个解释器，主要用于教学。而 GHC(Glasgow Haskell Compiler) 更加注重实践，

在本书中，我们假定你在使用最新版 6.8.2 版本的 GHC，这个版本是 2007 年发布的。大多数例子不要额外的修改也能在老的版本上运行。然而，我们建议使用最新版本。如果你是 Windows 或者 Mac OS X 操作系统，你可以使用预编译的安装包快速上手。你可以从 GHC 下载页面找到合适的二进制包或者安装包。

对于大多数的 Linux 版本，BSD 提供版和其他 Unix 系列，你可以找到自定义的 GHC 二进制包。由于这些包要基于特性的环境编译，所以安装和使用显得更加容易。你可以在 GHC 的二进制发布包页面找到相关下载。

我们在 [附录 A] 中提供了更多详细的信息介绍如何在各个流行平台上安装 GHC。

初识解释器 `ghci` `ghci` 程序是 GHC 的交互式解释器。它可以让用户输入 Haskell 表达式并对其求值，浏览模块以及调试代码。如果你熟悉 Python 或是 Ruby，那么 `ghci` 一定程度上和 `python`，`irb` 很像，这两者分别是 Python 和 Ruby 的交互式解释器。

The `ghci` command has a narrow focus We typically can not copy some code out of a haskell source file and paste it into `ghci`. This does not have a significant effect on debugging pieces of code, but it can initially be surprising if you are used to , say, the interactive Python interpreter. 在类 Unix 系统中，我们在 shell 视窗下运行 `ghci`。而在 Windows 系统下，你可以通过开始菜单找到它。比如，如果你在 Windows XP 下安装了 GHC，你应该从”所有程序”，然后”GHC”下找到 `ghci`。(参考附录 A 章节 Windows 里的截图。)

当我们运行 `ghci` 时，它会首先显示一个初始 banner，然后就显示提示符 `Prelude>`。下载例子展示的是 Linux 环境下的 6.8.3 版本。

```
$ ghci
GHCi, version 6.8.3: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
Prelude>
```

提示符 `Prelude` 标识一个很常用的库 `Prelude` 已经被加载并可以使用。同样的，当加载了其他模块或是源文件时，它们也会在出现在提示符的位子。

第 2 章 基本数据类型

内容提要

- 布尔类型
- 整数类型
- 浮点数类型
- 字符类型
- 字符串类型

这一章主要介绍 **Haskell** 的主要基本类型和相关的基本函数，并说明函数的定义方法。类型可以理解为一个集合。

2.1 布尔类型： `Bool`

最简单的类型是布尔类型，它只包含两个布尔值：`True` 和 `False`。名字中的大写很重要。作用于布尔值得操作符类似于 C 语言的情况：`&&` 表示“逻辑与”，`||` 表示“逻辑或”。

第 3 章 递归数据类型

第 4 章 规约模型及程序效率

第 5 章 抽象数据类型

第 6 章 副作用的抽象

第 7 章 应用 1: 并行计算

程序语言的成功的一个关键是重要的应用。Fortran 用于科学计算。C 并不是很大的创新的语言，但是它被用于 Unix，用于系统开发，特别是操作系统的开发。Cobol 用于商业应用。Java 有用于 WWW 开发。Lisp 用于 AI。

Hudak 通过多媒体上的应用，来说明 Haskell 的成功。Haskell 是非常适合于 DSL 设计和实现。

第 8 章 应用 2: 数据分析

第二部分

函数式程序的演算基础

第 9 章 基本概念

第 10 章 序列类型上的程序演算理论

第 11 章 递归函数的结构化

第 12 章 演算规则及其开发

第 13 章 应用 1: 最优化问题

第 14 章 应用 2: 并行自动化

第三部分

函数式算法的设计

第 15 章 基本概念

第 16 章 最小自由数问题

第 17 章 数独游戏

第四部分

多范式程序语言中的函数式思维

第 18 章 简介

第 19 章 OCaml

第 20 章 Scala

第 21 章 Javascript

第 22 章 Python

第五部分

Appendix

第 23 章 Hudak

第 24 章 Nots on Bird's Book

第六部分

Thompson

第 25 章 基本类型及定义

25.1 布尔类型

布尔类型表示为 `Bool`，包括两个布尔值 `True` 和 `False`。这些值来表示某种测试的结果，如测试两个数字是否相等，第一个数字是否小于第二个数字。布尔类型上的主要操作有逻辑与（`&&`），逻辑或（`||`）和逻辑非（`not`）。这些操作的语义可以用下面的真值表来表示。

t_1	t_2	$t_1 \ \&\& \ t_2$	$t_1 \ \ t_2$
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

t	<code>not t</code>
True	False
False	True

在这些基本操作的基础上，我们可以定义新的逻辑异或操作 `exor`：

```
exOr :: Bool -> Bool -> Bool
exOr x y = (x || y) && not (x && y)
```

第一行说明我们要定义一个函数名为 `exOr` 的操作，而这个操作是接受两个布尔值而返回一个布尔值。第二行给出了具体的定义，他接受两个参数 `x` 和 `y`，利用已定义的操作通过表达式来定义新的操作。在 `Haskell` 中，我们可以省略第一行，但是我们还是鼓励大家写出类型，增加可读性。

```
*Main> exOr True False
True
*Main> exOr True True
False
```

在函数定义式，我们可以在定义的左边用布尔值，增加可读性。比如，`exOr` 也可以定义如下。

```
exOr True y  = not y
exOr False y = y
```

这种定义方式叫做模式匹配（`pattern matching`），我们以后还会具体讨论。

对于新定义的函数，我们可以利用 `Quickcheck` 来测试函数是否满足某个性质。

```
prop_exOr :: Bool -> Bool -> Bool
prop_exOr x y = exOr x y == exOr y x
```

🔥 **练习 25.1** `x` 和 `y` 的异或为真如果 `x` 为真或 `y` 为伪, 或 `x` 为伪或 `y` 为真。根据这个说明给出 `exOr` 的另外一个定义。

🔥 **练习 25.2** 通过模式匹配, 我们可以给予逻辑操作的真值表。完成下面的 `xeOr` 的真值表的定义。

```
exOr True True    = ...
exOr True False   = ...
exOr False True   = ...
exOr False False  = ...
```

25.2 整数类型: `Integer` 和 `Int`

整数类型表示为 `Integer`, 包含所有的整数 (正整数, 副整数, 零), 可以任意大。

```
0
326
-1208
31415926535
```

整数上有我们熟悉的算术运算:

```
+    计算两个整数的和
-    计算两个整数的差
*    计算两个整数的积
div  计算整数除的商  div 10 3 = 3
mod  计算整数除的余数 div 10 3 = 1
```

整数上有下面的关系操作来比较两个整数:


```
>    大于
>=   大于等于
==   等于
/=   不等于
<=   小于等于
<    小于
```

利用上面的基本操作, 我们可以定义整数上的函数。比如, 我们可以定义以下函数来判断三个整数是否相等。

```
equal3 :: Integer -> Integer -> Integer -> Bool
equal3 x y z = (x==y) && (y==z)
```

注 尽管 `Integer` 能表示任意大的整数，但是实现起来会引入一定的开销。对于很多应用，只要整数足够打就可以了（比如可以用固定的 4 个字节来表示）。这时，我们就可以选用 `Int` 类型。`Integer` 和 `Int` 之间可以利用下面的函数进行相互转化。

```
fromInteger :: Integer -> Int
toInteger   :: Int   -> Integer
```

 **练习 25.3** 定义一个函数 `different3` 来判断三个整数是否都不相等。

在定义函数时，我们常常需要根据条件判断选择不同的计算。`Haskell` 的最简单的方法是用条件表达式

```
if condition then m else n
```


表示如果 `condition` 是 `True` 就返回 `m`，否则返回 `n`。比如，

```
max :: Integer -> Integer -> Integer
max x y = if x >= y then x else y
```

这里定义了函数 `max` 来选择两个输入中大的一个数。如果分歧条件比较多，或者条件比较复杂时，我们可以选用 `guard` 形式来更清晰地表述条件计算。

```
max :: Integer -> Integer -> Integer
max x y
  | x >= y    = x
  | otherwise = y
```

这个定义中有两个 `guard`，从上往下执行。如果第一个 `guard x>=y` 成立，它就返回 `x`，否则，它就是下一个 `guard`。`otherwise` 是一个特殊 `guard`，它总是成立。

 **练习 25.4** 定义一个函数返回三个整数的最大值。

25.3 字符类型: Char

字符类型包括所有字符，而字符使用单引号围绕着，如 `'a'` 在 `Haskell` 中表示字符 `a`，`'3'` 表示数字字符 `3`。还有一些特殊的字符：

```
'\t'  tab
'\n'  换行
'\'   反划线
'\''  单引号
'\"'  双引号
```

有一种标准的编码方式叫做 `ASCII` 编码，将字符对应于一个整数。下面两个函数表示了字符与整数之间的变换。

```
fromEnum :: Char -> Int
toEnum   :: Int  -> Char
```


在 ASCII 编码中, 大写字符, 小写字符, 数字都是连续从下到大排列的。利用这个性质, 我们可以对字符进行变换。设想我们希望定义一个函数将字符变成大写。为此, 我们首先计算大写字母和小写字母之间的间隔:

```
offset :: Int
offset = fromEnum 'A' - fromEnum 'a'
```


然后我们就可以定义这个函数:


```
toUpper :: Char -> char
toUpper c = toEnum (fromEnum c + offset)
```

字符按照他们的编码是可以比较的, 因此, 我们可以定义下面的函数来判断一个字符是不是数字。

```
isDigit :: Char -> Bool
isDigit c = ('0' <= c) && (c <= '9')
```

注 `toUpper` 和 `isDigit` 都是标准库 `Data.Char` 的标准函数。

 **练习 25.5** 定义一个将小写字符改为大写字符, 但是其他字符保持不变的函数。

 **练习 25.6** 定义一个将数字字符变为相应数字的函数。

```
char2Int :: Char -> Int
```

25.4 字符串类型: `String`

字符串类型 `String` 是有字符的序列构成。每个字符串用双引号括起来, 如 `"This is a string!"`。字符串上面重要的一个操作是字符串链接 `++`, 如, `"string" ++ "join" = "string join"`。由于字符串是字符的序列, 我们将来可以用序列上的函数来定义字符串上的新函数。

对于字符串, 有两个很有用的函数。一个是 `read`, 将一个字符串变为你想要的类型的值, 另一个是 `show`, 将一个类型的值变为字符串。

```
read :: String -> a
show :: a -> String
```


例如

```
read "True" = True
read "3"    = 3
show True   = "True"
show 3      = "3"
```

 **练习 25.7** 定义函数

```
onTwoLines :: String -> String -> string
```

使得它能接受两个字符串，返回一个字符串。这个字符串打印时能将两个字符串上下排列。

 **练习 25.8** 定义一个函数 `romanDigit` 将一个数字字符用罗马数字表示的程序, 如 `romanDigit '7' = "VII"`。


25.5 浮点数类型: Float


浮点数类型 `Float` 包含所有的浮点小数, 如

```
0.326
-12.08
31415.926
```

Haskell 库函数包括很多浮点数上的函数, 如平方根, 指数函数, 对数函数, 三角函数等。同时, 整数可以通过 `fromInt` 变为浮点数, 浮点数也可以通过 `ceiling`, `floor` 和 `round` 变成整数。下面是一些执行的例子。

```
Prelude> sin (pi/2) + sqrt 2
2.414213562373095
Prelude> floor 5.6
5
Prelude> ceiling 5.6
6
Prelude> round 5.6
6
```

 **练习 25.9** 给定三条边的长度为 a , b , c , 定义一个函数, 判定这三条边能否组成一个三角形。

 **练习 25.10** 假设 a, b, c 为二次方程 $ax^2 + bx + c = 0$ 的系数。定义一个函数, 求出这个方程解的个数。

第 26 章 函数式程序设计与开发

26.1 函数式程序设计

在写函数式程序之前，我们必须进行设计。

理解我们需要做什么

设计的第一步是确认我们是否理解了我们需要做什么。通常，需要我们解决的问题的描述是非形式化的，描述不完全或可能无解。比如说，我们需要返回三个数字的中间数。显然，如果给出的三个数是 2, 4, 3，其结果应该为 3，但是如果三个数是 2, 4, 2，那么结果应该是什么呢？你也可能说是 2，因为如果我们将这三个数排序，我们会得到 2, 2, 4，但是你也可以说无解，因为这三个数的最大值是 4，最小值是 2，而没有这两者之间的数。

在下面的讨论中，我们将考虑第一种选择。

清楚它的类型

设计函数时，我们应该清楚它的类型。对于上面的例子，我们可以写出下面的类型。

```
middleNumber :: Integer -> Integer -> Integer -> Integer
```

很显然，我们在给出定义时，如果它不满足上面的类型，他一定不会是我们要定义的函数。

我们已经有哪些可利用的信息


往往我们不需要从头开发一个程序。比如我们已经有了函数 `bigger` 和 `smaller` 能求出两个数字的最大值和最小值。这时，我们就可以轻松地定义上面的函数：

```
middleNumber x y z = smaller (bigger x y) z
```

将问题分割为简单的问题

如果我们一下子无法解决问题，我们可以试探着将它分解为简单的问题。然后从解决简单的问题着手。这时，我们可以问自己，假如我已有我想要的那个函数，那么我怎样解决问题。对于 `middleNumber`，我们可以说，如果我们有个韩式 `between x y z` 能判断 `y` 是中间数字，那我们可以写出下面的程序。

```
middleNumber x y z
  | between y x z = x
  | between x y z = y
  | otherwise    = z
```

 **练习 26.1** 给出 `between` 的函数定义。

第 27 章 组合类型

上一章我们介绍了基本类型及上面的操作。本章介绍两种构造组合类型的方法，组类型 (tuple) 和列表类型 (list)。组类型和列表类型都能将一些数据组合在一起，但是有所不同。一个组是组合一定个数的确定类型的元素，而列表是组合不定个数的同一个类型的元素。

为了看清楚差异，我们看一个超市模型。一个超市有很多物品，包括名字和它的价格，如

```
("Salt: 1kg", 10)
("Crisps", 5)
```

这些物品具有组类型 (String, Int)：我们用 String 来表示名字，用 Int 来表示价格。现在一个超市包括很多物品，我们可以用一个列表类型 [(String, Int)] 来表示。每一个元素表示一个物品，如

```
[("Salt: 1kg", 10), ("Crisps", 5), ("Sugar", 5)]
```

27.1 组类型

一个组类型是由很多简单的类型组合而成。一个组类型

$$(t_1, t_2, \dots, t_n)$$

包含下面的值

$$(v_1, v_2, \dots, v_n)$$

其中, $v_1 :: t_1, v_2 :: t_2, \dots, v_n :: t_n$. 为了增加可读性，我们可以给组类型一个名字，例如，对于超市的物品类型，我们可以定义为：

```
type ShopItem = (String, Int)
```

我们可以用组类型来定义一个函数返回一组值。比如下面的函数返回两个整数的最小值和最大值。

```
minAndMax :: Integer -> Integer -> (Integer, Integer)
minAndMax x y
  | x >= y    = (y,x)
  | othersise = (x,y)
```

给定一个组类型的值，我们可以通过模式匹配取出其中的元素。比如，我们定义一个函数，将一个二元组的两个整数相加。

```
addPair :: (Integer, Integer) -> Integer
addPair (x,y) = x + y
```

Haskell 里面提供了两个已定义的投影函数来取出二元组的元素。

```
fst (x,y) = x
snd (x,y) = y
```

下面一个例子是用二元组来计算斐波那契数列：

$$0, 1, 1, 2, 3, 5, \dots, u, v, u + v, \dots$$

除了开始的两个数字以外，后面的任何一个数都是他的前面的两个数的和。我们首先定义一个函数，给定 n ，它计算出两个连续的斐波那契数的二元组。

```
fib2 n = (fib n, fib (n+1))
```

给我一个连续的斐波那契数二元组 u, v ，我们可以通过下面的函数计算出下一个二元组：

```
fibStep (u,v) = (v, u+v)
```


因此，我们可以定义 `fib2` 如下：

```
fib2 :: Integer -> (Integer, Integer)
fib2 n
  | n==0      = (0,1)
  | otherwise = fibStep (fib2 (n-1))
```

利用 `fib2`，我们可以定义 `fib` 如下：

```
fib2 = fst . fib2
```

这里我们用了函数合成“.”将两个函数合成到一起。

 **练习 27.1** 定义一个函数计算某个直线与 x 轴的交叉点。

27.2 代数类型

代数类型给用户提供了一个定义新类型的方法。下面我们通过例子来说明代数类型的定义方法。

27.2.1 枚举类型

枚举类型顾名思义就是将类型中的元素通过枚举的方式来定义。比如，拳头/剪子/布的游戏的出拳的类型可以定义为：

```
data Move = Rock | Scissors | Paper
```

在枚举类型上的函数可以通过模式匹配，定义如何操作每一个元素。下面的函数定义了两个不同出拳的分数：

```

score :: Move -> Move -> Integer
score Rock Rock      = 0
score Rock Paper     = -1
score Rock Scissors  = 1
score Paper Rock     = 1
score Paper Paper    = 0
score Paper Scissors = -1
score Scissors Rock  = -1
score Scissors Paper = 1
score Scissors Scissors = 0

```

注 我们前面介绍的布尔类型内部就是通过枚举类型来定义的。

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

27.2.2 积类型

想组类型一样，积类型描述将几个不同类型的元素组合起来形成一个新的类型。比如，下面定义了一个人的类型。

```
data People = Person Name Age
```

其中 `Name` 和 `Age` 分别是 `String` 和 `int` 的代名词：

```

type Name = String
type Age  = Int

```

这里 `People` 是类型名，`Person` 是数据构造子。构造子 `Person` 可以理解为一个函数

```
Person :: Name -> Age -> People
```

它接受一个字符串的名字和一个整数的年龄来构造一个人的信息。例如，下面是 `People` 类型中的一些元素：

```

Person "Mary" 28
Person "Bob"  12

```

像枚举类型一样，积类型上的函数也是通过模式匹配的方式来定义的。

```

showPerson :: People -> String
showPerson (Person n a) = "Name: " ++ show n ++ " Age: " ++ show a

```

27.2.3 和类型

和类型是为了表示不同的选择的类型。比如，一个形状类型可能是圆形，有个半径，或是一个矩形，有高和宽。这可以定义为：

```

data Shape = Circle Float
           | Rectangle Float Float

```

这个类型可以包括以下的一些值：

```
Circle 3.0
Rectangle 10.0 20.0
```

在这样的类型上的函数也是通过模式匹配自然地定义。

```
isRound :: Shape -> Bool
isRound (Circle _)      = True
isRound (Rectangle _ _) = False

area :: Shape -> Float
area (Circle r)          = pi * r * r
area (Rectangle h w)    = h * w
```

- 🔪 **练习 27.2** 定义一个函数，求一个形状（`Shape`）的周长。
- 🔪 **练习 27.3** 将 `Shape` 类型扩充，增加一个三角形，并定义对应的 `isRound` 和 `area` 函数。
- 🔪 **练习 27.4** 将 `Shape` 类型扩充为 `NewShape`，使得圆形包括圆心，长方形包括对角顶点坐标。定义一个函数

```
move :: Float -> Float -> NewShape -> newShape
```

使得 `move x y s` 将形状 `s` 在横向平移 `x` 在纵向平移 `y`。

- 🔪 **练习 27.5** 定义一个函数判定两个 `NewShape` 的形状是否重叠。

27.3 列表

列表是一个函数式程序审计中非常重要的类型，就像数学里集合的重要性一样。一个列表表示同类型元素的序列。对于每个类型 `t`，列表类型 `[t]` 表示元素类型为 `t` 的序列。

```
[1,2,3,4] :: [Integer]
[True, False] :: [Bool]
['a','b','b'] :: [Char]
[(+), (-), (*)] :: [Integer -> Integer]
[[1,2],[2,3],[]] :: [[Integer]]
```

字符串 `String` 是字符的列表 `[Char]`。

- 🔪 **练习 27.6** 说明集合与序列的区别。

列表上有一些非常有用的略写。

- `[m..n]` 表示 `[m,m+1,...,n]`。如果 `m` 超过 `n` 则返回空列表。

```
[1..5] = [1,2,3,4,5]
[3.1 .. 7.0] = [3.1,4.1,5.1,6.1,7.1]
['a'..'e'] = "abcde"
```

- `[m, p..n]` 表示 `[m,p,p+s,...,p+ks]`，其中 `p+ks <= n < p+(k+1)s`。

```
[7,6..2] = [7,6,5,4,3,2]
[0.0,0.3 .. 1.0] = [0.0,0.3,0.6,0.8999999999999999]
['a','c'...'n'] = "acegikm"
```

对于集合我们有个非常有用的记法叫做集合闭包：

```
{ x | x <- {1,2,3,4}, isEven x }
```

表示从集合 1,2,3,4 中取出偶数而得到集合 [2,4] 。对于列表，我们有类似的记法，叫作列表闭包（list comprehension）。

```
[ x | x <- [1,2,3,4], isEven x ]
```

返回列表 [2,4] 。

下面给出一些例子，说明列表闭包非常强大的表现能力。

```
[ 2*n | n <- [2,4,7]]
[ isEven n | n <- [2,4,7]]
[ 2*n | n<- [2,4,7], isEven n, n>3]
```