Haskell 程序设计

著：Graham Hutton
译：Tony Bai[1]

October, 2010

[1] http://bigwhite.blogbus.com

Haskell 程序设计

待译

版权声明

由于尚未获得原作者授权，所以本项目仅用于学习和交流之用，不得用于任何商业用途。

序

> ... 这个世界上有两种设计软件的方法：一种是使设计尽量的简化，以至于明显没有任何缺陷；而另一种是使设计尽量的复杂，以至于找不到明显的缺陷。第一种方法更加困难。
>
> *-- Tony Hoare，1980 ACM 图灵奖演讲*

这本书讲述了一种以简单、清晰和优雅为关键目标的编程方法。更具体的说，它是一本使用 Haskell 语言介绍这种函数式编程方法的入门书籍。

函数式编程与当前大多数编程语言比如 Java，C++，C 和 Visual Basic 所提倡的风格相差很大。特别是当前大多数语言是与底层硬件紧密关联的，在这种意义上，编程的基本思想就是修改存储的值。与此相反，Haskell 则提倡一种更为抽象的编程风格，这种风格的基本思想则是将函数应用于参数。正如我们将要看到的，站上更高的层次可以让我们拥有更为简单的程序，并且支持一系列强大的构造和推导程序的新方法。

本书主要面对具备大专水平的学习计算机科学的学生，但也同样广泛适用于那些想了解和学习 Haskell 编程的读者。你不需要拥有任何编程经验，所有的概念都从基本原理讲起，并伴以精心挑选的例子。

本书使用的 Haskell 版本是 Haskell 98，Haskell 语言的标准版本，Haskell 设计者花费了 15 年才最终发布了这个标准。由于这里仅仅是入门引导，所以我们不会去涉及 Haskell 语言以及其相关库的所有方面。本书大约有一半内容是专门介绍该语言的主要特点的，另一半则包括了 Haskell 编程的例子和案例研究。每个章节还包括了一系列的习题以及关于进一步阅读更高级、更专业主题的建议。

本书基于课程材料编写而成，这些课程材料在诺丁汉大学经过了多年的改进和课程测试。通过 20 学时的授课以及大约 40 小时的自学、实验室实践和编程作业，你就可以学完本书的大部分内容。然而，你还需要更多的时间详细学习一下后面的一些章节以及一些编程例子。

本书的官方网站提供了一系列辅助资料，包括每个章节的幻灯片和一些扩展例子的 Haskell 代码。教师还可以通过发送电邮到 solutions@cambridge.org 得到每个章节练习题的标准答案以及大量带有标准答案的试题集。

致谢

诺丁汉大学编程社团为函数式编程的研究与教学提供了一个极好的环境。感谢学校提供假期让我可以进行本书的写作。感谢所有学生和讲师们关于我的 Haskell 课程的反馈; 感

谢 Thorsten Altenkirch, Neil Ghani, Mark Jones (现在在波特兰州), Conor McBrideh 和 Henrik Nilsson 在 FOP 社区与我进行的关于函数式编程的想法以及如何表达这些想法的交流和讨论。

我还要感谢 David Tranah 和 Dawn Preston 在剑桥大学出版社出色的编辑工作; 感谢 Mark Jones 的 Haskell 解释器; 感谢 Ralf Hinze 和 Andres Loh 提供的 lhs2TeX 排版系统; 感谢 Rik van Geldrop 和 Jaap van der Woude 关于使用本书草稿的反馈; 感谢 Kees van den Broek, Frank Heitmann 和 Bill Tonkin 指出的本书的错误; 感谢 Ian Bayley 和所有匿名评审者极富价值的评论; 感谢 Joel Wright 提供的倒计时程序。

Graham Hutton
诺丁汉, 2006

译者序

"A language that doesn't affect the way you think about programming, is not worth knowing".

-- Alan Perlis（ACM 第一任主席，图灵奖得主，1922-1990）

《程序员修炼之道》一书作者建议程序员每年应至少学习一门新的语言，以拓宽思维，避免墨守成规。今年我选择了函数式编程语言 Haskell。选择 Haskell 的理由正如 Alan Perlis 所说的那样，Haskell 是一门可以影响程序员编程思维的语言，我也期望通过学习 Haskell 来拓宽我的思维。

开始接触 Haskell 后，我才发现它在国内是如此的小众 (其实在国际上也很小众)，国内居然没有正式出版过 Haskell 相关的中文书籍[1]，唯一可参考的像样的中文资料就是网上流传的一本免费的由乔海燕翻译的《Yet Another Haskell Tutorial》，国内出版的影印版书籍似乎也只有《真实世界的 Haskell》(英文名：Real World Haskell) 这一本。

我开始学习 Haskell 时用的就是那本曾获得过 Jolt Award 大奖的《Real World Haskell》影印版，书很厚，是本 Haskell 大全。但后来发现似乎不太适合初学者。随后又在网上搜索资料，找到了 Graham Hutton 编写的《Programming in Haskell》[2]这本教程。与《Real World Haskell》比起来，《Programming in Haskell》这本书就显得"单薄"了许多，加起来总共不到 200 页。不过这本书却非常适合函数式编程和 Haskell 的初学者，因为这本书是基于英国诺丁汉大学课程讲义编制而成，经过了多年实际教学检验，并且在这本书的官方主页上还可以下载到与书配套的讲义幻灯片和习题答案。

同样是也是在这本书的主页上，我发现了这本书在 2009 年就已经出版了日文版和韩文版，这个让我很是受触动，为什么在好书引进方面我们也落后于日韩呢！突然脑中迸发出一个念头：要不我来试试翻译一下这本书，也算是为 Haskell 在中国的发展做出一些自己的贡献。

于是在 Google Code 上建立了这个《Programming in Haskell》中文版翻译项目[3]。

真诚的欢迎大家提出建议和意见，帮助我来审校翻译中存在的问题，共同完成这个项目。

另外这里需声明一点：自己仅是一个 Haskell 爱好者和初学者，非 Haskell 牛人。请大家读译稿后谨慎拍砖！

Tony Bai

October, 2010

---

[1] 据说 Real world Haskell 正在被翻译中

[2] http://www.cs.nott.ac.uk/~gmh/book.html

[3] http://code.google.com/p/programming-in-haskell-cn

中英文对照表

英文术语	中文翻译
apply...to...	把... 应用于...
arity	元数
component	元素
domain-specific language	领域专用语言
equation	等式
evaluate	求值
feature	特点
guards	守卫
high-order	高阶
lazy evaluation	惰性求值
list	列表
list comprehensions	待译
monad	待译
overloaded	重载的
pair	二元组
pattern matcing	模式匹配
polymorphic	多态的
primitive type	原子类型
program	程序
programming	编程
reason about	推导
section	段
singleton	单件
take arguments	接受参数
triple	三元组
tuple	元组
type inference	类型推断

目 录

第 1 章　导言

在这一章节中，我们为本书的后续部分展开打好了基础。我们从回顾函数的概念开始，然后介绍函数式编程的概念，总结 Haskell 的主要特点和它的历史，最后通过两个小例子"品尝"一下 Haskell 的味道。

1.1　函数

在 Haskell 中，一个*函数*是一个映射，它接受一个或多个参数，并产生唯一一个结果。我们可以通过一个等式来定义函数，等式中包含函数名、参数名以及详细描述如何依据参数计算出最终结果的函数体。

例如，一个函数 $double$，接受一个数字 x 作为参数，产生的结果为 $x + x$，它可通过如下等式定义：

$$double\ x = x + x$$

当一个函数被应用于实际参数时，其结果可通过将实际参数替换函数体中的参数名的方式获得。这个过程可能会立即产生一个不能被进一步简化的结果，比如一个数字。而更为常见的情况是，这个结果是一个含有其他函数程序的表达式，我们必须以同样的方式处理这个表达式才能得到最终的结果。

例如，程序 $double\ 3$ 将函数 $double$ 应用于数字 3 的结果可通过如下计算过程得出，每一步计算通过花括号里简短注释解释：

$double\ 3$
= {applying $double$}
$3 + 3$
= {applying +}
6

同样，两次应用 $double$ 的内嵌程序 $double\ (double\ 2)$ 的结果可以通过如下计算过程得出：

$double\ (double\ 2)$
= {applying the inner $double$}
$double\ (2\ +\ 2)$
= {applying + }
$double\ 4$
= {applying $double$ }
$4 + 4$

= {applying +}
8

另外，同样的结果也可以通过先从外层的函数 $double$ 开始计算获得：

double (double 2)
= {applying the outer $double$}
$double\ 2\ +\ double\ 2$
= {applying the first $double$}
$(2\ +\ 2)\ +\ double\ 2$
= {applying the first + }
$4\ +\ double\ 2$
= {applying $double$}
$4\ +\ (2\ +\ 2)$
= {applying the second +}
$4 + 4$
= {applying + }
8

但是，这个计算过程比我们原来的版本多出两步，因为表达式 $double\ 2$ 在第一步中被复制了一份并因此被化简了两次。一般来说，函数在计算过程中应用的顺序不会影响最终的结果值，但它可能会影响到所需步骤的数量，并可能影响计算过程是否终止的判断。本书第 12 章针对这些问题作了更为详细的探究。

1.2 函数式编程

什么是函数式编程？见仁见智，很难给出一个确切的定义。但总的来说，函数式编程可以被看作一种*编程风格*，这种风格的基本计算方式是将函数应用于实际参数。相应的，一门函数式编程语言就是*支持*和*鼓励*使用函数式风格的计算机编程语言。

为了说明这些概念，让我们考虑一个计算从 1 到 n 的整数和的任务吧。在当前大多数编程语言中，这个任务通常可以通过使用两个可随时改变的存储值变量实现，一个变量从 1 变到 n，另外一个变量用来累加总数。

例如，如果我们使用赋值符号:= 来改变一个变量的值，使用关键字 **repeat**和 **until**来反复执行一个指令序列，直到某个条件被满足，然后下面的指令序列会计算出所需的总和：

$count := 0$
$total := 0$
repeat
  $count := count + 1$
  $total := total + count$
until
  $count = n$

也就是说，我们首先将 counter 和 total 这两个变量初始化为零，然后反复递增 counter，并把这个值与变量 total 相加，直到 counter 达到 n，此时计算过程停止。

在上述程序中，计算的基本方法是改变存储的值，在某种意义上说，程序执行就是一系列的赋值操作。例如， $n = 5$ 时我们得到如下序列，其中最后赋给变量 $total$ 的值就是所需的总和：

$count := 0$
$total := 0$
$count := 1$
$total := 1$
$count := 2$
$total := 3$
$count := 3$
$total := 6$
$count := 4$
$total := 10$
$count := 5$
$total := 15$

通常，这种以改变存储值为基本计算方式的编程语言被称为*命令式语言*，因为用这类语言编写的程序由一系列命令式指令构成，这些指令精确描述了计算过程应该如何进行。

现在让我们考虑使用 Haskell 来计算从 1 到 n 的整数和。这通常可使用两个库函数实现，一个是 $[\,..\,]$，用于产生从 1 到 n 之间的数字列表; 另外一个是 sum，用于针对这个列表求和。

$sum[\,1..n\,]$

在这个程序中，计算的基本方法是将函数应用于参数。在这个意义上，程序的执行过程实际上是一系列的函数应用。比如当 $n = 5$ 时，我们得到如下序列，最终结果就是我们所需要的总和：

$sum[\,1..5\,]$
= { applying $[\,..\,]$ }
$sum[1,\ 2,\ 3,\ 4,\ 5]$
= { applying sum }
$1 + 2 + 3 + 4 + 5$
= { applying + }
15

大多数命令式语言都支持一些使用函数编程的形式，所以 Haskell 程序 $sum[\,1..n\,]$ 可以被转化成这些语言。但是，大多数命令式语言不鼓励使用函数式风格编程。比如，大多数语言不鼓励或禁止函数被存储在类似列表的数据结构中，禁止构建类似上面例子中数字列表那样的中间结构; 禁止接受函数作为参数或将函数作为返回值，禁止根据自己定义自己。相反，Haskell 在如何使用函数上没有这些限制，并且提供了一系列功能特点，使得使用函数进行编程既简单又强大。

1.3 Haskell 的特点

作为参考，Haskell 的主要功能特点都列在了下面，并伴随有提供进一步细节的章节号。

- 简明的程序 (第二章和第四章)

  由于函数式风格抽象层次高的本质，使用 Haskell 编写的程序往往比用其他语言更加*简明*，正如上一节例子中说明的那样。此外，Haskell 的语法设计充分考虑了简明的特点，尤其是拥有较少的关键字，并允许使用缩进来表明程序结构。虽然很难作出客观的比较，但 Haskell 编写的程序往往比用当前其他语言编写的程序短小 2-10 倍。

- 强大的类型系统（第三章和第十章）

  大多数现代编程语言都包含某种形式的*类型系统*来检测不兼容错误，如试图将一个数字和一个字符相加。Haskell 有一个类型系统，它仅从程序员那里获取很少量的类型信息，但却可以在程序执行之前使用一种被称为类型推断的过程自动检查出大量不兼容的错误。Haskell 的类型系统也比大多数现代编程语言更为强大，它允许函数是"多态的"和"重载的"。

- List Comprehensions（第五章）

  在计算中一种最常见的构造和操作数据的方法就是使用列表。为此，Haskell 提供列表作为语言的一种基本概念，并连同一个简单但功能强大的 *comprehension*记法，使用这些符号可以从已有列表中选择或过滤元素来构建新的列表。comprehension 记法的使用使得列表上许多公共函数以一种清晰、简明的方式定义出来，而不需要显式的递归。

- 递归函数（第六章）

  大多数实用程序都包含一些形式的重复或循环。在 Haskell 中，实现循环的基本机制是使用根据自己定义自己的*递归函数*。许多计算都能用递归函数给出一个简单和自然的定义，特别是使用"模式匹配"和"守卫"将不同情况分成不同等式时。

- 高阶函数（第七章）

  Haskell 是一门*高阶*函数式编程语言，这意味着在函数定义中你可以自由将函数作为参数和结果返回值。使用高阶函数接受常见的编程模式，例如组合两个函数或定义作为语言自身的函数。更常见的是在 Haskell 中高阶函数可以用于定义"领域专用语言"，比如列表处理、解析以及交互式编程。

- Monadic 作用（第八章和第九章）

  Haskell 中的函数都是纯函数，它们接受所有输入作为参数，将所有输出作为结果返回。但是，许多程序需要某种形式的*副作用*，这似乎与纯洁性有冲突，比如程序运行时从键盘读取输入或输出结果到屏幕。Haskell 提供了一个不损害函数纯洁性的基于*monad*数学概念的处理副作用的统一框架。

- 惰性求值（第十二章）

Haskell 程序的执行使用了一种叫*惰性求值*的技术，这种技术的基本思想是直到其结果是实际需要的时候，计算才应该被执行。除了避免不必要的计算，惰性求值保证程序适时结束，鼓励以使用中间数据结构的模块式风格进行编程，甚至允许使用拥有无穷元素个数的数据结构，比如一个无穷的数字列表。

- 程序推导

  因为在 Haskell 中程序是纯函数，所以简单的*等式推导*可用于执行程序，变换程序，证明程序属性，甚至能够从他们的行为规范中直接提取出程序。在结合使用归纳方法对递归函数进行推导时，等式推导尤为强大。

1.4 历史背景

Haskell 的许多特点并非首创，都是由其他语言首次引入的。为了帮助大家了解 Haskell 的背景，下面简要总结一下有关 Haskell 语言的一些主要的历史性的发展：

- 20 世纪 30 年代，Alonzo Church 发明了 lambda 演算，一种简单但功能强大的数学函数理论。
- 20 世纪 50 年代，John McCarthy 发明了 Lisp(列表处理器)，Lisp 被公认为是世上第一种函数式编程语言。Lisp 许多方面受到了 lambda 演算的影响，但同时仍然接受变量赋值作为语言的一个核心特征。
- 20 世纪 60 年代，Peter Landin 发明了 ISWIN(“If you See What I Mean”)，第一种纯函数式编程语言，它主要基于 lambda 演算，并且没有变量赋值。
- 20 世纪 70 年代，John Backus 发明了 FP("Fuctional Programming")，一种特别强调高阶函数和程序推导思想的语言。
- 同样也是在 20 世纪 70 年代，Robin Milner 和其他人一起开发了 ML(元语言)，第一种现代函数式编程语言，引入了多态类型和类型推断思想。
- 20 世纪 70 年代和 80 年代，David Turner 开发出许多惰性的函数式编程语言，最终造就了可获得商业支持的 Miranda (意为" 令人敬佩的") 语言的出现。
- 1987 年，一个国际研究委员会发起开发 Haskell 语言 (以逻辑学家 Haskell Curry 命名)，一个标准的惰性函数式编程语言。
- 2003 年，该委员会公布了 Haskell 的报告，报告中定义了一个期待已久的 Haskell 的稳定版本，该版本是该语言设计者们十五年工作的成果。

值得注意的是，上面提到的三个研究人员 - McCarthy, Backus 和 Milner 各自获得了等同于计算机领域诺贝尔奖的的 ACM 图灵奖。

1.5 品尝 Haskell

我们在结束本章之前通过两个小例子来品尝一下 Haskell 编程。首先我们回顾一下本章前面使用的函数 sum， sum 用于计算列表中一组数字的和。在 Haskell 中，这个函数可通过如下两个等式定义：

$$sum[\,] = 0$$
$$sum(x : xs) = x + sum\ xs$$

第一个等式定义一个空列表的总和是零，同时第二个等式定义一个非空列表的总和是由列表中的第一个数字和后续数字组成的列表 xs 的总和相加在一起获得的。例如，$sum\ [1,\ 2,\ 3]$ 的总和计算过程如下：

$sum\ [1,\ 2,\ 3]$
= { applying sum }
$1 + sum\ [2,\ 3]$
= { applying sum }
$1 + (2 + sum\ [3])$
= { applying sum }
$1 + (2 + (3 + sum[\,]))$
= { applying sum }
$1 + (2 + (3 + 0))$
= { applying + }
6

注意，即使函数 sum 使用自身定义自己而形成了递归，它也不会永远循环下去。尤其是每个 sum 都将列表参数的长度减一，直到列表最终变为空表，递归过程也随之终止。将零作为空表的总和再合适不过，因为加法里零不改变加法结果，也就是说对于任何数字 x，$0 + x = x$ 且 $x + 0 = x$。

在 Haskell 中，每个函数都有一个描述参数和返回值的*类型*，这个类型会自动从函数的定义中推断出来。例如，函数 sum 有以下类型：

$$Num\ a \Rightarrow [\,a\,] \to a$$

这个类型指出对于任何数字类型 a， sum 是一个将一组这样的数字列表映射到一个单一数字的函数。Haskell 支持许多不同类型的数字，其中包括整数，如 123，“浮点数”，如 3.14159。因此， sum 可以应用于一个整数列表，正如在上面的计算过程那样，也可以应用于一个浮点数列表。

类型提供了有关函数本质的有用的信息，但更为重要的是，类型的使用使得许多错误可以在程序被执行之前被自动检查出来。尤其是，对于一个程序中的每个函数都会检查其实际参数类型与函数本身的类型是否兼容。例如，试图将函数 sum 应用于一个字符列表将报错，因为字符不是数字类型。

现在让我们考虑一个关于列表的更为有趣的函数吧，这个函数说明了 Haskell 其他一些方面的特性。假设我们定义了一个名为 $qsort$ 的函数，它由以下两个等式构成：

$$qsort[\,] = [\,]$$

$qsort(x:xs) = qsort\ smaller\ ++ [x]\ ++ qsort\ larger$

where

smaller = $[a \mid a \leftarrow xs,\ a \leq x]$

larger = $[b \mid b \leftarrow xs,\ b > x]$

在这个定义里，+是一个连接两个列表的操作符。例如 $[1,\ 2,\ 3] ++ [4,\ 5] = [1,\ 2,\ 3,\ 4,\ 5]$。相应的，**where**是一个关键字，用于引入局部定义。在这个例子中，*smaller*列表由 xs 列表中所有小于等于 x 的元素组成，同时 larger 列表由 xs 列表中所有大于 x 的元素组成。例如，如果 $x = 3$ 且 $xs = [5,\ 1,\ 4,\ 2]$，那么 $smaller = [1,\ 2]$， $larger = [5,\ 4]$。

$qsort$ 究竟做了什么？首先我们要明确它对仅有一个元素的列表不起作用，在这种意义上，对于任何 x， $qsort\ [x] = [x]$:

$qsort\ [\ x\ ]$
= {applying $qsort$}
$qsort\ [\ ] ++ [x] ++ qsort\ [\ ]$
= {applying $qsort$}
$[\ ]\ ++ [x]\ ++ [\ ]$
= {applying ++ }
$[x]$

相应的，现在我们将 *qsort*应用到一个样例列表，使用上面的定义化简计算过程：

$qsort\ [3,\ 5,\ 1,\ 4,\ 2]$
= {applying $qsort$}
$qsort\ [1,\ 2] ++ [3] ++ qsort\ [5,\ 4]$
= {applying $qsort$}
$(qsort\ [\ ]\ ++ [1]\ ++ qsort\ [2])\ ++ [3]$
$++\ (qsort\ [4]\ ++ [5]\ ++ qsort\ [\ ])$
= {applying $qsort$, above property}
$([\ ]\ ++ [1]\ ++ [2])\ ++ [3]\ ++ ([4]\ ++ [5]\ ++ [\ ])$
= {applying ++}
$[1,\ 2]\ ++ [3]\ ++ [4,\ 5]$
= {applying ++}
$[1,\ 2,\ 3,\ 4,\ 5]$

总之， $qsort$ 将示例列表按照数字顺序进行了排序。更一般地说，这个函数产生了一个任意数字列表的排序版本。 $qsort$ 的第一个等式表明空列表是已经被排序了的，而第二个等式则表明任何非空列表都可以通过将第一个数字插入两个列表之间的方式排序，这两个列表是通过将剩余号码与该数字比较获得的，比这个数字小的号码集构成一个列表，比这个数字大的号码集构成另外一个列表。这种排序方法被称为快速排序，并且是已知的排序方法中最好的方法之一。

上面的 $quicksort$ 的实现是一个很好的体现 Haskell 功能强大、清晰又简明的例子。此外，函数 $qsort$ 也比预期更加通用，即不仅仅适合排序数字，同样适用于任何具备有序值的类型。更确切的说，类型

$qsort :: Ord\ a \Rightarrow [a] \rightarrow [a]$

表明对于任何具备有序值的类型， $qsort$ 是一个提供在这种值列表间映射的函数。Haskell 支持多种有序值类型，包括数字、单个字符比如'a' 以及字符串比如"abcde"。因此，$qsort$ 这个函数也可以用于对一个字符列表或一个字符串列表进行排序。

1.6 本章备注

Haskell 报告可以从 Haskel 主页 www.haskell.org 免费下载，同时这份报告也已经出版 (25)。另外 Hudak 的调查报告 (11) 更详尽的记录了关于函数式编程语言发展的历史。

1.7 习题

1. Give another possible calculation for the result of $double\ (double\ 2)$.
2. Show that $sum\ [x] = x$ for any number x .
3. Define a function $product$ that produces the product of a list of numbers, and show using your definition that $product\ [2, 3, 4] = 24$.
4. How should the definition of the function $qsort$ be modified so that it produces a reverse sorted version of a list?
5. What would be the effect of replacing ≤ by < in the definition of $qsort$?
   Hint: consider the example $qsort\ [2, 2, 3, 1, 1]$.

第 2 章　第一步

在本章中，我们将迈出使用 Haskell 的第一步。我们首先介绍 Hugs 系统和 Prelude 标准库，然后解释函数应用的记法，开发我们的第一个 Haskell 脚本，最后讨论一些关于脚本的语法惯例。

2.1　Hugs 系统

正如我们在前一章所看到的，我们可以手工执行一些小的 Haskell 程序，然而在实践中，我们通常需要一个可以自动执行程序的系统。在这本书中，我们使用一个被称为 *Hugs*的交互式系统，它也是使用最广泛的 Haskell 实现。

Hugs 的交互式的本质使得其非常适合教学和制作原型，并且它的性能可以满足绝大多数应用的要求。然而，如果需要更高的性能或独立的可执行程序，也有一些 Haskell 的编译器是可供选择的，这其中使用最广泛的是 Glasgow Haskell 的编译器，这个编译器也有一个和 Hugs 的运行方式类似的交互式版本，并且该版本可以很容易的在本书中使用。

2.2　标准 Prelude

当 Hugs 系统启动时，它首先加载一个名为 *Prelude.hs*的库文件，然后显示一个 > 提示符，表明系统正在等待用户输入待求值的表达式。例如，这个库文件定义了许多熟知的操作整数的函数，包括加，减，乘，除和求幂五个主要算术运算，如下所示：

```
> 2 + 3
5
> 2 - 3
-1
> 2 * 3
6
> 7 `div` 2
3
> 2 ^ 3
8
```

注意，整数除法操作符记作 `` `div` ``，如果结果是一个真分数，那么将向下圆整到最近的那个整数。

按常规数学惯例，求幂比乘法和除法具有更高的优先级，进而也具有比加法和减法更高的优先级。例如，`2 * 3 ^ 4` 表示`2 * (3 ^ 4)`，而`2 + 3 * 4` 表示`2 + (3 * 4)`。此外，求幂运算是右结合的，而其他四种算术操作符则是左结合的。例如，`2 ^ 3 ^ 4` 意味着`2 ^ (3 ^ 4)`，而`2 - 3 + 4` 则指的是`(2 - 3) + 4`。但实际上，在算术表达式里显式使用括号通常比依靠上述惯例表达得更为清楚。

除了操作整数的函数外，这个库文件还提供了一些有用的列表操作函数。在 Haskell 中，列表中的元素用方括号括上，并以逗号分隔。一些最常用的列表操作库函数说明如下：

- 从一个非空列表中选出第一个元素:

$$> head\ [1,\ 2,\ 3,\ 4,\ 5]$$
$$1$$

- 从一个非空列表中删除第一个元素:

$$> tail\ [1,\ 2,\ 3,\ 4,\ 5]$$
$$[2,\ 3,\ 4,\ 5]$$

- 选出列表中的第 n 个元素 (从 0 开始计数):

$$> [1,\ 2,\ 3,\ 4,\ 5]\ !!\ 2$$
$$3$$

- 选出列表中的前 n 个元素:

$$> take\ 3\ [1,\ 2,\ 3,\ 4,\ 5]$$
$$[1,\ 2,\ 3]$$

- 从列表中删除前 n 个元素:

$$> drop\ 3\ [1,\ 2,\ 3,\ 4,\ 5]$$
$$[4,\ 5]$$

- 计算列表的长度:

$$> length\ [1,\ 2,\ 3,\ 4,\ 5]$$
$$5$$

- 计算数字列表中元素之和:

$$> sum\ [1,\ 2,\ 3,\ 4,\ 5]$$
$$15$$

- 计算数字列表中元素之积:

$$> product\ [1,\ 2,\ 3,\ 4,\ 5]$$
$$120$$

- 连接两个列表:

$$> [1,\ 2,\ 3]\ +\!+ [4,\ 5]$$
$$[1,\ 2,\ 3,\ 4,\ 5]$$

- 反转列表:

$$> reverse\ [1,\ 2,\ 3,\ 4,\ 5]$$
$$[5,\ 4,\ 3,\ 2,\ 1]$$

针对某些参数值，标准 *Prelude*中的一些函数可能产生错误。比如试图除零或从一个空列表中选择第一个元素都会产生一个错误：

$$> 1\ `div`\ 0$$
$$Error$$
$$> head\ [\,]$$
$$Error$$

实际上，当出现错误时，*Hugs*系统也会产生一个消息，提供一些有关错误原因的信息。

作为参考，附录 A 介绍了标准 Prelude 最常用的一些定义，附录 B 显示了一些特殊的 Haskell 符号，如^ 和 $+$，以及如何使用键盘输入这些符号。

2.3 函数应用

在数学中，将函数应用于参数通常表示为用括号将参数括起来，且将两值相乘往往采用习惯性的表示，将两个值一个接着一个的写即可。例如，在数学中，表达式

$$f\ (a,\ b)\ +\ c\,d$$

意为将函数 f 应用于两个参数 a 和 b 上，并将结果与 c 和 d 的乘积相加。为了反映函数在语言中的主流位置，在 Haskell 中函数程序习惯性地使用空格表示，而两值相乘则显式地使用 $*$ 操作符表示。例如，上面的表达式使用 Haskell 编写如下：

$$f\ a\ b + c\ *\ d$$

此外，函数程序拥有比其它操作符更高的优先级。比如， $f\ a + b$ 意为 $(f\ a) + b$。下表给出了一些例子，来进一步说明函数记法在数学与 Haskell 之间的差异。

Mathematics	Haskell
$f(x)$	f x
$f(x, y)$	f x y
$f(g(x))$	f (g x)
$f(x, g(y))$	f x (g y)
$f(x)g(y)$	f x * g y

注意上面的表达式 $f\ (g\ x)$ 在 Haskell 中依然需要括号，因为 $f\ g\ x$ 本身会被解释为将函数 f 应用于两个参数 g 和 x。然而其本意却是将 f 应用于一个参数上，该参数即是将函数 g 应用于参数 x 上的结果。同样这个注意项也适用于表达式 $f\ x\ (g\ y)$.

2.4 Haskell 脚本

除了标准 Prelude 所提供的函数外，你也可以定义新的函数。你无法在 Hugs 的 > 提示符下定义新函数，只能在*脚本*中定义。脚本是一个由一系列定义组成的文本文件。按照惯例，Haskell 脚本通常用*.hs*作为文件后缀名以区别于其他种类的文件。

2.4.1 我的第一个脚本

当开发一个 Haskell 脚本时，保持两个窗口一直打开着是很有用的：一个窗口运行脚本的编辑器，另外一个运行 Hugs。举个例子，假设我们启动文本编辑器输入两个函数的定义，并保存脚本到一个名为 *test.hs*的文件中：

```
double x = x + x
quadruple x = double (double x)
```

相应的，假设我们保持编辑器窗口处于打开状态，而在另外一个窗口中启动 Hugs 并输入指令使其加载这个新脚本：

```
>: loadtest.hs
```

现在 *Prelude.hs*和 *test.hs*都已被加载，两个脚本中的函数都可以自由使用了。比如：

```
> quadruple 10
40
> take (double 2) [1, 2, 3, 4, 5, 6]
[1, 2, 3, 4]
```

现在保持 Hugs 处于启动状态，我们返回到编辑器窗口。将下面两个函数的定义添加到脚本中，并重新保存文件。

```
factorial n = product [ 1..n ]
average ns = sum ns `div` length ns
```

我们同样可以这样定义： $average\ ns = div\ (sum\ ns)\ (length\ ns)$, 但是将 *div*放在两个参数中间更加自然。一般情况下，任何接受两个参数的函数都可以写成将函数名用反单引号 (`) 括上后放在其参数之间的形式。

当脚本被修改后，Hugs 不会自动加载它们，所以在使用新定义之前必须执行一个 reload 命令：

```
>: reload
> factorial 10
3628800
> average [1, 2, 3, 4, 5]
3
```

作为参考，下表总结了一些 Hugs 中最常用命令的含义。请注意，每条命令都可以通过它的第一个字符进行缩写。例如，$:load$ 可以缩写为 $:l$。命令 $:type$ 将在后面的篇章中详细解释。

命令	含义
:load name	加载脚本 *name*
:reload	重新加载当前脚本
:edit name	编辑脚本 *name*
:type expr	显示 *expr*的类型信息
:?	显示所有命令
:quit	退出 Hugs

2.4.2 命名需求

定义一个新函数时，函数以及其参数的名字必须以小写字母开头，但后面可以跟随零个或多个字母（包括小写和大写），数字，下划线和正向单引号。例如，以下名字都是合法的：

$$myFun \quad fun1 \quad arg\,2 \quad x'$$

下面列表中的*关键字*在语言中都有着特殊的含义，并且不能作为函数或其参数的名字使用：

case class data default deriving do else
if import in infix infixl infixr instance
let module newtype of then type where

按照惯例，在 Haskell 中列表参数的名字中通常有一个后缀 *s*，表明它们可能含有多个值。例如，一个数字列表可能被命名为 *ns*，一个包含任意值的列表可能会被命名为 *xs*，一个字符列表可能被命名的 *css*。

2.4.3 布局规则

在一个脚本中，每个定义必须精确的从相同的列开始。这种*布局规则*使我们能够根据代码缩进确定定义分组。例如脚本：

$$\begin{aligned} &a = b + c \\ &\quad where \\ &\qquad b = 1 \\ &\qquad c = 2 \\ &d = a * 2 \end{aligned}$$

通过缩进可以很清楚的看出 b 和 c 是在 a 定义体中使用的局部定义。如果需要，这个分组可以显式的通过花括号将一系列定义括起来，并且定义之间可以用分号隔开。例如，上面的脚本也可以写成：

$$\begin{aligned} &a = b + c \\ &\quad where \\ &\qquad \{b = 1; \\ &\qquad c = 2\} \\ &d = a * 2 \end{aligned}$$

但一般来说，依赖布局规则来确定定义分组比使用显式语法更加清晰。

2.4.4 注释

除了新的定义，脚本也会包含注释，但注释将被 Hugs 忽略。Haskell 提供了两种类型的注释，分别称为*普通注释*和*嵌套注释*。普通注释以符号 -开始，作用延伸到当前行的结尾，如下面的例子所示：

```
—Factorial of a positive integer:
factorial n = product [1..n]
—Average of a list of integers:
average ns = sum ns `div` length ns
```

嵌套注释的开始和结束符号为 {-和 -}，嵌套注释可以跨多行，还可能包含其他注释。嵌套注释在临时删除脚本中的某段定义时特别有用，如下面的例子：

```
{-
double x
quadruple x
-}
```

2.5 本章备注

Hugs 系统可从 Haskell 主页 www.haskell.org 上自由下载，另外 Haskell 主页上还提供了其他有用的资源。

2.6 习题

1. 用括号显式标出下面算术表达式的结合情况：

```
2 ^ 3 * 4
2 * 3 + 4 * 5
2 + 3 * 4 ^ 5
```

2. 使用 Hugs 执行一遍本章所提供的例子

3. 下面的脚本中包含三处语法错误，纠正这些错误并使用 Hugs 确认你的脚本可以正常工作。

```
N = a `div` length xs
    where
      a = 10
    xs = [1, 2, 3, 4, 5]
```

4. Show how the library function last that selects the last element of a non-empty list could be defined in terms of the library functions introduced in this chapter. Can you think of another possible definition?

5. Show how the library function *init* that removes the last element from a non-empty list could similarly be defined in two different ways.

第 3 章　类型和类

在这一章中，我们将介绍 Haskell 中两个最基本的概念：类型和类。我们首先解释什么是类型以及在 Haskell 中如何使用它，然后介绍一些基本类型以及使用这些基本类型构造更大类型的方法，详细讨论函数类型，最后介绍一下多态类型和类型类的概念。

3.1　基本概念

*类型*是一组相关值的集合。例如，类型 $Bool$ 包含两个逻辑值 $False$ 和 $True$，而类型 $Bool \to Bool$ 则包含了所有将 $Bool$ 类型参数映射为 $Bool$ 类型结果的函数，如逻辑非函数 not。我们使用记法 $v :: T$ 表示 v是类型 T的一个值，并且可以说 v"具有类型"T。例如：

$$False :: Bool$$
$$True :: Bool$$
$$not :: Bool \to Bool$$

一般地说，符号:: 也可以用于尚未被求值的表达式，这种情况下，$e :: T$ 意思是对表达式 e 求值将产生一个类型 T 的值。例如：

$$not\ False :: Bool$$
$$not\ True :: Bool$$
$$not\ (not\ False) :: Bool$$

在 Haskell 中，每个表达式必须有一个类型，该类型通过一个先于表达式求值的过程计算得到，这个过程被称为*类型推断*。这个过程的关键在于一个函数应用的类型规则，其中规定如果 f 是一个将 A 类型参数映射为 B 类型结果的函数，且 e 是一个类型 A 的表达式，那么 $f\ \ e$ 具有类型 B：

$$\frac{f :: A \to B \quad e :: A}{f\ e :: B}$$

例如，$not\ False :: Bool$ 可通过这样的规则推断，该规则使用了这样的事实：$not :: Bool \to Bool$ 和 $False :: Bool$。另一方面，表达式 $not\ 3$ 通过上述规则无法推断类型，因为这需要 $3 :: Bool$，但 3 不是一个逻辑值，这是无效的。像 $not\ 3$ 这样的表达式无法确定类型，也可以说成是包含了一个类型错误，被视为无效表达式。

由于类型推断在求值过程之前，所以 Haskell 程序是*类型安全*的，也就是说在求值过程中不会发生类型错误。实际上，类型推断能检查出程序中所占错误比例较高的一类错误，它也是 Haskell 最有用的特点之一。但是注意使用类型推断并不能消除发生在求值阶段的其他类

错误的可能性，比如，表达式 $1\ `div`\ 0$ 可以通过类型推断检查，但在求值阶段报错，因为被 0 除的行为是未定义的。

类型安全的不足之处在于一些求值阶段成功的表达式却因类型原因而被拒绝。例如，条件表达式 **if** $True$ **then** 1 **else** $False$ 求值结果为 1, 但是因包含一个类型错误而被视为无效表达式。特别是，条件表达式的类型推断规则要求所有可能的结果都具有相同的类型，而在这里例子中，第一种结果为 1，是一个数字类型，而第二个结果是 $False$，是一个逻辑值类型。在实践中，程序员很快就学会了如何在类型系统的限制下工作以及如何避免这些问题。

在 Hugs 系统中，任意表达式的类型都可以通过 $:type$ 显示出来，例如：

$>: type\ not$
$not :: Bool \to Bool$

$>: type\ not\ False$
$not\ False :: Bool$

$>: type\ not\ 3$
$Error$

3.2 基本类型

Haskell 提供了很多内置到语言中的基本类型，其中最常用的类型描述如下：

$Bool$ - **逻辑值**
这个类型包含了两个逻辑值 $False$ 和 $True$。

$Char$ - **字符**
这个类型包含了普通键盘上提供的所有单个字符，如'a'，'A'，'3' 和'_'，以及拥有特殊效果的控制字符，如'\n'（移动到新的一行）和'\t'（移动到下一个制表位）。正如其他大多数编程语言的标准一样，单个字符必须用单引号'' 括起。

$String$ - **字符串**
这个类型包含了所有字符序列，诸如"abc"，"$1 + 2 = 3$" 以及空字符串""。同样正如其他大多数编程语言的标准，字符串必须用双引号" " 括起。

Int - **固定精度整数**
这个类型包含诸如 -100， 0 以及 999 这样的整数，计算机以固定大小的内存存储这些值。例如，Hugs 系统中 Int 类型的取值范围在 -2^{31} 和 $2^{31}-1$ 之间。超出这个范围将得到非预期的结果。例如，在 Hugs 系统中对2^31::Int（使用:: 将强制结果为 Int 类型而不是其他的数值类型）进行求值将得到一个负值，这是不正确的。

$Integer$ - **任意精度整数**

该类型包含所有的整数，我们使用足够多的内存储存这个类型的值，从而避免了对该类型值的范围强加上限和下限。例如，使用任何 Haskell 系统对`2^31::Integer` 求值都可以得到正确的结果。

除了对内存和精度的需求不同外，在 Int 和 $Integer$ 之间数字类型的选择还是性能考量之一。特别是，大多数电脑都内置了用来处理固定精度整数的硬件，而任意精度的整数通常必须被看成数字序列，通过速度较慢的软件来处理。

$Float$ - **单精度浮点数**
这个类型包含带小数点的数字，诸如 -12.34， 1.0 以及 3.14159，计算机以固定大小内存存储这些值。浮点一词源于这样一个事实：即小数点后允许的数字位数取决于数的大小。例如使用 Hugs 对 $sqrt\ 2 :: Float$ 求值结果为 1.41421（库函数 $sqrt$ 用于计算一个数的平方根），其中小数点后有五位数字; 而对 $sqrt\ 99999 :: Float$ 求值结果为 316.226，小数点后则只有三位数字。采用浮点数编程是一个专家话题，需要认真对待舍入误差。在本书入门性的文字中，我们将很少说到这种类型。

最后，我们注意到一个单个数字可能拥有不止一种数值类型。例如， $3 :: Int$， $3 :: Integer$ 和 $3 :: Float$ 对于数字 3 来说都是有效的类型。这就提出了一个有趣的问题：这些数字在类型推断过程中究竟应该被分配什么类型？这个问题将在本章后面考量类型类时回答。

3.3 列表类型

*列表*是一个由相同类型元素组成的序列，其元素放在方括号中，并使用逗号分隔。我们将元素类型为 T 的所有*列表*类型记作 $[T]$。比如：

$[False,\ True,\ False] :: [Bool]$
$['a',\ 'b',\ 'c',\ 'd'] :: [Char]$
$["One",\ "Two",\ "Three"] :: [String]$

一个列表中元素的个数称为列表的长度。长度为 0 的列表 [ ] 称为空列表，而长度为 1 的列表，如 $[False]$ 和 $['a']$，称为单件列表。注意 [[ ]] 和 [ ] 是两个不同的列表，前者是一个单件列表，组成该列表的唯一的元素是一个空列表，而后者则仅仅是一个空列表。

关于列表类型这里有三点需进一步注意。首先，一个列表的类型中没有传达其长度信息。例如， $[False,\ True]$ 和 $[False,\ True,\ False]$ 两者都是 $[Bool]$ 类型，即使他们的长度不同。其次，列表的元素没有类型限制。目前我们局限在我们所能给出的例子范围内，因为到目前为止我们介绍的唯一的非基本类型就是列表类型，但是我们可以定义由列表类型元素组成的列表，例如：

$[['a',\ 'b'],\ ['c',\ 'd',\ 'e']] :: [[Char]]$

最后，对一个列表的长度没有任何限制。特别是，正如我们将在第 12 章看到的那样，由于在 Haskell 中使用了惰性求值，具有无限长度的列表是自然且实用的。

3.4 元组类型

*元组*是一个由类型可能不同的元素组成的有限长度序列，其元素放在圆括号中，并使用逗号分隔。我们用 $(T_1, T_2, ..., T_n)$ 表示所有元组的类型。对于从 i 到 n 范围内的任意值 i，第 i 个元素具有类型 T_i。例如：

$(False, True) :: (Bool, Bool)$

$(False, 'a', True) :: (Bool, Char, Bool)$

$("Yes", True, 'a') :: (String, Bool, Char)$

一个元组中元素的个数称为*元数*（arity）。元数为 0 的元组 ( ) 称为空元组，元数为 2 的元组称为二元组，元数为 3 的元组称为三元组，等等。元数为 1 的元组，例如 $(False)$，是不允许使用的，因为它们将与显式设置求值顺序的括号的使用相冲突，如表达式 $(1 + 2) * 3$。

与列表类型一样，关于元组类型也有三点需进一步注意。首先，元组的类型传达了元数信息。例如，类型 $(Bool, Char)$ 包含了所有第一个元素为 $Bool$ 类型且第二个元素为 $Char$ 类型的二元组。其次，元组中的元素没有类型限制。例如，我们可以定义由元组类型元素组成的元组，由列表类型元素组成的元组以及由元组类型元素组成的列表：

$('a', (False, 'b')) :: (Char, (Bool, Char))$

$(['a', 'b'], [False, True]) :: ([Char], [Bool])$

$[('a', False), ('b', True)] :: [(Char, Bool)]$

最后，注意元组必须具有有限元数，以保证元组类型总是能在求值过程之前被计算出来。

3.5 函数类型

*函数*是一个从一种类型参数到另外一种类型结果的映射。我们用 $T_1 \to T_2$ 表示所有将 T_1 类型参数映射为 T_2 类型结果的函数。例如：

$not :: Bool \to Bool$

$isDigit :: Char \to Bool$

（库函数 $isDigit$ 用于判断一个字符是否是一个数字位）由于对函数的参数类型和结果类型没有任何限制，使用单一参数和结果的函数的简单概念就足以应付多个参数和结果的情况，只需将多个值使用列表或元组打包即可。例如，我们下面定义函数 add，用于计算一个整数二元组的元素之和；定义函数 $zeroto$，用于返回一个从 0 到给定上限值的整数列表：

$add :: (Int, Int) \to Int$

$add\,(x\,,\,y) = x + y$

$zeroto :: Int \to [Int]$

$zeroto\,n = [0\,..\,n]$

在这些例子中我们遵循了 Haskell 将函数类型放在函数定义之前作为参考文档的惯例。系统将检查由用户手工提供的类型与类型推断自动计算出的类型两者之间的一致性。

注意没有限制要求函数对它们的参数类型必须要有预期结果。这样一来，对于函数的某些参数来说，其结果可能是未定义的。比如当列表为空时，库函数 $head$ 从列表中选出第一个元素的行为就是未定义的。

3.6 Curried 函数

函数可以自由地将函数作为结果返回，利用这个事实，接受多个参数的函数也可以使用另外一种也许并不显而易见的方式处理。例如，考虑下面的定义：

$$add' \;::\; Int \to (Int \to Int)$$
$$add'\; x\; y \;=\; x \;+\; y$$

函数类型表明 add' 是一个函数，其接受的参数类型为 Int，返回结果为一个类型为 $Int \to Int$ 的函数。定义本身表明 add' 接受一个整数 x 为参数，后面跟着一个整数 y，返回结果为 $x + y$。更确切地说，add' 接受整数 x 为参数，返回一个接受整数 y 为参数且返回 $x + y$ 的函数。

注意函数 add' 产生的最终结果与上一节中的函数 add 相同。然而函数 add 将其两个参数打包为一个二元组后一同处理，而函数 add' 则一次仅接受处理一个参数，正如这两个函数的类型所反映的那样：

$$add \;::\; (Int,\, Int) \to Int$$
$$add' \;::\; Int \to (Int \to Int)$$

对于有两个以上参数的函数，也可以使用同样的技术处理，通过返回以函数为返回值的函数，等等。例如，函数 $mult$ 接受三个参数，每次接受一个，并返回它们的乘积，其定义如下：

$$mult \;::\; Int \to (Int \to (Int \to Int))$$
$$mult\; x\; y\; z \;=\; x \;*\; y \;*\; z$$

这个定义表明 $mult$ 接受一个整数 x 为参数并返回一个函数，后者依次接受一个整数 y 为参数并返回另外一个函数，最后这个函数接受整数 z 为参数，并最终返回结果 $x * y * z$。

诸如 add' 和 $mult$ 这样的每次接受一个参数的函数被称为 $curried$。除了本身有吸引力之外，curried 函数也比接受元组作为参数的函数更加灵活，因为一些有用的函数通常可以通过*部分应用*参数不完整的 curried 函数来实现。例如，一个完成递增功能的函数可以通过 curried 函数 add' 的部分应用 $add'\; 1 :: Int \to Int$ 实现，后者仅需要两个参数中的一个。

为避免在使用 curried 函数工作时过度使用括号，我们采纳了两个简单的惯例。首先，类型中使用的箭头 $\to$ 是右结合的，例如：

$$Int \to Int \to Int \to Int$$

意为

$$Int \to (Int \to (Int \to Int))$$

然而使用空格表示的函数应用则是左结合的，例如：

$$mult\; x\; y\; z$$

意为

$$((mult\; x)\; y)\; z$$

除非显式需要元组，Haskell 中的所有接受多个参数的函数一般都会被定义成 curried 函数，并且使用上面的两个惯例来减少需要使用的括号的数量。

3.7 多态类型

库函数 *length*用于计算任意列表的长度，无论列表中的元素是什么类型的。比如，它可以用于计算整型列表、字符串型列表甚至是函数类型列表的长度：

$> length\ [1,\ 3,\ 5,\ 7]$
4
$> length\ ["Yes",\ "No"]$
2
$> length\ [isDigit\ ,\ isLower\ ,\ isUpper]$
3

通过在类型中包含*类型变量*，使得将 $length$ 应用于由任意类型元素组成的列表的想法得以精确实现。类型变量必须以小写字母开头，通常命名为 a，b，c等。例如，*length*的类型如下：

$$length :: [a] \to Int$$

即对任意类型 a，函数 *length*具有类型 $[a] \to Int$。包含一个或多个类型变量的类型被称作*多态的*（“多种形式"），使用这种类型的表达式也是多态的。因此 $[a] \to Int$ 是一个多态类型，*length*是一个多态函数。更普遍的是，标准 Prelude 中提供的很多函数都是多态的，例如：

$fst\ ::\ (a,\ b) \to a$
$head\ ::\ [a] \to a$
$take\ ::\ Int \to [a] \to [a]$
$zip\ ::\ [a] \to [b] \to [(a,\ b)]$
$id\ ::\ a \to a$

3.8 重载类型

算术运算符 $+$ 用于计算任意两个相同数值类型数的和。例如，它可以用来计算两个整数的和或两个浮点数的和：

$> 1\ +\ 2$
> 3

$> 1.1\ +\ 2.2$
> 3.3

通过在类型中包含*类约束*，使得将 $+$ 应用于任意数值类型数字的想法得以精确实现。类约束写成 $C\ a$，其中 C 是类的名字，a 是一个类型变量。例如，$+$ 的类型如下：

$$(+)\ ::\ Num\ a \Rightarrow a \to a \to a$$

即对于任意一个数值类型类 Num 的实例类型 a，函数 $(+)$ 具有类型 $a \to a \to a$。（将操作符括起来将之转换为一个 curried 函数，下一章中将详细解释其中缘由。）一个包含一个或多个类约束的类型称为*重载的*，使用这种类型的表达式也是重载的。因此，$Num\ a \Rightarrow a \to a \to a$ 是一个重载类型，$(+)$ 是一个重载函数。更普遍的是，标准 Prelude

库中提供的大多数数值函数都是重载的，例如：

$(-) :: Num\ a \Rightarrow a \to a \to a$
$(*) :: Num\ a \Rightarrow a \to a \to a$
$(negate) :: Num\ a \Rightarrow a \to a$
$(abs) :: Num\ a \Rightarrow a \to a$
$(signum) :: Num\ a \Rightarrow a \to a$

此外，数值本身也是重载的。例如，$3 :: Num\ a \Rightarrow a$ 意为对于任意数值类型 a，数字 3 具有类型 a。

3.9 基本类

回顾一下，一个类型是一组相关值的集合。基于这个概念，一个类是一组支持重载操作的类型的集合，这些重载操作被称为方法。Haskell 提供一定数量的内置的基本类，下面描述了其中最常用的类：

Eq - **相等类型**
包含在这个类中的类型的值可以使用下面两个方法进行相等和不等比较：

$(==) :: a \to a \to Bool$
$(\neq) :: a \to a \to Bool$

所有的基本类型 $Bool, Char, String, Int, Integer$ 和 $Float$ 都是 Eq 类的实例。列表和元组类型也是一样，如果它们的元素类型是 Eq 类的实例，例如：

$> False == False$
$True$

$>' a' == 'b'$
$False$

> "abc" == "abc"
$True$

$> [1, 2] == [1, 2, 3]$
$False$

$> ('a', False) == ('a', False)$
$True$

Ord - **有序类型**
包含在这个类中的类型都是 Eq 类的实例，除此之外这些类型的值都是（线性）有序的，可以通过以下六个方法进行比较和处理：

$(<) :: a \to a \to Bool$
$(\leq) :: a \to a \to Bool$
$(>) :: a \to a \to Bool$
$(\geq) :: a \to a \to Bool$
$(min) :: a \to a \to a$
$(max) :: a \to a \to a$

所有的基本类型 $Bool, Char, String, Int, Integer$ 和 $Float$ 都是 *Ord*类的实例。列表和元组类型也是一样，如果它们的元素类型是 Ord 类的实例，例如：

$> False < True$
$True$

$> min\ 'a'\ 'b'$
$'a'$

$>$ "$elegant$" $<$ "$elephant$"
$True$

$> [1, 2, 3] < [1, 2]$
$False$

$> ('a', 2) < ('b', 1)$
$True$

$> ('a', 2) < ('a', 1)$
$False$

请注意，字符串，列表和元组都是按字典序排序的；也就是说与单词在字典中的顺序一样。例如，两个相同类型的二元组是有序的。如果它们的第一个元素是有序的，则无须考虑它们的第二个元素。或者如果它们的第一个元素相等，那么它们的第二个元素则必须是有序的。

Show - **可显示的类型**
包含在这个类中的类型的值都可以通过下面方法转换为字符串：

$show :: a \to String$

所有的基本类型 $Bool, Char, String, Int, Integer$ 和 $Float$ 都是 *Show*类的实例。列表和元组类型也是一样，如果它们的元素是 Show 类的实例，例如：

$> show\ False$
"False"

> $show\ 'a'$
"'a'"

> $show$ 123
"123"

> $show$ [1, 2, 3]
"[1, 2, 3]"

> show $('a',\ False)$
"$('a',\ False)$"

$Read$ - **可读的类型**
这个类与类 Show 是一对，包含在这个类中的类型的值可以通过下面方法从字符串转换得到：

$$read :: String \to a$$

所有的基本类型 $Bool, Char, String, Int, Integer$ 和 $Float$ 都是 *Read*类的实例。列表和元组类型也是一样，如果它们的元素是 Read 类的实例，例如：

> $read$ "$False$" $::\ Bool$
$False$

> $read\ 'a' ::\ Char$
'a'

> $read$ 123 $::\ Int$
123

> $read$ "[1, 2, 3]" :: [Int]
[1, 2, 3]

> $read$ "$('a',\ False)$"
$('a',\ False)$

例子中使用 :: 决定结果的类型。然而在实际中，通常可通过上下文自动推断出必要的类型信息。例如，表达式 $not\ (read$ "$False$") 不需要显式的类型信息，因为使用逻辑非的 not 的程序暗示了 $read$ "$False$" 必须具有 $Bool$ 类型。

注意，如果参数不符合语法要求，那么 $read$ 的结果将是未定义的。例如，表达式 $not\ (read$ "$hello$") 在求值时会产生一个错误，因为 "$hello$" 被 $read$ 的结果不是一个逻辑值。

Num - **数字类型**

包含在这个类中的类型都是 Eq 类和 $Show$ 类的实例，除此之外这些类型的值都是数字，可以通过以下六个方法进行处理：

$$(+) :: a \to a \to a$$
$$(-) :: a \to a \to a$$
$$(*) :: a \to a \to a$$
$$(negate) :: a \to a$$
$$(abs) :: a \to a$$
$$(signum) :: a \to a$$

（$negate$ 方法返回一个数的负数， abs 返回绝对值，而 $signum$ 则返回数的符号性。）基本类型 $Int, Integer$ 和 $Float$ 都是 Num 类的实例，例如：

$> 1 + 2$
3

$> 1.1 + 2.2$
3.3

$> negate\ 3.3$
-3.3

$> abs\ (-3)$
3

$> signum\ (-3)$
3

注意 Num 类没有提供除法，但是正如我们即将要看到的，Haskell 提供两个特殊的类单独处理除法，一个类用于整数数字，而另外一个用于分数。

$Integral$ - **整数类型**

包含在这个类中的类型都是 Num 类的实例，除此之外这些类型的值都是整数，支持整数除法和整数取余：

$$div :: a \to a \to a$$
$$mod :: a \to a \to a$$

实际中，这两个方法经常写在其两个参数之间，并用单引号括上。基本类型 Int 和 $Integer$ 是 Integral 类的实例。例如：

$> 7\ `div`\ 2$

3

$> 7 \text{ `}mod\text{` } 2$
1

考虑到效率，一些标准 Prelude 库中既涉及列表又涉及整型的函数（如 $length$， $take$ 和 $drop$）被严格应用在 Int 这样的有限精度整数上，而不是应用到任意 $Integral$ 类的实例上。如果需要，这些函数的通用版本在另外一个叫作 $List.hs$ 的库中提供。

$fractional$ - **分数类型**
包含在这个类中的类型都是 Num 类的实例，但除此之外这些类型的值都是非整数，支持小数除法和小数倒数：

$(/) :: a \to a \to a$
$(recip) :: a \to a$

基本类型 $Float$ 是分数类的一个实例，例如：

$> 7.0/2.0$
3.5

$> recip\ 2.0$
0.5

3.10 本章备注

用术语 $Bool$ 表示逻辑值类型是为了纪念 **George Boole**在符号逻辑领域做出的开创性贡献，而用术语 $curried$ 表示函数一次只接受一个参数是用于纪念 **Haskell Curry**(Haskell 语言本身也是以他命名的) 在这类函数领域所做的工作。Haskell 报告 (25) 中给出了关于类型系统的更详尽的描述，提供给专家的正式说明，可以在 (20; 6) 中找到。

3.11 习题

1. 下面的值是什么类型？

   $['a', 'b', 'c']$
   $('a', 'b', 'c')$
   $[(False, 'O'), (True, '1')]$
   $([False, True], ['0', '1'])$
   $[tail, init, reverse]$

2. 下面的函数是什么类型？

$$second\ xs\ =\ head\ (tail\ xs)$$
$$swap\ (x, y) = (y, x)$$
$$pair\ x\ y = (x, y)$$
$$double\ x = x * 2$$
$$palindrome\ xs = reverse\ xs\ ==\ xs$$
$$twice\ f\ x = f\ (f\ x)$$

提示：如果函数定义中使用了重载操作符，注意包含必要的类约束。

3. 使用 Hugs 检查一下你关于前两个问题的回答

4. 为什么在一般情况下将函数类型都作为 Eq 类的实例是不可行的？什么时候是可行的？
提示：两个类型相同的函数是相等的，如果两个类型相同的函数对于相等的参数始终返回相同的结果，那么这两个函数是相等的。

第 4 章　定义函数

在本章中我们将介绍一些在 Haskell 中定义函数的机制。我们首先介绍条件表达式和守卫等式，然后介绍一种简单却强大的模式匹配思想，最后介绍 lambda 表达式和段的概念。

4.1　以旧造新

也许定义新函数最直接的方法就是简单地将已有的一个或多个函数结合起来。例如，下面展示的一些库函数就是用这种方法定义的：

- 判断一个字符是否是数字

$$
\begin{aligned}
&isDigit \ :: \ Char \to Bool \\
&isDigit\, c \ = \ c \geq' 0' \ \&\& \ c \leq' 9'
\end{aligned}
$$

- 判断一个整数是否是偶数

$$
\begin{aligned}
&even \ :: \ Integral\, a \Rightarrow a \to Bool \\
&even\, c \ = \ n \ `mod` \ 2 == 0
\end{aligned}
$$

- 将一个列表在第 nth 个元素处拆分

$$
\begin{aligned}
&splitAt \ :: \ Int \to [a] \to ([a],\ [a]) \\
&splitAt\, n\, xs \ = \ (take\, n\, xs,\ drop\, n\, xs)
\end{aligned}
$$

- 倒数

$$
\begin{aligned}
&recip \ :: \ Fractional\, a \Rightarrow a \to a \\
&recip\, n \ = \ 1 \ / \ n
\end{aligned}
$$

注意上面 $even$ 和 $recip$ 类型中类约束的使用，精确的指明了这两个函数可以分别应用于任何整数类型和分数类型。

4.2　条件表达式

有一类函数，它们从许多种可能的结果中选择出一个最终结果，Haskell 提供了很多不同的方式来定义这类函数。最简单的方式就是使用*条件表达式*，条件表达式使用被称为*条件*的逻辑表达式在两个相同类型的结果中选出一个。如果条件为*真*，就选中第一个结果，否则选中第二个。例如：库函数 abs 的定义如下，该函数返回一个整数的绝对值：

$$
\begin{aligned}
&abs \;::\; Int \to Int\\
&abs\, n \;=\; \textbf{if}\; n \;\geq\; 0\; \textbf{then}\; n\; \textbf{else}\; -n
\end{aligned}
$$

条件表达式可以嵌套，它们可以包含其他条件表达式。例如，库函数 $signum$ 定义如下，它用来返回一个整型数的符号：

$$
\begin{aligned}
&signum \;::\; Int \to Int\\
&signum\, n \;=\; \textbf{if}\; n \;<\; 0\; \textbf{then}\; -1\; \textbf{else}\\
&\qquad\qquad\qquad \textbf{if}\; n \;==\; \textbf{then}\; 0\; \textbf{else}\; 1
\end{aligned}
$$

注意，与某些编程语言中的条件表达式不同，Haskell 中的条件表达式必须包含 **else**分支，这样就避免了众所周知的"else 悬挂"问题。例如，如果 **else**分支是可选的，那么表达式 **if** $True$ **then if** $False$ **then** 1 **else** 2 既可以返回结果 2，也可能产生一个错误，这取决于表达式中的 **else**分支是被看作是内部条件表达式的一部分还是被看作是外部条件表达式的一部分。

4.3 守卫等式

作为条件表达式的一种替代方案，函数还可以使用*守卫等式*来定义。在这类函数中，我们使用一些被称为*守卫*的逻辑表达式从一些类型相同的结果中选择函数的最终结果。如果第一个守卫等式为*真*，那么第一个结果被选中；否则如果第二个守卫等式为*真*，则第二个结果被选中，依此类推。例如，库函数 abs 也可以以下面的方式定义：

$$
\begin{aligned}
abs\, n \quad &| n \geq 0 \quad &= \; n\\
&| otherwise \quad &= \; -n
\end{aligned}
$$

符号 | 读作"满足于，使得"。守卫 $otherwise$ 在标准 Prelude 库文件中简单地定义为 $otherwise \;=\; True$。虽然以 $otherwise$ 作为一系列守卫的结尾不是必要的，但这样做为处理"所有其他情况"提供了一种便利的方式，同时也清楚地避免了因所有守卫都不为真而出错的情况。

较之条件表达式，使用守卫等式定义函数可读性更好。例如：使用如下守卫等式定义的库函数 $signum$ 更容易理解。

$$
\begin{aligned}
signum\, n \quad &| n \;<\; 0 \quad &= \; -1\\
&| n \;==\; 0 \quad &= \; 0\\
&| otherwise \quad &= \; 1
\end{aligned}
$$

4.4 模式匹配

使用*模式匹配*可以使许多函数拥有一个极为简单且直观的定义。在这类函数定义中，我们使用一些被称为*模式*的语法表达式从一些类型相同的结果中选择出函数的最终结果。如果第一个模式匹配成功，那么第一个结果被选中；否则，如果第二个模式匹配成功，那么第二个结果被选中，依此类推。例如，库函数 not 的定义如下，它用来返回一个对逻辑值取非的结果：

$$
\begin{aligned}
not \quad &:: \quad Bool \to Bool\\
not\, False \quad &= \quad True\\
not\, True \quad &= \quad False
\end{aligned}
$$

接受多个参数的函数也可以使用模式匹配定义，这种情况下，每个等式中各个参数的模式按顺序进行匹配。例如，库操作符 && 定义如下，该操作符返回两个逻辑值与后的结果：

$$
\begin{array}{lcl}
(\&\&) & :: & Bool \to Bool \to Bool \\
True \;\&\&\; True & = & True \\
True \;\&\&\; False & = & False \\
False \;\&\&\; True & = & False \\
False \;\&\&\; False & = & False
\end{array}
$$

然而，我们可以通过将最后三个等式合并为一个等式来简化这个函数的定义，合并后的等式使用可匹配任何值的*通配符*模式 _，并返回与两个参数值无关的结果 $False$。

$$
\begin{array}{lcl}
True \;\&\&\; True & = & True \\
\_ \;\&\&\; \_ & = & False
\end{array}
$$

根据第 12 章讨论的惰性求值，这个版本的函数定义还有这样的好处：如果第一个参数为 $False$，那么我们可直接返回结果 $False$, 而无须对第二个参数进行求值。在实际中，标准库 prelude 使用了同样具有这个属性的等式定义 &&。但只使用了第一个参数的值来选择哪个等式作为最终结果：

$$
\begin{array}{lcl}
True \;\&\&\; b & = & b \\
False \;\&\&\; \_ & = & False
\end{array}
$$

即如果第一个参数为 $True$，那么结果为第二个参数的值；如果第一个参数为 $False$，那么结果就是 $False$。

注意，因技术原因在一个等式中同样的名字不能被用于多个参数。例如，下面的操作符 && 的定义就是基于这个观察：如果两个参数相等，那么结果也是同样的值，否则结果为 $False$。但是由于上面的命名要求，这个定义是无效的：

$$
\begin{array}{lcl}
b \;\&\&\; b & = & b \\
\_ \;\&\&\; \_ & = & False
\end{array}
$$

然而如果需要，我们可以使用守卫等式来定义一个有效的版本，守卫等式用来判断两个参数是否相等：

$$
\begin{array}{lll}
b \;\&\&\; c & | \; b == c & = b \\
 & | otherwise & = False
\end{array}
$$

到目前为止，我们只考虑了基本模式，要么是值、要么是变量或是通配符模式。在本节其余部分，我们将介绍三种有用的将较小模式结合成较大模式的方法。

元组模式

一个模式元组的本身就是一个模式，它可以匹配任何元数相同且所有元素都可按顺序匹配对应模式的元组。例如，库函数 fst 和 snd 定义如下，它们分别返回二元组的第一个和第二个元素：

$$
\begin{array}{lcl}
fst & :: & (a,\, b) \to a \\
fst\,(x,\, \_) & = & x \\
snd & :: & (a,\, b) \to b \\
snd\,(\_,\, y) & = & y
\end{array}
$$

列表模式

同样，一个模式列表本身是一个模式，它可以匹配任何长度相同且所有元素都可按顺序匹配对应模式的列表。例如，用来判断一个列表是否精确的包含三个元素且第一个元素为'a'的函数 $test$ 定义如下：

$$
\begin{array}{lcl}
test & :: & [Char] \to Bool \\
test\ ['a',\ \_,\ \_] & = & True \\
test\ \_ & = & False
\end{array}
$$

到目前为止，我们一直将列表视为 Haskell 内置的原子类型。但事实上并非如此，它们实际上是使用操作符 : 从空列表 [ ] 开始一次一个元素的构造起来的。操作符 : 被称为 $cons$，它通过将一个新元素加到一个已存在列表的开始处来构造一个新的列表。例如，列表 $[1,\ 2,\ 3]$ 可以按如下分解：

$$
\begin{array}{l}
[1,\ 2,\ 3] \\
= \{\text{列表记法}\} \\
1\ :\ [2,\ 3] \\
= \{\text{列表记法}\} \\
1\ :\ (2:\ [3]) \\
= \{\text{列表记法}\} \\
1\ :\ (2:\ (3:\ []))
\end{array}
$$

即 $[1,\ 2,\ 3]$ 只是 $1\ :\ (2\ :\ (3\ :\ [\,]))$ 的一个缩写。为了避免在使用这样的列表时过度使用括号，cons 操作符被假定为是右结合的。例如，$1:\ 2:\ 3:\ [\,]$ 意为 $1:\ (2:\ (3:\ [\,]))$。

cons 操作符不仅可以用来构造列表，也可以用来构造模式，用于匹配任何非空且第一个元素及其余元素都可按顺序匹配对应模式的列表。例如，我们现在可以定义一个更通用的函数 $test$ 的版本，用于判断包含任意数量字符的列表是否以'a' 开头：

$$
\begin{array}{lcl}
test & :: & [Char] \to Bool \\
test\ ('a'\ :\ \_) & = & True \\
test\ \_ & = & False
\end{array}
$$

同样，库函数 $null$，$head$ 和 $tail$ 的定义如下，它们分别用于判断一个列表是否为空，从一个非空列表中选出第一个元素以及从一个非空列表中删除第一个元素：

$$
\begin{array}{lcl}
null & :: & [a] \to Bool \\
null\ [\,] & = & True \\
null\ (\_:\_) & = & False \\
head & :: & [a] \to a \\
head\ (x:\_) & = & x \\
tail & :: & [a] \to a \\
tail\ (\_:\ xs) & = & xs
\end{array}
$$

注意 cons 操作符必须用括号括上，因为函数优先级高于所有其他操作符。例如，不带括号的定义 $tail\ \_:\ xs\ =\ xs$ 意为 $(tail\ \_)\ :\ xs\ =\ xs$，它不仅含义不正确，而且还是一个无效的定义。

整数模式

作为一种有时很有用的特殊情况，Haskell 也允许 $n+k$ 形式的整数模式[1]，其中 n 是一个整数变量，$k>0$ 是一个整数常量。例如，函数 $pred$ 定义如下，它将 0 映射到本身且将任何正整数映射为该整数的前驱：

$$\begin{array}{lcl} pred & :: & Int \to Int \\ pred\,0 & = & 0 \\ pred\,(n\ +\ 1) & = & n \end{array}$$

关于 $n+k$ 模式有两点需要注意。首先，它们只是匹配 $\geq k$ 的整数。例如，对 $pred\,(-1)$ 求值将产生一个错误，因为 $pred$ 定义中的两个模式都不匹配负数。其次，和 cons 模式一样，整数模式必须用括号括起来。例如，不带括号的定义 $pred\,n+1\ =\ n$ 意为 $(pred\,n)+1\ =\ n$，这是一个无效的定义。

4.5 Lambda 表达式

作为使用等式定义函数的一种替代方案，函数还可以使用 *lambda 表达式*来构造。lambda 表达式包含一个用于参数匹配的模式以及一个详细说明了如何根据参数计算出结果的主体。lambda 表达式定义的函数没有函数名，换句话说，lambda 表达式是匿名函数。

例如，一个只接受一个数字为参数且返回结果 $x\ +\ x$ 的匿名函数可以按下面方式构造：

$$\lambda x \to x\ +\ x$$

符号 λ 是小写希腊字母"lambda"。尽管 lambda 表达式构造的函数没有名字，但它们仍然可以像其他函数那样使用。例如：

$$\begin{array}{l} >\ (\lambda x \to x\ +\ x)\ 2 \\ 4 \end{array}$$

除了自身有趣之外，lambda 表达式还有很多实际应用。首先，它们可以用来形式化 curried 函数定义的含义。例如，定义

$$add\,x\,y\ =\ x\ +\ y$$

可以被理解为

$$add\ =\ \lambda x \to (\lambda y \to x\ +\ y)$$

这里精确地表明 add 是一个函数，它接受一个数字 x 作为参数且返回一个函数，而这个返回的函数再接受数字 y 为参数且返回结果 $x\ +\ y$。

其次，当定义返回值的本质为函数而不是 curring 结果的函数时，lambda 表达式也非常有用。例如，库函数 $const$ 定义如下，它返回一个总是产生给定值的常量函数：

$$\begin{array}{lcl} const & :: & a \to b \to a \\ const\,x\ \_ & = & x \end{array}$$

但是，更吸引人地定义 $const$ 的方式是通过在类型中使用括号，在定义中使用 lambda 表达式以显式地指出用函数作为返回结果：

[1]译注：$n+k$ 模式在 Haskell 2010 标准中默认被禁用。若使用 $n+k$ 模式，在 GHC 7 及后续版本中必须开启 NPlusKPatterns 扩展。

$$
\begin{array}{lcl}
const & :: & a \to (b \to a) \\
const\ x & = & \lambda\_ \to x
\end{array}
$$

最后，lambda 表达式可用于避免给仅被引用一次的函数命名。例如，函数 $odds$ 定义如下，它用于返回前 n 个奇数：

$$
\begin{array}{lcl}
odds & :: & Int \to [Int] \\
odds\ n & = & map\ f\ [0..n-1] \\
 & & \textbf{where}\ f\ x\ =\ x\ *\ 2\ +\ 1
\end{array}
$$

（库函数 map 用于将一个函数应用于一个列表中的所用元素。）但是，由于本地定义的函数 f 只被引用一次， $odds$ 的定义可以使用 lambda 表达式简化为：

$$odds\ n\ =\ map\ (\lambda x \to x\ *\ 2\ +\ 1)\ [0..n-1]$$

4.6 段

$(\oplus)$

第 5 章　List comprehensions

在本章节中，我们将介绍列表推导式 (List comprehensions)[1]，它允许我们以一种简明的方式定义许多基于列表（lists）的函数。我们首先介绍生成器（generators）和守卫（guards），接着介绍库函数 zip 和字符串推导的有关概念，本章结尾部分给出一个破解凯撒密码的程序。

5.1　生成器

在数学中，记号 *comprehension*可基于已有的集合构建新的集合。例如，从推导式 $\{x^2|x \in \{1..5\}\}$ 可得集合 $\{1,4,9,16,25\}$，它是集合 $\{1..5\}$ 中元素 x 的平方的集合。在 Haskell 中，也可以使用类似的推导式基于已有的列表创建新的列表。例如：

$$
\begin{aligned}
&> [x \uparrow 2 \mid x \leftarrow [1..5]] \\
&[1,4,9,16,25]
\end{aligned}
$$

符号 | 和 $\leftarrow$ 分别读作“满足”和“取自”，称表达式 $x \leftarrow [1..5]$ 为一个生成器。一个列表推导式可以有多个生成器，连续的生成器间用逗号（，）隔开。例如，从列表 [1,2,3] 和 [4,5] 构造所有元素对的列表可以如下表示：

$$
\begin{aligned}
&> [(x,y) \mid x \leftarrow [1,2,3],\ y \leftarrow [4,5]] \\
&[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
\end{aligned}
$$

改变推导式中生成器的顺序得到的是相同的元素对集合，但是元素对的顺序不同，例如：

$$
\begin{aligned}
&> [(x,y) \mid y \leftarrow [4,5],\ x \leftarrow [1,2,3]] \\
&[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
\end{aligned}
$$

需特别指出的是，在本例中，元素对中的 x 比 y 变化更频繁（1,2,3,1,2,3 vs 4,4,4,5,5,5），而在前一例中，y 比 x 变化更频繁（4,5,4,5,4,5 vs 1,1,2,2,3,3）。这种行为可以按如下方式理解：更靠后的生成器可以看作更深层的嵌套，因此比它前面的生成器产生的值变化更频繁。

后面的生成器还可以依赖它前面生成器产生的值。例如，根据列表 [1..3] 生成的所有有序对可以用下面推导式生成：

[1] 参考 python 中的翻译，暂定，待议

```
>[(x, y) | x ← [1..3], y ← [x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

与此类似另一个例子是，库函数 $concat$，它将列表中的列表连接在一起，它使用一个生成器依次选择列表中的每个列表 xs，并使用另一个生成器选择 xs 中的每一个元素。

```
concat      ::  [[x]] → [a]
concat xss  =   [x | xs ← xss, x ← xs]
```

生成器中的通配符 _ 在丢弃列表中某些特定元素是非常有用。例如，返回列表中元素对中第一个组成的函数可以定义如下：

```
firsts      ::  [(a, b)] → [a]
firsts ps   =   [x | (x, _) ← xs]
```

与此类似，计算列表长度的库函数 $length$ 可以如下定义，首先将列表中的每个元素替换为数字 1，接着计算结果列表的和：

```
length      ::  [a] → Int
length xs   =   sum[1 | _ ← xs]
```

该定义中，生成器 _ ← xs 仅仅作为计数器来计算对应数目的 1 的和。

5.2 守卫

列表推导式还可以使用被称为守*卫 (guards)*的逻辑表达式来过滤前面生成器生成的值。如果守卫值为*真*，则当前值被保留，否则被丢弃。举例来说，推导式 $[x \mid x \leftarrow [1..10], even\ x]$ 的计算结果为列表 [1..10] 中所有偶数组成的列表 $[2,4,6,8,10]$。与此类似，生成某个正整数的全部因数的列表的库函数 $factors$：

```
factors     ::  Int → [Int]
factors n   =   [x | x ← [1..n], n `mod` x == 0]
```

例如：

```
> factors 15
[1,3,5,15]
> factors 7
[1,7]
```

回想下素数的定义：大于 1 且其正因数仅为 1 和它本身的数。因此，使用 $factors$ 可很简单的定义一个函数来判断一个整数是否为素数：

```
prime       ::  Int → Bool
prime n     =   factors n == [1,n]
```

例如：

```
> prime 15
False
```

```
> prime 7
True
```

注意，使用函数 $prime$ 来判断一个诸如 15 这样的数是否为素数并不需要它计算出它所有的因数。由于惰性求值，当计算结果中出现不同于数字 1 和它本身之外的任意因数时，立即就能得到结果 $False$，在本例中为因数 3。

返回列表推导式，使用函数 $prime$，我们可以定义一个函数得到给定上限下的所有素数的列表：

```
primes      ::   Int → [Int]
primes n    =    [x | x ← [2..n], prime x]
```

例如：

```
> primes 40
[2,3,5,7,11,13,17,19,23,29,31,37]
```

在第 12 章中，我们会使用著名的“埃拉托色尼过滤（sieve of Eratosthenes）算法”来构造更有效的程序来生成素数，Haskell 提供了该算法清晰而简单的实现。

作为守卫的最后的一个例子，我们给出由键值对构成的列表的查找表。函数 $find$ 查找并返回给对应键的值的列表，函数定义如下：

```
find       ::   Eq a ⇒ a → [(a, b)] → [b]
find k t   =    [v | (k', v) ← t, k == k']
```

例如：

```
> find 'b' [('a',1),('b',2),('c',3),('d',4)]
[2,4]
```

5.3 zip 函数

库函数 zip 通过将两个已有的列表对应元素配对生成新的列表，直到其中一个列表或全部结束。例如：

```
> zip ['a','b','c'] [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

库函数 zip 有时候在列表推导式中很有用。比如，假如我们定义函数 $pairs$，它返回一个列表，该列表由已有列表的相邻元素组成的数对组成，定义如下：

```
pairs      ::   [a] → [(a, a)]
pairs xs   =    zip xs (tail xs)
```

例如：

```
> pairs [1,2,3,4]
```

$[(1,2),(2,3),(3,4)]$

现在我们可以使用函数 $pairs$ 来定义新的函数，判断一个由任意有序类型元素组成的列表是否有序，只需要简单相邻元素组成的元素对保持正确的大小顺序：

$$\begin{array}{lll} sorted & :: & Ord\ a \Rightarrow [a] \rightarrow Bool \\ sorted\ xs & = & and\ [x \leq y \mid (x,y) \leftarrow pairs\ xs] \end{array}$$

例如：

$> sorted\ [1,2,3,4]$
$True$

$> sorted\ [1,3,2,4]$
$False$

与函数 $prime$ 类似，判断列表 [1,3,2,4] 为非有序，并不需要生成全部的相邻元素对。因为只要生成任意一个非有序元素对时，函数即可返回为*假*，在本例中为元素对 (3,2).

使用函数 zip 还可以定义函数 $positions$，它实现下面的功能：返回列表中某个给定值的位置的列表，先将列表中每个元素与其位置配对，然后选出所需值的那些位置：

$$\begin{array}{lll} positions & :: & Eq\ a \Rightarrow a \rightarrow [a] \rightarrow [Int] \\ positions\ x\ xs & = & [i \mid (x',i) \leftarrow zip\ xs\ [0..n], x\ ==\ x'] \\ & & \textbf{where}\ n\ =\ length\ xs\ -1 \end{array}$$

例如：

$> positions\ False\ [True, False, True, False]$
$[1,3]$

5.4 字符串推导式 (String comprehensions)

到现在为止，我们一直把 Haskell 中的字符串看作基本概念。但并非如此，实际上，字符串是由字符构成的列表。举例来说，$''abc'' :: String$ 仅仅是 $['a','b','c'] :: [Char]$ 的缩写形式。正因为字符串是特殊类型的列表，任何适用于列表的多态函数都可以应用于字符串。例如：

$>''abcde'' !!\ 2$
$'c'$

$> take\ 3\ ''abcde''$
$''abc''$

$> length\ ''abcde''$
5

```
> zip "abc" [1,2,3]
[('a',1),('b',2),('c',3)]
```

基于同样的理由，列表推导式也可以用于定义基于字符串的函数，比如返回字符串中小写字母或特定某个字符出现次数的函数分别定义如下：

```
lowers :: String -> Int
lowers xs = length [x | x <- xs, isLower x]
count :: Char -> String -> Int
count x xs = length [x' | x' <- xs, x == x']
```

例如：

```
> lowers "Haskell"
6

> count 's' "Mississippi"
4
```

5.5 凯撒密码 (The Caesar cipher)

5.6 本章备注

名词*推导（comprehension）*来自于集合论的"子集推导公理（axiom of comprehension）"[2]。此公理明确定义了如何通过选择满足给定性质的所有元素来构建一个集合。comprehensions 的更形式化的解释参见 Haskell Report[11]--？？？

5.7 习题

1. 使用列表推导式，给出一个表达式来计算 1-100 的平方和 $1^2 + 2^2 + \cdots 100^2$。

[2]译注：http://mathworld.wolfram.com/AxiomofSubsets.html