HZ BOOKS
华章教育

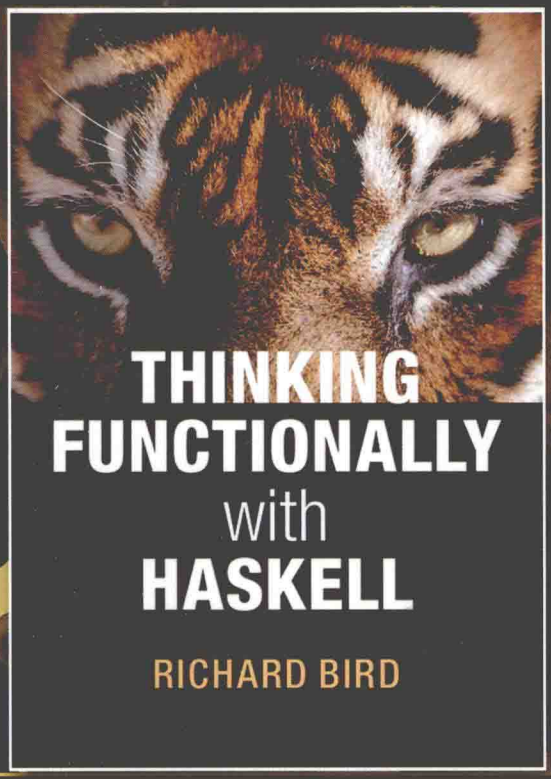
计 算 机 科 学 丛 书

CAMBRIDGE

Haskell函数式程序设计

[英] 理查德·伯德（Richard Bird） 著 乔海燕 译
牛津大学 中山大学

Thinking Functionally with Haskell



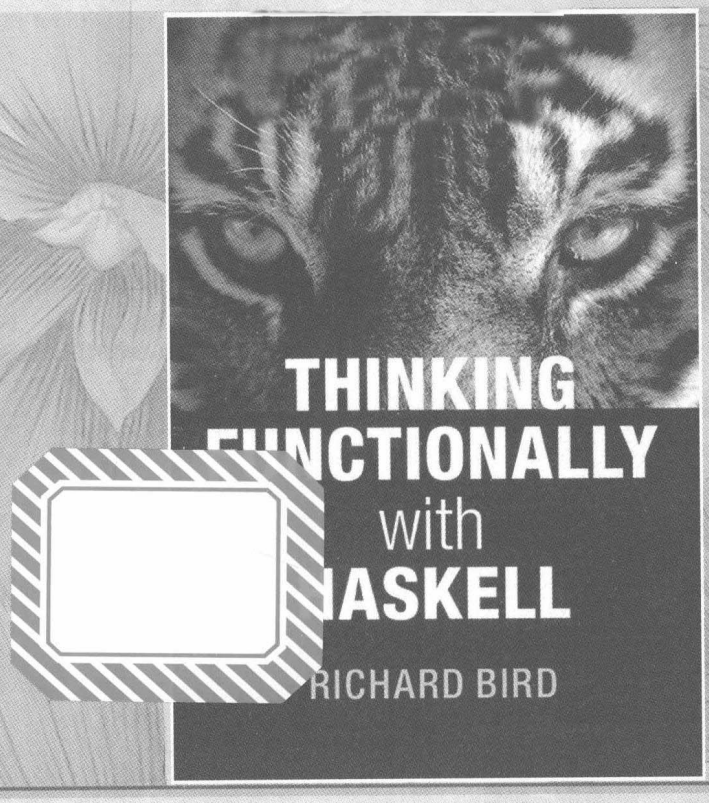
机械工业出版社
China Machine Press

计 [redacted] 学 丛 书

Haskell函数式程序设计

[英] 理查德·伯德（Richard Bird） 著 乔海燕 译
牛津大学 中山大学

Thinking Functionally with Haskell



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Haskell 函数式程序设计 /（英）伯德（Bird，R.）著；乔海燕译 . —北京：机械工业出版社，2016.3
（计算机科学丛书）
书名原文：Thinking Functionally with Haskell

ISBN 978-7-111-52932-3

I. H… II. ①伯… ②乔… III. 函数－程序设计 IV. TP311.1

中国版本图书馆 CIP 数据核字（2016）第 028759 号

本书版权登记号：图字：01-2015-5205

本书通过 Haskell 语言介绍函数式程序设计的基本思想和方法，讲解如何将数学思维应用于程序设计问题，以实现更高效的计算。本书涵盖 Haskell 的诸多特性，但并不是这门语言的参考指南，而是旨在利用丰富的实例和练习揭示函数式程序设计的本质。

本书不要求读者具备程序设计基础，所涉及的数学知识也不高深，既适合初学者阅读，也适合有经验的程序员参考。

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：曲　熠　　责任校对：董纪丽
印　　刷：中国电影出版社印刷厂　　版　　次：2016 年 3 月第 1 版第 1 次印刷
开　　本：185mm×260mm　1/16　　印　　张：15.25
书　　号：ISBN 978-7-111-52932-3　　定　　价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

文艺复兴以来，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自 1998 年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与 Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage 等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出 Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专门为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www. hzbook. com
电子邮件：hzjsj@ hzbook. com
联系电话：（010）88379604
联系地址：北京市西城区百万庄南街 1 号
邮政编码：100037

华章科技图书出版中心

译者序

Thinking Functionally with Haskell

命令式程序设计通过指令序列描述解决问题的方法，这些程序使用命令式语言的指令通过不断改变状态实现从输入到输出的过程，如 C、C ++ 、Java 等。这种程序中虽然也有函数，但是这些函数往往有副作用——不是数学意义上的函数，更准确地说是过程。

函数式程序设计将数学的思维方法应用于程序设计问题，强调一个程序是从输入集合到输出集合的数学函数，而程序员的任务就是定义可实现输入到输出转换的数学函数。近年来，许多函数式程序设计语言已成功用于工业和商业应用的开发，包括 Common List、Erland、F#、Haskell、Racket 和 Scheme 等。函数式程序设计对于程序设计者和程序员也产生了深刻的影响。许多命令式语言也融入了函数式程序的思想，如 C#和 Java 增加了便于使用函数式程序的构件。许多程序员即使在使用命令式语言时也会使用函数式方式编写程序。

Richard Bird 教授常年研究和讲授函数式程序设计和算法设计，本书便是他在多年积累的经验和素材基础上编写的。书中介绍了函数式程序设计的基本思想和方法，这特别要归功于他和 Lambert Meertens 教授创立的通过程序规格说明导出函数程序的方法。本书之所以选择 Haskell，主要在于 Haskell 是一种开放的语言，它实现了多数函数式程序设计的思想和方法，在学术界和工业界都比较流行，同时这些思想也广泛适用于其他函数式程序设计语言。函数式程序设计虽然强调数学思维或者函数思维，但是所涉及的数学知识并不高深，只需读者有高中数学基础，而且，本书也不需要读者有程序设计基础，所以既适用于初学程序设计的读者，也适用于有一定程序设计经验的程序员学习和参考。

由于译者水平有限，书中难免存在问题，请读者和同行批评指正！在此感谢 Richard Bird 教授在本书翻译过程中对我的问题的解答！感谢机械工业出版社的编辑给予的支持和帮助！感谢家人对我的支持！

乔海燕

于广州大学城

2015 年 11 月

本书是《Introduction to Functional Programming Using Haskell, Second Edition》的全新升级，主要变化有：重新组织部分介绍性内容，以适应一个学期或者两个学期课程的不同需要；几个新的实例；100 多道习题及其答案。与以前的版本一样，本书不需要读者具有计算机或者程序设计知识，因此本书适用于计算专业的第一门课程。

在编写教材时，每个作者各具风格，本书也不例外。尽管现在有很多关于 Haskell 的书、教程、文章和博客等，但是很少有人强调用数学思维思考函数式程序设计的能力，在我看来，正是这种能力使其成为有史以来最棒的程序设计方法。这其中所涉及的数学知识既不新也不复杂，任何学过高中数学（如三角函数）并且应用三角函数恒等式化简过正余弦表达式（一个典型的例子：将 $\sin 3\alpha$ 用 $\sin\alpha$ 来表示）的学生很快会发现，在程序设计问题中所要做的工作是完全类似的。使用函数式程序设计所获得的回报是更快的计算。即使在 30 年后，我依然使用这样的方法，并能从中得到很大的快乐：在解决问题时首先从一个简单、明显却不太高效的方法入手，然后应用一些熟知的恒等式，最后得到一个高效 10 倍的解。当然，如果我运气好的话。

如果上一段的最后一句让你失去兴趣，如果你一直在远离数学的“魔多”（Mordor），那么本书可能不适合你。我只是说有这种可能，但也不一定（没有人愿意失去读者）。我们在学习一种新的、令人兴奋的编程方法时仍能得到不少乐趣。即使是那些因为各种原因在日常工作中不能使用 Haskell，而且也没有时间计算更优解的程序员，仍然因学习 Haskell 所带来的享受而倍受鼓舞，而且非常赞赏它既简单又清晰简洁地表达计算思想和方法的能力。事实上，用纯函数式表达程序设计思想的能力已经慢慢地融入了主流的命令式程序设计语言，如 Python、Visual Basic 和 C#。

最后，也是最重要的一点：Haskell 是一种大规模语言，本书不能涵盖一切内容。本书不是 Haskell 的参考指南。尽管 Haskell 语言的细节在每一页出现，特别是在前几章，但是我的初衷是讲解函数式程序设计的本质，用函数思考程序的思想，而不是赘述一种特定语言的特点。但是，过去几年来 Haskell 已经吸收并实现了早期函数语言（如 SASL、KRC、Miranda、Orwell 和 Gofer）中表达的函数式程序设计的大多数思想，而且难以抵挡用这种超酷语言介绍所有这些特性的诱惑。

书中出现的大多数程序可以在下列网页上找到：

www. cs. ox. ac. uk/publications/books/functional

希望将来有更多习题（及答案）和编程项目的建议等可以添加进来。关于 Haskell 的更多信息，读者应该首选官网 www. haskell. org。

致谢

本书源于我基于第 2 版所写的讲义。来自助教和学生的意见和建议为本书增添了很多

光彩。另有很多读者通过电子邮件给出建设性的评论和批评，或者指出书中的打字错误和低级错误。这些读者包括：Nils Andersen，Ani Calinescu，Franklin Chen，Sharon Curtis，Martin Filby，Simon Finn，Jeroen Fokker，Maarten Fokkinga，Jeremy Gibbons，Robert Giegerich，Kevin Hammond，Ralf Hinze，Gerard Huet，Michael Hinchey，Tony Hoare，Iain Houston，John Hughes，Graham Hutton，Cezar Ionescu，Stephen Jarvis，Geraint Jones，Mark Jones，John Launchbury，Paul Licameli，David Lester，Iain MacCullum，Ursula Martin，Lambert Meertens，Erik Meijer，Quentin Miller，Oege de Moor，Chris Okasaki，Oskar Permvall，Simon Peyton Jones，Mark Ramaer，Hamilton Richards，Dan Russell，Don Sannella，Antony Simmons，Deepak D'Souza，John Spanondakis，Mike Spivey，Joe Stoy，Bernard Sufrin，Masato Takeichi，Peter Thiemann，David Turner，Colin Watson 和 Stephen Wilson。特别是 Jeremy Gibbons、Bernard Sufrin 和 José Pedro Magalhães 阅读了初稿，并提出了许多建议。

感谢剑桥大学出版社编辑 David Tranah 持续不断的建议和支持。我现在是牛津大学计算机系荣誉退休教授，感谢计算机系和系主任 Bill Roscoe 的一贯支持。

格式说明

习题

习题 A 请用 $\sin\alpha$ 表示 $\sin 3\alpha$。

答案

习题 A 答案

$$
\begin{aligned}
& \sin 3\alpha \\
= & \ \{\text{算术}\} \\
& \sin(2\alpha + \alpha) \\
= & \ \{\text{因为 } \sin(\alpha+\beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta\} \\
& \sin 2\alpha\cos\alpha + \cos 2\alpha\sin\alpha \\
= & \ \{\text{因为 } \sin 2\alpha = 2\sin\alpha\cos\alpha\} \\
& 2\sin\alpha\cos^2\alpha + \cos 2\alpha\sin\alpha \\
= & \ \{\text{因为 } \cos 2\alpha = \cos^2\alpha - \sin^2\alpha\} \\
& 2\sin\alpha\cos^2\alpha + (\cos^2\alpha - \sin^2\alpha)\sin\alpha \\
= & \ \{\text{因为 } \cos^2\alpha + \sin^2\alpha = 1\} \\
& \sin\alpha(3 - 4\sin^2\alpha)
\end{aligned}
$$

以上证明格式是由 Wim Feijen 发明的，本书将使用这种证明格式。

Richard Bird

第1章
Thinking Functionally with Haskell

何谓函数式程序设计

简而言之：

- 函数式程序设计是一种构造程序的方法，它强调的是函数和函数的应用，而非命令及其运行。
- 函数式程序设计使用简单的数学语言，使得问题的描述更清晰也更简洁。
- 函数式程序设计的数学基础简单，而且支持对函数的性质进行推理。

本书的目的是使用一种称为 Haskell 的函数语言展示以上 3 个特性。

1.1 函数和类型

本书将使用 Haskell 的如下记法：

```
f :: X -> Y
```

它表示 f 是一个函数，其参数类型是 X，返回值的类型是 Y。例如：

```
sin     :: Float -> Float
age     :: Person -> Int
add     :: (Integer,Integer) -> Integer
logBase :: Float -> (Float -> Float)
```

Float 表示像 3.14159 等浮点数类型；Int 表示有限精度整数类型，即满足 $-2^{29} \leqslant n < 2^{29}$ 的整数 n；Integer 表示无精度限制整数类型。在第 3 章将会看到，Haskell 包含了各种数值类型。

数学上用 $f(x)$ 表示将函数 f 应用于其参数 x。但是，也使用如 $\sin\theta$ 来表示 $\sin(\theta)$。在 Haskell 中可以始终使用 f x 表示将 f 应用于参数 x。函数的应用运算用一个空格表示。如果不使用括号，那么必须用空格避免多字母名可能引起的混淆：latex 是一个名，但是 late x 表示函数 late 应用于参数 x。

例如，sin 3.14、sin (3.14)或 sin(3.14)是函数 sin 应用于 3.14 的 3 种合法表示。

类似地，logBase 2 10、(logBase 2) 10 或(logBase 2)(10)都是以 2 为底 10 的对数的正确表示。但是，表达式 logBase (2 10)是错误的。式子 add (3,4)表示 3 与 4 之和，其中的括号是必需的，因为 add 的参数类型是一对整数，而且数对需要用括号和逗号表示。

再看看 logBase 的类型，其参数是一个浮点数，返回值是一个函数。初看起来可能有些奇怪，但再细看则不然：这里的 logBase 2 和 logBase e 恰好表示了数学函数 $\log_2$ 和 $\log_e$。

数学上有形如 log sin x 的表达式。对于数学家来讲，该式子表示 log (sin x)，因为(log sin) x 没有意义。但是在 Haskell 中，必须说明一个式子的含义，而且必须将

该式子写成 `log (sin x)`，因为 Haskell 将 `log sin x` 解释为 `(log sin) x`。在 Haskell 表达式中函数应用是*左结合*的，而且具有最高的*优先级*。（此外，`log` 是 `logBase e` 在 Haskell 中的简写。）

下面是另一个例子。在三角函数中，$\sin 2\theta = 2\sin\theta\cos\theta$。在 Haskell 中该式子写成

```
sin (2*theta) = 2 * sin theta * cos theta
```

我们不仅要显式地表示乘法，而且要使用括号表达确切含义。上式也可以添加更多括号，写成

2

```
sin (2*theta) = 2 * (sin theta) * (cos theta)
```

但是，多加的括号不是必需的，因为函数应用的优先级比乘法的优先级高。

1.2 函数复合

假设 `f :: Y -> Z` 和 `g :: X -> Y` 是两个函数，可以将这两个函数复合成一个新的函数：

```
f . g :: X -> Z
```

该函数将 `g` 应用于类型 `X` 的参数，得到类型 `Y` 的结果，然后将 `f` 应用于这个结果，最后得到类型 `Z` 的结果。我们将始终使用这样的术语：函数*输入参数*，*返回结果*。事实上，有

```
(f . g) x = f (g x)
```

复合的顺序是从右到左，这是因为我们把函数写在其应用的参数左边。英语的“green pig”中形容词“green”解释为函数，它应用于名词短语，得到名词短语。当然，在法语中情况相反。

1.3 例子：高频词

下面通过解决一个问题来说明函数复合的重要性。《战争与和平》中出现最多的 100 个词是哪些？《爱的徒劳》中出现最多的 50 个词是什么？下面将设计一个函数程序求得答案。不过，尽管现在还没到编写一个完整程序的时候，但是，可以通过构造足够的成分来展示函数式程序设计的精髓。

给定的输入是什么？答：一个文本，可视作由字符构成的一个列表。这里的字符既包含可见字符如`'B'`和`','`，也包含空白字符（blank character），如空格和换行符（`' '`和`'\n'`）。注意，单个字符用单引号表示。例如，`'f'`是一个字符，而 `f` 是一个名。Haskell 用 `Char` 表示字符类型，元素类型为 `Char` 的列表类型用`[Char]`表示。这种记法不只适用于字符，例如，`[Int]`表示整数列表，`[Float -> Float]`表示函数列表。

期待的输出是什么？答：如下形式的数据。

3

```
the: 154
of: 50
a: 18
and: 12
in: 11
```

以上显示也是一个字符列表，事实上可看成如下列表：

```
"  the: 154\n  of: 50\n  a: 18\n  and: 12\n  in: 11\n"
```

字符的列表用双引号表示。更多列表知识参见习题。所以，我们需要设计一个函数，不妨称为 commonWords，其类型为

```
commonWords :: Int -> [Char] -> [Char]
```

函数 commonWords n 获得一个字符列表作为输入，返回该列表中 n 个出现最多的词构成的串（字符列表的别名），形如前面所述列表。commonWords 的类型没有使用括号，当然也可以写成

```
commonWords :: Int -> ([Char] -> [Char])
```

当一个类型中有两个相邻的符号 ->时，结合的顺序是自右向左，与函数应用的结合顺序恰恰相反。因此，A -> B -> C 表示A -> (B -> C)。如果想表示类型(A -> B) -> C，那么必须使用括号。更多相关知识参见第 2 章。

明白了给定的输入和期待的输出后，不同的人有不同的解法，对问题的关注点也不尽相同。例如，什么是一个“词”？如何将字符列表转换为词的列表？"Hello"、"hello"以及"Hello!"是不同的词还是相同的词？如何计算词的数目？需要统计所有的词数，还是只要计算最常出现的词数？等等。有些人觉得这些过多的细节使人望而却步，大多数人似乎认为在计算过程中，某个时刻必须获得词与其出现频率的列表，但是如何由该列表实现最终目标呢？是扫描该列表 n 次，每次找出下一个出现次数最多的词，还是有其他更好的方法？

首先考虑词的概念，并简单地假定一个词是不含空格和换行符的最大字符序列。这样的定义允许把诸如"Hello!"、"3 * 4"和"Thelma&Louise"等看作词，但是这没关系。在一个文本中，一个词是有空白字符包围的字符序列，如"Thelma and Louise"包含 3 个词。

4

我们不准备考虑如何将一个文本分解成其组成元素（即词的列表），而是假定存在具有这种功能的函数：

```
words :: [Char] -> [[Char]]
```

诸如[[Char]]这样的类型显得难以记忆，不过在 Haskell 中总是可以引入类型同义词(type synonyms)：

```
type Text = [Char]
type Word = [Char]
```

现在可以这样表达类型 words :: Text -> [Word]，使其更便于记忆。当然，一个文本有别于一个词，前者可以包含空白字符，后者则不然，但是 Haskell 的类型同义词不能表达这种细微的区别。事实上，words 是 Haskell 的库函数，因此不必自定义。

另外一个问题是"The"和"the"是否表示同一个词。它们实际上是同一个词，解决这个问题的一种方法是将文本中的所有字母都转换成小写，其他字符不变。为此，需要一个函数 toLower :: Char -> Char，该函数将大写字母转换成小写字母，其他字符保持不变。为了将该函数应用于文本的每个字符，需要下面的通用函数：

```
map :: (a -> b) -> [a] -> [b]
```

使得 map f 应用于一个列表时，f 被应用于列表的每个元素。这样，将每个字母转换为小写由下列函数完成：

```
map toLower :: Text -> Text
```

好了，现在得到了将文本转换为小写字母词的列表函数 words . map toLower。下一个任务是计算每个词出现的次数。可以扫描词的列表，检查下一个词是第一次出现还是已经出现过，相应地开始新词的计数或者给对应词的计数器加 1。不过，另一种更简单的想法是对词的列表按照字典序排序，结果是所有重复出现的词排在了一起。人工操作时不会这样做，但是通过排序获得信息的思想或许是计算过程中最重要的算法思想。所以，假设存在一个函数：

```
sortWords :: [Word] -> [Word]
```

5 该函数将词的列表按照字典序排序。例如：

```
sortWords ["to","be","or","not","to","be"]
        = ["be","be","not","or","to","to"]
```

下一步需要计算在有序列表中每个词连续出现的次数。假定已有计算词数的函数：

```
countRuns :: [Word] -> [(Int,Word)]
```

例如：

```
countRuns ["be","be","not","or","to","to"]
  = [(2,"be"),(1,"not"),(1,"or"),(2,"to")]
```

其结果是按字典序排列的词及其出现次数的列表。

现在来考虑关键的思想：希望数据按照词的出现次数从大到小排列，而不是按照词的字典序排列。可以看出，这就是一种排序，无需设计其他更聪明的方法。如前所述，排序确实是程序设计中非常有用的方法。因此，假定已有函数：

```
sortRuns :: [(Int,Word)] -> [(Int,Word)]
```

该函数将词及其出现次数按照出现次数（列表元素的第一个分量）递减排序。例如：

```
sortRuns [(2,"be"),(1,"not"),(1,"or"),(2,"to")]
       = [(2,"be"),(2,"to"),(1,"not"),(1,"or")]
```

接下来只需取出结果列表中的前 n 个元素。为此，需要下列函数：

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

该函数使得 take n 取得一个列表的前 n 个元素。函数 take 并不关心列表中的元素是什么类型，这就是 take 的类型签名中出现了 a，而不是(Int,Word)的原因。第 2 章将解释这种思想。

最后的步骤仅仅是整理格式。首先将每个元素转换成一个串，例如，将(2,"be")转换为"be 2 \n"。将该函数称为

```
showRun :: (Int,Word) -> String
```

类型 String 是 Haskell 的预定义类型，实际上是[Char]的类型同义词。因此，下列函数将词及其次数列表转换为串列表：

```
map showRun :: [(Int,Word)] -> [String]
```

最后一步需要使用下列函数：

6

```
concat :: [[a]] -> [a]
```

该函数将元素的列表的列表串联成一个列表。同样，函数 concat 并不关心这里串联的是什么“元素”，这也是类型中出现 a 的原因。

下面定义函数：

```
commonWords :: Int -> Text -> String
commonWords n = concat . map showRun . take n .
                sortRuns . countRuns . sortWords .
                words . map toLower
```

函数 commonWords 定义中使用了 8 个分函数，并用函数复合将它们管道式粘合起来。并非每个问题都可以这样直接地分解成一系列子问题，但是，如果可行的话，最后的程序将是简单、迷人而且有效的。

需要注意的是分解问题的过程是如何在辅助函数的类型指导下进行的。第二个经验（第一个经验是函数复合的重要性）是，确定一个函数的类型是找到该函数合适定义的第一步。

本节的目的是设计一个解决高频词问题的程序。结果是利用辅助函数给出了 commonWords 的函数定义，这些辅助函数或者可以直接定义，或者由某个 Haskell 函数库提供。脚本（script）是一些定义的集合，所以，我们实际上构造了一个脚本。脚本中函数定义的顺序并不重要。函数 commonWords 的定义完全可以放在最前面，然后再定义辅助函数，或者先定义辅助函数，最后给出主要函数的定义。换言之，程序员可以在脚本中用任何顺序叙述故事。稍后将解释如何用脚本进行计算。

1.4 例子：数字转换为词

本节讨论另一个例子，并给出完整解。这个例子展示了求解问题的另一个基本方面，即解决一个复杂问题的好方法，首先是简化问题，然后考虑如何解决更简单的问题。

有时需要把数字写成词。例如：

7

What is functional programming?

```
convert 308000 = "three hundred and eight thousand"
convert 369027 = "three hundred and sixty-nine thousand and
                  twenty-seven"
convert 369401 = "three hundred and sixty-nine thousand
                  four hundred and one"
```

我们的目标是设计这样一个函数：

```
convert :: Int -> String
```

即对于一个给定的不超过 100 万的非负数，函数返回用词表示的数字。如上所述，String 是 Haskell 预定义的类型[Char]的同义词。

这里需要其中各个数字的名称。一种方法是用串的列表表示它们：

```
> units, teens, tens :: [String]
> units = ["zero","one","two","three","four","five",
```

```
>          "six","seven","eight","nine"]
> teens = ["ten","eleven","twelve","thirteen","fourteen",
>          "fifteen","sixteen","seventeen","eighteen",
>          "nineteen"]
> tens  = ["twenty","thirty","forty","fifty","sixty",
>     .    "seventy","eighty","ninety"]
```

以上每行开始的字符 > 表示什么？答案是，在一个脚本中，该字符表示一行 Haskell 代码，而不是注释。用 .lhs 做扩展名的 Haskell 文件称为 Haskell 文学脚本（Literate Haskell Script），习惯上脚本的每一行都是注释，除非出现符号 >，该符号表示随后的是 Haskell 代码行。Haskell 不允许代码行和注释紧邻，所以代码行和注释之间至少应该有一行空白。事实上，你正在阅读的本章就是一个合法的 .lhs 文件，完全可以将该文件载入 Haskell 系统并交互运行。在今后的章节将不再延续这种传统（除此之外，我们被迫用不同的名表示一个函数的不同定义），但是，本章展示的文学编程允许使用任何顺序讨论和书写函数的定义。

对于当前的任务，解决复杂问题的一个好方法是先解决一个更简单的问题。该问题的最简单情况是给定的数字只有一位数，即 $0 \leqslant n < 10$。假定用 convert1 解决这种简单情
8 况。现在马上可以定义：

```
> convert1 :: Int -> String
> convert1 n = units!!n
```

这个定义使用了列表索引运算（!!）。对于给定的列表 xs 和下标 n，表达式 xs!!n 返回 xs 中位置为 n 的元素，其中位置从 0 开始计算。特别地，units!!0 = "zero"。而且，units!!10 确实无定义，因为 units 只有 10 个元素，下标是 0 ~ 9。一般地，在一个脚本中定义的函数是*部分函数*或*不完全函数*，即并非对每个参数返回确切定义的结果。

这个问题的下一个最简单情况是数字 n 最多有两位数，即 $0 \leqslant n < 100$。假定 convert2用于处理这种情况。因为需要知道每位数字是什么，所以首先定义：

```
> digits2 :: Int -> (Int,Int)
> digits2 n = (div n 10, mod n 10)
```

数字 div n k 是 n 被 k 除的整数商，mod n k 是余数。也可以写成

```
digits2 n = (n `div` 10, n `mod` 10)
```

其中运算'div'和'mod'是 div 和 mod 的中缀形式，即运算写在运算数的中间，而不是写在运算数之前。这种形式非常便于改善可读性。例如，数学家会用 x div y 和 x mod y 表示这些表达式。注意，反引号（`）不同于表示单个字符的单引号（'）。

现在可以定义：

```
> convert2 :: Int -> String
> convert2 = combine2 . digits2
```

函数 combine2 的定义使用 Haskell 的*条件等式*（guarded equation）：

```
> combine2 :: (Int,Int) -> String
> combine2 (t,u)
>   | t==0          = units!!u
>   | t==1          = teens!!u
>   | 2<=t && u==0 = tens!!(t-2)
>   | 2<=t && u/=0 = tens!!(t-2) ++ "-" ++ units!!u
```

欲理解这段代码，需要知道 Haskell 表达等式和比较测试的如下符号： 9

==（等于）

/=（不等于）

<=（小于等于）

这些函数具有确切的类型，这将在稍后解释。

我们还需要知道两个测试的合取用 && 表示。因此，如果 a 和 b 都是 True，那么 a && b 返回布尔值 True，否则返回 False。实际上，有

```
(&&) :: Bool -> Bool -> Bool
```

第 2 章将进一步介绍类型 Bool。

最后，(++)表示两个列表串联的运算。该运算不关心列表的元素类型，所以

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

例如，下列等式将两个函数（函数类型均为 Float -> Float）列表串联：

```
[sin,cos] ++ [tan] = [sin,cos,tan]
```

也可以串联两个字符列表：

```
"sin cos" ++ " tan" = "sin cos tan"
```

函数 combine2 的定义是在仔细考虑了所有可能的情况后得到的。稍加思考后可以看出，这里有 3 种主要情况，即十位数为 0、1 或者大于 1 的 3 种情况。对于前两种情况，答案是直接的，但是第 3 种情况需要划分为两种情况，即个位数是 0 或者非 0。这些情况的书写先后顺序，也就是这些条件等式的先后顺序并不重要，因为这些条件互不相交（两种情况不会同时为真），并且覆盖了所有的情况。

也可以定义：

```
combine2 :: (Int,Int) -> String
combine2 (t,u)
  | t==0      = units!!u
  | t==1      = teens!!u
  | u==0      = tens!!(t-2)
  | otherwise = tens!!(t-2) ++ "-" ++ units!!u
```
10

但是，这里书写等式的顺序很重要。条件的计算是自上而下的，并将第一个计算为 True 的条件对应的等式右边作为函数定义的结果。标识符 otherwise 是 True 的同义词，所以它涵盖了所有其他情况。

定义 convert2 的另一种方法：

```
convert2 :: Int -> String
convert2 n
  | t==0      = units!!u
  | t==1      = teens!!u
  | u==0      = tens!!(t-2)
  | otherwise = tens!!(t-2) ++ "-" ++ units!!u
  where (t,u) = (n `div` 10, n `mod` 10)
```

这里使用了 where 子句。这种子句引入了*局部定义*，其*上下文或辖域*是 convert2 定义的所有等式右边部分。这种局部定义对于定义的组织并使得定义可读性更强是非常重要

的。对于本例来说，where 子句避免了显式地定义函数 digits2。

以上定义相对简单。现在考虑函数 convert3，其参数 n 满足 $0 \leqslant n < 1000$，即 n 最多有 3 位数。其定义如下：

```
> convert3 :: Int -> String
> convert3 n
>   | h==0      = convert2 t
>   | t==0      = units!!h ++ " hundred"
>   | otherwise = units!!h ++ " hundred and " ++ convert2 t
>   where (h,t) = (n `div` 100, n `mod` 100)
```

使用这样的方式将数字分解，是因为可以使用 convert2 处理小于 100 的数字。

现在假定 n 满足 $0 \leqslant n < 1\,000\,000$，即 n 可以有 6 位数。沿用以上的模式，可以给出如下定义：

```
> convert6 :: Int -> String
> convert6 n
>   | m==0      = convert3 h
>   | t==0      = convert3 m ++ " thousand"
>   | otherwise = convert3 m ++ " thousand" ++ link h ++
>                 convert3 h
>   where (m,h) = (n `div` 1000,n `mod` 1000)
```

11

对于 $0 < m$ 且 $0 < h < 100$，表示 m 的词与表示 h 的词之间需要一个连接词“and”，所以定义：

```
> link :: Int -> String
> link h = if h < 100 then " and " else " "
```

该定义使用了条件表达式：

```
if <test> then <expr1> else <expr2>
```

也可以使用条件等式：

```
link h | h < 100   = " and "
       | otherwise = " "
```

根据不同情况，有时一种表达式可读性较强，有时另一种可读性更强。这里的 if、then 和 else，以及其他的一些词，称为 Haskell 保留字，这也意味着程序员不可以使用这些词做其他定义的名称。

注意函数 convert6 是如何使用简单的函数 convert3 来定义的，同时注意 convert3 是如何用更简单的函数 convert2 来定义的。这是函数定义的一般方法。在本例中，简单情况的考虑都派上了用场，因为最后函数的定义使用了简单情况的定义。

另外一点，把所求的函数命名为 convert6，但是开始时称该函数为 convert。没关系，可以定义：

```
> convert :: Int -> String
> convert = convert6
```

下面需要做的是将函数 convert 应用于一些输入参数。怎么做呢？

1.5 Haskell 平台

访问网页 www. haskell. org 可以看到如何下载 Haskell 平台（Haskell Platform）。该平台