

提供各种IT类书籍pdf下载，如有需要，请 QQ: 2011705918

注：链接至淘宝，不喜者勿入！整理那么多资料不容易，请多多见谅！非诚勿扰！

更多资源请点击



[更多资源请点击](#)



Haskell 函数式编程入门

张 淞○编著

人民邮电出版社
POSTS & TELECOM PRESS

```
squareRoot :: Int -> Double -> Double
squareRoot 0 x = x
squareRoot n x = (squareRoot (n-1) x + x `div` squareRoot (n-1) x) `div` 2.0
c & x | c x == f x = x
      | otherwise = fix c f (f x)
data Exp = Const Double | Var String | Neg Exp | Add Exp Exp | Sub Exp Exp | Mult Exp Exp | Div Exp Exp | Power Exp Exp | Log Exp | Ln Exp | Sin Exp | L_Par Exp | R_Par Exp | Op_Expr String Exp | Option Exp | WS String
runWS :: (Show a, Monad w) -> w a
runWS (Fq s) = WS $ s
runWS (Show s) = WS $ s
runWS (Let (r, ns, m')) = runWS m' `then` r
runWS (Op op e1 e2) = WS $ op `then` e1 `then` e2
runWS (Option e) = WS $ e
runWS (L_Par e) = WS $ L_Par `then` e
runWS (R_Par e) = WS $ R_Par `then` e
runWS (WS s) = WS s
```

Haskell 函数式编程入门

张 淞◎编著

人民邮电出版社
北京

图书在版编目（CIP）数据

Haskell函数式编程入门 / 张淞编著. — 北京 : 人
民邮电出版社, 2014.3
ISBN 978-7-115-33801-3

I. ①H… II. ①张… III. ①函数—程序设计 IV.
①TP311.1

中国版本图书馆CIP数据核字(2013)第277063号

内 容 提 要

这是一本讲解 Haskell 这门经过精心设计和锤炼的纯函数式编程语言的书，同时也是一本通过 Haskell 来讲解函数式编程的方法与思想的书。全书共分三个部分。第一部分介绍函数式编程在解决数学与算法问题的精简与直观的特色，让不熟悉 Haskell 的读者对其建立初步的了解，同时通过解决一些算法问题，如裴波那契数列、八皇后问题、排序问题、24 点等，引发一些对函数式编程方式的思考；第二部分介绍一些略微深入的 Haskell 内容，包括函数、Monoid、IO 与 Monad 转换器等；最后一部分则涉及快速测试、惰性求值和并行编程等主题。

本书既适合对 Haskell 和函数式编程感兴趣的程序员阅读，又适合作为 Haskell 语言入门教程，供计算机科学与数学专业的学生参考。

◆ 编 著 张 淞
责任编辑 杨海玲
责任印制 程彦红 焦志炜
◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京艺辉印刷有限公司印刷
◆ 开本：800×1000 1/16
印张：23.5
字数：538 千字 2014 年 3 月第 1 版
印数：1~3 000 册 2014 年 3 月北京第 1 次印刷

定价：59.00 元

读者服务热线：(010)81055410 印装质量热线：(010)81055316
反盗版热线：(010)81055315

写给我亲爱的父母与曾经的自己。

前言

高级计算机语言的诞生大约是在 20 世纪 50 年代。虽然那时电子计算机的发展才刚刚起步，但计算机语言的设计却逐渐分成了两个阵营。

第一个阵营的设计直接基于计算机硬件的基本结构。这些语言的理念主要是让它们对计算机的内存、磁盘以及其他硬件的存储状态进行直接的操作，如 Pascal、C 语言等。它们常常被称为命令式（imperative）、顺序式（sequential）或者过程式（procedural）编程语言。在语句执行过程中，各个语句的先后顺序十分重要。

而另外一个阵营是希望计算机语言的设计直接从数学函数的角度出发，将计算过程抽象成函数运算，这种编程方式是对程序的一种更高层次的抽象。虽然起初在运行效率上并不是很高，但表达起来十分简洁，并且设计者也在努力，逐步使这类程序编译后更适应计算机的底层硬件以获得更高的运行效率，这一类语言就是函数式编程语言。

在顺序式计算机语言的发展进程中，它们的流行程度大致可以分成两种极端情况：一部分是像 C、C++、Java、C#这样的语言，一举成名，成为现在计算机编程语言的主流；另外是一些“冷门”的语言，可能是一些失败的实验探索性语言，也可能是一些未被很好地商业化的语言。但是，函数式编程语言在顺序式编程语言取得发展并广泛应用或者消失的几十年当中，被应用的程度却一直处于不温不火的状态，虽然没有取得广泛的商业化应用，但是有关的研究和发展从未间断。而软件行业发展到今天，由于顺序式编程语言在解决一些特定问题时会引起程序复杂度倍增和可靠程度降低等问题，人们的视角渐渐投向了函数式编程语言，这使得函数式编程语言在逐步兴起。

函数式编程语言的诞生可以追溯到 20 世纪 50 年代的 Lisp。那时的计算机处理器不是很强，函数式编程语言程序的效率要差一些，所以在当时一直没有变成主流。而如今的处理器速度的提升以及编译器的优化，使得函数式编程语言运行的效率已经不再是一个困扰性的问题。可是，顺序式语言也在发展，并且还加入了很多新的吸引人的特性，在软件工程的发展中占据了绝对的优势。即便这样，函数式编程仍然有着其他语言不可能有的优势，所以它的热度在一点一点地上升，比如微软公司就增强了自家.NET Framework 的函数式编程语言 F#的功能，并且也能在除 Windows 之外的其他操作系统上使用。除此之外，Java、C++、C#在新的版本中也引入了 λ 表达式，这个想法其实就是源于函数式编程的。其他的语言（如 Python、Ruby、Scala 等）中同样可以看到函数式编程思想的体现。

函数式编程语言有哪些呢？函数式编程语言其实也是一个很大的家族，它是一类语言，就

像命令式编程语言并不单单指 C 语言或 Java 一样，一些常见的函数式编程语言有：Lisp^①、Scheme、Ocaml、ML(Meta-Language)、Coq^②、Erlang^③、Haskell、Agda^④、F#、Clojure^⑤等。函数式语言表达程序十分精炼，但是这些并不说明像 Java 这样面向对象的顺序式编程语言冗长，各种语言在计算机发展进程中百花齐放，各有各的长处与用途。面向对象语言有很大的优势，各种设计模式在商业开发的路上也发展得非常成熟，而函数式编程的优势在于程序的严谨与可靠性，程序正确性的证明与测试时的简易性，另外，还有开发周期相对短，编写并发程序十分简洁且运行稳定。函数式编程的应用虽然在很长时间内都处于不温不火的状态，但它们的用途却非常广泛，常见的领域有人工智能、定理证明、无线通信、金融数据分析系统等。

另外值得一提的是，在 Matlab 和 Wolfram Mathematica 这类理工科、工程学软件中，也常常能见到函数式编程的身影与思想的体现，很多基本的函数应用其实都是基于函数式编程的思想。例如，在《Mathematica Cookbook》一书中，第 2 章讲的便是 Functional Programming，即函数式编程，并且书中说学习函数式编程是掌握 Mathematica 最重要的部分。所以，学习函数式编程不仅仅是学习一种新的编程语言，而是在更好地体会另外一种编程思想，这种编程思想在日后的学习与工作中可能会尤为重要。

本书通过 Haskell 这样一门语法经过精心设计和锤炼的纯函数式编程语言来讲解函数式编程的方法与思想。虽然很多例子都是解决数学与算法问题的，但也有一些例子是对 Haskell 在实际当中应用的展示，旨在说明函数式编程语言不单单是另外一种编程方式或者只是在教学与研究中使用的语言，同时也是一门可以解决现实编程问题的、非常务实的编程语言。Haskell 实现了很多软件中的精品，如窗口管理器 XMonad、Perl 6 的 Haskell 实现 Pugs 以及高性能的网页框架 Yesod、Snap 等。

书中的例子都尽量保证以短小为主，以方便读者根据需要做跳跃、选择性的阅读。前半部分将简单介绍函数式编程在解决数学与算法问题时精简与直观的特色，使一些不熟悉 Haskell 的读者对其建立初步的了解，通过解决一些数学算法问题，比如斐波那契数列、八皇后问题、排序问题、24 点等，引发一些对这种新的编程方式的思考。

中间部分则是一些关于 Haskell 略为深入的介绍，包括函数、monoid、IO 与 monad 转换器等。其中，monad 的概念在 Haskell 中十分重要，所以建议读者可以花相对比较长的时间来实践、理解并做一些其他相关的阅读。这里值得说明的是，由于 Haskell 语言的特殊性，笔者将处理输入与输出的内容安排在了中间这一部分，也就是说，在第 11 章以前，读者将不会见到可在操作系统上直接运行的程序。

这本书最后几章的主题则是关于快速测试、惰性求值还有并行与并发编程的。这几章的内

① 被认为是首个函数式编程语言，其他函数式编程语言的鼻祖，1958 由斯坦福大学教授 John McCarthy 开发。

② 基于 Ocaml，主要用于辅助定理证明，是一个证明辅助工具（proof assistant）。

③ Erlang 是瑞典爱立信计算机实验室为开发实时、高并发、分布式系统还有提高软件安全与稳定性而研发的语言。

④ Agda 是基于 Haskell 的一门依赖类型的函数式编程语言，与 Coq 相似，也是主要用于辅助定义证明的工具。

⑤ Clojure 为 Lisp 的变种，由 Rich Hickey 开发，可运行于 Java 虚拟机之上，具有良好的可移植性。

容相对零散，但是都讨论的是关于 Haskell 的非常重要的内容。其中，快速测试一章将讨论如何使用 QuickTest 库来测试函数的正确性。测试这一课题在软件开发中非常重要而且是比较困难的，而我们会看到 Haskell 却能以一种较好的方式处理。惰性求值是 Haskell 最重要的特性之一，也就是说，在函数估值计算时，仅仅计算需要计算的部分。这在某些情况下是个好主意，但是在另外一些情形下可能会严重降低程序的效率，所以了解它并且正确地用好它很重要。最后一章将讨论如何在 Haskell 中写可以并发运行的函数。由于 Haskell 的纯函数特性，它可以随意地并发多个线程来计算纯函数，因为纯函数计算不会互相干扰。当需要共享变量时，可以使用另外一种非常好的处理并发的机制就是软件事务内存（STM）。经过多年的研究，STM 已成为了一种使用简单、运行高效、易于扩展的处理并发计算的方式。

有些章节后会有一些开放性的思考题，但并不是很难，读者可以试着独立完成，也可以在 Haskell 的中文论坛上与其他读者讨论，得到解决问题的思路或者更好的解决方法。

书的大部分是由笔者在课余时间以及假期完成的，但其中部分章节是由来自安徽的中国科技大学不愿透露姓名的博士朋友执笔撰写的。这些部分是 9.5 节的定长列表以及 9.6 节的运行时重载，还有 11.3 节中的实现 printf 函数。但是笔者对其原稿的一些内容做了适当的调整与修改，以保证全书风格的一致和内容的连贯与顺畅，在此感谢他的帮助与支持。

如果读者没有学过任何编程语言，那么以 Haskell 函数式语言作为第一语言就不必对思维进行转换了，我想说你非常幸运。如果读者有一些其他函数式编程语言的基础，那么想再了解一下 Haskell 那就再好不过了。了解其他顺序式语言的读者在学习 Haskell 时需要注意，由于 Haskell 中是没有内存变量和循环的，因此不会出现变量的赋值、循环。所以，如果读者已经深谙 C 语言或者 Java 等顺序式编程之道的话，那么在开始学习函数式编程时可能会产生一些不适应，毕竟没有 for、while 循环还有内存变量的编程方式不太容易一开始就能让熟练使用 C 语言与 Java 的人所接受。但是，一旦适应了 Haskell 函数式编程的这种方式，读者可能会发现这种编程方式的效率是相当高的，也有助于看清与理解程序算法的本质。另外，值得再次指出的是，由于 Haskell 的纯函数特性，输入/输出是在 IO Monad 中处理的，所以笔者决定等读者对 Monad 有了一定的了解之后再讨论，因此，在学习 IO Monad 以前，只能在解释器（GHCi）中运行程序。有些读者可能会因此感觉单调，但是了解了 IO Monad 以后，读者会发现这种设计是很有道理的。此外，有一些理论部分可能看上去有些枯燥，读者也没有在其他编程语言中涉及过，并且乍看上去感觉并没有什么用途，但是在编程实践中却恰恰相反，所以读者一定要看懂理论部分，让理论来指导编程实践。

相信读者已经感觉到本书虽然是用中文编写，但在阅读的过程当中却少不了谈论一些英文术语。有些术语是无法生硬地一对一翻译的，所以笔者给出了自己的翻译的同时保留了其英文术语，也有的术语并未做出任何翻译，因为生硬的翻译并不会帮助读者理解，所以保留了原有的英文，以便读者参阅相关的英文资料。

本书编写的目的是希望可以让越来越多的计算机专业的同学以及其他计算机行业的人只

需要花费相对较少的时间就对函数式编程有一定的了解，同时也希望能够达到抛砖引玉的效果。因为本书的编写是从笔者自己对 Haskell 和函数式编程了解不多之时开始的，希望这能成为本书的优点，因为非常熟悉了解 Haskell 的人可能很难再以一个初学者的角度来阐述它。当然，这样的缺点则使书的内容会局限于我对 Haskell 的理解程度。如果读者对有些内容有更深入的理解或者认为书中有解释不合理或者内容性的错误，欢迎发邮件同我探讨。最后，当然无论作者与编辑多么努力，总还是会有一些错误藏匿书中，希望广大读者指出。

读者有任何问题欢迎通过 Haskell.Zhang.Song@hotmail.com 与我联系。

张淞
2013 年 10 月于诺丁汉

致 谢

在这里，我想衷心地感谢诺丁汉大学和那些曾经无私地帮助过我的人们。正因为有你们的关怀与教导才使我学到越来越多关于计算机科学的知识，让我看到了计算机科学的前景并意识到了在中国有关函数式编程中文书籍的缺乏，致使我萌生了写一本关于函数式编程的书的念头，正是由于你们不断地支持我，最后促使我完成这本书的编写。

首先感谢 Graham Hutton 教授在我学习高级函数式编程课的时候给予我的详细讲解和对我本科的 Haskell 程序毕业设计的悉心指导。

感谢 JP Moresmau 先生对 EclipseFP 的无偿开发与在 EclipseFP Github 论坛上对 EclipseFP 的使用中出现的问题给予的耐心解决。

这里，我想特别感谢 Nilsson Henrik 教授在百忙之中抽出时间对本书撰写的关注、支持，以及对我在编写本书中遇到的疑问的解答。是您的编译原理和编程基础两门课让我对 Haskell 与函数式编程有了更为深入的了解和认识。同时，我也非常荣幸能经常与您在诺丁汉大学 Jubilee 校园里的 Amenities Cafe 一起讨论函数式编程和 Haskell。

感谢我的私人导师 Roland Backhouse 教授在算法课上对于一些算法构造的细致讲解，以及对于我提出的有关算法构造与复杂度问题的耐心解答，也谢谢您在生活上给予我的帮助。此外，也感谢您为我研究生申请提供的推荐信，成为您的学生我感到无比的荣幸。

感谢 Thomas Anberree 讲师的函数式编程课，正是这门课使我了解了一种全新的编程方式。此外，我很感激您对学习函数式编程有困难的学生进行了课外辅导并且给出了更多的练习题，还有您在教师公寓楼下单独为我讲解复合函数的类型。

此外，我十分感谢 Roland Backhouse 教授的学生陈炜博士，感谢您在大学课后组织的数学和算法问题的讨论课，虽然范围很宽松，涉及排列组合、命题逻辑证明、机器学习和 RSA 加密等，但是让我学到了课堂以外的更多内容。同时，我也十分高兴在平时常常能有机会和您一起运动和讨论问题。

感谢 Haskell 中文论坛 <http://www.haskellcn.org> 的创始人吴海生为喜欢 Haskell 与函数式编程的朋友提供了一个非常好的平台。

感谢来自中国科技大学的朋友在博士在读期间帮忙编写函数响应式编程等章节，虽然函数响应式编程一章由于结构与内容的问题最终未出现在本书中，但还是特别感谢。

最后，我还想感谢曾经教过我的老师们，他们是费继娟老师、李丹老师、李加福老师和于华民老师。谢谢您们给我的关心与帮助，伴我一点一滴地从一个孩子成长起来。谢谢您们带我走入这个充满挑战的学科，让我踏上了一条充满奇幻色彩的路。

目 录

第 1 章 Haskell 简介.....	1
1.1 Haskell 的由来	1
1.2 Haskell 编译器的安装以及 编写环境	3
1.3 GHCi 的使用	4
1.3.1 GHCi 中的命令.....	5
1.3.2 在 GHCi 中调用函数.....	5
1.4 .hs 和.lhs 文件、注释与库函数	7
1.5 第一个 Haskell 程序 HelloWorld!.....	7
本章小结.....	8
第 2 章 类型系统和函数.....	9
2.1 Haskell 的类型与数据	9
2.1.1 Haskell 常用数据类型.....	9
2.1.2 函数类型	14
2.1.3 类型的别名	17
2.1.4 类型的重要性	18
2.2 Haskell 中的类型类	19
2.2.1 相等类型类: Eq	20
2.2.2 有序类型类: Ord	20
2.2.3 枚举类型类: Enum	21
2.2.4 有界类型类: Bounded	21
2.2.5 数字类型类: Num	22
2.2.6 可显示类型类: Show	25
2.2.7 小结	25
2.3 Haskell 中的函数	26
2.3.1 Haskell 中的值	26
2.3.2 函数思想入门	27
2.3.3 函数的基本定义格式	28
2.3.4 λ 表达式	30
2.3.5 参数的绑定	34
2.4 Haskell 中的表达式	35
2.4.1 条件表达式	35
2.4.2 情况分析表达式	36
2.4.3 守卫表达式	37

2.4.4 模式匹配.....	37
2.4.5 运算符与函数.....	38
2.4.6 运算符与自定义运算符.....	38
本章小结.....	41
 第3章 基于布尔值的函数.....	42
3.1 关键字 module 与 import 简介.....	42
3.2 简易布尔值的函数.....	43
3.3 与非门和或非门.....	46
本章小结.....	47
 第4章 库函数及其应用.....	48
4.1 预加载库函数.....	48
4.1.1 常用函数.....	48
4.1.2 基于列表的函数.....	50
4.1.3 定义历法公式.....	57
4.1.4 字符串处理的函数.....	58
4.2 字符与位函数库简介.....	60
4.2.1 Data.Char.....	60
4.2.2 Data.Bits.....	60
本章小结.....	61
 第5章 递归函数.....	62
5.1 递归函数的概念.....	62
5.2 简单递归函数.....	64
5.3 扩展递归与尾递归.....	66
5.4 互调递归.....	68
5.5 麦卡锡的 91 函数.....	69
5.6 斐波那契数列.....	69
5.7 十进制数字转成罗马数字.....	73
5.8 二分法查找.....	74
5.9 汉诺塔.....	75
5.10 排序算法.....	78
5.10.1 插入排序.....	78
5.10.2 冒泡排序.....	81
5.10.3 选择排序.....	83
5.10.4 快速排序.....	84
5.10.5 归并排序.....	86
小结.....	91
5.11 递归基本条件与程序终止.....	91
5.12 递归与不动点.....	92
5.13 无基本条件递归和惰性求值.....	94

本章小结.....	96
第 6 章 列表内包.....	97
6.1 列表生成器.....	97
6.2 素数相关趣题.....	99
6.3 凯撒加密.....	101
6.3.1 加密.....	102
6.3.2 解密.....	102
6.4 排列与组合问题.....	104
6.4.1 排列问题.....	104
6.4.2 错位排列问题.....	105
6.4.3 组合问题.....	106
6.5 八皇后问题.....	107
6.6 计算矩阵乘法.....	111
6.7 最短路径算法与矩阵乘法.....	112
本章小结.....	116
第 7 章 高阶函数与复合函数.....	117
7.1 简单高阶函数.....	117
7.2 折叠函数 foldr 与 foldl.....	119
7.3 mapAccumL 与 mapAccumR 函数.....	125
7.4 复合函数.....	126
本章小结.....	128
第 8 章 定义数据类型.....	129
8.1 数据类型的定义.....	129
8.1.1 枚举类型.....	129
8.1.2 构造类型.....	132
8.1.3 参数化类型.....	134
8.1.4 递归类型.....	138
8.1.5 杂合定义类型.....	140
8.2 类型的同构.....	142
8.3 使用 newtype 定义类型.....	146
8.4 数学归纳法的有效性.....	148
8.5 树.....	150
8.6 卡特兰数问题.....	151
8.7 霍夫曼编码.....	152
8.8 解 24 点.....	154
8.9 zipper.....	157
8.10 一般化的代数数据类型.....	159
8.11 类型的 kind.....	162
8.11.1 类型的 kind.....	162

8.11.2 空类型的声明	164
本章小结	165
第 9 章 定义类型类	166
9.1 定义类型类	166
9.2 Haskell 中常见类型类	169
9.2.1 常用类型类	169
9.2.2 Functor	171
9.2.3 Applicative	173
9.2.4 Alternative	177
9.2.5 简易字符识别器	179
9.2.6 Read 类型类	182
9.2.7 单位半群 (Monoid)	182
9.2.8 Foldable 与 Monoid 类型类	184
9.2.9 小结	186
9.3 类型类中的类型依赖	187
9.4 类型类中的关联类型	192
9.5 定长列表	193
9.6 运行时重载	197
9.7 Existential 类型	198
本章小结	199
第 10 章 Monad 初步	201
10.1 Monad 简介	201
10.2 从 Identity Monad 开始	204
10.3 Maybe Monad	206
10.4 Monad 定律	209
10.5 列表 Monad	210
10.6 Monad 相关运算符	210
10.7 MonadPlus	211
10.8 Functor、Applicative 与 Monad 的关系	213
本章小结	215
第 11 章 系统编程及输入/输出	216
11.1 不纯函数与副作用	216
11.2 IO Monad	218
11.3 输入/输出处理	222
11.3.1 Control.Monad 中的函数	222
11.3.2 系统环境变量与命令行参数	224
11.3.3 数据的读写	225
11.3.4 格式化输出 printf 函数	228
11.3.5 printf 函数的简易实现	229

11.4 星际译王词典.....	233
11.4.1 二分法查找.....	234
11.4.2 散列表的使用.....	237
11.5 简易异常处理.....	239
11.6 Haskell 中的时间.....	244
本章小结.....	245
第 12 章 记录器 Monad、读取器 Monad、状态 Monad.....	246
12.1 记录器 Monad.....	246
12.1.1 MonadWriter.....	248
12.1.2 记录归并排序过程.....	249
12.2 读取器 Monad.....	250
12.2.1 MonadReader.....	251
12.2.2 变量环境的引用.....	252
12.3 状态 Monad.....	253
12.3.1 状态 Monad 标签器.....	254
12.3.2 用状态 Monad 实现栈结构.....	255
12.3.3 状态 Monad、FunApp 单位半群和读取器 Monad 的关系.....	257
12.3.4 MonadState.....	258
12.3.5 基于栈的计算器.....	258
12.4 随机数的生成.....	270
本章小结.....	271
第 13 章 Monad 转换器.....	273
13.1 从 IdentityT Monad 转换器开始.....	273
13.2 Monad 转换器组合与复合 Monad 的区别.....	276
13.3 Monad 转换器的组合顺序.....	278
13.4 lift 与 liftIO.....	281
13.5 简易 Monad 编译器.....	282
13.6 语法分析器 Monad 组合子.....	286
13.6.1 简易语法分析器的实现.....	287
13.6.2 Parsec 库简介.....	291
13.6.3 上下文无关文法.....	296
13.6.4 基于语法分析器的计算器.....	300
本章小结.....	304
第 14 章 QuickCheck 简介.....	305
14.1 测试函数属性.....	305
14.2 测试数据生成器.....	308
本章小结.....	310
第 15 章 惰性求值简介.....	311
15.1 λ 演算简介.....	311

15.2	上Bottom.....	313
15.3	表达式形态和 thunk.....	314
15.3.1	WHNF、HNF 与 NF.....	314
15.3.2	thunk 与严格求值.....	315
15.4	求值策略.....	319
15.4.1	引值调用.....	319
15.4.2	按名调用.....	320
15.4.3	常序求值.....	320
15.5	惰性求值.....	321
15.6	严格模式匹配与惰性模式匹配.....	322
第 16 章 并行与并发编程.....		324
16.1	确定性的并行计算.....	325
16.2	轻量级线程.....	333
16.2.1	调度的不确定性.....	333
16.2.2	基本线程通信.....	334
16.2.3	信道.....	337
16.2.4	简易聊天服务器.....	337
16.3	软件事务内存.....	341
16.3.1	软件事务内存简介.....	341
16.3.2	软件事务内存的使用.....	343
16.3.3	哲学家就餐问题.....	347
16.3.4	圣诞老人问题.....	350
16.4	异步并发库简介.....	355
本章小结.....		357
参考文献		358
后记		359

第 1 章

Haskell 简介

本章首先介绍 Haskell 相关的历史，从 Lisp 诞生到多种函数式编程语言百花齐放，再到 Haskell 诞生的过程和现在发展的概况；接下来讲解安装 Haskell 编译器和编写 Haskell 程序所需要的软件，以及调试与测试函数的工具 GHCi；然后介绍 Haskell 中定义的两种源代码文件（一个是以 `lhs` 为扩展名，另外一个是 `hs` 为扩展名）有着怎样的不同；最后编写一个 Helloworld 程序当做学习 Haskell 旅程的第一步。

1.1 Haskell 的由来

要讲述 Haskell 的由来还要从函数式编程的诞生说起。函数式编程有着非常悠久的历史，比 C 语言还要久远。20 世纪 30 年代，美国数学家 Alonzo Church 引入了 λ 演算（Lambda Calculus），这是一个通过使用符号表达变量的绑定和替换来定义函数计算的系统，它是函数式编程语言的重要基石。也就是说，早在电子计算机还没有诞生的 20 世纪 30 年代，函数式编程语言就已经在孕育之中了。

1958 年，斯坦福大学的教授 John McCarthy 受卡内基梅隆大学开发的一个名为 IPL（Information Processing Language）语言的影响，开发了一个名为 Lisp 的函数式编程语言。虽然 IPL 并不是严格意义上的函数式编程语言，但是它已经有了些基本思想，如列表内包、高阶函数等，本书会在后面的章节中依次做介绍。这些特性深深地影响了 Lisp 的设计。在 Lisp 诞生之后，越来越多的人开始加入到函数式编程的阵营中来。同时，Lisp 的诞生也影响了很多编程语言的设计，时至今日，仍然有相当一部分人在使用 Lisp。

在 20 世纪 60 年代，牛津大学的 Peter Landin 和 Christopher Strachey 明确了 λ 演算对函数式编程具有极高的重要性，并于 1966 年开发出了一个名为 ISWIM（If you See What I Mean）的函数式编程语言，这是一个基于 λ 演算的纯函数式编程语言，一定程度上奠定了函数式编程语言设计的基础。

函数式编程发展到 20 世纪 70 年代早期的时候，爱丁堡大学的 Rod Burstall 和 John Darlington

使用函数式编程语言通过模式匹配来进行程序变换，并且明确地引入了列表内包的语法来更好地生成列表。几乎在同一时期，David Turner 教授在英国圣安德鲁斯大学，又开发了一个名为 SASL（St Andrews Static Language）的纯函数式编程语言，一定基础上奠定了圣安德鲁斯大学在函数式编程领域研究的地位。

1975 年，麻省理工学院的 Gerry Sussman 与 Guy Steele 开发了 Scheme，这是一个更接近 λ 演算的语言，广泛应用于实践编程与教学中。如今，Scheme 仍然被很多高校作为首要的课程来讲解。

不久，美国计算机科学家 John Backus 致力于函数式编程的研究，开发了一种名为 FP 的函数式编程语言。他引入了 BNF 符号系统以及对语言编译器系统的开发做出了突出的贡献，于 1977 年获得了图灵奖——他著名的图灵奖报告《编程是否能从冯诺依曼的体系风格中解放出来》（Can Programming be Liberated from the von Neumann Style?）^① 中提到的“解放”的方式，实际上指的就是函数式编程。

也几乎是在同一时期，剑桥大学的 Robin Milner 开发了 ML（Meta-Language），并发展出了多态类型系统——HM-System（Hindley-Milner type system），其中包括类型推断以及类型安全与异常处理机制。这个语言中的多态类型系统对函数式编程的发展意义重大，值得一说的是，Haskell 使用的正是这种类型系统。

Scheme 与 ML 都具有一定的顺序式语言的特性，但是这些语言改善了函数式编程的风格并且明确地引入了高阶函数的概念。20 世纪 70 年代末与 80 年代初，惰性求值这一概念被重新被发展。SASL 在 1976 年加入了惰性求值的功能，也出现了 Lazy ML，即 ML 的惰性版本。在程序的运行过程中，惰性求值可使计算机仅仅计算需要的数据。

在近几十年的时间里，各个大学的学者们通过对函数式编程的研究，涌现出了很多基于函数式编程的理论。当然，与此同时还出现了很多不同的函数式编程语言。1987 年，在美国俄勒冈州举行的函数式编程与计算机结构的会议（Functional Programming and Computer Architecture Conference, FPCA）上，与会者们就当时函数式编程语言种类过多、语法相似且大多数效率不高的现状进行了讨论。他们认为，这样下去的结果将会是越来越多的人研究和使用函数式编程语言，但是人们所使用的语言却得不到很好的统一。这种情形不利于函数式编程的研究、应用与发展。于是会议决定设计一个开源、免费、语法简单、运行稳定、效率高，可适用于函数式编程教学、研究，并且可编写应用软件的纯函数式编程语言来缓解函数式编程语言过多的混乱的局面，它的名字就是 Haskell。它的命名源于美国数学家 Haskell Brooks Curry，为纪念他在 λ 演算与组合逻辑（combinatory logic）方面作出的突出贡献。Haskell Curry 使得函数式编程语言的设计在理论上有了非常坚实的基础。由此可见，Haskell 是函数式编程发展了近 60 年的结晶，汇集了其他函数式编程语言的精华于一身，经过了 20 余年的发展，如今还在不断地壮大。

Haskell 仅仅意为一门计算机编程语言，语言标准的 1.0 版本于 1990 年发布，语法的主要设计者

^① 链接 http://www.thocp.net/biographies/papers/backus_turingaward_lecture.pdf。

之一是现任职于微软剑桥研究院的 Simon Peyton Jones^①教授。Haskell 这门语言的编译器有很多种，其中 GHC（Glasgow Haskell Compiler）是最常用的 Haskell 语言的编译器，名为哥拉斯哥是因为该编译器最初是基于 Kevin Hammond 教授在英国哥拉斯哥大学于 1989 年用 Lazy ML 所编写的原型。一年后，在 Simon Peyton Jones 的带领下，这个原型除分析器（parser）的部分全部被重写。第一个 GHC 的测试版于 1991 年 4 月 1 日发布，同时 GHC 也被作为英国学术研究项目之一并且开始接受英国政府的资金支持。除 GHC 外，Haskell 还有很多其他的编译器，比如由荷兰乌特勒支大学编写的 UHC（Utrecht Haskell Compiler），由 John Meacham 所编写的 JHC 等。随着发展，Haskell 语言的标准也开始发生变化，越来越多的特性与功能被逐渐加进来，两个里程碑的版本分别是 Haskell 98 和 Haskell 2010。本书的大部分代码是符合 Haskell 98 标准的，但有一部分用到了 Haskell 2010 中增加的内容。读者可以参阅 Haskell 98 report^② 和 Haskell 2010 Report^③ 来了解 Haskell 语法规则。

如果读者想要了解更多关于 Haskell 的历史，可以阅读由 Paul Hudak 和 John Hughes 等人写的《A History of Haskell: Being Lazy With Class》，文章的在线链接是 <http://research.microsoft.com/en-us/people/simonpj/papers/history-of-haskell/history.pdf>，还有（Hutton, 2007）的 1.4 节与（O’Sullivan et al., 2009）中的前言部分。

1.2 Haskell 编译器的安装以及编写环境

目前 Haskell 的主要编译器是 GHC，它可以将写好的程序编译后直接运行。但在教学中常常使用 Haskell 的两个解释器，它们是 Hugs 与 GHCi（Glasgow Haskell Compiler interpreter），这里使用的是 GHCi。GHCi 可以解析、调试 Haskell 程序而不必每一次都重新编译来测试代码，这在调试与测试代码时是一个非常大的优势。GHCi 是 GHC 的一部分，GHC 可以在 <http://www.haskell.org/ghc/> 下载到。这里推荐大家下载 Haskell Platform (HP)，它包含了所有开发所需要的工具，可以在 <http://www.haskell.org/platform/> 下载到。

Hugs 是一个遵循 Haskell 98 语言标准的解释器。由于 Hugs 不能将程序编译成可执行文件，也没有丰富的库函数，所以它很轻巧，适用于入门教学。Hugs 的下载地址是 <http://cvs.haskell.org/Hugs/pages/downloading.htm>。

WinGHCi 程序窗口如图 1-1 所示。Notepad++ 文本编辑器窗口如图 1-2 所示。

读者可以使用自己喜欢的任何编辑器，在 Windows 下，笔者喜欢使用 Notepad++，见图 1-2。这是一个非常轻巧的编辑器，可以在 <http://notepad-plus-plus.org/download> 下载到。Notepad++ 可以高亮显示 Haskell 中的关键字，看起来更加舒服，当选取了 Haskell 模式时还会默认保存为.hs 文件。Windows 下的记事本以及 Linux 或 Mac OS 下的 sublime、emacs、vim、gedit 都是非常好用的文本编辑器，并且很多也提供 Haskell 插件与设置。但是需要注意的是，由于 Haskell 代码

^① Jones 教授的主页：<http://research.microsoft.com/en-us/people/simonpj/>。

^② <http://www.haskell.org/onlineReport/>

^③ <http://www.haskell.org/onlineReport/haskell2010/>

的缩进与对齐有时非常重要，有的文本编辑器会让人将 Tab 与空格等字符搞混而引发一些错误。当然，正如其他语言一样，Haskell 也有一些非常好的集成开发环境，如用 Haskell 编写的 Haskell 集成开发环境 Leksah（Haskell 的逆写），可以在 <http://leksah.org/> 下载。笔者更喜欢用 EclipseFP，一个基于 Eclipse 的开源 Haskell 插件，具体安装可以详见 <http://eclipsefp.github.com/>。



图 1-1 WinGHCi 程序窗口



图 1-2 Notepad++文本编辑器窗口

1.3 GHCi 的使用

GHCi 是一个对函数进行测试与调试的工具，可以导入 Haskell 源代码文件，然后调用其中的函数、查看函数的信息等。本节先学习如何使用 GHCi 中的命令来对文件和库进行导入等。

再来了解如何在 GHCi 中调用函数。

启动 GHCi 后可以看到 GHCi 的版本，还有导入的库等，可以不用管它们，最后一行会有一个 Prelude>提示符，其中 Prelude 指的是 GHCi 在运行时一个默认的初始环境。它是一个定义了很多类型与函数的库。启动 GHCi 后，用户可以不做任何设置而直接使用其中定义的内容。下面来看一下 GHCi 中的一些命令。

1.3.1 GHCi 中的命令

下面介绍一些常用的 GHCi 命令，学习如何导入代码文件和库模块，以及如何改变 GHCi 的当前路径等。

- :load：简写为:l，用来导入当前路径或者指定路径下的文件，但在 Windows 下要注意使用转义的反斜杠。比如，导入作者桌面上 HelloWrold 文件夹下的 HelloWorld.hs，WinGHCi 的用户可以直接使用打开按钮来打开程序文件。
Prelude>:l "C:\\\\Users\\\\User\\\\Desktop\\\\HelloWorld\\\\HelloWorld.hs"
- :reload：简写为:r，用来重新导入当前的源代码文件。通常，在保存了源文件后，GHCi 不会自动重新导入修改后的文件，用户可以很方便地使用:r 来重新导入。WinGHCi 的用户可以使用刷新按钮来重新导入程序文件。
- :cd：改变当前 GHCi 的路径。这样做就不用每一次都输入绝对路径来导入文件了。例如：
Prelude>:cd C:\\Users\\User\\Desktop
- :edit：用默认的文本编辑器编辑当前导入的文件。如果使用 GHCi，它会读取系统环境变量中的 EDITOR，启动相应的编辑器。如果读者使用的是 Hugs，则需要设置 HUGSFLAG 环境变量来使得 Hugs 可以启动对应的文本编辑器。更多信息可以参阅 Hugs 用户手册 3.1 节，可以浏览 <http://cvs.haskell.org/Hugs/pages/users-guide/>。
- :module：导入一个库，简写为:m。使用:m +<module> 与:m - <module> 来增加与移除不同的模块。在后面会具体介绍如何使用这个命令。
- :quit：退出 GHCi。
- ?: 可以让 GHCi 输出帮助信息。

当然，GHCi 的命令还有很多，本书将在后面的章节再做介绍。这里约定：若没有特别说明，则 GHCi 指的就是 WinGHCi，而不是命令行下的 GHCi。

1.3.2 在 GHCi 中调用函数

很多数值比如整数、小数还有一些四则运算的函数都已经在上节中提过的 Prelude 初始环境中定义好了，所以可以直接使用。由于在 Prelude 中定义了各种数学运算符号，因此 GHCi 可以当做一个计算器来使用。比如：

```
>4+6*7/3
18.0
```

此外还有自然对数函数、三角函数及圆周率 π 等。

```
> log 2.71828
0.999999327347282

> sin (pi/3) / cos (pi/3)
1.7320508075688767

> tan (pi/3)
1.7320508075688767
```

除数字的类型以外，Prelude 中还定义了布尔类型，这种类型只有 `True` 与 `False` 两个值，表示真与假。Prelude 中也定义了基于布尔值的运算符，读者可以直接用 `&&` 运算符号对布尔值做逻辑与运算。例如：

```
> True && False
False
```

除了逻辑与运算 `&&` 外，Prelude 中还提供了逻辑或运算符 `||`，用户可以在 GHCi 中测试这个函数。

Prelude 中还提供非常实用的容器——列表。有了它就可以很灵活地对值进行存储和使用相关的函数。`[1..4]` 表示遍历整数 1~4，即 `[1, 2, 3, 4]`。

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

`sum` 是一个可以对列表中的数值进行求和的函数。也就是说，给定一个列表 `sum`，会求得该列表中所有元素的和。比如：

```
> sum [1..4]
10
```

Prelude 中的 `product` 函数可以求得一个列表的所有元素的乘积，读者可以在 GHCi 中计算 `[1..4]` 的乘积。

如果想引用之前调用的函数所计算的结果，可以使用 `it`。比如，计算了 1~4 之间的整数之和后想再加 100 可以写为：

```
> it + 100
110
```

因为 `it` 在 GHCi 中可以指代前一次函数计算的结果，所以在定义函数还有测试时不要使用 `it` 作为函数或者变量的名称。

最后，约定如下：如果书中只用 `>` 符号，然后调用函数或者输入 GHCi 命令，则表示在 GHCi

的提示符中的操作，而 C:>则是系统命令行的提示符。

1.4 .hs 和.lhs 文件、注释与库函数

GHCi 和 Hugs 可以解析扩展名为.hs 和.lhs 的文件。两者所写程序在语法上完全相同，它们的差别是.lhs (literate Haskell script) 文件是为了能让 Haskell 的代码生成文档。有效程序代码可以用“大于号 (>)”和空格开头。比如：

```
> add :: Int -> Int -> Int
> add x y = x + y
```

不以大于号和空格开头的内容将被视作注释处理，且注释与代码间必须有一行以上的间隔。还有一些.lhs 代码文件中的代码以\begin{code}以\end{code}结尾——如果.lhs 文件遵守一定的格式，就可以使用 lhs2tex 生成非常精美的文档以供人们阅读。如何使用读者可以参阅 <http://www.andres-loeh.de/lhs2tex/>。

使用.lhs 文件书写代码的优点很明显。函数式编程的代码往往蕴含了编写者很大的思考量，需要用大段的注释进行解释说明，而代码又不是特别长，这个时候用.lhs 最好不过了——哪部分是代码、哪部分是注释一目了然，还能生成.pdf 文件方便阅读与传播。

但是，当不需要多行注释还有生成文档的时候就可以用扩展名为.hs 的文件。.hs 文件里全局的函数要起头写，不可以有其他字符；单行的注释用两个减号 (--) 开头，多行注释用“{-”开头，以“-}”结尾。全书对编写的函数有非常多的阐述与解释，并不包括在源代码中，所以用.hs 文件就可以了。有时需要在文件头处对 GHC 与 GHCi 声明一些编译器参数，此时需要在文件首处定义，并且以“{-#”开头，用“#-}”结尾。在后面的章节中，再具体学习使用编译器参数。

在启动 GHCi 的时候，Prelude 中的一些预加载函数已经被导入了。Prelude 里有很多常用的函数以及数据。安装 hugs 的读者可以到 C:\Program Files\WinHugs\packages\ hugsbase\ Hugs 下找到 Prelude.hs 库函数文件，有兴趣的读者可以打开 prelude 或其他文件浏览代码。使用 GHCi 的读者可以在开始菜单的程序中找到 Haskell 的网页面档，打开 Library 链接，找到 Prelude，再点击 source 查看。读者可以通过查看这些源文件来感受 Haskell 程序的风格。

Haskell 的函数库是非常强大的。在发展的多年过程中，有很多库可供用户直接使用，也有像 Java 一样的 API 可以查阅。WinGHCi 的用户一样可以在开始菜单程序中网页面档中找到，也可以在 <http://www.haskell.org/ghc/docs/latest/html/libraries> 在线浏览。第 4 章将会对几个库函数使用做简要的介绍。

1.5 第一个 Haskell 程序 HelloWorld!

HelloWorld 程序虽然不能完全展示 Haskell 编程的风格与优势，但学习计算机编程都是从

这里开始的。现在写一个 HelloWorld 程序，当作开始学习 Haskell 的第一步吧！

文本编辑器中输入：

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

保存并命名为 Helloworld.hs。

这里可以看到，Haskell 和 C、Java 一样，都以一个名叫 main 的函数作为程序的开始运行的入口。保存代码文件，打开命令行窗口，即“开始”→“运行”，输入 cmd，按 Enter 键。在命令行中把目录改变到 HelloWorld.hs 所在的位置，输入 ghc HelloWorld.hs。这时，得到了一个可执行的 HelloWorld.exe 文件，运行该文件就可以了。

如果读者只想在命令行中运行程序而不想进行编译，可以使用 runghc 命令：

```
C:\> runghc Helloworld.hs  
Hello,World!
```

从 GHCi 中打开这个文件，在 GHCi 命令行里输入 main 也可以运行。除这个例子外，第 11 章之前的部分都在讨论纯函数，所有的函数均需要在 GHCi 中运行。因为 Haskell 处理输入/输出的特殊性，所以只有到第 11 章讨论输入/输出时读者才能写一些与操作系统互动的程序。为方便广大读者测试代码，本书的部分代码可以在 GitHub 下载到，同时还有对书中错误的更正，地址是 https://github.com/HaskellZhangSong/Introduction_to_Haskell。

本章小结

学习本章后，相信读者对 Haskell 应该有了大致的了解。GHCi 是一个常用的测试代码的工具，希望读者可以花更多的时间来熟悉。细心的读者可能会发现 Haskell 与 C 语言编译后可执行文件的大小有很大差异。其实，Haskell 使用内存空间和硬盘空间的效率是有些低的，这也是早期函数式编程没有比 C 一类的语言更流行的原因之一。但是，如今计算机硬件已经发展到内存和硬盘不会像以前那样限制函数式编程语言能力了，在时间和空间的效率上也可以手动或自动调试优化。因此，相信在不久的未来，函数式编程会以它精炼、缜密的代码与安全、可靠的运行渐渐受到更多人的青睐。

第2章

类型系统和函数

本章来简单了解一下 Haskell 中的类型系统与函数。用户操作的数值在计算机看来是没有任何意义的，它们只是一串由 0 与 1 组成的二进制数。而对于高级程序语言来说，需要将这些二进制数分成不同的类别以便处理，在计算机语言中就称为类型（type）。一个类型下可能会有一些数据：可以有 1 个值，也可以有多个值；可以是无限的，也可以是有限的；在 Haskell 中，甚至还可以为空，即没有值。整数类型下有 -7、11、98 等，布尔类型下面只有 True 和 False 两个值。Haskell 不但对这些二进制的值进行了类型的分类还对不同类型进行了整理，将有着共同属性的类型归为一种特定的类型类（typeclass），意为类型的类别。这里以布尔类型与整数类型为例，它们就有很多共同的属性。首先，它们都可以比较相等，如判定 1 与 2 是否相等，还可以判定 true 与 True 是否相等。而在 Haskell 中，可以判定一个类型的两个值是否可以相等就需要用到相等类型类 Eq。又如整数与布尔值都是可以打印在命令行上的，这就是因为 Prelude 预加载库中定义了可显示类型类 Show。也就是说，整数类型与布尔类型都是归于相等类型类 Eq 还有可显示类型类 Show 的。

对于 Haskell 的类型系统有了初步的了解以后，将会学习如何用 Haskell 中的各种表达式来定义函数。但在真正用表达式定义函数之前，需要了解函数与类型系统之间的关系，它们实际上并不是分离的，而是骨肉相连的。其中，函数的类型是“骨”，而函数的定义是“肉”，了解它们是如何相辅相成的，在今后的学习与编写 Haskell 程序中至关重要。

2.1 Haskell 的类型与数据

2.1.1 Haskell 常用数据类型

Haskell 是一种有着强类型（strong type）的语言，每一个数据和每一个函数都要有非常精

确的类型，这使得函数的定义更为严格。在 GHCi 的提示符里，可以输入:type（可简写为:t）命令来查看一个数据的类型或者一个函数的类型（读者将体会到，在 Haskell 中，数据与函数是没有本质区别的，函数只是数据的一种）。下面介绍一下 Haskell 中常用的类型。

1. 布尔类型：Bool

布尔类型是一个只有 True 与 False 两个值的数据类型。细心的读者可能已经发现，数据类型的首字母都是大写的。这里说明一下，Haskell 中约定类型的名字与该类型的数据的首字母要大写（当然，数字的类型除外）。布尔值的运算符号和其他语言相似，`&&` 表示“逻辑与”运算，`||` 表示“逻辑或”运算，`not` 表示“逻辑非”运算。例如，在 GHCi 里输入：

```
>True||False
True
```

```
>not True
False
```

可以使用:t 来查看 True 的类型：

```
>:t True
True :: Bool
```

GHCi 给了正确的答案，意思是说 True 有着 Bool 类型。

2. 字符型：Char

键盘上的字符都是 char 类型了，这一类型与其他语言是一致的。例如，在 GHCi 里输入：

```
>'a'
'a'
```

```
>:t 'a'
'a' :: Char
```

意思是说 'a' 是一个具有 Char 类型的数据。除此之外，还可以使用反斜杠与 ASCII 码的值来表示一个字符，比如从 97~122 为英文小写字母 a~z，从 65~90 为英文字母 A~Z。

```
> '\100'
'd'
```

有一些 ASCII 码用于控制输入和中断等，比如换行符\n、Tab \t，还有退出键\ESC。

常见的转义字符如表 2-1 所示。

比如，输出"\\"，那么只会得到一个\，因为第一个反斜杠表示为将后面反斜杠的功能去掉。

表 2-1 常用转义字符

\b	退格符
\n	换行符
\t	Tab
\&	空字符
\'	单引号
\\"	反斜杠

```
> putStrLn "\\"
```

再比如，\&为空字符：

```
> putStrLn "abc\&def"
```

3. 有符号整数：Int

Int 几乎是所有的计算机语言里都有的数据类型。它的范围与操作系统与 GHC 位数有关。这里使用的是 32 位的操作系统与 GHC，故整数的范围是 $-2^{31} \sim 2^{31}-1$ 。如果在 GHCI 里输入：

```
> 2^32::Int
0
```

结果是 0，相当于在 $2^{31}-1$ 后转回到了 -2^{31} ，最后又回到了 0。对于 64 位 GHC 来说 Int 的范围则是 $-2^{63} \sim 2^{63}-1$ ，所以 $2^{64}::Int$ ，结果是 0。 $::Int$ 的意思是让 Haskell 把 2^{64} 作为整数处理。为什么一定要声明为整数呢？因为 Haskell 里还有另外一个整数类型——任意精度整数，如果不指明类型，Haskell 会将 2^{32} 默认为任意精度整数处理。

4. 无符号整数：Word

Word 类型是无符号的整数，它的范围也是系统相关的。在 32 位的系统中，它的范围是 $0 \sim 2^{32}-1$ 而 64 位系统则为 $0 \sim 2^{64}-1$ 。Haskell 中的 Word 相当于 C 或者 Java 里的 unsigned int 类型。使用 Word 类型需要导入 Data.Word 库，在 GHCI 中可以使用 :module (简写为 :m) 来控制库的导入。

```
>:m +Data.Word
>-l::Word
4294967295
```

这样可以看到，虽然需要输入 -1，但 -1 对于无符号的 Word 类型来说是 4294967295，也就是 $2^{32}-1$ 。

:m -Data.Word 可以移除不需要的库。

5. 任意精度整数：Integer

与 Int 不同，Integer 类型可以表示任意大小的整数，限制它的大小范围的唯一因素就是

计算机的硬件。可以使用一个简单的 product 函数来试验一下，在 GHCI 里输入：

```
> product [1..10000]
```

结果将会是一个很大很大的数字，它是 1~10000 所有整数的乘积，也就是 10000 的阶乘。再比如，输入：

```
>2^32::Integer
```

GHCI 会给出正确的结果，即便用户不指定类型为 Integer，GHCI 也会默认 2^{32} 为此类型。其他的编程语言像 Java 需要用一些现有的类库来实现，而 Haskell 却很好地解决了这个问题。但付出的代价就是处理 Integer 的效率不如 Int 高。可是在处理一些数学问题，如 RSA 加密、大数字的运算等时，这会非常方便。

除了 Int、Word 与 Integer，在 Data.Int 库中，Haskell 还支持 Int8、Int16、Int32 和 Int64，即 8 位、16 位、32 位和 64 位有符号整数。Haskell 的 Data.Word 中还提供对应的 Word8、Word16、Word32 和 Word64 无符号整数，但只有在一些特殊的情况下才会用到。

6. 小数与有理数类型：Float、Double、Rational

Haskell 中的单精度浮点小数、双精度浮点小数与其他语言没有很大的区别。比如：

```
> pi :: Float  
3.1415927  
  
>pi :: Double  
3.141592653589793
```

除此之外，Haskell 还有有理数类型 Rational，即用两个任意精度的整数来表示一个小数，这在做高精度数学运算时有很多好处。

```
> 4.1332::Rational  
10333 % 2500
```

%相当于分数线，2500 为分母，10333 为分子，这样，一个小数就被转换为了分数。由于分子与分母可以为任意的整数，因此可以用这个类型来处理任意精度的有理数。

7. 字符串类型：String

在 Haskell 里，String 是通过其他类型定义的，这一点将在 2.1.3 节中讨论。String 的类型定义为 [Char]，也就是说 String 是一个 Char 的列表。这样的定义也是十分合理的。在 GHCI 里输入 ['H', 'e', 'l', 'l', 'o']，会得到 "Hello"。使用引号更为方便。

```
>['H', 'e', 'l', 'l', 'o']  
"Hello"
```

8. 元组类型

有时，需要一个数对来表示一些数据，比如，整数坐标的类型可以写做 (Int, Int)，这就是一个元组 (tuple)。元组中的两个整数称为该元组的元件 (component)。当函数需要返回两个

不同类型的结果时，就可以使用二元元组来存储这个两结果。不但有二元元组，还可有多元元组，多元元组中可以有很多个元件，如(5, True, "Hello"...)。但是，一旦确定了元组中元件个数，就不可以像列表那样伸缩了。同时，每一个元件的类型也不能改变。

对于二元的元组有两个重要的函数，它们是 `fst` 与 `snd`。它们会分别返回元组里的第一个元件和返回元组的第二个元件。例如，输入 `fst (5, True)`，GHCi 会返回 5。

```
> fst (5, True)
5
```

元组在 Haskell 里是非常实用的，已是一个非常重要的数据载体。比如，想记录一个学生的名字和学号，就可以用一个以 `(String, Int)` 为类型的元组来存储的，它可以是 ("Li Ming", 6509843)。这样的元组很像是数据库里的一个条目，如一个数据库记录着用户名、性别、年龄、电话、地址等。

```
> ("Ling Ming", "Male", 20, "123456")
("Ling Ming", "Male", 20, "123456")
```

9. 列表类型

这里来深入讨论一下列表类型的使用。列表（list）在 Haskell 里是一个非常重要的类型，很多函数都会用到。它是一个容器类型，也就是说，可以将各种值放在里边。放入的类型可以是任意的某一类型的值、布尔类型、整数类型，也可以存储函数类型，甚至可以是列表类型本身，但表中的数据类型是唯一的并且不可变的，比如，它不可以既存放字符又存放布尔值。列表在 Haskell 里是这样定义的：

- (1) 它可以是一个空的列表，即 []；
- (2) 它可以是一个元素与另一个列表的结合，即 `x:xs`。`x` 是一个元素，`xs` 是一个列表。`xs` 可以是一个有元素的列表，亦可以是一个空的列表。元素与后边的列表用冒号 (:) 连接。例如：

```
> 1:[]
[1]

> 1:[2]
[1,2]
```

这样就得到了整数类型的列表（这样说不是特别的准确，实质上这是一个 `Num a => [a]` 类型的列表，`Num` 是一个类型类，我们在 2.2 节中介绍类型类）。再比如，输入 `[True, False]`，就得到一个布尔类型的列表。

如果输入 `[]:[]`，就会返回 `[[]]`，这里 `[]:[]` 的意思是说，这是一个列表中的列表，即列表里边有一个列表并且里边的列表是空列表：

```
> []:[]
 [[]]
```

想一下，如果输入`[] : [] : []`，会返回什么作为结果呢？提示：`(::)`是右结合的运算，这里输入的其实是`[] : ([] : [])`。

有了列表中的列表这种结构，如`[[1, 2, 3], [3, 4, 5, 1, 4], [1, 2, 4, 4]] :: [[Int]]`，就有一个类似于 C 或者 Java 中二维数组一样的结构。

与 C 与 Java 中的数组相比 Haskell 中的列表比起数组更为灵活，因为可以伸缩，并且元素个数可以是有限的，还可以是无限的。另外，用户不需要指定它的长度。

读者已经知道，列表相当于一个容器，可以存放任何类型的值。在 Haskell 中函数与值的区别并不是那么明显的，函数不过是需要参数输入的值，那么列表一样也是可以存储函数的。在 C 语言中，可以存储函数的指针变量，但这种用法在 C 语言中并不常见，而在 Java 中却无法实现。Haskell 的列表可以非常自由地存储这些函数，在解决一些问题时非常方便，这也正是 Haskell 具有极强的表达能力的原因之一。比如，一个存储了加、减、乘函数的列表可以定义如下：

```
> :t [(+), (-), (*)]
[(+), (-), (*)] :: Num a => [a -> a -> a]
```

各种的类型实际上是可以互相组合使用的，比如，`([Int], [Bool])` 或 `[(Int, Bool)]`，或者再复杂一些的`[(Int -> Bool, [String])]`。如果需要，也可以存储为由不同类型组成的更为复杂的类型。

2.1.2 函数类型

函数可以理解为从参数到结果的一个映射，比如 `t1 -> t2`。每一个函数都符合这样一个定义，只不过是 `t1` 或 `t2` 可能代表着更复杂的类型，可能是一些其他特殊的类型，也可能是一些函数类型，`t1` 或 `t2` 若为函数，那么 `t1 -> t2` 函数可以称为高阶函数（high order function）。相关内容将在第 7 章中讲解。但无论是一个什么样的类型，它们都符合函数类型的定义。用户可以来定义一个函数，如整数加法：

```
add :: (Int, Int) -> Int
add (x, y) = x + y
```

很明显，加法是需要两个参数的函数。把这两个参数以一个二元元组的形式作为函数的输入，这样会得到一个整数结果。定义中的第 1 行就是这个函数的类型，`(Int, Int) -> Int` 表示给定一个整数元组返回一个整数。第 2 行是函数定义，即给一个二元元组，其中的值是 `x` 与 `y`，那么返回 `x` 与 `y` 的和。Haskell 中的函数定义格式基本如此，在 2.4 节中将着重介绍如何定义函数，这里仅仅是将函数当做函数类型的值来讨论。Haskell 中的函数有着不同的特点，本节将根据不同的特点来做一下简要的介绍。

1. 柯里化函数

柯里化函数（curried function）也是一种函数类型。当函数有多个参数时，参数可以一个一个地依次输入，如果参数不足，将返回一个函数作为结果，这样的函数就是柯里化的函数。比如，之前定义过的整数加法，可以在 GHCi 中定义如下：

```
let add' x y = x + y :: Int
```

这样的概念一开始可能有一些和直觉相背，但是在数学中常常使用这样的函数。比如，加号可以看做是一个需要两个数作为参数的二元函数，之前的 add 函数是这样定义的： $f(x,y)=x+y$ 。现在，如果已经给定第一个参数，那么希望这个函数就变成一个一元函数 $f(y)=4+y$ 。使用元组无法做到这一点，add' 却可以满足本例的需要，可以看一下 add' 4 的类型：

```
> :t add' 4
add' :: Int -> Int
```

这个类型意为给一个整数会返回一个整数，也就是函数 $f(x)=4+x$ 的类型。调用柯里化的函数时，如果参数不足，得到的是一个还需要参数的函数。这种类似的思想将在后边的小节中做更充分的解释。到这里，读者应该可以体会到，将参数依次传入到柯里化函数 add' 与用元组将参数一次性传入的非柯里化 (uncurried) 函数 add，实质上是等价的并且是可以相互转换的，即当一个函数需要多个参数得到结果时，使用一个元组将参数全部一次性给出，等价于一个一个连续地给出。

如果从非柯里化的 add 函数想得到对应柯里化的函数，即 $\text{add}(x,y)$ 到 $\text{add}' x y$ ，这个过程叫做柯里化 (curry)。

如果有柯里化的函数，想得到对应的非柯里化函数，即 $\text{add}' x y$ 到 $\text{add}(x,y)$ ，这个过程叫做非柯里化 (uncurry)。Prelude 如定义 3curry 与 uncurry 两个函数，例如：

```
> :t curry
curry :: ((a, b) -> c) -> a -> b ->c

> :t (curry add) 4
curry add 4 :: Int -> Int

> :t uncurry
uncurry :: (a -> b ->c) -> (a, b) ->c

> (uncurry add') (1,2)
3
```

柯里化与非柯里化的内容将在后边的章节中做更多的探讨。

2. 多态函数

很多函数的参数不一定要有明确具体的类型的。比如，元组类型中使用了 fst 函数，也介绍了 snd 函数，它们分别会返回元组中第一个元件和第二个元件。

```
> fst (1, True)
1

> fst ([1,2,3], False)
[1,2,3]
```

这时，第一个 fst 调用后返回的是一个整数类型，而第二个 fst 返回的则是整数的列表类型。这样，fst 函数包括了所有可能的类型，如 $(\text{Int}, \text{Int}) \rightarrow \text{Int}$ 、 $(\text{Int}, \text{Bool}) \rightarrow \text{Int}$ 、 $([\text{Int}], \text{Char}) \rightarrow [\text{Int}]$ 等，本书无法将它们一一地列举出来。像 fst 这样可以应用在多种类

型上的函数就称为多态函数 (polymorphic function)。它的类型是 $(a, b) \rightarrow a$ ，意为输入一个由任意的 a 类型与 b 类型组成的元组，会返回第一个为结果。

在 Haskell 中，这样的函数有很多。比如，`length` 是一个得到列表长度的函数，但对于列表里存的是什么类型，这个函数可以完全忽略。比如，在 GHCi 里输入：

```
> length [1,2,3,4]
4
> length [True,False,False]
3
```

这样，在函数的类型中，以一个小开头的名字来代替这些任意的类型，如 a 、 b 、 c 、 ok 、 key 等，它们是类型变量 (type variable)。例如，`length` 类型可以写为：

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

其中， a 只是一个名字，它可以改为任何的类型变量名。多态函数在 Haskell 十分常见，这里再给出两个例子。第一个例子是 `head`，它可以取得列表的首元素，若列表为空则报错，它的类型是 `head :: [a] -> a`。

```
> head [1,2,3]
1
```

第二个例子是将两个列表中的元素一一对应地合在一起，返回一个二元元组的列表的函数 `zip`，读者可以试一下，当一个列表的元素比另一个多时会怎样，如 `zip :: [a] -> [b] -> [(a,b)]`。

```
> zip [1,2,3] [4,5,6]
[(1,4),(2,5),(3,6)]
```

3. 重载类型函数

在前边的内容中，数字 5 一直是被当成整数。但是，它还可以是一个任意精度整数，或是一个小数。这样一来，类型上可能会有一些不协调，因为 5 是一个有着很多类型的值，Haskell 中用类型类 (typeclass) 这一概念来对这样的一些类型做了细致的分类。例如，在 GHCi 里输入：

```
:type 5
5 :: Num a => a
```

\Rightarrow 是类型类的限定符号，意思是说，5 有一个任意的 a 类型，但这个类型必须限定在一个名为 `Num` 的类型类中，即类型 a 必须是 `Num` 类型类的一个类型实例。这里还未介绍类型类，可以将 a 类型理解为一个数字，这个数字可以是整数，也可以是小数或其他数字类型。只有 a 符合相应的条件才能使用相应的数学计算函数，如加法。这样，通过对类型做一些限制，使得函数计算得以协调。下节简单讨论一下不同的类型类，在第 9 章再做进一步探讨。下面来看一下绝对值函数 `abs` 的类型：

```
:t abs
abs :: Num a => a -> a
```

GHCi 的意思就是说，`abs` 这个函数需要一个实现了 `Num` 类型类的类型 a 。由于 `abs` 不论是

对小数、整数还是其他类字类型都可以进行绝对值计算，但布尔值、字符串则不可以，因此 Haskell 引入了类型类的概念限定参数的类型，这样的函数就是重载函数。除 Num 类型类以外，还有 Ord 和 Show 类型类，下节中会一一介绍它们。

很多语言可以自由地支持函数的重载，但在 Haskell 中，函数或运算符的定义是唯一的，不可以出现第二种不同的定义，也就是说，对于每个函数，它的型只有一种。而在 Java 中，不但参数的类型可以不同，就连参数的个数也可以不同。Java 中可以这样做是因为在调用函数时都必须给出全部的参数。如果读者回顾之前的内容就会知道，Java 中的函数是非柯里化的函数。而在 Haskell 中，可以给出部分参数，而以函数作为返回的结果，这使得 Haskell 不能像 Java 一样对函数直接进行重载，但是借助类型类我们也可以做类似的事情。例如，在 Java 中定义：

```
static int function (int para1, boolean para2) {...}
static int function (int para1, char para2){...}
```

当使用这些方法的时候，要输入全部的参数。但是，在 Haskell 里定义如下：

```
function :: Int -> Bool -> Int
function para1 para2 = ..
function :: Int -> Char-> Int
function para1 para2 = ..
```

如果在 Haskell 中也可以这么做的话，那么 `function 5` 的类型就有两种结果了：一个是 `Bool->Int` 函数，另一个是 `Char->Int` 函数。这样，函数的类型就有了不确定性。

Java 重载是通过对有不同参数的函数用不同定义实现的，但是它们的名字完全相同。在调用函数时，Java 会根据参数的类型与个数匹配到不同的函数，比如 `print` 函数打印 `int`、`boolean`，其实也是用的重载。而 Haskell 中是使不同的类型实现 `Show` 类型类来做到的。对于所有的实现为 `Show` 类型类实例的类型都可以使用 `show` 函数。

2.1.3 类型的别名

在解决实际问题中，一个数据的类型可以由多个其他的类型组成。这时，需要将这些类型统一命名成其他的名字，这个就是这些类型的别名（type synonym）。比如，一个像素点其实是一组 RGB 的值，即 `(Int, Int, Int)`（其实这里用 `Word8` 更为恰当）。如果用户常常要用到这个类型，一直输入 `(Int, Int, Int)` 就显得有些麻烦。同样，对于研究函数的类型也很不方便。若一直用这些复杂的类型来写函数，这个类型的意义非常的不直观，不利于解决实际问题。在 Haskell 中，可以用 `type` 关键字将这些复杂的类型替换成为其他简单的名字，在定义时类型的名字要以大写字母开头。例如：

```
type RGB = (Int, Int, Int)
```

这样，Haskell 就会将 `RGB` 类型自动替换 `(Int, Int, Int)`。一幅图实质上是一个 `RGB` 的二维数组，这里可以用列表的列表来表示，即：

```
type Picture = [[RGB]]
```

再比如，管理一个图书馆的数据库，一本书在数据库里就是一个类型的条目：(Int, String, String, Int, String)。

看了上面这个类型后，读者可能还不知道哪部分都表示什么。但是，如果用 type 做一些替换就会好很多：

```
type ID      = Int
type BookName = String
type Author   = String
type ISBN     = Int
type Publisher = String
```

这样，对于一本书的条目可以定义为 (ID, BookName, Author, ISBN, Publisher)。如此一来，定义的类型就有意义多了并且可读性也更强了。然后，可以将这个类型再替换成 Book：

```
type Book = (ID, BookName, Author, ISBN, Publisher)
```

type 关键字只能做一些替换。在定义类型的别名时，它只是将已经有的数据类型组合成一个新的数据类型，并不能构造出新的数据类型。如何定义数据类型将于第 8 章讨论。

2.1.4 类型的重要性

为什么 Haskell 一定要强制用户使用类型？为什么不支持自由转型？下面讨论一下类型在编程中的重要性。看下面一段简单的程序：

```
while (x != 0) {
    x := x - 1;
}
```

这段程序所做的事情就是如果 x 不等于 0，那么 x 就减少 1。这段小程序会不会停呢？有人说是否停止取决于 x 的值，如果 x 是大于 0 的整数，那么最终 x 会减小到 0；如果是小于 0 的整数，这段程序不会停止，因为 x 会越来越小。其实，这段程序会不会停与 x 的类型有着更多的关系，而不是 x 的值。如果 x 为小数，并且小数部分不为 0，这段程序就不会停止。其实，即便小数等于 0，是否会停止也与!=运算符如何实现还有与不等号后面的 0 的类型有关。另外如果 x 的类型固定精度的整数，如 Int、Word，那么这个程序是会停的，因为即便 x 为负值，只要 x 小于 -2^{31} 时会再减 1 就会变为正数，最后程序会停止。但是，如果 x 为任意精度类型的整数那么情形就不一样了，所以类型的下面往往隐藏很多信息。

除了上边的例子，在很多的函数中类型都扮演有十分重要的角色。很多语言中支持运算符重载比如 1 + False，还有 1 + "Hello" 等，如果不知道这里+的类型，很难揣测+的意思。而 Haskell 中函数与数据的类型是十分明确的，每个函数都有类型签名，并且编译器会在编译前做类型检查，所以不会出现程序员还要猜测运算符意思的情况。

类型在程序语言中非常重要。在 Benjamin 的《Types and Programming》一书中简单阐述了类型系统的三个优点。

(1) 错误检查。函数式编程的本质上是给函数输入参数，由计算机来计算并输出结果。每个函数都有特定的输入与输出类型，这样类型系统就是对程序的一种限定。但是，为什么要做出这种限定？因为人是很容易犯错误的，虽然类型检查可以辅助用户检查程序的类型是否正确，但可以通过类型检查并不能说明函数完全没有问题。但是，如果类型不正确，则说明函数一定有问题。事实上，类型系统可以帮助用户检查出大部分的错误。如果读者有使用过汇编或者 Linux bash 脚本可能会体会到，汇编语言与 bash 中变量是无类型的（相比较 C 与 Java 而言，初学者很难写出没有问题的脚本），同时，汇编语言与 bash 在运行中出现的错误给出的错误信息也非常难懂（笔者也觉得 C 与 C++ 的错误信息多数情况下都非常的难懂）。而有了强大的类型系统，用户就可以少犯错误，并且即便程序有错误，类型系统可以辅助用户尽快地找到错误的位置。

(2) 对于程序的抽象。在对程序的抽象能力方面，Haskell 有着独到的优势，尤其是在大型的系统中，类型系统可以起到十分关键的作用。Haskell 中引入的 Monad，将一些不同的计算分成了不同的种类，这样在编写程序时，它们的代码不会互相干扰。将各种不同的计算分别独立地抽象出来，最终引导用户得到一个更好的设计，在后面的编程中会慢慢体会到这一点。

(3) 文档作用。类型签名不但可以辅助定义函数时，同时还会增加程序代码的可读性，作为补充文档便于软件维护，这在如今的软件开发过程中是至关重要的。

另外，类型还可以在团队开发时作为程序员讨论的媒介，现实开发中可以把问题分割成多种类型不同的函数，再分配给不同的程序员编写，此时函数的功能就类似于其他语言中使用的接口，这样所有人定义的函数最终会被很容易地组合起来，完成所需要的工作。

2.2 Haskell 中的类型类

Haskell 给很多“类型”分成了“类型类”，归为一类的类型有着共同的属性，不同类型所归的类就称为类型类。类型有着某种属性，意为该类型可以实现特定的函数，类型通过实现了类型类中声明的函数来成为其中的一份子，具体如何实现将在第 9 章中深入讨论。布尔类型与整数类型都实现了 Eq 类型类，即各给定两个布尔类型的数据和整数类型的数据都可以分别比较它们是否相等。例如：

```
> 5 == 6
False

> True == True
True
```

这些类型类隐含在已经介绍过的几个类型里。比如相等类型类 Eq、数字类型类 Num、可显示类型类 Show 和有序类型类 Ord。例如：

```
> :t 5
5 :: Num a => a
```

其中，Num 就为数字类型类：

```
> :t (==)
(==) :: Eq a => a -> a -> a
```

类型结果中的 `Eq` 所指的就是 `Eq` 类型类。又比如，比较大小用到的关系运算符——小于号的类型是：

```
> :t (<)
(<) :: Ord a => a -> a -> Bool
```

一个类型中的值是否有序也是一种属性，有序类型类 `Ord` 就是规定了这种函数的类型类。`Prelude` 库中的布尔类型实现了有序类型类，因此可以比较 `True` 与 `False` 的大小，如：

```
> True > False
True
```

下面一一介绍 Haskell 中常用的类型类。

2.2.1 相等类型类：`Eq`

可以比较是否相等的类型就需要实现相等类型类 `Eq`。用户可以比较两个布尔值是否相等，可以比较整数是否相等，似乎成了顺理成章的事情。但是，实现的时候其实并没有想象的那么容易。有一些类型可以很直观的就看出是如何比较相等的，有些则要用户自己动手来实现。这里读者只需要知道，一旦一个类型实现了相等类型类，就可以比较相等与否。

在 GHCi 里可以输入 `:t (==)` 来查看它的类型：

```
> :t (==)
(==) :: Eq a => a -> a -> Bool.
```

实际上 `(==)` 是一个函数，它的意思是说，`(==)` 是一个需要任意两个类型为 `a` 的值作为参数，然后会返回一个布尔值的结果的函数，若返回 `True` 表示给定的两个参数相等，`False` 表示不相等。但是，所应用的有着类型为 `a` 的参数一定要实现相等类型类 `Eq`，否则 `(==)` 不知道怎么比较两个值。如 Java 一样，如果要比较两个对象是否相等，需要在这个类中写 `equals` 方法，以后的章节里会提到如何实现类型类。同时，如果一个类型实现了 `(==)` 函数，就可以判定其中的两个值是否不相等，即使用 `/=` 运算符。它的类型与 `==` 是一样的，例如：

```
> :t /= 
/= :: Eq a => a -> a -> Bool.
```

这样，在定义函数时，如果直接或者间接需要对某一类型使用 `==` 或者 `/=` 运算，就需要该类型用相等类型类 `Eq` 做出限定，也就是使用 `Eq a => ...`。这里间接的意思指我们定义的函数并未使用 `==` 或 `/=`，但是该函数调用了一个使用了 `==` 或者 `/=` 的函数。具体如何用类型类限定一个类型，将会在后面讨论。

2.2.2 有序类型类：`Ord`

有序类型类囊括的是可以比较大小的类型。Haskell 中规定：一个有序类型一定为一个可比较相等的类型。也就是说，`Ord` 是基于 `Eq` 的，那么可以比较大小的类型一定可以判定该类型的

两个值是否相等。这样才能够使用`<`、`<=`、`>`、`>=` 等关系运算符。例如，可以比较数字的大小，因为数字是一个有序的类型。数字类型实现了 `Ord` 类型类，同时，也可以比较两个数字是否相等。另外一个例子是因为 Haskell 中的布尔类型实现成了 `Ord` 类型类，所以输入 `True > False`，GHCI 会返回 `True`。同样的，`Char` 类型、`String` 类型都是有序的。这里，由于 `Char` 类型是有序的，因此基于 `Char` 类型的列表 `[Char]`，即 `String` 也是有序的。例如：

```
> "Hello" < "World"
True
```

同理，也可以比较整数列表的大小，比较的方式是从首元素开始依次地进行比较：

```
> [1,2,3] < [2,3,4]
True
```

通过了解有序类型类 `Ord`，知道了它是基于相等类型类 `Eq` 的，也就是说，类型类之间也是有依赖关系的。它们之间的这种依赖关系十分简单，后面会介绍数字类型类 `Num` 里复杂的依赖关系。那么，如果需要对一个类型的值进行大小的比较，用到了`>`、`<`、`>=` 等运算就需要加这个类型类的限定。

2.2.3 枚举类型类：`Enum`

有一些类型可以按一定的顺序枚举的。`Integer` 是一个枚举类型，比如，上节里在 GHCI 里输入 `product [1..10000]`，为什么可以输入`..`来省略中间的数？因为 `Integer` 是一个枚举类型，即只要给定范围，Haskell 就可以一个一个地对这个类型的数据进行枚举。

```
> [1..10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

字符类型也实现了枚举类：

```
>['a'.. 'd']
['a', 'b', 'c', 'd']
```

除`..`以外，一个类型如果实现了枚举类型类，那么给定一个元素（若这个元素不是最后一个）后总是可以使用枚举类型类中定义的 `succ` 函数得到下一个元素。同样地，如果给定一个元素，若这个元素不是第一个，则总是可以使用 `pred` 函数得到它的前一个值。

```
> succ 1
2

> pred 'M'
'L'
```

2.2.4 有界类型类：`Bounded`

可以枚举定义的数据往往是有界的，比如 `Bool`、`Int` 都是有最大值和最小值的。布尔类型中只有两个值，一个是 `True`，它是布尔类型中的最大值；另一个是 `False`，它是布尔类型中的最小值。

```
> maxBound :: Bool
True
```

```
> minBound :: Bool
False
```

此前，已经讨论过 Int 类型，在 32 位系统上它的最大值是 $2^{31}-1$ 也就是 2 147 483 647，它的最小值是 -2^{31} ，也就是 -2 147 483 648。

```
> maxBound :: Int
2147483647

> minBound :: Int
-2147483648
```

字符类型也是有界的，有兴趣的读者可以在 GHCi 中查询一下最大的字符是什么。

2.2.5 数字类型类：Num

数字类型类 Num 指的就是日常生活中使用的整数、小数、有理数等，Int、Integer、Float，以及 Double 是数字类型中最基本的类型。Haskell 对各种不同的数字做了细致地分解，数字类型类下面分了很多其他类型类，如 Real、Fractional 等。它们之间的层级结构十分复杂，但是通过图 2-1 可以让大家理清楚它们之间的关系。这些类型大都被直接定义在了 Prelude 预加载库中，固定长度的 Word 与 Int 类型分别定义在 Data.Word 与 Data.Int 中，同时，复数类型 Complex 还有比值类型 Ratio 则需要分别从 Data.Complex 与 Data.Ratio 中导入。

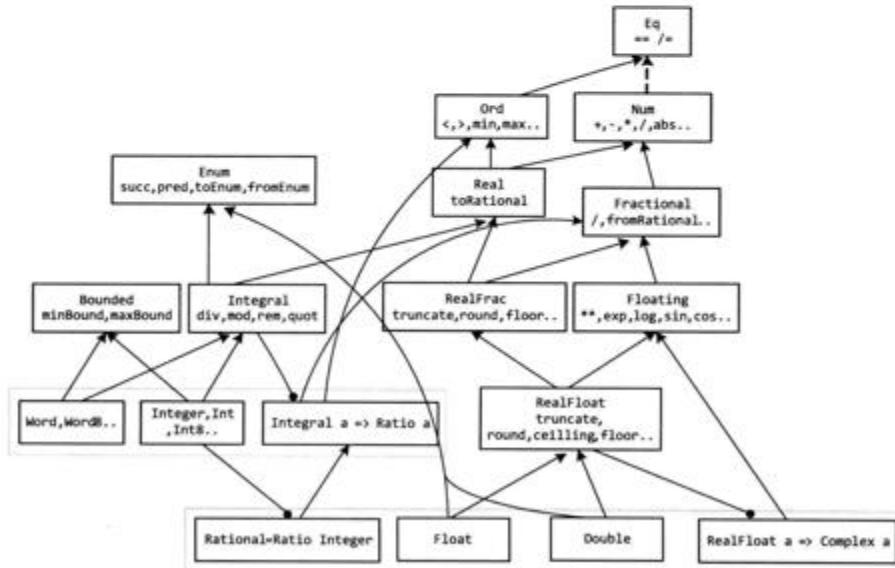


图 2-1 数字相关类型与类型类关系图

图 2-1 中灰色方框内所示的就是“类型”了，如 `Float`、`Integer` 等。有一些类型还需要其他参数，比如，比例类型 `Ratio a`，还有复数类型 `Complex a`。这种关系用一面是小球的线来表示，是因为该类型中还存在类型变量，这种类型的定义将在第 8 章中讲解。

不在灰色方框里的表示类型类。类型类间的箭头表示两个类型类之间的依赖关系。例如，实现有序类型类 `Ord` 的类型，一定要先实现相等类型类 `Eq` 才可以。对于函数的使用将在后边的章节中具体介绍。

类型与类型类间的箭头则说明这个类型直接实现了该类型类。比如，`Float` 与 `RealFloat` 之间有箭头，是因为 `Float` 类型实现了 `RealFloat` 中的一些预先定义好的函数，如 `truncate` 函数，它可以把小数取整。上图中每个类型类下面都写了一些该类型类中预定义的函数，读者可以自己试验一下它们的用途。这里仅仅介绍一些常见的类型类，在后边的章节还将定义自己的类型类与数据类型。

整数部分十分好理解，主要是带符号的整数与无符号的整数，它们都实现了整数除法与取余等运算。因为这些整数是实数的一部分，所以实现了 `Real` 类型类，而 `Real` 类型类又继承了 `Num` 类型类，因此它们就都是 `Num` 类型类的成员了。比值类型 `Ratio` 中需要一个整数类型，根据不同需要，可以用不同的整数来表示它，如 `Int`、`Word`、`Integer`，这样它会用给定的两个整数来表示一个比值。它实现了小数 `Fractional` 类型类，那么它也间接地是 `Num` 类型类的成员了。

小数这一面大约分了 3 级，但是 `Float` 与 `Double` 类型都实现了小数所有的功能，比如除法运算 (`/`)、小数乘方 (`**`)、正弦余弦等。而对于复数，还需要一个参数，可以是 `Float` 也可以是 `Double`。只要实现了 `RealFloat` 类型类的类型就可以，这样就用两个有序的小数表示了一个复数。复数也是可以进行乘方、正弦与余弦等运算的，所以它可以实现浮点小数 `Floating` 类型类。例如：

```
>:m +Data.Complex
> (5:+5) + (1:+1)
6.0 :+ 6.0
```

也可以做些三角函数的运算：

```
> sin (5 :+ 5)
(-71.16172106192103) :+ 21.048644880883543)
```

`Num` 这里仅仅是一个统称，所有的相同类型的数字都是可以比较相等的。这里，数字类型类与相等类型类间用了虚线箭头，这是因为这些类型类不都是间接地通过 `Num` 来实现的有的类型如 `Int`、`Word` 是直接实现的，而复数类型是间接实现的。

另外，可能会引起困惑的地方是 `Float` 与 `Double` 两个小数类型实际是 `Enum` 类型类，这并不代表可以像遍历整数那样遍历所有小数，而只是可以以固定的差值来遍历。例如：

```
> [1.0,1.5..3.0]
[1.0,1.5,2.0,2.5,3.0]
```

此时, Haskell 可以根据前两个值自推断出这个固定的差值是 0.5, 而 succ 与 pred 默认的长度是 1。例如:

```
> pred 0.1
-0.9
```

这些与数字有关的类型类内部规定了很多种运算, 根据不同的功能, 可能需要不同种类的数字。Haskell 中的类型系统对它们进行了细分, 这样, 无论将来处理普通的数字运算还是高精度的数字运算 Haskell 都可以很好地胜任, 强类型也可以保证结果与转型的正确。同时, 有了这种分类清楚的数字, 在以后的应用中也可以根据自己的需要来定制想要的数字类型。

介绍了数字的类型类, 再介绍一下数字类型转换的函数。在编程中, 数字间常常需要转型, 尤其是小数和整数之间的转型。Haskell 是一个类型非常严谨的语言, 所以不能像其他语言那样强制转型, 这样可以避免数字计算中的很多错误。下面介绍一些常用数字转型函数。

(1) `fromInteger :: Num a => Integer -> a` 函数可以将一个一个的整数转为一个重载的数字类型 a。比如, 有时需要将一个整数转为复数类型或者比值类型, 这时就可以使用它。例如:

```
>:m Data.Complex
> fromInteger 5 :: Complex Double
5.0 :+ 0.0

> :m Data.Ratio
> fromInteger 5 :: Ratio Int
5 % 1
```

由于这是一个重载的类型的函数, 因此应尽量指明返回结果的类型。下面介绍的函数都是如此。

(2) `toInteger :: Integral a => a -> Integer` 函数可以将实现为 `Integral` 类型类的类型转换为 `Integer`, 这些类型包括 2.1 节中讨论的 `Int`、`Integer`、`Int8`、`Word`、`Word8` 等。

(3) `fromRational :: Fractional a => Rational -> a` 函数可以将一个有理数化为一个实现了 `Fractional` 类型类的类型。例如:

```
> fromRational (5%1)
5.0
```

(4) `toRational :: Real a => a -> Rational` 函数会将有着实现了 `Real` 类型类的类型 a 化为有理数类型。

(5) `fromIntegral :: (Integral a, Num b) => a -> b` 函数将一个实现整数类型类的类型转换为更为一般数字类型类的类型。

(6) `truncate :: (Integral b, RealFrac a) => a -> b` 函数可以将一个小数的整数部分取出, 而 `properFraction :: (Integral b, RealFrac a) => a -> (b, a)` 函数则会将一个小数的两部分分成整数与小数部分以一个元组返回。

(7) `floor`, `ceiling`, `round` :: (`Integral b`, `RealFrac a`) \Rightarrow `a` \rightarrow `b` 这三个函数分别是下取整函数、上取整函数和四舍五入函数。它们的转型非常易懂和直接，这里就不再讨论了。

2.2.6 可显示类型类：`Show`

一个可以输出在 GHCI 中的类型就是可显示的类型。图 2-1 中没有画出 `Show` 类型类，因为所有的数字类型都是可以输出的。GHCI 中可以输出的结果都在 `Show` 类型类的成员。例如，比值 `Ratio` 类型输出两个整数，两个整数之间用%分开，复数 `Complex` 类型输出两个小数，中间用:+分开。

这些类型在 Haskell 中都可以显示出来的。可是有些类型在 Haskell 中不知道如何显示出来，原因可能是没有办法定义，也可能是需要手动定义。比如，在 Haskell 里，函数类型也是一种类型，可 GHCI 不知道怎么显示它们。因为函数类型没有实现 `Show` 类型类，所以这里函数是无法输出在命令行上的。例如，输出下面的存有函数的列表，GHCI 就报错了：

```
>[(+), (-)]
<interactive>:1:1:
No instance for (Show (a0 -> a0 -> a0))
  arising from a use of `print'
Possible fix:
  add an instance declaration for (Show (a0 -> a0 -> a0))
In a stmt of an interactive GHCi command: print it
```

加法与减法实际是二元函数，GHCI 向用户报告这些二元函数没有实现可显示类型类 `Show`，所以不能输出在 GHCI 中。即 GHCI 不知道如何将这两个函数输出在 GHCI 的命令行上。这引起了一个很有趣的思考题，用户输入 5，GHCI 打印 5，但为什么 GHCI 不能直接打印输入的 `[(+), (-)]`？

学过 Java 的读者可能会知道，在 Java 中打印一个对象需要实现 `toString` 方法。如果一个类没有实现 `toString` 方法，则输出的是根据该对象地址计算的十六进制的散列码（hash code），如@1a3ba42。Java 中的复写 `toString` 方法和 Haskell 中实现 `Show` 这个类型类是相似的道理，只是 Java 中没有把 `toString` 明确定义为接口。关于如何定义新类型以及实现类型类，将在第 8 章和第 9 章中进行讨论。

在定义函数的时候，函数的类型（即函数的标签）大多数情况下是可以省去的，但并不是所有的时候都能省去。对于每一个合法的函数，Haskell 都会用它所带的类型推断系统（type inference system）给出一个推断的合理的类型，这就是为什么它可以工作的原因。但是，这里建议读者在今后书写代码的时候将函数的类型写好，这样不但在定义函数时可以提供思路，也可以像注释一样在书写后检查类型是否匹配，日后对于自己和他人的理解都能提供帮助。

2.2.7 小结

相信读者已经可以区分前边的 `Bool`、`Int` 等类型与 `Eq`、`Ord` 等类型类了。它们是不同的概念。在 Haskell 里，类型是通过 `data` 关键字来定义的实体的类型，如 `Bool` 是这样定义的：`data`

`Bool = True | False.` `Int` 是一个 Haskell 中嵌入的类型，不是这样简单枚举定义出来的，它们都被称为类型。`Eq`、`Ord`、`Eum` 等类型类在 Haskell 里是用 `class` 关键字定义。Haskell 中的 `class` 写 Java 中的完全不同，却和 Java 的 `interface`（接口）有些相像。与 Scala 等一些语言中的 `trait` 十分相似。类型类中定义了一些函数，如果定义了一个新的类型，只要这个类型实现了类型类中声明的方法，这个类型就属于该类型类了，这样一来，一些基于该类型类的函数也就自然地可以用在新的类型上了。这是一种非常好的使用已有代码的机制，在后边的章节中会有更详尽的说明。

2.3 Haskell 中的函数

2.3.1 Haskell 中的值

Haskell 里变量的值在绑定后不会改变，所有变量一定意义上可以理解为定值。例如，创建一个名为 `Value.hs` 的文件，并在文件中定义：

```
a :: Int
a = 5
```

那么 `a` 就是一个全局的常量。在计算过程中，`a` 的值是不会改变的，用户只能引用值而不能修改值。可以在 GHCi 导入 `Value.hs` 文件，在命令提示符里引用 `a`，例如：

```
> 2*a
10

> 1+a
6
```

无论函数如何地被应用，`a` 的值是不变的。再比如，加入一个布尔值 `b`：

```
b :: Bool
b = False
```

由于改动了文件，所以保存文件后要在 GHCi 里重新导入。GHCi 工具栏里有一个快捷按钮 可以做到这一点，也可以在提示符中输入 `:r` 来重新导入文件。导入后在提示符中使用一下 `b`：

```
> b
False

> not b
True

> b || True
True
```

无论如何，定义过的值是没法再改变的。之前提到过，Haskell 值与函数是统一的，函数只

是需要其他参数输入的值。如果定义的是函数，那么这个函数的行为在运行过程中也是不会改变的，对于某一个特定的输入返回的结果总是确定的，这样的函数为纯函数。有人觉得不改内存状态的想法听上去很荒诞，甚至觉得这样是没有办法做计算的。其实，这两种想法都是错误的。不改变内存状态自有道理，而其他编程语言可以完成的工作 Haskell 一样可以完成。

接下来，了解一下这定义这些纯函数所需要的思想，然后学习如何在 Haskell 中定义函数。

2.3.2 函数思想入门

首先需要了解什么是函数。在函数类型中提到过，函数是从参数到结果的一个映射，将一个类型的值映射到另一个类型的值。这里的值不单单是狭义上的值，如 5、`True` 等常量也指函数，区别只是 5、`True` 不需要任何的输入。再说明一下，函数与值在 Haskell 中没有本质区别，函数可以是单一的定值，也可以是任意两个函数间的映射，如函数到函数的映射。

另外，在 Haskell 的世界里，所有的运算符号都可以被看做是函数，如加号+实际上也是一个函数——一个需要两个参数的函数。如果在 GHCI 的命令行里输入：

```
> (+) 5 7
```

结果会输出 12。这种写法只是将中缀运算符置前，`(+)` 只是一个符号，也可以使用 `add`、`plus` 等用户想要的名字。在 Haskell 中，实际上每一个函数都有一个唯一的类型，在命令行里输入`:t (+)` 就可以查看`(+)` 的函数类型：

```
> :t (+)
(+)
  :: (Num a) -> a -> a -> a
```

经过前面的学习，相信读者已经知道了，类型结果首先说明了这是一个柯里化的函数，然后说明了`(+)` 需要两个参数，但这两个参数必须实现了 `Num` 类，最后它会返回一个数字。比如，给定了 5 与 7 就会返回 12。如果只给定第一个参数 5 结果会怎样？例如：

```
> :t (5+)
(5+)
  :: Num a -> a -> a
```

这时，就从一个函数得到了另一个函数。`->` 是向右结合的符号，所以类型`(Num a) -> a -> a -> a` 指的是`(Num a) -> a -> (a -> a)`。如果已经给定了第一个参数 5，就会得到`(5+) :: Num a -> (a -> a)` 这样类型的函数。因为类型中的第一个 `a` 已经给定为 5，所以被消去了。这表明，`5+` 是需要一个数字作为参数返回一个数字的函数。这样，对于需要两个参数的函数，如果给出一个参数，那么实质上返回的是一个函数为结果的，这一点在很多语言中是无法做到的。在 Haskell 中，当给定第一个参数应用于一个 n 元函数，返回的结果是一个 $n-1$ 元函数。那些没有给出所有参数的函数应用称为函数的不完全应用（partial application，有时译为偏函数调用），这是柯里化函数的灵活之处。

很多数学上的概念，在某种程度上都可以理解为函数。例如，在数学中常常表示坐标 (x, y) ，

而事实上，这个元组的类型是 `Num a => (a, a)`。再如，平面中的直线方程是 $y=kx+b$ ，其中 k 与 b 为常数， x 为变量，则对于给定的 x 值总会得到一个 y 值。通俗地说就是，输入一个数，该函数就输出一个数作为结果，所以它的类型应该写为 `Num a => a -> a`。

求导数的过程实际上就是给定一个函数得一个新函数的过程。求一个函数的导数，实际上就是以一个函数作为参数输入，然后返回一个函数作为结果的函数。如果要写出求导函数的类型，则应该是 `diff :: Num a => (a -> a) -> (a -> a)`，这也就是前文中提过的从函数到函数的映射，如果有这样一个函数，那么 `diff (\x -> x + 5)` 的类型就是 `Num a => a -> a`，即 `(\x -> x + 5)` 替换了 `diff` 类型中的左面的 `(a -> a)`，得到了右面的 `(a -> a)`，即导函数的结果。

而积分则是返回一个曲线族，可以理解为返回一个函数与一个任意常数组成的元组，可以写成 `(Num a) => (a -> a) > ((a -> a), c)`。这里仅仅用 `c` 类型表示任意常数，如果不定义任意常数这个概念，也可以把积分理解为返回一个函数的列表的函数，也就是 `(Num a) => (a -> a) -> [(a -> a)]`。当然，这是一个无限的列表。像导数与积分这样的运算，以其他以函数作为输入的函数或者以函数作为返回结果的函数，称之为高阶函数。高阶函数将在第 7 章中具体探讨。

在图像处理中，图像处理滤镜其实本质上也是函数，假设有一个名为 `Picture` 的图片类型，对图片的水平翻转、锐化、模糊等实际是以一个图片为参数，对图片做处理，得到一张新图为结果的函数，它们的类型就应该是 `Picture -> Picture`。图片的旋转也是如此，仅仅是需要一个旋转的角度，给定一个整数的角度，再给定一张图片，返回旋转后的图片，这样它的类型应该写做 `Int -> Picture -> Picture`。

现实中的很多实例都可以当做函数来处理，如数学中的导数、微分、积分、曲面、平面及空间直线；图像处理中的翻转，放缩，旋转；文件的类型转换、压缩。再比如 C 的编译器，它可以理解为一个以字符串 `String`，也就是程序员的代码，作为输入，得到可执行的机器语言的函数，不过这个函数可能是由很多复杂函数复合而成的，但是它的类型可以简单地认为是 `string -> [Code]`，`Code` 表示二进制机器指令。在后面的章节中将会编写简单的计算器，到时候将会看到如何把一个问题看做一个函数，并将它们拆分成多个函数然后复合它们来解决。

2.3.3 函数的基本定义格式

现在，读者应该对 Haskell 的函数有了一定的了解，本节将讨论如何在 Haskell 中定义函数。一般情况下，先要在第一行定义函数名与函数的类型，它称为类型签名（type signature）。定义类型签名后，另起一行写出函数名、参数，最后定义函数体，在 Haskell 中定义的函数的大致格式是这样定义的：

函数名 :: 参数 1 的类型 \rightarrow 参数 2 的类型 $\rightarrow \dots \rightarrow$ 结果类型
函数名 参数 1 参数 2 .. = 函数体

:: 指定函数的类型。类型与集合是不同的。但是，如果把类型简单地当做集合来看待，:: 后面的就是函数的类型，那么 :: 可以理解为数学集合中的属于符号 \in ，比如定义：

```
i :: Int
i = 5
```

这里的 `i` 可以理解为 $i \in \text{Int}$ (数学上常用 \mathbb{Z} 表示整数集), `i` 的值为 5。对于函数时也可以这样理解, 若将 `Int -> Int` 看做一个函数的集合, 如果参数为 n , 阶乘函数 $n!$ 、将整数平方的函数 n^2 等都在该集合之中。

Haskell 有类型推断系统, 它可以自动推断出一个函数的类型。但是, 如果定义函数时可以给定该函数的类型, 那么 Haskell 的类型推断系统会检查系统得出的类型与我们给定的类型是否一致, 这在一定程度上可以保证函数定义的正确性。`::` 可以指定类型, 但具有相同类型的函数往往是定义在一起的, 例如数学上我们常常写 $x, y \in \mathbb{Z}$, 在 Haskell 中它们的类型也可以合并在一起, 函数名之间用逗号分隔。比如:

```
add, sub :: Int -> Int -> Int
add a b = a + b
sub a b = a - b
```

当有多个类型类限定在一个类型上的时候, 它们需要用括号写在一起, 并且中间要用逗号隔开, 或者像柯里伦那样使用`=>`依次连接。例如:

```
function :: (Show a, Ord a) -> a -> a -> a
Function :: Show/a -> Ord a => a -> a -> a
```

.hs 文件的全局函数的名必须要从每行最前端开始写, 因为 Haskell 格式里可以省去很多大括号`{}`与分号`;`, 所以这些格式对于 Haskell 编译代码十分重要。这里再重提一个细节, 在普通的文本编辑器中写 Haskell 代码最好不要用 Tab 键进行缩进, 最好多用一些空格代替, 因为在这样的情况下, GHCi 可能无法正确解析代码。在集成开发环境中, 会将所有的 Tab 键自动换成空格。另外要注意的是, 函数名不能以大写的英文字母和数字开头, 因为大写开头的字母单词是被当成类型或者类型数据使用的, 比如 `Bool` 类型是大写, 并且 `True` 与 `False` 都以大写字母开头的类型的值。下面来定义一些初等的数学函数来了解在 Haskell 如何定义函数。创建名为 `Function.hs` 的文件书写下边的代码。

在数学中, 常常将函数写成这样: $f(x)=4x+1$, 其中 x 是一个数字, 需要被限定在 `Num` 类型类中。它在 Haskell 里被写成这样:

```
f x :: Num a => a -> a
f x = 4 * x + 1
```

然后, 可以在 GHCi 中调用这个函数, 输入:

```
> f 5
21
```

这样就定义了一个名为 `f` 的函数: 把一个数同 4 相乘, 然后加 1。

现在来定义一个函数, 这个函数可以求一个圆的面积, 即 $\text{area}(r)=\pi r^2$, π 是 Haskell 预加载

库中预定义的常量，叫做 pi，所以面积函数可以直接定义为：

```
area x = pi * x ^ 2
```

在数学中，有多元的函数，比如： $f(x,y)=4x+5y+1$ 。

```
f :: Num a => (a,a) -> a
f (x,y) = 4*x + 5*y + 1
```

很明显，这个函数以一个元组即一个实数对作为输入，然后得到一个新的实数。也可以这样定义：

```
f' :: Num a => a -> a -> a
f' x y = 4*x + 5*y + 1
```

通过 2.1 节我们知道，这两个非柯里化与柯里化的函数是等价的，但非柯里化的函数是必须同时输入 x 和 y 后得到结果的。而柯里化的函数则可以对参数 x 和 y 一个接一个地输入，当 x 被输入成 5 的时候，会得到：

```
f'' :: Num a => a -> a
f'' y = 4*5 + 5*y + 1
```

从函数 f 到 f' 的过程就是之前提到的柯里化。

2.3.4 λ 表达式

Haskell 还有另外一种书写函数的格式，与函数的类型相互对应，这就是 λ 表达式。

```
函数类型 :: 参数 1 的类型->参数 2 的类型-> 结果的类型
函数名 = \参数 1 -> \参数 2 -> ... -> 函数体
```

λ 函数的写法源于 λ 演算，很多关于函数式编程的书中将这里的反斜杠写成希腊字母 λ ，有些语言里定义函数写成 $f = \lambda x \rightarrow \lambda y \rightarrow x + y$ 或者 $f = \lambda x.\lambda y.(x+y)$ ，不过各个写法表达的都是同样的意思，就是将函数的参数依次按次序输入到函数体中。这里输入的 x 与 y 不一定是值，也可以是函数，如 $g = \lambda x.\lambda y.x\ y$ 是把 y 应用于函数 x 。

```
> (\x -> \y -> x y) (abs) (-5)
5
> (\x -> \y -> x y) (+9) 5
14
```

那么上节中的 f' 函数可以这样定义：

```
f' :: Num a => a -> a -> a
f' = \x -> \y -> 4*x + 5*y + 1
```

可以看出，函数的类型与函数的定义非常好的对应上了。在代码编译后，函数最终都将会被化为 λ 表达式。参数依次输入的时候，在 Haskell 中可以省去参数间的箭头而简写成：

```
f'=\lambda x y -> 4*x + 5*y + 1
```

除 λ 表达式这种表达方法外, λ 演算中还有一些其他的重要概念被应用到了 Haskell 中, 它们是 α 替换、 β 化简与 η 化简, 其中 β 化简与 η 化简在代码中常常出现, 我们会在稍后一一介绍。

Haskell 为了避免重复计算和类型的歧义, 引入了单一同态限制 (MonomorphismRestriction) 这一概念, 这里我们简称它单态限定。单态限定的引入影响了我们前面提到的 η 化简。为了不让读者对它带来的问题感到困惑, 在下面的第 2 小节中我们会简单地讨论一下它。最后我们来看一下 λ 表达式的用途。

1. λ 演算中的 α 替换、 β 化简与 η 化简

α 替换 (α -conversion) 所指的是在不出现命名冲突的前提下可以将函数的参数重新命名。比如, 下面等号两边的定义是等价的:

$$\lambda x \rightarrow \lambda y \rightarrow x+y =_{\alpha} \lambda a \rightarrow \lambda y \rightarrow a+y$$

这里我们只是将 x 重新命名为 a 。但是, 如果有变量名有冲突那么两者则不相等, 例如:

$$\lambda x \rightarrow \lambda y \rightarrow x+y \neq \lambda y \rightarrow \lambda y \rightarrow y+y$$

β 化简 (β -reduction) 则指的是参数到函数体的替换。应用参数 N 于函数 $\lambda x \rightarrow M$, 相当于在不出现命名冲突的前提下, 把 M 中出现的 x 替换为 N 。

$$(\lambda x \rightarrow M) N \rightarrow_{\beta} [x/N]M$$

例如:

$$(\lambda x \rightarrow x+2) y \rightarrow_{\beta} y+2$$

这样, 就不必为 $x+2$ 这个函数再起一个名字, 借助 λ 表达式可以直接应用一个没有名字的函数到 y 。但是, 遇到下面有命名冲突的情形则不能简单地替换, 例如:

$$(\lambda x \rightarrow \lambda y \rightarrow x+y) y$$

所得的结果将是 $\lambda y \rightarrow y+y$, 这明显是错误的, 所以在遇到这种情况时要先做 α 替换, 把其中的 y 改成一个不冲突的名字, 然后才可以进行 β 化简, 这就是引入 α 替换的主要原因。下面我们来对前面的例子应用一下 β 化简, 手算一下函数应用与计算的具体过程:

```
(\x -> \y -> x y) (abs) (-5)
→_{\alpha} (\y -> abs y) (-5) (应用 \beta 化简替换 x)
→_{\beta} (abs (-5)) (应用 \beta 化简替换 y)
= 5 (根据 abs 绝对值函数定义计算得出)
```

η 化简 (η -reduction) 可以用来消去那些冗余的 λ 表达。在定义函数时, 可以将一个参数传给函数 M , 进行计算的函数和 M 本身是同一个函数:

$$(\lambda x \rightarrow M x) \rightarrow_{\eta} M$$

这里的意思是说, 我们将一个参数 a 应用到这个函数 $(\lambda x \rightarrow M x)a$, 在做 β 化简后所得的结果与直接将参数 a 应用到函数 M 所得的结果相同。

例如, 在 Haskell 中:

```
g :: Int -> Int -> Int
g = \y -> \x -> (+) x y
```

将加法的两个参数输入给加法函数来求和的一个函数，就是加法函数本身，所以函数 g 可以化简为 $g = \lambda x -> (+) x$ ，并且进一步化简为 $g = (+)$ 。

注意，在做 η 化简时尽量标明函数类型签名，因为如果不指明函数的类型，则可能会由于 GHC 默认的“单一同态约束”引发一些问题。下面我们来简单讨论一下单一同态限定，读者不必完全理解它，可以在对类型系统有了一定的了解后再来看这部分内容。

2. 单一同态限定

Haskell Report 2010 中给出引入单一同态限定的原因。引入它一是为了避免重复计算，二是为了消除类型歧义。我们分别来讨论一下。

(1) 重复计算：计算中可能产生类型为 $(\text{Num } a, \text{Num } b) \Rightarrow (a, b)$ 的结果，但是却可能被限定为 $(\text{Num } a) \Rightarrow (a, a)$ 。因为 $(\text{Num } a, \text{Num } b) \Rightarrow (a, b)$ 需要对 a 与 b 进行不同的两次运算，耶鲁大学的 Paul Hudak 构造了一个极端的函数，原本非常简单的函数应该很快被计算出来，但由于未限定为同一类型类使得计算有着指数级别的复杂度。

例如：在计算任意长度的列表时会用到 Data.List 库中的函数 genericLength :: Num a $\Rightarrow [b] \rightarrow a$ ，而下面定义的函数 $f = (\text{len}, \text{len})$ 中 $\text{len} = \text{genericLength } xs$ 。这里我们无法知道 f 中的两个 len 的类型是否一样，可能一个为整数，一个为小数，这样则可能需要计算两次，如果它们就是不一样的数字类型，那么 f 的类型则应当是 $(\text{Num } a, \text{Num } b) \Rightarrow (a, b)$ 。但是，由于单一同态限定，Haskell 默认它们是一样的。

```
> :m +Data.List
> :t genericLength
genericLength :: Num i => [b] -> i
> f xs = let len = genericLength xs in (len,len)
> let f xs = let len = genericLength xs in (len,len)
> :t f
f :: Num t => [b] -> (t, t)
```

我们可以看到这里本来应该不同的两种数字类型却被限定为同一种数字类型 t。如果关闭单一同态限定后重新定义函数 f 后会让 len 有不同的类型：

```
> :set -XNoMonomorphismRestriction
> let f xs = let len = genericLength xs in (len,len)
> :t f
f :: (Num t, Num t1) => [b] -> (t, t1)
```

(2) 类型歧义。在不给定类型上下文的前提下，在 GHCI 中 reads 函数的类型应当如下：

```
> :t reads
reads :: Read a => ReadS a
> :i ReadS
```

```
type ReadS a = String -> [(a, String)]
-- Defined in `Text.ParserCombinators.ReadP'
```

这样，对于在函数体中的变量绑定 `[{n, s}] = reads t` 中的 `s` 的类型则会被推断为 `Read a -> String`，这是一个有歧义的类型，因为类型签名中右侧并没有用到有类型类定义的类型。对于这种情形，GHC 默认将由类型类限定的类型限定到一个类型。对于数字类型类 `Num` 先试用 `Integer`，然后是 `Double`，依次类推，如果都不可以就失败。

所以这里对加法函数做 η 化简而不加类型签名是没有问题的。比如，上一小节中的函数 `g`，对于 `g 5 3` 的参数类型可以是 `Num` 类型中的任何值，结果可以返回 `Integer` 也可以返回 `Double` 类型，由于 GHC 在实现的时候对于数字运算有默认的规则，所以当定义 `g = (+)` 时，将文件导入 GHCI 中，会得到下面的警告：

```
Warning: Defaulting the following constraint(s) to type 'Integer'
  (Num a0) arising from a use of '+'
In the expression: (+)
In an equation for `g': g = (+)
```

意思是说，GHCI 默认 `g` 的类型为 `Integer -> Integer -> Integer`。

但是，如果不加类型签名定义基于相等类型类 `Eq` 的函数 `g = \x -> \y -> (x == y)`，还有 η 化简得到的 `g = \x -> (x==)` 和 `g = (==)`，则会得到下面的错误：

```
Ambiguous type variable `a0' in the constraint:
  (Eq a0) arising from a use of `=='
Possible cause: the monomorphism restriction applied to the following:
  g :: a0 -> a0 -> Bool
Probable fix: give these definition(s) an explicit type signature
  or use -XnoMonomorphismRestriction
```

这就是由单一同态限定（monomorphism restriction）引起类型歧义。对于无法像数字类型类 `Num` 那样把重载类型限定到 `Integer` 或者 `Double` 上的类型类，GHCI 会把它限定到单位类型 `()` 上去，`()` 是单位类型，它只有一个值，这个值记为 `()`。例如：

```
> let f = (==)
> :t f
f :: () -> () -> Bool
```

为了解决这个问题，我们只需要添加相应的类型声明或者在源文件顶端向 GHC 声明 `{-# LANGUAGE NoMonomorphismRestriction #-}` 将单一同态约束关闭即可，但是这样就需要对潜在的那两个问题做出适当的处理。在 GHCI 中我们可以使用命令来设置，例如：

```
>:set -XNoMonomorphismRestriction
> let f = (==)
> :t f
f :: Eq a => a -> a -> Bool
```

同理，当关闭了单态限定后，对于 η 化简后的定义 `g = (+)` 的类型就不会限定为整数了。

而是 $\text{Num } a \Rightarrow a \rightarrow a \rightarrow a$ 。更多关于单一同态限定的内容可以参阅 Haskell 2010 Report 4.5.5。

3. λ 表达式的应用

λ 表达式的应用主要有两个：对于柯里化函数，在不给定前一个参数的前提下给定后一个；另外一个应该是匿名函数（anonymous function）。在使用一些高阶函数时，如果不想定义新函数，可以使用 λ 函数来定义，这种函数也常常被称为匿名函数或者无名函数（nameless function），后面会见到这种情况。

使用中缀运算符，并且只给出部分参数时：

```
> :t (^) 2
(^) 2 :: (Num a, Integral b) -> b -> a
```

对于 $(^)$ 运算符，第一个参数为底数，第二个参数为幂。在使用的时候可以直接给出第二个参数而不给出第一个：

```
> :t (^3)
(^3) :: Num a => a -> a
```

这里只给出了 $(^)$ 运算符的第二个参数，跳过了第一个参数。若 $(^)$ 不为一个中缀运算符，通过 λ 表达式可以很容易做到这一点 $\backslash x \rightarrow (^) x 3$ ，这种情形下就不能做 η 化简了。

在上面的例子中，对于运算符其实可以不用 λ 表达式，但是对于之前定义的 f' ，如果只想给定变量 y ，而不想给定 x ，又该怎么办呢？借助入表达式很容易做到。

```
f' :: Num a => a -> a -> a
f' x y = 4*x + 5*y + 1
```

只要将函数写为 $\backslash x \rightarrow f' x 5$ 就可以了。

因为 λ 函数不需要函数名来定义，所以在 Haskell 中 λ 函数被称为匿名函数。当然，如果需要，那么也可以给它命名。在 GHCi 的命令行中，可以直接将函数应用到参数上：

```
> (\x -> (^) x 3) 2
8
```

若不借助 λ 函数，则需要对函数命名：

```
> let f x = x ^ 3
> f 2
8
```

2.3.5 参数的绑定

在数学上，求一个三角形的面积常常用到海伦公式。公式定义如下：

$$S = \sqrt{p(p-a)(p-b)(p-c)} \left(p = \frac{a+b+c}{2} \right)$$

a、b、c 是三角形的三边。由于 p 的值仅定义一次就好，这种情况就可以用 let..in.. 在函数定义中做替换。海伦公式在 Haskell 中可以定义为：

```
s :: Double -> Double -> Double -> Double
s a b c = let p = (a+b+c)/2 in sqrt (p*(p-a)*(p-b)*(p-c))
```

这样，在应用函数的时候，p 会自动被替换为 $(a+b+c)/2$ 。用户不仅可以绑定表达式，还可以绑定函数，例如：

```
>let f x = x+1 in f 5
6
```

多个表达式绑定可以用分号隔开：

```
>let x=2; y=2 in x+y
4
```

除了用 let..in.. 以外，还可以使用 where。依旧以海伦公式为例，可以先定义好函数，然后在 where 关键字后边定义参数 p 作为函数定义的补充。就像数学公式里的括号一样，where 是对公式的一个补充说明。

```
s' :: Double -> Double -> Double -> Double
s' a b c = sqrt (p*(p-a)*(p-b)*(p-c))
  where
    p = (a+b+c)/2
```

这样，在做计算的时候 Haskell 会计算 $(a+b+c)/2$ ，再代入到定义中。但是，使用 let..in.. 绑定要注意命名捕获（name capture），比如：

```
>let x = 6 in x * let x = 2 in x*x
24
```

表面上看这里计算的是三个 x 值相乘，但是这里的 x 值是不等的，第一个 x 的值是 6，后两个 x 的值是 2，所以计算的结果是 $6 \times 2 \times 2$ ，结果是 24。

但是，如果定义一个函数，用了 where 绑定，那么它在函数的全部的范围都是有效的。当然，也不可能用同一个变量的名。但是，where 也可以像 let 那样有多级，即在 where 内定义的函数内又有 where，读者可以自己尝试一下，但是这种用法并不常见。

2.4 Haskell 中的表达式

2.4.1 条件表达式

条件表达式是语言中最常用到的。例如，先来定义一个简单的函数叫做 isTwo，它的作用：如果输入的参数是 2，就返回 True；否则为假。

```
isTwo :: Int -> Int
isTwo n = if n == 2 then True else False
```

加载到 GHCi 中，在命令提示行里输入：

```
>isTwo 2
True

>isTwo 7
False
```

`isTwo` 函数非常简单而直接，事实上，只要写 `isTwo = (==2)` 就可以了。这里主要是为了介绍条件表达式。Haskell 里的 `if..then..else` 顺序式编程语言有些不同，就是 `else` 后的表达式不可省略。也就是说，必须定义条件成立的时候返回的值，也必须定义条件不成立的时候返回的值，并且返回的类型必须相同，这样一定程度上保证了函数定义的完整性。在顺序式语言中，`if..then..else..` 是一种表达式结构，而在 Haskell 中可以理解为一个运算符也可以理解为一个函数（运算符与函数其实是等价的，这一点将在后面的小节中详细说明），这个运算符既不像加号那样处于两个参数中间为中缀运算符（infix operator），也不像对数函数那样位于一个参数前边为前缀运算符（prefix operator）。它是一个混合位置运算符（mixfix operator），它是 `Bool -> a -> a -> a` 类型的一个函数，即需要一个条件参数，若满足，则返回第二个参数为结果；否则返回第三个。

2.4.2 情况分析表达式

情形分析表达式是用 `case` 与 `of` 关键字来对一个类型中不同的值进行讨论的，它与顺序式编程里的 `switch .. case` 类似。区别就是，Haskell 不用像 `switch` 那样选择到一个条件继续向下运行而不自动跳出。因此在 Haskell 中，不需要用到 `break` 关键字。顺序式语言中 `default` 关键字和 Haskell 里通配符类似，即一个下划线 “`_`”，表示除了上面的匹配外，匹配所有的值。下面用 `case..of..` 再来定义 12 月份不同天数的函数，中间的一些定义在这里省略了：

```
month :: Int -> Int
month n = case n of
    1 -> 31
    2 -> 28
    ...
    9 -> 30
    10 -> 31
    11 -> 30
    12 -> 31
    _ -> error "invalid month"
```

这里不再用等号作为返回结果的符号而是一个箭头 `->`，和函数类型里用的箭头是一样的。省略的部分也需要依次被定义。这里的 “`_`” 意为除 1~12 以外的输入都会得到错误提示。如果

有多个匹配满足，那么只有第一个会被返回。

2.4.3 守卫表达式

守卫表达式（guarded expression）是使用`|`将函数的参数按特定的条件分开，就像一个守卫一样，如果不能满足条件，它绝不会让不符合条件的表达式运算。不同条件的守卫表达式的需要对齐。本例用绝对值函数来演示：

```
abs :: Num a => a -> a
abs n | n > 0 = n
      | otherwise = -n
```

`otherwise` 是匹配时的默认定义。因为`|`后的一个一定是一个布尔类型，所以 `otherwise` 为布尔类型并且它的值很明显是 `True`。有兴趣的读者可以打开 `Prelude.hs`，找到 `otherwise` 的定义。这里需要说明的是，在定义函数时可能有很多条件，但是如果多个条件同时满足 Haskell 只会匹配第一个。

2.4.4 模式匹配

模式匹配（pattern match）中的 pattern 理解为模式，它指的是一个类型的值或者定义成的形式。模式匹配的本质可以通过 `case...of...` 表达式体现出来，即每个类型的数据（或者称为值）都可以看做是该类型的一个具体形式，我们仅仅是要把需要匹配的情形依次写下，Haskell 会从上到下的查询，直到找到第一个匹配，如果有多个复合的匹配，那么只有第一个匹配被执行。

```
month :: Int -> Int
month 1 = 31
month 2 = 28
month 3 = 31
month 4 = 30
month 5 = 31
month 6 = 30
month 7 = 31
month 2 = 40
month 8 = 31
month 9 = 30
month 5 = 20
month 10= 31
month 11= 30
month 12 = 31
month _ = error "invalid month"

> month 5
31
```

以上几种定义函数的方法并不一定非要分开使用，相反，它们常常在函数定义时同时使用。

这样就有了更多的自由度来对函数进行定义。这里，读者可以看到 `case..of..` 和模式匹配是对于类型值不同形式的分析，所以在一定意义上它们是等同的，而用竖线的守卫表达式和 `if..then.. else..` 都是对参数的条件讨论的表达式，所以它们在一定程度上是等同的。

另外，也可以对列表的两种不同模式进行匹配。如果读者记得列表的定义则会知道，一个列表只可能为空，或者为一个元素与另外一个列表的组合。这样，可以定义 `head` 函数来返回一个列表的第一个元素（因为 Prelude 中已经定义了 `head`，所以后加一个单引号）。

```
head' [] = error "empty list"
head' (x:_)= x
```

在定义函数的时候，如果我们并没有把所有的模式都分别定义好，那么当调用函数的时候可能会出现 `exception of non-exhaustive patterns` 的错误，即在模式与定义匹配的时候对于某些数据的模式有所遗漏，在今后写程序的时候应当注意这一点。比如，定义 `month` 函数时，我们使用了`_`通配符以匹配其他不合理的整数，让它们都返回 `error`。

各种表达方式其实可以混合使用，我们后面会常常见到这样的例子。

2.4.5 运算符与函数

前面已经提到，运算符（如加号、减号等）实质上是与函数是一样的。这里所说的运算符指的是所有的运算符，因为 Haskell 所有的运算符都是基于函数定义的。

例如，`(+)` `(-)` `(*)` 其实是有着同样类型的函数，它们的都是 `Num a => a -> a -> a`。而 `(/)` 略有些特殊，它的类型是 `Fractional a => a -> a -> a`，即所有参数都会被转换成小数进行运算。

运算符不过是一些规定了可以放在参数中间或者末尾的函数，并且用了一些特殊的符号表示，将它们写在最前端实质是一样的。例如在 GHCi 里输入 `(+) 5 6` 与 `5 + 6` 是一样的。读者可以将一些二元函数用反单引号 (`)（位于数字 1 键左侧的~键）来转换成位于参数中间的运算符。比如，`5 `div` 2` 表示 `div 5 2`，其中 `div` 是整数除法函数。同理，取余函数 `mod` 也可以写为中缀运算符的形式，`5 `mod` 2` 与 `mod 5 2` 是一样的。运算符号其实不过是为了简化书写而约定的，本质还是函数，在 Haskell 里函数与运算符是等价的。

2.4.6 运算符与自定义运算符

运算符可能有 3 种属性，即优先级、结合性和位置。在 Haskell 中，运算符号有 0~9，共十个优先级。结合性又分为左结合性、右结合性、无结合性。根据位置又可分为前缀、中缀和后缀运算符，一般使用的函数常常可以理解为前缀运算符。这样，在上节提到的函数也可以理解为运算符，定义的函数默认有着最高的优先级，比 9 还高并且为左结合。例如，`f g h i` 意为 `((f g) h) i`。因此，在某些情况下，要对函数的参数加括号才能正确使用这些函数。表 2-2 展示了一些常用运算符的优先级与结合性，它们都是中缀的运算符。

表 2-2 Haskell 中常用运算符的优先级与结合性

	左结合	无结合	右结合
9	<code>!!</code>		<code>.</code>
8			<code>^, ^^, **</code>
7	<code>*</code> , <code>/</code> , <code>'div'</code> <code>'mod'</code> , <code>'rem'</code> , <code>'quot'</code>		
6	<code>- +, -</code>		
5			<code>:, ++</code>
4		<code>==, /=, <, <=, ></code> , <code>>=</code>	
		<code>'elem'</code> , <code>'notElem'</code>	
3			<code>&&</code>
2			<code> </code>
1	<code>>>, >>=</code>		
0			<code>\$, \$!, \$!!`seq`</code>

有些运算符在前边没有提到。这里对这些运算符进行简单的介绍。

- `(!!)` 从一个列表中取出第 n 个元素（从 0 开始）。如果输入的数字大于等于列表的长度就会出现错误。例如，在 GHCi 里输入 `[1,2,3,4,5]!!3`，会返回 4。
`> [1,2,3,4,5]!!3`
`4`
- `(.)` 是复合函数的运算符号，将在第 7 章高阶函数中讨论。
- `(^)` 在 Haskell 里表示乘方。它的类型为 `(Num a, Integral b) => a -> b -> a`。由它的类型可知，底数可以是一个任意的数字类型，而指数一定要是整数类型。
- `(^^)` 也为乘方函数，这里规定底数 a 必须是 `Fractional` 类型类的实例。它的类型为 `(Integral b, Fractional a) => a -> b -> a`。
- `(**)` 还是乘方函数，但它的类型是 `Floating a => a -> a -> a`，意思就是说，这个函数的两个参数都可以为小数。比如，它可以计算 0.64 的 0.5 次方等。例如：
`> 0.64 ** 0.5`
`0.8`

根据不同的需要，可以使用不同的乘方运算符号。由于类型不同它们的计算效率也不尽相同。

- `mod` 和 `rem` 是取余函数，`quot` 和 `div` 是求商函数，但是在计算负数的时候，运算结果会有些不同。这里简单说明一下不同点。

在 GHCi 里运行一下：

- `div (-12) 5 得 -3;`
- `mod (-12) 5 得 3;`
- `quot (-12) 5 得 -2;`
- `rem (-12) 5 得 -2.`

可以看出，`div` 与 `mod` 是一组，因为商-3 乘以 5 再加上 3 正好是 12。同理，`quot` 与 `rem` 是一组。在计算的过程中，在保证余数 r 的绝对值小于除数的情况下，`div` 总是要结果逼近于负无穷，而 `quot` 总是需要将结果逼近 0。

很多情况下，一定要对负数加括号，否则出现会有一些“意外”的结果，比如 $-2 \bmod 6$ 我们会得到-2，而我们想要的结果是 4，注意，减号 “-” 的优先级比 `mod` 要低，所以这里实际上在计算 $-(2 \bmod 6)$ 。因为 Haskell 的语法中使用了大量的中缀运算符，所以在今后定义函数时要注意运算符的优先级，否则会因为括号遗漏或优先级没考虑而导致程序出现问题，从而没有得到预计的结果。

- `(:)` 正如前面所提到的，元素列表连接运算符也是一个函数，并且是一个有着类型 $a \rightarrow [a] \rightarrow [a]$ 的函数，即将一个新的元素加入一个列表的第一个位置。比如：

```
> l:[1,2,2]
[1,1,2,2]
```

- `(++)` 列表连接符，顾名思义是连接两个列表的，类型为 $[a] \rightarrow [a] \rightarrow [a]$ 。比如：

```
> [1,2]++[1,2]
[1,2,1,2]
```

优先级为 4 的第一行运算符就不做过多的解释了。`/=` 表示的是不等于，不要与其他语言混淆，写成 `!=` 或者 `<>` 都是错误的写法。

优先级为 1 的两个运算符是有基于 `Monad` 类型类的，这里先不讨论。直接看优先级为 0 的运算符。

- `(\$)` 的定义为：

```
(\$) :: (a -> b) -> a -> b
f \$ x = f x
```

这个函数是很实用的。当有多个函数应用时，Haskell 默认计算为左结合，而这里因为 `(\$)` 有着最低的优先级，并且是右结合，那么使用 `(\$)` 就可以避免使用过多的括号来定义函数。比如：当需要定义 `f (g (h x))` 时，可以写成 `f \$ g \$ h x`，使得代码更清楚并且十分方便，相当于把 x 输入给 h ，所得结果输入给 g ，依次类推。

想必读者应该知道为什么类型签名中的 `->` 为右结合而函数的应用却是左结合的了，即 `f g h` 意为 $(f g) h$ ，而 `a -> b -> c` 意为 $a -> (b -> c)$ ^①。因为类型签名中输入的类型在左，比

^① 有着 $a -> b -> c$ 这样类型的函数是无法定义的，这里只是举个例子，读者可以想想为什么有着这样类型的函数无法定义。

如 $f :: a \rightarrow b \rightarrow c$ 意为 f 是输入一个 a 类型的值，返回一个有着 $b \rightarrow c$ 类型的函数。这样，输入参数的类型在左，可是应用函数到参数时对应的输入参数却写在函数名的右侧，即 $f x$ ，如此一来，函数应用的结合性就与类型签名的结合性正好相反了。

以后读者在定义与使用函数时，不但需要给定函数参数，最好还要考虑类型在给定参数时是如何计算的，给定的参数替换中了函数类型中的哪些部分，会得到怎样的类型。只有明白了这些，才算是真正理清了函数与类型间的关系。

(`$!`)、(`$!!`) 与 `seq` 的使用与 Haskell 的惰性求值特性有关，有关惰性求值的内容将在第 15 章进一步探讨。

当然，这不是 Haskell 中所有的运算符，运算符只是定义了优先级的普通函数，在后面将见到更多的运算符。下面来定义一个运算符。虽然 Haskell 中通过类型类支持函数的重载，但 Haskell 不像 C++ 那样非常自由地支持函数与库运算符重载的。比如，不可以随意地重载加号，因为它的类型必须是 `Num a => a -> a -> a`。但在 Haskell 中，可以自由地定义自己的运算符号，但要声明它的结合性是怎样的，优先级是多少。声明运算符的关键字有三个：`infixl` 为左结合，`infixr` 是右结合，`infix` 为无结合。定义时先用这三个关键字说明结合性，再声明优先级，最后给出运算符的符号。如果需要定义多个结合性与优先级相同的运算符，那么它们之间要用逗号分开。

```
infixr 5 <->, <+>
(<->), (<+>) :: Int -> Int -> Int
(<->) x y = x - y
(<+>) x y = x + y
```

这样就定义了一个新的作用为减号的运算符。不同的是，它是右结合的。如果导入文件到 GHCi 里，输入 `10 <-> 5 <-> 2` 会得到 7，因为定义的这个符号 (`<->`) 是右结合的，这个运算实质上计算的是 $10 - (5 - 2)$ 的结果。

本章小结

本章介绍了 Haskell 中的类型系统以及函数。首先学习了类型与类型类，其次学习了如何使用各种表达式定义函数和运算符。类型系统与函数是紧密联系的，了解它们是如何统一起来的非常重要。有些人刚开始可能会觉得类型系统在编程实践中是一种束缚，因为在编程时不但要考虑计算是如何进行的，还需要考虑类型是否协调。同时，在使用函数时，心里不但要清楚返回的值是什么，还要清楚返回值的类型，这样才能通过 Haskell 的类型检查。刚开始的时候可能会不适应，但是久了就会发现，Haskell 的类型系统在把程序引到一个正确的方向上。

只有充分利用 Haskell 类型系统带来的好处，才能写出错误更少的代码。希望类型系统可以在定义函数时，成为你真正的朋友。

第3章

基于布尔值的函数

在上一章中，相信读者已经大约了解了如何在 Haskell 中定义函数的。这里，单独拿出一章的内容来复习它们。首先，学习 Haskell 中的 `module` 与 `import` 关键字是如何使用的，然后定义一些基于布尔类型的函数作为过渡，旨在加深在 Haskell 中通过各种表达式定义函数及运算符的理解。

3.1 关键字 `module` 与 `import` 简介

在编写 Haskell 代码时，第一行可以声明一些编译器参数（如果不需要编译器参数，那么这一行可以省略）。然后，以 `module XXX where` 的格式定义该模块的名字，通过这种方法，可以控制其他文件中代码对该模块下函数以及类型的访问。最后，定义该程序文件下的函数、类型等。这里假定模块名为 `Test` 有着一些函数，如 `f1`、`f2` 等，若想设定 `f3` 为私有的，就可以写为：

```
module Test (f1,f2) where  
  
f1 = ...  
f2 = ...  
f3 = ...
```

`Test` 后面括号内的内容表示对其他 Haskell 代码文件可见的函数——不但可以是函数，也可以是类型。此时，如果在另外一个文件中使用 `import` 关键字导入 `Test` 模块，那么函数 `f3` 不会被导入，因为这个函数是 `Test` 私有的。如果不写括号，则说明所有的函数都是对外可见的。如果不想把定义的文件当做模块在别的文件中调用，那么可以在 `Test` 后面加空括号 () 即可或者省略 `module` 的定义。

`import` 关键字可以用来导入需要用的库，它的使用非常灵活，可以从库中只导入特定的函

数，也可以导入所有函数。

如果只需要 Test 模块中的 f1 函数，那么可以写为：

```
import Test (f1)
```

这样，只有 f1 被导入当前的代码中。不但可以有选择性地从一个模块中导入一些函数，也可以导入一个模块中所有的函数，然后选择性地隐藏其中的一些函数。比如，一些库函数与用户定义的函数可能是有命名冲突的（或者不想使用），这时可以使用 hiding 关键字将它们的定义隐藏，例如：

```
import Prelude hiding (catch)
```

这样，catch 函数就被隐藏了起来，我们可以定义自己的 catch 函数。

有时，需要导入多个模块，但是其中的两个模块下有两个函数名称一样，可是又需要同时使用它们。这时，可以使用 qualified 关键字来对不同的模块命名，例如：

```
import qualified Test as T
```

这样，使用 Test 模块中的函数时需要加 T.。比如，T.f1 说明调用 Test 模块中的 f1 函数。

当然，也可以不用 import 关键字，但在使用函数时需要给出完整的函数所在路径，比如：

```
> :t Data.List.permutations
Data.List.permutations :: [a] -> [[a]]
```

关于 module 与 import 关键字的更多细节，读者可以参阅 Haskell 2010 Report 5.3。

3.2 简易布尔值的函数

逻辑在计算机科学领域有着重要的地位——计算机使用的 0 与 1 其实就是布尔逻辑。1 表示真 (True)，0 表示假 (False)。本章通过定义一些基于布尔值的函数与解决通用逻辑门，使读者进一步了解 Haskell 中的函数是如何通过形匹配、守卫表达等方式定义的。函数保存于 Boolean.hs 文件中。

先来定义一些简单的函数，逻辑非 not、逻辑与 and，其对应运算符为 (`&&`)；逻辑或 or，其对应运算符为 (`||`) 和相等 (`==`)。因为它们在预加载库 Prelude 中已经定义，虽然想重新以这些名字定义，但是功能与库提供的不一样。根据 3.1 节的内容，可以通过 import 与 hiding 关键字把它们隐藏起来。

之前提到过很多种表达式，如 if..then..else、case..of.. 等。读者可以根据自己的喜好来写代码，但常用的依旧是模式匹配，这样函数的可读性更强，也更工整。

```
--Boolean.hs
import Prelude hiding ((/=), (==), not, and, or, (&&), (||))
```

先定义等号:

```
(==) :: Bool -> Bool -> Bool
(==) True True = True
(==) False False = True
(==) _ _ = False
```

_为通配符，表示任何其他的情况，非常的方便。这样看起来很明显，除了输入一样的返回 True 以外，其他的情况都是 False。not 的定义如下：

```
not :: Bool -> Bool
not True = False
not _ = True
```

因为之前已经定义过(==)，在逻辑中，not 函数其实等价于输入的参数是否与 False 相等，经过 η 化简以后可以得到这样的定义：

```
not' = (==False)
```

代入 True 与 False 得出结果，就会知道这是符合 not 的定义的。在函数式编程里，用户知道的知识越多，想得越多，代码就越精炼。同样，也更容易被理解。下面将异或(/=)、and 与 or 合并在一起定义：

```
xor, and, or :: Bool -> Bool -> Bool
xor b1 b2 = not $ b1 == b2

and True b1 = b1
and False _ = False

or False b1 = b1
or True _ = True
```

当然我们，也可以用条件语句来定义 and 与 or，即：

```
{-
and b1 b2 = if b1 then b2 else False
or b1 b2 = if b1 then True else b2
-}
```

这里将它们注释了，读者应该很容易看出来，模式匹配使代码的可读性更强。

此外，我们可以定义 if then else。之前在 2.4.1 节中提到过 if then else 在 Haskell 中不过是一个函数。它的类型是这样的 $\text{Bool} \rightarrow \text{a} \rightarrow \text{a} \rightarrow \text{a}$ 。所以它可以这样定义：

```
condition :: Bool -> a -> a -> a
condition True t f = t
condition False t f = f
```

这些基本的函数已经定义完了，现在试着用运算符来代替它们。首先要声明符号是哪种类

型的，是无结合还是有结合，有的话向左还是向右，优先级是多少。

```
infix 4 ==
infix 4 /=
infixl 3 &&
infixl 2 ||
```

(||) 与 (&&) 的定义分别是 or 与 and，此处只是将它们替换为运算符号：

```
(||) = or
(&&) = and
(/=) = xor
```

|| 其实就是我们定义的 or 函数，而 && 其实是我们定义的 and 函数，所以等号右边直接定义就可以了。以上 3 个逻辑运算在电路的设计中常常用图 3-1 所示的符号表示。

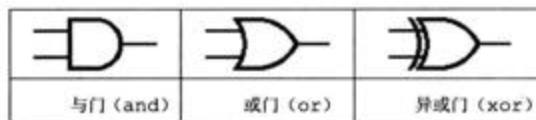


图 3-1 与门、或门与异或门符号

这些符号其实本身充当的是“黑匣子”的作用，纯函数也可以这样理解。将几个参数输入到这个匣子中，然后这个匣子会返回结果。在给定参数时不必在意匣子中是什么，也不必在乎函数是如何定义的，给定同样的值匣子总是会返回同一个值。

有了这些基本的逻辑运算，可以试着定义逻辑门电路中 1 位 (bit) 的半加法器，如图 3-2 和图 3-3 所示。



图 3-2 1 位半加法器

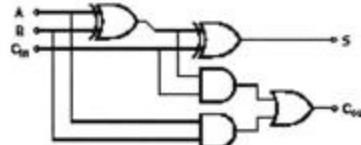


图 3-3 1 位全加法器

对于半加法器，如果输入的 A 与 B 都为 1，那么 S（和）为 0，C（进位）为 1；如果 A 与 B 中仅有一个 1，那么 S 为 1，C 为 0，表示这一位的加法运算不需要进位。对照图 3-3 就可以定义成：

```
hA :: Bool -> Bool -> (Bool,Bool)
hA a b = (a=/=b, a&&b)
```

从图 3-3 可以看出，全加法器是由两个半加法器还有一个或门定义的，所以有：

```
fA a b c = let {axb,aab} = hA a b in
            let {axbx,c,axbac} = hA axb c in (axbx,c,aab||axbac)
```

3.3 与非门和或非门

在逻辑门电路中，有两个非常重要的逻辑门——与非门 nand 与或非门 nor，即 not and 与 not or。这两个逻辑门称为“通用逻辑门”，它们的特殊之处在于仅仅用 nand 与 nor 中的一个就可以定义出其他所有的逻辑门。读者可以先找一些逻辑门的内容来阅读。在本节里，将带领读者来试着实现一下。因为需要用 nand 与 nor 实现其他的逻辑门，所以定义 nand 与 nor 的时候不能用到其他的函数，并且在定义其他函数的时候也不能用到之前定义过的函数，只能用 nand 与 nor，但是可以用定义过的函数来推导。

比如，nand 与 nor 的真值表如图 3-4 所示。

输入		A NAND B	A NOR B
A	B		
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

图 3-4 nand 与 nor 逻辑门真值表

根据图 3-4 所示的真值表，0 代表 False，1 代表 True，很容易用模式匹配来定义它们：

```
nand, nor :: Bool -> Bool -> Bool
nand True True = False
nand _ _ = True
nor False False = True
nor _ _ = False
```

下面，用它们来定义 not 函数：

```
not1, not2 :: Bool -> Bool
not1 b = nand b b
not2 b = nor b b
```

之后，用它们来定义 and 函数。因为 not nand 就是 and，所以第一个很容易定义出来。第二个其实也很容易，因为 nor b1 b2 可以表示为 $\neg(b_1 \vee b_2)$ ，根据德摩根定律 $\neg b_1 \wedge \neg b_2$ ，想得到 $b_1 \wedge b_2$ ，只需要将两个参数应用一次 not 函数即可。

```
and1, and2 :: Bool -> Bool -> Bool
--and1 b1 b2 = not1 $ nand b1 b2
and1 b1 b2 = nand (nand b1 b2) (nand b1 b2)
--and2 b1 b2 = nor (not2 b1) (not2 b2)
and2 b1 b2 = nor (nor b1 b1) (nor b2 b2)
```

接下来，分别用它们来定义 or。有了定义 and 的基础，这个就很简单了。

```
or1, or2 :: Bool -> Bool -> Bool
--or1 b1 b2 = nand (not1 b1) (not1 b2)
or1 b1 b2 = nand (nand b1 b1) (nand b2 b2)
--or2 b1 b2 = not2 $ nor b1 b2
or2 b1 b2 = nor (nor b1 b2) (nor b1 b2)
```

这样，`nand` 和 `nor` 就可以互相定义了。现在来分别定义不等价函数 `xor`。`xor` 的定义是 $(b_1 \wedge \neg b_2) \vee (\neg b_1 \wedge b_2)$ ，所以有：

```
xor1, xor2 :: Bool -> Bool -> Bool
--xor1 b1 b2 = orl (andl b1 (notl b2)) (andl (notl b1) b2)
--xor1 = nand (notl (andl b1 (notl b2)) (notl (andl (notl b1) b2)))
```

如果将上式展开，那么最后的结果会很长。所以，需要思考如何将其化简。这里我们用 F 表示假，T 表示真。

$$\begin{aligned}
 & (b_1 \wedge \neg b_2) \vee (\neg b_1 \wedge b_2) \\
 &= FV(b_1 \wedge \neg b_2) \vee FV(\neg b_1 \wedge b_2) \quad (\text{F 是 V 的单位元}) \\
 &= (\neg b_1 \wedge b_1) \vee (b_1 \wedge \neg b_2) \vee (\neg b_2 \wedge b_2) \vee (\neg b_1 \wedge b_2) \{(\neg b_1 \wedge b_1) = F\} \\
 &= (b_1 \wedge \neg(b_1 \vee b_2)) \vee (b_2 \wedge \neg(b_2 \wedge \neg b_1)) \quad (\wedge \text{是 } \vee \text{ 上满足分配律}) \\
 &= (b_1 \wedge \neg(b_1 \wedge b_2)) \vee (b_2 \wedge \neg(b_2 \wedge b_1)) \quad (\text{德摩根定律}) \\
 &= \neg((\neg b_1 \wedge \neg(b_1 \wedge b_2)) \wedge \neg(b_2 \wedge \neg(b_1 \wedge b_2))) \quad (\text{德摩根定律})
 \end{aligned}$$

在这里， $\neg(b_1 \wedge b_2)$ 刚好是 `nand` 逻辑门，整个化简过程其实就是在拼凑这种形式，所以 `xor` 可以表示为以下形式：

```
--xor1 b1 b2 = nand (nand b1 (nand b1 b2)) (nand b2 (nand b1 b2))
```

这里看到，`(nand b1 b2)` 被重复使用。在 Haskell 里，可以用 `where` 替换 `nand b1 b2`，写成：

```
xor1 b1 b2 = nand (nand b1 nb1b2) (nand b2 nb1b2)
  where nb1b2 = (nand b1 b2)
```

练习

读者可以用 `nor` 定义 `xor`，然后再试着用两个通用逻辑门定义 `xnor`，下面给出 `xnor` 的模式匹配定义：

```
xnor False False = True
xnor False True = False
xnor True False = False
xnor True True = True
```

本章小结

本章是对上一章的复习，主要是应用模式匹配还有条件表达式等学过的内容，这些内容本身并没有什么难点，但是这个过渡十分重要。在学习了第 2 章相当多的内容后，相信这样一个过渡会使读者对于 GHCi 的使用、Haskell 函数及运算符的定义有了更多的了解。

第 4 章

库函数及其应用

通过学习前两章，相信读者对 Haskell 的类型以及如何定义函数有了一些基本的了解。但是“巧妇难为无米之炊”，不可能一编写程序就从头开始定义所有的函数。在本章中，主要熟悉一下 Haskell 中的预加载库，即 `Prelude`，还有基于字符与位运算的库函数。熟练使用这些函数对于以后的编程十分重要。

4.1 预加载库函数

之前反复提到过，预加载库 `Prelude` 相当于一个初始的环境，里面定义了很多类型，如整数、布尔类型和很多函数、类型类。在 GHci 或者 Hugs 启动后就可以使用它们了。下面我们就来了解一下 `Prelude` 中的函数。

4.1.1 常用函数

1. 恒值函数 `id`

从恒值函数 `id :: a -> a` 的类型中可以看出，它是给定一个任何的值，返回这个给定的值。

```
>id 5
5
>id True
True
```

到这里，也许读者会觉得这个函数没有用，但读到后面就会发现其实不然。

2. 常值函数 `const`

函数 `const :: a -> b -> a` 的作用是给定两个元素，只返回第一个：

```
>const True 3
True
```

恒值函数 `id` 虽然简单，但其实是一个很常用的函数，常常用它来占位。比如，当需要定义一个给定两个参数，且只返回第二个参数的函数，就可以定义为 `const id`。

```
> :t const
const :: a1 -> b -> a1

> :t id
id :: a -> a

> :t const id
const id :: b -> a -> a
```

由于 `id` 是一个 $a \rightarrow a$ 的类型，当 `id` 作为 `const` 的第一个参数后， $a \rightarrow a$ 类型替换了 `const` 中的第一个参数的类型 `a1`。Haskell 的类型推断系统会得到 `const id` 的类型为 $b \rightarrow a1$ ，因为 `id` 替换的 `a1` 有着 $a \rightarrow a$ 的类型，所以 $b \rightarrow a1$ 中的 `a1` 也应该有着一个 $a \rightarrow a$ 的类型，这样 `const id` 的类型就应该为 $b \rightarrow a \rightarrow a$ 。这样它会返回二元函数的第二个参数。

```
>const id True 3
3
```

如果读者对这个结果感到奇怪，那么可以对整个计算过程进行演绎。这是函数式编程中最美的一部分。在后面的章节中将常常使用这种方法。

```
const id True 3
= (函数应用为左结合)
= (const id True) 3
= (应用 const 函数，将 id 函数返回)
id 3
= (应用 id 函数)
3
```

3. 参数反置函数 `flip`

函数 `flip :: (a -> b -> c) -> b -> a -> c` 可以将一个二元函数的两个参数的顺序颠倒，以后会了解到这个函数的用处。

```
>flip (:) [1,2,4] 5
[5,1,2,4]
>flip(-) 38
5
```

由于 `flip` 将 `(:)` 符号的参数反了过来，因此，当给定参数的时候也反过来，即先输入列表，再输入首元素。同样，`flip` 减法运算符则是计算 $8-3$ 的值。通过 `flip` 函数我们也可以定义同 `const id` 功能一样的函数：

```
>flip const True 3
3
```

4. 错误函数 error

函数 `error :: String -> a` 是抛出异常的函数。有些时候，程序出现异常导致全部的程序出错，这时，可以用 `error` 函数来返回错误，返回时，可以加一个字符串来告诉用户为什么有错误。第 1 章中的 `month` 函数当用户输入除 1~12 以外的整数的时候就会有 `invalidmonth` 异常。它有一个多态的类型 `a`，所以无论在返回什么类型的函数中返回异常都可以用它。

```
a = error "a is an error"

> a
*** Exception: a is an error
```

5. undefined

`undefined` 函数是由 `error` 定义的。

```
undefined :: a
undefined = error "Prelude.undefined"
```

由于 `undefined` 有多态的一个类型 `a`，因此有时有一些函数可以先不定义，暂时搁置一边，这是为了保证测试时 GHCI 不会报错，可以使用 `undefined`。例如：

```
month :: Int -> Int
month = undefined

days :: (Int, Int) -> Int
days (m, d) = month m + d
```

这样，如果代码文件中还定义了其他函数，就可以在 GHCI 中调试其他函数的同时，未定义的 `month` 函数不会让编译器报错。

6. min 与 max

这是一对常用的函数，`min :: Ord a => a -> a -> a` 将返回两个参数中较小的那个，而 `max` 则会返回两个参数中较大的那个。例如：

```
> min 5 6
5
```

4.1.2 基于列表的函数

列表是 Haskell 中的一个非常重要的数据结构，下面介绍的所有函数都是关于列表的。

1. null

它会判定是否一个列表为空，它的类型是 `[a] -> Bool`。

```
>null []
True

>null [1,3]
False
```

2. length

`length` 函数会返回一个列表的长度，它的类型是`[a] -> Int`，返回的长度的类型不是任意精度的整数而是`Int`。因此，如果处理过长的列表，就不可以用这个函数了。此时，可以使用`Data.List` 库中的`genericLength` 函数。

```
>length [1,3,4,5,8]
5
```

若想求一个数字列表的平均值，则这样定义函数是错的：

```
avg xs = sum xs / length xs
```

因为`(/)` :: `Fractional a => a -> a -> a`，而`Int` 与并没有限定在`Fractional` 类型类中，所以会有类型的错误。解决的方法是使用`fromIntegral` 函数：

```
avg xs = sum xs / fromIntegral $ length xs
```

或者，可以使用库`Data.List` 中的`genericLength` 函数，它的类型是`Num a => [b] -> a`。Haskell 会自动处理重载的数字类型`a`，不会引发类型错误。

```
import Data.List (genericLength)

avg xs = sum xs / genericLength xs
```

3. (!!)

之前已经介绍过这个函数。这个运算符可以取得给定列表中从 0 开始的第`n` 个元素。这个运算符的类型是`[a] -> Int -> a`。

```
>[1,2,3,4]!!0
1

>[True,False,True]!!2
True
```

如果取的元素位置比列表的元素多，则 GHCI 会报出错误。

```
>[] !! 4
*** Exception: Prelude.(!!): index too large
```

4. reverse

倒置函数`reverse` 可以将一个列表的顺序倒过来：

```
>reverse [1,2,3]
[3,2,1]
```

5. head 和 last

这两个函数分别取一个列表的第一个元素与最后一个元素。它们的类型都是 $[a] \rightarrow a$ 。如果列表为空，则会报出错误。

```
>head "Hello World!"
'H'

>last "Hello World!"
'!'
```

6. init 和 tail

它们分别将一个列表的最后一个元素与第一个元素去掉，得到一个新的列表。它们的类型是 $[a] \rightarrow [a]$ 。如果列表为空，则会报出错误。

```
>init [1..10]
[1,2,3,4,5,6,7,8,9]

>tail [1..15]
[3,5,7,9,11,13,15]
```

7. map

map 意为映射，是将一个函数应用到列表中的每一个元素，然后得一个新的列表的函数。它的类型是 $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$ 。比如，让列表中所的数字都加 1：

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

再比如，对列表中所有的布尔值取反：

```
> map not [True, False, False]
[False, True, True]
```

当然这个函数可以为更复杂的函数，比如将一个列表中的数先平方再加 1，借助 λ 匿名函数可以很容易定义它：

```
> map (\x -> x^2 + 1) [2,3,4,5]
[5,10,17,26]
```

$(\lambda x \rightarrow x^2 + 1)$ 意为将 x 平方后加 1，用在 map 中的意思就是将列表中的每个元素作为 x 然后平方加 1 得到一个新的列表。虽然本书还未讨论高阶函数，但是 map 只是一个简单的高阶函数，应该不难理解。

8. filter

`filter` 是过滤函数，是需要一个条件判断的函数，然后可以从一个列表中选出满足给定条件的元素。为了使用 `filter`，这里再介绍两个很简单的函数，即 `even` 与 `odd`。给定一个整数，通过这两个函数可以判断其奇偶性。

```
> odd 4
False
```

```
> even 4
True
```

下面看一下 `even` 函数类型。

```
>:t even
even :: Integral a => a -> Bool
```

下面使用 `filter` 函数来从一个列表中过滤出偶数来：

```
filter :: (a -> Bool) -> [a] -> [a]
>filter even [1,2,3,4,5,6]
[2,4,6]
```

同样，使用这个函数，也可以把大于等于 7 的函数过滤出来。`(>=7)` 是一个函数，它的类型是给一个数字，返回是真还是假，即：

```
>:t (>=7)
(Ord a, Num a) -> a -> Bool
```

这正是本例需要的类型。

```
>filter (>=7) [9,6,4,2,10,3,15]
[9,10,15]
```

与 `map` 一样，`filter` 也是一个高阶函数，因为它以一个函数作为输入。读者可以使用 λ 匿名函数写一个表达式，从整数列表过滤出所有平方后加 7 以后大于 30 的数字。

9. take 和 drop

`take` 函数可以从头任意连续地取一个列表的多个元素。它们的类型是 `Int -> [a] -> [a]`。

```
> take 5 [1..]
[1,2,3,4,5]
```

如果个数为负，则结果为空。如果要取的个数比总长度要大，则全部取下。`drop` 函数与 `take` 函数相反，而是将列表中前多个元素舍弃。

```
>drop 3 [1,2,3,4,5,6]
[4,5,6]
```

读到这里，你能通过学过的函数定义(!!)运算符吗？

10. span 和 break

这两个函数的类型都是 $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow ([a], [a])$ 。span 函数可以根据一个条件，从左至右，当遇到第一个不符合条件的元素时停止，将一个列表分成由两个列表组成的元组。

```
>span even [2,4,6,7,8,9]
([2,4,6],[7,8,9])
```

break 函数则与 span 函数相反，它会根据一个条件，从左至右，当遇到符合条件的时候停止。这里就不再举例了。

11. takeWhile 和 dropWhile

之前的 take 和 drop 函数是通过给定一个整数来取得或者去掉列表中的前几个元素，而 takeWhile 和 dropWhile 则需要一个条件来判断，条件不成立的时候停止，实质上取的是 span 结果的第一个列表。同理，dropWhile 是将一个列表左端不符合条件的舍去，实质上取的是 span 结果的第二个列表。

```
>takeWhile (>5) [6,7,8,3,5]
[6,7,8]
```

你能通过 span 函数来定义 takeWhile 与 dropWhile 两个函数吗？

12. splitAt

这个函数可以将一个列表在任何的位置分开。它的类型是 $\text{Int} \rightarrow [a] \rightarrow ([a], [a])$ 。例如：

```
>splitAt 5 "Hello World! "
("Hello"," World!")
```

其实，借助 take 与 drop 函数，splitAt 很容易定义：

```
splitAt' :: Int -> [a] -> ([a], [a])
splitAt' n xs = (take n xs, drop n xs)
```

13. repeat 和 replicate

重复函数 repeat 可以将一个元素在列表里重复无数次：

```
>repeat True
[True, True .....
```

这是一个无穷列表，点击回车或者按 Ctrl+C 组分键来终止程序。也可以用[True..]来表达这个无穷的列表：

```
> [True..]
[True, True .....
```

`replicate` 是复制函数，它的类型是 `Int -> a -> [a]`。这个函数可以将一个元素复制给定的次数，比如：

```
>replicate 5 True
[True, True, True, True, True]
```

实际上，我们有了 `take` 和 `repeat` 函数后很容易定义 `replicate` 函数：

```
replicate :: Int -> a -> [a]
replicate n a = take n (repeat a)
```

也就是说，从一个无穷的列表中取前 n 个元素。因为 Haskell 的惰性求值特性，这里的 `repeat` 不会永远重复下去。

14. any 和 all

这两个函数的类型为 `(a -> Bool) -> [a] -> Bool`，`any` 用来查询一个列表中是否存在符合给定条件的元素，而 `all` 会判定是否列表中所有的元素都符合给定条件。比如，查询一个列表中是否有偶数：

```
>any even [1,2,3,4]
True
```

判断一个列表中是否所有的数均为奇数：

```
>all odd [1,3,5]
True
```

15. elem 和 notElem

函数 `elem :: Eq a => a -> [a] -> Bool` 可以判断一个列表中是否存在某一元素。显然，`a` 一定要是可以比较相等的类型，所以在类型签名中需要 `Eq` 类型类。例如：

```
> elem 1 [4,5,1]
True
```

同样，`prelude` 中还有 `notElem`，`notElem` 是 `elem` 的否定。有了 `null` 和 `filter` 函数，`elem` 是非常好定义的，只要在文件中这样定义：

```
elem' :: Eq a => a -> [a] -> Bool
elem' a xs = not $ null (filter (==a) xs)
```

也就是说，只要过滤出来的结果不为空就说明给定的元素在列表中。

16. iterate

迭代函数 `iterate` 的类型为 `(a -> a) -> a -> [a]`，它可以将第 1 个函数应用在第 2 个参数上多次，借助 `iterate` 函数可以来生成以 1 为首项，以 2 为公比的等比数列：

```
>iterate (*2) 1
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144,
524288, 1048576, 2097152, 4194304, 8388608, 16777216, 33554432, 67108864, 134217728, 26843
5456, 536870912, ...]
```

这是一个无穷的列表，单击 或者按 **Ctrl + C** 组合键来终止程序。

17. until

函数 `until :: (a -> Bool) -> (a -> a) -> a -> a` 可以迭代地来生成数据直到满足给定的条件为止。它停止需要一个条件、一个迭代的函数还有一个迭代的初始值，首次到达条件时停止，然后返回结果。例如，想得到以 1 为首相，以 2 为公比的等比数列，且第一个比 500 大的数：

```
>until (>500) (*2) 1
512
```

18. zip 相关函数

`zip` 函数可以将两个列表结合成一个元组的列表。比如：

```
>zip [True, False, True, False] [2, 4, 5, 6, 7]
[(True, 2), (False, 4), (True, 5), (False, 6)]
```

当元素个数不相等的时候，多余的元素会被忽略。`zip` 函数是一个非常有用的函数。在 Haskell 里，列表访问没有索引值，有的时候需要记录元素的位置，这时就可以通过 `zip` 函数来加一个序数，并且不必知道要加序数的列表的长度。

```
>zip "Hello World" [0..]
[('H', 0), ('e', 1), ('l', 2), ('l', 3), ('o', 4), (' ', 5), ('W', 6), ('o', 7), ('r', 8), ('l', 9),
('d', 10)]
```

`unzip` 是把一个二元元素列表分成两个列表元素的函数，`unzip :: [(a, b)] -> ([a], [b])`，即将 `zip` 后的列表还原，比如：

```
>unzip [('H', 0), ('e', 1), ('l', 2), ('l', 3), ('o', 4), (' ', 5), ('W', 6), ('o', 7), ('r', 8),
('l', 9), ('d', 10)]
("Hello World", [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

`zipWith` 函数的类型是 `(a -> b -> c) -> [a] -> [b] -> [c]`，相信读者只要看类型就会知道这个函数可以做什么。它需要一个二元函数作为参数，然后再输入两个列表，最后把两个列表中的元素取出一一对应地进行第一个参数指定的二元运算。比如：

```
>zipWith (+) [5, 6, 7, 3] [2, 3, 4, 7]
[7, 9, 11, 10]
```

当然，有的时候要用到将三个列表的中元素合成有三个元件的元组，这时可以用到预加载库中的 `zip3`、`unzip3`。此外，`Prelude` 中还提供 `zipWith3` 等函数，读者可以自己查阅一下 API，然后试用一下。

19. concat

concat 函数可以将一个列表中的列表相连，它的类型是 $[[a]] \rightarrow [a]$ 。比如：

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

这样，就将一个列表的列表串连成一个列表。

20. concatMap

这个函数先使用 map 函数将 $[a]$ 计算为 $[[b]]$ 类型的结果，再使用 concat 函数来得到类型为 $[b]$ 的结果。例如：

```
> map (replicate 3) [1,2,3]
[[1,1,1],[2,2,2],[3,3,3]]

> concatMap (replicate 3) [1,2,3]
[1,1,1,2,2,2,3,3,3]
```

借助 map 与 concat 函数，读者可以自己试着定义 concatMap 函数。

4.1.3 定义历法公式

接下来，通过定义一个历法的公式来练习一下介绍过的函数。求任意一天是星期几的函数是这样定义的：

$$W = \left((y-1) + \left\lfloor \frac{y-1}{4} \right\rfloor - \left\lfloor \frac{y-1}{100} \right\rfloor + \left\lfloor \frac{y-1}{400} \right\rfloor + d \right) \bmod 7$$

$\lfloor x \rfloor$ 为下取整函数，也就是 floor。公式中 y 为给定的年份， d 为日期在当年累积的天数。

为了增加可读性，可以重命名一些类型。

```
type Weekday = Int
type Year = Int
type Month = Int
type Day = Int
```

然后，定义上方的数学函数，如下：

```
week' :: Year -> Day -> Weekday
week' y d = let y1 = y - 1 in
            (y1 + (div y1 4) - (div y1 100) + (div y1 400) + d) `mod` 7
```

由于需要求一个日期在某一年的累积天数，因此第一要判断这一年是否为闰年，第二要定义一个函数求每个月的天数。有了 elem 函数，就不必使用繁复的模式匹配了。最后，相加求得累积的天数。

先定义判定闰年的一——能被 4 整除，但不能被 100 整除的为闰年。比如，1996 年为闰年，但 1900 年为平年。但是，如果能被 400 整除，这一年为闰年。比如，2000 年为闰年，被称为千年大闰。这样的调整也是有误差的，大约三千年一次，这里就不做考虑了。

```
isLeapYear :: Int -> Bool
isLeapYear y = (mod y 4 == 0) && (mod y 100 /= 0) || (mod y 400 == 0)
```

再来定义一个函数，给定一个年份与月份，返回这个月的天数。年份在这里唯一的作用就是判定是否是闰年，如果是，那么 2 月有 29 天。

```
monthDays :: Year -> Month -> Int
monthDays y m | m == 2 = if not $ isLeapYear y then 28 else 29
               | elem m [1,3,5,7,8,10,12] = 31
               | elem m [4,6,9,11] = 30
               | otherwise = error "invalid month"
```

accDays 函数可以积累某一年内，该日期所经过的天数。这里，用 map 将 12 个月中所有的天数全部算出，但是只用 take 函数对前 $m-1$ 个月的天数进行求和，最后加上第 m 个月的天数 d 就得到了这个日期在一年中所积累的天数。

```
accDays :: Year -> Month -> Day -> Int
accDays y m d | d > monthDays y m = error "invalid days"
               | otherwise = (sum $ take (m-1) (map (monthDays y) [1..12])) + d
```

最后，就可以定义 week 函数求出给定日期是星期几了。

```
week y m d = week' y (accDays y m d)
```

有兴趣的读者可以找一些日期来测试一下 week 函数，比如今天。希望到这里，大家能对函数式编程有所了解。这里所说的了解是指习惯了函数的这种思想，即函数计算不是一个连续修改内存变量的过程，而是通过输入定义函数，然后计算输出的过程。

4.1.4 字符串处理的函数

1. show

show 函数的类型 `Show a => a -> String`，即可以将所有在 show 类里的数据作为字符串，`String` 类型输出，例如：

```
>show 6
"6"
```

如果想找出从 1~100 中所有包含 6 的数字，只要这样写：

```
contains6 :: [String]
contains6 = filter (elem '6') (map show [1..100])
```

即，将包涵“6”的字符串——`(elem '6')`，从把 0~100 转换成字符串后的列表，即 `map show [1..100]` 中过滤出来，即 `filter`。函数的定义直接，并且非常容易理解。

```
>contains6
["6","16","26","36","46","56","60","61","62","63","64","65","66","67","68","69",
 "76","86","96"]
```

2. read

`read` 函数类型 `read :: Read a => String -> a` 与 `show` 相反，可以将可读的类型从字符串 `String` 类型解析为类型 `a`。上图以数组作为输出，如果想输入 `Int` 类型，那么可以这样写：

```
contains6' :: [Int]
contains6' = map (\str->read str::Int) \$ filter (elem '6') (map show [1..100])

>contains6'
[6,16,26,36,46,56,60,61,62,63,64,65,66,67,68,69,76,86,96]
```

这里注意，因为同一个字符串可以被读成不同的类型（一个整数可以被当做 `Int`，还可以被当做 `Integer`），所以，要用`::`来指明想要的结果类型。

`read` 函数返回的结果的类型可能有很多种，需要指定一种，`::` 符号就是指定结果类型的符号。

3. lines 和 unlines

`lines` 函数读入的字符串以`\n`为分隔，变成一个 `String` 的列表，它的类型是 `String -> [String]`，例如：

```
>lines "first line\nsecond line"
["first line", "second line"]
```

`unlines` 是 `lines` 的反函数，即以一个 `[String]` 作为输入，将列表内的字符串以`\n`换行符分开，所以它的类型是 `[String] -> String`。例如：

```
>unlines ["first line", "second line"]
"first line\nsecond line"
```

4. word 和 unword

`words` 函数将一个字符串用空格分开，分成一个 `String` 的列表，`words :: String -> [String]`

```
> words "first second"
["first", "second"]
```

同样，`unwords` 是 `words` 的反函数。那么，`unwords` 的类型很明显是 `[String] -> String`。

下面来写一个函数，可以将一句话的单词倒过来。一共分三步，先将句子以空格分开，转换成一个 `[String]` 类型，即一个 `String` 的列表，然后用 `reverse` 倒置这个列表，最后再用 `unwords` 函数将其组合在一起，在文件中可以这样定义：

```
reverseSentence :: String -> String
reverseSentence str = unwords (reverse (words str))
```

括号使函数不是很漂亮，此时，可以用\$来定义：

```
reverseSentence str = unword $ reverse $ words str
```

这里可以用η化简，所以str可以省略。

```
reverseSentence = unword $ reverse $ words
```

直接在GHCI里使用测试函数，比如：

```
>unwords $ reverse $ words "This is a sentence"
"sentence a is This"
```

4.2 字符与位函数库简介

在本节里，将学习使用两个库函数，分别是字符与位运算的库函数。

4.2.1 Data.Char

编程中需要使用到很多重要的处理字符的函数。Haskell是强类型的所以并不能支持类型间的强制转型。因为强制转型也是导致程序错误的根源之一，Haskell要避免这些发生，所以取而代之的是一些有着严格定义类型的函数。它被定义在Data.Char中。

```
>:m +Data.Char
```

函数chr与ord可将ASCII码与字符相互转换。

```
>chr 90
'Z'
```

```
>ord 'Z'
90
```

此外，还有isDigit、isLower、isUpper、isAlpha、isControl等函数，分别为判定一个字符是不是数字、大写、小写、英文字母、控制字符。例如：

```
> isControl '\ESC'
True
```

更多关于字符的函数，读者可以参阅API文档中的Data.Char。

4.2.2 Data.Bits

在本节中，将了解一下Data.Bits库，因为计算时常常要做基于二制的比特运算。读者可

以参阅 GHC 库的文档来查看 Data.Bits 中的函数，也可以在 GHCI 中用 :browser (可简写为 :b) 一个库中定义的内容，比如：

```
> :browse Data.Bits
class Num a => Bits a where
  (.&)amp; :: a -> a -> a
  (.|.) :: a -> a -> a
  xor :: a -> a -> a
  complement :: a -> a
  shift :: a -> Int -> a
  rotate :: a -> Int -> a
  bit :: Int -> a
  setBit :: a -> Int -> a
  clearBit :: a -> Int -> a
  complementBit :: a -> Int -> a
  testBit :: a -> Int -> Bool
  bitSize :: a -> Int
  isSigned :: a -> Bool
  shiftL :: a -> Int -> a
  shiftR :: a -> Int -> a
  rotateL :: a -> Int -> a
  rotateR :: a -> Int -> a
```

确定一个数是否为偶数，只需要确定最低位是否为 0 即可，所以只需要与 1 做“且”运算即可。比如，由于 $0010 \& 0001 = 0$ ，因此 0010（也就是 2）为偶数。其实，even 函数可以定义如下：

```
isEven x = (x .&. 1) == 0
```

同理，应用且运算也可以十分快速地测试一个整数是否为 2 的整数次方。比如数字 8，二进制码为 1000，计算过程为 $1000 \& (1000-1) \rightarrow 1000 \& 0111 \rightarrow 0$ ，所以 8 为 2 的整数次方。

```
powerOf2 x = (x .&. (x-1)) == 0
```

shift 函数可以将一个数很快地做乘 2 与除 2 的运算。

```
mul2 x = shift x 1
div2 x = shift x (-1)
```

关于其他的函数，读者可以自己试着使用 :m +Data.Bits 来测试，这里就不再过多介绍了。

本章小结

本章介绍了 Haskell 中常用的函数，同时还利用这些函数解决了一些小问题。相信到了这里，读者应该对 Haskell 编程有了一个较为清楚的认识。如果读者了解其他的过程式编程语言，那么就应该感觉到 Haskell 与它们的不同了。到这章为止，Haskell 最基础的部分就已经结束了。倘若读者觉得还有一些部分不清楚，如 GHCI 的使用、函数定义的格式等，可以自己在课后做一些练习来熟悉它们。

第 5 章

递归函数

本章介绍一下递归函数（recursive function）。Haskell 中是没有其他语言里的 while 与 for 循环的。通过前面的内容，相信读者应该了解了，从某种程度上说，Haskell 中没有像 C 语言一样的内存变量的，所有的函数都是通过递归函数和普通的函数运算来完成的。这样做有很大的好处。如果每个函数都有明确的定义，那么 Haskell 中的函数代码很容易被数学化，这将使程序正确性的验证更为容易。因为数学化的函数更易于被证明出是可优化的，并且可以推导出优化后的函数。

首先，要知道什么是递归函数？简单来说，就是一个函数在自己调用自己，直到一种特殊的情况下，递归才会停止。

为什么递归如此重要？没有一个简单的答案能说明。计算机里的很多数据结构都是可以基于递归定义的，如自然数结构、树结构、列表结构等。计算机程序的循环结构是完全可以用递归来表达的，递归定义也使得函数式编程中函数的定义比起顺序式程序中的函数更加数学化，更便于数学推理演绎。在其他类型的语言中出错时，往往要追寻变量的内存是如何变化的。但是，在 Haskell 里就不必这样做，因为所有的数据在内存中都可以理解为是不变的，而正确性可以用数学方法来证明。

在本章的前几节中，先定义一些简单的递归函数，再来讨论更多有趣的递归问题，例如，麦卡锡的 91 函数、斐波那契数列、二分法查找、汉诺塔，以及一些排序算法。最后再来讨论一下关于递归的周边内容，如不动点函数（fixed point function）、无基本条件的递归等。

5.1 递归函数的概念

递归在函数式编程里是一个非常重要的概念，对于一个一般的递归函数，函数的定义可分为两部分：

- (1) 递归的基本条件（base case）；
- (2) 递归步骤（recursive step）。

递归函数会从给定的值一直调用它自己。在调用的过程中，函数的参数不断发生变化，向基本条件靠拢，调用一直到满足递归的基本条件才停止。例如，阶乘函数 factorial，对于阶乘函数基本条件为 factorial 0 = 1，然后是递归步骤的定义 factorial n = n * factorial (n-1)，这里计算一下 4 的阶乘是多少：

```

factorial 4
= 4 * factorial 3 (由 factorial 递归步定义)
= 4 * (3 * factorial 2) (由 factorial 递归步定义)
= 4 * (3 * (2 * factorial 1)) (由 factorial 递归步定义)
= 4 * (3 * (2 * (1 * factorial 0))) (由 factorial 递归步定义)
= 4 * (3 * (2 * (1 * 1))) (由 factorial 递归基本条件)
= 4 * (3 * (2 * 1)) (计算得出)
= 4 * (3 * 2) (计算得出)
= 4 * 6 (计算得出)
= 24 (计算得出)

```

回顾一下推理演绎的过程。当表达式中存在一个 factorial 函数的时候，这个函数就会再调用自己一次，直到参数减小到递归的基本条件为止。

有些读者可能会问：如果 factorial 里的参数 n 小于 0 会怎样？答案就是：这个递归停不下来，最终导致栈溢出。可以来演算一下：

```

factorial (-1)
= (-1) * factorial (-2)
= (-1) * (-2) * factorial (-3)
...

```

这样的结果，对于这个函数来说并不会让人感觉到意外，因为可以推导出来的。想要避免这种情况也很简单，只要再加一个“n 小于 0”的条件，然后返回错误就可以了。

```

factorial :: Integer -> Integer -> Integer
factorial n = if n < 0 then error "n is less than 0"
  else if n==0 then 1
  else n * factorial (n-1)

```

这里可以用 error 函数用来返回异常。由于结果可以当做是多态类型 a，即 String-> a，这样可以让返回的结果和其他的类型协调。a 在类型推断的时候会被替换成 Integer 来满足 factorial 函数的类型 Integer -> Integer。这样，如果输入 factorial (-1)，结果将会是：

```
*** Exception: n is less than 0.
```

再比如，最大公约数函数 gcd 也是递归定义的(gcd 也是 Prelude 中的函数，所以定义为 mygcd)：

```

mygcd :: Int -> Int -> Int
mygcd x y = if y == 0 then x else mygcd y (mod x y)

```

下面来计算一下 mygcd 12 8：

```

mygcd 12 8
= mygcd 8 (mod 12 8)

```

```
= mygcd 8 4
= mygcd 4 (mod 8 4)
= mygcd 4 0
= 4
```

由此可以看到，递归定义的函数非常数学化，非常便于数学演绎。当然，一个递归函数的基本条件可能有很多个，比如将要讨论的斐波那契数列和归并排序算法。同样，递归步也可能不止一个。下面，先写一些简易的递归函数来了解、学习并区分一些不同种类的递归，然后对那些有趣的递归问题进行——探讨。

5.2 简单递归函数

首先定义乘方函数 power，由于 $x^0=1(x \neq 0)$ 和 $x^n=xx^{n-1}(n \geq 1, x \neq 0)$ ，指数为自然数的乘方可以定义为：

```
power :: Int -> Int -> Int
power _ 0 = 1
power x n = x * power x (n-1)
```

这样一步一步计算是很慢的。当指数为偶数，由于 $x^{2n}=x^n x^n$ ，这样 x^n 就不需要计算两次，而 n 为奇数时 $x^{2n+1}=xx^n x^n$ ，这样也可以避免重复地计算 x^n 。因此，乘方函数可以定义为：

```
power :: Int -> Int -> Int
power _ 0 = 1
power x n | odd n = let p = power x ((n-1) `div` 2) in x * p * p
           | otherwise = let p = power x (n `div` 2) in p * p
```

在之前的章节中，介绍了很多库函数。现在，在本节里自己动手定义它们。首先定义 product 函数。它可以被理解为数学上的 $\prod_{k=1}^n A_k$ 公式， A 表示所求积的列表。在定义基于列表的递归函数时需要考虑两种情形，一是列表递归的基本定义，即 []；二是当列表至少有一个元素时的情形，即 $x:xs$ ， xs 代指余下的列表，并且余下的列表可能为空。

```
product' [] = 1          --(1)
product' (x:xs) = x * product' xs --(2)
```

我们可以很容易地对这个计算进行演绎：

```
product [1,2,3]
= product (1:[2,3])  (由：的定义 1:[2,3] = [1,2,3] )
= 1 * product (2:[3])  (由(2)有)
= 1 * (2 * product (3:[]))  (由(2)有)
= 1 * (2 * (3 * product []))  (由(1)有)
= 1 * (2 * (3 * 1))  (由乘法计算)
= 1 * (2 * 3)  (由乘法计算)
= 1 * 6  (由乘法计算)
= 6  (由乘法计算)
```

在使用列表时，`(:)`为一个函数。将一个元素放在一个列表的第一个位置，它常常被称为列表的构造（construction）函数，在有些书中常常写做 `cons`。

```
cons = (:)
> cons 5 [1,2,3]
[5,1,2,3]
```

下面，使用递归来定义一个函数 `snoc`，即 `cons` 的逆写，将一个元素加在一个列表的最后：

```
snoc :: a -> [a] -> [a]
snoc x [] = [x]
snoc x (y:ys) = y: snoc x ys
```

有兴趣的读者可以像 `Product` 那样推导 `snoc 4[1,2,3]` 的计算过程。

定义某些函数的时候，当列表为空时没有办法得出结果，可以使用 `error` 报出错误。比如，`last` 函数将返回列表的最后一个元素，但当列表为空时没有办法返回，所以报出错误。这里函数的递归基本条件为只有一个元素（即 `[x]`）的时候。

```
last' :: [a] -> a
last' [] = error "empty list"
last' [x] = x
last' (_:xs) = last' xs
```

`Prelude` 中的 `take` 函数也是递归定义的，定义起来也十分容易：

```
take' n _ | n <= 0 = []
take' _ [] = []
take' n (x:xs) = x : take' (n-1) xs
```

`elem` 函数可以判定一个元素是否在一个列表中。如果需要判定一个元素是否在列表中，就蕴含着这个类型中的元素之间可以判定是否相等，所以需要使用 `Eq` 类型类做限定。当到空列表时，则说明列表中没有该元素，那么为假。如果不为空，那么将列表的第一个元素与要查找的元素比较。如果相等则说明在列表中，那么返回 `True`，否则递归地继续判定，读者可以推一下 `elem' 3[1,2,3,4]` 的计算过程。

```
elem' :: Eq a => a -> [a] -> Bool
elem' _ [] = False
elem' a (x:xs) = if a==x then True else elem' a xs
```

练习

1. `Data.List` 中有一个函数为 `delete`，它可以删除列表中的元素，即它的类型为 `delete :: Eq a => a -> [a] -> [a]`。定义一个 `delete` 函数从列表中删去所有列表中出现的给定的元素。例如：

```
> delete 2 [4,2,4,6,2]
[4,4,6]
```
2. 请用与 `take` 类似的方法定义 `Prelude` 中的 `drop` 函数。

5.3 扩展递归与尾递归

在 5.1 节里，定义了阶乘与最大公约数函数。本节中，可以比较一下求阶乘函数与最大公约数函数的区别，虽然它们都是用递归来定义的，但还是有一些不同。

```

factorial :: Int -> Int
factorial n = if n == 0 then 1 else n * factorial (n-1)

mygcd :: Int -> Int -> Int
mygcd x y = if y == 0 then x else mygcd y (mod x y)

```

从第一节递归计算的过程中可以看出，阶乘函数在未达到递归的基本条件前一直在展开，并不能进行任何计算，所有的中间结果将会在暂时存储在内存的栈中，越叠越多。这样的递归称为扩展递归（augmenting recursion），即在递归步中除调用递归函数以外还参与了其他的函数计算。显然，在阶乘函数中这个函数为(n^*)。而像最大公约数这样的函数，不必展开就可以计算，即除递归地调用以外没有任何其他的函数计算，递归调用的结果即为函数的结果，即便递归函数的参数进行了其他的计算，在计算过程中也可以不必展开。像这样的递归被称为尾递归（tail recursion），即在递归的时候不需要向内存中暂存任何的值。尾递归是递归的一种特殊情况。

扩展递归非常常见，这里来定义一个函数 total，求一个列表的和。Total 函数的功能与 Prelude 中的 sum 函数相同。

```

total [] = 0
total (x:xs) = x + total xs

```

假设，想求 1~5 的和：

```

total [1,2,3,4,5]
= 1+total [2,3,4,5]
= 1+(2+total [3,4,5])
= 1+(2+(3+total [4,5]))
= 1+(2+(3+(4+total [5])))
= 1+(2+(3+(4+(5+total []))))
= 1+(2+(3+(4+(5+0))))
= 1+(2+(3+(4+5)))
= 1+(2*(3+9))
= 1+(2*12)
= 1+14
= 15

```

在计算的时候，要暂时存储中间的值。1, 2, 3, 4, 5 这些中间的值在计算到达基本条件前没有任何的用处，假设计算是这样进行的，那么当计算 n 个数的和时，计算机则需要分配 n 个存储单元来临时存放这些值。显然，这样对于内存空间的使用效率是不高的。那么，有没有办法在递归的过程中不去分配这些临时的空间呢？可以将这些函数写成尾递归的形式。

```
total' [] n = n
total' (x:xs) n = total' xs (n+x)

total xs = total' xs 0
```

这里的 `total` 是通过另外一个递归函数来定义的，这样的定义就是尾递归定义。可以看到，利用尾递归可以将代码优化，省去了分配临时空间。下面来演绎一个扩展递归是如何被转换为尾递归的，假设已知的非尾递归 `total` 的定义为：

```
total [] = 0
total (x:xs) = total xs + x
```

我们定义一个函数 `total'`，这个函数同时做着加法和递归两件事情。此时，需要增加一个累积器（*accumulator*）来作为函数的参数。通常，累积器的类型与函数结果的类型相同，还需要找到带有累积器为参数的函数与之前不为尾递归的函数之间的关系。在这里，这个关系很容易找到，它是 `total' xs x = total xs + x`。

下面则可以通过讨论列表的两种形式，为空或者有一个以上元素两种情况进行计算得到对应的尾递归函数。

(1) 递归基本条件：

```
total' [] n
= total [] + n { total' 与 total 的等价关系}
= 0 + n {应用加法函数}
= n
```

(2) 递归步骤：

```
total' (x:xs) n
= (total xs + x) + n { total' 与 total 的等价关系}
= total xs + (x + n) { 加法交换律 }
= total' xs (x+n) { 递用 total' 与 total 的等价关系 }
```

根据上面的计算，得到两种情形的结果，可以定义出 `total'`：

```
total' [] n = n
total' (x:xs) n = total' xs (x+n)
```

这个过程实际类似于数学归纳法。这样，数学归纳法不仅可以用于证明，还可以用于递归程序的转换。有些函数的转换非常简单，但是随着学习的深入，有一些递归函数想转化成尾递归就没那么容易了。表面上，这样的优化本来可以使 Haskell 不需要使用更多的空间，可是，Haskell 是一个默认设置为惰性求值的语言，还是会产生的问题。虽然本书还没有对惰性求值进行讲解，但在这里还是有必要提及的。

由于惰性求值在用尾递归也可能会产生和扩展递归一样的问题，因此，在 `total'` 函数调用到递归基本条件前，参数 `n` 只参与和 `x` 的加法运算，而并不作为结果使用，即 `n` 的具体值在递归到达

基本条件前不需要被计算。因此，Haskell 还是会把这些值暂时存于内存中，等到需要的时候才计算。

```

total [1,2,3] 0
= total' [2,3] (1+0)
= total' [3] (2+(1+0))
= total' [] (3+(2+(1+0)))
= (3+(2+(1+0)))
= (3+(2+1))
= (3+3)
= 6

```

这样，需要使用!模式（bang pattern）匹配或者(\$!)运算符来强制 Haskell 对 `total'` 的第二个参数进行求值。`!`模式是强制在参数匹配前计算参数的值，而`($!)`则为在调用函数时计算参数的值。这些内容将在惰性求值一章中介绍。

```

total' [] n = n
total' (x:xs) n = total' xs $! (x+n)
--total' (x:xs) !n = total' xs (x+n)

total' [1,2,3] 0
= total' [2,3] (1+0)
= total' [2,3] 1
= total' [3] (2+1)
= total' [3] 3
= total' [] (3+3)
= total' [] 6
= 6

```

如果读者想要了解更多关于对函数程序推理演绎的内容，可以参考《Hutton, 2007》的第 13 章。

在不使用控制求值策略的运算符时，只有用模式匹配一个具体值或需要者返回结果时数据才会被计算。比如阶乘函数 `factorial`，当给定了一个参数`(5+2)`，Haskell 需要先计算`(5+2)`的结果，然后判断是否为 0 如果不为 0，那么匹配递归步。而 `total`却不一样，`total'`的第二个参数中没有定义具体的值，因此，默认情况下，在结果返回前不需要被计算。

5.4 互调递归

互调递归（mutual recursion）是一种特殊的情形，即两个函数的定义分别都用到了对方。比如，`even` 函数的定义用到了 `odd` 函数，`odd` 函数的定义也用到了 `even` 函数。

```

even 0 = True
even n = odd (n-1)

odd 0 = False
odd n = even (n-1)

```

比如，在计算 `even 3` 的时候：

```

even 3
= odd 2
= even 1
= odd 0
False

```

这种互调递归是间接递归的一种特例，比如，函数 1 调用函数 2，函数 2 调用函数 3，函数 3 再调用函数 1，如此地反复。在后面的章节中，可以见到很多这样的例子。

5.5 麦卡锡的 91 函数

麦卡锡的 91 函数由 Lisp 的发明者约翰·麦卡锡 (John McCarthy)^①引入。它的数学定义是这样的：

$$M(n) = \begin{cases} n - 10 & (n > 100) \\ M[M(n+11)] & (n \leq 100) \end{cases}$$

在 Haskell 中，91 函数的定义与数学定义几乎完全相同：

```

mc n | n > 100 = n - 10
      | otherwise = mc (mc (n+11))

```

为什么这个函数称为 91 函数？因为当 $n \leq 100$ 时，函数的结果均为 91。当 $n > 100$ 时，所有的结果为 $n - 10$ ，也就是说，它与下面的定义是相同的：

$$M(n) = \begin{cases} n - 10 & (n > 100) \\ 91 & (n \leq 100) \end{cases}$$

这是一个多么有趣的性质！这个函数是递归定义的，但是递归定义的部分的结果却是常数。下面讨论一下，如何通过数学归纳法对 91 函数的这个性质给予证明。

当 $90 \leq n < 101$ 时， $M(n)=M[M(n+11)]$ ，由于 $101 \leq n+11 \leq 111$ ，故 $M[M(n+11)]=M[(n+11)-10]=M(n+1)$ 。也就是说，当 $90 \leq n \leq 100$ 时，有 $M(n)=M(n+1)=91$ 。

前面已经证明了 $90 \leq n < 101$ 的情况，接下来，需要对 $89 \leq n$ 的情况，将 $n \leq 89$ 的整数每 11 项一组进行归纳。现假设，当 $a \leq n < a+11$ 时 $M(n)=91$ 成立，那么当 $a-11 \leq n < a$ 时， $M(n)=M[M(n+11)]$ 由归纳假设，有 $M(n+11)=91$ ，故 $M(n)=M(91)$ ，由数学归纳法的基本前提有 $M(91)=91$ 。

5.6 斐波那契数列

斐波那契数列由伟大的意大利数学家莱昂纳多·斐波那契 (Leonardo Pisano Fibonacci) 引入。

^① 美国计算机科学家，也就是在前言中提到的 Lisp 的创始人。Lisp 语言对其他很多语言的设计有着深远的影响。人工智能 AI (Artificial Intelligence) 一词由他引入并在这领域取得了卓著的成就，因此，他有着“人工智能之父”之称，在 1971 年获得图灵奖，于 2011 年 10 月 24 日在美国斯坦福逝世。

这是一个似乎具有魔幻色彩的数列，有很多惊奇的性质。比如，前一个数与后一个数的比值逼近黄金分割值 $\frac{\sqrt{5}-1}{2}$ ，4个相连的整数的内积与外积相差1等。在数学中，它是这样定义的：

$$F(n) = \begin{cases} 0 & (n=0) \\ 1 & (n=1) \\ F(n-1)+F(n-2) & (n>1) \end{cases}$$

简单地说就是，下一项等于前两项之和。这里，读者可以看到，递归的基本条件不只有一个。它的定义为：当 $n=0$ 的时候，结果为 0；当 $n=1$ 的时候，结果为 1；否则的话就为前两项之和。在 Haskell 中，用模式匹配很容易定义：

```
fibonacci :: (Num a) => a -> a
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

这种递归函数称为多重递归 (multiple recursion) 或者多分支递归。它是一种特殊的递归，即在一个函数中递归调用多次，在 $n>1$ 时，这个函数调用了自身两次，所以它是一个多重递归函数。通过模式匹配，可以非常容易地定义出来，几乎与其数学定义一样了。这里的函数的类型其实就是 `Integer -> Integer`。但是，因为之后要对这些数进行除法运算，为避免再进行类型转换，只要把计算的数字类型限定在 `Num` 就可以了，以下的函数都是如此。

```
>fibonacci 10
55
```

接下来，想办法在列表中生成斐波那契数列。这个很简单，对所有的自然数用 `fibonacci` 函数映射一次即可：

```
fibs n = map fibonacci [0..n]
>fibs 30
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,
,28657,46368,75025,121393,196418,317811,514229,832040]
```

这样定义非常的数学化，也非常通俗易懂，很具美感。但可以看到，当计算到第 30 项前后的时候就非常慢了。原因是每计算下一项的时候要重新进行递归，而且要递归两部分，一部分是 `fibonacci (n-1)`，另一部分是 `fibonacci (n-2)`。所以，如果我们想要取得更高效的计算方法，必须要做进一步的思考。

如果想要加快运算的速度，可以记录中间运算的结果。因为在计算 `fibonacci (n-1)` 的时候已经计算过 `fibonacci (n-2)` 了，所以没有必要再计算一次。这样，不但减少了计算量，同时也将递归的部分从两部分减少到了一部分。观察斐波那契数列：

```
1,1,2,3,5,8,13,21,34,55,...v,u,u+v...
```

可以用一个元组存储相邻的两个斐波那契数，这样就不必再重复计算了。所以，定义一个函数：

```
fibStep :: Num a => (a, a) -> (a, a)
fibStep (u, v) = (v, u+v)
```

这样，每应用一次 `fibStep` 函数，就向下计算一次。然后定义：

```
fibPair :: Num a => a -> (a, a)
fibPair 0 = (0, 1)
fibPair n = fibStep (fibPair (n-1))
```

这个函数需要给定一个任意的整数，通过 `fibStep` 与递归计算到所需要的元组；然后取第一个就可以了。最终的函数如下：

```
fastFib = fst $ fibPair
```

这样，把函数 `fibs` 改成了：

```
fibs n = map fastFib [1..n]
```

可以计算 100，甚至更大的斐波那契数，而且效率非常高。在 GHCI 里输入：

```
>:set +s
```

这样，可以让 GHCI 在运算后打印出当前计算机运算所需要的时间与内存容量。看一下用 `fibs` 计算前 200 个斐波那契数所需要的时间与内存。

```
>fibs 200
[1,1..... 173402521172797813159685037284371942044301, 280571172 9925101400376119
32413038677189525]
(0.12 secs, 7347708 bytes)
```

可以看出，足足用了 0.12s，而且 GHCI 用了 7MB 左右的内存（这里内存与时间消耗并不等于编译后程序的运行时间，使用 GHC 编译后的运行要远远快于 GHCI 解释运行的速度，并且内存使用也会大幅度减少）。可以让程序更快一些吗？答案是：可以。因为在求第 n 项的时候，前 $n-1$ 项都不必计算了。在库中有一个 `iterate` 函数，通过对函数进行迭代做到这一点。这个函数总会使用前一个元素作为参数生成下一个元素，最后生成一个无穷长的列表。这里，只需要对 `fibStep` 进行迭代，取得二元组的第一个元件，然后取这个无穷长的列表的前 n 项即可。具体如下：

```
fibs' n = take n (map fst (iterate fibStep (0,1)))
```

然后运行：

```
>fibs' 200
[1,1.....]
(0.03 secs, 3151696 bytes)
```

所需计算时间仅仅为 30 ms，并且内存使用效率也提高了一倍左右。同样，也可以用尾调用

优化，将相邻的两个斐波那契数通过参数来暂存。读者可用 5.3 节中的方法推导演绎出函数的优化定义，即加入一个积累器作为参数。例如：

```
fib 0 f1 f2 = f2
fib n f1 f2 = fib (n-1) f2 (f1+f2)

fibonacci n = fib n 1 1
```

从以上的例子可以看出，只要正确使用，递归一样可以很高效。

接下来，随着这个神奇的数列的增长，观察相邻两项的比值是否趋近于黄金分割。基于 `fibs'`，只需要把 `map` 映射中的 `fst` 换成求元素中两个数字元件的商即可。

```
golden :: Fractional a => Int -> [a]
golden n = take n (map (\(x,y) -> x/y) (iterate fibStep (0,1)))
```

运行一下前 30 项的结果，可以看出，这组数的确是在逼近黄金分割的。

```
>golden 30
[0.0, 1.0, 0.5, 0.6666666666666666, 0.6, 0.625, ..., 0.6180339887482036, 0.6180339887505408]
```

用 Haskell 来粗略计算一下黄金分割的值：

```
>(sqrt 5 - 1)/2
0.6180339887498949
```

从列表中可以看出，随着这个数列的增长，相邻的两项的比值的确是在逼近黄金分割的。

接下来，我们来验证一下斐波那契数列的另一个性质，连续 3 项中，中间项平方与外两项之积是差 1 的。

$$F_{n-1}F_{n+1} - F_n^2 = (-1)^n$$

首先，要得到连续的 3 项，只要将 `fibPair` 生成的数据合成一个三元素的元组后，递归地组合后面的元组即可。但是，若不足两个元组，则返回空列表：

```
combine :: [(a,a)] -> [(a,a,a)]
combine ((f1,f2):(f3,f4):fs) = (f1,f2,f4):combine ((f3,f4):fs)
combine _ = []
```

再来定义生成连续二元元组列表的函数，只需要在 `fibs` 中映射 `fibPair` 即可。将这个函数定义为 `fibPairs`：

```
fibPairs :: Int -> [(Int,Int)]
fibPairs n = map fibPair [1..n]
```

重载文件，并在提示符里运行：

```
>combine $ fibPairs 7
[(1,1,2),(1,2,3),(2,3,5),(3,5,8),(5,8,13),(8,13,21)]
```

用 map 函数对三元组进行相应的运算，有了 λ 表达式映射的函数很容易就可以定义：

```
difference :: Int -> [Int]
difference n = map (\(f1,f2,f3)->f1*f3-f2*f2) (combine $ fibPairs n)
> difference 10
[1,-1,1,-1,1,-1,1,-1,1]
```

这个数列还有一些其他的性质，有兴趣的读者可以自行查阅一些相关资料。计算某一项斐波那契数可以更快，但是要通过矩阵乘法。如何表示矩阵，以及如何计算将在下一章中讨论。

5.7 十进制数字转成罗马数字

本节里，继续讨论递归函数。希望本书中的实例足够让读者适应递归这种思想。下面举一个使用递归把十进制数转换成为罗马数字的例子。罗马数字像是一种很有趣的五进制，说是五进制，但还不准确。在罗马数字中，I 为 1，V 为 5，X 为 10，L 为 50，C 为 100，但是 4、9、40、90 分别用 IV、IX、XL、XC 来表示，将小一级的罗马数字放在左边表示减法，它们需要特殊处理。1~10 的罗马数字如下：

I, II, III, IV, V, VI, VII, VIII, IX, X

这里只考虑 5000 以内的罗马数字。先来定义两个列表，分别对应罗马数字和与之对应的十进制数。

```
romeNotation :: [String]
romeNotation = ["N", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"]

romeAmount :: [Int]
romeAmount = [1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1]
```

使用 zip 函数将它们结合在一起，以便转换之用。

```
pair :: [(Int, String)]
pair = zip romeAmount romeNotation
```

给定一个十制数，只需要找到第一个不大于它的罗马数字，然后从十进制数字中减去，再将剩下的十进制数递归地转换成罗马数字即可。

先来实现一个函数，它将从 pair 列表中取得第一个比这个十进制小的数字，通过 dropWhile 将 pair 中比给定十进制数字大的元组去掉，再取得列表第一个元素。这样就得到了对应的罗马数字：

```
subtrahend :: Int -> (Int, String)
subtrahend n = head (dropWhile (\(a,_) -> a > n) pair)

> subtrahend 5
(5,"V")
```

```
> subtrahend 86
(50,"L")
```

下面定义函数 convert 来转换这两种数制，首先定义递归的基本条件。如果转换的数字是 0，那么返回空列表，因为罗马数字中没有表示 0 的符号，只需要返回 (0, "") 即可。0 在数字中其实是一个非常抽象的概念。在当时，也许罗马人也不知道用什么来表示 0，这里用的空字符串。下面再定义递归函数，使用 subtrahend 得到了减数，得到了对应的罗马数字 st，再递归地调用 convert 函数转换余下的十进制数，即 convert (n-i)，最后返回未转换的部分和两个罗马数字字符串连接：

```
convert :: Int -> (Int, String)
convert 0 = (0, "")
convert n = let (i,st) = subtrahend n in
            let (i',st') = convert (n-i) in (i',st++st')

> convert 12
(0,"XII")

> convert 109
(0,"CIX")

> convert 1925
(0,"MCMXXV")

> convert 4567
(0,"MMMMDLXVII")
```

练习

- 参照本章前面展示的那样，试着用等式推导的方法来演算 convert 17 的计算过程。
- 定义十进制转罗马数字函数的反函数，即将一个 5000 以内的罗马数字转换为一个十进制数字。
- 定义一个函数，将十进制转换为二进制，结果可以用只含有 1 与 0 的字符串来表示。

5.8 二分法查找

二分法查找 (binary search) 是一种常用的搜索方法，基于一个有序的结构。每一次搜索中间的元素，如果搜到自然好，若搜不到，将搜索的元素与中间元素进行比较，不论比较的结果如何，都可以减少一半的搜索量。在查字典的时候，不经意间就可能用到了这个方法。这种搜索算法非常简单，早在 1947 年就被提出了，但是第一个正确的、无错误的代码直到 1961 年才被发表。这说明了用顺序式语言很难直观地、一次性地写出毋庸置疑的，且没有错误的代码。相反，函数式编程在这一方面表现的尤为出色。

先来看递归的基本步骤，若所搜的列表为空，那么此元素必不在该列表内，只需要返回假即可。假定中间的元素为 m，列表为升序排列，若要搜的元素 a 小于 m，那么说明如果 a 在这个列表中，它只可能在 m 的左侧，需要继续对 m 左侧的列表进行递归地搜索；若大于 m，则说

明只可能在 m 右侧的部分，则需要递归地搜索右侧的部分。使用已有的 `split` 和 `length` 函数可以很容易地将这个列表分开。

```
search :: (Ord a) -> a -> [a] -> Bool
search a [] = False
search a xs | m < a = search a behind
            | m > a = search a front
            | otherwise = True
where (front,m:behind) = splitAt (length xs `div` 2) xs
```

使用强数学归纳法可以很容易地证明程序是正确的。当列表为空时，元素必定不在该列表内，所以这个基本条件是正确的。

现在，假设 `search a xs`，对于所有长度不大于 k 的列表 xs 都是正确的，下面证明 `search a (x:xs)` 也是正确的。

对于中间的元素 m 与所查找的元素间必有三种关系，即小于、大于、等于。

- 当 $a = m$ 时，则元素在此列表之中，返回 `True`。
- 当 $a < m$ 时，由于列表是有序的，因此搜索 m 左侧的元素则足以说明 a 是否在这个列表中。由归纳假设可知，`front` 的长度 $\leq k$ ，故 `search a front` 是正确的，因此，在该种情况下，这个函数也是正确的。
- 当 $a > m$ 时，应用归纳假设，则说明这个函数也是正确的。那么综上所述，这个二分法查找函数是正确的。

用这种归纳方法的出发点：任意一个列表都可以分段，成为由一个中间元素和左右两个列表组合而成的列表，且列表 $(x:xs)$ 的中间元素 m 左右两边的列表的长度至少要比 $(x:xs)$ 的长度小 1，从而保证了强数学归纳法的有效性。

在第 11 章中，写一个简单的字典程序，到时候会用到这个算法。

练习

这里的二分法查找仅仅是返回元素是否存在，能不能重新定义一个二分法查找函数，给定一个元素，如果该元素不在这个有序的列表中，那么返回 `[]`；如果在，那么返回一个包括这个元素的列表。注意：这个元素在列表中可能有多个，需要将它们全部返回，例如：

```
>search 4 [1,2,2,3,4,4,4,5,6,7,8]
[4,4,4]
```

5.9 汉诺塔

相信很多人对汉诺塔（Tower of Hanoi）问题已经非常了解了。但是，这里还是想讲述一下关于它的具有神话色彩的故事。在印度有一个古老的传说：在贝娜勒斯的圣庙里有一块黄铜板，

上面插着三根宝石针。当印度教的主神——梵天在创造世界时，在其中一根针上从下至上穿好了由大到小的 64 片金片，这就是所谓的汉诺塔。不论白天黑夜，总有一个僧侣在按照“一次只移动一片，不管在哪根针上，小片必须在大片上面”这样的规则移动这些金片。僧侣们预言，当所有的金片都从梵天穿好的那根针上移到另外一根针上时，世界就将在一声霹雳中毁灭，而梵塔、庙宇和众生也都将同归于尽。

这是一个非常好的递归问题。假设有 n 个金片的时候要移动 $h(n)$ 次。当有一个金片的时候，直接移动就好了，需要 $h(1)=1$ 次。当有 n 个金片的时候，需要将 $n-1$ 个金片移动到非目标针上，需要 $h(n-1)$ 次。至于怎么移动的不需要知道，然后将最底下的金片移动到目标针上。最后，再将非目标针上的 $n-1$ 个金片同样用那个不需要知道的方法移动到目标针上即可，此时也需要 $h(n-1)$ 次，移动过程如图 5-1 所示。这样，所需要移动的总次数为：

$$h(n)=h(n-1)+1+h(n-1)=2h(n-1)+1。$$

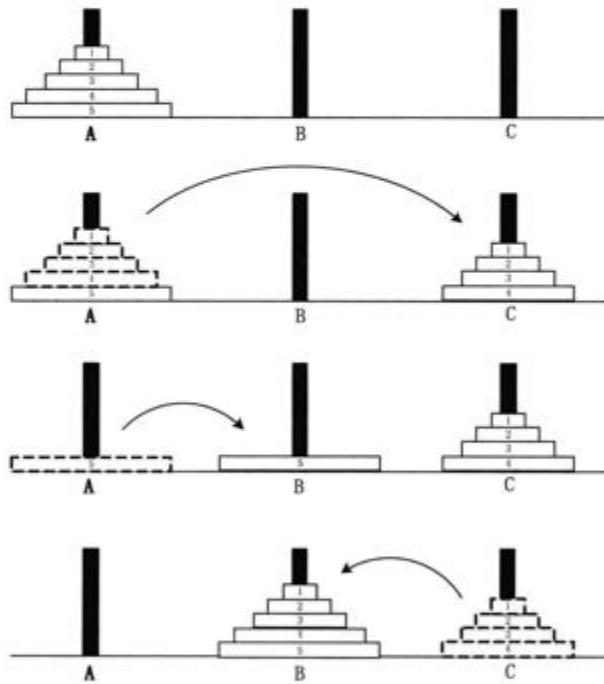


图 5-1 汉诺塔递归移动示意图

所以，当有 5 个金片的时候，移动次数的递归函数可以想像为先将上边的 4 片移动到 C 针上，再将第 5 片移动到 B 针上，最后，将 C 针上的 4 片移动到 B 上：

$$h(n) = \begin{cases} 1 & (n=1) \\ 2h(n-1)+1 & (n>1) \end{cases}$$

这个数列可以用一阶线性递归数列或者迭代来求得通项公式。其实，通过观察数列也可以知道数列 1,3,7,15…，通项公式为 $h(n)=2^n-1$ 。在 Haskell 中，可以像定义阶乘函数那样定义这个函数：

```
h 1 = 1
h n = 2 * h (n-1) + 1
```

也就是说，需要移动这些金片 $2^{64}-1$ 次才能让世界毁灭，并且这是最少的次数了。因为无论何时，想移动最底下的金片，总是要将的 $n-1$ 个移动到非目标针上去。这样，总是需要 $h(n-1)$ 次才行。移动的次数被递归地保证了最少，也就是说这就是最少的移动次数了。这个数字可以用 Haskell 计算一下，方法如下（或用 h 64）：

```
>2^64-1
```

需要 18446744073709551615 次。

现在来假设那位僧侣一秒可以移动一次，也就是说，要 18 446 744 073 709 551 615 秒才能完成。那么，这大约是多长时间呢？若不考虑闰年，一年有 $365\times 24\times 60\times 60$ 秒，计算一下需要多少年：

```
>div 18446744073709551615 (365*24*60*60)
584942417355
```

这样，通过非常精确的数学计算，终于可以让自己放心了，因为至少需要 5800 亿年以上才能移动完这些金片，而宇宙到现在为止的寿命才五百亿年左右。看来，如果按照这个预言，宇宙的寿命将会相当的长。从浩瀚宇宙的遐想中回来，想一想：具体怎么定义移动这些金片的函数呢？

创建名为 Hanoi.hs 文件编写代码。现在，假设那位僧侣要从第 1 根针上移动到第 2 根针上。然后，需要的函数是生成一个元组的列表记录移动的一系列的过程，如 $[(1, 2), (1, 3) \dots (3, 2)]$ ，即从 1 移动到 2，从 1 移动到 3……最后，从 3 移动到 2，结束。最终，需要一个类型为 $\text{hanoi} :: \text{Int} \rightarrow [(\text{Int}, \text{Int})]$ 的函数。

这个函数只是需要一个金片的个数，返回给用户移动的方法，而并不在乎怎么移动这个函数。另外，还需要一个辅助的函数 move：

```
move :: (Int, Int, Int, Int) -> [((Int, Int))]
```

这个函数的 4 个参数分别代表如下：

- 要移动多少个金片；
- 从哪里移动；
- 移动到哪里；
- 移动过程中可以借助哪根针。

当有一个金片的时候，从原始位置 from 移动到目标针 to 即可。

```
move (1, from, to, via) = [(from,to)]
```

当有多个金片的时候，按照前边所解释的方法，只需要把 $n-1$ 个金片从原位借助 to 针移动到 via 针上，via 为非目标针。然后，将最下边最大的金片的从原始位置 from 移动到目标针 to，最后，将原来移在 via 针上的 $n-1$ 个金片借助 from 针移到目标针 to 上即可，如下：

```
move (n, from, to, via) = move (n-1, from, via, to) ++ [(from, to)] ++
    move (n-1, via, to, from)
```

move 递归函数与递归公式 $h(n)=h(n-1)+1+h(n-1)$ 符合完好。由此，如何移动汉诺塔的函数为：

```
hanoi n = move (n, 1, 2, 3)
```

这里可以看到，解决递归问题的时候，并不必特别在乎是怎么移动 $n-1$ 个金片的，只需要找到基本条件和递推关系即可。

5.10 排序算法

排序问题一直是计算编程算法入门的必学问题。为什么排序如此重要？原因有很多。其中，重要的原因就是排序可以优化搜索。比如，在查字典的时候，字典里的字是按照字母顺序排序的，这样可以使人们更快地找到想要的字词，倘若里边的字词是胡乱堆砌的，那么每查一个陌生的字词可能就要翻遍一整部字典了。计算机中的英文字典词库就是同样的例子，单词只有按照一定的顺序排列，才会使查询更快；比如，银行的数据库中用户非常多，要在数据库里对姓名、卡号进行快速有效地搜索；再比如，手机内的联系人大多也是按照姓氏的拼音在字母表中的顺序排列。在操作系统中，也常常要对文件进行大小、时间、文件名、类型排序。计算机每天都不停地插入新的条目、排序、搜索，用户经常需要排序与搜索，这就是 Google、必应、百度等搜索引擎在日常生活中非常重要的原因。学会了排序，才算是进入了计算机算法的大门。下面，借助扑克牌的示例来探讨一些常用的排序算法。

5.10.1 插入排序

插入排序（insertion sort）是一种简单易懂的排序方法。在玩扑克牌时候，可能会在不知不觉中用过这个方法，下边的几张扑克牌是刚刚摸到时的顺序：



排序的过程是这样的，当摸到 7 时，不需要做任何的比较，直接拿在手里，即手中的牌是这样的：



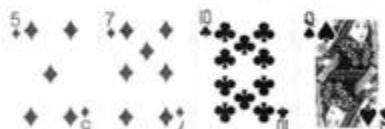
而当摸到 5 时，因为 5 比 7 小，所以将 5 放在 7 的左边。



当摸到 Q 时，比较一下，Q 比 5 大，所以应该是 5 的后面，而 5 后面是 7，要再比较一下 7 和 Q，Q 比 7 大，再向后比较，最后，7 后边没有牌了，所以 Q 只能放在最后了。



最后，当摸到 10 的时候，用同样的方法——10 比 5 大，10 比 7 大，10 比 Q 小，所以插入 Q 和 7 的中间。于是顺序就这样排好了。



现在，对于给定的任意列表，若元素类型限制为有序的，就可以用这种方法对它进行从小到大的排序。先来写一个插入的函数，即，将某一张牌插入到一个已经排序好的一叠牌里，就像插入 Q 时，5 和 7 已经是有序的牌中那样，插入 10 时，5、7、Q 也是有序的了。这样，无论何时插入牌进去，手里的牌一直保持有序，这就是要利用的性质。函数很容易定义，当列表为空的时候，相当于手里没有牌，直接拿就可以了。当手中有了一些有序的牌后，在插入新牌时，若它比当前比较的牌还小，放在前边就好，若比当前比较的牌大，只需要递归地将新牌插入到后边的牌里。这样，这个递归函数在插入的时候，会仍将新牌与手中未比较过的牌从左到右地依次比较，直到找到比它大的牌，插入在前边，或者它为最大，放在最后。这样最后没牌，即列表为空为止。

```

insert :: Ord a => a -> [a] -> [a]
insert x []      = [x]                      --(0)
insert x (y:ys) | x < y = x:y:ys  --(1)
                | otherwise = y:insert x ys --(2)

```

根据函数的定义，可以对这个算法演绎一下最后的步骤，将 10 插入到 [5, 7, 12] 的时候。

```

insert 10 5:[7,12]
{ 10>5 由函数中的(2)有 insert 10 5:[7,12] = 5:insert 10 7:[12] }
= 5:insert 10 7:[12]
{ 10>7 由函数中的(2)有 insert 10 7:[12] = insert 10 12:[] }
= 5:7:insert 10 12:[] 
{ 10<12 由函数中的(1)有 insert 10 12:[] = 10:12:[] }
= 5:7:10:12:[] 
(使用(:)函数有)
= [5,7,10,12]

```

现在得到了插入一张牌到一沓有序牌的函数，但怎样得到排序的函数呢？很容易，只需要将乱序的元素一个一个地使用 `insert` 函数放到另一个有序列表里就可以了。这个新的列表就是所要的结果了。这里用 `xs` 表示新的列表，用 `ys` 表示要排序的列表。当需要排的列表为空的时候，没有东西再需要排了；否则，将乱序列表中的第一个元素插入到有序的列表中。代码如下：

```

insertionSort :: Ord a => [a] -> [a] -> [a]
insertionSort xs []      = xs                      --(3)
insertionSort xs (y:ys) = insertionSort (insert y xs) ys --(4)

```

这样，在用这个排序方法的时候，在提示符中输入：`insertionSort [] [7,5,12,10]`。下面，对这种方法进行演绎：

```

insertionSort [] [7,5,12,10]
= insertionSort (insert 7 []) [5,12,10]
= insertionSort [7] [5,12,10]
= insertionSort (insert 5 [7]) [12,10]
= insertionSort [5,7] [12,10]
= insertionSort (insert 12 [5,7]) [10]
= insertionSort (5:insert 12 [7]) [10]
= insertionSort (5:7:insert 12 []) [10]
= insertionSort [5,7,12] [10]
= insertionSort (insert 10 [5,7,12]) [] (前边已演绎这步了，就不再详写了)
= insertionSort [5,7,10,12] []
= [5,7,10,12]

```

实际上，可以不需要中间那个列表的帮助，直接得到结果。即：当所排的列表为空的时候，返回空就好；当所排列表有多个元素时，只需要将第一个元素插入后边用这个算法排好的列表中即可。这里可以看到本例如何使用递归的。假定要排的列表为 `(x: xs)`，只需要将 `x` 插入到用这个算法排序好的 `xs` 中，即 `insert x (insertSort xs)`，这就是对这个算法使用递归的过程。函数如下：

```
insertionSort :: Ord a => [a] -> [a]
insertionSort []      = []
insertionSort (x:xs) = insert x (insertionSort xs)
```

下面，对这个函数的计算过程进行推导：

```
insertionSort [7,5,12,10]
= insert 7 (insertionSort [5,12,10])
= insert 7 (insert 5 (insertionSort [12,10]))
= insert 7 (insert 5 (insert 12 (insertionSort [10])))
= insert 7 (insert 5 (insert 12 (insert 10 (insertionSort []))))
= insert 7 (insert 5 (insert 12 (insert 10 (insert 10 []))))
= insert 7 (insert 5 (insert 12 [10]))
= insert 7 (insert 5 [10,12])
= insert 7 [5,10,12]
= [5,,7,10,12]
```

这两种插入的方法用的是完全一样的，只不过是第一个用了中间列表来存储，是尾递归，而后一种是扩展递归。

5.10.2 冒泡排序

冒泡排序（bubble sort）也是一个非常经典的排序方法。现在，延续之前的纸牌的示例。通过不断地将较大的牌向右交换，重复多次就可以得到一沓有序的牌，每运行到结尾，最后一张牌总是最大的，这个最大的被称为“泡”。比如：



以上图的牌为例，比较 7 与 Q。因为 7 比 Q 要小故不需要交换，即保留上图的顺序。



比较 Q 与 3。由于 3 是小于 Q 的，因此要交换，如下：



接下来，比较 Q 与 5。显然 Q 比 5 大，故交换它们。



最后，比较 Q 与 10。还是 Q 大，再进行一次交换，得到：



这样，就把最大的 Q 当成“泡”给冒出来了。之后，再用同样的方法去排前四张牌，依次类推。当排前一张的时候，算法结束。先来定义交换牌的函数。递归的基本情况很容易，当没有牌或者当只有一张牌的时候，没法进行交换，也不需要进行比较，所以就是它们本身了。当有两个元素以上的时候，就可以对两个元素进行比较与决定是否交换了。很简单，当前一个元素要小时，用这个函数交换后边的列表即可；当前一个元素要大一些的时候，只需要将其和后面较小的元素交换后，再递归应用交换函数即可。

```
swaps :: Ord a => [a] -> [a]
swaps [] = []
swaps [x] = [x]
swaps (x1:x2:xs) | x1 > x2 = x2: swaps (x1:xs)
                  | otherwise = x1: swaps (x2:xs)
```

在提示符里测试一下：

```
>swaps [7,12,3,5,10]
[7,3,5,10,12]
```

可以看到，12 作为一个最大的数，被交换到最后了。只要迭代这个函数多次，直到这个列表不变即可。

来定义一个不动点函数。这个函数可以一直调用 swaps，一直到列表不再发生变化为止：

```
fix :: Eq a => (a -> a) -> a -> a
fix f x = if x==x' then x else fix f x'
           where x' = f x
```

而冒泡排序不过是一个反复调用 swaps 的函数，所以有：

```
bubbleSort :: Ord => [a] -> [a]
bubbleSort xs = fix swaps xs
```

来测试一下：

```
>bubbleSort [7,12,3,5,10]
[3,5,7,10,12]
```

可以把 fix 隐含在 bubbleSort 里，则有：

```
bubbleSort' :: Ord a => [a] -> [a]
bubbleSort' xs | swaps xs == xs = xs
               | otherwise = bubbleSort' $ swap xs
```

可以看出，这里用的其实就是不动点函数的函数。在函数式编程里，这种函数有很多。比如，要重复化简一组数据，但是化简之后的数据还可以被继续化简，这一类的问题就可以交给 fix 函数来处理了，实质上不动点函数是可以代替顺序式语言中的循环的。

但是，这样做的效率并不高。既然最大的已经被交换到最右边了，就没有必要再使用交换函数了。因此，可以将交换完毕的列表分成前 $n-1$ 项与最后一项，最后一项可以不用动了，再递归调用 bubbleSort'' 来对前 $n-1$ 项排序。按照这个思路写出的函数如下：

```
bubbleSort'' :: Ord a => [a] -> [a]
bubbleSort'' [] = []
bubbleSort'' xs = bubbleSort'' initialElements ++ [lastElement]
  where swappedxs = swaps xs
        initialElements = init swappedxs
        lastElement = last swappedxs
```

有兴趣的读者可以尝试一下长一些的列表，比较一下时间效率，哪一个会低些，并想想为什么。

5.10.3 选择排序

选择排序（selection sort）是一种简单直观的排序方法。首先找到最小的元素，将其从列表中取出，放于另一个列表的第一个位置，依次类推，直到所有的元素全被取光。



搜索一次，3 最小，有：



再搜索余下的牌，5 最小，有：



依此类推，这样就可以得到一个有序的列表了。这里，从列表里选出最小的，可以使用库里的 `minimum` 函数实现找到最小的元素，放于数列最前，再将其从原列表中删除之后再找出最小的元素，直到列表为空，这个算法就结束了。先来写删除函数。

```
delete :: Eq a => a -> [a] -> [a]
delete _ [] = []
delete x (l:ls) | x == l = ls
| otherwise = l:delete x ls
```

这样，就可以将排序函数写出来了：

```
selectionSort []      = []
selectInSort xs = mini : selectionSort xs'
  where mini = minimum xs
        xs' = delete mini xs

>selectionSort [7,12,3,5,10]
[3,5,7,10,12]
```

这里，总是保证 `selectInSort` 函数会取得最小值，放于最前，然后递归应用到除去那个最小元素的列表。同理，可以对这个函数进行演绎：

```
selectionSort [7,12,3,5,10]
= 3: selectionSort [7,12,5,10]
= 3: 5: selectionSort [7,12,10]
= 3: 5: 7: selectionSort [12,10]
= 3: 5: 7: 10: selectionSort [12]
= 3: 5: 7: 10: 12: selectionSort []
= 3: 5: 7: 10: 12: []
= [3,5,7,10,12]
```

5.10.4 快速排序

快速排序（quick sort）由有计算机领域爵士之称的英国计算机科学家托尼·霍尔（Tony Hoare）于 1960 年引入的。它的过程非常简单。延续之前纸牌的例子，快速排序的基本思路是：取出一张牌，作为一个分界点，然后筛选出比其小的作为一组。比其大的作为一组，分别再用快速排序算法递归地对这两组进行排序后，把分界点与排序好的两组牌进行组合即可。比如，要对下列纸牌进行排序。



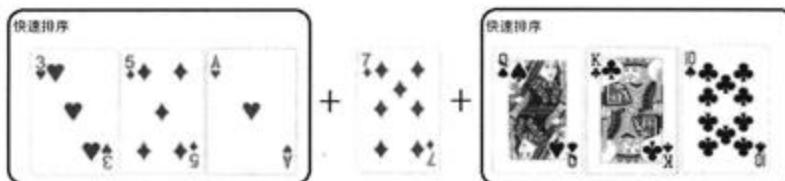
首先，选出 7 作为分界点，于是有：



然后，将余下的牌分成两组，一组比 7 小，一组比 7 大，于是有：



将两边的牌递归地调用算法排序，然后连接这三组牌，最终，会得到一叠有序的牌。



代码如下：

```
quickSort :: Ord a => [a] -> [a]
quickSort [] = []
quickSort (x:xs) = quickSort mini ++ [x] ++ quickSort maxi
  where mini = filter (<x) xs
        maxi = filter (≥x) xs

>quickSort [7,12,3,13,5,10,1]
[1,3,5,7,10,12,13]
```

可以看出，虽然这个函数一目了然，非常简单易懂，但有两个问题。其一，产生了很多的 `[]`，这个非常好解决，只要加一条 `quickSort [x] = [x]` 即可。其二，当筛选出比 `x` 小的元素时，要将列表遍历一次，筛选出大于等于 `x` 的元素时，又要遍历一次。因此，可以定义一个函数，当遍历列表时，一次性地将其分好组，然后被 `quickSort` 函数所用。这个函数需要一个条件函数来得知如何筛选，然后需要一个列表进行筛选，把结果的两部分当成一个元组返回。

下面，来定义这个函数：

```
filterSplit :: (a -> Bool) -> [a] -> ([a],[a])
filterSplit _ [] = ([],[])
filterSplit f (x:xs) | f x = ((x:[]),xs)
                     | otherwise = (l,(x:r))
where (l,r) = filterSplit f xs
```

这个函数会把成立的数放在左边，不成立的放在右边。这里，在用 where 做替换的时候，Haskell 会把 l 与 r 匹配到递归调用 filterSplit 函数的结果上。下面，重新定义 quickSort：

```
quicksort' :: Ord a => [a] -> [a]
quicksort' [] = []
quicksort' [x] = [x]
quicksort' (x:xs) = quicksort' l ++ [x] ++ quicksort' r
where (l,r) = filterSplit (<x) xs
```

5.10.5 归并排序

归并排序（merge sort）亦作合并排序或融合排序，由伟大的计算机之父约翰·冯·诺依曼（John von Neumann）于 1945 年引入。那时，第一台电子计算机还未正式运行，冯·诺依曼就能有这样超前的思想实在令人钦佩！

这个算法比起前几个排序算法略显复杂，诺依曼采用的其实是分治法（divide and conquer）的思想。延续之前的纸牌的示例，算法的过程大约是：先将牌均匀地分成两部分，将这两部分分别用归并排序算法排序，然后用一个函数将这两叠有序的牌合并为一叠有序的牌。那么，当用归并排序排那两部分的时候，实际上是又将这两部分牌继续均分，然后递归地使用这个排序方法，递归的基本条件是一张牌自己是有序的。比如，有一把纸牌：



将其均匀分成两组：



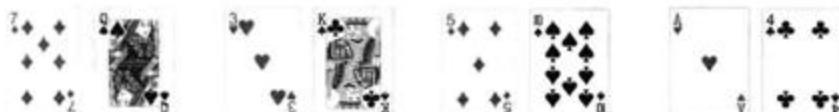
应用 mergeSort 的牌就可以当成是已经有序的了，想办法合并两组已经有序的牌。



依照算法再继续分:



最后分成一张一组:



由于每组只有一个元素，因此不用比较就排好序了，这就是递归的基本条件。第一次合并比较一次，后面直接按从小到大合并即可。



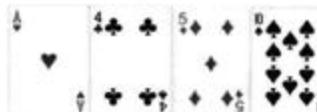
每一组合并时，比较 7 与 3 的大小，显然 3 小，将 3 取出；第二组比较 5 和 A 的大小，显然 A 小，将 A 取出，于是有：



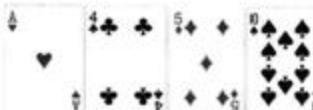
接着，比较 7 与 K 的大小，比较 5 与 4 的大小，有：



再比较 Q 与 K 的大小。由于后面这一组已经为空，因此直接放于 A 和 4 的后边即可：

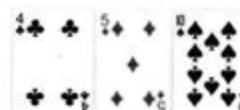


最后，也只余下一组了，把 K 放到 Q 的后边就完成了前一组的合并。



之后，按照同样的方法来合并这两组就可以了。合并两组有序的牌，最后得到的依旧是一组有序的牌。

1 比 3 小，有：



3 比 4 小，有：



4 比 7 小，有：



5 比 7 小，有：



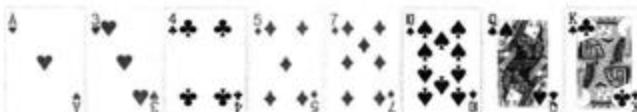
7 比 10 小，有：



10 比 Q 小，有：



另一组已经为空，Q与K已经是有序的了，所以直接放到最后即可，有：



这样，就完成了归并排序。下面来定义 merge 函数，它把两个有序的列表融合在一起。当有一个列表为空的时候，因为假定另一个列表是有序的了，所以直接返回另一个列表即可。然后，如果两个列表中都有元素，只需要比较第一个即可，因为两个列表都是有序的。这样，当有两个列表，并且列表中的元素是有序的时，应用 merge 函数后所得结果依旧是有序的。

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) | x > y = y:merge (x:xs) ys
| otherwise = x:merge xs (y:ys)
```

归并排序函数 msort 可将一列数分成两部分，然后递归地将它们排好序后，再使用 merge 函数将其合并在一起。

```
msort :: Ord a => [a] -> [a]
msort xs = merge (msort x1) (msort x2)
  where (x1,x2) = halve xs
        halve xs = (take l xs, drop l xs)
        l = (length xs) `div` 2
```

读者可以在 GHCi 中给出一些列表，比如 [6,4,7,9,1,4] 或字符串 "Haskell is fun"，来对它们进行排序。

练习

1. 上网查询一下 C 或者 Java 语言是如何实现这些排序的，比较这些语言与 Haskell 的不同。
2. 查询一下 Haskell API 的 Ord 和 List 库，在 GHCi 中使用 :m 导入它们并且使用一下 min、sort、sortBy、minimumBy 等函数，阅读一下它们的代码。

```
>:m +Data.List  
> sort [4,6,7,1]
```

小结

本章所讨论的很多问题，如斐波那契数列、汉诺塔、快速排序与归并排序，都有一个共同点，那就是在用定义时有很多递归调用的分支，比如：

- 斐波那契数列中的 `fibonacci n = fibonacci (n-1) + fibonacci (n-2);`
- 汉诺塔中的 `move (...) = move (...) ++ [(from, to)] ++ move (...);`
- 快速排序中的 `quickSort' (x:xs) = quickSort' l ++ [x] ++ quickSort' r;`
- 归并排序中的 `msort xs = merge (msort x1) (msort x2).`

这样的例子还有很多。这样的递归称为多分支递归（multi-branched recursion）。在解决问题时，像这样不断地将问题递归地转换成多个小问题的方法称为分治算法（divide and conquer algorithm），它是算法中的一种重要思想。更多的例子可以参阅相关的算法书籍。

通过解决上述有意思的算法问题，学过 C 语言的读者可能会感觉到函数式编程在解决这些问题上的威力了。解决斐波那契数列时，看到了同数学表达式几乎一样的函数定义，只需要进一步思考，就可以进一步提升生成斐波那契数列的速度；在汉诺塔问题里，解决这个问题实际上只需要两行非常易读的代码；解决排序问题的时候，马上能理解到算法的核心部分。

函数式编程语言是一类非常高级的语言，因为分配与回收内存、移动指针、声明变量等在这里几乎可以全部忽略，并且基于列表这种递归结构的容器来编程的自由度也很大。下一章里，会深入学习列表的使用。

5.11 递归基本条件与程序终止

有这样一类递归函数，无法确定递归基本条件一定可到达。比如：对于任意大于等于 1 的整数 n ，当 $n=1$ 时返回 1，结束程序，若 n 为偶数，则 $n=n/2$ ，否则 $n=3n+1$ 。到现在为止，对于这个程序，人们还无法证明它是否一定可以到达递归的基本条件，但人们试了很多数，这个程序是会到达递归的基本条件的，程序如下：

```
halt :: Integral a => a -> [a]
halt 1 = [1]
halt n | even n    = let n' = div n 2 in n':halt n'
        | otherwise = let n' = 3*n+1 in n':halt n'
```

这一类递归函数略有些不同。虽然不能很容易地直接看出或者证明它会停下来，但是它们总被认为是会停下来，可惜很难给出证明。另外，还有一些函数是没有停止条件的，它们会永远运算下去。这样的例子有很多，在以后的讨论中将会看到这一类函数，比如，在查找大于

n 的素数时，可以定义一个函数，检查 *n*+1 是否为素数，再检查 *n*+2 是否为素数等。但是，这个程序不会停止，为什么呢？因为自然数有无穷多个。理论上讲，给定一个自然数，总是可以找到比它大的素数，但是这是需要数学证明的。现在，读者可以知道，递归是否会到达基本条件然后停止，对于程序来说至关重要。

5.12 递归与不动点

之前在冒泡排序中提到了 `fix` 函数，它就是不动点函数（fixed point function）。不动点理论由荷兰数学家 Brouwer 引入。函数的不动点意为当参数应用到这个函数时，结果是这个参数本身。由于函数式编程是以函数为基础的，因此可以直接、自然地能引入它。关于本节的内容，也许有些读者还没有办法完全理解，可以先浏览一下，然后在学习高阶函数后再回来阅读。

虽然函数式编程基于 λ 演算，但 λ 演算中所定义的函数不能像 Haskell 那样简单地通过调用自身来定义递归函数。为了在 λ 演算中定义递归函数，就有人引入了不动点这一概念。函数式编程是以 λ 演算为理论基础的，而 λ 演算又使用了不动点来表达递归，所以说语义上，Haskell 中的递归是通过不动点函数来实现的。虽然在使用与定义递归函数时，我们并不使用不动点函数，但是 λ 演算的不动点函数承载了所有 Haskell 中定义的递归函数，包括我们在 Haskell 中定义的不动点函数 `fix` 本身，但是读者可能需要了解更多 λ 演算与 System F 的内容才会理解它们的联系，这里就不多介绍了。学习不动点函数不仅仅是出于函数式编程理论的需要，在实践编程中也可能会常常用到它。下面，看一下计算不动点的函数 `fix` 是如何定义的：

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

`fix` 函数将无限地调用函数 *f*：

```
fix f
= f (fix f)
= f (f (fix f))
= f (f (f (fix f)))
...

```

例如：

```
fix (l:)
= l: (fix (l:))
= l: l: (fix (l:))
...
= [l..]
```

感谢惰性求值，使用 `take 10 (fix (l:))` 可以得到该列表的前 10 项。

使用不定点函数可以定义其他递归函数。定义过程中，函数名就不会出现在函数体中了，

比如，阶乘函数可以通过 fix 函数这样定义：

```
factorial :: Int -> Int
factorial = fix (\f n -> if (n==0) then 1 else n * f (n-1))
```

\f n 中的 f 就可以代指 factorial 函数了。可以看到，在定义 factorial 的时候并没有用到 factorial 函数本身，而是通过函数 fix 递归得到。其中，f 的类型为 Int -> Int，因为它与 factorial 的类型相同，n 的类型为 Int，表达式结果的类型亦为 Int，所以 (\f n -> if (n==0) then 1 else n * f (n-1)) 的类型为 (Int -> Int) -> (Int -> Int)，应用到 fix 函数后得到一个以 Int -> Int 为类型的函数。其中，3 的阶乘的计算过程如下：

```
factorial 3
= fix (\f n -> if (n==0) then 1 else n * f (n-1)) 3
= (\f n -> if (n==0) then 1 else n * f (n-1))
  (fix (\f n -> if (n==0) then 1 else n * f (n-1))) 3
= 3 * ((fix (\f n -> if (n==0) then 1 else n * f (n-1))) 2)
...
= 3 * 2 * 1 * ((fix (\f n -> if (n==0) then 1 else n * f (n-1))) 0)
= 3 * 2 * 1 * ((\f n -> if (n==0) then 1 else n * f (n-1))
  (fix (\f n -> if (n==0) then 1 else n * f (n-1))) 0)
= 3 * 2 * 1 * 1
= 6
```

类似地，可以用 fix 函数定义斐波那契数列和其他递归函数。由于惰性求值，这里第一个参数 (fix (\f n -> if (n==0) then 1 else n * f (n-1))) 不会永远计算下去。从函数应用的顺序读者也能看得出来，我们不会一直应用 fix 函数，而是在需要的时候再应用。有些情况下我们不希望 fix 函数一直调用下去。比如，我们只需要调用函数 f，直到所得的结果不再发生变化就可以停止了，即 $f(x)=x$ ，这里， x 为 $f(x)$ 的不动点。当它们相等时，计算就不必再进行了，返回 x 即可。

```
fix :: (a -> a) -> a
fix f x | x == f x = x
         | otherwise = fix f (f x)
```

有时，也可设定停止的条件为相邻两次结果之间的差值小于某个精度。一个很好的例子就是，在写开方函数时，当结果到达一定的精度时就停止。下面以牛顿法开方为例，说明不动点函数的用途。

牛顿法开方

牛顿法开方一个数值 c ，相当于方程 $x^2 - c = 0$ 的根。从一个估计的值 t 开始，如果 $t = \frac{c}{t}$ ，那么 t 则为 c 开方的结果。也就是说我们不断将 t 逼近于结果就可以了，那么 $t + c/t$ 应该等于 $2t$ ，所以这个函数就是 \t -> (c/t + t) / 2.0，这样，开方时每一次计算都会更加逼近真实的结

果。在逼近的过程中，会得到一系列的数字。这其实可以定义一个简单的递归函数，通过给定迭代次数来得到一个近似的结果。

```

squareroot :: Int -> Double -> Double
squareroot 0 x = x
squareroot n x = squareroot (n-1) x + x / squareroot (n-1) x )/2

> squareroot 10 5
2.23606797749979

```

如果使用 fix 函数，就需要一个需要函数来判定何时停止，因为不需要函数永远计算下去，只是需要达到一定的精度即可。那么，这种不动点函数定义如下：

```

fix :: (t -> t -> Bool) -> (t -> t) -> t -> t
fix c f x | c x (f x) = x
           | otherwise = fix c f (f x)

```

借助定义的带停止条件的 fix 函数，其实可以更为方便地定义牛顿法开方，并且可以控制想要得到的精度：

```

newton :: Fractional a => a -> a -> a
newton c t = (c/t + t) /2.0

mysqrt :: Double -> Double
mysqrt c = fix (\a b -> a - b < 0.000001) (newton c) c

> mysqrt 3
1.7320508100147274

```

这里，可以看到不动点函数其实是对递归的一种更为高等的抽象。mysqrt 函数会把 newton 函数作为输入然后不停地迭代，所以它是一个高阶函数。关于不动点函数的使用，将在后边的章节中需要时再继续讨论，这里就不再举例了。

5.13 无基本条件递归和惰性求值

在 Haskell 中有着这样一类的递归：它们并没有递归的基本条件，而仅仅有一个递归步，比如上节中的不动点函数。这些函数在不断的调用自身，但却没有停止条件。另外，还有 [1, 1..] 这个列表可以用下面的这种递归方法来定义：

```

ones = 1:ones
ones
= 1:ones
= 1:(1:ones)
= 1:(1:(1:ones))
...

```

定义这种函数时，并没有将参数向某个数据结构的某种形式靠拢，而是用这个函数计算本

身去填充一个数据结构的构造器参数。列表的一个构造器需要两个参数，一个是为首的元素，另一个是为首元素后的列表，而 ones 自己填充了首元素后的列表。这样的例子还有很多，还可以加一些函数调用，比如自然数的列表 [0, 1..] 和还有斐波那契数列，使用这种定义如下：

```
nature = 0 : map (+1) nature
fibs = (0:1:zipWith (+) fibs (tail fibs))
```

在调用时，可以使用 take 函数来取这个列表中的前几项，虽然列表是无穷的，但是感谢惰性求值，用 take 会使取前几项的计算过程停止。

```
> take 5 ones
[1,1,1,1,1]

> take 5 nature
[1,2,3,4,5]

nature = 0 : map (+1) nature
= 0 : map (+1) (0 : map (+1) nature)
= 0 : 1 : map (+1) (map (+1) nature)
= 0 : 1 : map (+1) (map (+1) (0 : map (+1) nature))
= 0 : 1 : map (+1) (1 : map (+1) (map (+1) nature)))
= 0 : 1 : 2 : map (+1) (map (+1) (map (+1) (map (+1) nature)))
= ...
```

这也就说明了，Haskell 在计算时并没有完全计算整个列表，而仅仅计算它所要的部分。能以这种方法生成列表是 Haskell 重要的特性之一。对于斐波那契数列的惰性求值的定义与 nature 函数是相似的，读者可以自己根据定义写出计算的前几步。

变得懒惰

变得懒惰常常需要考虑无限的情形或者忽略那些用不到的参数。比如，下面的函数返回两个列表中较短的一个：

```
shorter :: [a] -> [a] -> [a]
shorter xs ys | x < y = xs
               | otherwise ys
where x = length xs
      y = length ys
```

如果对于 Haskell 的类型、函数的定义方式不熟悉，就很难看出这个函数定义是有问题的。严格地讲，上面函数的问题有两个，但不是大问题。其一，如果一个列表的长度是有限的，另外一个是无限的，那么这个函数则不会停止。其二，因为 length :: [a] -> Int，Int 是有范围的，所以，如果列表的长度大于 Int 的最大值，这样就可能会得到负值而导致错误的比较结果。但正常情况下，默认处理的列表长度都是小于 Int 的边界值的。所以，如果处理的列表长度可能超过 Int 的范围，就需要使用 Data.List 中的 genericLength 函数。这里，使用递

归来实现比较函数，然后将其作为 shorter 的辅助函数：

```
lazyShorter :: [a] -> [a] -> [a]
lazyShorter xs ys = short xs ys
  where short [] ys = True
        short xs [] = False
        short (x:xs) (y:ys) = short xs ys
```

由于递归地遍历这两个列表，这些问题在写 Haskell 程序的时候需要注意。当的确要处理非常长的列表时，就不可以使用 Int 与 length。这里的 short 函数就用了惰性求值的特性，只要有一个列表为空时，就直接返回结果，而不再去计算另外一个列表了。这样，即便另外一个个列表是无限的，也不会使这个计算卡住。

练习

有一列数被称为汉明数（Hamming Number），它是这样定义的：1 属于该序列，然后，如果 h 是一个汉明数，那么 $2h$ 、 $3h$ 、 $5h$ 都是汉明数。你可以没有重复并且升序地生成所有的汉明数吗？提示：首先定义一个函数 merge，这个 merge 与归并排序中的 merge 相似，但是会将重复的数字去掉，然后再像定义 nature 那样定义名为 ham 的函数，将 map (*2) ham、map (*3) ham 与 map (*5) ham 使用第一步定义的 merge 函数融合起来。这个数列的前 15 项如下：

```
> take 15 ham
[1,2,3,4,5,6,8,9,10,12,15,16,18,20,24]
```

本章小结

本章是书里非常重要的一章，因为递归函数在实践编程中是非常重要的思想，并且在编写 Haskell 程序时几乎不可能避免递归。递归也有各种各样的形式，有些时候区分它们显得非常重要。因此，如果读者对于递归理解得不是很好，那么就需要花多一些的时间来消化本章的内容，认真品读本章中的示例代码。

第6章

列表内包

列表内包（list comprehension）的 comprehension 一词，在这里意为列表内涵、内包，具体指的是 Haskell 提供一种精简的语法来对列表中的元素进行处理，如过滤、应用函数或者组合，这种表达形式源于数学中定义集合的符号表示。下面介绍列表内包的使用。

6.1 列表生成器

通常，用已经有的集合来构造新的集合，比如，有一个自然数集 N ，可以定义一个集合，这个集合是 10 以内自然数的平方组成的集合，可以书写成 $\{x^2 | x \in \{1..100\}, x < 10\}$ 。在 Haskell 中，用列表可以非常直观地写成：

```
> [x^2|x <- [0..100], x < 10]
[0,1,4,9,16,25,36,49,64,81]
```

其中 $x <- [0..100]$ 为生成器，它的意思是说， x 取从 0 开始的自然数直到 100，即表达了集合表达式中的 $x \in \{1..100\}$ 的部分，而 $x < 10$ 则是对 x 的条件限制， x^2 则对符合条件的 x 进行转换，应用平方函数，从而得到了一个新的列表。这里其实可以简写为 $[x^2 | x <- [1..10]]$ 。

生成列表的时候可以使用多个生成器，但是生成的过程是有一定的顺序的，写在后边的生成器将会被先遍历。

```
> [(x,y) | x <- [1..4], y <- [1..4]]
[(1,1),(1,2),(1,3),(1,4),(2,1),(2,2),(2,3),(2,4),(3,1),(3,2),(3,3),(3,4),(4,1),(4,2),(4,3),(4,4)]
```

用列表生成器可以定义 map 函数，定义如下：

```
map' f xs = [f x | x <- xs]
```

列表内包可以对元素进有条件的限制，比如，上面例子中的 `x < 0`，这也正是 `filter` 函数的作用，使用列表内包的 `filter` 函数定义如下：

```
filter' f xs = [x | x <- xs, f x]
```

当然，限定的条件可能不止一个，可以再加入一些限定的条件，这些限定条件之前是且的关系。比如：

```
> [x | x <- [0..], even x, x > 10]
[12, 14, 16, 18, 20, 22, 24, 26, 28, ...]
```

这是一个无穷列表，后面的 `even x` 与 `x > 10` 为限定条件，也就是说这组数要同时满足：

- (1) 来自自然数集；
- (2) 必须为偶数；
- (3) 要大于 10。

使用列表内包时，如果用不到生成器中产生的值，可以使用通配符来匹配。例如，使用列表内包定义 `length` 函数的话就可以将列表中的元素全部映射成 1，然后求和就可以了。这里，不必在乎列表中的元素是什么，所以可以使用通配符形式，那么 `length` 函数可以这样定义：

```
length' xs = sum [1 | _ <- xs]
```

使用列表内包可以很容易地表达顺序式语言中的循环。用户可以十分方便地生成各种类型的值，比如，如果需要生成乘法口诀表的列表，只需要一行就可以了：

```
> [show x ++ "*" ++ show y ++ "=" ++ show (x * y) | y <- [1..9], x <- [1..y]]
```

再比如，早在 14 世纪，印度数学家 Madhava 就通过级数表示 π 了，但是这一方法并没有被人们熟知。后来，数学家莱布尼茨再次独立用积分发现了它才引起了广泛关注。莱布尼茨知道，因为其实 π 是 $4 \times \arctan 1$ 的级数展开，写做：

$$\pi = 4 \cdot \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots \right)$$

借助列表内包可以很容易表达这个序列。

```
series :: Int -> [Double]
series n = [1 / (2 * (fromIntegral k) + 1) * (-1)^k | k <- [0..n]]
```

接下来，可以使用这个公式来粗略地计算 π 的值。由于这个收敛较慢，要应用 20 万次以上才能精确到第 5 位小数，所以并不是什么高效的方法。

```
> 4 * (sum $ series 200000)
3.141597653564762
```

当然如果列表较短就不能算出 π 值，并且在计算的过程中使用 `Double` 可能由无限循环小

数等问题引起的精度丢失。

练习

大数学家欧拉在不久后找到了收敛更快的公式，由于：

$$\frac{\pi}{4} \arctan\left(\frac{1}{2}\right) + \arctan\left(\frac{1}{3}\right)$$

那么，使用级数展开则有：

$$\pi = 4 \left(\frac{1}{2} - \frac{1}{3} \cdot \left(\frac{1}{2}\right)^3 + \frac{1}{5} \left(\frac{1}{2}\right)^5 + \dots \right) + 4 \left(\frac{1}{3} - \frac{1}{3} \cdot \left(\frac{1}{3}\right)^3 + \frac{1}{5} \left(\frac{1}{3}\right)^5 + \dots \right)$$

你能通过这种级数展开用列表内包计算出 π 的近似值吗？

6.2 素数相关趣题

本节主要讨论素数，素数又称质数，是数论中的重要一类数。它在整数中的分布也是没有规律的，没有统一的通项公式可以表达。素数在密码学中有着非常重要的位置。比如，RSA 公钥加密时就需要生成两个非常大的素数。另外，著名的歌德巴赫猜想（每个不小于 4 的偶数，都能表示成 2 个素数之和）是一个看似容易却困扰数学家们多年的问题。素数的定义非常简单，对于一个整数，除 1 和它本身外，再没有其他因数了，这个数就是素数。就是说，只除以 1 或者本身时余数为 0，除以其他数所得的余数均不为 0。下面定义一个函数对一个数进行因数分解。

我们已经知道素数的定义是这样的：对于一个整数，除 1 和它本身外，再没有其他因数了。换句话说，就是这个数 p 除以 $[2..n-1]$ 的余数均不为 0。返回一个整数的所有因数的列表的函数可以定义如下：

```
factors :: Integral a => a -> [a]
factors n = [x | x <- [1..n], mod n x == 0]
```

这个函数会依次对 $1 \sim n$ 的数尝试，把余数为 0 的取出。来试一下：

```
> factors 24
[1, 2, 3, 4, 6, 8, 12, 24]
```

这样，检验一个数是否为素数的函数，只需检验它的因数是否只有 1 和它自己，于是有：

```
isPrime :: Integral a => a -> Bool
isPrime n = factors n == [1, n]
```

这两个函数完整并且精简地表达了素数的本质。下面来生成素数的列表。使用列表生成器，这个就很容易了：

```
primes :: Integral a => a -> [a]
primes n = [x | x <- [1..n], isPrime x]
```

这样，生成素数的效率是很低的，因为每一次求因数的时候，要从 $1 \sim n$ 全部遍历一次。下面，来看看如何减少一些运算量。从上面可以看出，对于每一个数，如果想要判断它是否是素数，需要把 $1 \sim n$ 的数全部取一次余。下面做进一步的思考，简化一些过程：

(1) 除去 2 以外，所以的素数都必须是奇数；

(2) 素数必须是大于等于 2 的整数；

(3) 对于一个整数 N ，若它有一个因数为 p_1 ，必有另外一个因子 p_2 使得 $N=p_1p_2$ ，并且 p_1 与 p_2 一定分布在 \sqrt{N} 的两端或 $p_1=p_2=\sqrt{N}$ ；

(4) 因为比 2 大的素数必须是奇数，所以它一定也不会有偶因数。

所以，对 2 进行单独讨论，然后对于其他的数只需要用它去除以所有 $3 \sim \sqrt{N}$ 的所有奇数，若余数全不为 0，则这个数为素数。

这样，对于 `isPrime` 函数，可以这样定义：

```
isPrime' :: Integral a => a -> Bool
isPrime' 2 = True
isPrime' p = p > 1 && (all (\n-> p `mod` n /= 0) $ takeWhile (\n->n*n <=p) [3,5..])
```

`takeWhile (\n->n*n <=p) [3,5..]` 求出了 $3 \sim \sqrt{N}$ 的所有奇数组成的列表，用 `(\n-> p `mod` n /= 0)` 检验其中的某一个，然后使用 `all` 函数检验是否所有的符合不余零的条件，如果余数都不为 0，就是素数了。下面这个生成质数的列表就会运行得非常快了。

在加密的时候，常常需要生成非常大的素数，现在可以写一个函数，以一个任意大的正整数作为参数，称为“种子”，然后返回大于或等于它并且距离最近的素数。思路是这样的，若这给定的这个“种子”为奇数，先来判断它本身是否为素数，若为素数直接返回即可，否则需要递归地判断这个数加 2 是否为素数。若这个“种子”是偶数，递归地判断这个数加 1 是否为素数即可。函数如下：

```
nextPrime :: Integer -> Integer
nextPrime a | odd a = if isPrime a then a else nextPrime (a+2)
            | otherwise = nextPrime (a+1)
```

这个是一个递归函数，但仔细观察，它与前边的递归函数略有不同。这个函数一直在保持增长，直到参数为素数停止。而前边的递归函数都是就参数的某一具体实例为基本条件停止，像整数乘方和阶乘递归那样，确定递归的基本条件一定可以达到，但在这里，`nextPrime` 函数却不是的。这里函数停止的原因是素数有无穷多个的，总是可以找到大于某个数的素数。理论上递归的基本条件是可以达到的。

埃拉托斯特尼筛法

之前遍历素数的方法还是有些慢，下面介绍一种能更快地生成素数的方法。这种生成方法

就是埃拉托斯特尼筛法 (Eratosthenes sieve)。给定从 2 开始连续的一列数，2 为素数。那么，在 2 之后有 2 为因数的数均不为素数，可以被筛掉。下一个数为 3，3 之后的所有为 3 的倍数的数就全被筛掉了，因为 4 为 2 的倍数，这在第一轮中已经被筛掉了，下一个是 5，假定列表是无穷的，那么按着这个方法可以遍历所有的素数。

```
sieve:: (Integral a) -> [a] -> [a]
sieve (p:xs) = p: sieve [x | x <- xs, x `mod` p /= 0]

primes = sieve [2..]
```

有了惰性求值，这种写法非常直接地表达了埃拉托斯特尼筛法的思想，可以处理一个无限长度的列表。

```
>primes
[2, 3, 5, 7, 11, ...]
```

练习

定义一个函数 primeFactors，可以将一个整数做质因数的分解，如 $36=2 \times 2 \times 3 \times 3$ ，其中有 2 个 2，2 个 3，那么有：

```
> primeFactors 36
[(2,2), (3,3)]
```

6.3 凯撒加密

提到间谍情报工具，可能人们都会认为它们大多都是现代的高科技产品，或者至少并不是那么古老的。其实早在古罗马时代，凯撒就已经使用密码术对信息进行加密了。图 6-1 所示的密码盘就是凯撒密码加密和解密的工具了，它的年代一直可以追溯至美国内战。这种加密方法当时由美国南方邦联采用。



图 6-1 凯撒加密密码盘

6.3.1 加密

凯撒加密的基本原理就是将明文信息中的字母进行移位，从而使得信息看上去变得混乱。在英文中加密过程大约是这样：两人约定一个私钥，即字母移动的位数 k ，明文被向右移动 k 位，对在原文的字母 m 我们向右移动 $k \bmod 26$ ，得到密文字母 c 。对应的就是旋转密码盘 k 位得到明文与密文的对应，加密公式如下：

$$c = m + k \bmod 26$$

比如， a 向右移动 3 位，就变成了 d 。再比如， e 向右移动 1 位，就变成了 f 。这样，给定一个字符串，就有了 25 种加密的方法。下面我们来写一下加密的算法。首先，定义两个简单的函数，第一个函数可以将一个字符转换成 ASCII 码，另外一个函数则是第一个函数的反函数，可以将 ASCII 转换成字符。需要用到两个函数，即 `chr` 与 `ord`，它们都在 `Data.Char` 里。

```
import Data.Char (ord, chr, isLower)
```

这里，默认为明文中只有小写字母：

```
char2int :: Char -> Int
char2int c = ord c - ord 'a'

int2char :: Int -> Char
int2char n = chr (ord 'a' + n)
```

下面这个函数可以将字符转换成数字，进行移位，然后再转换成密文：

```
shift :: Int -> Char -> Char
shift n c | isLower c = int2char ((char2int c + n) `mod` 26)
          | otherwise = c
```

通过列表内包与 `shift` 函数定义加密函数，即给定移的位数与原文，就可以算出密文。

```
encode :: Int -> String -> String
encode n xs = [shift n x | x <- xs]

> encode 13 "see you this afternoon"
"frr lbh guvf nsgreabba"
```

6.3.2 解密

这种加密方式是很好被破解的，当看到英语文本和密文的时候，能很快地分出明文与密文，英语中的单词的组成的解是有一些规律可循，比如元音和辅音的出现或者常用单词拼写等。而且解密的运算量并不是很大，只要再移 25 次位，用计算机算出 25 组解，如果是英文文本，一个懂英语的人就应该可以从这 25 组中找出明文了。但是，用计算机来找的话应该使用怎样的方法呢？在英语里，26 个字母出现的频率已经被统计出来了，如果给定了大量的英文文本，那么文本中每一个字母出现的频率应当是接近那个统计出来的频率的，也就是说，其余 25 组移位结

果中，最接近这个频率统计的就最可能是想要的结果。这里注意，因为用的是频率统计的方法，所以如果没有足够的英文文本，可能是无法得到想要的结果的，这种解密方式常常称为频率攻击。计算英文本频率与统计频率接近程度的公式如下：

$$\sum_{i=0}^{n-1} \frac{(os_i - es_i)^2}{es_i}$$

其中 os_i 表示明文中的某一字符在英文文本中的概率， es_i 是某一字符在密文中的概率。在 Haskell 中这个公式可以这样定义：

```
chisqr :: [Float] -> [Float] -> Float
chisqr os es = sum [(o - e) ^ 2 / e | (o, e) <- zip os es]
```

当然，也可以用 `zipWith` 来定义。

这个公式就是在量度某一文本与大量英文文本各个字母出现概率的距离，在 25 组移位的统计中，如果这个距离最小，则最有可能是想要的结果。英文字母 a~z 在英文文本中出现的百分比如下：

```
table :: [Float]
table = [8.2, 1.5, 2.8, 4.3, 12.7, 2.2, 2.0, 6.1, 7.0, 0.2, 0.8, 4.0, 2.4, 6.7, 7.5,
1.9, 0.1, 6.0, 6.3, 9.1, 2.8, 1.0, 2.4, 0.2, 2.0, 0.1]
```

首先，定义 `count` 函数来统计某一英文字符在文本出现的次数：

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

接下来，定义统计概率需要的函数。首先是求百分比函数。由于是小数，因此需要使用 `fromIntegral` 将 `Int` 类型化为一个具有 `Num b => b` 的类型的值，最后求比值。

```
percent :: Int -> Int -> Float
percent n m = (fromIntegral n / fromIntegral m) * 100
```

再来统计文本中有多少个小写英文字母。使用列表内包很容易定义这个函数：

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

接下来，就可以定义统计文本中各个字母出现概率的函数了。只需要查每个字母在文中有多多个就可以了。同样地，使用列表内包很容易实现它：

```
freqs :: String -> [Float]
freqs xs = [percent (count x xs) n | x <- ['a'..'z']]
where n = lowers xs
```

我们需要将统计出的概率移位 25 次，以便将移位的每一次都与 `table` 中的概率应用于 `chisqr` 函数来计算与 `table` 中概率的距离，所以我们定义一个 `rotate` 函数来移动列表中的元素所在的位置：

```
rotate :: Int -> [a] -> [a]
rotate n xs = drop n xs ++ take n xs
```

最后，只需要定义解密函数 `crack`，从 25 组值中找出最小的，它在表中所在的位置就是移动的位数。首先，得到密文中各个英文字符的频率，记为 `table'`，`table'` 与原文频率表 `table` 依次应用 `chisqr`，这样就得到了所有的距离，从这些距离中找到最小的，其所在的位置就是解密需要移动的次数。

```

crack :: String -> String
crack xs = encode (-factor) xs
where
    factor = head (positions (minimum chitab) chitab)
    chitab = [chisqr (rotate n table') table | n <- [0..25]]
    table' = freqs xs

positions :: Eq a => a -> [a] -> [Int]
positions x xs = [i | (x', i) <- zip xs [0..], x == x']

```

凯撒加密的内容来自 (Hutton, 2007) 中第 5 章的第 5 节，有兴趣的读者可以阅读这本书。

6.4 排列与组合问题

排列 (permutation) 与组合 (combination) 是高中数学概率部分的重要内容，同时也是离散数学中计数原理的重要内容。在解决实际问题时也常常要用到排列组合，本节中主要讨论它们。

6.4.1 排列问题

有些时候，人们想通过生成一个列表的全排列解决某问题，这些方法常常称为蛮力法 (brute force)。比如，后面章节的八皇后问题与 24 点问题，这样就要生成所有的组合，要用到一个全排列函数：

```

>permutation [1,2,3]
[[3,1,2],[1,3,2],[1,2,3],[3,2,1],[2,3,1],[2,1,3]]

```

得到一个列表的全部排列可以使用两种方法。第一种方法采用的是递归地插入的思想。现在，假设我们要得到 `[1,2,3]` 的全部排列并且已经求出列表 `[1,2]` 的全排列，即 `[[1,2],[2,1]]`，如何生成 `[1,2,3]` 的全排列呢？如图 6-2 所示，只需要分别将 3 插入到 `[1,2]` 与 `[2,1]` 的中间和两边的空隙里就可以了。两个数，有三个位置可以插入，所以最后的全排列是 6 种：`[3,1,2]`，`[1,3,2]`，`[1,2,3]`，`[3,2,1]`，`[2,3,1]`，`[2,1,3]`。

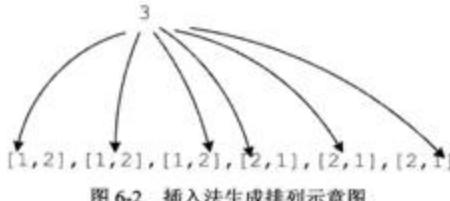


图 6-2 插入法生成排列示意图

那么，当有 $n-1$ 个元素的时候，就有 n 个位置，只需将第 n 个元素插入到 $n-1$ 个元素生成

的全排列的每一个列表的每一个位置即可，最后排列的总数即为 $nA_{n-1}^{n-1}=A_n^n$ ，与数学公式完全相符。下面来定义插入函数，即将某一元素插入列表中的所有位置。

```
insert :: a -> [a] -> [[a]]
insert n [] = [[n]]
insert n (n':ns) = (n:n':ns):[n':ns'|ns'<-insert n ns]
```

那么，全排列函数就是将第一个 x 插入到后边列表全排列的每一个元素中，于是有：

```
permutation [] = [[]]
permutation (x:xs)= concat [insert x permuxs|permuxs<-permutation xs]
```

另外一种递归的思路是：在全排列的过程中，所有在列表中的元素必须开头至少一次。然后，当某一元素 p 开头时，只要对除这一元素的列表进行排列，就得到了以 p 开头的排列，那么若生成每一个元素开头的排列，就得到了整个列表的全部排列。

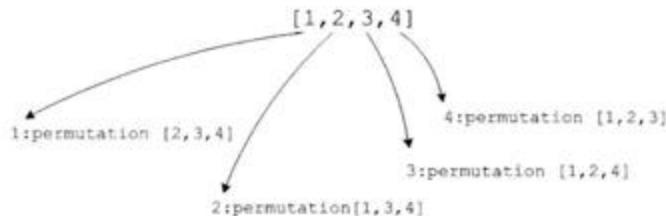


图 6-3 删除法生成排列示意图

当有 n 个元素时，各个元素开头共有 n 种情况。对于某一元素开头，将其与余下 $n-1$ 个元素的全排列结合，就会有 A_{n-1}^{n-1} 个，则对于某一元素会开头 A_{n-1}^{n-1} 次，最终，有 $nA_{n-1}^{n-1}=A_n^n$ ，具体如图 6-3 所示。这种思想依旧符合公式。

```
import Data.List (delete)

permutation :: Eq a => [a] -> [[a]]
permutation [] = [[]]
permutation xs = [y:ys | y<-xs, ys<-permutation (delete y xs)]
```

但是，这里由于需要使用 `delete` 函数，类型 `a` 需要可以比较相等。有些情况下，不需要得到所有元素的全排列，只需要得到 n 中的 m 个的排列。它可以这样定义：

```
permutation' :: Eq a => Int -> [a] -> [[a]]
permutation' 0 _ = [[]]
permutation' n l = [x:xs | x<-l, xs <- permutation' (n-1) (delete x l)]
```

6.4.2 错位排列问题

假设给定一列从 1 开始的连续的 n 个整数，需要得到一些排列，这些排列满足数字 i 不在

第 i 个位置，怎样生成那些符合这种条件的列表呢？想法很简单，只需要将给定的数字依次取出放在第 n 个，第 $n-1$ 个直到第 1 个位置，但是放的过程中，这个整数与余下的整数的总个数不能相等，即 i 不在第 i 个位置上。由于 Haskell 列表的定义方式是一个元素跟一个列表，所以将左手边视为高位，右手边视为低位，从右自左地来放置这些整数。最后，只需要将它们依次倒置即可。

```
import Data.List (delete)

derangements :: [Int] -> [[Int]]
derangements [] = [[]]
derangements l = [x:xs | x <- l, xs <- derangements (delete x l), x /= length l]

derangements' n = map reverse (derangements [1..n])
```

n 为 2 时应该有一种情况，就是 $[2, 1]$ ，而 n 为 3 时应该只有二种情况， n 为 4 时应该有九种情况。下面，测试一下这个函数：

```
> derangements' 3
[[2,3,1], [3,1,2]]
```

```
> derangements' 4
[[4,3,2,1], [3,4,2,1], [2,3,4,1], [4,3,1,2], [3,4,1,2], [3,1,4,2], [2,4,1,3], [4,1,2,3], [2,1,4,3]]
```

6.4.3 组合问题

在一些情况下，想要得到一个集合的所有子集，比如：

```
> powerSet [1,2,3]
[[], [1], [2], [3], [1,2], [1,3], [2,3], [1,2,3]]
```

这些函数通过列表实现起来是十分方便的。对于集合每个元素，只可能有两种可能，就是存在于某一子集中或者不存在于某一子集中。所以 powerSet 函数可以定义如下：

```
powerSet = choice (\x -> [True, False])

choice :: (a -> [Bool]) -> [a] -> [[a]]
choice _ [] = [[]]
choice f (x:xs) = [if choose then x:ys else ys | choose <- f x, ys <- choice f xs]
```

这样，对于我们第一个元素映射成为一个两个元素的列表 $[True, False]$ ，如果为 $True$ 则说明该元素在列表中，我们只需要返回 $x:ys$ 即可， ys 可以递归地生成；当不在列表中时，直接返回 ys 即可。这样任意一个元素不在列表中的情形都被考虑到了，所以得到的就是所有的子集。当然，还有另外一种方式来定义它。首先该集合为空集时，它的子集也为空集。当该集合不为空集时，对于第一个元素 x ，只有包括 x 与不包括 x 两种情形。当包括 x 时，所有的子集为 $[x:r | r <- powerSet xs]$ ；而不包括 x 时，所有的子集为 $powerSet xs$ ，所以有：

```
powerSet :: [a] -> [[a]]
powerSet [] = [[]]
powerSet (x:xs) = [x:r | r <- powerSet xs ] ++ powerSet xs
```

如果要从一个集合中选出 n 个元素，要有序地来选择这些元素，并且保证每个元素只被选了一次。当选择 0 个元素时，结果为空集；当不为空时，在一个组合中，每个元素只可能被选一次。先使用 tails 函数求出列表所有后缀。比如：

```
> Data.List.tails [1,2,3,4]
[[1,2,3,4],[2,3,4],[3,4],[4],[]]
```

假定需要从 $[1,2,3,4]$ 中选取 n 个元素，当开头的元素被选择时，只需要从后面的元素中再递归地选出 $n-1$ 个元素即可。例如，当需要从 $1:[2,3,4]$ 中取出 3 个元素时，如果包括 1 时，只需要从 $[2,3,4]$ 中递归地再取出 2 个即可；然后，再对 $[2,3,4]$ 、 $[3,4]$ 、 $[4]$ 、 $[]$ 列表进行同样的处理，即从 $2:[3,4]$ 中选出 3 个。因此，组合函数定义如下：

```
import Data.List (tails)

combinations :: Int -> [a] -> [[a]]
combinations 0 _ = [[]]
combinations n xs = [y:ys | y:xs' <- tails xs, ys <- combinations (n-1) xs']

> combinations 2 [1..4]
[[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]
```

练习

定义一个函数 choose :: Int -> [a] -> [[a]]，但与之前不同的是列表元素可以重复选取。比如：

```
> choose 2 [1,2]
```

```
[[1,1],[2,2],[1,2]]
```

6.5 八皇后问题

八皇后问题是回溯算法的典型问题。它是由有“数学王子”之称的德国数学家高斯于 1850 年提出的。

国际象棋中的皇后是非常厉害的，上下、左右及两对角线八个方向上的棋子都可以攻击。这个问题正是要求在 8×8 格的国际象棋棋盘上摆放八个皇后（如图 6-4 所示），使她们互相不能攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法。高斯那个时候认为有 76 种方案。在 1854 年在柏林的象棋杂志上不同的作者发表了 40 种不同的解。不久，高斯于 1855 年去世。后来，有人用图论的方法解出 92 种结果，92 也是正确的结果。计算机发明后，有多种方法可以解决此问题。

有些读者可能听说过解决这个问题所用到的回溯算法，但此处先不考虑什么是回溯算法。八皇后问题本质上是一个有条件限制的搜索问题，刚得到这个问题的时候，未学过算法的读者读到回溯算法就认为这一定很高深，其实不然。相信很多读者已经有思路了。作者在这里就有一个很天真的想法：

- (1) 先想办法生成所有棋盘上皇后的位置；
- (2) 写一个限制条件函数，根据限制的条件进行逐个筛选。

可以用一个长度为 8 的列表来表示一个棋盘，从左到右有 8 列，每列上的数就是皇后的位置，这样就直接避免了皇后可以在同一列的情况。那么，可以用列表将图 6-4 中的棋盘表示为 [6, 1, 5, 2, 8, 3, 7, 4]。

首先，想办法用列表递归地生成所以皇后可能的位置。假定棋盘为 $n \times n$ ，对于每一列的皇后都有 n 种选择，我们只需要一列一列地生成即可。有 0 列的时候为 [[]]，如果不为 0 时，只需将第 k 列皇后能在的 n 个位置与递归生成的 $k-1$ 列的所有棋盘组合即可，这样就有：

```
positions 0 n = []
positions k n = [x:xs | x <- [1..n], xs <- positions (k-1) n]
> positions 4 4
[[1,1,1,1],[1,1,1,2],[1,1,1,3],[1,1,1,4],[1,1,2,1],[1,1,2,2],[1,1,2,3],[1,1,2,4],...]
```

下面判断任意两个皇后不在同一行。这个问题好解决，只需要判断每个元素是否存在于它后边跟随的列表。

```
noSameRow [] = True
noSameRow (x:xs) = (not $ elem x xs) && noSameRow xs
```

接下来，写一个函数来判断对角线。如果给定一个函数 $q(n)$ ，参数 n 为棋盘上的某一列，它会返回的值为该列上皇后所在的位置，那么两个皇后不在对角线上的条件等价于下面的公式：

$$|q(n)-q(m)|-|n-m|\neq 0$$

也就是说，棋盘上的两个皇后水平位置之差的绝对值不能等于竖直位置之差的绝对值。要用到列表元素的位置，其实只需要用 `zip [1..] xs` 就可给每个元素加上次序，这样，新的元组的第一个元件就表示元素在列表的第几个位置。这样没有皇后在对角线上的函数 `noSamediaig` 就可以定义为：

```
noSameDiag [] = True
noSameDiag xs@(_:_:xs') = and [abs (i1-i)/=abs (p1-p) | (i,p)<-ip] && noSameDiag xs'
  where (i1,p1):ip = zip [1..] xs
>noSameDiag [6,1,5,2,8,3,7,4]
True
```

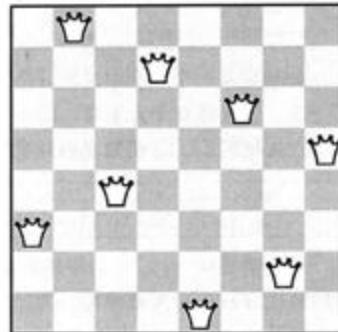


图 6-4 八皇后棋盘

```
>noSameRow [6,1,5,2,8,3,7,4]
True
```

这里，`xs@(x:xs')`中的@意为用`xs`来代指`(x:xs')`，引用`(x:xs')`与引用`xs`是一样的。下面这个函数就可以使用列表生成器或过滤器定义出来了。

```
queen n = [xs | xs <- positions n n, noSameRow xs, noSameDiag xs]

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

程序没有问题，但是对于8个皇后的情形普通的计算机是不能在一个可以接受的时间内给出结果的。因为每列有8个格子，有8列，所以列表总数是 8^8 ，也就是 $2^{24}=16\,777\,162$ 种情况，近1700万个，这是一个很大的数，目前的个人计算机无法很快筛选出来。因此，要想一个办法来优化。先来观察`positions`是如何生成这些位置的：

```
> positions 8 8
[[1,1,1,1,1,1,1,1],[1,1,1,1,1,1,2],[1,1,1,1,1,1,3].....
,[1,1,1,1,1,2,1],[1,1,1,1,1,1,2,2],[1,1,1,1,1,1,2,3]....]
```

正如所观察到的，`positions`的生成是从右手边开始的，同查数一样，但是以八进制向前进位，这个过程是从右至左的。当最右边最后一位为1的时候，由于问题的约束，倒数第二位一定不能为1了，因为在同一排；也不能为2，因为在一对角线上。像这些情况就不必再继续生成下去了。因此，只需要在`positions`函数生成左手边的`p`时，看`p`在`ps`中是否符合两个条件的限制，即不与`ps`中的元素相同，并且不与`ps`中的元素在一条对角线上，我们只需要额外定义一个函数`isSafe`就可以了。如果不符，那么终止这一列表的递归的生成：

```
positions' 0 n = [[]]
positions' k n = [p:ps | ps <- positions' (k-1) n, p <- [1..n], isSafe p ps]

isSafe p ps = not (elem p ps) || (sameDiag p ps)
    where sameDiag p ps = any (\(dist,q) -> abs (p-q)==dist) $ zip [1..] ps
```

这样，八皇后问题就解决了。

```
queens = positions' 8 8

>length 5 queens
92
(0.19 secs, 10450792 bytes)
```

其实，这本质上就是回溯算法了。解决这个题的出发点是终止那些不必要的棋盘，因为那些不必要的棋盘太多了，所以在那些不必要的棋盘生成之前，就把它们删除，然后再试着放在其他位置，从而生成下一个。

搜索过程如图6-5所示。先从最右边的1开始搜索，倒数第二列（即第7列）第一个成立的位置是第3行，然后再搜索下一列第一个成立的是放在第5行，第5列第一个成立的位置是

第2行，第4列第一个成立的是第4行，但再继续向下的时候，第3列没有成立的，然后回到前一个，也就是第4列，搜索第4列的下一个位置。依此次序搜索，就会得到第一个结果 $[4, 2, 7, 3, 6, 8, 5, 1]$ 。每一个位置下有8个位置需要被判定是否符合条件。如果读者知道树结构的话，实际上这是一个“八叉树”，而这里使用的是深度优先搜索。

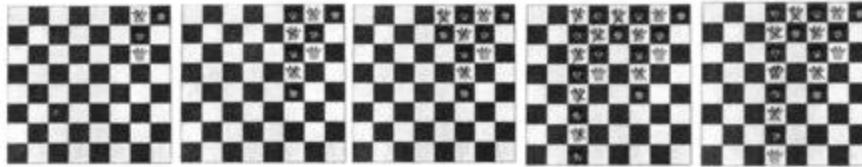


图 6-5 八皇后问题搜索过程示意图

前一个方法是终止继续生成已经不符合条件的棋盘。有没有其他法可以进一步减少那些不必要的棋盘呢？有的，因为没有两个皇后在同一排上的，其实，如果生成的是 $[1, 2, 3, 4, 5, 6, 7, 8]$ 的全排列，那么在同一行的情况就在生成组合的时候避免了。这样，只要让计算机从 $A_8^8=8!=40320$ 种里筛选出不在对角线上的即可。40 000 种情况对一台普通的计算机来说，还是可以接受的。

下面学以致用，用插入法来全排列一个列表：

```
insert :: a -> [a] -> [[a]]
insert n [] = [[n]]
insert n (n':ns) = (n:n':ns) : [n':ns' | ns' <- insert n ns]

permutation :: [a] -> [[a]]
permutation [] = [[]]
permutation (x:xs) = concat [insert x permux | permux <- permutation xs]
```

只要筛选出不在同一对角线上的情况就可以了：

```
queen' :: Int -> [[Int]]
queen' n = [xs | xs <- permutation [1..n], noSameDiag xs]

>length $ queen' 8
92
(2.25 secs, 123659176 bytes)
```

应用列表全排列的方法蛮力地来解决问题是一种常见的思想，在后边的章节解决 24 点问题的时候还要用到它。如果全排列的数值太大，比如要在 100×100 的棋盘上就不推荐这种方法了。`permutation` 函数在 `Data.List` 中被定义了，事实上我们只需要定义限定皇后不在同一条对角线函数和主函数 `queen` 就可以了，虽然非常简单，但是这里从近 4 万个列表中筛选，用了近 2s 的时间，而前一个回溯搜索的方法仅仅用了 190 ms 左右。那么可以通过优化全排列的方法来提高效率么？笔者认为是有些难度的，因为在采用插入或是删除生成全排列的过程中，原来不符合要求的列表可能因为插入或者删除就变得符合要求了。在排列生成的过程中，找不到一个固定的规律来限定结果的生成。相反，回溯法则是从右到左地渐渐增长列表，在增长的过

程中一直生成符合条件的列表，最终得到所有结果。

虽然解决了著名的八皇后问题，但是，如果读者仔细回想解决问题的过程可以发现，解决问题的时候可以先从一些天真、幼稚的想法开始，通过不断地让这些想法“成长”、“成熟”，最终可以非常好地解决问题。

6.6 计算矩阵乘法

计算矩阵时，常常需要计算矩阵乘法，它是这样定义的：

$$\sum_{r=1}^n a_{ir} b_{rj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \dots$$

这里，在 Haskell 中定义一个矩阵乘法运算符 `(*)`。它把矩阵 `a` 的行和矩阵 `b` 的列依次相乘，然后求和，最后得到一个新矩阵。使用 `Data.List` 库中的 `transpose` 函数可以将矩阵沿对角线翻转，使它的列变为行。例如：

```
> transpose [[1,2,3],[4,5,6]]
[[1,4],[2,5],[3,6]]
```

使用列表内包与递归，这个函数其实很容易定义：

```
transpose :: [[a]] -> [[a]]
transpose [] = []
transpose ([]:xss) = transpose xss
transpose ((x:xs) : xss) = (x : [h | (h:_) <- xs]) :
                           transpose (xs : [t | (_:t) <- xss])
```

可以直接导入来使用它：

```
import Data.List (transpose)
infixl 5 (*)
```

对照公式，可以使用 `sum` 与 `zipWith` 函数来定义矩阵乘法函数：

```
(*) :: Num a => [[a]] -> [[a]] -> [[a]]
(*) a b = [[ sum $ zipWith (*) ar bc | bc <- transpose b ] | ar <- a]
```

斐波那契数列与矩阵乘法

斐波那契数列可以使用矩阵来计算，计算的公式如下：

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

根据公式，可以很容易地定义这个函数。但是，在定义的时候可以优化矩阵乘法，分为奇偶来讨论。正如定义乘方函数那样，这样可以避免重复的计算。

```

unit = [[1,1],[1,0]]

fib 1 = unit
fib n | even n = let m = fib (div n 2) in m |*| m
      | otherwise = let m = fib (div (n-1) 2) in m |*| unit |*| m

> fib 10
[[89,55],[55,34]]

```

6.7 最短路径算法与矩阵乘法

给定一个地图，上面有很多点，点和点之间有权重，权重可以表示两点间的距离或者达到的时间，这样的一个地图是一个二元函数，它的类型为 $\text{Point} \rightarrow \text{Point} \rightarrow \text{Distance}$ ，即给定两点，可得一权重，没有关系的两点间的距离为正无穷，这样的二元函数可以用一个正方形矩阵来表示。例如，对于给定的两点，在矩阵中，以起点为行，以终点为列，就可以在矩阵中很容易地得到任意两点的距离。这样，图 6-6 所示的地图就可以用图 6-7 所示的矩阵表示了。因为这张地图是无向图，所以矩阵是对称。

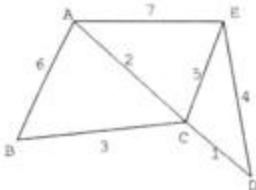


图 6-6 无向图

	出发				
A	0	6	2	∞	7
B	6	0	3	∞	∞
C	2	3	0	1	5
D	∞	∞	1	0	4
E	7	∞	5	4	0

图 6-7 无向图对应矩阵

这样，最初的矩阵 M 可以理解为是在 1 步之内，任意两点的最短距离。正无穷就意味着 1 步之内无法从起始点到达终点。用 M^n 来表示进行 $n-1$ 次迭代后的矩阵。用 m_{ij}^n 来表示迭代后矩阵 M^n 中的行 i 列 j 元素，即从 i 至 j 走 n 步的最短距离。

这样对于一个 $p \times p$ 的矩阵，进行如下运算就可得到任意两点的最短路径：

$$m_{ij}^{n+1} = \min(m_{ik}^n + m_{kj}^1) \quad (k=1..p)$$

为什么这样计算可以得到最短的路线呢？这里的 m_{ij}^n 表示 n 步从 i 到 j 的最短距离， m_{kj}^1 是 1 步从 k 走至 j 的最短距离，那么 $m_{ik}^n + m_{kj}^1$ 就表示走 $n+1$ 步经过第 k 个位置的距离，在这些所有的中取得最短即为从 i 到 j 走 $n+1$ 步最短。

以两步以 A 为起点，计算出 2 步内到达 D 的距离。

$$\begin{aligned}
m_{14}^2 &= \min(m_{11}^1 + m_{14}^1, m_{12}^1 + m_{24}^1, m_{13}^1 + m_{34}^1, m_{14}^1 + m_{44}^1, m_{13}^1 + m_{34}^1) \\
&= \min(0 + \infty, 2 + \infty, 2 + 1, \infty + 0, 7 + 4) \\
&= \min(\infty, \infty, 3, \infty, 11) \\
&= 3
\end{aligned}$$

通过运算的过程可以看出，实质上是计算 A->A->D, A->B->D, A->C->D, A->D->D, A->E->D 中的最小值。这样，一直保持 m_{ik}^n 最小，然后，所能走的一步亦为最小，所以计算出的下一步也为最小。

需要 List 库中的 transpose 与 minimumBy 函数，transpose:: [a] -> [a] 函数可以对矩阵进行翻转。例如：

```
> :m +Data.List
> transpose [[1..4],[5..8]]
[[1,5],[2,6],[3,7],[4,8]]
```

而 minimumBy:: (a -> a -> Ordering) -> [a] -> a 则可以使用给定的方法来得出一个列表中的最小值。例如，根据二元元组的第一个元素选出列表中的最小值：

```
> :m +Data.Ord
> minimumBy (comparing fst) [(1,2),(2,3)]
(1,2)
```

上面用到的 comparing 函数在 Ord 库中，它的类型是 comparing :: Ord a => (b -> a) -> b -> b -> Ordering, comparing fst 就是对比较方法的声明。

要导入它们：

```
import Data.List (transpose,minimumBy)
import Data.Ord (comparing)
```

可以用列表来实现，为了方便简单直观，先绑定一些类型。Distance 表示距离，这里用 Double 小数来表示：

```
type Distance = Double
```

下面，绑定 Name 与 String 来表示地点的名称：

```
type Name = String
```

走的方向也是一个字符串，如“A->B->C”，用 String 来表示。在现实中，不但想计算出最短的距离，还想记录最短距离的路线。

```
type Direction = String
```

一个带有距离的线段称为权重。这里用一个元组来表示，前边是距离，后边则是走这条最短路径的路线：

```
type Weight = (Distance, Direction)
```

在一个矩阵中输入 (5.0, "Alice's home->Blair's home") 是有些麻烦的，可以定义一个函数将第 1 步的矩阵与地点的名字进行绑定，变成一个元组。

首先是一个辅助函数，分别以目的地、某一行的起始离目的地的距离、所有名字为参数，函数得到一个 Weight 的列表：

```
zipD :: [Name] -> [[String]]
zipD ns = [(start++->"++des") | des <- ns] | start <- ns

zipW :: [[Distance]] -> [Name] -> [[Weight]]
zipW ds ns = [zip d n | (d, n) <- zip ds (zipD ns)]
```

为了得到两个元素中距离之和与路线，再定义一个函数 tuplePlus，将第一个元件相加，然后把路线相连接，加在 m_k^n 路线的后边。这个运算符定义的是公式 $m_y^{n+1} = \min(m_x^n + m_y^1)$ 中 $m_x^n + m_y^1$ 的加号。

```
tuplePlus :: Weight -> Weight -> Weight
tuplePlus (d1, n1) (d2, n2) = (d1+d2, n1++destination)
  where (from, destination) = break (=='-') n2

> tuplePlus (5, "A->B") (10, "B->C->D")
(15.0, "A->B->C->D")
```

有了以上的函数，下面来定义算法的核心部分。计算最短路径公式 $m_y^{n+1} = \min(m_x^n + m_y^1)$ ，其实就是两个列表中的元素分别相应地使用 tuplePlus 运算，然后，以元组中的第一个元件，也就是两点间的距离为标准选出最小的即可，借助 comparing 函数就很容易了：

```
type RouteMap = [[Weight]]

step :: RouteMap -> RouteMap -> RouteMap
step a b = [[minimumBy (comparing fst) $ zipWith tuplePlus ar bc | bc <- transpose b] | ar <- a]
```

对比矩阵乘法的定义可以看到，这个公式实际上仅仅是矩阵乘法的变种：

```
infixl 5 `*`

(`*`) :: Num a => [[a]] -> [[a]] -> [[a]]
(`*`) a b = [[sum $ zipWith (*) ar bc | bc <- transpose b] | ar <- a]
```

矩阵乘法中的求和运算 sum 被替换成求最小值运算 minimum，乘法运算 (*) 被替换成 tuplePlus 运算。

接下来要做的，就是将这个计算重复多次，看我们可以走到哪里，先来定义一个迭代的函数，使得这个算法可以自动迭代一个一元函数多次。

```
iteration :: Int -> (a -> a) -> a -> a

iteration 0 f x = x
iteration n f x = iteration (n-1) f (f x)
steps :: Int -> RouteMap -> RouteMap
steps n route = iteration n (step route) route
```

`steps` 函数以走的步数和 M^1 为参数，最后求得 M^{n+1} 。

在前边，可以自定义走一定的步数，得到某一步数内任意两点的最短距离。但是，往往想要计算机自动计算好最短距离，直到再计算矩阵不会再发生变化为止，通过学过的不动点函数可以很容易做到。由此，又要定义一个不动点函数。这个不动点函数有些特殊，因为随着计算，虽然最短路线的距离不变了，但是路线还会发生变化，所以在计算不动点的时候，要验证下一次的距离列表与此次的距离列表相等，故要将二元组的两个元件用 `unzip` 分离。若两次距离矩阵相等则停止计算，否则继续计算。

```
fix f x = if dss == dss' then x else fix f x'
  where
    x' = f x
    dss = [fst $ unzip ds|ds<-x']
    dss' = [fst $ unzip ds|ds<-x']
```

最后定义 `path` 函数来计算一个图上的所有最短路经：

```
path :: [[Distance]] -> [Name] -> RouteMap
path dis ns = fix (step route) route
  where route = zipW dis ns
```

这样，所有的函数就全部定义完了。最后，可以定义一些数据来测试算法了。因为距离可能是有正无穷的，即在某几步之内无法到达，所以，先定义一下正无穷，Haskell 中是没有直接定义正无穷给用户使用，但是可以通过小数除法 $1/0$ 来得到正无穷。

```
infinity :: Fractional a => a
infinity = 1/0
```

用 `i` 来表示正无穷：

```
i = infinity
```

那么将图所示的地图在 Haskell 里表示成列表，则有：

```
graph :: [[Distance]]
graph = [[0, 6, 2, i, 7],
          [6, 0, 3, i, i],
          [2, 3, 0, 1, 5],
          [i, i, 1, 0, 4],
          [7, i, 5, 4, 0]]
```

再给这些线段的端点取名字：

```
names = ["A", "B", "C", "D", "E"]
```

下面就可以试用一下函数了：

```
> path graph names
[[ (0.0, "A->A->A"), (5.0, "A->C->B"), (2.0, "A->A->C"), (3.0, "A->C->D"), (7.0, "A->A->E") ],
[(5.0, "B->C->A"), (0.0, "B->B->B"), (3.0, "B->B->C"), (4.0, "B->C->D"), (8.0, "B->C->E") ],
[(2.0, "C->A->A"), (3.0, "C->B->B"), (0.0, "C->C->C"), (1.0, "C->C->D"), (5.0, "C->C->E") ],
[(3.0, "D->C->A"), (4.0, "D->C->B"), (1.0, "D->C->C"), (0.0, "D->D->D"), (4.0, "D->D->E") ],
[(7.0, "E->A->A"), (8.0, "E->C->B"), (5.0, "E->C->C"), (4.0, "E->D->D"), (0.0, "E->E->E") ]]
```

这样，得到从某一点到另一点就很容易了。前面试的地图都太小了，有兴趣的读者可以测试大一点的地图，可以测试有向图，也可以测试无向图。

本节讨论的算法来自（Cormen et al., 2009）第六部分第 25 章第 1 节。

练习

- 写一个函数，需要 4 个参数、图、名字、出发地、目的地，返回一个 Weight 的元组。提示：你可能用到(!!) 函数。
- 这个算法是可以被优化的，当计算出两步的时候，用两个 2 步就可以计算出 4 步的值，然后 8 步的值，这样有助于我们更快地找到最少步数的最短路径。算法只需要改动一下即可：

$$m_g^{2n} = \min(m_k^n + m_{gk}^n) \quad (k=1..p)$$

由于有了前边的铺垫，因此代码书写起来十分容易。有兴趣的读者可以自己定义。本节到这里就结束了，相信读者对列表，应该有了非常好的理解。

本章小结

本章主要学习了列表内包的使用。可以看到，列表内容在使用起来非常地清晰明了。在生成各种列表时，有列表内包还有基于列表的递归我们可以很轻易地应对各种问题，这要归功于列表的灵活性还有 Haskell 对列表内包定制的精致的语法了。这种表达法在后面的内容中会常常遇到，读者一定要学会熟练地使用。

第 7 章

高阶函数与复合函数

在这一章中，不再探讨那些数学及算法上的问题，而是对编写过和库中的函数进行一下小结，并深入了解一些复杂的函数，这些函数是一些以很多其他函数为参数的函数，或者以函数作为结果返回的函数，也就是高阶函数。由于以函数作为返回结果只是柯里化函数的特点而已，所以高阶函数常常特指那些以函数为参数的函数。理解这些函数有助于写出更为精炼的代码，同时可以让人们省去很多重复的工作。

高阶函数是一些以其他函数作为参数或者返回其他函数作为结果的函数。Haskell 的函数与顺序式语言的函数有所不同，Haskell 中的函数和值没有任何区别，只是它是需要一个或者多个参数的值。

7.1 简单高阶函数

其实前面介绍过的很多函数都是高阶函数，比如，`map` 以一个类型为 `a -> b` 的函数和类型为 `[a]` 的列表为参数，返回以 `[b]` 为类型的列表。`span` 函数需要一个以 `a -> Bool` 为类型的函数作为参数等。

此外，我们知道非柯里化函数与柯里化函数等价的，那么它们之间则是可以互相转化的，其中转化的函数的 `curry` 与 `uncurry` 很明显都是高阶函数。读者可以试着去定义它们，这里我们只考虑二元函数，那么它们的类型为：

```
curry :: ((a,b) -> c) -> (a -> b -> c)
uncurry :: (a -> b -> c) -> ((a,b) -> c)
```

高阶函数常常是某一些有共同特点的函数更为一般的形式。比如，将一个列表中的所有数字平方：

```
power2 :: Num a => [a] -> [a]
power2 [] = []
power2 (x:xs) = x^2 : power2 xs
```

或者，将所有数字都加 1 的函数：

```
plus1 :: Num a => [a] -> [a]
plus1 [] = []
plus1 (x:xs) = (x+1) : plus1 xs
```

显然，这两个函数一个等价于 `map (^2)`，而另一个等价于 `map (+1)`。读者可以在 GHCi 中测试这些函数是否会返回一样的结果。这样，`map` 则是可以表达 `plus1` 与 `power1` 这一类函数的更为一般的函数。

之前在 5.8 中定义的很多种不动点函数，它们都是高阶函数，比如：

```
fix1 :: (a -> a) -> a
fix1 f = f (fix1 f)

fix2 :: (a -> a) -> a
fix2 f x | x == f x = x
          | otherwise = fix2 f (f x)

fix3 :: (t -> t -> Bool) -> (t -> t) -> t -> t
fix3 c f x | c x (f x) = x
          | otherwise = fix3 c f (f x)
```

首先看它们的类型，都是需要类型为 `(a -> a)` 作为参数的函数，其中，`fix3` 还需要一个判定停止条件的函数，在牛顿法开方时用以控制精度。这一类函数就是为了应用一个函数多次而编写的更为一般的函数。

之前在 4.1.2 节中介绍过使用 `iterate` 函数计算等比数列，但是，有时不想得到该数列，而是想得到该数列的某一项，可以这样定义：

```
> let geopro n = last $ take n (iterate (*2) 1)
```

但是，这样定义是不好的，因为计算过程中存储了很多不必要的东西。这里可以写成 `(*2) ((*2) (... ((*2) 1)...))`。显然，我们想求一个某一项不可能写多次 `*2` 来得到结果，这里就可以定义一个函数 `apply`，它将给定的有着 `a -> a` 类型的函数应用给定的次数，正如上一章最短路径中我们可以控制走的步数的 `steps` 函数。

```
apply :: (a -> a) -> Int -> a -> a
apply f 0 x = x
apply f n x = apply f (n-1) (f x)
apply f _ x = x
```

这样，通过应用 `apply` 就可以得到等比数列中的任意一项。

```
> apply (*2) 5 1
32
```

此时，`geopro` 只需要定义为 `\n -> apply (*2) (n-1) 1` 即可。这样，高阶函数就捕捉

到了将一个函数应用多次的这种更为高级的抽象。

找到某些函数更为一般的形式在编程是至关重要的，这使得程序可以更为模块化，很容易提高函数库的质量从而避免重复工作，减少人的工作量，而 Haskell 的高阶函数使得它非常擅长做这样的事情。在 Haskell 库中提供了最重要的几个高阶函数，它为 fold 函数与复合函数，这一章将着重介绍它们。

7.2 折叠函数 foldr 与 foldl

常常，用户需要对一个列表中的元素做一个二元运算符的操作。当列表为空时，返回一个特殊的值。比如 sum 函数，当列表为空时返回 0；否则，递归地使用 sum 函数将第一个元素与后面列表的和相加。同样，product、and、or 函数都是这样的函数，如果给定了这个特殊的元素还有这个二元运算，它们的定义可以被更为一般地写为：

```
f [] = e
f (x:xs) = x ⊕ f xs
```

很多情况下，e 为 x 类型的单位元，加法单位元为 0、乘法单位元为 1，这些将于第 9 章中的单位半群做更为深入的讨论。比如，sum 与 and 可以这样定义：

```
sum [] = 0
sum (x:xs) = x + sum xs

and [] = True
and (x:xs) = x && and xs
```

库函数中的 foldr 函数和 foldl 函数恰恰是这一类函数的一般形式。fold 意为折叠，foldr 意为向右折叠 (fold right)，foldl 意为向左折叠 (fold left)，借助这样一个更为一般的函数 sum 与 and 可以被重新定义为：

```
sum xs = foldr (+) 0 xs
and xs = foldr (λ x y. x && y) True xs
```

虽然现在还没有给出这个高阶函数的定义，但是看到 sum 与 and 函数定义的共同的部分，可以从中推断出如果有这样一个高阶函数 fold，其中 sum 与 and 可以如何通过它来定义。读者可以先试着定义它。

foldr 函数应该是一个以一个右结合的二元函数、一个特殊值 v、一个列表作为输入，将列表内的元素做这个二元操作，把这个列表“折叠”起来的高阶函数。它的定义如下：

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

从 foldr 的定义可以看出它是一个扩展递归函数。因为在递归调用之外还有函数 $f x$ ，所以

它需要向内存的栈中不断叠加递归过程中产生的中间值。比如：

```

foldr (+) 0 [1,2,3]
= 1 + (foldr (+) 0 [2,3])
= 1 + (2 + (foldr (+) 0 [3]))
= 1 + (2 + (3 + (foldr (+) 0 [])))
= 1 + (2 + (3 + (0)))
= 6

```

这样，在做计算的时候会比较消耗存储空间。由于加法是左右均可结合的，因此，先不必考虑结合性的问题。这个折叠函数对列表进行了右结合的折叠计算，所以将这个函数称为 `foldr`。

这里，`foldr ⊕ e [x1,x2,x3 ... xn]`是在计算 $x_1 \oplus (x_2 \oplus (x_3 \oplus \dots (x_n \oplus e)))$ 的值，这里 \oplus 必须为右结合的二元运算符，它的类型需要为 $(a \rightarrow b \rightarrow b)$ 形式，并且在很多情况下， e 是 \oplus 的右单位元。

下面，可以用 `foldr` 来定义列表连接运算符：

```

(++) :: [a] -> [a] -> [a]
(++) = foldr (:)

```

这个函数通过 `foldr` 将第二个列表中的值一个一个地加在第一个列表的前面。

```

> foldr (:) [1,2,3] [4,5,6]
[4,5,6,1,2,3]

```

如果想加在后面也很容易，只需要用 `flip` 函数即可。之前给出的例子大多是左右均可结合的运算符。现在，给出一个仅为右结合的二元函数作为例子。在排序算法中，插入排序函数 `isort` 可以通过使用 `foldr` 函数来折叠 `insert` 函数得到。

```

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]
isort xs = foldr insert [] xs

```

对 `isort` 的计算过程进行演绎：

```

isort [4,2,1,5]
= foldr insert [] [4,2,1,5]
= insert 4 (foldr insert [] [2,1,5])
= insert 4 (insert 2 (foldr insert [] [1,5]))
= insert 4 (insert 2 (insert 1 (foldr insert [] [5])))
= insert 4 (insert 2 (insert 1 (insert 5 (foldr insert [] []))))
= insert 4 (insert 2 (insert 1 (insert 5 [])))
= insert 4 (insert 2 (insert 1 [5]))
= insert 4 (insert 2 [1,5])
= insert 4 [1,2,5]
= [1,2,4,5]

```

下面，再给出一个小问题来练习一下 foldr 函数。定义一个函数 compress 将一个有多个连续的元素的列表压缩为一个没有连续相同元素的列表，比如：

```
> compress [1,1,1,3,5,5,5]
[1,3,5]
```

为了解决这个问题，先定义一个二元函数 skip :: a -> [a] -> [a]，若列表首元素与第一个参数相同则不会插入，否则与：函数相同。

```
> skip 3 [1,2]
[3,1,2]

> skip 1 [1,2]
[1,2]
```

它很容易定义。

```
skip :: a -> [a] -> [a]
skip x [] = [x]
skip x (y:ys) | x == y = (y:ys)
| otherwise = x:y:ys
```

这样，compress 函数可以定义为：

```
compress :: Eq a => [a] -> [a]
compress = foldr skip []
```

在计算的整个过程中，skip 这个函数将会像 foldr insert 那样展开得到 skip a1 (skip ..(skip an [])..)。比如：

```
compress [1,1,2,2]
= foldr skip [] [1,1,2,2]
= skip 1 (foldr skip [] [1,2,2])
= skip 1 (skip 1 (foldr skip [] [2,2]))
= skip 1 (skip 1 (skip 2 (foldr skip [] [2])))
= skip 1 (skip 1 (skip 2 (skip 2 (foldr skip [] []))))
= skip 1 (skip 1 (skip 2 (skip 2 [])))
= skip 1 (skip 1 (skip 2 [2]))
= skip 1 (skip 1 [2])
= skip 1 [1,2]
= [1,2]
```

除此之外，还有很多这样的函数可以使用这样的定义。比如，将一个元素放于列表的最后的函数 snoc 可以定义为：

```
snoc :: a -> [a] -> [a]
snoc x = foldr (:) [x]
```

这样，库函数中的(++)就可以定义为 foldr snoc，将第二个列表的元素依次地放在第一个列表的末尾。

个列表的后面。有了`(++)`，可以定义`concat`函数，因为`concat`的意义就是将一个列表中的列表使用`(++)`连接起来，就像把一个自然数列表中的数字用加号连接起来一样。所以，`concat`可以定义为：

```
concat :: [[a]] -> [a]
concat = foldr (++) []
```

再比如，预加载库函数中的`map`函数也可用与`sum`一样形式的递归方法来定义：

```
map f [] = []
map f (x:xs) = f x : map f xs
```

从`map`的定义可以看出，`map f`的定义与之前所定义的`sum`的结构是没有区别的，仅仅是第一个元素需要应用一次`f`，然后`sum`中的加法对应`(::)`运算符，那么，它也可以用`foldr`来定义：

```
map' :: (a -> b) -> [a] -> [b]
map' f = foldr (\l1 ls -> f l1 : ls) []
```

有兴趣的读者可以演绎一下`snoc`与`map'`的计算过程，这里就不再多讲了。如果这个二元运算是左结合的，可以用`foldl`来计算它，`foldl`函数是这样定义的：

```
foldl :: (a -> b -> a) -> a -> [b] -> a2
foldl f v [] = v
foldl f v (x:xs) = foldl f (f v x) xs
```

很明显，`foldl`是尾递归定义的，在计算的过程中，很明显就能看到这一点：

```
foldl (+) 0 [1,2,3]
= foldl (+) (0+1) [2,3]
= foldl (+) ((0+1)+2) [3]
= foldl (+) (((0+1)+2)+3) []
= (((0+1)+2)+3)
= 6
```

其实，实际的计算过程要比写在上面的复杂很多，但是在这里，不必太过追究。

`foldl ⊕ e [x1, x2, x3...xn]`是在计算`(..((e⊕x1)⊕x2)...⊕xn)`的值，这里，`⊕`必须为左结合的二元运算符，它的类型应该满足`(a -> b -> a)`，常常`e`为`⊕`运算符的左单元。

但是，由于惰性求值，这样的尾递归定义还是会消耗过多的存储单元，可以使用`foldl`的严格求值版本`foldl'`，它在`Data.List`库中。`foldl'`在计算的过程大致如下：

```
foldl' (+) 0 [1,2,3]
= foldl' (+) (0+1) [2,3]
= foldl' (+) (1+2) [3]
= foldl' (+) (3+3) []
= 6
```

下面，用`foldl`来定义`reverse`函数：

```

reverse' :: [a] -> [a]
reverse' = foldl (flip (:)) []

reverse' [1,2,3]
= foldl (flip (:)) [] [1,2,3]
= foldl (flip (:)) (flip (:) [] 1) [2,3]
= foldl (flip (:)) (flip (:) [1] 2) [3]
= foldl (flip (:)) (flip (:) [2,1] 3) []
= flip (:) [2,1] 3
= [3,2,1]

```

使用 foldl 时，其中的二元运算符需要是左结合的，而使用 foldr，则二元运算符需要为右结合的。下面介绍一下 foldl1 和 foldr1，在调用这两个函数前需要保证列表中至少有个一元素，所以函数命名时后边有 1，foldr1 的定义为：

```

foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 _ [x]      = x
foldr1 f (x:xs)  = f x (foldr1 f xs)
foldr1 _ _        = error "foldr1 empty list"

```

比如，unword 函数则可以用 ($\backslash w\ s \rightarrow w\ ++\ '':s$) 函数来定义：

```

unwords :: [String] -> String
unwords [] = ""
unwords ws = foldr1 (\w s -> w ++ '':s) ws

unwords ["s1","s2"]
let f = (\w s -> w ++ '':s)
= foldr1 (\w s -> w ++ '':s) ["s1","s2"]
= foldr1 (\w s -> w ++ '':s) "s1" ["s2"]
= f "s1" (foldr1 f ["s2"])
= f "s1" "s2"
= (\w s -> w ++ '':s) "s1" "s2"
= "s1"++ '': "s2"
= "s1 s2"

```

foldl 对应的 foldl1 函数定义如下：

```

foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)  = foldl f x xs
foldl1 _ [] = error "foldl1 empty list"

```

Data.List 库中也有对应的严格求值版本 foldl1'。有了 foldl1 就可以定义很多函数了，比如，Prelude 中的 maximum、minimum 函数，求出一个列表的最大值与最小值函数：

```

maximum', minimum' :: Ord a => [a] -> a
maximum' = foldl1 max
minimum' = foldl1 min

```

有了这两个函数，可以很容易地判断某两个列表中的元素是否全部相等，即它的最大值是否等于最小值，即 `constant xs = minimum xs == maximum xs.`

有很多两元运算都可以通过 `fold` 相关函数定义，比如求一列数的最大公约数函数和最小公倍数的函数：

```
gcds = foldrl gcd
lcms = foldrl lcm

> foldrl gcd [72,180,144]
36
```

虽然 `scanl` 与 `scanr` 函数同 `foldl` 与 `foldr` 函数相似，但是它会记录二元函数计算的中间结果。

```
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanl f a [] = [a]
scanl f a (x:xs) = a: scanl f (f a x) xs

> scanl (+) 0 [1,2,3]
[0,1,3,6]
```

这里就不再讨论 `scanr::: (a -> b -> b) -> b -> [a] -> [b]` 了，有兴趣的读者可以试着自己定义这个函数，然后可以到文档中查找它的定义来检查自己定义与库中的是否一致。

`foldr` 与 `foldl` 是基于列表这种数据结构的基本函数，也就是说，通过它可以定义出其他所有的基于列表的函数，如 `filter`、`concatMap` 等，有兴趣的读者可以尝试着定义一下这些函数。`foldr` 与 `foldl` 之间可以相互转换，因为运算符的结合性可以通过 `flip` 来改变，这样对于一个函数来说，有一个 `foldr` 的定义就会有一个对应的 `foldl` 定义，反之亦然，第 9 章中将会讨论这个性质。

理论上，可以得到所有的组合，但是这种表达方式只可能遍历以 1 为第一个元件的元组，即 `[(1,1), (1,2), (1,3), (1,4), (1,5)...]`，也就是图 7-1 中的第一行，那么对于(2,1)将不会被遍历，也就是说，我们无法给定一个自然数 n ，这个自然数 n 是(2,1)在列表中所在的位置。

	1	2	3	4	5	...
1	(1, 1)	(2, 1)	(3, 1)	(4, 1)	(5, 1)	...
2	(1, 2)	(2, 2)	(3, 2)	(4, 2)	(5, 2)	...
3	(1, 3)	(2, 3)	(3, 3)	(4, 3)	(5, 3)	...
4	(1, 4)	(2, 4)	(3, 4)	(4, 4)	(5, 4)	...
5	(1, 5)	(2, 5)	(3, 5)	(4, 5)	(5, 5)	...
6	(1, 6)	(2, 6)	(3, 6)	(4, 6)	(5, 6)	...
...

图 7-1 整数对统计表

请找到一种方法，一个不落地生成所有整数对的组合，即对于任何给定的一对整数，它一定会在定义的列表中的某一个确定的位置出现。

提示 使用练习 1 的 interLeave 函数与 foldr 函数定义一个函数 interLeaveLists，它与 concat 功能相似。再使用 interLeaveLists 对 $[(x,y) \mid y < [1..] \mid x < [1..]]$ 进行连接。关于更多内容可以参阅 http://en.wikipedia.org/wiki/Rational_number.

练习

1. 定义一个类似于 concat 的函数 interLeave，这个函数可以将两个无穷的列表合在一起。比如：

```
> interLeave [1..] [100..]
[1,100,2,101...]
```
2. 当给定两个无限的列表时，需要遍历其中所有两两的二元元组，比如：

```
> [(x,y) | x < [1..], y < [1..]]
```
3. 定义一个函数 removeDup :: Eq a => [a] -> [a]，它可以移除一个列表重复的元素。（提示：先定义一个函数 removeOne :: Eq a => a -> [a] -> [a]，再使用 foldr 定义 removeDup，这与之前的 skip 函数十分相似。）

7.3 mapAccumL 与 mapAccumR 函数

Data.List 中还提供了 mapAccumL 与 mapAccumR 函数，mapAccumL 是这样定义的：

```
mapAccumL :: (acc -> x -> (acc, y)) -> acc -> [x] -> (acc, [y])
mapAccumL _ s [] = (s, [])
mapAccumL f s (x:xs) = (s'',ys)
  where (s', y) = f s x
        (s'',ys) = mapAccumL f s' xs
```

这两个函数可以理解为，在计算的时候常常需要返回一些状态。比如，在将某个列表在求和的过程中，记录当前和的奇偶性：

```
>:m +Data.List
> mapAccumL (\sum -> \a -> (sum + a, even (sum + a))) 0 [1,3,4,5,6]
(19,[False,True,True,False,False])
```

首先，1 不为偶数，故为 False，1+3 为偶数，故为 True，然后依次类推。这样。不但计算了列表的和，同时还记录求和中间临时的结果是否为偶数。

mapAccumR 与 mapAccumL 是很相似的，除了在累计的过程中 mapAccumR 是由右至左的，而 mapAccumL 是由左至右的。

```
mapAccumR :: (acc -> x -> (acc, y)) -> acc -> [x] -> (acc, [y])
mapAccumR _ s [] = (s, [])
```

```

mapAccumR f s (x:xs) = (s'', y:ys)
  where (s'', y) = f s' x
        (s', ys) = mapAccumR f s xs

> mapAccumR (\sum -> \a -> (sum + a, (sum + a))) 0 [1,3,4,5]
(13, [13,12,9,5])

```

Data.List 中还有很多非常实用的函数，如 subsequences、intersect、(\ \)、union 等，希望读者可以通过 API 来了解它们。

7.4 复合函数

数学中常常用到复合函数（function composition），比如 $f(x)=4x+1$, $g(x)=x^2+1$ ，那么， $f(x)$ 复合 $g(x)$ 的函数 $h(x)=f(g(x))$ ，即 $h(x)=4g(x)+1=4(x^2+1)+1$ ，也就是说，先要求得函数 g 的结果，然后再传递给函数 f 。在 Haskell 中将这些函数定义如下：

```

f,g,h :: Num a => a -> a
f x = 4*x+1
g x = x^2+1

```

$h x = f (g x)$ 也可以写为 $f \circ g x$ 。

Haskell 中提供复合函数运算符 $(.)$ 来复合两个函数， η 化简后， h 可以直接写做 $h = f . g$ ，有时书面上也写为 $f \circ g$ 。

这样，在表达函数的时候可以更为精简。下面来看一下复合函数运算符的定义：

```

(.) :: (b -> c) -> (a -> b) -> (a -> c)
(.) f g = \x -> f (g x)

```

也就是说，定义 $(f.g) x$ 与 $f (g x)$ 是一样的。定义中的 $(b -> c)$ 类型对应函数 f ， $(a -> b)$ 类型对应函数 g ，最后的类型 a 对应 λx 。第一次见到这个类型可能感觉有些奇怪，理解了 $->$ 为右结合与函数应用为左结合这其实就并不奇怪了，因为通过复合函数运算符复合在一起的函数在计算时是从右至左的。比如 $f.g.h x$ ，这里函数计算的优先级比运算符高，那么， $h x$ 首先被计算然后传入 g ，然后传入 f ， $(.)$ 的参数的顺序与计算是顺序是反的，所以类型签名才会显得有些怪异。可以用 $flip$ 定义一个反置参数输入的顺序，这里使用和 F# 中相同的 $(>>)$ 作为类似管道的运算符，在 Haskell 中有时使用 $(>>>)$ 作为这个运算符的符号：

```

infix 9 >>
(>>) :: (a -> b) -> (b -> c) -> (a -> c)
(>>) = flip (.)

```

使用 $>>$ 来复合函数，参数则会从左至右地依次计算。这样则会看到与类型与函数会对应好，比如：

```

> ((`div` 2) >> even >> not) 4
False

```

如果这里将函数看做“黑匣子”，那么这3个“黑匣子”应该是以图7-2所示的方式连接的，但我们还是需要将4放在最右边。

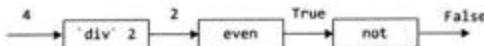


图7-2 |>接连接函数示意图

首先，除以2，然后判断是否为偶数，最后取这个布尔值的反。如果使用(.)，则为（见图7-3）：

```
> (not.even.(`div` 2)) 4
False
```



图7-3 (.)复合函数示意图

此时参数需要写在函数体之后，怎样写更好，读者可能有自己的看法，但是Haskell中常用的还是复合函数运算符。有些人为了把参数写在函数体的前面，以便更符合直觉，所以定义了这样一个有趣的类似管道的运算符：

```
(|>) :: b -> (b -> c) -> c
(|>) = flip ($)

> 4 |> div 8 |> even |> not
False
```

之前，读者已经了解过any与all函数如何用来判定一个列表中的元素是否存在或者全部符合某一条件，使用复合函数可以像下面那样定义它们。首先，使用map函数将判定条件映射到每个元素上得一布尔值的列表，再来判定这些列表中是否有True或者全为True即可。

```
any, all :: (a -> Bool) -> [a] -> Bool
any p = or . map p
all p = and . map p

map p :: [a] -> [a]
or :: [a] -> Bool
```

(|>)这个运算符被预定义在了F#中，在F#中可以直接使用。这个运算符把参数4输出到div 8这个函数得到2，再把2输入到even函数中得到True，最后将True输出给not，得false。这样从左到右的书写的确更符合人的直觉。这个运算符的定义也更接近图7-2。

根据map p与or的类型，可以看到这样定义是符合(.)类型定义的。接下来看一组略复杂的使用复合函数定义的函数。elem与notElem用来判定一个元素是否在一个列表之中，它是通过复合any、all与(==)、(/=)定义的函数。

```
elem, notElem :: Eq a => a -> [a] -> Bool
elem     = any . (==)
```

```
notElem = all . (/=)
--notElem x = not.elem x
```

(==) 符号得一个以类型为 $a \rightarrow \text{Bool}$ 的函数，而 any 以这样一个类型的函数作为参数，所以可以复合在一起：

```
(==) :: Eq a => a -> (a -> Bool)
any :: (a -> Bool) -> ([a] -> Bool)
```

我们可以推理一下这样定义的 elem 是如何工作的：

```
(any . (==)) 5 [1..10]
=\x -> any (x==) 5 [1..10] {复合函数的定义}
=any (5==) [1..10] {β化简}
```

concatMap 函数是很有用的，它的作用是将一个类型为 $[a]$ 用一个函数 f 映射成有着 $[[b]]$ 类型的列表，然后再对其使用 concat。使用 foldr 函数与复合函数可以非常容易地定义它。它的定义等价于 $\lambda f \, xs \rightarrow concat (map f \, xs)$ ，而 concat 是通过 $(++)$ 定义，也就是在应用 $(++)$ 前再用函数 f 映射一次，这其实就是 $(++)$ 与 f 函数的复合。所以可以写为：

```
concatMap :: (a -> [b]) -> [a] -> [[b]]
concatMap f = foldr ((++) . f) []
```

读者可以取 f 为 replicate 3 为例，演绎一下这个 concatMap 函数计算的过程。

练习

不使用 GHCi 尝试自己推断出下面函数的类型：foldl、foldr、foldl Const、foldr Const、(.)、(.)(.)、(.)、(.) map map、map.map。最后使用 GHCi 来看你写出的类型是否正确。

本章小结

高阶函数是函数式编程语言最重要的特性之一，由于函数可以当做参数来传递，无论在解决问题时还是在开发库函数时都将具有很高的灵活性。读到本章最后一节时，细心的读者可能会感觉到，在 7.4 节复合函数中的 any 与 all 等函数是通过 map、or、and 来定义的，而 map、or、and 又都是可以通过 fold 函数定义的，那么说明 any、all 等函数也可以通过 fold 函数来定义，而 elem 与 notelem 函数则是通过 any 与 all 来定义的，这直接说明了它们都可以用 fold 函数定义出来，这也侧面反映了 fold 函数在列表这种结构上具有极高的一般性。可以使用 fold 定义出来的函数称为基本递归函数（primitive recursive function），有兴趣的读者可以查阅相关的资料。

第 8 章

定义数据类型

前面的几章用到了各种数据类型，如布尔类型、整数类型、列表等。在本章中，将为读者讲解如何在 Haskell 中定义数据类型。

定义新的类型主要是使用 `data` 与 `newtype` 关键字。在定义新类型时，可以使用很多种不同的方式，在本章里会一一讨论。定义了新的类型后会发现，这些类型可能会与读者曾经用过的类型是等价的，这种等价称为同构（isomorphism）。同构关系的引入有助于了解类型的本质。不仅如此，知道类型是如何定义的可以更为清楚地理解数学归纳法的本质，理解为什么它的证明过程是有效的，它是证明中的重要的手段之一，在验证程序的正确性时十分重要。

定义新的数据结构与类型，其实是为了更好地解决现实中的问题。有些问题用特定的数据结构来解决十分方便并且高效，而用其他的方法可能会很低效，所以在解决具体问题时，如何定制类型就显得非常 important 了。在本节里，通过用定制的类型解决霍夫曼编码问题与 24 点游戏来加深对类型的认识。

8.1 数据类型的定义

8.1.1 枚举类型

布尔类型在 Haskell 中就是枚举定义的，这个类型里只有两个值，即 `True` 与 `False`。在 `Prelude` 中它是这样定义的：

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

`data` 是定义新类型的关键字，后边跟想要定义的类型名称，类型名称起始字母要大写。然后定义布尔类型的值，即 `True` 和 `False`。

布尔类型是一个非常简单的类型，这个类型只可能有两个值，也可称为两种形式，即 `True` 或者 `False`。这里只需要将它们依次写出，用 “|” 分开即可。这种定义的方式可以被称为枚举定义，即定义的值可以以一定的顺序一一地枚举出来。模式匹配就是匹配的 “|” 之间不同的定义模式这与 C 语言和 Java 语言中用 `enum` 定义类型十分相似，但是在后面我们会看到 Haskell 中的 `data` 关键字要比 C 语言和 Java 语言中的 `enum` 强大的多。

再比如，星期是可以枚举的，从星期一到星期日，只有 7 个不同的值，下面来定义星期：

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

当在 GHCi 中输入：

```
> True  
True
```

由于 Prelude 中 `Show` 类型类里定义了如何输出 `Bool` 类型的值，但是对于 `Day` 类型：

```
>Mon  
<interactive>:1:1:  
  No instance for (Show Day)  
    arising from a use of 'print'  
  Possible fix: add an instance declaration for (Show Day)  
  In a stmt of an interactive GHCi command: print it
```

得到了以上的错误信息，简单的说，就是 Haskell 知道 `Mon` 是一个 `Day` 类型的值，但是却不知道如何来打印这个类型的值，因为没有将 `Day` 实现为 `Show` 类型类的实例。这里可以先使用 `deriving` 关键字来让这个类型的数据自动实现一些类型类，比如 `Show`、`Eq`、`Enum`、`Ord` 等，使用这个关键字后 Haskell 会自动导出定义在类型类中的对应的函数，这些类型类中定义了哪些函数将于下章着重介绍。

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun deriving (Show, Eq, Ord, Enum)
```

这样，在 GHCi 里输入 `Mon`，就会有打印的结果输出了：

```
> Mon  
Mon
```

同时，实现了相等类型类 `Eq`，所以可以比较两个值是否相等：

```
> Mon == Mon  
True
```

除了相等类型类 `Eq`，还实现了基于它的有序类型类 `Ord`，这样就可以比较这个类型值之间的大小：

```
> Mon < Sun  
True
```

由于有序类型类 `ord` 是基于相等类型类 `Eq` 的，因此一定要实现相等类型类 `Eq` 后才能实现

有序类型类 `Ord`。

由于使用 `deriving` 还实现了 `Enum` 类型类，因此可以使用..遍历定义的类型中的值：

```
> [Mon .. Sun]
[Mon, Tue, Wed, Thu, Fri, Sat, Sun]
```

相信读者已经熟悉了用模式匹配来定义函数。事实上，如何模式匹配跟所用数据的定义有关。`data Day` 后的定义就是模式匹配中需要匹配的形式了，比如，要写一个函数叫“明天”就可以用模式匹配来书写：

```
tomorrow :: Day -> Day
tomorrow Mon = Tue
tomorrow Tue = Wed
tomorrow Wed = Thu
tomorrow Thu = Fri
tomorrow Fri = Sat
tomorrow Sat = Sun
tomorrow Sun = Mon
```

这样，可以看到之前做了很多无谓的工作。其实，实现了枚举类型类 `Enum` 后，有很多函数可以立即使用，如 `succ`、`pred` 及其他的函数，这里只简单介绍 `succ`（successor 的简写）函数，它返回参数的下一个枚举类型的值，而 `pred`（predecessor 的简写）会返回给定参数的前一个枚举类型的值。如果给定的参数已经为边界值，则会出现异常，但这种异常很好处理。“明天”函数可以这样定义：

```
tomorrow Sun = Mon
tomorrow d = succ d
```

因为定义的星期类型导出了 `Enum` 类型类，那么它就有有着可以遍历的性质。这样的设计是不是帮助省了很多不必要的工作呢？类似地，“昨天”函数可这样定义：

```
yesterday Mon = Sun
yesterday d = pred d
```

除了以上的 4 个比较常用的类型类，还有可读类型类 `Read`，即可以使用 `read` 函数来将一个字符串读成 `Day` 的类型的数据。

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun deriving (Show, Read)
> read "Mon" :: Day
Mon
```

这样，就将“Mon”这个字符串读成了 `Day` 的类型的数据。

值得一提的是，如果一个数据类型 `a` 是可读的，另一个 `b` 类型也可读，那么依赖于 `a` 的类型 `b` 也是可读的。比如，数据 `Day` 是可读的，基于 `Day` 的列表 `[Day]` 也是可读的。关于可读类型类 `Read` 将在下一章中介绍。

```
> read "[Mon,Tue]" :: [Day]
[Mon,Tue]
```

一个数据类型是如何定义的，实现了哪些相应的类，这些信息在 GHCi 中是可以查寻的。可以用：info（简写为：i）来查寻一个类型的值的相关信息或者数据类型的相关信息。

比如，我们想查看数据 Tue 的信息：

```
>i Tue
data Day = ... | Tue | ...    -- Defined at Dat.hs:1:18-20
```

这样，就输出了 Tue 这个值是如何定义的，并且也输出了定义的位置，还可以看出 Tue 不是 Day 类型的第一个值，也不是最后一个。

也可以用 info 来查看一个类型的信息：

```
>i Day
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
          -- Defined at Dat.hs:1:6-8
instance Read Day -- Defined at Dat.hs:1:62-65
instance Show Day -- Defined at Dat.hs:1:67-70
```

这里，可以看到 Day 类型是如何定义的，并且 Day 类型实现了 Read 与 Show 类型类，所以是可读的，并且也是可以打印输出的。

Int 类型也可以用枚举类型来定义，但在 Haskell 中它是一个嵌入式的类型。它的具体定义是通过原始类型实现的，需要通过 MagicHash 编译器选项使用 Int#类型，这里不讨论。

别外，在 Haskell 中，有时会需要用到类型 ()。它也被称为单位类型（unit datatype）这个类型只有一个值 ()，可以理解为它是这样定义的：

```
data () = ()    --理想化的定义，不合法
```

它的类型名称与值是一样的。虽然现在看来它好像没有什么用途，但其实它是很有用的，在讨论 Monad 时会用到。此外，在进程同步时也可能用到这类型，但是关于进程同步本书将不讨论。

8.1.2 构造类型

定义有些类型时，类型中有些数据需要一些其他类型作为参数，比如定义“书”这种类型：

```
type Name = String
type Author = String
type ISBN = String
type Price = Float

data Book = Book Name Author ISBN Price deriving (Show, Eq)
```

这里，构造一个类型为 Book 的数据需要 4 个参数，即书名、作者、ISBN 和价格，data Book

中的 Book 是类型的名字（也称为类型构造器，但是这里它没有参数，下一节中介绍参数化类型），而 Book Name Author ISBN Float 中的 Book 被称为数据构造器（data constructor），意思是说，Book 可以构造出 Book 类型的数据的东西。虽然在很多类型的定义中类型的名称与数据构造器的名称是相同的，但绝不能混淆两者，时刻要清楚我们是在讨论类型还是数据。其实，数据构造器的本身是一种特殊的函数，不过首字母会大写。下面来看一下 Book 构造器的类型：

```
>:t Book
Book :: Name -> Author -> ISBN -> Price -> Book
```

通过查看 Book 的类型可以知道，构造器的本质实际上是将那些参数^①作为输入，然后返回一个以类型为 Book 的数据作为结果的函数。

当给定一个 Book 类型的数据，需要得知书的信息，即访问 Book 构造器的参数。如此一来，就要写一些重复的、烦人的函数当做访问器。比如：

```
name  (Book n _ _ _) = n
author (Book _ a _ _) = a
isbn   (Book _ _ i _) = i
price   (Book _ _ _ p) = p
```

当构造器中的参数少时还好，但是如果多的话，做这样重复的工作，即浪费时间，也毫无意义。不过，Haskell 语言的设计者们提供了另外一种定义的语法，这个语法使得访问器函数在这个类型定义的同时也被定义出来。

```
data Book = Book {
    name   :: Name,
    author :: Author,
    isbn   :: ISBN,
    price   :: Price
}
```

访问器的名称，如 name、author 等，被称为字段名。在定义的过程中注意，各个部分需要用逗号隔开，访问器函数名后需要指定这一数据的类型。通过使用这样的语法来定义，可以省去了很多时间，提高了写代码的效率。

当定义函数的时候，可能同时用到构造出的值和这个构造器中的参数。这时，可以不用写出这个构造的数据的形式而仅仅是用一个名字指代，然后用访问器函数来得到这个值中的内容，并且用该名字来引用这个构造出的值。用户还可以将这个值的形式写出来，可是当再次用到这个值的时候还要再写一次。这两种定义方式都显得多余而冗长。比如，书店中的一些书要涨价，书店需要分别记录原价和涨价后书的信息，所以需要一个如下类型的函数：

```
increasePrice :: ([Book], [Book]) -> Book -> Float -> ([Book], [Book])
```

如果不给出这个构造的值的形式，仅仅使用访问器，就要定义成下面的形式：

^① 构造器的参数在英文中为 field。

```
incrisePrice (b1,b2) b pri =
  ((b:b1),(Book (name b) (author b) (isbn b) (price b + pri))
```

而当仅仅使用模式匹配来定义时，需要再输入一遍原价时书的信息，如：

```
incrisePrice (b1,b2) (Book nm ath isbn prc) pri =
  ((Book nm ath isbn prc):b1 ,(Book nm ath isbn (prc+pri)):b2)
```

这两种定义都显得冗长。当遇到这样的情况时，可以使用`@`符号来用一个名字来指代这个构造的数据，减少一些不必要的重复，如：

```
incrisePrice (b1,b2) b@ (Book nm ath isbn prc) pri =
  (b:b1,(Book nm ath isbn (prc+pri)):b2)
```

这里，`Book`构造器是需要一些其他参数填充的，而有的时候，构造器与值其实并没有显著的区别，比如，可以认为`True`与`False`就是布尔类型的构造器，它们不需要参数，称为零元数据构造器（nullary data constructor），还有上一节中定义的星期类型也是如此。

8.1.3 参数化类型

参数化类型是一些需要类型参数的类型。Haskell 的预加载库中定义了很多重要的参数化类型，最常用的就是列表了，此外还有`Maybe`和`Either`。列表的定义更为复杂一些，稍后再进行讨论。本节中，先来看一些简单的参数化类型：`Maybe`和`Either`。参数化类型的定义中的类型名称后可以加一个任意的类型参数以供这个类型的数据构造器使用，这样的构造器可以暂时简单理解为一个容器，与构造类型不同，这里的参数可以是任何的多态类型，而不是一个确定的类型。

```
data Maybe a = Nothing | Just a
```

`Maybe`可以理解为一种容器类型，里面可以放一些东西在里面。`Just`内的类型`a`可以是任意的，可以是`Float`，也可以是`Double`，还可以是元组，甚至也可以是函数类型。这里`Just`就是一个构造器，与前一节的`Book`类型不同的是这个构造器是一个多态类型的函数，它的类型是`a -> Maybe a`。查询一下`Maybe`的信息：

```
>:i Maybe
data Maybe a = Nothing | Just a    -- Defined in Data.Maybe
instance Eq a => Eq (Maybe a) -- Defined in Data.Maybe
instance Monad Maybe -- Defined in Data.Maybe
instance Functor Maybe -- Defined in Data.Maybe
instance Ord a => Ord (Maybe a) -- Defined in Data.Maybe
instance Read a => Read (Maybe a) -- Defined in GHC.Read
instance Show a => Show (Maybe a) -- Defined in GHC.Show
```

从以上的信息可以看出，`Eq a => Eq (Maybe a)`。`Nothing`和`Nothing`是一定相等的，而当容器中的类型`a`是可以比较相等的，那么`Maybe`也是可以比较相等的类型。同理，`Maybe`也是可以比较大小，当`a`类型是一个有序的类型，那么`Just`也可以比较大小并且`Nothing`要小于`Just a`。而

当 a 不是一个有序的类型，Nothing 与 Just 是无法比较大小的。比如，函数是无序的类型，那么：

```
> Nothing < Just (+3)
<interactive>:1:9:
    No instance for (Ord (a0 -> a0))
      arising from a use of '<'
    Possible fix: add an instance declaration for (Ord (a0 -> a0))
    In the expression: Nothing < Just (+ 1)
    In an equation for 'it': it = Nothing < Just (+ 1)
```

这个错误的意思是说，函数的类型没有实现有序类，所以不能使用“<”进行比较。在任何语言中函数都是无法直接比较相等还有大小的。

同时，Maybe 实现了 Read 类，也就是说，Maybe 可以从字符串读入。如果 a 类型是可读的，那么 Maybe a 也是可读的。例如：

```
>read "Just 5" :: Maybe Int
Just 5
```

同样地，如果 a 类型是可显示的，那么 Maybe a 也是可显示的。至于信息中的 Functor 和 Monad，将于后面的章节进行讲解，这里先不探讨。

Maybe 类型有什么作用呢？简单地说，Maybe 是一种简单的程序处理异常的方式。因为有些时候，程序需要带着出现的异常继续运行，而不是简单地就抛出异常然后终止。当出现异常的时候，返回 Nothing，比如，head [] 的时候就抛出异常。

```
>head []
*** Exception: Prelude.head: empty list
```

这样，可以用 Maybe 来写一个不出现异常的、安全的 head 函数：

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

在写其他函数调用 safeHead 得出结果的时候，所得结果需要分成 Nothing 和 Just a 两种情况分开考虑，使得函数可以不返回异常，当需要继续计算时，可以对 Nothing 的情况进行处理。由此一来，Maybe 就可以被认为是一种简单的处理异常的机制了。

同样地，如果一个整数除以 0，那么也会发生异常。另外，也可以用 Maybe 来处理，即当有一个整数除以 0 时，返回 Nothing，否则正常输出结果。这样，当程序有异常的时候，并没有终止，而是带着这个异常让程序继续运行。

```
safeDiv :: Integral a => a -> a -> Just a
safeDiv a 0 = Nothing
safeDiv a b = Just (div a b)
```

1. 类型构造器

到这里，相信读者已经对参数化类型的定义有所了解了。比如，Maybe Bool 是一个类型，

而 Nothing 和 Just True 和 Just False 是这个类型中的值，也可称为数据。Just 是一个函数，会将一个类型为 Bool 的数据输入然后输出一个类型为 Maybe Bool 的数据作为结果。之前介绍的运算都是基于数据的，而像 Maybe 需要其他类型作为参数来构造一个新的类型的类型，被称为类型构造器 (type constructor)。

在前面，本书将值归为不同的类型，比如 False 为布尔类型，1 为 Int 类型。这些类型有着一些共同的属性，本书将它们归为了类型类。现在了解了类型构造器，可以将类型划为不同的 kind。类型构造器是基于类型的运算，所有的类型通常写成星号 *，意为所有类型的类型，这里类型构造器的类型就称为 kind。* 是一个零元类型构造器 (nullary type constructor)，即不需要其他类型，自己本身就是一个“完整的”类型。比如，Maybe Bool :: * 与 Maybe Int :: * 都是完整的类型。Haskell 中在 GHCi 里使用 :kind (简写为 :k) 来查询一个类型构造器的 kind：

```
>:k Int
Int :: *
```

而 Maybe 需要另外一个类型来得到一个完整的类型，故它的 kind 为 * -> *，比如：

```
>:k Maybe
Maybe :: * -> *
```

Int 是一个不需要其他类型作为参数的 kind。那么，Maybe Int 有什么样的一个 kind 呢？

```
>:k Maybe Int
Maybe Int :: *
```

相信读者应该没有太多的惊讶，Maybe Int 是一个不需要其他类型输入的 kind。这里的 Int 类型被应用到了 Maybe 构造器，成为它的参数替换了第一个星号从而得到了一个完整的类型，这与函数在应用时，参数的类型替换了函数类型签名中箭头前的类型是一样的道理。

除 Maybe 外，Haskell 预加载库中另外一个重要类型是 Either，称为“或者”类型。它需要两个不同的类型作为输入，组合成一个新的类型。常常同时需要一种类型或者另外一种类型的时候用到。Either 是一种需要两个类型来共同构成的类型构造器。Either 这是这样定义的：

```
data Either a b = Left a | Right b
```

下面我们来查询一下 Either 的 kind 是什么：

```
>:k Either
Either :: * -> * -> *
```

Either 需要两个类型作为参数来返回一个新的类型，所以它的 kind 为 * -> * -> *。在使用列表的时候，不可以放不同的类型，当我们一定需要两种不同的类型时，可以将两种类型组合成一个 Either a b 类型，然后使用列表来存放 Either a b 这个类型的数据。

举一个例子。假设需要按一定的顺序存储学生的成绩，成绩为整数，而有的学生可能没有成绩，没有成绩的学生可能分为很多种情况，有可能学生作弊了，还有可能学生因病缺考了，

总之，可能的情况有很多。这样，可以用 `Left Int` 来记录学生的成绩，而用 `Right String` 来记录学生没有成绩的原因。

```
>:t [Left 80,Right "Cheated",Left 95,Right "Illness"]
[Left 80,Right "Cheated",Left 95,Right "Illness"] :: Num a => [Either a [Char]]
```

这里的 `Right` 和 `Left` 都是构造器，`Either` 也可以理解成为一个容器。它由两种任意的类型组成，这个容器中的内容或者是 `a` 类型，或者是 `b` 类型。相比于 `Maybe` 类型，如果用 `Either` 来处理异常有时是一个更好的选择，因为我们不但可以知道运算过程有异常，还可以存储异常的信息，而 `Maybe` 中 `Nothing` 却无法做到这一点。这里的 `Either` 相当于集合中的不相交并集（disjoint union）。在 Haskell 中，将两个类型可能不同的列表合成一个列表就可以使用这样的函数。

```
disjoint :: [a] -> [b] -> [Either a b]
disjoint as bs = map Left as ++ map Right bs
```

当需要把 `Either` 中的值映射为另一个值时，要为 `Left` 与 `Right` 分别提供一个函数，这两个函数返回的类型相同，在这里都为 `c` 类型：

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f _ (Left x) = f x
either _ g (Right y) = g y
```

也可以将一个 `Either` 类型的列表分成两个列表，这个相当于 `disjoint` 的反函数：

```
partitionEithers :: [Either a b] -> ([a],[b])
partitionEithers = foldr (either left right) ([],[])
where
    left a (l, r) = (a:l, r)
    right a (l, r) = (l, a:r)
```

`Either` 的相关函数可以在 `Data.Either` 中找到，有兴趣的读者可以看一下其中定义了哪些函数。

在声明时类型时，还可以对类型参数进行限定，比如定义形状类型，但是参数必须实现了 `Num` 类型类。

```
data (Num a) => Shape a = Rectangular a a
```

但是 `type` 关键字是不可以这样限定类型参数的。虽然在声明类型时可以加入一些限定，但是在定义计算 `Shape` 类型的函数时，`Num` 也必须在类型签名中声明，否则就会得到错误。

```
area :: Num a => Shape a -> a
area (Rectangular a b) = a * b
```

下面讨论一下 `Pair` 类型，可以将两个类型合在一起：

```
data Pair a b = Pair a b

pfirst (Pair a b) = a
psecond (Pair a b) = b
```

由两个组成类型合并为一个类型的 Pair，其实是与 (a, b) 是等价的。这种等价的关系，将在同构的类型一节中介绍。那么，Haskell 中的二元元组的构造器是什么呢？它写作 $(,)$ 。

```
>:t (,)
(,) :: a -> b -> (a,b)
```

这样， $(1, 2)$ 可以写成 $(,) 1 2$ 。同理，三元元组的构造器为 $(,,)$ 。

```
>:t (,,)
(,,) :: a -> b -> c -> (a,b,c)
```

它们的类型构造器与数据构造器的写法是一样的，正如前面提到过的 Book 那样。

```
>:k (,,)
(,,) :: * -> * -> * -> *
```

2. 函数类型

$a \rightarrow b$ 为一个函数的类型，它也是有类型构造器的。它的类型构造器是 (\rightarrow) ，也可以写做 $(\rightarrow) a b$ 。但它是嵌入在 Haskell 中的，不是直接定义的。它的 kind 为 $* \rightarrow * \rightarrow *$ 。

```
> :k (\rightarrow)
(\rightarrow) :: * -> * -> *
```

可以看到，即便是函数这样的类型也是通过构造来定义的：

```
> :i (\rightarrow)
data (\rightarrow) a b      -- Defined in 'GHC.Prim'
```

但这些都是通过原始类型定义的。一些旧版本的 GHC 中可能会将 (\rightarrow) 的 kind 返回为 $? \rightarrow ? \rightarrow *$ 。可能是由于这个原因，笔者认为，这可能是 GHC 中的一个小错误，但是它与 $* \rightarrow * \rightarrow *$ 的意思是一样的。

8.1.4 递归类型

递归函数的概念已经在前面的章节中做了比较详细的探讨。在本节中，来定义一下递归的类型。什么是一个递归定义类型？就是定义类型时用到了正在定义的该类型本身。

数学中的自然数是可以递归定义的，类型的名字定义可以取为 Nat。首先，自然数是从 0 开始的，0 就是递归定义的基本情况，这里定义为 Zero，然后其他每一个自然数都有一个后继（successor），也就是比当前的数大 1 的数，这个后继也是一个自然数，定义为 Succ Nat——一个自然数的后继。这就是自然数的递归定义，还有基于这种表示法的计算称为皮亚诺算术（Peano arithmetics）。它的意思是说，如果给定一个类型为 Nat 的自然数 a，那么 Succ a 的类型也为 Nat。既然它是自然数的另一种定义，那么就可以在这种结构上做基于自然数的四则运算了。这里应该看得出，递归的基本形式 zero 不是任何自然数的后继。在 Haskell 中，这种自然数可以这样定义：

```
data Nat = Zero | Succ Nat (Show, Eq)
```

在 Nat 的定义里可以看到，在定义 Nat 的时候又用到了 Nat 类型本身。如此一来，根据前文中提到的自然数的属性，就可以用这些数来代替用阿拉伯数字或者二进制表示的自然数，每使用一次 Succ 函数，实际上就是将输入的自然数加 1，如表 8-1 所示。

表 8-1 递归定义自然数与阿拉伯数字对照表

递归定义自然数	对应的阿拉伯数字
Zero	0
Succ Zero	1
Succ (Succ Zero)	2
Succ (Succ (Succ Zero))	3
...	...

下面来定义两个函数，使得定义的自然数和 Int 类型的自然数部分互相转换：

```
natToInt :: Nat -> Int
natToInt Zero = 0
natToInt (Succ n) = 1 + natToInt n

int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

对于整数的模式匹配也可以写为 `int2nat (n+1) = Succ (int2nat n)`。将参数写为 `(n+1)` 是可以的，但是，读者觉得这种定义的方式好不好呢？

既然 Nat 是定义的自然数，那么就可以写一些基于这种定义的函数，比如四则运算。下面来定义基于 Nat 类型的加法，可在第一个参数上进行递归，亦可以在第二个参数上进行递归。当选择在第一个参数上递归时，第一个参数可以分为两种情况，一种是为 zero，这样直接返回第二个参数 n；另外一种情况是这个自然数是 m 的后继，即 `Succ m`，只需要返回 `m` 与 `n` 之和再多 1，即 `Succ (add m n)`，比如， $5 + 3 = (4 + 3) + 1$ ， $5+3$ 与 $4+3$ 中的 $+$ 就是定义的 add，而最后的 $+1$ 则是 Succ。

```
add :: Nat -> Nat -> Nat
add Zero n = n
add (Succ m) n = Succ (add m n)
```

同样地，add 函数也可以这样定义：

```
add :: Nat -> Nat -> Nat
add Zero n = n
add (Succ m) n = add m (Succ n)
```

这种定义的方法是说，将第一个参数一个一个地加到第二个参数上，直到第一个参数为 0，然后返回第二个参数作为结果。后面的定义为尾递归定义。

上面的递归的估值计算的实际运行过程需要不断在内存上进行堆栈操作，可以用一个列表来代替这个过程：

```

eval' :: [BoolExp] -> Bool
eval' [TRUE] = True
eval' [FALSE] = False
eval' [(IF TRUE b1 b2):xs] = eval' (b1:xs)
eval' [(IF FALSE b1 b2):xs] = eval' (b2:xs)
eval' [(B(IF con b1 b2)):xs] = eval' (con:l:xs)
eval' [(TRUE:(IF con b1 b2)):xs] = eval' (b1:xs)
eval' [(FALSE:(IF con b1 b2)):xs] = eval' (b2:xs)

test = IF (IF FALSE FALSE TRUE) (IF FALSE TRUE FALSE) FALSE

```

有兴趣的读者可以做一些测试。

3. 参数化递归类型

通常，参数化类型与递归类型是一起使用的。预加载的库中定义了一个非常重要的类型，就是列表，它就是用了参数化与递归的方式定义的，读者可以在 GHCi 中查看它的定义：

```

> :i []
data [] a = [] | a : [a]
...

```

列表在 Haskell 中是一个参数化的递归定义类型。这里的[]如同构造类型一节中的 Book 一样，Book 既是一个类型构造器的名字，又是一个数据构造器的名字，在使用中一定要分清。对于不为空的列表定义 a : [a] 中的构造器就是常用的运算符(:)。

```

> :k []
[] :: * -> *
> :k [] Bool
[Bool] :: *

> :t []
[] :: [a]

>:[2] :: [] Float
[2.0]

>:[2] :: [Float]
[2.0]

```

在 Haskell 中实现一个功能与 Prelude 中列表相同的类型，可以这样定义：

```
data List a = Nil | Cons a (List a) deriving (Eq, Show)
```

也就是说，一个列表要么为空列表，定义为 Nil，要么是一个 a 类型的元素结合一个 a 类型的列表。这里，Nil 为递归定义的基本情况相当于列表定义中的[]，而 Cons a (List a) 为

递归定义。cons 是一个数据构造器，在这里相当于`(::)`运算符。List 是类型的名字，Nil 与 Cons 为构造器函数。如果不想列表为空，那么就可以像下面这样定义：

```
data List a = Nil a | Cons a (List a) deriving (Eq, Show)
```

这样，取得列表第一个元素的 lhead 函数就可以定义为：

```
lhead :: List a -> a
lhead (Nil a) = a
lhead (Cons a _) = a
```

再回到到与 Haskell 功能相同的、可以为空的列表。既然定义好了这种与 Haskell 相同的列表，可以看出这个定义与 Haskell 中定义的列表的形式是一样的，它们是等价的，那么它们是可以之间相互转换的。下面我们定义它们相互转换的函数：

```
listToMylist Nil = []
listToMylist (Cons x xs) = x:(listToMylist xs)

mylistToList [] = Nil
mylistToList (x:xs) = Cons x (mylistToList xs)
```

这两种列表的关系十分重要，下节将主要讨论类型间的这些关系。

8.2 类型的同构

上节中，定义了列表类型，它与 Haskell 中的列表类型功能是相同的，这里的功能相同指的是什么呢？其实就是两个类型“等价”，这种等价称为同构（isomorphism）。如果两个类型 A 与 B 之前可以相互转换，可以定义出转换的函数，并且它们均为一一对应的函数且互为反函数，则可称类型 A 与 B 是同构的。即可以定义一个函数 $f:A \rightarrow B$ 将 A 映射到 B，并且可以定义 f 的反函数 $g=f^{-1}:B \rightarrow A$ 将 B 映射到 A 且满足 $f \circ g=id_A$ 和 $g \circ f=id_B$ （ \circ 为复合函数运算符），那么类型 A 与类型 B 为同构，记做 $A=B$ 。类型与函数的关系如图 8-1 所示。

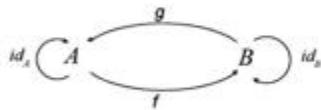


图 8-1 同构类型示意图

以定义的列表与 Haskell 的列表为例，这两个函数就是上一节中的 `listToMylist` 和 `mylistToList`。两个类型同构则说明定义在类型 A 上的函数在类型 B 上一样可以定义。例如，我们可以在定义的列表上写出常用的函数，比如 `head`、`last` 等，例如 `head` 函数：

```
listHead Nil = undefined
listHead (Cons x xs) = x
```

同构现象在函数式编程中十分常见，比如下面两个用枚举法定义的类型：

```
data ThreeNum = One | Two | Three
data Level = Low | Middle | High
```

像上边两个类型那样，所枚举的值的个数是相等的。那么，显然可以写两个函数相互转换，并且能够满足同构的条件。对于这种情况，可以清楚地总结出更为一般的规则：给定两个使用枚举定义的类型，若它们所定义的值的个数相等，那么这两个类型是同构的。

```
f :: TreeNum -> Level
f One = Low
f Two = Middle
f Three = High
g :: Level -> ThreeNum
g Low = One
g Middle = Two
g High = Three
```

若一个类型的值有无穷多个，则需要了解基数（cardinal number）的概念来区别不同等级的无穷才能更为一般地判定两个类型是否同构。这些概念超出了本书讨论的范围，所以在此不做深入探讨。但需要理解到，我们定义的列表与 Haskell 中的列表都有无穷个值，它们是同构的，如果读者仔细看它们的定义会发现，它们定义的形式完全相同，这种同构关系也非常明显。

```
data [a] = [] | a:[a]
data List a = Nil | Cons a (List a)
```

List a = [a]

列表是一个参数化定义类型，若给定这个类型参数为 Unit，并且 Unit 类型中仅定义有一个值——Unit，那么可以看出，() 与 Unit 类型是同构的，而 List Unit 与 Nat 是同构的。

```
data Unit = Unit
data List a = Nil | Cons a (List a)
data Nat = Zero | Succ Nat
```

```
List Unit = Nil | Cons Unit (List Unit)
Nat = Zero | Succ Nat
```

下面来定义这两个函数，转换 List Unit 与 Nat 类型。

```
list2Nat Nil = Zero
list2Nat (Cons x xs) = Succ (list2Nat xs)

nat2List Zero = Nil
nat2List (Succ n) = Cons Unit (nat2List n)
```

所以，可以得到结论：

List Unit = Nat

显然，用这种类型的列表是可以表示自然数的。[] 表示 0 即 zero，[Unit,Unit] 表示 2，即 Succ (Succ Zero)。也许这就是最原始的自然数表达的方法，或许可以称之为“一进制”。在 Nat 类型上可以进行的加法，也可以在类型[Unit] 上定义，其实质是列表连接运算符 (++)，同样地，乘法、乘方等也一样可以在这种类型上被定义。

下面来介绍一下新类型的构建与更为一般的同构类型。若给定类型 A、B 和 C，那么下列基于它们的元组显然也是同构的：

$$(A, B) = (B, A)$$

$$(A, (B, C)) = ((A, B), C) = (A, B, C)$$

函数类型之间与可以是同构的，只是此时对应的转换函数 f 与 g 均为高阶函数。

$$A \rightarrow (B, C) = A \rightarrow (C, B)$$

$$(A, B) \rightarrow C = A \rightarrow (B \rightarrow C)$$

关于第二个同构关系，相信读者已经有所了解了。转换这两个函数的函数，即为在本书开始和上一章高阶函数中介绍的 *curry* 与 *uncurry*。

像元组这样定义的类型，有一个构造器，构造器内含有多个类型，在书面表达时常常用 \times 来表示，如二元元组可定义为 `data Pair A B = Pair A B`，记作 $A \times B$ ，称作积类型（product type），在集合中称为笛卡尔积（Cartesian product）。比如，`Pair Bool Level` 中有 6 个值，`Pair True Low`、`Pair False Low`、`Pair True Middle` 等，这个类型中值的个数是 A 与 B 类型的值的个数之积。可以用 $|A|$ 来表示类型中值的个数，那么，则有 $|A \times B| = |A| \times |B|$ 。

而像 `Either` 类型这样，由多个构造器（或称多模式）枚举定义的类型，则用 $+$ 来表示，比如，`data Either A B = Left A | Right B`，记作 $A + B$ ，称作和类型（sum type），从集合的角度可以理解为不相交并集。即，在结合两个类型时，即便它们有重合的元素，但 `Left` 与 `Right` 构造器将作为标记对它们加以区分。如 `Either Bool Bool` 类型中有 `Left True`、`Left False`、`Right True`、`Right False` 这 4 个值，并且 `Right True` 与 `Left True` 是不同的值，显然这个类型值的个数是 A 与 B 类型值的个数之和，那么，则有 $|A + B| = |A| + |B|$ 。

此外，还有函数类型，如果是 `data Fun = Fun (A -> B)` 这样的函数类型，记作： B^A ，即这样的函数有 $|B^A| = |B|^{|A|}$ 个。这一点也很好理解，给定两个类型，可以构造出 $|B|^{|A|}$ 种函数，因为对于类型 A 中的每一个值在类型 B 中有 $|B|$ 种对应的方法。

像布尔值类型这样以具体值，或者称为零元构造器定义的和类型有时称为累计类型（counting），如布尔值类型可以写为 `1+1`，第一个 1 表示 `False`，第二个 1 表示 `True`，可以看到它与 `Either () ()` 同构。而自然数的定义则写为 `Nat = 1 + Nat`。

这样由其他类型通过积类型、和类型还有函数类型等方式构造出来的类型，称为代数数据类型（Algebraic Data Type, ADT）。比如，元组、列表、树、四则运算表达式等，也正是这种定义数据的方式，使得在解决问题时更为灵活。在使用积来定义时，一个构造器可能有多个参数。

```
data T = Con a b c d . .
```

其中，`Con` 为构造器，`a`、`b`、`c`、`d` 为构造器参数，即 `field`。这里使用多个 `field` 与使用一个元组定义是等价的。`T` 的类型可记作 $a \times b \times c \times d$ 。

如果定义类型时使用了和的方法来定义一个类型，那么这个类型的值就有了多种模式（pattern），也称为变体（variant）。这就是杂合定义类型：

```
data T = A a b c.. | B d e f.. | C g h i.. | ...
```

这里给出了类型 T 定义的三种形式，还可能有更多。每一个形式都有一个独一无二的构造器，这里的 T 类型可以记作 $(a \times b \times c \dots) + (d \times e \times f \dots) + (g \times h \times i \dots) + \dots$

这样，之前介绍的几个同构关系可以记作：

$$\begin{aligned} A \times B &= B \times A \\ (A \times B) \times C &= A \times (B \times C) = A \times B \times C \\ (B \times C)^A &= (C \times B)^A \\ C^{A \times B} &= (C^B)^A \end{aligned}$$

此外，对于：

$$A \times (B + C) = A \times B + A \times C$$

`(A, Either B C)` 和 `Either (A, B) (A, C)` 显然是同构的。根据定义，很容易地就可以定义出那两个函数：

```
f :: (a, Either b c) -> Either (a,b) (a,c)
f (a, Left b) = Left (a,b)
f (a, Right c) = Right (a,c)

g :: Either (a,b) (a,c) -> (a, Either b c)
g (Left (a, b)) = (a, Left b)
g (Right (a, c)) = (a, Right c)
```

这样，使用模式匹配或者 `case` 来定义基于某一类型的函数时，每一条匹配实际上匹配的是该类型的和中的某一个变体，即某一个构造器与该构造器的参数，这个参数可能是一个积或者函数。而需要对一个类型进行多个模式匹配则是因为该类型是由多个数据构造器的和定义的。比如，`Either` 的定义为 `data Either a b = Left a | Right b`，用模式匹配应该定义为：

```
f (Left a) = ...
f (Right a) = ...
```

而在定义函数时，需要定义多个模式匹配，这正是因为数据类型的值可能有多种形式。同时，读者也应该意识到 `non-exhaustive patterns` 异常产生的原因了，即在定义函数时，没有把所有的数据形式都考虑进来。也就是说，有一些形式在我们的函数中没有被定义，但是调用函数该时出现了这种形式却没有匹配。在编译的过程中，GHC 会对这些遗漏给出一些警告，即便定义时确定不会出现这种形式，也最好通过 `error` 或者其他方式来定义补上这个缺口。

这样，相信读者对于模式匹配将会有更加深入的了解了。很久以前，笔者刚刚学习 Haskell 函数式编程时，笔者的一位同学犯过一个非常经典的错误。不过，那是在学习定义类型之前，那时他还不知道如何做模式匹配。老师要求写一个函数取得列表中第 4 个元素，他是这样写的：

```
get4 :: [a] -> a
get4 ([a1,a2,a3,a4]++xs) = a4
```

这就是由于当时没有了解构造器引起的。因为++不是列表的构造器，仅仅是一个基于列表类型的函数，所以是不能出现在模式匹配中的。

可是在 Haskell 中，可以定义 `int2nat (n+1) = Succ (int2nat n)` 这样的函数，即便加号是函数，也可以写在左侧。读者在其他书上可能也见过这样定义的阶乘函数。这样做是不好的，会让使用 Haskell 的人感觉到困惑，Int 类型不是以加号作为构造器的，但是却可以在函数定义等式的左侧。这也许是引起那位同学出错的原因。`n+k` 这种匹配是 Haskell 98 的标准，在 Haskell 2000 标准中已经不允许了，最新的 GHC 中也默认禁止了这种匹配。

8.3 使用 newtype 定义类型

除使用 `data` 外，还可以使用 `newtype` 来定义新类型。使用 `data` 与 `newtype` 关键字定义的类型有一些区别。其中，最明显的区别在于 `newtype` 只能定义单一构造器，并且该构造器只能有且仅有一个参数。这样一来，`data` 关键字是可以完全代替 `newtype` 的，但反之则可能不成立。

所以，`newtype` 不可以通过枚举来定义类型，即定义的类型不能为多模式（multi-pattern）的，比如：多构造器定义：

```
newtype T = One | Two
```

会得到对应的错误：

```
A newtype must have exactly one constructor, but 'T' has two
In the newtype declaration for 'T'
```

这段错误提示大致意为 `newtype` 定义只能有一个构造器，而定义的类型 `T` 却有着两个构造器。

对那些使用 `newtype` 定义的有多参数的构造器：

```
newtype T a b = NewType a b
```

GHCi 也会报错：

```
The constructor of a newtype must have exactly one field
but 'NewType' has two
In the definition of data constructor 'NewType'
In the newtype declaration for 'T'
```

这段错误提示是说，`newtype` 的构造器必须有且只有一个参数，但是定义的 `newtype` 构造器有两个。这里可以看到构造器的参数被称为字段（field）。

但是下面的定义是合法的，因为 `(a,b)` 将作为 `NewType` 构造器的唯一一个参数：

```
newtype T a b = NewType (a, b)
```

首先，值得注意的是，它与类型的别名是有区别的，类型的别名不创造新类型，而只是把当前某些类型重命名，而 `newtype` 定义的类型是一个新的类型。并且 `newtype` 关键字也可以递

归地定义一个类型。比如：

```
newtype Stream a = Cons (a, (Stream a))
```

值得注意的是，这样的递归定义的类型是没有基本条件的。

另外，使用 newtype 定义一个类型可以理解为是将已有的某些

类型整合为一个类型，只是在这个类型外多了一个标记（tag）而已。比如：

```
newtype T a b = NewType (a, b) 或者 newtype T a b = NewType (a -> b)
```

另一点与 type 重命名一个类型不同，在定义时，可以加一些类型类的限定，比如：

```
newtype (Num a, Eq b) -> T a b = NewType (a -> b)
```

那么，只能定义仅有一个参数的构造器的意义是什么呢？既然已经有了 data 关键字，为什么还要有 newtype 呢？使用 data 定义类型会在编译系统类型检查与运行中产生额外的计算，因为需要构造数据而加一层包装。而当定义这种只有一个构造器且构造器仅有一个参数的类型时，用 data 就会显得有一种“杀鸡用牛刀”的感觉。而 type 关键字又不能完全满足要求，比如，需要实现 Show 类型类，使用 newtype 容器中的类型都是整数，但是意义却不尽相同，想让打印的方式有所不同，这时用 type 就不行了，因为在 Haskell 标准语法中，默认它是不可以被实例化的。例如，现实生活中的速度、秒、重量都可以用整数来表示：

```
newtype Velocity = Velocity Int
newtype Weight = Weight Int
newtype Second = Second Int
```

我们希望它们可以不一样地输出，虽然读者还没有学实现类型类，但是可以简单地看一下。在 Haskell 中，实现类型类使用的是 instance 关键字，可显示类型类 Show 中需要定义 show 函数：

```
instance Show Velocity where
    show (Velocity n) = show n ++ " m/s"

instance Show Weight where
    show (Weight w) = show w ++ " kg"

instance Show Second where
    show (Second 0) = "0 second"
    show (Second 1) = "1 second"
    show (Second n) = show n ++ " seconds"

> Second 5
5 seconds
> Weight 5
5 kg
```

newtype 定义的类型在运行时不会使用构造器对其中的值进行包装，从而减少了运行时的负担。读者可以把 newtype 理解为 type 与 data 为了争取效率的一个折衷。其实，newtype 与 data 定义的类型还有着一些其他区别，在这里先不讨论。

定义了一种类型，在一定程度上就可以理解为定义一种数据结构了，用户总是可以定义一些函数来对已经定义的类型进行操作而达到某些特定目的。像刚刚定义过的自然数类型就是一种结构，因为可以写 add、mul 等函数来做一些基于自然数类型的运算。再比如，Haskell 中定义的列表以及基于列表的函数其实也是一种数据结构。同时，定义的类型在一定意义上也可以抽象地理解为一个对象（object），比如，Book 类型的定义实际上更多的是在描述一本书的细节，书名（作者、ISBN 等）。如果读者了解 Java，则可能体会到枚举定义实际上是用 enum 关键字定义的一列有序的数据，比如，Java 中 Vector 这样的容器类是通过泛型（generics）定义的，比如在声明 `Vector<Integer>` 时，则说明这个 Vector 容器中只能存放整数对象，这其实和 `[Int]` 是十分相似的，通过这个例子可以知道，Java 中的泛型实际上在一定程度上对应的是 Haskell 中的参数化类型。如果用 Java 写一个 Book 类，则 Book 类型中的 Book 构造器函数实际上和 Haskell 也是十分相似的，并且 Java 中的构造器也是返回一个 Book 类型的值，即这个对象的一个实例。

8.4 数学归纳法的有效性

本节中我们来讨论一下数学归纳法。读到这里，读者可能会感觉有些突兀，之前所学的类型跟数学归纳法有什么关系？其实关系非常大，数学归纳法为什么可以用来证明、它的证明为什么有效以及什么时候可以应用数学归纳法都与类型的定义有着密切的关系。本节将会讨论这些问题。

首先，简单讨论一下一些逻辑与证明的知识。现在有两个命题 P 与 Q，已知 P 蕴含着 Q，还知道 P，那么通过 P 蕴含 Q 这个命题就可以证明 Q。可以将这种证明规则写为：

$$\frac{H0: P \quad H1: P \rightarrow Q}{Q}$$

横线上的内容是证明的前提，可以理解为已知条件，横线下的内容是可得到结论。这个证明规则很容易理解。比如，P 命题是：努力学习， $P \rightarrow Q$ 可以是努力学习意味着成绩很好，如果想要成绩很好，不但需要有努力学习与成绩好这种因果关系，还要有努力学习的前提。这种规则在某种意义上也可理解为函数的应用，即将条件 H1 应用到条件 H0 就会得到结果 Q。

在使用数学归纳法的时候，一般基于自然数。先证明一个基本情况 $n=0$ 时成立，然后通过假定 $n=k$ 时成立证明 $n=k+1$ 时成立。比如，有一个关于自然数的命题 $P(n)$ ，对于所有的自然数 n 都成立，这种证明规则可以写为下面的形式：

$$\frac{H0: P(0) \quad H1: P(n) \rightarrow P(n+1)}{\forall n. P(n)}$$

这样的证明方式实际上是基于自然数递归的结构的。首先，证明 zero 成立，假设对于任意 Nat 成立，然后，证明若 Succ Nat 也成立。由于 zero 被最初地证明成立了，根据递归步，如果任意 Nat 类型成立，那么 Succ Nat 也成立，这样，则 Succ Zero 成立，同理，有 Succ (Succ

`zero`), 依次类推。由于自然数为递归定义, 因此这就像是一排多米诺骨牌, 推倒了第一个 `zero`, 后边的 `Succ Nat` 依次全部倒下。

$$\begin{array}{c} H0 : P(0) \\ H1 : P(0) \rightarrow P(1) \\ H2 : P(1) \rightarrow P(2) \\ H3 : P(2) \rightarrow P(3) \\ H4 : P(3) \rightarrow P(4) \\ \dots \\ \hline \forall n. P(n) \end{array}$$

如果将证明规则写为上面的形式, 那么应用 H_1 到 H_0 可以得到 $P(1)$, 将 H_2 应用到 $P(1)$, 就会得到 $P(2)$, 然后依次类推, 就证明了对于所有的自然数 n , $P(n)$ 都成立。

数学归纳法还有加强的形式, 也就是强数学归纳法。在证明时, 可以假设对于所有小于 n 的自然数 $P(n)$ 全部成立然后证明 $P(n+1)$ 成立。这种规则可以写为下面的形式:

$$\begin{array}{c} H0 : P(0) \\ H1 : P(0) \wedge P(1) \dots \wedge P(n) \rightarrow P(n+1) \\ \hline \forall n. P(n) \end{array}$$

这种加强的形式也很容易理解, 因为在证明 $P(2)$ 时, 之前的 $P(0)$ 、 $P(1)$ 全部成立, 所以使用它们也是可以的。根据多米诺骨牌的原理, 就相当于某个牌倒下了, 可以十分确定地说它前面全部的牌必定都是倒下的, 这样, 可以用到所有之前的结论。

正因为自然数类型为递归定义, 才使得基于自然数的数学归纳法有效。若一个类型不是像自然数这样递归定义的, 则数学归纳法是不起作用的, 比如, 我们不会在实数类型上定义递归函数或者使用数学归纳法证明实数的某些性质。如果需要使用数学归纳法证明函数的性质, 那么这个函数所操作的类型必须是递归定义。例如, 证明下面的 `eval` 函数估值运算过程不会卡住并且总会得到 `Bool` 类型的结果。很明显, `BoolExp` 类型为递归定义, `TRUE` 与 `FALSE` 为基本条件。下面, 对 `eval` 的这一性质用数学归纳法给予证明:

```
data BoolExp = TRUE | FALSE | IF BoolExp BoolExp BoolExp

eval :: BoolExp -> Bool
eval TRUE = True
eval FALSE = False
eval (IF con b1 b2) | eval con == True = eval b1
                     | eval con == False = eval b2
```

当 `eval` 的参数为 `TRUE` 时, 结果为 `True`, 同理, 为 `FALSE` 时, 结果为 `False`。故, 当参数为 `TRUE` 和 `FALSE` 时, 估算运算是会停止的, 并且结果的类型为 `Bool`。

现假设, 对于任意的 `BoolExp` 的值 `b`, `eval b` 的计算会停止且结果类型为 `Bool`。那么, 当情况为 `IF con b1 b2` 时, 由于 `con`、`b1`、`b2` 的类型均为 `BoolExp`, 因此由归纳假设, 对于它们的估值计算都会停止并且结果均为 `Bool`, 因为 `eval con` 的估值会停止并且结果必为 `Bool`。同时, `eval (IF con`

$b_1 \ b_2$ 的结果不为 $\text{eval } b_1$ 就为 $\text{eval } b_2$ ，由归纳假设，它们也必然以 Bool 为结果。以上的证明足以说明对于任意有着类型 BoolExp 的值 b ， $\text{eval } b$ 的估值计算总会停止并且结果的类型为 Bool 。

这个例子的结论虽然无关紧要，但是却足以说明，数学归纳法并不一定需要基于自然数，在对 BoolExp 这样递归定义的结构上同样数学归纳法也是有效的。函数式编程的好处之一就是，以这种方式定义的数据类型在证明上比起顺序式语言要容易很多。

8.5 树

树（tree）在编程中是一个很重要的数据结构。树是一个颇为有趣的容器类型，根据不同需要，树也可以被定义成很多种，如二叉树、2-3-4 树等。在数据库的设计、虚拟机上变量的存储中常常用到。接下来，定义一些树的类型。下面我们来看一种树类型的定义：

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

这种定义方法的意思是，一个树的类型可以是基本形式——“叶子”，也可以是一个递归的形式，即一个节点，节点后跟两个子树。树的节点和树的最基本形式“叶子”都是容器。这样分两叉的树称为二叉树（binary tree），下面我们来看另一种二叉树：

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

这种树与上一种不同之处在于，它的“叶子”不是容器。正如列表中的 Nil 一样，不存放任何东西，仅仅是一个基本定义。显然，以上两种树的结构不是同构的，因为第一个 Leaf 可作为容器，可以用第二种树的定义写为 $\text{Node } a \ \text{Leaf} \ \text{Leaf}$ 。但是，第二种定义的 Leaf 中不存放任何东西，在第一种定义里没有对应。

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

这种树只有基本形式“叶子”是容器，而树的“节点”不为容器，不存储任何东西。

```
data Tree a = Node a [Tree a]
```

上面的定义就不仅仅是二叉树，而是可以有很多叉或者没有叉的树。它为什么不需要定义基本形式 Leaf 呢？因为当后跟的列表为空的时候就可以理解为一个基本形式了。这种方式也是 Haskell 库中树的定义，`Data.Tree` 中源文件中是这样定义的：

```
data Tree a = Node {
    rootLabel :: a,
    subForest :: Forest a
}
```

```
type Forest a = [Tree a]
```

此外，在很多时候，需要节点中的元素不止一个，这样就需要这样定义树类型：

```
data Tree a = Node [a] [Tree a]
```

如果这样定义树类型的话，在写函数的时候就要注意子树或者节点本身都有可能为[]的情况。

Haskell 中的树与其他语言里的树有很大差别的，纯函数中不能定义 C 语言中的指针变量。Haskell 定义树的各个结点间是不会形成环的，而 Java 或者 C 语言中定义的树是通过对象引用或者指针变量定义。这样，一个结点的子树可以为同一个。

```
#include <stdio.h>

struct node {
    char *word;
    struct node *left;
    struct node *right;
};

int main(int argc, char** argv) {
    struct node parent = {"parent",NULL,NULL};
    struct node child = {"child",NULL,NULL};
    parent.left = &child;
    parent.right = &child;
}
```

当 left 和 right 指针变量是一样时，就不是一个真正意义上的二叉树了，而在 Haskell 中不会出现这种可能的错误。

在解决不同的问题时，可能会用到不同的树。比如，将讨论的霍夫曼编码，该算法用到的就是 Tree a = Leaf a | Node (Tree a) (Tree a) 的定义。而二叉树搜索则用的是第一种或者第二种。当然，多叉树时用到的则是 data Tree a = Node a [Tree a]。下面来定义一些基于树的问题。

8.6 卡特兰数问题

卡特兰数又称作卡塔兰数，是用比利时数学家欧仁·查理·卡塔兰 (Eugène Charles Catalan) 的名字命名的。卡特兰数频繁地出现在各种计数问题中。它从 0 开始，前几项是 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786。卡特兰数的问题有很多，它们在一定程度上都是类似的。比如，给定 n 个结点，一共能组成多少种二叉树，所得到的数列就是卡特兰数。比如，有三个节点的树一共有五种，如图 8-2 所示。

这里的树不需要装任何值，它不是一个容器类型，所以直接定义为：

```
data Tree = Leaf | Node Tree Tree deriving Show
```

现在需要定义一个函数，当给定节点个数时，这个函数返回所有可能的树。这个很容易定义，思路就是将一个节点取出作为父节点，然后假定余下的节点个数为 n ，将 n 拆分成不同整

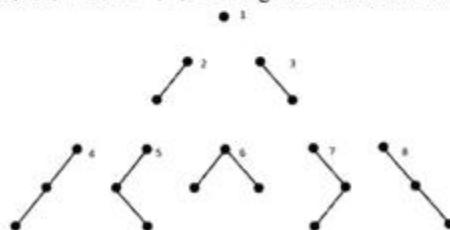


图 8-2 节点个数分别为 1、2、3 的二叉树

数的和，可以得到列表 $\{(0, n), (1, n-1), (2, n-2) \dots\}$ ，然后对于一个整数对 (l, r) 递归地用这个函数将树生成出左树与右树后加在父节点的左右就可以了。

```
treess :: Int -> [Tree]
treess 0 = [Leaf]
treess n = [Node lt rt | l <- [0..(n-1)], lt <- treess l, rt <- treess (n-1-l)]
```

与卡塔兰数相关的一个问题是生成合法的括号。合法的括号与这种二叉树有着对应关系。每一对括号都对应一个树的节点。如果一对括号在另一对括号的内部，可以等价为树的子节点在父节点的左侧，如果一对括号在另一对括号的外部，则可等价为树的子节点在父节点的右边。

```
brace :: Tree -> String
brace Leaf = ""
brace (Node l r) = '(:brace l ++")' ++ brace r

> map brace (treess 3)
["()", "()", "()", "()", "()", ""]
```

从图中可以看到，树这种类型也是可以数的，与自然数也可以有着一一对应关系。另外从图 8-2 可以得到一个并不意外的结论，这种树类型与自然数是同构的，也就是说，可以用这种方法来计数，有兴趣的读者可以试着定义那两个转化的函数，你可能需要用到卡塔兰数的通项公式。由此可以知道，两种类型的同构并不一定需要有着像 List Unit 与 Nat 那样相似的定义。其实，很多结构都可以被证明与自然树同构，但在这里就不进一步讨论了。

练习

试着不借助 Tree 类型定义函数 brace :: Int -> [String]，给定一个整数 n，brace 可以返回所有合法的 n 对括号。

8.7 霍夫曼编码

霍夫曼编码广泛用于图像、文本的压缩。在计算机中，英文字符是以长度相等的 ASCII 编码存储的。假定每一个 ASCII 都占 8 位，在讨论凯撒加密的时候读者已经知道不同的字符在英文文本中出现的频率是不同的，那么把出现频率高的字符和出现频率很低的字符编码成相同的长度显然对存储空间的使用效率是不高的，尤其是需要进行文件传输的情况下。霍夫曼编码就是对出现频率高的字符以较短的编码，而对于出现频率低的字符则给以较长的编码，从而使得编码的平均长度达到最短，这种不定长度的编码方式就被称为霍夫曼编码。

首先，这里简单介绍一下这个算法。霍夫曼编码的生成是一个贪婪算法，始终选取最小的两个概率相加，它的算法过程如下。

假设 A 地发生互斥随机事件的概率如下：

```
[("p1", 0.4), ("p2", 0.3), ("p3", 0.1), ("p4", 0.1), ("p5", 0.06), ("p6", 0.04)]
```

每隔1小时，发报员需要从A地向B地发送事件的发生情况，发报机只能发送0与1，编码这些事件就需要霍夫曼编码了。首先，选取概率最小的两个事件，即p5与p6，将它们组合成一个概率为0.1的事件的两种情况，即一棵树的两个分支，如图8-3所示。

继续选取两个概率最小的两个事件。这里的p3、p4和已经构建的这个事件的概率均为0.1，所以可以选取任意两个，这里取p3与p4，如图8-4所示。

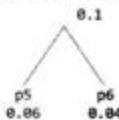


图8-3 霍夫曼编码生成步骤1

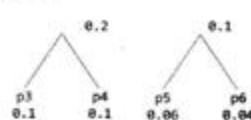


图8-4 霍夫曼编码生成步骤2

类似地，再选取两个概率最小的两个事件，它们就是以上两个组合的事件，如图8-5所示。

最后，会得到图8-6所示的一棵树。

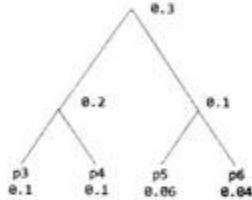


图8-5 霍夫曼编码生成步骤3

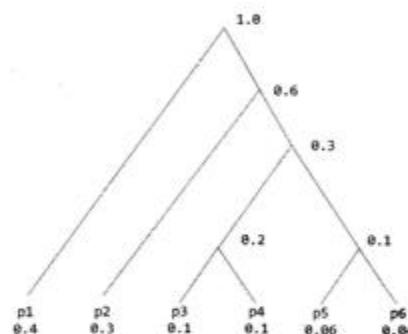


图8-6 霍夫曼编码生成树的结果

在编码的时候，遇到左叉补0，遇到右叉补1。这样，位置较深的节点，由于它们的概率较小，所以它们的编码就会较长，反之深度小的节点的编码就会较短。当然，结果可能不止一种，编码结果如图8-7所示。

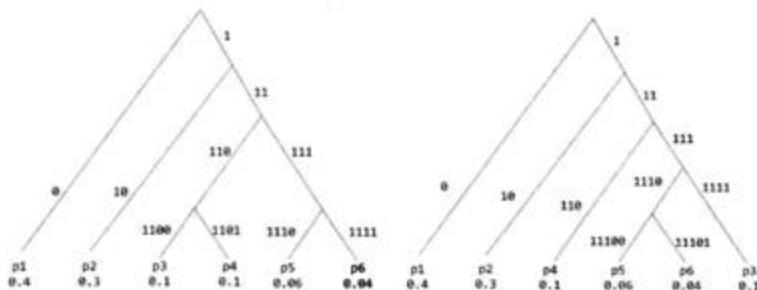


图8-7 其他霍夫曼编码生成树的结果

```
import Data.List (insertBy, sortBy)
import Data.Ord (comparing)
```

为了生成霍夫曼编码，需要定义这个树类型，这里用到的就是只有叶子为容器的树类型，它定义为：

```
data HTree a = Leaf a | Branch (HTree a) (HTree a) deriving Show
```

这里用 `(a, HTree b)` 类型来记录概率与生成的树，用一个列表来存储没有构建的树。由于总是需要取最小的两个树来构建一个新树，因此在生成的时候可以先假定列表一直保持升序的。当列表中只有一个元素时，就不必再构建了，直接返回结果，这时理论上概率应为 1.0，若还有两个以上的元素，则再选取两个最小的元素继续构建树。构建好后，使用 `insertBy` 函数将它插入余下的树的列表中，以保证概率依然是升序的。最后，递归地调用这个函数就可以了。

```
htree [(_, t)] = t
htree ((w1,t1):(w2,t2):wts) = htree $ insertBy (comparing fst) (w1 + w2, Branch t1 t2)
wts
```

当给定了一棵树后，就要进行编码了。左支补 0，右支补 1，这里为了方便，我们补在前面，所以有：

```
serialize (Leaf x) = [(x, "")]
serialize (Branch l r) = [(x, '0':code) | (x,code) <- serialize l] ++ [(x, '1':code) | (x,code) <- serialize r]
```

实现了以上两个函数，`huffman` 函数就非常容易了。在生成时，需要将这些树按照概率排序，最后，使用 `serialize` 函数生成编码即可。

```
huffman :: (Ord a, Ord w, Num w) => [(a,w)] -> [(a,[Char])]
huffman freq = sortBy (comparing fst) $ serialize
    $ htree $ sortBy (comparing fst) $ [(w, Leaf x) | (x,w) <- freq]
> huffman (zip ['a'] [0.4,0.3,0.1,0.1,0.06,0.04])
[("a","0"),("b","10"),("c","1111"),("d","110"),("e","11101"),("f","11100")]
```

练习

在凯撒加密小节中，读者知道了应当如何统计一个字符在文本中的概率。定义一个函数，使它的类型为 `String -> String`，参数为英文文本，结果为用霍夫曼编码压缩的二进制字符串。

8.8 解 24 点

24 点是用扑克牌玩的一种算术游戏，即用 4 张牌上的点数，做四则运算来得到 24。多人竞赛有更多乐趣，同时还可以提高心算能力。但有时大家常常想不出来，有可能是真的无解，也有可能是解太少以至于很难想。现在，用 Haskell 来解决这个问题。

首先，介绍表达式树。在日常生活中，使用四则运算都是写于一行的，并且事先知道各个符号

的优先级。但是，如果式子很长就会看起来有些混乱，倘若将其表示成一个树状的结构的话，即使不知道运算符的优先级也可以计算，并且非常直观。比如， $(3+4)*2$ 的表达式树如图 8-8 所示。

通过定义表达式树类型，这个表达式可以写为 `Mult (Add (Val 3) (Val 4)) (Val 2)`，这样，可以通过组合生成所有可能的表达式树，并计算生成表达式树的结果是否为 24，从而达到解决 24 点的目的。这里，直接使用库中的全排列函数。

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

首先，定义四则运算的表达式树类型，这里的类型使用的是 `Double`。由于需要用到小数除法，因此为了避免再转换类型，使用 `Double` 就好了，其实这里使用 `Rational` 类型更精确。

```
data Exp = Val Double
| Plus Exp Exp
| Sub Exp Exp
| Mult Exp Exp
| Div Exp Exp deriving (Show, Eq)
```

这样，整个问题可以被抽象为一个有着类型为 `[Double] -> [Exp]` 的函数，即以一系列的整数为输入，返回多个计算结果为 24 的算术表达式。定义了算术表达式树结构，递归地计算一个表达式树的结果：

```
eval :: Exp -> Double
eval (Val a) = a
eval (Plus a b) = eval a + eval b
eval (Sub a b) = eval a - eval b
eval (Mult a b) = eval a * eval b
eval (Div a b) = eval a / eval b
```

类似地，将一个表达式树输出成单行的形式。

```
showExp :: Exp -> String
showExp (Val a) = show a
showExp (Plus a b) = "(" ++ showExp a ++ "+" ++ showExp b ++ ")"
showExp (Sub a b) = "(" ++ showExp a ++ "-" ++ showExp b ++ ")"
showExp (Mult a b) = "(" ++ showExp a ++ "*" ++ showExp b ++ ")"
showExp (Div a b) = "(" ++ showExp a ++ "/" ++ showExp b ++ ")"
```

然后，将一个数字的列表分割成列表的元组，比如 $[1, 2, 3] \rightarrow [[(1), (2, 3)], ((1, 2), (3))]$ ，这样，二元元组的两个元件间可以插入 4 种运算符号，然后对于长度不为 1 的列表可以递归地生成它所有可能的表达式树。先来定义分割列表的函数：

```
divide :: [a] -> [[[a], [a]]]
divide xs = [(take n xs, drop n xs) | n <- [1..(length xs - 1)]]

> divide [1,2,3,4]
[[[1], [2, 3, 4]], ((1, 2), [3, 4]), ((1, 2, 3), [4])]
```

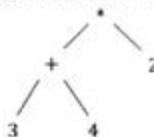


图 8-8 $(3+4)*2$ 的表达式树示意图

下面定义两个函数，将列表中所有可能的表达式树遍历出。这里先不必对其进行排列，稍后将用 permutations 函数来得到所有的排列。

通过使用递归可以很简单地遍历所有可能的表达式。给定两个表达式树（如图 8-9 所示），可以通过这 4 种运算符连接来构成一个新的表达式树。使用列表内包做到这一点十分容易。

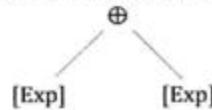


图 8-9 递归生成表达式树示意图

```
buildExpressions :: ([Exp], [Exp]) -> [Exp]
buildExpressions (es1, es2) = [op e1 e2 | e1 <- es1, e2 <- es2, op <- [Plus, Sub, Mult, Div]]
```

toExpression 函数会递归地生成一个表达式树的左侧和右侧，然后使用 buildExpressions 函数将生成的树组合起来。

```
toExpressions :: [Double] -> [Exp]
toExpressions [] = []
toExpressions [x] = [Val x]
toExpressions xs = concat [buildExpressions (toExpressions l, toExpressions r) | (l, r) <- divide xs ]
```

再定义一个函数将给定的列表全排列一次，再生成每一个排列所有可能的表达式树从而达到遍历的目的。

```
generate :: [Double] -> [Exp]
generate ns = concatMap toExpressions (permutations ns)
```

解 24 点，即让表达式树的结果为 24，通过使用列表内包很容易做到。

```
twentyfour :: [Double] -> [String]
twentyfour ns = [showExp x | x <- generate ns, eval x == 24.0 ]
```

这样，这个问题就被解决了，是不是很简单？有兴趣的读者可以找几组数来测试一下 twentyfour 函数。

练习

1. 我们将每一个运算表达式都加了括号，如 $((2*3)+(9+9))$ 则为 $2*3+9+9$ 。定义一个函数 showExp :: Exp -> String 可以省略那些不必要的括号。提示：需要要定义乘法、除法与加法减法的优先级，然后根据优先级来判断。
2. 给定一个由 0、1、&、|、^ 组成的布尔表达式，& 是逻辑与，| 是逻辑或者，^ 是逻辑异或，这些逻辑操作前面介绍过了。下面写一个函数，当给定一个逻辑表达式还有布尔值时，这个函数会返回使得这个逻辑表达式成立的结果是这个布尔值的所有可能的结合方式。例如，给出表达式 $1^0|0|1$ 还有布尔值 False，这里可以用 0 来表示，那么结果应该有两个，即 $1^((0|0)|1)$ 和 $1^((0|(0|1)))$ 。
3. 往往比较难以解的 24 点题目是只有一组解的，我们的牌可以表示为 [1..13] 的列表，请写出一个函数来遍历那些只有一组解的组合。提示：你需要用一个函数来把重复的排列去掉，读者可以自己定义，也可以使用 Data.List 中的 nub 或者其他相关的函数。

8.9 Zipper

之前定义的函数都是一次性地历遍一个容器，当我们需要在一个容器中反复多次游历时需要在游历的过程中保持容器结构与其中值的完整性。比如，`sum`求和时进行 `sum (x:xs) = x + sum xs` 计算后，程序运行时指向 `x` 的指针变量就消失了，再无法取回 `x` 的值。而当我们需要在列表中来回游历时就需要使用 `zipper`（拉锁）类型。

先来讨论如何游历于列表容器，下面就来看一下游历于列表类型的拉锁类型是如何定义的：

```
data Zipper a = Zipper [a] a [a] deriving Show
```

首先，定义 `fromList` 函数，将一个列表转换成一个 `zipper` 类型。

```
fromList :: [a] -> Zipper a
fromList (x:xs) = Zipper [] x xs
fromList _      = error "empty!"
```

当前所在的位置为 `a`，向后移动一次就将 `zipper` 当前的元素暂时存入左面的列表中，再从右端列表中取出一个元素作为当前位置。再向前移动也是类似的。

```
next :: Zipper a -> Zipper a
next (Zipper ys y (x:xs)) = Zipper (y:ys) x xs
next z = z

prev :: Zipper a -> Zipper a
prev (Zipper (y:ys) x xs) = Zipper ys y (x:xs)
prev z = z

> next $ fromList [1,2,3,4]
Zipper [1] 2 [3,4]
```

这样，就可以使用 `next` 与 `prev` 函数在这个列表中游历了。当然也可以使用一个元组类型来代替 `type Zipper a = ([a], a, [a])` 或者也可以使用 `([a], [a])`。

再看基于树类型的拉锁，这里定义一个递归类型 `Accumulate` 来暂存树的其余部分：

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
data Accumulate a = Empty | R (Accumulate a) a (Tree a)
                  | L (Accumulate a) a (Tree a)
```

`Accumulate` 有三种形式，`Empty` 表达什么也没存，`R` 与 `L` 用来标记左与右。最后，定义 `Zipper` 类型。它的第一个元件为树类型，也就是当前所在树中的位置，第二个元件就是 `Accumulate` 类型了。

```
type Zipper a = (Tree a, Accumulate a)
```

在这种树的类型中，游历只需要有三个操作就可以，一个是到游历到树的左支，一个是到

游历树的右支，最后一个分支是返回上一级。游历时，类型 Accumulate 中记录了之前累积的树的父节点和另外一支并且使用 R 与 L 进行了标记。

```
right, left, up :: Zipper a -> Zipper a
right (Node n l r, a) = (r, R a n l)
right a = a

left (Node n l r, a) = (l, L a n r)
left a = a

up (t, R a n l) = (Node n l t, a)
up (t, L a n r) = (Node n r t, a)
up z@t, Empty )= z
```

Accumulate 是使用一个用户自定义的递归参数化类型来暂时存储的，还可以使用一个列表来存储，当然，列表也是递归定义的。

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

这里定义 Branch 类型，用来存储父节点与树的一支，使用 R 与 L 区分左支与右支。

```
data Branch a = R a (Tree a) | L a (Tree a)
```

元组的第一个元件为当前游历时，当前所在的容器中的位置，第二个元件用来存储父节点与支干。

```
type Zipper a = (Tree a, [Branch a])

right, left, up :: Zipper a -> Zipper a

right (Node n l r, t) = (r, R n l: t)
right z@ (Leaf a, t) = z

left (Node n l r, t) = (l, L n r :t)
left z@ (Leaf a, t) = z

up (r, (R n l: t)) = (Node n l r ,t)
up (l, (L n r: t)) = (Node n l r ,t)
up z@t, [] )= z
```

这里可以略微比较一下 Accumulate 类型与 Branch 类型跟列表类型的组合：

```
data Accumulate a = Empty | R (Accumulate a) a (Tree a)
                  | L (Accumulate a) a (Tree a)

data Branch a = R a (Tree a) | L a (Tree a)
data List b = Nil | Cons b List b
```

列表类型与 Branch 类型的组合 List (Branch a) 只能有三种可能：

```
Nil | Cons (R a (Tree a)) (List (Branch a))
      | Cons (L a (Tree a)) (List (Branch a))
```

那么，它们组合起来与 Accumulate 显然是同构的，Accumulate 中的 Empty 对应 Nil，Accumulate 中的 R ... a (Tree a) 对应 List (Branch a) 中的 R a (Tree a)。

拉锁是一种很有用的类型，比如在编写窗口管理器的时候，可以将多个窗口存储于一个列表，但是可能需要用户在多个窗口中反复切换，那么这时就需要拉锁类型了。再比如，文件系统可以看做一个树，目录就好比节点，我们可以访问一个目录，也可以退到上一级目录，也可以当做一个拉锁类型。

本节的内容参考了 (Lipovac, 2011) 的第 14 章，还有斯坦福大学计算机专业 Haskell 中的函数系统这门课程，读者可以访问 <http://www.scs.stanford.edu/11au-cs240h/notes/zipper-slides.html> 来了解更多相关的内容。

8.10 一般化的代数数据类型

读者之前对代数数据类型 (ADT) 应该有些了解了，这里来讨论一下一般化的代数数据类型 (Generalized Algebraic Datatypes, GADT)。前面定义的类型有着诸多 Haskell 类型系统带来的限制，有时可能会给用户带来很多不便。

当需要实现一个简单的“特定领域语言 (Domain Specific Language, DSL)”，比如解 24 点时用的算术语言，先定义这个语言的语法树，然后在对语法树进行计算等。这里将这个语言做一下简单的扩展，除 Int 类型外，再引入 Bool 类型，为了简化问题，只保留加法。引入了 Bool 值的原因是因为想引入一些关系运算符，如 ==、>= 等，这里只引入 ==。如果使用 ADTs，则实现如下：

```
data Exp = ValInt Int
         | ValBool Bool
         | Add Exp Exp
         | Equa Exp Exp
Deriving (Eq, Show)
```

这样，在 eval 函数中就需要对 Exp 类型中 Equa 的形式增加一个匹配，并且需要用 Either 来区别加法与判断相等运算的结果：

```
eval :: Exp -> Either Int Bool
eval (ValInt a) = Left a
eval (ValBool b) = Right b
eval (Add e1 e2) = case eval e1 of
                      Left a -> case eval e2 of
                                      Left b -> Left (a + b)

eval (Equa e1 e2) = case eval e1 of
                      Left a -> case eval e2 of
                                      Left b -> Right (a==b)
```

很明显，这两个运算符会有类型问题，因为它允许 Add (ValInt 1) (ValBool True) 这样的表达式，但是在 eval 的定义却没有给出相应的匹配。所以需要添加更多的类型信息来保

证类型不会出现这样的问题，从而使得这样的表达式不能通过编译，这样就可以在程序的编译期发现更多的问题。这里将 `Exp` 类型参数化，增加一个类型参数 `a`。

```
data Exp a = ValInt Int
           | ValBool Bool
           | Add (Exp a) (Exp a)
           | Equa (Exp a) (Exp a)
deriving (Eq, Show)
```

这里的类型 `a` 称为虚幻类型（phantom type），一个有着 `Exp a` 类型的值不会包括一个有着 `a` 类型的值，比如 `ValInt 5`、`ValBool True`、`Add (ValInt 1) (ValInt 2)`，而像列表 `[a]` 这样的类型，则是说类型为某一类型 `a` 的列表，如 `[1, 2]`，这里的 `a` 为整数类型。可以看到，在定义类型时，递归的基本形式与类型参数无关时，那么这个无关的类型则为虚幻类型。引入这个虚幻类型的是为了记录在计算中的类型。但是，不但需要记录这些类型，还要通过将虚幻类型与 `Int` 与 `Bool` 联系起来从而加入一些限制，否则在这里引入虚幻类型就没有意义了。比如，下面的表达式一样是合法的，同时也反映了虚幻类型这一名字的意义：

```
> :t ValInt 5
ValInt 5 :: Exp a

> ValInt 5 :: Exp Bool
ValInt 5
```

Haskell 中提供的 GADT，可以让构造器携带更多的类型信息来对类型做出需要的限制。但是，需要在文件首处加入编译器参数 (`# LANGUAGE GADTs #-`) 或者在 GHCI 中使用 `:set -XGADTs`，使用它定义类型时，格式是这样的：

```
data TypeName arg1 arg2 ... where
  Con1 :: Type1
  Con2 :: Type2
  ...
```

比如，`Maybe` 类型用 GADT 可以重写为：

```
data Maybe a where
  Nothing :: Maybe a
  Just :: a -> Maybe a
```

通过使用 GADT，上边的表达式就可以定义为：

```
data Exp a where
  ValInt :: Int -> Exp Int
  ValBool :: Bool -> Exp Bool
  Add    :: Exp Int -> Exp Int -> Exp Int
  Equa   :: Exp Int -> Exp Int -> Exp Bool
```

这样，就限制 `Add` 构造器的结果类型为 `Exp Int`，`Equa` 结果类型为 `Exp Bool` 而避免了之

前可能引起的错误。但是，使用 GADT 的代价之一就是不能通过简单地 `deriving` 来导出 `Eq`、`Show` 等类型类。从上面的表达式也可以看出，将 `ValInt` 与 `Exp Int` 中的 `Int` 类型相关联了，而不是仅仅写做 `ValInt Int`。此时，`ValInt 5` 的类型就与它的表达式类型联系起来了：

```
> t : ValInt 5
ValInt 5 :: Exp Int
```

这样就可以重新定义 `eval` 函数了：

```
eval :: Exp a -> a
eval (ValInt i) = i
eval (ValBool b) = b
eval (Add e1 e2) = eval e1 + eval e2
eval (Equa e1 e2) = eval e1 == eval e2
```

由于已经在定义 `Exp` 类型时做出了相应的限制，因此不会出现 `Add (ValInt 1) (ValBool True)`，这样就不会出现不满足类型限定的表达式了。

```
> eval $ Equa (ValInt 5) (ValInt 5)
True

> eval $ Add (ValInt 5) (ValInt 5)
10

> eval $ Add (ValInt 5) (ValBool True)

<interactive>:191:24:
Couldn't match expected type 'Int' with actual type 'Bool'
Expected type: Exp Int
Actual type: Exp Bool
In the return type of a call of 'ValBool'

In the second argument of 'Add', namely '(ValBool True)'
```

如此一来，一些不合法的表达式就会被类型系统检查出来，不会通过编译从而不会得到因未定义某些匹配而导致运行时错误，但同时也付出了另一些代价。这个代价就是 `eval` 函数类型签名中的 `a` 类型所指的并不是所有类型。这里仅仅是指 `Bool` 与 `Int` 类型，所以这个类型签名不能准确地表达出 `eval` 函数的类型。但是，这里的类型签名也不能省略，因为如果不给定这个类型签名，GHC 就会向我们抱怨不能匹配 `Bool` 与 `Int` 类型，因为在做类型推断时，`eval` 中的前两个定义一个返回的是 `Int` 类型，另一个返回的是 `Bool` 类型，这里建议使用注释对这个类型签名进行适当的注释。

使用 GADT 定义数据类型和使用常规方法定义数据类型，其区别在于：在 GADT 里，需要明确指出每个构造器的类型，而在常规方法里，每个构造器的类型由编译器自动生成而且严格受限。例如，下面是这们讨论过的二叉树类型：

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

这里，`Branch` 的类型是 `Branch :: Tree a -> Tree a -> Tree a`，所以也可以用

GADT 重写为：

```
data Tree a where
  Leaf :: Int -> Tree a
  Branch :: Tree a -> Tree a -> Tree a
```

可以为结果指定任意类型，返回一个 Tree Int 类型的二叉树，比如：

```
data Tree a where
  Leaf :: a -> Tree Int
  Branch :: Tree a -> Tree a -> Tree Int
```

这一点在之前是无法做到的，GADT 给了用户更多的自由。但是，在享受更多自由的同时，我们还需要对获得的“自由”加一些限定，从而保证可以让 Haskell 的类型系统找到程序的一些错误。下面先来深入地了解一下 kind 这一概念。

8.11 类型的 kind

Haskell 是一门极具抽象能力的语言，所以了解 Haskell 是如何对于事物进行抽象对于我们来说非常重要，这也是使得 Haskell 学习之路陡峭的原因之一，因为我们需要先理解它对事情的抽象然后才能去使用它。这一小节里我们介绍一下 kind，它是对类型进一步的抽象，这里我们为了讨论方便使用英文 kind。在 8.1 节中，我们对于 kind 有了一个简单的了解，它可以简单理解为类型的类型，下面我们来深入地了解一下它。

Haskell 中的类型把值还有函数归类了，它们在 Haskell 常常可以称为 terms。而 Haskell 中的 kind 把类型又进行归类了。此外 Haskell 的设计者又通过把类型类限定定义为一种新的 kind，它写为 Constraint，我将在 9.2 节中简单地提一下它。

8.11.1 类型的 kind

当我们在声明新类型的时候，比如 `data Triple a b c = Triple a b`，这里类型构造器的 kind 与数据构造器的类型分别为：

```
> :k Triple
Triple :: * -> * -> * -> *
> :t Triple
Triple :: a -> b -> c -> Triple a b c
```

这里的 Triple 的类型构造器需要 3 个类型参数，每个类型参数都用一个*代表，返回一个完整的类型。而数据构造器只需要 2 个参数就可以构造出一个该类型的值。对于有类型类限定的类型也可以通过这种方式来查看，如：

```
> :k (Num a) -> a
(Num a) -> a :: *
```

像函数一样，kind 也可以是高阶的。比如，我们定义两种树类型：

```
data P a = P (a,a)
data BTee a      = BLeaf a | BNode (P (BTee a)) deriving (Show, Eq)
data RoseTree a = RLeaf a | RNode [RoseTree a] deriving (Show, Eq)
```

现在我们可以通过引入另外一个类型把这两种树类型的公共部分提取出来，定义成一个更为一般的树：

```
data AbsTree k a = Leaf a | Node (k (AbsTree k a))
```

这样，那两个树就可以定义为 AbsTree P a 还有 AbsTree [] a 了。那么 AbsTree 这个类型的 kind 是什么样的呢？

```
> :k AbsTree
AbsTree :: (* -> *) -> * -> *
```

这样我们就通过引入一个有`* -> * kind` 签名的类型把两种不同的树整理成为了一种抽象树，在这种抽象树的基础上来表达其他的树。借助 GHC 的语法扩展 GADTs 与 KindSignatures 我们可以声明我们定义类型的 kind，这里以列表还有 AbsTree 为例。

```
({-# LANGUAGE GADTs , KindSignatures , TypeSynonymInstances , FlexibleInstances #-})
data T :: * -> * where
  NIL :: T a
  CONS :: a -> T a -> T a

data AbsTree k a = Leaf a | Node (k (AbsTree k a))

data Tree :: (* -> *) -> * -> * where
  L :: a -> Tree k a
  N :: k (Tree k a) -> Tree k a

type RoseTree a = Tree [] a

instance Show a => Show (RoseTree a) where
  show (L a) = show a
  show (N tree) = show tree

test :: RoseTree Int
test = N [L 5, L 8, N [L 1 , L 2], N[L 3]]
> test
[[5,8,[1,2]],3]
```

这里 RoseTree 使用的是 type 关键字定义的，如果要它实现 Show 类型类则需要两个额外的编译器参数。虽然在下一章中我们才会学习如何实现类型类，但这里我也将实现类型类的代码给出，其实不难，只是使用 instance 关键字，然后定义一个 show 函数。读者可以读过下一章后再重要看这一小节的内容。

GHC 中默认 kind 都是使用`*`来表示的，但是新版本的 GHC 允许我们自己来定义 kind 的名字，下一节中我们就来讨论一下它。

8.11.2 空类型的声明

从上一小节中的虚幻类型中可以看到，类型参数可以在类型声明中作为标记来提供一些很有用的额外信息。有的时候，定义其他类型时仅仅需要另外一个类型的名字，而不需要这个类型中有任何值^①，这时我们就需要一个空类型。在声明空数据类型前需要在文件首处加上下面的编译器参数：

```
(-# LANGUAGE GADTS, EmptyDataDecls #-)
```

下面来简单讨论一下如何定义一个安全的 head 函数，之前用了很多方法，比如：

```
data List a = Nil a | Cons a (List a) deriving (Eq, Show)
```

还有，使用 Maybe 等能不能从根本上杜绝这种情况的发生呢？借助于空类型是可以的。只需要让 Nil 与 Cons 两个构造器携带更多的类型信息，即不仅要说 Nil 是一个列表，还要说 Nil 是一个为空的列表。对于 Cons 构造器的结果，需要注明一定为非空的列表。

```
data Empty
data NonEmpty
```

这里定义的两个类型 Empty、NonEmpty 的 kind 均为 *。下面定义 List 类型，这里使用 Empty 与 NonEmpty 对于空与非空的列表加以区分：

```
data List a b where
    Nil :: List a Empty
    Cons :: a -> List a b -> List a NonEmpty

safeHead :: List a NonEmpty -> a
safeHead (Cons x _) = x

> safeHead Nil
<interactive>:1:9:
  Couldn't match expected type 'NonEmpty' with actual type 'Empty'
  Expected type: List a0 NonEmpty
  Actual type: List a0 Empty
  In the first argument of 'safeHead', namely 'Nil'
  In the expression: safeHead Nil
```

这样，就使用了两个不同的类型来区别为空的列表与不为空的列表。这两个类型的作用也只是这个，所以它们的值对于我们没有任何用处。在模式匹配时，safeHead 函数只对不为空的列表进行匹配来到达目的。这样的定义与之前声明的：

```
data List a = Nil a | Cons a (List a) deriving (Eq, Show)
```

它们有很大区别，这个列表其实是不能为空的。事实上，它是以列表中仅有一个元素的情形为递归的基本条件的类型，而这里使用空类型定义的列表类型是可以为空的。

^① 其实即使是空类型中也是有值的，它是 ()，详细内容需要参阅本书 15.2 节。

这样，声明虽然解决了空列表的问题，但是却产生了其他的问题，其实需要限定：

```
Cons :: a -> List a b -> List a NonEmpty
```

定义 Cons 时，b 类型需要被限定为只能是 Empty 与 NonEmpty 类型，这就需要我们自己来定义一个 kind，这个 kind 下有两个值，借助 DataKind 就可以做这样的事情。

```
{-# LANGUAGE GADTs , DataKinds , KindSignatures #-}
```

DataKinds 编译器参数可以将我们定义的类型中的值作为类型使用：

```
data KEmpty = Empty | NonEmpty
data List :: * -> KEmpty -> * where
    Nil :: List a Empty
    Cons :: a -> List a b -> List a NonEmpty
>:t Empty
Empty :: KEmpty
>:k Empty
Empty :: KEmpty
```

这样定义的 List :: * -> KEmpty -> * 它类型构造器的第二个参数就只能为 Empty 与 NonEmpty，不可以为其他的 kind 了。

```
> :k List
List :: * -> KEmpty -> *
```

这样 List 的 kind 的签名就不是 List :: * -> * -> *，即第二个类型参数必须为 KEmpty 中的值，使用这种方法我们就对列表做出了更严格的限定。GHC 中的 kind 系统还在研究与实现阶段，笔者这里认为 DataKinds 这种实现并不好，UHC 中将 kind 作为关键字来定义新的 kind 与类型，GHC 7.8 中会引入类似的关键字使我们可以明确地定义 kind。虽然在 GHC 7.4 中新的 kind 系统还处于实验阶段，但是已经可以在类型上做很多运算，可以借助它来实现带有长度信息的列表，这就需要在类型上做加法运算。相信它会在 GHC 7.6 与将来的 GHC 7.8 版本中更加强大。关于 kind 我们就不再深入讨论了，有兴趣的读者可以参阅 (GHC Team, 2012) 的 7.8 节以及关注新版本 GHC 用户手册。

本章小结

本章里我们学习了如何定义数据的类型，使用 data 关键字来定义类型非常灵活。此外，我们还了解了类型的同构、数学归纳法等相对理论的内容，但是它们与类型是息息相关的。两个类型如果同构，那么可以在两个不同类型上执行一样的操作。另外，只有对于递归定义的类型数学归纳法才有效，只有理解了类型才能完全了解数学归纳法的思想。8.10 节的一般化数据类型可以让我们更为灵活地定义类型，打破之前已有的束缚，在后面的内容中我们还需要使用它。在 8.11 节中了解了一些关于 kind 的内容，它相当于类型的类型，了解它不但有助于了解 Haskell 的类型系统，更有助于在编程中写出正确的代码。

第 9 章

定义类型类

之前我们了解了类型类，如 Eq、Ord、Show 等，本章中简单介绍如何定义它们。具有同一类性质或者属性的类型可以在 Haskell 中归为一类，在 Haskell 称为类型类（typeclass），这些属性其实只是一些预先设定好的函数。类型类使用 class 关键字定义。Haskell 中的 class 关键字完全不同于 Java 与其他语言的 class，是完全不同的概念，但却和 Java 中接口的概念有几分相似之处。使用 Haskell 的类型类的可以非常容易地复用已经有的代码，从而减少编码工作量，提高了库的质量。

本章先来讨论 Haskell 中常用的一些类型类是如何定义的，如 Eq、Ord、Enum、Num 等，然后再定义自己的类型类。在定义有多个参数的类型类时，我们会发现有时这些定义会让 Haskell 在进行类型推断时产生歧义，不过我们会学习如何消除这些歧义。

9.1 定义类型类

在 Haskell 中类型类通过使用 class 关键字定义，定义它的格式如下：

```
class C a b c ... where  
  fun1 = ...  
  fun2 = ...
```

c 为类型类的名称，需要大写。后跟类型类中声明的函数所需要的类型，可以为一个，也可以为多个。当一个类型类有多个类型参数时需要使用 `# LANGUAGE MultiParamTypeClasses #-`。在类型类中有一些函数类型声明和函数定义，比如 fun1 与 fun2。如果把某个类型变为该类型类的成员，需要对该类型类中声明的某几个函数作出对应的实现。比如，Eq 类型类是这样定义的。

```
class Eq a where  
  (==) :: a -> a -> Bool
```

```
(/=) :: a -> a -> Bool
x == y = not (x /= y)
x /= y = not (x == y)
```

其中有两个函数类型声明 (`==`) 与 (`/=`)，实现其中之一其实就实现了另外一个，因为下边函数定义用 (`==`) 定义了 (`/=`)，并且用 (`/=`) 定义了 (`==`)。这里的意思是，实现了其中一个，就会自动地得到另一个。例如，我们定义一个类型 `MyNum`:

```
data MyNum = O | Zero | One
```

将一个类型实现为一个类型类的实例需要使用 `instance` 关键字，例如，将 `MyNum` 实现为相等类型类 `Eq` 的实例的定义如下：

```
instance Eq MyNum where
  O    == Zero = True
  O    == O     = True
  Zero == Zero = True
  One  == One  = True
  _    == _    = False

> Zero /= One
True
```

有一些类型类中的函数已经预定义好了，所以不用再重新定义，只需要将实现该类型类的类型写出即可，例如：

```
instance Eq MyNum where
```

这里只声明一行，而不实现函数。这里引发的问题是 (`==`) 与 (`/=`) 互相调用，因为在 `Eq` 类型类中它们就是这样定义的，如果我们不给出这就是一个互调递归，那么计算就不会停止。如果在类型类中声明的函数不为这种情况，而是已经全部都定义好了则不会出现这种情形。对于这种特殊情况，本书将在异常处理一节中遇到。

当实现一个参数化类型时，可否相等取决于其中的参数是否可以比较相等。因为我们需要对参数使用对应的 (`==`) 函数，所以参数的类型需要受到限定。比如，将 `Maybe` 类型声明为 `Eq` 类型类的实例时：

```
instance (Eq m) -> Eq (Maybe m) where
  Just x == Just y = x == y
  Nothing == Nothing = True
  _ == _ = False
```

在定义 `Just x == Just y` 时就用到了参数 `x` 与 `y` 的 (`==`) 函数，所以需要使用 `Eq m => Eq (Maybe m)` 来限定。函数定义中使用这样的表达也是一样的道理。

到这里有读者可能有定义函数之间的相等的想法，我们前面提出过这个问题，但是没有给出答案。判定任意的两个函数是否相等是不可行的，因为函数相等，如 `f = g`，意为对于所有的符

合类型条件的参数 x 都有 $f x = g x$ (这里假定 f 与 g 是一元函数, 如果是多元函数则需要将余下的参数依次引入), 即对于同一个 x 作为输入都会输出相等的结果。有一些函数的相等是显然的, 如 $\text{id}=\text{id}$, 再如 $\text{flip const} = \text{const id}$, 其实, 在 λ 表达式中, β 化简 flip const 与 const id 结果是完全相同的, 它们的确是相等的函数。而另外一些函数, 如快速排序、插入排序、归并排序等, 它们的计算过程是不一样的, 按照这个定义它们都是相等的, 只是它们的效率不同。显然不可能有某个方法判断任意两个函数是否相等, 所以也就无法将函数实现 Eq 类型类实例。

定义类型类时, 也可以加入一些对于类型类的约束, 也就是类型类间的依赖关系。比如, Haskell 中 Ord 类型类:

```
class (Eq a) => Ord a where
  ...
```

由于 Ord 类型类是依赖于 Eq 的, 因此无论是通过导出实现 Ord 类型类还是自定义来实现, 都必须先将 Eq 类型类实现才可以, 例如:

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun deriving Ord
```

我们会得到如下的错误:

```
No instance for (Eq Day)
  arising from the 'deriving' clause of a data type declaration
Possible fix:
  add an instance declaration for (Eq Day)
  or use a standalone 'deriving instance' declaration,
  so you can specify the instance context yourself
When deriving the instance for (Ord Day)
```

这段错误的意思是说, GHC 没有找到 Day 类型实现 Eq 类型类的实例, 因为 Ord 依赖于 Eq 类型类。GHC 还给出了解决方法, 用户可以自己定义 Eq 的实例, 也可以使用 Haskell 自动导出的实例。这里让 Haskell 自动导出:

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun deriving (Eq, Ord)
```

一个类型如果实现了 Ord 类型类, 那么所有限定于 Ord 类型类的函数就都可以使用了, 如最大值函数 maximum 、最小值函数 minimum , 以及各种排序算法等。

从上边的例子可以知道, Ord 类型类的定义依赖于 Eq 类型类, 即类型 a 需要先实现 Eq 类型类才能实现为 Ord 类型类的实例。但是, 在定义类型类的过程中需要注意, 多个类型类间的依赖不可以成环, 比如:

```
class (A a) => B a where
  ...
class (B a) => C a where
  ...
class (C a) => A a where
  ...
```

如果需要现在实现一个 A 类型类的实例，由于它依赖于 C 类型类，因此要先实现 C 类型类的实例，而 C 类型类又依赖于 B 类型类，那么，实现 C 之前要实现 B 类型类的实例，可是 B 类型类又依赖于 A 类型类。因此，如果定义中有环，就是不可能实现的。

9.2 Haskell 中常见类型类

9.2.1 常用类型类

本节将介绍 Haskell 常用的类型类，即 `Ord`、`Bounded` 和 `Enum`、`Show`。之前，简单地介绍过它们，但本章中，将了解在 Haskell 中它们是如何定义的。`Ord` 类型类的定义如下：它规定实现 `Ord` 类型类至少需要定义 `compare` 或者 `<=` 函数。

```
class (Eq a) => Ord a where
    compare      :: a -> a -> Ordering
    (<), (=<), (>) , (=>) :: a -> a -> Bool
    max, min     :: a -> a -> a

    compare x y = if x == y then EQ
                   else if x <= y then LT
                   else GT

    x < y = case compare x y of { LT -> True; _ -> False }
    x <= y = case compare x y of { GT -> False; _ -> True }
    x > y = case compare x y of { GT -> True; _ -> False }
    x >= y = case compare x y of { LT -> False; _ -> True }

    max x y = if x <= y then y else x
    min x y = if x <= y then x else y
```

`<=` 与 `compare` 函数互相定义，所以定义一个就可以了。因为 `<=` 定义了 `max` 与 `min` 函数，所以，当定义了 `<=` 或者 `compare`，就直接得到了 `min` 与 `max` 函数。那么，一个类型实现了 `Ord` 类型类后，我们可以直接使用这些函数。

```
class Bounded a where
    minBound :: a
    maxBound :: a
```

`Bounded` 类型类的实例都理论上都是可以遍历的，所以都可以实现 `Enum` 类型类，反之却不能成立。比如，`Int` 是有界的，也是可以遍历的，而 `Integer` 可以遍历但没有界。

```
class Enum a where
    toEnum      :: Int -> a
    fromEnum    :: a -> Int

    succ, pred   :: a -> a
```

```
enumFrom    :: a -> [a]          -- [n..]
enumFromThen :: a -> a -> [a]      -- [n,n'..]
enumFromTo   :: a -> a -> [a]      -- [n..m]
enumFromThenTo :: a -> a -> a -> [a] -- [n,n'..m]
```

一般来说，只需要定义 `toEnum` 与 `fromEnum` 就算实现 `Enum` 类型类了，这样就在 `Int` 与该类型间做了一个映射。由于下面 6 个函数的定义使用的都是 `toEnum` 与 `fromEnum`，所以在实现了 `Enum` 后，它们就都可以使用了，读者可以自己试验一下。

在 Haskell 中，对于那些没有参数的构造器，即零元构造器来说是可以通过 `deriving` 关键字来实现 `Enum` 类型类的，例如上一章中定义的星期等。

上一章中定义了 `Nat` 自然数类型，并且还定义了在 `Nat` 与 `Int` 类型间转换的函数，所以可以很容易地将 `Nat` 实现为 `Enum` 的类型类。但是到这里，有些读者可能会有这样一个疑问：`Enum` 类型类的 `toEnum` 和 `fromEnum`，两个函数用的 `Int` 类型是有界的，而有限的 `Int` 值是不能对应无限多个 `Nat` 或者 `Integer` 值的。那么，它们应该如何更准确地实现 `Enum` 类型类呢？GHC 中的任意精度整数 `Integer` 在遍历时可以超过 `Int` 的最大值，是因为它的定义与读者熟悉的定义类型的方法不同。它用到了原始类型、`ByteArray` 还有外部函数接口等。关于这几部分的内容，以及如何精确地定义 `Nat` 与 `Integer` 类型为 `Enum` 类型类的实例将不在本书中讨论。

如果想要一个类型以 `String` 的形式输出在命令行上，就需要实现 `Show` 类型类。如果直接用 `deriving` 关键字导出它的实例，那么显示的内容和定义的是完全一样的。但有时不想这样，此时就可以自己定义 `Show` 类型类中的 `show` 函数。

```
data MyNum = One | Two | Three

instance Show MyNum where
  show One = "1"
  show Two = "2"
  show Three = "3"

> One
1
```

除定义 `show` 函数外还可以有另外一种选择，就是定义 `showPrec` 函数。在介绍它之前，先来看一下 `ShowS` 类型：

```
type ShowS = String -> String
```

`ShowS` 为一个函数。当一个值的类型为 `ShowS` 时，它还需要一个字符串为参数，将这两个字符串结合起来。借助 `ShowS` 类型，使用复合函数来连接各个字符串时所用的时间为常数级。函数 `shows :: (Show a) -> a -> ShowS`，可以将一个可显示的类型计算为一个 `ShowS` 类型。此外，为了处理转义字符的单引号与双引号，Haskell 中还定义了 `showChar` 与 `showString` 函数。

数，如图 9-1 所示。

> show "string"	> show 'c'	>showString "Hello" " World"	> showChar 's' ""
"\"string\\""	'c''	"Hello World"	"s"

图 9-1 显示类型类相关函数

同样，使用 `Shows` 也很容易将多个类型的值通过复合函数连接显示出来，例如：

```
> (shows 5.showString " is my lucky number".showChar '!') " How about you?"
"5 is my lucky number! How about you?"
```

这样就不用再使用`++`来连接多个字符串了，显得整洁了许多。`showPrec :: Int -> a -> Shows` 函数就是为此而定义的函数。显然，通过附加一个空字符串也就定义了 `show` 函数。当数据构造器为运算符时，`showPrec` 会根据第一个参数来打印括号。在实现 `Show` 类型类时我们常常可以忽略这个参数。

9.2.2 Functor

到这里相信读者已经对 `map` 函数还有列表相当熟悉了，但是当我需要把 `Maybe`、树等参数化的类型中的值也像列表中那样映射为其他类型的值，如果不借助类型类就可能需要这样定义：

```
data List a = Nil | Cons a (List a) deriving (Show, Eq)
data Maybe a = Nothing | Just a deriving (Show, Eq)
data Tree a = Leaf | Node a (Tree a) (Tree a) deriving (Show, Eq)

mapList :: (a -> b) -> (List a -> List b)
mapList = ...
mapMaybe :: (a -> b) -> (Maybe a -> Maybe b)
mapMaybe = ...
mapTree :: (a -> b) -> (Tree a -> Tree b)
mapTree = ...
```

相信读者已经看出来了这三个 `map` 函数的公共部分，它们除类型构造器不同以外，其余部分都是相同的。为了把这种公共的部分抽象出来，Haskell 中就定义了函子类型类。函子类型类中仅仅定义了一个函数 `fmap`，意为给定类型 `a` 与类型 `b` 间有一个映射，可以返回另一个参数化类型上的映射，这个参数化类型的 `kind` 必须为`* -> *`。这样 `fmap` 实质上是一个以函数作为输入，以函数作为结果的高阶函数。比如，我们可以将`[a]`通过一个类型为`a->b`的函数映射成为类型为`[b]`的结果，而 `Functor` 中的 `fmap` 是对其他容器这一性质的抽象并且为 `fmap` 函数定义了运算符`<$>`，它们在 `Data.Functor` 中的定义为：

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

为了使用方便，库中为 `fmap` 定义了运算符 `infixl 4 <$>`：

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

`Functor` 需要一个类型 `f` 作为参数，这个类型的 Kind 为`*->*`。对于列表类型，构造器 `f` 为`[]`，而这个 `fmap` 函数与 `map` 是一样的。这样 Haskell 中通过定义的函数类型类总结出了这种映射关系，其他容器（如树、`Maybe`、`Either` 等类型）都可以很容易地实现为函数类型类的实例，这样就通过类型类而将功能相似的函数整理在了一起。

```
newtype Containter a = Containter a
instance Functor Containter where
    fmap f (Containter a) = Containter (f a)

instance Functor [] where
    fmap = map

instance Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

函数也是构造出的类型，`(->) r` 也为一个函数。这里，读者可能有些疑惑了，`(->) r` 类型其实本质与其他的参数化类型并无不同。接下来看`(->) r` 是如何实现函数类型类的：

```
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x)) --或用 η 化简写做 fmap f g = f ∘ g
```

在这种情况下，整个 `fmap` 类型为`(a -> b) -> ((->) r a) -> ((->) r b)`，我们可以将`->`写为中缀符号`(a -> b) -> (r -> a) -> (r -> b)`，可以很清楚地看到，`fmap` 实际上是将函数 `f` 与 `g` 复合了，函数类型类实例的声明可以写成：

```
instance Functor ((->) r) where
    fmap = (.)
```

当然，树也是容器，它也可以实现函数类型类，读者可以自己定义它。一般来讲，所有的容器类型构造器均可实现函数类型类，这是一个较为普遍的现象，但是在实现时要满足下面的定律。

函数在数学上要符合以下的定律：

`fmap id = id`

如果，使用 `id` 函数进行映射，那么与 `id` 函数本身是一样的。

`fmap (f . g) = fmap f . fmap g`

第二条定律则说明 `fmap` 函数在复合函数运算符上是服从分配律的。

值得注意的是，实现了 `Functor` 的 `fmap` 函数，但不满足 `Functor` 定律则该类型不是 `Functor`。在实现 `fmap` 函数时，除了在 `a`、`b` 两个类型间做映射以外，如果还做了其他的操作，那么很可能就打破了函数的两个定律。比如：

```
data Container a = Container a Int
instance Functor Container where
    fmap g (Container x i) = Container (g x) (i+1)
```

这样就破坏了第一条定律，因为使用 `fmap` 时，在 `Container` 中有加法操作，`fmap id`

与 `id` 的计算结果就不相等了。同样，它也破坏了第二条定律，`fmap (f . g)` 只应用了一次 `fmap` 函数，加法也只应用了一次，而 `fmap f . fmap g` 则会应用两次加法。所以，在定义函子的时候需要注意，除了映射以外，不可以有其他的操作。

练习

定义这样一个树类型，`data Tree a = Leaf a | Branch (Tree (a,a)) deriving Show`，它的 `Branch` 中用一个二元元组来表示二叉树，请比较一下它与我们之前定义的树的不同，然后把这个树类型实现为函子类型类的实例。（提示：这里的类型是嵌套定义的，所以在匹配 `Branch x` 时讨论 `x` 是没有用的，你需要想如何变换映射的函数 `f`。）

9.2.3 Applicative

上一节介绍了 `Functor` 函子类型类，实现函子类型类以后，可以通过一个函数将容器内的一种类型的值映射到另外一种类型的值。比如，从 `Maybe a` 映射到 `Maybe b`，但是如何能更好地在这些类型上应用已有的函数呢？比如，我们需要想计算 `Just 1` 与 `Just 2` 的和，最后得到 `Just 3`。显然 Haskell 的标准语法不支持使用 `Just 1 + Just 2`，这里可以把 `Maybe Int` 实现为 `Num` 类型类的实例，但是这里我们需要的是更为一般地把任意的函数应用到任意类型的构造器内的值。在 Haskell 中，`Applicative` 类型类已经帮助我们做了这样的工作，所以我们可以直接使用，它被定义在了 `Control.Applicative` 库中。

```
>:m +Control.Applicative
> Just (+) <*> Just 1 <*> Just 2
Just 3
```

其实，如果 `Maybe` 的定义只有 `Just` 构造器，那么它与普通的加法运算是没有差别的，就如同 `(+)` 一样。可是由于 `Maybe` 类型有着 `Nothing` 来表示计算失败的结果，因此当参数或者函数为 `Nothing` 时，结果也为 `Nothing`。

```
> Just (+) <*> Nothing <*> Just 2
Nothing
```

下面来看 `Applicative` 是如何定义的。

```
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

首先，它的定义是基于函子类型类的，所以在实现它的时候需要先实现函子类型类。定义中的 `pure` 函数可以将一个值或者函数存放于 `f` 容器中，`<*>` 运算符就是进行计算的函数了。下面就是 `Maybe` 类型实现 `Applicative` 类型类实例的声明。

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> arg = fmap f arg
```

很明显，如果有 Nothing 参与计算，那么则结果为 Nothing，在这里，Nothing 可被称为`<*>`运算符的零元，它就像乘法运算符的 0 一样，任何数乘以 0 结果都为 0。如果第一个参数不为 Nothing，那么计算的过程只是将容器中的函数从参数做一个映射。例如，前面的 Just (+) `<*> Just 1 <*> Just 2`。

```
>:t Just (+) :: Num a => Maybe (a -> (a -> a))
Just (+) <*> Just 1 <*> Just 2
= (fmap (+) Just 1) <*> Just 2
= Just (1+) <*> Just 2
= fmap (1+) (Just 2)
= Just 3
```

它应用的函数与前面用到的`(\$)`运算符是类似的。

```
(\$) :: (a -> b) -> a -> b
(<*>) :: f (a -> b) -> f a -> f b
```

下面来比较一下 Functor 与 Applicative Functor 中二元运算符的区别，其类型分别为：

```
(<$>) :: (a -> b) -> f a -> f b
(<*>) :: f (a -> b) -> f a -> f b
```

Applicative 类型类除了具有 Functor 的特性以外，能做的只是调用函数容器内的函数。因此，常常把这个类型类称作 Applicative Functor，意为可应用函数的函数，如 Maybe 类型。

```
> (+) <$> Just 5 <*> Just 4
Just 9
> pure (+) <*> Just 5 <*> Just 4
Just 9
```

为了更方便的表达这种计算，GHC 的库中定义了 liftA、liftA2、liftA3 函数，lift 意为抬升，意思是将一个函数运算的参数分别放置于实现了 Applicative 类型类的容器中：

```
liftA :: Applicative f => (a -> b) -> f a -> f b
liftA f a = pure f <*> a
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b
> liftA2 (+) (Just 5) (Just 4)
Just 9
```

除此之外，Applicative 中还定义了另外两个运算符：`<*>`和`*>`。

```
(*>) :: f a -> f b -> f b
u *> v = pure (const id) <*> u <*> v
(<*>) :: f a -> f b -> f a
u <*> v = pure const <*> u <*> v
```

在使用`*>`运算符时，先计算第一个参数并忽略其结果（这里注意忽略结果并不等于计算没有意义），然后再计算第二个参数但将其返回为结果，`<*`则刚好相反。用户也可以很容易地使用`liftA2`函数来定义它们，然后应用 η 化简。

```
(*>) = liftA2 (const id)
(<*) = liftA2 const

> Nothing <* Just 1
Nothing

> Nothing *> Just 1
Just 1

> undefined *> Just 1
*** Exception: Prelude.undefined
```

由于容器内都为常数，因此可以暂时理解为直接返回第一个或者第二个参数。但是，在第3个测试中可以看到，由于`undefined`是一个错误值，虽然`*>`只需要返回第二个参数，但是还是需要对`undefined`进行匹配计算，当然，它抛出异常了。这里值得留意一下的是`<*>`、`<*`和`*>`运算符的结合性与优先级，它们都是左结合的，优先级都为4。

```
infixl 4 <*>, <*, *>
```

列表类型也是可以实现`Applicative`类型类的。若将列表实现为`Applicative`的实例后就得到了类似于列表内包的表达方法。

```
instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x <- xs]
```

比如，想表达`[x+1 | x <- [1,2,3,4]]`，即`map`函数，可以写做：

```
>(\x-> x+1) <*> [1,2,3,4]
[2,3,4,5]

>(\x-> x+1) *> [1,2,3,4]
[2,3,4,5]
```

表达`[x+y | x<- [1,2], y<- [3,4]]`可以写做：

```
>(+) <*> [1,2] <*> [3,4]
>[(+)] <*> [1,2] <*> [3,4]
[4,5,5,6]
```

表达`[f x y | f<- [(+), (*)], x<- [1,2], y<- [3,4]]`可以写做：

```
>[(+), (*)] <*> [1,2] <*> [3,4]
[3,4,6,8,4,5,5,6]
```

读者应该还记得上节中的 $(-\>) x$ 类型，它也可以实现为Applicative类型类的实例：

```
instance Applicative ((->) x) where
    -- pure :: a -> (x -> a)
    pure x _ = x
    -- (<*>) :: (x -> a -> b) -> (x -> a) -> x -> b
    (<*>) f g x = f x (g x)
```

这里对两个函数的类型做了注释，以便读者阅读。从定义可以看出，二元运算符 $<*>$ 需要“三个”参数。为什么是三个参数？其实函数式编程学习到这里，参数个数在这里不应该是一个让人迷惑的问题了，因为以 $(x -> a -> b) -> (x -> a) -> (x -> b)$ 类型的函数，当给定两个参数后，返回的是一个一元函数：

```
> :t (<*>) (+) (+1)
(<*>) (+) (+1) :: Num b => b -> b
```

这个一元函数也正是想要的结果，即 $((-\>) x) b$ ，只不过这里的 x 与 b 的类型相同，均为 $\text{Num } b \Rightarrow b$ 。虽然看上去略有些复杂，但其实它的计算过程很简单：

```
> (<*>) (+) (+1) 5
11
(<*>) (+) (+1) 5
= (5+) (5+1)
= 11
```

实现的Applicative类型类实例应该满足以下定律：

单位元 (identity) <code>pure id <*> v = v</code>

单位元应该很容易理解，比如，乘法的单位元为1，但是为1乘以任何数都是那个数本身，这里也是一样的，这里的`pure id`为 $<*>$ 运算符的单位元。例如：

```
> pure id <*> Just 5
Just 5
```

复合定律 (composition) <code>pure (.) <*> u <*> v <*> w = u <*> (v <*> w)</code>
--

复合定律可以理解为想要复合容器内的函数，所以它与一般的函数复合是相似的(f,g,h)

$x = f (g (h x))$:

```
> (Just not) <*> ((Just even) <*> (Just 5))
Just True
> not (even 5)
True
> pure (.) <*> (Just not) <*> (Just even) <*> (Just 5)
Just True
> (not,even) 5
True
```

同态定律 (homomorphism) `pure f <*> pure x = pure (f x)`

同态定律可以理解为在容器外使用`<*>`使用函数计算的结果等于在容器内直接应用函数`f`。

```
>Just (+1) <*> Just 5
Just 6
>Just (5+1)
Just 6
```

互换定律 (interchange) `u <*> pure y = pure ($ y) <*> u`

互换定律如下：

```
> Just (+1) <*> pure 5
Just 6

> pure ($ 5) <*> Just (+1)
Just 6
```

互换定律中用了`$`运算符，这里`y`为`$`运算符的第二个参数，所以`(\$ 5)`的类型应为：

```
> :t ($ 5)
(\$ 5) :: Num a => (a -> b) -> b
```

也就是说，`pure ($ y) = pure (\f -> f \$ y)`，所以有：

```
pure (\f -> f \$ 5) <*> Just (+1)
= Just (\f -> f \$ 5) <*> Just (+1)
= Just ((+1) 5)
= Just 6
```

可以看出互换定律通过`$`运算符用`y`参数构造了一个 λ 表达式，而这个 λ 表达式需要一个函数作为参数，这样，`u`内的函数就可以放在`pure ($ y)`表达式之后以达到调用函数的目的。

9.2.4 Alternative

刚刚讨论了 Applicative 类型类，本节来讨论另外一个很重要的类型类 Alternative。Alternative 意为可二选一的、后备的。这个类型类依赖于 Applicative 类型类，并且在 Control.Applicative 中是这样定义的：

```
infixl 3 <|>

class Applicative f => Alternative f where
    empty :: f a
    (<|>) :: f a -> f a -> f a
```

这个类型类中定义的运算符`<|>`意为其他的选择。比如，当给定多个值的时候，需要选择合理的，如下：

```
> Nothing <|> Nothing <|> Just 1 <|> Just 2
Just 1
```

Nothing 代表了计算的失败，那么就不要选它，遇到第一个不失败的计算结果为止。比如，使用折叠函数来取得一个列表中第 1 个不为 Nothing 的值：

```
> foldr1 (<|>) [Nothing, Nothing, Just 1, Just 2]
Just 1
```

为了实现具有这样一个性质的类型类，Maybe 类型的实例应定义为：

```
instance Alternative Maybe where
    empty = Nothing
    Nothing <|> p = p
    Just x <|> _ = Just x
```

通过上面的函数计算与<|>的定义，读者应该可以看出，empty 定义的为 <|> 运算符的单位元，正如<*>运算符的单位元为 pure id 一样。

列表类型可以代表一些结果可能有多种情况的计算，比如，将 [(1,2), (2,4), (1,3)] 根据元组第一个元件排序，结果可以为 [(1,2), (1,3), (2,4)] 或者为 [(1,3), (1,2), (2,4)]。这时，需要将所有可能的结果全部保留，结果应为 [[(1,2), (1,3), (2,4)], [(1,3), (1,2), (2,4)]]，所以列表类型实现 Alternative 类型类实例时，<|> 运算符应该与++运算符是等价的。

列表也是可以实现 Alternative 类型类的，当计算失败时可以返回 []：

```
instance Alternative [] where
    empty = []
    (<|>) = (++)
```

这里 [] 为 ++ 运算的单位元，因为任何列表与 [] 进行 ++ 操作，所得的结果都是它本身。Alternative 类型类中还定义好了另外两个函数 some 与 many 定义，使用了 Applicative 的 <*> 与 Alternative 的 <|> 运算符：

```
some :: f a -> f [a]
some v = some_v
    where many_v = some_v <|> pure []
        some_v = (:) <*$> v <*> many_v

many :: f a -> f [a]
many v = many_v
    where many_v = some_v <|> pure []
        some_v = (:) <*$> v <*> many_v
```

some 函数先检查 v 值是否为 empty 单位元，如果为单位元，那么返回这个单位元，否则返回 v 内的值多次，即尝试返回 v 内的值 1 次或者更多次。而 many 函数先判定 v 值是否为 empty 单位元，如果为单位元，那么返回 []，意为返回 v 内的值 0 次或者更多次。而定义中的本地函数 many_v 与 some_v 为互调递归定义，若 some_v 不能返回 1 次以上，则返回 []，这正是 many 的意思。同理，将 v 内的值返回一次，然后再返回多次也正是 some_v 的意思。例如：

```
> some Nothing
Nothing
```

`Nothing` 为 `Maybe Alternative` 类型类中`<|>`的单位元，所以 `some` 试着想返回 1 次以上 `Maybe` 容器类型内的值，得一个 `Just [a]` 为结果，但是它失败了，于是返回 `Nothing`。如果在 GHCi 中运行 `some (Just 5)`，那么函数的运行不会停止。因为 5 为一个纯数值，所以它的计算在 Haskell 中永远不会失败，其结果可以理解为 `Just [5,5,5,5..]`。而 `many` 函数为：

```
> many Nothing
Just []
```

`many` 函数也试图取得 `Maybe a` 容器类型内的值，但是它只能从 `Nothing` 中取 0 次，于是只有返回 `Just []`。而 `many (Just 5)` 的结果就与 `some (Just 5)` 相同了，它从 `Just 5` 中一直取 5 而不会停止。

库中还定义了 `optional` 函数，即返回 `v` 中的值 0 次或者 1 次，如果没有值，则会返回 `f Nothing`。

```
optional :: Alternative f => f a -> f (Maybe a)
optional v = Just <$> v <|> pure Nothing

> optional [1,2,3]
[Just 1,Just 2,Just 3,Nothing]

> optional []
[Nothing]
```

9.2.5 简易字符识别器

本节先不继续讨论 Haskell 中的其他类型类了。因为相信很多读者看到这里，觉得上边讨论的这么多理论内容在实践编程中是没有用处的。先来看一个实践的例子，意在说明现实中却恰恰相反，这些类型类其实是有用的。`many` 与 `some` 函数在计算某一特定的值时是不会失败的，所以会一直进行下去。例如 `many [5]`，但是对于一个实现了 `Alternative` 类型类的类型 `f`，`f` 内的计算可能失败也可能成功，那么 `many` 与 `some` 的定义就非常有意义了。下面来定义一个非常简单的字符串识别器，看一下这些类型类的用途，顺便可以进一步熟悉一下这些类型类与讨论过的函数。需要下面的库：

```
import Control.Applicative
import Data.Char
```

首先，定义 `Parser` 类型。在识别字符串的时候，常常希望字符串满足要求的格式，如日期 `1998-01-01`。如果识别成功，那么最后剩余的将为空字符串。可是有时也需要识别失败。因为识别字符串的过程常常不能一次性地完成，也有可能字符串不符合要求。比如，识别日期先要识别年，然后横线，再然后月等，识别时的输入的类型也可能不是合法的，如 “`efds-fe-ii`”，还有可能月份大于 12 等。所以，在定义时需要使用 `Maybe` 类型，那么 `Parser` 类型就可以定义

为 `String -> Maybe (a, String)` 的类型，如果识别失败，就返回 `Nothing`，如果这部分识别成功，就返回成功的部分 `a` 与余下的字符串。所以，一个字符分析器类型可以定义为：

```
newtype Parser a = Parser { runParser :: String -> Maybe (a, String) }
```

首先来实现 `Functor` 类型类，这个非常容易。

```
instance Functor Parser where
    fmap f p = Parser $ \str -> case runParser p str of
        Just (a, s) -> Just (f a, s)
        Nothing -> Nothing
```

再来实现 `Applicative` 类型类。`pure` 函数非常容易实现。对于`<*>`运算符，下面的代码中参数 `fp` 有着 `Parser (a->b)` 的类型即 `String -> Maybe (a -> b, String)` 类型，在运行了第一个识别器 `fp` 后，只需要将它 `Just` 构造器中的函数 `ab` 应用到第二个识别器 `a` 的结果中去就可以了，这也正是 `Applicative` 类型类的意义。

```
instance Applicative Parser where
    pure a = Parser $ \str -> Just (a, str)
    (<*>) fp a = Parser $ \str -> case runParser fp str of
        Nothing -> Nothing
        Just (ab, s) -> case runParser a s of
            Nothing -> Nothing
            Just (at, s1) -> Just (ab at, s1)
```

最后实现 `Alternative` 类型类。失败的识别可以返回 `Nothing` 作为结果。所以`<|>`要定义为：如果用第一个识别器识别失败，那么使用第二个识别器识别；如果成功，那么返回结果即可。

```
instance Alternative Parser where
    empty = Parser $ \_ -> Nothing
    (<|>) a b = Parser $ \str -> case runParser a str of
        Nothing -> runParser b str
        just -> just
```

为了方便地使用这个识别器，来定义几个函数。首先为 `satisfy` 函数，即给定一个判定条件，如果满足，就识别，否则返回 `Nothing` 作为失败的结果。使用 `satisfy` 就可以定义一个 `char` 函数来识别特定的字符了，只需要定义成 `satisfy (==c)` 就可以了。

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy f = Parser $ \str -> case str of
    [] -> Nothing
    s:ss -> if f s then Just (s, ss) else Nothing

char :: Char -> Parser Char
char c = satisfy (==c)
```

下面，在 GHCi 中测试一下 `many` 与 `some` 函数，读者可以看出它们的不同，`many` 可以识别 0 次，返回空字符串，而 `some` 一定要识别一次以上，如果第一次失败就返回 `Nothing`。分

别用 many 与 some 识别 “123abc” 与 “abc” 中的数字多次与一次以上为例：

```
> runParser (many (satisfy isDigit)) "123abc"
Just ("123", "abc")

> runParser (many (satisfy isDigit)) "abc"
Just ("", "abc")

> runParser (some (satisfy isDigit)) "123abc"
Just ("123", "abc")

> runParser (some (satisfy isDigit)) "abc"
Nothing
```

<*> 运算符在这种情况下就是很有用的了，它可以帮助用户忽略一些不需要的部分。比如，在用语法分析器分析 HTML 的时候，只需要尖括号内的标签而可以忽略括号。所以，可以写成 `char '<' *> many (satisfy isDigit) <* char '>'`。当输入为 “<321>” 时，在计算 `char '<'` 后余下的字符串为 “321>”，但是识别的结果被忽略了。它只返回 `many (satisfy isDigit)` 识别 “321>” 的结果，所以看到下边的函数的结果中只有 321 而没有尖括号了。

```
> (runParser $ char '<' *> many (satisfy isDigit) <* char '>') "<321>"
Just ("321", "")
```

下面，定义一个识别器来直接得到一个 `Int` 类型。第一步需要将一位的数字字符用 `digitToInt` 转型为整数，然后每识别一位数字字符，分析成功的整数乘以 10 再加上刚刚识别的数字作为识别的结果。它的定义如下：

```
number :: Parser Int
number = fmap (foldl (\x y -> 10*x+y) 0) (many digit)
  where digit = fmap digitToInt (satisfy isDigit)

> runParser num "12345a"
Just (12345, "a")
```

在识别字符串的时候，常常需要做连续的识别，识别了一个字符后还需要对后面的字符串识别。这样，定义一个 `sequ` 函数来辅助用户做这件事情。可以连续地识别字符，就可以定义另外一个函数来识别一个字符串了。通过使用 `foldr` 函数，定义它很容易。

```
sequ :: Parser a -> Parser [a] -> Parser [a]
sequ x y = Parser $ \str -> case runParser x str of
  Nothing -> Nothing
  Just (s,ss) -> case runParser y ss of
    Nothing -> Nothing
    Just (s1,ss1) -> Just (s:s1,ss1)
```

这时，识别字符串的函数就可以定义为：

```
parseStr :: [Char] -> Parser [Char]
parseStr strs = foldr sequ (Parser $ \str -> Just ("",str)) [char s | s <- strs]
```

```
> (runParser (parseStr "hello")) "helloworld"
Just ("hello","world")
```

这样，语法字符识别器就实现好了。前面只定义了几个很简单的函数，实现了几个类型类就得到了可以分析上下文无关文法（Context Free Grammar）的函数了。到这里，相信读者对于 `Functor`、`Applicative`，以及 `Alternative` 有了一定的了解。关于字符识别和语法分析的内容，将在第 13 章关于 `Monad` 转换器的部分里继续深入讨论。下面来看一些其他重要的类型类。

本节的内容主要参考了由 Graham Hutton 教授与 Erik Meijer 的一篇文章——《Monadic Parser Combinators》，可以登录 <http://eprints.nottingham.ac.uk/237/1/monparsing.pdf> 下载。另外，还有在微软剑桥研究院任职的 Daan Leijen 与 Erik Meijer 写的《Parsec: Direct Style Monadic Parser Combinators for The Real World》，有兴趣的读者可以打开链接 <http://research.microsoft.com/pubs/65201/parsec-paper-letter.pdf> 阅读。

9.2.6 Read 类型类

之前讨论了一个简易字符识别器。其实，Haskell 中的 `Read` 类型类做的事情就是前面介绍过的，但是库中定义的类型是这样的：

```
type ReadS a = String -> [(a, String)]
```

这里库中使用的是列表，而在上面一小节使用的是 `Maybe`，可读类型类中定义了一个函数：

```
class Read a where
  readsPrec :: Read a -> Int -> ReadS a
```

其中的类型为 `Int` 的参数为构造器的优先级，这里可以不考虑。如果列表中只有一个元素，那么它的使用是与 `Maybe` 的版本是一样的。如果结果的列表为空，则说明解析失败了。如果分析器会得到多种可能的结果，那么说明语法有歧义，可以将分析的结果全保留，也可以取得第一个元素，需要根据实际情况来定。前面由于讨论了 `Maybe` 的实现，这里的 `Read` 就不再做过多阐述了。

9.2.7 单位半群（Monoid）

数学中的半群（semigroup）是闭合于一个有结合性质的二元运算之下的集合 S 构成的代数结构。更严谨地说，对于集合 S 和基于 S 的二元运算 $\oplus : S \times S \rightarrow S$ ，若 \oplus 满足结合律，即 $\forall x, y, z \in S$ ，则有 $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ ，有序对 (S, \oplus) 被称为半群。倘若在半群 (S, \oplus) 上还存在一个单位元 1 ，那么这三元组 $(S, \oplus, 1)$ 为组成的代数结构就称为单位半群（也作幺半群），即满足 $\{\exists 1 \in S, \forall s \in S, 1 \oplus s = s \oplus 1 = s\}$ ，单位元 1 亦作“幺元”。

十二进制的钟表的时间及其加法运算可以理解为一个单式半群。集合为 $\{1..12\}$ ，运算符为 $+$ ，那么 12 则为单位元——对于任意的时间，加上 12 小时以后所得到的时间和当前时间相同。

再比如, $(\mathbb{Z}, \min, +\infty)$ 和 $(\mathbb{Z}, \max, -\infty)$, 即不论 a 为何值, $\min(+\infty, a)$ 的结果总是 a 。如果将定义的数据类型简单地理解为一个集合, 那么很多数据类型可以抽象为单位半群, 也就是说, 它们都存在一个满足结合律的二元运算和一个单位元:

```
class Monoid a where
    mempty :: a
    mappend :: a -> a -> a
```

多个实现单位半群类型类的类型所组成的元组也是单位半群。

```
instance (Monoid a, Monoid b) -> Monoid (a,b) where
    mempty = (mempty, mempty)
    (a1,b1) `mappend` (a2,b2) =
        (a1 `mappend` a2, b1 `mappend` b2)
```

很多类型都为 `Monoid` 类型类的实例, 下面对它们进行一一列举。

```
instance Monoid a => Monoid (Maybe a) where
    mempty = Nothing
    Nothing `mappend` m = m
    m `mappend` Nothing = m
    Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)
```

`(Bool, &&, True)` 与 `(Bool, ||, False)` 均为单位半群, 因为 `True` 是 `&&` 的单位元, 并且 `&&` 运算是满足结合律的。同理, `(Bool, ||, False)` 也为单位半群。

```
newtype All = All { getAll :: Bool }
deriving (Eq, Ord, Read, Show, Bounded)

instance Monoid All where
    mempty = All True
    All x `mappend` All y = All (x && y)

newtype Any = Any { getAny :: Bool }
deriving (Eq, Ord, Read, Show, Bounded)

instance Monoid Any where
    mempty = Any False
    Any x `mappend` Any y = Any (x || y)
```

$(\mathbb{Z}, *, 1)$ 与 $(\mathbb{Z}, +, 0)$ 也均为单位半群。1 为乘法的单位元, 并且乘法是满足结合律的。同样, 在整数集上的+法与 0 也可以构成一个单位半群。

```
newtype Product a = Product { getProduct :: a }
deriving (Eq, Ord, Read, Show, Bounded)

instance Num a => Monoid (Product a) where
    mempty = Product 1
    Product x `mappend` Product y = Product (x * y)

newtype Sum a = Sum { getSum :: a }
```

```

deriving (Eq, Ord, Read, Show, Bounded)

instance Num a => Monoid (Sum a) where
    mempty = Sum 0
    Sum x `mappend` Sum y = Sum (x + y)

```

列表是一个也单位半群，它的二元运算符为`++`，即字符串拼接（concatenation）运算符：

```

instance Monoid [a] where
    mempty = []
    mappend = (++)

```

`(f:{a->a}, ., id)`也为单位半群，若函数`g`有具有类型`a -> a`，根据高阶函数一章中复合函数的一节可知，常值函数为复合运算符的单位元，即`id.g = g = g.id`，并且它满足结合律，即`(f.g).h = f.(g.h)`。

```

newtype Endo a = Endo { appEndo :: a -> a }
instance Monoid (Endo a) where
    mempty = Endo id
    Endo f `mappend` Endo g = Endo (f . g)

```

之前在复合函数中提到的`(.)`运算符是一个将函数从右到左复合于一起的运算符，此外还提到了一个运算符将函数从左至右地复合在一起，`(f:{a->a}, >>>, id)`，它也是单位半群，但在 Data.Monoid 中没有被定义，可以给它命名为`FunApp`。

```

newtype FunApp a = FunApp { appFunApp :: a -> a }
instance Monoid (FunApp a) where
    mempty = FunApp id
    FunApp f `mappend` FunApp g = FunApp (g . f)

```

在高阶函数一章中讨论了`foldr`与`foldl`函数，这里可以看出，实现`Monoid`类型类实例的列表是可以被其对应的二元运算符折叠为一个值的。比如：

```

compose :: [a -> a] -> a -> a
compose = foldr (.) id

```

这个例子正对应`Endo`单位半群。同理，`foldr (+) 0`对应的正是`sum`单位半群，`foldr (*) 1`对应的正是`Product`单位半群。所以，可以使用`foldr`的函数与类型也具有一类性质的，在 Haskell 中被抽象成为了`Foldable`类型类，下节来简单介绍一下`Foldable`类型类。

9.2.8 Foldable 与 Monoid 类型类

所有的`Foldable`类型类的实例也可以理解为一个容器。实现`Foldable`的类型可以折叠为一个值。虽然在定义时 Haskell 库 Data.Foldable 中并未要求所有的`Foldable`均为函数`Functor`类型类的实例，但事实证明，满足`Foldable`类型类的实例都为`Functor`的实例。实现`Foldable`类型类至少要实现`foldMap`或者`foldr`函数中的一个：

```

class Foldable t where
    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

```

由于刚刚学习 Monoid 类型类，下面可以使用 foldMap 函数来测试一下。

```

>:m +Data.Monoid
>:m +Data.Foldable
> foldMap (Any.(even)) [1,2,3,4]
Any (getAny = True)

```

值得提的一个细节是，Any 是一个类型为 `Bool -> Any` 的构造器，它不能直接与 even 复合。比如，`Any.even` 是错误的，因为除了函数的复合，它还可以被解释为是名为 Any 模块下的 even 函数，这样就有冲突了，用户可能会得到 `Not in scope: `Any.even'` 的错误。所以，有时构造器与函数复合需要写成 `((Any).even)` 或者 `(Any.(even))`。

下面，将树定义为 Foldable 类型类的实例：

```

import Data.Foldable
import Data.Monoid

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

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

```

实现了 Foldable 类型类，就可以很容易地定义一个函数，将树“压平”（flatten）而成为一个列表：

```

flatten = foldMap (: [])
> flatten (Node (Leaf 1) 2 (Leaf 3))
[1,2,3]

```

同样，将一个类型为 `Num a => Tree a` 的树求和的函数也非常容易定义：

```

sumTree = foldMap Sum
> sumTree (Node (Leaf 1) 2 (Leaf 3))
Sum {getSum = 6}

```

读者应该可以感觉到，实现了这些类型类对于定义函数的精简起到了非常重要的作用。用户不必再像原来那样要很啰嗦地定义 `flatten` 与 `sumTree` 函数等。

这里了解了 `Monoid` 中二元运算符的结合性，了解了它的单位元，同时也了解了 `Foldable` 类型类中的折叠函数。接下来，可以来了解一下折叠函数的二元性定理（duality theorem）了。

这里的“二元性”指的是两种相对立的情形一定同时存在，并且可以互相转换。

(1) 若 (M, \oplus, u) 为一个单位半群，其中 M 为类型， \oplus 为二元运算符， u 为单位元，那么有 $\text{foldr } \oplus \ u \ xs = \text{foldl } \oplus \ u \ xs$ 。它称为第一二元性定理，其中 xs 为有限元素的列表。

(2) 如果存在两个二元运算符 \oplus 、 \otimes ，还有一个值 e ，满足 $x \oplus (y \otimes z) = (x \oplus y) \otimes z$ ，且 $x \oplus e = e \otimes x$ ，那么 $\text{foldr } \oplus \ e \ xs = \text{foldl } \otimes \ e \ xs$ 。它称为第二二元性定理，其中 xs 为有限元素的列表。它是第一定理更为一般的形式。

(3) 对于所有的有限列表 xs ，都有 $\text{foldr } f \ u \ xs = \text{foldl } (\text{flip } f) \ e \ (\text{reverse } xs)$ 。它称为第三二元性定理。

以上三个定义的意义在于将使用 foldr 的函数转换为了使用 foldl 的函数。 foldr 函数是通过扩展递归定义的而 foldl 是通过尾递归定义的。所以，如果使用得当，这些定理会帮助用户在计算函数时使用较少的内存空间。比如， $\text{foldl}' (+) 0$ 的空间使用效率就是要比 $\text{foldr} (+)$ 高的，通过第一二元性定理很容易知道它们是相等的。由第二与第三二元性定理可知， $\text{foldl}' (\text{flip } (:)) []$ 同 foldr snoc [] 是等价的。

本节内容参考了(Bird, 1998)的第4章，这本书中对各个性质提供了证明，有兴趣的读者可以深入研究，证明一下。

9.2.9 小结

本节讨论了 Haskell 中一些比较重要的类型类，它们是 Functor、Applicative、Alternative、Monoid 与 Foldable。它们之间关系如图 9-2 所示。

图 9-2 中部分实线箭头表示在定义这些类型类时用到了这个关系，而虚线箭头表示在 Haskell 中定义这些类型类时是没有用到这些关系的，但理论上它们是符合这种关系的。这里的虚线表示了实现 Alternative 类型类的类型都可以实现 Monoid。由于它们有着这种关系，因此有时也称 Alternative 为基于 Applicative 函子的单位半群。可以比较一下这两个类型类的定义，它们几乎是相同的。

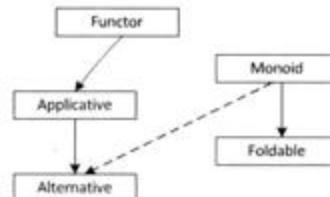


图 9-2 常用类型类关系示意图

```

class Monoid a where
  mempty :: a
  mappend :: a -> a -> a

class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
  
```

`mempty` 对应 `empty`，`mappend` 对应 `<|>`。在将列表实现为 `Alternative` 时，读者应当还记得列表类型实现 `Alternative` 时的 `<|>` 的定义就是 `(++)` 运算符，并且这两个运算都既为左结合

又为又结合的。

有时，我们不是很确定类型类中的类型参数有什么样的 kind，但却发现没有办法向 GHCi 询问类型类定义中类型参数的 kind 签名。在新版本的 GHC 中，类型类被定义成为了一种统一的 kind，称为 constraint kind，我们可以通过使用-XConstraintkinds 编译器参数来对它们进行查询（新版本的 GHCi 已经不需要设置了），比如：

```
> :set -XConstraintKinds
> :k Functor
Functor :: (* -> *) -> Constraint
> :k Monad
Monad :: (* -> *) -> Constraint
```

这样，借助 GHCi 中的:type、:kind、:info 等命令我们就可以方便地查询 Haskell 中定义的值、类型、类型类的信息了。

9.3 类型类中的类型依赖

在 Haskell 中，可以定义多参数的类型类（multi-parameter typeclass），而之前的类型类都只有一个参数，如 class Eq a where。这样，判断两个 Haskell 中的值是否相等的机制则不能用于两个不同的类型，即不能比较(5::Int)==(5.0::Float)。但是，如果希望做到这一点，则需要声明两个类型参数。所以在定义时，需要在文件首处声明要在此文件中定义多参数的类型类，即{-# LANGUAGE MultiParamTypeClasses #-}。

```
{-# LANGUAGE MultiParamTypeClasses #-}
class GEq a b where
    equals :: a -> b -> Bool

data Nat = Zero | Succ Nat

instance GEq Nat [a] where
    equals a b = eq a b
    where eq Zero [] = True
        eq (Succ n) (_:xs) = eq n xs
        eq _ _ = False

> equals (Succ Zero) []
True

> equals (Succ Zero) []
False
```

又从上面的例子可以看到，实现了 GEq 类型类，在类型 Nat 与 [a] 之间定义了一个相等的函数，即可以比较一个自然数是否与一个列表“相等”。这里的相等并不是它们真正意义上的相同，而是任意定义的一种“等价”。直到这里，定义的类型类还并不存在歧义，但是当类型类中的类型参数作为声明函数返回结果的类型时，则有可能产生歧义，例如：

```
(-# LANGUAGE MultiParamTypeClasses #-}
class Fun a b where
    fun :: a -> b

instance Fun Int Nat where
    fun a = Zero

instance Fun Int Int where
    fun _ = 0
```

若在 GHCi 中运行 `fun (5::Int)`，则会引起歧义，因为返回的类型可以是 `Nat`，还可以是 `Int`。那么，为了消除歧义，无论有多少个 `Fun` 类型类的实例，一定要使用`::`来明确结果的类型得到想要的结果，如`(fun (5::Int))::Int`。

但是，有时在实现类型类的实例时，我们可以保证对于给定的类型 `a` 仅仅只有一个与之对应的类型 `b`，即类型 `a` 决定 `b` 的类型，或者类型 `b` 依赖于类型 `a`，此时，不必在每一次调用函数时都声明结果的类型。因为在调用类型类中的函数时，对于给定类型为 `a` 的数值，返回的结果类型只可能是 `b`，所以不必声明返回结果的类型，GHC 也可以知道去调用对应的函数。这样，为了不必每一次调用函数的时候都声明结果的类型，要在文件首处声明`(-# LANGUAGE FunctionalDependencies #-)`，并且在定义类型类时声明类型参数间的依赖关系。

```
(-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies #-}
class Fun a b | a -> b where
    fun :: a -> b
instance Fun Int Nat where
    fun a = Zero

> fun (5::Int)
Zero
```

这里类型类定义 `Fun a b` 后面的 `a -> b` 意为类型 `a` 决定类型 `b`。那么，对于一个类型 `Int`，不可以同时对应 `Nat` 和 `Int` 两种类型，所以一定要舍去一个。从这个例子中可以看出，`a` 与 `b` 的关系看做是基于类型映射，也就是说 `b` 的类型依赖于 `a`，给定一个 `a` 之后 `b` 的类型才可确定，这种类型间的关系称为函数依赖 (functional dependency)。因为函数中不可以有一个输入对应多个输出的情形，这样它就定义了从一些类型到另一些类型的映射关系，这种映射与函数相同，但是是定义在类型层面上的，即不可能会有一个类型类的实例 `instance Fun a b`，同时还有另外一个 `instance Fun a c`，这样在类型推断的时候，给定类型 `a` 时结果的类型 `b` 可以直接被 Haskell 的类型系统推断出来。

Haskell 中加法的类型为 `Num a => a -> a -> a`，但正如==那样，不能将两个不同的数字类型用加号相加，比如：

```
> (5::Int) + (5.0::Float)
<interactive>:1:13:
Couldn't match expected type `Int' with actual type `Float'
```

```
In the second argument of `(+)', namely `(5.0 :: Float)'
In the expression: (5 :: Int) + (5.0 :: Float)
In an equation for `it': it = (5 :: Int) + (5.0 :: Float)
```

错误提示说 GHC 期望的“+”的第二个参数类型为 Int，而给定的却是 Float，是不能这样计算的。所以，如果想进行这样的计算，可以为加法定义一个更为一般的类型类：

```
{-# LANGUAGE MultiParamTypeClasses #-}
import GHC.Float
class (Num a, Num b, Num c) => GPlus a b c where
    plus :: a -> b -> c

instance GPlus Int Float Float where
    plus a b = fromIntegral a + b

instance GPlus Int Float Double where
    plus a b = fromIntegral a + float2Double b

instance GPlus Double Double Double where
    plus a b = a + b
```

同理，这样定义的类型实例就会产生歧义了。需要用如下的定义来消除这种歧义：

```
{-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies #-}
class (Num a, Num b, Num c) => GPlus a b c | a b -> c where
    plus :: a -> b -> c
```

向量的数量积与外积，比如，把“有乘积”这种性质定义为一个类型类：

```
{-# LANGUAGE MultiParamTypeClasses #-}
class Mult a b c where
    mult :: a -> b -> c

data Vector = Vector Int Int Int deriving (Eq, Show)

instance Mult Vector Vector Int where
    mult (Vector x1 y1 z1) (Vector x2 y2 z2) = x1*x2 + y1*y2 + z1*z2

instance Mult Vector Vector Vector where
    mult (Vector x1 y1 z1) (Vector x2 y2 z2) = Vector (y1*z2-z1*y2) (z1*x2-x1*z2)
                                                (x1*y2-y1*x2)
```

此外，一个整数也可以与一个向量相乘，那么可以这样实现：

```
instance Mult Int Vector Vector where
    mult i (Vector x y z) = Vector (i*x) (i*y) (i*z)
```

定义两个向量值：

```
m1 ,m2 :: Vector
m1 = Vector 1 1 1
m2 = Vector 1 1 0
```

很明显 `m1 `mult` m2` 的类型是有歧义的，因为它的结果的类型可以是 `Int`，也可以是 `Vector`。

```
>:t m1 `mult` m2
m1 `mult` m2 :: Mult Vector Vector c => c
```

这里，`c` 的类型只可能是 `Int` 或者 `Vector`，那么，类似于 `(m1 `mult` m2) `mult` m1` 的表达式就是有歧义的，可以使用 `::` 来声明想要的结果的类型。但是，如果不想让这种歧义发生，就可以声明类型间的依赖关系。改写文件头与类型类的定义如下：

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

修改 `Mult` 类型类的定义如下：

```
class Mult a b c | a b -> c where
    mult :: a -> b -> c
```

这样就会得到下面的错误提示。这段错误提示的意思是说，实现的两个类型类的实例之间与声明的类型类参数间的依赖关系有冲突。

```
Functional dependencies conflict between instance declarations:
  instance Mult Vector Vector Int
    -- Defined at C:\src\FunDep.hs:17:10-31
  instance Mult Vector Vector Vector
```

这里 `a b -> c` 的意思是 `a` 与 `b` 的类型唯一决定 `c` 的类型，可以理解为这是一个基于类型的函数 $(a, b) \rightarrow c$ ，对于特定的 (a, b) ，结果 `c` 是唯一的，即对于特定的 `a` 与 `b` 不可以有两个 `c` 类型的实例，所以必须舍去其中的一个来达到消除类型歧义的目的。另一种乘积可以再定义一个类型类。

使用这种方式在现实编程中是很有用的，这里再给出一个例子。比如，定义一个容器的类型类，一个容器类型类中定义了三个基本的操作，这个容器为空，插入一个新元素，判断一个元素是不是在这个容器中。这里，需要使用 `FlexibleInstances` 编译器参数是因为 `a` 与 `[a]` 之间有依赖关系，而默认情况下，GHC 不允许这种情况的发生。

```
{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}
class Collection e ce where
    empty :: ce
    insert :: e -> ce -> ce
    member :: e -> ce -> Bool
```

这里可以将列表实现为 `Collection` 的一个实例。在实现 `Collection` 类型类时用到了两次类型 `a`，所以要对 GHC 使用 `FlexibleInstances` 参数。

```
instance Eq a -> Collection a [a] where
    empty = []
    insert x xs = (x:xs)
    member = elem
```

`empty` 的类型为：

```
>:t empty
empty :: Collection e ce => ce
```

这里，虽然 GHCi 没有报出错误，但 e 类型出现在类型类限制的声明中却不在=>右侧的类型声明里，这样类型是存在歧义的。有类型歧义的定义在可能会导致程序的语义上的歧义，所以在正常情况下，下面的这种定义在 Haskell 中是不允许的：

```
ambiguous :: Eq a => b -> b
ambiguous = id

Ambiguous constraint 'Eq a'
At least one of the forall'd type variables mentioned by the constraint
must be reachable from the type after the '>='
In the type signature for 'ambiguous': ambiguous :: Eq a => b -> b
Failed, modules loaded: none.
```

由于 Haskell 中一个容器只能存放一种指定类型的元素，因此下面的函数调用应当是不合法的。但是，由于定义有歧义，Haskell 还是会给出类型推断。

```
>:t insert 5 (insert 'c' empty)
insert 5 (insert 'c' empty)
:: (Collection Char ce, Collection el ce, Collection e ce, Num e) => ce
```

容器中元素的类型与容器类型显然是有依赖关系的。若容器的类型为 [a]，则其中的元素类型必为 a；如果容器类型为 Tree b，那么其中的元素的类型必为 b。也就是说，容器的类型决定其中元素的类型，所以要通过额外声明一个类型依赖消除这种歧义，Collection 类型类应该定义为：

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

class Collection e ce | ce -> e where
    empty :: ce
    insert :: e -> ce -> ce
    member :: e -> ce -> Bool

>:t insert 5 (insert 'c' empty)
<interactive>:1:8:
  No instance for (Num Char)
    arising from the literal `5'
  Possible fix: add an instance declaration for (Num Char)
  In the first argument of `insert', namely `5'
  In the expression: insert 5 (insert 'c' empty)
```

这样，就不能将类型为 Int 的值插入到一个盛装类型为 Char 的容器中了。关于更多类型依赖的内容，可以阅读由 Gregory J. Duck 和 Simon Peyton Jones 等人写的《Sound and Decidable Type Inference for Functional Dependencies》，可以访问 <http://research.microsoft.com/en-us/um/people/simonpj/papers/fd-chr/esop04.pdf>。

9.4 类型类中的关联类型

关联类型（associated type）是另外一种消除歧义的方式，即在类型类中声明一个关联的类型来起到消除歧义的作用，它其实与类型依赖功能是一样的。但是，作者认为关联类型使用起来更加灵活。这里继续以定义 GPlus 类型类为例：

```
{-# LANGUAGE MultiParamTypeClasses ,TypeFamilies #-}
class (Num a, Num b) => GPlus a b where
    type SumType a b :: *
    plus :: a -> b -> SumType a b
```

GHC 参数-XTypeFamilies 允许在类型类的声明中加入相关的类型。在这里，SumType 为类型类 GPlus 所关联的类型。当定义 GPlus 类型类实例的时候，对于 a 与 b 不能存在两对分别同样的类型，即同时定义下面两个有着一样类型的实例是不合法的：

```
instance GPlus Int Int where
    type SumType Int Int = Int
    ...

instance GPlus Int Int where
    type SumType Int Int = Float
    ...
```

所以二者只能选其一，type SumType a b :: * 就意为 a 与 b 两个类型共同决定另外一个类型 c，同类型依赖| a b -> c 要表达的意思是相同的。类似地，定义容器类型类时，使用关联类型的方式定义与之前用类型依赖定义的类型类意思是相同的。

```
{-# LANGUAGE TypeFamilies #-}
class Collection ce where
    type Element ce :: *
    empty :: ce
    insert :: Element ce -> ce -> ce
    member :: Element ce -> ce -> Bool

instance Eq a => Collection [a] where
    type Element [a] = a
    empty = []
    insert x xs = x:xs
    member x xs = elem x xs
```

关于类型关联就简单介绍到这里，更多的相关内容可以浏览 Simon Peyton 和 Chung-C hieh Shan 的演讲稿“Fun with type functions”，演讲稿可以在 <http://research.microsoft.com/en-us/um/people/simonpj/papers/assoc-types/fun-with-type-funs/FunWithTypeFuncs-Apr09.pdf> 中下载，而 Simon Peyton Jones 的演讲可以登录 <http://channel9.msdn.com/posts/MDCC-TechTalk-Fun-with-type-functions> 观看。

小结

在前面的几节里，介绍了 Haskell 中类型与类型类的概念。对初学者而言，Haskell 的类型系统看上去只是一个更严格的 C/C++ 类型系统。有人可能会认为，虽然 Haskell 的类型系统可以给程序员带来程序编译时可以检查出更多错误的好处，与此同时，却大大限制了程序员的发挥。读者也许还会觉得，类型系统能在编译期发现的错误其实很有限，优秀的程序员不该依赖和受制于类型系统。其实，这两种观点都有道理，但 Haskell 一直在进化，它的类型系统也是如此。如今的 Haskell 已经可以让程序员在享受类型检查机制好处的同时，解除掉类型系统加诸程序的绝大多数限制。这使得程序员能通过细致地设计类型，在编译期间检查出其他语言根本无法发现的错误。Haskell 的类型系统相当复杂，下面的几节将会对类型系统做进一步的讨论，但也仅能起到简单介绍和引起兴趣的作用。每节的末尾都为读者留出几道思考题，读者在解决这些问题时，可能需要参考网上的相关资料或者相关书籍。

9.5 定长列表

通常，在使用列表索引运算符 (!!) 时，如果给定的索引太大，就会出现运行错误：

```
> [1, 2] !! 5
*** Exception: Prelude.(!!): index too large
```

此类问题有没有办法在编译时就找出来？换句话说，如果在编译期就知道索引值是 m ，列表长度为 n ，而 $m > n$ ，是否可以光靠编译器就报出错误来？想做到这一点，必须让列表的类型里带有长度信息。要让列表的类型不仅能表达“这是一个整数的列表”，还要让它能提供“这是一个长度为 10 的整数列表”的信息。如果说前者的类型是 List Integer（在 Haskell 里表示为 [Integer]），那么后者的类型应该为 FList 10 Integer，这种类型被称为依赖类型（dependent type），因为类型本身（FList 10 Integer）依赖于值 10。包含依赖类型的类型系统通常非常复杂，Haskell 并没有直接支持依赖类型。在 Haskell 里，类型是类型，值是值，两者之间有严格的区分。为了克服该限制，只能抛弃值层面上的整数概念，直接使用第 8 章中类似的方法将自然数在类型层面上表达出来：

```
(-# LANGUAGE GADTs #-)
data O = VO
data S n = VS n
```

本书约定用 o 代表 0， $(s\ n)$ 代表 $n+1$ 。按此约定，1 表示成 $(s\ o)$ ，2 表示成 $(s\ (s\ o))$ ，依次类推。有了类型层面上的自然数，就可以定义定长列表了：

```
data FList n t where
  FNil :: FList O t
  FCons :: t -> FList n t -> FList (S n) t
```

从上节中可以知道，GADT 的意义在于构造器的类型可以任意指定。这样就减少了常规方

法里的限制。如果用常规方法定义 `FList`, 就只能写成:

```
data FList n t = FNil | FCons t (FList n t)
```

这样一来, `FNil` 的类型就被编译器自动设置为 `FList n t` (长度为 n 的列表), 而不是想要的 `FList 0 t` (`FNil` 是空列表, 长度只可能是 0)。`FCons` 的类型也被设置为:

```
FCons :: t -> FList n t -> FList n t
```

加入新元素后, 列表长度不变, 而此处想要:

```
FCons :: t -> FList n t -> FList (S n) t
```

加入新元素后, 列表长度加 1。测试一下 `FList` 的定义:

```
> :t (FCons 'a' (FCons 'b' (FCons 'c' FNil)))
(FCons 'a' (FCons 'b' (FCons 'c' FNil)))
```

```
:: FList (S (S (S 0))) Char
```

可以看出, 此处给了一个长度为 3 的字符串列表, 编译器自动推断出它的类型是:

```
FList (S (S (S 0))) Char
```

这里可以看到, Haskell 的类型系统是非常强大的, 甚至可以在类型的层面上做计算。这里只是基于自然数的计算, 其实它可以实现更复杂的计算系统。现在可以用一个类型来追踪列表长度的变化了。下面, 讨论如何使用索引、将两个定长列表连接在一起等内容, 因为实现它们需要引入新的类型类。

之前成功地在类型里包含了列表的长度信息, 下面继续讨论它。正如第一部分开始时所说的那样, 用户希望有安全的索引操作, 该操作的任何错误在编译期就能报出来:

```
fIndex :: m -> List n t -> t
```

只有索引值小于列表长度时, 索引操作才有意义。为了实现该限制, 可使用 type class 来规定 `m` 和 `n` 的关系:

```
class FIndex m n where
    fIndex :: m -> FList n t -> t
```

另外, 0 小于一切大于等于 1 的整数, 而对于索引 0, 应当取出列表的首元素:

```
instance FIndex 0 (S n) where
    fIndex 0 (FCons x _) = x
```

当 $m < n$ 时, 则有 $m+1 < n+1$, 而对于索引 $m+1$ 和长度为 $n+1$ 的列表, 去掉列表的首元素, 然后用索引 m 去找剩下长度为 n 的列表:

```
instance FIndex m n => FIndex (S m) (S n) where
    fIndex (S m) (FCons _ xs) = fIndex m xs
```

下面来测试一下：

```
xs = FCons 'a' (FCons 'b' (FCons 'c' FNil))
test1 = fIndex (VS VO) xs
> test1
'b'
```

如果想用索引 3 (即 VS (VS (VS VO))) 去找列表 xs 里的元素：

```
test2 = fIndex (VS (VS (VS VO))) xs
```

编译器会直接报错：

```
No instance for (FIndex O O)
    arising from a use of `fIndex'
    Possible fix: add an instance declaration for (FIndex O O)
    In the expression: fIndex (VS (VS (VS VO))) xs
In an equation for `test2': test2 = fIndex (VS (VS (VS VO))) xs
```

因为只为 $m < n$ 的情况定义了 fIndex 的实例，因此，在 $m=n$ 的情况下，编译器类型检查时找不到实例，从而在实际运行前就有效发现和报告了错误。除了索引操作，定长列表还消除了许多其他操作的运行时异常，把它们的报错时间也都提早到了编译期，例如：

```
fHead :: FList (S n) t -> t
fHead (FCons x _) = x

fTail :: FList (S n) t -> FList n t
fTail (FCons _ xs) = xs

> fHead FNil
Couldn't match expected type 'S n0' with actual type 'O'
    Expected type: FList (S n0) t1
    Actual type: FList O t0
    In the first argument of `fHead', namely `FNil'
In the expression: fHead FNil
```

定长列表在提早发现错误的方面带来了好处，但在类型里增加了不少的信息，正确传递与修改这些信息，是程序编写人的责任。

例如，两个长度分别为 m 和 n 的定长列表，连接后的长度为 $m+n$ 。这种长度的计算过程必须在类型里时刻体现出来，也就是说，定长列表的连接操作，类型应当是下面这样：

```
fAppend :: FList m t -> FList n t -> FList (Sum m n) t
```

其中， $(\text{Sum } m \ n)$ 表示类型领域整数 m 和 n 在类型领域上的和。使用以前章节介绍的 associated types 来正确定义 fAppend 和 Sum。

首先，在程序文件首处加入：

```
(-# LANGUAGE TypeFamilies,
  GADTs, MultiParamTypeClasses,
  FlexibleInstances #-)
```

然后定义类型类：

```
class FAppend m n where
  type Sum m n :: *
  fAppend :: FList m t -> FList n t -> FList (Sum m n) t
```

我们知道，0 加上任何数 n，结果都是 n，而长度为 0 的列表拼接任何长度为 n 的列表 ys，结果都是 ys：

```
instance FAppend 0 n where
  type Sum 0 n = n
  fAppend FNil ys = ys
```

我们还知道， $m+1$ 与 n 的和，等于 m 与 n 的和加 1，而长度为 $m+1$ 的列表，其表头和表尾若为 x 和 xs ，它与长度为 n 的列表 ys 拼接，列表等效于 xs 与 ys 先拼接，再在头部加上 x ：

```
instance FAppend m n => FAppend (S m) n where
  type Sum (S m) n = S (Sum m n)
  fAppend (FCons x xs) ys = FCons x (fAppend xs ys)
```

测试 `fAppend`：

```
test3 = fIndex (VS (VS (VS VO)))
  (fAppend (FCons 1 (FCons 2 FNil))
    (FCons 3 (FCons 4 FNil)))
> test3
4
```

不光是列表拼接，对所有的定长列表操作，都必须在类型上追踪列表长度的变化和关系。这给程序的编写带来了不小的负担。但是，有时候为了消除运行时错误，其实这样的负担是值得的。而在另一些场合，宁愿为了缩短编程周期，而付出运行时可能出错的代价。最快速潦草的原型设计和最细致严格的类型检查是两个极端，程序员需要找到一个最佳的中间点。新版本的 GHC 中的 Kind 系统可以让我们定义类型层面上的计算更加容易，我们也可以使用它来定义各种有特殊功能的列表，但是也会给编写人带来一定的难度，所以没有万能的解决方法。这里可以记住一句话，也许会有帮助：Good design needs good compromise。即，好的设计需要好的折衷与妥协。

练习

1. 你可以将我们定义的定长列表实现为 `Show` 类型类的实例吗？
2. 列表过滤是常见的列表操作。对于定长列表，实现过滤操作会遇到什么困难？你能不能查找更多资料，并提出解决困难的方法呢？
3. 从以前你写过的 Haskell 程序里，找出一个频繁使用列表的程序，把列表都改成本节介绍的定长列表，并解决随之而来的各种问题。

9.6 运行时重载

从前面的介绍可以看到，Haskell 的类型类为用户提供了函数重载功能。例如，在上一章枚举构造类型中，讨论了如何使用 `data` 关键字定义矩形与圆形，也可以再加一些图形，它们有一些共同的特征，比如，它们都有面积、周长等。可以像下面这样定义它们：

```
data Shape = Rect Double Double | Circle Double | ...
```

然而，如果考虑程序的可扩展性，未来可能会定义任意多的新的数据类型来表示新的形状，因而列举的方法不可行，所以，需要将它们的定义分开，然后定义一个 `HasArea` 类型类，让所有图形的类型都实现这个类型类：

```
data Rect = Rect Double Double
data Circle = Circle Double
```

下面为它们定义一个 `HasArea` 类型类：

```
class HasArea t where
    area :: t -> Double
```

在这里，`Rect` 和 `Circle` 数据类型都能通过实例化 `HasArea` 的方式实现 `area` 方法：

```
instance HasArea Rect where
    area (Rect a b) = a * b

instance HasArea Circle where
    area (Circle r) = pi * r * r

> area (Rect 2.0 3.0)
6.0

> area (Circle 1.5)
7.0685834705770345
```

熟悉面向对象语言的读者会很快发现，这种重载是静态的。如果动态生成一个列表，里面放不同的形状，它们都实现了 `area` 方法。`area` 理应可以遍历不同形状组成列表。如果读者学过些面向对象语言，就知道在面向对象语言里，这一点非常容易做到，只要使用一个泛型容器就可以了。比如，在 Java 中只要先定义 `HasArea` 接口，再使用 `Vector<HasArea>` 即可。之前在 Haskell 使用列表容器时，容器存储的都是具体的类型。比如，`[Integer]` 在 Java 中对应的是 `Vector<Integer>`，这里 `Integer` 为对象，`HasArea` 为接口，Java 可以不必区分。但在 Haskell 里则有所不同，这样，如果想在 Haskell 中实现这种功能就可能要面对两个问题。

(1) Haskell 的列表要求所有元素是同样的类型。而 `Rect` 和 `Circle` 虽都实现了 `area` 方法，但却仍然是不同类型。如何把它们放到同一个列表里？

(2) 假使解决了第一个问题，把 `Rect` 和 `Circle` 都转换成了同一个类型，放到同一个列表中。

如何在运行时根据它们实际是 Rect 还是 Circle，从而正确选择相应的 area 方法？

对第一个问题，定义统一的数据类型，称为 Shape。但是前面已经知道了，不能像下面这样直接地把可能的形状都列举出来：

```
data Shape = RectShape Rect | CircleShape Circle | ...
```

那么该怎么办？想要的其实是下面这样的个数据类型：

```
data Shape = (所有实现了 HasArea 类型类型的)
```

这里，可以通过两种方法来表达一样的意思。通过使用 GADT（需要开启 GADT 选项），按上面的要求，Shape 可以定义成：

```
data Shape where
  Shape :: HasArea t => t -> Shape
```

测试一下：

```
> :t [Shape (Rect 2 3), Shape (Circle 1.5)]
[Shape (Rect 2 3), Shape (Circle 1.5)] :: [Shape]
```

加上 Shape 这一层包装之后，就成功地把 Rect 和 Circle 放进了一个列表里。

对第二个问题，为 shape 数据类型实现 area 方法。按照 Shape 的 GADT 定义，所有数据类型，在包装进 Shape 之前，都被强制要求实现 area 方法。因此，从 Shape 里拿出数据结构时，直接调用其对应的 area 方法即可：

```
instance HasArea Shape where
  area (Shape shape) = area shape
```

回到本节开始时的问题，用 area 遍历一个 Shape 的列表。Shape 可能包装了 Rect、Circle，还有可能在未来包装更多的数据类型，表示更多形状：

```
shapes = [Shape (Rect 2 3), Shape (Circle 1.5)]
test1 = map area shapes
> test1
[6.0, 7.0685834705770345]
```

练习

- 试着从 Shape 里取出被包装的数据类型，在此过程中会出现什么问题？
- 如果你了解面向对象语言，请找出一个频繁使用对象和类的实例，并用本节介绍的方法，将它用 Haskell 重新实现。

9.7 Existential 类型

当在定义类型时，类型参数总是需要在 type、newtype 或者是 data 等的左边出现。但在

有些情况下，不想在类型构造器中指定它们，而是将它们限定在右侧，这时就可以用 Existential 类型来实现，需要使用 `XExistentialQuantification` 编译器参数。

这样，就可以用一个构造器来存储不同的类型，比如：

```
(-# LANGUAGE ExistentialQuantification #-}
data Showy = forall a. (Show a) -> Showy a

instance Show Showy where
    show (Showy a) = show a

showType :: [Showy]
showType = [Showy (1::Int), Showy "String", Showy 'c']

> sequence_ $ map (\x -> putStrLn $ show x) showType
1
"String"
'c'
```

如此一来，上一节的问题也可以用 Existential 类型来解决。也就是有了另外一种方法可以表达所有实现了 HasArea 类型类的类型。

```
data Shape = forall a. (HasArea a) -> Shape a

instance HasArea Shape where
    area (Shape a) = area a

shapes :: [Shape]
shapes = [Shape (Rect 2 3), Shape (Circle 1.5)]
```

本章小结

Haskell 中的类型系统是非常强大的。前面所讨论的其实只是 Haskell 类型系统中的一小部分，它甚至强大到可以在 Haskell 的类型系统层面之上实现一个完整的 λ 演算系统，但在这两章里，只是想让读者们对 Haskell 的类型系统有一些较为深入的认识。在第 11 章中还有一个延伸的例子，就是 11.3.5 实现 C 语言中的 `printf` 函数，由于还没有讨论 IO，所以暂时先不讨论。事实上仅仅是借助于 Haskell 的类型实现 `printf` 这样的参数数量可变函数（variable arguments function）是完全没有问题的。这里可以简单介绍一下。

Haskell 中的函数都是有着明确的类型签名的，比如：

```
> :t (+)
(+) :: Num a => a -> a -> a
```

(+) 运算符只能应用到两个参数上，能不能再支持多些的参数或者参数数量不定呢？比如：

```
> plus 3 4
7
```

```
> plus 1 2 3 4
10
```

答案是可以的，而且不会与类型系统冲突。先定义一个 `Addition` 类型类，它可以将整数映射到一个类型 `t` 上：

```
class Addition t where
    add :: Int -> t
```

再将 `Int` 与 `Int -> t` 同时实现为 `Addition` 的类型类，这样，`Int -> Int -> t` 是 `Addition` 类型类的实例，`Int -> Int -> ... -> t` 也是 `Addition` 类型类的实例，这样就可以接受多个参数了。不过，这里需要指定参数与结果的类型。

```
instance Addition Int where
    add x = x

instance (Addition t) => Addition (Int -> t) where
    add i = \x -> add (x+i)

> add (5 :: Int) (4 :: Int) :: Int
9

> add (5 :: Int) (4 :: Int) (3 :: Int) :: Int
12
```

这样就实现了有一个或者多个参数的函数。如果在这里不是很明白也没关系，本书将在 11.3.5 节做更深入地讨论。

在接下来的三章里会讨论 Haskell 至关重要的内容——`Monad`，希望读者可以做好准备。虽然它们不是很难，但是很多内容需要读者在实际的编程中细细体会。

第 10 章

Monad 初步

在 Haskell 中，**Monad** 是一个很重要的类型类，起源于数学中的范畴论。它的引入对函数式编程影响深远，Haskell 甚至还为它增加了一些语法特性。**Monad** 在函数式编程里代表一种伴有某些其他计算或者行为的数据结构，所以它是一个类型类，是一类类型的共同的属性。它可以把某些数据特定的连续的计算行为抽象出来自动处理，而不暴露在外。也就是说，程序员在编写程序时不必在乎 **Monad** 的内部发生了什么，而仅仅是去使用 **Monad** 的一些基本操作，如 `return` 与 `>>=`。**Monad** 的字面意思可理解为“单一”，也有人将 **Monad** 译为“单子”。这里，**Monad** 在一定程度上可以理解为代码的整合、合并，这样可以消减掉重复、冗余代码的一种结构或者编程方式。

在编写函数程序时带来极大便利的同时，**Monad** 将使程序的结构看起来更为清晰，同时也提供了很好的处理异常和程序副作用（side effect）的机制（关于程序的副作用将在第 13 章末讨论）。通过这些特性，将会看到使用 **Monad** 可以非常好地控制程序代码的复杂程度。

Monad 是函数式编程里的重要内容，并且它十分抽象。也正因如此，关于它的参考资料是相当多的。很多学者也对 **Monad** 做了很久的研究，所以想要很好地理解 **Monad** 可能需要一些时间来思考与实践，仅通过这样一本入门的书是不太可能完全掌握 **Monad** 的全部内涵的。建议在学习完本书之后，对 **Monad** 有兴趣的读者可以找些其他相关的英文文献来阅读。

10.1 Monad 简介

先来看一个在 Haskell 中定义简单的四则运算表达式的例子。在讨论过 24 点游戏后，读者应该很清楚应当如何定义表达式树了，与之不同的是，这里使用是整数而非小数。

```
data Exp = Lit Integer
          | Add Exp Exp
          | Sub Exp Exp
          | Mul Exp Exp
```

 | Div Exp Exp

 eval 函数将对表达式树进行递归计算求值。

```
eval :: Exp -> Integer
eval (Lit n) = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Sub e1 e2) = eval e1 - eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Div e1 e2) = eval e1 `div` eval e2
```

表面上看来，这些函数似乎没有任何问题，可是，如果用 eval (Div (Lit 8) (Lit 0)) 的时候，就会得到一个异常，在运算中除数是不能为 0 的。浮点小数除以 0 可以得正无穷大，但整数除以 0 的异常是没有办法解决的。

```
> eval (Div (Lit 8) (Lit 0))
*** Exception: divide by zero
```

这些函数以及它们的类型都没有问题，但是结果却得到了一个异常，这是因为函数类型并没有把全部的情况考虑进来，正确的函数类型下面却掩盖了可能有的异常，表面上没有问题的函数实际上是可能会引起运行时错误的。如果用 Maybe 类型来处理这样的异常，使程序在有可能出现异常的情况下依旧可以返回结果，因为有时用户希望自己来处理结果为 Nothing 的情形，而不是简单地抛出错误，那么借助 Maybe 类型，这个四则运算的估值函数要这样定义：

```
safeEval :: Exp -> Maybe Integer
safeEval (Lit n) = Just n
safeEval (Add e1 e2) = case safeEval e1 of
    Nothing -> Nothing
    Just n1 -> case safeEval e2 of
        Nothing -> Nothing
        Just n2 -> Just (n1 + n2)

safeEval (Sub e1 e2) = case safeEval e1 of
    Nothing -> Nothing
    Just n1 -> case safeEval e2 of
        Nothing -> Nothing
        Just n2 -> Just (n1 - n2)

safeEval (Mul e1 e2) = case safeEval e1 of
    Nothing -> Nothing
    Just n1 -> case safeEval e2 of
        Nothing -> Nothing
        Just n2 -> Just (n1 * n2)

safeEval (Div e1 e2) = case safeEval e1 of
    Nothing -> Nothing
    Just n1 -> case safeEval e2 of
        Nothing -> Nothing
        Just n2 -> if n2 == 0
```

```
then Nothing
else Just (n1 `div` n2)
```

从上面的代码可以很明显地看到，当想要处理这个异常的时候，如果使用 Maybe 将会产生非常多的重复代码，不但长度暴增而且还会显得非常繁琐。现在，要找到重复代码公共的部分，将这些公共的部分用一个简单的函数代替。

通过仔细观察可以看到，用户要做的事情只是把 e1 中的计算结果与 e2 的结果做一个运算，这个运算可能是四则运算里的任意一个。如果 e1 为 Nothing，那么全部计算结果就为 Nothing，不必再继续计算；若 e2 为 Nothing，那么最终的计算结果也为 Nothing，即 e1 和 e2 任意一个有计算时有异常出现都为 Nothing。只有当 e1 和 e2 的计算都无异常，那么才使用它们进行计算。此外，这是一个连续的估值再计算的顺序。首先，需要计算出 e1 的值，然后计算 e2 的值，最后再对它们进行对应的计算。可以写一个函数辅助做这个事情。如果从一个 Exp 里计算的值是 Nothing，那么直接得 Nothing；如果没有计算异常，那么将 Just 内的结果传给另一个函数，至于新的结果有无异常，取决于给定的函数使用 e1 的计算。

```
evalSeq :: Maybe Integer -> (Integer -> Maybe Integer) -> Maybe Integer
evalSeq mi f = case mi of
    Nothing -> Nothing
    Just a -> f a
```

其实，发现这样一个函数的并不是那么容易的，而是通过了很久的理论与实践才得到的。下面来测试一下 evalSeq 函数：

```
> Just 5 `evalSeq` (\x-> Nothing)
Nothing

> Just 5 `evalSeq` (\x-> Just (x+5))
Just 10

> Just 5 `evalSeq` (\x-> (Maybe 6 `evalSeq` (\y-> Nothing)))
Nothing

> Just 5 `evalSeq` (\x-> (Maybe 6 `evalSeq` (\y-> Just (x+y)))
11
```

在编辑的时候，可以将这样表达式的函数分成多行，写成如下的格式，这样将会显得更为工整：

```
mb = Just 5 `evalSeq` \n1 ->
    Just 6 `evalSeq` \n2 ->
        Just (x+y)
```

这样，通过使用 evalSeq 函数，safeEval 函数可以写成下面的形式：

```
safeEval (Add e1 e2) =
    safeEval e1 `evalSeq` \n1 ->
```

```

safeEval (Add e1 e2) =
    safeEval e1 `evalSeq` \n1 ->
    safeEval e2 `evalSeq` \n2 ->
    Just (n1+n2)

safeEval (Sub e1 e2) =
    safeEval e1 `evalSeq` \n1 ->
    safeEval e2 `evalSeq` \n2 ->
    Just (n1-n2)

safeEval (Mul e1 e2) =
    safeEval e1 `evalSeq` \n1 ->
    safeEval e2 `evalSeq` \n2 ->
    Just (n1*n2)

safeEval (Div e1 e2) =
    safeEval e1 `evalSeq` \n1 ->
    safeEval e2 `evalSeq` \n2 ->
    if n2==0
    then Nothing
    else Just (n1/n2)

```

这样就定义好了。比较一下，是不是省了一些重复的代码，并且更为工整了呢？用 evalSeq 来辅助判断是不是 Nothing，如果是，那么整个计算的结果就是 Nothing；如果不是，则把 Just 容器里的结果传给下一个函数继续运算。而下一个函数，则通过 λ 表达式来定义。

从以上的例子和 Maybe 的定义可以知道，Maybe 是代表可能产生失败结果的一种数据结构，可以理解为一个容器。如果计算失败，则结果为 Nothing，什么也没有；如果计算成功，那么返回的结果为 Just a。所以，Maybe 类型可以被抽象为一个 Monad。也可以看出，safeEval 中计算 e1 的结果是否失败、得 Nothing 还是得 Just a 的情况都被隐含在了 evalSeq 函数中，不必在 safeEval 函数中处理。这样，只需要检查 evalSeq 函数的正确性，就可以知道 safeEval 函数的正确性了，evalSeq 为用户做了那些重复工作，这样就省去了很多重复的代码。

本节的例子来自于诺丁汉大学 Henrik Nilsson 教授的《编译原理》讲议的第 9 节——A Versatile Design Pattern: Monads，如果想要了解更多相关的内容，可以在 <http://www.cs.nott.ac.uk/~nhn/G53 CMP/LectureNotes-2011/lecture09.pdf> 下载到讲义。

10.2 从 Identity Monad 开始

之前给了一个例子，相信读者大约对 Monad 应该有些模糊的概念了。其实，Maybe 只是 Monad 家族中的一个小小的成员。为了了解这样一个庞大的家族，可以从最基本的 Monad 开始，一步一步地了解它们。在 Haskell 中，Monad 类型类是这样定义的：

```

class Monad m where
  (">>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a

```

```
(>>) :: m a -> m b -> m b
m >> k = m >>= \_ -> k

fail :: String -> m a
fail s = error s
```

如果一个数据类型要实现 Monad 类型类，需要实现以下两个函数 return 函数与 (>>=) 运算符。return 函数将一个数值 a 映射为 monad 值。(>>=)^① 运算符可以将连续的、从左至右的运算串连起来。

Monad 类型类中还定义了 (>>) 运算符，它不关心前一步产生什么样的结果，但还是会有计算的过程，此后，直接计算并返回第二个参数。fail 函数则用来抛出异常。但 Monad 的计算真的有错误并且不能补救，我们才会用到 fail。虽然 >> 与 fail 在 Monad 类型类中有默认的定义，但是在一些特定的情形下我们可以自己定义它们。

在 Control.Monad.Identity 库中，单位元 Monad (Identity Monad) 是最简单地具备了 Monad 性质的类型，它仅仅是定义了一个容器。它除了可以将一个类型的值装进来，其他的事情什么也不做，也就是说，在顺序试计算的同时不伴有任何其他行为。

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

instance Monad Identity where
    return a = Identity a
    (Identity m) >>= k = k m
```

return a 函数把一个普通的数据转换成类型为 Identity a 的数据，即返回成 Monad 值。这里 return 函数其实就是相当于 Identity 构造器函数，可以通过 η 化简写做 return = Identity，因为两个函数是等价的，而 Identity Monad 单独使用起来也似乎毫无意义。

```
> Identity 5 >>= \a -> Identity (odd a) >>= \b -> Identity (not b)
Identity False
```

它与正常的函数应用是没差别的。

```
> not $ odd $ 5
```

如果读者还记得高阶函数那一章的复合函数中，为了把函数的参数写在函数名的前面我们定义了 (<|>) 一个中缀运算符，这里我们可以演示一下如何使用它

```
> let (||>) = flip ($)
Prelude> :t (||>)
(||>) :: b -> (b -> c) -> c
> 5 ||> \x -> odd x ||> \y -> not y
False
```

我们可以比较运算符 \$> 的类型与 >>= 的类型：

^① 这个符号在英文中常被称为 bind。

```
(|>) :: b -> (b -> c) -> c
(>>=) :: m b -> (b -> m c) -> m c
```

它们是十分相似的，`|>`的目的就在于可以把参数写在左侧，

`Identity` `Monad` 只能进行基于某一类型的计算，并没有其他的行为。但是，如果再给 `Identity` 类型的定义加上另外一个值，那么 `Identity` 则可以来管理基于任意类型 `a` 上计算的异常，这就是 `Maybe` `Monad`。

10.3 Maybe Monad

```
data Identity a = Identity a
data Maybe a = Just a | Nothing
```

这里将数据构造器中的 `Identity` 改成 `Just`，再加一个值，这个值就是 `Nothing`，`fail` 我们不用 `error` 定义，因为 `Maybe` 正是管理异常的类型，下面来实现 `Maybe` `Monad`：

```
instance Monad Maybe where
    return = Just
    (Just a) >>= f = f a
    Nothing >>= _ = Nothing
    fail _ = Nothing
```

`>>=` 符号就是之前的 `evalSeq` 函数了，给定这个函数，将一个 `Monad` 值 `ma` 和一个返回 `Monad` 值的一元函数作为参数 `f`，它会计算一个 `Monad` 值作为结果。这样，`>>=` 将第一个参数的评估结果输入到函数 `f` 中，从而把估值和下一步的计算巧妙地连接了起来。这里，来看使用 `Maybe` 时，`>>=` 与 `>>` 运算符的类型：

```
(>>=) :: (Maybe a) -> (a -> Maybe b) -> Maybe b
(>>) :: Maybe a -> Maybe b -> Maybe b

> return 3 :: Maybe Int
Just 3
```

这样，在做四则运算的时候还伴随着其他的计算，就是异常的处理，这个异常会在 `Maybe` `Monad` 内部将会被自动处理。由于 `Monad` 有很多种，所以在没有类型上下文的情况下，Haskell 是无法做出类型的推断的，需要指定转换的 `Monad` 的类型。下节中，读者将会知道，列表也是一个 `Monad`，所以可以将任意的数据通过使用 `return` 转换成列表。

```
> return 3 :: [] Int
[3]
```

由于 `>>=` 相当于前文例子中的 `evalSeq` 函数，因此它可以将第一个 `Maybe` 的结果传给下一个函数进行计算。`>>=` 可以理解为一个传送带，将 `Monad` 值的数据传递给之后的函数进行计算。

也可以理解为一个链条，将一个 Monad 值与函数相连，这样的连接可以很长。实现了 Maybe Monad，就可以很容易地来定义 10.1 节中的 safeEval 函数了。

```
safeEval (Add e1 e2) =
    safeEval e1 >>= \n1 ->
    safeEval e2 >>= \n2 ->
    return (n1+n2)
```

为什么 $\gg=$ 符号会有 $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$ 这种奇怪的、不对称的类型呢？因为连续的函数计算需要这样的运算符有这样的类型，正如我们在高阶函数中定义的 $|>$ 一样。可以看到， $(\lambda n1 \rightarrow safeEval e2 >>= \lambda n2 \rightarrow Just (n1+n2))$ 是一个 $Int \rightarrow Maybe\ Int$ 类型的函数，而 $\lambda n2 \rightarrow Just (n1+n2)$ 还是有着一个 $Int \rightarrow Maybe\ Int$ 的类型。这种形式保证了 $\gg=$ 符号的左端一直是一个 Monad 值，而右端一直是一个结果为 Monad 值的一元函数。这样， $\gg=$ 符号将连续的计算连接起来就可以始终保持结果是一个 Monad 值。比如，有一个连续使用 $\gg=$ 的运算：

```
\a \rightarrow Just a >>= \b \rightarrow Just (b+5) >>= \c \rightarrow Just (c*3)
```

可以看到， $\gg=$ 的两端均为 $Int \rightarrow Maybe\ Int$ 类型，但如果像图 10-1 所示这样，画出这个表达式的结构，可以看出， $\gg=$ 符号保证了函数的总体的函数类型是 $Int \rightarrow Maybe$ ，而方括号内的部分的类型始终是一个 Monad 值。 $\gg=$ 左端虽然是一个 Monad 值为输入，右端是一个以 Monad 值为结果的一元函数，但这里方括号外是可以有参数输入的。而大多数情况下第一个参数是已经给定的情形，所以表达式的类型一直保持为一个 Monad 值的类型，也就是图 10-2 所示的情形。

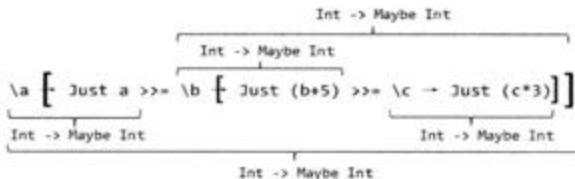


图 10-1 $\gg=$ 运算符类型结构示意图 1

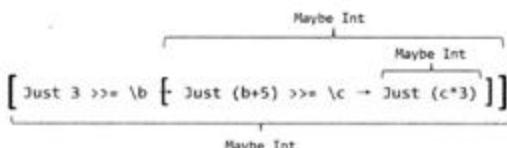


图 10-2 $\gg=$ 运算符类型结构示意图 2

所以说， $\gg=$ 运算符有着 $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$ 的类型也就不足为奇了。其中，每一个类型为 $Int \rightarrow Maybe\ Int$ 的表达式都为一个次计算，比如：

```
> (\b \rightarrow Just (b + 5)) 3
Just 8
```

那么，`>>=`将这些类型为 $a \rightarrow m b$ 的计算像链条一样连接起来，构成一个新的、看起来是由左向右的顺序式的计算。而常常计算的初始的参数是给定的，所以整个计算保持 **Monad** 值的类型。

实现了 **Monad**，就可以使用 `do` 关键字和 `<-` 符号了。`do` 和 `<-` 符号其实是 Haskell 中的一个“语法糖”，比如，C 语言里，有时用 `for` 循环代替 `while` 循环一样，语法糖只是让函数书写起来更加容易，可读性更强，但它们的本质是一样的。比如，以下三个函数第一个是用 `>>=`，后两个是用 `do` 来表达，这三种定义方式的意思是完全相同的。

```
safeEval (Add e1 e2) = safeEval (Add e1 e2) = do
    safeEval e1 >>= \n1 -> n1 <- safeEval e1
    safeEval e2 >>= \n2 -> , n2 <- safeEval e2
    return (n1+n2)           return (n1+n2)

safeEval (Add e1 e2) =
    do { n1 <- safeEval e1; n2 <- safeEval e2; return (n1+n2) }
```

之前，当不借助 **Monad** 的时候，`safeEval` 要定义如下：

```
safeEval (Add e1 e2) = case safeEval e1 of
    Nothing -> Nothing
    Just n1 -> case safeEval e2 of
        Nothing -> Nothing
        Just n2 -> Just (n1 + n2)
```

可以看到，使用 `>>=` 运算符会简化用户的工作，程序异常的部分都被 `>>=` 自动处理了。写成用 `do` 关键字的话，`n1` 可以简单地理解为从 `safeEval e1` 这个容器中提取出来的结果，然后继续运算。实质上 `n1` 和 `n2` 是 `>>=` 符号第二个参数的参数，而根据 **Monad** 运算符 `>>=` 的定义，输入的参数也的确是 `>>=` 符号第一个参数容器中的值，所以，这样理解是合理但却是不精确的。

Applicative 与 **Monad** 类型类中定义的两个函数是十分相似的。不同的是 **Applicative** 实例的二元运算符是用来应用函数的，而 **Monad** 中的 `(>>=)` 是用来表达连续的具有某些伴随效应的计算的。这样，顺序式语言的特性在某种程度上就被 `>>=` 运算符表达了，从而让 Haskell 这种函数式语言有了表达顺序式程序的能力。但是，**Monad** 的意义并不仅仅于此。

理论上，**Monad** 与 **Applicative** 还有 **Functor** 是有联系的，但是在里并没有体现出来。有一个结论是，所有的 **Monad** 都为必为 **Applicative** 和 **Functor**。可以实现一个函数 `ap`，将所有实现 **Monad** 类型类的实例实现为 **Applicative** 来印证这一点。

```
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b

class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b

ap :: Monad m => m (a -> b) -> m a -> m b
ap mab ma = do
```

```
f <- mab
a <- ma
return $ f a
```

10.4 Monad 定律

在讨论函数时有过一个例子：一个类型实现了函数类型类，但是它不符合函数的定律，所以它并不一定真的是一个函数。这里也同样，实现了 Monad 类型类的两个基本函数也并不代表这个类型就具有 Monad 的特性。Haskell 可以帮助用户检查实现 Monad 的函数的类型正确与否，但是不会检查用户所写的 Monad 是否满足 Monad 的定律。一个 Monad 要有以下的定律：

```
> return 5 >>= (\x -> Just (x+6))
Just 11
```

左单位元 $\text{return } x \text{ } >>= f = f \text{ } x$

很明显， $(\lambda x \rightarrow \text{Just } (x+6)) \text{ } 5$ 也得 $\text{Just } 11$ 。把一个值放在 Just 中，再将 Just 中的值通过 $>>=$ 传递给 f 函数，跟直接将这个值应用于 f 函数是完全一样的。

```
>Just 5 >>= (\x -> return x)
Just 5
```

右单位元 $m \text{ } >>= \text{return } = m$

右单位元的意思是，将 m 中的数据通过 $>>=$ 传递给 return 函数，和 m 是一样的。同样，这个定律也很好理解。将 $\text{Just } 5$ 中的数据 5 传给 return ，而 $\text{return } 5$ 又会返回 $\text{Just } 5$ 。

结合律 $(m \text{ } >>= f) \text{ } >>= g = m \text{ } >>= (\lambda x \rightarrow f \text{ } x \text{ } >>= g)$

在定义 $>>=$ 运算符的时候，它被定义左结合，但是根据结合律，左结合与右结合所计算的结果是相等的。例如：

```
>(return 5 >>= \x -> Just (x+2)) >>= \x -> Just (x*2)
Just 14
```

```
>(return 5) >>= (\x -> Just (x+2)) >>= \x -> Just (x*2)
Just 14
```

$>>=$ 符号始终保证左端是一个 Monad 值，右端是一个结果为 Monad 值的一元函数，所以，不论是 $m \text{ } >>= f$ 的结果传入函数 g 进行计算，还是 m 的结果先传入 $(\lambda x \rightarrow f \text{ } x \text{ } >>= g)$ 函数，最终的结果是一样的。所以， $>>=$ 简单理解为一个传送带的话，将 m 中的结果传给 f ，计算得出的结果再传给 g ，然后依次类推。同样， $>>$ 运算符也有着结合的性质：

```
(m1 >> m2) >> m3 = m1 >> (m2 >> m3)
```

Maybe Monad 的相关函数在 `Control.Monad.Maybe` 中，用 `Either` 也可以实现 Monad。

但是它和用 Maybe 实现 Monad 的过程的区别是不大的。这里，Left e 大约等价于 Nothing，但是 e 类型可以表示一些返回的异常信息。

```
instance Monad (Either e) where
    return = Right
    Right m >>= k = k m
    Left e >>= _ = Left e

instance Monad Maybe where
    return = Just
    Just a >>= k = k a
    Nothing >>= _ = Nothing
```

10.5 列表 Monad

列表可以被抽象为一个 Monad。它可以代表一系列结果可能为多个或者可能失败的计算。当计算失败时，则返回空列表 []。定义 $xs \gg= k$ 时，如果列表中有多个元素，那么每一个都需要参与 k 函数的计算，这样返回的也是多个结果，因此计算的结果是不确定的。从某种角度讲，列表 Monad 是 Maybe Monad 更为一般的形式，即结果可能失败，也有可能有多种结果。可以这样定义列表为 Monad 类型类的实例：

```
instance Monad [] where
    return x = [x]
    xs >>= f = concatMap f xs
    fail _ = []
```

如果列表类型实现了 Monad 类型类，使用 do 关键字其实表达也是与列表内包是很相似的，正如列表类型实现了 Applicative 那样。

比如，表达 $[x + y \mid x \leftarrow xs, y \leftarrow ys]$ 实现如下：

```
plus :: Num b => [b] -> [b] -> [b]
plus xs ys = do
    x <- xs
    y <- ys
    return (x + y)

> plus [1,2,3] [4,5,6]
[5,6,7,6,7,8,7,8,9]
```

至于用这种方式实现对应的 map 与 filter 等其他基于列表的相关函数，读者可以自己试着实现一下。

10.6 Monad 相关运算符

为了方便使用 Monad，Control.Monad 中定义的很多关于 Monad 的函数，这里先来看几

个简单的运算符，更多的函数将在下章讨论。

首先来看一下函数的复合的类型：

```
f :: b -> c
g :: a -> b
```

函数 f 与 g 复合的类型如下：

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g :: a -> c
```

$\langle\langle=\rangle\rangle$ 运算符将不同的 Monad 计算从右至左地复合在一起，它与函数的复合是十分相似的。

类似于管道的顺序式计算运算符我们已经提过：

```
(|>) :: a -> (a -> b) -> b
```

而 $(>>=)$ 运算符的类型为：

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
\ a -> Just (ord a) >>= \ b -> Just (odd b)
```

在图 10-1 中， $>>=$ 运算符号类型其实可以当做这样一种类型： $(a -> m b) -> (b -> m c) -> (a -> m c)$ ，库中的确存在这样一个函数，它写做 $>=>$ ：对应我们在复合函数中提到的 $>>>$ 。

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
(>>>) :: (a -> b) -> (b -> c) -> (a -> c)
```

$>=>$ 可以将多个 Monad 计算从左至右地复合在一起，就像 $>>>$ 将函数复合在一起一样。它将多个运算从左至右地连接起来，但是 $>=>$ 定义为 $m a -> (a -> m b) -> m b$ ，在操作上有更多的灵活性。

库中还提供一个函数 $(<\<)$ ： $Monad m => (a -> m b) -> m a -> m b$ ，它是这样定义的 $f = <\< x = x >>= f$ ，相当于 $flip (>=>)$ ，对应的当然就是 $(\$)$ 运算符了。

10.7 MonadPlus

MonadPlus 与 Alternative 很类似。表示在计算时可以有些选择。如果失败，则返回`mzero`，如果有一个成功，那么计算就是成功的。这与 Maybe 中的 $>>=$ 还有 $>>$ 是不同的。使用 $>>=$ 与 $>>$ 时，如果第一个参数为`Nothing`，那么整个计算都为`Nothing`。有的时候为了从`Nothing` 这些失败计算中恢复，可以使用`MonadPlus`。

```
class Monad m => MonadPlus m where
    mzero :: m a
```

```
mplus :: m a -> m a -> m a
```

其中，该 Monad 类型类的实例可以实现单位半群。类似地，也可以实现为 Alternative 类型类。

```
instance MonadPlus m => Monoid (m a) where
    mempty = mzero
    mappend = mplus
```

实现 MonadPlus 类型类与实现 Alternative 类型类几乎是一样的，以列表与 Maybe Monad 为例：

```
instance MonadPlus [] where
    mzero = []
    mplus = (++)

instance MonadPlus Maybe where
    mzero = Nothing
    Nothing `mplus` ys = ys
    xs `mplus` _ys = xs
```

由于该 Monad 满足可以实现单位半群，因此该 Monad 的值、mplus 二元运算与 mzero 满足单位半群的 3 个定律：

```
mzero `mplus` m = m
m `mplus` mzero = m
m `mplus` (n `mplus` o) = (m `mplus` n) `mplus` o
```

并且，mzero 常常为连续运算的零元。其中，它为>>=运算符的左零元，同时为>>右的零元。

```
mzero >>= f = mzero
f >>= (\x-> mzero) = mzero
v >> mzero = mzero
```

mzero 是否需要为>>运算的左零元呢？根据不同需要可以不同定义。如果不能忽略这种计算的失败，那么 mzero 为左零元。根据>>运算符在 Monad 类型类中默认的定义它是左零元的，但有时，如果想忽略这种计算的失败，那么 mzero 则不为左零元。比如，定义一个类型 Possibility，它与 Maybe 类似：

```
data Possibility a = Failure | Success a deriving (Show, Eq)

instance Monad Possibility where
    return = Success
    Failure >>= f = Failure
    Success a >>= f = f a

    Failure >> a = a
    Success a >> b = b
```

这样，在定义 MonadPlus 时，mzero 就不为>>运算符的左零元了。

mplus 函数在 >>=运算符上是符合分配律的。这个定律也很好理解，mplus a b 或者返回

a，或者在 a 计算失败时返回 b。这与或者返回 $a \gg= k$ ，或者 a 计算失败导致 $a \gg= k$ 失败，从而返回 $b \gg= k$ 是一样的。

```
mplus a b >>= k = mplus (a >>= k) (b >>= k)
```

10.8 Functor、Applicative 与 Monad 的关系

由于一些历史原因，Haskell 标准库中的 Functor、Applicative、Monad 等类型类的定义间的依赖关系并不是非常严格的。如果需要重新定义它们的话，我们可以更为严谨的方式来定义这些类型类。

函数类型类之前我们已经讨论过，它的定义为：

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)
```

实质上它是将 a 与 b 类型间的映射照搬到了 $f a$ 与 $f b$ 上，并且我们知道它保留了恒值映射，即满足等式 $fmap id = id$ ，但实质上，这两个 id 函数应用到的类型是不同的，第一个是 $a -> a$ ，而等号右侧的类型为 $f a -> f a$ ，只是因为 Haskell 有多态类型所以不加以区分。另外，函数还保留了复合性，即满足等式 $fmap (f.g) = fmap f . fmap g$ 。

之前并没有介绍过 Pointed 这个类型类，Haskell 中也没有对应的定义。它的定义非常简单：

```
class Functor f => Pointed f where
    point :: a -> f a
```

`point` 函数只是将一个值放到构造器中，得到只有一个元素的数值。比如，`point 5 :: [Int]` 就会得到 `[5]`。很明显，这里的 `point` 函数相当于 Applicative 中的 `pure` 与 Monad 中的 `return`。这种只有一个元素的值称为 `singleton`，这样的函数把一个值注入到类型 f 内，这样的映射称为注入映射（injection）。

在 Functor 与 Pointed 的基础之上，我们可以重新定义 Applicative 与 Monad 类型类。Applicative 做的事情就是将 f 内类型为 $a -> b$ 的函数应用到有着类型 $f a$ 值上。

```
class Pointed f => Applicative f where
    pure :: a -> f a
    ( $\langle\ast\rangle$ ) :: f (a -> b) -> f a -> f b
    pure = point
```

有了前面的铺垫，我们就可以定义 Monad 类型类了。数学上定义的 Monad 并不是采用的 `return` 与 `>>=` 定义，而是通过 `fmap`、`return` 与 `join`，它们是等价的两种定义，因为借助 `fmap` 与 `id` 函数 `join` 与 `>>=` 可以互相定义。

```

class Applicative m => Monad m where
    return :: a -> m a
    join :: m (m a) -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    (>>)  :: m a -> m b -> m b

    return = pure
    join mma = mma >>= id
    (>>=) ma m = join $ fmap m ma
    (>>) ma mb = ma $>>= \_ -> mb

```

从类型签名我们可以看到 `join` 函数类型是 $m(m a) \rightarrow m a$ ，也就是将构造器脱去一层。如果 Haskell 中的 `Monad` 类型类是这样定义的，那么声明 `Monad` 类型类的实例时只需要定义 `join` 或者 `>>=` 其中之一就可以了。数学上，`join` 函数满足下面的定律：

```

join.return = id = join.fmap return
join.join = join.fmap join

```

第一个等式体现的是对于同一种 `Monad` 类型类的实例 m ，该实例的值有着 $m a$ 类型，将这个 `Monad` 值直接返回成类型为 $m(m a)$ 的值，再应用 `join` 函数脱去一层构造器得到 $m a$ 类型的值，即 `join.return :: m a -> m a`。这个过程与先将构造器内部的有着 a 类型的值，用 `fmap` 映射为 $m a$ 类型之后，再用 `join` 脱去一层构造器后所得的结果相同。这里以列表 `Monad` 为例，列表 `Monad` 中 `join` 函数对应的就是 `concat` 函数：

```

(join.return) [2,3]
= join (return [2,3])
= join [[2,3]]
= [2,3]

(join.fmap return) [2,3]
= join (fmap return [2,3])
= join [return 2, return 3]
= join [[2],[3]]
= [2,3]

```

第二个等式表达的也是类似的道理，即将 $m(m(m a))$ 从外向内脱去构造器得到类型为 $m a$ 的结果与先对内部的 $m(m a)$ 通过 `fmap` 应用 `join` 函数得 $m(m a)$ 后再应用 `join` 是一样的。这里反映的是针对构造器 m 的变换顺序对结果是没有影响的。读者可以使用列表 `Monad` 或者其他 `Monad` 试验一下。

使用 `join` 的时候要留意，因为 `join` 去掉了一层 `Monad` 就意味着可能丢失了一些信息，比如：

```

> :m +Control.Monad
> join [[1,2,3],[4,5,6]]
[1,2,3,4,5,6]

```

虽然我们完整地保留了所有的元素，但是从`[[Int]]`到`[Int]`我们将原内部列表长度的信息丢失了。`join [[1, 2], [3, 4, 5, 6]]`也会得到相同的结果。此外，我们不可能再对`[Int]`使用`join`函数了，虽然我们可以使用`sum`等类型将其求和，但这样列表`Monad`的效应将会完全消失。再如`Maybe Int`类型，我们可以对`Just (Just 5)`使用`join`函数，但是`join Nothing`与`join (Just Nothing)`结果都为`Nothing`，这种计算的失败是怎样导致的我们就不得而知了。它是一个多对一的函数，即对于多个不同的输入可能会得到相同的结果。但是一般我们不会将`Maybe a`类型再使用`join`函数计算到`a`类型，因为列表与`Maybe`都是多形式定义的，这样`[] :: [[Int]] -> [] :: [Int]`就会丢失信息，而同理`(Nothing :: Maybe (Maybe a)) -> (Nothing :: Maybe a)`也会丢失信息，但是`Identity Monad`应用一次`join`则不会丢失任何信息，因为我们总是可以从结果推出函数定义域中对应的值，同样一个函数`Identity a -> a`也不会丢失任何信息。

有兴趣的读者可以阅读“Brent Yorgey The Typeclassopedia, The Monad.Reader Issue 13”，链接为<http://www.haskell.org/wikiupload/8/85/TMR-Issue13.pdf>。

本章小结

`Monad`总是代表着有一系列效应(effect)的运算，例如`Maybe Monad`。如果一个计算不会产生失败的结果，那么用与不用`Maybe`是完全相同的。`>=>`与`>>>`运算符号非常相似，`>>>`运算符号仅仅是做连续运算的复合函数，`>=>`会复合结合为`Monad`值的函数，将值放在一个“容器”中可以在计算的过程中有其他的行为或称为效应，`m`类型构造器使得计算时可以伴有更多的信息。比如，`Maybe`中除了`Just`容器，还有可以让计算产生失败的结果`Nothing`，这样就代表了失败的效应。除了本章介绍的`Monad`以外，还有记录函数计算过程的记录器`Monad (Writer Monad)`、维持并连续计算一些状态的状态`Monad (State Monad)`、从某一不变的数据结构中读取信息的读取器`Monad (Reader Monad)`等。这些`Monad`将在第12章中详细介绍。下一章中我们主要介绍Haskell是如何使用`Monad`来处理输入与输出的。

第 11 章

系统编程及输入/输出

在本章中，简要介绍一下在 Haskell 中的系统编程，如日期、时间，从 Haskell 中读入一些系统参数、处理输入/输出等。但是在介绍之前，需要了解一下为什么处理系统输入、输出的部分在本书相对靠后的位置才讨论。

如果读者学过 C 语言，可能首先就会学到，如何在命令行上打印各种值、如何从键盘上读值、访问操作系统文件等，并且它们理解起来很直接。而这本书到第 11 章才真正开始学习用 Haskell 解决输入与输出的问题，这是因为 Haskell 的纯函数的特性。与操作系统互动的过程，比如打印一个字符串在命令行上的 `putStrLn` 函数，当调用了这个函数时，Haskell 做计算了么？如果把它当做函数计算，那么它应当返回什么值作为结果呢？这些在顺序式语言中看似容易的问题，在 Haskell 这样一个纯函数式编程语言中并不能很容易地解决，它花费了 Haskell 设计者们很多智慧与精力。想要了解为什么 Haskell 需要和如何解决这些问题，需要了解一个重要的概念，那就是程序的副作用（side effect）。

11.1 不纯函数与副作用

之前定义的函数都是纯函数（pure function），什么是纯函数？纯函数指的是对于相同的输入，这个函数总是会返回相同的结果，如数学中的正弦函数、`length` 函数、`map` 函数等。而不纯函数则指的是一些函数的返回结果可能与当前计算机的系统状态有关，如返回今天的日期、产生一个随机数、从键盘中读入一个值、改变计算机存储器状态等，这些都是不纯的函数。为了解决实际问题，在 Haskell 中，这些不纯的函数是无法避免的，所以需要一种机制来更好地处理它们。

另一个相关的概念是程序的副作用，它与不纯函数的区别十分微妙。在编程中，程序的副作用所指的是程序的一种效应。有一些函数可能不仅去计算函数的结果，而且连带地引起了其他的计算或者效应。对于最常见的处理输入/输出的相关函数，什么是副作用呢？从操作系统的角度说，这些函数可能会打印一串字符在命令行上、从键盘中读取一行字符串、从磁盘上读取

文件、发送数据到网络的端口、发送信号到打印机等；从计算机存储器状态的角度说更改（*mutate*）内存存储单元中的值，如静态变量（*static variable*）或者全局变量（*global variable*）、在屏幕上画一些点等，这样会改变显示卡的存储状态。再或者，一个函数调用了另外一个有副作用的函数等，以上这些都是程序的副作用。而副作用不仅仅可以指输入/输出操作。副作用的字面意思是当做某一件事情时有着连带的其他的效应产生。例如药物的副作用，人们在服用感冒药时会出现头晕、失眠等副作用。其他有副作用的函数还有不确定性计算、异常处理、并发函数等，本书将一点一点地讨论与理解它们。

为什么要引入这样一个概念？因为一但有了副作用，那么函数的结果将和这些存储状态有关，再或者，程序运行的可能结果决定于函数被调用的顺序或者时间，而非函数本身。

下面来看 C 语言的一段代码（如果读者没学过顺序式语言也没有关系）：

```
int globalVar = 50;

int addGlobalVar (int a){
    return a + globalVar;
}

void setGlobalVar(int a){
    globalVar = a;
}

int foo(int a){
    setGlobalVar(20);
    return addGlobalVar(10);
}
```

其中的 `setGlobalVar` 函数就是对于全局的变量进行操作，而不是返回一个值作为结果，即 `void`。它的参数是多少，`globalVar` 在内存中的值就被设置成了多少。函数返回的值还可能与系统当前的状态有关。例如，`addGlobalVar` 函数，这个函数的结果不但取决于输入的参数，还取决于全局变量 `globalVar` 的值。这样，它返回的结果取决于其他函数对于这个全局变量的操作，这些都是程序的副作用，它们都是不纯的函数。而在 Haskell 中避免了变量，也避免了不返回任何值而直接去操作计算机存储状态的函数，所以，在 Haskell 中如果定义了 `globalVar` 值，那么它在任何函数中的引用都为 50，而且 `setGlobalVar` 也无法实现用纯函数实现，因为 `globalVar` 的值在函数运行过程中无法改变。

```
globalVar :: Int
globalVar = 50

addGlobalVar :: Int -> Int
addGlobalVar a = a + globalVar
```

这些函数不会引起计算机存储器状态的改变，也不会引起其他的效应，不接受从操作系统的输入，而只是计算结果。如果给定参数确定，那么这个函数的结果也是确定的，这样的函数就被称为纯函数。这些函数的调用可以直接被函数计算的结果代替，这种性质被称为引用透明性（referential

transparency)。正是它使得我们可以演绎一个纯函数的运行过程来推导出唯一确定的结果。

可是在实践编程中，程序不可能不与操作系统交流，同时也可能完全不去改变计算机存储器的状态，也就是说，副作用和不纯函数在一定程度上是不可以避免的。为了解决这些问题，Haskell 引入了 Monad。这样，处理前面提到的有关输入/输出、并发与异常处理的函数时，不会破坏 Haskell 的纯洁性，也不需要再对 Haskell 的语法或者语义做任何的改动。

11.2 IO Monad

一个有着 IO Monad 类型的值，如 `IO a`，代表着一个或者一系列的输入/输出的操作，最后返回一个类型为 `a` 的值，如图 11-1 所示。

可以将它理解为一个容器，这样，一个输入/输出操作就对应了一个 Monad 值。比如，`IO Int` 意为进行了一些输入/输出操作后会返回一个整数类型，又如，`IO ()` 可以理解为进了一列表的输入/输出操作返回了 `()` 这个值。同操作系统交流不是一个简单的问题，比如，在从键盘读取参数时不可能保证引用的透明性、读取文件时也可能会引发权限、文件不存在或者文件损坏等异常，这些问题全部都发生在 `IO Monad` 内部，不会影响到其他部分的代码。这样，Haskell 用 `IO Monad` 值代表了输入/输出的操作，它是纯与不纯两个世界间的桥梁，在使用时会产生内存、磁盘或者网络读取等副作用，但是定义纯函数时可以不必在意副作用的产生，因为它们不会发生在纯函数里。

如果将整个计算系统状态当成一个值，这个值的类型为 `OS_State`，那么一个 `IO` 行为可以理解为一个纯函数，它的类型为 `OS_State -> (a, OS_State)`。由于操作系统的状态 `OS_State` 会发生改变，因此将其放于 `IO Monad` 中处理，并将其私有而不暴露给其他的纯函数，这样也就不会破坏纯函数的引用透明性了。其实，它与之后要学习的状态 `Monad` 十分相似。

下面来看两个最基本的处理输入与输出的函数，即 `getChar :: IO Char` 与 `putChar :: Char -> IO ()`，如图 11-2 所示。

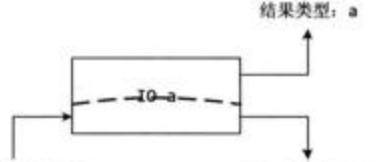


图 11-1 IO Monad 行为示意图

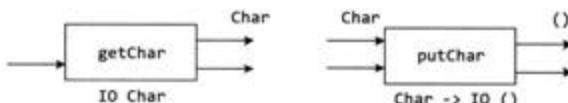


图 11-2 `getChar` 与 `putChar` `IO Monad` 函数示意

`getChar` 函数通过键盘从命令行中读取一个字符，返回一个 `Char` 类型，而 `putChar` 则为一个函数，以一个 `Char` 类型作为参数打印在操作系统命令行上，用户可以在 GHCi 的命令行调用这些函数。

```
> getChar
ab --输入 ab
'a'

> putChar 'a'
a
```

当再一次运行 `getChar` 会有什么结果？GHCi 会直接返回 '`b`'，因为刚刚输入的 `ab` 中的 `b` 被存入了一个缓冲区中，并没有消失：

```
> getChar
'b'
```

在上一章介绍了 `Monad`，`IO` 类型实现了 `Monad` 类型类，因为 `IO` 的行为常常是连续的，比如，从 GHCi 命令行中（见图 11-3）：

```
> getChar >>= putChar
f --输入 f
f
```

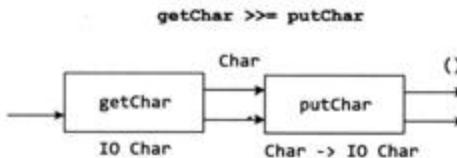


图 11-3 连接 `getChar` 与 `putChar` `IO Monad` 函数示意图

Haskell 也有 `print` 函数，它的类型为 `print :: Show a => a -> IO ()`，也就是说，它可以将所有实现 `Show` 类型类实例的类型打印出来。比如 `print True`、`print 5`、`print "helloworld!"` 等。不从操作系统中读入数据的 `IO` 行为的类型是 `IO ()`，其中，`()` 可以理解为有些顺序式语言的 `void`，但与之不同的是，Haskell 会将 `()` 值作为结果返回，如下：

```
> putChar 'c' >>= print
c()
```

`putChar` 函数打印了 '`c`'，而 `print` 打印了 `putChar 'c'` 的结果，也就是 `()`，所以输出为 `c()`。Prelude 中还提供了另外 3 个处理命令行输入与打印字符串的函数，分别是 `getLine`、`putStr`、`putStrLn`，对应为可以从命令行中读取一行、打印一个字符串和打印一个字符串另起一行。

```
main::IO ()
main = do
    putStrLn "what is your name?"
    name <- getLine
    putStrLn $ "Hello " ++ name
```

```
> main
what is your name?Alexander
Hello Alexander
```

`main` 函数为 Haskell 程序的入口，它的类型必须为 `IO ()`。有了 `main` 函数，程序就可以编译使用了，就像在第 1 章中介绍的那样。`IO` 类型除了实现了 `Monad` 类型类以外，还实现了 `Functor` 和 `Applicative` 类型类，如果需要的话，基于这些类型类的相关函数都可以使用。

从一个函数的类型签名中就可以看出一个函数的行为是否存在副作用。比如，`map toLower` 就是一个纯函数，因为它的类型为 `String -> String`，当给定一个 `String` 为参数，结果是确定且唯一的，它就是纯函数。Haskell 中读文件的函数 `readFile` 的类型为 `String -> IO String`，为什么 `readFile` 不是一个 `String -> String` 的函数呢？因为它的结果与系统状态有关，在读文件期间，可能会有因系统路径不存在、文件正在被加了写入锁、文件权限不够等系统问题引发的异常。即便没有异常，计算机同一个文件可以被修改，每一次返回的内容也就不同，这样就破坏了函数的引用透明性，所以，结果需要放在 `IO Monad` 内处理。再比如，从键盘读取一行字符串的函数 `getLine`，它的类型为 `IO String`，因为每一次读入的字符串是不同的。

使用 `IO Monad` 将带有 `IO` 行为的函数与其他函数分离开来，不会有将不纯的类型作为输入而返回纯类型的函数，比如 `IO String -> String`，因为大家知道 `IO` 构造器是私有的，所以不可能从“不纯的世界”再回到“纯的世界”，输入与输出的行为只能发生在 `IO Monad` 内部。

再来回顾上一章为处理 `div` 函数除数为 0 引发的异常，用的是 `Maybe Monad`。它代表了可能失败的计算，它的异常可能影响到之后的计算。但 `div` 函数是纯函数，首先，它的计算不依赖于系统状态；其次，对于确定的参数它的结果也是确定的，这里除数为 `0:Int` 引发的异常也是一种确定的结果，并且我们很确定什么时候会发生计算异常，所以并未破坏引用透明性。这里引发的异常也有着一个 `Int` 类型，可以详见 15.2 节。那么，如同类型为 `IO String -> String` 的函数一样，如果使用了 `Maybe` 处理了计算失败的话，又定义了一个有着类型为 `Maybe a -> a` 的函数，就可能使得之前用 `Maybe Monad` 处理这种效应变得没有意义了。因为在定义这个函数时，可能没有值与 `Nothing` 匹配而又会导致异常。

引入了 `IO Monad`，那么就可以在 `IO Monad` 中进行一些更改内存状态的工作了，但也仅仅限于 `IO Monad` 内部。`Data.IORef` 库中提供了 `IORef` 容器来以及相应的函数（常用 `IO` 引用函数如表 11-1 所示）来直接引用或者修改容器内部的值，可称为 `IO` 引用。

```
bool :: IO ()
bool = do
    bRef <- newIORef True
    modifyIORef bRef not
    b <- readIORef bRef
    print b
    writeIORef bRef True
    b <- readIORef bRef
    print b
```

```
> bool
False
True
```

表 11-1 常用 IO 引用函数

函 数	功 能
<code>newIORef :: a -> IO (IORef a)</code>	生成一个 <code>IORef</code> 并将给定的参数存入其中
<code>readIORef :: IORef a -> IO a</code>	从 <code>IORef</code> 读取值
<code>writeIORef :: IORef a -> a -> IO ()</code>	向 <code>IORef</code> 写入值
<code>modifyIORef :: IORef a -> (a -> a) -> IO ()</code>	向 <code>IORef</code> 中的值应用函数，类似于 <code>map</code>

这样的表达方式就与一般的顺序式语言差别不大了。此外，在 `Data.IORef` 中还提供了两个函数：

```
atomicModifyIORef :: IORef a -> (a -> (a, b)) -> IO b
mkWeakIORef :: IORef a -> IO () -> IO (Weak (IORef a))
```

第一个是用来处理多线程访问 `IORef` 容器的，当只有一个 `IO` 引用时，`atomicModifyIORef` 可以阻止多个线程访问它可能产生的竞争状态。如果一个值的引用消失了，那么 GHC 的垃圾回收器会自动回收内存。但有时，用户想自己来管理内存就可以用弱指针（weak pointer），关于它的内容将不在本书中讨论，读者可以参阅 API 的 `System.Mem.Weak` 来了解，这里不再介绍了。

如此一来，Haskell 使用 `IO` Monad 这种机制将纯与不纯的函数分开，所以由系统引发的异常一定不在纯的函数中，这对寻找程序中的错误带来了极大的方便。此外，也正是因为 Haskell 纯函数中没有其他语言中所谓“变量”产生的副作用这使得 Haskell 在函数计算时变量可以被共享，并且在并行计算和惰性求值时有了顺序式编程语言所没有的优势。

读到这里，相邻读者也许了解了纯函数式编程语言中“纯”的意思了。纯函数式编程语言不是指完全不可以有处理输入/输出、修改内存状态的函数存在，而是这些函数的这些不纯的行为会通过不同的封装而不暴露给纯函数。比如，当用户从键盘读入了一个字符串后使用 `map toLower` 纯函数来处理这个字符串，这个纯函数对于这个字符串是什么，从哪里来一无所知。这里，`IO` Monad 就代表了输入/输出的行为。`IO` Monad 不但可以处理输入与输出，也能用通过 `IORef` 修改内存变量，这种更改状态的行为在 Haskell 中可以用 `ST` Monad 来代表，有兴趣的读者可以参阅 API 中的 `Control.Monad.ST` 来了解。通过不同的 `Monad`，Haskell 将不同的行为分开，达到了把纯函数和各种有着不同副作用的函数分隔开来的目的，这对程序编写、维护以及除错有很大帮助。

并不是所有的函数式编程语言都可以被称为纯函数式编程语言，如 F#。F# 支持原生可变的数据（mutable data）和循环，所以它是不纯的函数式编程语言。OCaml、Scheme、Clojure 也为不纯的函数式编程语言，但它们都具有“纯”的部分。保证语言的纯洁性还需要程序员来遵守相应的编程纪律。

本节的内容主要参考 Simon Peyton Jones 教授的“Tackling the Awkward Squad:monadic

input/output, concurrency, exceptions, and foreign-language calls in Haskell”，可以打开地址：<http://research.microsoft.com/en-us/um/people/simonpj/papers/marktoberdorf/mark.pdf> 在线阅读。

11.3 输入/输出处理

本节将主要介绍一下 Haskell 中文件的处理。但是在讲文件处理前，先来认识一些基本的 Monad 函数，然后了解如何读取操作系统的环境变量、操作系统命令行参数和程序名。这样，在下节可以实现一个查字典的小程序。

11.3.1 Control.Monad 中的函数

Control.Monad 中的函数如表 11-2 所示。

表 11-2 Control.Monad 中的常用函数

函 数	功 能
forever :: Monad m => m a -> m b	永远执行给定参数
when :: Monad m => Bool -> m ()-> m ()	若符合给定条件执行第二个参数
unless :: Monad m => Bool -> m ()-> m ()	若不符合给定条件执行第二个参数

forever 函数会一直运行给定的 Monad 值产生的行为。比如，在写网络服务监听网络端口时可能就会用到。forever 不会返回任何的结果，但是结果的类型为 mb，说明它可以当做任何类型使用。

```
main = forever (do
    print "Can you tell me your name?"
    name <- getLine
    print ("Hello " ++ name))

> main
"Can you tell me your name?"
Alexander
"Hello Alexander"
"Can you tell me your name?"
Ben
"Hello Ben"
"Can you tell me your name?"
Cherry
"Hello Cherry"
"Can you tell me your name?"
...
...
```

when 与 unless 相当于没有 else 分支的条件判断表达式。when 意为当给定条件成立时执行第一个参数，否则将被忽略。unless 与 when 相反，意为除非当给定的条件成立时执行第二

个参数，否则将被忽略。when 函数在 debug 程序时十分有用，可以将一个值设置为 True 或者 False 来控制程序输出一些信息帮助用户找到错误。例如：

```
debug = True

main = do
    ...
    when debug (print "Debugging")
    ...


```

sequence 函数（如表 11-3 所示）将一系列的 Monad 行为 [m a] 从左至右地执行一遍，并将结果串连起来，成为 m [a]。

表 11-3 sequence 函数

函 数	功 能
sequence :: Monad m => [m a] -> m [a]	将多个 Monad 值串连起来并记录结果
sequence_ :: Monad m => [m a] -> m ()	将多个 Monad 值串连起来并忽略结果

例如：

```
> str <- sequence $ replicate 3 getLine
Haskell
is
fun!
> str
["Haskell","is","fun!"]
```

而有时，在执行某一系列 Monad 行为 [m a] 时，可能不介意返回的结果。比如，结果类型可能为 [()]，这样可以使用 sequence_。

```
fun = sequence $ replicate 3 $ print "Haskell is fun!"

> fun
"Haskell is fun!"
"Haskell is fun!"
"Haskell is fun!"
[(),(),()]
```

而当使用了 sequence_ 时，则不会出现最后的结果的列表。Monad 相关函数名中最后的下划线可以理解为将结果省略掉。

```
fun = sequence_ $ replicate 3 $ print "Haskell is fun!"
```

加这个下划线相当于函数对结果使用 void :: Functor f => f a -> f ()，它可以把返回结果忽略。

```
fun = void (sequence $ replicate 3 $ print "Haskell is fun!")
```

为了更好地处理基于列表的 Monad，库中提供了 replicateM 与 replicateM_ 函数。因此，

之前的函数还可以定义为：

```
fun = replicateM_ 3 (print "Haskell is fun!")
```

也就是将 (print "Haskell is fun!") 这一 IO 行为复制三次，并忽略结果。mapM 与 mapM_ 函数的类型如表 11-4 所示，它们只是在调用 sequence 函数进行了一次函数的映射。

表 11-4 mapM 函数

函数	功能
mapM :: Monad m => (a -> m b) -> [a] -> m [b]	mapM f = sequence . map f
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()	mapM_ f = sequence_ . map f

这两个函数就把纯的值映射为了带有特定 Monad 行为的值，比如：

```
> mapM_ print [1,2,3]
1
2
3
```

此外，库中还提供了与 mapM 等价的定义为 forM 函数，它们在功能上没有区别，仅仅是参数调换了一下位置，借助 flip 函数可以直接定义。但对于一部分读者来说，这样用起来可能更符合直觉，即对于列表中每一个值应用映射基于 Monad 的函数 f，产生一系列的 Monad 行为。

```
forM :: Monad m => [a] -> (a -> m b) -> m [b]
forM_ :: Monad m => [a] -> (a -> m b) -> m ()
```

liftM 会将一个普通的一元函数计算转换成为一个带有 Monad 行为的一元函数计算。它的定义非常容易，但是却非常实用，与讨论 Applicative 类型类时的 liftA 函数是类似的。Control.Monad 中还提供了 liftM2 等函数，即将二元函数化为带有 Monad 行为的计算。

```
liftM :: (Monad m) => (a -> r) -> m a -> m r
liftM f m1 = do { x1 <- m1; return (f x1) }
```

当然，除此之外还有很多函数，比如 foldM、filterM、zipWithM 等，在这里就不过多赘述了，读者可以自己查阅 API 来熟悉它们。

11.3.2 系统环境变量与命令行参数

与系统环境变量及输入参数有关的函数（如表 11-5 所示）在 GHC 的 System.Environment 库中。在这里约定，如果见到 C:\> 的提示符就在操作系统命令行中运行。比如，从系统中读取 PATH 环境变量就可以这样做：

```
>:m +System.Environment
> path <- getEnv "path"
```

```
> path
"C:\Program Files (x86)\Haskell\bin;C:\Program Files (x86)\Haskell
Platform\2012.2.0.0\lib\extralibs\bin;C:\Program Files (x86)\Haskell
Platform\2012.2.0.0\bin;C:\MinGW\msys\1.0\bin;C:\Windows\System32;C:\\
cygwin\bin..."
```

下面的代码展示了 getArgs 函数的功能，它可以从系统命令行中得到参数：

```
--Argstest.hs
import System.Environment

main = do
    args <- getArgs
    case args of
        [] -> print "please input some arguments"
        arg -> mapM_ print args
```

表 11-5 环境变量与命令行参数相关函数

函数	功能
getArgs :: IO [String]	将命令行参数作为 string 的列表读入
getProgName :: IO String	读取程序本身的文件名
getEnv :: String -> IO String	读取指定的系统环境变量

在 GHCi 中，可以使用 :set 来设置系统命令行参数。当然，如果不想编译也可以用 runghc，但在 GHCi 中，测试要方便一些。

```
>:set args first second third
> main
"first"
"second"
"third"

C:\> runghc Argstest.hs first second third
"first"
"second"
"third"
```

GHCi 也支持使用 :main [<arg1>, <arg2>, ...] 的方式来把多个命令行参数传入 main 函数，如下：

```
> :main ["first", "second", "third"]
"first"
"second"
"third"
```

11.3.3 数据的读写

对文件的访问与操作是和操作系统相关的。为了访问文件和文件夹，GHC 库中 System.IO.Windows 与支持 POSIX 标准的操作系统的 System.IO.Posix 库中定义了与文件路径相关的类型与函数，其中文件路径类型定义为字符串。

```
type FilePath = String
```

对于文件或者数据流的操作需要句柄 (handle)。句柄是一个值，当用户访问文件时，就会得到一个句柄，它就像用户在写文本时的光标，在光标的位置进行插入字符，选取文本等操作。`stdin :: Handle, stdout :: Handle` 就是从操作系统命令行输入与输出的句柄。可以通过句柄来得到文件的属性，比如使用 `hFileSize :: Handle -> IO Integer` 来得到文件的大小。`System.IO` 中定义了处理输入/输出的 4 种模式（如表 11-6 所示），它们被定义为：

```
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

表 11-6 打开文件模式

模 式	说 明
ReadMode	只能读文件，文件必须存在，否则返回 <code>isDoesNotExistError</code> 异常
WriteMode	只能写文件，若文件已存在，内容会被清空，如果不存在会自动新建
AppendMode	只能写文件，若文件已存在，内容会加在末尾，如果不存在会自动新建
ReadWriteMode	可以读也可以写，若文件已存在，内容不变，如果不存在会自动新建

当不需要再对一个句柄操作时，就要关闭它，所以要用 `hClose :: Handle -> IO ()`，比如关闭一个不再操作的文件。

根据不同的需要，Haskell 已经定义好了对应的文件操作函数（如表 11-7 所示），我们不必再自己定义了。

表 11-7 打开文件函数

模 式	函 数
ReadMode	<code>readFile :: FilePath -> IO String</code>
WriteMode	<code>writeFile :: FilePath -> String -> IO ()</code>
AppendMode	<code>appendFile :: FilePath -> String -> IO ()</code>
ReadWriteMode	<code>openFile :: FilePath -> IOMode -> IO Handle</code>

使用 `ReadWriteMode` 时，就可以通过对文件句柄来自由地控制文件的读取与写入了。但有时可能得不到文件句柄，因为如果得到了第一个 `ReadMode` 的句柄后，Haskell 会将其加写保护锁，此时，对于同一个文件的 `ReadMode` 的句柄还是可以返回的，但无法返回其余另外三个模式的句柄了。同样，如果已经得到了文件写模式的句柄，即，除 `ReadMode` 以外的三个模式，再想返回一个句柄就会得到 `isAlreadyInUseError` 异常。

当给定一个句柄时，就可以使用 `hSeek :: Handle -> SeekMode -> Integer -> IO ()` 在文件中游走了，它需要一个移动模式和一个整数。移动的模式分为三种（如表 11-8 所示），定义如下：

```
data SeekMode = AbsoluteSeek | RelativeSeek | SeekFromEnd
```

表 11-8 在文件中移动模式

模 式	说 明
AbsoluteSeek	绝对移动，即从 0 开始，移动给定的字节数
RelativeSeek	相对移动，即从当前的位置移动给定的字节数，可以为负值
SeekFromEnd	从文件末尾向前移动给定的字节数

文件内容读取函数如表 11-9 所示。

表 11-9 文件内容读取函数

函 数	功 能
hGetChar :: Handle -> IO Char	返回一个字符，并且向后移动 1 字节
hGetLine :: Handle -> IO String	返回一行字符串，并且向后移动对应长度
hLookAhead :: Handle -> IO Char	保持当前位置不变，返回下一个字符，若当前位置已经在末尾则抛出 isEOFError 异常
hGetContent :: Handle -> IO Char	返回当前位置后的所有数据，并关闭句柄，若当前位置已经在末尾则抛出 isEOFError 异常

hTell 会返回一个句柄当前访问的绝对位置：hTell :: Handle -> IO Integer。

在操作的过程中可以使用 hIsEOF :: Handle -> IO Bool 来判定是不是已经读到了末尾处。之前使用过的 getChar 可以定义为 hGetChar stdin，同理，getLine 也是使用 hGetLine 定义的。

在从数据流（如从文件、网络端口）读取时数据时常常不会一次性都读过来，当然也不会一个字符一个字符的读，因为一次性读取可能要花很久，并且其间很难在读取的同时对数据进行处理。一个字符一个字符地读取也会降低读取速度，因为处理一个字符的时间要远比从磁盘、网卡中读取一个字符的时间短。同理，写入文件也是一样的。所以这些数据常常会被读到或者写入到一段内存单元中，一段一段处理，可称为缓冲区（buffer）。Haskell 的 IO 提供了 3 种不同的缓冲模式，如表 11-10 所示。如果打印字符串在命令行上但在运行时却不会有输出，那么很有可能是没有设置缓冲区的缘故。

```
data BufferMode = NoBuffer | LineBuffering | BlockBuffering (Maybe Int)
```

表 11-10 缓冲区模式

模 式	说 明
NoBuffer	无缓冲模式，字符将逐个被操作
LineBuffering	行缓冲模式，字符串以行为单位被操作
BlockBuffering (Maybe Int)	块缓冲模式，字符串被存储在指定字节长度的缓冲区中被操作，若为 Nothing，Haskell 将使用默认大小的缓冲区

可以使用函数 `hSetBuffering :: Handle -> BufferMode -> IO ()` 来改变缓冲的模式。当使用行缓冲模式或者块缓冲模式时，如果缓冲区中没有\n 或者缓冲区没有满，那么可能会出现字符串无法在命令行上输出，如果读者使用 OpenGL 的话，容器内的图形可能没有被画在窗口中。这时需要用 `hFlush :: Handle -> IO ()` 函数来强制将数据写给操作系统并清空缓冲区。另外，当用户使用 `hClose` 函数关闭一个句柄时，会自动调用 `hFlush` 函数。`hGetContents` 与 `readFile` 的实现使用了惰性缓冲区，所以可以读很大的文件进来，而不必特意担心内存分配等问题。虽然它使用起来十分方便，但是有时惰性 IO 可能会消耗大量的内存空间。

最后来了解一下文本输出函数，如表 11-11 所示，从它们的类型签名中就可以很确定这些函数的功能了，所以这里就不深入讨论了。

表 11-11 输出函数

函 数	功 能
<code>hPutChar :: Handle -> Char -> IO ()</code>	输出一个字符
<code>hPutStr :: Handle -> String -> IO ()</code>	输出一个字符串
<code>hPutStrLn :: Handle -> String -> ()</code>	输出一个字符串并在末尾添加一个换行符
<code>hPrint :: Show a => Handle a -> IO ()</code>	输出任意实现 <code>Show</code> 类型类的实例 a

`System.IO` 中还有很多有用的函数，例如 `hWaitForInput`、`hReady`，但是这里就不再一一列举了。读者可以参阅 API 自己动手来了解它们。

11.3.4 格式化输出 `printf` 函数

学过 C 语言的读者可以略过本节，因为本节只是简要地介绍 C 语言中 `printf` 函数的使用。`printf` 是 C 语言的库函数，它的使用范围非常广泛。究其原因，它简单和直观的接口功不可没。GHC 的库中其实提供了功能完整的与 C 语言功能相同的 `printf` 函数（格式输出控制符参见表 11-12）。

表 11-12 `printf` 格式输出控制符

格式输出符			说 明			
c	字符	O	八进制数	x X	十六进制数	s
D	十进制数	F	小数	e E	科学计数法小数	-
+	总是打印正负号	0	空位补 0	.	打印固定精度	

在输出字符串的过程中，常常需要对格式进行输出，比如日期 2012-10-1，如果在 Haskell 中生成这个字符串可能有些麻烦的，但借助于 `Text.Printf` 的 `printf` 函数可以很容易地生成它们。

```
>:m +Text.Printf
> printf "%d-%d-%d" 2012 10 1
2012-10-1
```

而之前，可能需要先使用 `show` 函数然后再将这些字符串用`++`相连再输出，这样显得笨拙麻烦。

```
import Text.Printf
main = do
    a <- getLine
    b <- getLine
    printf "Hello %s, I am %s.\n" a b

>main
Alexander
Ben
Hello Alexander, I am Ben.
```

其中，一个%与一个英文字符对应一个类型的参数：

```
> printf "%5d\n" 10
10
```

左侧会有 5 位的空格，如果想将空格补 0，则只需要改为`%05d`即可。如果需要左对齐，只要改为`%-5d`就可以了。

```
> printf "%.5f\n" pi
3.14159
```

`.5f` 意为只打印小数点后 5 位。`e` 是为了表达科学计算法而定义的格式。

```
> printf "%e\n" (2^10::Double)
1.024e3
```

用科学计数法表示时注意，这里的参数需要被指定为小数。

这里可以看到，`printf` 和之前了解的函数有很大不同。调用 `printf "%.5f\n" pi` 时，`printf` 函数的类型为 `printf :: String -> Double -> IO ()`，而调用 `printf "Hello %s, I am %s.\n" a b` 时，`printf` 函数的类型为 `printf :: String -> String -> String -> IO ()`。像这样的函数被称为参数数量可变函数。它的实现似乎是与 Haskell 的类型系统冲突了，而事实上，只要给出正确的设计在 Haskell 中是完全可以实现的。其实，使用 GHC 有很多种方法实现这样的函数，比如借助外部函数接口与 Template Haskell，但它们都不是安全使用 Haskell 的方式。下面，使用 Haskell 中的类型类来实现它，所以下面小节相当于是对第 9 章的一个延伸。

11.3.5 `printf` 函数的简易实现

首先，观察下面两个在 C 语言中的 `printf` 函数表达式：

```
printf("%s and %s are friends.", s1, s2);
printf("%s, %s and %s are friends.", s1, s2, s3);
```

应该可以看出以下两点内容。

(1) `printf` 使用格式控制字符串，将其余参数按格式插进字符串中，然后再显示。

(2) `printf` 可以接受紧跟在格式控制字符串之后任意类型和数量的参数。

第一个特性，用 Haskell 来实现没有任何问题。然而对于第二个特性，如果想在 Haskell 中实现，似乎类型系统会和用户作对。正如上一节所讨论过的那样，单独拿第一个例子来考虑，`printf` 的类型应该是：

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

但在第二个例子，`printf` 的类型似乎又应该是：

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

这样，它们看起来是冲突的。应该如何为 `printf` 函数确定类型？怎样才能写出 `printf` 函数？回想第9章中对类型类的介绍：当把不同的类型上的不同操作用相同的函数来表示时，会用一个类型类来归并这些类型。在这两个例子里：

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

下划线部分不一样，试着把它抽象成一个类型类：

```
class Printf t where  
    printf :: String -> t
```

当 `printf` 除了格式控制字符串外没有任何额外参数时，`t` 应该就是 `IO ()`：

```
instance Printf (IO ()) where  
    printf cs = putStrLn cs
```

但是，GHCi 会报告如下错误：

```
Illegal instance declaration for `Printf (IO ())'  
(All instance types must be of the form (T a1 ... an)  
where a1 ... an are *distinct type variables*,  
and each type variable appears at most once in the instance head.  
Use {-# LANGUAGE FlexibleInstances #-} if you want to disable this.)  
In the instance declaration for `Printf (IO ())'
```

出现该提示的原因是，Haskell 对 type class 实例加了相当严格的限制，它只能接受 (`IO a`) 这样的类型实例，其中，`a` 是类型参数。而类型实例是 (`IO ()`)，其中，`()` 是类型常量。随着 Haskell 的发展，这类严格的限制可以被解除。这里，可以在程序代码的顶部加上：`{-# LANGUAGE FlexibleInstances #-}` 来解除这个限制。接下来，讨论如何支持可变长度的参数。首先，实现将参数插入格式控制字符串里的功能：

```
format :: Show t => String -> t -> String  
format ('%' : 's' : cs) cs' = show cs' ++ cs  
format (c : cs) cs' = c : format cs cs'
```

```

format "" cs' = ""

> putStrLn $ format "We formatted an integer: %s" 123
We formatted an integer: 123

> putStrLn $ format "We formatted a string: %s" "abc"
We formatted a string: "abc"

```

对于一个额外参数，`Printf` 应该定义成：

```
instance Show t => Printf (t -> IO ()) where
    printf cs x = putStrLn (format cs x)
```

两个额外参数如下：

```
instance (Show u, Show t) => Printf (u -> t -> IO ()) where
    printf cs y x = putStrLn (format (format cs y) x)

...
```

显然不能一直这样写下去，因为 `printf` 原则上可以接受任意数量的参数。然而，可以从这些实例里找到共性。在一个额外参数的情况下，其定义可以这样重写：

```
printf cs x = putStrLn (format cs x)

= (写为 λ 表达式形式)
  printf cs = \x -> putStrLn (format cs x)

= (这里认为 putStrLn 与 printf 等价)
  printf cs = \x -> printf (format cs x)      (1)
```

两个额外参数的情况下，定义也可以进行下面的化简：

```
printf cs y x = putStrLn (format (format cs y) x)

= (写为 λ 表达式形式)
  printf cs = \y -> \x -> putStrLn (format (format cs y) x)

= (-> 为右结合的并且 putStrLn 与 printf 是等价的)
  printf cs = \y -> (\x -> printf (format (format cs y) x))
```

= (对划线部分的 λ 表达式做 n 化简)

```
printf cs = \y -> printf (format cs y)
```

最后一步之所以成立，是因为第一个式子告诉我们，对任意 `cs` 和 `x`，`printf cs` 和 `\x -> printf (format cs x)` 都是可以互换的。从这两个实例能够发现，对于任意数量大于 1 的额外参数，`printf` 都能通过递归写成下面的定义：

```
printf cs = \x -> printf (format cs x)
```

而每次递归时，传递给 `printf` 的参数数量就会减少一个。因此，将所有参数数量大于 1

的情况归结成一种，实现 `Printf` 类型类的实例定义如下：

```
instance (Show u, Printf t) => Printf (u -> t) where
    printf cs = `x -> printf (format cs x)
```

回到本节开头 C 语言的例子，测试一下 Haskell 版本的 `printf`：

```
test1 :: IO ()
test1 = printf "%s and %s are friends." "Mike" "Jane"

test2 :: IO ()
test2 = printf "%s, %s and %s are friends." "Mike" "Jane" "Chris"

> test1
"Mike" and "Jane" are friends.

> test2
"Mike", "Jane" and "Chris" are friends.
```

事实上，`printf` 比 C 语言里的使用起来更方便，例如：

```
x :: Int
x = 3
c :: Char
c = 'a'

test3 :: IO ()
test3 = printf "The pair of %s and %s is %s." x c (x,c)

> test3
The pair of 3 and 'a' is (3,'a').
```

在这个例子中，无论是对于整数 `x`，字符 `c`，还是二元组 `(x, c)`，都用 `%s` 来标记，而不像 C 语言那样需要用不同的标记符号来控制输出方式。其实这点很容易理解，它只是以前介绍过的 `show` 函数的威力而已，C 语言中没有类型类与重载函数，所以没法做到这一点。

练习

- 在 4.1.2 中，我们定义了一个历法的函数，通过计算可以得出任意一天是星期几。读者可以写一个万年历程序，使它从命令行中读取年份与月份，可以使用 `read` 函数将字符串解析为整数，然后输出当月的月历。比如：

```
C:\>runghc Calendar.hs 2012 10
          October 2012
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
    7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

了解了 printf 这种格式化输出函数，解决这类问题就很容易了。

2. printf 和 C 语言的 printf 是有些区别的。比如：
- ```
> printf "%s and %s are friends." "Mike" "Jane"
"Mike" and "Jane" are friends.
```

```
printf("%s and %s are friends.", "Mike", "Jane");
Mike and Jane are friends.
```

你能修改我们的 printf 定义，让其在输出字符串时的行为与 C 版本的 printf 一致吗？

3. 在本节里定义的 printf 统一使用 %s 来输出任意类型参数，依靠 show 来确定如何显示，这比 C 的 printf 更容易使用。然而，C 语言的 printf 也有它独到的好处，就是可以在参数类型相同时，也能改变它的输出方式。例如：printf("%d", x); 与 printf("%x", x);。

对同样的整数 x 可能会有不同输出。读者可以修改我们的 printf 定义，使它支持该特性吗？

## 11.4 星际译王词典

本节以一个词典程序为例，主要介绍如何在 Haskell 使用 IO。首先，先介绍一下星际译王词典。星际译王词典最初版由马苏安开发，后来被胡正加以用户界面重新开发，并使用 GPL 授权。程序本身并不包括字典文件，使用词典的人可以根据需要自己下载特定的词典文件。由于词典文件与程序是分开的，所以星际王词典有着非常好的可移植性，可任意在很多移动设备上使用。经过多年的发展，星际王词典也不仅仅局限于英汉、汉英词典，也有很多西语到西语的词典。有兴趣的读者可以登录 <http://www.stardict.cn/> 了解更多内容。

星际译王词典文件主要包括三个类型的文件，它们的扩展名分别是 .ifo 、 .idx 、 .dict 。有的星际王词典的扩展名为 .idx.dz 、 .dict.dz ，这是经过 dictzip 压缩的，读者可以通过 dictzip 或者 gzip 解压。但是，如果正确使用压缩词典可以提高空间使用效率。这里仅仅使用解压后的文件。下面简单介绍一下这 3 个文件。

- .ifo 文件携带的是词典的基本信息。
- .idx 是单词的索引文件，存放了单词在 .dict 文件中的位置以及释意的长度。
- .dict 则是存储单词释意的文件。比如，以下载的朗文当代英语词典的 .ifo 文件为例：

---

```
StarDict's dict ifo file
version=2.4.2
wordcount=43052
idxfilesize=771616
bookname=Longman Dictionary of Contemporary English
description=Made by Hu Zheng
date=2005.10.23
sametypesequence=m
```

---

文件中包括了以下信息：版本、单词数、索引文件大小、字典名称、描述等信息，.idx 文件中单词的索引一般是这样存储的：

```
<单词>\0<单词在.dict 文件位置中偏移><单词在.dict 文件中释意长度>
```

文件中每个单词以 ‘\0’ 作为结尾，后跟 8 个字节，前 4 个字节组成的 32 位整数是单词在.dict 文件中的位置，也就是偏移量，偏移单位为字节，后 4 个字节组成的 32 位整数是此单词在字典中释意的长度，单位也为字节。所有的单词按字母表顺序排列而不是 ASCII 码表的顺序。若找到单词，用索引文件中的偏移量去找到释意的位置，然后取出索引文件中存储的长度即可。

.dict 文件中的内容使用了 UTF-8 字符编码方式来保存单词的释意。

#### 11.4.1 二分法查找

下面，先来写索引文件的搜索。

---

```
--WordIndex.hs
module WordIndex where
import Data.Char (toLower)
```

---

这里使用 toLower 函数是因为单词在索引文件中是按字母表顺序排列的，而不是按 ASCII 表排列的，比如词典中最后一个词与 about 这个词，即在 Haskell 中比较：

---

```
> "Zulu" < "about"
True
```

---

而显然，在词典中，Zulu 在 about 的后面，所以需要在比较两个单词前，把索引文件改成小写。

---

```
data WordIdx = WordIndex {
 word :: String,
 offset :: Int,
 expLen :: Int
} deriving (Show)
```

---

WordIdx 中的三个值很容易理解：word 是单词、offset 是偏移量、expLen 是释意长度。这就是需要存储索引的数据类型。下面来定义二分法查找算法：

---

```
searchWord :: String -> [WordIdx] -> Maybe WordIdx
searchWord str [] = Nothing
searchWord str xs | wrd < str = searchWord str behind
 | wrd > str = searchWord str front
 | otherwise = Just b
 where (front,b:behind) = splitAt (length xs `div` 2) xs
 wrd = map toLower (word b)
```

---

以上就是二分法查找了，也是字典的核心部分，由于在前面的章节中讲过二分法查找，在此不过是类型的类型与返回结果有些变化，所以在这里就不再赘述了。

下面来解决将词典索引文件字符串解析成 WordIdx 数据类型、词典搜索、输入/输出等问题。

---

```
--Dict.hs

module Main where

import Data.Char (ord)
import System.Environment (getArgs)
import System.IO
import qualified Data.ByteString.Char8 as DB
import Text.Printf
import WordIndex
```

---

getIndexList 函数将 idx 文件中的全部字符串解析成一个 WordIdx 的列表作为输出。这个函数需要一个可以将 String 转换成整数的函数 byteInt，即，将那些 byte 组成的 String 计算机成 WordIdx 列表。

---

```
getIndexList :: String -> [WordIdx]
getIndexList [] = []
getIndexList str = WordIndex w (byteInt o) (byteInt e) : getIndexList left
 where
 w = takeWhile (/='\0') str
 off = dropWhile (/='\0') str
 o = take 4 (drop 1 off)
 e = take 4 (drop 5 off)
 left = drop 9 off
```

---

显然，w 为读到的单词，off 为取出单词余下的部分，然后需要处理偏移量与释意长度。o 则为单词释意在 dict 文件中的偏移，e 为释意的长度，只需要将 o，e 转换成 Int 即可。left 为余下未处理的字符串，只需要递归地调用 getIndexList 直到字符串为空为止，这样就会得到所有 idx 文件中的单词索引信息。byteInt 函数用来将读到的 4 字节字符串转换成整数。下面来看一下 byteInt 函数。

给定一个以字节为单位的整数的列表，需要将其移位然后组合来得到 32 位的位表示的整数。

---

```
shift :: [Int] -> Int
shift xs = sum [i * 2^sh | (i, sh) <- zip xs (iterate (+8) 0)]
```

---

shift 函数将多个字节的整数值移位并相加，得到整数值。这里，列表的第一个元素为低位。计算前，将列表倒置输入给这个函数即可。

---

```
>shift [1,2,0,0]
513
```

---

这里的 [1,2,0,0] 表示二进制数 00000000 00000000 00000010 00000001 的大小。

---

```
byteInt :: String -> Int
byteInt xs = shift int
 where int = reverse $ map ord xs
```

---

主函数的定义也非常简单，先从命令行参数中得到要查找的单词，再读索引文件，如果有

这个单词，那么打开释意文件，取得单词的解释：

---

```
main :: IO ()
main = do
 arg <- getArgs
 case arg of
 [] -> print "Usage: Dict <word>"
 (a:_) -> do
 idctIdx <- readFile "./Dict/longman.idx"
 let is = getIndexList idctIdx
 let result = searchWord a is
 case result of
 Nothing -> printf "Word \'%s\' not found" a
 Just wrd -> do
 inh <- openFile "./Dict/longman.dict" ReadMode
 hSeek inh RelativeSeek (toInteger $ offset wrd)
 hSetEncoding inh utf8
 explanation <- DB.hGet inh (expLen wrd)
 hClose inh
 DB.putStrLn explanation
```

---

整个字典的搜索程序仅有 60 行左右的代码，而且，约 1/3 部分被处理输入/输出的 main 函数所占据。从上述的代码可以看到，在写 Haskell 程序时不需要考虑内存的分配与回收，也不需要考虑列表的长度等细节问题。二分法查找也可以很容易并且非常严格地被证明是正确的，不存在 C、Java 中可能出现数组访问越界等问题。只要考虑如何搜索，如何将.idx 文件读取成所需要的 WordIdx 的结构，如何用搜索到的结果来访问词典释意文件就可以了。这里，有些读者应该可以感觉到，用 Haskell 编程，其生产效率是非常高的，并且程序的准确程度也非常好。读者可能会想，这里我们的字典每一次只查询一个单词，没有必要使用二分法查找，因为我们在构建索引列表时已经遍历过全部的单词，完全可以在遍历时进行查找。这种想法是没有错的，但本节主要是为了展示有副作用的函数与纯函数是如何一起使用的，并且我们会在后面的练习中修改代码，使它可以让用户一直地输入单词，不断地查找。这节只是使用二分法查找作为例子。

在命令行中输入 ghc --make Dict 即可将写的字典编译成可执行文件。只要输入 Dict.exe 和想查询的单词即可。

---

```
> :set args programmer
> main
n [C] someone whose job is to write computer programs//
```

---

```
> :set args algorithm
> main
n [C] [Date: 1600-1700; Origin: algorism 'algorithm' (13-19 centuries), from
Medieval Latin algorismus, from Arabic al-khuwarizmi, from al-Khuwarizmi
9th-century Arab mathematician; influenced by Greek arithmos 'number']//
technical a set of instructions that are followed in a fixed order and used for solving
a mathematical problem, making a computer program etc//
```

---

读者可能会发现，程序运行可能会有些慢，因为 `String` 是由列表的字符组成的，它的效率很低。当然，GHC 提供 `ByteString` 可以提高对字符串的处理效率。

这里可以看到，处理输入/输出的部分全部在 `main` 函数中，没有在其他的函数中使用 `IO` 类型，那些函数是纯函数。`IO` 的操作是 Haskell 语言和外界联系的过程，这个过程会产生副作用，系统可能出现字典文件不存在或者因其他异常而无法读取的问题。但是，这些情况全然不在 `searchWord` 和 `getIndexList` 这些纯的函数中，它们中绝对不会出现与程序或者系统状态有关的异常。也正因 Haskell 将纯与不纯的代码分离开来，如果系统出现异常，那么一定是在 `main` 函数中，也就是处理带有输入/输出副作用的函数里。Haskell 中将那些纯的函数与带有副作用的不纯函数分开，从而避免了很多顺序式编程语言中可能的错误。

#### 11.4.2 散列表的使用

除了使用列表结构进其中的单词使用二分法快速搜索外，还有可以用其他的方式来搜索，`Prelude` 就提供 `lookup :: Eq a => a -> [(a, b)] -> Maybe b` 给定元素 `a` 与存有 `(a, b)` 类型的列表，从左至右地查找 `b`，但是这样并不高效。还可以使用另一种容器来对单词与其所在的位置与释意长度进行快速的索引，它就是散列表（hash table）。它在 `Data.HashTable` 中，其计算的主要方法是定义一个散列函数 `f :: String -> Int32`，这样，使用对于一个单词都有一个 `Int32` 与之对应，这个函数会尽可能地让不同的字符串对应不同的数字而达到可以快速搜索的目的。需要使用下面的库：

---

```
import Prelude hiding (lookup)
```

---

散列表中的 `lookup` 与 `Prelude` 中的 `lookup` 冲突，所以要将 `Prelude` 中的 `lookup` 隐藏。

---

```
import Data.Char
import Data.HashTable
import System.Environment
import Text.Printf
import System.IO
import qualified Data.ByteString.Char8 as DB
```

---

下面，定义一些类型，每个单词的所在位置 `Offset` 与释意长度 `ExpLen`。

---

```
type Offset = Int
type ExpLen = Int
type Idx = (Offset, ExpLen)
```

---

散列表则需要两个类型，一个是索引类型 `key`，另一个是查找的类型 `value`。这里，通过单词 `String` 来查找 `(Offset, ExpLen)`，将这种类型命名为 `Idx`。

---

```
type WordTable = HashTable String Idx
```

---

向散列表中插入单词是要改变内存的。倘若不改变内存，每插入一个新的单词就生成一个新的散列表，无论从时间还是空间，效率是非常低的。所以，对散列表操作这种改变内存状态

的副作用要在 IO Monad 中进行。库中提供了新建散列表的函数。

---

```
new :: (key -> key -> Bool) -> (key -> Int32) -> IO (HashTable key val)
newHint :: (key -> key -> Bool) -> (key -> Int32) -> Int -> IO (HashTable key val)
```

---

new 函数的第一个参数是索引比较函数，即给定两个索引值如何来比较它们是否相等。对于字符串来说，这个函数是 (==)，第二个参数是散列函数，这里可以使用库中提供的 hashString。newHint 的前两个参数是相同的，它的第 3 参数可以指定散列表的容量。

还有 insert、delete 与 lookup 函数，相信不需要写出它们的类型读者也能自己推断出来。下面定义一个函数，把 idx 文件中读入的字符串解析并存入到散列表中。

---

```
getIndexTable :: String -> WordTable -> IO ()
getIndexTable "" _ = return ()
getIndexTable str table = do
 insert table w (byteToInt o, byteToInt e)
 getIndexTable left table
 where
 w = takeWhile (/='\'0') str
 off = dropWhile (/='\'0') str
 o = take 4 (drop 1 off)
 e = take 4 (drop 5 off)
 left = drop 9 off
```

---

shift 与 byteToInt 函数与之前的二分法相同，这里就不再解释了。

---

```
shift :: [Int] -> Int
shift xs = sum [i * 2^sh | (i, sh) <- zip xs (iterate (+8) 0)]

byteToInt :: String -> Int
byteToInt xs = shift int
 where int = reverse $ map ord xs
```

---

下面就是 main 函数，与之前的二分法查找几乎完全相同，只是在查找时使用了 lookup 函数。

---

```
main :: IO()
main = do
 arg <- getArgs
 case arg of
 [] -> print "Usage: Dict <word>"
 (a: _) -> do
 idctIdx <- readFile "./Dict/longman.idx"
 wordTable <- new (==) hashString
 getIndexTable idctIdx wordTable
 result <- lookup wordTable a
 case result of
 Nothing -> printf "Word \"%s\" not found" a
 Just (offset, expLen) -> do
 inh <- openFile "./Dict/longman.dict" ReadMode
 hSeek inh RelativeSeek (toInteger offset)
```

---

```

hSetEncoding inh utf8
explanation <- DB.hGet inh expLen
hClose inh
DB.putStrLn explanation

```

### 练习

- 现在，词典每查一个单词就需要启动一次，通过使用 Control.Monad 中的函数，可以写一个互动的词典以便用户可以一次启动词典查询多个单词，使用 :quit 退出词典程序。例如：  

```

C:\>Dict.exe
Welcome to Star Dictionary, use :quit to exit
please input you word: programmer
n [C] someone whose job is to write computer programs//
please input you word: algorithm
...
please input you word: :quit
C:\>

```
- 使用以 [Char] 定义的字符串类型，在很多情况下是低效的，因为它相当于一个单向的链表，很多情况下，操作它的效率是不如 C 语言中的数组的，但是 Haskell 在 Data.ByteString 中提供了字节字符串 (ByteString) 类型与操作它的相关函数。其中的 Data.ByteString.Lazy 通常是比较好的：首先，它使用的大小将会很好地适应 CPU 一级高速缓存，GHC 的垃圾回收器也会对不用的部分尽快回收；其次，CPU 中读取内存的过程中，如果需要某一内存单元中的值，它只是将它读入高效缓存，而且还会对它周围的内存块进行预读来降低高速缓存错失 (cache miss) 的概率，以提高内存访问效率。然而，单向链表在内存的存储可能是十分分散的，这样，CPU 预读带来的高效可能无法体现出来。这里，请读者浏览一下 ByteString 的相关函数，将字典程序改为使用 ByteString 的版本。

## 11.5 简易异常处理

之前讨论了很多内容，其中，很多都涉及了程序可能产生的异常。如果不处理这些异常，最后就会成为程序中的错误。通常情况下，异常是那些在程序运行中不能预见的问题，比如读取文件时磁盘损坏、权限不够等问题，它们通常是与外界交流引起的。由于纯函数计算引起的错误必须预见得到，并且使用 Maybe、Either 或者其他异常类型来处理。本节简单讨论如何在 Haskell 中处理异常。

在 Java 等面向对象的语言中，异常被当做是一个对象，也就是 Exception class，其余各种异常可以通过继承 Exception class 或者其子类来实现。这样，就有一个非常明确的等级结构，然后可以使用 try 与 catch 关键字来处理异常。如果 try 内出现了异常，那么对应的 catch 部分中的代码会被运行。这样，catch 关键字需要判断由 try 抛出的异常是否是该异常类型或者是该异常类型的子类。比如：

---

```
try {...}
catch (MyException e) {...}
catch (Exception e) {...}
```

---

如果 try 的部分有异常出现，那么 Java 会根据不同的异常进行匹配。倘若抛出的异常不是 MyException 或者它的子类，那么就会匹配到最后一行的 Exception e。而有着静态类型的 Haskell 却不容易实现类似的机制，因为这里第一个 catch 的类型是 MyException  $\rightarrow$  IO a，而第二个则是 Exception  $\rightarrow$  IO a，显然这是不可能的。

首先，异常处理需要将各种异常归类，然后分级，这样使得每一个 catch 都可以处理属于它的异常。其次，可以在这个等级森严的异常系统中自定义一些异常，以满足用户自己的需要。最后，异常可以通过使用同样的函数处理，不至于用户为每个异常都要定义不同的 catch 函数。

库中定义了最顶级的异常类型定义如下：

---

```
data SomeException = forall e . Exception e => SomeException e
deriving Typeable
```

---

也就是说，它是一个容器类型，内盛一种异常。这里的 Typeable 是一种将类型具体化的方法。实现 Typeable 类型类后，就可以得到一个代表数据类型的类型，还可以对类型进行比较等。它的定义如下：

---

```
class Typeable a where
 typeOf :: a -> TypeRep
```

---

typeOf 函数取得任意一个类型 a，然后得到可以代表其他类型的一个类型，称为 TypeRep (Type Representation)。在计算这个函数时，typeOf 的参数会被忽略，因为被需要的只是其类型，所以只要声明了类型，即使参数为 undefined 也会得出结果。

---

```
> :m +Data.Typeable
> typeOf True

Bool
> typeOf 5
Integer
> typeOf (undefined:: Bool)
Bool
```

---

而库中的 cast 函数可以用来判定一个类型是否与给定的类型相等，例如：

---

```
> (cast 5) :: Maybe Integer
Just 5
> (cast 5) :: Maybe Double
Nothing
```

---

这里可以看到，GHC 默认 5 为任意精度整数。如果使用 Typeable，那么类型是要限定类型为单一形态的，它不能支持多态与重载，虽然不支持这些类型，但是对于简单的异常处理已

经足够了。需要 `DeriveDataTypeable` 编译器参数来使用 `deriving` 关键字让类型自动生成 `Typeable` 类型类所用的函数。

部分异常的相关的类型类与函数被定义在了 `Control.Exception` 中，其中 `Exception` 类型类里有两个函数，即 `toException` 与 `fromException`。这两个函数可以在调用 `throw` 函数抛出异常与用 `catch` 处理异常时对进行类型的转换，读完后面的内容，读者就会明白为什么要这样设计。

---

```
class (Typeable e, Show e) => Exception e where
 toException :: e -> SomeException
 fromException :: SomeException -> Maybe e

 toException = SomeException
 fromException (SomeException e) = cast e
```

---

可以看出，这个类型类默认定义了所有声明的函数，所以一个类型在实例化 `Exception` 时，如果不定义 `toException` 与 `fromException` 函数，那么这个异常类型将直接属于类型 `SomeException`。例如：

---

```
instance Exception SomeException where
 toException se = se
 fromException = Just
```

---

现在，来实现一系列等级结构分明的异常，首先是 `SomeA`，它隶属于 `SomeException`，然后 `SomeA` 下又有另外两个子异常，分别为 `SomeSubA1` 与 `SomeSubA2`。最后定义 `SomeSubA1` 与 `SomeSubA2` 的异常类型实例，即 `subA1` 与 `subA2`。异常类型的容器前加 `Some` 的话，则 `SomeXXXException`，一般是用来表示它只是异常类型等级树中的节点，而不以 `Some` 开头的异常习惯上命名为 `XXXException`，它们为异常类型等级树中的叶子。

---

```
({-# LANGUAGE
 ExistentialQuantification, DeriveDataTypeable, ScopedTypeVariables,
 NoMonomorphism Restriction #-})
```

---

`ScopedTypeVariables` 编译器参数可以让我们在定义 `λ` 表达式时指定输入的类型。同时需要隐藏预加载库中的 `catch` 函数，并且需导入要另外两个库：

---

```
import Prelude hiding (catch)
import Control.Exception
import Data.Typeable
```

---

首先定义 `SomeA`，它是一个用来盛装异常的容器类型，并且它与 `SomeException` 的定义是同构的。

---

```
data SomeA = forall e. Exception e => SomeA e deriving Typeable

instance Show SomeA where
 show (SomeA e) = show e
```

---

由于它直接隶属于 SomeException，所在实现 Exception 类型类时什么也不用写：

---

```
instance Exception SomeA
```

---

下面定义 SomeSubA1 与 SomeSubA2。它们也都是异常结构中的一级，所以也定义为容器类型。

---

```
data SomeSubA1 = forall e. Exception e => SomeSubA1 e deriving Typeable

instance Show SomeSubA1 where
 show (SomeSubA1 e) = "SomeSubA1, " ++ show e
```

---

将 SomeSubA1 实现为 Exception 类型类的实例。在定义 toException 时，要先将它放到 SomeA 中，然后再对 SomeA 调用 toException 函数，最后就会得到 SomeException (SomeSubA1 ...) 这样一个值。而 fromException 所做的事情恰恰相反，它做的事情是将异常从一层层的包装中取出，但是会使用 cast 的返回的值。

---

```
instance Exception SomeSubA1 where
 toException = toException.SomeA
 fromException x = do
 SomeA e <- fromException x
 cast e
```

---

SomeASub2 的定义与 SomeSubA1 的定义是一样的：

---

```
data SomeSubA2 = forall e. Exception e => SomeSubA2 e deriving Typeable

instance Show SomeSubA2 where
 show (SomeSubA2 e) = "SomeSubA2, " ++ show e

instance Exception SomeSubA2 where
 toException = toException.SomeA
 fromException x = do
 SomeA e <- fromException x
 cast e
```

---

最后，定义 SubA1 与 SubA2 下的异常，它们在异常这棵等级树中是叶子：

---

```
data SubA1 = SubA11 | SubA12 deriving (Typeable, Show)

instance Exception SubA1 where
 toException = toException.SomeSubA1
 fromException x = do
 SomeSubA1 e <- fromException x
 cast e

data SubA2 = SubA21 | SubA22 deriving (Typeable, Show)

instance Exception SubA2 where
 toException = toException.SomeSubA2
 fromException x = do
```

---

---

```
SomeSubA2 e <- fromException x
cast e
```

---

下面可以测试一下定义的异常。由于 Haskell 为函数式语言，它的异常处理与其他语言就略有不同，try 与 catch 的意思与其他语言是不一样的。库中主要提供了 catch 函数来对异常进行处理，它的类型是 catch :: Exception e => IO a -> (e -> IO a) -> IO a。第一个参数是 IO 操作，第二个参数是如果出现了异常应该如何处理。

---

```
> throw (SubAll) `catch` (\(e::SomeSubA1) -> putStrLn ("Caught " ++ show e))
Caught SubAll

> throw (SubAll) `catch` (\(e::SomeException) -> putStrLn ("Caught " ++ show e))
Caught SubAll

> throw (Overflow) `catch` (\(e::SomeSubA1) -> putStrLn ("Caught " ++ show e))
*** Exception: arithmetic overflow
```

---

因为 SubAll 是属于 SomeSubA1 与 SomeException 的，所以异常都会被捕捉到。

这样，通过 throw 一层一层地加了包装，然后 catch 时再一层一层地去包装，就有了这种将异常分级处理的方法。虽然这不是效率很高的方法，但是异常处理的效率往往不会作为首要条件考虑，而且在现实编程中，异常的数量也不会多到影响程序的效率的地步。

如果想处理多种异常，那么可以使用 catches 函数，它的类型是 catches :: IO a -> [Handler a] -> IO a，其中，Handler 是处理某一异常的函数，它是这样定义的：

---

```
data Handler a = forall e . Exception e => Handler (e -> IO a)
```

---

也就是将 catch 的第二个参数加了一层构造器。这样，使用 catches 函数来处理多种类型的异常时可以像下面这样来定义函数：

---

```
test :: Exception e => e -> IO ()
test = \e -> (throw e) `catches`
 [Handler \$ \(e::SubA2) -> putStrLn ("caught SubA1 " ++ show e),
 Handler \$ \(e:: SomeSubA1) -> putStrLn ("caught SomeA2 " ++ show e)]
> test SubAll
caught SomeA2 SomeSubA1, SubAll

> test SubA22
caught SubA1 SubA22

> test Overflow
*** Exception: arithmetic overflow
```

---

可以看到，相应的异常被正确地处理了。如果发生的异常不能匹配列表中所有 Handler 的 λ 表达式的输入的类型，那么 GHC 会抛出这个异常。

try 函数是通过 catch 函数定义的，如果第一个参数没有返回异常，那么得一个 Right v；

否则返回 `Left e`, 其中, `e` 是异常。这样, 可以根据异常做出一些反应。

---

```
try :: Exception e -> IO a -> IO (Either e a)
try a = catch (a >>= \ v -> return (Right v)) (\e -> return (Left e))
```

---

如果没有足够的类型信息, 在使用 `try` 函数时, 需要指定这里异常的类型。通常, 该函数被指定为库中的 `IOErrorType` 类型, 其中包括了各种在处理输入与输出时所能异常情况。有时, 也仅仅用 `SomeException` 暗指得到所有的异常。

除 `try` 与 `catch` 外, 库中还提供了很多其他函数处理异常, 比较常用的有 `finally::: IO a -> IO b -> IO a`, 它的作用是执行完第一个 IO 操作后执行第二个 IO 操作。即便是第一个 IO 操作有异常抛出。还有 `onException :: IO a -> IO b -> IO a`, 它与 `finally` 类似, 但是, 只有当第一个 IO 操作中有异常时, 才会执行第二个 IO 操作。这两个函数常用于在获得某一硬件资源后, 在运行时发生异常需要释放, 如果不这样做, 那么其他的线程或者程序无法再继续使用它了。

关于异常的处理就先简单介绍到这里, 想进一步了解的读者可以参阅 `Control.Exception` 的 API。本节的内容主要是来自 Simon Marlow 写的“An Extensible Dynamically-Typed Hierarchy of Exceptions”, 可以访问地址: <http://community.haskell.org/~simonmar/papers/ext-exceptions.pdf> 来在线阅读。

#### 练习

试着完善上节中的词典程序, 处理输入/输出所引发的异常。比如, 如果文件不存在, 那么要给出一些信息, 告知用户需要将词典文件放到词典需要的文件夹下。

## 11.6 Haskell 中的时间

GHC 的库中提供了非常多的系统库函数来对日期、时间、文件还有目录操作的系统函数, 熟练地使用它们对平时的编程工作非常有帮助, 这里简单地介绍一下。

首先, 简单地介绍一下 GHC 中有关日期、时间的库函数 `System.Time` 与 `Data.Time`。因为系统的时间与系统的当前状态有关, 所以也需要在 `IO Monad` 中处理。其中, GHC 不推荐使用 `System.Time`, 应使用 `Data.Time`。

时钟时间 `ClockTime` 类型是这样定义的:

---

```
data ClockTime = TOD Integer Integer
```

---

第一个参数是世界协调时间距离 1970 年 1 月 1 日 0 时 0 分 0 秒的秒数, 第二个参数则是当前时间余下的皮秒 (picosecond) 数, 1 皮秒为  $10^{-12}$  秒。

---

```
> :m +System.Time
> time <- getClockTime
> time
Thu Dec 24 12:10:18 GMT Standard Time 2012
```

---

上面的打印方式非常便于人的阅读，若希望操作构造器中的秒与皮秒数，只需要将 TOD 构造器匹配即可。

```
> (TOD sec pico) <- getClockTime
> sec
1359031414
```

库中提供了将时钟时间作差与作和的函数：diffClockTimes, addToClockTime。可以得到可读的时间是因为库中定义了日历时间 CalendarTime，它更加符合日常读时间的习惯。构造器的参数很多，读者可以查阅 API 来了解。日历时间与时钟时间是可以用 toCalendarTime 与 toClockTime 函数相互转换的。

GHC 现在推荐使用 Data.Time 来处理时间，这个库里面有 4 个模块。第一个是 Calendar，可以处理使用儒略日（Julian Day）以日来记录的时间，其中有现在常用的格里高利历法（Gregorian Calendar），并有它们之前的转换函数。第二个是 Clock，也就是时钟时间，其一是世界时（Universal Time, UT），它基于地球的运转，世界时又分为 3 个系统 UT0、UT1 与 UT2，这里使用的是 UT1。第二种时间就是 UTC 了，比如北京时间使用的就是 UTC 时间加 8 小时，记为 UTC+8，夏威夷的时间要比 UTC 少 10 小时，记为 UTC-10，这个时间是由原子钟记录的，它不会考虑地球的运转，所以在发生地震或者其他因素影响了地球的运转时，这个时间可能需要被调整。第三个库中的相关函数为了操作是本地时间，提供了得到当前时区的时间以及计算时区的函数。Format 库中提供了类似于 printf 的函数，可以用来将时间以不同的格式打印，此外，还有解析时间的函数 parseTime 等。

Haskell 库中也提供了 CPU 时间的库，即 System.CPUTime，其中的 getCPUTime 可以计算当前程序运行所用的皮秒数。

时间是一个非常抽象的概念，有着不同的标准。因此，编写一个准确无误的日历程序还是非常具有难度的，只有程序员对历法与时间十分了解，才能写出一个日历跟日程提醒的程序。苹果与谷歌公司的移动设备上的日历或者时间全都出现过不同的问题，这就是由于编写日历的人对于历法理解得不好的缘故。另外，在大多数语言中，与系统通过有符号的 32 整数位来记录距 1970 年 1 月 1 日子夜的秒数来记录时间，也将在 2038 年 1 月 19 日 3 时 14 分 7 秒时出现问题，这时的二进制时间是 0 后跟 31 个 1，下一秒则为 -2147483648，这样表示的年数就可能会小于 1970 年。而 Haskell 中使用的是任意精度整数等类型来计算时间，虽然效率可能是不高的，但是一定程度上却更有远见。

## 本章小结

本节主要了解了程序的副作用，学习了 Haskell 中的输入与输出，相信读者已经明白为什么 Haskell 中要将 IO 这设计成为 Monad，也明白了为什么到这章才学习输入与输出的原因。通过一个实战的程序——星际译王词典，读者应该可以熟悉如何在 Haskell 中使用 IO。此外，还简单介绍了异常处理，算是对于第 9 章一些知识的总结。在最后的一节中，简单讨论了如何在 Haskell 中使用时间相关的库，相信这会在以后的编程中派上用场。下章将介绍更多 Monad 相关的内容。

## 第 12 章

# 记录器 Monad、读取器 Monad、状态 Monad

第 10 章初步介绍了 Monad，第 11 章介绍 io Monad，本章我们来学习一下 Haskell 中定义的更多 Monad。它们有着不同的作用，分别是记录器 Monad、读取器 Monad 还有状态 Monad。了解了状态 Monad 就可以学习在 Haskell 中如何生成伪随机数，而了解了 io Monad 就会明白 Haskell 是如何通过系统当前时间来生成随机数的。

### 12.1 记录器 Monad

在计算时，可能常常需要对一个计算的过程做些记录。比如，给定一个整数数轴，轴上有一个指针，可以向左移动，也可以向右移动，这两个移动的函数可以定义为 left 和 right，多次使用这两个函数只会得到一个最终的结果，但是不会记录移动的过程。如果想记录这个过程，则需要使用一个二元元组。

```
left,right :: Int -> (Int , String)
left x = (x-1, "move left\n")
right x = (x+1, "move right\n")

move i = let (x,str1) = left i in
 let (y,str2) = left x in
 (y,str1++str2)
> move 4
(2,"move left\nmove left\n")
```

这样，向左移动了两次，同时也记录了移动的过程。在 Haskell 中，常常需要记录计算的

过程，而这种方式并不能很好地满足这种需要，因此，可以将计算的元组  $(a, w)$  抽象为一个 `Monad` 来做连续的计算与记录。记录器 `Monad` 在做基于类型  $a$  的计算的同时，还可以用  $w$  类型来对这个计算做一些记录，所以称之为记录器 `Monad` (`Writer Monad`)。类型  $w$  需要限定为一个 `Monoid`，关于记录器 `Monad` 的定义在 `Control.Monad.Writer` 中。其中，`Writer` 构造器为私有的，但是可以使用 `writer` 函数来代替，读者可以查阅 API 来了解相关内容。

记录器 `Monad` 是这样定义的：

---

```
import Data.Monoid
newtype Writer w a = Writer { runWriter :: (a,w) }

writer = Writer

instance (Monoid w) => Monad (Writer w) where
 return x = Writer (x, mempty)
 (Writer (x,v)) >>= f = let (Writer (y,v')) = f x
 in Writer (y, v `mappend` v')
```

---

这里的 `>>=` 与前面的 `move` 函数定义十分相似，我们正是把这种记录过程放到记录器 `Monad` 中自动处理了。这里可以使用任何除 `String` 之外的实现了 `Monoid` 类型类的实例来记录计算过程中需要的信息。有了 `Writer Monad`，上边的例子可以改写为：

---

```
left', right' :: Int -> Writer String Int
left' x = writer (x-1, "move left\n")
right' x = writer (x+1, "move right\n")
```

---

这样，使用 `do` 关键字就可以做连续的计算了，记录的部分被隐含在了 `left'` 与 `right'` 函数中。不用在 `move'` 中做处理，例如：

---

```
move' i = do
 x <- left' i
 y <- left' x
 return y

> runWriter $ move' 4
(2,"move left\nmove left\n")
```

---

这样，借助于 `Writer Monad` 就可以在连续计算的同时记录结果了。但是，如果要移动很多次，那么程序看上去可能会是这个样子：

---

```
move' i = do
 a1 <- left' i
 a2 <- right' a1
 a3 <- left' a2
 a4 <- left' a3
 a5 <- left' a4
 return a5
```

---

这样就显得非常麻烦了，这是由于 `Writer Monad` 总是会返回计算的结果，而没有在程序

的背后保持结果的状态。如果读者想写成下面的样子，就需要用 Writer Monad 与 State Monad 的组合或者复合，State Monad 将在 12.3 节讨论，如何组合各个 Monad 的部分将在下章中讲解。

```
{-
move' = do
 left'
 right'
 left'
 left'
 return ()
-}
```

### 12.1.1 MonadWriter

MonadWriter 类型类是对所有 Writer Monad 行为的一种总结。它是这样定义的：

```
class (Monoid w, Monad m) => MonadWriter w m | m -> w where
 tell :: w -> m ()
 listen :: m a -> m (a, w)
 pass :: m (a, w -> w) -> m a
```

可以看到，它的定义中使用了类型依赖，即对于一个 Monad m 只能有一个记录类型 w 与之对应。其中，tell 函数不会做任何计算，而是直接将记录信息追加到末尾；listen 函数在会返回计算的结果的同时返回记录的值。而 pass 则会对记录器 Monad 输入一个函数。

```
move' i = do
 x <- left' i
 tell "moved left once!\n gonna move again\n"
 y <- left' x
 return y

> runWriter (move' 4)
(2,"move left\nmoved left once!\n gonna move again\nmove left\n")

move'' i = do
 x <- left' i
 y <- listen (right' x)
 return y

> runWriter (move'' 4)
(4,"move right\n"), "move left\nmove right\n")
```

这样，listen 不仅仅把当前的位置 4 返回，也将 right' 所记录的值一并返回了。

```
foo x = writer ((x-1, map toUpper), "foo")

move' i = do
 x <- left' i
 pass (foo 10)
```

```
>runWriter (move' 10)
(9,"move left\nFOO")
```

`pass` 函数在对记录内容前会用函数映射一次。通过 `listen` 可以定义一个叫 `listens` 的函数，`listens` 函数会对记录的内容应用函数 `f`，然后返回结果和映射后记录的值，原有记录的状态是不变的。另外一个函数是 `censor`，`censor` 是使用 `pass` 定义的，它是将函数 `f` 直接应用到了记录的内容上。

```
listens :: (MonadWriter w m) -> (w -> w) -> m a -> m (a, w)
listens f m = do
 (a, w) <- listen m
 return (a, f w)

censor :: (MonadWriter w m) -> (w -> w) -> m a -> m a
censor f m = pass $ do
 a <- m
 return (a, f)

move' i = listens (map toUpper) (do
 x <- left' i
 y <- left' x
 return y)

> runWriter (move' 4)
((2,"MOVE LEFT\nMOVE LEFT\n"), "move left\nmove left\n")

move' i = censor (map toUpper) (do
 x <- left' i
 y <- left' x
 return y)

> runWriter (move' 4)
(2,"MOVE LEFT\nMOVE LEFT\n")
```

### 12.1.2 记录归并排序过程

在本节中，使用记录器 Monad 来实现一个记录归并排序过程的函数。合并两个列表的 `merge` 函数读者应该已经非常熟悉了，它定义为：

```
import Control.Monad.Writer

merge [] xs = xs
merge xs [] = xs
merge (x:xs) (y:ys)
| x <= y = x : merge xs (y:ys)
| otherwise = y: merge (x:xs) ys
```

为了使得记录看起来更加美观，这里定义 `indent` 与 `nl` 函数来控制缩进与换行。可以很方便地使用它们来格式化记录。

便地用复合函数来取代`++`运算符来连接多个字符串。

---

```
indent :: Int -> ShowS
indent n = showString (take (4 * n) (repeat ' '))

nl :: ShowS
nl = showChar '\n'
```

---

最后就是定义 `mergesort` 函数了，它的类型签名中，第一个参数为缩进的级别；第二个参数为要排序的列表。当列表为空或者仅仅有一个元素时，直接返回。当有多个元素时，用 `tell` 函数来记录要排的列表，之后分成两份，再用 `tell` 来记录合并的两个列表。然后，递归地调用 `mergesort` 函数即可，这里需要使用 `liftM2` 函数将一个普通的函数二元函数转换为有 `Monad` 行为的二元函数。它的类型为 `liftM2 :: Monad m => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r`。

---

```
mergesort :: Int -> [Int] -> Writer String [Int]
mergesort l [] = do
 return []
mergesort l s@[x] = do
 return [x]
mergesort l s@xs = do
 tell $ (indent l.showString "mergesort: ".shows s.nl) ""
 let (a1,a2) = splitAt (length s `div` 2) xs
 tell $ (indent (l+1).showString "merge".shows a1.shows a2.nl) ""
 liftM2 merge (mergesort (l+2) a1) (mergesort (l+2) a2)

> putStrLn $ execWriter (mergesort 0 [5,4,3,6])
mergesort: [5,4,3,6]
merge[5,4][3,6]
mergesort: [5,4]
merge[5][4]
mergesort: [3,6]
merge[3][6]
```

---

## 12.2 读取器 Monad

如果只需要从一个不变的数据中读取信息，就可以使用读取器 `Monad` (`Reader Monad`)。这个数据就相当于只读的，比如给定一个字符串为读入的数据，可以读一个字符，也可以返回它的长度，但就是不能对它进行操作，比如加一个后缀、应用 `reverse` 函数等。它定义在 `Control.Monad.Reader`。同样，`Reader` 构造器是私有的，但是可以使用 `read` 来替代构造器。它是这样定义的：

---

```
newtype Reader r a = Reader { runReader :: r -> a }

instance Monad (Reader r) where
 return a = Reader $ _ -> a
 m >>= k = Reader $ \r -> runReader (k (runReader m r)) r
```

---

其中，Reader r a 中的 r 类型就是给定的数据，可以从中提取想要的信息。构造器中不再是一个普通的值而是一个函数。类型 r 的值称为环境，即从该环境下读出类型为 a 的值来，所以读取器 Monad 有时也称为环境 Monad (Environment Monad)。>>= 运算符在给定的环境 r 下执行 m 后，将结果输入函数 k 中。比如，给定了一个很长的字符串，如果需要读取它的长度，那么读取器 Monad 的类型就应当为 Reader String Int。定义如下：

---

```
readLen :: Reader [a] Int
readLen = reader $ \r -> length r

> runReader readLen "12345678"
8
```

---

当然我们还可以读一个列表的第一个元素。

---

```
readHead :: Reader [a] a
readHead = reader $ \x:xs -> x
```

---

### 12.2.1 MonadReader

MonadReader 类型类进一步地抽象了 Read Monad 的行为。ask 函数可以将当前的环境读取进来。而 local 相当于在读入时将信息应用函数 f 一次，因为只是局部有效，所以称为 local。

---

```
class (Monad m) => MonadReader r m | m -> r where
 ask :: m r
 local :: (r -> r) -> m a -> m a

instance MonadReader r (Reader r) where
 ask = Reader id
 local f m = Reader $ runReader m . f

 test :: Reader [Int] [Int]
 test = do
 xs <- local (map (+1)) ask
 ys <- ask
 return ys
```

---

这里 test 函数中的 xs 则会使环境的参数全部加 1，但是当再读时，ys 还是原来的环境。库中还提供了 withReader 与 mapReader 函数。withReader 对环境应用函数 f，得到一个新的读取器。mapReader 相当于 local 的更为一般的形式，因为给定的函数不仅仅限定于一个类型，mapReader 事实上是把 Reader r 类型实现成为了函子类型类。

---

```
withReader :: (r' -> r) -> Reader r a -> Reader r' a
withReader f m = Reader $ runReader m . f

mapReader :: (a -> b) -> Reader r a -> Reader r b
mapReader f m = Reader $ f . runReader m
```

---

### 12.2.2 变量环境的引用

下面举一个简单的例子，在解决24点游戏定义表达式树时，所有的叶子需要为数值。现在来加一个变量环境，比如，Haskell中可以有`let x = 4 in x + 5`这样的定义。如果用元组的列表来表示环境，这里`x + 5`的变量环境就为`[(x, 4)]`。在计算时，需要来管理变量的环境，并且在计算前将变量的值替换下来。

---

```
import Control.Monad.Reader
import Data.List (lookup)
```

---

为了简化问题，这里只定义加法：

---

```
data Exp = Val Int
 | Var String
 | Add Exp Exp
 | Decl Bind Exp deriving (Show, Eq)

type Bind = (String, Int)
type Env = [Bind]
```

---

声明变量时不考虑命名冲突的问题。

---

```
updateEnv :: Bind -> Env -> Env
updateEnv = (:)
```

---

`resolve`函数就是通过读取器 Monad 定义的了。它从环境中读取变量然后替换到表达式中，如果变量不存在，那么返回`Nothing`。

---

```
resolve :: Exp -> Reader Env (Maybe Exp)
resolve (Val i) = return (Just (Val i))
resolve (Var s) = do
 env <- ask
 case lookup s env of
 Nothing -> return Nothing
 Just v -> return (Just (Val v))

resolve (Add e1 e2) = do
 rel1 <- resolve e1
 case rel1 of
 Nothing -> return Nothing
 Just a -> do
 rel2 <- resolve e2
 case rel2 of
 Nothing -> return Nothing
 Just b -> return (Just (Add a b))
```

---

当声明新的变量时，只需要使用`local`函数将变量加入到当前的环境，就可以递归地调用`resolve`函数了。

---

```

resolve (Decl b e) = local (updateEnv b) (resolve e)

-- let x=3 in let y=5 in x + (y+6)
test1 :: Exp
test1 = Decl ("x",3) (Decl ("y",5) (Add (Var "x") (Add (Var "y") (Val 6)))))

--let x = 2 in x + let x = 3 in x
test2 :: Exp
test2 = Add (Decl ("x",2) (Var "x")) (Decl ("x",3) (Var "x"))

```

---

可以来测试这个函数。给定一个空的环境，即[]。下面测试一下这个函数：

---

```

> runReader (resolve test1) []
Just (Add (Val 3) (Add (Val 5) (Val 6)))

```

---

对于 test2，两边都有定义变量 x，但是左侧的 x 仅仅在左侧局部有效，而右侧也仅仅在右侧有效，所以结果应该为 Add (Val 2) (Val 3)。使用这种方法保证了不同级别内相同变量名不会冲突。

---

```

> runReader (resolve test2) []
Just (Add (Val 2) (Val 3))

```

---

## 12.3 状态 Monad

状态 Monad (State Monad) 代表一类需要保持状态的计算。在函数计算时，有一状态可能会随着计算而改变并且有着相应的输出，状态 Monad 是最重要的 Monad 之一，相对而言要难理解一些。状态 Monad 的相关函数定义在库 Control.Monad.State 中，同样地 State 构造器是私有的，但是可以用 state 来替代。它是这样定义的：

---

```

newtype State s a = State { runState :: s -> (a,s) }

```

---

State s a 可以理解成代表一个有着状态为类型 s，输出结果为类型 a 的计算。State 构造器内是一个函数，而不是直接可以返回的值，这个函数可以理解为传入一个状态，做一次计算，这个计算产生一个类型为 a 的结果和一个新状态，用元组表示成(a,s)。作为 Monad，它是被这样定义的：

---

```

instance Monad (State s) where
 return x = State $ \s -> (x,s)
 (State h) >>= f = State $ \s -> let (a, newState) = h s
 (State g) = f a
 in g newState

```

---

return 函数需要输入一个值 x 并让这个 Monad 返回它，这样，这个值正是这个有状态计算的结果，即输入的 x 就是结果，而状态保持不变。>>= 运算符把一个计算状态的函数 h 传入函数 f，我们来分析一下函数 f，函数 f 是一个类型为 ( $\lambda s_1 \rightarrow (x, s_1)$ )  $\rightarrow$  State ( $\lambda s_2 \rightarrow (x_2, s_2)$ ) 的函数，前边的 ( $\lambda s_1 \rightarrow (x, s_1)$ ) 就是 h，函数 h 用输入的状态生成了新的状态，并返回一个结果，可以记作 (a, newState)，f 函数用了新的结果 a，得到一个新的 Monad 值，f a

中的 Monad 值内又存储了函数 `g`, 将之前新的状态结果输入给 `g` 函数, 就得到了最终的结果与状态。这样, 通过这种方式将状态的计算交给 State Monad, 状态会在 State Monad 里一直保持并更新着; 需要一个新的结果时, State Monad 会返回这个结果, 并自动计算下一个状态。这样我们就可以省下更多的精力来关注真正要编写的部分。

关于 State Monad 的函数都在 Control.Monad.State 库中。这个库提供了一个函数 `state :: (s -> (a,s)) -> State s a` 来代替构造器。还有 `evalState :: State s a -> s -> a` 和 `execState :: State s a -> s -> s` 两个函数, 它们分别相当于应用 `runState` 函数后, 分别再应用 `fst` 与 `snd` 函数收回结果与状态的值。

### 12.3.1 状态 Monad 标签器

以深度优先 (depth first) 的顺序来对一个二叉树的各个结点与叶子进行访问, 即访问左支时一直遍历到底, 然后访问右支。在访问时, 对希望访问的顺序予以标记。这样, 需要保持一个状态计数器来记录目前已经数到第几个节点。不用状态 Monad 时可以这样定义:

---

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

labelTree :: Tree a -> Tree (a, Int)
labelTree t = fst $ ntAux t 0

ntAux :: Tree a -> Int -> (Tree (a, Int), Int)
ntAux (Leaf a) n = (Leaf (a, n), n+1)
ntAux (Node l a r) n = let (nn, n') = ((a, n), n+1) in
 let (ln, n'') = ntAux l n' in
 let (rn, n''') = ntAux r n''' in
 (Node ln nn rn, n''')
```

---

`ntAux` 是辅助函数, 它的第二个参数即为之前数到某个节点的整数。若当前的节点为叶子, 那么计数器应当加 1。当遇到节点时, 计数器也加 1, 然后递归地对树的左支与右支遍历并且标记。

---

```
test :: Tree Int
test = Node (Node (Leaf 5) 3 (Leaf 2)) 7 (Leaf 9)

> labelTree test
Node (Node (Leaf (5,2)) (3,1) (Leaf (2,3))) (7,0) (Leaf (9,4))
```

---

使用 `let .. let .. in .. in ..` 这样的方式来表达连续地让状态计数器加 1 的计算。显然, 这种写法十分麻烦、易出错并且可读性差。下面, 使用状态 Monad 来重新定义它。

---

```
import Control.Monad.State

data Tree a = Leaf a | Node (Tree a) a (Tree a) deriving (Show, Eq)

increase :: State Int Int
increase = state $ \i -> (i, i+1)
```

---

当调用 increase 时，输入的整数 i 将会被返回，同时，它会更新存储的状态为 i+1。这样，每调用 increase 一次，就会自动加 1。状态 Monad 通过这种方式代表了顺序式语言中 i++ 这种效应。使用 increase 定义 ntAux 函数就相当直观了。increase 会将状态中的整数读取出来，自动加 1，状态 Monad 会处理好整数的更新，这样就不必在 ntAux 中处理它了。

---

```
ntAux :: Tree a -> State Int (Tree (a,Int))
ntAux (Leaf a) = do
 nl <- increase
 return (Leaf (a,nl))

ntAux (Node l n r) = do
 nl <- increase
 lt <- ntAux l
 rt <- ntAux r
 return (Node lt (n, nl) rt)
```

---

labelTree 中的参数 0 是初始状态，也就是说从 0 开始计数。

---

```
labelTree t = evalState (ntAux t) 0

test :: Tree Int
test = Node (Node (Leaf 5) 3 (Leaf 2)) 7 (Leaf 9)

> labelTree test
Node (Node (Leaf (5,2)) (3,1) (Leaf (2,3))) (7,0) (Leaf (9,4))
```

---

这个例子也是来自于诺丁汉大学 Henrik Nilsson 教授的《编译原理课程》的第 9 节——A Versatile Design Pattern: Monads。读者可以到 [http://www.cs.nott.ac.uk/~nhn/G53CMP/Lecture\\_Notes-2011/lecture09.pdf](http://www.cs.nott.ac.uk/~nhn/G53CMP/Lecture_Notes-2011/lecture09.pdf) 浏览讲义的内容。

### 12.3.2 用状态 Monad 实现栈结构

计算机的栈（stack）数据结构可以抽象地用一个 State Monad 表示，并且需要对栈进行 push 压栈、pop 出栈、peek 观栈操作。在这种结构中我们只能知道栈顶的内容，最后进入的元素在使用 pop 时会第一个出来，而最先入栈的元素则会最后一个出来。而这种栈里的内容可以理解为一个状态，push、pop 函数导致这个状态的变化由状态 Monad 计算。例如：

---

```
import Control.Monad.State
type Stack = [Int]

pop :: State Stack Int
pop = state $ \x:xs -> (x,xs)
```

---

pop 出栈函数很简单，当需要 pop 一个结果时，需要一个 Stack 类型的值作为输入，将栈顶的元素作为结果返回，与此同时，栈顶的元素也从栈中移除了，xs 也就是新的状态。

---

```
peek :: State Stack Int
peek = state $ \x:xs -> (x, x:xs)
```

---

peek 观栈函数也很简单，只是看一下栈顶的内容是什么，而不改变栈内元素的状态。

---

```
push :: Int -> State Stack ()
push i = state $ \xs -> ((), i:xs)
```

---

push 压栈函数中用到了 () 值，() 值具有 () 类型，这个类型中也只有这么一个值。它的使用有些像是 C 或者 Java 中的 void 关键字，意思是说，push 函数不从 State Monad 中返回任何结果，而只是去改变这个 State Monad 的状态。这里和 IO () 很像，IO () 的意思是说，不从输入/输出设备里读入任何值，而是做一些输出操作，完全不返回任何值在 Haskell 中是无法做到的，这里仅仅用 () 类型来返回一个没用的值。

我们看一下 State Monad 是如何工作的：

---

```
instance Monad (State s) where
 return x = State $ \s -> (x, s)
 (State h) >>= f = State $ \s -> let (a, newState) = h s
 (State g) = f a in g newState
```

---

向 [0] 这个栈的状态中 push 5，然后 pop：

---

```
runState (push 5 >>= \s -> pop) [0]
= runState (State \xs -> (((), 5:xs)) >>= (\s -> State \x:xs -> (x, xs))) [0]
```

---

对应一下 State Monad 实例中的 h 与 f：

---

```
h = \xs -> (((), 5:xs))
f = \s -> State \x:xs -> (x, xs)

= runState State $ (\s -> let (a, newState) = h s
 (State g) = f a
 in g newState) [0]
```

---

此时：

---

```
h [0] = (((), [5, 0]))
f () = State \x:xs -> (x, xs)

= runState State $ (\s=[0] -> let (a, newState) = (((), [5, 0]))
 (State g) = (\s -> State \x:xs -> (x, xs) ())
 in g newState)

g = \x:xs -> (x, xs) newState = [5, 0]
g newState = (\x:xs -> (x, xs)) [5, 0]
=(5, [0])
```

---

这样，最后的结果就是 (5, [0])。可以看到 push 5 后，[5, 0] 是新状态，函数调用 h [0] 后得一个结果和一个新的状态，而 push 5 后再 pop，pop 不需要其他输入，所以这里 () 的输入

仅仅是起一个函数参数占位的作用，将`\s -> State \x:xs->(x, xs)`中输入的`s`替换成`()`，得到 Monad 值。因为`pop`函数在计算不需要任何输入，所以无论函数参数输入什么都会有结果。但是，这是一个函数类型，因此一定要有一个参数输入。在运行一个不需要返回任何值的函数时，就使用`()`作为返回的值，然后又将`()`值当做函数的参数输入，从而得到一个新的 Monad 值。比如：

```
f = do:
 push 5
 a <- pop
 push (a+5)
f = push 5 >>= (\b ->
 pop >>= (\a ->
 push (a+5)))
```

在 Haskell 计算时，如果不使用`do`关键字，而是用`>>=`运算符，可以看到这里的`b`其实是`()`。接下来，可以写些函数来对一个 State Monad 操作，将栈上边的两个数加在一起，然后压到栈中。

---

```
addStack :: State Stack Int
addStack = do:
 a1 <- pop
 a2 <- pop
 let a3 = a1+a2
 push a3

>runStack addStack [5,6]
(),[11]
```

---

这样的书写看起来非常的舒服，也节省了很多用于控制栈的操作的代码。栈的状态一直保持在 Monad 内部，时刻准备着接受下一个栈的操作，因此不必在代码中特别地加以处理。

### 12.3.3 状态 Monad、FunApp 单位半群和读取器 Monad 的关系

其实，State Monad 可以理解为由 FunApp 单位半群和读取器 Monad 组合成的一个 Monad。下面来看一下它们的定义：

---

```
newtype FunApp s = FunApp { appFunApp :: s -> s }
newtype Reader s a = Reader { runReader :: s -> a }
newtype State s a = State { runState :: s -> (a,s) }
```

---

从上面的定义可以看出，FunApp 单位半群是只能计算状态，却不能从状态中计算结果的一种类型。而 Reader Monad 是一种只能从状态中读取信息却不能改变的状态的类型。

以栈为例：

---

```
import Data.Monoid
import Control.Monad.Reader

(>|>) :: Monoid a => a -> a -> a
(>|>) = mappend

newtype FunApp a = FunApp { appFunApp :: a -> a }
instance Monoid (FunApp a) where
```

---

---

```

mempty = FunApp id
FunApp f `mappend` FunApp g = FunApp (g . f)

push :: Int -> FunApp Stack
push i = FunApp $ \xs -> i:xs

pop :: FunApp Stack
pop = FunApp $ \x:xs -> xs

m :: FunApp Stack
m = push' 3
 |> push' 1
 |> pop

```

---

这样，使用 FunApp 单位半群可以顺序地从左至右改变栈的存储状态，但是却无法从中读值。因为构造器中只有状态，而没有返回值，所以 FunApp 上的操作可以理解为复合只对状态进行操作的函数。而对于 Reader Monad，只能从中读取某一个元素、列表总长等信息，因为它只有返回值，所以不能修改这个存储状态，也就是读取器 Monad 中的环境。

---

```

readLength :: Int -> Reader Stack Int
readLength n = reader $ \xs -> length xs

```

---

但是借助于状态 Monad 可以同时做两件事，第一件事就是保持并且修改状态，第二件事就是从这个状态中读值并返回。它可以理为读取器 Monad 和 FunApp 单位半群的组合。

#### 12.3.4 MonadState

像记录器 Monad 与读取器 Monad 一样，状态 Monad 同样也提供了更高一级的抽象 MonadState。MonadState 中提供了两个函数，即 get 与 put。其中，get 函数会返回当前的状态，而 put 则会设置当前的状态，这里就不再举例介绍了。

---

```

class (Monad m) -> MonadState s | m -> s where
 get :: m s
 put :: s -> m ()

instance MonadState s (State s) where
 get = State $ \s -> (s, s)
 put s = State $ _ -> ((), s)

```

---

#### 12.3.5 基于栈的计算器

用状态 Monad 定义的栈这种抽象的模型（或者称为数据结构）在计算机编程中是十分有用的。一般栈的操作只有两个——push 和 pop，它们看似简单，但很多看似复杂的问题实际上都可以仅使用这两个操作来解决。比如，使用浏览器浏览网页时，前进与后退就是通过栈来实现的；再比如，Java 的虚拟机其实也是用指令来对堆栈的模型进行操作来运行的。在本节里，用

栈来实现一个简易的数学计算器。在本书开始提到过 GHCi 可以当做计算器来使用，下面在 GHCi 中输入一个数学计算式，比如

---

```
> 4+5*6/3+2^2
18.0
```

---

如果让人来做这个计算当然是很容易的，但 GHCi 是如何将输入的字符串计算成数字作为结果返回的？在编译 Java 程序时，程序的代码中常常有一些算术表达式，Java 编译器知道如何计算它们。可是，当需要一个程序将字符串计算成数字时应该怎么办？是否也可以用 Haskell 中写出一个程序像卡西欧(CASIO)科学计算器那样可以计算一串很长的数学计算式的程序呢？答案当然是可以的。很多有编程经验的人写这样的程序往往很直接地想到使用语法分析器(parser)，或者借助语法分析器生成器(parser generator)来建立语法树(syntax tree)，然后对语法树进行计算求值，比如 Haskell 的 Happy、C 语言有 Yacc、Java 有 ANTLR 等。例如，下面是 24 点一节中的表达式语法树，可以先使用语法分析器生成器生成一个语法分析器，定义一个类型为 `String -> Exp` 的函数，最后再对表达式树递归地估值即可。

---

```
data Exp = Val Double
| Plus Exp Exp
| Sub Exp Exp
| Mult Exp Exp
| Div Exp Exp deriving (Show, Eq)

eval :: Exp -> Double
eval (Val a) = a
eval (Plus a b) = eval a + eval b
...
```

---

可能有的人想把估值的计算运行很多轮，从优先级最高的开始计算，计算到表达式中没有运算符号为止。但是这些方法往往效率不高，实现起来也相对复杂。事实上，一般的数学表达式计算只需要对两个栈进行操作就可以了，这种算法由 Edsger Dijkstra 在 1960 年提出。

这里要明确，我们写的这个程序其实只是一个函数，是一个以 `String -> Float` 为类型的函数(这里统一返回结果类型为 `Float`)。在定义函数时，需要将数字的值(Lit)与运算符(Op)分开，但却需要将它们存放在一个列表中，所以需要使用 Either 合并它们。可以先将 String 解析成 [Either Lit Op] 类型，Either Lit Op 为这个数学语言的基本单位。这里，每一个数值、每一个运算符都是一个基本单位，在编译原理中，它们被称为“记号”(token)。将字符串分析成记号后，再对这些记号进行相应堆栈的操作就可以计算出结果了。简单来说，需要将 `String -> Float` 分成两个函数来写，第一个函数用来分析字符串，它的类型为 `String -> [Either Lit Op]`；另外一个函数用来做计算，它的类型是 `[Either Lit Op] -> Float`。这里先来考虑第二部分，然后再来讨论第一部分。

计算的过程主要是应用状态 Monad，在状态 Monad 中维护两个栈，然后通过状态 Monad 对两个栈进行操作，一个栈存储数值，另一个存储运算符号。现假设这两个栈分别为栈 0 跟栈 1，栈 0 用于存储数字，栈 1 用于存储符号。

每个运算符号都有自己的优先级，当然，还有位置以及结合的属性，这里只考虑一元前置运算符，例如：以 2 为底的对数函数  $\log$ 、正弦函数  $\sin$ 、余弦函数  $\cos$ ，还有二元中缀运算符，如  $+ -$  等。当遇到数字的时候，将数字压入栈 0 内；当遇到符号时，如果一元前置运算符，如对数则直接压入栈 1 内。但是，如果是二元中缀运算符，那么是否压栈还需要对运算符的优先级与结合性进行讨论。下面就来讨论这种具体情况。

如果栈顶的运算符为一元前置运算符，并且优先级大于当前输入的运算符，那么应当分别取出栈 0 顶部的数字和栈 1 顶端的运算符进行计算。然后，将结果压入栈 0，再将这个二元运算符递归地压入栈中，因为下一次压入时，栈 1 顶端仍旧可以是一个一元运算符，并且依旧可以继续运算。

例如，假设需要计算  $\log \log 4 + 5$ 。在加号入栈前，两个栈如图 12-1 所示。此时，当加号要入栈时，由于对数运算的优先级大于加法，因此可以计算  $\log 4$  的值。然后，当加法第二次试图入栈的时候，它会发现还可以计算  $\log 2$  的值。

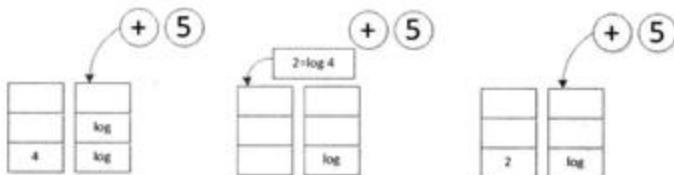


图 12-1 入栈计算过程示意图 1

但如果此二元运算符的优先级大于这个一元前置运算符，则需要将这个二元运算符直接压入栈 1 中，如  $\log \log 4^5$ ，如图 12-2 所示。

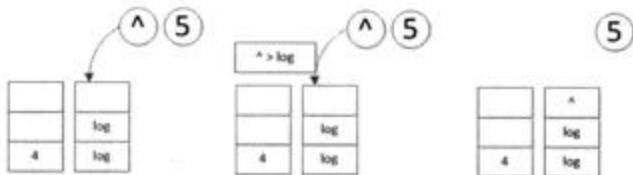


图 12-2 入栈计算过程示意图 2

由于幂运算的优先级要比对数高，因此应该直接入栈而不能去计算  $\log 4$  的值，而幂运算还需要等待它的第二个参数。

如果栈 1 的顶端是一个二元运算符，如  $4+5+3$ ，那么当栈操作进行到图 12-3 所示的过程时，由于加法的优先级不大于加法，所以在压入第二个加号前，可以进行栈 1 顶端运算符的计算。那么，这里则说明加法在计算的时候是左结合的，减法、乘法与除法这里都是左结合的。可是，当压入的运算符优先级大于栈 1 顶端的运算符时，则不可以进行计算，需要直接入栈，比如  $4+5*3$ ，当乘号需要入栈时，如图 12-4 所示。

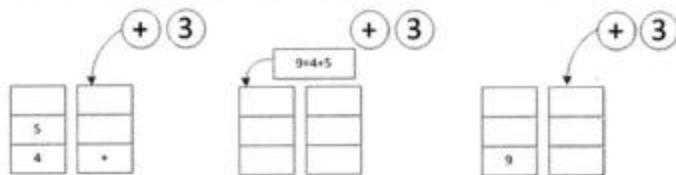


图 12-3 入栈计算过程示意图 3

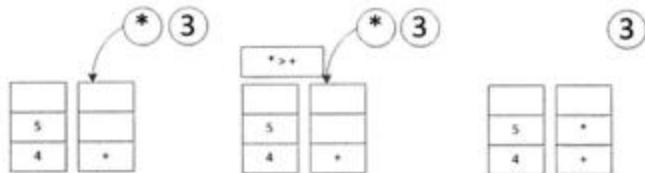


图 12-4 入栈计算过程示意图 4

如果栈 1 为空，那么直接将运算符压入栈 1 中即可。

相信读者现在对于用栈计算的过程已经有了一定的了解，下面先来写用栈计算这一部分。首先，定义一些需要的数据类型，然后用状态 Monad 来定义栈对它们进行计算操作。

---

```
module Calculator where
import Control.Monad.State
```

---

首先，定义参与计算的量。定义 `Lit` 时，将值与常量分开，`Val` 中可以存储任何的小数，而在 `Const` 中存储一个字符串，这样可以引入一些常用的数学常量比如圆周率  $\pi$  和自然对数的底数  $e$ ，分别记作 `Const "pi"` 和 `Const "e"`。这里定义的 `Empty` 可以在一元运算的计算时起到一个占位的作用，而不必把一元运算和二元函数估值的函数分开。

---

```
data Lit = Val Float | Const String | Empty deriving (Eq, Show)
```

---

然后，定义哪些运算符可以在计算器中使用，虽然正号与加法、负号与减法用的符号分别都相同，但是意义不同，需要分开定义。同时，左括号和右括号被当做运算符处理。`OpBottom` 仅仅是一个标记，如果 `opBottom` 出栈，则说明已到达栈 1 的栈底。

---

```
data Op = Posi | Nega | Plus | Minus
 | Mult | Divi | Power
 | Log | Ln | Sin | Cos | Sqrt
 | L_Par | R_Par
 | OpBottom
deriving (Eq, Show)
```

---

最后，定义运算符为几元的类型 `Order`。`Unary` 为一元运算符，`Binary` 为二元运算符，`Null` 则是为了判断括号，`Bottom` 则用来判断是否为栈底。

---

```
data Order = Unary | Binary | Null | Bottom
```

---

下面定义一个函数，将这些运算符根据不同元次归类。

---

```
nary :: Op -> Order
nary op = case op of
 Plus -> Binary
 Minus -> Binary
 Mult -> Binary
 Divi -> Binary
 Power -> Binary
 Posi -> Unary
 Nega -> Unary
 Log -> Unary
 Ln -> Unary
 Sin -> Unary
 Cos -> Unary
 Sqrt -> Unary
 OpBottom -> Bottom
 _ -> Null
```

---

当然，如果读者不喜欢这样定义 `nary` 函数，也可以用 `elem` 来对参数进行分析。接下来，定义这些运算的优先级。括号优先级为零，此外，加减法最低，幂最高。这里需要注意运算符的结合性质，加法与乘法是有结合律的，左结合还是右结合对它们不会有影响。减法与除法为左结合运算符号，而幂函数为右结合运算符号。因此，这里默认其他运算符为左结合的情况下，幂运算符入栈时需要被特殊考虑。

---

```
priority :: Op -> Int
priority op = case op of
 Plus -> 1
 Minus -> 1
 Mult -> 2
 Divi -> 2
 Log -> 3
 Sin -> 3
 Cos -> 3
 Posi -> 4
 Nega -> 4
 Sqrt -> 4
 Power -> 5
 _ -> 0
```

---

为了方便，要给常用的类型重新命名。由于栈中可以存放两种不同的类型，因此需要用 `Either` 来联合两种类型。

---

```
type LitOp = Either Lit Op
```

---

接下来定义的就是这两个栈了，这两个栈都是以 `LitOp` 为类型的列表，只不过左侧的只存放 `Lit`，右侧的只存储运算符。

---

```
type Stack = ([LitOp], [LitOp])
```

---

给定运算符和两个 Lit，则 evaluate 函数会进行求值。如果是一元运算符，就使用 Empty 占位。这个函数定义起来看上去还是有些啰嗦的。

---

```
evaluate :: Op -> LitOp -> LitOp -> State Stack ()
evaluate op (Left (Val f1)) (Left (Val f2)) = case op of
 Plus -> push $ lv (f1+f2)
 Minus -> push $ lv (f1-f2)
 Mult -> push $ lv (f1*f2)
 Divi -> push $ lv (f1/f2)
 Power -> push $ lv (f1**f2)
evaluate op (Left (Val f1)) (Left Empty) = case op of
 Posi -> push $ lv f1
 Nega -> push $ lv (-f1)
 Log -> push $ lv (logBase 2 f1)
 Ln -> push $ lv (log f1)
 Sin -> push $ lv (sin f1)
 Cos -> push $ lv (cos f1)
 Sqrt -> push $ lv (sqrt f1)

lv :: Float -> LitOp
lv x = Left $ Val x
```

---

接下来，定义 pop0、pop1 来对两个栈进行出栈操作，定义 push 对运算符进行压栈操作。这个 push 没有任何条件地将元素压入栈中。由于值与运算符有 Left 和 Right 之分，因此入栈函数没有必要定义成两个。同时，在压栈时，可以将一些常量名直接转换为对应的值。

---

```
pop0, pop1 :: State Stack LitOp
pop0 = state $ \(ls, rs) -> case ls of
 [] -> error "Number stack underflow"
 (h:hs) -> (h, (hs, rs))
pop1 = state $ \(ls, rs) -> case rs of
 [] -> error "Operator stack underflow"
 (h:hs) -> (h, (ls, hs))

push :: LitOp -> State Stack ()
push (Left (Const "pi")) = push $ lv 3.1415926
push (Left (Const "e")) = push $ lv 2.7182812
push (Left (Const c)) = error $ "Unknown Constant"++c
push l@{Left a} = state $ \(ls, rs) -> ((()), (l:ls, rs))
push r@{Right a} = state $ \(ls, rs) -> ((()), (ls, r:rs))
```

---

下面的 pushIn 函数就是这个栈计算器的核心部分了。如果需要入栈的是数字或者左括号，那么直接入栈即可。

---

```
pushIn :: LitOp -> State Stack ()
pushIn l@{Left num} = push l
pushIn p@{Right L_Par} = push p
```

---

当入栈的是右括号时，则说明这个计算式中括号内部的表达式可以被计算了，因为对于括号内的表达式计算的先后顺序不会影响整体的结果。如果入栈时，发现栈顶运算符的元次为 Null，则说明为栈 1 的顶端必为左括号，遇到左括号则说明这个括号内的表达式已经计算好了，那么左括号出栈后不必再做任何操作，故 return ()即可。

如果出栈的为二元运算符，那么从栈中取出两个数字，做这个运算符的求值计算。类似地，如果是一元运算符，则仅取出一个数字做这个运算符的求值计算。如果遇到栈底，则说明右括号比左括号多一个，那么就是表达式有错误了。整个对栈的操作过程中，右括号是不会存放于栈中的，它最终会和某个左括号消去或者出现错误。

```

pushIn p@{Right R_Par} = do
 Right top <- popl
 case nary top of
 Null -> return ()
 Unary -> do
 f1 <- pop0
 evaluate top f1 (Left Empty)
 pushIn p
 Binary-> do
 f1 <- pop0
 f2 <- pop0
 evaluate top f2 f1
 pushIn p
 Bottom -> error "Expected Left Bracket\n"

```

如果即将入栈的运算符为一元前置运算符，那么它需要等待其后的参数才能计算，因此直接入栈。

如果即将入栈的运算符为二元中缀运算符，那么就要像已经在本节开始阐述的那样，需要对于栈顶端的运算符的元次和优先级进行讨论。

当栈顶为一元运算符时，如果栈顶运算符优先级大于此二元运算符，则可以计算。例如  $\log$   $\log 4 + 5$ ，加号尝试入栈时， $\log 4$  的值可以被计算，因为对数函数的优先级比加法高，当再次尝试入栈时，还可以计算  $\log 2$  的值，所以要递归地使用 `pushIn` 函数。然而，如果这个二元运算符的优先级比栈顶的一元运算符大，比如  $\log 4^5$ ，当 $^$ 需要入栈时，则直接入栈即可。

---

```

pushIn o
False -> do
 push (Right top)
 push o

```

---

当栈顶为二元运算符时，首先需要考虑这个运算是否为幂运算。由于连续的幂运算是向右结合的，因此，如果要入栈的运算符为幂运算符，那么无论栈顶是什么运算符都要入栈，包括幂运算符本身，因为幂运算的优先级最高并且为右结合。比如  $2^3^2$ ，其计算的表达式为  $2^{(3^2)}$ ，也就是  $2^9$ 。如果栈顶不是幂运算符，并且优先级等于或者大于将要入栈的运算符，则可以做计算。比如  $4+3*5$ ，当减号入栈时，可以计算  $4+5$  的和。倘若栈顶的优先级小，比如  $4+3*5$ ，当乘号入栈时，加号在栈顶，加号优先级小于乘号，这时不可以计算，乘号要入栈来等待它的第二个参数。

---

```

Binary -> do
 case op of
 Power -> do
 push (Right top)
 push o
 -> do
 let pri=priority top >= priority op
 case pri of
 True -> do
 f1 <- pop0
 f2 <- pop0
 evaluate top f2 f1
 pushIn (Right op)
 False -> do
 push (Right top)
 push o

```

---

如果是左括号或者已经到达栈底，左括号和栈底类似，可以理解为一个新的表达式环境，或者说一个空栈，括号内的表达式是相对独立于外边的表达式的，这种只需要将出栈的左括号压入栈中后再压入这个运算符即可。

---

```

- -> do --- L_Par and OpBottom
 push (Right top)
 push o

```

---

这一部分已经基本完成了，最后只需要一个函数递归地使用 `pushIn` 函数将一个 `LitOp` 的列表依次压入栈中计算即可。从上面的代码中可以看出，这样的语言表达方法使得多情况的讨论、分支比起顺序式语言的 `if .. then ... else` 或者 `elseif` 要清晰很多，而且不容易产生遗漏。

若列表为空，则说明已经到达了表达式的末尾，只需要将栈中余下的记号进行计算即可。若栈 1 的顶端为栈底，则说明栈 0 内的值即为结果，所以直接操作栈 0 出栈即可。如果还有运算符，那么需要做运算，然后递归地调用 `calc []`。例如，计算  $4+2*3^2$  时，由于三个运算符的优先级一个比一个高，那么在末尾的 2 入栈之前是不会有任何计算的，但是记号列表已经为空，这时需要对栈中的幂、

乘法、加法依次运算。如果列表不为空，那么直接 pushIn，然后递归地 pushIn 余下的记号即可。

---

```

calc :: [LitOp] -> State Stack LitOp
calc [] = do
 Right op <- popI
 case nary op of
 Bottom -> popO
 Unary -> do
 f1 <- popO
 evaluate op f1 (Left Empty)
 calc []
 Binary -> do
 f1 <- popO
 f2 <- popO
 evaluate op f2 f1
 calc []
 Null -> error "Expected right bracket"

calc (t:ts) = do
 pushIn t
 calc ts

```

---

使用状态 Monad 需要一个初始的状态，显然，这个初始的状态应该在栈 1 中初始一个 OpBottom 来判断栈底。

---

```

inits :: ([LitOp], [LitOp])
inits = ([], [Right OpBottom])

```

---

下面来写一些测试：

---

```

test1 = [Right Nega, Left (Const "pi")]

test2 = [Right L_Par, Left (Val 5.0), Right Plus, Left (Val 6), Right R_Par, Right Mult,
Left (Val 3.0)] -- (5+6)*3

test3 = [Right Log , Right L_Par, Left (Val 8), Right Plus, Left (Val 8), Right R_Par,
Right Plus , Left (Val 5)] -- log (8+8) + 5

test4 = [Right Log , Left (Val 8), Right Power, Left (Val 2), Right Plus , Left (Val
5)] -- log 8^2 + 5

test5 = [Right Sqrt , Left (Val 4), Right Plus, Left (Val 4)] --sqrt 4 + 4

> (runState (calc test1)) inits
(Left (Val (-3.1415925)), ([], []))

> (runState (calc test2)) inits
(Left (Val 33.0), ([], []))

> (runState (calc test3)) inits

```

---

```
(Left (Val 9.0), ([]))
> (runState (calc test4)) init
(Left (Val 11.0), ([]))
> (runState (calc test5)) init
(Left (Val 6.0), ([]))
```

---

之前的工作已经做好了，下一步需要定义一个函数，将字符串转换成 LitOp，即写一个函数 scanner :: String -> [LitOp]。

计算表达式的结构是有一定的规则的，比如，乘号不会出现于表达式的开头，数字的后边不是右括号就是二元运算符，否则它就为表达式的末尾。可以用上下文无关文法（Context Free Grammar, CFG）来描述这个语言。在分析语法时，不必考虑结合性与优先级的问题，这些将由栈计算的过程来处理。规定一些术语和符号：表达式(Expression)、一元前置运算符开头的表达式(UnaryExpression)、二元中缀运算符开头的表达式(BinaryExpression)、数字(Number)、一元运算符 $\ominus$ 、二元运算符 $\oplus$ 、 $\epsilon$ 为空字符串。那么，算术表达式的语法规则如下定义：

---

```
Expression → ⊖ Expression
| (Expression) BinaryExpression
| Number BinaryExpression

BinaryExpression → * Expression | ε
```

---

以 Expression 为起始，Number 可以生成所有的数字， $\ominus$  与  $\oplus$  最后可以转换为一元与二元运算符。通过这种方法，可以生成所有的算术表达式。比如  $\log \sin 10 * (4+2)$ 。

---

```
Expression
-> ⊖ Expression
-> log Expression
-> log ⊖ Expression
-> log sin Expression
-> log sin Number BinaryExpression
-> log sin 10 BinaryExpression
-> log sin 10 * Expression
-> log sin 10 * Expression
-> log sin 10 * (Expression) BinaryExrepssion
-> log sin 10 * (Number BinaryExpression) BinaryExrepssion
-> log sin 10 * (4 BinaryExpression) BinaryExrepssion
-> log sin 10 * (4 * Expression) BinaryExrepssion
-> log sin 10 * (4 + Expression) BinaryExrepssion
-> log sin 10 * (4 + Number BinaryExrepssion) BinaryExrepssion
-> log sin 10 * (4 + 2 BinaryExrepssion) BinaryExrepssion
-> log sin 10 * (4 + 2) BinaryExrepssion
-> log sin 10 * (4 + 2)
```

---

如果对于这个语言的语法有了一些了解，就可以将输入的数学表达式字符串分析成记号的列表了。

```
--Scanner.hs
module Scanner where
import Data.Char
import Calculator
```

空格是可以被忽略的。先考虑一元前置运算符，一元运算符后面跟的还是一个 Expression 所以递归调用 scanExp 函数。

```
scanExp :: String -> [LitOp]
scanExp [] = error "Expected an expression"
scanExp (' ':ts) = scanExp ts
--scan prefix unary operator
scanExp ('l':'o':'g':ts) = Right Log : scanExp ts
scanExp ('s':'i':'n':ts) = Right Sin : scanExp ts
scanExp ('c':'o':'s':ts) = Right Cos : scanExp ts
scanExp ('s':'q':'r':t:ts) = Right Sqrt : scanExp ts
scanExp ('+':ts) = Right Posi : scanExp ts
scanExp ('-':ts) = Right Nega : scanExp ts
scanExp ('(':ts) = Right L_Par : scanExp ts
```

一个数学表达式的开头若不是一元前置运算符，也不是左括号，则必为一个数值，所以有：

```
scanExp xs = scanNum xs
```

数值后必为一个二元运算符或者右括号，所以，在解析数值后需要解析二元运算符与右括号。

```
scanNum :: String -> [LitOp]
scanNum ('e':ts) = Left (Const "e") : scanBin ts
scanNum ('p':'i':ts) = Left (Const "pi") : scanBin ts
scanNum xs | null num = error "Expected a number or constant"
| otherwise = case rest of
 ('.': r) -> let (float,r') = span isDigit r in
 Left (Val (read (num ++ "." ++ float) :: Float)) : scanBin r'
 r -> Left (Val (read num :: Float)) : scanBin r
 where (num, rest) = span isDigit xs
```

这里，将右括号归于二元运算符中，若解析的为一个二元运算符，则运算符后跟的是表达式，所以应该调用 scanExp 函数。如果是一个右括号，那么括号右边还应该为一个二元运算符或者还是右括号，所以要递归地调用 scanBin 函数，如果读者学习过上下文无关文法则会有更深入的了解。

```
scanBin :: String -> [LitOp]
scanBin [] = []
scanBin (' ':ts) = scanBin ts
scanBin ('+':ts) = Right Plus: scanExp ts
scanBin ('-':ts) = Right Minus: scanExp ts
scanBin ('*':ts) = Right Mult: scanExp ts
scanBin ('/':ts) = Right Divi: scanExp ts
scanBin ('^':ts) = Right Power: scanExp ts
scanBin (')':ts) = Right R_Par: scanBin ts
scanBin _ = error "Expected an infix binary operator"
```

从上面的代码可以看到，将字符串分析成记号的过程是多个函数通过间接递归实现的。表达式的语法也正是间接递归。

现在，有了一个分析记号的函数 `scanExp :: String -> [LitOp]`，还有一个计算函数 `calc :: [LitOp] -> State Stack LitOp`。最后，只需要再写一个函数 `calculate :: State Stack LitOp -> Float`。通过复合 `evalState`、`calc`、`scanExp` 函数很容易做得到。

---

```
--Main.hs
module Main where
import Calculator (calc,inits,Lit (Val),LitOp)
import Scanner (scanExp)
import System.Environment (getArgs)
import Control.Monad.State (evalState)

cal :: String -> LitOp
cal exp = (evalState.calc.scanExp) exp inits

num :: LitOp -> Float
num (Left (Val a)) = a
num _ = error "input error"

calculate = num.cal
```

---

最后，只需要写一个 `main` 函数，从命令行得到参数，然后计算，打印结果即可。

---

```
main :: IO ()
main = do
 expr <- getArgs
 print $ calculate $ concat expr
```

---

写好 `main` 函数，这个计算器就可以编译使用了。写完这个程序可以看到，借助 Haskell 中的状态 Monad 可以很灵活地对栈进行操作。用模式匹配定义函数，使得程序的结构十分简明、准确和清晰，美观并且可读性很强。

#### 练习

1. 有一种数学表达式称为逆波兰式（Reverse Polish Notation），比如  $3\ 1\ -$ ，将减号放在二元运算符的后面表示为  $3-1$ ，再如  $3\ 1\ -\ 2\ +$ ，表示为  $3-1+2$ 。这样的表达式使用计算过程十分方便，因为只需要一个栈就可以实现了。例如，计算  $3\ 1\ -\ 2\ +$  时，只需要将  $3$  入栈， $1$  入栈，然后读到减号时，将栈中的  $3$  与  $1$  作差得  $2$ ，再压  $2$  入栈。当读到加号时，将栈中的两个数作和得  $4$ 。定义自己想要的运算符，用一个栈来实现可以计算形式为逆波兰式的算术表达式的栈计算器。
2. 在定义了一个简单的字符识别器后，其实它就是一个简单的语法分析器了。试着定义 `Parser [Either Lit Op]` 类型的函数。其实，当我们在下一章中学习了语法分析器 Monad 时，这种任务会变得相当容易。

## 12.4 随机数的生成

本节中，简单讨论一下生成随机数（random number）。因为随机函数不可能为纯函数，并且每次随机数生成函数都会生成一个不同的值，所以这个函数是不可能具有引用透明性的。这种效应被 IO Monad 代表，因为随机生成的时候读取了计算机系统的时间。GHC 中生成随机数的库为 System.Random。

首先，库中有一个 RandomGen 类型类，称为随机生成器类型类，通过借助一种类型作为生成器来随机生成其他的类型。该类型类中定义了函数 next :: g -> (Int, g)，即返回一个 Int 值与一个新的生成器，这实际就是生成伪随机数的过程。相信学过了状态 Monad，读者应该很了解它了，它实质上是一个状态 Monad。当输入的生成器确定了，返回的结果也是确定的。

库中还定义了类型 StdGen，它通过用 Int 类型实现了 RandomGen 类型类来作为标准的生成器，其他类型值的生成可以借助这个标准生成器。StdGen 构造器为私有的，但是可以使用函数 mkStdGen :: Int -> StdGen 来得到一个标准生成器。

最后，库中还定义了 Random 类型类，即如何通过实现了 RandomGen 类型类的生成器来生成某一个类型的数据。例如，函数 random :: (RandomGen g, Random a) => g -> (a, g)，可以随机给定一个生成器的值 g 来生成一个值与新的生成器。其实，这正是状态 Monad 做的事情。

---

```
> (fst $ random (mkStdGen 2010)) :: Int
711767236
```

---

但是，如何随机地生成这个随机生成器？答案就是使用系统当前的时间。因为某一时刻的时间具有一定的随机性，而得到系统时间就需要 IO Monad，所以随机数的生成由 IO Monad 所代表。可以使用函数 newStdGen :: IO StdGen，得到一个新的生成器，这样就可以随机地生成需要的类型了。更为方便地是，库中定义了 getStdRandom :: (StdGen -> (a, StdGen)) -> IO a 可以直接随机生成一个值。

在 Random 类型类中，除 random 外，randomR 与 randomRs 可以在指定的范围内生成一个值和列表。

---

```
randomR :: RandomGen g -> (a, a) -> g -> (a, g)
randomRs :: RandomGen g -> (a, a) -> g -> [a]

> d <- getStdRandom (randomR (1, 6))
> d
6

> d <- getStdRandom (randomR (1, 6))
> d
4
```

---

比如，掷六次骰子的函数可以定义如下：

---

```
rollDice :: Int -> IO ()
rollDice n = do
 gen <- newStdGen
 print $ take n (randomRs ((1,6)::(Int,Int)) gen)

> rollDice 6
[4,4,3,1,2,6]
> rollDice 6
[2,5,4,3,6,3]
```

---

在随机生成各种类型的值时，需要指定结果的类型，否则将会因产生歧义而得到下面的错误：

---

```
Ambiguous type variable `a0' in the constraints:
 ...
 ...
```

---

## 本章小结

到这里，相信读者已经对 Monad 有了一定的了解。很多的类型都可以被抽象为 Monad，它的意义在于，对于一种结构只需要定义一些对应的操作（比如 `return`、`>>=`）就可以自动地管理计算过程中其他的效应，而使用者完全可以不必在乎发生了什么，只需要在乎其计算结果，而不必在乎伴随的计算是如何发生的，产生了什么样的效应。

学习 `IO` Monad 时，库中提供了很多函数，比如 `getLine`。定义的函数不必知道 `getLine` 函数是如何与操作系统交流的，只在乎返回的结果。又如，实现栈计算器时，定义了 `push`、`pop0`、`pop1`，在使用它们的过程中不会在乎这个栈是如何维护的，元素的进入与弹出全被自动地处理了。

以 `Reader`、`Writer`、`State` Monad 为类型的函数都是纯函数，所提供的操作在一定程度上可以理解为一种语法糖。使用记录器 Monad 的 `tell` 函数还有用状态 Monad 定义的对栈操作的函数 `push` 与 `pop` 时，从使用者的角度来看，我们不必知道发生了什么，对 Monad 内部列表的操作是一种副作用。有些人甚至将这些操作理解为通过 Monad 在 Haskell 中为操作不同各类副作用而实现的 DSL。比如，`push` 与 `pop` 就是为栈而定义的操作，可以理解为操作栈而定义的一种语言。有人说这两个函数在面向对象语言里很容易实现，也不需要借助 Monad，这样说是没错，但是在 Haskell 异常处理时所用的 `catch` 与 `finally` 等函数，借助 `IO` Monad 与高阶函数，它们是很容易实现的，在使用这些函数时也是非常直观与方便的。又如，将在第 16 章介绍的软件事务内存（Software Transactional Memory）Monad，它可以用来处理并发，其中提供了事务的原子、重试与选择操作，它们是 `atomically`、`retry` 与 `orelse`。同时，它们也可以被当做普通函数来处理，也相当于一个小型的控制并发程序的语言。

现在试想一下，要在一个没有异常处理机制的顺序式语言上加 `catch` 和 `finally`，再加入

软件事务内存的特性使得该语言可以更好地处理并发，并且同时还要简单易用，那么有可能需要对语言的语法进行改动，引入一些关键字，然后定义这些关键字的语义，再修改运行时系统，最后发布语言的新版。而读者将会看到，相比之下，由于 Haskell 为函数式语言，通过 Monad 加入一些新的特性并不需要这么麻烦，大部分需要做的只是发布新的库，所以 Haskell 在进化的道路上更具有优势。GHC 的设计者们对于新特性的加入也是非常小心的，新增的特性常常使用编译器参数来扩展，以保证对于旧版本的兼容，还有对标准语法的统一。Haskell 通过引入 Monad 把可能产生副作用的计算用一个类型来代表，让纯函数式编程语言可以用一些类型来代表伴有其他行为的计算，并且可以将所代表的计算行为隐含的中间状态用一种类型处理而不暴露给外界，这种特点还需要我们在学习后面的 Parser Monad 与 STM Monad 时细细品味。

## 第 13 章

# Monad 转换器

Monad 转换器（Monad transformer，其中，transformer 意为转换器、变形器或者转接器）意为转换了的 Monad，以便用来组合其他的 Monad 从而得到一个功能更为强大的 Monad。每一种特定的 Monad 都是伴随着特定行为的计算，当需要伴随着的行为不止一个时（比如，希望在栈操作的时候同时记录哪些元素入了栈，哪些元素出了栈），就需要组合 Writer Monad 与 State Monad。再比如，在操作栈的时候可能会出现栈中没有元素了，但如果继续做 pop 出栈操作就会出现 stack underflow 的异常，这实际上，这是由于列表为空而没有模式可匹配引起的，这个时候若希望处理这个异常，则可以用 Maybe 类型返回 Nothing 为结果，这个时候就需要组合 Maybe Monad 和 State Monad。本章就来探讨如何组合它们。

### 13.1 从 IdentityT Monad 转换器开始

首先，来看一下最简单的 Monad 转换器。Identity 的 Monad 转换器为 IdentityT，从它的定义可以看出，IdentityT 相当于内部又嵌套了一个 Monad m。由于 Identity Monad 不伴随着任何的其他行为，所以对于任意的 Monad m 与 IdentityT Monad 转换器组合所得到的 Monad 在功能上与 Monad m 并无差别。

```
import Control.Monad.State

newtype Identity a = Identity { runIdentity :: a }
newtype IdentityT m a = IdentityT { runIdentityT :: m a }
```

下面来看一下当类型 m 为一个 Monad 时，Identity m 如何实现为一个 Monad 的实例：

```
instance (Monad m) => Monad (IdentityT m) where
 return a = IdentityT $ return a
 m >>= k = IdentityT $ do
```

```
a <- runIdentityT m
runIdentityT (k a)
```

将 `IdentityT m` 实现为 `Monad` 的实例的过程中，可以看到 `IdentityT` 并没有任何的计算行为，而是直接将内部的值传递给函数 `k`。读者可以试着组合 `IdentityT` 与 `State` `Monad`，然后来定义对栈操作的函数。

```
type IState s a = IdentityT (State s) a

push :: Int -> IState [Int] ()
push x = IdentityT $ State $ \xs -> ((), x:xs)

pop :: IState [Int] Int
pop = IdentityT $ State $ \(x:xs) -> (x, xs)
```

可以看出，这里的 `push` 与 `pop` 函数同单独用 `State` `Monad` 定义的没有差别。相信到这里还记得类型同构的读者已经想到了，这里的 `Identity (State s)` 和 `State` 功能相同，它们是同构的。

$$\text{IdentityT}(\text{State } s) \cong \text{State } s$$

定义 `f` 与 `g` 函数也非常容易，这里就不再赘述了。如果读者对于同构关系的定义生疏了，那么可以参阅第 8 章中 8.2 节。

下面，来讨论一下 `Maybe` 类型的 `Monad` 转换器。

```
data Maybe a = Nothing | Just a
data MaybeT m a = MaybeT {runMaybeT :: m (Maybe a)}

instance Monad m => Monad (MaybeT m) where
 return x = MaybeT $ return (Just x)
 MaybeT a >>= f = MaybeT $ do
 result <- a
 case result of
 Nothing -> return Nothing
 Just x -> runMaybeT (f x)
```

试着将 `MaybeT` 同 `Identity` `Monad` 组合起来，可以写一个 `safeHead` 函数来得到一个列表的第一个元素：

```
safeHead :: [a] -> MaybeT Identity a
safeHead [] = MaybeT $ Identity Nothing
safeHead (x:xs) = MaybeT $ Identity $ Just x
```

很明显，它和仅用 `Maybe` 定义的 `safeHead` 的功能是一样的，所以 `MaybeT Identity` 与 `Maybe` 是同构的。

$$\text{MaybeT Identity} \cong \text{Maybe}$$

事实上，对于任意的 `Monad m` 及其对应的转换器 `mT`，可以得到：

$$\text{IdentityT } m \cong mT \text{ Identity} \cong m$$

Writer Monad、Reader Monad 与 State Monad 都有其对应的转换器，即 WriterT、ReaderT 和 StateT，下面来看一下状态 Monad 转换器：

---

```
newtype StateT s m a = StateT { runStateT :: s -> m (a, s) }

instance (Monad m) => Monad (StateT s m) where
 return a = StateT $ \s -> return (a, s)
 m >>= k = StateT $ \s -> do
 (a, s') <- (runStateT m) s
 runStateT (k a) s'
```

---

之前的 Identity 与 State Monad 的组合 IState s a 与 State s a 是同构的，它们之间的同构关系很容易验证，因为两个转换的函数 f 与函数 g 十分容易定义。比如，函数地 f :: IState s a -> State s a 的定义如下：

---

```
f :: IState s a -> State s a
f (IdentityT ms) = state $ runState ms
```

---

显然，StateT Monad 转换器与 Identity Monad 的组合也是与 state 同构的。事实上，库中的 State Monad 就是通过组合 StateT 与 Identity Monad 来定义的，Reader Monad 与 Writer Monad 都是如此。

---

```
> :m +Control.Monad.State
> :i State
type State s = StateT s Data.Functor.Identity.Identity
```

---

同样地，也可以实现栈操作的两个函数。

---

```
type State s a = StateT s Identity a

push :: Int -> State [Int] ()
push x = StateT $ \xs -> Identity ((), x:xs)

pop :: State [Int] ()
pop = StateT $ \(x:xs) -> Identity (x,xs)
```

---

现在，读者应该意识到，Identity Monad 可以用来占位与其他 Monad 转换器组合来定义该转换器对应的 Monad，而一个 Monad 转换器与 IdentityT 组合得到的结果还是一个 Monad 转换器，并且得到的结果与该转换器本身同构。

---

```
newtype IdentityT m a = IdentityT { runIdentityT :: m a }

> :k IdentityT (StateT Int Identity)
IdentityT (StateT Int Identity) :: * -> *

> :k StateT Int (IdentityT Identity)
StateT Int (IdentityT Identity) :: * -> *
```

---

组合的 Monad 转换器可以任意多，比如：

---

```
> :k StateT Int (IdentityT (IdentityT Identity))
```

---

前面的例子都有 `Identity` 和 `IdentityT` 参与，所以不增加 `Monad` 的功能。接下来，组合 `State` `Monad` 与 `Maybe` `Monad`，使得在栈操作的时候可以处理栈的列表为空但继续出栈时导致的下溢出（underflow）异常。

---

```
import Control.Monad.State
import Control.Monad.Trans.Maybe
```

---

首先，用 `StateT` `Monad` 转换器来组合 `Maybe` `Monad`：

---

```
pushSM :: Int -> StateT [Int] Maybe ()
pushSM x = StateT $ \xs -> Just (((),x:xs))

popSM :: StateT [Int] Maybe Int
popSM = StateT $ \xs -> case xs of
 [] -> Nothing
 (x:xs) -> Just (x,xs)
```

---

这里看到，如果这样定义，得到的结果类型是 `Maybe (a,s)`。如果栈下溢了，那么代表栈状态的状态 `Monad` 效应也连同消失了。而使用 `MaybeT` 来组合状态 `Monad`，则不会出现这种问题：

---

```
pushMS :: Int -> MaybeT (State [Int]) ()
pushMS x = MaybeT $ state $ \xs -> (Just (),x:xs)

popMS :: MaybeT (State [Int]) Int
popMS = MaybeT $ state $ \xs -> case xs of
 [] -> (Nothing, xs)
 (y:ys) -> (Just y, ys)
```

---

由于这样组合得到的结果类型为 `(Maybe a,s)`，因此，如果栈为空返回值为 `Nothing`，就不会导致状态 `Monad` 效应的消失。

## 13.2 Monad转换器组合与复合 Monad 的区别

只是需要管理状态与基于某一类型计算的失败时，其实只要复合 `State` 与 `Maybe` 就可以了，即用 `State s (Maybe a)`。在上一章读取器 `Monad` 的例子中也用到了 `Reader Env (Maybe Exp)` 类型，可能那时读者就已经感觉到连续地对 `Maybe` 类型进行模式匹配是一件很麻烦的事情，这里再简单地举一个例子：

---

```
push :: Int -> State [Int] ()
push x = state $ \xs -> (((),x:xs))

pop :: State [Int] (Maybe Int)
pop = state $ \xs -> case xs of
```

---

---

```

[] -> (Nothing,[])
(x:xs) -> (Just x, xs)

stack :: State [Int] ()
stack = do
 push 5
 pop
 pop
 push 4

> runState st []
((),[4])

```

---

但是, `pop :: State [Int] (Maybe Int)` 类型返回的是一个 `Maybe Int` 类型, 因此, 如果需要对 `Int` 进行计算, 那么在用 `do` 关键字定义函数时, 就需要对返回的 `Maybe Int` 值进行模式匹配。这样就会显得十分麻烦。而用 `Monad` 转换器定义的 `pop :: MaybeT (State [Int]) Int` 会直接返回 `Int` 类型, 不需要再进行模式匹配。

---

```

stack1 :: State [Int] (Maybe Int)
stack1 = do
 push 5
 a <- pop
 case a of
 Nothing -> return Nothing
 Just a -> return (Just (a+1))

stackMS1 :: MaybeT (State [Int]) Int
stackMS1 = do
 pushMS 5
 i <- popMS
 return (i+1)

```

---

当使用 `State [Int] (Maybe Int)` 复合多个 `Monad` 类型时, 它仅仅是一个状态 `Monad`。`>>=` 只会对状态 `Monad` 进行连续操作, 而 `Maybe` 类型则要另外处理。但是, 使用 `Monad` 转换器则是将多个 `Monad` 的效应组合成了一个统一的, 同时处理多个 `Monad` 效应的、更强大的 `Monad`。`>>=` 运算符会将这些组合的 `Monad` 的效应一次处理好。

此外, 当需要同时用很多个 `Monad` 时, 这些 `Monad` 还有着另一层的抽象, 因为它们组合的 `Monad` 可以是 `Writer` 与 `State` 等, 所以可以被抽象成为 `MonadWriter`、`MonadState` 等类型类, 对应的 `Monad` 转换器类型也实现了它们。这样使用 `Monad` 转换器来组合要更为方便。比如, 库中的 `StateT` `Monad` 转换器内部的 `Monad` 实现了 `MonadReader` 时, `StateT` `Monad` 也可以实现 `MonadReader` 与 `MonadWriter` 类型类:

---

```

instance (MonadReader r m) -> MonadReader r (StateT s m) where
 ask = lift ask
 local f m = StateT \$ \s -> local f (runStateT m s)

```

---

同理，对于 MonadReader 也是如此。这样，在组合时 StateT 时，也可以使用 MonadReader 提供的函数，十分方便。

---

```
instance (MonadWriter w m) => MonadWriter w (StateT s m) where
 tell = lift . tell
 listen m = StateT $ \s -> do
 (a, s'), w <- listen (runStateT m s)
 return ((a, w), s')
 pass m = StateT $ \s -> pass $ do
 (a, f), s' <- runStateT m s
 return ((a, s'), f)
```

---

正因为有了它们，使用状态 Monad 转换器后，如果要对其内部组合的记录器 Monad 或者读取器 Monad 操作，就不用嵌套多层表达式来处理内部的 Monad 了。

其次，Monad 转换器不仅仅是简单地复合两个 Monad。在 Monad 转换器定义时，它会严格地限定它们组合的方式。比如，状态 Monad 与记录器 Monad 可以简写为：

---

```
type St s a = s -> (a, s)
type Wt w a = (a, w)
```

---

如果只是简单地复合它们，即  $St\ s\ (Wt\ w\ a)$  得到  $a -> ((a, w), s)$ ，而复合  $Wt\ w\ (St\ s\ a)$  会得到  $(s -> (a, s), w)$ 。忽略 newtype 定义类型的构造器，WriterT 的定义是  $m\ (a, w)$ ，StateT 的定义为  $s -> m\ (a, s)$ 。

---

```
newtype WriterT w m a = WriterT { runWriterT :: m (a, w) }
newtype StateT s m a = StateT { runStateT :: s -> m (a, s) }
```

---

对于 WriterT，如果组合的 Monad 为状态，Monad 就得到了  $State\ s\ (a, w)$ ，也就是  $s -> ((a, w), s)$ ，即与  $St\ s\ (Wt\ w\ a)$  相同。反过来，用 StateT s 与 Writer w 组合时，得到  $s -> Writer\ w\ (a, s)$ ，也就得到了  $s -> ((a, s), w)$ 。显然，它与  $Wt\ w\ (St\ s\ a) = (s -> (a, s), w)$  的类型是不同的。

这里也可以看到，在使用 Monad 转换器组合 Monad 的情况下，当有多个 Monad 转换器时，组合它们，类型就像是一个栈一样。这里组合的  $WriterT\ w\ (State\ s)$  等价于复合的  $St\ s\ (Wt\ w\ a)$ ，正好反了过来。那么，观察可得知， $StateT\ s\ (WriterT\ w\ Maybe)\ a$  组合了 3 个 Monad 的类型，其中 Maybe 在最右侧，然后是 Writer，最后在最左侧的是 State。也就是说，State 效应在最里边与 a 类型在一起。如果运行这个 Monad，给定了一个输入的状态，得到的结果类型将会是  $Maybe\ ((a, s), w)$ 。

### 13.3 Monad 转换器的组合顺序

再重新回到组合的状态 Monad 与 Maybe Monad。可以比较一下在组合 MaybeT 与 State s

和 StateT s 与 Maybe 时的类型，Maybe 与 State Monad 及其转换器类型如表 13-1 所示。

表 13-1 state Monad 与 Maybe Monad 及其对应转换器

| 原 Monad | 构造器内类型      | 对应 Monad 转换器 | 转换器构造器内类型     |
|---------|-------------|--------------|---------------|
| State s | s -> (a, s) | StateT s     | s -> m (a, s) |
| Maybe   | a           | MaybeT .     | m (Maybe a)   |

通过表格可知，当组合 StateT s 与 Maybe 时，会得到类型 s -> Maybe (a, s) 类型。显然，当基于类型 a 的计算失败时，整个元组的计算都会失败。这样，Maybe Monad 会使状态 Monad 的行为一起消失。而当组合 MaybeT 与 State s 时，会得到类型 s -> (Maybe a, s) 的类型，所以，当基于类型 a 的计算失败时，不会引起整个元组的计算失败。

同样，正如前面所提到的 Writer Monad 也是有其对应的 Monad 转换器的，它定义为：

```
newtype WriterT w m a = WriterT { runWriterT :: m (a, w) }
instance (Monoid w, Monad m) -> Monad (WriterT w m) where
 return a = WriterT $ return (a, mempty)
 m >>= k = WriterT $ do
 (a, w) <- runWriterT m
 (b, w') <- runWriterT (k a)
 return (b, w `mappend` w')
```

现在，可以使用 WriterT String 与 State [Int] 组合来记录栈操作的过程。很显然，当 WriterT String m a 内部的 Monad 为 State [Int] 时，组合成的类型可近似为 State [Int] (a, String)，最终，State 构造器内的类型为 [Int] -> ((a, String), [Int])。

```
import Control.Monad.State
import Control.Monad.Writer

pushWS :: Int -> WriterT String (State [Int]) ()
pushWS x = WriterT $ state $ \xs -> ((((), " push "++ show x), x:xs))

popWS :: WriterT String (State [Int]) Int
popWS = WriterT $ state $ \(x:xs) -> ((x, " pop "++ show x), xs)
```

那么，反过来组合又会怎样？组合 StateT s 与 WriterT w 成的 Monad，在功能上会与 WriterT w 与 State s 组合的 Monad 有所不同么？可以来试一下，首先应当明确 StateT 构造器内的类型，StateT [Int] (Writer String) a = [Int] -> Writer String (a, [Int])，即 [Int] -> ((a, [Int]), String)，那么对应的 push 与 pop 函数可以定义为：

```
push :: Int -> StateT [Int] (Writer String) ()
push x = StateT $ \xs -> writer ((((), xs), " push "++ show x))

pop :: StateT [Int] (Writer String) Int
pop = StateT $ \(x:xs) -> writer ((x, xs), " pop "++ show x)
```

事实证明，这两种组合在功能上是没有任何差别的。可以仔细看一下它们为 Monad 时的类型和对应的转换器类型，如表 13-2 所示。

表 13-2 状态 Monad 与记录器 Monad 及对应转换器

| 原 Monad  | 构造器内类型     | 对应转换器     | 转换器构造器内类型    |
|----------|------------|-----------|--------------|
| State s  | s -> (a,s) | StateT s  | s -> m (a,s) |
| Writer w | (a,w)      | WriterT w | m (a,w)      |

构造器只是相当于一个类型的标记，可暂时将它忽略。那么，当 WriterT w 与 State s 组合时，WriterT 转换器构造器内的类型等价于  $s \rightarrow ((a,w), s)$ ，而当 StateT s 与 Writer w 组合时，StateT 转换器构造器内部的类型等价于  $s \rightarrow ((a,s), w)$ ，相信读者早已经发现，这两个类型是同构的，即：

$$s \rightarrow ((a,w), s) = s \rightarrow ((a,s), w)$$

所以有：

$$\text{StateT S(Writer W)A} \cong \text{WriterT W(State S)A}$$

显然，它们与类型  $s \rightarrow (a,s,w)$  也是同构的。那么，可以用这个类型来定义一个 Monad，它同 State Monad 与 Writer Monad 组合是等价的：

```
newtype WS s w a = WS {runWS::s -> (a, s ,w)}
instance Monoid w => Monad (WS s w) where
 return a = WS $ \s -> (a,s,mempty)
 k >>= f = WS $ \s -> let (a, s',m) = runWS k s in
 let (r,ns,m') = runWS (f a) s' in
 (r,ns,m `mappend` m')
```

为什么在组合 MaybeT 与 State s 时没有这一交换的性质呢？因为 Maybe 类型的定义与 Writer 不同。Maybe 类型是一个通过多个构造器（或者称为多模式）定义的类型，即 Nothing | Just a。而 Writer 的定义为 Writer (a,w)，是由参数只有一个的单一构造器定义的类型。状态 Monad 的情形与记录器 Monad 的情形是一样的。像 Maybe、Either 这样由多形式构造器定义的类型，在参与 Monad 组合时，Nothing 零元构造器会使组合它的 Monad 转换器的行为消失。同理，Either e Monad 也是由多构造器定义。由于组合的顺序不同，使用 Monad m 对应的转换器 mT 与 Either 的组合跟使用 EitherT 与 m 组合得到的 Monad 在功能上也会有所不同。

在这里，笔者认为如果组合的多个 Monad 均通过仅单一构造器定义，如 State、Writer、Reader 等，那么在使用相对应的转换器组合时，它们之间组合的顺序可以交换而不影响组合后 Monad 的功能，但是我目前还没有找到严格的证明或者相关的内容，也可能存在特例。但如果这种观点是正确的，那么对于满足这样条件的 Monad  $m_1$  和对应的转换器  $m_1T$ ，以及另一个这满足这样条件的 Monad  $m_2$  和对应的转换器  $m_2T$  有：

$$m_1T m_2 = m_2T m_1$$

当组合的 Monad 中有通过多构造器定义的类型参与，例如 Maybe 及对应转换器 MaybeT、

Either 及对应的转换器 EitherT 等类型，那么则可能不符合这种交换的规则。

Monad 的组合像洋葱一样，是一层一层的。如果需要，就可以一直组合下去，使得每一层都会让计算伴有不同的行为。也有人将这种组合比作栈，称为 Monad 栈。也是因为这种组合是有顺序的，由内到外的。无论将它比作什么，这种 Monad 的组合最终形成了一个非常独立统一，并且功能强大的 Monad。

## 13.4 lift 与 liftIO

Monad 转换器可以组合一个原始的 Monad，那么，这个原始的 Monad 总是可以转换为与某个 Monad 转换器的组合。这种性质被抽象为了一个类型类 MonadTrans，其中，定义的函数可以称为“提升函数”(lift)。

---

```
class MonadTrans t where
 lift :: Monad m => m a -> t m a
```

---

例如，态 Monad 转换器实现 MonadTrans 类型类定义为

---

```
instance MonadTrans (StateT s) where
 lift m = StateT \$ \s -> do
 a <- m
 return (a, s)
```

---

当原始的 Monad 类型为 State [Int] ()，这样，将这个操作加上 Maybe 的效应，需要定义函数 pushMS:: Int -> MaybeT (State [Int]) ()，它就可以通过 lift 转换状态 Monad 来直接定义：

---

```
import Control.Monad.State
import Control.Monad.Trans.Maybe

push :: Int -> State [Int] ()
push x = state \$ \xs -> (((),x):xs)

pushMS :: Int -> MaybeT (State [Int]) ()
pushMS x = lift \$ push x
```

---

这样，已有的 Monad 则可以十分方便地转换为需要组合 Monad 类型。用户只需要分别定义对应的 Monad 类型的值，然后使用 lift 函数将其提升为对应的组合后的 Monad 的类型即可。

如果一个 Monad 转换器组合的 Monad 有 IO 行为，那么 IO Monad 一定是在组合多个 Monad 最里边。如果 Monad 中的效应有 IO 参与，就可以直接使用 liftIO。由于 IO Monad 没有对应的 Monad 转换器，因此在定义多个 Monad 组合时，IO 一定在组合 Monad 类型的最右侧。MonadIO 是这样定义的：

---

```
class (Monad m) -> MonadIO m where
 liftIO :: IO a -> m a
```

---

也就是说，对于任意的 Monad  $m$  可以伴随着 IO 操作。这里可以给一个例子来组合 Maybe、Writer 与 IO。从键盘读取满足安全需要的密码，即长度大于 8 个字母，同时有大写字母、小写字母和数字：

---

```
isPasswordValid :: String -> Bool
isPasswordValid s = length s >= 8 && check s
 where check :: String -> Bool
 check s = and [f s | f <- map any [isUpper, isLower, isNumber]]
```

---

然后，使用 Maybe 来处理不满足条件的密码，如果密码满足条件，将其用一个记录器 Monad 记录下来。

---

```
setPassword :: MaybeT (WriterT (String) IO) ()
setPassword = do
 liftIO $ putStrLn "Please set a password"
 pass <- liftIO $ getLine
 guard (isPasswordValid pass)
 tell [pass]

> runWriterT $ runMaybeT setPassword
Please set a password
abcdef
(Nothing, [])

> runWriterT $ runMaybeT setPassword
Please set a password
abcDEF123
(Just (), ["abcDEF123"])
```

---

这样，在 IO 操作的同时，同时可以得到密码的设置是否成功，如果成功则会自动地记录下来，如果不成功则不会记录。

## 13.5 简易 Monad 编译器

本节里，将实现一个最简单的通用编程语言（General-purpose Programming Language, GPL）的语法树，然后编译成一个虚拟机能运行的指令。本节主要不是讨论编译原理，因此实现的过程非常的粗糙与简略，旨在让大家学会使用 Monad 转换器。但是，如果有一些编译原理的知识就会更好。

基于栈的虚拟机其实相当容易，只要根据相应的指令对栈进行相应的操作即可。比如，对于之前定义定的四则运算表达式，如果将它们译成一种基于栈操作的指令，那么可以这样译：

---

```
data Exp = Val Double
 | Plus Exp Exp
 | Sub Exp Exp
 | Mult Exp Exp
 | Div Exp Exp deriving (Show, Eq)
```

---

比如，对于表达式 Plus (Val 5) (Mul (Val 3) (Val 2)) 就可以译成[PUSH 5, PUSH 3, PUSH 2, MUL, ADD]。先压 5，再压 3、2 入栈，得到[Val 2, Val 3, Val 5]，左侧为栈顶，然后将栈顶的两值取出，做乘法，然后将结果再压入栈中得[Val 6, Val 5]然后再做加法。如果引入 If、While 等语句，则只是需要在指令中加入 JUMP，以便在机器运行指令时可以跳转。先来实现语法树，再来实现机器的指令集与这个机器，最后定义一些函数将语法树译成机器代码。

首先我们需要下面的库：

---

```
import Data.List
import Control.Monad.Writer
import Control.Monad.State
```

---

为了简化问题，用字符来表示变量名（不用 String）。

---

```
type Name = Char
```

---

下面定义语言的表达式，它们可以是整数值、变量名或四则运算。

---

```
data Exp = Val Int | Var Name | App Op Exp Exp deriving Show
data Op = Add | Sub | Mul | Div deriving (Show, Eq)
```

---

有了之前计算器的例子，将表达式译为逆波兰的函数 comexp 定义如下：

---

```
comexp :: Expr -> Code
comexp (Val int) = [PUSH int]
comexp (Var name) = [PUSHV name]
comexp (App op e1 e2) = comexp e1 ++ comexp e2 ++ [DO op]
```

---

接下来就要定义程序语言的语法树了。它只能做四件事情：第一个是将一个表达式的结果赋值给一个变量名。第二个就是条件，也很简单，判断一个表达式是否为非 0，如果为非 0 则运行 If 中定义第一个程序，否则运行第二个。第三个为循环，也就是说，如果 While 中的 Exp 为非 0，那么一直运行给定的程序；否则结束循环。最后一个为顺序，即将多个程序连接在一起。

---

```
data Prog = Assign Name Exp
 | If Exp Prog Prog
 | While Exp Prog
 | Seqn [Prog]
deriving Show
```

---

这样，语法树部分就实现完了。比如，来用这个语言定义一个计算阶乘的程序。

---

```
factorial :: Int -> Prog
factorial n = Seqn [Assign 'A' (Val 1),
 Assign 'B' (Val n),
 While (Var 'B') (
 Seqn [Assign 'A' (App Mul (Var 'A') (Var 'B')),
 Assign 'B' (App Sub (Var 'B') (Val 1))])]
```

---

给定 n 的值就可以计算 n 的阶乘了。其实用伪码写起来非常容易：

---

```
a := 1
b := readInt();
while (b) {
 a = a * b;
 b = b - 1;
}
```

---

a 的值就为 readInt() 用户给定的数字 n 的阶乘。

接下来实现机器指令。首先，定义一下栈，因为需要操作一个栈，所以在这里它就是一个整数的列表。

---

```
type Stack = [Int]
```

---

代码 Code 定义为一系列的指令，可以用列表来表示。机器的指令定义如下：

---

```
type Code = [Inst]
data Inst = PUSH Int --将一个整数压入栈中
 | PUSHV Name --将一个变量压入栈中
 | POP Name --将一个栈顶的值出栈，然后以给定的名字存储到存储器中
 | DO Op --做指定的 Op 二元运算
 | JUMP Label --无条件跳转到指令中的某一标签
 | JUMPAZ Label --若栈顶为 0，则跳转到指令中某一标签
 | LABEL Label --为跳转而定义的标签
deriving (Show)
```

---

上述内容就是机器的全部指令了，它非常简单。

下面来定义编译器。首先，这个编译器需要做两件事情。

- (1) 保持一个状态，可以不断地生成不同的标签值以便程序跳转，也就是状态 Monad。
- (2) 需要记录生成的机器指令，这是记录器 Monad 的功能。

由此可以知道，需要将这两个 Monad 用 Monad 转换器组合起来，定义为 WT a。

---

```
type WT a = WriterT Code (State Int) a
```

---

定义一个 fresh，可以返回一个新的整数作为跳转时的标签。

---

```
fresh :: WT Int
fresh = WriterT $ state (\s -> ((s, mempty), s+1))
```

---

下面，可以用这个 Monad 来定义一个函数来编译程序了。

---

```
mlabel :: Prog -> WriterT Code (State Int) ()
```

---

首先，将一个表达式赋给一个变量，相当于先对表达式求值，然后再将其出栈存入存储器中：

---

```
mlabel (Assign name expr) = do tell $ comexp expr
 tell [POP name]
```

---

编译条件时，先计算表达式。如果栈顶为 0，即为假，需要跳到第二个分支 n；如果不为 0，则为真，那么可以运行 prog1 后跳到末尾的标签 m。

---

```
mlabel (If expr prog1 prog2) = do n <- fresh
 tell $ comexp expr
 tell [JUMPZ n]
 mlabel prog1
 tell [JUMP m]
 tell [LABEL n]
 m <- fresh
 mlabel prog2
 tell [LABEL m]
```

---

循环与条件其实差不多，先对循环的首部加一个标签 n，然后计算条件表达式。如果为假，则说明条件不满足了，那么跳至循环程序的末尾处标签 m；否则运行 prog 循环体，然后跳到循环程序的开始，再查检条件是否满足。

---

```
mlabel (While expr prog) = do n <- fresh
 m <- fresh
 tell [LABEL n]
 tell $ comexp expr
 tell [JUMPZ m]
 mlabel prog
 tell [JUMP n]
 tell [LABEL m]
```

---

如果是多个程序编在一起，那么也很好编译，只需要递归地使用 mlabel 函数就可以了。

---

```
mlabel (Seqn []) = do tell []
mlabel (Seqn (c:cs)) = do mlabel c
 mlabel' (Seqn cs)
```

---

最后，定义函数来将一段“程序”编译为机器代码。

---

```
comp :: Prog -> Code
comp prog = snd $ fst $ (runState $ runWriterT $ mlabel' prog) 0
```

---

下面来测试一下这个编译器，编译 factorial 3 程序的机器代码结果如下：

---

```
> comp (factorial 3)
[PUSH 1, POP 'A', PUSH 3, POP 'B', LABEL 0, PUSHV 'B', JUMPZ 1, PUSHV 'A', PUSHV 'B', DO Mul, POP
'A', PUSHV 'B', PUSH 1, DO Sub, POP 'B', JUMP 0, LABEL 1]
```

---

来看一下这段程序是如何运行的。首先，[PUSH 1, POP 'A']将变量 A 初始化，这两条指令结束后，栈为空，存储器中有 {A=1}。[PUSH 3, POP 'B']后，栈还为空，而存储器中有 {A=1, B=3}，这样两个变量就被初始化了。然后 PUSHV 'B'，即 3 入栈，此时栈顶为 3，所以 JUMPZ 1 不会跳转，在判断 B 是否为 0 的时候，B 已经出栈了，所以此时栈又为空。[PUSHV 'A', PUSHV 'B', DO Mul, POP 'A']就非常好理解了，A 入栈，B 入栈，然后相乘，将结果放到

A 中。此时栈为空，存储器为 {A=3, B=3}。[PUSHV 'B', PUSH 1, DO Sub, POP 'B']则会将 B 减小 1，最后无条件地跳到标签 0，下一次则会将 2 乘到 A 上，B 会再减少 1。这样下去，当 B 中的变量为 0 时程序就结束了，此时，程序的存储器中 A 的值即为 3 的阶乘。

这个例子基于诺丁汉大学函数式编程的一次的作业，意在考查学生对 Monad 的理解。这里没有定义一个机器运行机器代码，有兴趣的读者可以试着定义一下虽然不是很难，但是很烦，这里可以给些提示，但读者也可以不根据提示来做：

先定义两个类型，一个是 Machine，另一个是程序计数器 Counter——记录运行到了哪条指令。

---

```
type Machine = (Code, Stack, Mem, Counter)
type Counter = Int
```

---

此外，还要定义 Mem。由于类型只有整数，因此没有布尔与字符等类型。另外，还需要一个存储器将变量名与值存储起来，以便可以读取它们，可以用列表定义一个简单的映射 [(Name, Int)]，也可以使用库中的 Map。

Machine 中的 4 个值其实就是机器的状态，正如前言中说的那样，机器做计算实际上就是对存储器的状态进行的一系列的改变。因此，给定一条指令可以将机器从一个状态计算到另一个状态，即：

---

```
execline :: Inst -> Machine -> Machine
```

---

其中，Counter 可以一直记录机器运行到指令中的位置，当跳转的时候也需要对 Counter 进行相应的修改。最后，定义一个函数 execute 来执行 Code 中的指令，不断地使用 execline 运行程序计数器的下一条指令，直到程序计数器到达指令的末尾。当然这只是两个最重要的函数，读者还需要定义其他函数来辅助。最后，返回存储器中的变量的状态即可。

#### 练习

使用 StateT Int (Writer Code) a 来实现 mlabel 函数，有兴趣的读者可以不用 Monad 转换器实现，看一下实现的代码会是什么样子。

## 13.6 语法分析器 Monad 组合子

在 9.2.5 节中，已经实现了一个简单的字符识别器。当处理字符串信息的时候经常需要将这一系列的字符串分析成想要的结构。这种结构通常是树状的，常常是上下文相关文法 (Context Sensitive Grammar)，比如下面左侧 HTML 的字符串，使用下面的这种类型定义分析成右侧这种树状结构。

---

```
data Node = Tag String [Node]
 | Text String
deriving (Show, Eq)

<html>
 <head>Hello world!</head>
 <body>Hello again!</body>
Tag "html" [
 Tag "head" [Text "Hello world!"],
 Tag "body" [Text "Hello again!"]]
```

---

```
</html> 1
```

另外一个例子是将数学表达式通过语法分析器分析成表达式树，或者分析成上一章的栈计算器中提到的“记号”(token)，进一步分析成为算术表达式树。这些任务其实都是可以通过Parser Monad 轻松实现的。之前的简单语法识别器只能分析上下文无关文法，不能分析上下文相关文法。比如这里的 XML 文件，需要匹配开始的 tag <html>与结束的 tag </html>。但是，如果需要的话，Monad 语法分析器则可以分析上下文相关文法。

本节将讨论 Parser Monad 大体上是如何实现的，再了解 GHC 带的 Parsec 库，到时候很自然地就会知道它应当如何使用了，只不过多了很多额外的功能。

### 13.6.1 简易语法分析器的实现

首先，语法分析时需要记录当前的位置、所在行数与列数，此外可能还有一些可能输出信息、警告或者错误。

一个字符串分析器可以定义为下面的类型：

---

```
newtype Parser a = Parser (String -> Int -> Int -> ParseResult a)
```

---

类型参数 a 为分析器返回的类型，其中的构造器中函数的参数分别为分析的字符串、行、列。最后返回一个结果为 ParseResult a 类型的值，这个类型包括余下的字符串，更新的当前的位置，还有一些信息、警告或者错误，一旦出现问题，就会知道为什么分析器的分析会失败。这里，用一种更为清晰的方式来定义语法分析器。

此时，会需要下面的库：

---

```
import Control.Applicative
import Data.Char
```

---

这里，将分析的位置定义为一个新的类型——Pos 来追踪分析的位置。

---

```
type Line = Int
type Column = Int

data Pos = Pos { getLine :: Line ,
 getColumn :: Column } deriving (Eq, Show)
```

---

在处理字符的时候，每分析一个字符，都要更改当前的位置。如果为换行符，则行数加 1；如果是 tab，那么移动到最近可以被 8 整除的位置；如果是其他字符，那么列数加 1。

---

```
updatePos :: Pos -> Char -> Pos
updatePos (Pos l c) char = case char of
 '\n' -> Pos (l+1) 1
 '\t' -> Pos l ((c+8-(c-1) `mod` 8))
 _ -> Pos l (c+1)
```

---

再来定义一个初始化位置的函数，对字符串的语法分析都是从(1,1)开始的。

---

```
initialPos :: Pos
initialPos = Pos 1 1
```

---

下面定义一个状态类型，这里想要定义一个更为一般的语法分析器，所以输入不一定非是字符串，有可能是其他类型。比如，计算器中的[Either Lit Op]也可以作为分析的类型，最后将它分析为算术表达式的语法规树，即 Exp。在这个类型里还可以包含输入和所在的位置。

---

```
data State s = State { stateInput :: s , statePos :: Pos } deriving (Eq, Show)
```

---

这里，分析语法时采用的是 LL(1)，在分析时向后看一个记号，可以不向后看这个记号，直接返回结果，即 Empty；也可以使用这个记号，返回一个结果，即 Consumed。定义这两种类型，对于分析的结果加以区分。这种分析的方法称为 Consumer-based 分析法。

---

```
data Consumed a = Consumed a
| Empty a
deriving (Eq, Show)
```

---

对于返回的结果，定义一个类型 Reply 可代表，如果分析成功，那么返回值 ok，包括成功的分析结果与当前的状态，还有一些诊断信息；如果为错误，那么返回 Error 值，并且将错误信息加到诊断信息类型当中。

---

```
data Reply s a = Ok a (State s) ParseError | Error ParseError deriving (Eq, Show)
```

---

下面就是定义信息类型了，根据不同问题的严重性，信息也不同。如果分析成功，但是想让分析器加一些信息，则可以使用 Info。有时问题相对严重，但还不能导致分析失败，可以加一个警告 Warn。最后，如果计算失败，则需要错误 Err。

---

```
data Message = Info String | Warn String | Err String deriving (Eq, Show)
```

---

再来定义语法分析过程中可能出现的信息类型，一个工具函数可以帮助用户对分析信息进行操作。

---

```
data ParseError = ParseError [Message] deriving (Eq, Show)

appendError :: ParseError -> Message -> ParseError
appendError (ParseError a) msg = ParseError (msg:a)
```

---

最后就可以定义分析器类型了。首先，里面的函数的第一个参数为状态类型，当前的位置和分析的内容都在里面。第二个参数是分析的信息，最后返回一个 Consumed (Reply s a) 为结果，因为分析器可能没有用到输入信息也就对应 Empty，也有可能用了一些对应 Consumed。

---

```
data Parser s a =
Parser { runParser :: State s -> ParseError -> Consumed (Reply s a) }
```

---

下面实现函数类型类，一对地写下来，定义起来还是非常容易的。

---

```
instance Functor (Parser s) where
fmap f p = Parser $ \st error -> case runParser p st error of
```

---

---

```

Consumed (Ok r st' err) -> Consumed (Ok (f r) st' err)
Consumed (Error err) -> Consumed (Error err)
Empty (Ok r st' err) -> Empty (Ok (f r) st' err)
Empty (Error err) -> Empty (Error err)

```

---

再来实现 Monad 类型类，同函数一样，也可以一对一地写出。

---

```

instance Monad (Parser s) where
 return inp = Parser $ \st error -> Empty (Ok inp st error)
 p >>= f = Parser $ \st error -> case runParser p st error of
 Consumed (Ok r st' err) -> runParser (f r) st' err
 Consumed (Error err) -> Consumed (Error err)
 Empty (Ok r st' err) -> runParser (f r) st' err
 Empty (Error err) -> Empty (Error err)

```

---

前面学习过了，通过 Monad 类型类，可以很容易地定义 Applicative 类型类：

---

```

instance Applicative (Parser s) where
 pure = return
 (*)<*> pf pa = do
 f <- pf
 m <- pa
 return $ f m

```

---

最后来定义 Alternative 类型类，empty 如同 Maybe 中的 Nothing 一样。在定义`<|>`运算时，如果不需要任何的输入就可以分析成功，那么再来看第二个分析器的运行情况，如果第二个分析器需要一个值，那么这个就使用第二个分析器，这里总是贪婪地尽可能向右分析。

---

```

instance Alternative (Parser s) where
 empty = Parser $ \st err -> Empty (Error err)
 p <|> q = Parser $ \st err -> case runParser p st err of
 Empty (Error err') -> runParser q st err
 Empty o@(Ok r st' err') -> case runParser q st err of
 Empty _ -> Empty o
 consumed -> consumed
 consumed -> consumed

```

---

所有需要的类型类的实例已经定义好了，下面就可以定义一些应用函数，这样可以在使用分析器时更加方便。第一个就是`<?>`函数，如果分析失败，那么会将信息附加到结果上，否则没有作用。

---

```

(<?>) :: Parser s a -> Message -> Parser s a
(<?>) p msg = Parser $ \st err -> case runParser p st err of
 Empty (Error err') ->
 Empty (Error (appendError err' msg))
 Consumed (Error err') ->
 Consumed (Error (appendError err' msg))
 result -> result

```

---

下面则可以为字符串分析定义一些应用函数了。最重要的一个高阶函数就是 `satisfy`，也就是给定一个条件判定是否可以成功。

---

```

satisfy :: (Char -> Bool) -> Parser String Char
satisfy f = Parser $ \(State str pos) err ->
 case str of
 c:cs -> if f c
 then Consumed
 (Ok c (State cs (updatePos pos c)) err)
 else Empty
 (Error (ParseError [Err ("error at " ++
 show pos)]))
 [] -> Empty
 (Error (ParseError [Err ("error at " ++
 show pos ++
 " input exhausted.")]))

```

---

有了 `satisfy` 函数，定义其他的函数来分析字母、特定的某一字符与字符串就相当容易了。可以使用`<?>`运算符在后面加上一些错误信息。

---

```

char :: Char -> Parser String Char
char c = satisfy (==c) <?> Info ("expect a character " ++ show c)

letter :: Parser String Char
letter = satisfy isAlpha <?> Info "expect an alpha"

```

---

有了 `char` 函数，分析一个字符串的函数很容易定义。

---

```

string :: String -> Parser String String
string [] = return []
string (s:str) = do
 c <- char s
 cs <- string str
 return (c:cs)

```

---

定义一个函数 `parse` 来直接地得到字符串分析的结果。如果有错误，就使用 `error` 函数来抛出异常。

---

```

parse :: String -> Parser String a -> a
parse str p = case runParser p (State str initialPos) (ParseError []) of
 Consumed (Ok r st' err) -> r
 Consumed (Error err) -> error $ show err
 Empty (Ok r st err) -> r
 Empty (Error err) -> error $ show err

```

---

现在回到本节开始的地方，想分析一个简单的 XML，它的语义树定义如下：

---

```

data Node = Tag String [Node]
 | Text String deriving (Eq, Show)

```

---

分析 `Node` 很容易，只需要逐个分析字符，直到`<=`为止。将这个字符串转换成 `Text String` 这种值即可。可以使用 `fmap`，也可以使用 `do` 表达式。

---

```
textNode :: Parser String Node
textNode = fmap Text $ some $ satisfy (/='<')
```

---

再来分析 tag。首先要得到 tag 的名字，使用`>`与`<`忽略两侧的尖括号，使用`many letter`就可以得到尖括号内的字符串了。然后，要做的就是递归地分析 tag 内的部分。最后，检查末尾的 tag 与开始的 tag 是否一致，由于不必在意分析结果，因此使用`>>`忽略，这里使用`>`与`>>`其实是相同的。

---

```
tagNode :: Parser String Node
tagNode = do
 tagName <- char '<' *> many letter *> char '>'
 subNode <- many $ tagNode <!> textNode
 string "</>" >> string tagName >> char '>'
 return $ Tag tagName subNode

xml :: String
xml = "<html><head>Hello world!</head><body>Hello again!</body></html>"

parseTest :: Parser String Node
parseTest = parse xml tagNode

> parseTest
Tag "html" [Tag "head" [Text "Hello world!"], Tag "body" [Text "Hello again!"]]
```

---

当然，这个语法分析器是最简单的一个，有的时候根据不同需要，用户可以自己定制语法分析器，比如加入缓冲区，使用 LL(2) 来分析语法，这样就更为复杂了此处不再讨论。大多数情况下，Haskell 的 Parsec 库就可以满足用户的需要，下节主要介绍它。

本节的内容主要参阅了 Daan Leijen 与 Erik Meijer 发表的“Parsec: A practical parser library”，可以在 <http://research.microsoft.com/en-us/um/people/emeijer/papers/parsec.pdf> 中浏览。此外，还有 Daan Leijen 的“Parsec, a fast combinator parser”，可以在 <http://legacy.cs.uu.nl/daan/download/parsec/parsec.pdf> 中浏览。

#### 练习

上一章中实现了一个基于栈的计算器，有兴趣的读者可以使用本节所学的内容重新编写一个语法分析器来分析数学表达式，即定义一个类型为`Parser [Either Lit Op]`的语法分析器。

### 13.6.2 Parsec 库简介

Parsec 库是一个专门为语法分析而编写的库，它与之前实现的`Parser`大体上是相同的，但是它的实现会更为复杂一些，包括了更多的辅助函数，可以更为简便地对文件或者其他各类型的输入进行语法分析，并且可以提供更有效的错误信息。其中，不但包括了`Char`、`String`等库对字符与字符串进行分析，还有`Text.Parsec.Expr`可以用来对表达式进行分析，让用户

可以对有各种性质的运算符进行处理。此外，它还提供了 `Text.Parsec.Perm` 库来对无序的输入进行分析。有兴趣的读者可以简单浏览一下 `Parsec` 库。在这一小节中不会涵盖所有关于它的所有内容，只是简单了解一下如何使用它。

`Parsec` 库中定义了 `ParserT`，即对应的 `Monad` 转换器类型 `data ParsecT s u m a`。其中，`s` 为输入类型，通常为字符串类型；`u` 为用户状态，它提供了一种机制让分析器的使用者可以进行语法分析时保持一个状态，实质上与状态 `Monad` 是一样的，也就是说，`Parsec` `Monad` 本身含有一个状态 `Monad`。比如，有时候不但要解析字符串，还需要对字符串中的内容进行统计，比如，查询有多少个变量名或者某个人名出现了多少次，这些都是可以通过它来记录的。关于这个状态，一会儿再介绍它。定义中的 `m` 为组合的 `Monad` 类型，如果需要在解析时进行输出，那么需要组合 `IO` `Monad`，如果需要记录一些内容，那么需要组合记录器 `Monad`。最后的 `a` 为返回的类型。

很多情况不会用到 `ParsecT`，那么只需要组合一个 `Identity` `Monad` 即可。`type Parsec s u a = ParsecT s u Identity a`，类型的定义也是可以用  $\eta$  化简的，即 `type Parsec s u = ParsecT s u Identity`。这里的 `State` 定义中加入了 `stateUser`，也就是刚刚提过的状态 `Moand`。

---

```
data State s u = State {
 stateInput :: s,
 statePos :: SourcePos,
 stateUser :: u
}
```

---

下面来看一下 `Parsec` 具体是如何使用的。

---

```
import Text.Parsec
```

---

导入这个库以后，就可以使用 `satisfy`、`letter`、`char`、`digit` 函数了，还有一些比较常用的，比如 `tab`、`space`、`spaces` 等。例如，需要一个函数 `word`，可以分析一个英语单词，它可以定义为：

---

```
word :: Parsec String () String
word = manyl letter
```

---

可以使用 `parseTest :: Parsec s () a -> s -> IO ()` 函数来测试：

---

```
> parseTest word "Hello World"
"Hello"
```

---

这里并未指定任何用户状态，所以使用了 `()`，因为现在还不需要它。

有一对函数也是非常有用的，它们是 `oneOf` 与 `noneOf`，它们的定义是：

---

```
oneOf, noneOf :: [Char] -> ParsecT s u m Char
oneOf cs = satisfy (\c -> elem c cs)
noneOf cs = satisfy (\c -> not (elem c cs))
```

---

`oneOf`，顾名思义，就是当字符在给定的字符串里，分析都会成功，如果不在此，则分析会

失败，而 `noneOf` 与它刚好相反。

另外，除了 `many` 与 `many1` 之外，还有 `skipMany` 与 `skipMany1`，它们的类型是 `skipMany :: ParsecT s m a -> ParsecT s m ()`，只会跳过多个记号，而不返回任何的值。这样，当不需要一些记号时，就可以用这个跳过它们。

还有一组比较有用的函数是 `sepBy` 与 `sepBy1`，它们对一连串的输入进行分析，用户可以对规则加些限定。如果读者还记得 `words` 函数，则应该知道它不会忽略单词间的标点，通过语法分析，可以只将一个句子的单词提取出来，并且限定句子中的单词要通过空格或者逗号分开，并且一定要以句号、问号或者叹号结尾，但是在分析结果中忽略标点：

---

```
> words "Hi, Haskell is fun!"
["Hi", "Haskell", "is", "fun"]

separator :: Parsec String a {}
separator = skipMany1 (space <|> char ',')

sentence :: Parsec String () [String]
sentence = do
 words <- sepBy1 word separator
 end <- oneOf ".?!"
 return words

> parseTest sentence "Hi, Haskell is fun."
["Hi", "Haskell", "is", "fun"]
```

---

在使用 `<|>` 运算符时需要注意，当计算 `p <|> q` 时，如果 `p` 已经使用了一些输入，当中途产生错误时，那么它不会再选择 `q`，比如：

---

```
test1 :: Parsec String () String
test1 = string "(a)" <|> string "(b)"

> parseTest test1 "(b)"
parse error at (line 1, column 1):
unexpected "b"
expecting "(a)"
```

---

这样，因为 `string` 已经取得了 `(`，这样它会继续分析字符 `a`，在继续分析时发现给定的输入是 `b`，所以给出了错误。遇到这种情况，需要试探性地对 `"(a)"` 进行分析，如果失败，则使用 `<|>` 的第二个参数。

---

```
test2 :: Parsec String () String
test2 = try (string "(a)") <|> string "(b)"

> parseTest test2 "(b)"
"(b)"
```

---

下面介绍一下 `stateUser`，在解析时，总是需要统计一些信息。比如，在分析一个句子时，

需要清点这个句子中有多少个单词或者用了多少个标点，这时有了 `stateUser` 就非常方便了。

可以使用 `updateState` 来对状态进行操作，它的类型是 `updateState :: (Monad m) => (u -> u) -> ParsecT s u m ()`，如果给定一个函数，那么这个函数就可以用来更新这个状态。下面来定义 `word` 函数，每分析一个单词状态就加 1，所以有：

---

```
word' :: Parsec String Int String
word' = do
 word <- many1 letter
 updateState (+1)
 return word
```

---

`separator` 与 `sentence` 函数同之前的是一样的，只是类型有所不同，需要指定分析过程中保持的状态为整数。

---

```
separator' :: Parsec String Int ()
separator' = skipMany1 (space <|> char ',')

sentence' :: Parsec String Int [String]
sentence' = do
 words <- sepBy1 word' separator'
 oneOf ".?!"
 return words
```

---

最后就是字数统计的函数了，`getState` 函数可以返回当前的状态值：

---

```
wordCount :: Parsec String Int Int
wordCount = do
 words <- sentence'
 c <- getState
 return c
```

---

由于这里状态不再为 `()`，因此不能使用 `parseTest` 来测试。可以用 `runP :: Parsec s u a -> u -> SourceName -> s -> Either ParseError a` 来测试分析器，它需要给定被的状态还有分析来源的名字，随意指定为 `n`：

---

```
> runP wordCount 0 "n" "Hi, Haskell is fun!"
Right 4
```

---

有读者可能会觉得，既然可以得到单词的列表，那么可以用 `length` 函数来取得单词的个数。虽然这是没问题的，但是常常遇到的情况会更为复杂。比如，分析某种程序语言时对变量进行维护，可能就需要一个 `Map` 或者 `Set` 作为容器来管理它们，或者需要变量引用计数器（reference counter），这些都是可以用状态 `Monad` 来实现的，`Parsec` 中原生地加入了状态 `Monad`，给使用带来了极大的方便。

`Parsec` 库为了可以让使用者更好地处理空格、变量名、预留关键字、数字、括号以及注释等问题提供了词法分析的相关函数，都在 `Text.Parsec.Token` 里。可是，不同的语言可能对

于变量名、关键字还有注释所声明的要求是不同的，可以通过 `GenLanguageDef` 类型来定义一门语言的词法，也就是它会有什么样的记号。然后可以使用其中的 `makeTokenParser :: GenLanguageDef s u m -> GenTokenParser s u m` 函数来通过给定词法的定义生成对应的记号分析器。在 `Text.Parsec.Language` 中定义了 Haskell 与 Java 的词法，有兴趣的读者可以看看它们是如何定义的，这里将用到的是 `emptyDef`，即空的词法定义。在实现栈计算器的一节中，将字符串分析成记号时，对小数的处理是很麻烦的，分析出一个数字的代码如下：

---

```
scanNum xs | null num = error "Excepted a number or constant"
 | otherwise = case rest of
 ('.': r) -> let (float,r') = span isDigit r in
 Left (Val (read (num ++ "." ++ float) :: Float)) : scanBin r'
 r -> Left (Val (read num :: Float)) : scanBin r
 where (num, rest) = span isDigit xs
```

---

可以看到，上面的这个方法非常乱，看起来很不舒服，而有了 `Parsec` 的帮助，可以非常容易地实现这个函数：

---

```
import Text.Parsec
import qualified Text.Parsec.Token as T
import Text.Parsec.Language (emptyDef)

lexer :: T.TokenParser ()
lexer = T.makeTokenParser emptyDef

lexeme :: Parsec String () a -> Parsec String () a
lexeme = T.lexeme lexer

float :: Parsec String () Double
float = T.float lexer

> parseTest float "1.2"
1.2
```

---

其中，`lexeme` 可以帮助用户在分析语法过程中处理好尾部的空格问题。因为在 `T.float` 的定义中已经使用到了它，所以在用 `float` 时，不必考虑空格问题。

---

```
chars :: Parsec String () [Char]
chars = do
 c1 <- lexeme $ char 'a'
 c2 <- lexeme $ char 'b'
 return [c1,c2]
```

---

如果不用 `lexeme`，就会得到下面的错误：

---

```
> parseTest chars "a b"
parse error at (line 1, column 2):
unexpected " "
expecting "b"
```

---

每一次分析时，相当于得当了一个记号，而 `lexeme` 会将该记号后面的空格忽略。

库中所分析的小数不能为负数，不过，经过简单的改动就可以分析有符号或者无符号的小数了。

---

```
floatl :: Parsec String () Double
floatl = do
 f <- lexeme sign
 n <- T.float lexer
 return (f n)

sign :: Num a -> Parsec String () (a -> a)
sign = (char '-' >> return negate)
 <|> (char '+' >> return id)
 <|> return id

> parseTest floatl "-1.2"
-1.2
```

---

当然，还有处理空格的 `whiteSpace`、处理正整数的 `natural` 和处理带符号整数的 `integer`，这里就不再一一介绍了。想要深入研究的读者可以阅读本节的参考文献。下面详细地讨论一下上下文无关文法，然后用 `Parsec` 库来定义一个与之前类似的计算器，本节的代码在下节中还会用到。

本节的内容来自 Daan Leijen 的《`Parsec, a fast combinator parser`》可以在 <http://research.microsoft.com/en-us/um/people/daan/download/parsec/parsec.pdf> 中浏览。

#### 练习

上一章中的基于栈的计算器是先将表达式分析成为 `[Either Lit Op]` 再进行栈操作。

它是复合了两个类型为 `String -> [Either Lit Op]` 还有 `[Either Lit Op] -> Float` 得到的，这里你可以使用 `ParserT` 组合一个状态 `Monad` 来同时做这两件事情。

### 13.6.3 上下文无关文法

之前，在定义栈计算器时，想必读者对上下文无关文法（Context Free Grammar, CFG）已经有些了解了，本节来简要地讨论一下它们。

一个上下文无关文法实际上是一个四元组，语法规则用  $G$  表示，即  $G = (N, \Sigma, P, S)$ 。其中， $N$  为非终止符号（non-terminal）的集合，元素习惯用大写字母表示； $\Sigma$  为终止符号（terminal）的集合，元素习惯用小写字母或者特殊的符号表示； $P$  为生成规则（production rule）； $S$  为起始符号， $S$  一般是集合  $N$  中的元素。

比如，只考虑加法与乘法，那么算术表达式的语法是这样的：

---

```
Exp -> Exp + Exp | Exp * Exp | (Exp) | Number
Number -> 所有可能的数字
```

---

其中，`Exp` 是起始符号，一个算术表达式的生成是由它开始的。`Number` 是一个非终止符号，它最后可以生成所有的数字，数字是终止符号。类似地，`+` 与 `*` 为运算符，运算符也为终止符号。

但是这个语法是有歧义的，每次运行规则时，只应用最左侧的非终止符号，即便是这样得  $4+2*3$  有不同的方法，如图 13-1 所示。

Exp	Exp
$\rightarrow \text{Exp} + \text{Exp}$	$\rightarrow \text{Exp} * \text{Exp}$
$\rightarrow \text{Number} + \text{Exp}$	$\rightarrow \text{Exp} + \text{Exp} * \text{Exp}$
$\rightarrow 4 + \text{Exp}$	$\rightarrow \text{Number} + \text{Exp} * \text{Exp}$
$\rightarrow 4 + \text{Exp} * \text{Exp}$	$\rightarrow 4 + \text{Exp} * \text{Exp}$
$\rightarrow 4 + \text{Number} * \text{Exp}$	$\rightarrow 4 + \text{Number} * \text{Exp}$
$\rightarrow 4 + 2 * \text{Exp}$	$\rightarrow 4 + 2 * \text{Exp}$
$\rightarrow 4 + 2 * \text{Number}$	$\rightarrow 4 + 2 * \text{Number}$
$\rightarrow 4 + 2 * 3$	$\rightarrow 4 + 2 * 3$

图 13-1 使用 CFG 生成  $4+2*3$

这种歧义就是由于没有考虑到加法与乘法的优先级还有结合性导致的。下面来看如何消除这种歧义。

如果需要一个二元运算符  $\otimes$  的优先级比二元运算符  $\oplus$  高，那么语法应该写成：

$$A \rightarrow A \oplus A \mid B$$

$$B \rightarrow B \otimes B \mid C$$

$$C \rightarrow (A) \mid \dots$$

而如果需要一个运算符是在结合的，那么它需要被写成  $A \rightarrow A \oplus B$ ，如果是右结合的，那么可以写成  $A \rightarrow B \oplus A$ ，如果是没有结合性那么写做  $A \rightarrow B \oplus B$ 。

所以说，上边的加法与乘法的数学表达式的上下文无关文法应当写成：

---

```
Exp -> Exp + Mul | Mul
Mul -> Mul * Num | Num
Num -> (Exp) | Number
Number -> 所有可能的数字
```

---

这样就满足了乘法比加法的优先级高，并且都为左结合的。下面试着来写对应的分析器，看看可能会遇到什么问题。

---

```
data Exp = Add Exp Exp | Mul Exp Exp | Val Double deriving (Eq, Show)

parseExp :: Parsec String () Exp
parseExp = do
 e1 <- parseExp
 char '+'
 e2 <- parseMul
 return (Add e1 e2)
<|> parseMul
```

---

可以看到，`parseExp` 是一个递归函数，但是这样的定义后果就是在该函数被调用时 `parseExp` 马上就递归地调用自己，而没有做任何计算，最后导致了一个无限的递归，它的运行是不会停止的。这里的问题在于左递归，比如，定义为 1 的常数列表时，不会这样定义：

---

```
ones :: [Int]
ones = ones ++ [1]
```

---

如果这样定义这个列表，即便是有惰性求值也没有办法使用 `take 4 ones` 来得到结果。这就需要将语法的左递归消除（left recursion elimination），就像将 `ones` 定义为 `1:ones` 一样，那么，对于下面这样的左递归语法：

---

```
A → Aα1 | Aα2 ... | Aαn
| β1 | β2 ... | βn
```

---

它等价于下面的右递归语法：

---

```
A → β1A' | β2A' | ... | βnA'
A' → α1A' | α2A' | ... | αnA' | ε
```

---

$\epsilon$  表示空字符串，这样语法就转换好了，于是就可以将之前的语法改成下面这种形式：

---

```
Exp → Mul Exp'
Exp' → + Mul Exp' | ε
Mul → Num Mul'
Mul' → * Num Mul' | ε
Num → (Exp) | Number
```

---

这样，有了这个语法就可以定义了分析器，然后对表达式树递归求值了。来定义递归求值函数：

---

```
eval (Val v) = v
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```

---

假定已经有了分析器 `parseExp`，那么 `calculate` 函数定义如下：

---

```
eval' :: Parsec String () Double
eval' = do
 el <- parseExp
 return $ eval el

calculate :: String -> IO ()
calculate e = parseTest eval' e
```

---

接下来就可以对照着语法来定义分析器了，会看到语法中的竖线|相当于<|>运算符，而对于每一种情形，只需用 `do` 表达式连续地分析即可。这里， $\epsilon$  用 `Nothing` 来表示，也就是说，`Exp'` 与 `Mul'` 返回的是 `Maybe` 的类型。使用语法分析组合子定义的函数可以跟上下文无关文法对应得十分工整。

---

```
Exp → Mul Exp':
parseExp :: Parsec String () Exp
```

---

```

parseExp = do
 e1 <- parseMul
 e2 <- parseExp'
 case e2 of
 Nothing -> return e1
 Just e -> return (e e1)

 Exp' -> + Mul Exp' | ε

parseExp' :: Parsec String () (Maybe (Exp -> Exp))
parseExp' = do
 char '+'
 e1 <- parseMul
 e2 <- parseExp'
 case e2 of
 Nothing -> return (Just (\e -> Add e e1))
 Just e -> return (Just (\e' -> e (Add e' e1)))
 <|> return Nothing

Mul -> Num Mul':

parseMul :: Parsec String () Exp
parseMul = do
 el <- parseNum
 e2 <- parseMul'
 case e2 of
 Nothing -> return el
 Just e -> return (e el)

Mul' -> * Num Mul' | ε

parseMul' :: Parsec String () (Maybe (Exp -> Exp))
parseMul' = do
 char '*'
 el <- parseNum
 e2 <- parseMul'
 case e2 of
 Nothing -> return (Just (\e -> Mul e e1))
 Just e -> return (Just (\e' -> e (Mul e' e1))) <|> return Nothing

Num -> (Exp) | Number

parseNum :: Parsec String () Exp
parseNum = do
 char '('
 el <- parseExp
 char ')'
 return el <|> (do {num <- float1; return (Val num)})

```

下面来测试一下函数：

---

```
> parseTest parseExp "1.0+2.0*3.9"
Add (Add (Val 1.0) (Val 2.0)) (Val 3.0)
```

```

> calculate "1.0+2.0+3.0"
6.0

> parseTest parseExp "1.0+2.0*3.0"
Add (Val 1.0) (Mul (Val 2.0) (Val 3.0))

> calculate "1.0+2.0*3.0"
7.0

> parseTest parseExp "(1.0+2.0)*3.0"
Mul (Add (Val 1.0) (Val 2.0)) (Val 3.0)

> calculate "(1.0+2.0)*3.0"
9.0

```

可以看出，这种方法比起使用栈的计算器简单，并且具有非常好的灵活性，如果为基于栈的计算器添加一种运算符，那可能就非常的麻烦了。但是，如果计算器是用这种上下文无关语法实现的则会简单很多，也更容易出错。现在，相信大家已经对上文无关语法有所了解，其实各种计算机语言的语法都是通过它来定义的，只是更加复杂一些。不过，一旦有了语法的定义，用 Parsec 库来实现起分析器来是非常容易的。下节中，将完整地定义一个基于语法分析器的计算器，功能与之前基于栈的计算器大体相同。

#### 13.6.4 基于语法分析器的计算器

需要下面的库。

---

```
--Calculator.hs
import Text.Parsec
import qualified Text.Parsec.Token as T
import Text.ParserCombinators.Parsec.Language
```

---

先来定义表达式树类型，即计算器可以进行的计算表达式，再定义 eval 估值函数，看上去十分的冗长。

---

```

data Exp = Plus Exp Exp | Minus Exp Exp | Mult Exp Exp | Divi Exp Exp
 | Power Exp Exp | Nega Exp | Posi Exp | Sqrt Exp | Log Exp | Ln Exp
 | Sin Exp | Cos Exp | Val Double deriving (Eq, Show)

eval :: Exp -> Double
eval (Plus e1 e2) = eval e1 + eval e2
eval (Minus e1 e2) = eval e1 - eval e2
eval (Mult e1 e2) = eval e1 * eval e2
eval (Divi e1 e2) = eval e1 / eval e2
eval (Power e1 e2) = eval e1 ** eval e2
eval (Nega e1) = negate $ eval e1
eval (Posi e1) = eval e1
eval (Log e1) = logBase 2 (eval e1)
```

---

---

```

eval (Ln el) = log $ eval el
eval (Sin el) = sin $ eval el
eval (Cos el) = cos $ eval el
eval (Sqrt e) = sqrt $ eval e
eval (Val e) = e

```

---

下面就需要对语法进行分析了。如果不考虑优先级与结合性，原始的语法大致是与上面的表达式类型一致。这里用 $\ominus$ 表示所有一元运算符，语法为：

---

```

Exp -> Exp + Exp | Exp - Exp | Exp * Exp | Exp / Exp | Exp ^ Exp
| ⊖ Exp | (Exp) | Number

```

---

其中，Number 可以生成所有可能的数字。接下来，运用上一节的结论，根据运算符结合性与优先级分离这些生成规则：

---

```

Exp -> Exp + Mul | Exp - Mul | Mul
Mul -> Mul * UExp | Mul / UExp | UExp
UExp -> ⊖ UExp | Power
Power -> Num ^ Power | Num
Num -> (Exp) | Number

```

---

最后，消除左递归有：

---

```

Exp -> Mul Exp'
Exp' -> + Mul Exp' | - Mul Exp' | ε
Mul -> UExp Mul'
Mul' -> * UExp Mul' | / UExp Mul' | ε
UExp -> ⊖ UExp | Power
Power -> Num ^ Power | Num
Num -> (Exp) | Number

```

---

有了语法树，实现起来就是非常容易了，先来定义几个基本的分析器：

---

```

lexer :: T.TokenParser()
lexer = T.makeTokenParser emptyDef

lexeme :: Parsec String () a -> Parsec String () a
lexeme = T.lexeme lexer

constant :: Parsec String () Double
constant = do
 choice [lexeme $ string "pi" -> return 3.1415926,
 lexeme $ string "e" -> return 2.71828]

```

---

这里分析数字时不必考虑其符号，可以把正负号当做一元运算符来处理。

---

```

float :: Parsec String () Double
float = do
 n <- T.float lexer
 return n

number :: Parsec String () Double

```

---

---

```
number = try (do
 int <- T.integer lexer
 return (fromIntegral int)) <|> try float <|> constant
```

---

现在定义 Exp 与 Exp'：

```
Exp -> Mul Exp'
Exp' -> + Mul Exp' | - Mul Exp' | ε

parseExp :: Parsec String () Exp
parseExp = do
 e1 <- parseMul
 e2 <- parseExp'
 case e2 of
 Nothing -> return e1
 Just e -> return (e e1)

parseExp' :: Parsec String () (Maybe (Exp -> Exp))
parseExp' = do
 op <- choice $ (map (lexeme.char)) ['+', '-']
 e1 <- parseMul
 e2 <- parseExp'
 case e2 of
 Nothing -> case op of
 '+' -> return (Just (\e -> Plus e e1))
 '-' -> return (Just (\e -> Minus e e1))
 Just e -> case op of
 '+' -> return (Just (\e' -> e (Plus e' e1)))
 '-' -> return (Just (\e' -> e (Minus e' e1)))
 <|> return Nothing
```

---

然后，再定义 Mul 与 Mul'，它们与 Exp 跟 Exp' 几乎是相同的，其实复制修改一下就可用了。

```
Mul -> UExp Mul'
Mul' -> * UExp Mul' | / UExp Mul' | ε

parseMul :: Parsec String () Exp
parseMul = do
 e1 <- parseUExp
 e2 <- parseMul'
 case e2 of
 Nothing -> return e1
 Just e -> return (e e1)

parseMul' :: Parsec String () (Maybe (Exp -> Exp))
parseMul' = do
 op <- choice [lexeme $ char '*', lexeme $ char '/']
 e1 <- parseUExp
 e2 <- parseMul'
 case e2 of
 Nothing -> case op of
```

---

```

 '*' -> return (Just (\e -> Mult e el))
 '/' -> return (Just (\e -> Divi e el))
 Just e -> case op of
 '*' -> return (Just (\e' -> e (Mult e' el)))
 '/' -> return (Just (\e' -> e (Divi e' el)))
 <|> return Nothing

```

---

最后，定义一元运算、乘方和 Num。这里需要注意的是，由于一元前缀运算符可能有公共起始字符，比如 log 与 ln 均为 l 开头，所以需要使用 try。

最后实现下面 3 个生成规则：

---

```

UExp -> ⊕ UExp | Power
Power -> Num ^ Power | Num
Num -> (Exp) | Number

parseUExp :: Parsec String () Exp
parseUExp = do
 op <- choice (map (try.lexeme.string)
 ["-","+","log","ln","sin","sqrt","cos"])
 el <- parseUExp
 case op of
 "-" -> return $ Neg a el
 "+" -> return $ Posi el
 "sqrt" -> return $ Sqrt el
 "log" -> return $ Log el
 "ln" -> return $ Ln el
 "sin" -> return $ Sin el
 "cos" -> return $ Cos el
 _ -> fail "expect an unary operator"
 <|> parsePower

parsePower :: Parsec String () Exp
parsePower = try (do
 num <- parseNum
 lexeme $ char '^'
 el <- parsePower
 return (Power num el))
 <|> parseNum

parseNum :: Parsec String () Exp
parseNum = try (do
 lexeme $ char '('
 el <- parseExp
 lexeme $ char ')'
 return el) <|> (do {num<- number;return (Val num)})

```

---

从以上过程中可以看到，使用语法分析器来定义计算器的整个过程是非常清晰的。读者可以写一些表达式来测试一下这个计算器。语法分析是一个较为繁重的工作，所以小型设备上的计算器一般都是基于栈的，但是这并不会影响我们讨论语法分析器。本节只是以计算器为例，

目的是让读者可以比较熟练地应用它。读到这里，希望读者可以感觉到 Haskell 的高产性与 Parsec 库的易用性，这样一个程序只需要花费不到 1 个小时就可以完成。在平时的编程实践中，可能常常需要写一些处理文本的程序，使用 Haskell 与 Parsec 库可以在工作中带来极大的便利。

## 本章小结

本章学习了 Monad 转换器，了解了如何组合多个 Monad 得到一个功能更多的 Monad，使读者明白以何种顺序组合它们在日后的编程实践中非常重要。后面，又学习了使用 Parsec 库来做语法分析，Parsec 可以很容易并且直观地解决各类语法分析问题，这为分析字符串带来了很大的方便。Monad 的内容还远远不止于此，读到这里，读者应该感觉到 Monad 这一概念在函数式编程中非常重要并十分抽象。Monad 的用途非常多，它不只是一个简单的数学概念，还是一种设计模式。除了介绍过的，Monad 还有很多应用。其中，分析器 Monad 只是一项重要应用。此外，还有 ST Monad (state threads monad)、Iteratee Monad 与 Template Haskell 中的 Q Monad 等。由于本书不会讨论它们，因此读者需要阅读更多的相关内容才能了解 Monad 的真谛。

## 第 14 章

# QuickCheck 简介

测试一直是软件开发中比较重要的部分。虽然 Haskell 的类型系统可以帮助用户找到很大的一部分的错误，但是在应用开发中，测试还是非常重要的。Haskell 的纯函数特性给函数的测试带来了相当大的便利，由于没有副作用，纯函数的结果不依赖于内存状态与函数被调用的顺序，加之 Haskell 强大的类型系统，因此使函数的自动测试成为可能——QuickCheck 就是为完成这一任务而编写的库。

### 14.1 测试函数属性

QuickCheck 主要是对函数的结果属性进行的测试，也就是说，结果一定要满足的条件。我们从最基本的一个例子开始，它是 even 函数的属性。首先，需要导入 Test.QuickCheck。

```
import Test.QuickCheck
```

对于两个整数的和，它的奇偶性满足 `even (x+y) == (even x == even y)`，例如：

```
even (4 + 5)
= even 9
= False
```

```
even 4 == even 5
= True == False
= False
```

这是一个已经被证明的性质，下面使用 QuickCheck 来测试这个性质。这个性质可以定义为：

```
prop_even x y = even (x+y) == (even x == even y)
```

定义好了这个函数就可以使用 `quickCheck` 来测试这个属性是否成立了，它们类型为：

```
> :t quickCheck
quickCheck :: Testable prop => prop -> IO ()
```

---

```
> quickCheck prop_even
+++ OK, passed 100 tests.
```

---

这就是说，quickCheck 生成了 100 个案例，全部通过了。测试就是这样，虽然有 100 个案例通过了，但是也不能绝对地保证正确性，对于函数的性质，数学证明要优于单纯的函数测试。在这里，不必指定 prop\_even 的类型，因为它的类型可以根据 even 自动推断。

测试时，可以使用 verboseCheck 来看 QuickCheck 究竟用了哪些案例对该函数进行了测试。下面再来看一些其他的例子。

reverse 函数有一些比较有趣的性质，用户可以使用 QuickCheck 来依次测试它们。首先，如果列表的元组只有一个，那么倒置后的结果与输入是相等的，即：

---

```
prop_reverseUnit x = reverse [x] == [x]
```

---

在测试时，要指定需要测试的类型，否则 QuickCheck 只会默认为 () 类型。

---

```
> quickCheck (prop_reverseUnit :: Int -> Bool)
+++ OK, passed 100 tests.
```

---

另外一个属性为 reverse (xs ++ ys) == reverse ys ++ reverse xs。

---

```
prop_reverseConcat xs ys = reverse (xs ++ ys) == reverse ys ++ reverse xs
```

---

同样，这个属性也是可以通过测试的。还有一个性质是将一个列表倒置两次，但还是这个列表本身，即：

---

```
prop_reverseTwice xs = (reverse.reverse) xs == xs
```

---

```
> quickCheck (prop_reverseTwice :: [Int] -> Bool)
+++ OK, passed 100 tests.
```

---

当然，属性 (reverse.reverse.reverse) xs == reverse xs 也是成立的，进行的测试实质上就是看两个函数 reverse.reverse.reverse 与 reverse 是否相等。

然而，当属性需要有函数作为输入的时候则会产生问题：其一，如何随机地生成一个满足类型的函数；其二，如果测试的属性在这个函数上失败了，那么如何打印出这个函数？例如，下面的这个性质是显然成立的。

---

```
prop_reverseMap f xs = (map f.reverse) xs == (reverse.map f) xs
```

---

针对这两个问题是没有什么很好的解决方法的。虽然可以用一个有限的二元组的列表来定义出一些函数，但是在处理高阶函数或其他更为复杂的情形时，对于函数  $f$  可能需要用户自己定义，然后再进行测试。

下面需要对列表的排序的相关函数来做一些测试，需要用到 Data.List 库。对于一个升序排列的列表，它应具备一个属性，这个属性可以叫做 ordered，它的定义如下：

---

```
ordered :: Ord a -> [a] -> Bool
ordered [] = True
ordered [x] = True
ordered (x:y:xs) = x <= y && ordered (y:xs)
```

---

一个空列表是有序的，仅有一个元素的列表也是有序的，这两个属性可以被认为是无关紧要的真理；而如果列表长度大于 2，那么只有比较前两个，然后递归地再判断第二个元素与余下的列表是否满足有序的属性。这样就可以对排序函数的结果做出了断言，即排序函数的结果一定要满足这个属性。

---

```
prop_ordered xs = ordered $ sort xs
```

---

如果能通过测试，则说明函数应该没有问题。有时还会在定义递归函数时找到些计算过程中的不变量，了解这些不变量不仅仅在 Haskell 中很重要，在其他语言中使用循环或者递归时同样非常重要。例如，在定义插入排序时，如果将一个元素用 `insert` 函数插入到一个有序的列表中，那么结果还是有序的。定义了这个属性，就可以测试 `insert` 函数的定义是否是正确的了。

---

```
prop_insertOrdered x xs = ordered $ insert x (sort xs)
```

---

有时在对函数测试时，需要参数满足特有的条件，这时，可以使用`==>`来做出限定。例如，当一个升序的列表的第一个元素应当是该列表的最小元素，也就是说，排序函数的结果应该会有这样的属性：

---

```
> :t (==>)
(==>) :: Testable prop -> Bool -> prop -> Property
```

---

如果直接测试这个属性：

---

```
prop_headMin xs = head (sort xs) == minimum xs

> quickCheck prop_headMin
*** Failed! Exception: 'Prelude.head: empty list' (after 1 test):
[]
```

---

显然，想测试这个属性的前提是要保证列表是不能为空的。QuickCheck 会首先检查列表为空的情况，这些特殊情况也正是定义函数时容易出错的情形。这样，可以使用`==>`来让 QuickCheck 只会使用符合条件的案例进行测试。`==>`可以理解为蕴含着、意味着或者前提条件，那么，这里的意思就是，若给定的列表不为空，就意味着将其按升序排列后得到列表的第一个元素为该列表的最小值。

---

```
prop_headMin' :: Ord a -> [a] -> Property
prop_headMin' xs = not (null xs) ==> head (sort xs) == minimum xs

> verboseCheck (prop_headMin' :: [Int] -> Property)
Passed:
[4783, 14289, -18342, -15871, 13308, 19029, 20689, -6292, -20977, 13059]
```

---

```
Passed:
[-38998, 37865, -48802, 59889, 29579, -2987, -28919]
...
+++ OK, passed 100 tests.
```

## 14.2 测试数据生成器

随机测试数据要根据数据的类型来生成。QuickCheck 中定义了 `Arbitrary` 类型类，其中的 `arbitrary` 函数定义了如何生成测试数据。

```
class Arbitrary a where
 arbitrary :: Gen a
```

如果想要某个类型的值，可以随机地生成，只需要实现这个类型类即可。其中，`Gen a` 类型用到了随机数生成一节中的 `StdGen`，它的功能就是随机生成一个 `a` 类型的值。`Gen` 实现了 `Monad` 类型类，所以可以使用 `do` 关键字来定义新的生成器。定义数据生成器时，可能需要取随机数或者对一些列表元素进行随机选取等，QuickCheck 中提供了一些函数来满足用户的需要。

表 14-1 QuickCheck 中生成数据常用函数

函 数	功 能
<code>elements :: [a] -&gt; Gen a</code>	从一个列表中随机生成某个元素
<code>choose :: Random a -&gt; (a, a) -&gt; Gen a</code>	在一个范围内随机地生成一个值
<code>oneof :: [Gen a] -&gt; Gen a</code>	从多个生成器中随机地抽取一个

假定，想测试基于栈计算器一节中定义的 `scanner` 函数。在现实开发中，程序的编写与测试是分开的，现在，假定做为测试人员，需要为 `scanner` 生成大量的合法的算术表达式来测试 `scanner` 函数是否正确。现实中可能不会做这样的测试，因为生成的规则是由 `scanner` 所用的语法规则得来的，所以，在一般情况下并不需要测试，这里只是给出一个生成测试数据的例子以供参考。

需要使用下面的库：

```
import Test.QuickCheck
import Control.Monad
```

首先，定义一个 `Exp` 类型，`Exp` 构造器内是一个字符串，将生成的算术表达式存在这里。

```
newtype Exp = Exp String deriving (Show, Eq)
```

对照算术表达式的语法：

```
Expression ->
 | Expression -> Expression
 | (Expression) BinaryExpression
 | Number BinaryExpression

BinaryExpression -> * Expression | ε
```

首先，需要在 0, 1, 2 中随机一个数，分别对应 Expression 的 3 个生成规则，然后再对 3 个规则分别讨论。在生成运算符时，可以使用 elements 对多个运算符进行随机地选择，而在生成 Number 时，可以生成数字，也可以生成常数。这时，定义两个生成器，然后使用 oneof 函数来随机选取一个生成。

---

```

instance Arbitrary Exp where
 arbitrary = do
 n <- choose (0,2) :: Gen Int
 case n of
 0 -> do
 unaryOp <- elements ["","","+","-","sin ","cos ","log ","ln ","sqrt "]
 (Exp exp) <- arbitrary :: Gen Exp
 return (Exp (unaryOp ++ exp))
 1 -> do
 Exp exp <- arbitrary :: Gen Exp
 let bracketExp = "(" ++ exp ++ ")"
 nl <- choose (0,1) :: Gen Int
 case nl of
 0 -> do
 binaryOp <- elements ["+","-","*","/","^"]
 (Exp expl) <- arbitrary :: Gen Exp
 return (Exp (bracketExp ++ binaryOp ++ expl))
 1 -> do
 return (Exp bracketExp)
 2 -> do
 num <- oneof [liftM show (choose (-20 ,20) :: Gen Int),
 elements ["pi","e"]]
 nl <- choose (0,1) :: Gen Int
 case nl of
 0 -> do
 binaryOp <- elements ["+","-","*","/","^"]
 (Exp expl) <- arbitrary :: Gen Exp
 return (Exp (num ++ binaryOp ++ expl))
 1 -> do
 return (Exp (num))

```

---

下面可以使用 sample 函数来测试一下这个测试数据生成器：

---

```

> sample (arbitrary :: Gen Exp)
Exp "-9+pi"
Exp "-e^e+pi-2"
Exp "-4/(-ln +(6)*18)/-7/ln cos e+ln log -15"
Exp "sqrt (e)"
Exp "(e)"
Exp "pi-(-14)"
Exp "ln +(e*6)"
...

```

---

有方法可以随机生成这些样本，就可以非常容易地对函数进行测试了，QuickCheck 库中还

有很多其他的函数可以辅助用户对测试数据进行生成。比如：

---

```
frequency :: [(Int, Gen a)] -> Gen a
```

---

`frequency` 可以给不同的生成器一个权重来控制某个生成器被选择的概率。

---

```
suchThat :: Gen a -> (a -> Bool) -> Gen a
```

---

`这个函数可以从一个生成器中得到满足给定条件的值。`

---

```
listOf ,listOf1 :: Gen a -> Gen [a]
```

---

这两个函数可以调用一个生成器多次来生成一个列表，若 `listOf` 可能为空，则 `listOf1` 至少会有一个元素。除这些外，还有其他更多的函数，读者可以查阅 QuickCheck 文档来了解。QuickCheck 就介绍到这里了，有兴趣的读者可以试着修改一下上边的测试生成器，对生成表达式的长度做出限定，让生成的表达式不要过长。

这一章的内容主要参考了由 John Hughes 与 Koen Claessen 发表的“QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs”，可以访问 <http://www.cs.tufts.edu/~nr/cs257/archive/john-hughes/quick.pdf> 阅读。

#### 练习

我们在定义数据类型时介绍了类型间的同构关系，定义两个同构的类型 A 与 B，以及它们间的转换函数 f 与 g，最后，使用 QuickCheck 测试 f.g 还有 g.f 是否与 id 函数相等。

## 本章小结

QuickCheck 是 Haskell 在对函数测试时用到的最具代表性的方法之一。除 QuickCheck 这种测试框架（testing framework）以外，还有与 Java 测试所用到的 JUnit 相似的 HUnit。Haskell 中也提供了两者统一的测试框架。另外，还有 SmartCheck 可以帮助用户更好地测试基于代数数据类型的函数。但是，想要了解它们，可能需要更多的书中没有介绍的内容，有兴趣的读者可以通过浏览对应的库，以及在网上查找些相关资料来了解它们。

# 第 15 章

## 惰性求值简介

前面的章节中我们一直在提 Haskell 的惰性求值。本章来深入讨论一下 Haskell 中的惰性求值。但是在讨论惰性求值之前，需要先简单了解函数式编程的基石  $\lambda$  演算与其他一些符号以及相关术语。

### 15.1 $\lambda$ 演算简介

$\lambda$  演算由美国数学家 Alonzo Church 引入。虽然它的定义简单，但这种演算可以被看做是一个完整的编程语言，因为它具备了表达所有数据结构与算法的能力。下面看一下  $\lambda$  演算的语法。

$\lambda$  语法的定义非常简单，它的定义如下：

---

```
Expression -> Constant
| Variable
| Expression Expression
| λ Variable. Expression
```

---

Constant 意为常量（如布尔类型中的 `True` 与 `False` 等），也可以是加法、减法等函数。但是，它在个定义中是可以略去的，因为所有的常量以及其他函数都可以通过另外 3 个构造出来的。

- Variable 为变量名称，可以使用任何的字符串来代指一个变量，但是约定使用小写字母，而大写字母用来指代一整个的  $\lambda$  表达式。
- Expression Expression 相当于读者了解的函数应用 (application)。
- $\lambda$  Variable. Expression 则为  $\lambda$  函数定义，称为  $\lambda$  抽象 ( $\lambda$  abstraction)。若一个变量  $x$  在函数体  $t$  中通过  $\lambda$  抽象引入了，那么则称  $x$  在  $t$  内被限定了 (bounded)。比如，在  $\lambda x. t$  中，表达式  $t$  中的变量  $x$  被则被限定了。若  $\lambda$  表达式中的变量没有通过  $\lambda$  抽象被限定，那么这个变量被称为游离的 (free)。在  $\lambda$  表达式中，如果不存在游离的变

量，那么这个表达式称为闭合的（closed）；否则称为开放的（open）。

其中， $\lambda$  抽象默认为右结合的，比如： $\lambda x.x\ y$  意为  $(\lambda x.(x\ y))$  而非  $((\lambda x.x)\ y)$ 。而函数应用默认为左结合的，比如： $f\ x\ y$  所指的为  $((f\ x)\ y)$  而非  $(f\ (x\ y))$ 。所以表达式  $\lambda x.\ \lambda y.\ \lambda z.x\ y\ z$  意为  $(\lambda x.\ (\lambda y.\ (\lambda z.(x\ y)\ z)))$ 。

在第2章中定义了3种对 $\lambda$ 式子转换与化简的方式，本节来深入地讨论一下它们。

- $\alpha$  替换： $\lambda x.\lambda y.x + y =_{\alpha} \lambda a.\lambda y.a + y$  称为  $\alpha$ -conversion。 $\alpha$  替换实际上就是在不出现变量名冲突的情况下对变量进行重命名。这个就不再多介绍了。
- $\beta$  化简： $(\lambda x.M)N+ \rightarrow_{\beta} [x/N]M$  称为  $\beta$ -reduction。
- $\eta$  化简： $(\lambda x.Mx) \rightarrow_{\eta} M$  称为  $\eta$ -reduction。

若一个 $\lambda$ 表达式不能再进行 $\beta$ 化简（也有将化简译为归约的），则称这个 $\lambda$ 表达式为 $\beta$ 常数形式（ $\beta$ -normal form），可以简写为 $\beta$ -nf；如果进一步地连 $\eta$ 化简也不能进行，则称为 $\beta\eta$ 常数形式（ $\beta\eta$ -normal form，也有将其译为范式的）可简写为 $\beta\eta$ -nf。比如 $(\lambda x \rightarrow M\ x)$ 为 $\beta$ -nf，却不为 $\beta\eta$ -nf。若某 $\lambda$ 表达式 $E$ 被化简为表达式 $N$ ，若 $N$ 为 $\beta$ -nf，那么 $N$ 称为 $E$ 的 $\beta$ -nf。所有的常数都为 $\beta$ -nf，因为它们无法被化简，比如 True 和函数 $(\lambda x.(x\ True))$ 均为 $\beta$ -nf。而 $(\lambda x \rightarrow x + 3)\ 4$ 还可以进行 $\beta$ 化简，所以它不为 $\beta$ -nf。

为了方便讨论，定义几个 $\lambda$ 表达式：

$$I = \lambda x.x \quad K = \lambda x_1.\lambda y_1.x_1 \quad W = \lambda w.w\ w$$

如果计算 $W\ W$ 的值，那么它是永远不会停止的，它的结果可以记为上。上符号会在下一节中简单讨论。

$$\begin{aligned} & W\ W \\ &=_{\beta} (\lambda w.w\ w) \quad (\lambda w.w\ w) \\ &=_{\beta} (\lambda w.w\ w) \quad (\lambda w.w\ w) \\ &\dots \end{aligned}$$

这样就会产生一个问题，比如在计算 $K\ I\ (W\ W)$ 时， $W\ W$ 的计算不会停止，那么 $K\ I\ (W\ W)$ 的计算是否会停止呢？它可以不停的原因是 $(W\ W)$ 的计算不会停止，而可以停止的原因是 $(W\ W)$ 与计算的结果无关。如果按以下的方式来求值，则计算会停止。

$$\begin{aligned} & K\ I\ (W\ W) \\ &= (\lambda x_1.\lambda y_1.x_1) \quad (\lambda x.(x)) \quad ((\lambda w.w\ w) \quad (\lambda w.w\ w)) \\ &= (\lambda y_1. \quad (\lambda x.(x))) \quad ((\lambda w.w\ w) \quad (\lambda w.w\ w)) \\ &= \lambda x.(x) \end{aligned}$$

但是，如果在2步一直计算 $W\ W$ 的值，那么这个计算就不会停止。所以，当计算一个表达式时，计算是否会停止与使用的求值策略有关。不同的求值策略将在15.4节中讨论。

$\lambda$ 演算就暂时介绍到这里。在求值策略一节中还会继续讨论 $\lambda$ 演算。接下来介绍一些其他

关于求值的相关术语与表达式形态。

## 15.2 $\perp$ Bottom

在讨论函数式编程语言语义时，可以用 $\perp$ 表示计算不可能完成的结果。有一些 $\lambda$ 表达式不会被化简为 $\beta\text{-nf}$ ，那么这种表达式的语义或者结果就可以表示为 $\perp$ ，如上一节中的`w w`，在语义上就可表示为 $\perp$ 。在没有特殊符号的时候，很多文档将它写为`_|_`。在 Haskell 中，它可以代表多种形式，如`forever` 的结果、异常或者是`undefined` 函数。

任何类型的计算都是可能没有结果的或者失败的。在使用`data` 定义类型时，Haskell 会自动为每一种类型附加了一个额外的值 $\perp$ ，这样就可以区别不同类型上计算的失败或者不终止，比如我们可以说`div a 0 = ⊥`。包括 $\perp$ 值的类型称为 lifted。如果这样，严格来说在定义`data MyInt = MyInt Int` 与 `Int` 为同构时，我们需要从 `Int` 中取出一个特殊值`n` 来对应 `MyInt ⊥`，也要将 `MyInt` 对应回去。同时，在之前讨论自然数 `Nat` 与 `[Unit]` 同构时是不精确的，因为还需要附加另外一个匹配`nat2List ⊥Nat = ⊥[Unit]`，但是在定义函数时是不能匹配上的，但是数学上有时常常会将其写出。

而`newtype` 类型则仅仅是相当于把一个类型的值放在一个构造器内重新命名。比如速度大小、距离、高度、面积等，都可以用`Double` 表示，但有时需要区分它们，就可以用`newtype`。但是与`data` 相比，`newtype` 不会引入 $\perp$ 。例如：

---

```
newtype N = N Int
data D = D Int
n (N i) = 42
d (D i) = 42
```

---

计算`n ⊥ = 42`，因为`newtype` 定义中没有默认定义`n ⊥ = ⊥`，也就是说，对于所有的值结果都是 42，这样，基于`N Int` 的计算就会得到结果。由于使用`data` 定义类型，类型中会有 $\perp$  一个值，而`d ⊥ = ⊥`，所以有：

---

```
> n undefined
42

> d undefined
*** Exception: Prelude.undefined
```

---

这里，使用`data` 定义新类型与使用`newtype` 定义新的类型就会有所不同。在函数式编程中，如果一个函数`f` 是严格的（strict）那么当且仅当`f ⊥ = ⊥`。也就是说，如果函数`f` 的参数求值不成功，那么函数`f` 的计算也不会成功。C 语言、Java 语言等用的就是这种方式，它们中定义的函数都是严格的，称为严格的语言，反之称为非严格的语言，比如 Haskell、Miranda 等中的函数。注意，等号两边的 $\perp$ 可能有着不同的类型，比如`Just ⊥ = ⊥`，左边可能为`Int` 类型的 $\perp$ ，而右边是有着`Maybe` 类型的 $\perp$ 。需要在语法上区别不同类型的 $\perp$ ，但是它们的语义在

某种程度上是相同的，可以不做区分。这里的函数 n 就为不严格的，而函数 d 为严格的，这样，相同的函数在使用不同关键字定义时行为就会有所不同。

但是在计算的时候，因为  $\perp$  为 Int 类型中一个的值，事实上 d  $\perp$  为 D Int 中的一个值。这样，由于 Haskell 的惰性求值，在计算 d(D undefined) 时函数会返回 42，因为它必去严格计算 D undefined 的值。

---

```
> d (D undefined)
42
```

---

如果要明确 D 构造器中的严格地计算 D 构造器中的 Int 值时，那么需要在类型前加一个!，即：

---

```
data D2 = D2 !Int
d2 (D i) = 42

> d2 (D undefined)
*** Exception: Prelude.undefined
```

---

但是，在使用 newtype 时，则不能使用感叹号来限定构造器内的类型。强制 Haskell 严格求值可以使用 seq :: a -> b -> b 函数与 (\$!) :: (a -> b) -> a -> b 运算符。seq 函数会直接返回第二个参数为结果，但是在返回的时候会计算严格地计算第一个参数的值，从语义的角度出发，它的定义可以理解为 seq a b = if a ==  $\perp$  then  $\perp$  else b。而 (\$!) 与 (\$) 是类似的，只是在将参数传给函数前会严格地计算第二个参数，如果为  $\perp$ ，那么计算的结果就是  $\perp$ 。

---

```
> const id undefined 0
0
> (const id $! undefined) 0
*** Exception: Prelude.undefined

> seq undefined 0
*** Exception: Prelude.undefined
```

---

## 15.3 表达式形态和 thunk

### 15.3.1 WHNF、HNF 与 NF

一个  $\lambda$  表达式称为 WHNF (Weak Head Normal Form)，当且仅当它是符合以下形式及条件：

- $F E_1 E_2 \dots E_n$  ( $n$  为自然数)。
- 当  $F$  为一个 Haskell 中的函数或者为一个值时，如 sqrt、True。又如 (f ((+ 5 5)))， $f$  在这里为游离的，它也可以是一个构造器，比如 Just (5+5) 就为一个 WHNF。

另一种情况可能是  $F$  为一个  $\lambda$  抽象或者一个内嵌的函数，并且  $F E_1 E_2 \dots$  不能继续化简，

如  $F$  为  $(+)$ ,  $E1$  为  $((-) 2 1)$ , 那么  $F E1 = (+) ((-) 2 1)$ , 之前我们知道, 四则运算表达式可以写成树状结构。其实, 不仅仅是这种数学表达式, Haskell 中所有的表达式都可以写成这种树状结构。其中,  $(+)$  为最顶层函数, 由于它的参数不够而不能进行化简, 所以内部的  $2-1$  对于 Haskell 来说不必计算, 因为计算了也无法得到最后的结果, 所以它的计算可以被延后。再如, 当  $n=0$  时,  $(\lambda x. (+) 1 1)$  也是一个 WHNF, 因为表达式最顶层函数缺少参数, 那么  $1+1$  可不必计算。

简而言之, 可以说当且仅当表达式树的最顶层无法化简时, 那么这个表达式就是一个 WHNF。还有一个与本节相关的概念是 HNF (Head Normal Form), 它与 WHNF 常常难以区别, 因为在很多情况下, 它们的确没有区别。它的定义是这样的。

一个表达式为 HNF (Head Normal Form), 当且仅当它可以写为如下的形式:

$\lambda x_1. \lambda x_2. \dots. \lambda x_n. (v M_1 M_2 M_3 \dots M_m)$  ( $m$  和  $n$  为自然数) 并且对于所有的  $p \leq m$ , 在  $(v M_1 M_2 M_3 \dots M_p)$  都不能被化简。其中,  $v$  可以为数据, 也可以是一个嵌入的函数, 但不可以为  $\lambda$  抽象, 因为这样就可以做  $\beta$  化简了。这样, 根据定义, 所有为 HNF 的表达式都为 WHNF, 因为  $\lambda x_1. \lambda x_2. \dots. \lambda x_n. (v M_1 M_2 M_3 \dots M_m)$  的最高层为  $\lambda$  抽象, 它无法被化简, 所以 HNF 是 WHNF 的一种特例。

例如,  $\lambda x. (\lambda y. x+y) 3$  为 WHNF, 而不是一个 HNF, 因为  $(\lambda y. x+y) 3$  可以被化简。 $\lambda x. x ((\lambda y. y) 5)$  为 HNF, 因为这里的  $\beta$  化简在内部。Haskell 中的  $\backslash x \rightarrow (\backslash y \rightarrow \text{Either} (x+y) (5+6))$  就是 HNF, 因为  $\text{Either} (x+y) (5+6)$  不可再进行  $\beta$  化简。当化简遇到最外层的构造器  $\text{Either}$  时, 对于这个值的计算就可以停止了, 这里的  $\text{Either}$  对应为定义中的  $v$ 。由于表达式最高层无化简, 因此它也是 WHNF。之前所介绍的 `seq` 函数就会将表达式计算为 HNF。但在它们的差别不是那样重要时, 只讨论 WHNF, 而不说 HNF。

另一个相关的概念则是 NF (Normal Form)。它的定义是: 如果一个表达式中不包括任何的化简, 即它被完全地计算了, 那么这个表达式就称为(NF), 与  $\beta$ -nf 中的 nf 是一个意思,  $\beta$ -nf 就是说一个  $\lambda$  表达式不可以再进行  $\beta$  化简了。比如, HNF 中  $\lambda x_1. \lambda x_2. \dots. \lambda x_n. (v M_1 M_2 M_3 \dots M_m)$ , 如果  $v$  和  $M_1$  或者其他某个表达式中有可以被化简的, 那么这个表达式就不是一个 NF。那么, 根据定义, NF 也是 WHNF 的一种特殊形式。所以, WHNF 包括了 HNF, HNF 包括了 NF, 它们之前是真包涵的关系。于是, 它们的关系如图 15-1 所示。

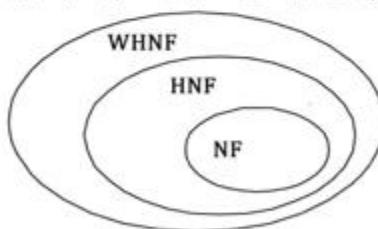


图 15-1 WHNF、HNF 和 NF 之间的关系

### 15.3.2 thunk 与严格求值

thunk 意为形实替换程序 (有时也称为延迟计算, suspended computation)。它指的是在计算的过程中, 一些函数的参数或者一些结果通过一段程序来代表, 这被称为 thunk。可以简单地把

thunk 看做是一个未求得完全结果的表达式与求得该表达式结果所需要的变量环境组成的函数，这个表达式与环境形成了一个无参数的闭包（parameterless closure），所以 thunk 中有求得这个表达式所需的所有信息，只是在不需要的时候不求而已。比如，`[1..]`这样无限的列表，在计算的时候可能不需要该列表中的所有元素。惰性求值过程就是尽可能地拖延对它的估值，当某个值不需要的时候就不去计算它，但是，需要通过一段程序来代替它保证在需要这个值的时候可以计算出来。

还以`[1..]`为例，当计算时候，它是一个未被计算的 thunk。当需要第一个值时，它的第一个值就被计算出来了。它将被计算为`thunk:thunk`，因为需要第一个值又可以分为需要第一个值占位，或者需要第一个值计算。如果只是需要它占位，那么求到这里就可以了；如果需要它的值参与计算，那么第一个 thunk 会被计算成 1，就得到`1:thunk`。迫使 thunk 求值的过程被称为胁迫计算（forcing）。thunk 有自己的表达式与环境，它不需要任何的参数，也许可以被理解为一个不需要参数的函数。当需要它求值时，就可以从原来的 thunk 中返回一个新值与一个新的 thunk，或者所有的值已经完全被计算好了而结果中不再包括任何的 thunk 了。也可以将 thunk 理解为一个容器，所装的内容在不需要时可以不用考虑，只有需要时才会从中取。这样，thunk 通过替代构造器内的值而拖延了计算，所得到的形式就为 WHNF。以`1:thunk`为例，这个表达式最顶层为`(::)`，thunk 代指了 1 后面的列表。由于再不能化简，因此是 WHNF，同时如果不需要 thunk 内的值，也不需要计算。

这里为了展示 thunk 的求值，我们使用 GHCI 中的`:sprint` 命令，它可以在打印结果时不严格地求值，对于未求出的部分仅仅是用下划线\_表示。此外，Haskell 中定义的函数很多默认使用了严格求值，这里为了测试，我们定义一些自己的函数来测试 Haskell 的 thunk 大体上是如何工作的

---

```

head' (x:xs) = x
length' [] = 0
length' (x:xs) = 1 + length' xs

> let xs = map (+1) (take 10 [0..])
*Main> :sprint xs
xs = _
*Main> head' xs
1
*Main> :sprint xs
xs = 1 : _
*Main> length' xs
10
*Main> :sprint xs
xs = [1,_,_,_,_,_,_,_,_,_]

```

---

我们可以看到，在使用`head'`时，其参数`xs`的结果并没有全部求出，而是仅仅使用了`_`代表。而`length'`函数也不需要知道列表中的值，所以所有的值都用`_`来代替。下面我们来继续测试`seq`函数对于表达式带来的影响

---

```
> seq xs ()
()
*Main> :sprint xs
xs = [1,_,_,_,_,_,_,_,_,_,_]
```

---

这里 `seq` 对于 `xs` 的求值没有任何的影响，它已经是一个 HNF 了，所以 `seq` 并不能求列表中的值。

在讨论  $\perp$  的时候，定义的 `D` 类型中的构造器内为  $\perp$  时， $D \perp = \perp$ ，并且  $d \perp = \perp$ 。可是，函数 `d (D undefined)` 一样可以返回结果。

---

```
data D = D Int deriving Show

d :: D -> Int
d (D i) = 42

> D undefined
D *** Exception: Prelude.undefined

> d undefined
*** Exception: Prelude.undefined

> d (D undefined)
42
```

---

这里说明，Haskell 在计算 `d (D undefined)` 时并没有计算 `D` 构造器内的值，而是使用一个 thunk 来代表，当需要时才对它进行计算。`D undefined` 为 WHNF，因为这个表达式的最高层没有办法再化简了，其中，`undefined` 可以使用 thunk 来表示，并且在计算的过程中不需要胁迫计算 thunk 中 `undefined` 的值，所以 `d` 函数会直接返回结果。

所以一些函数在不会用到构造器中的值时，那么其中的值就不会被计算，比如，`length [undefined, 1]` 会得到 2，即便使用 `seq` 时 `undefined` 也不会被计算。而当需要这个值参与计算时，如 `sum [undefined, 1]`，就会得到异常，因为我们需要列表中的值来计算结果。如果一定要将一个表达式计算为 NF，只需要使用 `Control.DeepSeq` 中的 `deepseq` 函数还有 `$!!` 运算符，它们会将数据的结构完全地计算成为 NF。

---

```
>:m +Control.DeepSeq
> let x= [undefined]::[Int] in x `seq` length x
1

> let x= [undefined]::[Int] in x `deepseq` length x
*** Exception: Prelude.undefined

> :t ($!!)
($!!) :: NFData a => (a -> b) -> a -> b
Prelude Control.DeepSeq> let x= [undefined]::[Int] in length $!! x
*** Exception: Prelude.undefined
```

---

下面我们来看 seq 与 deepseq 在对表达式求值时产生的影响:

---

```
> let xs = [1..10]
> :sprint xs
xs = _
> seq xs ()
()
> :sprint xs
xs = _ : _
> deepseq xs ()
()
> xs
[1,2,3,4,5,6,7,8,9,10]
```

---

这样，使用 deepseq 还有\$!!运算符就可以对构造器内部的式子进行递归地严格的估值计算了，即便列表内部的还是列表也会被求出来。[1..]即便使用 seq 也不会被严格求值，因为通过 thunk 它会被计算成为 HNF。而使用 deepseq 时，这些 thunk 都会被胁迫求值，最后，这个列表就被计算成为了 NF，所以这个无穷列表的计算不会停止，最终导致内存溢出异常。

---

```
> let xs = map (\x -> [x,x,x]) [1..3]
> :sprint xs
xs = _
> deepseq xs ()
()
> xs
[[1,1,1],[2,2,2],[3,3,3]]

> let x = [1..] :: [Int] in seq x (take 5)
[1,2,3,4,5]

> let x = [1..] :: [Int] in deepseq x (take 5)
Stack space overflow
```

---

同样，Haskell 还提供与\$、\$!对应的运算符\$!!，即将参数计算为 NF 后传给函数。Data.DeepSeq 中定义了 NFData 类型类，表示所有可以被计算为 NF 的类型。该类型类中定义了函数 rnf :: a -> ()（为 reduce to normal form 的简写），它将类型为 a 的数据计算为 NF，然后返回()。Maybe 与列表是这样实现 NFData 类型类的：

---

```
instance NFData a => NFData (Maybe a) where
 rnf Nothing = ()
 rnf (Just x) = rnf x
```

---

由于 Nothing 是一个具体值，当 rnf 函数对 Maybe 类型的值进行匹配时需要求出是哪种构造器，所以 Nothing 不必再求，而对于 Just 则需要对构造器中的 x 进行严格求值。

---

```
instance NFData a => NFData [a] where
 rnf [] = []
 rnf (x:xs) = rnf x `seq` rnf xs
```

---

要对列表中所有的元素依次求值，这样当 `xs` 是一个列表的列表，或者有更复杂的数据类型时 `rnf` 保证将它们全部求了出来。

通过对构造器内的数据递归地使用 `seq` 函数，最后就得到了表达式的 NF。使用 `NFData` 类型类中的 `rnf` 函数就可以定义 `deepseq` 了。

---

```
deepseq :: NFData a -> a -> b -> b
deepseq a b = rnf a `seq` b
```

---

使用 `deepseq` 时，也可以定义 `$!!` 运算符来替换。此外，库中还提供了另外一个常用的函数 `force`，它会将一个值计算为 NF 并返回。

---

```
force :: (NFData a) -> a -> a
force x = x `deepseq` x
```

---

到这里，读者应该了解了，在 Haskell 中使用 `seq` 与 `deepseq` 来严格求值是不同的。`seq` 常常只是称为严格求值，而 `deepseq` 称为深度严格求值或者完全严格求值（fully strict evaluation）。这样，可以将表达式计算为不同的形式，使用 `seq` 可以计算为 HNF，使用 `deepseq` 可以将表达式计算为 NF。

## 15.4 求值策略

在 15.1 节中看到，对于同样的表达式，使用不同的求值方法可能在步骤还有计算是否停止的方面有所不同。本节来讨论不同求值策略（evaluation strategy）。

### 15.4.1 引值调用

引值调用（call by value）是编程语言中最常用的调用函数的方法之一。在调用一个参数前，需要先把参数的值求出。在 C 语言中，函数的调用就是这种方法，因为在调用函数前，参数需要被计算好后才能被传入函数，如 `and (length [3] > 0) (null [1,2])`。在 C 语言中就要同时求得两个参数的值。

下面来对函数 `double (average 2 4)` 进行计算，其中 `double` 与 `average` 定义如下：

---

```
add :: Int -> Int -> Int
add = (+)

double :: Int -> Int -> Int
double x = add x x

average :: Int -> Int -> Int
average x y = (add x y) `div` 2

double (average 2 4)
```

---

---

```

double (average 2 4)
= double ((add 2 4) `div` 2)
= double (6 `div` 2)
= double 3
= add 3 3
= 6

```

---

这样，引值调用就是等参数计算为值后才被应用到函数。

### 15.4.2 按名调用

按名调用 (call by name) 这种求值方式是指在应用函数时，不必计算参数的值，而是将参数直接替换到函数体当中。如果这个参数在函数体中没有参与结果的计算，那么它就不会被计算。

---

```

double (average 2 4)
= plus (average 2 4) (average 2 4)
= plus (div (add 2 4) 2) (div (add 2 4) 2)
= plus (div 6 2) (div 6 2)
= plus 3 3
= 6

```

---

上面的函数调用不能演示按名调用的优势。这里再来看一下计算 `and (length [3] < 0) (null [1,2])`。其中，`and` 定义为 `(\a -> \b -> if a then b else False)`。

---

```

and (length [3] < 0) (null [1,2])
= (\a -> \b -> if a then b else False) (length [3] > 0) (null [1,2])
= (\b -> if (length [3] < 0) then b else False)
= if (length [3] < 0) then (null [1,2]) else False
= if (1 < 0) then (null [1,2]) else False
= if False then (null [1,2]) else False
= False

```

---

可以看出，因为 `(null [1,2])` 的值与结果无关，所以它不会被计算。这里它不为上，但就算这个表达式即便它的值为上，这样的估值策略也会返回结果。

### 15.4.3 常序求值

常序求值 (normal order evaluation) 首先从左手边最外侧的不在  $\lambda$  函数定义内部的式子开始化简。比如在  $\lambda$  表达式  $(e_1 e_2)$  进行化简时，当  $(e_1 e_2)$  化简为  $((\lambda x.e) e_2)$  时，即便  $e$  中有一些可以表达式继续化简，我们也先将  $e$  中的  $x$  替换为  $e_2$ ，也就是说，应用函数的步骤发生在对函数抽象化简之前。

如果一个表达式为 HNF 并且它存在  $\beta$ -nf，那么这种方法一定可以将它化简计算为  $\beta$ -nf。这样，使用这种方法计算一定可以得到这个  $\beta$ -nf。常序求值就会从一个表达式的左手边的外侧开始化简，比如：

---

```

(\x.((\y.y) (1+x)) 2
= (\y.y) (1+2)

```

---

---

```
= (1+2)
= 3
```

---

与之对应的另一种求值策略，是从最内侧开始化简。即在应用函数前，将函数体内的表达式先化简。

---

```
(λx.((λy.y)(1+x)) 2
= (λx.(1+x)) 2
= (1+2)
= 3
```

---

但是在由内向外化简的时候，比如化简  $(\lambda y. y)(1+x)$  时，表达式中的  $x$  为游离变量，所以可能产生命名冲突的问题。这里虽然并未考虑这个问题，但是在真正实现时它会带来一些麻烦。

使用常序求值这种方式在计算 double 函数时会比引值调用多很多的步骤。

---

```
double (average 2 4)
= plus (average 2 4) (average 2 4)
= plus (div (add 2 4) 2) (average 2 4)
= plus (div 6 2) (average 2 4)
= plus 3 (div (add 2 4) 2)
= plus 3 (div 6 2)
= plus 3 3
= 6
```

---

## 15.5 惰性求值

之前讨论了很多内容，相信读者到这里对惰性求值 (lazy evaluation) 与严格求值 (strict evaluation) 有了一定的了解。惰性求值不是一种固定的求值策略，而是结合了多种求值策略的优势的求值方法。当结果为 WHNF 计算时就会停止，同时，使用 thunk 来代表还不需要的结果。对于相同的参数会被相同的 thunk 替代，这使得计算函数的时候能避免重复的化简，比如：

---

```
> let x = (1+1)
> :sprint x
x = _
> let y = (x,x)
> :sprint y
y = (_,_)
> x
2
> :sprint y
(2,2)
```

---

从上面的测试我们可以看到，对  $x$  求值一次那么元组  $y$  内的两个  $x$  值都被求出，这里  $x$  是被共享的。用这种方法就能够使用相对较少的步骤来计算出一个表达式的值，与此同时计算可以被尽可能地延后。通过这种方式我们就能够在无穷的数据结构上进行计算，比如无穷的列表、

数据流等。惰性求值是 Haskell 的一个非常重要的特性，但有很多例子说明它不总是好的，所以我们需要理解它才能正确地使用它。

事实上把这种求值方式称为惰性求值实际上不是很准确，更为正式的说法应该是非严格求值（non-strict evaluation）。实现它的一个重要手段是图的化简（graph reduction），但是本书不会讨论它。相关的内容可以参阅（Peyton Jones, 1987）中的第二部分。

## 15.6 严格模式匹配与惰性模式匹配

当我们使用模式匹配或者 case 关键字定义函数时，当所匹配的形式为值时，它会被严格地计算，因为如果不被严格地计算出来就无法进行匹配。例如，Haskell 中的 `&&` 可以定义为：

---

```
(&&) False b = False
(&&) True b = b
```

---

当调用 `(&&)` 时，它的第一个参数一定要被严格地计算以便于匹配不同的定义。之前我们了解到 `undefined` 函数在 Haskell 可以理解为一种错误：

---

```
> undefined
*** Exception: Prelude.undefined
```

---

而 `const` 函数只返回第一个参数作为结果，它不会去计算第二个参数，所以即便如此还是可以返回结果。这就是惰性求值的另一个简单体现。

---

```
> const 0 undefined
0
```

---

这里的逻辑与 `(&&)` 也是如此，计算时如果不需要是不会去碰第二个参数的，比如：

---

```
> False && undefined
False
```

---

由于类型已经检查过了，第二个值的类型为布尔类型，因此 `&&` 运算不必再匹配第二个参数，直接可以返回 `False`。可是，当将两个参数调换一下，理论上也可以得到 `False`，但结果却是异常。

---

```
> undefined && False
*** Exception: Prelude.undefined
```

---

是否可以在 Haskell 中定义一个纯函数的逻辑与运算，满足 `False && undefined` 和 `undefined && False` 的结果都为 `False` 呢？答案是否定的，无法定义一个这样的纯函数。因为在使用模式匹配的时候总是一个在前一个在后，没有“同时”这种概念，而对于它的证明需要一些更深入的  $\lambda$  演算的内容，这里就不讨论了。这样，不能对所有的参数都保证为惰性的，也就是说这种懒惰都是有限度的。

之前，要对一个参数进行严格计算可以使用 `seq` 函数，但是连续地使用 `seq` 函数有时是略

显麻烦的。Haskell 提供了叹号形式 (BangPatterns) 来严格地匹配参数，它是模式匹配的一个扩展，需要使用 BangPatterns 编译器参数。

函数 `f x = True` 为常函数，`x` 的值可以不必计算。但是，如果需要计算 `x` 的值，可以定义为 `f !x = True`。这种定义与使用 `seq` 是等价的，即 `f x = seq x True`。因此，使用 `!` 来匹配一个形式时，表达式在匹配形式时会首先被计算为 HNF。`f (x, y) = True` 与 `f !(x, y)` 是等价的，因为将 `(,)` 理解为 `Pair x y`，这里无论 `x` 或者 `y` 的值是否被计算，根据定义它们都为 HNF 与 WHNF。而 `f !(x, y) = True` 则会严格地将 `x` 求出。同理，在定义函数时，如果读者理解了 HNF 与 WHNF，匹配 `[x, y, z]` 与 `![x, y, z]` 也没有区别。

Haskell 中除提供严格模式匹配外还提供惰性模式匹配 (lazy pattern match)，它通过在需要匹配的形式前加 `~`，有时也称波浪字符模式 (tilde pattern)。当使用模式匹配的时候，实际上匹配的是数值的构造器，而惰性形式如果不匹配则不会匹配对应的构造器。例如：

---

```
> let f (x,y) = 5 in f undefined
*** Exception: Prelude.undefined

> let f ~(x,y) = 5 in f undefined
5
```

---

可以看到，使用了 `~` 后 `undefined` 可以将 `(,)` 构造器匹配为 `undefined` 了，而不会得到 `f ⊥` 上的情形。但是需要注意的是，如果使用了惰性模式匹配，那么在匹配的时候构造器不会被计算，所以，当类型的值由多种形式定义并且全部使用惰性模式匹配时，只有第一个会被匹配。例如：

---

```
head' :: [Int] -> Int
head' [] = 0
head' ~(x:xs) = x
```

---

无论参数是什么，结果都将为 0：

---

```
> head' [1,2,3]
0
```

---

这一章的内容主要参阅了 Peter Sestoft 发表的《Demonstrating  $\lambda$  Calculus Reduction》，可以在 <http://www.itu.dk/people/sestoft/papers/sestoft-lamreduce.pdf> 中进行在线阅读。此外还参考了 (Peyton Jones, 1987) 的第 2 章与第 11 章。

# 第 16 章

## 并行与并发编程

并行 (parallel) 与并发 (concurrent) 编程一直是一个研究比较活跃的领域。如今的计算机基本上已经都是双核心或者更多核心的处理器了，大型的服务器集群也需要同时处理多个任务。所以，如何能够高效地同时处理多个程序并让它们很好地互相配合，一直都是在编程中很关心的话题。

如果要了解并行与并发编程，那么需要了解一些基本概念，首先要了解的概念就是进程 (process)。进程是操作系统层面的抽象，它代表了程序运行的指令、内存单元以及当前的程序状态等。另外一个概念为线程 (thread)，它是进程层面的抽象，代表了一个程序的运行可能需要同时执行多个任务。进程在运行时，内存分配受到操作系统的保护，这样，如果一个进程出现崩溃，则不会影响另一个进程。但是，进程间协作时，如果需要共享内存时就会不够灵活。而线程则隶属于进程，没有独立的内存空间，所以在共享时协作时需要更为灵活。

下面再来看一下什么是并行与并发。并行与并发是指同时处理多个进程或者线程的过程，但它们的概念是不同的。并行的概念是多处理器的不同核心同时处理多个进程或者线程，计算过程是严格意义上的同时进行。这样做主要是为了更好地利用多 CPU 核心的硬件资源。而并发指的则是单个处理器同时处理多个进程或者线程的过程。这里的同时指的是进程或者线程在处理器中快速切换，对于每个进程或线程只运行一小部分的时间，所有进程和线程在 CPU 中快速切换以至于用户感觉是同时的，进程或，线程中切换的过程称为调度 (scheduling)。并发的主要在于某个程序在等待输入/输出或者其他信号时，除他以外的程序也有机会在 CPU 中运行而不会因中止导致浪费时间与 CPU 资源。在现在的计算机语言中，平行与并发的概念已经并不是那么的清楚了，因为解决并行与并发计算使用都是几乎相同的机制。

并行与并发地运行程序时要解决几个关键问题，其一是死锁，其二是要保证公平性，避免线程饥饿 (starvation)。死锁是并行与并发系统中最常见的问题。由于不能决定 CPU 什么时候对线程进行调度，因此，在共享内存时，线程之间对内存的操作需要保证是互斥的 (mutual exclusive)，即线程 1 在操作一段共享内存时，线程 2 是不能对这段区域操作的，这样就需要对

内存加锁，因为若不加锁，那么线程 1 未操作完毕的内存单元可能会被其他线程修改了，这样则可能会得出不正确的结果。当线程 1 与线程 2 都需要对同样两段内存操作时，线程 1 对第一段加锁后打算对第二段加锁，而此时，线程 2 已经对第二段加锁并且打算对第一段加锁，两个线程谁也不肯放弃。这样的情形就为互斥等待，也就导致了死锁。公平性所指的是在多个线程中切换的时候，要保证没有线程因遗漏而一直不被运行。一个简单的例子就是汽车过窄桥的问题，桥是共享的资源，由于桥很窄，一次只能过一辆车，右边的车要过到左边，左边的车要过到右边，因为不可以出现只有一边的车在桥上通过的情形，更不可以出现撞车。

很多语言通过锁、线程监听器来达到互斥与协调线程的目的。可是，通过这些方法往往很难非常高效地复合多个处理并行与并发计算的函数，并且，由于副作用与多线程产生的不确定性给并行与并发程序正确性的证明与寻找程序的错误带来了较大的难度。同时，这些方法的可扩展性也许要差一些，当线程非常多时，运行效率就会明显下降，即便增多 CPU 核心的数量性能也不会有明显提升。而在前面的章节中提到，由于 Haskell 是纯函数式编程语言，这个特性使用它在编写并发程序上有着其他很多语言所没有的优势。第一个优势就是确定性的并行 (deterministic parallel) 计算，无副作用函数的计算可以任意地并行；此外，GHC 还提供了轻量级线程来减少并行与并发计算时需要付出的代价。另外还有复合性良好、使用简单并且易于证明的软件事务内存 (STM)。最后需要明确的一点是，使用多线程并不一定永远是好的。如果单一进程能够在可接受的时间内完成任务，那么也未尝不是非常好的方法，因为这样就不必花时间去编写更为复杂的并发与并行的程序。

## 16.1 确定性的并行计算

确定性并行计算主要受益于 Haskell 为纯函数式语言的特性。因为纯函数的计算互相没有影响，可以任意并发。它需要并行 Monad 库，如果读者不能在 GHCi 中载入 Control.Monad.Parallel 库，说明这个库没有安装，可以使用如下的命令进行安装：

```
C:\> cabal update
Downloading the latest package list from hackage.haskell.org

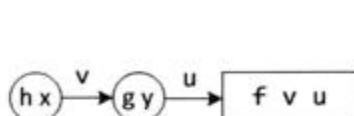
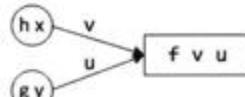
C:\>cabal install monad-par
...
Registering monad-par-0.3..
```

Haskell 中的纯函数是无副作用的，需要同时计算两个纯函数的时候也就不必要对其加锁。但是，同样由于函数为纯函数，因此不必在计算时候共享数据。这种方法非常简单，因此可以不必进行过多的修改就可以提高程序的效率。

比如，在计算  $f(h(x))(g(y))$  的结果时，可以严格地计算  $h(x)$ ，然后严格地计算  $g(y)$ ，最后应用到函数  $f$ ，如图 16-1 所示。

但是，当  $hx$  与  $gy$  的计算都需要较长时间时，计算  $hx$  的值后需要等待  $gy$  的计算结果才能

应用函数  $f$ , 这样就浪费了一些时间。如果函数  $h$  与  $g$  均为纯函数, 那么由于它们的计算而互不影响, 因此没有必要互斥或者加锁, 可以直接并行计算, 如图 16-2 所示。而程序会何时运行这两个函数, 哪个在先, 哪个在后是不知道的, 因为纯函数计算的顺序是不影响结果的。这样, 计算得到的结果总是确定的, 因此称为确定性并行计算。

图 16-1 依次按顺序计算  $f$  的参数图 16-2 并行计算  $f$  的参数

简单地说, 并行 Monad 就是为了解决这一问题而设计的库。下面来了解一下这个库中都定义了哪些内容。

类型 `Par a` 代表了并行的计算。由于构造器只有一种形式且计算应当没有副作用, 因此可以定义 `runPar` 函数来得到并行计算的结果。

---

```

newtype Par a
 runPar :: Par a -> a

```

---

`Par` 类型分别实现了函子、`Applicative` 以及 `Monad` 类型类。下面来看一下计算的结果是如何传递的。

在并行计算的过程中, 程序员需要得到并行计算的结果并输入到其他函数中继续计算。为此, `Parallel` 中定义了一个容器, 以容器来传递函数计算的结果, 称为 `IVar`<sup>①</sup>。`IVar` 使用 `IORef` 定义, 在计算时实际上对内存进行了操作, 加快了运行的速度。

---

```

data IVar a = IVar (IORef (IVarContents a))

```

---

`IVar` 的构造器为私有的, 提供 4 个操作 `IVar` 的函数, 如表 16-1 所示。

表 16-1 操作 `IVar` 的相关函数

函数	作用
<code>new :: Par (IVar a)</code>	新建一个空的 <code>IVar</code> 容器
<code>newFull :: NFData a =&gt; a -&gt; Par (IVar a)</code>	新建一个 <code>IVar</code> 容器并把数据存入
<code>get :: IVar a -&gt; Par a</code>	从 <code>IVar</code> 中获得其中的值
<code>put :: NFData a =&gt; IVar a -&gt; a -&gt; Par ()</code>	向 <code>IVar</code> 中存储一个值

这里需要注意的是, 使用 `newFull` 与 `put` 时, 这个容器中所有的值必须实现 `NFData` 类型类, 因为在并行计算的时候不希望计算结果被惰性求值拖延, 而是在线程中计算完成, 所以想进行确定性并行计算的类型就需要实现 `NFData` 类型类。向一个 `IVar` 中写入一个计算后, 就再

<sup>①</sup> Control.Monad.Par API 文档中的原文为 Information flow is described using variables called IVars.

也不能使用 `put` 了，否则会引起程序错误。这样做是为了保证 `Par Monad` 的纯洁性。下面来构造一个很慢的计算，这个计算在多线程计算时会明显提高速度。比如，同时验证两个数是否是素数，如果是就返回这两个素数的乘积，否则返回 0。

首先，写出定义过的 `isPrime` 函数：

---

```
import Control.Monad.Parallel
isPrime :: Integer -> Bool
isPrime 2 = True
isPrime p = p > 1 && (all (\n -> p `mod` n /= 0) $ takeWhile (\n -> n * n <= p) [3, 5 ..])
```

---

给定两个整数，先并行地分别判断它们是否为素数，如果为素数，返回它们的乘积；如果不是，返回 0。

---

```
generateKey :: Integer -> Integer -> Integer
generateKey x y = runPar $ do
 [k1, k2] <- sequence [new, new]
 fork $ put k1 (isPrime x)
 fork $ put k2 (isPrime y)
 p1 <- get k1
 p2 <- get k2
 return $ if p1 && p2 then x * y else 0
```

---

这里给定两个比较大的素数，素数的检查是纯函数并且会很耗时，所以用多核心来判断两个素数就一定要比单核心要快一些。

---

```
p1, p2 :: Integer
p1 = 2646507710984041
p2 = 1066818132868207

main = do
 print $ generateKey p1 p2
```

---

如果需要使用多核心计算，那么在使用 GHC 编译的时候有所不同。这里先来了解一下 GHC 的运行时系统（runtime system，RTS）。GHC 的运行时系统是通过 C 与 C--<sup>①</sup>两种语言语言编写的，它的主要用途是为 Haskell 程序运行提供一个平台，主要包括内存管理、垃圾回收、并行计算、线程调度等。GHC 的运行时系统默认只支持一个 CPU 核心，但是在每一个 CPU 核心上运行是支持多线程并发的。但是可以用 `-threaded` 参数在编译时，告诉运行时系统这个程序可能在多核心上运行，这样运行时系统就可以在多 CPU 核心上管理多个系统线程。

下面再来了解一下运行时系统的选项。编译时使用 `-rtsopts`，可以在运行时通过 `+RTS` 与 `-RTS` 从命令行给运行时系统一些选项。如果在运行时使用了 `-s` 参数，就会得到一些程序运行时的数据，包括内存空间的分配、初始化时间、运行时间等。使用 `-N[n]` 可以指定使用  $n$  个 CPU 并行地运行的程序<sup>②</sup>。GHC 的运行时系统会将 `+RTS` 与 `-RTS` 之间的部分作为运行时参数处理。

① C--是一门可移植的汇编语言，更多关于它的内容可以登录 <http://www.cminusminus.org/>。

② 想要了解更多编译参数与运行系统参数的内容读者可以参阅《The Glorious Glasgow Haskell Compilation System User's Guide》第 4 章第 14 节。

其他的则仅仅当做程序参数处理。

```
C:\> ghc --make Main.hs -O2 -rtsopts -threaded
[1 of 1] Compiling Main (Main.hs, Main.o)
Linking Main.exe ...

C:\> .\Main.exe +RTS -s
> .\Main.exe +RTS -s
2823342414853307021661733284487
Main.exe : 9,501,199,292 bytes allocated in the heap
 1,110,713,976 bytes copied during GC
 298,796,760 bytes maximum residency (10 sample(s))
 37,676,908 bytes maximum slop
 618 MB total memory in use (0 MB lost due to fragmentation)
 Tot time (elapsed) Avg pause Max pause
Gen 0 18338 colls, 0 par 2.71s 2.46s 0.0001s 0.0058s
Gen 1 10 colls, 0 par 2.54s 2.93s 0.2930s 1.5338s
Parallel GC work balance: nan (0 / 0, ideal 1)

 MUT time (elapsed) GC time (elapsed)
Task 0 (worker) : 32.79s (33.63s) 5.24s (5.39s)
Task 1 (worker) : 0.00s (39.02s) 0.00s (0.00s)
Task 2 (bound) : 0.00s (39.02s) 0.00s (0.00s)

SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)
INIT time 0.00s (0.01s elapsed)
MUT time 32.79s (33.63s elapsed)
GC time 5.26s (5.39s elapsed)
EXIT time 0.00s (0.00s elapsed)
Total time 38.05s (39.03s elapsed)
Alloc rate 289,746,590 bytes per MUT second
Productivity 86.2% of total user, 84.0% of total elapsed

gc_alloc_block_sync: 0
whitehole_spin: 0
gen[0].sync: 0
gen[1].sync: 0
```

下面来看运行时系统都给出了哪信息<sup>①</sup>。

- `xxx bytes allocated in the heap` 指的是在程序运行的过程中所分配的字节数，这个数字在运行过程中可能随着不段地分配内存而不断增加。
- `xxx bytes maximum residency (10 sample(s))` 指的是内存使用的最大值，在计算这个值的时候考虑了垃圾回收器（garbage collector）回收的部分。因为这个结果是被通过使用垃圾回收器采样而得出的，所以可能不准确，后面的 10 指的就是采

<sup>①</sup> 关于 GHC RTS 的垃圾回收器实现与性能测量方法就不讨论了，读者可以参阅 Simon Marlow、Simon Peyton Jones 和 Tim Harris 等写的“Parallel Generational-Copying Garbage Collection with a Block-Structured Heap”，可在线阅读 (<http://community.haskell.org/~simonmar/papers/parallel-gc.pdf>)。

样次数。

- `xxx bytes maximum slop`, 运行时系统在分配内存时是以块来分配的, Slop 指的剩余是在块的尾部由于容量不足而不能被使用而浪费的内存空间。
- INIT、MUT、GC、EXIT、Total 分别后面跟了两个时间, 下面依次来讨论一下它们。第一个为 CPU 运行时间, 而在括号内的为消逝的真实时间 (elapsed real time)。CPU 运行时间指的是程序在 CPU 中运行所经过的时间, 不包括 IO 等待、内存或者磁盘读取等其他延时, 仅仅是 CPU 指令在 CPU 中运行过程中消耗的时间, 当有多 CPU 核心时, 这个时间为所有 CPU 运行该程序所用 CPU 时间的总和。而消逝的真实时间是指一个程序从开始到结束所花的时间, 这个时间也称为时钟时间 (wall clock time)。它就是我们感知的程序运行时间, 包括了所有的 IO 其他信号的等待还有并发调度其他线程的时间。

如图 16-3 所示, 线程 P1 的 CPU 时间为  $t_1+t_5$ , 而时钟时间为  $t_1+t_2+t_3+t_4+t_5$ 。那么, 如果给定 2 个 CPU 核心, 一个并行运行程序的两个线程在两个核心上同时开始工作并且工作量是相等的, 若没有任何信号等待与其他线程并发所引起的延迟, 那么 CPU 时间应该为时钟时间时间的 2 倍。所以, 在比较程序运行效率的时候需要比的是总的时钟时间。使用多线程时, 需要保证让各个线程的工作量不会较少, 因为使用多线程会产生很多额外的费用。如果任务量较小使用单线程就可以了; 如果函数需要多个线程的计算结果, 那么尽力让各个线程的工作量相等, 以保证线程间互相等待的时间不会过长。

P1运行	P1等待	P2	P3	P1完成
$t_1$	$t_2$	$t_3$	$t_4$	$t_5$

图 16-3 线程运行时间示意图

再回到这些运行时系统输出的信息。INIT 所指的是程序的初始化时间, MUT 指的是程序运行代码需要的时间, GC 为垃圾回收器所用的时间, EXIT 表示程序 GHC 的运行时系统关闭所用的时间, Total 为总时间。在比较性能时, 需要比较的就是 Total 中的时钟时间。

下面使用-N2 来给这个程序提供两个 CPU 核心, 看看效率上是否有所提升。

```
C:\> .\Main.exe +RTS -s -N2
2823342414853307021661733284487
Main.exe : 10,088,993,772 bytes allocated in the heap
 1,197,924 bytes copied during GC
 30,652 bytes maximum residency (1 sample(s))
 30,272 bytes maximum slop
 2 MB total memory in use (0 MB lost due to fragmentation)
 Tot time (elapsed) Avg pause Max pause
Gen 0 12220 colls, 12219 par 0.34s 0.30s 0.0000s 0.0096s
Gen 1 1 colls, 1 par 0.00s 0.00s 0.0002s 0.0002s

Parallel GC work balance: 1.00 {279141 / 278458, ideal 2}
 MUT time (elapsed) GC time (elapsed)
```

```

Task 0 (worker) : 13.35s (19.75s) 0.25s (0.56s)
Task 1 (worker) : 0.00s (20.31s) 0.00s (0.00s)
Task 2 (bound) : 0.00s (20.31s) 0.00s (0.00s)
Task 3 (worker) : 19.58s (19.96s) 0.17s (0.35s)

SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)
INIT time 0.00s (0.00s elapsed)
MUT time 33.03s (20.01s elapsed)
GC time 0.34s (0.30s elapsed)
EXIT time 0.00s (0.00s elapsed)
Total time 33.37s (20.31s elapsed)
Alloc rate 305,491,839 bytes per MUT second
Productivity 99.0% of total user, 162.6% of total elapsed

gc_alloc_block_sync: 149
whitehole_spin: 0
gen[0].sync: 0
gen[1].sync: 0

```

从上边的例子可以看到，单 CPU 核心时，程序耗费的时钟时间为 39.03 s；而使用了两个 CPU 核心时，时钟时间为 20.31 s，效率大约提升了 48%<sup>①</sup>。在使用至强 CPU 的小型工作站上运行时，单 CPU 核心的时间为 5.91 s，而 2 个核心则只要 2.63 s，效率提升大约为 55%<sup>②</sup>，有两个核心同时来判断素数时间当然会大约提高一倍。当然，这个结果会与计算机硬件、进程优先级与 CPU 当前负载等因素有很大关系，同时也存在偶然性，读者也可以多运行几次来得到更可靠的结果。

GHC.Conc 库中提供了一些基本的控制并发的函数，这里先介绍简单的几个。`numCapabilities :: Int` 会得到在运行时系统+RTS `-N[n]` 参数中指定的 CPU 核心个数，但是，因为运行时使用的 CPU 核个数 `n` 可能会不同，所以不是严格的纯函数，建议库中使用 `getNumCapabilities :: IO Int` 函数来得到程序当前可以使用 CPU 核心数量<sup>③</sup>。可以通过 `getNumProcessors :: IO Int` 来得到当前计算机有的 CPU 数量<sup>④</sup>。`setNumCapabilities :: Int -> IO ()` 可以设置当前程序运行所使用的核心数量<sup>⑤</sup>。

并行运行时并不是线程越多越好，因为使用线程会产生额外的负担，同时性能也与 CPU 核心的数量有关。读者应该还记得斐波那契数列与归并排序，它们都使用的是分治算法，这类算法有个好处就是很容易地在多个核心上使用，现在在这里我们实现它的多线程的版本。

首先是斐波那契数列，只需要通过一个 CPU 核心来求得 `fib (n-1)`，另一个核心求得 `fib (n-2)`，然后将其相加。

---

```

parfib :: Int -> Par Int
parfib n | n <= 2 = return 1

```

---

<sup>①</sup> 测试的计算机使用的是 Intel(R) Core(TM) i7 CPU L640 2.13GHz 低电压移动平台处理器。

<sup>②</sup> 测试的计算机使用的是 Intel(R) Xeon(R) CPU E5630 2.53GHz 小型工作站处理器。

<sup>③</sup> 对于使用超线程技术(Hyper-Threading Technology(HTT))的 CPU，返回的数值为逻辑核心数量而不是物理核心数量。

<sup>④</sup> 同脚注 3，若 CPU 使用了 HTT 技术返回的值为逻辑 CPU 核心数，而非物理 CPU 核心和数量。

<sup>⑤</sup> 同脚注 3，若 CPU 使用了 HTT 技术，则设置的值为逻辑核心数；若多于逻辑核心则程序运行时会报错。

---

```

| otherwise = do
 x <- spawn $ parfib (n-1)
 y <- spawn $ parfib (n-2)
 x' <- get x
 y' <- get y
 return (x' + y')

```

---

这里的 `spawn :: NFData a => Par a -> Par (IVar a)` 函数定义为：

---

```

spawn p = do r <- new
 fork (p >>= put r)
 return r

```

---

它仅仅是新建一个线程来计算，然后存在 `ivar` 容器中返回。同理，归并排序也是这样的：将一个列表分成两组，第一组用一个 CPU 核心来排序，第二组用另外一个，然后将组合合并起来。

---

```

parmsort :: [Int] -> Par [Int]
parmsort [] = return []
parmsort xs = let l = length xs `div` 2 in
 do
 m1 <- spawn $ parmsort $ take l xs
 m2 <- spawn $ parmsort $ drop l xs
 m1' <- get m1
 m2' <- get m2
 return (merge m1' m2')

merge :: [Int] -> [Int] -> [Int]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) | x > y = y:merge (x:xs) ys
 | otherwise = x:merge xs (y:ys)

```

---

但是它们的效率非常低，如果数字很大或者列表过长，那么可能要低几十倍左右。因为这里递归地创建线程来计算，一个函数调用会创建两个线程，而创建线程的代价虽然很低，但也不是没有。如果过多反而会拖慢程序的速度。

可以加一个限制条件，当需要解决的问题足够小时，就不再继续并行计算了。这样就可以定义一个高阶函数来同时解决斐波那契数列与归并排序问题。

---

```

import Data.List (genericLength, sort)
import Control.Monad.Par

divAndConq :: (NFData sol, Integral size)
 -> (prob -> size) -- 量度(measure)问题大小的函数
 -> (size -> Bool) -- 停止创建更多线程条件(condition)
 -> (prob -> [prob]) -- 分解问题的函数(divide)
 -> (sol -> sol -> sol) -- 合并结果的函数
 -> (prob -> sol) -- 单(single)线程解决问题的函数
 -> (prob -> sol)

```

```

divAndConq m c d f sg prob = runPar $ complete prob
 where complete p | c.m $ prob = return $ sg p
 | otherwise = do
 let probs = d p
 let subsolve = map (spawn.complete) probs
 results <- sequence (map (get.runPar) subsolve)
 return $ foldl1 f results

divFib c n = divAndConq id (< c) (\m -> [m-1,m-2]) (+) fibonacci n

```

这样，当要计算的数  $n$  比  $c$  小时，我们直接使用一个线程进行计算，否则生成多个线程。

```

fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n-1) + fibonacci (n-2)

```

同样，在定义归并排序的时候，如果列表的长度小于我们给定的 size，那么我们则使用单线程进行计算；否则将列表分成两个，分别生成两个线程计算。

```

divMsort size xs = divAndConq genericLength (<size)
 (\ls -> let m = div (length xs) 2
 in [take m ls, drop m ls])
 merge sort xs .

```

本节学习了如何使用 Par Monad 来对纯函数进行并行计算，还讨论了如何使用 GHC 的运行时系统来评估并发程序的性能，还定义了一个高阶函数将来提升分治算法性能。确定性并行计算的函数都是纯函数，不会处理跟 IO 有关的效应。由于纯函数只是做计算，因此，一般这样的程序都是计算密集型的，也就是说，这个计算的全部都一定需要占用 CPU<sup>①</sup>资源。其他有一些任务是非常容易分为多个 CPU 核心处理来提高性能的，例如：映射一个函数从列表[a]~[b]，如果列表很长，则可以考虑使用多核心来运行。Control.Monad.Par 库中提供了相应的函数来完成，如 parMap、parMapReduceRangeThresh 等，读者可以参阅 API 文档了解这些函数。

本节的内容主要参考了（GHC Team, 2012）的 7.22 节，可以在 [http://www.haskell.org/ghc/docs/7.4.2/users\\_guide.pdf](http://www.haskell.org/ghc/docs/7.4.2/users_guide.pdf) 中下载这本参考手册。

### 练习

1. 我们编写星际王词典时，如果单词在词典里是乱序的，那么需要查询整个词典，同样也可以用这种方法来进行多线程的查询，返回一个 Maybe WordIdx 类型后再用 Applicative 中的 `<|>` 运算符合并结果即可。读者可以自己试着定义一下（如果一个线程找到了结果，那么其他线程可以停止查询了。然而，因为使用确定性的并行计算时这些线程无法共享信息，所以无法知道其他线程进行的情况，了解下节内容后可以解决这个问题）。
2. 读者可以随机产生一些数然后对它们进行排序，先看一下单线程的效率，然后再比较一下使用 `divAndConq` 的效率。

<sup>①</sup> 不考虑 GPU 计算。

## 16.2 轻量级线程

GHC 中提供了产生轻量级线程 (light weight thread) 的函数 `forkIO :: IO () -> IO ThreadId`。为什么称 GHC 中的线程为轻量级呢？因为使用 `forkIO` 来产生新的线程与调度这些线程所需要的代价很低，产生一个线程所需要的内存代价大约在几 KB 左右。调度这些线程的工作全部由 GHC 的运行时系统来完成，不需要操作系统提供任何的支持。由于这些原因，它有很好的伸缩性，在一台普通的计算机上可以很容易地创建数百万级别的线程，同时，当不需要的时候也非常容易地回收它们。

### 16.2.1 调度的不确定性

`Control.Current` 库中提供了 `forkIO :: IO () -> IO ThreadId` 函数来创建一个新的轻量级的线程，它会返回一个线程的 Id，读者可以将这个 `ThreadId` 类型理解为一个整数。这样，创建一个线程就可以同时做多件事情了。

---

```
--Main.hs
import Control.Concurrent
import Control.Monad
import System.IO

main :: IO()
main = do
 hSetBuffering stdout LineBuffering
 forkIO $ replicateM_ 20 (putStrLn "Thread 1 is printing")
 forkIO $ replicateM_ 20 (putStrLn "Thread 2 is printing")
 return ()
```

---

这样，就一定会有 40 行的打印结果，但是打印 1 与打印 2 的顺序是不确定的。还有可能在 `putStrLn` 执行的过程中线程就被调度了，所以，在 GHCI 中可能输出下面这样的结果多次：

---

```
ThTrheraeda d1 2i si sp rpirnitnign
G
```

---

`putStrLn` 不是原子的操作，理论上会出现这种情形，但这只是 GHCI 中的情形。使用 `runghc` 或者编译后再未遇到这种情况，这可能与 `runghc` 跟 `ghc` 的运行时系统有关。这里也侧面反映了使用多线程就会使程序变得复杂，不确定性难以处理。

如果在命令行中运行编译 `Main.hs`，每次得到的结果会不一样，不但打印的顺序有先有后，而且打印的个数也可能不等。打印的顺序不一样是因为 CPU 调度线程是相对随机的。多个线程共享一个命令行，第一个线程打印几次，然后第二个线程打印几次，这就是并发的意义。个数不等是因为这里程序的主线程不会等待两个子线程，即主线程结束则整个程序结束。

### 16.2.2 基本线程通信

最基本的线程通信是使用一个容器类型，它称为 MVar (Mutable Variable)。同 IVar 十分类似，MVar 也是一个容器，只能存储一个值，它有两个状态，为空或者为满。库中提供的函数如表 16-2 所示。

表 16-2 操作 MVar 的相关函数

函 数	作 用
<code>newEmptyMVar :: IO (MVar a)</code>	新建一个空的 MVar 容器
<code>newMVar :: a -&gt; IO (MVar a)</code>	新建一个 MVar 容器并把数据存入
<code>takeMVar :: MVar a -&gt; IO a</code>	取出 MVar 中的值，若为空，则该线程会等待
<code>putMVar :: MVar a -&gt; a -&gt; IO ()</code>	向 MVar 中存储一个值，若为满，则该线程会等待
<code>tryTakeMVar :: MVar a -&gt; IO (Maybe a)</code>	试探性地从一个 MVar 容器中取值，若为空，则不会导致线程等待
<code>tryPutMVar :: MVar a -&gt; a -&gt; IO Bool</code>	试探性地向一个 MVar 容器中存值，若为满，则不会导致线程等待
<code>swapMVar :: MVar a -&gt; a -&gt; IO a</code>	取出 MVar 中的值，然后存入另外一个值，它是通过 <code>takeMVar</code> 与 <code>putMVar</code> 定义的
<code>readMVar :: MVar a -&gt; IO a</code>	从一个 MVar 中读值，也是通过 <code>takeMVar</code> 与 <code>putMVar</code> 定义的
<code>isEmptyMVar :: MVar a -&gt; IO Bool</code>	判定一个 MVar 容器是否为空
<code>threadDelay :: Int -&gt; IO ()</code>	线程会等待给定的时间，单位为微秒，这里不能保证在等待结束后线程马上会被调度运行
<code>yield :: IO ()</code>	建议当前线程放弃对 CPU 的占用
<code>killThread :: ThreadId -&gt; IO ()</code>	终止给定的线程

这里需要注意的是，`putMVar` 不像 `IVar` 那样存储的是数据的 NF，它存储的是惰性的值。如果不需，`MVar` 中的值不会被计算。但是，计算为惰性还是严格的可以由用户自己来决定的。`swapMVar` 与 `readMVar` 在执行的时候可能会被干扰，因为线程的调度是不确定的。例如，`readMVar` 是通过下面的方法定义的：

```
readMVar :: MVar a -> IO a
readMVar m = do
 a <- takeMVar m
 -- 注释
 putMVar m a
 return a
```

这样，如果另外一个线程在注释的位置被调入 CPU 中并且恰好会向该 `MVar` 中写入值，那么这两个线程对同一个 `MVar` 的操作就不为互斥了。GHC 的运行时系统保证了公平性，当有多

个线程同时需要访问一个 MVar 时，它们会排好队，当操作可以进行时，第一个等待的线程会第一个执行，然后依次类推，这样的数据结构称为队列（queue），这样的入队与出队顺序称为先进先出（first in first out, FIFO）。这样，公平性有了保证，虽然如果 MVar 使用得当，可以当做一种互斥的机制使用，但是它是无法避免死锁的。

本章开始提到过一种互斥等待的情形，使用 MVar 是不可能完全避免的。如果两个线程同时要取得两个 MVar 中的值，线程 1 取得第一个，线程 2 取得第二个，两个线程互相等待对方的 MVar，那么就引起了死锁。虽然 MVar 会引起死锁，但是可以用它实现简单的线程通信来传递一些计算结果还有互斥。当需要从多个 MVar 中读取值且还需要避免死锁时，最好不要使用轻量级线程，推荐使用下节将学的软件事务内存。

如果需要主线程等待子线程，那么只需要在主线程处创建一个空 MVar，在子线程运行结束后使用 putMVar 存入一个值，然后在主线程的末尾读就可以了。

---

```
main :: IO ()
main = do
 hSetBuffering stdout LineBuffering
 a <- newEmptyMVar
 forkIO $ do
 replicateM_ 20 (putStrLn "Thread 1 is printing")
 putMVar a ()
 b <- newEmptyMVar
 forkIO $ do
 replicateM_ 20 (putStrLn "Thread 2 is printing")
 putMVar b ()

 takeMVar a
 takeMVar b
 return ()
```

---

读者应该可以看到创建一些让主线程等待的子线程时的公共的部分，实际上就是创建一个新的空 MVar，然后创建一个子线程，在子线程结束后将随便一个值存入，然后在主线程中使用 takeMVar 进行等待就可以了。

下面，定义 startThread 函数来创建一个新的线程后返回 (ThreadId, MVar a) 类型，元组中的第二个元素就是用来被等待的 MVar 容器。

---

```
startThread :: IO a -> IO (ThreadId, MVar a)
startThread work = do
 var <- newEmptyMVar
 tdid <- forkIO $ do
 w <- work
 putMVar var w
 return (tdid, var)
```

---

下面写 wait 函数，给定一个 (ThreadId, MVar a)，等待 MVar 中的值。

---

```
wait :: (ThreadId, MVar a) -> IO a
wait = takeMVar.snd
```

---

这样，之前的 main 函数就可以定义为：

---

```
main :: IO ()
main = do
 hSetBuffering stdout LineBuffering
 tdinfo1 <- startThread $ replicateM_ 20 (putStrLn "Thread 1 is printing")
 tdinfo2 <- startThread $ replicateM_ 20 (putStrLn "Thread 2 is printing")
 wait tdinfo1
 wait tdinfo2
 return ()
```

---

也可以进一步地抽象这些函数，将 startThread 与 wait 整合起来，定义一个线程管理器来管理线程。

首先，线程可以分为两种，一种是守护进程（daemon），它不是程序非常重要的部分，当主程序结束时，守护进程也就结束了；另一种是非守护进程（non-daemon），只有当所有的非守护进程结束后，整个程序才能结束。GHC 的库中默认 forkIO 创建的线程都是守护进程。首先来定义它们：

---

```
data ThreadType = Daemon | NonDaemon deriving (Show, Eq, Ord)
```

---

下面定义线程管理器。可以定义一个 Map ThreadId (MVar ()) 来存储所有的非守护线程。其实，用列表类型 [(ThreadId, MVar ())] 也是可以的。

---

```
newtype ThreadManager = TdMgr (MVar (Map ThreadId (MVar ())))
```

---

下面这个函数会创建一个新的线程管理器，Map 容器内存储的映射为空映射。

---

```
newThreadManager :: IO ThreadManager
newThreadManager = fmap TdMgr (newMVar empty)
```

---

接着就是创建线程的函数了。如果为守护线程，那么直接创建，不做任何其他的事情。

---

```
niceFork :: ThreadType -> ThreadManager -> IO () -> IO ThreadId
niceFork Daemon _ work = forkIO $ work
```

---

如果为非守护线程，那么为它新建一个空的 MVar，创建新的线程来运行给定的程序，线程结束时向对应的 MVar 存入 ()。在线程运行的同时，将这个线程的信息存入线程管理器中。

---

```
niceFork NonDaemon (TdMgr mvar) work = do
 var <- newEmptyMVar
 tdid <- forkIO $ do
 work
 putMVar var ()
 con <- takeMVar mvar
 putMVar mvar (insert tdid var con)
 return tdid
```

---

最后需要定义一个函数，让主线程等待所有的非守护线程完成。这个也很容易，只需要遍历一次，将所有线程对应的 MVar 中的值取出即可，如果某一个线程的 MVar 中没有值，那么就等待。

---

```
waitAll :: ThreadManager -> IO ()
waitAll (TdMgr mvar) = do
 con <- takeMVar mvar
 traverse_ takeMVar con
 return ()
```

---

这样，有了线程管理器，就可以用下面的方法来定义之前的函数了：

---

```
main::IO ()
main = do
 msg <- newThreadManager
 niceFork NonDaemon msg (replicateM_ 20 (putStrLn "Thread 1 is printing"))
 niceFork NonDaemon msg (replicateM_ 20 (putStrLn "Thread 2 is printing"))
 waitAll msg
```

---

关于更多线程配合与管理的部分，将在 16.4 节中简单讨论。

### 16.2.3 信道

信道（channel）与 MVar 类似，也是一个容器，但区别是，信道是一个无限长的队列，多个线程可以向一个信道内写多个值，也可以有多个线程来读这个信道。由于它是一个队列，因此，顺序就是先入列的先被读，即先进先出（first in first out, FIFO）。它提供了表 16-3 所示的几个基本函数。

表 16-3 操作信道的相关函数

函 数	功 能
newChan :: IO (Chan a)	新建一个信道
writeChan :: Chan a -> a -> IO ()	向信道中写一个值，这个操作总是会成功
readChan :: Chan a -> IO a	从信道中读取一个值，若信道为空，则线程将等待
dupChan :: Chan a -> IO (Chan a)	从零开始复制一个信道

使用 dupChan 复制的信道从空开始，但在此之后，向一个信道写入的值在复制的信道中都可见。如果信息为空，那么这个 readChan 就会卡住，而 writeChan 对信道的写入操作一起会成功。需要注意的是，如果读写的比率不同，尤其是当写入远多于读取时，那么信息就会在信道中积累起来，越来越多，最后导致内存溢出。

### 16.2.4 简易聊天服务器

下面用学过的多线程与信道来实现一个简单的网络聊天服务器，它的功能是：用户以一个名字登录，然后可以以广播的形式向所有人发送信息。其实，实现的是一个类似于 IRC（Internet

Relay Chat) 的程序。

先来了解一下网络编程，GHC 提供了 Network 库，网络是 IO 的一部分。第一个需要了解的是端口 Socket 是什么，可以理解为通信的一个端点，信息可以从这里接收并且可以从里发出。这个端口，需要在 IP 地址后指定。

---

```
listenOn :: PortID -> IO Socket
```

---

对于监听的端口，需要指定一个 PortID，这个 PortID 可以用 PortNumber 构造器来得到。当然，还有其他方法，PortNumber 只是 PortID 类型定义中的一种形式。

---

```
PortNumber :: PortNumber -> PortID
```

---

accept 将会接受 listenOn 所建立的端口，它们返回一个句柄，可以用来与客户端传递信息。这个句柄可读也可写：

---

```
accept :: Socket -> IO (Handle, HostName, PortNumber)
```

---

网络在使用之前需要被初始化，所以，需要 withSocketDo 函数来做这件事情。

---

```
withSocketDo :: IO a -> IO a
```

---

在 Haskell 中，函数的命名常常使用 withXXX 来说明这个函数会对资源进行的分配与回收，withSocketDo 意为初始化一个网络端口处理给定 IO 操作。

下面就可以写一个简单的客户与服务器对话的程序了，服务器会向客户端发送当前服务器的时间。先导入需要库：

---

```
import Network
import Text.Printf
import Control.Monad
import System.IO
import Control.Concurrent
import System.Time
import Control.Exception
```

---

定义端口号为 60000：

---

```
port :: Int
port = 60000
```

---

下面定义端口上进行的服务，首先新建一个端口并且监听，然后每来一个客户就建立一个新的线程，交给 communicate 函数去处理得到的句柄。

---

```
service :: IO a
service = do
 sock <- listenOn (PortNumber (fromIntegral port))
 putStrLn $ "Listening on port " ++ show port
 forever $ do
```

---

---

```
(hd, host ,p) <- accept sock
printf "Connection from %s:%s" host (show p)
forkIO (communicate hd `finally` hClose hd)
```

---

与客户交流时，这里使用的是 telnet 做客户端连接，除了得到 quitr 会退出以外，其余直接返回服务器的当前时间，然后发送给客户端：

---

```
communicate :: Handle -> IO ()
communicate handle = do
 hSetBuffering handle NoBuffering
 loop
 where loop = do
 line <- liftM init (hGetLine handle)
 print line
 if line == "quit"
 then hPutStrLn handle "Connection closed"
 else do
 time <- getClockTime
 hPutStrLn handle (show time)
 loop

main::IO()
main = withSocketsDo service
```

---

聊天服务器要做的就是将一个人发送的信息发给所有人，这个通过 dupChan 很容易做到，即为每一个线程复制一个信道，这样，向一个信道写入信息后，每一个客户端对应的信道都可以收得到该信息。但是，这里需要注意以下两点。

- (1) 最原始的信道没有线程需要读取它但是也需要被读，否则信息就会在该信道中一直积累最后导致内存溢出。
- (2) 发信人发出的信息不会在自己的屏幕上再次出现，所以需要给定每个客户端一个 id，只有信息不是自己发的时候服务器才会发送该信息。

---

```
module Main where

import Network
import Control.Monad
import System.IO
import Control.Concurrent
```

---

端口依旧是 60000

---

```
port :: Int
port = 60000
```

---

要开启服务，先要建立一个信道，然后监听端口。为了防止内存溢出，要建立一个线程来不断地读这个原始的信道。然后将端口还有信道交给 mainLoop 函数，它会不断地接受新的客户。

---

```

service :: IO ()
service = do
 chan <- newChan
 sock <- listenOn (PortNumber (fromIntegral port))
 forkIO $ forever $ do
 (_,msg) <- readChan chan
 return ()
mainLoop sock chan 0

```

---

一个信息是由发信部的 Id 与字符串来代表，这里用一个整数表示用户 Id。

---

```

type Message = (Int, String)

```

---

mainLoop 函数需要做的事情就是接受端口连接，得到网络通信用的句柄，再为用户建立一个新的线程，调用 communicate 函数来与用户交互。

---

```

mainLoop :: Socket -> Chan Message -> Int -> IO ()
mainLoop sock chan iden = do
 c@{hd,hn,pn} <- accept sock
 forkIO (communicate c chan iden)
 mainLoop sock chan (iden+1)

```

---

下面就是处理用户信息的函数 communicate 了。首先定义一个函数 broadcast，它会向信道里面写内容。由于将使用 dupChan，一个用户向该信道里写内容，在其他用户的信道里也会可见，实际上信息传入的其实是原始的信道。第 1 次运行时需要询问用户的名字，读取进来时，将最后一个字符去掉，它常常为 '\r'。然后通知所有人这个人在线了，之后再为这个用户复制一个信道后建立一个线程 reader，不断地从这个信道中接收信息。之前，由于该用户没有信道，因此在线信息不会打印到自己的命令行上。如果信道中有信息，则需要判定这个信息是不是由该用户发出。只有当信息发送者不为当前发送者时服务器才会将信息发送，即 when (iden /= id') (hPutStrLn hd msg)。最后还要处理用户发送的信息。这个也很简单，如果发送的是 quit，那么退出，否则向所有人发送这条信息。

---

```

communicate :: (Handle, HostName, PortNumber)
 -> Chan Message -> Int -> IO ()

communicate (hd,hn,pn) chan iden = do
 let broadcast msg = writeChan chan (iden,msg)
 hSetBuffering hd NoBuffering
 hPutStrLn hd "What is your name?\r"
 name <- liftM init (hGetLine hd)
 broadcast (">>> " ++ name ++ " is online.\r")
 hPutStrLn hd ("Welcome, " ++ name++"\r")
 dChan <- dupChan chan
 reader <- forkIO $ forever $ do
 (id',msg) <- readChan dChan
 when (iden /= id') (hPutStrLn hd msg)
 handle (\(SomeException _) -> return ()) $ forever $ do

```

---

---

```

line <- liftM init (hGetLine hd)
case line of
 "quit" -> do
 hPutStrLn hd "Bye bye, have a nice day.\r"
 killThread reader
 hClose hd
 _ -> broadcast ("From:"++name++": "++line++"\r")
killThread reader
hClose hd

main :: IO ()
main = withSocketsDo service

```

---

从 communicate 中的代码可以看到，GHC 网络库中提供的 Chan 极大地简化了这个问题，仅仅需要 50 行左右的代码就能写出一个简单的聊天服务器程序。运行该程序后，可以开启多个系统命令行窗口使用 telnet 来测试它们，测试的过程中可能会遇到系统权限问题，需要用管理员权限运行。

## 16.3 软件事务内存

### 16.3.1 软件事务内存简介

软件事务内存（Software Transactional Memory, STM）是一种比较理想的解决并发程序的方案。它的想法来自数据库处理事务的机制。数据库中处理的事务意为一系列的读取写入数据库的指令，一个数据库的事务有以下 4 种特征。

(1) 原子性 (atomicity): 即一个事务的所有指令一次性地全部被执行，或者一条指令也没有被执行。也就是说，一次事务的一系列指令不可分割。一次事务只能完成或者对数据库不产生任何影响。

(2) 一致性 (consistency): 操作数据库的过程中，数据库将会有一些不变的属性，而一致性（或协调性）就是保证在一次事务结束前与结束后这个属性是不变的。

(3) 隔离性 (isolation): 事务与事务之间在执行不会互相影响，一次事务也不会感知到另一次事务在执行时的中间状态。

(4) 持久性 (duration): 一旦一次事务完成，那么它对数据库的修改将永久地保存于数据库中，以便随后的事务可以在它基础之上对数据库进行操作。

可以看到，数据库所保证的四种特征正是并发与并行计算时保证各个线程在内存共享时所需要的。原子性保证了成功计算完整性与失败计算对内存的无影响性。隔离性将保证并发的事务不会互相干扰，就如同没有事务并发执行一样。但软件事务内存不像数据库那样会对网络端口、磁盘进行操作，而只是对内存的数据进行读写，所以称为事务内存。这样，软件事务内存的持久性就与数据库就略有不同，所以，软件事务内存所指的持久性可变相地理解为在程序运

行期间内事务的执行在内存上是持久的。

事务内存的实现有很多种方式，不同设计所要达到的目的也不尽相同。有些的实现是为了减少事务执行时产生的额外负担，尽可能快地完成一次事务的执行，有的是为了更好地扩展到多个 CPU 核心上，还有的是为了尽量减少事务执行失败的次数。在比较各种 TM 的实现与其他并行并发计算的机制时，不但需要比较这些实现，还要比较 CPU 的结构，如共享高速缓存的机制等，因为硬件可能对软件事务内存的性能有着较大的影响。当性能较为重要时，还可以通过定制硬件来获取较高的性能，称之为硬件事务内存（Hardware Transactional Memory，HTM）。它提供了完整的事务内存的实现，并且可以提高部分软件事务内存的性能。

软件事务内存对事务提供原子的操作来保证事务的原子性。具有原子性的事务在运行时，首先试探性地对数据进行操作并记录内存的日志。若经过检查此次事务是有效的，则会最终执行此次事务，否则会根据内存日志对内存状态进行回滚，恢复到此次事务执行前的内存状态，然后试着重新执行。事务的有效性是指没有多个事务同时写同一段内存单元或者有读有写的情形，也可以是当有冲突时某一次事务被优先选取作为有效的事务执行的情形，这样可以保证程序不会停滞。

软件事务内存的主要优势如下。

(1) 不会因互斥等待引起死锁 (deadlock)。因为在软件事务内存层面上对并行与并发的抽象没有提供锁，所以也就不会引起死锁。

(2) 可扩展性 (scalability)。在 CPU 核心比较少时，软件事务内存的性能可能不如使用锁的版本，但是随着 CPU 核心数量的增加，程序的性能提升可能会比单纯使用锁机制的高很多。

(3) 健壮性 (robust)。在使用锁来处理一个线程时，如果引发了异常，那么程序很可能就处在一个与之前不一致的状态，这可能影响到整个程序。下面以银行转账为例，假设账户为一个对象并且提供这些操作：lock、unlock 对账户加锁与释放锁，withdraw 和 deposit 分别为取钱与存钱。那么，转账函数 transfer 应该定义如下：

---

```
void transfer (Account a1, Account a2, int amount) {
 a1.lock();
 a1.withdraw(amount);
 a1.unlock();
 throw UnknownException(); // 抛出异常
 a2.lock();
 a2.deposit(amount);
 a2.unlock();
}
```

---

如果这里的异常没有在 transfer 函数中处理，那么从账户 a1 取出的钱就很难从调用 transfer 的上一级函数中恢复。而使用软件事务内存的 atomic 操作时：

---

```

void transfer(Account a1, Account a2, int amount) {
 atomic {
 a1.withdraw(amount);
 throw UnknownException(); //抛出异常
 a2.deposit(amount);
 }
}

```

---

软件事务内存中的操作发生异常时，事务首先会回滚，恢复到 `atomic` 执行与之前一致的状态，然后再抛出异常。这样，一个线程引起的异常对整个程序带来的影响就被最小化了。

(4) 模块化 (modularity) 和复合性 (composition)。这两个人性一直是软件开发中重要的概念。模块化使得函数简短、代码易于维护并且可以提升代码的可重用性 (reusability)。而复合性是指将不同代码拼合在一起而使程序有特定用途。GHC 的 STM 库只提供了几个简单的操作，很好地体现了这两种性质，如 `retry` 和 `orElse`，复合这两个简单操作就能很好地完成我们的任务。其实，Haskell 语言与库的设计都较好地反映了这两种性质。

当然，软件事务内存不是解决所有问题的最佳方案。比如，在运行时产生的额外负担可能要高些，但是经过多年研究，STM 库与 GHC 优化的已经较为完善了，而且很多情况下性能也不作为首要条件考虑。其次需要说明的是，软件事务内存会尽力保证各个事务执行的公平性，几乎不会出现事务饥饿的情形，但是，软件事务内存是否已经完全消除了这种情形还需要更多的实践来验证。

STM 在 Haskell 里被定义为 Monad，它实际上是 IO Monad 的一部分，但它只是代表了那些事务内存的操作的效应，而不能对文件、网络与命令行进行输入/输出进行操作，这也是原子操作 `atomically :: STM a -> IO a` 可以将软件事务内存映射为 IO 类型的原因，因为 IO 中的内存操作只是 IO 的一部分，所以 STM 可以看作 IO 的一个子集，那么这个函数为包含映射 (inclusion map)。不把 STM 单纯地定义为 IO 类型的原因也很简单，因为这些输入/输出操作很难或者不可能在事务失败时回滚，并且软件事务内存中的 M 也意为 Memory，也就是只限于对内存进行操作。如果读者安装了 Haskell Platform，那么软件事务内存相关的库就已经随之安装了；如果没有，则需要另外下载。下节来看如何在 GHC 中使用它。

在本节中，关于软件事务内存的内容主要参考了 (Harris et al., 2010) 的第 1 章内容。

### 16.3.2 软件事务内存的使用

下面用 Haskell 来实现之前提到的银行转账的例子。

软件事务内存的库中提供了一个容器类型 `TVar a` (*transactional variable*) 来存储共享的数据。它与之前介绍的 `IORef a`、`Par a`、`MVar a` 十分类似，`TVar` 可以理解为事务的指令用来读取与写入的容器（操作 `TVar` 相关函数见表 16-4）。同样，STM 也实现了 `Monad` 类型类，我们可以用 `do` 关键字。

表 16-4 操作 TVar 相关函数

函数	功能
<code>newTVar :: a -&gt; STM (TVar a)</code>	新建一个 TVar 容器，并且内部存储一个值
<code>readTVar :: TVar a -&gt; STM a</code>	从一个 TVar 容器中读取一个值
<code>writeTVar :: TVar a -&gt; a -&gt; STM ()</code>	向一个 TVar 容器中写入一个值

这里我们以银行转账为例。首先需要导入 STM 库：

```
import Control.Concurrent.STM
import Control.Concurrent
```

定义 Account 类型，它由账户名与余额组成：

```
type Name = String
data Account = Account (TVar (Name ,Int))
```

给定一个账户名与金额就可以开户，可以使用 newTVar 来完成。

```
newAccount :: String -> Int -> STM Account
newAccount name amount = do
 tvar <- newTVar (name,amount)
 return $ Account tvar
```

下面两个函数就是取钱与存钱的函数了，由于都只是对账户进行操作，因此它们的类型都为 STM ()。

```
withdraw :: Account -> Int -> STM ()
withdraw (Account tvar) amount = do
 (name,balance) <- readTVar tvar
 writeTVar tvar (name,(balance-amount))

deposit :: Account -> Int -> STM ()
deposit acc amount = withdraw acc (-amount)
```

STM Monad 代表了事务内存的效应，使用 `atomically :: STM a -> IO a` 可以将一次事务原子地执行。虽然知道 STM Monad 是 IO Monad 的一部分，但将事务内存与 IO 分离的好处是事务内存操作避免了与其他 IO 操作混用。使用 withdraw 与 deposit 函数就可以定义转账函数了。

```
transfer :: Account -> Account -> Int -> IO ()
transfer a1 a2 amount = atomically $ do
 withdraw a1 amount
 deposit a2 amount
```

这样，这个转账过程原子性地执行了，其他的事务不会觉察到这个事务执行的中间状态。下面开两个账户，一个为 mom，另一个为 son。mom 会将账户里的钱分两笔同时转给 son，然后

再查看一下两个账户的余额。

---

```
getBalance :: Account -> STM (Name,Int)
getBalance (Account v) = readTVar v

test :: IO()
test = do
 son <- atomically $ newAccount "son" 0
 mom <- atomically $ newAccount "mom" 2000
 forkIO $ transfer mom son 1000
 forkIO $ transfer mom son 1000
 ball <- atomically $ getBalance mom
 bal2 <- atomically $ getBalance son
 putStrLn $ show ball
 putStrLn $ show bal2
```

---

正如所期望的那样，两笔钱被全部转到了 son 的账户里。

---

```
> test
("mom",0)
("son",2000)
```

---

但这样定义函数其实是不正确的。这里转账的两个线程很短，完成要比主线程快。实际应用中，需要让主线程等待子线程完成。这里只是简单看一下 STM 如何使用。

`retry` :: STM a 函数可以将一次事务回滚，回到最初的状态，然后重新开始执行这个事务。比如，当一个账户里余额不足了，则不能取钱了，这里就需要 `retry`。这样，重新定义 `withdraw` 函数：

---

```
withdrawl :: Account -> Int -> STM ()
withdrawl (Account tvar) amount = do
 (name,balance) <- readTVar tvar
 if (amount > 0) && (amount > balance)
 then retry
 else writeTVar tvar (name,balance - amount)
```

---

这样，当余额不足时，事务会不断地回滚，最终导致程序卡住。

---

```
test1 :: IO()
test1 = do
 son <- atomically $ newAccount "son" 0
 atomically $ withdrawl son 100

test2 :: IO()
test2 = do
 son <- atomically $ newAccount "son" 0
 draw <- newEmptyMVar
 depo <- newEmptyMVar
 forkIO $ do
 atomically $ withdrawl son 100
 putMVar draw "Got money!"
 forkIO $ do
```

---

---

```

 atomically $ deposit son 100
 putMVar depo "Saved money!"
takeMVar draw
takeMVar depo
bal2 <- atomically $ getBalance son
putStrLn $ show bal2

> test2
("son",0)

```

---

事务执行到 `retry` 时，只会回滚而不会马上重新执行该事务。因为 STM 知道，TVar 中的值如果没有变化，即使马上执行了该事务，也需要再次遇到 `retry`。在运行时，如果遇到 `retry` 时，该事务会等待其他事务对这个 TVar 进行写入操作，然后才会重新执行，这样，再次遇到 `retry` 的概率就会低很多。

为了使 `retry` 容易使用，库中定义了 `check` 函数，它的定义如下：

---

```

check :: Bool -> STM a
check True = return undefined
check False = retry

```

---

这里为了保证函数类型的正确返回了 `undefined`，但是它永远也没有必要被计算，所以不会产生影响。使用 `check` 定义 `withdraw` 如下：

---

```

withdraw2 :: Account -> Int -> STM ()
withdraw2 (Account tvar) amount = do
 (name,balance) <- readTVar tvar
 check (amount > 0 && amount > balance)
 writeTVar tvar (name,balance - amount)

```

---

函数 `orElse` :: `STM a -> STM a -> STM a` 可以用来组合多个事务得到一个另一次事务。这个函数的意思为，当第一次事务调用 `retry` 时，执行第二个事务，如果第二个事务也调用 `retry`，那么整个事务回滚，然后重新执行。

下面假设有三个账户，即 `son`、`mom` 和 `dad`。定义一个函数需要从 `mom` 中转账到 `son` 中，如果 `mom` 中的余额不足，则从 `dad` 账户中转。这里我们在使用之前定义的 `transfer` 函数时就遇到问题了，因为它的类型为 `Account -> Account -> Int -> IO ()`。它直接通过 `atomically` 函数转换为了 `IO` 行为，所以在定义函数时不要过早地使用 `atomically`，而是提供类型 `STM a` 为结果以保证之后还可以组合这些事务。

---

```

transferSTM :: Account -> Account -> Int -> STM ()
transferSTM a1 a2 amount = do
 withdraw2 a1 amount
 deposit a2 amount

transfer2 :: Account -> Account -> Account -> Int -> STM ()
transfer2 s m d a = transferSTM m s a `orElse` transferSTM d s a

test3 :: IO ()

```

---

```

test3 :: IO ()
test3 = do
 thread1 <- newEmptyMVar
 son <- atomically $ newAccount "son" 0
 mom <- atomically $ newAccount "mom" 500
 dad <- atomically $ newAccount "dad" 2000
 forkIO $ do
 atomically $ transfer2 mom dad son 1000
 putMVar thread1 ()
 takeMVar thread1
 balson <- atomically $ getBalance son
 putStrLn $ show balson
 balmom <- atomically $ getBalance mom
 putStrLn $ show balmom
 baldad <- atomically $ getBalance dad
 putStrLn $ show baldad

>test3
("son",1000)
("mom",500)
("dad",1000)

```

STM 实现了以下类型类：Functor、Applicative、Alternative、Monad、MonadPlus。其中，MonadPlus 与 Alternative 的实现是相同的，二元运算符均为 orElse，单位元为 retry。而事实也正是如此，因为 retry ‘orElse’  $b = b$  并且  $a ‘orElse’ \text{retry} = a$ 。

<pre> instance MonadPlus STM where     mzero = retry     mplus = orElse </pre>	<pre> instance Alternative STM where     empty = retry     (&lt; &gt;) = orElse </pre>
--------------------------------------------------------------------------------	----------------------------------------------------------------------------------------

除 TVar 外 GHC 的 STM 也提供了其他的通信模型，比如 TChan 事务通道、TQueue 事务有界队列和 Queue 事务无界队列等，读者可以查阅 STM 的 API 来做进一步的了解，这里就不再做过多解释了。

本节的内容主要参考了（Peyton Jones, 2007）的第一部分，读者可以在 Simon Peyton Jones 教授的主页上在线阅读 <http://research.microsoft.com/en-us/um/people/simonpj/papers/stm/beautiful.pdf>。

### 16.3.3 哲学家就餐问题

哲学家就餐问题是并发问题当中非常经典的一个，它在 1965 年由 Edsger Dijkstra 作为考试题提出，后由 Tony Hoare 重新表述为哲学家就餐问题。题目是这样的：有 5 个哲学家环坐在一个圆桌上，它们只能安静地吃饭或者思考。每个哲学家的两侧分别有一支叉子，因为是圆桌所以一共有 5 支叉子。可是，哲学家一定要用两支叉子一起才能吃饭，而在任意时刻一支叉子只能被一个哲学家拿着，只有当叉子没有人用时哲学家才能将其拿起。需要写一个程序来分配这些叉子，



图 16-4 哲学家就餐问题示意图

并且让这些哲学家们不断地思考与吃饭，但是不能引起死锁，即不会出现每一个哲学家都只拿一个叉子，谁也不主动放下叉子、谁也吃不了饭并且也不能思考的情形。这个问题用 STM 解决起来就非常容易了，对于一个哲学家来说，只需要将拿一个叉子的事务组合在一起成为拿两个叉子的事务即可，放下叉子也是一样的。

需要用到下边的库：

---

```
import Control.Monad
import Control.Concurrent
import Control.Concurrent.STM
import System.IO
import System.Random
```

---

```
type Name = String
```

先举出 5 个哲学家的名字，它们分别是罗素、柏拉图、霍布斯、洛克还有亚里士多德。

---

```
philosopherNames :: [Name]
philosopherNames = ["Russell", "Plato", "Hobbes", "Locke", "Aristotle"]
```

---

再定义一个函数来取得这个列表中的名字。

---

```
getName :: Int -> String
getName i = philosopherNames !! i
```

---

由于叉子是一个共享的变量，因此需要将其放在 TVar 容器中用来并发地读取与写入，这里用一个 Bool 类型来代表叉子是否在使用。

---

```
type Fork = TVar Bool
```

---

再定义一个函数来初始化一支叉子。

---

```
newFork :: STM Fork
newFork = do
 fork <- newTVar True
 return fork
```

---

分别定义拿起叉子函数与放下叉子函数。当拿起叉子时，如果叉子中的变量为 False，则说明在被别人使用，这时需要 retry。

---

```
takeFork, putFork :: Fork -> STM ()
takeFork fork = do
 isFree <- readTVar fork
 if isFree
 then writeTVar fork False
 else retry
```

---

放下叉子不需要有任何的前提条件，因为一个叉子在某一时刻只可能有一个人或者没有人用。

---

```
putFork fork = writeTVar fork True
```

---

这里为避免程序运行过快，可让线程有些延迟。有了随机数，可以对让线程随机地延迟地段时间。

---

```
randomDelay :: IO ()
randomDelay = do
 time <- getStdRandom(randomR (1,3))
 threadDelay (time * 10^6)
```

---

一个哲学家这样的对象，可以看做是一个名字跟两支叉子的元组。

---

```
type Philosopher = (Name, Fork, Fork)
```

---

哲学家要做的事情就是思考，拿起两支叉子吃饭，然后把叉子放下再思考，如此反复。使用递归可以很容易地实现，但是拿起两支叉子与放下两支叉子需要为原子操作。

---

```
philosopher :: Philosopher -> IO ()
philosopher (name,fork1,fork2) = do
 putStrLn (name ++ " is thinking.")
 randomDelay
 atomically $ do
 takeFork fork1
 takeFork fork2
 putStrLn (name ++ " is eating.")
 atomically $ do
 putFork fork1
 putFork fork2
 philosopher (name,fork1,fork2)
```

---

最后就该定义主函数了，首先设置命令行输出的缓冲模式，然后新建一个 MVar 让主线程等待子线程，这里的子线程永远不停止。再用 replicateM 函数新建 5 支叉子，最后用 forM 建立 5 个线程分别代表 5 个不同的哲学家，再把 5 支叉子分配给它们就可以了。这样，在运行时，GHC 的运行时系统会决定如何分配叉子。

---

```
main :: IO ()
main = do
 hSetBuffering stdout LineBuffering
 tvar <- newEmptyMVar
 forks <- replicateM 5 (atomically newFork)
 forM_ [0..4] $ \i -> forkIO
 (philosopher (getName i,forks!!i,forks!!((i+1) `mod` 5)))
 takeMVar tvar
```

---

这样，这个问题就解决了。使用 STM 简单并且方便，下面来编译测试一下函数。

---

```
C:\> ghc --make Main.hs
[1 of 1] Compiling Main (Main.hs, Main.o)
Linking Main.exe ...
C:\> .\Main.exe
Russell is thinking.
```

---

```

Plato is thinking.
Hobbes is thinking.
Locke is thinking.
Aristotle is thinking.
Locke is eating.
Locke is thinking.
Hobbes is eating.
Aristotle is eating.
...

```

---

### 16.3.4 圣诞老人问题

圣诞老人问题由 William Stallings 在他的《操作系统核心与设计原则》<sup>①</sup>一书中提出。此问题由 3 种对象参与，分别为圣诞老人、小精灵和驯鹿。圣诞老人是一直睡觉的，只有当他被需要的时候才被叫醒，如果工作做完了就再回去睡觉。他的工作有两个：一个是帮助小精灵做玩具；另一个是将雪橇栓在驯鹿上好让它们可以派送礼物。系统中只有一个圣诞老人。小精灵一直在做玩具，但是可能需要圣诞老人的帮助，小精灵的数量可能有很多，这里假定有 10 个。驯鹿平时是休息的，只有在圣诞节前夕会去派送礼物，但是需要圣诞老人需要把雪橇准备好才行。驯鹿共有 9 只。所以，加上圣诞老人一共有 20 个线程。只有当所有的驯鹿都到齐了圣诞老人才会被唤醒。为了尽量少打搅圣诞老人休息，直到有 3 个小精灵在门外时才可以唤醒圣诞老人，如果多于 3 个，其他的小精灵需要等待。然后，当这 3 个小精灵的问题解决后，圣诞老人需要等待它们全部离开才能帮助下一组小精灵。若驯鹿同时到齐并且还有不少于 3 个精灵时需要帮助时，驯鹿派送礼物这项工作有更高的优先级。比起前一个哲学家就餐问题，这个问题要复杂一些，不但参与对象的种类多了，而且还需要满足一定条件才能唤醒圣诞老人。书中希望读者使用 semaphores 来解决，但是这里借助 STM 会更加容易。

需要下边的库：

---

```

import System.IO
import System.Random
import Control.Concurrent
import Control.Monad
import Control.Concurrent.STM

```

---

圣诞老人将小精灵与驯鹿分成两组。小精灵与驯鹿总是试着加入它们对应的组。如果成功了，它就会得到两个“门”作为返回，第一个门可以用来控制小精灵进入圣诞老人家里得到帮助，第二个门控制小精灵离开圣诞老人家，只有当一个小组准备好了圣诞老人才会把门打开，所以初始的门应该是关闭的，即小精灵不能通过。对于驯鹿也是类似的。圣诞老人要做的就是等待两个小组准备好。整个程序在一个无限递归当中，小精灵与驯鹿不断地加入到对应的组中，

<sup>①</sup> 《Operating System: Internals and Design Principle》第 5 版的第 5 章，练习 5.24。

当个数足够了则会通过那些门得到圣诞老人帮助，然后离开，等一段时间之后再试着加入，如此反复。

首先定义一个门（Gate）类型，这个门由一个固定的总容量与一个可变的剩余容量组成，当另外一个函数 passGate 通过门时，剩余容量就会被减 1；如果剩余容量为 0，那么 passGate 操作会等待。初始化时门中剩余的容量为 0，当一个小组准备好时，圣诞老人使用 operateGate 函数来打开门来将剩余的容量设置为 n，这样，小精灵与驯鹿就可以通过进入圣诞老人家里得到相应的帮助，然后离开。

---

```

data Gate = MkGate Int (TVar Int)

newGate :: Int -> STM Gate
newGate n = do
 tv <- newTVar 0
 return (MkGate n tv)

passGate :: Gate -> IO {}
passGate (MkGate n tv) = atomically (do
 n_left <- readTVar tv
 check (n_left > 0)
 writeTVar tv (n_left - 1))

operateGate :: Gate -> IO {}
operateGate (MkGate n tv) = do
 atomically (writeTVar tv n)
 atomically (do
 n_left <- readTVar tv
 check (n_left == 0))

```

---

下面，使用门就可以来定义小组了。一个小组由固定的容量组成，TVar 中的值为剩余的容量以及两个门，一个门是进入时使用，另一个是离开时使用。

---

```

data Group = MkGroup Int (TVar (Int, Gate, Gate))

newGroup 可以初始化一个小组。

newGroup :: Int -> IO Group
newGroup n = atomically (do
 g1 <- newGate n
 g2 <- newGate n
 tv <- newTVar (n,g1,g2)
 return (MkGroup n tv))

```

---

圣诞老人要做的就是等待一个小组准备好了。如果有小组准备好了，那么新建两个门，写入这个小组，然后返回旧的两个门，圣诞老人得到这两个门后将它们打开即可。这样，在帮助一组的同时，其他对象可以继续排队，而不影响圣诞老人正在帮助的一组。

---

```

awaitGroup :: Group -> STM (Gate, Gate)
awaitGroup (MkGroup n tv) = do
 (n_left, g1,g2) <- readTVar tv
 check (n_left == 0)
 new_g1 <- newGate n
 new_g2 <- newGate n
 writeTVar tv (n,new_g1,new_g2)
 return (g1,g2)

```

---

下面是随机延时的函数，这里就不再多讲述了。

---

```

randomDelay :: IO ()
randomDelay = do
 time <- getStdRandom(randomR (1,3))
 threadDelay (time * 10^6)

```

---

小精灵与驯鹿都要试着加入相应的小组中，但只有当剩余的容量大于 0 才可以。如果成功加入，那么小组的容量要减 1，然后得到两个门，等待圣诞老人把它们打开。

---

```

joinGroup (MkGroup n tv) = atomically (do
 (n_left,g1,g2) <- readTVar tv
 check (n_left > 0)
 writeTVar tv (n_left-1,g1,g2)
 return (g1,g2))

```

---

下面的函数就其实是小精灵与驯鹿共用的函数，这两个对象要做的就是加入小组。如果成功，就得到两个门。小组的对象全都在等圣诞老人开门，它们依次通过第一个门，得到圣诞老人的帮助后从第二个门离开。

---

```

task :: Group -> IO () -> IO ()
task group do_task = do
 (in_gate,out_gate) <- joinGroup group
 passGate in_gate
 do_task
 passGate out_gate

elf', reindeer':: Group -> Int -> IO ()
elf' group e_id = task group (getHelp e_id)
reindeer' group deer_id = task group (deliverToys deer_id)

```

---

下面则是两个不重要的输出操作，一个是来自得到帮助的小精灵，另一个是托着雪橇去送玩具的驯鹿。

---

```

getHelp, deliverToys :: Int -> IO ()
getHelp e_id = putStrLn ("Elf "++ show e_id ++" is getting help from Santa")
deliverToys d_id = putStrLn ("Reindeer " ++ show d_id ++ " delivering toys")

```

---

这里就可以定义小精灵与驯鹿的线程了。分配给每个对象一个 id，线程要做的就是之前定义的 task 函数，这里仅仅是要区分小精灵与驯鹿。

---

```
elf,reindeer :: Group -> Int -> IO ThreadId
elf gp e_id = forkIO (forever (do {elf' gp e_id; randomDelay}))
reindeer gp d_id = forkIO (forever (do {reindeer' gp d_id; randomDelay}))
```

---

最后则是圣诞老人线程了。这里，他在主线程中，不必新建一个线程。他要做的事是等待两个小组，如果有一个满了就打开门，但是这个问题有特别的要求，就是把驯鹿拴上雪橇去送玩具要比帮助小精灵重要，使用 orElse 可以很容易地解决。

---

```
santa' :: Group -> Group -> IO ()
santa' elf_gp rein_gp = do
 putStrLn "-----"
 (do_task, (in_gate, out_gate)) <- atomically (orElse
 (chooseGroup rein_gp "Let's deliver toys!")
 (chooseGroup elf_gp "Let me help with the toys!"))
 putStrLn ("Ho! Ho! Ho! " ++ do_task)
 operateGate in_gate
 operateGate out_gate

where
 chooseGroup :: Group -> String -> STM (String, (Gate, Gate))
 chooseGroup gp do_task = do
 gates <- awaitGroup gp
 return (do_task, gates)
```

---

像上面这样定义圣诞老人函数是没有问题的，但是人们希望找到一个更为一般的解决方案，因为在多线程编程中常常有这些种情形：某个线程决定从一系列的事务中选取一个进行操作，接着又会有一连串的事务操作，这里就是等待小组。然后，圣诞老人开门，两种对象分别通过定义的门。还有一个线程从其他线程中收取消息，对这个消息进行操作。

想要做到这些，可以定义一个 choose 函数，将 (STM a, a -> IO ()) 元组的列表 foldr1 orElse 会选取第一个可以执行的事务，然后只对事务内存中的值进行操作。choose 将为圣诞老人选择小组，得到两个门，然后开门。但是无需在意是哪一次事务，这里只打印消息，所以只要放在元组的第二个元件 IO 操作中处理即可。原子地执行类型为 STM (IO ()) 的事务会得到 IO (IO ()) 类型，可以使用 join 转为 IO ()。

---

```
choose :: [(STM a, a -> IO ())] -> IO ()
choose choices = join (atomically (foldr1 orElse actions))
where
 actions :: [STM (IO())]
 actions = [do {val <- g ; return (rhs val)}
 | (g, rhs) <- choices]
```

---

这样，圣诞老人要做的就是从两组中选取一组，运行给定的 IO 操作即可。

---

```
santa :: Group -> Group -> IO ()
santa elf_gp rein_gp = do
 putStrLn "-----"
```

---

---

```

choose [(awaitGroup rein_gp, run "Let's deliver toys!"),
 (awaitGroup elf_gp, run "Let me help with the toys!")]
where
 run :: String -> (Gate, Gate) -> IO ()
 run do_task (in_gate, out_gate) = do
 putStrLn ("Ho! Ho! Ho! " ++ do_task)
 operateGate in_gate
 operateGate out_gate

```

---

最后则是主函数。首先要设置缓冲区模式，然后为小精灵建立容量为 3 的小组，再创建 10 小精灵线程，它们一直要加入这个小组；同样为驯鹿建立容量为 9 的小组，然后创建 9 个驯鹿线程，驯鹿也一直试着加入这个小组。而圣诞老人要做的就是监听这两个小组，是否满足开门的条件。

---

```

main::IO()
main = do
 hSetBuffering stdout LineBuffering
 elf_group <- newGroup 3
 sequence_ [elf elf_group n | n <- [1..10]]
 rein_group <- newGroup 9
 sequence_ [reindeer rein_group n | n <- [1..9]]
 forever (santa elf_group rein_group)

```

---

编译测试一下。

---

```

C:\> ghc Main.hs
[1 of 1] Compiling Main (Main.hs, Main.o)
Linking Main.exe ...
C:\> .\Main.exe

Ho! Ho! Ho! Let's deliver toys!
Reindeer 1 delivering toys
Reindeer 2 delivering toys
Reindeer 4 delivering toys
Reindeer 3 delivering toys
Reindeer 5 delivering toys
Reindeer 6 delivering toys
Reindeer 7 delivering toys
Reindeer 8 delivering toys
Reindeer 9 delivering toys

Ho! Ho! Ho! Let me help with the toys!
Elf 1 is getting help from Santa
Elf 2 is getting help from Santa
Elf 3 is getting help from Santa
...

```

---

本节的内容与代码来自 (Peyton Jones, 2007) 的第二部分。读者可以在线阅读 <http://research.microsoft.com/en-us/um/people/simonpj/papers/stm/beautiful.pdf>。

## 16.4 异步并发库简介

之前讨论了轻量级线程、STM 等内容。轻量级线程提供了一种灵活的方式来编写多线程程序。可以看出，它仍是很底层的库，需要用户自己实现一个简单的线程管理器来控制这些线程的运行等待以及其他的问题。STM 也是一种非常好的并发机制，它让并发事务可以非常容易地复合在一起，虽然库较为小巧，但是用起来可以非常灵活，可是它也是很底层的库。在实际开发过程中，需要很多其他的功能，比如线程的异步等待、异常处理、线程运行超时、线程竞争、建立线程树等。`async` 库就是建立在轻量级线程与 STM 上的库，提供了高一层的抽象。库内主要是一些函数可以异步地等待并发计算的结果，实现所用到的大部分内容也是之前讨论过的，本节大致地了解一下如何把一些问题做更高的抽象。

安装 `Async` 库可以使用下面两条命令，也可以在 Haskell 的网站上下载 `Async` 库，地址 <http://hackage.haskell.org/package/async>。不过，很有可能在这本书出版的时候，Haskell Platform 已经加入了这个库，读者可以查阅自己的 Haskell Platform 的 API，如果安装过就不必再安了。

---

```
C:\> cabal update
C:\> cabal install async
```

---

`Async` 库假设现在需要使用 2 个线程来做两件事情并返回结果，在轻量级线程中已经介绍过了，大约应当像下面这样实现：

---

```
do
 mvar1 <- newEmptyMVar
 mvar2 <- newEmptyMVar
 forkIO $ do
 result <- doIOAction ...
 putMVar mvar1 r
 forkIO $ do
 result <- doIOAction ...
 putMVar mvar2 r
 result1 <- takeMVar mvar1
 result2 <- takeMVar mvar2
 return (r1,r2)
```

---

之前定义过一个简单的线程管理器找到了这段程序的公共部分，这里需要异步的原因是，在创建了一个线程运行程序后，它并不能马上得到结果，而是需要调用它的线程进行等待，所以需要用 `MVar` 来作为一种消息传递机制。

`Async` 库就是用来处理这些问题的，其中，有非常多的函数处理各种异步并发的问题，它的实现主要用到了 `Control.Exception` 与 `STM`。首先，它定义了 `Async` 类型，代表了异步的 IO 操作并且考虑了异常。可以使用 `async` 函数来产生一个异步线程，这个线程的 `Id` 可以用 `asyncThreadId :: Async a -> ThreadId` 来得到。下面大致地看一下其中一些函数的定义与用法。

---

```
data Async a = Async !ThreadId! (TMVar (Either SomeException a))
```

---

async 函数可以新建一个异步并发的线程。

---

```
async :: IO a -> IO (Async a)
```

---

然后，库中定义了 wait 函数来等待并得到结果。这里的 wait 用了 readTMVar 而没有用 takeTMVar，这样，可以对一个 Async 类型使用多次 wait 函数来得到结果。

---

```
wait :: Async a -> IO a
wait = atomically . waitSTM
```

---

```
waitSTM :: Async a -> STM a
waitSTM (Async _ var) = do
 r <- readTMVar var
 either throwSTM return r
```

---

这样，之前的程序就可以很简单地写为：

---

```
do
 async1 <- async $ doIOAction ...
 async2 <- async $ doIOAction ...
 result1 <- wait async1
 result2 <- wait async2
 return (result1, result2)
```

---

在使用多线程的时候，用户常常想终止一个线程的运行，Control.Exception 里提供了 throwTo :: Exception e => ThreadId -> e -> IO () 函数，可以引发使得引种终止的异常，这样，取消一个异步线程就定义为：

---

```
cancel :: Async a -> IO ()
cancel (Async t _) = throwTo t ThreadKilled
```

---

为了更好地处理多个线程的任务，库中提供了很多其他非常重要的函数，其中，有两个非常重要的函数 concurrently 与 race。其中，concurrently 是像上面的函数一样。

---

```
concurrently :: IO a -> IO b -> IO (a, b)
```

---

现在，借助 concurrently 函数，只需要一行就可以定义之前的函数了，这个函数很好地把一大类问题抽象了出来。它是由库中的 waitBoth :: Async a -> Async b -> IO (a,b) 实现的。与 concurrently 对应的函数就是 race：

---

```
race :: IO a -> IO b -> IO (Either a b)
```

---

它是通过库中的库中的 waitEither :: Async a -> Async b -> IO (Either a b) 定义的。这样，给定两个 IO 操作，race 函数将并发地运行它们，先完成的线程结果将会被返回，未完成的线程会被终止。

异步并发库中还有很多非常实用的函数。比如，将多个线程连接起来，这样任意一个线程发生异常时会同时影响两个线程，这对线程管提供了非常大的方便。此外，还有 `waitAny`，可以等待多个线程中最先完成的那个，借助折叠函数 `fold`，也是不难实现的。由于这本书不是主要介绍 Haskell 并发编程的书，因此，关于异步并发的内容就讨论到这里，读者想了解更多的相关函数，可以参阅相关 `Control.Concurrent.Async` 的 API。

## 本章小结

并发一直是软件开发中十分关心的内容，在服务器集群上使用 Haskell 进行分部式计算一直是一个较为热门的话题。本章仅仅是简单介绍了一些常见的方法。除之前介绍的表达并发程序的方法以外，还有基于 Tony Hoare 教授于牛津大学引入的 Communicating Sequential Processes (CSP) 的 Communicating Haskell Processes (CHP) 库，它由英国 Kent 大学开发。CSP 是一种表达并发程序的数学语言，将这些程序数学化对于证明、演绎有着非常重要的帮助。读者可以在 <http://www.cs.kent.ac.uk/projects/ofa/chp/> 中了解相关内容。另外一个是 Actor 模式，它也是为表达并发程序建立的数学模型，很多语言中都有实现，如 Erlang、Scala 等，读者可以在网上查找相关的资料来了解。关于更多使用 Haskell 并行与并发的内容可以登录 [http://www.haskell.org/haskellwiki/Applications\\_and\\_libraries/Concurrency\\_and\\_parallelism](http://www.haskell.org/haskellwiki/Applications_and_libraries/Concurrency_and_parallelism) 来了解。另外可以阅读由 Simon Marlow 编写的《Parallel and Concurrrent Programming in Haskell》，读者可以在 <http://chimera.labs.oreilly.com/books/123000000929/index.html> 免费阅读。

## 参 考 文 献

- Bird,Richard.** 1998 *Introduction to Functionl Programriog using Haskell.* s.l.:Prentice Hall,1998.
- Cormen, Thmomas H, et al.** 2009. *Introduction to Algorithm.* s.l. : The MIT Press, 2009.
- GHC Team.** 2012. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.4.2. 2012.*
- Harris, Tim, Larus, James and Rajwar, Ravi.** 2010. *Transactional Memory.* s.l. : Morgan & Claypool, 2010.
- Hutton, Graham M.** 2007. *Programming in Haskell.* s.l. : Cambridge University Press, 2007.
- Lipovača, Miran.** 2011. *Learn You a Haskell for Great Good.* s.l. : no starch press, 2011.
- O'Sullivan, Bryan, Goerzen, John and Stewart, Don.** 2009. *Real World Haskell.* s.l. : O'Reilly, 2009.
- Peyton Jones, Simon L.** 2007. *Beautiful Concurrency. Beautiful Code.* s.l. : O'Reilly, 2007.
- . 1987. *The Implementation of Functional Programming Language.* s.l. : Prentice Hall, 1987.

## 后记

首先，十分感谢你购买这本书并且能把它读完！其实，把这本书当成是一本入门书并不是很恰当，因为这并不是严格意义上的入门书籍，其中的一些内容已经涉及了 Haskell 中比较深入的部分。虽然内容上相对深入了，但是读完这本书并不是学习 Haskell 的终点，而仅仅是一个起点。下面，我想简要地介绍一下其他的相关的文章、讲义和书籍。

我个人觉得在学习 Haskell 时，参考英文资料是必不可少的。中文书籍仅仅能起到引路的作用，毕竟大部分的 Haskell 与函数式编程的文章都采用英文撰写。如果读者在读一些文献有困难的话，不妨从语言简练的《Programming in Haskell》开始，这本书由多年教学经验的 Graham Hutton 教授编写，它很薄，内容很实在。另一本就是语言随意轻快的《Learn You a Haskell for Great Good》，全书语言简单、有亲和力，如果作为第一本英文的 Haskell 读物也会是一个比较不错的开始（网上有英文版可免费阅读，可以登录 <http://learnyouahaskell.com/chapters>）。请读者不要认为看过本书以后再去读简单的入门的书籍只是浪费时间，其实不然，我常常可以从一些简单的书中想到新的东西，正可谓温故而知新。

如果你想了解更多用 Haskell 来解数学与算法的趣题的内容，那么我推荐你阅读由已经从牛津大学退休的教授 Richard Bird 编写的《Pearls of Functional Algorithm Design》。书里面有很多算法问题，都是用函数式编程语言来解的，相信想用 Haskell 语言学习算法的读者一定会喜欢。另外一个很好的资料就是 Haskell Wiki 上的《函数式编程 99 题》，是《Lisp 99 题》的 Haskell 版本。本书从中选取了几个例子，读者可以访问 [http://www.haskell.org/haskellwiki/99\\_questions](http://www.haskell.org/haskellwiki/99_questions)。网站上不但有题目，还有答案，虽然个别答案是错误的，但对于喜欢算法类趣题的读者是一个不错的资源。

有些读者可能对如何使用 Haskell 开发商业软件更感兴趣，那么，由 O'Reilly 出版的 Real World Haskell 是不应该错过的。这本书的作者之一是在斯坦福大学从事函数式编程多年研究与教学的 Bryan O'Sullivan（网上可以免费阅读，<http://book.realworldhaskell.org/read/>）。书中所用的 GHC 版本有些旧了，不过书的新版应该已经在紧锣密鼓的计划当中，相信离出版上市不会很远了。

并发与并行一直是一个饱受关注的话题。如果读者想进一步了解并行与并发，可以阅读曾在微软剑桥研究院工作的 Simon Marlow 撰写的，由 O'Reilly 出版的《Parallel and Concurrent Programming in Haskell》。这是一本不错的书，部分内容来自 Simon 夏期课程的讲义。Simon Marlow 曾是 GHC 项目领导者——Simon Peyton Jones 的同事。Simon Marlow 现任职于 Facebook，为 Facebook 用 Haskell 更好地解决一些实践中的问题。他主要负责编写了 GHC 的运行时系统、

异常处理、语法分析器生成器 Happy 以及文档生成工具 Haddock 等。他发表的关于 Haskell 的学术文章、讲义或者博客都是值得一读的。

若读者想对 Haskell 的语法与 GHC 的使用做更深入、更全面的了解，那么《Haskell 2010 Language Report》和《The Glorious Glasgow Haskell Compilation System User's Guide》可以作为查阅的手册使用，网上也有免费的 PDF 版本可供阅读。同样，我推荐由 Simon Peyton Jones 与 Simon Marlow 一起编写的“*The Glasgow Haskell Compiler*”一文，文中讲述了 GHC 的结构和特性。可以登录 <http://www.aosabook.org/en/ghc.html> 阅读。另外，GHC 作为一个免费开源的软件，大家可以在 <http://www.haskell.org/ghc/> 下载 GHC 的源代码，以便研究之用。

如果想进一步了解函数式编程语言的设计与实现，那么可以参阅 Simon Peyton Jones 教授编写的《*The Implementation of Functional Programming Language*》在 Simon Peyton Jones 教授所在的微软剑桥研究院的主页上可以免费下载到这本书的 PDF 扫描版本。期间，你可能要更深入地了解  $\lambda$  演算、类型与 System F 的相关内容，其中， $\lambda$  演算与类型的内容可以参阅 J.Roger Hindley 编写的《*Lambda-Calculus and Combinators*》与《*Basic Simple Type Theory*》。你还可以阅读一本非常经典的、理论与实践相结合的书，是宾夕法尼亚大学 Benjamin C.Pierce 教授编写的《*Types and Programming Language*》。

如果你不知道需要读什么，但又很想对 Haskell 做进一步的了解，那么我建议你下载 Simon Peyton Jones 教授与 Simon Marlow 发表的相关论文，如“*A History of Haskell*”，其中讲述了 Haskell 的由来与历史，记录了 Simon Peyton Jones 教授在实现 Haskell 类型类时遇到的困难与解决过程，还是相当有趣味性的。

本书出版之后，在我继续学业的同时，还会在业余时间计划在优酷视频网站上开一个频道，将 Youtube 网站上关于 Haskell 的视频讲座转传到优酷上供大家观看。如果有时间，可能会加一些中文字幕，也有可能自己做一系列的中文讲座，到时敬请大家关注。

最后，希望大家能通过这本书喜欢上 Haskell，喜欢上函数式编程，愿 Haskell 可以给你学习编程带来乐趣，能给你的工作带来轻松和愉快。

# Haskell 函数式编程入门

作为Haskell语言的设计者和编译器的作者之一，我十分高兴有一本关于Haskell的中文书籍上市了。

Haskell是一门纯函数式编程语言。也就是说，计算主要是通过函数来完成的（像在数学中一样），而不是通过“先做这个，再做那个”的命令式操作顺序进行的（像在主流编程语言中一样）。首先，让人惊讶的是，即便有这样的限制，你也可以做所有你想做的事情，这一点与没有这些限制的顺序式语言完全一样。Haskell中的高阶函数、惰性求值和表达能力极强的类型系统等特性会完全改变你对编程的态度。

阅读这本书，不仅仅是会让你了解Haskell语言，而且会让你成为一名更好的程序员，无论你常用的是何种编程语言，因为你会从一个全新的视角来看待你在做的事情。祝你阅读愉快！

—— Simon Peyton Jones



张鹏 1989年1月出生于黑龙江省绥芬河市，酷爱数学、物理，进入大学后开始喜欢学习与研究各类编程语言，并成为了一名Haskell爱好者。2012年7月于英国诺丁汉大学获得计算机科学本科学位。2013年11月于英国牛津大学获得计算机科学硕士学位。目前喜欢学习与研究λ演算、类型系统、抽象代数、范畴论、逻辑证明、组合数学等内容。

（姜皓桐 摄于牛津大学Pembroke学院）



分类建议：计算机/程序设计

人民邮电出版社网址：[www.ptpress.com.cn](http://www.ptpress.com.cn)



ISBN 978-7-115-33801-3



ISBN 978-7-115-33801-3

定价：59.00 元

任文杰 封面设计：任文杰



# Document Outline

- [封面](#)
- [前言](#)
- [目录](#)
- [第1章 Haskell简介](#)
  - [1.1 Haskell的由来](#)
  - [1.2 Haskell编译器的安装以及 编写环境](#)
  - [1.3 GHCi的使用](#)
    - [1.3.1 GHCi中的命令](#)
    - [1.3.2 在GHCi中调用函数](#)
  - [1.4 hs和.lhs文件、注释与库函数](#)
  - [1.5 第一个Haskell程序HelloWorld!](#)
  - [本章小结](#)
- [第2章 类型系统和函数](#)
  - [2.1 Haskell的类型与数据](#)
    - [2.1.1 Haskell常用数据类型](#)
    - [2.1.2 函数类型](#)
    - [2.1.3 类型的别名](#)
    - [2.1.4 类型的重要性](#)
  - [2.2 Haskell中的类型类](#)
    - [2.2.1 相等类型类: Eq](#)
    - [2.2.2 有序类型类: Ord](#)
    - [2.2.3 枚举类型类: Eenum](#)
    - [2.2.4 有界类型类: Bounded](#)
    - [2.2.5 数字类型类: Num](#)
    - [2.2.6 可显示类型类: Show](#)
    - [2.2.7 小结](#)
  - [2.3 Haskell中的函数](#)

- [2.3.1 Haskell中的值](#)
- [2.3.2 函数思想入门](#)
- [2.3.3 函数的基本定义格式](#)
- [2.3.4 入表达式](#)
- [2.3.5 参数的绑定](#)
- [2.4 Haskell中的表达式](#)
  - [2.4.1 条件表达式](#)
  - [2.4.2 情况分析表达式](#)
  - [2.4.3 守卫表达式](#)
  - [2.4.4 模式匹配](#)
  - [2.4.5 运算符与函数](#)
  - [2.4.6 运算符与自定义运算符](#)
- [本章小结](#)
- [第3章 基于布尔值的函数](#)
  - [3.1 关键字module与import简介](#)
  - [3.2 简易布尔值的函数](#)
  - [3.3 与非门和或非门](#)
  - [本章小结](#)
- [第4章 库函数及其应用](#)
  - [4.1 预加载库函数](#)
    - [4.1.1 常用函数](#)
    - [4.1.2 基于列表的函数](#)
    - [4.1.3 定义历法公式](#)
    - [4.1.4 字符串处理的函数](#)
  - [4.2 字符与位函数库简介](#)
    - [4.2.1 Data.Char](#)
    - [4.2.2 Data.Bits](#)
  - [本章小结](#)
- [第5章 递归函数](#)

- [5.1 递归函数的概念](#)
- [5.2 简单递归函数](#)
- [5.3 扩展递归与尾递归](#)
- [5.4 互调递归](#)
- [5.5 麦卡锡的91函数](#)
- [5.6 斐波那契数列](#)
- [5.7 十进制数字转成罗马数字](#)
- [5.8 二分法查找](#)
- [5.9 汉诺塔](#)
- [5.10 排序算法](#)
  - [5.10.1 插入排序](#)
  - [5.10.2 冒泡排序](#)
  - [5.10.3 选择排序](#)
  - [5.10.4 快速排序](#)
  - [5.10.5 归并排序](#)
- [小结](#)
  - [5.11 递归基本条件与程序终止](#)
  - [5.12 递归与不动点](#)
  - [5.13 无基本条件递归和惰性求值](#)
- [本章小结](#)
- [第6章 列表内包](#)
  - [6.1 列表生成器](#)
  - [6.2 素数相关趣题](#)
  - [6.3 凯撒加密](#)
    - [6.3.1 加密](#)
    - [6.3.2 解密](#)
  - [6.4 排列与组合问题](#)
    - [6.4.1 排列问题](#)
    - [6.4.2 错位排列问题](#)

- [6.4.3 组合问题](#)
  - [6.5 八皇后问题](#)
  - [6.6 计算矩阵乘法](#)
  - [6.7 最短路径算法与矩阵乘法](#)
  - [本章小结](#)
- [第7章 高阶函数与复合函数](#)
  - [7.1 简单高阶函数](#)
  - [7.2 折叠函数foldr与foldl](#)
  - [7.3 mapAccumL与mapAccumR函数](#)
  - [7.4 复合函数](#)
  - [本章小结](#)
- [第8章 定义数据类型](#)
  - [8.1 数据类型的定义](#)
    - [8.1.1 枚举类型](#)
    - [8.1.2 构造类型](#)
    - [8.1.3 参数化类型](#)
    - [8.1.4 递归类型](#)
    - [8.1.5 杂合定义类型](#)
  - [8.2 类型的同构](#)
  - [8.3 使用newtype定义类型](#)
  - [8.4 数学归纳法的有效性](#)
  - [8.5 树](#)
  - [8.6 卡特兰数问题](#)
  - [8.7 霍夫曼编码](#)
  - [8.8 解24点](#)
  - [8.9 zipper](#)
  - [8.10 一般化的代数数据类型](#)
  - [8.11 类型的kind](#)
    - [8.11.1 类型的kind](#)

- [8.11.2 空类型的声明](#)
  - [本章小结](#)
- [第9章 定义类型类](#)
  - [9.1 定义类型类](#)
  - [9.2 Haskell中常见类型类](#)
    - [9.2.1 常用类型类](#)
    - [9.2.2 Functor](#)
    - [9.2.3 Applicative](#)
    - [9.2.4 Alternative](#)
    - [9.2.5 简易字符识别器](#)
    - [9.2.6 Read类型类](#)
    - [9.2.7 单位半群\(Monoid\)](#)
    - [9.2.8 Foldable与Monoid类型类](#)
    - [9.2.9 小结](#)
  - [9.3 类型类中的类型依赖](#)
  - [9.4 类型类中的关联类型](#)
  - [9.5 定长列表](#)
  - [9.6 运行时重载](#)
  - [9.7 Existential类型](#)
  - [本章小结](#)
- [第10章 Monad初步](#)
  - [10.1 Monad简介](#)
  - [10.2 从Identity Monad开始](#)
  - [10.3 Maybe Monad](#)
  - [10.4 Monad定律](#)
  - [10.5 列表Monad](#)
  - [10.6 Monad相关运算符](#)
  - [10.7 MonadPlus](#)
  - [10.8 Functor、Applicative与Monad的关系](#)

- [本章小结](#)
- [第11章 系统编程及输入/输出](#)
  - [11.1 不纯函数与副作用](#)
  - [11.2 IO Monad](#)
  - [11.3 输入/输出处理](#)
    - [11.3.1 Control.Monad中的函数](#)
    - [11.3.2 系统环境变量与命令行参数](#)
    - [11.3.3 数据的读写](#)
    - [11.3.4 格式化输出printf函数](#)
    - [11.3.5 printf函数的简易实现](#)
  - [11.4 星际译王词典](#)
    - [11.4.1 二分法查找](#)
    - [11.4.2 散列表的使用](#)
  - [11.5 简易异常处理](#)
  - [11.6 Haskell中的时间](#)
  - [本章小结](#)
- [第12章 记录器Monad、读取器Monad、状态Monad](#)
  - [12.1 记录器Monad](#)
    - [12.1.1 MonadWriter](#)
    - [12.1.2 记录归并排序过程](#)
  - [12.2 读取器Monad](#)
    - [12.2.1 MonadReader](#)
    - [12.2.2 变量环境的引用](#)
  - [12.3 状态Monad](#)
    - [12.3.1 状态Monad标签器](#)
    - [12.3.2 用状态Monad实现栈结构](#)
    - [12.3.3 状态Monad、FunApp单位半群和读取器Monad的关系](#)
    - [12.3.4 MonadState](#)

- [12.3.5 基于栈的计算器](#)
- [12.4 随机数的生成](#)
- [本章小结](#)
- [第13章 Monad转换器](#)
  - [13.1 从IdentityT Monad转换器开始](#)
  - [13.2 Monad转换器组合与复合Monad的区别](#)
  - [13.3 Monad转换器的组合顺序](#)
  - [13.4 lift与liftIO](#)
  - [13.5 简易Monad编译器](#)
  - [13.6 语法分析器Monad组合子](#)
    - [13.6.1 简易语法分析器的实现](#)
    - [13.6.2 Parsec库简介](#)
    - [13.6.3 上下文无关文法](#)
    - [13.6.4 基于语法分析器的计算器](#)
  - [本章小结](#)
- [第14章 QuickCheck简介](#)
  - [14.1 测试函数属性](#)
  - [14.2 测试数据生成器](#)
  - [本章小结](#)
- [第15章 惰性求值简介](#)
  - [15.1 λ演算简介](#)
  - [15.2 ⊥Bottom](#)
  - [15.3 表达式形态和thunk](#)
    - [15.3.1 WHNF、HNF与NF](#)
    - [15.3.2 thunk与严格求值](#)
  - [15.4 求值策略](#)
    - [15.4.1 引值调用](#)
    - [15.4.2 按名调用](#)
    - [15.4.3 常序求值](#)

- [15.5 惰性求值](#)
- [15.6 严格模式匹配与惰性模式匹配](#)
- [第16章 并行与并发编程](#)
  - [16.1 确定性的并行计算](#)
  - [16.2 轻量级线程](#)
    - [16.2.1 调度的不确定性](#)
    - [16.2.2 基本线程通信](#)
    - [16.2.3 信道](#)
    - [16.2.4 简易聊天服务器](#)
  - [16.3 软件事务内存](#)
    - [16.3.1 软件事务内存简介](#)
    - [16.3.2 软件事务内存的使用](#)
    - [16.3.3 哲学家就餐问题](#)
    - [16.3.4 圣诞老人问题](#)
  - [16.4 异步并发库简介](#)
  - [本章小结](#)
- [参考文献](#)
- 后记 359