Haskell– 函数程序设计技艺

Simon Thompson 著

乔海燕译

（内部资料，请勿外传）
(译文初稿, 欢迎提出意见.
Email: isdqhy@zsu.edu.cn)

2006 年 8 月 30 日

目 录

第一章 函数程序设计简介 **15**
§1.1 计算机与建模 15
§1.2 什么是函数? 16
§1.3 图形与函数 17
§1.4 类型 18
§1.5 函数程序设计语言 Haskell 19
§1.6 表达式与计算 19
§1.7 定义 20
§1.8 函数定义 22
§1.9 一个图形模型 24
§1.10 证明 26
§1.11 类型和函数程序设计 27
§1.12 计算与求值 28

第二章 Haskell 解释系统 Hugs **31**
§2.1 第一个 Haskell 程序 31
§2.2 使用 Hugs 34
§2.3 标准库 Prelude 和 Haskell 函数库 37
§2.4 模块 37
§2.5 第二个例子：Pictures 38
§2.6 错误与错误信息 40

第三章 基本类型和定义 **43**
§3.1 布尔类型 Bool 43
§3.2 整数类型 Int 45
§3.3 重载 47
§3.4 守卫 47
§3.5 字符型 Char 50
§3.6 浮点数 Float 51
§3.7 语法 54

第四章 设计与编写程序 **59**
§4.1 从何处出发设计一个 Haskell 程序? 59
§4.2 递归 63
§4.3 实践中的原始递归 66
§4.4 递归的一般形式 68

§4.5 程序测试 70

第五章 数据类型：多元组和列表 75
§5.1 多元组, 列表和串简介 75
§5.2 元组类型 76
§5.3 定义列表 79
§5.4 Haskell 的列表 79
§5.5 列表概括 81
§5.6 一个图书馆数据库 83
§5.7 通用函数：多态 87
§5.8 Haskell 引导库 Prelude 中的列表函数 89
§5.9 类型 String 91

第六章 列表程序设计 95
§6.1 再次讨论 Picture 一例 95
§6.2 扩展的练习：有位置的图形 98
§6.3 局部定义 99
§6.4 扩展练习：超市帐单 104

第七章 定义列表上的函数 109
§7.1 再谈模式匹配 109
§7.2 列表与列表模式 110
§7.3 列表上的原始递归 112
§7.4 寻找原始递归定义 113
§7.5 列表上的一般递归 117
§7.6 例子：文本处理 119

第八章 程序推理 125
§8.1 理解定义 125
§8.2 测试与证明 126
§8.3 定义性、终止性和有限性 127
§8.4 一点逻辑知识 128
§8.5 归纳法 129
§8.6 归纳证明的进一步例子 132
§8.7 推广证明目标 135

第九章 推广：计算模型 139
§9.1 列表上的计算模式 139
§9.2 高阶函数：函数作为参数 141
§9.3 折叠与原始递归 146

§9.4 推广：折分列表 149

第十章 函数作为值 151

§10.1 函数层定义 151
§10.2 函数的复合 152
§10.3 函数作为值和结果 154
§10.4 部分应用 157
§10.5 再谈 Picture 161
§10.6 更多的例子 164
§10.7 Currying 和非 Currying 164
§10.8 例子: 建立索引 166
§10.9 验证与通用函数 171

第十一章 程序开发 179

§11.1 开发周期 179
§11.2 实践中的程序开发 182

第十二章 重载和类型分类 183

§12.1 为什么使用重载？ 183
§12.2 引进类型分类 184
§12.3 Signatures 和特例 186
§12.4 Haskell 的预定义类 191
§12.5 类型与类 195

第十三章 类型检测 199

§13.1 单态类型检测 199
§13.2 多态类型检测 201
§13.3 类型检测与分类 208

第十四章 代数类型 211

§14.1 代数类型简介 211
§14.2 递归代数类型 217
§14.3 多态代数类型 223
§14.4 实例研究：程序错误 226
§14.5 使用代数类型设计系统 229
§14.6 代数类型与类型分类 233
§14.7 代数类型的推理 238

第十五章 实例研究：Huffman 编码 243
§15.1 Haskell 的模块 . . . 243
§15.2 模块设计 . . . 246
§15.3 编码与译码 . . . 247
§15.4 实现一 . . . 249
§15.5 构造 Huffman 树 . . . 251
§15.6 设计 . . . 252
§15.7 实现二 . . . 252

第十六章 抽象数据类型 259
§16.1 类型表示 . . . 259
§16.2 Haskell 抽象数据类型机制 . . . 260
§16.3 队列 . . . 263
§16.4 设计 . . . 266
§16.5 仿真 . . . 267
§16.6 实现仿真 . . . 269
§16.7 查找树 . . . 272
§16.8 集合 . . . 277
§16.9 关系和图 . . . 283
§16.10 评论 . . . 289

第十七章 惰性计算 291
§17.1 惰性计算 . . . 291
§17.2 计算规则与惰性计算 . . . 293
§17.3 再谈列表概括 . . . 297
§17.4 数据导向编程 . . . 303
§17.5 实例：分析表达式 . . . 306
§17.6 无穷列表 . . . 314
§17.7 为什么使用无穷列表 . . . 319
§17.8 实例：仿真 . . . 322
§17.9 再谈证明 . . . 323

第十八章 交互式程序设计 331
§18.1 为什么 I/O 是一个问题 . . . 331
§18.2 输入/输出 . . . 332
§18.3 do 记法 . . . 334
§18.4 迭代与递归 . . . 337
§18.5 计算器 . . . 341
§18.6 进一步的 I/O . . . 343

§18.7 再谈 do 记法 344
§18.8 函数程序设计中的 monads 345
§18.9 树上的 monadic 计算 350

第十九章 时间和空间行为 **355**
§19.1 函数的复杂度 355
§19.2 计算的复杂度 358
§19.3 集合的实现 362
§19.4 空间行为 363
§19.5 再谈折叠 366
§19.6 避免重复计算：记忆 369

第二十章 结论 **375**

附录一 函数式，命令式和 OO 程序设计 **379**

附录二 名词解释 **387**

附录三 Haskell 运算符 **389**

附录四 理解程序 **391**

附录五 Haskell 的实现 **393**

附录六 Hugs 错误 **395**

索 引 **398**

前言

计算机技术的更新可谓日新月异，然而其基本理论却几乎保持不变。现代计算机的尺寸和性能与其诞生初有着巨大的差别，但是它的基本结构自其半个世纪诞生以来几乎没有变化。在程序设计领域，诸如面向对象的现代思想从研究领域走向商业主流经历了几十年。从这个意义上讲，像 Haskell 这样的函数程序设计语言是相对年轻的，但是它的影响力将与日俱增。

- 函数程序设计语言被不断地应用于大型系统的**组成部分**，如在 Fran(Elliott, Hudak 1997) 中，Haskell 被用于描述动画，然后被转换为低级语言。在这里，中间运算的引入并没有牺牲函数程序语言语义的优美。
- 函数程序设计语言提供了一个**构架**，现代程序设计的中心思想在其中体现得淋漓尽致。为此，函数程序设计语言已经成为流行的计算机科学教学语言并且影响着其他语言的设计。例如，G-Java 的设计，其中的通用性便直接建立在 Haskell 多态模式上。

本书是 Haskell 函数程序设计简介。下面的引言将首先简单解释函数程序设计以及学习它的原因。之后解释本书的讲解方式和主要内容。对于读过第一版的读者，请留意第二版在方式和内容上的变化的说明。最后一节说明阅读本书内容的不同方式。

什么是函数程序设计?

函数程序设计为用户提供了一个高层的程序设计视野，由此带来的各种特点可以帮助用户建立优美、强大和通用的函数库。函数程序设计的中心概念是函数。一个函数计算其输入值的输出结果。

函数程序语言的表现力与通用性的一个例子是函数 `map`，它通过某种方式作用于列表的每个元素。例如，`map`可用于将一列数中的每个数加倍，或者将一个图形列表中的每个图形反色。

函数程序设计的简洁性是函数定义方式的结果：一个等式用于说明函数对任意输入的结果。一个简单的例子是函数 addDouble，它将两个整数相加，然后将和加倍。其定义如下：

```
addDouble x y = 2*(x+y)
```

其中 `x`和 `y`是输入，`2*(x+y)`是结果。

函数程序设计的模型简单利落：欲计算下列表达式的值

```
3 + addDouble 4 5
```

只需使用表达式所含的函数定义的等式，即

```
  3 + addDouble 4 5
⇝ 3 + 2*(4+5)
⇝ ...
⇝ 21
```

这便是计算机如何求一个表达式的值的过程，这也是使用纸和笔计算表达式的值的过程，由此使得过程的实现具有透明性。

我们还可以讨论程序的一般特性。对于上例 addDouble，利用`x+y`和`y+x`相等的性质，我们可以断言，对于所有的 `x`和 `y`，`addDouble x y`和`addDouble y x`相等。这样的证明思想在函数程序语言中较传统的命令式语言和面向对象 (OO) 语言更容易实现。

Huskell 与 Hugs

本书使用程序设计语言 Haskell。对于多数计算机系统 Haskell 提供了免费的编译器和解释器。本书使用 Hugs 解释器，它为初学者提供了理想的平台，并且具有编译周期快，界面简洁的特点。Hugs 提供 Windows, Unix 和 Macintosh 免费版本。

Haskell 始于二十世纪八十年代后期，原设计为惰性函数程序设计的标准语言，并历经多种变化与修改。本书使用 Haskell 98 编写。Haskell 98 建立在到目前为止 Haskell 的最新成果上，并将保持稳定；进一步的扩展将形成 Haskell 2, 但是，这之后系统将仍然支持 Haskell 98。

虽然本书涵盖了 Haskell 98 的大部分特性，但是，本书是一本程序设计教程，而不是语言参考手册。有关语言及其函数库的细节参见语言报告和库报告，并可从 Haskell 网页下载：`http://www.haskell.org`。

为什么要学习函数程序设计？

函数程序设计语言提供了一个简单的程序设计模型：一个值（结果）是在其他值（输入）的基础上计算出来的。

函数程序设计语言的简单基础为现代计算的中心概念提供了最清楚的解释，其中包括抽象（函数中），数据抽象（抽象数据类型），通用性，多态和重载。这表明函数程序语言不仅提供了现代程序设计思想的理想介绍，而且对于更传统的命令式语言和面向对象的语言也提供了一个有益的视角。例如，Haskell 中树的数据类型是最直接的，而在其他语言中用户不得不使用指针链接的数据结构。

Haskell 不仅是一种好的教学语言；而且是一种实用的程序设计语言，并且支持 C 函数和其他基于模块编程语言等的接口。Haskell 已经被用于许多实际项目编程。有关扩展和项目等的详细信息参见最后一章。

谁应该阅读本书?

本教程是为计算机科学和其他学科的本科生准备的函数程序设计介绍。本书也可作为计算机科学的初学者，或者具备一定知识、初次学习函数程序设计的学生使用，两组学生均会发现内容既新颖又富有挑战性。

本书也可由程序员，软件工程师和其他想学习函数程序设计的人自学使用。

本书自成体系，但是有关命令和文件等基本知识在使用 Haskell 的实现时仍然是需要的。本书引入了一些逻辑记号，并且在引入时作了解释。如果读者了解函数 log, n^2 和 2^n 的图像，对于阅读第十九章会有帮助。

本书的讲解方式

在编写函数程序设计语言教程时，作者希望尽可能早地介绍语言的所有特性，同时不会给读者造成太大的负担。本教程的第一版引入了“自底向上”的概念，这意味着在介绍任何一个大型例子之前，教程需要花费 100 多页来讲解基本概念。

第二版采取了不同的途径：在第一章通过一个“图形”的实例研究非正式地引入关键概念，然后随着教程的深入再对这些概念做正式和详细的解释。Haskell 有一个很大的预定义函数库，特别是列表上的函数，本书对此进行了探讨，并且鼓励读者在未看到函数的定义之前使用这些函数。这种方式使读者学习进展更快，而且符合实际工作习惯：大多数程序建立在预定义的大型函数库上，因此，从开始便采用这样的方式对于读者是非常有益的经历。有关第二版的其他变化将在引言的后面一节介绍。

本书还具有下列特色：

- 本书包含了对于函数程序推理的详细介绍。推理的重点对象是自然数上的函数及列表函数，其原因有二：证明函数在列表上的性质似乎更实际；而且列表上的结构归纳原理更容易被学生接受。
- 图形一例在第一章引入，并贯穿全书；由此读者可以看到同一问题的不同编程方式，并且有机会对不同的设计进行比较和思考。
- 第四章和第十一章强调了函数设计（编码之前完成）。
- 强调了 Haskell 是一种实用程序设计语言，所采取的方式是及早地介绍了模块的概念，I/O 和其他基于 Monad的应用的 do 表示法。
- 类型在本书中具有重要的作用。在定义每个函数或者对象的同时，其类型也一并引入。类型的引入不仅便于检查一个定义具有作者期望的类型，而且我们把类型视为一个定义文档的重要组成部分，因为一个函数的类型精确地描述了如何使用这个函数。
- 分阶段介绍一些实例研究：前面提到的图形例子，交互式计算器程序，基于 Huffman 编码的编码和译码系统以及一个小队列模拟程序包。通过这

些例子来介绍新概念，并且说明现有的技术如何协同工作。

- 最后一章包括有关 Haskell 的支持材料，以及大量的网站链接。附录包括了其他的有用信息，如哪些实现是可以使用的；Hugs 常见的错误；以及函数程序设计，命令式和 OO 程序设计之间的比较。
- 其他的支持材料可以参阅本书的网页 `http://www.cs.ukc.ac.uk/people/staff/sjt/craft2e/`

内容概要

第一章的简介包括函数程序设计的基本概念：函数与类型，表达式与求值，定义与证明。一些更深入的思想，如高阶函数和多态，通过由字符构成的图形一例作了简单介绍。第二章介绍 Haskell 的 Hugs 实现，传统脚本和文学式脚本的加载和运行，模块系统的基本知识，并且包含一个使用 Picture 类型的练习。这两章共同构成 Haskell 函数程序设计课程的基础。

第三章包括如何在整数、字符和布尔值上构造简单程序。自本章开始，我们将通过习题复习基本内容。在此基础上，第四章再次讨论定义函数的各种策略，特别是强调了使用其他函数（系统提供的函数或者用户编写的函数）的重要性。本章还讨论了“分治法”的概念，介绍了自然数上的递归。

第五章介绍数组和列表形式的结构化数据。在引进列表概念之后，我们介绍应用两种资源在列表上编程：使用有效地表达了函数 map 和 filter 的表现力的列表概括；使用一阶引导库和其他库函数。几乎所有的引导库列表函数都是多态函数，为此在这里引入了多态的概念。第六章包括各种扩展的例子；第七章介绍了列表上的原始递归，随后介绍了一个更复杂的文本处理的例子。

第八章引入列表处理程序的推理，其中包括一系列相关逻辑知识的内容。本章介绍了构造归纳证明的指导原则，并且专设一节来介绍如何从失败中构造成功的证明。

第九章和第十章介绍高阶函数。第九章首先讨论函数参数，并且证明函数参数使我们容易实现列表上的许多计算模式。第十章介绍将函数作为结果，这样的函数既可以定义为 λ 表达式，也可以定义为部分应用的函数；通过图形例子和索引例子进一步解释这些概念。随后的第十一章讨论程序设计中开发周期的作用。

类型分类允许函数重载，即函数在不同的类型上其实现也不同；第十二章包括类型分类和 Haskell 的各种内置分类。因为类型分类的存在，Hasekll 的类型系统比较复杂，所以，第十三章探讨 Haskell 的类型检测方法。一般地讲，类型检测的任务是解决函数的定义置于其类型上的各种约束。

在编写大型程序时，用户必须能够自己定义类型。Haskell 提供两种支持自定义类型的方式。代数类型（如树）是第十四章的主题，本章包括代数类型

从设计和证明到它们与类型分类的交互作用等各个方面，并且介绍了许许多多实践中的代数类型例子。这些例子在第十五章中得到进一步的巩固，本章讨论使用 Huffman 编码对信息进行编码和译码的实例研究。实例研究首先介绍此方法的基础，然后讨论程序的实现。该系统被分解为容易处理的多个模块，为此介绍了 Haskell 模块系统的进一步特性。

一个抽象数据类型 (ADT) 提供了通过一个限制的函数集对一个实现的访问。第十六章介绍 Haskell 的 ADT 机制并且通过大量的例子说明如何使用 ADT，如队列，集合，关系等，此外给出了一个模拟例子的基本函数。

第十七章介绍 Haskell 的惰性计算。惰性计算为程序员提供了一种处理回溯和无穷数据结构的独特方法。作为回溯的例子，我们研究语法分析，并且使用无穷列表编写 “进程式程序” 和一个随机数生成器。

Haskell 程序通过 `IO`类型实现输入和输出。第十八章介绍 `IO`类型。`IO`类型的成员表示基于动作的程序。许多程序很容易用 do 表示法编写。本章先介绍 do 表示法，并给出一系列例子，最后以一个交互式前端计算器结束。do 表示法的基础是 Monad。Monad还可用于不同类型的基于动作的程序设计，本章的后半部分将对此作进一步介绍。

第十九章将讨论程序的性能，即程序计算其结果所需的时间和空间。本书的最后一章即第二十章, 包括各种应用的概述，Hasekll 的扩展和进一步的研究方向，其中提供了许多的网页地址和参考资料。

附录涵盖各种背景资料。附录 A 提供了有关函数程序设计和 OO 程序设计的链接；附录 B 是函数程序设计常用词汇表；其他附录包括 Haskell 运算符和 Hugs 错误汇集，以及有助于理解 Haskell 程序和 Haskell 的各种实现的信息。

本书的所有例子的 Haskell 代码以及其他背景资料均可以从本书的网页上下载。

第二版的变化

致读者

这本 Haskell 函数程序设计简介是一个有机整体，但是, 读者可以跳过某些内容，或者按照不同的顺序阅读某些内容。

本书的内容是按照作者认为自然的顺序编排的。这种安排反映了内容之间的某些逻辑依赖关系，但是某些在书中后面出现的内容也可以在早些时候阅读。特别地，第十八章介绍 I/O 的前四节和第十四章有关代数类型的前几节均可以在第七章后阅读。第六章介绍的由 where 和 let 引入的局部定义可以在第三章后阅读。

如果读者只想阅读某个主题的部分内容，一种方法是读者在阅读有关章节时不必读完整章内容。下面将介绍其他剪接内容的方法。

程序验证是始于第八章，贯穿于 10.9,14.7,16.10 和 17.9 节的线索，读者可以根据需要选择阅读这些内容。类似地，第十九章关于程序的时间和空间性能的内容自成一体，读者也可以根据需要选择阅读。

书中有些内容是技术性的，读者可以在第一次阅读时省略。这些内容包括在 8.7 节和 13.2 节，并有“可省略”的字样。

最后，读者可以跳过某些例子和实例研究。例如，6.2 和 6.4 节扩展的习题集可以省略；文本处理 (7.6 节) 和索引 (10.8 节) 也可以省略，这些例子的目的在于强化训练以及显示 Haskell 在大型系统中的应用。在后面的章节中，读者也可以跳过某些例子，如 14.6 和 16.7-16.9 节以及第十七章的例子，但是，略去太多的例子也会失去一些有关动机的材料。

第十五章的前两节介绍模块，剩余的部分是 Huffman 编码的实例研究，这部分也是可以选读的。最后，分布于后几章的计算器和模拟实例研究也是可以选读的。但是，略去计算器的实例研究也将丢掉语法分析和代数类型及抽象代数类型的重要内容。

鸣谢

我特别感谢 Ham Richards, Bjorn von Sydow 和 Kevin Hammond 对第一版的反馈意见，感谢那些指出书中印刷错误的读者。感谢阅读第二版手稿的 Tim Hopkins, Peter Kenny 和匿名审稿人。

非常感谢 Addison-Wesley 的 Emma Mitchell 和 Michael Strang 自本书接受出版之日起所给予的支持。

Jane 对本书前两章的阅读和评论对我非常有帮助，在此我表示特别的感谢，同时感谢她在我编写本版的一年中所给予的支持。最后，感谢 Alice 和 Rory 毫不犹豫地与 Haskell 分享我们的家庭电脑。

Simon Thompson
Caterbury, 1999 年 1 月

第一章　函数程序设计简介

本章将以 Haskell 函数程序设计语言为背景。本章的目的有三个：

- 介绍函数程序设计的主要思想，解释函数和类型的概念。介绍求表达式的值以及如何书写求值的过程。介绍如何定义一个函数，并且证明一个函数具有某种特定的性质。
- 我们将以图形为例，来解释以上的概念。
- 最后，我们将介绍函数程序设计中更具表现力，并且与众不同的思想。由此可以看出，函数程序设计是如何不同于其他方式的程序设计，如面向对象的程序设计，以及为什么我们认为函数程序设计对于学习计算机科学的人们具有重要意义。

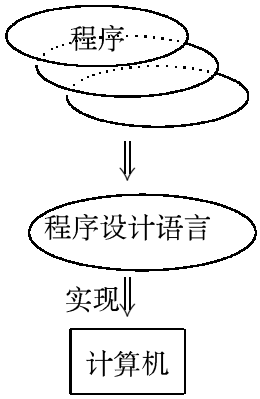
§1.1　计算机与建模

在过去的五十年中，计算机已经由庞大、昂贵、稀少、低速及不可靠的机器过渡到小型、廉价、普及、快速和（相对）可靠的机器。第一代计算机是“孤立”的机器，而现代的计算机可以实现很多不同的功能，可以组织成网络，可以嵌入汽车和洗衣机等家用电器，或者安装在个人计算机 (PCs) 中，等等。

尽管如此，计算机的基本原理在这个时期没有多大的变化：计算机的目的是处理符号信息。这些信息可以是简单的情况，如超市的购物清单，或者更复杂的情况，如关于欧洲的气象系统。给定这些信息，我们的任务是计算超市购物的总支出，或者计算英格兰南部 24 小时的气象预报。

这些任务如何得以完成呢？我们需要描述信息是如何处理的。这个过程描述称为一个 **程序**，它是用一个 **程序设计语言** 编写的。一个程序设计语言是一个计算机指令的人工形式化语言。换言之，这种语言用于编写控制计算机 **硬件** 的 **软件**。虽然计算机自诞生以来，其结构基本保持相似，但是，计算机的编程方式已经经历了巨大的变化。最初的程序是用直接控制计算机硬件的指令编写的，而现代程序设计语言和其要解决的问题处于同一个” 高” 层次，而不是处于” 低” 层次，或者机器层。

程序设计语言由一个 **实现** 来完成它在计算机上的工作，这个实现本身也是一个程序，其任务是运行由高阶语言编写的程序。

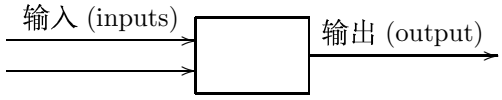


我们的任务是设计程序，所以我们将关心的是上图的上半部分，而非实现的细节（有关实现参见 Peyton Jones 1987; Peytong Jones 和 Lester 1992）。

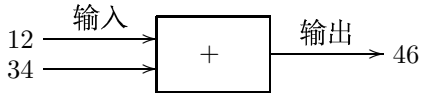
我们讨论的主题是 **函数程序设计**。函数程序设计是众多不同的程序设计方式或者 **种类** 的一种，其他的程序设计方法包括面向对象 (OO)，结构程序设计和逻辑程序设计等。为什么有不同的程序设计方法，以及它们有何不同呢？一种研究程序设计语言的有效方法是分析它在计算机中如何为实际问题或者想象的问题 **建模**。每类程序设计语言提供了不同的建模工具，这些不同的工具允许我们 - 或者迫使我们 - 用不同的方式思考问题。例如，函数程序员考虑的是值之间的关系，而 OO 程序员考虑的是对象。在介绍函数程序设计之前，我们需要先弄清函数的概念。

§1.2 什么是函数？

一个 **函数** 可以如下图示为一个方盒，一些输入和一个输出：



函数产生一个依赖于 **输入** 值的 **输出**。我们常用 **结果** 这个术语来代替输出，用术语 **自变量** 或者 **参数** 代替输入。数的加法运算 + 便是一个简单的函数例子。给定两个输入 12 和 34，对应的输出将是 46。



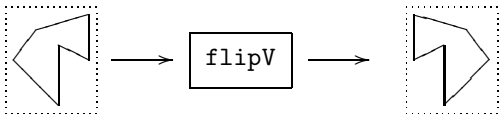
为函数提供特定输入的过程称为 **函数的应用**，(12+34) 表示将函数 + 应用于 12 和 34。加法是一个数学函数，但是函数出现在许多其他情形下，例如：

- 输出两个城市（输入）间距离（输出）的函数；
- 超市的结账程序: 给定一系列扫描过的条形码（输入），计算出相应的账单（输出）；
- 化工厂的阀门控制器，其输入是传感器的信息，输出是传到阀门控制器的信号。

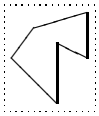
我们在前面提到，不同种类的程序设计语言的特点是其提供了不同的建模工具：在函数程序设计语言中，函数将是模型的主要组成部分。我们将会在下面贯穿本书的图形例子中看到这一点。

§1.3 图形与函数

本章将以二维图形为例，研究它们在计算机系统中的表示。这个例子也将贯穿本书。在这里，我们只想说明图形间的很多关系是由函数来表述的。在本节，我们将讨论一系列这样的例子。在垂直镜子中的反射将两个图形相关联，这种关系可以由一个函数 `flipV`来描述：



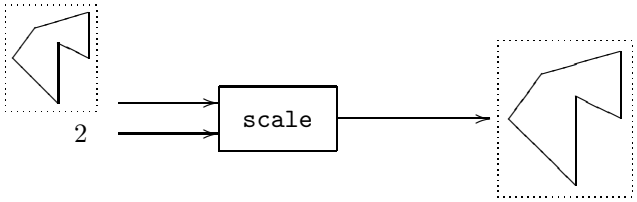
在这里我们演示了将如下“马”的图形翻转后的结果：



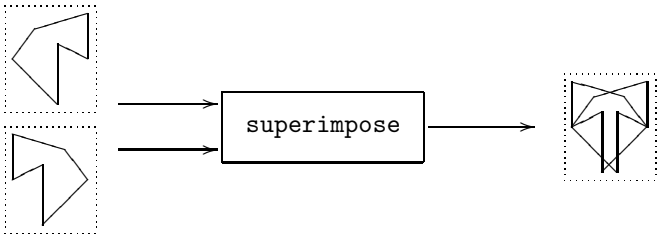
类似地，我们可以用一个函数 flipH 表示在水平镜子中的翻转。

一个表示将（单色）图形反色的函数如下：

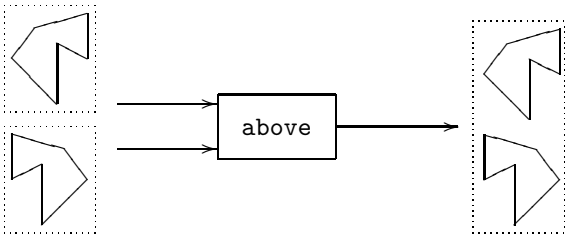
某些函数具有多个参数，比如将图形拉伸的函数：



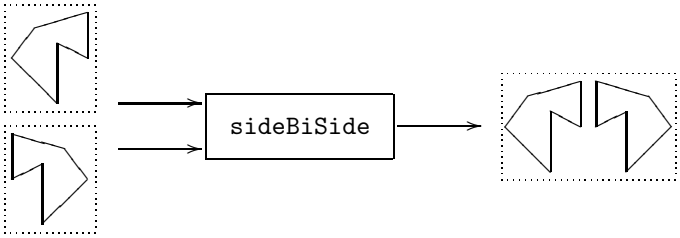
一个将两个图形叠加的函数：



将一个图形放在另一个图形之上的函数：



以及将两个图形并列的函数：

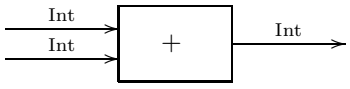


我们现在已经认识了函数的概念和一些函数的例子。在解释函数程序设计之前，我们先来了解另一个概念：类型。

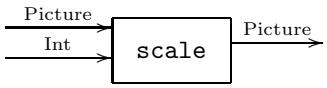
§1.4 类型

函数程序中的函数将涉及各种不同种类的值：加法函数把两个数相结合，给出另一个数；`flipV`将一个图形转换为另一个图形；`scale`取一个图形和一个数值，然后给出另一个图形，等等。

一个 **类型** 是一些同类值的一个集合，例如，数构成的集合，图形构成的集合。同类的值尽管各不相同，如 2 不同于 567，但是可以应用到这些值的函数是相同的。例如，求两个数中的较大者是有意义的，但是，比较一个图形与一个数是无意义的。如果我们看一下加法函数 `+`，只有两个数相加有意义，而两个图象相加无意义。这个例子说明，我们所讨论的函数本身有一个类型，并且可以用下图表示之：



上述图表表示 + 由两个整数作为输入，返回一个整数作为结果。类似地，函数 `scale`可以用下述图表表示：



它的第一个参数是一个 `Picture`，第二个参数是一个 `Int`, 其结果是一个 `Picture`。在函数程序设计的中心思想中，至此我们已经介绍了其中的两个：一个类型是一个值的集合，如整数的集合；一个函数是带有一个或者多个参数，并能返回一个结果的运算。 这两个概念是相互关联的：函数作用于特定的类型上：拉伸图形的函数带有两个参数，一个类型是 Picture, 另一个的类型是 `Int`。在为一个问题建模时，一个类型可以表示一个概念，如" 图形"，一个函数表示这些对象可以被处理的一种方法，比如，将一个图形搁置在另一个图形之上。我们将在 1.11 节对类型继续讨论。

§1.5 函数程序设计语言 Haskell

Haskell (Peyton Jones 和 Hughes 1998) 是我们将在本书中使用的函数程序设计语言。但是，本书中的许多内容具有更普遍的意义，同样适用于其他的函数程序设计语言 (参见第二十章)，也适用于一般的程序设计。不过，本书对于使用 Haskell 的函数程序设计具有更重要的价值。

Haskell 是以发明 λ 演算的先驱之一 Haskell B. Curry 命名的。λ 演算是关于函数的数学理论，它为多种函数语言的设计者提供了灵感。Haskell 的设计始于二十世纪八十年代后期，并且经历了多次修改，达到目前的标准状态。

Haskell 有多种实现。本书将使用 **Hugs** (1998) 系统。我们认为 Hugs 为初学者提供了最好的环境，包括免费的 PC 版本，Unix 版本以及 Macintosh 系统版本，并且具有高效、紧凑和用户界面灵活等特点。Hugs 是一个解释器。这意味着像我们在纸上作计算一样，Hugs 一步步地计算表达式的值。因此，它的效率不及将程序直接翻译成计算机机器语言的编译器。经编译的诸如 Haskell 语言的程序其运行速度与更传统的 C 和 C++ 等程序相似。有关 Haskell 不同实现的细节参见附录 E 以及 Haskell 主页：https://www.haskell.org/。

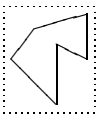
从现在开始，我们将以 Haskell 函数程序设计语言和 Hugs 系统为基础，介绍函数程序设计思想。有关如何获得 Hugs 系统，参见附录 E。本书中的所有程序和例子均可从本书的网页上下载：`http://www.cs.ukc.ac.uk/people/staff/sjt/craft2e/`

§1.6 表达式与计算

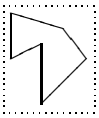
在小学一年级我们学会了计算一个表达式的值，如`(7 - 3)*2`

表达式 值
$(7-3)*2 \Longrightarrow 8$
求值

结果是 **值** 8 。此表达式是由表示数的符号和数上的函数的符号（减法 - 和乘法 *）构成的。表达式的值是一个数。这样的计算过程在计算器中是自动化的。在函数程序设计中，我们所做的工作正是如此：计算表达式的值，只是表达式包含了为特定问题建模时定义的函数。例如，在为图形建模时我们需要计算其值为图形的表达式。如果下面的图形称为 `horse`,



那么我们可以将 `flipV`应用于 `horse`，从而构成一个表达式。这样的函数应用表示为函数名后跟随其参数，如`flipV horse`, 其计算过程为

表达式 值
`flipV horse` $\Longrightarrow$ 
求值

一个更复杂的表达式是`invertColour (flipV horse)` 其结果是将一个 horse 经垂直镜子反射，即`flipV horse`, 然后将其反色，其结果是

总之，在函数程序设计中，我们通过计算包含问题领域函数的表达式之值来解决问题。我们可以把一个函数程序设计语言看作与电子计算器类似的东西：输入一个表达式，系统计算出表达式的值。程序员的任务是编写将问题模型化的函数。

一个 **函数程序** 是由一系列函数和其他值的定义组成的。下面介绍如何书写这些定义。

§1.7 定义

一个 Haskell 函数程序由一系列的 **定义** 组成。一个 Haskell 定义将一个 **名**（或者 **标识符**）与一个特定 **类型** 的值相联系。最简单的定义具有如下形式：

```
name  :: type
name = expression
```

例如

```
size :: Int
size = 12+13
```

在这个定义中，位于左边的名 size与右边表达式的值25关联，其类型为 Int，即整数的类型。符号"::" 读作" 具有类型"，上述定义的第一行读作"size 具有类型 Int"。注意，函数和值的名要以小写字母开头，而类型名要以大写字母开头。假设我们有了 horse 和前面讨论过的 Picture上的各种函数的定义（我们将在第二章讨论如何下载这些定义以及在一个程序中使用它们），那么现在可以利用这些图形上的运算。例如，我们可以定义

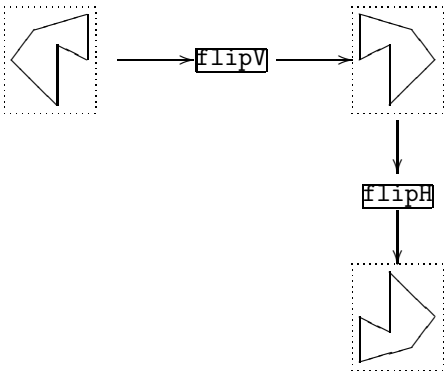
```
blackHorse :: Picture
blackHorse = invertColour horse
```

所以，与 blackHorse 联系的 Picture是由函数 invertColour 应用到 horse 得到的，其结果为

另一个例子是下列的定义

```
rotateHorse :: Picture
rotateHorse = flipH (flipV horse)
```

我们可以将右边表达式的计算图示如下：



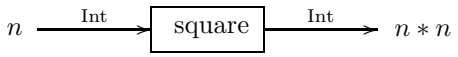
这里假定函数 flipH 的作用是将一个 Picture 在水平镜子中反射。两个反射的结果是将图形旋转 180° 。在 1.6 节我们将 Hugs 系统解释为与计算器非常类似的系统。那么 Hugs 如何计算象 size-17 的表达式呢? 利用先前给出的 size 的定义, 我们把 size 用其右边的表达式代替, 由此得到 (12+13)-17, 经过简单的算术运算, 得到表达式的值为 8。到目前为止，我们看到的只是常数值的定义，下面介绍如何来定义函数。

§1.8 函数定义

我们也可以定义函数，先看一些简单的例子。我们可以定义一个求整数平方的函数：

```
square :: Int -> Int
square n = n*n
```

此定义也可以用下面的图表示



函数 square 的 Haskell 定义的第一行说明它的类型：square 是一个函数。一个函数类型由箭头表示。箭头左边表示其输入参数的类型 (这里是 Int)，箭头右边表示其结果的类型 (这里是 Int)。定义的第二行给出函数的定义：当函数 square 应用于一个 **未知量** 或者 **变量**n 时，其结果是 n*n。如何来解释此定义呢？定义等式中的 n 是任意的，或者未知的。定义说明，无论 n 取何值，等式均成立。例如，

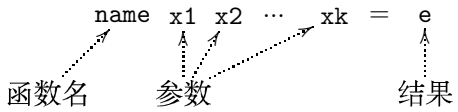
```
square 5 = 5*5
```

以及

```
square (2+4) = (2+4)*(2+4)
```

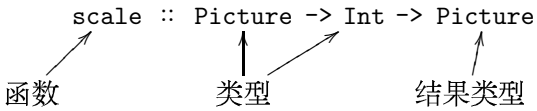
这便是利用等式计算包含 square 的表达式的方法。若要计算将 square 应用于表达式 e，将表达式`square e`用定义等式中右边的表达式代替，即 e*e。

一般地，一个简单的函数定义具有如下的形式：

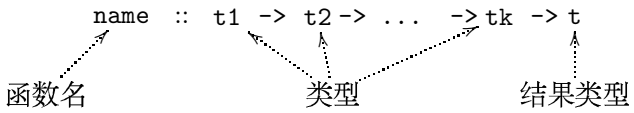


在函数定义的等式左边的变量称为 **形式参数**，因为它们代表参数（有时称为 **实际参数**）的任意值。在大多数情况下，两种参数的含义不言自明时，我们将两者统称为“参数”，只有需要将两者区分开来时，才冠以“形式”或者“实际”。

伴随着函数的定义的是它的 **类型说明**。我们以函数 scale 为例说明其格式:



一般地，一个函数的类型说明具有如下格式:



由 1.7 节的函数 rotateHorse 可以设想一个旋转函数的定义。可以通过两个反射得到一个图形的旋转, 为此定义

```
rotate :: Picture  -> Picture
rotate pic = flipH (flipV pic)
```

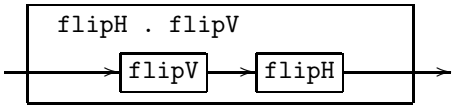
上述等式表示: 若要旋转图形 pic, 首先将 flipV 作用于 pic 得到 (flipV pic), 然后将后式在水平镜中反射, 从而得到 flipH (flipV pic)。利用此定义, rotateHorse 的定义可表示如下:

```
rotateHorse :: Picture
rotateHorse = rotate horse
```

即: rotateHorse 的结果是将 rotate 应用于图形 horse。rotate 定义的模式在 Haskell 中具有普遍性, 即将一个函数应用于其参数, 然后将另一个函数应用于前面的结果。这种模式给出一种将两个函数相结合的方法。我们定义

```
rotate :: Picture -> Picture
rotate = flipH . flipV
```

定义中的符号"." 代表 **函数的合成**, 其中一个函数的输出是另一个函数的输入, 如下图所示:



通过将两个函数的输入和输出相连接, 构造出一个新的函数。由此可以设想, 函数可以有许多其他的连接方式。我们将在以后的章节中作进一步介绍。

函数的直接合成提供了函数程序设计威力的第一个例子: 我们可以利用运算"." 将函数结合, 就如使用运算"+" 将数结合在一起一样。这里我们使用了 **"运算"** 这个术语, 而非函数, 因为符号'.' 写在其参数中间, 而非其参数前面。我们将在 3.7 节详细讨论运算。

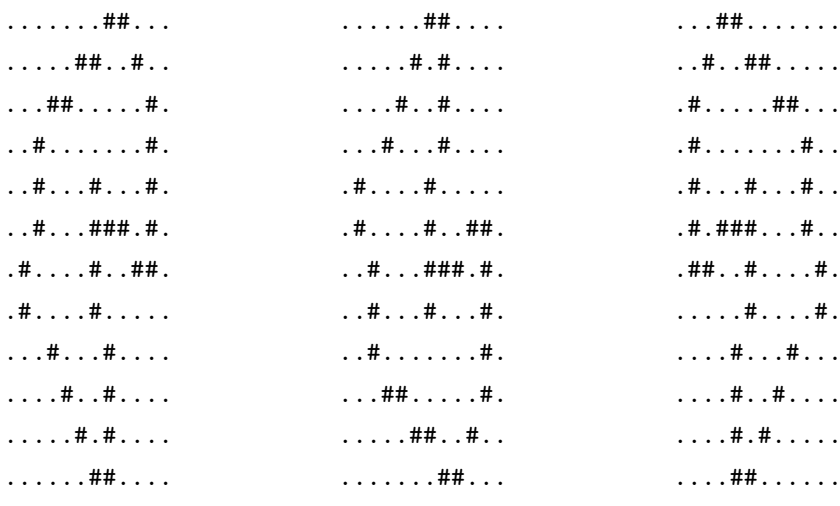
类型抽象

在介绍新内容之前, 我们指出今后将在本书中探讨的另一个重要概念。我们已经定义了

```
blackHorse :: Picture
rotate     :: Picture -> Picture
```

其中使用了类型 Picture 和其他定义在 Picture 上的函数, 如 flipH 和 flipV。我们可以定义 blackHorse 和 rotate而无需了解类型 Picture的或者其上的"翻转"函数的任何细节, 而只需知道这些函数给出我们期望的结果。

如此处理类型的方式称为 **类型抽象**: 类型的使用者不需要关心类型是如何定义的。其优点是, 无论图形是如何刻画的, 用户的定义永远适用。或许在不同的场合, 图形有不同的模型, 但是, 在任何情况下, flipH . flipV 的结果



都是将图形旋转 180°。第十六章将详细讨论 Haskell 支持类型抽象的机制。下一节介绍 Haskell 的其他重要特色。

§1.9 一个图形模型

在第一章介绍本节内容的原因有二。首先，由此出发介绍 Pictures 在Haskell中建模的一种直观方法。其次，概述 Haskell 的特色，正是这些特色使得Haskell成为一种有别于其他语言的有力编程工具。在介绍这些特色的时候，我们将说明在本书的哪些地方讨论这些内容。

我们的模型是由 **字符** 构成的单色二维图形。字符是单个的字母，数字，空格等可由计算机键盘键入并且在计算机屏幕上显示的符号。在 Haskell 中，字符的类型是内置的 Char.

这个模型的特点是在计算机屏幕上显示时简单易行。另一方面，我们可以建立更复杂的图形模型，参见第 19 页所示的本书网页。

上图是 horse 的图形以及它在水平和垂直镜子中反射的图形，其中小园点表示图形的白色部分。

如何由字符构造图形呢？在我们的模型中，一个图形是由一些线段的 **列表** 组成的，即线段从上到下的有序排列。每个线段也可以看成为一个字符的列表。因为在编写程序时，经常需要处理一些元素的集合，所以列表在 Haskell 中是内置的。具体地说，给定任意类型（如字符或者线段），Haskell 包含一个此类型的列表类型。特别地，我们可以使用前面解释的方法为图形建模，即使用字符的列表表示线段，线段的列表表示图形。

建立了这样的 Pictures 模型后，我们可以考虑如何实现图形上的函数。第一个定义很简单：一个图形在水平镜子中反射不改变每一条线段，但是线

段的顺序要颠倒，也就是将线段列表逆转：

```
flipH = reverse
```

其中reverse 是一个将列表中元素逆转的内置函数。那末如何在垂直镜子中反射一个图形呢？线段的顺序不受影响，但是每条线段被逆转。其表示如下：

```
flipV = map reverse
```

其中map是一个将函数分别作用于列表中每个元素的 Haskell 函数。由上述flipH和filpV 的定义可以看出 Haskell 函数程序设计的威力和优美。

- 我们使用了reverse在flipH 中逆转线段的列表，在filpV中逆转每条线段: 这是因为函数reverse的同一个定义可以应用于任何类型的列表。这是多态，或者通用程序设计的例子。5.6 节将对此作详细讨论。
- 函数flipV的定义使用了函数map应用于其参数reverse, 此参数本身也是一个函数。这使得map成为一个非常通用的函数，因为通过改变其参数, map可以实现在列表上的任何动作。这将是第九章讨论的内容。
- 最后，map应用于reverse的结果仍然是一个函数。这是第十章讨论的内容。

后两个事实表明函数是“一等公民”，因此可以像对待任何其他对象 (如数和图形) 一样地对待函数。这一点和多态相结合意味着在函数程序设计语言中，我们可以编写如reverse和map这样的非常通用的函数，并将其应用于许多不同的场合。

前面所举的例子带有很大的普遍性。图形上的其他函数同样具有类似的简单定义。将一个图形放置在另一个之上可以通过将两个相应的列表首尾相接, 形成一个新列表来实现[1]：

```
above = (++)
```

将两个图形并列放置可以通过将相应的线段连接来实现：

```
.......##...      ++     ......##....
.....##..#..      ++     .....#.#....
...##.....#.      ++     ....#..#....
..#.......#.      ++     ...#...#....
..#...#...#.      ++     .#....#.....
..#...###.#.      ++     .#....#..##.
.#....#..##.      ++     ..#...###.#.
.#....#.....      ++     ..#...#...#.
...#...#....      ++     ..#.......#.
....#..#....      ++     ...##.....#.
.....#.#....      ++     .....##..#..
......##....      ++     .......##...
```

[1]运算++ 由括号 (...) 包围后可解释为函数。3.7 节将对此加以说明。

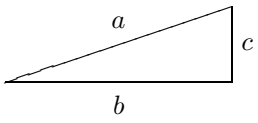
并且可用函数zipWith实现。函数zipWith应用某个函数将两个列表相应元素组合在一起，在此例中为运算 (++)：

```
sideByside = zipWith (++)
```

函数superimpose是更复杂的应用zipWith的函数。函数invertcolour也可以利用函数map来定义。在第十章我们将继续讨论这些例子。

§1.10 证明

本节讨论函数程序设计的另一个特点：证明。一个证明是一个说明某命题在所有情况下都成立的逻辑的或者数学的论证。例如，给定任意一个特定的直角三角形，

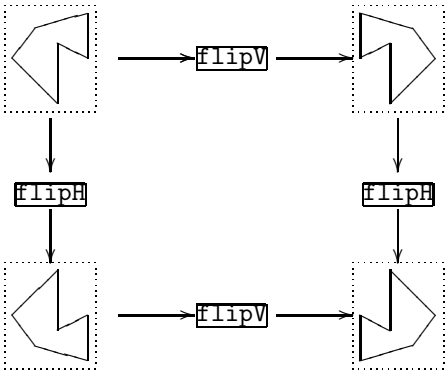


我们可以检验 $a^2 = b^2 + c^2$ 是否成立。在任何情形，这个公式均成立，但这并不能说明公式对于所有的 a, b 和 c 均成立。另一方面，一个证明是说明公式 $a^2 = b^2 + c^2$ 对于任意直角三角形均成立的通用论证。

那么证明和函数程序设计有何关系呢？要回答这个问题，先来观察 Picture 上的一个例子。由 1.8 节可知，我们可以定义

```
rotate = flipH . flipV
```

有趣的是，如果颠倒反射函数应用的次序，则合成的效果是一样的，如下图所示：



我们可以将此性质用函数间的简单等式表示：

```
flipH . flipV = flipV . flipH          (flipProp)
```

更进一步，我们可以根据flipV和flipH的实现，给出这些函数具有如上性质 (flipProp) 的逻辑 **证明**。其主要依据是两个函数的作用互不影响：

- 函数flipV只影响每个线段，而所有的线段保持原次序，
- 函数flipH将线段列表逆转，而并不改变任何线段。

因为两个函数对于线段的不同部分有影响，所以先应用哪一个并不重要，两种次序应用的最后结果均是：

- 逆转每条线段，逆转线段的列表。

对于多数程序设计语言来说，对程序进行证明是可能的，但是程序证明对于函数程序设计语言比其他任何类型的语言简单得多。程序证明将是本书的一个内容。第八章将开始探讨对列表函数的证明。

给出诸如(`flipProp`)的性质的证明有什么好处呢？证明确保我们的函数具有某个特定的性质。和计算中通常进行的 **测试** 做比较：我们选取某些点测试函数的值。测试只能告诉我们函数在测试点具有某种性质，而在原则上并不能告诉我们函数在其他情形是否具有此性质。在事关生命财产安全的情况下，我们期望确保程序按照计划运行，在这里证明便是最好的保证。不过，我们并没有说测试不重要。测试和证明在软件开发过程中具有互补的作用。

更具体地说，(`flipProp`)的含义是：我们可以确信，无论是使用`flipH . flipV` 还是`flipV . flipH`，其结果是一样的。因此，我们可以把一个使用`flipH . flipV`程序 **转换** 为使用相反顺序合成的函数`flipV . flipH`，并且保证新程序与原程序具有相同的作用。在 10.9 节我们将看到，这样的思想既可以应用于语言的实现，也可应用于程序的开发。

§1.11 类型和函数程序设计

类型在函数程序设计中的作用是什么呢？给定一个函数的类型，其首要的信息是如何使用此函数。如果给出

```
scale :: Picture -> Int -> Picture
```

我们立即得知下面两点：

- 首先，`scale`具有两个参数，第一个参数的类型是 Picture，第二个参数的类型是 Int，因此，`scale`可以应用于`horse`和`3`。
- 将`scale`应用于 Picture 和 Int 的结果是 Picture。

因此，类型有两个作用。第一，它表示了一种如何使用函数`scale`的 **约束**：`scale` 必须应用于一个 Picture 和一个 Int。第二，类型说明当函数被正确地使用时结果的类型，在这里是 Picture。

函数和其他表达式的类型不仅说明如何使用它们，而且有可能使得检查函数是否被正确地应用自动化，这个发生在 Haskell 中的过程称为 **类型检测**。如果我们使用表达式

```
scale horse horse
```

我们将会被告知，把`scale`应用于两个图形造成了类型错误，因为`scale`需要一个图形和一个数作为其参数。更进一步地，类型检测的完成不需要`scale`或者`horse`的值，只需要有关对象的类型。因此，像这样的 **类型错误** 是在计算表达式或者执行程序之前检测到的。

值得注意的是，无论是编程新手还是经验丰富的程序员，他们犯下的许许多多错误都是由于输入的错误或者对问题的误解而造成的。因此，类型系统能帮助用户编写正确的程序，避免一大部分明显的和细小的错误。

§1.12 计算与求值

我们可以把 Hugs 看作一个通用计算器，它可以使用函数程序中定义的函数和其他值。当我们求一个表达式的值时，如

```
23 - (double (3+1))                (dd)
```

我们需要使用下列函数的定义：

```
double :: Int -> Int
double n = 2*n                     (dbl)
```

将定义(dbl)中的未知数n由表达式 (3+1) 代替， 得到

```
double (3+1) = 2*(3+1)
```

现在可以用2*(3+1)替换(dd)中的double (3+1), 并继续求值.

函数程序设计的一个与众不同的特点是，这样的简单计算机模型有效地描述了函数程序的计算。因为模型是如此直观，求值过程可以 **逐步** 地进行。我们称这样的逐步求值为 **计算**。作为例子，我们说明本节开始讨论的表达式的计算：

$$
\begin{array}{ll}
23-(\mathtt{double}(3+1)) & \\
\rightsquigarrow 23-(2*(3+1)) & \text{using}(\mathtt{dbl}) \\
\rightsquigarrow 23-(2*4) & \text{arithmetic} \\
\rightsquigarrow 23-8 & \text{arithmetic} \\
\rightsquigarrow 15 & \text{arithmetic}
\end{array}
$$

其中 $\rightsquigarrow$ 表示一步计算， 并且在每一行的右边标明如何到达这一步。比如，第二行

$$\rightsquigarrow 23-(2*(3+1)) \quad \text{using}(\mathtt{dbl})$$

表示这一步计算是应用double函数的定义(dbl)得来的。

有时在计算过程中使用下划线标明下一步修改的部分，也是我们在阅读计算时关注的部分。将上面的计算加上下划线后的计算过程：

$$
\begin{array}{ll}
23-\underline{(\mathtt{double}(3+1))} & \\
\rightsquigarrow 23-(2*\underline{(3+1)}) & \text{using}(\mathtt{dbl}) \\
\rightsquigarrow 23-\underline{(2*4)} & \text{arithmetic} \\
\rightsquigarrow \underline{23-8} & \text{arithmetic} \\
\rightsquigarrow 15 & \text{arithmetic}
\end{array}
$$

在今后每当一个新的概念引入时，我们将使用这样的逐行求值模型，以探讨包含在这些新概念中的新思想。

小结

如我们开始所述，本章有三个目的。我们想介绍函数程序设计的基本思想, 并且通过图形的例子加以说明, 同时使读者感受到函数程序设计如何与众不同。我们在此汇总已经介绍的定义如下：

- 一个函数是将其输入转变为一个输出的东西；
- 一个类型是具有相同属性的对象的集合，例如整数或者图形；
- 每个对象有一个明确定义的类型，此类型在定义对象时说明；
- 程序中定义的函数用于编写表达式；
- 表达式的值可以通过手工计算，或者使用 Hugs 解释器计算。

在今后的章节，我们将探讨定义新类型和新函数的不同方法，进一步介绍多态的概念，函数作为参数和结果，数据抽象和证明等这些在本章接触到的题目。

第二章　Haskell 解释系统 Hugs

第一章介绍了 Haskell 函数程序设计的基础。本章的主要目的是介绍如何使用 Hugs 解释系统进行实际的程序设计。

在开始学习程序设计时，我们还将介绍 Haskell 模块系统的基本知识。使用模块系统，程序可以分解成多个相互独立的文件，并且可以使用引导库 Prelude 和其他函数库的内置函数。

我们将仍然使用第一章的 Picture 以及简单的数值计算作为程序设计例子。为此，我们将介绍如何下载本书的程序和其他背景资料，以及如何获得 Hugs。

最后，我们概述在使用 Hugs 系统时可能得到的错误信息。

§2.1　第一个 Haskell 程序

第一个 Haskell 程序或者 **脚本** 是第一章的几个数值例子。一个脚本包含定义和注释。

脚本中的一个 **注释** 是为读者（而非机器）提供的信息。一条注释可能说明一个函数的功能，如何使用一个函数等。

Haskell 脚本有两种不同的格式，也反映了两种不同的编程观点。

在传统的脚本中，所有的符号均解释为程序文本，除非明显地标明一段文本为注释。图 2.1 便属于这一类。传统风格的脚本使用'.hs' 作为扩展名。

注释可用两种方法说明。一行注释可用符号 '–' 开始。多行注释可用'{-' 和'-}' 括起来。后者可以包含任意长的注释，也可以包含其他注释，故称为 **嵌套注释**。

另一种书写脚本的方式称为 **文学型**：文件中的所有文字均为程序的注释，而将程序用某种方法明显地标明。图 2.2 是一个文学型脚本的例子。其中，程序文本用符号'¿' 开始，并用空白行与注释区别开来。这种脚本的扩展名为'.lhs'。

两种脚本强调了程序设计的不同方面。传统的脚本强调的是程序，而文学型的脚本强调，程序设计不仅仅是定义要正确。设计决策需要解释，使用函数等的条件应该写下来。这些不仅对于用户是有益的，而且对于我们程序员同样是非常有益的。比如，当我们想修改一个以前编写的程序时。我们也可以把本书视为扩展的 "文学型脚本"，其中的注释中穿插了使用打字机字体的程序。今后，证明和 URL 也将使用打字机字体。

下载程序

本书中所有的程序，以及其他有关资料，Haskell 和一般函数程序设计链

```
{-############################################################

        FirstScript.hs

        Simon Thompson, June 1998

  此脚本的目的是演示整数(Int)上的一些简单定义和第一个脚本例子。

############################################################-}

-- size 的值是一个整数(Int)，其定义为12与13的和。

size :: Int
size = 12+13

--   计算整数平方的函数。

square :: Int -> Int
square n = n*n

--  计算一个整数的两倍的函数。

double :: Int -> Int
double n = 2*n

--   使用double，square和size的例子。

example :: Int
example = double (size - square (2+2))
```

图 2.1: 一个传统的脚本例子

```
############################################################

  FirstLiterate.lhs

  Simon Thompson, June 1998
```

此脚本的目的是演示整数(Int)上的一些简单定义和第一个文学型脚本例子。

```
############################################################ size
```

size的值是一个整数(Int)，其定义为12与13的和。

```
>       size :: Int
>       size = 12+13
```

计算整数平方的函数。

```
>       square :: Int -> Int
>       square n = n*n
```

计算一个整数的两倍的函数。

```
>       double :: Int -> Int
>       double n = 2*n
```

使用double，square和size的例子。

```
>       example :: Int
>       example = double (size - square (2+2))
```

图 2.2: 文学型脚本的例子

图 2.3: 启动 Hugs 的 Windows 版本

接，均可以在本书的网页上找到：

`http://www.cs.ukc.ac.uk/people/staff/sjt/craft2e/`

其中的定义使用了文学型的脚本。

§2.2 使用 Hugs

Hugs 是 Haskell 的一个实现。它既可以运行在使用 Windows95 和 NT 的 PC 机上，也可以运行在 Unix 和 Linux 系统上。系统可以在 Haskell 主页上免费下载：`http://www.haskell.org/hugs/`

这个网页也包含有关 Haskell 及其实现的许多资料。有关下载和安装 Hugs 的详细说明，参见附录 E。

在这里，我们介绍 Hugs 的字符型界面，如图 2.3，因为此界面在 Uinx 和 Windows 上是相同的。有经验的 PC 机用户，一旦懂得 Hugs 是如何运行的，将会很容易使用 Winhugs 系统。Winhugs 使用了 Window 风格的界面，在其中运行 Hugs 命令。

启动 Hugs

在 Unix 下启动 Hugs, 可键入 hugs; 使用一个特定的文件启动 Hugs, 键入 hugs 和此文件名, 例如，

```
hugs FirstLiterate
```

在 Windows 系统上, Hugs 可以通过开始菜单启动；双击一个文件图标可以在一个文件上启动 Hugs[1]。

[1]假定使用了标准的安装，完成了相应的注册项目

Haskell 脚本带有扩展名.hs 或者.lhs（文学型脚本）；只有这样的文件可以调入 Hugs 系统，而且这些扩展名可以在 Hugs 启动或者在 Hugs 系统内用:load 命令调入文件时省略。

在 Hugs 中求表达式的值

如 1.6 节所述，Hugs 解释器将对键入提示符后的表达式求值。如图 2.3，`size`的值是 25，`example`的值是 18。又如下面的例子：

```
Main> double 32 - square (size - double 3)
-297
Main> double 320-square (size - double 6)
471
Main>
```

这里我们使用了斜体表示系统的输出，非斜体表示用户的输入。**提示符** `Main>` 将在 2.4 节介绍。

由这些例子看出，可以使用当前的脚本计算表达式的值。在此例中，当前脚本是`FirstLiterate.lhs`或者`FirstScript.hs`。

Hugs 界面的一个优点是用户很容易通过键入各种表达式的值来对函数进行实验。如果用户想计算一个复杂表达式的值，最好将其定义写在程序中。例如，

```
test :: Int
test = double 320 - square (size - double 6)
```

用户只需在提示符`Main>`下键入`test`即可。

Hugs 命令

Hugs 命令以冒号':' 开始。下列是 Hugs 的主要命令：

`:load parrot`	加载程序`parrot.hs`或`parrot.lhs`。其中的扩展名`.hs`和`.lhs`可以省略。
`:reload`	重复执行前一个加载命令。
`:edit first.lhs`	使用缺省编辑器编辑文件`first.lhs`。注意，这里的扩展名`.hs`或`.lhs`不可省略。关于编辑脚本参见下一节。
`:type exp`	计算表达式`exp`的类型。如，`:type size +2`的结果是`Int`。
`:infor name`	显示有关名为`name`的信息。
`:find name`	用编辑器打开含有`name`的文件。
`:quit`	退出系统。
`:?`	列出 Hugs 的命令。
`!com`	转去执行 Unix 或者 Dos 命令`com`。

所有的':' 命令可以简化为第一个字母，如:`l parrot`等。有关其他命令，请使用 Web 浏览器阅读在线 Hugs 文档。如果 Windows 用户使用了标准安装，文档的位置是

```
C:\hugs\docs\manual-html\manual_contents.html
```

一般地，用户需要根据安装的位置查找相应的文档。

编辑脚本

Hugs 可以和一个“缺省编辑器关联，当用户执行命令:`edit`或:`find`时 Hugs 便调用这个缺省编辑器。Unix 的缺省编辑器是`vi`; Windows 用户可以使用`edit`或者 notepad。有关如何使用设置缺省编辑器等，参见附录 E。

命令:`edit`会使编辑器打开相应的文件。退出编辑器后，修改后的文件自动载入系统。但是，更方便的方法是在另一个窗口运行编辑器，并且通过下列方法加载文件：

- 在编辑器中存储文件（不需退出），然后
- 使用 Hugs 命令:`reload`或者:`reload filename`重新加载文件。

使用这种方法，用户可以随时修改文件。

下面列出在第一个程序例子上使用 Hugs 的练习。

Hugs 系统练习

1. 在 Hugs 系统加载`FirstLiterate.lhs`, 并计算下列表达式的值：

   ```
   square size
   square
   double (square 2)
   $$
   square (double 2)
   23 - double (3+1)
   23 - double 3+1
   $$ + 34
   13 `div` 5
   13 `mod` 5
   ```

   你能说出 $$ 的作用吗？
2. 使用 Hugs 命令给出上面表达式（除 $$ 外）的类型。
3. 下列表达式的结果是什么？试解释你得到的结果。

   ```
   double square 2 double
   ```

4. 编辑文件`FirstLiterate.lhs`，添加下列整数到整数的函数定义：
   - 将输入加倍，然后将此结果平方；
   - 将输入平方，然后将此结果加倍。

请在函数的定义中写明其类型。

§2.3 标准库 Prelude 和 Haskell 函数库

在第一章我们看到 Haskell 包含了一些内置类型，如整数，列表和这些类型上的函数，包括算术函数和列表上的函数`map`和`++`等。所有这些类型和函数的定义包含在 **标准引导库**`Prelude.hs` 中。当 Haskell 启动时，系统会自动装载标准引导库。在图 2.3 中可以看到，当运行 Hugs 时

```
Reading file: "C:\HUGS\lib\Prelude.hs"
```

在加载`FirstLiterate`之前进行(译者: 在译本中提供的图片中没有显示"Reading ...")。

在过去的十多年，随着 Haskell 的发展，引导库也越来越大。为了限制 Prelude 的规模，并且让用户可以使用其中的许多名，许多定义已经移入 **标准函数库**，用户需要时可以调入。在讨论语言的有关部分时，我们会进一步讨论这些库。

除标准函数库外，Hugs 还包括支持并行和函数式动画等由用户开发的各种函数库。我们会在今后的章节中讨论这些库。其他 Haskell 系统已包含用户开发的函数库，但是所有的系统均支持标准库。

为了使用函数库，我们需要介绍 Haskell 的模块。这也是下节的内容。

§2.4 模块

一个典型的计算机软件将包含几千行程序。为了便于管理，我们将软件分解成一些较小的组成部分，称之为模块。

一个 **模块** 是 Haskell 定义的一个命名集合。例如，一个名为`Ant`的模块以下列形式开始：

```
module Ant where
    ...
```

一个模块还可以 **输入** 其他模块的定义。例如，模块`Bee`使用`import`语句输入模块`Ant`的定义：

```
module Bee where
import Ant
    ...
```

其中 import 语句表示我们在模块 Bee 中可以使用 Ant 中的所有定义。在处理模块时，我们将遵守下列习惯：

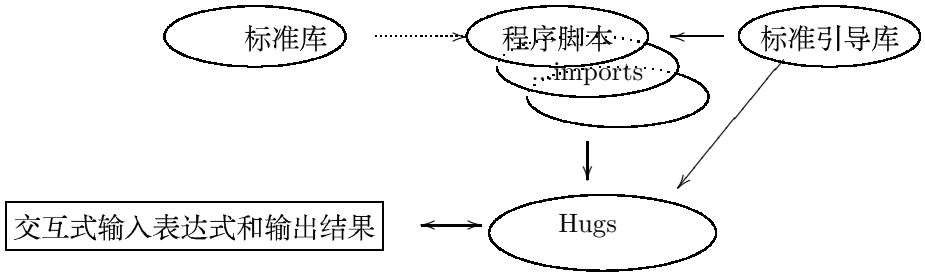
- 每个文件只包含一个模块；
- 文件 Blah.hs 或者 Blah.lhs 包含模块 Blah。

模块机制支持 2.1 节讨论的函数库，但是我们也可以用模块机制包含我们自己或者他人的代码。

模块机制允许我们控制如何输入定义，哪些定义是可见的或者被一个模块输出，从而可以供其他模块使用。我们将在第十五章对此做进一步的讨论，并且探讨如何最好地使用模块以支持软件系统的设计。

现在我们解释为什么 Hugs 系统显示提示符 Main¿。提示符表示目前调入 Hugs 系统的顶层模块名，如果模块没有名，则系统使用缺省名 “Main”（见第十五章）。

我们可以将使用 Hugs 的过程图示如下：



当前脚本可以访问标准引导库以及使用 import 输入的模块；这些模块包括与标准引导库位于同一个目录的标准库。用户与 Hugs 交互作用：用户输入表达式或者其他命令，系统返回计算结果。

下一节再次讨论第一章介绍的图形例子，以演示模块的实际应用。

§2.5 第二个例子：Pictures

贯穿第一章的例子是图形。图 2.4 显示实现图形脚本的一部分。我们省略了部分定义，用省略号代体之。这个模块称为 Pictures, 可以在 34 页提到的网页下载。用下列语句可将此模块输入另外一个模块：

```
import Pictures
```

模块 Pictures 唯一的新东西是下列函数：

```
printPicture :: Picture -> IO ()
```

它用于在屏幕上显示一个图形。类型 `IO`是 Haskell 输入/输出 (I/O) 机制的一部分。我们将在第十八章对此做深入探讨，目前读者只需知道，如果 horse 是先前图形例子的名，那么函数应用`printPicture horse`的效果是

```
.......##...
.....##..#..
...##.....#.
..#.......#.
..#...#...#.
..#...###.#.
.#....#..##.
.#....#.....
...#...#....
```

```
>   module Pictures where

>   type Picture = [[Char]]

The example used in Craft2e, and a completely white picture.

>   horse, white :: Picture
>   horse = ....
>   white = ....

Getting a picture onto the screen.

>   printPicture :: Picture -> IO ()

>   printPicture = ....

Reflection in vertical and horizontal mirrors.

>   flipV, flipH :: Picture -> Picture
>   flipV = map reverse
>   flipH = reverse

One picture above another. To maintain the rectangular property,
the pictures need to have the same width.

>   above :: Picture -> Picture -> Picture
>   above = (++)

One picture next to another. To maintain the rectangular property,
the pictures need to have the same height.

>   sideBySide :: Picture -> Picture -> Picture
>   sideBySide = zipWith (++)

Superimose one picture above another. Assume the pictures to be
the same size.

>   superimpose :: Picture -> Picture -> Picture
>   superimpose = ....

Inverting the colours in a picture.

>   invertColour :: Picture -> Picture
>   invertColour = ....
```

图 2.4: 模块 Pictures

```
....#..#....
.....#.#....
......##....
```

任何 Picture 都可以用类似的方式显示。

本节以使用模块 Picutres 的一系列练习结束。

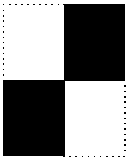
习题

2.1 定义一个模块UsePictures, 输入模块Pictures, 并定义blackHorse, rotate和 rotateHorse。注意, Pictures已经包括rotate, 所以在import 模块Pictures时需要隐藏rotate。

在以下的练习中, 你将在模块UsePictures中添加其他定义。

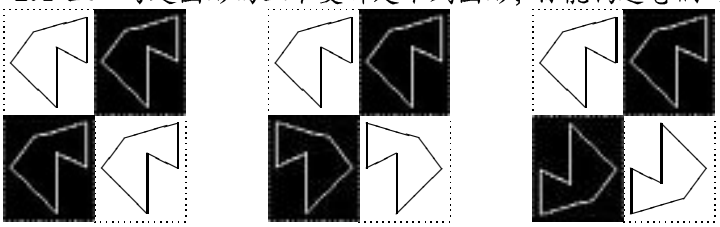
2.2 如何定义一个黑色矩形?假定可以使用第 *18* 页讨论的函数superimpose, 而不使用white, 又如何定义黑色矩形。

2.3 你能建立如下图形吗?

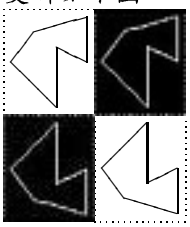


试用两种方法构造之。利用你的答案构造一个 8×8 的国际象棋棋盘。

2.4 上一习题图形的三个变种是下列图形, 你能构造它们吗?



2.5 给出前一习题图形的另一个变种, 说明如何构造它? 注意, 其中的一个变种如下图



§2.6 错误与错误信息

没有一个系统能保证用户的输入总是有意义的, Hugs 也不例外。如果在用户输入的表达式中或者脚本中有错误, 用户会得到一个错误信息。在 Hugs 中试输入

```
2+(3+4
```

上述错误在语法中，就像一个英语句子不合语法规则一样，如"Fishcake our camel"。表达式中缺少一个匹配的右括号")"。错误信息说明'4' 之后的输入不是系统期望的：

```
ERROR: Syntax error in expression (unexpected end of input)
```

类似地，输入`2+(3+4))`将产生下列错误信息：

```
ERROR: Syntax error in input (unexpected ')')
```

再试着输入下列表达式

```
double square
```

系统将产生一个**类型**错误，因为 double 应该应用于一个整数，而不是函数 square。

```
ERROR: TYPE error in application
*** expression        : double square
*** term              : square
*** type              : Int -> Int
*** does not match    : Int
```

这个信息显示，系统期望的类型是 Int, 而实际输入的类型是 `Int -> Int`。在这里，double 期望的参数类型是 `Int`, 但是占据其参数位置的 square 的类型为 `Int -> Int`。

当你得到如上的错误信息时，你需要检查为什么其中的 **项 (term)** 与包含它的 **上下文 (context)** 不匹配。上例中的项是 square, 其类型为第三行 `Int -> Int`, 上下文为第二行的`(double square)`，它所要求的项的类型是最后一行的 `Int`。

类型错误并不能永远给出详细的错误信息。输入`4 double`或者`4 5`将得到如下的错误信息：

```
ERROR: ... is not an instance of class ...
```

我们将在今后的章节中讨论这些信息的细节，目前读者只需知道它们是类型错误即可。

最后一种错误是 **程序错误**。试输入

```
4 `div` (3*2-6)
```

因为 0 不可以做除数，所以我们将得到下列信息：

```
Program error: {primDivInt 4 0} (or divide by zero)
```

这表示系统检测到 4 被 0 除。有关 Hugs 生成的错误信息的更多细节参见附录 E。

小结

本章的主要目的是实用性，使读者熟悉 Haskell 的 Hugs 实现。我们介绍了如何书写简单的 Hugs 程序，将其调入系统，然后使用模块中的定义求表达式的值。

大型 Haskell 程序应该组织成模块，这些模块可以被用户输入到其他模块。模块支持 Haskell 的函数库体系。我们使用 Picture 一例演示了模块系统。

最后，我们概括地介绍了提交给 Hugs 系统的表达式或者脚本中可能包含的语法错误，类型错误和程序错误。

第一章和第二章为本书今后的章节使用 Haskell 和 Hugs 解释器探讨函数程序设计的许多特色奠定了理论和实践基础。

第三章　基本类型和定义

我们已经介绍了函数程序设计的基本知识以及如何编写、修改和运用 Haskell 程序。本章讲述 Haskell 最重要的 **基本类型** 以及如何定义 **分情况** 函数。最后以 Haskell 的 **语法** 内容结束本章。

Haskell 包含了许多数值类型。我们已经使用了 Int 类型，本章还要介绍 **浮点** 分数类型 Float。我们在编写程序时经常要根据某个特定的 **条件** 是否成立来选择一个值。这些条件包括测试一个数是否大于另一个数，两个数是否相等，等等。测试的结果 True（如果条件成立）和 False（若条件不成立）称为 **布尔** 值。以十九世纪逻辑学家 George Boole 命名的这两个值构成 Haskell 的类型 Bool。本章将介绍布尔值，以及在函数定义中如何使用它们通过 **守卫** 选择函数值。

最后介绍包含字母，数字和空格等的 Haskell 字符类型 Char。

本章提供了基本类型的参考资料。读者可以跳过有关 Float 和 Char 的内容，待后面需要时返回来参考。

本章每一节包括一些函数的例子和建立在这些例子上的习题。

下一章将在本章的基础上，讨论各种不同程序设计方法。

§3.1　布尔类型 Bool

布尔值 True 和 False 表示测试结果，比如，比较两个数是否相等，或者第一个数是否大于第二个数。Haskell 的布尔类型记作 Bool。Haskell 提供了下列布尔运算：

```
&&    and
||    or
not   not
```

因为 Bool 只有两个值，布尔运算可以用 **真值表** 表示。真值表表示应用布尔运算与参数的所有可能组合的结果。例如，在下面的真值表中，第三行表示 `False&&True` 的值是 False, `False || True`的值是 True.

t1	t2	t1 && t2	t1 || t2
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

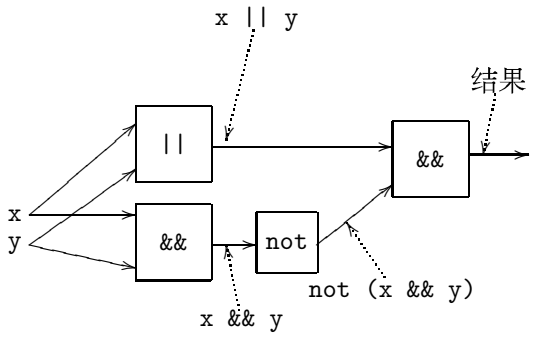
t1	not t1
T	F
F	T

布尔值可以是函数的参数，也可以是函数的结果。下面是一些例子。” 异或” 是这样的函数，只有当它的两个参数中仅有一个为 True 时，其结果才为 True。这就像一个饭馆的菜单中的” 或”：你可以选择素食茄盒或者鱼作为主菜，但不能同时选择两者! Haskell 的 “内置或” ||是” 包含的”，因为当其中

一个参数为 True 或者两个参数均为 True 时，其结果均为 True。

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

像第一章一样，我们用方盒表示函数，线段表示值。进入方盒的线段表示参数，离开方盒的是结果。



布尔值可以使用运算== 和 /= 比较相等和不相等, 他们均有如下类型。

```
Bool-> Bool -> Bool
```

注意运算 /= 和 exOr 是两个相等的函数，因为当两个参数中恰有一个为 True 时，两个函数均返回 True。

原子和定义

True 和 False 及数 2 等这样的表达式称为 **原子**。原子的值是直接给出的，也就是说，原子的值便是原子本身。我们可以利用原子 True 和 False 定义 not:

```
myNot :: Bool -> Bool
myNot True  = False
myNot False = True
```

在函数定义等式的左边也可以混合使用原子和变量，如

```
exOr True  x = not x
exOr False x = x
```

上述定义用了两个等式：当 exOr 的第一个参数是 True 时使用第一个等式，当它的第一个数为 False 时，使用第二个等式。等式左边使用 True 和 False 的定义比左边只使用变量的定义更容易理解。这是 Haskell 的 **模式匹配** 的简单例子，模式匹配将在 7.1 节介绍。

习题

3.1 按如下方法给出”异或”的另一个定义：”只有当 x 为 True, y 为 False, 或者x为 False, y为 True 时, x与y的异或为 True。”

3.2 用方盒与线段图描述上题的解。

3.3 利用等式左边为原子的定义方式，可以把一个函数的真值表转换为它的 Haskell 定义。使用这样的方法完成下列定义：

```
exOr True True = ···
exOr True False = ···
```

3.4 给出具有下列类型的函数 nAnd 的两个不同定义：

```
nAnd :: Bool -> Bool -> Bool
```

除了两个参数均为 True 的情况外，函数均返回 True。用一个框图描述你的定义。

3.5 利用你给出的 nAnd 的定义，给出下面式子的逐行求值过程

```
nAnd True True
nAnd True False
```

§3.2 整数类型 Int

Haskell 类型包含用于计数的整数，记作 0, 45, -3452, 2147483647 等。类型 Int 表示一个固定区间的整数，因此只包含了有限个整数。maxBound 表示此类型的最大值，在这里是 2147483647。Int 类型适用于大部分有关整数的运算，如果需要更大的整数，可以使用 Integer 类型。后者可表示任意大的整数[1]。下列运算和函数可应用于整数的算术，同样也适用于 Integer 类型：

`+`	两个整数之和
`*`	两个整数之积
`^`	幂, 如 `2^3` 的结果为`8`
`-`	作为中缀运算符时, 表示两个整数之差, 如`a - b`; 作为前缀运算符时, 表示一个整数之相反数
`div`	整数除法, 如`div 14 3`的结果为`4`; 也可记作`14 ‘div‘ 3`
`mod`	整数除法的余数, 例如, `mod 14 3`(或者`14 ‘mod‘ 3`) 为`2`
`abs`	一个整数的绝对值
`negate`	求一个整数的相反数

注意，用 **反引号** 括起来的 ‘`mod`‘ 应写在他的两个参数中间，它是函数 mod 的 **中缀** 形式。任何函数都可以用这种方式变为中缀形式。

注 1 负原子 负原子求反时常会出现错误。例如，负十二记作`-12`，其中的前缀 ’`-`’ 常与减法运算混淆，因此导致难以理解的类型错误信息。例如，

```
negate -34
```

被解释为”对 *-34* 取反“，结果导致 Hugs 错误信息。

```
ERROR: a -> a is not an instance of class "Num"
```

[1]本章介绍的许多 Haskell 标准函数使用 Int 类型，所以我们选择了介绍 Int 类型

如果你不明白错误的原因，而你处理的表达方式中包含负数，那么你应该将负数放在括号内，在这里应该是 negate(-34)。详细情况参见 *3.7* 节。

关系运算

整数上具有顺序关系和相等 (不相等) 关系。实际上，所有的基本类型上都有这些关系。整数上的这些关系是输入为两个整数，输出是布尔型，即值为 True 或 False 的函数。这些关系是

¿	大于
¿=	大于或者等于
==	等于
/=	不等于
¡=	小于等于
¡	小于

使用这些函数的一个简单例子是检查三个整数是否相等的函数：

```
threeEqual :: Int -> Int -> Int -> Bool
threeEqual m n p = (m==n)&&(n==p)
```

习题

3.6 解释下面定义的函数的作用：

```
mystery :: Int -> Int -> Int -> Bool

mystery m n p = not(m==n)&&(n==p)
```

提示：如果直接回答此问题有困难，那么先给函数一些具体输入以检查函数的功能。

3.7 定义下列函数

```
threeDifferent :: Int -> Int -> Int -> Bool
```

只有当三个数 m, n 和 p 全不相同时threeDifferent m n p才为True。试问 threeDifferent 3 4 3 的结果是什么? 为什么?

3.8 下列函数返回 True, 仅当它的四个参数完全相同时:

```
fourEqual ::Int->Int->Int->Int->Bool
```

试利用前面定义threeEqual的方式定义此函数。再一次绘出fourEqual的定义, 并在定义中使用threeEqual, 并比较你的两个定义。

3.9 逐步求下列表达式的值:

```
threeEqual (2+3) 5 (11 `div` 2)
mystery (2+4) 5 (11 `div` 2)
threeDifferent (2+4) 5 (11 `div` 2)
fourEqual (2+3) 5 (11 `div` 2) (21 `mod` 11)
```

§3.3 重载

虽然整数和布尔值具有不同的类型, 但是整数与布尔值均可以使用相同的符号 == 做相等比较运算。事实上, 在任何可以定义相等比较运算的类型上, 我们都将使用 == 来表示之。这意味着 (==) 具有下列类型

```
Int -> Int -> Bool
Bool -> Bool -> Bool
```

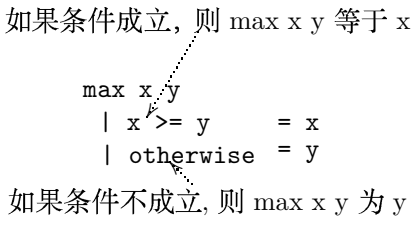
以及 `t -> t -> Bool`, 只要类型 `t`上有相等运算。这种使用同一个符号或名表示不同运算的机制称为 **重载**。Haskell 重载了许多符号。在第十二章我们将介绍 Haskell 类型系统如何处理重载，以及用户如何定义自己的重载运算符或名。

§3.4 守卫

本节讨论在函数定义中如何使用条件或者守卫选择函数的值。一个 **守卫** 是在函数定义中表示不同情况的布尔表达式。本节将以比较整数大小的函数和求两个整数中最大值的函数为例。当两个整数相等时，取他们的公共值为最大值。

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

如何理解 Haskell 标准库 Prelude 的这个函数呢?



一般地，如果第一个条件（此处为 x¿=y) 为 True，则相应的值便是函数的结果（此处为 x)。另一方面，如果第一个条件为 False, 那么顺序检查第二个条件, 第三个条件等。条件 otherwise 对任意参数均成立, 因此, 在上例的情况, 当 x¿=y 时，max 的值为 x，否则，即 y¿x 时，其值为 y。下面是定义中含有多个守卫的函数，它返回三个整数中的最大者：

```
maxThree :: Int -> Int -> Int -> Int
maxThree x y z
  | x>=y &&x >=z = x
  | y>=z         = y
  | otherwise    = z
```

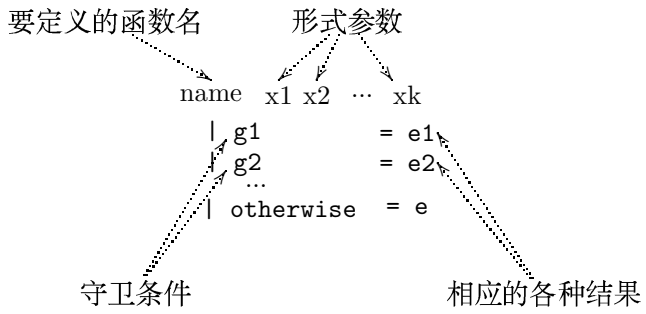


图 3.1: 使用守卫定义函数的一般格式

那么如何理解这个定义呢？第一个条件 `x>=y && x>=z` 测试 x 是否三个输入中最大者，如果条件为 True, 则相应的结果是 x。如果条件不成立，则 x 不是最大值，所以必须在 y 和 z 之间选择。所以第二个条件为`y>=z`。如果此条件成立，则结果为 y，否则结果为 z。在 4.1 节，我们将继续讨论 maxThree。

在第一章，我们介绍了简单函数的定义，现在给出带有条件的函数定义的一般模式，见图 3.1。注意，其中的条件 otherwise 并非必需的。

第一章还介绍了表达式的逐步求值。如何在这个计算模型中处理条件呢？当一个函数应用于它的参数时，我们必须确定哪一种情形适用，为此要顺序计算条件的值，直至找到一个其值为 True 的条件，此时，便可计算表达式的值。以 maxThree 为例，下面给出两个例子，其中条件的计算以 `??` 开头。

```
maxThree 4 3 2
??  4 >= 3 && 4 >= 2
?? ⇝ True && True
?? ⇝ True
⇝ 4
```

在此例中，第一个条件 `4>=3&&4>=2` 的值为 True, 故函数的结果为相应的值 `4` 。在第二个例子中，我们要计算更多的条件。

```
maxThree 6 (4+3) 5
??  6>=(4+3) && 6>=5
?? ⇝6>=7 && 6>=5
?? ⇝ False && True
?? ⇝ False
??  7>=5
?? ⇝ True
⇝ 7
```

在这个例子中，我们首先计算第一个条件，`6>=(4+3)&&6>=5`，其结果为 False；所以我们继续计算第二个条件`7>=5`, 其值为 True，所以结果为 7。第二

个参数4+3的值一旦计算过，再一次遇到它时，不需要重新计算它的值。这不仅仅是我们所懂得的技巧。Hugs 系统对（4+3）这样的参数，只计算一次它的值，并将其值保存起来以备后用，如上例的情形。这将是第十七章讨论的惰性计算的一个特点。

条件表达式

守卫是函数定义中区别不同情形的条件。我们还可以使用 Haskell 的if...then...else结构编写一般条件表达式。当一个条件表达式 if condition then m else n 的条件condition为True时，其值为m，当条件condition为False时，其值为n。所以，表达式if False then 3 else 4的值为 4。表达式if x>=y then x else y表示 x 与 y 的最大值。因此，函数max可以用下列方式定义

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

我们通过使用守卫的格式定义分情形函数，但有的时候使用if... then ...else显得更自然。下面会看到这样的例子。

注 2 重新定义引导库函数 Prelude.hs包含了max函数。如果一个脚本maxDef.hs包含如下定义：

```
max :: Int -> Int-> Int
```

那么此定义与 Prelude.hs 中的 max 函数定义发生冲突，Hugs返回下列信息

```
ERROR - maxDef.hs(line3) : Definition of variables "max" clashes
with import
```

若要重新定义 Prelude 中的函数 max 和 min，在脚本中加上下列语句

```
import Prelude hiding (max,min)
```

此语句应置于 module 语句之后。本书中许多函数是 Prelude 中定义的。若要重新定义这些函数，应该使用上述方法。

习题

3.10 计算下列表达式的值

```
max (3-2) (3*8)
maxThree (4+5) (2*6) (100`div`7)
```

3.11 给出下列函数的定义

```
min      :: Int -> Int -> Int
minThree :: Int -> Int -> Int -> Int
```

这两个函数分别计算两个数及三个数中的最大值。

§3.5 字符型 Char

人与计算机通过键盘输入与屏幕输出进行交流。输入和输出是 **字符** 序列，包括字母，数字和 “特殊” 字符，如空格，制表符 (tab), 换行和文件尾等特殊符号。Haskell 包括一个内置的字符类型 Char。单字符要用单引号括起来, 如 `'d'` 在 Haskell 中表示字符 d. 又如`'3'`表示字符 3。一些特殊字符的表示如下表:

制表符 (tab)	`'\t'`
换行符 (newline)	`'\n'`
反斜线 (backslash(\))	`'\'`
单引号 (single quote ('))	`'''`
双引号 (double quote("))	`'"'`

将字符用整数来表示的标准编码称为 ASCII 编码。在这个编码下, 大写字母'A' 到'Z' 的编码为 65 到 90, 小写字母'a' 到'z' 的编码为 97 到 122。例如, 编码为 34 的字符可以用 `'\34'` 来表示。因此'a' 和 `'\57'` 具有相同的意义。ASCII 最近被扩展成 Unicode 标准, 其中包含了英语之外的其他语言的字符。

字符与其数值编码之间可以使用下列函数相互转换。

```
ord:: Char -> Int
chr:: Int -> Char
```

这些编码函数可用于定义 Char 上的函数。把小写字母转换成大写字母, 需要加一个偏移量:

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

toUpper :: Char -> Char
toUpper ch = chr (ord ch + offset)
```

注意, 我们给偏移量一个名`offset`, 而设有把它直接写在函数`toUpper`中, 如

```
toUpper ch = chr (ord ch+ (ord'A'-ord'a'))
```

这是一种标准的编程方式, 它使得程序更易于阅读和修改。若要修改偏移量的值, 我们只须修改`offset`的定义, 而不需修改使用`offset`的函数。字符之间可以通过它们的编码比较先后顺序。因此, 我们可以检查一个字符是否数字:

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

这是因为数字0 到9 的编码是连续的 48 到 57。标准引导库包含了如`toUpper`和`isDigit`等函数, 详情参见 Prelude.hs。Char 上的其他函数在库 Char.hs 中定义。

注 3 字符与名 *a 和'a' 容易混淆。其区别是,a 是一个名或一个变量, 其类型可能是任意的, 而'a' 是一个字符, 其类型为 Char。类似地, 注意数 0 和字符'0' 的区别。*

习题

3.12 定义一个函数, 它将小写字母转换为大写字母, 而非小写字母保持不变.

3.13 定义如下函数

```
CharToNum :: Char ->Int
```

它把像'8' 这样的数字转换为其值 8, 而把非数字符转变为 0.

§3.6 浮点数 Float

在 3.2 节我们引入了 Haskell 整数类型 Int。在数值计算中, 我们需要使用分数。在 Haskell 中使用类型 Float 表示 **浮点数**。本书很少使用 Float, 为此, 读者可以跳过本节, 待有需要时返回来参考。

Haskell 系统用一个固定的空间存储每一个 Float。其结果是, 并非所有的分数都能用浮点数表示, 分数上的算术运算结果并非永远是准确的。为了有更高的精确度, 可以使用双精度浮点数类型, 或者建立在 Integer 上的全精度分数类型 Rational。因为本书是编程简介, 所以我们将限于类型 Int 和 Float。在第十二章, 我们将对数值类型做简单总结。

在 Haskell 中浮点数可以用小数表示, 如 0.31426

-23.12

567.347

4523.0

这些数称为浮点数，因为小数点的位置并非对所有 Float 都是一样的，根据一个特定数的情况，更多的空间可用于整数部分或者小数部分。

Haskell 也允许用 **科学计法** 表示浮点数。其表示形式如下，其中表中右边表示他们的值。

231.61e7	$231.61 * 10^7 = 2,316,100,000$
231.6e-2	$231.61 * 10^{-2} = 2.3161$
-3.412e03	$-3.412 * 10^3 = -3412$

对于很大的数或很小的数，这种表示法比小数更方便。例如，数字 2.1^{444} 的表示，小数点前有超过 100 位数字，这对于有限的小数表示（通常最多为 20 位）是不可能的。使用科学表示法，可计作 1.162433e+143。

Haskell 在 Prelude 中定义了 Float 上的一些运算和函数。表 3.2 给出他们的名、类型及功能的简单描述，其中包括

- 标准的数学运算：平方根，幂，对数和三角函数；
- 将整数转换为浮点数的函数：`fromInt`，以及将浮点数转换为整数的函数`ceiling`,`floor`和`round`。

Haskell 可用作计算器。试着在 Hugs 的提示符下键入表达式 `sin(pi/4)*sqrt 2`。

重载的数字和函数

在 Haskell 中，数字 4 和 2 既属于类型 Int, 又属于类型 Float，如 3.3 节

`+ - *`	`Float -> Float -> Float`	加, 减, 乘
`/`	`Float -> Float -> Float`	分数除法
`^`	`Float -> Int -> Float`	幂 `x^n` = x^n, `n` 为自然数
`**`	`Float -> Float -> Float`	幂 `x**y` = x^y
`==, /=,<,>, <=, >=`	`Float -> Float -> Bool`	相等和关系运算
`abs`	`Float -> Float`	绝对值
`acos, asin, atan`	`Float -> Float`	`cosine,sine,tangent` 之逆
`ceiling floor round`	`Float -> Int`	取上、取下和最接近的整数
`cos, sin tan`	`Float -> Float`	三角函数
`exp`	`Float -> Float`	e 的幂
`fromInt`	`Int -> Float`	将 Int 转换为 Float
`log`	`Float -> Float`	以 e 为底的对数
`logBase`	`Float -> Float -> Float`	任意底 (第一个参数) 的对数
`negate`	`Float -> Float`	求相反数
`pi`	`Float`	常数 pi
`signum`	`Float -> Float`	根据输入为正数、0 或者负数 结果分别为 1.0, 0.0 或者 -1.0
`sqrt`	`Float -> Float`	(正数) 平方根

图 3.2: 浮点数运算和函数

所述，它们是重载的。一些数值函数也是重载的。例如，加法同时具有下列两个类型:

```
Int -> Int -> Int
Float -> Float -> Float
```

又如，关系运算 == 等在所有基本类型上有定义。在第十二章讨论类时，我们将详细讨论重载的思想。

注 4 把整数转换为浮点数 虽然数字是重载的，但是并不存在由Int到Float的自动转换。一般地，如果把一个整数（如floor 5.6）加到一个浮点数（如6.7）上，直接计算表达式(floor 5.6)+ 6.7，系统会返回错误信息，因为我们在把两个不同类型的量相加。要完成以上的加法，我们必须把Int 转换为Float，即fromInt(floor 5.6)+6.7 这里fromInt将一个Int型量转换为相应的Float型。

习题

3.14 定义一个求三个整数平均值的函数：

```
averageThree :: Int -> Int -> Int -> Float
```

使用上述函数，定义函数

```
howManyAboveAverage:: Int -> Int -> Int -> Int
```

其功能是返回输入中大于其平均值的个数。

后面的问题与一元二次方程有关

$$a * X^2 + b * X + C = 0.0$$

其中 a,b 和 c 是实数。

- 当 $b^2 > 4.0 * a * c$ 时，方程有两个实根；
- 当 $b^2 == 4.0 * a * c$ 时，方程有一个实根；
- 当 $b^2 < 4.0 * a * c$ 时，方程无实根。

以上假定 a 是非零实数，方程称为 **非退化的**。对于退化的方程，有下列三种情形：

- 如果b/=0.0 则方程有一个实根；
- 如果b==0.0且c/=0.0, 则方程无实根；
- 如果b ==0.0且c ==0.0, 则每个实数均为方程的根。

3.15 定义一个函数：

```
numberNDroots :: Float-> Float-> Float-> Int
```

对于给定的系数 *a,b* 和 *c*, 函数返回方程根的个数，假定方程是非退化的。

3.16 使用上题定义的函数，定义函数

```
numberRoots :: Float-> Float-> Float-> Int
```

对于给定方程的系数 *a,b* 和 *c*，函数返回方程根的个数。当所有实数均为根时，函数返回结果 *3*。

3.17 求二次方程根的公式为

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

定义下列函数

```
smallerRoot, lagerRoot:: Float-> Float-> Float -> Float
```

计算方程的较小根和较大根。当方程无实根或所有实数均为根时，两个函数均返回 *0*。

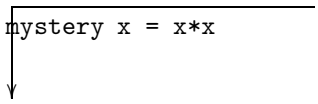
§3.7 语法

一个语言的语法描述所有合法的程序。本节介绍 Haskell 的语法，特别是初看上去不易理解的部分。

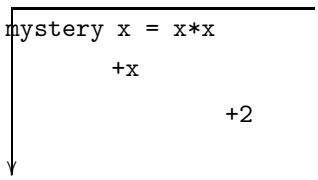
定义与书写格式

一个脚本包含了一个接一个的定义。如何分辨一个定义的结束，另一个定义的开始呢？英语用句点 '.' 表示一个句子的结束。

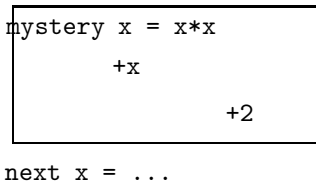
一个定义是由和此定义开始的位置处于同一列或其左边的字符结束的。当我们书写一个定义时，它的第一个字符标示着如下的一个方框：



所有写在方框内的部分均属于此定义。



当方框左边框遇到一个字符，或左边框左边有字符出现时，方框闭合。



因此，在书写一系列定义时，所有定义开始位置应该处于同一列。我们在书写脚本时将按照顶层定义从页面的左边开始，在文学型脚本中每个定义

在一个制表符后开始。这种版面格式规则称为 **边界规则**, 因为这使我们联想到足球中的边界。此规则也适用于有多条语句构成的条件等式，如 max 和 maxThree.

实际上，象英语中用'.' 显式表示一个句子的结束一样，Haskell 的显式结束符为';'。如果我们要把多个定义写在同一行，可以使用';'。例如

```
answer=42; facSix=720
```

注 5 格式错误 即使不使用';', 也有可能得到有关';' 的错误信息。如果我们违反格式规则，如

```
funny x=x+
1
```

则会得到如下错误信息：

```
ERROR...:Syntax error in expression (unexpected '.')
```

这是因为系统内部会在 *1* 之前插入';' 以示定义的结束，但这并不是我们期望的结束位置。

推荐的格式

边界规则允许各种不同的书写格式。本书中有数个子句构成的条件等式定义将选用下列格式：

```
fun v1 v2....vn
  | g1        = e1
  | g2        = e2
   ...
  | otherwise = er (or | gr = er)
```

在这个格式中，每个子句另起一行，所有守卫及结果上下对齐。同时注意，我们总是写明所定义函数的类型。

如果一个表达式 ei 或者守卫 gi 很长, 那么它仍可以分成多行, 如下式

```
fun v1 v2 ... vn
     一个很长的守
  |  卫, 可能占用多
     行
                 一个很长的表达
           =     式, 可能占用多
                 行
  | g2       = e2
  ...
```

Haskell 的命名

迄今为止，我们已经在定义和表达式中使用了各种命名。在下列定义中

```
addTwo :: Int -> Int -> Int
```

```
addTwo first second = first+second
```

其中的名或 **标识符** `Int`，`addTwo`，`first` 和`second`用于命名类型，函数和两个变量。Haskell 的标识符必须由大写或小写字母开始，后面可以加上字母，数字，下划线和单引号构成的序列。

在定义中用于命名值、变量和类型变量的标识符必须以小写字母开始。另一方面，类型名，如 Int;**构造符**, 如 `True` 和 `False`；模块名及今后要介绍的类名均要以大写字母开始。

如果试图以大写字母开始命名一个函数，如

```
Fun x =x+1
```

则系统输出错误信息`Undefined constructor function "Fun"`.

标识符的选择还有其他的限制。**保留字** 集不可用于做标识符，这些保留字包括

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

特殊标识符`as`,`qualified`和`hiding`在特殊的上下文中有特殊的含义，但可用作普通标识符。

按照常规，当一个标识符由多个词构成时，第二个及以后的各个词的第一个字母用大写字母，如`maxThree`。

同一个标识符可用于命名一个函数和一个变量，或者一个类型和一个类型构造符，但我们强烈建议不使用这种命名方式，因为这样只能带来混乱。

如果要 **重新命名** 一个在引导库或者其他库中的函数，我们必须在输入这些库时 **隐藏** 这些函数。详情参见第 49 页。

Haskell 是构建在 Unicode 标准字符集上的。Unicode 标准字符集包括了 ASCII 标准字符集以外的字符。这些字符可用于标识符中。Unicode 字符用 16 位二进行制序列表示, 可以用 `\uhhhh` 的形式输入, 其中每个`h`是一个 8 进制数 (4 位)。本书将只使用 Unicode 的子集 ASCII[2]。

运算

Haskell 包括了各种运算, 如 +,++ 等。运算是 **中缀** 函数, 即运算写在其参数中间, 而不是像普通的函数那样写在其参数前面。

原则上，所有运算的应用可以通过括号来表示，如`(((4+8)*3)+2)`。但这样的表达式很快会变得难于理解。下面的两个性质允许我们去掉某些括号。

结合性

如果将三个数 4, 8 和 99 相加，则可用 4+ (8+99) 或 (4+8) + 99 表示，其结果是一样的，这个性质称为加法的 **结合性**。因此，我们可以用 4+8+99

[2]本书作者在撰写这本书时, Hugs 尚不支持 Unicode

表示这个和。但是，并非所有的运算有结合性。例如，对于表达式4-2-1有两种不同的加入括号方法：

(4-2)-1=2-1= 1　　左结合
4-(2-1)=4-1=3　　右结合

在 Haskell 中每个非结合的运算被划分为左结合或者右结合的。如果一个运算是左结合的，则当此运算出现两次时，括号加到左面；如果是右结合的，则扩号加到右面。虽然选择是任意的，但是要尽可能符合习惯。特别地，'-' 是左结合的。

结合力

运算的结合性使我们能够解决像 2^3^3 这种同一运算出现两次的表达式的运算次序。但是，不同的运算符出现时如何处理运算的次序呢？如

```
2+3*4
```

```
3^4*3
```

在这里我们需要比较不同运算的 **结合力** 或 **凝聚力**。* 的结合力是 7，+ 的结合力是 6，所以在表达式 2+3*4 中 3 和 4 先做 * 运算，即

```
2+3*4=2+(3*4)
```

类似的，^ 地结合力是 8，较 * 的结合力强，所以

```
3^4*2=(3^4)*2
```

在附录 C 中列出了 Haskell 预定义的运算的结合性和结合力。在下面的“自定义运算”一节，我们讨论如何在脚本中定义运算，如何通过说明设置或改变运算的结合性和结合力。

注 6 函数应用 *结合力最强的是函数的应用，表示为函数名置于其参数的前面。如*`f v1 v2....vn`*。因此，*`f n+1`*的解释为*`f n`*加* `1`*，即*`(f n)+1`*，而非* `f(n+1)`*。在有疑问时，可以将每个参数括起来。类似地，由于'-'既是一个中缀运算，也是一个前缀运算。所以有时会混淆。*`f -12`*被解释为f减* *12*，而不是f作用于* *-12*。消除这种混淆的办法是将参数括起来。*

运算与函数

中缀运算符也可放在其参数的前面，方法是将运算符放在圆括号中。例如：

```
(+)::Int-> Int-> Int
```

所以

```
(+)2 3=2+3
```

今后介绍函数作为函数的参数时，上述转换是必需的。我们还可以把函数转换成运算，方法是反引号将函数名括起来，如`name`。如前面定义的最大值函数

```
2 `max` 3 = max 2 3
```

这种记法便于理解包含 **二元** 函数或有两个参数的函数的表达式。运算的结合力和结合性是可以控制的，参见附录 C。

自定义运算

Haskell 允许用户定义中缀运算，其定义方式与函数的定义完全相同。运算符名由下列 ASCII 运算符和 Unicode 符构成：

```
! # $ % & * + . / < = > ? \ ^ | : - ~
```

运算符不可以用冒号开始。

例如，定义最小函数为中缀运算符`&&&`：

```
(&&&) :: Int-> Int->Int
x &&& y
   | x>y       = y
   | otherwise = x
```

运算的结合性和结合力可由用户设置，详见附录 C。

习题

3.18 使用推荐的格式重新书写前面习题的答案。

3.19 给定下列定义

```
funny x = x+x
peculiar y = y
```

当你去掉`peculiar`前的空格时，什么现象会发生？请解释之。

小结

本章介绍了基本类型`Int`、`Float`和`Bool`以及其上预定义的函数。通过求两个整数的最大值的例子，我们介绍了如何使用布尔表达式 – 称为守卫 – 来定义分情形函数。求最大值的例子包括两种情形，一种是第一个参数大于第二个参数，另一种是第二个参数更大的情况。最后，我们介绍了 Haskell 程序的格式的重要性 – 一个定义由与此定义处于同一边界的程序片断隐式地结束。我们还概括介绍了 Haskell 的运算。

以上这些内容为我们解决编程问题提供了一个工具箱。下一章我们探讨使用这个工具箱解决实际问题的各种方式。

第四章　设计与编写程序

本章将从前面的 Haskell 细节转向讨论如何构造程序，我们介绍一些程序设计的一般方法，即在开始书写程序细节前如何计划程序的编写。这里将给出的建议很大程度上是独立于 Haskell 的，将适用于任何程序设计语言。

讨论将由递归入手。我们先重点解释递归的工作原理，然后解释**如何**找出原始递归定义，最后介绍更一般的递归形式。

当我们完成一个定义时，应该提问自己，定义是否具有它应有的功能。本章最后将讨论程序测试原理及一些测试例子。

§4.1　从何处出发设计一个 Haskell 程序？

本书的一个主题是如何 **设计** 用 Haskell 书写的程序。设计一词在计算科学中由许多含义。在这里它的含义如下：

定义 1 *设计是指书写详细的Haskell代码之前的阶段。*

本节将集中讨论一些例子和定义函数的不同方法，但同时我们也将给出如何开始编写程序的一般性建议。所有这些都是我们在解决编程问题遇到困难时可以提出的问题。

我理解我需要做什么吗？

在开始解决一个编程问题之前，我们需要弄清楚必须做什么。问题的描述通常是非正式的，这意味着或者问题的描述不完善，或者这样描述的问题没有办法解决。

假设问题是返回三个数的中间数。如果这三个数是 2，4 和 3，那么我们应该返回 3 作为结果。但是，如果这三个数是 2，4 和 2，那么由两种可能的答案:

- 我们可以说 2 是中间数，因为将三个数按序写成 2，2，4 后，2 便是处于中间位置的数。
- 我们也可以说，在这种情况下没有中间数，因为 2 是最小的数，而 4 是最大的数，因此不存在中间数。

从以上示例中我们能学到什么呢？

- 首先，即使是简单问题，我们在编程前也要仔细思考问题。
- 第二，应该意识到，在上面列出的两种选择中**没有正确的答案**，这需要由程序的用户与程序员一起决定哪一种结果是他们想要的。
- 第三，一种检验我们是否理解问题的好方法是考虑如何解决各种**实例**。
- 最后，值得注意的是，类似这样的问题经常发生在编程阶段；这样的问题发现得越早，越能节省我们的精力。

出现这种问题的另一例子是 3.4 节的 max 定义，我们必须回答当两个参数相同时函数返回什么样的结果。在那个例子中，返回 3 和 3 中的最大者仍为 3 是明知的答案。

在此阶段我能知道类型吧?

在此阶段，我们可以考虑各种对象的类型。我们可以写出返回三个数的中间数的函数名与类型，而无需知道如何实现这个函数：

```
middleNumber :: Int -> Int-> Int-> Int
```

无论如何，这是解决问题的一个进展，因为如果我们定义的函数`middleNumber`不具有上述类型，则函数一定不是我们期望的。

我已经知道哪些信息？如何使用这些信息？

这些是程序设计者需要考虑的关键问题，我们需要知道解决手中的问题有哪些资源是可用的：已经完成的定义中哪些是有用的？引导库和函数库提供了哪些有用的定义？自然地，随着课本的深入，我们对后者会了解得越来越多。但是，即使我们只是定义了很少的函数，我们也应该时刻考虑如何使用已有的函数解决手中的问题。例如，在定义 3.4 节的函数 maxThree 时，我们知道一个返回两个数中的最大者的函数 max 已经有定义。

在了解哪些资源可用的同时，还应知道如何使用它们。有两种使用已有函数的方法。

我们可以把一个函数的定义看作我们欲解决的问题的模型

在定义 maxThree 时，我们定义过 max。我们可以把 max 的定义看作如何定义 maxThree 的模型。

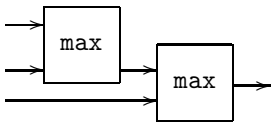
在 max 定义中，当 x 是两个数中的最大值时，即`x>=y`时，我们定义其结果为 x。maxThree 的定义是类似的，只是将两个值上的条件变为三个值上的条件，即

```
x>=y&&x>=z
```

这是定义 maxThree 时使用 max 的一种自然方法，但并非唯一的方法。

我们在新定义中可以使用已有的函数

我们要求三个数中的最大值，同时我们有一个求两个数中最大值的函数 max。如何使用 max 来定义 `maxThree`呢？我们可以先求前两个数的最大值，再求它和第三个数的最大值，如下图所示



在 Haskell 中可表示为

```
maxThree x y z = max (max x y) z
```

或者用中缀形式

```
maxThree x y z = (x `max` y) `max` z
```

这种使用 max 的方式使得 maxThree的定义更简短，更易于理解。如果在某个时刻我们改变了 max 的定义，或者使之成为预定义函数，那么maxThree便可使用 max 的新定义。这个优点在这个小例子中或许显得不突出，但是在大型的软件系统中，软件在其整个生命周期过程需要不断地修改或者扩充，上述优点显得尤其重要。

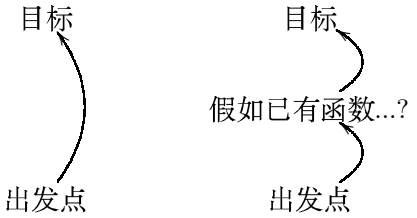
我能把问题分解成更简单的小问题吗？

如果我们不能直接解决一个问题，可以考虑将原问题分解成更小的问题。这种" 分而治之" 的原理是所有大规模程序设计的基础：分别解决这些小问题，然后将这些解组合起来成为原问题的解。

那么如何将一个问题分解成小问题呢？我们可以考虑解决一个比较简单的问题，然后在此基础上构造原问题的完整答案。我们也可以考虑下列问题。

假如我得到希望的函数，如何用这些函数构造问题的解呢？

这里的" 假如" 是问题的中心，因为它把问题分解为两部分。首先，假定我们可以得到要求的辅助函数而无需考虑这些辅助函数如何实现，我们必须给出问题的解。然后我们必须分别定义这些辅助函数。



在这里，一个由起点到终点的单级跳跃由两个更短、更容易的跳跃来代替。这种方式称为 **自上而下** 的，因为我们从整个问题的顶部出发，然后将问题分解为小问题。

上述过程可以重复进行，这样一个问题便可以由一系列的小跳跃来解决。下面是一个例子。本节习题中会有更多的例子。假设需要按照第 59 页中的第一选择定义下列函数

```
middleNumber :: Int -> Int -> Int -> Int


middleNumber x y z
   | x 为解的条件   = x
   | y 为解的条件   = y
   ... ...
```

现在的问题是如何定义这些条件，不过，我们假定已有一个函数检测这些条件，称之为 between, 它带有三个参数，返回布尔型值：

```
between :: Int -> Int -> Int -> Bool
```

between m n p返回True, 当且仅当n介于m和p之间。现在我们可以完成middleNumber的定义：

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

函数 between 的定义留给读者作为练习。

本节介绍了一些可以帮助我们开始解决问题的一般方法。显然，因为程序设计是一种创造性劳动，所以不存在一套能永远机械地找到问题的规则。另一方面，前面提出的问题能帮助我们起步，并提供了一些如何设计一个程序的不同策略，我们将在第十一章继续讨论这个题目。

习题

4.1 定义返回四个整数中最大者的函数：

```
maxFour :: Int-> Int-> Int-> Int-> Int
```

试给出函数的三个定义：第一个以maxThree为模型，第二个使用max, 第三个使用max和maxThree，然后用图示法表示第二和第三个定义，最后比较三个定义各自的优点。

4.2 给出本节讨论中函数between的定义

```
between :: Int -> Int ->Int ->Bool
```

你的定义应该与middleNumber的功能协调一致，仔细考虑一个数介于两个数之间的可能方式。或许下面的函数会有帮助：

```
weakAscendingOrder :: Int -> Int -> Int -> Bool
```

weakAscendingOrder m n p 为True当且仅当 m, n和 p是非递减序排序的，如2 ,2 ,3。

4.3 定义一个函数

```
howManyEqual :: Int-> Int-> Int-> Int
```

函数返回三个参数中相同参数的个数，如

```
howManyEqual 34 25 36 = 0
howManyEqual 34 25 34 = 2
howManyEqual 34 34 34 = 3
```

想一想你看到的函数中（如习题），哪些可以应用到你的定义中。

4.4 定义一个函数

```
howManyOfFourEqual::Int-> Int-> Int-> Int-> Int
```

其功能与howManyEqual类似。或许你需考虑”假如...”。

§4.2 递归

递归是一种重要的程序设计方法：一个递归函数或对象的定义中包含了此函数或对象本身。本节重点解释递归的思想和工作原理。特别地，对于自然数上阶乘的原始递归定义，我们使用两个互补解释来说明它的工作原理。下一节介绍如何在实践中应用递归。

第一个例子：关于阶乘

我们知道，一个自然数的阶乘是从 1 到这个数（包括此数）的乘积，如

```
fac 6 = 1*2*3*4*5*6
```

假如我们画一个阶乘表，其中 0 的阶乘为 0，则阶乘表如下

```
n  fac n
0  1
1  1
2  1*2 = 2
3  1*2*3 = 6
4  1*2*3*4=24
```

不过，要注意到，在画此阶乘表时，我们做了大量的重复计算。例如，在计算`1*2*3*4`时，我们重复计算了`1*2*3`:

$$\boxed{\texttt{1*2*3}}\texttt{*4}$$

这表明我们可以用另一种方式生成阶乘表：首先说明从何处开始，

```
fac 0 = 1                                    (fac.1)
```

因此，表的开始如下：

```
n  fac n
0  1
```

然后说明如何从表的一行构造下一行

```
fac n = fac (n-1) * n                        (fac. 2)
```

由此我们得到下表

```
n   fac  n
0    1
1    1*1 = 1
2    1*2 = 2
3    2*3 = 6
4    6*4 = 24
```

等等。

这个故事的寓意何在呢？我们从阶乘表的一种描述出发，但后来发现，我们需要的所有信息是`(fac.1)`和`(fac.2)`。

- `(fac.1)`告诉我们表的第一行是什么

- (fac.2)告诉我们如何由表的一行得到下一行。

阶乘表是阶乘函数的一种表示形式，因此，我们可以把(fac.1)和(fac.2)看作计算阶乘函数的描述，也就是

```
fac :: Int -> Int
fac n
  | n == 0    = 1
  | n > 0     = fac (n-1) * n
```

这种形式的定义称为递归的，因为我们在定义fac时使用了fac本身。这种描述听起来似乎像悖论：怎么能使用对象本身来描述一个对象呢？然而，前面的故事确定是有意义的，因为它们提供了

- 一个起始点：fac在 0 处的值；
- 一个计算方法：fac 从一个点的值，fac (n-1), 到下一点的值即fac n的一种计算方法。

这些递归规则将给出 fac在任意参数 n(n¿0) 上的值fac n, 如前面看到的，我们只需写出表的几个行。

递归与计算

上一节的故事描述了阶乘定义如何用于生成阶乘表：

```
fac :: Int -> Int
fac n
  | n == 0   = 1                          (fac.1)
  | n >0     = fac (n-1) * n              (fac.2)
```

从fac 0出发，然后计算fac 1,fac 2, 由此可以计算我们希望的任何n上的值。

我们也可以用计算的方式解读定义，并给出递归的另一种解释，以fac 4为例。

```
fac 4
⇝ fac 3 * 4
```

利用(fac.2)将求fac 4代替为更简单的目标求fac 3 (以及其结果与 4 相乘)。继续使用(fac.2)得到下列结果

```
fac 4
⇝ fac 3 * 4
⇝ (fac 2 * 3) * 4
⇝ ((fac 1 * 2) * 3) * 4
⇝ (((fac 0 * 1) * 2) * 3) * 4
```

现在我们到达了最简单的情况 (或 **递归基**)，它可以用(fac.1)来求得。

```
⇝ (((1 * 1) * 2) * 3) * 4
⇝ ((1 * 2) * 3) * 4
⇝ (2 * 3) * 4
⇝ 6 * 4
⇝ 24
```

在上述计算过程中，我们利用 **递归步**(fac.2) 将目标转化为递归基。我们又一次得到了希望的结果，因为递归步把复杂的情况转化为简单的情况，最终转化为最简单的情况 (此处为 0)，其值是已知的。

通过 fac一例，我们看到递归原理的两种解释：

- **自底向上** 的观点：fac 的两个等式可看作从 0 的递归基开始逐行生成 fac 的值；
- **自顶向下** 的观点：由欲计算的目标出发，利用递归步逐步化简到递归基。

这两种观点是相关的，自顶向下的解释也可以看作表的生成，只是在这种情况下的表是按需要生成的。由目标fac 4出发，我们需要生成 0 至 3 行。

上述递归形式称为 **原始递归**。我们将在下一节对此进行更多的讨论，包括如何寻找递归定义。在结束本节之前，我们讨论 fac函数的另一特点。

无定义或错误值

阶乘的定义包括 0 和正整数，将 fac应用于负整数会有什么结果呢？在 Hugs 中计算fac (-2)，我们会得到

```
Program error : {fac (-2)}
```

因为在负数上阶乘没有定义。可以将阶乘的定义扩展到负数上，并定义其值为 0，即

```
fac :: Int -> Int
fac  n
  | n == 0     = 1
  | n > 0      = fac(n-1)*n
  | otherwise  = 0
```

也可以在定义中加入我们自己的错误信息如下：

```
fac :: Int -> Int
fac  n
  | n = = 0   = 1
  | n > 0     = fac(n-1)*n
  |otherwise  = error "fac only defined on natural numbers"
```

当计算fac (-2)时，我们会得到下列信息

```
Program error: fac only defined on natural numbers.
```

错误信息是一个 Haskell 串。第五章将讨论串。

习题

4.5 定义函数rangeProduct, 对于任意自然数 m和 n函数返回下列乘积

```
m*(m+1)*···*(n-1)*n
```

定义应该包括函数的类型。当 n小于 m时，函数返回 *0*。提示：你不必使用递归。如果你愿意，也可以使用递归。

4.6 因为 fac是rangeProduct的一个特例，使用rangeProduct表示 fac。

§4.3 实践中的原始递归

本节通过一系列例子介绍如何在实践中使用递归。

原始递归的模式说明，我们可以用下列方式定义自然数 0，1，2，…上的函数；定义函数在 0 上的值，给出由函数在 n-1 处的值计算函数在 n 处值的方法。我们可以给出这种定义的 **模板**:

```
fun n
  | n == 0  = ···
  | n > 0   =  ··· fun ( n-1 ) ···
```

其中，两个等号的右边需要由我们提供。

如何判断一个函数是否可以用这种方法定义呢？与本章开始类似，我们提出一些问题，这些问题汇总了应用原始递归时的实质性质。

如果给出fac (n-1)的值，如何由它定义fac n的值呢？

我们将通过一些例子介绍原始递归如何应用于实践中。

例 4.1 假如需要定义一个函数计算 *2* 的自然数的幂次：

```
power2 :: Int -> Int
```

其中pow2 n为 2^n，即 2 的 n次乘积。模板函数为

```
power2  n
  | n == 0   = ...
  | n > 0    = ... power2 (n-1)...
```

对于 0 的情况，结果为 1。一般地，2^n 可表示为 2^{n-1} 与 2 的乘积，所以我们定义

```
power2  n
  | n ==0   = 1
  | n > 0   = 2*power2(n-1)
```

例 4.2 第二个例子考虑函数

```
sumFacs :: Int -> Int
```

其中

```
sumFacs n = fac 0 + fac 1 +···+ fac (n-1) + fac n
```

如果我们知道sumFacs 4 的结果是 34, 那么可用一步计算出sumFacs 5, 我们只须在前者的基础上加上fac 5, 即 120, 结果是 154, 这个事实对一般 n 都成立, 由此得到函数的下列定义:

```
sumFacs :: Int -> Int
sumFacs  n
   | n == 0    = 1
   | n > 0     = sumFacs(n-1)+fac n
```

事实上, 如果将前面的函数 fac换成类型为 Int -> Int的任意函数 f, 则前面的模式同样适用。因此, 我们可以定义

```
sumFun :: (Int -> Int ) -> Int -> Int
sumFun f n
   | n == 0    = f 0
   | n >0      = sumFun f (n-1) + f n
```

其中函数值被累加的函数本身是函数sumFun的参数。下面是应用sumFun的一个例子

```
sumFun fac 3
⇝sumFun fac 2 + fac 3
⇝sumFun fac 1 + fac 2 + fac 3
⇝sumFun fac 0 + fac 1 + fac 2 + fac 3
⇝fac 0 + fac 1 + fac 2 + fac 3
⇝ ...
⇝10
```

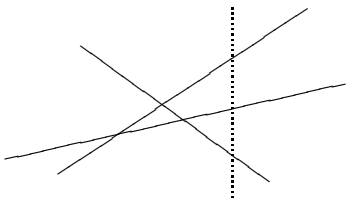
同时, 我们可以用sumFun定义sumFacs:

```
sumFacs n = sumFun fac n
```

第一章曾对函数作为数据做过简短介绍, 第九章将对此做详尽的讨论。第一章提到函数作为参数的威力, sumFun便是一个很好的例子: 对于任意类型为 Int -> Int的函数, 其参数在 0 与 n 之间的函数值求和可用同一定义完成。

例 4.3 最后一个例子是几何问题。

试求在一张纸上画若干线段可将纸分割的最大区域数。没有分割时的区域数为 1。一般情况如何呢? 假设纸上已有 n-1 条线分割, 现在再加上一个新的直线分割。



当新直线与原有的每条直线相交时, 我们得到最多的区域。又因为新直

线只能与每条直线相交一次，所以新直线恰好穿过 n 个区域，并将其一分为二。所以，加上第 n 条直线后可以新增加 n 个区域。由此可以填充原始递归函数模板，得到下列定义

```
regions :: Int -> Int
regions  n
  | n == 0 = 1
  | n > 0  = regions(n-1) + n
```

习题

4.7 使用自然数上的加法函数，给出自然数上乘法的递归定义

4.8 一个正整数 n 的平方根是其平方小于或等于 n 的最大整数。

例如，*15* 和 *16* 的平方根分别是 *3* 和 *4*。给出这个函数的原始递归定义。

4.9 给定一个类型为 Int -> Int的函数 f，定义一个类型为 Int -> Int的递归函数，对于输入 n，函数返回f 0，f 1，…，f n中的最大值。或许 *3.4* 节定义的函数 *max* 对你有帮助。

要测试这个函数，可在脚本中加入函数 f的一些值的定义：

```
f  0   = 0
f  1   = 44
f  2   = 17
f  _   = 0
```

等，然后测试函数在不同参数上的值。

4.10 给定一个类型为 Int -> Int的函数，定义一个类型为 Int -> Bool的函数，对于输入 n，当f 0，f 1，…，f n中有一个以上的值为 *0* 时，函数的值为 True，否则函数的值为 False。

4.11 你能使用sumFun而不使用递归给出regions的另一个定义吗？

4.12 (难度较大) 试求用一定数目平面切割一个固体所能得到的最大切割面（平面）数。本题答案不同于纸片的直线切割区域一例。

§4.4 递归的一般形式

如 4.2 节所述，函数 fac的递归定义利用fac (n-1)的值求得fac n的值。我们看到，fac(n-1)较fac n更接近于fac 0，因此更简单些，只要保持这种更简单的性质，我们可以使用不同的递归模式。这是本节讨论的内容。这种更一般形式的递归称为 **一般递归**。在使用递归定义一个函数时，我们需要回答下列问题：*在定义f n 时，哪些f k是有帮助的？*

例 4.4

Fibonacci 数列由 0 和 1 开始，其后的每个数均为最后两个数之和，因此其后的数为 0 + 1 = 1，1 + 1 = 2 等。我们可以给此序列一个递归定义

```
fib :: Int -> Int
fib n
  | n == 0   = 0
  | n == 1   = 1
  | n >1     = fib (n-2) + fib (n-1)
```

其中的一般情况`fib n`不仅依赖于`fib (n-1)`, 还依赖于`fib (n-2)`。

虽然此定义给出 Fibonacci 数列的一个清晰描述，但是它的计算效率是很低的。当计算`fib n`时，我们需要计算`fib (n-1)`和`fib (n-2)`，而在计算`fib (n-1)`时，我们需要再一次计算`fib (n-2)` 。我们将在 5.2 节讨论解决这个问题的方法。

例 4.5

用一个正整数除以另一个正整数有许多种实现方法，其中一种最简单的方法是重复地从被除数中减去除数。我们将给出这样的程序。事实上，我们将定义两个函数

```
remainder :: Int -> Int -> Int
divide    :: Int -> Int -> Int
```

它们分别给出除法的余数和商。

在寻求一个定义时，先看几个例子常常是有帮助的。假设要用 10 除 37，我们希望得到

```
remainder 37 10 = 7
divide    37 10 = 3
```

如果被除数 37 减去除数 10，函数的值有何关系呢？

```
remainder 27 10 = 7
divide    27 10 = 2
```

余数保持不变，新商是原商减 1。递归基的情况是什么呢？

```
remainder 7 10 = 7
divide    7 10 = 0
```

由这些例子作指导，我们可以给出下列定义

```
remainder m n
  | m < n     = m
  | otherwise = remainder (m-n) n


divide m n
  | m < n     = 0
  | otherwise = 1+divide (m-n) n
```

这些定义也表明了递归的另一重要特性：一个一般递归函数不一定永远给出结果，一个求值过程可能永远不会停止。例如，下列求值过程

```
remainder 7 0
⇝remainer (7-0) 0
⇝remainder 7 0
⇝ ....
```

这个计算过程将永远循环下去。事实上，用 0 作除数是有问题的。但是，如果用负数作除数，同样会发生问题，例如

```
divide 4 (-4)
⇝ divide (4-(-4)) (-4)
⇝ divide 8 (-4)
⇝ ....
```

由此得到的经验是用递归定义的函数不能保证永远 **终止计算**。如果使用原始递归，或者递归的情况总是把复杂的情况化为简单的情况，那么终止计算是可以保证的。上例的问题是，减去一个负数使结果远离递归基，造成函数应用于更复杂的参数。

习题

4.13 给出求两个正整数最大公因子函数的递归定义。

4.14 设要计算 *2* 的 n次幂。当 n为偶数时，如为2*m，则 $2^n = 2^{2*m} = (2^m)^2$。如果 n为奇数，如2*m +1，则 $2^n = 2^{2*m+1} = (2^m)^2 * 2$ 利用上面的事实，给出计算 2^n 的递归定义。

§4.5 程序测试

一个程序被 Haskell 系统接受，并不意味着程序具有它应有的功能。我们如何能确认一个程序按我们期望的方式运行呢？一种方法是第 1.10 节提到的，用某种方法证明程序会按照期望的方式运行。然而，证明需要付出昂贵的代价。通过测试程序在某些有选择的输入上的运行，可以在一定程度上确保程序能正确运行。因此，测试的艺术是选择尽可能详尽的输入，也就是说，我们希望测试数据代表了函数的所有不同" 种类" 的输入。

如何选择测试数据呢？有两种选择测试数据方式。

我们只知道函数的 **规格说明**，并且在此基础上选择测试数据。这种方法称为 **黑盒测试**，因为我们不能看到包含函数的盒子的内部。另一方面，在 **白盒测试** 中，我们可以根据函数的定义选择测试数据。下面我们通过测试求三个整数中最大值函数maxThree分别介绍这两种方法。

黑盒测试

如何合理地选择一个函数的测试数据，而不是简单地随机选择呢？

我们需要将输入分成不同的 **测试组**，其中函数在同一组的输入上有类似的运行方式。在选择测试数据时，我们希望确保在每个组中至少选取一个代

表。

我们还应该特别注意一些**特殊情况**，即各组的" 边界" 情况。例如，如果分组为正数和负数，则须特别注意 0 的情况。

maxThree 的测试组是什么呢？对此没有一个简单的正确答案。但是，我们可以考虑哪些因素可能与此问题相关，哪些因素不相关。对于 maxThree，整数的大小和符号是不相关的，决定其结果的是它们的相对顺序。我们可以做以下的首次划分：

- 所有三个数均不相同；
- 所有三个数均相同；
- 其中两个数相同，第三个不同，事实上这里有两种情况，
  - 两个相等的数为最大值；
  - 第三个不同的数为最大值。

为此我们可以选择下列的测试数据：

```
6 4 1
6 6 6
2 6 6
2 2 6
```

如果用这些数据测试 3.4 节的 maxThree定义，我们看到程序给出正确的答案。对上述数据，下列定义同样给出正确答案：

```
mysteryMax :: Int -> Int -> Int ->Int
mysteryMax  x y z
  | x>y && x>z  = x
  | y>x && y>z  = y
  | otherwise   = z
```

那么我们能得出mysteryMax计算三个输入的最大值吗？如果我们做出这样的结论，那就错了。这是因为

```
mysteryMax 6 6 2 ⇝ 2
```

这是一个很重要的例子：它告诉我们，仅仅靠测试不能确保一个函数是正确的。如何设计测试数据使得上述错误暴露出来呢？应该说，我们不仅要考虑以上的分组，而且应该考虑数据的所有可能顺序，即

- 三个值均不相同：6 个不同的顺序；
- 三个值均相同：1 个顺序；
- 二个值相同，第三个与其不等。这里有两种情况，对每种情况，需要考虑 3 种顺序。

最后一种情况生成测试数据 6 6 2 , 从而发现上述错误。

前面提到过特殊情况：在这里的特殊情况为当两个相同的参数为最大值时。显然，mysteryMax的作者考虑的是三个参数均不相等的一般情况，所以，这个例子说明了测试特殊情况的重要性。

白盒测试

在设计白盒测试数据时，我们将遵循适用于黑盒测试的原理，但同时可借助程序帮助我们选择测试数据。

- 如果一个函数包括守卫，那么对定义中的每一种情况都应提供测试数据。我们还要注意" 边界条件"。例如，当守卫使用>=或者>时，测试相等的情况。
- 如果函数使用递归，应该测试 0 的情况，1 的情况和一般情况。

对于mysteryMax一例，按照上述原则，应该能选择测试数据 6 6 2，因为前两个输入处于下列守卫的边界

```
x>y && x>z      y>x && y>z
```

在第八章讨论证明时，我们将继续讨论本节的思想。

习题

4.15 设计下列函数的测试数据

```
allEqual:: Int -> Int ->Int ->Bool
```

此函数测试三个输入参数是否完全相等。

4.16 使用前一问题的测试数据测试下列函数。

```
solution m n p = ((m+n+p) == 3*p)
```

试讨论你的结果。

4.17 设有函数

```
allDifferent :: Int -> Int -> Int -> Bool
```

只有当三个输入均不相同时，函数返回 True，试设计此函数的黑盒测试数据。

4.18 使用上题的测试数据测试下列函数

```
attempt m n p =(m /= n)&&(n /= p)
```

4.19 为下列函数设计测试数据

```
howManyAboveAverage :: Int -> Int -> Int -> Int
```

此函数计算三个输入中大于其平均数的个数。

4.20 为 *2* 的正整数幂函数设计测试数据

小结

本章介绍了程序设计的一些基本原理。

- 我们应该考虑如何最好地使用已有的知识。如果我们已经定义了一个函数 f，那么有两种使用它的方法：
  - 可以用 f的定义作为新定义的模型。
  - 可以在新定义中直接使用 f
- 我们应该考虑如何把一个问题分解为更小、更容易解决的问题。我们应该考虑假如我有...?
- 使用递归定义函数。

我们还介绍了递归的基本知识，以及如何在实践中应用递归定义各种函数。在第七章介绍列表上的递归时，我们会看到更多的例子。

最后我们指出一种设计测试数据的原则性方法，而非凭空想象选择测试数据。

第五章　数据类型：多元组和列表

我们已经介绍了 Int, Float 和 Bool 等基本类型上的程序，以及程序设计的一般方法。但是，在实际问题中我们希望表达更复杂的事物，如第一章中看到的 Picture 一例。

本章介绍在 Haskell 语言中构造复合数据的两种方法；它们是 **多元数组** 和 **列表**。这两种类型足以使我们表示许多不同种类的“结构”信息。我们将在第十四章和第十六章介绍其他自定义类型的方法。

本章重点介绍 Haskell 提供的定义和处理多元组和列表的方法。多元组上的预定义函数不多，但是，Haskell 语言为列表提供了许多预定义函数和运算。除此之外，我们还可以使用“列表概括”表示法描述如何用列表生成新的列表。

为了描述列表上的预定义函数，我们需要解释 **多态**。多态是一个 Haskell 函数可以应用于多种类型的机制：例如，列表上的 length 函数可应用于任何类型的列表上。

在本章的基础上，下一章将介绍更多有关列表的例子。

§5.1　多元组，列表和串简介

多元组和列表是由一些数据构成的单个对象，但是它们具有不同的性质。一个多元组是由一定数目确定类型的值构成的单个对象，这些类型可能互不相同。一个列表是任意数目的同类型值构成的单个对象。

用一个例子可以说明两者的区别。假设我们要构造一个超市的简单模型，其中的一个组成部分是记录一个顾客的购物清单。一件商品具有品名和价格（用便士表示），我们需要将这两个信息结合起来。为此，我们使用多元组，如

```
("Salt: 1 kg",139)
("Plain crisps",25)
```

其中每个元组由一个 String 和一个 Int 构成。用双引号括起来的串表示商品的名，Int 表示其价格。String 实质上是字符串的列表，这将在 5.9 节讨论。两个值`("Salt:  1kg",139)`, `("Plain crisps",25)`属于 **多元组类型**

```
(String, Int)
```

正如类型 (String, Int) 的说明所示，此类型的每个成员有两个分量，一个 String 和一个 Int。因此，如果给定此类型的一个成员，那么我们可以确定其分量的类型，这也表示我们可以检查这些分量的使用方式是否合理。例如，我们可以检查第二个分量是以 Int 类型来对待的，而非以 Bool 类型来对待。所以，我们可以保持在第一章提到的性质：我们可以在程序执行前对程序进行类型检测，程序中的任何类型错误可以在程序运行前发现。

如何表示购物清单呢？我们预先不知道一个顾客购买了多少东西。有的顾

客购买了一件，有的购买了三件。但是，每件商品是由同一种方式表示的，即均为 (String,Int) 类型的成员，所以我们把一个顾客所购商品表示为这些成员的一个 **列表**，如`[("Salt:1kg",139),("Plain crisps",25),("Gin:1t",1099)]`它是类型`[(String, Int)]`的一个成员。此列表类型的其他成员包括空列表`[]`，`[("Salt:1 kg",139),("Plain crisps",25),("Plain crisps",25)]`等。因为列表的每个成员具有相同的类型，所以我们可以预测列表中每件商品的类型。与此相对的是其成员可以有不同类型的列表：如果选择列表中的第一个元素，我们不能预测其类型，因此失去了在程序运行前进行类型检测的能力。因为我们想保持这种类型检测的重要性质，所以 Haskell 的列表必须包含同类型的元素，但是，不同类型的列表可以包含不同类型的元素。

在 Haskell 中可以命名类型，以便于阅读。在本例中我们命名两个类型。

```
type ShopItem = (String,Int)
type Basket   = [ShopItem]
```

其中关键字 type 说明后面引进一个类型定义，而非值的定义。由 3.7 节所注，我们也可以由类型名`ShopItem`和`Basket`中开始的大写字母辨别出来，系统中预定义了类型

```
type String = [Char]
```

所以 Haskell 把串作为一个特殊的列表类型对待。`ShopItem`和 String 称为它们所命名的类型的 **同义词**。

下一节详细介绍元组类型以及它在实际应用中的例子。

§5.2 元组类型

上节介绍了元组类型的概念，一般地，一个元组类型是由简单类型的分量构成的。类型 $(t_1, t_2, ..., t_n)$由元组值 $(v_1, v_2, ..., v_n)$构成，其中 $v_1 :: t_1, v_2 :: t_2, \cdots, v_n :: t_n$。换言之，元组的每个分量 v_i 必须具有元组类型相应位置的类型 t_i。

” 元组” 名称来源于二元组，三元组，四元组等，为此在这里统称为（多）元组。其他的程序设计语言中称之为记录或结构，详细的比较参见附录 A。

我们可以用下面定义的类型`ShopItem`表示超市的商品

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

我们看到它包括了象（`"Gin, 1lt",1099`）这样的成员。Haskell 把上面这样的类型定义视为别名，`ShopItem`与`(String, Int)`的作用完全一样。这样的类型定义便于程序的阅读，也有利于类型错误信息的理解。

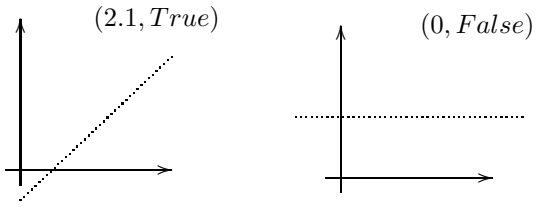
元组类型在程序中是如何使用的呢？我们来看几个例子。

例 5.1 *首先，利用元组可以使函数返回一个复合结果，例如，下列函数同时返回两个整数的最小值与最大值：*

```
minAndMax :: Int -> Int -> (Int,Int)
minAndMax x y
```

```
  | x >= y    = (y,x)
  | otherwise = (x,y)
```

例 5.2 假设要求一个问题的数值解，但我们不能确定是否在每一种情况下都存在解：例如，试问一直线是否与水平轴相交。



一种办法是让函数返回一个类型为(Float, Bool) 的二元组。如果布尔分量是 False, 则表示无解；若布尔分量为 True, 则表示有解，如(2.1, True)表示 *2.1* 是解。

模式匹配

下面讨论如何定义元组上的函数。元组上的函数通常是采用 **模式匹配 (pattern matching)** 定义的。在定义中不使用单个变量表示类型 (Int, Int) 的参数，而使用一个 **模式 (pattern)** (x,y)：

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

在应用函数中，模式中的分量与实际参数的相应分量匹配。如当函数addPair应用于参数 (5, 8) 时，5 与 x匹配，8 与 y匹配，计算过程为

```
addPair (5, 8)
⇝ 5+8
⇝ 13
```

模式可以包含原子和嵌套模式，如下例

```
addPair (0,y) = y
addPair (x,y) = x + y

shift :: ((Int, Int),Int) -> (Int, (Int, Int))
shift ((x,y),z) = (x,(y,z))
```

选取元组的某些分量的函数可以用模式匹配定义。如类型ShopItem上的函数：

```
name  :: ShopItem -> String
price :: ShopItem -> Int
name  (n,p) = n
price (n,p) = p
```

Haskell 预定义了二元组上的 **选择函数**：

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

有了这些选择函数，我们可以不使用模式匹配。例如，可以重新定义函数addPair:

```
addPair:: (Int, Int) -> Int
addPair  p = fst p + snd p
```

但是，一般地，模式匹配定义较使用选择函数的定义更易读。

例 5.3 4.4节介绍了 *Fibonacci* 数列

```
 0,1,1,2,3,5, ... ,u,v,(u+v),...
```

并且给出此序列的一个低效的递归定义。使用元组可以给出此问题的一个高效解。序列中的下一个值为前两个值之和，所以我们定义一个返回两个相邻值的函数。换句话说，我们要定义具有如下性质的函数fibPair:

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

给定这样的一个二元组(u,v)，可以得到下一个二元组(v,u+v)，这个任务由函数fibStep完成：

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

由此得到下面的“*Fibonnacci* 二元组”函数

```
fibPair  :: Int -> (Int, Int)
fibPair  n
 | n ==0     = (0,1)
 |otherwise  = fibStep (fibPair (n-1))
```

最后可以定义

```
fastFib :: Int -> Int
fastFib = fst . fibPair
```

其中'.'是函数的复合运算：fibPair的结果成为fst的输入，后者取其第一个分量。

注 7 一个二元组还是两个参数？要注意下列两个函数的区别

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

fibStep有一个参数，参数为一个二元组；而fibTwoStep有两个参数，每一个参数是一个数。今后我们会看到，第二个函数的使用比第一个函数更灵活。这里要注意这两个函数的区别，如果混淆两者，会得到类型错误，如fibStep 2 3和fibTwoStep (2,3)。我们将在 *10.7* 节继续讨论这两个函数间的关系。

习题

5.1 定义下列函数maxOccurs :: Int -> Int -> (Int, Int)，函数返回两个整数中的最大值及其出现的次数。在此基础上定义下列函数

```
maxThreeOccurs :: Int -> Int -> Int -> (Int, Int)
```

函数的功能与前一函数类似。

5.2 定义一个函数

```
orderTriple :: (Int, Int, Int) -> (Int, Int ,Int)
```

函数将一个整数三元组按递增序排列。你可以使用早先定义的函数`maxThree`, `middle`和`minThree`。

5.3 定义一个求得一直线与 x 轴交点的函数，你需要考虑如何给函数提供直线信息。

5.4 为前面各题设计测试数据，并解释你在每种情况所做的选择。用测试数据测试每个函数。

§5.3 定义列表

列表是表现力很丰富的数据类型。在此我们只列出三个潜在的应用：一个文本可表示为一些文本行的列表，而每个文本行是一些词的列表；一个信息集可表示为数据项的列表，如超市购物清单；一个度量仪上的一系列读数可表示为 Float 的列表。

与此同时，许多不同的事物都可用列表表示，其中包括第一章的 Picture 的实现。给定一个列表，我们可以根据不同的标准将其分类，对其排序，从中选取某些元素或者将其所有元素按某种方式变换。我们可以把列表合并，如将列表连接或者将列表的相应元素放在一起。我们可以使用某种运算将列表的所有元素合并，如求它们的和，求它们的最大值或它们的合取等。Haskell 的引导库 Prelude 和库模块`List.hs`包含了许多列表上的预定义函数和运算。

因为 Haskell 提供了如此多的列表预定义函数，我们可以用两种截然不同的方法讨论列表。我们可以自己定义列表处理函数，然后待我们理解库函数的定义后再使用它们[1]。另一方面，我们可以采纳" 工具箱" 的方法，只讨论库函数以及如何使用它们，我们在这里想做的是将两种方法结合起来：在给出函数定义之前，先介绍这些函数的使用，然后介绍这些函数是如何定义的，以及如何自定义其他函数。

为了充分欣赏列表上的通用运算，我们必须讨论 Haskell 如何处理通用或**多态函数**（见 5.7 节）以及**高阶函数**的概念（见 9.2 节）

本章的剩余部分将介绍 Haskell 列表上的主要函数；随后的章节将介绍如何使用这些函数以及如何自定义这些函数和其他一些列表函数。

§5.4 Haskell 的列表

Haskell 的一个列表是一个给定类型的若干元素的集合。对于每个类型 t，存在一个 Haskell 类型`[t]`，其元素类型为 t 的列表。

[1]基本上，这是本书第一版采用的方法

```
[1,2,3,4,1,4] :: [Int]
[True]        :: [Bool]
```

将其读为“[1,2,3,4,1,4]是 Int 的一个列表”,“[True]”是 Bool 的一个列表。String 是[Char]的别名，下列两个列表是同一个列表：

```
['a','a','b'] :: Sting
"aab"         :: String
```

我们可以建立任意类型的列表，如函数的列表，整数列表的列表：

```
[fac,fastFib]       :: [Int -> Int]
[[12,2],[2,12],[ ]] :: [[Int]]
```

可以看出，元素为e1,e2到en的列表可用方括号将其扩起来表示：

```
[e1,e2,...,en]
```

一个特别情况是不含任何元素的空列表 [], 它是任何列表类型的元素。

列表中元素的顺序是重要的，同样重要的是一个元素在列表中出现的次数。下列三个列表互不相同：

```
[1,2,1,2,2]
[2,1,1,2,2]
[2,1,1,2]
```

前两个列表的长度为 5，第三个列表的长度为 4；第一个列表的第一个元素是 1，而第二个列表的第一个元素是 2。集合是另一个类型，其中一个元素出现的次序和次数是无关紧要的。我们将在第十六章介绍集合。

Haskell 提供了数的列表，字符列表和其他枚举类型列表的其他表示方法：

- [n .. m]表示列表[n,n+1, ... , m]；如果 n超过 m，则列表为空；

  ```
  [2 .. 7]     = [2,3,4,5,6,7]
  [3.1 .. 7.0] = [3.1,4.1,5.1,6.1]
  ['a' .. 'm'] = "abcdefghijklm"
  ```

- [n, p .. m]表示这样的列表：前两个元素为 n和 p, 其元素构成差为p-n的等差数列，最后一个元素为 m。例如

  ```
  [7,6 .. 3]  =[7,6,5,4,3]
  [0.0,0.3 .. 1.0]=[0.0,0.3,0.6,0.9]
  ['a','c' .. 'n'] ="acegikm"
  ```

- 从以上两种情况中可以看出，如果 m 不属于等差数列，则列表的最后一个元素是等差数列中小于等于/大于等于 m 的最大/最小值。也可能出现这样的情况：Float 上的舍入错误使得列表不同于预期的结果, 参见习题。

下一节将介绍一种很强的列表表示法，使用这种方法可以定义列表上各种函数。

习题

5.5 表达式[0,0.1 .. 1]的值为何？在Hugs系统中检验你的答案，如果答案不同于你的预期结果，解释两者之间可能存在的差别。

5.6 列表[2,3]包含多少个元素？列表[[2,3]]呢？列表[[2,3]]的类型是什么？

5.7 表达式[2 .. 2]求值的结果是什么？表达式[2,7 .. 4]呢？试求[2,2 .. 2]的值。欲在Hugs中（在Windows或者Unix下）**中断计算**，请按Ctrl - C。

§5.5 列表概括

函数语言的一个重要特征是它的列表概括记法，这是其他语言不具备的。

使用列表概括，一个列表可用另一个列表来描述。一个列表 **生成** 元素，这些元素经过**测试**和**转化**形成结果列表的元素。本节介绍含一个生成器的列表概括，17.3 节将介绍一般情况。但是，本节介绍的简单情况在设计列表处理程序时是非常有用的。我们将通过一系列的例子介绍这个概念。

例 5.4 设列表ex为[2,4,7]，则列表概括[2*n| n<-ex]表示列表[4,8,14]，它包含列表ex的所有元素的 2 倍。我们可将上述列表概括读作”取所有的2*n，其中 n是ex的元素”。其中符号”<-”模仿数学符号属于”∈”。我们可用下表来表示上述列表的计算过程：

```
[2*n | n<-[2,4,7]]
 n   = 2  4  7
 2*n = 4  8  14
```

例 5.5 类似地，[isEven n |n <-ex] ⇝ [True,True, False]其中isEven 的定义如下

```
isEven :: Int -> Bool
isEven  n = ( n 'mod' 2 ==0)
```

在列表概括中n<-ex称为 **生成器**，因为结果列表建立在它生成的元素上。在符号”<-”的左边是一个变量，右边是从中取得元素的列表。

例 5.6 一个生成器可以与一个或几个**测试**相结合，每个测试是一个布尔表达式：

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

上述列表概括可读作“取所有的2*n，其中 n取自列表ex，n是偶数且大于 3”。

同样，我们可以用表格表示它的计算过程：

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

```
2*n      =     8
```

因此，最后的结果是列表[8]，因为 *4* 是列表中[2,4,7]中唯一满足是偶数且大于 *3* 的元素。

例 5.7 在符号"*|-*" 的左边除使用变量外，还可使用模式，如

```
addPairs :: [(Int, Int)] -> [Int]
addPairs pairList = [m+n|(m,n) <- pairList]
```

在此例中，我们选取列表pairList的所有元素，将每个元素的分量相加形成结果列表的元素，例如

```
[m+n| (m,n) <- [(2,3),(2,1),(7,8)]]
 m   = 2 2 7
 n   = 3 1 8
 m+n = 5 3 15
```

结果为

```
addPairs [(2,3),(2,1),(7,8)] ⇝ [5,3, 15]
```

例 5.8 在以上的列表概括中还可以加上测试。

```
addOrdPairs :: [(Int, Int)] -> [Int]
addOrdPairs pairList =[m+n |(m,n) <- pairList, m<n]
```

使用同样的输入列表，列表概括的计算如下：

```
[m+n | (m,n)<- [(2,3),(2,1),(7,8)], m<n]
m   = 2 2 7
n   = 3 1 8
m<n = T F T
m+n = 5   15
```

其结果是 addOrdPairs [(2,3), (2,1),(7,8)] ⇝ [5,15] 因为列表中的第二个二元组未通过测试。

例 5.9 注意，我们只利用布尔表达式对元素做测试，其结果是过滤出列表中满足测试条件的元素。为了求出一个串中的所有数字，可以定义

```
digits :: String -> String
digits st = [ch |ch<-st, isDigit ch]
```

其中函数

```
isDigit :: Char -> Bool
```

是引导库中定义的函数，函数对数字*'0'*, *'1'* 到*'9'* 的字符返回True。

例 5.10 一个列表概括可以用于函数定义中。假设要检查一个列表中的所有元素是否均为偶数或者均为奇数，可以定义

```
allEven xs = (xs == [x | x<- xs, isEven x])
allOdd  xs = ([] == [x | x<- xs, isEven x])
```

在下一节讨论一个简单的图书馆数据库时，我们将看到列表概括在实践中的应用。

习题

5.8 定义将整数列表中所有元素加倍的函数：

```
doubleAll :: [Int] -> [Int]
```

5.9 定义将串中所有小写字母转换为大写字母其他字符保持不变的函数；

```
capitalize :: String -> String
```

你能否修改上述函数使之删除列表中的所有非字母字符？

```
capitalizeLetters :: String -> String
```

检查模块Char是否有对解决此问题有帮助的函数。

5.10 定义函数

```
divisors :: Int -> [ Int]
```

如果输入是一个正整数，则函数返回所有因子构成的列表，否则返回空列表。例如，`divisors 12` ⇝ `[1,2,3,4,6,12]`。一个素数是只有因子 *1* 和它本身的正整数。使用divisors和其他函数定义下列函数

```
isPrime :: Int -> Bool
```

它检查一个正整数是否素数（如果输入不是正整数，则返回 `False`）.

5.11 定义下列函数

```
matches :: Int -> [Int] -> [Int]
```

它返回第一个参数在第二个参数中的所有出现。例如`matches 1 [1,2,1,4,5,1]` ⇝ `[1,1,1]`, `matches 1 [2,3,4,6]` ⇝ `[]`。使用matches或者其他函数重新定义下列函数：

```
elem :: Int -> [Int] -> [Int]
```

它检查第一个参数是否第二个参数的元素。例如，

```
elem 1 [1,2,1,4,5,1] ⇝ True
elem 1 [2,3,4,6] ⇝ False
```

因为 *elem* 是引导库函数，所以你需要按 *49* 页所述隐蔽此函数。

§5.6 一个图书馆数据库

本节介绍图书馆借阅数据的一个简单模型，并由此演示列表概括在实践中的应用。

图书馆利用一个数据库记录读者借阅的图书。我们首先讨论描述这个数据库的类型以及从数据库中抽取数据的函数。然后讨论如何描述数据库的变化，最后探讨如何测试数据库函数。

类型

在对这个问题建立模型时，我们首先考虑对象的类型。读者和书用串来表示：

```
type Person = String
```

```
type Book   = String
```

数据库可用各种不同的方式表示，其中包括下列三种选择：

- 每次借阅用二元组 (Person, Book) 来表示；
- 我们可以使用二元组(Person, [Book])把每个人与他借阅的书联系起来，或者
- 我们可以把每本书与借阅此书的读者列表相关联，即([Person], Book)。

在这里我们选择用二元组 (Person, Book) 列表来表示数据库。如果列表中包含二元组("Alice", "Asterix"), 则它表示"Alice"借阅名为"Asterix"的图书。为此，我们定义

```
type Database = [(Person, Book)]
```

这种表示方法的优点是简单，并且读者和书以同样方式处理，而不是用非对称的方法处理。这个类型的对象如下例

```
exampleBase :: Database
exampleBase
 = [("Alice", "Tintin"), ("Anan", "Little Women"),
       ("Alice", "Asterix"), ("Rory","Tintin")]
```

定义好对象的类型之后，我们来考虑数据库上的函数。

- 给出一个读者，求该读者借阅的图书；
- 给出一本书，求借阅此书的读者（假定一本书有多个拷贝）；
- 给出一本书，问此书是否被借出；
- 给出一读者，求该读者借阅的图书数目。

每个**查询函数**取一个数据库，一个人或一本书，然后返回查找结果。它们的类型是

```
books       :: Database -> Person -> [Book]
borrowers   :: Database -> Book -> [Person]
borrowed    :: Database -> Book -> Bool
numBorrowed :: Database -> Person -> Int
```

注意到 borrowers 和 books 返回的结果是列表，其中可以包含 0 个，1 个或多个元素，特别地，空列表表示某书没有借出或某人没有借书。

此数据库还需要定义两个函数。我们需要描述借出一本书和归还一本书后数据库的变化。这两个函数取一个数据库和借阅或归还信息，返回一个**不同**的数据库。即原数据库加上或去掉相应的借阅信息。这些**更新**函数具有下列类型

```
makeLoan   :: Database -> Person -> Book -> Database
returnLoan :: Database -> Person -> Book -> Database
```

定义查询函数

我们将定义函数

```
books :: Database -> Person -> [Book]
```

其他查询函数的定义类似。对于 exampleBase, 我们有下列结果

```
books exampleBase  "Alice" = ["Tintin", "Asterix"]
books exampleBase  "Rory"  = ["Tintin"]
```

如何得到这些结果呢? 对于"Alice"的情况, 从列表的第一个元素开始顺序查找第一个分量为"Alice"的二元组, 将其第二个分量列入结果列表中。也可以用列表概括表示

```
[book | (person,book)<-exampleBase, person =="Alice"]
person  =        "Alice"     " Anna"          "Alice"     "Rory"
book    =        "Tintin"  "Little Women"    "Asterix"    "Tintin"
(person=="Alice") =  T          F               T            F
book    =         "Tintin"                     "Asterix"
```

由此可定义函数 books 如下:

```
books  :: Database -> Person -> [Book]            (books.1)
books  dBase findPerson
    = [book | (person, book) <- dBase, person == findPerson]
```

注意, 在上述定义中Person是一个类型, 而person是类型为Person的变量。其他查询函数的定义与 books 的定义类似, 留给读者作为练习。

定义更新函数

数据库由函数 makeLoan 和 returnLoan 修改或更新。读者借阅一本书时, 需要在数据库中添加一个二元组

```
makeLoan :: Database -> Person -> Book -> Database
makeLoan dBase pers bk = [(pers,bk)] ++dBase
```

其中运算 ++ 用于将两个列表连接起来, 即单元素列表[(pers, bk)]与原数据库dBase。

当读者归还一本书时, 我们需要在数据库中删除二元组(pers,bk)。为此, 我们需要检查数据库中的所有元素, 保留那些不等于(pers, bk)的元素, 即

```
returnLoan ::Database -> Person -> Book -> Database
returnLoan dBase pers bk
      = [pair | pair <-dBase, pair /=(pers,bk)]
```

注意, 在上述定义中使用了一个简单变量pair, 而不是模式。这是因为我们只需检查一个二元组是否等于二元组(pers, bk), 而不需要对其分量进行分别处理。另一方面, 我们可以在定义中使用模式: [(p,b) | (p,b) <- dBase, (p,b) /=(pers, bk)], 其结果是完全一样的。

我们定义的函数 returnLoan 将删除数据库中**所有的**二元组(pers, bk), 我们将在 9.3 节的习题中讨论此问题。

测试

Haskell 解释器如同一个计算器，这一点对测试数据库函数是很有用的。任何函数的测试都可以通过在 Hugs 提示符下键入表达式进行。例如

```
makeLoan [] "Alice" "Rotten Romans"
```

欲测试更大的例子，最好将测试数据写入脚本，例如，在exampleBase的定义中加入下列测试：

```
test1 :: Bool
test1 = borrowed exampleBase  "Asterix"
test2 :: Database
test2 = makeLoan exampleBase "Alice"  "Rotten Romans"
```

将这些表达式加入脚本后，在每次测试时不需要键入整个表达式。另一个有帮助的建议是使用 $$，它表示“上一次求值的表达式”。下列序列表示借阅一本书，再借阅一本书，然后归还第一次借阅的书：

```
makeLoan exampleBase  "Alice"  "Rotten Romans"
makeLoan $$  "Rory"  "Godzilla"
returnLoan  $$  "Alice"  "Rotten Romans"
```

注 8 列表概括中的变量 列表概括中变量的作用需要注意。上述books的定义(books.1)也许显得有些过于复杂。或许我们想如下定义books

```
books dBase findPerson
  = [book | (findPerson, book)<-dBase]            (books.2)
```

其结果是返回**所有**读者借阅的所有图书，而不仅仅是一个特定读者findPerson所借阅的图书。其原因是在(findPerson, book)中的findPerson是一个新变量，而不是定义左边的同名变量。实际上，上述定义(books.2)与下列定义等价：

```
Books dBase findPerson = [book | (new,book)<- dBase]
```

显然，在这里不存在new的值与findPerson的值相等的约束。

习题

5.12 求下列表达式的值

```
books exampleBase "Charlie"
books exampleBase "Rory"
```

5.13 定义函数borrowers, borrowed和numBorrowed。你或许在定义numBorrowed时需要返回列表长度的函数length。

5.14 求下列表达式的值

```
returnLoan  exampleBase  "Alice"  "Asterix"
returnLoan  exampleBase  "Alice"  "Little Women"
```

5.15 假如你使用`[(Person, [Book])]`描述数据库，你会如何实现数据库函数。

§5.7 通用函数：多态

在介绍 Haskell 引导库及其他库中列表上的函数之前，我们需要了解 **多态** 的概念，其字面含义是“有许多形态”。如果一个函数“有许多类型”，则称之为多态函数。许多列表函数是多态函数。例如返回列表长度（`Int`）的函数 `length`，此函数可应用于许多类型的列表：

```
length :: [Bool] -> Int
length :: [[Char]] -> Int
```

等等。如何将这样的信息在 length 的类型中表达呢？

我们定义

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

其中 a 是一个 **类型变量**。任何以小写字母开头的标识符均可做类型变量。习惯上用字母表中的开始几个字母 a, b, c 等作类型变量。正如在下列定义中

```
square x = x*x
```

变量 x 代表任意值一样，类型变量代表任意类型，所以我们看到诸如类型`[Bool] -> Int`，`[[Char]] -> Int`等，它们是将类型变量 a 用特别类型 Bool 和`[Char]`代替的结果。类型`[Bool] -> Int`称为类型`[a] -> Int`的一个 **特例**。因为 length 的每个类型均为`[a] -> Int`的一个特例，我们称`[a] -> Int`为 length 的 **最通用类型** 或者 **最广类型**。

将两个列表连接的运算 ++ 的类型为

```
[a] -> [a] -> [a]
```

类型变量 a 表示” 任意类型”，但是要注意所有的 a 表示同一类型，正如在下列定义中

```
square x = x*x
```

所有的 x 表示同一个（任意）值一样。`[a] -> [a] -> [a]`的特例包括

```
[Int] -> [Int] -> [Int]
```

但不包括类型

```
[Int] -> [Bool] -> [Char]
```

显然，我们不能期待将一个数的列表与一个布尔值列表连接，结果为串。

另一方面，函数 zip 和 unzip 在列表的二元组和二元组的列表间转换，这其中涉及两个类型变量：

```
zip   :: [a] -> [b] -> [(a,b)]
unzip :: [(a,b)] -> ([a],[b])
```

zip 类型的特例包括`[Int] -> [Bool] -> [(Int, Bool)]`其中 a 和 b 用不同的类型 (Int 和 Bool) 代替。当然，也可以用同一类型代替两个不同的类型变量

如[Int] -> [Int] -> [(Int, Int)] 或者一般地[a] -> [a] -> [(a,a)]。

类型与定义

如何定义多态函数呢？考虑直接返回其参数的恒等函数的定义

```
id x = x
```

此定义对 x 的类型没有任何限制，函数所做的是直接返回参数 x。由此可知，函数的输出类型与输入类型相同，其最广类型为

```
id :: a -> a
```

寻找函数最广类型的原则是函数的类型尽可能通用，并且和定义对类型的限制协调，对于 id 函数，唯一的限制是输入与输出类型相同，类似地，考虑下列函数的定义:

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

其中对 x 与 y 没有任何限制，所以它们可以有不同的类型 a 和 b：

```
fst :: (a,b) -> a
```

最后一个例子是

```
mystery (x,y) = if x then 'c'  else 'd'
```

在上述定义中，x 在等式右边具有类型 Bool，而 y 在右边没有出现，对 y 的类型无任何限制，故mystery的类型为

```
(Bool, a) -> Char
```

我们将在第七章检查引导库中的许多函数，并将看到一个函数或者对象具有尽可能广的、与定义的限制一致的类型。在第十二章我们将讨论类型检验机制。

Hugs 的:type命令可用于求一个函数定义的最广类型。如果你已说明了一个函数的类型，那么在求类型前先注释掉类型说明。

多态与重载

多态与重载是一个函数名可用于不同类型的机制，但两者有重大区别。

一个多态函数在所有类型上的定义是相同的，例如 fst

```
fst ( x y) = x
```

即函数在所有特例上的定义是相同的。

另一方面，诸如==这样的重载名在不同的类型上有不同的定义，所以，同一个名在不同的类型上表示不同但类似的函数。例如，==在 Int 上是预定义的，而在二元组上定义如下：

```
(n, m) == (p, q) = (n == p)&& (m == q)
```

有关重载的详细内容见第十二章。

习题

5.16 求下列函数snd和sing的最广类型：

```
snd (x, y) = y
sing x     = [x]
```

5.17 试解释为什么[[a]] -> [[a]]是id的类型，而不是它的最广类型。

5.18 考虑本章的例子

```
shift :: ((Int, Int), Int) -> (Int, (Int, Int))
shift  ((x, y), z) = (x, (y, z))
```

注释掉shift的类型说明，求shift的最广类型。

§5.8 Haskell 引导库 Prelude 中的列表函数

具备了上一节的知识后，我们可以看一下引导库 Prelude 中的多态列表运算，如图 5.1。表中列出了函数或运算的名，它的类型，其功能的简单描述和一个例子。例如，length 函数

`length [a] -> Int` 列表的长度

`length "word"` $\rightsquigarrow 4$

除表 5.1 中的多态函数外，标准引导库还提供了特殊类型上的各种运算，表 5.2 列出其中的一部分。重载函数 sum 和 product 将在第十二章做进一步讨论。

类型的重要意义

一个函数的最重要信息是它的**类型**，尤其是表 5.1 中列出的函数的多态类型。假设我们要寻找这样的一个函数：由一个元素的若干拷贝构造一个列表，那么函数一定带有一个元素和一个数值，然后给出一个列表，因此其类型必为

```
Int -> a -> [a]    a -> Int -> [a]
```

检查表 5.1，我们很快发现函数 replicate 具有其中的一个类型，它确实是我们需要的函数。如果要将一个列表前后倒置，那么它的类型必为`[a] -> [a]`。尽管有多个函数具有这样的类型，但是寻找可用类型范围迅速减小。

以上的观察不限于函数语言，但是它对于支持多态或者通用函数或运算的语言尤其有用。

其他函数

基于两个不同的原因，我们还没有描述引导库中的所有函数。第一，有些函数是 **高阶的**，我们将推后对此讨论；第二，有些函数，如`zip3`，显然是我们这里讨论的函数的变型。类似的原因，我们没有列出库`List.hs`中的所有函数。请读者参考库文件，其中包含函数的类型和功能等信息。

下一章我们探讨如何使用引导库函数构造自己的函数定义。在此之前，我们介绍串，列表类型的一个例子。

:	a -> [a] -> [a]	在列表的前面添加一个元素 3:[2,3] ⇝ [3,2,3]
++	[a] -> [a] -> [a]	将两个列表连接在一起 "Ron" ++ "aldo" ⇝ "Ronald"
!!	[a] -> Int -> a	xs!!n返回xs的第 n 个元素 [14,7,3]!!1 ⇝ 7
concat	[[a]] -> [a]	将一个列表的列表连接成一个列表 concat [[2,3],[],[4]] ⇝ [2,3,4]
length	[a] -> Int	列表的长度 length "word" ⇝ 4
head, last	[a] -> a	返回列表的第一个/最后一个元素 head "word" ⇝ 'w' last "word" ⇝ 'd'
tail, init	[a] -> [a]	除第一个/最后一个的所有元素 tail "word" ⇝ "ord" init "word" ⇝ "wor"
replicate	Int -> a -> [a]	构造参数元素的 n 个拷贝的列表 replicate 3 'c' ⇝ "ccc"
take	Int -> [a] -> [a]	从列表的前端取 n 个元素 take 3 "Peccary" ⇝ "Pec"
drop	Int -> [a] -> [a]	从列表的前端删除 n 个元素 drop 3 "Peccary"⇝"cary"
splitAt	Int -> [a]->([a],[a])	在给定的位置拆分列表 splitAt 3 "Peccary"⇝("Pec","cary")
reverse	[a] -> [a]	将列表元素顺序倒置 reverse [2,1,3]⇝ [3,1,2]
zip	[a] -> [b]->([a],[b])	用两个列表生成一个二元组的列表 zip [1,2] [3,4,5]⇝ [(1,3), (2,4)]
unzip	[(a,b)] -> ([a], [b])	将二元组的列表转换为列表的二元组 unzip [(1,5), (3,4)]⇝([1,3], [5,4])

图 5.1: 引导库 Prelude.hs 中的一些多态函数

`and`	`[Bool] -> Bool`	一个布尔值列表的合取 `and [True ,False]` ⇝ `False`
`or`	`[Bool] -> Bool`	一个布尔值列表的析取 `or [True ,False]` ⇝ `True`
`sum`	`[Int] -> Int` `Float -> Float`	一个数值列表的累加和 `sum [2,3,4]` ⇝ `9`
`product`	`[Int] -> Int` `[Float] -> Float`	一个数值列表的乘积 `product [2,3,4]` ⇝ `24`

图 5.2: 引导库 Prelude.hs 中的一些单态函数

§5.9 类型 String

类型 String 是列表的一个特例

```
type String = [Char]
```

表 5.1 中的所有多态引导库函数均可用于串上。3.5 节介绍了如何使用'escape'键书写特殊字符, 如换行符 '\n' 和制表符 '\t'。这些字符可用于构造串, 如

```
"baboon"
" "
"\99a\116"
"gorilla \n hippo \n ibex"
"1 \t 23 \t 456 "
```

如果我们在 Hugs 下计算这些串, 其结果与输入完全一致。为了消除 escape 字符及双引号, 我们需要完成输出运算。这可以用下列 Haskell 原始函数完成:

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

其结果是将参数串显示在屏幕上。将 putStr 应用于以上串将得到如下结果

```
baboon

cat
gorilla
hippo
ibex
1       23       456
```

可以用"++" 将串连接起来, 如 `"cat"++"\n"++"fish"` 输出到屏幕的结果是

```
cat
fish
```

注 9 名、串和字符 *a, 'a'* 和*"a"* 容易混淆。其区别总结如下:

a 是一个名或变量，它可能具有任何类型

'a' 是一个字符

"a" 是一个串，它碰巧由一个单字符构成。

类似地，下列两者有区别

`emu` 一个Haskell名或者变量

`"emu"` 一个串

串上的其他函数可在库`String.hs`中找到。

串和值

重载函数 show 和 read 是 Haskell 预定义函数，它们将一个值转换为一个串或者将串转换为值，例如

```
show (2+3)              ⇝  "5"
show (True || False)    ⇝  "True"
```

相反地，函数 read 将串转换为值，如

```
read "True"   ⇝  True
read "3"      ⇝  3
```

在某些情况下，read 的类型不明确，此时可以给表达式一个类型说明，例如

```
(read "3" ) :: Int
```

其结果是3，类型为 Int。

有关 read 和 show 的类型详细说明见第十二章。

习题

5.19 定义一个函数，将小写字母转化为大写字母，非小写字母不变。

5.20 定义函数`romanDigit :: Char -> String` 它将一个数字转化为相应的罗马数字。如，将*'7'* 转换为*"VII"* 等。

5.21 定义函数`onThreeLines :: String -> String -> String -> String` 它将三个输入串转换为一个输出串，屏幕打印输出串时，三个输入串分别显示在三行上。

5.22 定义函数`onSeparateLines :: [String] -> String` 屏幕打印函数的输出结果时输入列表中的各串分行显示。

5.23 定义函数`duplicate ::String -> Int -> String` 其输入为一个串和一个整数 n，结果是输入串的 n个拷贝连接在一起。如果 n小于等于 *0*，则结果为空串*""*，如果 n为 *1*，则结果是输入串本身。

5.24 定义函数`pushRight ::String -> String` 它将在输入串前添加若干空格，形成一个长为linelength的串。例如，如果linelength为 *12*，则`pushRight "crocodile"`为`"  crocodile"`。如何使linelength成为函数的一个参数？

5.25 你能指出前一函数说明的不足吗？找出一种函数没有定义的情况。这是一种例外情况。

5.26 定义函数`fibTable ::Int -> String` 它将生成一个 *Fibonacci* 数列构成的表。例如, `putStr(fibTable b)` 的结果是

```
n  fib n
0  0
1  1
2  1
3  2
4  3
5  5
6  8
```

5.27 定义函数使得 *5.6* 节中数据库的输出更易读。

小结

本章介绍了结构类型元组和列表以及它们之间的区别：对于一个给定的元组类型 $(t_1, t_2, ..., t_n)$, 它的所有元素具有相同的形式 $(v_1, v_2, ..., v_n)$, 其中 v_i 是相应类型 t_i 的元素。另一方面, 列表类型包含不同长度的元素 $[e_1, e_2, ..., e_n]$, 而且所有的 e_i具有相同的类型 t。

我们介绍了元组上的模式匹配 – 一个模式, 如`(x,y)`可用来表示二元组类型的任意元素 – 并且看到使用模式匹配使得定义更容易理解。

本章的主要内容是 Haskell 的列表函数, 它们包括

- 各种书写基类列表元素的方法, 如`[2,4 ..  12]`;
- 列表概括, 其中列表的元素是由另一个列表的元素生成, 测试和转换而来的, 例如`[toUpper ch |ch <-String, isAlpha ch ]`, 它表示选取`String`中的字母字符并将其转换为大写字母。
- 标准引导库和`List.hs`库包含的函数;
- String 作为列表类型`[Char]`。

为了理解引导库函数, 我们讨论了多态的概念, 即一个函数可以有“许多类型”。这样的函数类型由类型变量描述, 例如`reverse ::  [a] -> [a]` 表示`reverse`可应用于任何类型的列表 (a 是一个类型变量), 返回同类型列表的元素。在随后的章节中, 我们将利用本章介绍的列表函数构造我们自己的定义, 并且介绍引导库的其他函数是如何定义的。

第六章　列表程序设计

本章目标有三个

- 再次讨论并扩展 Picture 一例以演示前一章介绍的概念；
- 讨论函数或表达式中的 **局部** 定义。局部定义对于大型程序显得尤其重要，因为它使得程序更易读而且更有效；
- 引进两组较大的练习，这些练习不同于以前的小练习，读者有机会编写较大的程序。这两组练习包括：
  - 扩展Pictures使其包含一个（空间）位置；
  - 超市结帐系统，它将根据条码机读入的一系列条码，输出一个格式化的帐单。

下一章将讨论如何使用递归实现列表上的主要函数。读者可以在阅读完 6.1 节和 6.3 节关于Picture和局部定义后跳到下一章。

§6.1　再次讨论 Picture 一例

本节再一次讨论在第一章介绍并且在 2.5 节做过进一步讨论的 Picture 例子。我们将讨论如何在如下类型上实现 Picture 上的某些运算

```
type Picture = [[Char]]
```

其中的一些运算将作为库函数。在水平镜子中反转一个图形，只需将图形各行顺序颠倒

```
flipH :: Picture -> Picture
flipH = reverse
```

要将一个图形置于另一个图形之上，只需将相应的两个列表连接在一起：

```
above :: Picture -> Picture  -> Picture
above = (++)
```

其中运算符 ++ 置于括号内后成为一个（前缀）函数。

如何在垂直镜子中反转一个图形呢？我们必须将每一行逆转，即将列表中的每个元素用某种方式转换。这便是列表概括的一个特色，因此可以定义

```
flipV :: Picture -> Picture
flipV  pic = [reverse line | line <-pic]
```

我们可以将上述定义解读为 “reverse pic中的每条line”。这便是将一个函数 f 应用于一个列表 xs 的每个元素的运算的例子，使用列表概括表示为

```
[f x | <- xs]
```

我们将在第九章看到这种运算本身是一个高阶函数。

下面讨论如何将两个图形并排放置。我们希望两个图形的对应行连接在一起，如第 26 页所示。如何做到这一点呢？如 flipV 一样，我们想把每一对对应行用++连接一起，不过，在此之前，我们需要将对应行联系在一起。这正是

引导库函数 zip 的目的，它取两个列表，将两个列表中的对应元素组成二元组，因此，可以定义

```
sideBySide :: Picture -> Picture -> Picture
sideBySide picL picR
   = [lineL++lineR | (lineL, lineR)  <- zip picL picR]
```

如果两个列表的长度不同，则 zip 将配对到较短的输入列表，舍弃较长表的剩余元素，因此sideBySide将剪掉较长图形底部的长出部分；如果两个列表长度相同，则无裁减。我们也可以用高阶函数 zipWith 定义 sideBySide; 我们将在第九章讨论之。

在我们的图形中，白色用点 '.' 表示，黑色用符号 '#' 表示。要将一个字符反色，可以定义

```
invertChar :: Char -> char

invertChar ch = if ch=='.'then '#' else '.'
```

上述定义将字符 '.' 和 '#' 互换（其他字符也转换为 '.'），那么，如何将整个图形反色呢？我们需要将一行的每个字符反色：

```
invertLine :: [Char] -> [Char]

invertLine line = [invertChar ch|ch <- line]
```

我们希望将此函数应用于图形的所有行：

```
invertColour :: Picture -> Picture
invertColour pic
     = [invertLine line | line  <- pic]
```

我们也可以将上述定义合并为一个定义

```
inverColour :: Picture -> Picture
invertColour  pic
    = [[invertChar ch | ch  <- line] | line  <- pic]
```

不过，使用辅助函数 inverLine 的定义更易读。

下一节我们将扩展图形的模型，除了图形内容之外，给图形添加一个位置。

习题

6.1 定义函数

```
superimposeChar :: Char -> Char -> Char
```

字符'.' 与它本身的重叠为'.', 与其他字符的重叠均为 '#'。

6.2 定义函数

```
superimposeLine :: [Char] -> [Char] -> [Char]
```

它利用superimposeChar将两条线（设长度相同）的对应字符叠加，例如

```
superimposeLine ".##." ".#.#" =  ".###"
```

或许你需要使用zip。

6.3 类似于superimposeLine, 定义函数

```
superimpose :: Picture -> Picture  -> Picture
```

它将两个图形叠加, 可假设两个图形有相同的尺寸。

6.4 使用函数putStr :: String -> IO() 使得

```
printPicture [".##.", ".#.#" ,".###" , "####"]
```

的结果是在字符终端上显示

```
.##.
.#.#
.###
####
```

6.5 图形的另一种表示法是使用如下类型

```
[[Bool]]
```

其中 True和 False分别表示图形的黑色和白色点。如何修改Picture上的函数以适应这种变化? 两种表示法的优点和缺点各是什么?

6.6 (较难) 定义函数

```
rotate90 :: Picture  -> Picture
```

它将图形顺时针旋转 *90* 度。例如, rotate90 作用于前题中的图形的结果是

```
#...
####
##.#
###.
```

提示: 原图形每行第 i 个元素构成的列表在水平镜子中的反射构成新图形的一行。

6.7 使用rotate90或其他函数定义将图形逆时针旋转 *90* 度的函数。

6.8 (较难) 定义函数

```
scale :: Picture -> Int -> Picture
```

它将输入图形放大到第二个参数的整数倍。例如, 如果exPic表示图形

```
#.#
..#
```

则scale exPic 2的结果是

```
##..##
##..##
....##
....##
```

如果第二个参数为 *0* 或负数, 则返回空图形。

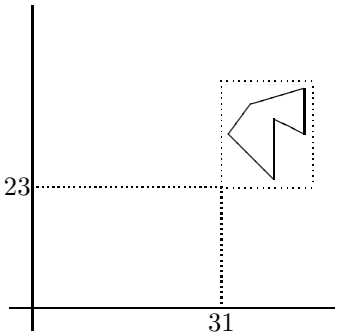


图 6.1: 一个图形的例子

§6.2 扩展的练习：有位置的图形

我们使用Picture建模的图形没有定位于空间中任何一点，我们可以将其想象为在纸上的图形，可以将其连接，叠加，旋转等。

一个不同的图形模型是给予每个图形一个空间位置，于是我们可以考虑移动图形，叠加图形等。

基本定义

如何表达定位的图形呢？首先我们需要考虑如何在整数网格上表示位置。一个位置是由一对整数表示的：

```
type Position = (Int,Int)
```

我们将使用Image表示定位的图形，即

```
type Image = (Picture, Position)
```

图 6.1 是一个例子，我们将 horse 的左下角或者 **引用点** 置于 (31, 23)。

本节的剩余部分是编写处理这些图形函数的练习。你可以使用前一章介绍的列表函数，也可以使用Picture上的已定义函数。

习题

6.9 定义函数

```
makeImage :: Picture -> Position -> Image
```

它将利用一个Image和一个Position构建一个图形。

6.10 定义函数

```
changePosition :: Image -> Position -> Image
```

它不改变Image的Picture，而将Position改为第二个参数给出的位置。

6.11 定义函数

```
moveImage :: Image -> Int -> Int -> Image
```

moveImage img xMove yMove的结果是将img在水平方向移动xMove，在垂直方向移动yMove。

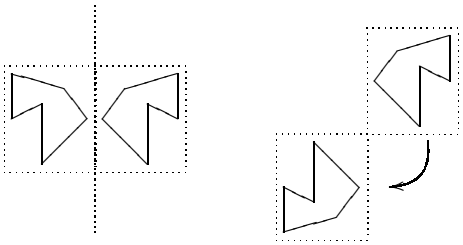


图 6.2: 几何观点的 flipV 和 rotate

6.12 定义函数printImage :: Image -> IO() 其功能与printPicture类似。

变换

我们可以将类型Picture上的变换扩展到类型Image上，不过我们需要考虑这些变换在位置上的作用。

一种将变换由Picture提升到Image的方法是简单地将图形置于原来的位置，称之为**单纯**的观点。如果我们考虑空间的反射和旋转，那么结果很可能像图 6.2 所示，其中变换后的图形的位置已经改变。旋转的重心是参考点，反射变换相对于通过参考点的水平线或者垂直线进行；一般地，这些变换将改变图形的参考点。称这样的变化为几何的观点。

习题

6.13 在单纯的观点下实现Image上的flipH, flipV, ratate 和rotate90等类似运算。

6.14 在几何观点下实现Image上的flipH, flipV, ratate 和rotate90等运算。

叠加

定位的图形叠加会更加复杂。如图 6.3.

我们看到一种叠加方法是利用Picture的叠加，首先要将两个图形的边界用白色扩展使其具有相同的尺寸与位置。

6.15 定义如图 *6.3* 所示，将一个Picture用白色扩展的函数。在开始实现之前，你需要仔细考虑这些函数的预期效果。你将需要函数参数以表示图形的上、下、左、右的白色扩充量。特别注意，扩展后图形的位置可能会变化.

6.16 利用边界扩展函数定义Image上的叠加函数。

6.17 如何利用Image上的叠加定义Image上的above和sideBySide?

§6.3 局部定义

在继续深入之前，我们需要进一步讨论函数定义。

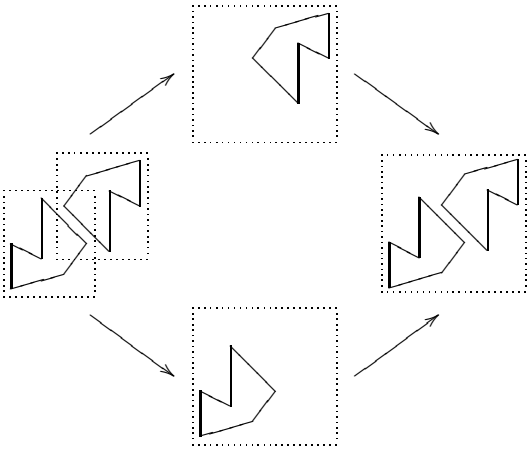


图 6.3: 两个图形的叠加

每个（条件）等式均可跟随几个 **局部** 于所定义的函数或对象的定义。这些局部定义写在关键字 where 之后。在讨论局部定义的可视范围、计算等之前，我们先看几个例子。

例 6.1 一个简单的例子是求两个数的平方和函数

```
sumSquares :: Int -> Int -> Int
```

函数的结果是两个值sqN和sqM之和，

```
sumSquares n m = sqN+sqM
```

这两个值的定义可置于跟随等式的where之后:

```
sumSquares n m
   =  sqN + sqM
      where
      sqN = n*n
      sqM = m*m
```

在这个简单例子中，或许很难看出局部定义的意义，但是，在实践中，有许多情况下使用局部定义使得一个解更易读，更有效。下面看这样的一个例子。

例 6.2 考虑下列函数

```
addPairwise :: [Int] -> [Int] -> [Int]
```

它将两个列表的相应元素相加，舍弃那些无伴的元素。例如

```
addPairwise[1, 7] [8, 4, 2] = [9, 11]
```

上述函数可用类似于 *6.1* 节中的sideBySide的方式定义，所以

```
addPairwise intlist1 intList2
  = [m+n |(m,n) <- zip intList1 intList2]
```

假如我们希望将无伴的元素添加到列表的尾部:

```
addPairwise' :: [Int] -> [Int] -> [Int]
```

例如，

```
addPairwise' [1,7] [8,4,2,67] = [9,11,2,67]
```

如何解决此问题呢？使用图 *5.1* 介绍的函数take 和drop，我们可以将参数列表分裂。如果minLength是两个列表的最短长度，那么结果列表的开始部分为

```
addPairwise (take minLength intList1)
            (take minLength intList2)
```

结果的剩余的部分呢？参数列表的剩余部分为

```
drop minLength intList1
drop minLength intList2
```

上面两个列表至少有一个为空列表，所以我们可以将这两个列表连接到一起，从而形成结果列表的剩余部分。因此，我们得到如下定义

```
addPairwise' intList1  intList2
   = front++rear
     where
     minLength = min(length intList1) (length intList2)
     front = addPairwise (take minLength intList1)
                         (take minLength intList2)
     rear = drop minLength intList1++ drop minLength intList2
```

在这里，我们在效率上有所收获，因为minLength只计算一次，尽管它在定义中使用了 *4* 次。另外，我们的定义更易读；定义说明，结果由两部分front和rear构成，我们可以分别阅读两个定义。事实上，我们可以进一步改进函数的效率，方法是用图 *5.1* 中的函数splitAt代替take和drop的多次调用。

```
addPairwise' intList1 intList2
   = front++rear
     where
     minLength = min(length intList1) (length intList2)
     front = addPairwise front1 front2
     rear  = rear1++rear2
     (front1, rear1) = splitAt minLength intList1
     (front2, rear2) = splitAt minLength intList2
```

在这个例子中，我们看到where子句的第三次应用。我们可以在等式的左边使用一个模式，如(front1,rear1)，其结果是将front1和rear1与等式右边表达式的相应分量匹配，即列表inList1分裂的结果。这种形式的模式称为 **一致的 (conformal)**，因为定义右边的表达式必须与左边的模式一致，否则定义会失败。另一点值得注意的是，不同定义的次序是无关紧要的，特别是，一个值在未定义之前可以被使用：front和rear的定义在它们使用的front1和rear1的定义之前。这个规则对脚本也是成立的，即顶层函数定义的次序是无关紧要的。

版面格式

在带有 where 子句的定义中，版面格式是很重要的，系统用边界规则确定 where 子句定义的结尾。where 子句必须在它所属的定义中，所以它必须位于它所属的定义起始位置的右边。在 where 子句中，同样的规则适用。所以，定义垂直对齐很重要，否则会引起错误。我们推荐的版面格式为

```
f p1 p2 ... pk
  | g1         = e1
   ...
  | otherwise  = er
   where
   v1 a1 ... an = r1
   v2 = r2
   ...
```

这里的 where 子句是附属于整个条件等式，也就是附属于条件等式的所有子句。这个例子也表明局部定义可以包含函数，这里v1是一个局部函数定义。类似于顶层函数有类型说明，同样，用 where 定义的对象也可以有类型说明。如果在上下文中难以确定一个局部定义的对象的类型，我们的习惯是添加其类型说明。

let 表达式

同样可以使定义局部于一个表达式。例如，

```
let x = 3+2 in x^2 +2*x -4
```

其结果是 31。如果同一行包含多个局部定义，应该用分号将它们隔开来，如

```
let x = 3+2; y = 5-1 in  x^2 + 2*x-y
```

我们将看到，这种形式只是偶尔会用到。

作用域

一个 Haslcell 脚本由一系列定义组成。一个定义的 **作用域** 是程序中可以使用此定义的那部分程序。Haskell 顶层函数的作用域是它们的定义所在的整个脚本，也就是说，脚本中的所有定义均可使用顶层定义，特别地，顶层定义可以在它们的定义之前被使用。例如，

```
isOdd, isEvea :: Int -> Bool

isOdd n
  | n <= 0    = False
  | otherwise = isEven (n-1)

isEven n
```

```
  | n<0       = False
  | n == 0    = True
  | otherwise = isOdd (n-1)
```

由 where 给出的局部定义是不期望在整个脚本中可见的，而只在它们出现的条件等式中可见。这也适用于函数定义中的变量；它们的作用域是它们所出现的整个条件等式。

特别地，在下例中sqx，sqy和sq以及变量 x 和 y 的定义的作用域用大方框表示，变量 z 的作用域用小方框表示。

```
maxsq x y
    | sqx > sqy   = sqx
    | otherwise   = sqy
      where
       sqx = sq x
       sqy = sq y
       sq ::  Int -> Int
       sq z = z*z
```

要特别注意下列几点

- 出现在函数定义左边的变量 (如上面的 x和 y) 可用于局部定义；在上例中，x和 y在sqx和sqy中被使用；
- 局部定义可以在定义之前使用；如 sq 在 sqx 中使用；
- 局部定义可以在结果、守卫和其他局部定义中使用。

一个脚本可以包含两个同名的定义或变量。在下例中，变量 x出现两次。那么在各个位置哪一个定义起作用呢？

最局部的变量是被使用的变量。

```
maxsq x y
  | sq x > sq y  = sq x
  | otherwise    = sq y
      where
      sq x = x*x
```

在这个例子中，可以想象成小方框在大方框内开了一个洞，所以外层 x的作用域排除了sq的定义。当一个定义包含在另一个定义中时，最好的办法是内层定义使用不同的变量或者命名，除非有特别的理由使用同样的名。最后注意，在同一层不可以有同名的重复定义。如果因为模块的组合出现同名冲突，其中的一个应被隐蔽起来。

计算

前面关于计算的书写方式可扩展到 where 子句。例如，前节的sumSquares函数一例：

```
sumSquares 4 3
  = sqN+sqM
```

```
  | where
  | sqN = 4*4 = 16
  | sqM = 3*3 = 9
  = 16+9
  = 25
```

在 where 之下的局部定义的值在需要时被计算，在 where 之下的所有局部定义右缩格书写。要查看顶层值，只需查看左边的计算。其中的垂线用于连接有 where 中间计算结果的连续计算步骤。这些垂线可以省略。

习题

6.18 定义函数`maxThreeOccurs :: Int -> Int -> Int -> (Int,Int)`它返回三个整数中的最大者及最大者出现的次数。一个自然的解是找出最大值，然后检查它在三个整数中出现的次数。如果不允许使用`where`定义，你如何解决此问题？

6.19 利用你前题的解给出下列计算例子

```
maxThreeOccurs 4 5 5
maxThreeOccurs 4 5 4
```

§6.4 扩展练习：超市帐单

这一组练习讨论超市帐单[1]。练习的用意是利用第五章介绍的列表处理技术。特别地，我们将使用那里介绍的列表概括和引导库函数。同时，我们也将在适当的地方使用 6.3 节介绍的局部定义。

问题的提出

超市收银台的条码扫描仪将由一篮商品生成一个条码列表，如

```
[1234,4719,3818,1112,1113,1234]
```

这个条码列表必须被转换为一个帐单

```
        Haskell Stores


Dry Sherry, 1lt .............5.40
Fish Fingers ................1.21
Orange Jelly ................0.56
Hula Hoops (Giant) ..........1.33
Unknown Item ................0.00
Dry Sherry, 1lt .............5.40
```

[1]我非常感激澳大利亚新南威尔斯大学计算机系的 Peter LindSay 等，是他们的讲义激发我编写了这个例子

```
Total .....................13.90
```

我们必须首先决定如何表示所涉及的对象。条码和价格（便士）可用整数表示；商品名可用串表示。因此，我们有

```
type Name    = String
type Price   = Int
type BarCode = Int
```

条码到帐单的转换将基于一个条码、名称和价格构成的数据库。像图书馆一例一样，我们用一个列表描述这个关系

```
type Database = [(BarCode,Name,Price)]
```

我们将使用的数据库例为

```
codeIndex :: Database
codeIndex = [  (4719, "Fish Fingers", 121),
               (5643, "Nappies", 1010),
               (3814, "Orange Jelly", 56),
               (1111, "Hula Hoops", 21),
               (1112, "Hula Hoops(Giant)", 133),
               (1234, "Dry Sherry, 1lt", 540)]
```

程序的目标是将一个条码列表转换为二元组(Name,Price)的列表，然后将此列表转换为可以打印出上面形式帐单的串。我们定义如下类型

```
type TillType = [BarCode]
type BillType = [(Name,Price)]
```

我们希望定义的函数为

```
makeBill :: TillType -> BillType
```

此函数将一个条码列表转换为一个商品和价格二元组列表。下列函数

```
formatBill :: BillType -> String
```

将把品名/价格二元组列表转换为格式化的帐单。函数

```
produceBill :: TillType -> String
```

将结合两个函数makeBill和formatBill的功能：

```
produceBill = formatBill . makeBill
```

帐单每行的长度确定为 30。这是一个常数，因此

```
lineLength  :: Int
lineLength = 30
```

以这种方式定义lingLength为常数后，如果需要改变一行的长度，我们只需修改一个定义，但是，如果在每个格式化函数中都使用 30，那么我们不得不在使用 30 的所有地方做相应的修改。随后的练习将完成脚本的剩余部分。

格式化帐单

我们首先自底向上定义函数formatBill：设计格式化价格、格式化各行和格式化总计的函数，然后使用这些函数构造函数formatBill。

习题

6.20 给定便士数，如 *1023*，则磅和便士部分可由下列方式得到：1023‘div‘100为磅数，1023 ‘mod‘ 100为便士数。利用这个事实和show函数，定义下列函数

```
formatPence :: Price -> String
```

使得formatPence 1023 = "10.23"。要特别注意像"10.23"这样的情况。

6.21 利用函数formatPence定义下列函数

```
formatLine :: (Name, Price) -> String
```

它将格式化帐单的每一行，如

```
formatLine("Dry Sherry,1lt",540)
        = "Dry Sherry,1lt ...... 5.40\n"
```

这里'n'是转行字符，*++* 可用于连接两个串，*length* 给出串的长度。你或许会发现函数 *replicate* 有帮助。

6.22 利用函数formatLine定义formatLines :: [(Name,Price)] 它将formatLine应用于每个二元组(Name, Price)，并将结果连接到一起。

6.23 定义函数makeTotal :: BillType -> Price 它将计算一个(Name, Price)二元组列表的价格的总价。例如formatTotal[("...",540),("...",121)] = 661

6.24 定义函数

```
formatTotal :: Price -> String
```

使得其功能如下例

```
formatTotal 661 = "\nTotal.....................6.61"
```

6.25 利用函数formatLines,makeTotal和formatTotal定义

```
formatBill :: BillType -> String
```

使得对于下列输入，函数输出本节开始的帐单格式：

```
[("Dry Sherry,11t",540),("Fish Fingers,121),
 ("Orange Jelly",56),
 ("Hula Hoops(Giant)",133), ("Unkonw Item",0),
 ("Dry Sherry,11t",540)]
```

制作帐单：条码列表转换为品名和价格

我们必须查询数据库才能实现条码列表到品名和价格的转换

6.26 定义函数look :: Database -> BarCode -> (Name, Price) 它将返回数据库中对应于BarCode的二元组(Name, Price)。如果数据库中不存在BarCode，则返回("Unkonw Item",0)。提示：利用图书馆数据库的思想，你可能发现你

```
        Haskell Stores

  Dry Sherry, 1lt .............5.40
  Fish Fingers ................1.21
  Orange Jelly ................0.56
  Hula Hoops (Giant) ..........1.33
  Unknown Item ................0.00
  Dry Sherry, 1lt .............5.40

  Discount ....................1.00

  Total .......................12.90
```

图 6.4: 带折扣的账单

得到一个(Name, Price)的列表，而不是一个二元组。你可以假设数据库中每个条码只出现一次，所以你可以取列表的头元素（如果列表不空）。

6.27 定义一个函数lookup :: BarCode -> (Name, Price) 它利用look 在一个特定的数据库codeIndex中查找一个商品。此函数与引导库中的同名函数发生冲突，参阅 *49* 页找出解决问题的方法。

6.28 定义函数

```
makeBill :: TillType -> BillType
```

它将lookup应用于输入列表中的每个元素。例如，当lookup应用于

```
[1234,4719,3814,1112,1113,1234]
```

时，结果将是习题 *6.25* 中的二元组列表。注意1113在codeIndex中不存在，故转换为("Unkonw Item",0)。至此，我们完成了makeBill的定义，再利用formatBill，转换程序得以实现。

扩展的问题

最后给出一些进一步的练习。

6.29 顾客买多瓶Sherry时应有折扣：每两瓶折扣为 *1.00* 磅。对于条码列表

```
[1234,4719,3814,1112,1113,1234]
```

则帐单应像图 *6.4* 那样打印。你或许需要定义函数

```
makeDiscount   :: TillType -> Int
formatDiscount :: Int -> String
```

然后利用它们重新定义函数

```
formatBill :: TillType -> String
```

6.30 定义修改条码数据库的函数。你需要定义一个函数实现在数据库中添加一个条码和一个(Name, Price)二元组，同时删除任何在库中已存在的对此

条码的引用项。

6.31 重新定义你的系统使得库中不存在的条码在帐单中没有对应项。至少有两种处理办法。

- 保持makeBill不变，修改格式化函数
- 修改makeBill函数，删除"unknown Item"项。

6.32 (项目) 设计一个分析一组销售的程序。给定一个TillType列表，生成统计每种商品销售量的统计表。你还可以分析帐单，记录哪些商品是一起购买的，以便于超市组织货架。

小结

本章介绍了局部定义的概念，最重要的是附属于函数定义中条件等式的 where 子句。我们演示了使用局部定义使函数定义更易读和避免重复计算的方法。例如，addPairwise'中的minLength。我们给出了包含守卫和 where 子句的函数定义的一般模式。

我们还看到列表概括与引导库函数的结合提供了一个强有力的工具箱，由此可以构造特定列表类型上的函数。这一点在 Picture 一例以及超市实例中得以显示。同时我们有机会看到如何用一组相关函数来构造一个较长的程序。

第七章　定义列表上的函数

我们已经看到如何使用列表概括和引导库的列表处理函数相结合的方法定义列表上的函数。本章察看其中的原理, 并解释如何利用递归定义列表上的函数。我们可以使用递归定义其他引导库函数。我们还将看到递归的更广泛应用，包括排序和一个文字处理程序。我们先介绍模式匹配机制，并且给出递归工作原理的解析，以响应第四章的讨论；然后讨论使用原始递归和一般递归定义的各种函数例子；最后以上面所述实例研究结束本章。

§7.1　再谈模式匹配

我们已经看到，可以用条件等式定义函数，如

```
mystery :: Int -> Int -> Int
mystery x y
  | x==0      = y
  | otherwise = x
```

这里使用了守卫来选择结果；我们可以将上述定义表示为下列等式：

```
mystery 0 y = y                                    (mystery.1)
mystery x y = x                                    (mystery.2)
```

这里使用了一个模式 – 原子 0 – 来区分两种情况。像守卫一样，系统按顺序应用两个等式，只有在不能应该(mystery.1)时，系统才应用(mystery.2)。上述定义的另一个特点是 y在(mystery.2)中不出现。为此，在这种情况下，我们不需命名第二个参数。可以用一个与任何参数匹配的 **通配符**'_' 代替之。

```
mystery 0 y = y
mystery x _ = x
```

由此我们看到, 模式匹配可用于函数定义中区分不同的情形。我们还看到模式匹配可用于命名元组的分量，如

```
joinStrings :: (String, String) -> String
joinStrings (st1,st2) = st1++"\t" ++st2
```

其中变量st1与st2将分别与参数的第一个和第二个分量匹配。在下一节我们会看到，对于列表，区分不同的情形和抽取分量常常会共同使用。

模式小结

一个模式可以是

- 一个**原子值**, 如 24, 'f' 或者 True; 一个参数等于这个值时，参数便与模式匹配。
- 一个**变量**, 如 x或longVariableName; 任何参数值与变量模式匹配。
- 一个**通匹符**'_'; 任何参数值与通匹符匹配；

- 一个**元组模式**($P_1, P_2, ..., P_n$)。一个参数值必须具有形式 ($v_1, v_2, ..., v_n$), 并且每个 v_k与相应 P_k匹配时, 参数值与元组模式匹配。
- 一个**构造符**应用于某些模式; 我们将在下一节和第十四章讨论这种情况。

在有多个条件等式的函数定义中, 每个等式的左边是函数作用于一系列模式。当一个函数被引用时, 我们用参数值按顺序与这一系列等式左边的模式匹配, 并选择第一个匹配成功的等式对应的结果。所以, 在 Haskell 中, 模式匹配是**顺序的**, 这与检查守卫所表示的条件类似。

§7.2 列表与列表模式

每个列表或者为**空**, `[]`, 或者是非空的。对于后者, 如`[4,2,3]`, 列表可表示为形如 `x:xs`, 其中 x 是列表的第一个元素, xs 是列表的剩余部分; 对于上例, 列表可表示为 `4:[2,3]`。我们称4为列表的**头**,`[2,3]`为列表的**尾**。

另外, 每个列表可由空列表重复应用`':'`而构成, 事实上, 这也是 Haskell 列表的内部表示法。上例列表可看作如下逐步构成的:

```
[]    3:[] = [3]     2:[3] = [2,3]     4:[2,3] = [4,2,3]
```

我们还可以重复使用`':'`来表示列表, 如`4:2:3:[]` 注意, `':'`是**右结合的**, 所以对任意 x , y 和 zs, `x:y:zs =x:(y:xs)`。

不难看出, `4:2:3:[]`是使用`':'`构造`[4,2,3]`的唯一方法。因此, 具有类型`a -> [a] -> [a]` 的运算符`':'`对于列表具有特殊的功能: 它是一个 **构造符**, 因为每个列表可由 `[]`和`':'`唯一地表示。由于历史的原因, 我们有时称这个构造符为**cons**。并非所有的函数都是构造符: ++ 可用于构造列表, 但这种构造不具有唯一性, 如

```
[1]++[2,3]=[1,2,3]=[1,2]++[3]
```

模式匹配定义

如果一个定义适用于列表的所有情况, 我们可以用下列形式的定义

```
fun xs = ...
```

但是, 更普遍的情形是我们需要区分空和非空列表两种情形, 如引导库函数

```
head     :: [a] -> a
head (x:_)  = x

tail     :: [a] -> [a]
tail (_:xs) = xs

null     :: [a] -> Bool
null []     = True
null (_:_)  = False
```

这里 head 取非空列表的第一个元素，tail 取非空列表除第一个元素外的剩余部分，null 检查一个列表是否为空。在 null 的定义中，模式 (_:_) 与任何非空列表匹配，但是它没有给予表头和表尾任何名；如果我们需要命名它们，可以使用另一种模式，如在 tail 的定义中，模式 (_:xs) 给予列表的尾一个名 xs。

在 Haskell 中习惯使用 xs 和 ys（读作"exes"，"whyes"）等作为列表变量，用 x, y 等作为列表元素变量。在使用短变量名时，我们将遵守这个习惯。

现在可以解释最后一种模式了。列表上的一个 **构造符模式** 或者是[]或者形如(p:ps)，其中 p 和 ps 本身也是模式。

- 当一个列表为空时，列表与[]匹配；
- 当一个列表不空时，并且其头与模式 p 匹配，尾与 ps 匹配时，此列表与模式 p:ps 匹配.

如果模式为 x:xs，则任何非空列表与之匹配；列表的头与 x 匹配，尾与 xs 匹配。一个包含构造符(:)的模式需要用括号括起来，因为函数应用的结合力强于任何其它运算。

case 结构

前面介绍了如何在函数的参数上进行模式匹配；有时我们希望在其他值上进行模式匹配。case 表达式可用于这样的目的，下面看一个例子。

假设我们要找出串st中的第一个数字，如果找不到数字，则返回 '\0'。我们可以使用 5.5 节的函数 digits，返回串中的所有数字构成的列表digits st。如果此列表不空，即它与 (x:_) 匹配，则 x 便是我们求的结果，如果列表空，则返回 '\0'。

可见，我们希望对(digits st)的值进行模式匹配。我们可以用 case 表达式来进行：

```
firstDigit :: String -> Char
firstDigit st
  = case (digits st) of
        []     -> '\0'
        (x: _) -> x
```

一个 case 表达式可用于区分不同的分支选择，在这里是空列表和非空列表；还可以通过带变量的模式与值的匹配提取值中的组成部分。将表达式e与 (x:_) 匹配，x 与e的头相关联；因为我们使用了通配符模式，e的尾不与任何变量关联。

一般地，case 表达式具有下列形式

```
case e of
   p1 -> e1
   p2 -> e2
   ...
```

```
    pk -> ek
```

其中e是将与模式p1,p2, ..., pk顺序匹配的表达式。如果pi是与e匹配的第一个模式，则ei便是结果，其中pi所含的变量与e的相应部分关联。

习题

7.1 利用模式匹配定义一个返回列表第一个整数加一的函数，如果列表不包含任何元素，则返回 *0*。

7.2 使用模式定义下列函数；如果一个列表至少包含两个整数，则返回前两个整数之和；如果列表包含一个整数，则返回列表的头，否则返回 *0*。

7.3 不使用模式匹配定义上述函数。

§7.3 列表上的原始递归

假设我们要计算一个整数列表的和。像 4.2 节计算阶乘一样，我们可以设想把 sum 的值列一个表：

```
          sum [] = 0
...       sum [5] = 5    ...
...       sum [7,5] = 12    ...
...       sum [2,7,5] = 14    ...
...       sum [3,2,7,5] = 17    ...
...
```

如阶乘一样，此表可以通过说明第一行以及如何由一行计算下一行来描述：

```
sum :: [Int] -> Int
sum []     = 0                              (sum.1)
sum (x:xs) = x + sum xs                     (sum.2)
```

sum的上述定义称为 **列表上的原始递归**。在这样的定义中，我们给出

- 一个开始点:sum在 []上的值，以及
- 由sum在一个特定点的值sum xs得到另一行的值，即sum (x:xs)的方法。

如 4.2 节，我们可以利用计算过程解释递归的工作原理。考虑sum[3,2,7,5]的计算过程，重复使用(sum.2)可得到

```
sum [3,2,7,5]
⇝ 3 + sum [2,7,5]
⇝ 3 + (2 + sum [7,5])
⇝ 3 + (2 + (7 + sum [5]))
⇝ 3 + (2 + (7 +(5 + sum [])))
```

现在可以使用等式sum.1和整数的算术运算，得出

```
⇝ 3 + (2+ (7+ (5+0)))
⇝ 17
```

可以看出，定义sum的递归可以给出任意有穷列表的和，因为每个递归步使我们更接近递归基，即sum作用于 []。

在下一节我们将看到更多原始递归定义的例子。

习题

7.4 定义函数product :: [Int] -> Int，它返回整数列表中所有整数的乘积；当列表空时返回 *1*。为什么选择 *1* 作为空列表上的值呢？

7.5 定义函数and，or :: [Bool] -> Bool 它们分别返回布尔值列表的合取与析取。例如

```
and [False,True] = False
or  [False,True] = True
```

在空列表上，and返回 True，or返回 False，请解释这些选择的理由。

§7.4 寻找原始递归定义

前一节对原始递归的工作原理给出了两种解释：函数表格和计算函数的值。本节介绍一系列列表上原始递归定义的例子。列表上原始递归定义的模板是

```
fun []     = ...
fun (x:xs) = ... x...xs...fun xs...
```

找到原始递归定义的关键问题是:

假如我们知道fun xs的值，如何由此定义fun (x:xs)呢？

我们将通过一系列例子说明之。

例 7.1 许多函数和sum类似，可以通过“折叠”一个运算而得到。引导库的函数product,and 和or均为这样的例子。下面我们看如何定义引导库函数concat。

```
concat :: [[a]] -> [a]                          (concat.0)
```

其结果为

```
concat  [e1,e2,...,en]= e1++e2++...++en
```

定义可以如下开始

```
concat  []    = []
concat (x:xs) = ...
```

如果我们知道concat xs，如何求concat (x:xs)呢？检查当 x:xs 为 $[e_1, e_2, ..., e_n]$的情形。concat xs的值为e2++...++en，而我们想要的结果为e1++e2++...++en，所以，我们可以将 x 连接到已连接在一起的列表之前，由此得到下列定义

```
concat  []    = []                              (concat.1)
concat (x:xs) = x ++ concat xs                  (concat.2)
```

由定义看出，x:xs 是列表的列表，因为在(concat.2)中，它的元素与另一个列表相连接；x 的类型将是结果的类型。由此可断定，输入的类型为 *[[a]]*，输出的类型为 *[a]*；这与(concat.0)给定的类型相符。

例 7.2 前一节使用的函数 *++* 又是如何定义的呢？能用原始递归定义 *++* 吗？一种方法是查看一些例子，例如，x为2，xs为[3,4]，则

```
[2,3,4]++[9,8] = [2,3,4,9,8]
[3,4]++[9,8]   = [3,4,9,8]
```

可见，将2置于[3,4]++[9,8]之前，便可得到[2,3,4]++[9,8]。当第一列表为空时，[]++[9,8]=[9,8]。由这些例子可得如下定义

```
(++) :: [a] -> [a] -> [a]
[] ++ ys     = ys
(x:xs) ++ ys = x : (xs++ys)
```

注意，*++* 的类型允许连接任意列表类型，只要这两个列表类型相同即可。

例 7.3 第三个例子是检查一个整数是否一个整数列表的元素。

```
elem :: Int -> [Int] -> Bool
```

显然，任何元素均不是 []的元素。但是，在什么情况下 x是 y:ys 的元素呢？如果你不知如何回答此问题，那么先看一二个例子。返回刚才的问题，因为 y:ys 是在 ys之前加上 y而得，如果 x是 y:ys 的一个元素，则

- x与 y相等，或者
- x是 ys的一个元素。

在第二种情形，我们使用了值elem x ys，由此得到elem的如下原始递归定义

```
elem x []     = False                          (elem.1)
elem x (y:ys) = (x == y)|| (elem x ys )        (elem.2)
```

注 10 模式中的重复变量 elem的另一个候选定义是

```
elem x (x:ys)= True                            (elem.3)
elem x (y:ys)= elem x ys
```

在这里检查相等在(elem.3)的左边进行。不幸的是Haskell不允许在模式中使用重复变量名。

例 7.4 假如我们欲将整数列表中的所有元素加倍

```
doubleAll :: [Int] -> [Int]
```

最简洁的解是使用列表概括

```
doubleAll xs =[2*x | x <- xs]
```

我们也可以考虑是否能利用原始递归完成定义。看几个例子

```
doubleAll [2,3]   = [4,6]
doubleAll [4,2,3] = [8,4,6]
```

可见，要给 x:xs 的元素加倍，需要给 *xs* 的元素加倍，然后在此列表前加上2*x即可，故此有下列定义

```
doubleAll []     = []                      (doubleAll.1)
doubleAll (x:xs) = 2*x : doubleAll xs      (doubleAll.2)
```

例 7.5 假设我们要从一个整数列表中选出偶数元素

```
selectEven :: [Int] -> [Int]
```

这个函数可以用列表概括实现

```
selectEven xs = [x | x <- xs, isEven x]
```

可否用原始递归定义此函数呢？对于空列表，结果仍为空列表

```
selectEven [] = []                              (selectEven.1)
```

对非空列表如何处理呢？考虑下列例子

```
selectEven [2,3,4] = [2,4] = 2 : selectEven [3,4]
selectEven [5,3,4] = [4]   =     selectEven [3,4)
```

可见，结果是在selectEven xs上根据 x的奇偶性决定是否加上 x。由此可定义

```
selectEven (x:xs)
  |isEven x  = x : selectEven xs              (selectEven.2)
  |otherwise = selectEven xs
```

例 7.6 最后一个例子。假设我们要将一个数的列表按递增序排序，例如将列表[7,3,9,2]排序。一种方法是先将尾部排序为[2,3,9] 然后将头7插入列表中适当的位置，最后结果为[2,3,7,9]。下面给出这种程序iSort的定义 *(*其中’ i’表示 **插入** 排序*)*

```
iSort :: [Int] -> [Int]
iSort []     = []                                (iSort.1)
iSort (x:xs) = ins x (iSort xs)                  (iSort.2)
```

这是自顶向下（见4.1节）定义的典型例子。我们假定有ins的定义，然后定义了iSort。程序的开发分成了两部分：利用一个简单函数ins定义iSort和函数ins本身的定义。解决两个子问题比解决原问题来得简单。现在我们需要定义

```
ins :: Int  -> [Int] -> [Int]
```

要了解如何定义此函数，先看几个例子。前面给出了将7插入2,3,9的结果，下面是将1插入同一列表的结果[1,2,3,9]。由这两个例子看出

- 对于1的情况，插入的元素不大于列表的头元素，则将插入元素添加到列表的前面；
- 对于7的情况，插入的元素大于列表的头元素，则将其插入到列表的尾部，并将列表的头元素添加到插入结果之前：

  ```
  2 : [3,7,9]
  ```

  由此可得下列函数定义

```
ins x []    = [x]                                (ins.1)
ins x (y:ys)
 | x<= y      = x : (y : ys)                     (ins.2)
 | otherwise  = y : ins x ys                     (ins.3)
```

下面是计算iSort[3,9,2]的过程

```
iSort [3,9,2]
⇝ ins 3 (isort [9,2])                by (iSort.2)
⇝ ins 3 (ins 9 (iSort [2]))          by (iSort.2)
⇝ ins 3 (ins 9 (ins 2 (iSort [])))   by (iSort.2)
⇝ ins 3 (ins 9 (ins 2 []))           by (iSort.1)
⇝ ins 3 (ins 9 [2])                    by (ins.1)
⇝ ins 3 (2:ins 9 [])                   by (ins.2)
⇝ ins 3 [2,9]                          by (ins.1)
⇝ 2:ins 3 [9]                          by (ins.3)
⇝ 2:[3,9]                              by (ins.2)
⇝ [2,3,9]
```

上述函数的开发过程显示了在设法定义函数时查看一些例子的优点；通过查看例子我们可以看出如何将问题分情形处理或者递归的模式是什么样的等。我们还看到自顶向下的设计可以将一个大问题分解成更容易解决的小问题。

下一节我们将介绍一般形式的递归。

习题

7.6 利用列表上的原始递归定义函数elemNum :: Int -> [Int] -> [Int]使elemNum x xs返回 x在 xs中出现的次数。你能利用列表概括和预定义函数而不使用原始递归定义elemNum吗？

7.7 定义函数unique :: [Int] -> [Int] 使得unique xs返回 xs中所有仅出现一次的元素。例如, unique [4,2,1,3,2,3]的结果是[4,1] 。可以考虑两种解法: 使用列表概括和不使用列表概括。

7.8 写出引导库函数reverse和unzip的原始递归定义。

7.9 你能使用iSort找出一个数的列表中的最小元素和最大元素吗？如果不使用iSort, 如何找到这些元素？

7.10 设计函数ins的测试数据。你的测试数据应该考虑到不同的插入点, 以及所有特殊情况。

7.11 修改ins的定义可以改变排序函数iSort的功能。用下列两种不同的方法重新定义ins:

- 列表以降序排列
- 重复元素被删除, 如iSort [2,1,4,1,2]=[1,2,4,]

7.12 设计删除重复元素排序函数iSort的测试数据, 解释你的选择。

7.13 通过修改ins和iSort定义一个对二元组列表排序的函数。顺序为 **字典序**。先比较二元组的第一个分量, 当第一个小量相同时, 才比较第二个分量。例如(2,73)小于(3,0), 而(3,0)小于(3,2)。

§7.5 列表上的一般递归

如 4.4 节所述，一个函数的递归定义不一定总使用函数在列表尾上的值；对任何更简单的列表上的递归调用都是合法的；因此在列表上的递归可以有多种形式。使用递归在列表上定义函数时，我们需要回答这样的问题：

在定义f (x:xs)时，哪些值f ys有助于f (x:xs)的计算?

例 7.7 在两个参数上同时使用递归是可能的。引导库函数zip便是一个例子。

```
zip :: [a] -> [b] -> [(a,b)]
```

例如

```
zip [1,5] ['c', 'd']     = [(1,'c'), (5,'d')]
zip [1,5] ['c', 'd','e'] = [(1,'c'), (5,'d')]
```

如果两个列表均不空，则由它们的头元素构成一个二元组，然后zip它的尾:

```
zip (x:xs) (y:ys) = (x,y) : zip xs ys          (zip.1)
```

但对于其他的情况，即至少有一个是空列表时，结果是空：

```
zip _ _  =  []                                 (zip.2)
```

我们还可以将(zip.2)用显式模式表示

```
zip (x:xs) (y:ys)  =  (x,y):zip xs ys
zip (x:xs) []      = []
zip [] ys          = []
```

在zip的第二个定义中，三种不同的情形分别用三个等式定义。下面是使用第一个定义计算的例子

```
zip [1,5] ['c','d','e']
⇝ (1,'c'):  zip [5] ['d','e']      by (zip.1)
⇝ (1,'c'):(5,'d') :zip [] ['e']    by (zip.1)
⇝ (1,'c'):(5,'d'):[]               by (zip.2)
⇝ (1,'c'):[(5,'d')]                by defn of :
⇝ [(1,'c'),(5,'d')]                by defn of :
```

注意，我们使用了':' 的右结合律。

例 7.8 函数take用于在一个列表中取出给定数目的元素。例如

```
take 5  "Hot Rats" = "Hot R"
take 15 "Hot Rats" = "Hot Rats"
```

在此例中，我们在整数和列表上进行递归

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

对于特殊的情形，当整数为 *0* 或列表为空时，我们有

```
take 0 _  = []                                 (take.1)
take _ [] = []                                 (take.2)
```

一般情形呢？也就是说，当列表非空，整数不为 *0* 的情况呢？我们可以在列表的尾部取 *n-1* 个元素，然后把头置于其前，即

```
take n (x:xs)
   |n >0     = x : take (n-1) xs                    (take.3)
```

对于其他的情况，我们给出错误信息

```
take _ _ = error "PreludeList.take: negative argument"   (take.4)
```

例 7.9 最后一个例子是列表的另一种方排序方法。**快速排序** 算法是通过对排序的两次递归调用实现的。例如，对下面列表排序

```
[4,2,7,1,4,5,6]
```

我们取出头元素4，然后将其余的部分2,7,1,4,5,6分成两部分

```
       [1,2,4]           [5,6,7]
```

第一部分包含不大于4的元素，第二部分包含大于4的元素。然后对这两部分排序，得出

```
       [1,2,4]           [5,6,7]
```

最后的排序结果为

```
       [1,2,4]++[4]++[5,6,7]
```

下面是排序算法的定义

```
qSort :: [Int] -> [Int]
qSort []      = []                                (qSort.1)
qSort (x: xs)
   = qSort [y|y <- xs, y<=x] ++[x]++ qSort[y|y <- xs,y>x]
                                                  (qSort.2)
```

可以看出，快速排序算法的定义与它的非正式描述是何等地接近，这种表现力是函数式程序的优点之一。上述递归对于每个有穷列表均可给出答案，因为qSort的两个递归调用的参数为xs的子列表，它们均小于 x:xs。

在第十九章，我们将介绍各种算法的效率，并证明在一般情况下快速排序优于插入排序。在下一节我们将介绍一个利用一般递归的大型例子。

习题

7.14 以函数take为例，定义引导库函数drop和splitAt。

7.15 根据函数take的定义，take (-3) []的值是什么？如何修改定义使得当整数参数为负数时，函数给出错误信息。

7.16 如何定义函数zip3用于zip三个列表？分别用递归和zip给出两种定义。这两种定义各有什么优缺点？

7.17 如何修改qSort将列表按递减序排列？如何确保qSort删除重复元素。

7.18 一个列表是另一个列表的子列表，如果第一个列表的所有元素都是第二个列表的元素，且出现的顺序相同。例如，"ship"是 "Fish&Chips" 的子列表，而不是"hippies"的子列表。一个列表是另一个列表的子序列，如果第一个列表是第二个列表中的某一段元素。例如"Chip"是 "Fish&Chips" 的子序列，而不是"Chin up"的子序列。定义判断一个串是否为另一个串的子列表或

者子序列的函数。

§7.6 例子：文本处理

在文本处理系统中，我们习惯上把各行添满后自动中断，以利于文本显示的美观。本书也不例外。下列形式的输入

```
The heat bloomed      in December
  as the   carnival   season
             kicked into gear.
Nearly helpless with sun and glare, I avoided Rio's brilliant
sidewalks
  and glittering breaches,
panting in dark  corners and waiting out the inverted southern
summer.
```

将各行添满后转换为

```
The heat bloomed in December as the
carnival season kicked into gear.
Nearly helpless with sun and glare,
I avoided Rio's brilliant sidewalks
and glittering breaches, panting in
dark corners and waiting out the
inverted southern summer.
```

为了使左右对齐，在除最后一行的各行字间加上额外的空格；

```
The heat bloomed in December as the
carnival  season  kicked into gear.
Nearly helpless with sun and glare,
I avoided Rio's brilliant sidewalks
and glittering breaches, panting in
dark  corners  and  waiting out the
inverted southern summer.
```

Haskell 的一个输入文件可视为一个字符串，所以字符串处理函数在这里具有重要的作用。另外，因为串也是列表，本例可看成一般列表函数的练习。

整体方案

本节将给出一个自底向上的程序开发例子：我们首先考虑解决问题所需的组成部分，而不是将问题自顶向下分解。

处理文本的第一步是将输入串划分为**词**，并舍弃所有空白；然后将词组织成所需长度的文本行。这些行中可以加入空格以调节文本。为此，我们先看如何将文本分解为词。

抽取词

首先，给定一个字符串，如何定义一个函数从串的前面抽取第一个词。

一个词是不包含**whitespace(空白字符)**如空格、制表符和换行符的字母序列：

```
whitespace =['\n','\t',' ']
```

在定义getWord 时，我们将使用检查一个元素是否一个列表的元素的标准函数elem。例如elem 'a' whitespace为 False。

为了定义此函数，考虑下列例子：

- getWord " boo"应为""，因为第一个字符为空白;
- getWord "cat dog"应为"cat"。我们将'c' 置于"at"之前，也即getWord "at dog"之前，即可得到所需的词。

由此得到下列定义

```
getWord :: String  -> String                    (getWord.1)
getWord [] = []
getWord (x:xs)
 | elem x whitespace = []                       (getWord.2)
 | otherwise         = x : getWord xs           (getWord.3)
```

考虑下例

```
getWord "cat dog"
⇝ 'c':getWord "at dog"          by (getWord.3)
⇝ 'c':'a':getWord "t dog"       by (getWord.3)
⇝ 'c':'a':'t':getWord " dog"    by (getWord.3)
⇝ 'c':'a':'t':[]                by (getWord.2)
⇝ "cat"
```

用类似的方法，一个串中的第一个词可以被舍弃

```
dropWord :: String -> String
dropWord []  = []
dropWord (x:xs)
 | elem x whitespace = (x:xs)
 | otherwise         = dropWord xs
```

容易验证 dropWord "cat dog"=" dog"。我们将利用getWord 与dropWord将串分解成组成串的词。注意，在串" dog"中取词之前，我们应该先删除前面的空白字符。此功能由函数dropSpace实现：

```
dropSpace :: String -> String
dropSpace [] = []
dropSpace (x:xs)
 | elem x whitespace = dropSpace xs
 | otherwise         = (x:xs)
```

如何将一个串st分解为词呢？假设st的开始无空白字符

- 输出的第一个词由getWord作用于st而得；
- 其余的词将由去掉第一个词及其后面的空格后分解而得：即dropSpace (dropWord st)。

顶层函数splitWord首先删除串开始的空格，然后调用split。

```
type Word = String

splitWord :: String -> [Word]
splitWord st = split (dropSpace st)

split  ::  String  -> [Word]
split [] = []
split st
 = (getWord st) : split (dropSpace (dropWord st))
```

考虑一个简单例子

```
splitWords " dog cat"
⇝ split "dog cat"
⇝ (getWord "dog cat"):split (dropSpace (dropWord "dog cat"))
⇝ "dog":split (dropSpace " cat")
⇝ "dog":split "cat"
⇝ "dog":(getWord "cat"):split (dropSpace (dropWord "cat"))
⇝ "dog":"cat":split (dropSpace [])
⇝ "dog":"cat":split []
⇝ "dog":"cat":[]
⇝ ["dog","cat"]
```

分解为文本行

现在考虑如何将词的列表分解为文本行。同前面的讨论类似，我们先看如何从词的列表中取出第一行。

```
type Line = [Word]

getLine :: Int -> [Word] -> Line
```

getLine带有两个参数：第一个参数是行的长度，第二个是从中提取一行的词列表。定义使用 length 求一个列表的长度。函数的定义分三种情况：

- 如果列表中没有任何词，则结果行是空行。
- 如果第一个词是 w，并且此行有足够的空间给 w；它的长度length w不能大于行的长度len。此行的剩余部分由列表的剩余部分并且长为len - (length w+1)的行组成。
- 如果第一个词大于行的长度，则行为空。

```
getLine len [] = []
getLine len (w:ws)
 | length w <= len    = w : restOfLine
 | otherwise          = []
   where
   newlen     = len - (length w+1)
   restOfLine = getLine newlen ws
```

为什么当前行的剩余部分长度为`len -(length w +1)`呢? 我们必须为词 w 和其之后的空白分隔符分配空间。下面是一个例子

```
getLine 20 ["Mary", "Poppins", "looks","like",...
⇝"Mary":getLine 15 ["Poppins", "looks","like",...
⇝"Mary":"Poppins":getLine 7 ["looks","like",...
⇝"Mary":"Poppins":"look":getLine 1 ["like",...
⇝"Mary":"Poppins":"look":[]
⇝["Mary","Poppins","look"]
```

就如`getWord`有一个伴侣函数`dropWord`一样, `getLine`也需要一个伴侣函数:

```
dropLine :: Int -> [Word] -> Line
```

将一个词的列表分解为长至多为 lineLen 的文本行的函数可定义为

```
splitLines :: [Word] -> [Line]
splitLines  [] = []
splitLines  ws
  =getLine lineLen ws : splitLines (dropLine lineLen ws)
```

结论

为了将一个文本串分解成文本行, 我们定义

```
fill :: String -> [Line]
fill = splitLines . splitWords
```

要将此结果转换为一个串, 我们需要下列函数

```
joinLines :: [Line] -> String
```

此函数的定义留给读者作为练习。

习题

7.19 定义本节说明的函数`dropLine`

7.20 定义函数`joinLine :: Line -> String` 它将一个文本行转换为可打印形式。例如

```
joinLine ["dog", "cat"]="dog cat"
```

7.21 使用函数`joinLine`或其他函数定义下列函数

```
joinLines :: [Line] -> String
```

它把文本行用换行符分隔后连接成一个串。

7.22 在上例中，我们分别为词和行定义了'take'和'drop'函数。应用“拆分”函数，如引导库函数splitAt，重新设计程序。

7.23 (较难) 修改函数joinLine使之能通过在词之间加入空格后将行的长度调整为lineLen。

7.24 设计一个函数`wc :: String -> (Int,Int,Int)` 对于给定的文本，它返回文本串中的字符数，词数和行数。一个文本行以字符 '\n' 结尾。定义一个类似的函数

```
wcFormat :: String -> (Int,Int,Int)
```

它返回文本中插入空白调整行长度后的相应统计数目。

7.25 定义函数`isPalin :: String -> Bool` 它检查一个串是否一个回文，即顺读和倒读是同一个串。例如，

```
Madam Z'm Adam
```

注意，检查忽略分号和空格，并且不区分大小写。你可以先定义一个函数检查一个串正读和倒读是否完全一致。之后，再考虑忽略分号和大写字母。

7.26 (较难) 定义函数

```
subst :: String -> String -> String -> String
```

使得`subst oldSub newSub st` 为将oldSub在st中的第一次出现用newSub代替后的结果。例如

```
subst "much" "tall" "How much is that?"
      ="How tall is that?"
```

如果oldSub在st中不出现，则结果为st.

小结

本章介绍了如何在列表上定义递归函数，从而完成了介绍定义列表函数的不同方法。通过一些例子，我们讨论了在第四章提出的程序设计原则，包括“分治发”和设计递归程序的一般方法。文本处理一例提供了一个定义函数库的自底向上的方法。

第八章　程序推理

在 1.10 节我们介绍了证明：一个**证明**是说明某个特定的命题成立的论据。一个命题通常是关于某类对象**普遍**成立的陈述。在数学上我们可以证明毕德格拉斯定理：直角三角形的三边具有关系 $a^2 + b^2 = c^2$。

在程序设计中我们可以证明一个程序对于所有的输入具有某个性质。这样的性质能够确保无论在何种条件下，程序都将按照我们的要求运行。与程序测试比较，测试只能保证程序在某个特定的输入集合上按照要求运行，由此只能建立程序将在所有输入上正确运行的信念，然而，数学家不会因为一个命题在某个有限的测试数据集上有效而承认这个命题的普遍有效性。

推理能够应用于函数程序设计的主要原因是，我们可以把一个函数定义理解为它的行为的逻辑描述；我们将在本章的开始对此作深入的探讨。在讨论了推理与测试的关系之后，我们介绍有关程序设计与逻辑的某些课题，然后介绍有限列表上的归纳的中心概念。

归纳证明遵循一个模式，我们将通过一系列例子进行演示。我们还将给出如何寻找归纳证明的建议。本章最后介绍一个更具挑战性的例子，初学者可跳过。

§8.1　理解定义

假设我们问自己一个看似明显的问题："如何理解一个函数的作用？" 答案有很多种。

- 我们可以计算函数在特定输入的值，比如用 Hugs。
- 我们可以用手逐行进行上述计算。其优点是我们可以看到程序如何求得其结果，其缺点是慢，并且除很小的程序外，几乎对所有的程序是不现实的。
- 我们可以推断程序的一般行为。

第三个答案，即在 1.10 节介绍过的对程序进行推理，将是本章的主题。

考虑下面的简单程序

```
length []     = 0                                   (length.1)
length (x:xs)= 1 + length xs                        (length.2)
```

利用这个定义，我们可以计算任意列表的长度，如`[2,3,1]`，

```
length [2,3,1]
⇝ 1+length[3,1]          by (length.2)
⇝ 1+(1+length [1])       by (length.2)
⇝ 1+(1+(1+length[]))     by (length.2)
⇝ 1+(1+(1+0))            by (length.1)
⇝ 3
```

我们也可以将(length.1)和(length.2)读作对函数 length 一般行为的**描述**。

- (length.1)说明length []是什么;
- (length.2)说明，**无论** x 和 xs 取何值，length(x:xs)都等于1+length xs。

第二种情况是 length 的一般性质：它表明长度在非空列表上的行为。在这些等式的基础上，我们可以断言:

```
length [x] = 1                                          (length.3)
```

为什么呢？我们知道(length.2)对所有的 x 和 xs 均成立，所以特别地将 xs 替换为 [] 时，

```
length [x]
= length (x:[])          by defn of [x]
= 1+length []     by defn of (length.2)
= 1+0                    by (length.1)
= 1
```

上述讨论说明, 我们可以用 (至少) 两种方法解读一个函数的定义。

- 我们可以把定义视为如何计算特定结果的描述，如length [2,3,1]。
- 我们也可以把定义视为对函数的行为的**一般描述**。

根据一般描述, 我们可以推导出其他的事实, 有的是直截了当的, 如(length.3), 有的表达了两个或更多函数之间的关系，如

```
length (xs++ys) = length xs + length ys          (length.4)
```

我们将在 8.6 节证明(length.4)

另一种阅读(length.3)的证明的方式是，我们在进行 **符号计算**，而不是计算 length 在特定输入上的值。我们将看到，符号计算是证明的一个重要部分，但是，递归函数的多数证明需要使用另一个原理 — 归纳。

总之，我们看到函数程序直接地“描述它本身”。如果你熟悉命令式语言，如 Pascal, C 或者 Java, 设想如何在这些语言中进行类似于(length.3)和(length.4) 的推理。如何在这些语言中陈述这些性质并非明显的，如何证明这些性质将是更困难的。

§8.2 测试与证明

在 4.5 节介绍程序测试时，我们考虑了如下例子：

```
mysteryMax :: Int -> Int -> Int -> Int
mysteryMax x y z
  | x > y && x > z    = x
  | y > x && y > z    = y
  | otherwise         = z
```

这是求三个数中最大者的一种尝似，我们可以试着证明这个定义是正确的，这需要考虑三个值的各种顺序。

如果我们首先考虑下列情况

```
x>y && x>z
y>x && y>z
z>x && z>y
```

那么，在这些情况下 mysteryMax 将给出正确的答案。对于其他的情况，至少有两个值是相等的。如果三个值均相等，

```
x == y && y == z
```

函数定义也正确，最后考虑恰好有两个值相等的情况。对于

```
y == z && z>x
```

函数仍然是正确的。但是，对于

```
x == y && y>z
```

函数的定义给出错误的结果 z。

我们可将上述证明的尝试过程看作测试函数的一般方法，这是一种 **符号测试**；我们轮流考虑各种情况，直至发现错误。由此看出，推理不仅呈示给我们一个更传统的观念，即证明一个程序满足规格说明的要求，另外，通过分析正确性不成立的原因，推理提供了一种有力的诊断方法。

另一方面，正如 4.5 节所述，找到一个证明是很困难的，所以，在可靠性软件的开发过程中，证明和测试都有其不可替代的地位。

§8.3 定义性、终止性和有限性

在继续讨论证明之前，我们需要讨论程序设计的两个方面。

定义性和终止性

求一个表达式的值有两种结果：

- 求值可以停止或 **终止**，并给出一个结果，或者
- 求值可以一直进行下去。

如果我们定义

```
fact  :: Int -> Int
fact n
  | n == 0       = 1
  | otherwise    = n * fact(n-1)
```

那么下面两个表达式分别给出对应两个结果的例子

```
fact 2    fact(-2)
```

因为，对于后一表达式

```
fact (-2)
⇝(-2)* fact(-3)
⇝(-2)*((-3)* fact(-4))
⇝ ...
```

对于求值过程不能终止的情况，我们说表达式的值 **无定义**，因为求值过程不能到达一个有定义的值。在进行证明时，我们往往限于考虑那些值有定义的情况，因为许多熟悉的性质仅对于有定义的值成立。一个最简单的例子是表达式`0*e`。我们期望不论`e`的值如何，`0*e`的值均为 0。如果`e`有一个定义的值，这是显然的。但是，如果`e`为`fact(-2)`，则`0*fact(-2)` 的值将是无定义的，而不是 0。在许多证明中，我们将说明结果对所有的定义值成立。这个限制在实际中不会造成任何问题，因为在大部分情况下，我们感兴趣的情形是有定义的情形。我们感兴趣的无定义值的情形是，当我们期望函数给出一个值而它没有给出值时，即符号诊断的情形。

有穷性

到目前为止，我们还未讨论 Haskell 中表达式求值的顺序。事实上，Haskell 的计算是**惰性的**，即函数的参数只有在需要时才被求值。惰性计算赋予 Haskell 一种独特的性质，我们将在第十七章对此进行探讨。更重要的是，惰性计算允许我们定义和使用无穷列表，如`[1,2,3, …]` 以及**部分定义的列表**。下面我们将主要考虑**有限**列表，即长度有限且元素有定义的列表。例如

```
[]    [1,2,3]    [[4,5],[3,2,1],[]]
```

对惰性程序的推理将在 17.9 节讨论。

习题

8.1 给定`fact`的上述定义，试求下列表达式的值

```
(4>2) || (fact (-1) == 17)
(4>2) && (fact (-1) == 17)
```

试回答你为何得到这样的结果。

8.2 试定义乘法函数`mult :: Int -> Int -> Int` 使得`mult 0 (fact(-2))`⇝`0`。计算`mult (fact (-2)) 0`的结果是什么？为什么？

§8.4 一点逻辑知识

对函数程序进行推理不需要形式逻辑的背景。然而，在讲解证明之前，值得讨论逻辑的两个概念。

证明中的假设

首先，我们观察包含假设的证明。举一个初等算术的例子。如果假设每升汽油售价 27 便士，那么我们可以证明 4 升汽油的价格为 1.08 磅。这说明

什么呢？它并没有告诉我们 4 升汽油的花费是多少，只是告诉我们在假设成立时的花费。要确认花费为 1.08 磅，我们需要假设成立的证据：这种证据可以是另一个证明，如基于每加仑售价为 1.20 磅，或者是直接的证据。我们可以把以上证明的命题写成一个公式：

1 升售价为 27 便士 $\Rightarrow$ 4 升售价为 1.08 磅

其中的箭头是 **蕴涵** 的逻辑符号，它表明第二个命题由第一个命题推出。

我们已经看到，在证明 $A \Rightarrow B$ 时，我们假定 A，然后证明 B。如果我们能找到 A 的一个证明，那么蕴涵 $A \Rightarrow B$ 的成立将保证 B 是成立的。

另一种解读蕴涵的方法是将 $A \Rightarrow B$ 的证明视为将 A 的一个证明转变为 B 的一个证明的机器。我们在归纳证明中将借助这样的思想，因为归纳证明包含一个称为归纳步的过程，其任务是假设一个性质，然后证明另一个性质。

自由变量和量词

当我们写下如下等式时

```
square x = x*x
```

我们想表达的是此等式对自由变量 x 的**所有值**均成立。如果我们想显式地表达“所有的”，可以使用一个 **量词**∀x(square x = x*x)。这里的全称量词”∀x”读作“对所有的 x...”。

下面我们转入归纳，这是证明程序性质的主要技术。

§8.5 归纳法

我们在第七章看到，定义列表上函数的一种一般方法是利用原始递归，如

```
sum  ::  [Int] -> Int
sum  []     = 0                          (sum.1)
sum  (x:xs) = x+sum xs                   (sum.2)
```

我们在这里直接给出函数在 [] 上的值，并且利用sum xs的值来定义sum(x:xs)的值。结构归纳法是一个证明原理。

定义 2 (列表结构归纳原理) 要证明一个逻辑性质P (xs)对所有**有穷**列表xs成立，我们需要做下列两件事。

- **归纳基**：证明P([])成立。
- **归纳步**：假设P(xs)成立，证明P(x:xs)成立。换言之，证明 P(xs) $\Rightarrow$ P(x : xs) 成立。这里的P(xs)称为归纳假设，因为它是在证明P(x:xs) 时所做的假设。

有趣的是归纳原理与原始递归非常相似，所不同的是，在归纳中，我们构造的不是函数的值，而是证明。相同的是，我们把 [] 作为基，然后构造从 xs 到 x:xs 的一般情况，在函数定义中，我们使用fun xs定义fun(x:xs)，而在P(x:xs)的证明中，我们可以使用P(xs)。

合理性

正如我们辩论递归不是循环，同样的论据适用于分阶段构造的所有有限列表的证明。假设给定P([])和P(xs) ⇒ P(x : xs) 的证明，我们想证明P([1,2,3]).

列表[1,2,3]是利用 [] 和 cons 如下构造的

```
1:2:3:[]
```

我们可以利用与上述逐步构造过程类似的方法构造P([1,2,3])的证明。

- P([])成立；
- P([]) ⇒ P([3]) 成立，因为它是 P(xs) ⇒ P(x : xs) 的一种特殊情况；
- 回顾上面 ⇒ 的讨论，如果我们知道 P([]) ⇒ P([3]) 和P([])均成立，则可推出P([3])成立。
- 因为 P([3]) ⇒ P([2, 3]) 成立，同理，P([2,3])成立。
- 最后，因为 P([2, 3]) ⇒ P([1, 2, 3]) 成立，所以P([1,2,3])。

上述解释以一个特别列表为例，但是对任意列表也成立：如果列表有 n 个元素，那么如上的证明过程有 n+1 步。总之，如果我们知道归纳原理的两个要求均满足，则P(xs)对每个有穷列表 xs 成立。

第一个例子

我们已经看到函数 sum 的定义，再回顾将列表的所有元素加倍的函数

```
doubleAll []     = []                              (double.1)
doubleAll (z:zs) = 2*z :doubleAll zs               (double.2)
```

现在，我们期望 doubleAll 与 sum 如何交互作用呢？如果我们将列表的所有元素加倍，然后求和，其结果与将列表的和加倍是一样的：

```
sum (doubleAll xs) = 2* sum xs                      (sum+dbAll)
```

构建归纳

如何证明此性质对所有 xs成立呢？根据结构归纳原理，我们有两个归纳目标。第一个是归纳基

```
sum (doubleAll []) = 2* sum []                    (base)
```

第二个是归纳步，证明

```
sum (double (x:xs)) = 2* sum (x:xs)               (ind)
```

可利用的归纳假设为

```
sum (doubleAll xs) = 2* sum xs                    (hyp)
```

在今后的证明中，我们将用(base),(ind)和(hyp)标注这几种情况。

归纳基

如何证明(base)呢?我们可以使用的资源只有等式(sum.1),(sum.2),(doubleAll.1)和 (doubleAll.2)，所以我们必须利用它们。在证明一个等式时，我们可以考虑化简等式的两边。先化简等式的左边

```
sum (doubleAll [])
= sum []              by (doubleAll.1)
= 0                         by (sum.1)
```

再化简等式的右边

```
2* sum []
= 2*0       by (sum.1)
= 0               by *
```

这表明两边相等，故归纳基成立。

归纳步

我们需要证明(ind)。同归纳基一样，我们可以利用定义 doubleAll 和 sum 的等式，但是我们也可以 – 通常应该 – 应用归纳假设(hyp)。与归纳基的证明类似，先利用定义尽可能地化简要证明的等式的两边。首先化简左边

```
sum (doubleAll (x:xs))
= sum (2*x :doubleAll xs)   by (doubleAll.2)
= 2*x + sum(doubleAll xs)         by (sum.2)
```

然后化简右边

```
2*sum(x:xs)
= 2* (x+ sum xs)    by (sum.2)
= 2*x + 2* sum xs      by arith
```

我们已经利用定义的等式化简了(ind)的每一边。证明两边相等的最后一步可由归纳假设(hyp)得出，即利用(hyp)化简左边

```
sum (doubleAll (x:xs))
= sum (2*x:doubleAll xs)
= 2*x + sum(doubleAll xs)
= 2*x + 2* sum xs           by (hyp)
```

在归纳假设成立的假设下，最后一步使得(ind)的两边相等。由此完成了归纳步的证明，也完成了性质的整个证明。 □

我们方框用来表示一个证明的结束。

寻找归纳证明

观察前一个例子，我们可以总结出证明递归函数的性质的一些建议：

- 陈述归纳的目标和归纳的两个子目标(base)和 (hyp) ⇒ (ind)。
- 为避免混淆，改变有关定义中的变量名使其不同于归纳证明中的变量。
- 唯一可利用的证明资源是有关函数的定义和算术运算规则。使用这些资源化简子目标。如果子目标是一个等式，分别化简等式的每一边。
- 在证明归纳步(ind)时，应该使用归纳假设(hyp)；如果没有使用(byp)，那么很可能证明是不正确的。
- 在证明中给每一步标以根据：每个根据通常是函数的一个定义等式。

在下一节我们将看到一系列的例子。

§8.6 归纳证明的进一步例子

本节我们介绍有限列表上结构归纳证明的两个例子。

例 8.1 (length 和 ++)

首先考虑本章开始介绍的例子(length.4)

```
length(xs++ys) = length xs + length ys        (length.4)
```

回顾 length 和 ++ 的定义

```
length []     = 0                             (length.1)
length (z:zs) = 1+(length zs)                 (length.2)
[ ]++ zs      = zs                            (++.1)
(w:ws)++ zs   = w:(ws++zs)                    (++.2)
```

其中我们选择了新的变量名以避免与目标中的变量名发生冲突。

因为(length.4)包含两个变量 xs 和 ys, 所以如何进行证明需要一些思考。由定义作为向导可以看出, ++ 是在第一个变量上的递归。所以, (length.4)的证明方法是对 xs 进行归纳, 证明(length.4)对所有的 xs 和 ys 成立, 因为 ys 是表示任意列表的变量, 正如先前(length.3)的证明中 x 表示列表的任意元素。

命题 我们可以陈述归纳证明的两个目标。

归纳基的情况需要证明

```
length([]++ ys) = length []+ length ys   (base)
```

在归纳步的情况需要证明

```
length ((x:xs)++ys) = length (x:xs) + length ys   (ind)
```

可利用的归纳假设为

```
length (xs++ys) = length xs + length ys   (hyp)
```

归纳基证明 分别检查(base)的两边, 首先是左边。

```
length ([]++ys)
 = length ys                  by (++.1)

length [] + length ys
= 0 + length ys           by (length.1)
= length ys
```

这表明(base)成立。

归纳步的证明 首先化简(ind)的左边:

```
length ((x:xs)++ys)
 = length (x:(xs++ys))       by (++.2)
 = 1+length (xs++ys)     by (length.2)
```

利用定义不能进一步化简上式, 但可以利用(hyp)得到

```
  = 1+ length xs + length ys  (hyp)
```

再来看(ind)的右边

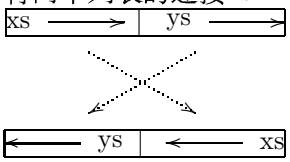
```
  length (x:xs) + length ys
  = 1+length xs + length ys   by (length.2)
```

可见(ind)可由(hyp)推出，由此完成了第二部分的证明，也结束了整个证明。

□

例 8.2 (reverse 与 ++)

将两个列表的连接xs ++ ys求逆的结果是什么呢？



结果是两个列表求逆，并将结果交换位置

```
 reverse (xs++ys) = reverse ys ++ reverse xs  (reverse++)
```

其中我们定义

```
  reverse []       = []                  (reverse.1)
  reverse (z:zs)   = reverse zs ++ [z]   (reverse.2)
```

我们将利用对 xs 的归纳证明(reverse++)对于所有的有穷列表 xs 和 ys 成立。

命题 归纳基为

```
 reverse ([]++ys) = reverse ys ++ reverse []  (base)
```

归纳步的证明目标为利用归纳假设

```
 reverse (xs++ys) = reverse ys ++ reverse xs  (hyp)
```

证明

```
 reverse ((x:xs)++ys) = reverse ys ++ reverse (x:xs)  (ind)
```

递归基的证明 化简(base)的两边

```
 reverse ([]++ys)
 = reverse ys                    by (++.1)

 reverse ys ++ reverse []
 = reverse ys ++ []         by (reverse.1)
```

但是，要证明两式相等，我们必须证明在一个列表尾部连接空列表是恒等运算，即

```
 xs++[] = xs  (++.3)
```

我们将上述命题的证明留作练习。

归纳步的证明：首先化简等式的左边

```
reverse ((x:xs)++ys)
= reverse (x:(xs++ys))                    by (++.2)
= reverse (xs++ys) ++ [x]              by (reverse.2)
= (reverse ys ++ reverse xs) ++ [x]        by (hyp)
```

再化简等式的右边

```
reverse ys ++ reverse (x:xs)
= reverse ys ++ (reverse xs ++[x])  by (reverse.2)
```

以上两式几乎相等, 只是连接的先后顺序不同. 我们需要证明 ++ 的另一个性质, 即**结合性**:

```
xs ++ (ys ++ zs) = (xs ++ ys) ++ zs  (++.4)
```

此性质的证明留作练习。

上述证明也是很典型的证明：通常在证明的过程中需要使用其他定理或者引理 (数学家为“小定理”所起的名称)。如果我们进行任何严肃的证明, 我们将构造由这些引理构成的库, 其中(++.3)和(++.4)是关于 ++ 的基本性质, 这些引理可以在其他证明中随时引用。可以想象, 这些库和引导库类似, 并且在使用引导库函数时可以随时引用其性质。本节末尾的多数练习要求读者证明引导库函数的性质。

习题

8.3 证明对于所有的有穷列表xs和ys

```
sum (xs++ys) = sum xs + sum ys  (sum++)
```

8.4 证明 ++ 的两个性质：

```
xs ++ [] = xs                          (++.3)
xs ++ (ys ++ zs) = (xs ++ ys) ++ zs  (++.4)
```

对于所有的有穷列表xs、ys和zs成立。

8.5 证明对于所有的有穷列表xs

```
sum (reverse xs)    = sum xs
length (reverse xs) = length xs
```

从这两个证明中你能观察到什么共同因素。

8.6 证明对于所有的有穷列表xs和ys

```
elem z (xs ++ ys) = elem z xs || elem z ys
```

8.7 证明对于所有的有穷列表ps

```
zip (fst (unzip ps)) (snd (unzip ps)) = ps
```

列表xs和ys在什么样的条件下有

```
unzip (zip xs ys) = (xs, ys)
```

请给出证明。其中unzip的定义如下

```
unzip [] = ([], [])
unzip ((x, y): ps)
```

```
  = (x:xs, y:ys)
    where
    (xs, ys) = unzip ps
```

8.8 证明对于所有的有穷列表xs和有定义的n

```
take n xs ++ drop n xs = xs
```

§8.7 推广证明目标

使用归纳直接证明一个目标并不总是那么容易的。本节讨论一个例子，我们经历两次失败后才构造出一个性质的证明。本节较本章的其他各节都更富挑战性，读者在初次阅读时可略过。

转移函数

函数 shunt 将一个列表的元素移至另一个列表:

```
shunt :: [a] -> [a] -> [a]

shunt []     ys = ys                    (shunt.1)
shunt (x:xs) ys = shunt xs (x:ys)       (shunt.2)
```

例如，第二个参数由空列表开始，我们有

```
shunt [2,3,1] []
⇝ shunt [3,1] [2]
⇝ shunt [1] [3,2]
⇝ shunt [] [1,3,2]
⇝ [1,3,2]
```

由此看出, 可以利用此函数将列表前后倒置:

```
rev ::  [a] -> [a]
rev xs = shunt xs []   (rev.1)
```

下面我们证明函数 rev 的性质。

第一次证明尝试

将一个列表倒置两次的结果为原列表本身，所以我们将证明对所有的有穷列表xs

```
rev (rev xs) = xs            Q(xs)
```

归纳基的证明很容易。但是当证明归纳步时，我们遇到了第一个问题。

```
rev (rev (x:xs))
= shunt (shunt (x:xs) []) []     by (rev.1)
= shunt (shunt xs [x]) []      by (shunt.2)
```

上式与归纳假设没有直接关系，因为归纳假设只涉及函数 rev。问题是 rev 并非用直接递归定义的函数，它只是 shunt 的一个特例。可否将Q(xs)推广，使之显式地表达 shunt 的性质，并可使用归纳法证明。

一般地，shunt xs ys的结果是

```
(reverse xs) ++ ys
```

如果倒置此列表，我们将得到

```
(reverse ys) ++ xs
```

所以，我们应该能证明

```
shunt (shunt xs ys) [] = shunt ys xs
```

当 ys 被[]代替时，我们得到Q(xs)。因此，我们希望证明上述推广的命题。

第二次尝试

我们的目标是证明对于所有的有穷列表 xs和 ys

```
shunt (shunt xs ys) [] = shunt ys xs
```

当 xs为 []时，证明很简单。对于归纳步

```
shunt (shunt (x:xs) ys) []
  = shunt (shunt xs (x:ys)) []   by (shunt.2)
```

我们希望能证明上式等于shunt (x:ys) xs, 为此我们需要下面的结果

```
shunt (shunt xs (x:ys)) [] = shunt (x:ys) xs
```

而不是下面的归纳假设

```
shunt (shunt xs ys) [] = shunt ys xs
```

为了解决此问题，我们将归纳假设强化为

```
shunt (shunt xs zs) [] = shunt zs xs
```

对所有的有穷列表 zs成立。特别地，当 zs为 x:ys 时也成立。下面我们证明强化的命题。

成功的证明

用逻辑表示法，我们的目标是对 xs归纳证明

$$\forall zs(\texttt{shunt (shunt xs zs) [] = shunt zs xs})$$

命题 下面是我们需要证明的命题。归纳基为

```
∀zs(shunt (shunt [] zs) [] = shunt zs [])  (base)
```

归纳步要证明

```
∀ zs(shunt (shunt (x:xs) zs) [] = shunt zs (x:xs))  (ind)
```

归纳假设为

```
∀ zs(shunt (shunt xs zs) [] = shunt zs xs)  (hyp)
```

归纳基 我们需要证明对于所有的 zs

```
shunt (shunt [] zs) [] = shunt zs []
```

等式的左边可以一步化为右边

```
shunt (shunt [] zs) []
  = shunt zs []          by (shunt.1)
```

归纳步 我们证明对于所有的 zs

```
shunt (shunt (x:xs) zs) [] = shunt zs (x:xs)
```

化简等式的左边

```
shunt (shunt (x:xs) zs) []
= shunt (shunt xs (x:zs)) []    by (shunt.2)
```

利用归纳假设 (hyp), 将全称变量 zs用 x:zs 代替得出：

```
= shunt (x:zs) xs        by (hyp)
= shunt zs (x:xs)    by (shunt.2)
```

这便是欲证命题等式的右边，由此完成了 (ind) 的证明，也完成了整个归纳证明。 □

这个例子表明，为了使用归纳法，我们有时必须将证明的命题推广。这听起来有些矛盾：显然欲证明的命题变得更难了。但是，另一方面，我们也使得归纳假设更强，所以，在证明归纳步时，我们有更多的资源可以利用。

习题

8.9 证明对所有的有穷列表 xs和 ys

```
rev (xs ++ ys) = rev ys ++ rev xs
```

8.10 应用函数

```
facAux :: Int -> Int -> Int
facAux 0 p = p
faxAux n p = facAux (n-1) (n+p)
```

我们可以定义

```
fac2 n = facAux n 1
```

试证明对所有有定义的自然数 n

```
fac n = fac2 n
```

小结

本章显示，我们可以给 Haskell 程序以逻辑的解读，由此可以对它们进行推理。对列表推理的主要方法是结构归纳，结构归纳之于证明正如原始递归之于定义。

我们给出了如何构造函数程序证明的一些提示，并且通过引导库函数，如 sum, ++和 length，以及一些练习对此作了演示。

第九章　推广：计算模型

软件**重用**是软件业的一个重要目标。像 Haskell 这样的现代函数程序设计语言的一个重要特征是我们可以定义**通用**函数，这些通用函数可用于许多不同的场合。例如，Haskell 引导库的列表函数构成一个我们经常使用的工具箱。

我们已经看到**多态**的这种通用性，即**同一个**程序可用于许多**不同**的类型上。第五章介绍的列表上的许多函数，包括 length，++ 和 take 便是这样的例子。这些函数在每一个类型上的作用是相同的。例如，length 计算任意类型列表的长度。本章介绍通用性的另一种机制，使用这种机制我们可以定义嵌入了一种**计算模式**的函数；下面两个例子对此做出解释。

- *用某种方式对一个列表的所有元素做变换*。例如，我们可以将每个字母字符转换为大写字符，或者将每个数加倍。
- *利用某种运算将一个列表的元素结合起来*。例如，将一个数的列表的元素加在一起。

如何定义实现这种模式的函数呢？我们需要将这样的变换或者运算作为通用函数的**参数**；换句话说，我们需要定义将其它函数作为参数的函数。这些**高阶函数**是本章的主题。与此相对的是将在第八章介绍的其**结果**为函数的函数。

我们将先回顾先前遇到的列表上的计算模式，然后讨论如何用 Haskell 高阶函数来实现它们。我们还将看到，原始递归函数的定义推广了利用运算结合列表元素的过程。

最后我们介绍一个推广的例子：将 String 上的函数推广为一个多态、高阶函数。其方法是找出函数中仅适用于 String 的部分，并将其作为函数的参数。这个例子可作为推广函数的一种模式，使一个函数可适用于多种场合，从而变为可重用的函数。

§9.1　列表上的计算模式

我们前面看到的列表处理函数可划分为几类。本节回顾前几节的内容，讨论其中出现的模式。这些模式将在后几节用 Haskell 高阶函数实现。

应用于所有的元素 — 映射

许多函数需要将列表的所有元素用某种方式转换，我们把它称为映射。我们在第一章已经看到这样的例子，在那里为了把一个图形在垂直镜子中反转 –flipV– 我们需要将图形 (一个线段的列表) 的每一个线段倒置。

在第五章的第一个列表概括的例子中我们也看到了映射的例子，它将一个数的列表的每个一元素加倍。

```
doubleAll [2, 3, 71] = [4, 6, 142]
```

其它的例子包括

- 在一个二元组列表中取每个元素的第二个分量, 如图书馆数据库的例子;
- 在超市帐单的例子中, 将条码列表中的每一项转换为相应的 (Name, Price) 二元组;
- 将列表中的每一项 (Name, Price) 格式化。

选择元素 — 过滤

在一个列表中选出具有给定性质的所有元素也是常见的操作。第五章包含这样的例子：在一个串中选出所有的数字：

```
digits "29 February 2004" = "292004"
```

我们遇到的其他例子包括

- 选取其第一个分量为特定的人的所有二元组;
- 选取不等于借阅二元组的所有二元组。

将元素结合在一起 — 折叠

第七章的第一个例子是求一个整数列表的和 sum。整数列表的和可以通过折叠函数 + 求得：

```
sum [2,3,71] = 2+3+71
```

类似地,

- 将 ++ 折叠入列表的列表实现列表的连接, 例如 concat 的定义;
- 将 && 折叠入布尔值的列表从而实现它们的合取：引导库函数 and 便是这样定义的;
- 将 max 折叠入一个整数列表可求出其中最大的整数。

拆分列表

在第七章的文本处理一例中, 一个常用的模式是在一个列表中当元素具有某性质时继续选取或者舍弃此元素。第一个例子是 getWord,

```
getWord "cat dog" = "cat"
```

在这里, 当字符为字母时我们继续选取。其它例子包括 dropWord, dropSpace 和 getLine。其中最后一个函数不只依赖于列表的元素, 而且依赖于当前选出的部分。

结合

以上计算模式常常是在一起使用的。在图书馆数据库一例中, books 的定义需要返回某人借阅的所有图书, 为此, 我们过滤出所有与此人有关的二元组, 然后在此结果上取出所有的第二个分量。列表概括便是映射与过滤的结合, 因此, 列表概括特别适合于许许多多的例子, 如图书馆一例。

函数的其他结合也是常见的。

- 在图形实例研究中, 函数 invertColour 通过使每一条线段反色达到反色的目的; 每条线段的反色需要使每个字符反色, 所以这里需要嵌套的映射。
- 格式化超市帐单涉及用某种方法对每一项进行处理, 然后用 ++ 将结果结合起来。

原始递归与折叠

许多函数的定义形式是原始递归的。插入排序是一个经典的例子:

```
iSort []     = []
iSort (x:xs) = ins x (iSort xs)
```

Haskell 提供了一种将像 ins 这样的前缀函数转换为中缀形式的机制, 其方法是用反引号将函数名括起来, 如 ‘ins‘, 因此

```
iSort (x:xs) = x ‘ins‘ (iSort xs)
```

将其应用于一个列表例子, 我们得到

```
iSort [4,2,3] = 4 ‘ins‘ 2 ‘ins‘ 3 ‘ins‘ []
```

使用这种方式, 上述定义就像把 ‘ins‘折叠入列表`[4,2,3]`。我们将在 9.3 节继续讨论这个例子。

最后的 10%

到目前为此我们讨论的定义都是原始递归的, 也就是说, 我们可以用`xs`的结果定义`x:xs`上的结果。据说列表上 90% 的函数是原始递归的。非原始递归的函数的例子包括第七章的函数 quicksort 和 splitLines,

```
splitLines [] = []
splitLines ws
  = getLine lineLen ws : splitLines (dropLine lineLen ws)
```

对于非空列表 ws, `splitLines ws`的计算并未使用 splitLines 在 ws 的尾部的调用, 而是使用了`(dropLines lineLen ws)`上的调用。注意到在列表 ws 中没有一个词的长度大于行宽 lineLen 的情况下, `(dropLine lineLen ws)`总是比 ws 短, 所以, 这种形式的递归将终止。

§9.2 高阶函数：函数作为参数

一个 Haskell 函数称为 **高阶** 函数, 如果它的一个参数是函数或者其结果是函数。本节向大家介绍如何使用函数参数定义各种函数, 包括上一节讨论的模式。

映射 – 函数 map

我们可以用两种方式将整数列表的所有元素加倍, 或者使用列表概括

```
double :: [Int] -> [Int]
double xs = [2*x | x <- xs]
```

或者利用原始递归

```
double []     = []
double (x:xs) = 2*x : double xs
```

对于这两种方式, 我们看到特定的乘 2 运算被应用于列表的一个元素, 即表达式`2*x`。

假如我们要用另一种运算修改列表的每个元素, 例如, 将 Char 转换为 Int 的函数 ord, 我们可以修改上面的一个定义, 将`2*x`改为`ord x`, 从而得到一个不同的定义。

使用这样的定义方式, 我们将定义大量的函数, 其不同之处仅在于所应用的转换函数。另一种方法是定义一个函数, 转换函数变为它的参数。这种函数的一般定义为

```
map f xs = [ f x | x <- xs]                    (map.0)
```

或者使用递归

```
map f [] = []                                  (map.1)
map f (x:xs) = f x : map f xs                  (map.2)
```

将列表所有元素加倍的函数可以用 map 来实现：将 map 应用于转换函数 double 以及要处理的列表。

```
doubleAll xs = map double xs
```

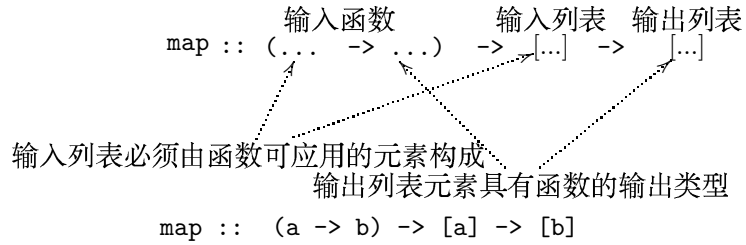
其中`double x = 2*x`。类似地, 将所有字符转换为它们编码的函数可定义为

```
convertChrs :: [Char] -> [Int]
convertChrs xs = map ord xs
```

在图形实例研究中, 将一个图形在垂直镜子中反转的函数可定义如下

```
flipV :: Picture -> Picture
flipV xs = map reverse xs
```

函数 map 的类型是什么呢? 它有两个参数, 第一个是一个函数, 第二个是一个列表, 其结果是一个列表。



上图表示函数的类型与列表的关系, 由此推出 map 的类型

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

这里 a 和 b 是类型变量, 代表任意类型。map 类型的特例包括在 doubleAll 定义中使用的 map, 其中 map 被应用于类型为 `Int -> Int`的函数

```
map :: (Int -> Int) -> [Int] -> [Int]
```

以及 convertChrs 中 map 的应用

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

将性质表示为函数

在定义一个函数用于过滤或者选取列表中具有特定性质的元素时, 我们需要考虑如何在 Haskell 中描述这些性质。例如, 以前提到的过滤字符串中数字的函数 digits。性质 “是一个数字” 是如何描述的呢？引导库包含一个函数

```
isDigit :: Char -> Bool
```

将此函数应用于一个特定的字符, 如'd', 其结果为布尔值 True 或者 Flase, 由此可判定这个字符是否数字。

这也是我们描述任意类型 t 上性质的方法。类型 t 上的性质可以用下列类型的函数表示:

```
t -> Bool
```

如果`f x`的值为 `True`, 则元素 `x`具有这个性质。我们已经看到一个例子 isDigit; 其他例子包括

```
isEven :: Int -> Bool
isEven n = (n`mod` 2 == 0)


isSorted :: [Int] -> Bool
isSorted xs = (xs == iSort xs)
```

这里延用了性质以'is' 开始的习惯。

过滤函数 filter

由上述对性质的讨论可以看出, 函数 filter 以一个性质和一个列表作为参数, 并返回列表中具有此性质的元素。

```
filter p [] = []                                (filter.1)
filter p (x:xs)
   | p x       = x : filter p xs                (filter.2)
   | otherwise =     filter p xs                (filter.3)
```

如果列表空, 则结果为空。对于非空列表 `x:xs`, 分二种情况。如果守卫条件`p x`为真, 则 x 为结果列表的第一个元素; 结果的其余部分从 `xs`选出那些满足条件 p 的元素。如果`p x` 为 `False`, 则 x 不包含在结果列表中, 结果由 `xs`中具有性质 p 的元素构成。

列表概括也可用于定义 filter:

```
filter p xs = [x | x <- xs, px]                  (filter.0)
```

从中看出, x 包含在结果中的条件是它满足性质 p。

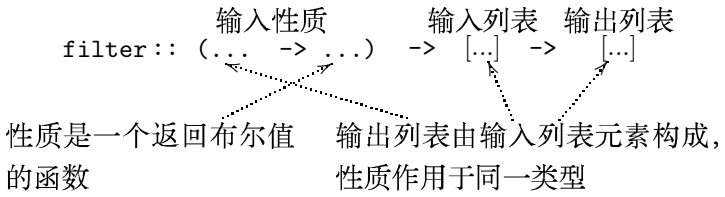
函数 digits 可以用 filter 定义如下

```
digits xs = filter isDigit xs
```

其他应用 filter 的例子

```
filter isEven [2,3,4,5]                        ⇝ [2,4]
filter isSorted [[2,3,4,5],[3,2,5],[],[3]]   ⇝ [[2,3,4,5],[],[3]]
```

函数 filter 的类型是什么呢？它的参数为一个性质和一个列表，结果为一个列表。




```
filter :: (a -> Bool) -> [a] -> [a]
```

结合 zip 和 map– 函数 zipWith

我们已经看到下列的多态函数

```
zip :: [a] -> [b] -> [(a,b)]
```

它将两个列表的相应元素组成元素对，形成二元组的列表。例如

```
zip [2,3,4] "Frank" = [(2,'F'),(3,'r'),(4,'a')]
```

假如我们要做的不是将相应元素组成对，而是另一种运算，那么情况会怎么样呢？回顾第一章图形实例研究中的函数 sideBySide，我们想把对应的线段用 (++) 连接起来。为此，我们定义了函数 zipWith，它结合了拉链函数和映射的效果

```
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _       _     = []
```

对于第一种情况，两个列表均不空时，我们将函数 f 应用于两个列表的头元素并将其作为结果的头元素，随后用类似的方式将 f 应用于两个列表的尾部。对于第二种情况，当至少有一个列表为 []时，结果也是 []，这与 zip 的定义类似。

回到 Picture 实例，我们可以定义

```
sideBySide :: Picture -> Picture -> Picture
sideBySide pic1 pic2 = zipWith (++) pic1 pic2
```

函数 zipWith 的类型是什么呢？函数有三个参数。第二个和第三个参数为任意类型的列表，分别为[a]和[b]。结果也是任意类型的列表，设为[c]。第一个参数被应用于两个输入列表的元素，并返回输出列表的一个元素，所以它的类型必然是a -> b -> c。由此我们得到下面的类型

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

在下面的习题中，我们将介绍其它的高阶函数并进一步讨论这些函数的类型。

习题

9.1 利用 doubleAll 的三种不同定义，列表概括、原始递归和map，分别写出doubleAll [2,1,7]的三步逐行计算过程。

9.2 如何利用 map 和 sum 定义 length?

9.3 给定函数

```
addUp ns = filter greaterOne (map addOne ns)
```

其中

```
greaterOne n = n > 1
addOne n     = n+1
```

如何重新定义 addUp，要求在 map 之前应用 filter，例如

```
addUp ns = map fun1 (filter fun2 ns)
```

9.4 描述下式的结果

```
map addOne (map addOne ns)
```

设f和g为任意函数，你能推断出表达式map f (map g xs)的一般性质吗?

9.5 下列表达式的结果是什么?

```
filter greaterOne (filter lessTen ns)
```

其中lessTen n = n < 10。更一般的表达式filter p (filter q xs)的效果如何呢? 其中p和q表示任意性质。

9.6 给出下列函数的定义，其输入为一个整数列表 ns

- 返回 ns 中元素的平方的列表;
- 返回 ns 中元素的平方和;
- 检查 ns 中的元素是否均大于 0。

9.7 尽可能利用已定义的函数，定义下列函数

- 求函数 f 在输入 0 至 n 上的最小值;
- 测试 f 在 0 至 n 上的值是否全部相等;
- 测试 f 在 0 至 n 上的值是否全大于 0;
- 测试f 0, f 1至f n是否呈递增序。

9.8 定义函数 twice 并写出它的类型：twice 的输入是一个整数到整数的函数和一个整数，输出是将第一个参数两次应用于第二个参数。例如，当输入为 double 和 7 时，结果为 28。你定义的函数的最广类型是什么?

9.9 定义函数 iter 并给出其类型

```
iter n f x = f (f (f ... (f x) ...))
```

其中f在等式右边出现了n次。例如，

```
iter 3 f x = f (f (f x))
```

而且iter 0 f x应该返回x。

9.10 利用 iter 和 double 定义一个函数，对于输入 n 函数返回 2^n。

§9.3 折叠与原始递归

本节考虑一类特别的高阶函数，它们将一种运算或者函数折叠入一个列表。我们将看到，这种折叠运算比我们想象的要简单，而且事实上，列表上的大多数原始递归函数均可以用折叠运算来定义。

函数 foldr1 和 foldr

我们在这里考虑两种折叠函数。首先考虑将一个函数折叠入一个非空列表的函数；在 Haskell 引导库中称之为 foldr1；稍后我们将讨论其命名的来由。

foldr1 的定义分两种情况。将 f 折叠入单元素列表 `[a]` 的结里是 a。将 f 折叠入更长列表的结果如下定义

```
foldr1 f [e1, e2, ..., ek]
   = e1 `f` (e2 `f` ( ... `f` ek) ...)
   = e1 `f` (foldr1 f [e2, ..., ek])
   = f e1 (foldr1 f [e2, ..., ek])
```

因此，我们有如下的 Haskell 定义

```
foldr1 f [x] = x                              (foldr1.1)
foldr1 f (x:xs) = f x (foldr1 f xs)           (foldr1.2)
```

foldr1 的类型为

```
foldr1 :: (a -> a -> a) -> [a] -> a
```

foldr1 的类型表示它有两个参数。

- 第一个参数是类型 a 上的二元函数，例如，Int 上的 `(+)`。
- 第二个参数是类型 a 上的列表，例如，`[3,98,1]`，列表的元素将被运算结合在一起。

运算的结果是类型为 a 的值，例如

```
foldr1 (+) [3,98,1] = 102
```

其他使用 foldr1 的例子有

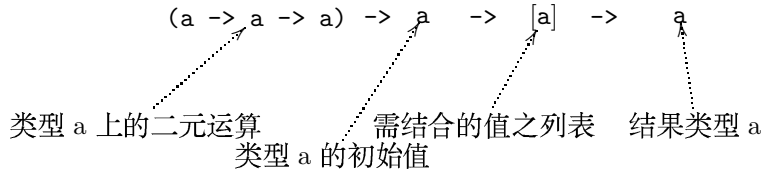
```
foldr1 (||) [False, True, False] = True
foldr1 (++) ["Freak ", "Out", "", "!"] = "Freak Out!"
foldr1 min [6] = 6
foldr1 (*) [1..6] = 720
```

当 foldr1 被应用于一个空列表时，系统产生一个错误信息。

我们可以修改上述定义，给函数一个额外的参数，表示在空列表上的结果，由此给出一个定义在所有有限列表上的函数。这个函数称为 foldr，其定义如下

```
foldr f s []     = s                          (foldr.1)
foldr f s (x:xs) = f x (foldr f s xs)         (foldr.2)
```

其中的 'r' 表示" 折叠、括号靠右"。可以预见函数 foldr 的类型为



利用这个更一般的函数, 我们可以定义 Haskell 的一些标准函数

```
concat :: [[a]] -> [a]
concat xs = foldr (++) [] xs

and :: [Bool] -> Bool
and bs = foldr (&&) True bs
```

返回本节开始的问题, 可以看出 foldr1 命名的来由: 它是在至少有一个元素的列表上的折叠函数。我们也可以用 foldr 定义 foldr1

```
foldr1 f (x:xs) = foldr f x xs                    (foldr1.0)
```

更通用的折叠 foldr

定际上, foldr 的最广类型比我们预测的要广。假设起始值的类型为 b, 列表元素的类型为 a, 则

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

我们将在 13.2 节解释这个类型是如何推导出来的。

了解 foldr 的类型后, 我们便可以利用它定义一大批列表上的函数。例如, 我们可以求一个列表的逆

```
rev :: [a] -> [a]
rev xs = foldr snoc [] xs

snoc :: a -> [a] -> [a]
snoc x xs = xs ++ [x]
```

因为这个函数象 'cons', :, 的逆, 所以传统上称之为 snoc。

我们也可以用折叠进行排序

```
iSort :: [Int] -> [Int]
iSort xs = foldr ins [] xs
```

在继续往下介绍之前, 我们再来看一下 foldr 的定义

```
foldr f s []     = s                              (foldr.1)
foldr f s (x:xs) = f x (foldr f s xs)             (foldr.2)
```

foldr f s 的效果是什么呢? 我们有两种情况

- 在空列表上的结果由 s 直接给出;
- 在 (x:xs) 上的值是由函数在 xs 的值与 x 本身定义的。

这就像第七章介绍的列表上原始递归一样[1]。因此, 使用 foldr 可以定义许多原始递归函数也就不奇怪了。通常可以机械地把原始递归定义转变为使用 foldr 的定义。

这两种方法各有何特点呢？通常在开始的时候, 用递归考虑问题更容易些, 而后再将其转换为 foldr 的应用。进行这种转换的一个优点是, 借助折叠我们可能发现函数的某些性质。我们将在 10.9 节看到通用函数如 map, filter 和 foldr 等的证明。在第十九章我们将介绍其他折叠函数。

习题

9.11 如何使用map和foldr定义自然数从 *1* 到 n 的平方和。

9.12 定义一个求列表中正整数的平方和的函数。

9.13 使用foldr定义引导库函数unZip, last和init。例如,

```
last "Greggery Peccary" = 'y'
init "Greggery Peccary" = "Greggery Peccar"
```

9.14 下列函数的功能是什么*?*

```
mystery xs = foldr (++) [] (map sing xs)
```

其中 `sing x = [x]`。

9.15 函数formatLines的目的是使用下列函数格式化线段列表中的每个线段

```
formatLine :: Line -> String
```

定义下列函数

```
formatList :: (a -> String) -> [a] -> String
```

它用第一个参数格式化第二个列表参数中的每个元素。如何利用 `formatList` 和 `formatLine` 定义 `formatLines`?

9.16 定义函数

```
filterFirst :: (a -> Bool) -> [a] -> [a]
```

使得 `filterFirst p xs` 删除了xs中第一个不满足性质p的元素。利用这个函数给出returnLoan的另一个定义, 它只返回一本书的一个拷贝。如果一个列表的所有元素都具有性质p, 你的函数会做什么呢？

9.17 你能定义下列函数吗？

```
filterLast :: (a -> Bool) -> [a] -> [a]
```

它删除列表中最后一个不具有性质p的元素。你能用 `filterFirst` 定义上述函数吗？

9.18 你能利用你学过的高阶函数简化前述函数的定义吗？你可以利用已学过的函数重新定义有关超市帐单的练习。

---

[1]原来对原始递归的刻画有些模糊。在使用原始递归定义 g 时, `g (x:xs)`的值是用`g xs`以及 x 和 xs 定义的。这就使得原始递归较使用 foldr 的折叠更广泛。

§9.4 推广：折分列表

作为本章的最后一个例子, 我们将函数推广为一个多态、高阶函数。这个推广模式将适用许多其他的情形。

许多列表函数涉及将列表以某种方式折分。一种方法是选取列表中具有一个特殊性质的某些元素或者所有元素，例如 filter。其他方法包括在列表的前面选取或者舍弃某些元素, 我们在文本处理中看到了这样的例子。如果我们知道要选取或者舍弃的元素个数，那么可以使用下列函数

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

其中 `take n xs` 和 `drop n xs` 分别在列表 xs 的前面选取或者舍弃 n 个元素。这些函数是在第七章定义的。

在第七章文本处理的例子中, 我们需要将列表折分为词和文本行。那里定义的函数 getWord 和 dropWord 在空格处折分, 它们不是多态函数。

函数程序设计的一个一般原则是程序通常可以用通用的多态和高阶函数重写, 我们将在此证明这一点。

函数 getWord 的原定义为

```
getWrod :: String -> String
getWord []      = []                              (getWord.1)
getWord (x:xs)
  | elem x whitespace = []                        (getWord.2)
  | otherwise         = x : getWord xs            (getWord.3)
```

使得这个函数在字符串上有效的是(`getWord.2`), 测试 x 是否 whitespace 的成员。我们可以把性质的测试看成一个参数, 从而将函数推广。

我们讲过，类型 a 上的性质可以表示为类型`a -> Bool`的函数。将上述测试看成参数后, 我们得到下列定义

```
getUntil :: (a -> Bool) -> [a] -> [a]
getUntil p []      = []
getUntil p (x:xs)
  | p x           = []
  | otherwise     = x : getUntil p xs
```

其中测试 `elem x whitespace` 被任意性质 p 应用于 x 的测试 `p x` 代替。当然，我们可以由此得到 getWord 的定义

```
getWord xs = getUntil p xs
   where
      p x = elem x whitespace
```

Haskell 的库函数 takeWhile 和 dropWhile 类似于 getUntil 和 dropUntil, 只是它们在条件真时选取或者舍弃元素。例如,

```
takeWhile :: (a -> Bool) -> [a] -> [a]
```

```
takeWhile p []     = []
takeWhile p (x:xs)
  | p x         = x : takeWhile p xs
  | otherwise   = []
```

函数 getUntil 可以用 takeWhile 定义，反之亦然。

习题

9.19 给出函数 dropWord 的推广函数 dropUntil 的类型和定义。

9.20 如何使用 dropUntil 定义函数 dropSpace？如何利用 getUntil 定义 takeWhile？

9.21 如何利用 getUntil 和 dropUntil 将一个字符串折分成文本行。

9.22 第七章的函数 getLine 的多态类型是什么？如何推广函数中的测试？做这样的推广后，函数的类型是否变得更广了？试解释之。

9.23 你能将文本处理函数 getLine，dropLine 和 splitLines 推广为多态、高阶函数吗？

小结

本章演示了如何将列表上非正式的定义模式用高阶和多态函数实现，如 map，filter 和 foldr。我们看到了如何提出这些函数，如何推导出它们的类型以及如何用它们解决问题。

最后介绍了一个函数推广的例子，此例来自文本处理的实例研究，但是，推广函数的方法具有一般性。

本章的重点是如何编写函数作为参数的高阶函数。那么这些函数参数来自哪里呢？函数参数一方面可以是已经定义的函数，另一方面它们本身可以是 Haskell 函数的结果，这将是我们在下一章讨论的内容。

第十章　函数作为值

我们在前一章看到，函数可以作为**高阶函数**的参数。在本章我们将看到，函数还可以是其他函数或运算的结果。这样一来，我们不仅能在 Haskell 脚本中定义函数，而且还可以在程序中将函数作为值构造出来。

这种机制允许我们将某些函数的结果作为其他高阶函数的参数，而且使我们能够充分探讨这些函数的特性。在这种机制下我们将看到所谓的函数的函数层定义，其中使用了我们先前看到的一些通用函数。与传统的定义相比这些定义显得更简洁、更易读, 也更容易实现证明和程序变换。

我们将先介绍一些描述函数的方法，然后介绍如何将函数作为其他函数的结果返回，特别是通过**部分应用**和**部分运算**。我们还将考虑如何将这些思想应用于以前的例子，特别是图形的例子。

一个更长的例子是建立文件的索引，这个例子用于演示如何将这些新思想用于程序开发。本章的结尾是高阶多态函数的验证的例子，从中可以看到已证明定理的重用与函数本身的重用毫无二致。

§10.1　函数层定义

函数程序设计所以称为“函数”的原因之一是在这样的语言中我们可以象处理整数或者列表那样来处理函数。由于这个原因, 我们在本章将看到一个函数通常可以用函数层定义给出。什么意思呢？我们不用解释函数如何作用于它的参数, 例如

```
rotate :: Picture -> Picture
rotate pic = flipV (flipH pic)                    (rotate.1)
```

函数层定义给出函数的直接定义，如

```
rotate = flipV . flipH                            (rotate.2)
```

在这里我们把 rotate 描述为两个反射的复合; 当然, 两个定义 (rotate.1) 和 (rotate.2) 是完全一样的, 但是, 后一种方法有两个优点。首先，第二个定义更易读也更容易修改; 定义很**明确地**表示为两个函数的复合, 而不是如 (rotate.1) 中那样表示为右边如何应用的结果。

更重要的是，用这种形式描述一个定义后，我们在分析 rotate 的性质时可以使用 '.' 的性质。这表明在证明中可以使用复合的性质，可以做程序变换。一般地，这些说明同样适用于高阶多态函数，我们将在 10.9 节看到这样的例子。

我们已经见过其他的直接定义，如

```
flipH :: Pciture -> Picture
flipH = reverse                                   (flipH.1)
```

注意这个定义与下述定义有相同的效果

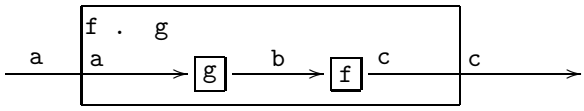


图 10.1: 函数的复合

```
flipH pic = reverse pic
```

因为，如果我们将 (flipH.1) 应用于一个图形, 如 horse, 那么计算的第一步便是

```
flipH jorse
⇝ reverse horse   by (flipH.1)
```

其中的 flipH 被 reverse 代替。

§10.2 函数的复合

我们已经使用了 Haskell 的函数复合运算 '.'; 本节将对其做详细介绍, 特别是它的类型。

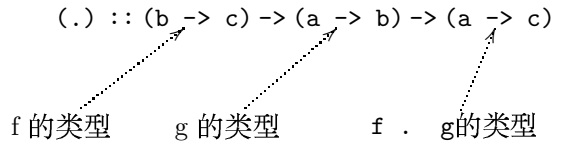
组织程序的一种最简单方法是将一系列任务一个接一个地完成, 而且每一部分都可以独立地设计和实现。在函数程序设计语言中可以通过一系列函数的复合来达到这个目的: 如图 10.1 所示，一个函数的输出成为另一个函数的输入。其中箭头上所标的是元素的类型。

对于任意函数 f 和 g, `f . g` 的结果由下列定义给出

```
(f . g) x = f (g x)                                    (comp.1)
```

并非任意两个函数都可以复合。g 的结果 `g x` 成了 f 的输入，所以，g 的输出类型必须和 f 的输入类型相同。例如，在 10.1 节 rotate 的例子中, flipH 的输出类型与 flipV 的输入类型均为 Picture。

一般地，函数复合的约束是由 '.' 的类型表示的



它表示，如果我们称 f 为第一个输入，g 为第二个输入，则

- f 的输入和 g 的输出具有相同的类型 b;
- 结果 `f . g` 与 g 具有相同的输入类型 a，与 f 有相同的输出类型 c。

复合是可结合的，即对所有的 f, g 和 h, `f . (g . h)` 等于`(f . g) . h`。因此，我们可以毫不含糊地用 `f . g . h` 表示 "先做 h, 再做 g, 然后做 f" [1]。

[1]由于技术上的原因，在 Haskell 引导库中 '.'是右结合的。

向前复合

复合 `f . g` 的顺序是很重要的，并且可能会引起混乱; `f . g` 表示 “先应用 g, 然后在其结果上应用 f” , 所以, 我们必须从右向左阅读函数的复合。

我们将复合 “先 g 后 f” 记作 `f . g` 的原因是函数的参数位于函数的右边。因此，在复合函数的应用 `(f . g) x` 中, 参数 x 更接近于 g，这种记法也与嵌套的应用 `f (g x)` 一致。

在 Haskell 中很容易定义一个与 ‘.‘ 顺序相反的复合运算 `>.>`, 其定义如下

```
infixl 9 >.>

(>.>) :: (a -> b) -> (b -> c) -> (a -> c)
g >.> f = f . g                                    (fcomp.1)
```

这个定义有以下的效果

```
(g >.> f) x = (f . g) x = f (g x)                  (fcomp.2)
```

其中 f 与 g 的顺序在应用之前被交换了。使用这种复合, rotate 可定义如下

```
rotate = flipH >.> flipV
```

我们可以将其读作”flipH 然后 flipV”, 并且先应用左边的函数，后应用右边的函数。记号 `>.>` 中包含一个 ’.’ 表示一种复合，箭头表示信息流的方向。在有多个函数复合时，我们会使用 `>.>`, 这样就不必从多行的尾部往回读来理解复合的含义。

复合中需注意后问题

在使用复合时有两点需要特别注意:

- 由函数应用的结合力而引起的错误。常见的错误是把 `f . g x` 理解成 `(f . g)` 应用于 x。因为函数的结合力最强, 所以 `f . g x` 被系统解释为 `f . (g x)`, 因此往往导致类型错误。

  例如，计算下列表达式

  ```
  not . not True
  ```

  将得到下列类型错误信息

  ```
  ERROR: type error in application
  *** expression      : not . not True
  *** term            : not True
  *** type            : Bool
  *** does not match  : a -> b
  ```

  因为系统试图将 `not True` 看作与 not 复合的函数。这样的函数必须有类型 `a -> b`, 而它的实际类型为 Bool。

  因此，在使用复合时应将其括起来，如 `(not . not) True`。
- 函数应用与函数复合可能会混淆。函数复合将两个函数结合成一个函数，

而函数应用将一个函数与其参数结合 (当然, 参数可以是函数)。

例如, 如果 f 具有类型 `Int -> Bool`, 则

- `f . x` 表示 f 与函数x 的复合, 因此, x 必须具有类型 `s -> Int`, s 是某个类型; `f x` 表示 f 应用于对象 x, 所以 x 必须是一个整数。

习题

10.1 使用复合重新定义 *6.4* 节有关超市帐单习题中的函数printBill。再应用向前复合 >.> 定义此函数。

10.2 设id是多态恒等函数, 其定义为 `id x = x`。试解释下列表达式的行为

```
(id . f)      (f . id)      id f
```

如果 *f* 的类型为 `Int -> Bool`, 那么在每个表达式中用到了id的最广类型 `a -> a` 的哪个特例? 如果 `f id` 是类型正确的, 那么f的类型是什么?

10.3 定义一个函数composeList, 它将一个函数列表复合为一个函数。请给出composeList的类型并解释之。你定义的函数在空列表上有何结果?

§10.3 函数作为值和结果

在本节我们介绍将函数作为其他函数结果的方法; 下一节介绍部分应用的重要技术。

我们已经看到在函数定义的右边可以使用复合运算 '.' 和向前复合 >.> 将函数结合在一起。最简单的例子是

```
twice f = (f . f)                                    (twice.1)
```

其中 f 是一个函数, 结果是 f 与自己复合。为此, f 必须有相同的输入类型和输出类型, 所以

```
twice :: (a -> a) -> (a -> a)
```

这表示 twice 的输入为类型 `a -> a` 的函数, 输出也是类型为 `a -> a` 的函数。例如, 如果 succ 是整数上的加 1 函数

```
succ :: Int -> Int
succ n = n+1
```

那么将 twice 应用于 succ 得到

```
(twice succ) 12
⇝ (succ . succ) 12   by (twice.1)
⇝ succ (succ 12)     by (comp.1)
⇝ 14
```

我们可以将 twice 推广, 用一个参数表示函数参数与它自己复合的次数

```
iter :: Int -> (a -> a) -> (a -> a)

iter n f
  | n>0        = f . iter (n-1) f              (iter.1)
```

```
  | otherwise  = id                                    (iter.2)
```

这是整数参数上的一个标准原始递归函数; 对于正整数的情况，我们取 f 的 n-1 次复合再与 f 复合。对于 0 的情形，我们应用 f 零次, 所以结果是一个对其参数没有任何作用的函数，即 id。利用标准列表函数, 我们可以给出一个构造性定义。

```
iter n f = foldr (.) id (replicate n f)          (iter.3)
```

在上述定义中, 我们创建了 f 的 n 个拷贝的列表

```
[f,f,...,f]
```

然后将复合折叠其中，构成下列的函数

```
f . f . ... . f
```

例如，如果 double 将其参数加倍，那么 2^n 可定义为 `iter n double 1`。

定义函数的表达式

如何书写描述函数的表达式呢？在书写函数定义时，我们可以用 where 定义一个函数。

例如，给定一个整数 n, 我们返回一个将 n 加到其参数上的函数 (类型为 `Int -> Int`)，此函数可定义如下

```
addNum :: Int -> (Int -> Int)
addNum n = addN
           where
           addN m = n+m
```

其结果是名为 addN 的函数，而且 addN 本身是由 where 子句定义的。这是一种间接的定义方法: 我们说返回名为 addN 的函数，然后再定义这个函数。

Lambda 记法

除了先命名再定义函数的间接定义外，我们还可以直接写下这个函数。在定义 addNum 时, 我们可以将结果直接写成

```
\m -> n+m
```

如何解释这个表达式呢？

- 箭头左边的表示参数，在这里有一个参数 m;
- 箭头右边表示结果，这里是 n+m。

上述表达式以符号 '\' 开始，表明它是一个函数。符号 '\' 是 ASCII 字符中与希腊字母 λ 最接近的符号, 而 λ 是函数的数学理论即 lambda 演算中使用的表示函数的符号。现在 addNum 的定义可以表示为

```
addNum n = (\m -> n+m)
```

在下一节我们将看到 addNum 的另一种定义。

图 10.2 所示的 "接插" 是使用 lambda 记法的例子。图中所示的对象是一个函数, 其参数为 x 和 y, 结果为

```
g (f x) ( f y)
```

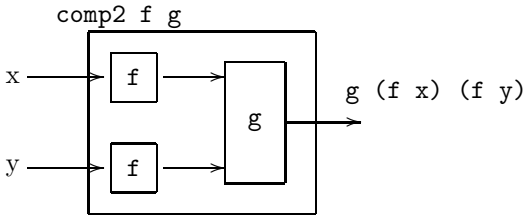


图 10.2: f 和 g 的接插

所以，它的整体效果是一个函数: 在将 g 作用于这个函数的两个参数之前，先将 f 应用于每个参数。函数的定义很直接地描述了这一点:

```
comp2 :: (a -> b) -> (b -> b -> c) -> (a -> a -> c)


comp2 f g = (\x y -> g (f x) (f y))
```

例如，将 3 和 4 的平方相加可表达为

```
comp2 sq add 3 4
```

其中 sq 和 add 的定义是显然的。

一般地，由 lambda 定义的函数是我们先前定义的命名函数的匿名版本。换言之，下列等式定义的函数

```
f x y z = result
```

与下列 lambda 定义的函数

```
\x y z -> result
```

两者的效果是完全一样的。

在下一节我们将看到，部分应用使许多定义更直接，包括本节所定义的函数。另一方面，lambda 更具通用性，因此可用于部分应用不适用的场合。

习题

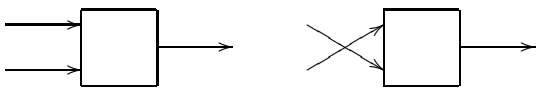
10.4 计算下列表达式

```
iter 3 double 1
(comp2 succ (*)) 3 4
comp2 sq add 3 4
```

10.5 下列函数的类型和结果是什么?

```
\n -> iter n succ
```

10.6 给定一个类型为 `a -> b -> c` 的函数f，用lambda表达式定义一个类型为 `b -> a -> c` 的函数，它的功能与f类似，但参数的次序相反，如下图所示



10.7 利用前一个习题或者其他函数，定义下列函数

```
flip :: (a -> b -> c) -> (b -> a -> c)
```

它将函数参数的参数次序颠倒。

10.8 利用lambda表达式, 布尔函数not和预定义函数elem描述一个类型为Char -> Bool的函数, 它的值只有在非空白字符上为True, 即当字符不是列表 " \t\n" 的元素时。

10.9 定义一个函数total

```
total :: (Int -> Int) ->(Int -> Int)
```

使得 total f 是一个函数, 它作用于一个数n将给出下列结果

```
f 0 + f 1 + ... + f n
```

10.10 (较难) 定义一个函数

```
slope :: (Float -> Float) -> (Float -> Float)
```

它取一个函数参数f, 返回它的 *(近似)* 导数 f'。

10.11 (较难) 定义函数

```
integrate :: (Float -> Float) -> (Float -> Float -> Float)
```

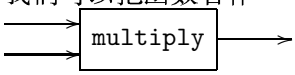
它取一个函数参数f, 返回一个 *(近似)* 二元函数, 其功能是返回f在两个点间图形的面积。

§10.4 部分应用

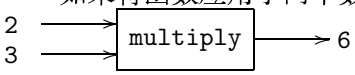
函数 multiply 将两个参数相乘

```
multiply :: Int -> Int -> Int
multiply x y = x*y
```

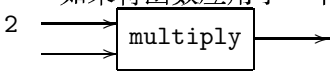
我们可以把函数看作一个方盒, 有两个输入箭头, 一个输出箭头。



如果将函数应用于两个数, 结果是一个数; 例如, multiply 2 3 等于 6:



如果将函数应用于一个参数会怎么样呢? 例如, 将其作用于 2



由图看出, 它表示一个仍在等待另一个输入的函数。当输入一个等侍值 y 时, 函数将其加倍, 即输出 2*y。

这种现象具有普遍性: 任何二元或者多元函数都可以 **部分地应用** 于一个或多个参数。我们由此得到一种构造函数作为结果的有力方法。

为了说明这一点, 我们再来看一个例子: 将列表的元素加倍。函数可以定义如下

```
doubleAll :: [Int] -> [Int]
doubleAll = map (multiply 2)
```

这个定义包含两个部分应用:

- `multiply 2` 是将 multiply 应用于一个参数, 其结果是由整数到整数的函数;
- `map (multiply 2)` 是从 `[Int]` 到 `[Int]` 的函数, 由部分应用 map 所得。

部分应用在这里有两种不同的用途:

- 第一种情况, `multiply 2`, 部分应用用于在定义 doubleAll 时构造传给 map 的函数参数" 乘 2";
- 第二个部分应用, `map (multiply 2)`, 可以通过给 doubleAll 带上参数来避免, 即

  ```
  doubleAll xs = map (multiply 2) xs
  ```

  但是, 如 10.1 节所述, 这种形式的定义有其优点。

  在 10.3 节我们定义 addNum 如下

```
addNum n = (\m -> n+m)
```

addNum 应用于一个整数 n 时返回一个将其参数与 n 相加的函数。部分应用提供了一种更简单的机制, 因为

```
addNum n m = n+m
```

部分应用的思想很重要。我们已经看到许多函数可以定义成通用运算的特殊化, 如 map, filter 等的特殊化。这些特殊化是通过把一个函数传给通用运算来完成的, 而所传的函数通常是一个部分应用。例如, 在第一章图形的例子中

```
flipv       = map reverse
sideBySide  = zipWith (++)
```

在 10.5 节我们将进一步研究图形的例子。

并不是任何时候都可以使用部分应用, 这是因为我们要应用的参数并非总是第一个参数。考虑下列函数

```
elem :: Char -> [Char] -> Bool
```

我们可以用下面的式子测试一个字符是否空字符

```
elem ch whitespace
```

其中 whitespace 表示串 `"\t\n"`。如用 elem 部分应用于 whitespace 来表达一个测试空格字符的函数, 在这里行不通。我们可以定义 elem 的一个变种以改变其参数顺序

```
member xs x = elem x xs
```

然后把所定义的函数写成

```
member whitespace
```

另一种方法是用 lambda 表达式来定义函数

```
\ch -> elem ch whitespace
```

类似地, 要过滤一个串中的所有非空白字符, 我们可以选择使用下列的部分应用

```
filter (not . member whitespace)
```

```
filter (\ch -> not (elem ch whitespace))
```

部分应用的类型

部分应用的类型是如何确定的？我们有一个简单的规则来解释之。

定义 3 (消去规则) 假定一个函数的类型为

```
t1 -> t2 -> ... -> tn -> t
```

将其应用于下列参数

```
e1::t1, e2::t2, ..., ek::tk
```

则其结果的类型为消去 t1 到 tk 后的结果，即

```
t(k+1) -> ... -> tn -> t
```

例如，使用上述规则我们得到下列类型

```
multiply 2      :: Int -> Int
multiply 2 3    :: Int
doubleAll       :: [Int] -> [Int]
doubleAll [2,3] :: [Int]
```

函数应用与->的语法

函数应用是**左结合的**，所以

```
f x y = (f x) y
```

```
f x y ≠ f ( x y)
```

函数空间符号 '->' 是**右结合的**，所以a -> b-> c等同于

```
a -> (b -> c)
```

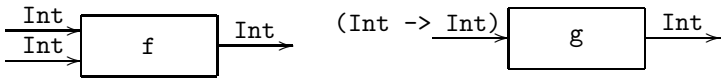
而不是

```
(a -> b) -> c
```

箭头是非结合的。如果

```
f :: Int -> Int -> Int
g :: (Int -> Int) -> Int
```

如下图所示



那么当 f 应用于一个整数时得到一个 Int 到 Int 的函数，例如multiply 2。另一方面，当给 g 一个类型为 Int -> Int 的函数时，结果是一个整数。例如，

```
g :: (Int -> Int) -> Int
g h = (h 0) + (h 1)
```

这里定义的函数取一个函数 h 作为参数，返回参数函数在 0 和 1 上的函数值的和，所以，g succ 的结果是 3。

函数有多少个参数?

部分应用看起来可能有些迷惑: 在有的上下文中函数似乎带有一个参数, 而在其他情况下带有多个参数。定际上, Haskell 的每个函数恰好带有一个参数。如果一个函数应用结果是一个函数, 那么这个函数可以进一步应用于一个参数。再考虑乘法函数

```
multiply :: Int -> Int -> Int
```

它是下列类型的简写

```
multiply :: Int -> (Int -> Int)
```

所以它可以应用于一个整数。例如

```
multiply 2 :: Int -> Int
```

此函数可以再应用于一个整数, 例如

```
(multiply 2) 5 :: Int
```

因为函数应用是左结合的, 所以可写作

```
multiply 2 5 :: Int
```

本书以前的解释与现在的完整解释是一致的。我们隐藏了以下事实：表达式

```
f e1 e2 ... ek
```

```
t1 -> t2 -> ... tn -> t
```

分别是下列式子的简写

```
( ...((f e1) e2) ... ek)
```

```
t1 -> (t2 -> (... (tn -> t)...))
```

但是, 这些并未妨碍我们理解如何使用 Haskell 语言。函数应用的左结合性与 -> 的右结合性便是为了支持这样的简写。

我们将会看到更多部分应用的例子, 以及部分应用用于简化和澄清以前的许多例子。其中三个简单例子来自文本处理中的函数:

```
dropSpace   = dropWhile (member whiteSpace)
dropWord    = dropWhile (not . member whiteSpace)
getWord     = takeWhile (not . member whiteSpace)
```

在介绍部分运算之后, 我们将在下一节讨论更多的例子。

部分运算

Haskell 的运算也可以部分地应用, 得到所谓的 **部分运算 (operator sections)**。例如,

`(+2)` 在参数上加 2 的函数。
`(2+)` 在参数上加 2 的函数。
`(>2)` 返回其参数是否大于 2 的函数。
`(3:)` 将 3 加到列表前的函数。
`(++"\n")` 在串尾加上换行符的函数。
`("\n"++)` 在串首加上换行符的函数。

这里的一般规则是部分应用的运算将把参数置于使运算完整的一边，即

```
(op x) y = y op x
(x op) y = x op y
```

当部分运算与高阶函数相结合时，如 map, filter 和复合等, 其表达力是相当强的, 而且非常简洁优美, 这种结合由此使我们能在函数层定义大量的函数。例如，

```
filter (>0) . map (+1)
```

这个函数在列表的每个元素上加 1，然后删除那些不大于 0 的数。

习题

10.12 利用部分应用定义 *10.3* 节及其习题中的函数`comp2`和`total`。

10.13 定义部分运算`sec1`和`sec2`使得

```
map sec1 . filter sec2
```

等同于下列函数

```
filter (>) . map (+1)
```

§10.5 再谈 Picture

在介绍完高阶函数，尤其是部分应用之后，我们再回过来考虑图形的例子并完成类型 Picture 上函数的定义。这个实例研究是在第一章介绍的，并且在 2.5 节和 6.1 节得到进一步发展。

一个图形是线段的一个列表，每条线段是一个字符列表:

```
type Picture = [[Char]]
```

我们首先定义在水平镜子中的反射，它可以通过将线段列表逆转来实现

```
flipH :: Picture -> Picture
flipH = reverse
```

要实现在垂直镜子中的反射，我们需要将每条线段逆转，显然可以用 map 实现

```
flipV :: Picture -> Picture
flipV = map reverse
```

将两个图形放在一起有两个函数。将一个图形放在另一个之上，可以通过将两个线段列表连接在一起来实现

```
above :: Picture -> Picture -> Picture
above = (++)
```

而将两个图形并排放置需要将两个列表的对应线段用 (++) 连接，这个函数可以通过 9.2 节介绍的函数 zipWith 实现

```
sideBiSide :: Picture -> Picture -> Picture
sideBiSide = zipWith (++)
```

其他的函数还包括

```
invertColor  :: Picture -> Picture
superimpose  :: Picture -> Picture -> Picture
printPicture :: Picture -> IO()
```

下面我们给出其定义。一个图形的反色需要每条线段反色，所以

```
invertColour = map ...
```

其中... 将是每条线段反色的函数。使一条线段反色仍然需要 map, 因为线段是字符的列表。这里 map 的参数是 6.1 节定义的函数 invertChar。由此得到下列定义

```
invertColour :: Picture -> Picturee
invertColour = map (map invertChar)
```

我们可以将上述定义读作：“应用 map invertChar 于 Picture 的每条线段，也就是将 invertChar 应用于 Picture 的每个字符，因为一个 Picture 是字符列表的列表”。

假设我们有一个函数

```
combineChar :: Char -> Char -> Char
```

此函数实现两个字符的重叠，那么如何实现两个图形的重叠呢？回顾函数

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

`zipWith f xs ys` 是将 f 应用于 xs 和 ys 的对应元素而构成的列表, 例如，

```
zipWith (*) [2,0] [3,1] = [6,0]
```

将两个图形叠加可以通过叠加相应的线段实现，所以

```
superimpose = zipWith ...
```

其中... 将两条线段叠加。为此，我们需要将相应字符叠加，这又需要使用 zipWith，而字符的叠加可以用 combineChar 实现，所以我们有下列定义

```
superimpose :: Picture -> Picture -> Picture
superimpose = zipWith (zipWith combineChar)
```

最后一个定义是将图形输出到屏幕上的函数 printPicture。输出一个串可以使用函数

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

所以，我们只需将表示图形的列表转换成线段间用换行符分隔的串，然后应用 putStr 即可。将表示图形的列表转换为串可以用下式实现

```
concat . map (++"\n")
```

其结果是先在每条线段之后加上换行符，map (++"\n")，然后用 concat 将一个串的列表连接成一个串。所以，我们可以如下定义输出函数

```
printPicture :: Picture -> IO()
printPicture = putStr . concat . map (++"\n")
```

习题

在本节的练习中我们将实现图形上的更多运算。

10.14 定义函数

```
chessBoard :: Int -> Picture
```

使得chessBoard n成为n行n列的棋盘。

10.15 如果Picture定义为[[Bool]]，如何实现invertColor，superimpose和printPicture。

10.16 定义一个函数

```
makePicture :: Int -> Int -> [(Int, Int)] -> Picture
```

其中的两个整数参数表示图形的宽度和高度，列表参数表示图形中黑点的位置。例如，

```
makePicture 7 5 [(1,3),(3,2)]
```

将生成下列图形

```
.......
...#...
.......
..#....
.......
```

显然，位置行列号均从 *0* 开始计数。

10.17 定义一个函数

```
pictureToRep :: Picture -> (Int, Int, [(Int, Int)])
```

它是makePicture的逆。例如，如果pic如下

```
....
.##.
....
```

则 pictureRep pic 将是 (4, 3, [(1,1),(1,2)])。

10.18 如果我们定义

```
type Rep = (Int, Int, [(Int, Int)]
```

试问在这种图形表示下如何定义旋转、反射和叠加函数。比较两种图形表示法的优点和缺点。

10.19 利用前面四章的知识重新解决 *6.2* 节有关定位图形的练习。

§10.6 更多的例子

本节探讨如何使用部分应用和部分运算化简其他例子。我们通常可以使用部分应用返回函数的方法免去显式定义函数。例如，第七章将列表元素加倍的函数可定义为

```
doubleAll :: [Int] -> [Int]
doubleAll = map (*2)
```

其中利用部分应用 map 直接给出函数的定义，并且利用部分运算 (*2) 代替了函数 double。

考虑选取数值列表中的偶数。我们需要检查除 2 的余数是否等于 0，这样的函数可表示为

```
(==0) . (`mod` 2)
```

这是两个部分运算的复合: 首先求除 2 的余数，然后检查它是否等于 0。(为何不能写作 (`mod` 2 == 0)?) 所以，过滤函数可定义为

```
getEvens :: [Int] -> [Int]
getEvens = filter ((==0) . (`mod` 2))
```

最后一个例子来自列表的折分。我们曾定义

```
getWord xs
  = getUntil p xs
    where
    p x = elem x whitespece
```

使用部分运算定义性质 p 后, 局部定义便可省去

```
getWord xs = getUntil (`elem` whitespace) xs
```

注意我们将一个函数部分应用于它的第二个参数形成部分运算。这种表示法是有效的，因为

```
(`elem` whitespace) x
= x `elem` whitespace
= elem x whitespace
```

最后，函数 getWord 可以利用部分应用直接定义

```
getWord = getUntil (`elem` whitespace)
```

这个定义看似一个非正式的解释: 要取一个词，取字符直至遇到一个空白字符。

§10.7 Currying 和非 Currying

在 Haskell 中二元或者多元函数的描述方式有两种。我们通常使用所谓的 **Curry 形式 (curried form)** 表示函数，即函数一次应用于一个参数。它

是以 λ- 演算的先驱 Haskell Curry 命名的[2], Haskell 语言也是以他命名的。例如，两个数相乘的函数通常定义为

```
multiply :: Int -> Int -> Int
multiply x y = x*y
```

而 **非 Curry(uncurrying)** 版本将两个参数合成为一个二元组:

```
multiplyUC :: (Int, Int) -> Int
multiplyUC (x, y) = x*y
```

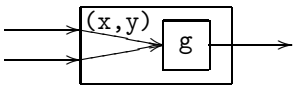
为什么我们经常选择 Curry 形式呢? 这里有几个理由。

- 这种记法更灵巧; 我们将一个函数应用于一个参数时, 用两个符号的并列表示 `f x`, 将函数应用于两个参数时只要将此记法扩展为 `g x y`;
- 这种记法允许部分应用。对于乘法，我们可以使用`multiply 2`表示乘 2 的函数，而在将两个参数在一起的情况是不可能的，如 multiplyUC。

在任何时候我们都可以在 Curry 形式和非 Curry 形式间转换，而且这种转换可以通过两个高阶函数实现。

首先, 假设我们要将类型为 `(a,b) -> c` 的非 Curry 函数 g 转换为 Curry 形式

`curry g`



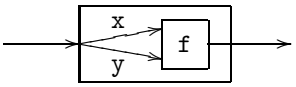
函数 g 需要一个二元组参数，而它的 Curry 版本, `curry g`, 将分别取其两个参数

```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry g x y = g (x,y)
```

`curry multiplyUC` 将等同于函数 multiply。

假设 f 是 Curry 形式, 类型为 `a -> b -> c`。

`uncurry f`



函数 `uncurry f` 的参数为二元组，所以在将 f 应用于参数之前必须将参数分开

```
uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry f (x,y) = f x y
```

`uncurry multiply` 将等同于函数 multiplyUC。函数 curry 和 uncurry 互为逆函数。

[2]事实上，第一次提出这个概念的是'Schönfinkeling', 但是'Schönfinkeling' 听起来并不那么爽快。

部分应用是将函数从左到右应用到它的参数上，所以一个函数不能直接应用到它的第二个参数上。我们可以间接地得到这样的结果: 先变换参数的次序，然后再使用部分应用。

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f x y = f y x
```

`flip map` 将变换 map 的参数次序，第一个参数为列表，第二个参数为函数，将其部分应用于第一个参数与 map 应用其第二个参数有相同的效果。

另一种构造部分应用 (`elem` whitespace) 的方法是使用 flip。我们有

```
flip elem :: [Char] -> Char -> Bool
```

(当然此函数有多个类型) 所以，我们可以构造下列部分应用

```
flip elem whitespace
```

下面我们转去讨论一个更复杂的例子，在这个例子中, 复合、部分应用和部分运算的思想以各种方式得到了应用。

§10.8 例子: 建立索引

本节探讨我们已经研究过的文本处理的另一个问题, 即如何自动生成一个文件的索引。通过这个例子我们将演示如何在最后的程序中使用高阶函数。多态函数可以应用于许多不同的类型，其函数参数表明在不同的场合可以有不同的效果。

为了缩短例子的长度, 我们讨论一个小规模的索引问题, 这仅仅是为了表述的方便，但是, 系统的所有重要方面都将得到探讨。

规格说明

我们首先说明程序要完成的任务。输入是一个文本串，其中文本行之间用换行符 `'\n'` 分隔。索引将给出一个词出现的所有行。我们只考虑至少含 4 个字母的词，索引结果按字母顺序排列。在每个索引项中，行号不应重复。例如，对于下列输入

```
"cathedral doggerel cathedral\nbattery doggerel cathedral\ncathedral"
```

我们希望得到下列索引

```
battery    2
cathedral  1,2,3
doggerel   1,2
```

程序设计

我们可以用一个列表表示索引，每个索引项为一个列表元素。那么索引项如何表示呢？一个索引项必须将文本中的某个词与它出现的一系列行号关联起来; 因此，我们可以把索引项表示为由一个类型为 `[Int]` 的行号列表和类型为 String 的词构成的二元组。创建索引的顶层函数将是

```
makeIndex :: Doc -> [([Int], Word)]
```

其中使用了下列类型别名以区别类型串在下列设计中的不同应用:

```
type Doc  = String
type Line = String
type Word = String
```

注意它们是相同的类型; 我们使用这些名字使其带有更多的信息: 例如, `'Line'` 的定义可读作 "String 视为文本行的表示"。

如何来设计程序呢? 我们将把重点放到程序生成的**数据结构**上, 并且将程序看作对输入数据的一系列变换。这种数据导引的设计在 Haskell 函数程序设计中是很普遍的。

在设计的顶层, 问题的解是一系列函数的复合。这些函数依次完成下列运算:

- 将文本 (一个 Doc) 折分, 形成类型为 [Line] 的对象;
- 给每个文本行标以它的行号, 结果为类型 [(Int, Line)] 的对象;
- 将文本行折分成词, 并将每个词与它所在的行关联, 结果是类型为 [(Int, Word)] 的列表;
- 将上述列表按词的字典序排列, 结果是同类型的列表;
- 修改上述列表使得每个词与不含重复行号的列表配对, 结果类型为 [([Int],Word)];
- 将同一词的索引项并入同一个行号列表, 结果为类型为 [([Int], Word)] 的列表;
- 删除那些少于 4 个字母词的索引项, 给出一个类型为 [([Int], Word)] 的列表。

下面给出定义。注意, 我们用注释给出向前复合中各分函数的定义:

```
makeIndex
  = line        >.>     -- Doc              -> [Line]
    numLines    >.>     -- [Line]           -> [(Int, Line)]
    allNumWords >.>     -- [(Int, Line)]    -> [(Int, Word)]
    sortLs      >.>     -- [(Int, Word)]    -> [(Int, Word)]
    makeLists   >.>     -- [(Int, Word)]    -> [([Int], Word)]
    amalgamate  >.>     -- [([Int], Word)]  -> [([Int], Word)]
    shorten             -- [([Int], Word)]  -> [([Int], Word)]
```

一旦给出函数的类型, 每个函数均可以独立开发。使用一个函数的唯一信息是它的类型, 这些类型已经在上述定义中说明。下面分别给出这些函数的定义。

分函数的实现

首先考虑将一个文本串折分为文本行的列表, 折分必须在换行符 `'\n'` 处进行。如何定义这个函数呢? 一种方法是编写类似于在 splitWord 中使用的函

数 getWord 和 dropWord。另一种方法是使用第七章介绍的函数 getUntil 和 dropUntil。第三种选择是使用引导库中已定义的函数 lines。我们将使用最后一种解法。

```
lines :: Doc -> [Line]
```

下一个函数给每行配以它的行号。如果用 linels 表示文本行的列表，则行号列表为

```
[1 .. length linels]
```

然后我们可以用拉链函数 zip 将两个列表结合成二元组的列表:

```
numLines :: [line] -> [(Int, Line)]
numLines linels
   = zip [1 .. length linels] linels
```

所有的文本行又需要折分成词，并配以行号。我们先考虑一行的折分:

```
numWords :: (Int, Line) -> [(Int, Word)]
```

我们可以将第七章的函数 splitWord 稍加修改来完成这项工作。在定义 splitWord 时我们保留了标点符号，因为它们将仍然出现在文本处理的输出中。相反地，在这里标点符号将被删除，所以空白字符包括标点符号。我们定义

```
whitespace :: String
whitespace = " \n\t;:.,\'\"!?()-"
```

然后我们给一个文本行的每个词配以相同的行号，这可以用 map 来完成:

```
numWords (number, line)
  = map (\word -> (number, word)) (splitWords line)
```

或者使用列表概括

```
numWords (number, line)
  =  [(number, word) | word <- splitWords line]
```

将上述运算应用整个文本，即将 numWords 应用于每一行，这可由 map 完成；最后将结果连接起来，这可以用 concat 实现

```
allNumWords :: [(Int, Line)] -> [(Int, Word)]
allNumWords = concat . map numWords
```

目前的结果是什么呢？文本被变换成一个行号和词对的列表，索引将在此基础上生成。例如，文本

```
"cat dog\nbat dog\ncat"
```

将被转换成

```
[(1, "cat"), (1, "dog"), (2, "bat"), (2, "dog"), (3, "cat")]
```

这个列表需要按词排序，生成一个词及其出现的行号的有序列表。行号与词对的顺序如下定义

```
orderPair :: (Int, Word) -> (Int, Word) -> Bool
orderPair (n1,w1) (n2,w2)
  = w1 < w2 || (w1 == w2 && n1 < n2)
```

其中词的顺序为字典序。如果两个二元组包含相同的词, 则按行号排序。

对列表排序可以使用 quicksort 的一个版本。列表被分成两部分, 一部分小于给定的元素, 另一部分大于给定的元素; 在每部分上分别排序, 然后将结果合在一起。

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

sortLs []     = []
sortLs (p:ps)  = sortLs smaller ++ [p] ++ sortLs larger
```

其中列表 smaller 和 larger 是 ps 中小于或者大于 p 的元素构成的列表。注意, 重复元素将被删除。在这里与 p 相同的元素被删除, 它们不会出现在 smaller 和 larger 之中。

如何定义以上两个列表呢? 它们是用某个性质从 ps 中选出的, 这可以用 filter 或者列表概括实现:

```
sortLs (p:ps)
  = sortLs smaller ++ [p] ++ sortLs larger
    where
    smaller = [q | q <- ps, orderPair q p]
    larger  = [q | q <- ps, orderPair p q]
```

将排序应用于前例得到

```
[(2,"bat"}, (1,"cat"), (3,"cat"), (1,"dog"), (2,"dog")]
```

同一个词的项应该收集在一起。首先将每一项转换为词与行号对的列表

```
makeLists :: [(Int, Word)] -> [([Int], Word)]
makeLists
  = map mklis
    where
    mklis (n, st) = ([n], st)
```

对于上例, 我们得到

```
[([2],"bat"}, ([1],"cat"), ([3],"cat"),
 ([1],"dog"), ([2],"dog")]
```

然后将与同一词相关的二元组合并为一个二元组。

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

amalgamate []   = []
amalgamate [p]  = [p]
amalgamate ((l1,w1):(l2,w2):rest)
  | w1 /= w2  = (l1,w1):amalgamate ((l2,w2):rest)       (amalg.1)
  | otherwise = amalgamate ((l1++l2, w1):rest)          (amalg.2)
```

前两个等式很简单, 第三个等式的作用如下

- 如果两个相邻项包含不同的词，即情况 (amalg.1)，则不需给第一项添加任何行号，只需合并尾部；
- 对于情况 (amalg.2)，两个相邻项包含相同的词，则将其合并，并在其结果上调用本函数。这是因为后面的项仍可能包含同一个词，需要进一步并入前面的项。

  考虑一个例子

```
amalgamate [([2],"bat"}, ([1],"cat"), ([3],"cat")]
⇝ ([2],"bat"): amalgamate [([1],"cat"), ([3],"cat")]   by (amalg.1)
⇝ ([2],"bat"): amalgamate [([1,3],"cat")]              by (amalg.2)
⇝ ([2],"bat") : [([1,3], "cat")]
⇝ [([2],"bat"), ([1,3], "cat")]
```

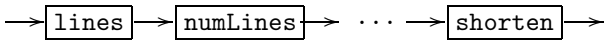
  为了满足设计要求，我们需要删除短词。

```
shorten
  = filter sizer
    where
    sizer (n1,wd) = length wd > 3
```

函数 filter 再一次证明了它的实用性。现在可以得到索引函数的完整定义如下

```
makeIndex :: Doc -> [([Int], Word)]
makeIndex
  = lines >.> numLines >.> allNumWords >.> sortLs >.>
    makeLists >.> amalgamate >.> shorten
```

如本节开始所述，函数复合是组织设计的有力方法：程序设计成传送数据的运算 **管道 (pipeline)**。



如何修改这样的设计是显然的。例如，上述索引程序在最后一步滤掉短词。在上图的运算链中，有多个位置可以实施这个运算。作为练习，留给读者考虑。

习题

10.20 使用第九章的函数getUntil和dropUntil，或者预定义函数takeWhile和dropWhile定义函数 lines。要注意，当文件中含有空行时，如，"cat\n\ndog"，"fish\n"，你的函数不要输出空字。

10.21 如何使用lambda表达式代替makeList和shorten中的局部定义？如何使用列表概括定义这些函数？

10.22 在本书的索引中，下列形式的索引项

```
cathedral        3,5,6,7,9,10
```

是用不同的区间表示的，如

```
cathedral    3, 5-7, 9-10
```

如何重新设计你的程序给出上述格式的输出。提示: 首先考虑新索引表示的类型，然后考虑在makeIndex的向前复合中增加一个函数。

10.23 如何重新定义sortLs使得重复元素不被删除。由此产生的结果是，如果一个词在同一行出现两次，如在 *123* 行，则 *123* 将在相应的索引项中出现两次。

10.24 如何在amalgamate的定义中使用函数getUntil和dropUntil。

10.25 试解释如何用预定义函数和部分运算定义局部函数sizer。函数sizer的作用是选取二元组的第二个分量，求出其长度，然后将结果与 *4* 比较。

10.26 在amalgamate的下列定义中，为什么第二个条件等式是错误的？用一个例子说明你的答案。

```
amalgamate ((l1, w1):(l2,w2):rest)
  | w1 /= w2    = (l1,w1) : amalgamate ((l2,w2):rest)
  | otherwise   = (l1++l2,w1) : amalgamate rest
```

10.27 定义一个函数

```
printIndex :: [([Int], Word)] -> IO()
```

它将索引如本节开始那样显示在屏幕上。你或许需要定义下列函数

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

10.28 修改程序使少于 *4* 个字母的词在函数allNumWords中被删除。

10.29 修改函数makeIndex使得结果不是一个词出现的行号列表，而是一个词出现的总次数。注意一个词在同一行的多次出现要计数。有两种方法解决这个问题:

- 尽可能少修改程序。例如，你可以返回一个列表的长度而不是列表本身。
- 以程序的结构为指导，编写一个 *(*更简单的*)* 程序直接计算出现的次数。

10.30 修改程序使得大写字母开头的词 *(*"Dog"*)* 列在小写字母开头的相应词 *(*"dog"*)* 的索引项中。但是，此规则不适用于专有名词，如 "Amelia"。对此你有何办法处理？

10.31 函数sortLs仅限于类型为 [(Int, Word)] 的列表，因为它调用了函数orderPair。重新定义排序函数使比较函数作为它的参数。新定义函数的类型是什么？

10.32 如果程序用于建立Haskell脚本的索引，如何修改程序？提示: 你需要考虑在这种情况下哪些可以忽略。

§10.9 验证与通用函数

对于高阶多态函数，验证具有不同的特征。我们可以证明函数的相等，而不仅是函数值的相等。我们还将看到，我们可以证明具有通用性和重用性，从而可应用于许多场合的定理。

函数层的验证

我们在 10.3 节指出函数 iter 是 twice 的推广，因为

```
iter 2 f
  = f . iter 1 f          by (iter.1)
  = f . (f . iter 0 f)    by (iter.1)
  = f . (f . id)          by (iter.2)
  = f . f                 by (compId)
  = twice f              by (twice.1)
```

在上述证明中我们使用了两个函数的相等

```
f .id  = f                                          (compId)
```

如何证明上述等式呢？我们可以检查等式两边作用于任意参数 x 的结果

```
(f .id) x
  = f (id x)
  = f x
```

可见两个函数在任意参数 x 上的作用是一样的。因此，两个函数作为黑盒是一样的。因为我们感兴趣的是函数的行为，所以我们说它们是相等的。我们把这种相等的 “黑盒” 概念称为 **外延的 (extensional)**。

定义 4 (外延原理) *如果两个函数作用于任意参数的结果相等，则这两个函数相等，。*

外延的相对概念是内涵。我们说两个函数内涵相等，如果它们有相同的定义。此时的函数不再是黑盒，因为我们可以察看函数的内部工作机制。如果我们关心的是程序的结果，那么重要的是函数给出的结果，而不是这些结果是如何求得的。所以我们在 Haskell 中对函数进行推理时将使用外延相等的概念。但是，如果我们关心的是程序的效率或者性能，那么求得结果的方法将是很重要的。我们将在第十九章对此做进一步讨论。

习题

10.33 *使用外延原理证明函数的复合满足结合律，即对于任意的*`f`，`g`*和*`h`

```
f . (g . h) = (f . g) . h
```

10.34 *证明对所有的*`f`

```
id . f = f
```

10.35 *证明 10.7 节定义的函数*`flip`*满足*

```
flip . flip = id
```

提示: *可以证明对任意*`f`

```
flip . (flip f) = f
```

10.36 *两个函数*`f`*和*`g`*互为逆，如果可以证明*

```
f . g = id              g . f = id
```

证明 *10.7* 节定义的函数curry和uncurry互为逆。你能举出其他逆函数对的例子吗？

10.37 利用归纳法证明, 对于任意自然数n

```
iter n id = id
```

10.38 一个函数f称为 **幂等的 (idempotent)**, 如果下式成立

```
f . f = f
```

证明函数abs和signum是幂等函数。你能举出其他幂等函数的例子吗？

高阶证明

到目前为止的验证仅限于一阶单态函数。正如 map 和 filter 推广了计算模式一样，我们将看到有关这些函数的证明也推广了我们已经得到的结果。举例来讲, 不难证明

```
doubleAll (xs++ys) = doubleAll xs ++ doubleAll ys
```

对于所有的 xs 和 ys 成立。当 doubelAll 定义为 `map (*2)` 时，显然上式是一个更一般性质的一个特例

```
map f (xs++ys) = map f xs ++ map f ys            (map++)
```

上式对任意函数 f 均成立。在以前的一个习题中我们曾指出

```
sum (xs++ys) = sum xs + sum ys                  (sum.3)
```

对于任意有穷列表 xs 和 ys 成立。函数 sum 是通过折叠(+)定义的:

```
sum = foldr (+) 0
```

而且, 如果 f 是可结合的, st 相对于 f 是单位元, 即对于任意 x, y 和 z

```
x `f` (y `f` z) = (x `f` y) `f` z
x `f` st = x = st `f` x
```

那么下列等式对所有的 xs 和 ys 成立:

```
foldr f st (xs++ys) = f (foldr f st xs) (foldr f st ys)   (foldr.3)
```

显然, (+)是可结合的，而且 0 是单位元，所以 (sum.3) 是 (foldr.3) 的特例。下面我们用同样的方式给出三个证明的例子。

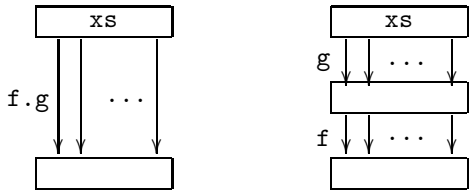
map 与复合

第一个例子是关于 map 与复合。先回顾它们的定义

```
map f []     = []                               (map.1)
map f (x:xs) = f x : map f xs                   (map.2)
(f .g ) x    = f (g x)                          (comp.1)
```

不难看出，我们可以证明下式对于任意有穷列表 xs 成立

```
map (f . g) xs     = (map f . map g) xs         (map.3)
```



将 (f . g) 应用列表的每个元素相当于将 g 应用于列表的每个元素，然后将 f 应用前面结果列表的每个元素。这一点很容易用结构归纳原理证明。归纳基需要证明等式对空列表成立:

```
map (f . g) [] = []     by (map.1)

(map f . map g) []
 = map f (map g [])    by (comp.1)
 = map f []             by (map.1)
 = []                   by (map.1)
```

假设下列归纳假设成立

```
map (f . g) xs     = (map f . map g) xs        (hyp)
```

我们需要证明

```
map (f . g) (x:xs) = (map f . map g) (x:xs)    (ind)
```

我们只需分析等式的两边

```
map (f . g) (x:xs)
  = (f . g) x : map (f . g) xs      by (map.2)
  = f (g x) : map (f . g) xs        by (comp.1)

(map f . map g) (x:xs)
  = map f (map g (x:xs))            by (comp.1)
  = map f (g x : map g xs)          by (map.2)
  = f (g x) : map f (map g xs)      by (map.2)
  = f (g x) : (map f . map g) xs    by (comp.1)
```

然后利用归纳假设证明等式两边相等，从而完成整个证明。 □

每个 Haskell 列表类型不仅包含有穷列表，而且包含无穷列表和部分列表。我们将在第十七章解释这些概念并证明 (map.3) 对了所有的列表成立。因此，下列函数等式成立

```
map (f . g) = (map f) . (map g)
```

map 与 filter

上述证明显示如何利用函数定义直接证明函数程序的性质。这些性质可以陈述程序的特性，如一个排序函数返回一个有序的列表，也可以陈述一个

程序与另一个程序之间的关系。后者是函数程序设计中的 **程序变换** 的基础。本节介绍一个称为过滤提升的例子，它是一个非常有用的基本函数变换。

```
filter p . map f = map f . filter (p . f)
```

等式表明：一个 map 跟随一个 filter 可以用一个 filter 跟随一个 map 代替。右式比左式的潜在效率高，因为 map 被应用于一个更短的列表，一个只有满足性质 `(p . f)` 的元素构成的列表。一个例子是 10.4 节定义的函数

```
filter (0<) . map (+1)
```

上述函数可以由下式代替

```
map (+1) . filter ((0<) . (+1))
  = map (+1) . filter (0<=)
```

显然，变换之后的版本效率更高，因为 `(<=)` 的测试比 `(<)` 的测试更容易。我们使用结构归纳法证明: 对任意有穷列表 xs 下式成立

```
(filter p . map f) xs = (map f . filter (p . f)) xs
```

首先列出 map, filter 和复合的定义:

```
map f []        = []                  (map.1)
map f (x:xs)    = f x : map f xs      (map.2)

filter p []     = []                (filter.1)
filter p (x:xs)
  | p x         = x : filter p xs   (filter.2)
  | otherwise   =     filter p xs   (filter.3)

(f . g) x        = f (g x)            (comp.1)
```

对于归纳基的情况，我们需要证明

```
(filter p . map f) [] = (map f . filter (p . f)) []       (base)
```

我们只需化简等式两边即可

```
(filter p . map f) []
  = filter p (map f [])          by (comp.1)
  = filter p []                   by (map.1)
  = []                          by (filter.1)

(map f . filter (p . f)) []
  = map f (filter (p . f) [])    by (comp.1)
  = map f []                    by (filter.1)
  = []                             by (map.1)
```

对于归纳步的情况，我们有下列归纳假设

```
(filter p . map f) xs = (map f . filter (p . f)) xs        (hyp)
```

需要证明

```
(filter p . map f)(x:xs)=(map f . filter (p . f))(x:xs)    (ind)
```

首先分析 (ind) 的左边

```
 (filter p . map f) (x:xs)
   = filter p (map f (x:xs))      by (comp.1)
   = filter p (f x : map f xs)     by (map.2)
```

我们需要考虑两种情况[3]: `p (f x)`为 True 或者 False。对于`p (f x)`为 True 的情况，继续化简 (ind) 的左边

```
   = f x : filter p (map f xs)            by (filter.2)
   = f x : (filter p . map f) xs            by (comp.1)
   = f x : (map f . filter (p . f)) xs        by (hyp)
```

下面化简等式的右边，同样假设 `p (f x)` 为 True

```
 (map f . filter (p . f)) (x:xs)
   = map f (filter (p . f) (x:xs))         by (comp.1)
   = map f (x: (filter (p . f) xs))      by (filter.2)
   = f x : map f (filter (p . f) xs)        by (map.2)
   = f x : (map f . filter (p . f)) xs     by (comp.1)
```

上述证明表示在 `p (f x)` 为 True 的情况下 (ind) 成立。

对于`p (f x)`为 False 的情况，可以类似地证明 (ind) 成立。由此完成了归纳步的证明，也结束了整个证明。 □

map, reverse 和 Picture 实例研究

在第一章介绍 Picture 实例研究时，我们断言：可以证明以两种顺序应用 flipV 和 flipH 的结果是相同的。两个函数的实现定义如下

```
filpH = reverse
flipV = map reverse
```

同时可以看出

- reverse 改变列表元素的顺序而不改变任何元素;
- `map reverse` 改变列表的每个元素，而不改变列表元素的顺序。

其中第二点是 map 函数的性质，而且我们还可以证明更普遍的性质，即对于所有的有穷列表 xs 和函数 f

```
map f (reverse xs) = reverse (map f xs)        (map/reverse)
```

如果用 reverse 代替 f, 则可推出

```
flipV (flipH xs) = flipH (flipV xs)
```

我们将在第十七章证明 `(map/reverse)` 对所有列表 xs 成立, 由此建立下列函数等式

```
map f . reverse = reverse . map f
flipV . flipH = flipH . flipV
```

[3]我们还应该考虑当`p (f x)`没有定义的情况，此时等式两边均无定义，故相等。

我们现在用归纳法证明 (map/reverse) 对于所有的有穷列表成立。

函数 map 的定义在前面已经给出。下面是 reverse 的定义:

```
reverse []     = []                                (reverse.1)
reverse (x:xs) = reverse xs ++ [x]                 (reverse.2)
```

命题 我们首先证明归纳基:

```
map f (reverse []) = reverse (map f [])            (base)
```

然后假定下列归纳假设

```
map f (reverse xs) = reverse (map f xs)            (hyp)
```

证明归纳步

```
map f (reverse (x:xs)) = reverse (map f (x:xs))    (ind)
```

归纳基的证明 化简 (base) 的两边

```
map f (reverse [])
  = map f []          by (reverse.1)
  = []                    by (map.1)

reverse (map f [])
  = reverse []            by (map.1)
  = []                by (reverse.1)
```

由此建立了等式 (base)。下面证明归纳步。

归纳步的证明 先化简 (ind) 的左边

```
map f (reverse (x:xs))
  = map f (reverse xs ++ [x])   by (reverse.2)
```

不难证明 (留给读者作为练习)

```
map f (ys++zs) = map f ys ++ map f zs              (map++)
```

我们利用 (map++) 可以继续化简 (ind) 的左式

```
  = map f (reverse xs) ++ map f [x]            by (map++)
  = map f (reverse xs) ++ [f x]       by (map.1), (map.2)
  = reverse (map f xs) ++ [f x]                  by (hyp)
```

再来看右式的化简

```
reverse (map f (x:xs))
  = reverse (f x : map f xs)        by (map.2)
  = reverse (map xs) ++ [f x]   by (reverse.2)
```

由此证明了 (ind) 成立, 从而完成 (map/reverse) 的证明。 □

定理库

由此可见, 我们可以证明通用函数如 map, filter 和 foldr 等的性质。它的意义是, 如果我们使用这些函数定义了一个新函数, 例如使用 map, 那么我们可以使用 map 性质的定理库, 包括下列对所有的有穷列表 xs 和 ys 成立的性质

```
map (f . g) xs        = (map f . map g) xs
(filter p . map f) xs = (map f . filter (p . f)) xs
map f (reverse xs)    = reverse (map f xs)
map f (ys++zs)        = map f ys ++ map f zs
```

我们已经看到，定义新函数时可以使用通用函数, 如 map, filter 等，而不必使用递归从头做起。同样地，我们在证明一个函数的性质时，可以引用定理库中的定理而不必重新写一个归纳证明。

习题

10.39 证明对于任意ys和zs下列等式成立*:*

```
map f (ys++zs) = map f ys ++ map f zs  (map++)
```

这是在证明有关map和reverse的定理时使用的一个性质。

10.40 假设f是可结合的，st是f的单位元 *(*这些概念的定义参见第 *173* 页*)*，试证明等式(foldr.3)，即对于所有的有穷列表xs和ys

```
foldr f st (xs++ys) = f (foldr f st xs) (foldr f st ys)
```

10.41 试说明下列结果是(foldr.3)的特例

```
concat (xs ++ ys) = concat xs ++ concat ys
```

其中concat的定义为

```
concat = foldr (++) []
```

10.42 证明对于所有的有穷列表xs和函数f

```
concat (map (map f) xs) = map f (concat xs)
```

10.43 试在类型Int上证明

```
(0<) . (+1) = (0<=)
```

这是在有关map和filter的定理中用到的性质。

10.44 试证明对于任意有穷列表xs下式成立*:*

```
filter p (filter q xs) = filter (p &&& q) xs
```

其中 &&& 的定义为

```
p &&& q = \x -> (p x && q x)
```

小结

我们在本章介绍了如何定义结果为函数的函数。其意义在于我们可以在程序中使用 map, filter 和 foldr 等运算创建函数，而且在函数程序设计语言中可以把函数作为“一等公民”来对待。其结果是我们可以解释第一章中 Picture 上某些运算的定义。

本章介绍的创建函数的主要机制是将函数或者运算应用于少于规定数目的参数，形成部分应用或者部分运算。我们还介绍了 Haskell 的类型系统和语法如何适应多元函数依次应用于其参数的 Curry 形式定义。

本章最后展示了我们可以证明通用函数的一般性质，并且可以建立由这些性质构成的定理库。当通用函数被重用时，有关的定理也可以被应用。

第十一章　程序开发

这一小章将从 Haskell 程序设计的细节转向在第四章的基础上进一步讨论程序开发的周期。尽管有些是针对 Haskell 的，但多数内容具有普遍性，同样适用于使用其他语言的程序开发。

对于程序开发中的理解、设计、实现和回顾 4 个阶段如何进行，我们给出一个图表说明。这 4 个阶段很大程度上来自于 Polya 提出的解决数学问题的方法。最后，我们对前述建议给出 Haskell 示例。

§11.1　开发周期

我们可以把程序开发视为一个环形过程。

首先，我们必须**理解程序设计问题**，即我们要解决的问题。这是我们在第四章讨论过的内容。完成这个任务后，我们可以**计划**或者**设计**如何利用程序设计语言的所有资源、语言的库和已有的程序来解决面对的问题。在第四章我们曾指出，在进入下一个编写程序阶段之前，设计阶段是至关重要的。

一旦完成程序的编写，我们可以 **反思** 或者 **回顾** 程序是否达到了其设计目标：我们可以测试程序，如第四章所讲，也可以设法证明程序的性质。此时我们也可以回顾原问题 – 这是不是用户想要解决的问题，或者面对程序的运行，用户是否需要改变问题的说明。

四个阶段构成的顺时针环形是实践中从练习程序到大型工业项目的程序开发的简化模型。即使对于我们在本书中编写的程序，回顾我们所做的也是非常重要的活动。在图中我们用虚线箭头表示这些活动。

- 在设计程序的过程中，我们进一步完善程序的说明：或许我们发现遗漏的情况，或者规格说明没有回答的问题，此时我们需要返回去修改规格说明，解决发现的问题；
- 在设计过程中我们或许有几种解决问题的方法：我们需要考虑哪一种方

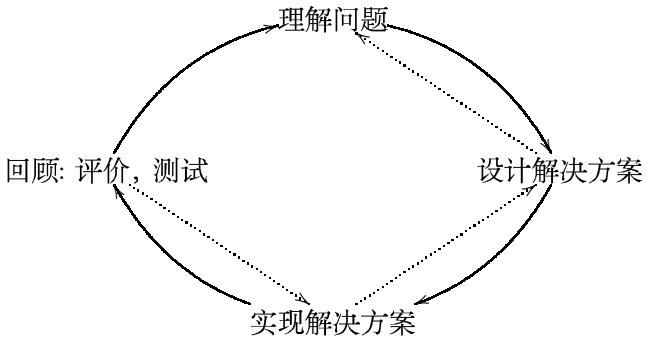


图 11.1：开发周期

法更好；如果一种选择行不通，我们不得不改变我们的方法;

- 在编写程序的过程中，我们或许会发现如何改进设计。例如，在 10.8 节建立索引的例子中: 我们一直把短词保留到复合的最后阶段，实际上我们可以在很早的时候舍弃它们，例如，在把文本行折分成词的时候。

特别地，我们在学习程序设计时，批评自己的程序或者别人的程序是一个好习惯。确实，有的时候我们写完程序后发现，我们对问题和解决问题的方法有了更深刻的理解，因此，为了反映我们的新认识，使得问题的解更清晰，我们舍弃第一个解，并重新编写新的解。

对于如何编写程序很难给出一般性的建议，不过，图表 11.1 包含了一些重要的思想。这里的建议受到 Polya 解决数学问题的方法的强烈影响。Polya 的著作《如何解题》(Polya 1988) 包含如何寻求问题之解的大量建议，其中的许多建议也适用于程序设计。错误记录是 Humphrey 所著《个人软件进程》(Humphrey 1996) 的一部分。

在下一节，我们将以 Haskell 程序设计例子说明开发周期中的某些思想。

理解问题

首先，我们需要理解要解决的问题。

- 问题的输入和输出是什么？对于输入和输出有无特殊条件？
- 研究问题的一些特例有助于澄清问题。
- 问题能解决吗？问题的说明完整吗？有无需要进一步澄清的地方？
- 如果问题说明有不同的理解方法，设法弄清说明者的原意是什么。
- 问题本身有结构吗？问题是否由几个部分构成，而且每个部分可以分别解决？框图是否有助于描述问题？

设计一个解

在编写程序之前，我们需要计划如何解决问题。

- 以前遇到过类似的问题吗？如果遇到过类似的问题，你可以借此帮助解决问题。
- 你能解决一个更简单而且相关的问题吗？如果能解决这样的问题，你或许可以使用这个解或通过修改这个解来获得原问题的解。
- 你能把问题推广吗？解决推广的问题或许比原问题更容易。
- 问题的体系结构是什么？你能把问题分解成几部分，每部分能够 (相对) 独立解决吗？除解决部分问题外，你还需要考虑如何把部分解合并成为整个问题的解。
- 考虑如何把输入转化为输出。这是一个由底向上的过程。在这个过程中，可以使用中间数据为指导。还可以考虑有了哪些资源后你便可以解决问题，这种"假设" 方法称为由顶向下的方法。
- 即使在计划阶段，了解你拥有的资源也是很重要的。要弄清楚程序设计语言及其库提供了什么。另一个重要资源是你自己所写的程序。

- 设计时要考虑到将来的修改。如果你的程序是实用的，那么在它的生命周期中它可能会被多次修改。

编写程序

在编写程序之前，你需要明白你使用的程序设计语言提供哪些资源。你还需要遵守非正式的设计或计划。

- Haskell 提供了大量的支持列表上程序设计的库函数。有些函数是通用的多态高阶函数，可用于多种情形；要尽可能使用它们。
- 我们将看到，在其他数据类型上我们也可以定义类似的通用函数。使用这些函数往往比一切从头开始编写程序更容易。
- 通用函数是由特例抽象来的。特别地，特例 – 如乘 2 – 可以变为通用函数的函数参数，如 map。
- 大多数语言允许分情形定义；Haskell 也提供模式匹配，用以区分不同的情形和选取一个对象中的成分。
- 递归是列表和整数数据类型上程序设计的一般方法。要定义一个递归函数 f 在参数 x 上的值，你需要考虑" 如果已知 f 在…的值，那么如何求 f 在 x 上的值呢？"。
- 列表概括是列表的一种表现力很丰富的表示法。
- 在开始定义函数时，你可能需要引进其他函数。这些函数可能由 where 子句定义，也可能在程序的顶层定义。
- 如果你不能定义你需要的函数，那么先定义一个比较简单的函数。简单函数可能成为你的最终函数的模型，或者被应用于最终函数的定义中。

回顾

回顾你所完成的工作可能会影响你的程序、程序的设计以及问题本身的说明。

- 你能测试你的解吗？你需要认真考虑所有的特殊情形，以及程序有相似表现的测试分组。
- 如果你的测试发现了错误，设法找出其来源。错误来源于偶然的错误吗？还是来源于对 Haskell 语言的理解问题？还是对问题之解的要求的误解？还是对问题本身的误解？还是其他原因？
- 从错误之中吸取教训。设法记录你所犯的错误及造成错误的原因。这样便于帮助你今后不再重复这样的错误。
- 你能证明你的程序完成它本应完成的工作吗？如果不能，试问为什么，由此找出程序中的错误或者设计中的错误。
- 假如你重新编写同一个程序，你会用不同的方式来做吗？
- 假如要求你修改或者扩展你的程序，这个任务是否容易？如果这个任务很困难，你能考虑如何设计与编写程序使得修改或扩展更容易吗？
- 程序的运行时间是否合理？如果不合理，你能找到瓶颈在哪里吗？你能

看出如何修改程序使之效率更高吗？

§11.2 实践中的程序开发

本节将通过一系列编程例子解释第四章及图 11.1 中的建议。

推广问题

假设我们要自己定义列表 [1 .. n]。或许第一次尝试是

```
[1 .. n]=1: [2 .. n]                    (..1)
```

但是，[2 .. n] 并不是我们要定义的列表的特例。其中 2 的出现提醒我们应该解决更广的从任意值开始的列表，而不是从特殊值开始的列表。为此，我们定义 [m .. n]:

```
[m .. n]
  | m>n        = []                     (..2)
  | otherwise  = m : [m+1 .. n]
```

另一个解是

```
[1 .. n]
  | 1>n        = []
  | otherwise  = [1 .. n-1] ++ [n]      (..3)
```

但是，(..3) 的效率比 (..2) 低的多。我们将在第十九章讨论计算的效率问题。

小结

本章讨论了环形的程序开发过程：首先我们清楚地陈述要解决的问题，然后设计一个解决问题的方案，最后实现这个解。

在每个阶段中，我们应该回顾和评价我们所做的：对于学习程序设计者来讲，这一点尤其重要。例如，弄清我们所犯的错误有助于我们今后避免犯这样的错误。此外，如果我们设法用新技术来解决过去已经解决的问题，那么我们不仅能学到新的技术，而且能理解新技术与已有知识间的联系。这也是我们不断地返回来讨论 Picture 实例研究的目的。

第十二章　重载和类型分类

我们已经看到两种可以应用到多个类型上的函数。一种是**多态**函数，如length，这种函数有一个适用于所有类型的定义。另一种是**重载**函数，如相等，+ 和 show 等可应用于不同的类型，但在不同的类型上有不同的定义。

本章首先讨论重载的优点，然后讨论**类型分类**。一个类型分类是一些类型的集合，其成员的共同点是均定义了某些函数。例如，相等类型分类 Eq 的成员是定义了相等函数 == 的类型。因此，类型分类是赋予重载函数类型的一种机制。

我们将介绍如何定义类型分类和属于这些分类的类型，我们称之为分类的特例。我们还将看到与面向对象程序设计相关的分类之间的某种继承性。我们将在第十六章继续讨论继承性。

Haskell 引导库和函数库包含一些分类以及特例，特别是数值类型。我们将对此做简单总结，详细内容请读者参阅 Haskell 报告 (Peyton Jones and Hughes 1998)。

§12.1　为什么使用重载？

本节通过一系列例子讨论在 Haskell 中引入重载的原因。

假设 Haskell 设有重载，而且我们想检查一个特定元素是否一个 Bool 列表的一个成员。我们将定义如下函数：

```
elemBool :: Bool -> [Bool] -> Bool
elemBool x []  = False
elemBool x (y:ys)
    = (x ==Bool y || elemBool x ys)
```

其中 $==_{\texttt{Bool}}$ 表示 Bool 上的相等函数。

假设我们想检查一个整数是否为一个整数列表的成员，我们需要定义一个新的函数

```
elemInt :: Int -> [Int] -> Bool
```

它与 elemBool 的区别仅在于使用了 $==_{\texttt{Int}}$ 代替了 $==_{\texttt{Bool}}$。每次检查一个不同类型列表的属于关系，我们必须再定义一个非常类似的函数。

解决这个问题的一种办法是将相等函数看作一个通用函数的参数

```
elemGen :: (a -> a -> Bool) -> a -> [a] -> Bool
```

但是这样的函数在某种意义上太通用，因为它使用类型为`a -> a -> Bool`的任意参数，而不仅仅是检查相等的函数。另外，每次使用这个函数必须明确地写出这个参数，例如

$$\texttt{elemGen (}\ ==_{\texttt{Bool}}\texttt{)}$$

这使得程序难于阅读。

另一种办法是定义一个使用重载相等的函数

```
elem :: a -> [a] -> Bool
```

其中类型 a 必须限于那些有相等运算的类型。这种方法的优点是

- **重用**：函数 elem 的定义可用于具有相等运算的所有类型。
- **可读性**：==较 $==_{\texttt{Int}}$ 等易读性好得多。尤其是对于数值运算符，使用 $+_{\texttt{Int}}$，$*_{\texttt{Float}}$ 等令人厌烦得多。

上述讨论表明我们需要一种机制给类似于 elem 的函数赋予类型，这正是类型分类的目的。

§12.2 引进类型分类

函数 elem 似乎具有下列类型

```
elem :: a -> [a] -> Bool
```

但是这个类型仅适用于具有相等函数的类型。如何表示这种限制呢？我们需要用某种方式说明在一个类型上是否定义了相等运算。我们把那些定义了某一个函数的类型的集合称为 **类型分类 (type class)** 或简称类。例如，定义了相等 == 的类型的集合称为相等类，记作 Eq。

定义相等类

如何定义一个类型分类，如 Eq 呢？我们需要说明一个类型属于一个类型分类的条件。一个类型 a 属于 Eq 类，如果 a 上定义了一个类型为`a->a->Bool`的函数 ==。

```
class Eq a where
  (==) :: a -> a -> Bool
```

一个类型分类的成员称为它的 **特例 (instances)**。Eq 包含的预定义特例包括基类型 Int，Float，Bool 和 Char。其他特例包括由 Eq 特例构成的多元组和列表，例如 (Int,Bool) 和 [Char]。

并非所有的类型具有相等运算。例如，出于信息隐蔽的考虑，我们选择不定义相等；或者在某个类型上不存在自然的相等定义，例如，函数类型`Int->Int`不是 Eq 的特例，因为不存在算法判定 Int 上的两个函数是否相等。

不幸的是，特例这个术语在 Haskell 中有两种不同的含义。在 5.7 节我们称一个类型 t_1 是另一个类型 t_2 的特例，如果 t_1 可以在 t_2 中做一个变量替换得到。在这里我们说一个类型是一个分类的特例。

使用等式的函数

许多函数使用了特定类型上的相等。下列函数判定三个整数是否全等：

```
allEqual :: Int -> Int -> Int -> Bool
allEqual m n p = (m==n) && (n==p)
```

我们检查定义时发现，定义并不包含整数的特殊性质，唯一的约束是比较 m,n 和 p 是否相等。它们的类型可以是 Eq 类中的任何类型。由此得到 allEqual 的最广类型：

```
allEqual :: Eq a => a -> a -> a -> Bool
allEqual m n p = (m==n) && (n==p)
```

符号=>前面的部分称为 **上下文 (context)**。我们可以将上述类型读作：”如果类型 a 属于 Eq 类，即 a 上定义了==，则 allEqual 具有类型`a -> a -> a -> Bool`”。这表明，allEqual 可用于下列类型

```
allEqual :: Char -> Char -> Char -> bool
allEqual :: (Int,Bool) -> (Int,Bool) -> (Int,Bool) -> Bool
```

因为 Char 和 (Int,Bool) 均属于 Eq。如果我们不顾这个约束，比较函数的相等会发生什么呢？如果我们定义

```
suc :: Int -> Int
suc = (+1)
```

并试图计算

```
allEqual suc suc suc
```

我们将得到下列信息

```
ERROR: Int -> Int is not an instance of class "Eq"
```

它表示`(Int -> Int)`不属于 Eq 类，因为它不是 Eq 的一个特例。

更多相等的例子

12.1 节的例子 elem 将获得类型

```
elem :: Eq a => a -> [a] -> Bool
```

此函数可以用于下列类型

```
 Bool -> [Bool] -> Bool
 In -> [Int] -> Bool
```

我们定义的许多函数已经使用了过载的相等。我们可以利用 Hugs 系统推导一个函数的最广类型，如 5.6 节中图书馆数据库的 books 函数，先将 books 的原类型注释

```
-- books :: Database -> Person -> [Book]
```

然后在提示符下键入

```
:type books
```

结果为

```
books :: Eq a => [(a,b)] -> a -> [b]
```

或许初看起来有点惊奇。如果我们重新书写它的定义，其类型会变得明显。将 books 重命名为 lookupFirst，因为它要查找所有第一个分量是为特定值的二元组，并返回相应的第二个分量。其定义如下

```
lookupFirst :: Eq a => [(a,b)] -> a -> [b]
```

```
lookupFirst ws x
  = [z | (y,z) <- ws, y==x]
```

从定义明显看出，此函数不只适用书或者人，它只要求在二元组的第一个分量上可以比较相等即可，所以它是多态函数。其中条件用上下文`Eq a`表达。类似地，5.6 节的其他函数也是多态的

```
borrowed     :: Eq b => [(a,b)] -> b -> Bool
numBorrowed  :: Eq a => [(a,b)] -> a -> Int
```

小结

本节引进了类的概念。一个类是一些类型，或者它的特例的集合，在这些类型上均定义了某些函数。我们可以把一个类视为一个**形容词**：任意类型或者属于这个类，或者不属于这个类，正如在某一时刻的天气或者是晴朗的，或者不是晴朗的。

我们看到如何将相等运算视为在 Eq 类的所有类型上定义的函数。因此，以前定义的许多函数可以视为多态函数，可用于 Eq 类的任何类型。

从下节开始，我们介绍如何定义类和特例，并且探讨类在 Haskell 程序设计中所带来的影响。

习题

12.1 如何利用相等运算==定义不相等运算/=？其类型是什么？

12.2 定义函数numEqual，其输入为一个列表xs和另一个参数x，函数返回x在xs中出现的次数。函数的类型是什么？如何用numEqual定义member？

12.3 定义函数

```
oneLookupFirst  :: Eq a => [(a,b)] -> a -> b
oneLookupSecond :: Eq b => [(a,b)] -> b -> a
```

函数oneLookupFirst的第一个输入是一个二元组列表xs，第二个参数为x，输出是列表xs中第一个分量等于x的第一个二元组相应的第二个分量。解释如果列表中不存在这样的二元组时，函数会输出什么？函数oneLookupSecond返回列表中第二个分量等于x的第一个二元组的第一个分量。

§12.3 Signatures 和特例

我们在上一节看到相等运算==被重载。这使得==可用于许多类型，也使得使用==的函数可以定义在 Eq 类的所有特例上。本节解释如何引入类以及声明它的特例。由此我们不仅可以使用 Haskell 的预定义类，还可以使用自定义的类。

定义一个类

如前所见，一个类是由如下的声明引入的

```
class Visible a where
  toString :: a -> String
  size     :: a -> Int
```

上述声明定义了一个类，其名为 Visible(可视)，然后是构成类的 signature 的函数声明，说明 Visible 类的成员必须定义两个函数：

- 函数 toString，它将类型 a 的对象转换为一个 String，
- 函数 size，它返回一个整数作为参数大小的度量。

我们可以通过函数`toString`察看可视类的对象，还可以用`size`度量对象的大小：列表的大小可以是其长度，一个布尔值的大小可以是 1。一个类的一般定义形如

```
class Name ty where
  ... 有关类型变量ty的signature
```

那么，一个类型如何成为一个类的特例呢？

定义一个类的特例

一个类型上定义了一个类的 Signature 函数便可声明为该类的成员或特例。例如，

```
instance Eq Bool where
  True  == True   = True
  False == False  = True
  _     == _      = False
```

上述特例声明说明 Bool 是 Eq 类的一个特例。数值类型 Int 和 Float 等属于 Eq 类以及其他预定义类的特例声明涉及到系统实现所提供的原始相等函数。尽管我们称 Eq 类为相等类，但是并未要求函数`==`具有相等的一般性质，只要求其类型与相等的类型相同。如何定义`==`，完全取决于用户。再拿上例来说，我们可以定义

```
instance Visible Char where
  toString ch  = [ch]
  size _       = 1
```

它表明如何将字符转换为串，并度量其大小。在这里，字符被转换成长度为 1 的串。我们也可以将 Bool 声明为 Visible 的一个特例：

```
instance Visible Bool where
  toString True  = "True"
  toString False = "False"
  size _         = 1
```

假设类型 a 是可视的: 这表明我们可以度量 a 中值的大小并将其转换为一个串。利用 a 上的这些函数，我们可以定义 [a] 上的相应函数，所以我们可以声明如下特例

```
instance Visible a => Visible [a] where ...
```

其中使用了**上下文**`Visible a`，表明我们只把列表元素可视的列表可视化。最后，通过定义 Signature 函数完成特例声明：

```
instance Visible a => Visible [a] where
  toString = concat . map toString
  size = foldr (+) 1 . map size
```

将 a 上的一个列表转换成 String，我们把其中的每个元素转换为一个串 (map toString) 并使用 concat 将它们连接在一起。类似地，a 上一个列表的大小为所有元素大小 (map size) 之和加 1，可以通过 foldr (+) 1 实现。

在上面函数定义的右边，我们使用了类型 a 上的函数 toString 和 size，这表明上下文"a 是一个 Visible 类型" 是必要的。

Haskell 对于哪些类型可以声明为特例，即哪些类型可以出现在`=>`(在有上下文的情况) 后面有一些限制。声明为特例的类型必须是一个基类，如 Int，或者类型构造符应用于不同的类型变量，如 [...] 或者 (...,...)。例如，我们不能声明 (Float, Float) 为特例，也不能使用命名类型 (用 type 定义的类型)。更详细的说明参见 Haskell 报告 (Peyton Jones and Hughes 1998)。我们将在介绍了类型构造符后，讨论更多有关类的例子。

缺省定义

Haskell 的相等类实际上是如下定义的

```
class Eq a where
  (==),(/=) :: a -> a -> Bool
  x/=y     = not (x==y)
  x==y     = not (x/=y)
```

除相等运算外，Eq 还包含一个不相等运算`/=`。此外，`==`和`/=`都有一个缺省定义。这些定义有两个目的：它们提供了所有相等类型上的定义，但是，它们可以被特例声明所覆盖。

对于每个特例，至少应该提供`==`和`/=`其中之一，根据缺省定义，一个定义足以同时提供两个定义。

在一个特例声明中，也可以同时定义两个运算。例如，如果我们想在 Bool 上定义一个不同于缺省定义的`/=`，我们可以在特例声明中加上下面的定义

```
  x/=y  = ... 自己的定义 ...
```

如果不想覆盖缺省定义，可以将该运算的定义置于顶层，而不是放在 Signature 中。对于 Eq 类，如果不想`/=`的定义被覆盖，我们将下列定义置于顶层

```
(/=) :: Eq a => a -> a -> Bool
```

```
x /= y  = not (x==y)
```

这样的定义在所有定义了==运算的类型上有效。

对于有些情况，使用可覆盖的缺省定义是更好的选择，而不是不可覆盖的顶层定义。例如，在数值类型上，一个实现完全可以提供基于硬件指令的运算，其效率比缺省定义高得多。

派生类

函数和特例可以依赖于某些类；对于类也有同样的情形。Haskell 中最简单的例子是有序类型的类 Ord。一个类型属于有序类，除相等运算外，还需要提供运算>, >=等。我们记作

```
class Eq a => Ord a where
 (<),(<=),(>),(>=) :: a -> a -> Bool
 max, min          :: a -> a -> a
 compare           :: a -> a -> Ordering
```

一个类型 a 属于 Ord 类，我们不仅要提供 Eq 定义中的运算，还要定义 Ord 的其他运算。由<的定义，我们可以定义其他运算的缺省定义。例如，

```
x <= y     = (x < y || x == y)
x >  y     = y < x
```

我们将在 12.4 节解释类型 Ordering 及其函数 compare。

在 Ord 类的类型上定义的简单函数例子是第七章的插入排序函数 iSort，其最广类型为

```
iSort :: Ord a => [a] -> [a]
```

事实上，任何使用基于<=的排序函数均具有上述类型。

从另一种观点看，我们可以认为 Ord 类 **继承** 了 Eq 的运算；继承是面向对象程序设计的一个中心思想。

多个约束

前面看到的上下文只有一个约束条件，如`Eq a`。很自然，在类型上可以有多个约束。本节介绍多个约束出现的例子和表示法。

假如我们想对一个列表排序，然后将结果显示为串，我们可以将函数定义为

```
vSort = toString . iSort
```

要将元素排序，列表的元素类型必须属于 Ord 类。要将结果转化为一个 String，类型 [a] 必须属于 Visible，参照第 188 页有关 [a] 的特例说明，我们知道只要 a 属于 Visible 即可。因此有

```
vSort :: (Ord a,Visible a) => [a] -> String
```

这表明类型 a 必须既属于 Ord 类又属于 Visible 类。这样的类型包括 Bool 和 [Char] 等。

类似地，如果我们想把 lookupFirst 的结果可视化，则可写成

```
vLookupFirst xs x = toString (lookupFirst xs x)
```

此时在列表类型 [(a,b)] 上有双重约束。我们需要比较列表的第一部分，所以需要约束 Eq a。我们还需要将第二部分转化为串，即需要 Visible b。由此得到类型

```
vLookupFirst :: (Eq a, Visible b) => [(a,b)] -> a -> String
```

多重约束可以出现在特例声明中，例如

```
instance (Eq a, Eq b) => Eq (a,b) where
  (x,y) == (z,w) = x==z && y==w
```

这里的声明表示 Eq 的类型构成的二元组类型也属于 Eq。多重约束也可以出现在类的定义中，

```
class (Ord a,Visible a) => OrdVis a
```

在这样的声明中，类 OrdVis 继承了 Ord 和 Visible 的运算。

在上面这个特殊情况下，类型声明包含一个空的 Signature，一个类型属于 OrdVis，只要 a 属于 Ord 并且属于 Visible 即可。因此，我们可以修改 vSort 的类型为

```
vSort :: OrdVis a => [a] -> String
```

当一个类建立在两个以上的类之上时，我们称之为 **多继承**。多继承将对编程方式产生影响，我们将在 14.6 节探讨之。

小结

本节解释了 Haskell 类机制的详细内容。我们看到，一个类的定义声明了一个 Signature，定义该类的特例时需要提供 Signature 中每个运算的定义，这些运算的定义将覆盖 Signature 中的缺省定义。上下文是对出现在多态类型、特例声明和类声明中类型变量的约束。

习题

12.4 如何使得Bool，二元组类型(a,b)和三元组类型(a,b,c)成为Visible类型？

12.5 定义一个将整数转换为String的函数toString，然后说明Int为Visible的特例。

12.6 说明下列函数的类型

```
compare x y = size x <= size y
```

12.7 完成类Ord的缺省定义。

12.8 完成下列特例说明

```
instance (Ord a,Ord b) => Ord (a,b) where ...
instance Ord b => Ord [b] where ...
```

其中二元组和列表上的顺序为字典序。

§12.4 Haskell 的预定义类

本节简单介绍 Haskell 的预定义类。许多类是为了处理数值类型上的重载数值运算而定义的，如整数类型，浮点实数，复数和有理数 (整数分数，如 $\frac{22}{7}$) 等。我们将列出这些数值类型的主要特征，忽略其细节。

相等类 Eq

我们重新给出 Eq 类的定义如下：

```
class Eq a where
  (==),(/=) :: a - >a -> Bool
  x /= y  = not (x == y)
  x == y  = not (x /= y)
```

有序类 Ord

类似地，我们在 Eq 类上建立有序类

```
class (Eq a) => Ord a where
  compare              :: a -> a -> Ordering
  (<),(<=),(>),(>=)    :: a -> a -> Bool
  max, min             :: a -> a -> a
```

类型 Ordering 包括三个值 LT，EQ 和 GT，分别表示比较两个元素的三个可能结果。我们将在第十四章第 212 页介绍 Ordering 是如何定义的。

使用 compare 的优点是一次函数应用便可确定两个输入之间的确切关系，而使用返回布尔值的次序运算符时，需要两次比较。事实上，我们看到在使用==和<=的 compare 缺省定义中，需要两次比较才能确定其值是 LT 还是 GT。

```
compare x y
  | x == y     = EQ
  | x <= y     = LT
  | otherwise  = GT
```

缺省定义还包括利用 compare 定义的次序运算符

```
x <= y     = compare x y /=GT
x <  y     = compare x y == LT
x >= y     = compare x y /= LT
x >  y     = compare x y == GT
```

实际上，Ord 包含所有运算的缺省定义，但在定义其特例时，只需提供 compare 或者<=即可。最后，最大值和最小之运算的缺省定义如下

```
max x y
  | x >= y     = x
  | otherwise  = y
```

```
min x y
  | x <= y     = x
  | otherwise  = y
```

许多 Haskell 类型属于相等类和有序类，但是，函数类型和第十六章介绍的抽象数据类型除外。

枚举类

我们可以使用枚举表达式生成列表，例如`[2,4..8]`。枚举也可以在其他类型上进行：字符型，浮点数等。类的定义为

```
class (Ord a) =>  Enum a where
  toEnum          :: Int -> a
  fromEnum        :: a -> Int
  enumFrom        :: a -> [a]             -- [n..]
  enumFromThen    :: a -> a -> [a]        -- [n,m..]
  enumFromTo      :: a -> a -> [a]        -- [n..m]
  enumFromThenTo  :: a -> a -> a -> [a]   -- [n,n'..m]
```

其中 enumFromTo 和 enumFromThenTo 的缺省定义留作练习。

类的 Signature 还包括在此类型和 Int 间转换的运算 fromEnum 和 toEnum。对于类型 Char，这些转换函数称为 ord 和 chr，可以如下定义

```
ord :: Char -> Int
ord = fromEnum

chr :: Int -> Char
chr = toEnum
```

枚举类的特例包括 Int，Char，Bool 和其他有限类型如 Ordering。在这些类型上可以定义前驱和后继函数

```
succ, pred :: Enum a => a -> a

succ = toEnum . (+1) . fromEnum
pred = toEnum . (subtract 1) . fromEnum
```

有界类型 Bounded

有界类的声明如下

```
class Bounded a where
  minBound,maxBound :: a
```

而且这两个值给出这些类型的最小和最大值。类型 Int，Char，Bool 和`Ordering`属于有界类。

把值转换为串的类 Show

在介绍类型分类时，我们介绍了 Show 类，它包含其值可以转换为串的类型。

```
type ShowS = String -> String

class Show a where
  showsPrec :: Int -> a -> ShowS
  show      :: a -> String
  showList  :: [a] -> ShowS
```

函数 showPrec 支持大型数据的灵活有效转换。但是，在一般情况下，只要定义将值转换为串的函数 show 即可。Show 类包含了 showPrec 和 show 相互间的缺省定义。有关 showPrec 的更多细节参见 Hudak, Fasel 和 Peterson (1997)。

多数类型属于 Show 类，即使类型的值不能够完全显示出来，我们也可以用某种文本来表示。例如，一个函数可以表示为`<<function>>`。对于其他类型，特例声明可以是

```
instance Show Bool where
  show True  = "True"
  show False = "False"

instance (Show a,Show b) => Show (a,b) where
  show (x,y) = "("++show x ++","++ show y ++")"
```

将串转换为值：Read 类

Read 类包含其值可以从串中读出的类型。使用这个类时，只需要理解下列函数即可

```
read :: (Read a) => String -> a
```

read 的结果可能无定义：在输入串中应该只包含该类型的一个值（输入串有时包含了空格或者嵌套注释），否则 read 失败。有关 read 如何分析一个串的细节见 17.5 节。

同样重要的是，在许多情况下，我们需要说明 read 的返回类型，因为 read 的潜在返回类型可以是任意的。例如

```
(read "   1  " ) :: Int
```

表明我们要求 read 的结果是 Int。

Read 类和 Show 类是互补的。因为 show 产生的串通常可以用 read 返回其值。许多类型是可读的，但函数类型不包括在内。

Haskell 数值类型和分类

设计 Haskell 的一个重要目的是建立一个具有强类型系统的函数程序设计语言，并且发生在定义或表达式中的任何类型错误均可在计算之前发现。当然，它包含了丰富的数值类型以解决大量的实际问题。Haskell 的数值类型包括

- 固定精度整数 Int 和完全精度整数 Integer，后者忠实地表示所有整数。
- 浮点数 Float 和双精度浮点数 Double。
- 有理数，即用整数之比表示的分数；预定义 Rational 是 Integer 元素之比。
- 可以在其他类型上（如 Float）上构造的复数。

Haskell 的设计要求包括重载通常的运算如 + 和/以及数 23 和 57.4 等。例如，Int 和 Integer 有相同的运算[1]和数值表示。有关整数和浮点数上的运算介绍见 3.2 节和 3.6 节。在这样的重载情况下，有些表达式的类型是不确定的，此时，需要给表达式一个显式的类型，如

```
(2+3) :: Int
```

Haskell 报告 (Peyton Jones and Hughes 1998) 讨论了一种给予表达式缺省类型的方法。

数值函数的重载是通过定义类的划分实现的。详细内容请参考 Haskell 报告（Peyton Jones and Hughes 1998）和标准引导库 Prelude.hs。下面是 Prelude.hs 的一个简单介绍。

包含所有数值类型的基类为 Num，其 Signature 为

```
class (Eq a,Show a) => Enum a where
  (+),(-),(*)  :: a -> a -> a
  negate       :: a -> a
  abs, signum  :: a -> a
  fromInteger  :: Integer -> a
  fromInt      :: Int -> a

  x - y        = x + negate y
  fromInt      = fromIntegral
```

从 Signature 看出，所有的数值类型具有相等和 show 函数，以及加法、减法、乘法和相关的运算。Int 和 Integer 还可以转换为任意数值类型的值。

整数具有任何数值类型，如

```
a :: Num a => a
```

整数类型属于 Integral 类，其 Signature 包含下列方法

```
quot, rem :: a -> a -> a
div, mod  :: a -> a -> a
```

[1] Char 的编码和解码，take 和 drop 等例外

包含分数部分的数值具有相当丰富的类结构。这个类在 Num 上增加了分数的除法和倒数

```
class (Num a) => Fractional a where
  (/)           :: a -> a -> a
  recip         :: a -> a
  fromRational  :: Rational -> a

  recip x      = 1/x
```

Float 和 Double 的浮点数属于类 Floating, 其中包含了一些数学函数,

```
class (Fractional a) => Floating a where
  pi           :: a
  exp,log,sqrt :: a -> a
  (**),logBase :: a -> a -> a
  sin,cos,tan  :: a -> a
  ...
```

完整的 Signature 可在 Prelude.hs 中找到。有关数值类型的详细内容参见引导库、标准库及 Haskell 文档。

习题

12.9 定义类型Bool和 $(t_1, t_2, ..., t_k)$ 上的关系<。

12.10 定义将布尔函数显示为表格的函数

```
showBoolFun :: (Bool -> Bool) -> String
```

将上述函数扩充为函数

```
showBoolFunGen :: (a -> String) -> (Bool -> a) -> String
```

其中第一个参数给出类型a的元素显示的串, 用于显示第二个参数对应的表格元素。你能推广到多个变元的布尔函数吗?

12.11 利用前一问题的解说明如何定义Bool -> Bool为类Show的特例(但是, 要注意, Bool -> Bool不具有特例的正确形式, 所以不是合法的Haskell代码)。

§12.5 类型与类

本节讨论 Haskell 类型分类和面向对象语言中类的关系, 初学者可跳过。

我们可以认为函数在 Haskell 的类型系统中只有单个类型。例如, 下面的多态类型

```
show :: Show a => a -> String
```

可以看作一系列类型说明的简写, 如

```
show :: Bool -> String
show :: Char -> String
```

即对于属于类 Show 的每个类型，对应一个类型说明。

在 Haskell 中，一个类是类型的一个集合。在其他语言中，如 C++，一个类型和一个类是相同的。按照后者的方法，引进可视对象的类将形成一个类型[2] ShowType 。这个类的接口将用下列函数描述

```
show :: ShowType -> String
```

ShowType 类将包含 Bool 和 Char 等作为其子类（或子类型）。于是我们可以书写下列的值

```
[True, 'N', False] :: [ShowType]
```

更进一步，若将如上列表转换为一个 String，则可定义

```
concat . map show :: [ShowType] -> String
```

在列表的不同对象上，我们将使用不同版本的 Show 函数；在第一个元素上使用 Bool 的函数，而在第二个元素上使用 Char 的函数。这种 **动态捆绑** 是许多面向对象语言包括 C++ 的突出特征，但并不为 Haskell 98 所具有。Haskell 的一个扩展允许这种动态捆绑，参见 Laufer（1996）。

返回上面的例子，在 Haskell 中`concat.map`的类型是什么呢？不难看出，其类型为

```
Show a => [a] -> [Char]
```

所以函数可应用于 [Bool] 和 [Char] 等，但不能应用于 [True,'N',False] 这种在 Haskell 中不合法的列表。

Java 允许用户定义由 Signature 构成的接口。一个类的一部分定义可以说明类实现了哪些接口。这一点很类似于 Haskell 类型分类的特例声明，只是在 Haskell 中特例声明不必成为类型定义本身的一部分，这样产生的后果是，我们可以扩展类型所支持的运算，而对 Java 中的类是不可行的。

小结

本章介绍了函数名如 read 和 show 以及运算符 + 等的重载，虽然它们在不同的类型上有不同的定义。这种机制是由 Haskell 的类系统实现的。一个类的定义包含一个 Signature。Signature 由属于此类的类型必须提供的运算名和类型构成。对于一个特定的类型，这些函数在特例声明中定义。

在说明一个函数的类型，引入一个类或特例时，我们可以提供一个上下文，以约束其中出现的类型变量。例如

```
member :: Eq a => [a] -> a -> Bool
instance  Eq a => Eq [a] where ...
class     Eq a => Ord a where ...
```

从这个例子中可以看出，member 只能用于 Eq 类的类型；类型 a 上列表可以定义相等，只要类型 a 上可以定义相等；Ord 类的类型必须已经是 Eq 类的类型。

[2]用 C++ 的术语，这是一个抽象基类，如 Bool, 其子类必须继承和实现基类的方法

在介绍了各种例子之后，我们还介绍了 Haskell 标准引导库中定义的类，最后讨论了 Haskell 的类型分类与面向对象语言的类的关系。在本书的最后，我们将再次讨论类型分类，以及如果利用它们组织大规模系统。

第十三章　类型检测

Haskell 中的每个值都有一个类型，这个类型可能是单态的，多态的，或者包含分类约束的上下文。例如

```
'w' :: Char
flip :: (a -> b -> c) -> (b -> a -> c)
elem :: Eq a => a -> [a] -> Bool
```

强类型意味着我们可以在计算表达式或者使用定义之前来 **检测** 它们是否满足 Haskell 的类型规则。这种检测的优点是显然的：我们不需要运行一个程序便可检查出程序中的一类错误。

除此之外，类型也是程序文档的宝贵组成部分：当我们看到一个函数时，第一条信息便是它的类型，因为类型说明如何使用一个定义。一个函数的类型说明它可以应用到什么类型的值，以及结果是什么类型的值。

类型还可用于在一个函数库中搜索函数。假设我们想定义一个删除列表中重复元素的函数，例如，将`[2,3,2,1,3,4]`转换为`[2,3,1,4]`。这样的函数具有类型

```
(Eq a) => [a] -> [a]
```

在引导库 Prelude.hs 和 List.hs 中搜索的结果，只有一个函数具有这样的类型，即 nub，这正是我们所寻找的函数。在现实中，或许有多个函数具有给定的类型（或者由于参数顺序的不同而遗漏），但是，类型为库函数提供了一个”把柄”。

本章是类型检测的非形式概述。我们首先探讨在单态构架中的类型检测。在这种情形下，每个类型正确的表达式都有一个类型。然后我们讨论多态的情形。这种情形的类型检测可以理解为构造表达式时置于表达式类型上的约束。类型检测的一个关键概念是**合一**，它是将约束合并在一起的过程。本章最后讨论上下文。用于重载的上下文包含了类型变量成为分类成员应该满足的条件。

§13.1　单态类型检测

本节讨论没有多态和重载的单态类型检测。这里的重点是函数应用的类型检测。我们下面看到的是简化的内容，也是稍后讨论的一般类型检测的基础。在涉及到多态运算时，我们将使用其单态特例，并用一个类型下标表示。例如

$+_{\texttt{Int}}$ :: Int -> Int -> Int

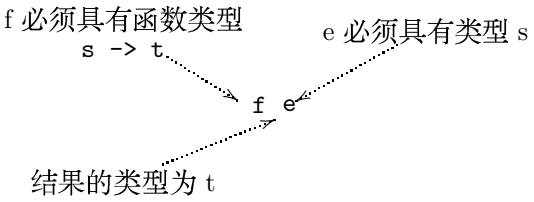
$\texttt{length}_{\texttt{Char}}$ :: [Char] -> Int

我们首先讨论如何检测表达式的类型，然后讨论定义的类型检测。

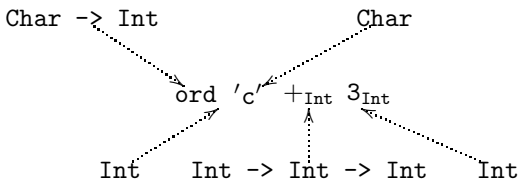
表达式

一般地，一个表达式是一个原子，一个变量或者一个常量，或者由一个函数应用于一些参数构成，这些参数本身也是表达式。函数应用包含的情况比我们想象的多。例如，表达式 [True False] 可看作构造符函数的应用，即`True:[False]`。同理，运算符和条件构造符`if ...then...else`的作用与函数一样，只是使用了不同的语法。

函数应用的类型检测规则如下图所示，即一个类型为`s -> t`的函数必须应用于类型`s`的参数。一个类型正确的函数应用结果是类型`t`的表达式。

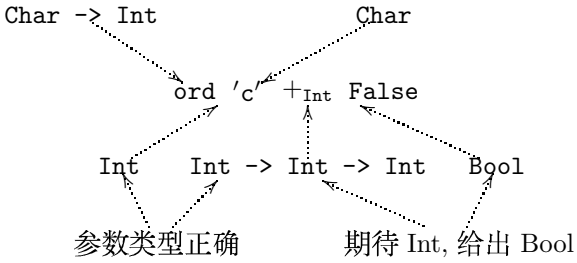


下面看两个例子。第一个是 (`ord 'c' +`$_{\texttt{Int}}$ `3`$_{\texttt{Int}}$)，这是一个类型正确的表达式，其类型为 Int.



ord 应用于'c' 的结果是类型 Int 的表达式，$+_{\texttt{Int}}$ 的第二个参数也是一个 Int，所以 $+_{\texttt{Int}}$ 的应用类型正确，结果类型为 Int。

如果将上例改为 `ord 'c' +`$_{\texttt{Int}}$ `False`，我们将得到一个类型错误，因为运算 $+_{\texttt{Int}}$ 被应用于一个布尔值 False, 但它的期待值类型应该是 Int。



函数定义

现在考虑下列单态函数定义的类型检测

```
f :: t1 -> t2 -> ... -> tk -> t                    (fdef)
f p1 p2 ... pk
  | g1     = e1
  | g2     = e2
  ...
  | gl     = el
```

我们需要检测下列三点

- 每个守卫 g_i 必须具有类型 Bool;
- 每个子句返回的值 e_i 必须具有类型 t;
- 模式 p_j 必须与其参数的类型 t_j 一致。

称一个模式与一个类型**一致**, 如果模式与此类型的某些元素匹配。下面是各种可能的情形。一个变量与任意类型一致; 一个原子与其类型一致。如果 p 与 t 一致, 并且 q 与 [t] 一致, 那么模式 (p:q) 与类型 [t] 一致。例如, (0:xs) 与类型 [Int] 一致, (x:xs) 与任意列表类型一致。定义的其他情形是类似的。

由此结束了单态类型情况下的类型检测。下节讨论多态类型检测。

习题

13.1 说明下列定义会产生什么类型错误

```
f n    = 37 + n
f True = 34

g 0 = 37
g n = True

h x
 | x>0       = True
 | otherwise = 37

k x = 34
k 0 = 35
```

试将上述定义键入一个Haskell脚本, 然后调入Hugs系统验证你的答案。注意, 你可以用:type得到一个表达式的类型。

§13.2 多态类型检测

在单态的情形下, 一个表达式或者类型正确, 具有一个类型, 或者类型不正确, 因此没有类型。在 Haskell 这样的多态语言中, 情况更复杂, 因为一个多态对象恰好具有许多类型。

在本节我们首先再次讨论多态的含义，然后解释通过 **约束满足** 的类型检测，其中心概念是合一，由此找出同时满足两个类型约束条件的类型。

多态

我们熟悉的下列函数是多态的

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

但是如何理解类型中的类型变量 a 呢？我们可以把 (length) 读作 “length 具有一簇类型”

```
[Int] -> Int
[(Bool, Char)] -> Int
 ...
```

即包含了所有类型`[t] -> Int`，其中 t 是一个**单一类型**，即不含类型变量的类型。

当 length 被应用于一个参数时，我们必须确定在上面的类型中 length 使用的是哪一个类型。例如，对于

```
length ['c','d']
```

我们可以看出 length 应用于 Char 的列表，所以在这里 length 类型是`[Char] -> Int`。

约束

如何解释一般的检测过程呢？可以看出，一个表达式的不同部分在它的类型上加以不同的约束。在这种解释下，类型检测变成检查类型是否满足这些约束。在 9.2 节讨论 map 和 filter 的类型时，我们已经看到这样的例子。下面是更进一步的例子。

例 13.1 *考虑下列定义*

```
f (x,y) = (x, ['a' .. y])
```

f 的参数是一个二元组，所以我们需要考虑 x 和 y 的类型。x 是完全无约束的，作为结果的第一个分量原样返回。另一方面，y 被用于表达式`['a' .. y]`，它表示枚举类型的一个范围，以字符`'a'`开始，由此迫使 y 具有类型 Char，故得 f 的类型为

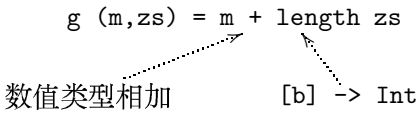
```
f :: (a, Char) -> (a, [Char])
```

例 13.2 *考虑下列定义*

```
g (m, zs) = m + length zs
```

参数 m 和 zs 上的约束是什么呢？我们可以看出 m 是加法运算的运算数，故 m 必具有数值类型，具体属于哪一个数值类型，有待进一步分析。加法的另一个运算数是`length zs`，这说明两点。首先，zs 的类型是 [b]，而且结果的类型是 Int。这种约束导致 + 的参数类型为 Int，由此得

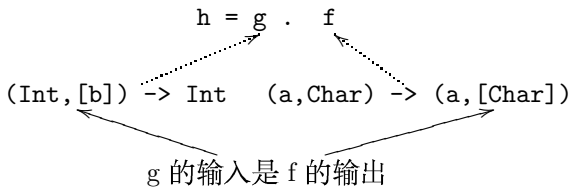
```
g :: (Int, [b]) -> Int
```



例 13.3 最后考虑前两个例子的复合

```
h = g . f
```

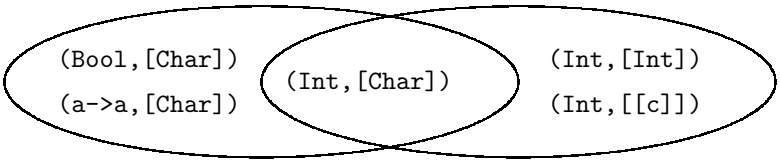
在复合`g . f`中, f 的输出成为 g 的输入



我们需要回想一下带有类型变量的类型的意义; 这样的类型可以看作一些类型的集合的缩写。f 的输出具有类型`(a, [Char])`, g 的输入具有类型`(Int, [b])`。因此, 我们必须寻找同时满足这两种描述的类型。我们将讨论这类一般问题的解决方法, 并在适当的时候返回这个例子。

合一

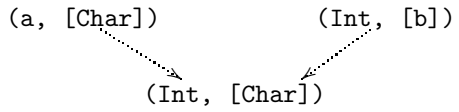
如何来描述满足两个描述`(a, [Char])`和`(Int, [b])`的类型呢?



利用类型作为集合的概念, 我们寻找`(a, [Char])`和`(Int, [b])`表示的集合的交集。如何描述这个交集呢? 我们先介绍一些术语。

一个类型的特例是将类型中的一个或几个类型变量用类型表达式代替后的结果。一个类型表达式是两个类型表达式的公共特例, 如果它是每个表达式的特例。两个类型表达式的一个公共特例 mgci 称为 **最广公共特例**, 如果这两个表达式的所有特例均是 mgci 的特例。

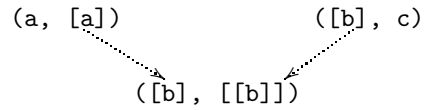
现在我们可以描述这两个类型表达式所表示的交集。我们称之为两个类型表达式的合一, 它是两个类型表达式的最广公共特例。再返回来考虑例 3。在此例中, 我们有



即最广公共特例是`(Int, [Char])`。由此得到下列类型

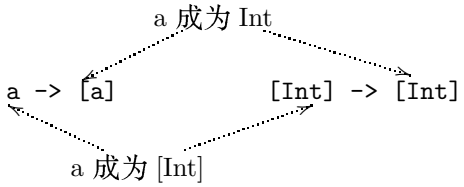
```
h :: (Int, [Char]) -> Int
```

合一的结果不一定总是单态类型。例如 (a, [a]) 和 ([b], c) 的合一结果是 ([b], [[b]])。这是因为表达式 (a, [a]) 要求类型的第二个分量的类型是第一个分量类型的列表，而表达式 ([b], c) 要求第一个分量是一个列表。因此，最后得到类型 ([b], [[b]])。



在上例中，两个类型表达式的公共特例有很多，例如，([Bool], [[Bool]]) 和 ([[c]], [[[c]]])，但每一个都不是 mgci，因为 ([b],[[b]]) 不是它们的特例。另一方面，前两个类型是 ([b],[[b]]) 的特例，因此它是最广公共特例。

并非任意一对类型均可以合一：考虑类型`[Int] -> [Int]`和`a -> [a]`。

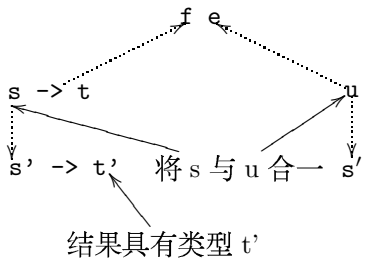


将这两个参数类型合一要求 a 成为 [Int]，而结果类型合一要求 a 成为 Int；显然这些约束不一致，故合一失败。

表达式类型检测

如 13.1 节所述，函数应用是构造表达式的主要方法。这表明函数应用的类型检测也是关键问题。

多态函数应用的类型检测



将一个函数`f::s -> t`应用于一个参数`e::u`时，我们并不要求 s 和 u 相等，而要求 s 和 u 必须能够合一成类型 s'，使得`e::s'`并且`f::s' -> t'`，结果的类型为 t'。例如，考虑函数应用`map ord`，其中

```
map :: (a -> b) -> [a] -> [b]
ord :: Char -> Int
```

将a -> b和Char -> Int合一的结果是 a 成为 Char, b 成为 Int 故有

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

```
map ord :: [Char] -> [Int]
```

与单态情况类似, 类型推导和函数应用的讨论可以用于解释表达式的类型检测。下面再看一个例子, 然后讨论类型检测的技术细节。

例 13.4 *折叠函数*foldr

在 19.3 节我们定义了函数foldr

```
foldr f s []      = s                              (foldr.1)
flodr f s (x:xs)  = f x (foldr f s xs)             (foldr.2)
```

此函数可用于将一个运算折叠入一个列表, 如

```
foldr (+) 0 [2,3,1] = 2+(3+(1+0))
```

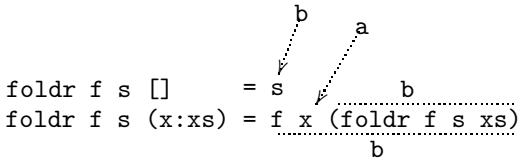
初看上去, foldr 具有下列类型

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

事实上, foldr 的最广类型比上述类型广泛。假设初始值的类型为 b, 列表元素类型为 a

```
foldr :: (... -> ... -> ...) -> b -> [a] -> ...
```

则定义中的类型如下图所示



s 是第一个等式的结果, 所以 foldr 本身的结果类型为 b, 即 s 的类型

```
foldr :: (... -> ... -> ...) -> b -> [a] -> b
```

在第二个等式中, f 被应用于第一个参数 x, 故有

```
foldr :: (a -> ... -> ...) -> b -> [a] -> b
```

f 的第二个参数是 foldr 的结果, 故有类型 b

```
foldr :: (a -> b -> ...) -> b -> [a] -> b
```

最后, 第二个等式的结果是 f 的应用, 其结果类型必须同 foldr 的结果类型 b 相同

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

从对 foldr 的分析看出, 我们可以利用 foldr 定义许多列表函数, 例如插入排序

```
iSort :: Ord a => [a] -> [a]
iSort = foldr ins []
```

其中 ins 的类型为

```
ins :: Ord a => a -> [a] -> [a]
```

多态定义和变量

本节讨论多态定义类型检测中的技术细节，初学者可略过。

同一个表达式中的函数和常量可以具有不同的类型。最简单的例子是

```
expr = length ([]++[True]) + length ([]++[2,3,4])        (expr)
```

空列表[]的第一次出现具有类型 [Bool]，而第二次出现具有类型 [Int]。这是完全合法的，而且是多态函数的优点之一。现在假定用一个变量代替[]，定义

```
funny xs = length (xs++[True]) + length (xs++[2,3,4])   (funny)
```

则变量必须具有类型 [Bool] 和类型 [Int]；即 xs 的类型是多态的。这在 Hadkell 中是不允许的，因为 funny 的类型无法表示。你或许会觉得下列类型是正确的

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

但是，这表示 funny 将具有所有的类型特例

```
funny :: [Int] -> Int
funny :: [[Char]] -> Int
...
```

显然这是不成立的。结论是: 常量和变量要区别对待，常量在同一表达式中可以有不同的类型，但变量则不同。

禁止定义 (funny) 和允许定义 (expr) 的意义何在？首先，在 (expr) 的定义中[]::[a]有一个多态定义，而且在表达式中出现了两次；第一次出现的类型为 [Bool]，第二次出现的类型为 [Int]。允许这样独立应用的出现，一个应用的类型检测不影响其他应用的类型检测。

另一方面，定义 (funny) 为何被禁止呢？当我们检测一个变量的类型时，我们不能将变量的每次出现的类型作为一个独立的特例。假如对变量xs::t没有初始约束。xs 的第一次出现使xs::[Bool]，第二次出现要求xs::[Int]；这两个约束不能同时满足，所以定义 (funny) 类型检测失败。

从这个例子应记取的经验是，一个函数的定义不能强迫它的参数具有多态类型。

函数定义

在类型检测一个函数定义时，如第 201 页的 (fdef)，我们必须遵守类似于单态情况下的规则

- 每个守卫 g_i 必须具有类型 Bool;
- 每个子句返回的值 e_i 必须具有一个类型 s_i, 使得 t 是 s_i 的一个特例
- 每个模式 p_j 必须与相应参数的类型 t_j 一致。

下一节我们讨论类型分类的类型检测。

习题

13.2 下列垂直的类型对能合一吗？如果可以，给出一个最广合一类型；如果不可以，解释为什么合一失败。

```
(Int -> b)                (Int, a, a)
(a -> Int)                (a, a, [Bool])
```

13.3 说明我们可以将(a, [a])和(b,c)合一为(Bool, [Bool])。

13.4 下列函数能应用于(2,[3])，(2,[])和(2,[True])吗？

```
f :: (a,[a]) -> b
```

如果可以，结果的类型是什么？试解释你的答案。

13.5 对于下列函数重复回答上述问题：

```
f :: (a,[a]) -> a
```

试解释你的答案。

13.6 假定函数f具有如下类型

```
f :: [a] -> [b] -> a -> b
```

试给出f [] []的类型。

下列函数h的类型是什么？

```
h x = f x x
```

13.7 如何使用Haskell系统检测两个类型表达式是否可以合一？如果可以，它们的合一结果是什么？提示：可以在Haskell中定义哑元，如zircon等于它自己

```
zircon = zircon
```

这样定义的哑元可以具有任何你希望的类型。

13.8 (较难) 回顾函数curry和uncurry的定义，说明下列表达式的类型

```
curry id
uncurry id
curry (curry id)
uncurry (uncurry id)
uncurry curry
```

试解释为什么下列表达式类型检测失败

```
curry uncurry
curry curry
```

13.9 (较难) 给出一个判定两个类型表达式是否可以合一的算法。如果可以合一，你的算法应该返回一个最广合一代换；如果不可以合一，算法应该解释合一失败的原因。

§13.3 类型检测与分类

Haskell 的类型分类限制某些函数只能应用于定义了相应函数的类型上。例如，== 只能应用于属于 Eq 的类型。这些限制在某些含有上下文的类型中是显然的。例如，如果定义

```
member []     y = False
member (x:xs) y = (x==y) || member xs y
```

其类型将是

```
Eq a => [a] -> a -> Bool
```

因为在定义中类型 a 的元素 x 和 y 用于相等的比较，迫使 a 属于相等分类 Eq。

本节讨论涉及重载的类型检测方法。讨论将通过一些例子说明。

假定我们将函数 member 应用于表达式 e，其类型为

```
Ord b => [[b]]
```

这表示 e 是带有顺序运算的类型上的列表的列表。在没有上下文的情况下，将 member 与 e 的类型合一得到

```
member :: [[b]] -> [b] -> Bool          e :: [[b]]
```

所以，应用表达式 member e 的类型为`[b] -> Bool`。使用同样的方法，并将合一应用于上下文得到下列上下文

```
(Eq [b], Ord b)                                    (ctx.1)
```

下面我们来检测并简化上下文

- 在上下文中的约束只适用于类型变量，所以需要消去像`Eq [b]`这样的约束。消去这种约束的唯一办法是使用特例说明

  ```
  instance Eq a => Eq [a] where ...
  ```

  对于这种情况，以上特例允许我们将 (ctx.1) 中的`Eq [b]`用`Eq b`代替，由此得上下文

  ```
  (Eq b, Ord b)                                      (ctx.2)
  ```

  重复这个过程直至没有适用的特例。

  如果我们不能将约束简化为只包含类型变元约束的上下文，则函数应用类型检测失败，并生成一个错误信息。例如，如果将 member 应用于 [id]，则得到

  ```
  ERROR: a -> a is not an instance of class "Eq"
  ```

  因为 id 是一个函数，其类型不属于 Eq 分类。
- 进一步利用 class 定义简化上下文。对于此例，上下文包含 Eq b 和 Ord b，但是

  ```
  class Eq a => Ord a where ...
  ```

  所以 Ord 的任意特例自动成为 Eq 的特例；于是上下文 (ctx.2) 可简化为

  ```
  Ord b
  ```

重复这个过程，直至不能再简化。

对于我们的例子，最后得到类型

```
member e :: Ord b => [b] -> Bool
```

这种合一，类型检测（使用特例）和简化的三阶段过程是 Haskell 中涉及上下文类型检测的一般模式。

最后，我们应该解释上下文是如何引入类型中的。上下文源于类型分类说明中函数的类型，例如，

```
toString :: Visible a => a -> String
size     :: Visible a => a -> Int
```

如果一个函数使用了这些过载函数，则函数的类型检测将传播和结合这些上下文，如前面的分析所见。

我们已经看到 Haskell 类型系统如何适应于类型分类的过载名的类型检测。更详细的资料，包括需要加在某些多态约束上的“单态限制”的讨论，请读者参看 Haskell 98 报告（Peyton Jones and Hughes 1998）。

习题

13.10 给出下列定义中每个条件等式的类型，并讨论它们共同定义的函数的类型。

```
merge (x:xs) (y:ys)
  | x<y        = x:merge xs (y:ys)
  | x==y       = x:merge xs ys
  | otherwise  = y:merge (x:xs) ys
merge (x:xs) []    = (x:xs)
merge []    (y:ys) = (y:ys)
merge []     []    = []
```

13.11 定义一个多态排序函数，并说明它的类型是如何由顺序关系的类型推出的。

```
compare :: Ord a => a -> a -> Ordering
```

13.12 讨论下列数值函数的类型；你将发现这些类型引用了一些预定义的数值分类。

```
mult x y = x*y
divide x = x `div` 2
share x  = x/2.0
```

注意，通过显式地说明类型，这些函数可以具有更局限的类型，如

```
divide :: Int -> Int
```

小结

本章解释了 Haskell 表达式和定义的类型检测过程。我们先讨论了单态的情形，然后扩展到多态。对于后者，类型检测是一个提取和统一约束的过程，

统一约束通过对包含类型变量的类型表达式进行合一完成。最后，我们讨论了如何处理类型中的上下文，也就是重载在 Haskell 类型系统中的处理方法。

第十四章　代数类型

到目前为止，我们可以用下列类型描述数据

- 基本类型：Int, Float, Bool 和 Char;
- 复合类型：多元组 $(t_1, t_2, ..., t_n)$; 列表类型 $[t_1]$ 及函数类型 t_1 -> t_2，其中 $t_1, t_2, ..., t_n$是类型。

这些类型为我们提供了广泛的类型选择，包括很复杂的结构，例如，文件的索引可以用类型的适当组合表示，在那个例子中，我们使用了类型 [([Int],[Char])]。但是，存在很多其他的类型，很难使用上面所述的类型刻画，这样的例子包括

- 月份构成的类型：January, ···, December;
- 元素可能是数，也可能是串的类型；例如，一幢房子可以用一个数字表示也可以用一个名表示；
- 树的类型，如图 14.1 所示。

所有这些类型都可以用 Haskell 的代数类型刻画，这便是本章的内容。

§14.1　代数类型简介

代数 (数据) 类型 是由关键字 data 引入的，后接类型名称，等号以及所定义的类型的**构造符**。类型名和构造符名要以大写字母开头。我们将给出一系列由简单到复杂的例子，然后讨论这种类型定义的一般格式。

枚举类型

最简单的代数类型的定义是列出类型的元素。例如，

```
data Temp   = Cold | Hot
data Season = Spring | Summer | Autumn | Winter
```

这里定义了两个类型。类型 Temp 有两个成员，Cold 和 Hot; 类型 Season 有四个成员。Cold 和 Hot 称为类型 Temp 的 **构造符**。若要定义这些类型上的函数，可以使用模式匹配；模式可以是一个原子值，也可以是一个变量。要描述

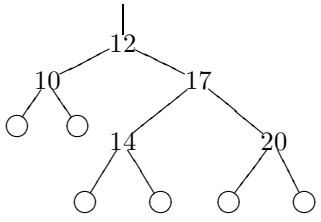


图 14.1: 整数树例子

(英国!) 天气，我们可以定义

```
weather :: Season -> Temp
weather Summer = Hot
weather _      = Cold
```

模式匹配是顺序进行的；第一个匹配成功的模式是我们将使用的等式。上式定义表示，英国的天气在夏天是 Hot，而在其他时间是 Cold。预定义的布尔类型也是一个代数类型，其定义为

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

在 Ord 类中使用的类型 Ordering 也是代数类型，它的定义为

```
data Ordering = LT | EQ | GT
```

正如我们看到的，模式匹配可用于定义代数类型上的函数。例如，我们可以用模式匹配定义 Temp 上的相等运算

```
Cold == Cold = True
Hot  == Hot  = True
_    == _    = False
```

然后将 Temp 声明为 Eq 类的特例。

在每个新定义的类型上定义相等运算是繁琐的，为此, Haskell 系统提供了自动生成相等、顺序枚举和转换为串的函数。我们将在本章最后介绍其细节。

乘积类型

我们也可以不使用多元组类型，而是定义一个由几个类型构成的称为乘积的代数类型。例如

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

其中 Name 是 String 的一个别名，Age 是 Int 的别名，在 Haskell 中记作

```
type Name = String
type Age  = Int
```

People 的定义可读作："要构造类型 People 的一个元素，我们需要提供两个值：一个是类型 Name 的值，记作 st，另一个是类型 Age 的值，记作 n。由此形成的 People 的元素记作`Person st n`"。

这个类型的值的例子包括

```
Person "Electric Aunt Jemima" 77
Person "Ronnie" 14
```

如前所述, 函数可以用模式匹配定义。类型 People 的元素的一般格式是`People st n`, 所以我们可以在定义的等式左边使用这个模式,

```
showPerson :: People -> String
showPerson (Person st n) = st ++ "--" ++ show n
```

(注意 show 给出一个 Int 的串表示，因为 Int 属于 show 类)。例如，

```
showPerson (Person "Electric Aunt Jemima" 77)
  = "Electric Aunt Jemima -- 77"
```

在这个例子中，类型含有一个构造符，它由两个元素构造类型 People 的一个值，故称为二元的。枚举类型 Temp 和 Season 中的构造符称为 0 元的，因为它们不需要参数。代数类型定义中的构造符可以像函数一样使用，所以`Person st n`是将函数 Person 应用于参数 st 和 n 的结果；People 的定义可以解释为给出**构造符的类型**，如

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

类型 People 的另一个定义是使用类型别名

```
type People = (Name, Age)
```

使用代数类型的优点有三个

- 类型的每个成员都有一个显式的标志，以表明此元素的目的，在上述情况标志表示一个人；
- 一个人必须使用构造符 Person 构造，所以我们不可能将任意的由串和数构成的二元组作为 People 的成员对待；
- 类型会出现在任何由于类型错误引起的错误信息中；一个类型别名将被它代表的类型代替，所以类型别名不会出现在任何错误信息中。

使用多元组类型别名也有一些优点：

- 类型的成员更紧凑，定义更短；
- 使用多元组，特别是二元组，我们可以使用多元组上的许多多态函数，如 fst , snd 和 unzip；这在代数类型上是不可行的。

在为一个系统建模时，我们必须在这些不同的建模方法中做出选择；我们的选择取决于我们如何使用产品，以及系统的复杂程度。对于目前的情况，同样可以利用一元构造符表示 Age

```
data Age = Years Int
```

其元素为`Years 45`等。显然，这里的 45 是以年为单位的年龄，而不是一个纯粹的数量。缺点是我们不能在 Age 上直接使用 Int 上的函数。

我们也可以用类型名做构造符名，如 Person 既可以是类型名，同时可以做构造符名：

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

但是，我们将避免这样做，以免引起误解。

上面的类型例子是我们下面将看到的类型的一个特殊情形。

多个不同的选择

在一个简单的几何程序中，一个形状或者是一个圆或者是一个矩形。这些不同的选择由下列类型表达

```
data Shape = Circle Float |
             Rectangle Float Float
```

它表示，构造 Shape 的元素有两种方式：一种是给出 Circle 的半径，另一种是给出 Rectangle 的两个边。这个类型的对象例子有

```
Circle 3.0
Rectangle 45.9 87.6
```

利用模式匹配可以分情况定义函数，如

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

我们还可以使用组成元素的成分

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

解读 Shape 定义的另一种方法是：类型 Shape 有两个 **构造函数**，其类型为

```
Circle    :: Float -> Shape
Rectangle :: Float -> Float -> Shape
```

因为类型的元素是通过应用这些函数构造的，故称这些函数为 **构造函数**。本节的习题将讨论这个类型的扩展，即给对象附加一个位置信息。

代数类型定义的一般形式

我们目前看到的代数类型定义的一般形式为

```
data Typename                          (Typename)
   = Con1 t11 ... t1k1 |
     Con2 t21 ... t2k2 |
        ...
     Conn tn1 ... tnkn |
```

每个 $\mathtt{Con_i}$ 是一个构造符，其后跟随着 $\mathtt{k_i}$ 个类型，$\mathtt{k_i}$ 是一个非负整数，可以是 0。通过应用构造符函数于它的参数可以构造类型 Typename 的元素，所以

$$\mathtt{Con_i\ v_{i1} \ldots v_{ik_i}}$$

是类型 Typename 的一个成员，其中 $\mathtt{v_{ij}}$ 是 $\mathtt{t_{ij}}$ 的元素 ($1 \leq \mathtt{j} \leq \mathtt{k_i}$)。

将构造符视为函数，Typename 的定义给出构造符的下列类型

```
Coni  ::  ti1 -> ... ->tiki -> Typename
```

在下列几节，我们将看到以上定义的两种扩展。

- 类型可以是递归的；Typename 定义中的任何 $\mathtt{t_{ij}}$ 可以是正在定义的类型 Typename。由此可以得到列表、树和许多其他类型。
- Typename 可以跟随一个或者多个类型变量，这些变量可用于定义的右边，由此可得到多态类型。

递归多态类型结合了这两种思想，而且这种有力的结合提供了可以在许多不同场合下重用的类型 — 预定义的列表便是这样的一个例子。以后我们会介绍其他例子。

在继续讨论代数类型之前，我们来对 type 和 data 定义做一比较。由 type 定义的类型别名只是一个缩写，而且一个类型别名总是可以展开为它代表的类型，因此可以在程序中删除。另一方面，一个 data 定义创建一个新类型。因为别名只是缩写，别名定义不可能是递归的；data 定义可以是，也常常是递归的。

导出类的特例

如我们以前看到的，Haskell 有一些预定义类，包括

- Eq 类，给出相等和不相等运算；
- 建立在 Eq 上的 Ord 类，给出类型的元素上的序；
- Enum 类，允许类型元素被枚举出来，因此可以使用像 [n..m] 的表达式；
- Show 类，允许类型的元素表达成文本形式，以及 Read 类，允许从串中读出类型的值。

在定义一个新代数类型时，如 Temp 和 Shape，我们可能希望它成为相等和枚举等的特例。我们可以让系统提供这些特例函数：

```
data Season = Spring | Summer |Autumn | Winter
              deriving (Eq, Ord, Enum, Show, Read)
data Shape  = Circle Float |
              Rectangle Float Float
              deriving (Eq, Ord, Show, Read)
```

由此我们便可比较季节的相等和顺序，或者用下列的表达式

```
[Sping .. Autumn]
```

表示列表 [Spring,Summer,Autumn], 或者显示类型的值。我们同样也可以导出 Shape 的特例函数，只是不能枚举图形；属于 Enum 的类型只适用于象 Season 这样的枚举类型。

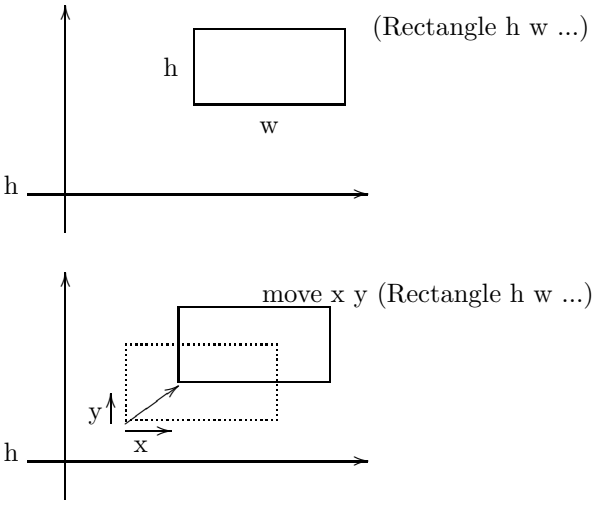
我们也可以不使用系统导出定义，而是自己给出特例定义。例如，可以定义半径为负数的所有圆相等；showPerson 的定义可用于声明 People 为 Show 类的特例。

习题

14.1 使用守卫或者if重新定义函数`weather :: Season -> Temp`。你认为那一种定义更好？

14.2 定义表示月份的Haskell代数类型，然后定义一个将月份映射到季节的函数。你可能需要类型上的导出序，如前所述。

14.3 如何定义地处南半球与英国同纬度的新西兰的天气函数weather？如何定义地处赤道上的巴西的天气函数？



14.4 定义类型Shape上的几何图形的周长函数，并说明函数的类型。

14.5 在类型Shape上增加一个三角形构造函数，然后扩充函数isRound, area和perimeter的定义。

14.6 圆，正方形和等边三角形称为正则图形。试定义一个判定一个Shape是否正则图形的函数。

14.7 检查Temp和Shape的导出函数，例如，次序和show函数是如何定义的？

14.8 定义Shape上的相等==使得半径为负数的圆均相等。如何处理边长是负数的三角形呢？

14.9 类型Shape没有考虑图形的位置和方向。在决定了点的表示后，如何修改Shape的定义使其包含每个对象的中心？你可以假定矩形的边与两个轴分别平行，如下图

14.10 将新定义的形状类型称为NewShape。试定义函数

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

其作用是将图形移动给定的偏移量。

14.11 定义判定两个图形是否有重叠部分的函数。

14.12 一个房子可能有一个编号，也可能有一个名。作为房子地址的部分内容，如何定义类型“串或者数字”？定义一个用文本形式显示房子的函数。定义名的类型和地址类型。

14.13 使用代数类型（如People而不是二元组）重新实现 *5.6* 节的数据库，并比较这两种实现的特点。

14.14 第 *5.6* 节的数据库可以做如下扩充

- 读者不仅可以借书，而且可以借CD和录像；
- 库中记录图书和CD的作者和书名（标题）等信息；
- 借阅期限：图书一个月，CD一周，录像三天。

试解释如何修改相应的类型以及函数。试实现系统的下列操作，说明每个操作的类型和定义

- 查找某个读者借阅的所有东西；
- 查找库中所有到期的借阅，某个日期到期的所有借阅，以及某个人所有到期的借阅。
- 借阅后修改数据库；可以假定常数today包含了今天的日期，日期形式自定。

若使得数据库可用，还应该定义哪些函数？给出它们的类型。

§14.2 递归代数类型

类型经常可以很自然地用它们本身来描述。例如，一个整数表达式或者是一个原子整数，如 347，或者是由算术运算符（如加或减）和两个表达式组合成的一个新表达式，如 (3+1)+3。

```
data Expr = Lit Int |
            Add Expr Expr |
            Sub Expr Expr
```

类似地，一颗树或者是空，或者由一个值和两颗子树构成。例如，数 12 和图 14.2 的结合形成图 14.1 的树。

作为 Haskell 类型，其定义为

```
data NTree = NilT |
             Node Int NTree NTree
```

最后，我们已经用到的列表类型：一个列表或者是空（[]）或者由一个头和一个尾 – 另一个列表 – 通过列表构造符':'构成。列表是使用递归（和多态）定义的很好例子。特别地，由列表的例子可看出如何在其他代数类型上定义一般多态高阶函数，以及如何验证程序。下面我们详细地介绍一些例子。

表达式

类型 Expr 是前面讨论的简单数值表达式的模型。例如，这个类型可用于实现一个简单的数值计算器。

```
data Expr = Lit Int |
            Add Expr Expr |
            Sub Expr Expr
```

下面是一些表达式例子，左边是非形式的表达式，右边是它们在类型 Expr 中的表示。

```
2                 Lit 2
2+3               Add (Lit 2) (Lit 3)
(3-1)+3           Add (Sub (Lit 3) (Lit 1)) (Lit 3)
```

给定一个表达式，我们可能希望

- 计算它的值;
- 将其转换成串，然后输出;
- 估算其大小，如运算符的个数。

每一个函数都可以利用原始递归用相同的方式定义。因为类型本身是递归的，所以处理类型的函数也是递归的也就不奇怪了。同时，递归函数定义的形式与类型定义的形式一样。例如，要计算一个带运算符的表达式的值，我们先计算参数的值，然后使用运算符将这些值结合起来。

```
eval :: Expr -> Int
eval (Lit n)      = n
eval (Add e1 e2)  = (eval e1)+(eval e2)
eval (Sub e1 e2)  = (eval e1)-(eval e2)
```

原始递归定义由两部分构成：

- 递归基的情况，其定义是直接的，在这里是 (Lit n);
- 在递归的情况，表达式的值用子表达式值来计算，在此例中，即 eval e1 和 eval e2。

函数 show 有相同的形式

```
show :: Expr -> String
show (Lit n)    = show n
show (Add e1 e2)
  = "("++show e1 ++"+"++show e2 ++")"
show (Sub e1 e2)
  = "("++show e1 ++"-"++show e2 ++")"
```

计算表达式中运算符个数的函数也有相同的形式，留给读者作为练习。本节的其他练习讨论表达式的另一种不同表示，其中一个类型用于表示不同的运算符。下面我们将介绍另一个递归代数类型，之后再返回 Expr 并且给出用一种特定的方式重新组织表达式的非原始递归定义。

整数树

像图 14.1 那样的整数树可以用下列类型描述

```
data NTree = NilT |
             Node Int NTree NTree
```

空树由 NilT 表示，图 14.2 的树可表示为

```
Node 10 NilT NilT
Node 17 (Node 14 NilT NilT)(Node 20 NilT NilT)
```

树的许多函数定义是原始递归的，例如

```
sumTree, depth :: NTree -> Int

sumTree NilT           = 0
```

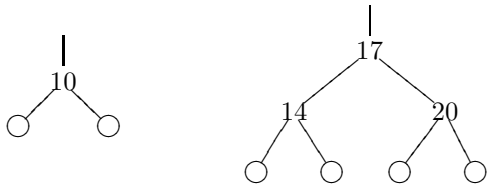


图 14.2: 树的例子

```
sumTree (Node n t1 t2) = n + sumTree t1 + sumTree t2


depth NilT           = 0
depth (Node n t1 t2) = 1 + max (depth t1) (depth t2)
```

使用这些函数的例子

```
sumTree (Node 3 (Node 4 NilT NilT) NilT) = 7
depth   (Node 3 (Node 4 NilT NilT) NilT) = 2
```

另一个例子是求一个给定的数 p 在树中出现的次数。根据树的不同形状，原始递归分两种情形：

- 对于空树 NilT，答案是 0；
- 对于非空树 Node n t1 t2 ，可以通过递归调用求 p 在 t1 和 t2 中出现的次数，然后根据 p 是否在这个特定的根结点出现，即是否 p==n 分情形定义。

最后的定义是

```
occurs :: NTree -> Int -> Int
occurs NilT p = 0
occurs (Node n t1 t2) p
  | n==p      = 1+ occurs t1 p + occurs t2 p
  | otherwise =    occurs t1 p + occurs t2 p
```

本节末尾的练习包括树上使用原始递归的其他例子。下面我们看一个使用不同形式递归的特殊例子。

重组表达式

下一个例子是使用更一般递归的例子。在说明一般递归的必要性之后，我们说明这样定义的函数是完全的，即函数在所有合法表达式上有结果。

整数上的加法是可结合的，所以括号的次序是无关紧要的。为此，我们可以决定用什么样的方式将包含'+' 的表达式括起来。此例的目的是编写一个将表达式变为括号向右结合的形式，如下表所示

```
(2+3)4                    2+(3+4)
((2+3)+4)+5               2+(3+(4+5))
```

```
((2-((6+7)+8))+4)+5        (2-(6+(7+8)))+(4+5)
```

程序要做的是什么呢？主要目的是找出下列形式的表达式

```
  Add (Add e1 e2) e3                    (AddL)
```

并将其转换为

```
 Add e1 (Add e2 e3)                     (AddR)
```

首次尝试的程序可能是

```
 try (Add (Add e1 e2) e3)
  = Add (try e1) (Add (try e2) (try e3))
 try ...
```

这里的定义是原始递归的；在定义的右边函数 try 仅用于参数的子表达式。这个函数可以将（AddL）转换为（AddR），不幸的是，(AddExL) 被转换为 (AddExR)：

```
 ((2+3)+4)+5                            (AddExL)
 (2+3)+(4+5)                            (AddExR)
```

问题在于转换 (AddL) 为 (AddR) 的过程中可能会产生我们在顶层寻找的另一模式：这恰恰是发生在 (AddExL) 到 (AddExR) 转换过程中的现象。为此，我们必须在重组的结果上再次调用函数。

```
 assoc :: Expr -> Expr
 assoc (Add (Add e1 e2) e3)
   = assoc (Add e1 (Add e2 e3))         (Add.1)
```

定义中的其他情况确保一个表达式的组成部分正确重组

```
 assoc (Add e1 e2)
   = Add (assoc e1) (assoc e2)          (Add.2)
 assoc (Sub e1 e2)
   = Sub (assoc e1) (assoc e2)
 assoc (Lit n)
   = Lit n
```

等式 (Add.2) 只用于 (Add.1) 不适用的情况，即 e1 或者是 Sub 或者是 Lit 表达式。这是模式匹配的规则：使用第一个匹配的等式。在使用原始递归时，我们确信递归将终止，并且给出一个答案：因为递归调用只应用于更小的表达式，所以，经过有限步后必然到达递归基。函数 assoc 要复杂一些，我们需要说明函数一定会给出答案。等式 (Add.1) 是说明的关键，但是，直观上，我们可以看出转换过程在取得进展，树的“重量”已经由左移至右。特别地，一个加法运算符组织到了右边。在其他等式中不存在加法向其他方向移动的现象，所以，在使用 (Add.1) 有限次后，加法运算不再出现在表达式的顶层的左边。这表明递归不可能无限的进行下去，所以函数一定会给出一个结果。

中缀构造符

我们知道，函数可以表示为中缀形式，这也适用于构造符函数。例如，我们可以重新定义函数 assoc：

```
assoc ((e1 `Add` e2) `Add` e3)
  = assoc (e1 `Add` (e2 `Add` e3))
  ...
```

其中使用了用反引号括起来的构造符函数的中缀形式。

当显示这个表达式时，它仍然以前缀形式出现，如表达式

```
(Lit 3) `Add` (Lit 4)
```

将显示为

```
Add (Lit 3) (Lit 4)
```

在 Haskell 的 data 定义中，我们可以定义作为运算符的构造符。这些构造符与运算符有相同的语法，不同的是它们的开始字符必须是':', 注意，':' 本身是一个中缀构造符。对于整数表达式类型，我们可以定义

```
data Expr = Lit Int |
            Expr :+: Expr |
            Expr :-: Expr
```

当含有运算构造符的表达式被显示时，构造符出现在中缀位置。

请读者完成在重新定义的类型 Expr 上的其它函数的定义。

相互递归

在描述一个类型时，我们经常需要使用其他类型，而这些类型可能会引用原来的类型：由此得到一对相互递归的类型。一个人的描述可能包含生平简历，生平又会涉及到其他人。例如：

```
data Person = Adult Name Address Biog |
              Child Name
data Biog   = Parent String [Person] |
              NonParent String
```

对于父母的情况，生平包含一些文字，以及他们的子女，后者可以视为类型 Person 上的列表。

假如我们想定义一个函数来把一个人的信息显示为一个串。这些信息的显示包括一些生平信息，而这些信息本身包含了有关人的进一步信息。于是我们必须使用两个相互递归的函数。

```
showPerson (Adult nm ad bio)
  show nm ++ show ad ++ showBiog bio
showBiog (Parent st perList)
  = st ++ concat (map showPerson perList)
  ...
```

习题

14.15 试计算下列式子

```
eval (Lit 67)
eval (Add (Sub (Lit 3) (Lit 1)) (Lit 3))
show (Add (Lit 67) (Lit -34))
```

14.16 定义下列计算表达式中运算符个数的函数

```
size :: Expr -> Int
```

14.17 在类型Expr中添加乘法和除法，然后重新定义函数eval，show和size。除数为零时如何定义eval?

14.18 我们可以如下定义类型Expr

```
data Expr = Lit Int |
            Op Ops Expr Expr
data Ops = Add | Sub | Mul | Div
```

在此类型上重新定义函数eval，show和size，并说明如果添加一个整除运算Mod，你的定义必须做哪些修改。

14.19 写出下列式子的逐行计算过程

```
sumTree (Node 3 (Node 4 NilT NilT) NilT)
depth   (Node 3 (Node 4 NilT NilT) NilT)
```

14.20 在使用中缀构造符的类型Expr定义之后完成Expr上函数的定义。

14.21 定义分别返回一棵NTree树的左右子树的函数。

14.22 定义判定一个整数是否属于一棵NTree树的函数。

14.23 定义一个将一棵NTree树的左右子树（递归）反转的函数reflect。试问反转两次reflect . reflect的结果是什么?

14.24 定义下列函数

```
collapse, sort :: NTree -> [Int]
```

函数collapse顺序列出左子树上的值，结点上的值和右子树上的值。函数sort将树上的元素按照非递减序排列。例如，

```
collapse (Node 3 (Node 4 NilT NilT) NilT) = [4,3]
sort     (Node 3 (Node 4 NilT NilT) NilT) = [3,4]
```

14.25 完成函数showPerson和showBiog的定义。

14.26 我们可以将类型Expr扩充使其包含条件表达式If b e1 e2，其中b是属于类型BExp的元素。当b的值为True时，If b e1 e2的值是e1，否则是e2。

```
data Expr = Lit Int |
            Op Ops Expr Expr |
            If BExp Expr Expr
data BExp = BoolLit Bool |
            And BExp BExp |
            Not BExp |
```

```
            Equal Expr Expr |
            Greater Expr Expr
```

以上的五个构造符具有明显的解释

- 布尔值：BoolLit True和BoolLit False;
- 两个表达式的合取；如果两个表达式的值均为真，则结果为真；
- 表达式的否定；Not be的值为True当且仅当be的值为False。
- 表达式Equal e1 e2的值为True当且仅当e1与e2的值相等。表达式Greater e1 e2的值为True当且仅当e1的值大于e2的值。

试利用相互递归定义下列函数以及这些类型上的show函数：

```
eval  :: Expr -> Int
bEval :: BExp -> Bool
```

§14.3 多态代数类型

代数类型定义可以包含类型变量 a, b 等，从而定义多态类型。定义的格式如前，定义中使用的类型变量出现在定义左边类型名之后。一个简单的例子是

```
data Pairs a = Pr a a
```

其中包含的元素有

```
Pr 2 3     :: Pairs Int
Pr [] [3]  :: Pairs [Int]
Pr [] []   :: Pairs [a]
```

测试一个二元组的两个分量是否相等的函数定义为

```
equalPair :: Eq a => Pairs a -> Bool
equalPair (Pr x y) = (x==y)
```

下面将列出更多的多态类型。

列表

预定义列表类型由下列定义给出

```
data List a = NilList | Cons a (List a)
              deriving (Eq, Ord, Show, Read)
```

其中 [a],[ ] 和':' 分别用于表示 List a, NilList 和'Cons'。列表类型是递归多态类型的一个很好范例。特别地，我们能看到在这些类型上定义一类函数的可能性，以及通过使用结构归纳法来进行程序验证的方法。

二叉树

14.2 节的树上每个结点带有一个整数；整数在这里不具有特殊性，我们可以在每个结点上记录任何类型的元素：

```
data Tree a = Nil | Node a (Tree a)(Tree a)
              deriving (Eq, Ord, Show, Read)
```

函数 depth 和 occurs 的定义可以不加改变地照搬到这里：

```
depth :: Tree a -> Int
depth Nil            = 0
depth (Node n t1 t2) = 1 + max (depth t1) (depth t2)
```

同样，14.2 节习题中许多有关的函数也适用于上述定义的类型。其中的一个函数是将一颗树转化为一个列表。这个过程可以通过中根次序遍历二叉树完成，即先遍历左子树，再访问根，再遍历右子树：

```
collapse :: Tree a -> [a]
collapse Nil = []
collapse (Node x t1 t2)
  = collapse t1 ++ [x] ++ collapse t2
```

例如

```
collapse (Node 12
               (Node 34 Nil Nil)
               (Node 3 (Node 17 Nil Nil) Nil))
 = [34, 12, 17, 3]
```

我们还可以定义许多其他的高阶函数，例如，

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f Nil = Nil
mapTree f (Node x t1 t2)
  = Node (f x) (mapTree f t1) (mapTree f t2)
```

我们将在 16.7 节继续讨论树，特别是“查找”树。

联合类型 Either

类型定义可以带有两个以上的参数。我们曾看到元素或者是一个名或者是一个数的类型。一般地，我们可以定义一个类型，其元素或者来自于 a 或者来自于 b：

```
data Either a b = Left a | Right b
                  deriving (Eq, Ord, Read, Show)
```

“联合”或者“和”类型的元素是 (Left x)，其中 x::a 和 (Right y)，其中 y::b。或者名或者数的类型可表达为 Either String Int , 并且

```
Left "Duke of Prunes" :: Either String Int
Right 33312           :: Either String Int
```

我们可以利用下列函数区分一个元素是否属于联合的前半部分

```
isLeft :: Either a b -> Bool
isLeft (Left _)  = True
```

Either a b

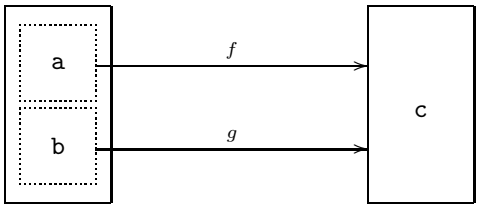


图 14.3: 连接两个函数

```
isLeft (Right _) = False
```

要定义由 Either a b 到另一个类型，如 Int 的函数，我们必须分两种情况，

```
fun :: Either a b -> Int
fun (Left x)  = ... x ...
fun (Right y) = ... y ...
```

在第一种情况，等式右边 x 被转换为 Int，由一个从 a 到 Int 的函数完成；在第二种情况，y 被转换成一个 Int，因此由一个从 b 到 Int 的函数完成。由上面分析看出，我们可以定义一个高阶函数，它将定义在 a 和 b 上的函数结合起来，从而定义 Either a b 上的函数，其定义如下，如图 14.3 所示:

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left x)   = f x
either f g (Right y)  = g y
```

如果我们有一个函数`f::a->c`，而且希望将其应用于 Either a b 上的一个元素，这里有一个问题：如果参数为 Either a b 的右边部分，即形如 (Right y)，函数的值如何决定呢？最简单的答案是返回一个错误信息

```
applyLeft :: (a -> c) -> Either a b -> c
applyLeft f (Left x)  = f x
applyLeft f (Right y) = error "applyLeft applied to Right"
```

在下节我们将讨论处理错误的其他方法。

习题

14.27 在第 *14.2* 节习题中哪些树上的函数是多态的？

14.28 定义函数`twist`，它将交换并的次序：

```
twist :: Either a b -> Either b a
```

`twist . twist`的结果是什么？

14.29 如何使用函数`either`定义函数`applyLeft`？

14.30 试说明任何类型为`a -> b`的函数都可以转化为下列类型的函数

```
a -> Either b c
a -> Either c b
```

14.31 如何将`either`推广为`join`使其具有类型

```
join :: (a -> c) -> (b -> d) -> Either a b -> Either c d
```

你或许发现前一习题对于使用either定义join有帮助。

前面定义的树称为二叉树：每个非空树有两个子树。我们可以定义一般的树，即一个结点有一个子树列表：

```
data GTree a = Leaf a | Gnode [GTree a]
```

随后的习题讨论这样的树。

14.32 定义下列函数

- 计算一个GTree的叶数。
- 求一个GTree的深度。
- 计算GTree Int上数字之和。
- 判定一个元素是否出现在一个GTree上。
- 将一个函数映射到GTree叶结点元素上。
- 将一个GTree转换成一个列表。

14.33 在GTree中如何表示一颗空树？

§14.4 实例研究：程序错误

一个程序如何处理不应该发生的情况呢？这样的情况包括

- 试图用 0 做除数，取负数的平方根等错误的算术表达式；
- 对一个空列表求首元素，这是代数类型定义中未定义的一种特殊情形。

本章给出处理这些情况的三种方法。最简单的是停止计算，并报告错误的来源。这也是Haskell处理上述情况所采取的措施。我们可以在定义中使用函数error

```
error :: Sring -> a
```

计算表达式error "Circle with negative radius"的结果是：在屏幕上显示下列结果并终止计算

```
Program error: Circle with negative radius
```

这种方法的缺陷是计算过程中的所有有用信息丢失。下面介绍的两种方法将用某种方法处理错误，而不会完全终止计算。

哑值

函数tail应该返回一个列表的尾，对于空列表则返回一个错误信息：

```
tail :: [a] -> [a]
tail (_:xs) = xs
tail []     = error "Prelude.tail: empty list"
```

我们可以重新定义此函数

```
tl :: [a] -> [a]
tl (_:xs) = xs
tl []     = []
```

现在，对于任意列表取尾都会成功。类似地，我们可以定义

```
divide :: Int -> Int -> Int
divide n m
  | (m/=0)    = n `div` m
  | otherwise = 0
```

使得 0 做除数也有一个结果。函数tl和divide在错误的情况下有一个较明显的值，但是，对于函数head情况并非如此，对于这种函数，我们可以给函数提供一个额外的参数作为错误情况下的值。

```
hd :: a -> [a] -> a
hd y (x:_) = x
hd y []    = y
```

这种方法具有普遍适用性；如果函数f（假定它有一个参数）在条件cond成立时会引起一个错误，我们可以定义一个新函数

```
fErr y x
  | cond      = y
  | otherwise = f x
```

这种方法的缺陷是，单从结果不能判定什么时候发生了错误，因为结果 y 可能是错误情况下的值，也可能是正常情况下的值。另一种方法是利用错误类型捕获错误并处理错误。

错误类型

前一种方法在错误发生时返回一个哑值。何不返回一个错误值作为结果呢？我们定义类型

```
data Maybe a = Nothing | Just a
               deriving (Eq, Ord, Read, Show)
```

此类型在类型 a 上添加了一个特殊值Nothing。我们现在可以定义一个除法函数 errDiv

```
errDiv :: Int -> Int -> Maybe Int
errDiv n m
  | (m/=0)     = Just (n `div` m)
  | otherwise  = Nothing
```

一般地，如果函数 f 在条件cond成立时引起错误，可以定义

```
fErr x
  | cond       = Nothing
  | otherwise  = Just (f x)
```

这些函数的结果类型不是原来的类型如a，而是类型Maybe a。这种 Maybe 类型允许我们列出潜在的错误。对于潜在的错误我们可以做两种处理。

- 我们可以在函数中传递错误，如函数mapMaybe的作用；

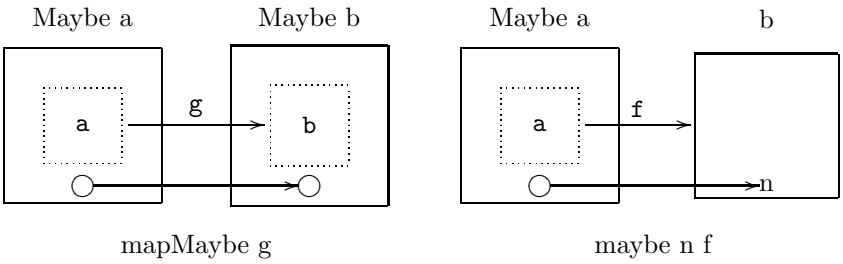


图 14.4: 错误处理函数

- 我们可以捕获错误，函数maybe的作用。

这些函数的作用如图 14.4 所示，下面给出定义。

函数mapMaybe通过函数应用 g 传递错误值。假定函数 g 的类型为a->b，我们可以将它的参数类型提升为Maybe a。假如参数为Just x，则可以将 g 应用于 x，得到类型为 b 的结果g x，然后应用构造符 Just 将其置入类型Maybe b。如果参数为 Nothing，则结果也是 Nothing。

```
mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe g Nothing  = Nothing
mapMaybe g (Just x) = Just (g x)
```

在捕获到一个错误时，我们希望由输入类型Maybe a返回一个类型 b 的值。有两种情况需要处理：

- 在 Just 的情况下，应用从 a 到 b 的函数；
- 在 Nothing 的情况，我们必须提供类型 b 的值（正如前面看到的函数 hd 的情况。）

完成这种任务的高阶函数是maybe，其参数 n 和 f 分别应用于 Nothing 和 Just 两种情况：

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing  = n
maybe n f (Just x) = f x
```

在下面的例子中可以看到这两个函数的应用。在第一个例子中，0 做除数导致 Nothing，这个错误被提升后捕获，因此，最后返回 56：

```
maybe 56 (+1) (mapMaybe (*3) (errDiv 9 0))
= maybe 56 (+1) (mapMaybe (*3) Nothing)
= maybe 56 (+1) Nothing
= 56
```

在第二个例子中，除法返回Just 9，然后乘 3，最后maybe应用加 1 并去除了 Just:

```
maybe 56 (+1) (mapMaybe (*3) (errDiv 9 1))
= maybe 56 (+1) (mapMaybe (*3) (Just 9))
= maybe 56 (+1) (Just 27)
```

```
= 1+27
= 28
```

这种方法的优点是，我们可以先定义无错误处理的系统，然后加上错误处理，方法是使用函数mapMaybe和maybe以及加上错误处理的函数。我们已经多次看到，将一个问题一分为二使得每一部分的解以及整个解更容易理解。

我们将在第 18.8 节看到，Maybe 类型是一个更通用的程序设计结构 monad 的一个例子。我们还将探讨函数 mapMaybe 和 map 在列表上的关系。

习题

14.34 使用函数mapMaybe和maybe定义下列函数

```
process :: [Int] -> Int -> Int -> Int
```

使得process xs n m返回xs中第n个和第m个元素的和。一个长度为 p 的列表的下标是 *0,1,...,p-1*。如果process的任何一个下标参数超出列表的下标界，则结果为 *0*。

14.35 讨论本节介绍的三种错误处理方法各有哪些优点和缺点。

14.36 类型Maybe (Maybe a)的值是什么？定义函数

```
squashMaybe :: Maybe (Maybe a) -> Maybe a
```

它将Just (Just x)“挤压”成Just x，其他值“挤压”成Nothing。

14.37 类似于mapMaybe，定义错误处理函数的复合

```
composeMaybe :: (a -> Maybe b) ->
                (b -> Maybe c) ->
                (a -> Maybe c)
```

如何使用mapMaybe，复合运算和挤压函数定义composeMaybe？

14.38 我们可以推广类型Maybe使得Nothing包含错误信息：

```
data Err a = OK a | Error String
```

如何在这个类型上定义相应的函数mapMaybe，maybe和composeMaybe？

§14.5 使用代数类型设计系统

代数数据类型为构造问题中的类型和解决问题的程序中的类型提供了强大的工具。本节介绍构造代数类型的三阶段方法，并将这种方法应用于求两个词的“编辑距离”和一个仿真问题。

本节的一个重要原则是：开始的类型设计独立于程序本身。无论系统的规模有多大，我们都应该遵循这样的原则，因为分别考虑系统的不同部分使系统的实现更容易成功。

在接下来的两章，我们将会讨论更多有关数据类型的设计。

编辑距离：问题的陈述

在讨论设计阶段时，我们将以求两个串之间的 **编辑距离** 为例。两个串之间的编辑距离是将一个串转换成另一个串的最短简单编辑运算序列。

这个例子是一个实际问题的版本：为了及时更新显示（窗口或者文本），更新的速度是很关键的。因此，我们希望更新可以用尽可能少的基本操作完成；这便是编辑距离程序所完成的任务。

假定串上的基本编辑操作有五种：将一个字符修改为另一个字符，拷贝一个字符，删除一个字符，插入一个字符和删除到串的尾部。我们假定除拷贝运算外，其他运算的开销是相同的，而拷贝不需要任何开销。

将串"fish" 转换成"chips"，一种方法是删除整个串，然后逐个插入字符，共需 6 个操作。另一个解决方法是拷贝尽可能多的字符：

- 插入字符'c'；
- 将'f' 修改为'h'；
- 拷贝'i'；
- 插入'p'；
- 拷贝's'；
- 最后，删除剩余部分"h"。

接下来我们设计一个表示编辑步骤的类型，在介绍完数据类型设计的另一个例子后，我们给出从一个串到另一个串的最优编辑步骤序列的函数。

这里的分析也可用于描述任意类型的两个列表之间的差别。如果列表的元素是一个文件的文本行，那么函数的功能类似于 Unix 程序 diff，其功能是返回两个文本文件之间的差异。

编辑距离的设计阶段

下面详细讨论定义代数类型的三阶段。

- 首先，我们必须确定问题涉及的数据的类型。对于目前的例子，我们必须定义表示编辑运算的类型

```
data Edit = ...
```

- 接下来，我们必须确定每个类型中的不同种类的数据。每种数据有一个构造符。在本例中，我们可以修改、拷贝、删除或者插入一个字符，以及删除至串尾。所以，类型的定义形如

```
data Edit = Change ... |
            Copy ... |
            Delete ... |
            Insert ... |
            Kill  ...
```

其中... 表示构造符的类型还未定。

- 最后，对于每个构造符，我们需要确定它的组成部分或者参数是什么。有些构造符，如 Copy, Delete 和 Kill，不需要其他信息；其他构造符需要说明新插入的字符，所以

```
data Edit = Change Char |
            Copy   |
            Delete |
            Insert Char |
            Kill
            deriving (Eq,Show)
```

  由此完成了类型的定义。

  在给出编辑距离解之前，我们说明如何用类似的方式定义其他类型。

仿真

假设我们要模仿或者 **仿真** 邮局或者银行的队列的变化情况；或许我们想确定在一天的某些时刻需要多少雇员服务。系统将把客户的到来作为输入，客户的离开作为输出。所有这些可以用类型来建模。

- Inmess 是输入信息的类型。在给定的时刻，有两种可能性：
  - 无客户到来，用 0 元构造符 No 表示；
  - 某客户到来，用构造符 Yes 表示。此构造符的参数包括客户到达的时间和需要服务的时间。

  为此，我们定义

```
data Inmess = No | Yes Arrival Service


type Arrival = Int
type Service = Int
```

- 类似地定义输出信息类型 Outmess。或者没有客户离开 (None)，或者一个客户离开 (Discharge)。客户的相关信息包括他们等待的时间以及到达时间和服务时间。为此，我们定义

```
data Outmess = None | Discharge Arrival Wait Service
type Wait = Int
```

我们将在第 16 章返回这个仿真例子。

编辑距离之解

我们的目标是找到将一个串转换成另一串的最小代价编辑序列。我们可以如下定义这个函数

```
transform :: String -> String -> [Edit]


transform [] [] = []
```

```
transform xs [] = [Kill]
transform [] ys = map Insert ys
```

将 xs 转换成[], 只需删除 xs; 将[]转换为 ys, 必须依次插入每个字符。

对于一般的情况，我们有多种选择：首先使用 Copy, Delete, Insert, 还是 Change? 如果两个串的首字符相等，那么应该使用 Copy；否则，没有显然的选择。为此，我们尝试所以的可能，并选择其中代价最小的序列。

```
transform (x:xs) (y:ys)
  | x==y       = Copy : transform xs ys
  | otherwise  = best [Delete   : transform xs (y:ys),
                       Insert y : transform (x:xs) ys,
                       Change y : transform  xs ys]
```

如何选择最好的序列呢? 我们选择代价最小的序列。

```
best :: [[Edit]] -> [Edit]
best [x] = x
best (x:xs)
 | cost x <= cost b = x
 | otherwise        = b
   where
   b = best xs
```

编辑运算的代价均为 1 ，拷贝除外，其代价为 0 ：

```
cost :: [Edit] -> Int
cost = length . filter (/=Copy)
```

习题

前四个习题帮助读者理解如何设计数据类型。这些问题没有唯一的 “正确” 答案，读者应该确认你定义的类型表达了问题中的数据。

14.39 假设一个停车场需要记录使用此停车场的机动车。试设计一个代数数据类型表示之。

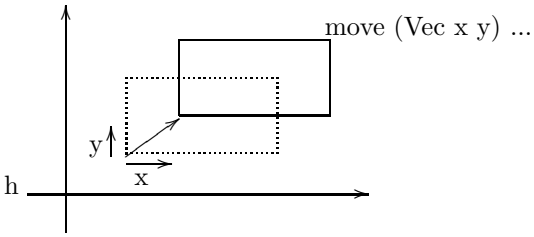
14.40 假设机动车记录用于燃料效率的测试，如何修改上题的答案使其适用于本应用。

14.41 讨论学生成绩数据库中使用的数据类型，并解释你使用的任何代数数据类型。

14.42 什么样的数据类型可以表示交互式绘画程序所绘出的对象？更有挑战性的是考虑对象的组合，对象的多个拷贝以及对象的放大和缩小。

14.43 如何修改编辑距离程序使其可以包含交换运算Swap，例如，"abxyz"可以经一次交换变成"baxyz"。

14.44 定义一个函数：给定一个编辑序列和一个串，函数将编辑序列依次应用于输入串。



14.45 试计算`transform "cat" "am"`。由此你对`transform`的效率有何结论？

§14.6 代数类型与类型分类

现在我们可以展示在第十二章介绍的类型分类的更大的例子了。

可移动物体

我们来构造一类 (class) 类型，类型的成员是二维几何物体。这个类上的运算包括在各个方向上移动物体的运算。

下面给出类的定义，见图 14.5。有些移动是由向量表示的，所以，我们定义

```
data Vector = Vec Float Float
```

类的定义如下

```
class Movable a where
  move      :: Vector -> a -> a
  reflectX  :: a -> a
  reflextY  :: a -> a
  rotate180 :: a -> a
  rotate180 = reflectX. reflectY
```

定义说明物体可以移动的方式。首先，物体可以移动一个向量，如下图所示

我们还可以相对于 x 轴（水平轴）或者 y 轴（垂直轴）将物体反射，或者关于原点旋转 180°。旋转`rotate180`可以分解成相对于 y 轴的反射和相对于 x 轴的反射，类似于第一章`Picture`上的旋转实现。

我们可以定义一系列可移动物体。首先是`Point`:

```
data Point = Point Float Float
             deriving Show
```

我们需要定义`Point`上的函数`move`, `refelctX`和`refelctY`使其成为 Movable 的特例。

```
move (Vec v1 v2) (Point c1 c2) = Point (c1+v1) (c2+v2)
reflectX (Point c1 c2) = Point c1 (-c2)
reflectY (Point c1 c2) = Point (-c1) c2
```

我们使用一个更有效的定义覆盖`rotate180`的定义。

```
totate180 (Point c1 c2) = Point (-c1) (-c2)
```

利用点的类型可以构造图形：

```
data Figure = Line Point Point |
              Circle Point Float
```

在Figure的Movable特例的定义中，我们可以使用Point上的运算，见图 14.5, 例如

```
  move v (Line p1 p2) = Line (move v p1) (move v p2)
  move v (Circle p r) = Circle (move v p) r
```

类似的方法适用于可移动物体的列表

```
instance Movable a => Movable [a] where
  move v   = map (move v)
  reflectX = map reflectX
```

这种重载具有许多优点，例如

- 代码更容易阅读：在移动一个点时，可以书写move，而不需使用movePoint等。
- 我们可以重用定义；Movable [a]的特例定义使得任意类型可移动物体的列表也成为可移动物体，包括点的列表和图形的列表。没有重载我们很难达到上述目的。

命名物体

许多类型的数据包含某种名称，例如一个 String 标明一个物体。对于这样的名称我们可以做什么呢？

- 我们应该能够识别一个物体的名称；
- 我们应该能够赋予一个物体一个名称。

这些运算可以嵌入到Named类中

```
class Named a where
  lookName :: a -> String
  giveName :: String -> a -> a
```

一个命名物体类型的例子是

```
data Name a = Pair a String
```

此类型由一个构造符构成，它的两个成分类型分别是 a 和 String，使其成为Named的特例定义是

```
instance Named (Name a) where                          (1)
  lookName (Pair obj nm) = nm
  giveName nm (Pair obj _) = (Pair obj nm)
```

类的组合

面向对象软件开发的一个特色是一个类（子类）可以建立在另一个类（父类）上，子类可以重用父类上的运算。本节讨论如何将Movable和Named组合起来以表示既可移动又可命名的物体。本节内容比较深，初学者可以略过。

```
data Vector = Vec Float Float

class Movable a where
  move      :: Vector -> a -> a
  reflectX  :: a -> a
  reflectY  :: a -> a
  rotate180 :: a -> a
  rotate180 = reflectX . reflectY

data Point = Point Float Float
             deriving Show

instance Movable Point where
  move (Vec v1 v2) (Point c1 c2) = Point (c1+v1) (c2+v2)
  reflectX (Point c1 c2)  = Point c1 (-c2)
  reflectY (Point c1 c2)  = Point (-c1) c2
  rotate180 (Point c1 c2) = Point (-c1) (-c2)

data Figure = Line Point Point |
              Circle Point Float
              deriving Show

instance Movable Figure where
  move v (Line p1 p2) = Line (move v p1) (move v p2)
  move v (Circle p r) = Circle (move v p) r

  reflectX (Line p1 p2) = Line (reflectX p1) (reflectX p2)
  reflectX (Circle p r) = Circle (reflectX p) r

  reflectY (Line p1 p2) = Line (reflectY p1) (reflectY p2)
  reflectY (Circle p r) = Circle (reflectY p) r

instance Movable a => Movable [a] where
  move v   = map (move v)
  reflectX = map reflectX
  reflectY = map reflectY
```

图 14.5: 可移动物体

假设我们希望给可移动物体添加名称，如何完成这个任务呢？我们在这里介绍一种方法，在习题中探讨另一种方法。

我们的方法是构造类型Name a，其中 a 的元素是可移动的，即Movable a成立。然后我们说明类型Name a同时属于类Movable和Named。我们已经在 (1) 中说明后者，下面说明前者。

注意到命名并不依赖于被命名的类型，所以类型上的运算可以提升到被命名的类型上：

```
mapName :: (a -> b) -> Name a -> Name b

mapName f (Pair obj nm) = Pair (f obj) nm
```

我们可以说明类Movable上的运算均可提升：

```
instance Movable a => Movable (Name a) where          (2)
  move v   = mapName (move v)
  reflectX = mapName reflectX
  reflectY = mapName reflectY
```

我们已经知道Named (Name a)，现在可以定义一个结合两个属性的类

```
class (Movable b, Named b) => NamedMovable b          (3)
```

然后说明如下特例

```
instance Movable a => NamedMovable (Name a)
```

这是因为当用Name a代替 (3) 中的 b 时，其中的两个约束均满足，即在Movable a的前提下，约束 (1) 和 (2) 满足。

由此我们完成了在Movable a的前提下说明NamedMovable (Name a)。值得注意的是，这些是由 Haskell 系统自动完成的，我们只需输入图 14.6 的内容。

本节已经开始说明类如何用于软件的开发过程。特别地，我们看到如何命名可移动物体并重用移动物体的所有代码。

习题

14.46 另一种结合类Named和Movable的方法是构造下列特例

```
instancee (Movable b, Named c) => NamedMovable (b,c)
```

方法是说明下列特例

```
instance Movable b => Movable (b,c) where ...
instance Named c   => Named (b,c) where ...
```

完成上述特例说明。

14.47 说明前一个习题所述的方法可用于结合任意两类的特例。

14.48 本节最后一个例子说明如何将类Movable的任意特例a与类Named的一个特例String结合。试说明如何用这种方法结合一个类的任意特例与另一个类的某个特例。

```
data Name a = Pair a String

exam1 = Pair (Point 0.0 0.0) "Dweezil"

instance Named (Name a) where                    (1)
 lookName (Pair obj nm) = nm
 giveName nm (Pair obj _) = (Pair obj nm)

mapName :: (a -> b) -> Name a -> Name b

mapName f (Pair obj nm) = Pair (f obj) nm

instance Movable a => Movable (Name a) where       (2)
  move v   = mapName (move v)
  reflectX = mapName reflectX
  reflectY = mapName reflectY

class (Movable b, Named b) => NamedMovable b       (3)

instance Movable a => NamedMovable (Name a)
```

图 14.6: 命名可移动物体

14.49 扩充可移动物体的运算，使其包括收放和任意角度的旋转。你可以重定义Movable，也可以在Movable上定义一个类MovablePlus。试说明哪一种方法更好。

14.50 设计一个类的集合来为银行账户建模。你能使用类Named吗？

§14.7 代数类型的推理

代数类型的验证遵循第八章讨论的列表验证的例子。代数类型上的结构归纳原理的一般模式是：证明结论对于每个构造符成立；如果一个构造符是递归的，我们可以在证明中使用相应的归纳假设。本节先介绍一些有代表性的例子，最后是一个相当复杂的证明例子。

树

类型Tree上的结构归纳原理如下。

树上的结构归纳

欲证明性质P(tr)对于类型Tree t的所有有穷元素tr成立，我们需要证明下列两点：

Nil情况　证明P(Nil)

Node情况　证明P(Node x tr1 tr2)对于类型t的所有x成立
假定P(tr1)和P(tr2)成立。

第八章介绍的寻找证明的建议很容易搬到这种情况。这里给出一个证明例子。我们证明对于所有的有穷树tr下列性质成立

```
map f (collapse tr0 = collapse (mapTree f tr)      (map-collapse)
```

即如果将一个函数映射到一棵树上，然后将其转化为列表，其结果等同于先将树转化为列表，然后将函数映射到列表上。其中的函数定义如下

```
map f []       = []                          (map.1)
map f (x:xs)   = f x : map f xs              (map.2)

mapTree f Nil = Nil                                  (mapTree.1)
mapTree f (Node x t1 t2)
  = Node (f x) (mapTree f t1) (mapTree f t2)         (mapTree.20

collapse Nil = []
collapse (Node x t1 t2)
  = collapse t1 ++ [x] ++ collapse t2
```

基的情况 对于Nil的情况，我们只需简化等式两边：

```
map f (collapse  Nil)
  = map f []                                by (collapse.1)
```

```
  = []                                          by (map.1)

collapse (mapTree f Nil)
  = collapse Nil                              by (mapTree.1)
  = []                                       by (collapse.1)
```

由此完成了归纳基的证明.

归纳步 对于Node的情况，我们证明

```
map f (collapse (Node x tr1 tr2))              (ind)
  = collapse (mapTree f (Node x tr1 tr2))
```

其中归纳假设为

```
map f (collapse tr1) = collapse (mapTree f tr1)        (hyp.1)
map f (collapse tr2) = collapse (mapTree f tr20        (hyp.2)
```

我们首先简化 (ind) 的左边

```
map f (collapse (Node x tr1 tr2))
 = map f (collapse tr1 ++ [x] ++ collapse tr2)              by (collapse.2)
 = map f (collapse tr1) ++ [f x] ++ map f (collapse tr2)    by (map++)
 = collapse (mapTree f tr1) ++ [f x] ++ collapse (mapTree f tr2)
                                                            by
                                                            (hyp.1,hyp.2)
```

其中最后一步由归纳假设推出；前一步 (map++) 由下列定理推出（见第十章)

```
map g (ys++zs) = map f ys ++ map g zs
```

再检查 (ind) 的右边

```
collapse (mapTree f (Node x tr1 tr2))
 = collapse (Node (f x) (mapTree f tr1)
                        (mapTree f tr2))      by (mapTree.2)
 = collapse (mapTree f tr1) ++ [f x] ++
   collapse (mapTree f tr2)                   by (collapse.2)
```

由此完成归纳步的证明，同时完成了性质 (map-collapse) 的证明。 □

类型 Maybe

类型Maybe t上的结构归纳变成分情形证明，因为这个类型不是递归的，在证明中不需要使用归纳假设。

Maybe 类型上的结构归纳法

欲证明性质P(tr)对于类型Maybe t的所有有定义元素tr成立，我们需要证明下列两点：

Nothing情况	证明P(Nothing)
Just情况	证明P(Just x)对于类型t的所有有定义x成立

下面是一个证明例子，证明对于类型Maybe Int的所有有定义值 x

maybe 2 abs x ≥ 0

证明 1 证明分两种情况。第一种情况x是Nothing：

```
maybe 2 abs x ≥ 0
  = 2 ≥ 0
```

第二种情况是 x 为Just y:

```
maybe 2 abs (Just y)
  = abs y ≥ 0
```

两种情况均成立，所有性质成立。

其他形式的证明

我们已经看到，并非所有的函数都是原始递归的。例如，14.2 节定义的表达式重组的函数assoc:

```
assoc (Add (Add e1 e2) e3)
  = assoc (Add e1 (Add e2 e3))              (assoc.1)
assoc (Add e1 e2)
  = Add (assoc e1) (assoc e2)               (assoc.2)
assoc (Sub e1 e2)
  = Sub (assoc e1) (assoc e2)               (assoc.3)
assoc (Lit n)      = Lit n                  (assoc.4)
```

其中 (assoc.1) 是非原始递归的情况。我们将证明，这种重组并不会影响表达式的值：

```
eval (assoc ex) = eval ex                   (eval-assoc)
```

即证明上式对于任意有穷表达式 ex 成立。类型Expr上的归纳原理分三种情况：

Lit的情况　证明P(Lit n)成立。

Add的情况　证明P(Add e1 e2)成立，假定P(e1)和P(e2)成立。

Sub的情况　证明P(Sub e1 e2)成立，假定P(e1)和P(e2)成立。

欲证明(eval-assoc)，我们必须证明上述三种情况均成立。其中 Lit 和 Sub 的情况由 (assoc.3) 和 (assoc.4) 分别给出，但是，Add 的情况更复杂。为此，我们需要证明

```
eval (assoc (Add e1 e2)) = eval (Add e1 e2)
```

方法是对于 e1 顶层中的左嵌套 Add 数目使用归纳法。注意，我们正是通过说明这个数目递减的方法说明函数 assoc 终止。如果 e1 的顶层没有 Add, 则方程 (assoc.2) 说明 (eval-assoc) 成立。否则，我们做如下调整

```
eval (assoc (Add (Add f1 f2) e2))
= eval (assoc (Add f1 (Add f2 f3)))          by (assoc.1)
```

因为 f1 的顶层含有较少的 Add, 所以, 计算可以如下继续

```
= eval (Add f1 (Add f2 f3))
```

```
 = eval (Add (Add f1 f2) f3)                    +的结合律
```

由此完成归纳步证明，也结束了性质 (eval-assoc) 的整个证明。 □

这个例子也说明，我们可以对于非原始递归的更一般函数定义进行验证。

习题

14.51 证明 *14.1* 节定义的函数weather与下列函数具有相同的功能

```
newWeather = makeHot . isSummer
```

其中

```
makeHot True   = Hot
makeHot False  = Cold
issummer = (==Summer)
```

14.52 第 *14.1* 节定义的每个Shape的面积area均是正的吗？如果是，给出证明，否则，给出一个反例。

14.53 假如定义NTree的size如下

```
size (NilT          = 0
size (Node x t1 t2) = 1 + size t1 + size t2
```

试证明对于所有类型为NTree的有穷tr

```
size tr < 2^(depth tr)
```

14.54 试证明对于所有类型为NTree的有穷tr

```
occurs tr s = length (filter (==x) (collapse tr))
```

下面两个习题需要参考 14.3 节习题。

14.55 证明函数twist具有下列性质

```
twist . twist = id
```

14.56 解释类型GTree上的结构归纳原理。陈述并证明此类型上有关map，mapTree和collapse之间等价关系的定理。

小结

代数类型增强了我们在程序中建立类型的能力：我们在本章看到如何定义简单的由穷类型如Temp以及更复杂的Either和递归类型。许多递归类型是树的变种：我们介绍了数值树；类型Expr的元素也可以视为表示算术表达式结构的树。

列表类型对于理解递归类型具有指导意义。

- 列表类型的定义是递归的、多态的，并且在列表类型上可以定义许多多态高阶函数；这些对于树类型和错误类型如Maybe也成立。
- 结构归纳是对列表推理的简单原理，也是对于代数类型进行推理的原理。

本章同时介绍了定义代数类型的方法。定义分三部分：首先是类型名，然后是构造符，最后是构造符的成分的类型。代数类型的这种分解成分方法有助于程序员构造简单、正确的系统。

有了代数类型的概念后，我们可以介绍规模更大的分类及其特例的例子。我们看到，由分类引进的重载使得代码更容易阅读，也更容易重用；特别地，我们看到系统的扩充只需要对代码做很小的修改后便可完成。

在接下来的章节中，代数类型将成为系统不可分割的部分，在下一个实例中我们将展示这些类型的各种特点。我们还将探讨定义类型的另一种途径：抽象数据类型，并将看到抽象代数类型与代数类型的互补性。

第十五章　实例研究：Huffman 编码

本章利用一个实例研究来说明前一章讲到的多态、代数类型和程序设计，并且演示 Haskell 的模块系统。

§15.1　Haskell 的模块

如 2.4 节所述，一个模块由类型、函数等定义构成，而且有一个说明此模块输出哪些定义的界面。使用模块组织一个大型程序有下列优点:

- 系统的某些部分可以独立地开发。假设我们需要鉴视一个网络的流量：可能有一个模块用于产生统计数字，另一个模块用适当的方法显示这些数字。如果我们确定了显示哪些统计数字，即界面，那么系统的两个模块可以独立地进行开发。
- 系统的某些部分可以分别编译；这对于任何系统都是一个优点。
- 通过输入适当的模块，系统组成部分的函数库可以得到重用。

在 Haskell 的定义中，并没有明确模块与文件之间的对应。不过，我们将选择一个文件只包含一个模块。在介绍实例之前，我们先来介绍 Haskell 模块的详细内容。

模块头

每个模块有一个名，如命名为 Ant 的模块形如

```
module Ant where
data Ants  = ...
anteater x = ...
```

注意，所有定义与关键字 module 在同一列开始。最保险的方法是所有定义从文件的最左列开始；对于文学型的脚本，所有定义从一个制表符位置开始。

我们将名为 Ant 的模块所在的文件署名为 Ant.hs 或者 Ant.lhs。

输入一个模块

模块的基本操作是将一个模块输入另一个模块，例如，在定义 Bee 时输入 Ant

```
module Bee where
import Ant
beekeeper = ...
```

在这里，Ant 中所有 **可视** 的定义都可以在 Bee 中使用。一个模块中缺省的可视定义是那些出现在模块中的定义。如果定义

```
module Cow where
import Bee
```

那么 Ants 和 anteater 在 Cow 中是不可视的。要使得它们在 Cow 中可视，或者在 Cow 中明确地输入 Ant，或者使用下面讨论的输出控制方法说明 Bee 输出哪些定义。

主模块

每个模块系统应该包含一个称为 Main 的顶层模块，其中包含一个定义 main。在编译系统中，main 是编译后代码运行时计算的表达式；而在解释系统中，如 Hugs，main 的意义不明显。注意，一个没有命名的模块被系统作为 Main 对待。

输出控制

在介绍 import 时我们讲过，一个模块缺省的输出定义是所有的顶层定义。

- 这样的缺省输出可能输出太多的定义：或许我们不希望输出那些顶层的辅助函数，如下面的 shunt 函数

```
reverse :: [a] -> [a]
reverse = shunt []

shunt :: [a] -> [a] -> [a]
shunt []     ys = ys
shunt (x:xs) ys = shunt xs (x:ys)
```

  因为 shunt 的作用只是定义 reverse。
- 另一方面，缺省的输出也许太少；或许我们希望将输入模块中的某些定义也输出。如上面的例子 Ant，Bee 和 Cow。

我们可以在模块名后列出要输出的定义名以控制输出哪些定义。例如，对于模块 Bee

```
module Bee (beekeeper,Ants(..),  anteater) where
...
```

输出列表列出了已定义的对象名，如 beekeeper，以及数据类型 Ants。其中数据类型名后附有(..)，它表示构造符与类型一起输出；如果省略符号(..)，则这样的类型称为 **抽象数据类型**，我们将在下一章对此做介绍。对于利用 type 定义的类型，符号(..)是不必要的。

以上列表逐个列出输出的对象；我们也可以表达输出一个模块中的所有定义，如

```
module Bee (beekeeper, module Ant) where
...
```

或者

```
module Bee (module Bee, module Ant) where
...
```

其中列表中模块名前加上 module 表示将模块中的所有定义输出。再如

```
module Fish where
```

等价于

```
module (module Fish) where
```

输入控制

我们可以像控制输出那样控制输入的对象，方法是在 import 语句中列出那些要输入的对象、类型或者分类。例如，假如我们不想输入 Ant 的 anteater，则可用下列输入语句表示

```
import Ant (Ants(..))
```

它表示我们只想输入类型 Ants；我们也可以换一种说法，即列出要隐蔽的那些定义

```
import Ant hiding (anteater)
```

假设在我们定义的模块中有一个定义 bear，同时在模块 Ant 中也有一个定义 bear。那么如何来同时使用这两个不同的定义呢？方法是输入模块中的对象使用 **受限的** 名 Ant.bear，bear 留给局部定义的对象。一个受限的名由模块名和对象名中间加一点构成。注意，在两个名和'.'之间无空格，以区别于函数的复合。使用受限名时，应该如下输入模块

```
import qualified Ant
```

在受限输入语句中也可以列出要输入的对象或者隐蔽的对象，方法与不受限的情况完全一样。另外，我们可以给输入的模块一个局部名，如

```
import Insect as Ant
```

即将输入的模块 Ant 局部命名为 Insect。

标准引导模块

标准引导库 Predude.hs 默默地被输入每个模块。我们也可以修改输入语句，如隐蔽某些定义

```
module Eagle where
import Prelude hiding (words)
```

这样我们可以定义自己的 words。如果将 Eagle 输入另一个模块，那么这个模块也必须明确地隐蔽引导库中的 words 以避免定义的冲突，由此可见，重新定义的引导库函数不可能不可视。

如果我们希望使用 words 的原定义，我们可以使用受限的输入

```
import qualified Prelude
```

并使用受限名 Prelude.words。

更多的细节

有关 Haskell 模块系统的进一步资料，参见语言报告 (Peyton Jones and Hughes 1998)；注意有些细节随不同的实现而不同。

习题

15.1 你能使用import取得输出控制的效果吗？你能使用输出控制取得import的效果吗？讨论为什么Haskell包含了import和export两个命令。

15.2 试解释你认为为什么输入的定义没有被输出成为缺省的定义。

15.3 假设给模块的输出控制和输入定义加上一个可选项：如果使用-module Dog，那么模块Dog的所有定义不输出或者不输入。讨论此建议的优点和缺点。如何在目前的模块系统中获得上述效果？

§15.2 模块设计

任何实用的计算机系统在其生命周期内都需要修改，这种修改可能是由实现系统的人来实现的，更有可能是由别人来完成的。为此，所有的系统在设计之初都应该考虑到将来的修改。

我们在讲到系统建档时提到了这一点：说明所有顶层定义的类型，给每个脚本和定义加上说明。另一种有用的描述是将每个定义与其相关的证明相关联；如果我们知道一个函数的逻辑性质，那么对它的目的我们会有更明确的概念。

文档使得一个脚本容易理解，因此也容易修改，但是，如果一组定义可以分解成几个模块或者脚本，每个脚本是整个系统的一个独立组成部分，那么我们可以给这些定义一个**结构**。文件目录应该说明这些部分如何构成系统。如果我们想修改系统的某一部分，我们应该修改一个模块（至少在开始的时候），而不是将整个系统做为一个整体进行修改。

如何把一个系统设计成一组模块呢？下面给出的一些建议将使得修改尽可能简单。

- 每个模块有**明确定义的功能**；
- 每个模块完成一个任务。如果一个模块有两个不同的目的，应该将其分为两个独立的模块。这样可以减小修改一个模块对另一个模块带来的影响。
- 系统的每个部分应该由一个模块完成：每个模块彻底完成一个任务；换言之，一个模块应该自成一体。如果系统的一部分功能由两个模块完成，那么或者应该将两个模块的代码合并，或者应该定义一个新模块，其作用是将两个模块合并为一个模块。
- 每个模块只输出那些必要的定义。这样一来，输入语句的作用会更清楚：输入的恰好是那些需要的函数。在软件工程中这个过程通常被称为**信息隐蔽**，这也是一般程序的设计原则。
- 模块要短小。一个原则是一个模块应该能打印在二至三面打印纸上。

我们还讲过，设计系统时要考虑重用，特别是在使用多态类型和高阶函数的时候。模块将是重用的单位，一个库将通过输入语句被使用。类似的原则也

适用于库的设计。每个库应该有明确定义的目的，例如，一个类型与它的基本操作的实现组织在一个库里。此外，我们还建议

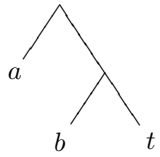
- 在输入一个通用模块时，可以隐蔽那些不使用的定义
- 使用受限的输入可避免命名的冲突：尽管函数名的选择有无穷多种，但实际上我们往往只使用一个很小的子集。

以上的建议可能看似空洞，我们将在随后的实例研究中加以说明。在下一章介绍抽象数据类型时，我们将应用信息隐蔽的思想。以下将研究 Huffman 编码。

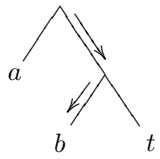
§15.3 编码与译码

人们每天在机器之间传送各种电子信息。这些信息通常是作为二进制序列传输的。为了快速传送，信息需要进行有效地编码。我们将要讨论的是如何对信息编码，即将字符转换为二进制序列，并使得这样的序列尽可能紧凑。

树可用于对信息编码和解码。考虑下图表示的树



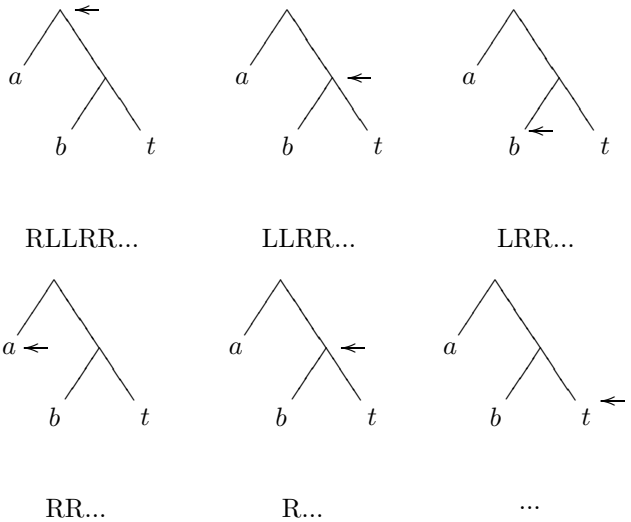
我们可以将此树解释为一种给字母 a、b 和 t 的编码：每个字母的编码是从根到达此字母的**路径**。例如，到达 b 的路径：从根结点往右，然后向左：



由此得到 b 的编码为 RL。类似地，a 的编码为 L，t 的编码为 RR。

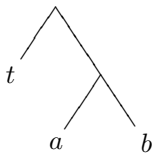
由树得出的编码是 **前缀码**：在这些码中没有一个字母的编码是另一个字母编码的前缀。这是因为从根结点到一个叶的路径不可能是从根结点到另一个叶的路径的前缀。有关 Huffman 编码与算法的详细资料参见 Cormen, Leiserson 和 Rivest (1990)。

一条信息的解码也是通过同一棵编码树完成的。例如，信息 RLLRRRRLRR 的解码过程：由根结点出发，按照信息所表示的路径先向右，再向左，到达叶结点 b，即路径 RL 对应的字母为 b，按同样的方法对剩余的路径 LRRRRLRR 解码，先后得出 a 和 t 等，所以原信息解码后以 bat 为前缀。在下图中我们在树下列出解码剩余的路径。



原信息的最后译码是 battat，编码的长度为 10 位长。注意，不同字母的编码长度不一；a 的码长为 1 位，其他字母的码长为 2 位。对于字母 t 出现较多的信息编码，以上编码树是不是一种好的选择呢？

采用下面的编码树，battat 的编码为 RRRLLLRLL，长度为 9。

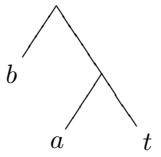


Huffman 编码所采取的策略是，最常出现的字母的编码长度最小，出现较少的字母具有较" 昂贵" 的编码，这是因为它的稀少而决定的。Morse 码是一种常用的 Huffman 编码。

本章的剩余部分将探讨 Huffman 编码的实现，并演示 Haskell 的模块系统。

习题

15.4 使用下列树对信息battat编码后的结果是什么呢？比较你的结果和先前的结果。



15.5 利用第一颗树将RLLRLRLLRR解码。哪颗树能给出最好的编码？试比较三种结果。

```
        Types.lhs
        Huffman编码中使用的类型
模块Types的界面在模块名后显式表示。
>   module Types (Tree(Leaf, Node),
>                 Bit(L,R),
>                 HCode,
>                 Table) where
表示字母相对频率和编码的树
>   data Tree = Leaf Char Int |
>               Node Int Tree Tree
二进制位，Huffman编码和Huffman编码表的类型
>   data Bit = L | R deriving (Eq, Show)
>   type HCode = [Bit]
>   type Table = [(Char, HCode)]
```

图 15.1: 文件 Types.lhs

§15.4 实现一

我们将用一系列 Haskell 模块实现 Huffman 编码和解码。系统的总体结构见本章最后的图 15.4。与往常一样，我们先来定义系统中使用的类型。

类型 –Types.lhs

编码为二进制序列，所以我们定义

```
data Bit = L | R deriving (Eq, Show)
type HCode = [Bit]
```

在编码过程中，我们将 Huffman 树转换为一个表以便于编码

```
type Table = [(Char, HCode)]
```

Huffman 树的叶结点带有字符信息。在构造 Huffman 树时我们还需要每个字母出现的频率；所以，Huffman 树的叶结点和内部点将包含字符和整数信息。

```
data Tree = Leaf Char Int |
            Node Int Tree Tree
```

包含此模块的文件见图 15.1。文件名与其目的写在文件的开始；每个定义之前是有关定义目的的注释。注意，我们在模块名后列出了此模块输出哪些定义。对于数据类型 Tree 和 Bit，它们的构造符也一起输出，我们也可以在这些数据类型名后加上(..)获得同样的效果。这个模块的界面也是可以省略的，但是，我们在这里将其作为模块的界面文档。

编码和解码 –Coding.lhs

本模块使用了 Types.lhs 的类型，故用下列输入语句

```
import Types (Tree(Leaf,Node),Bit(L,R),HCode, Table)
```

我们在此列出了输入的对象名；语句 `import Types` 将有同样的效果，但是失去了文档的意义。

本模块的目的是定义编码函数和解码函数；我们将只输出这两个函数，不输出在定义这两个函数时可能使用的辅助函数，故模块首为

```
module Coding (codeMessage, decodeMessage)
```

在根据一个编码表为一条信息编码时，我们先在表中查找每个字符的码，然后将结果连接在一起。

```
codeMessage :: Table -> [Char] -> HCode
codeMessage tbl = concat . map (lookupTable tbl)
```

有趣的是函数的定义恰好是函数目的注释的实现；部分应用和函数复合的使用使得定义更清晰。

我们先定义 lookapTable，这是在表中查找对应于一个关键字的值的标准函数

```
lookupTable :: Table -> Char -> HCode
lookupTable ((ch, n):tb) c
    | ch==c     = n
    | otherwise = lookupTable tb c
```

根据 module 的说明，此函数没有被输出。

对于一个二进制序列信息 (HCode 的元素的序列) 进行解码，我们使用了树

```
decodeMessage :: Tree -> HCode -> [Char]
```

在 15.3 节我们看到，根据树 tr 解码分两种情况：

- 如果当前处于树的内部结点，则根据码的第一位选择子树；
- 如果当前处于叶结点，则读出相应的字符，然后从树 tr 的根结点开始对剩余的码进行解码。

当所有的码都解完时，便得到解码信息

```
decodeMessage tr
  = decodeByt tr
    where
    decodeByt (Node n t1 t2) (L:rest)
      = decodeByt t1 rest
    decodeByt (Node n t1 t2) (R:rest)
      = decodeByt t2 rest
    decodeByt (Leaf c n) rest
      = c : decodeByt tr rest
    decodeByt t [] = []
```

其中局部函数称为 decodeByt，因为它” 根据 tr” 来解码。

15.3 节的第一棵编码树和信息可表示如下

```
exam1 = Node 0 (Leaf 'a' 0)
               (Node 0 (Leaf 'b' 0) (Leaf 't' 0))
```

此信息的解码过程如下

```
decodeMessage exam1 mess1
⇝ decodeByt exam1 mess1
⇝ decodeByt exam1 [R,L,L,R,R,R,R,L,R,R]
⇝ decodeByt (Node 0 (Leaf 'b' 0) (Leaf 't' 0))
                                  [L,L,R,R,R,R,L,R,R]
⇝ decodeByt (Leaf 'b' 0) [L,R,R,R,R,L,R,R]
⇝ 'b' :  decodeByt exam1 [L,R,R,R,R,L,R,R]
⇝ 'b' :  decodeByt (Leaf 'a' 0) [R,R,R,R,L,R,R]
⇝ 'b' :  'a':  decodeByt exam1 [R,R,R,R,L,R,R]
```

在继续实现之前，我们先来看一下如何构造 Huffman 编码树。

习题

15.6 完成上述计算`decodeMessage exam1 mess1`。

15.7 设有下表

```
table1 = [('a',[L]),('b',[R,L]),('t',[R,R])]
```

试计算`codeMessage table1 "battab"`。

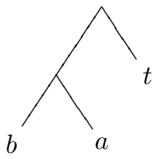
§15.5 构造 Huffman 树

给定一段文本，如"battat"，如何求它的最优编码树呢？我们按照 Cormen, Leiserson 和 Rivest (1990) 17.3 节介绍它的构造过程。

- 首先计算每个字母出现的频率，如
  `[('b',1),('a',2),('t',3)]`
- 算法的主要思想是找出频率最小的两个字母并将其作为一个单字符（或者树）。重复这个过程直至最后成为一棵树。下列步骤给出其构造细节。
- 将`[('b',1),('a',2),('t',3)]` 中的每个元素转换为一棵树，由此得出
  `[Leaf 'b' 1, Leaf 'a' 2, Leaf 't' 3]`
  这些树按其频率排序
- 然后开始合并树：将两棵最小频率的树合在一起，并按照频率从小到大的顺序将其插入适当位置。
  `[Node 3 (Leaf 'b' 1) (Leaf 'a' 2), Leaf 't' 3]`
- 重复上述过程，直至剩下一棵树。例如
  `Node 6 (Node 3 (Leaf 'b' 1)(Leaf 'a' 2)) (Leaf 't' 3)`
  其图形如下

```
    Frequency.lhs
>   module Frequency (frequency) -- [Char] -> [(Char, Int)]
    MakeTree.lhs
>   module MakeTree (makeTree) -- [(Char, Int)] -> Tree
>   import Types
    CodeTable.lhs
>   module CodeTable (codeTable)  -- Tree -> Table
>   import Types
```

图 15.2: 构造 Huffman 的模块



- 将此树转换为一个 Table
  `[('b', [L,L]), ('a',[L,R]),('t',[R])]`
  下面讨论如何在 Haskell 中实现这个算法。

§15.6 设计

系统的实现需要我们设计完成上一节所述各阶段的模块。我们首先确定需要哪些模块，以及它们实现哪些函数。这在很大程度上相当于**分治法**：我们将问题分解为易于处理的几部分，并分别解决，然后使用 import 和 module 将它们组织在一起。在实现这些函数之前，我们先设计这些模块的界面。

算法的三个阶段（见图 15.2）分别构成三个模块。我们给输出的对象加上它们的类型作为注释，这样的模块首部包含了所输出函数的足够信息；尽管用户不知道函数的定义，但知道如何使用这些函数。

因为函数 frequency 和 makeTree 不会分别使用，所以当三个文件合在一起时，我们将这两个函数合成在模块 MakeCode.1hs 中。详见图 15.3。

我们的下一个任务是分别实现每个模块。

§15.7 实现二

本节分别讨论三个模块的实现。

```
    Makecode.lhs
    Huffman coding in Haskell
>   module MakeCode (code, codeTable) where
>   import Types
>   import Frequency (frequency)
>   import MakeTree (makeTree)
>   import CodeTable (codeTable)
将计算频率和转换为树合在一起
>   codes :: [Char] -> Tree
>   codes = makeTree . frequency
```

图 15.3: 模块 MakeCode.lhs

统计字符 –Frequency.lhs

函数 frequency 的作用是将一段文本转换为字母和它们在文本中出现的频率的列表, 并按频率由小到大排列。如输入文本"battat", 输出`[('b',1),('a',2),('t',3)]`。我们通过三步实现这个目标。

- 首先给文本中每个字母配上计数 1,
  `[('b',1),('a',1),('t',1),('t',1),('a',1),('t',1)]`
- 然后按字母顺序对列表排序, 将相同字母的计数加到一起
  `[('a',2),('b',1),('t',3)]`
- 最后按频率对列表排序。

上面用到了两种排序, 一种在字母上排序, 另一种在频率上排序。可否定义一个排序函数使其可用于上述两种排序呢? 我们可以定义一个通用的排序函数, 其方法是将前后两部分排好序的列表归并为一个有序列表。

```
mergeSort :: ([a] -> [a] -> [a]) -> [a] -> [a]
mergeSort merge xs
   | length xs < 2     = xs
   | otherwise
      = merge (mergeSort merge first)
              (mergeSort merge second)
        where
        first  = take half xs
        second = drop half xs
        half   = (length xs) `div` 2
```

函数 mergeSort 的第一个参数是归并函数, 它将两个有序列表归并为一个有序列表。正是通过将归并函数作为参数, mergeSort 成为一个可重用的排序函数。

在字符上排序时，我们将相同字符的项合并

```
alphaMerge :: [(Char, Int)] -> [(Char, Int)] -> [(Char, Int)]
alphaMerge xs [] = xs
alphaMerge [] ys = ys
alphaMerge ((p,n):xs) ((q,m):ys)
  | (p==q)     = (p,n+m):alphaMerge xs ys
  | (p<q))     = (p,n):alphaMerge xs ((q,m):ys)
  | otherwisee = (q,m):alphaMerge ((p,n):xs) ys
```

在频率上排序时，我们比较频率，当两个二元组有相同的频率时，按照字母顺序归并。

```
freqMerge xs [] = xs
freqMerge [] ys = ys
freqMerge ((p,n):xs) ((q,m):ys)
  | (n<m || (n==m && p<q))
     = (p,n) : freqMerge xs ((q,m):ys)
  | otherwise
     = (q,m) : freqMerge ((p,n):xs) ys
```

现在可以写出顶层函数 frequency 的定义

```
frequency :: [Char] -> [(Char, Int)]
frequency
  = mergeSort freqMerge . mergeSort alphaMerge . map start
    where
    start  ch = (ch,1)
```

可以看出，这是前面非形式描述的三个阶段的直接结合。注意，在这个模块中定义的函数中，只有 frequency 被输出。

构造 Huffman 树 –MakeTree.lhs

根据字符与频率列表构造 Huffman 树分两步完成。

```
makeTree :: [(Char, Int)] -> Tree
makeTree = makeCodes . toTreeList
```

其中

```
toTreeList :: [(Char,Int)] -> [Tree]
makeCodes :: [Tree] -> Tree
```

函数 toTreeList 将每个字符数字二元组转换为一棵树，故

```
toTreeList = map (uncurry Leaf)
```

注意，在这里我们利用引导库函数 uncurry 得到构造符函数 Leaf 的非 Curry 形式。

函数 makeCodes 将两棵相邻的树合并为一棵树

```
makeCodes [t] = t
makeCodes ts = makeCodes (amalgamate ts)
```

如何合并树呢？我们必须将列表中的前两棵树合并（因为列表中的树按频率有序），然后将结果插入到列表中以保持列表的有序性。采用由顶向下的方法，我们定义

```
amalgamate :: [Tree] -> [Tree]
amalgamate (t1:t2:ts) = insTree (pair t1 t2) ts
```

当合并两棵树时，我们需要合并它们的频率

```
pair :: Tree -> Tree -> Tree
pair t1 t2 = Node (v1+v2) t1 t2
             where
             v1 = value t1
             v2 = value t2
```

其中 value 定义如下

```
value :: Tree -> Int
value (Leaf _ n)    = n
value (Node n _ _ ) = n
```

函数 insTree 的定义类似于插入排序中的插入，留给读者作为练习。此外，输出函数使用了各种在" 外部不可视" 的各种定义。

编码表 –CodeTable.lhs

下面定义函数 CodeTable，它将一个 Huffman 树转换为一个编码表。在转换树(`Node n t1 t2`)时，我们必须转换`t1`，并在码的前面添加 L；转换`t2`，并在这之前添加 R。为此，我们定义下列更通用的转换函数

```
convert :: HCode -> Tree -> Table
```

其中第一个参数是树中当前的路径，函数的定义为

```
convert cd (Leaf c n)
    = [(c,cd)]
convert cd (Node n t1 t2)
    = (convert (cd++[L]) t1) ++ (convert (cd++[R]) t2)
```

函数 CodeTable 由上述转换函数定义，初始码为空串

```
condeTable :: Tree -> Table
codeTable = convert []
```

例如下面的计算

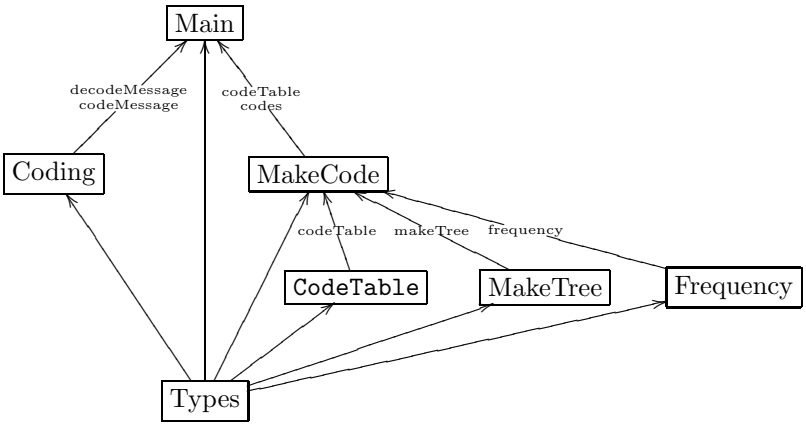


图 15.4: Huffman 编码模块系统

```
codeTable (Node 6 (Node 3 (Leaf 'b' 1) (Leaf 'a' 2)))
                      (Leaf 't' 3)
⇝ convert [] (Node 6 (Node 3 (Leaf 'b' 1) (Leaf 'a' 2)))
                      (Leaf 't' 3)
⇝ convert [L] (Node 3 (Leaf 'b' 1) (Leaf 'a' 2)) ++
    convert [R] (Leaf 't' 3)
⇝ convert [L,L] (Leaf 'b' 1) ++
    convert [L,R] (Leaf 'a' 2) ++
    [('t',[R])]
⇝ [('b',[L,L]), ('a',[L,R]), ('t',[R])]
```

顶层文件 –Main.lhs

我们可以将系统的各部分组合在一起形成顶层文件。

```
module Main (main) where

import Types (tree(Leaf,Node), Bit(L,R), HCode, Table)
import Coding (codeMessage, decodeMessage)
import MakeCode (codes, codeTable)
```

系统的整体结构见图 15.4。模块用矩形表示，A 至 B 的箭头表示 A.lhs 输入 B.lhs。箭头上的标识表示输入模块输出的函数，例如，codes 和 codeTable 由模块 MakeCode.lhs 输出到 Main.1hs。

如果这个编码系统用作一个更大系统的一个子系统，可以将模块重命名，并在 import 命令中说明 4 个函数和类型中哪些由模块输出。特别要注意，要使用这些函数，类型必须被输出。

习题

15.8 给出使用预定义顺序<=的归并排序定义，并说明其类型。

15.9 修改上题定义，使得排序删除重复元素。

15.10 定义一个使用比较函数作为参数的归并排序定义

```
ordering :: a -> a -> Ordering
```

解释如何用此定义实现mergeSort freqMerge，并讨论为何不能使用此定义实现mergeSort alphaMerge。

15.11 试定义函数insTree。

15.12 试计算

```
makeTree [('b',2),('a',2),('t',3),('e',4)]
```

15.13 定义下列函数

```
showTree  :: Tree -> String
showTable :: Table -> String
```

它们分别给出可打印的Huffman树和编码表。一种通用的打印树的方法是标识树的结构。例如，

```
      left sub tree, 空 4 个字符
  value(s) at Node
      right sub Tree, 空 4 个字符
```

小结

在编写任何规模的程序时，我们需要将任务分解。Haskell 模块系统允许一个脚本包含在另一个脚本中。在模块的界面，我们可以控制模块输出哪些定义以及输入哪些定义。

我们提出了系统模块化设计的几个原则。最重要的是，一个模块完成一个明确定义的任务，并且只输出必须的信息，这是信息隐蔽的原则。下一章介绍抽象数据类型时我们将进一步探讨这个原则。

本章提出的设计原则在 Huffman 编码系统中得以实践。特别地，我们提出对于文件 MakeCode.1hs 及其三个子模块，模块的设计可以从它们的界面出发，即模块输出的定义（以及类型）。所以，设计过程发生在所有实现之前。

第十六章　抽象数据类型

Haskell 模块系统允许被调入文件中的函数和对象隐蔽起来。这些隐蔽函数只用于定义输出函数，将其隐蔽可使得两个文件的接口更清晰：只有那些需要的函数是外部可见的。

本章将说明，信息隐蔽同样适用于类型，由此可以定义抽象数据类型，或者 ADT。我们将解释抽象数据类型机制，并介绍一些 ADT 例子，包括队列，集合，关系和仿真实例中用到的一些基本类型。

§16.1　类型表示

我们先讨论抽象数据类型机制的目的和运作。

假定我们要构造一个数值计算器。表达式的类型如 14.2 节的类型 Expr，但是，表达式中包含了变量。计算器的功能包括用变量构造表达式，设置变量的值等。

系统的一个组成部分将说明变量的当前值，称之为计算器的 *存储器*，如何来表达这样的 *存储器*呢？自然的选择包括

- 整数和变量二元组的列表：`[(Int, Var)]`
- 变量到整数的函数：`Var -> Int`

这两个模型均允许我们查看变量的值，以及为 *存储器*设置初值。这些运算具有下列类型

```
initial :: Store
value   :: Store -> Var -> Int                          (StoreSig)
update  :: Store -> Var -> Int -> Store
```

但是，每个 *存储器*模型允许更多的运算，例如，将一个列表置逆，将两个函数复合等等。尽管类型 Store 的目的只限于上述三种运算，但是，Store 模型可能以我们不希望的方式使用。

如何为 *存储器*建立更好的模型呢?答案是定义一个只有 initial,value 和`update`三个运算的类型，以避免类型的滥用。为此，我们隐蔽类型的实现信息，只允许用户使用定义 (StoreSig) 中出现的运算来处理类型的对象。

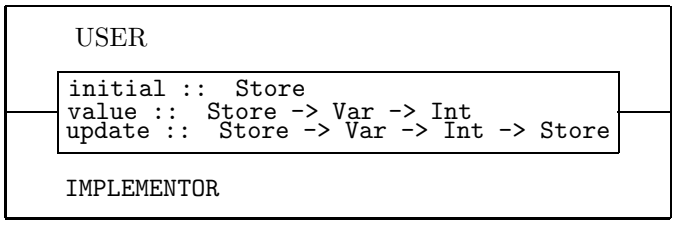


图 16.1:

当我们通过一组运算给一个类型提供了一个接口时，称这样的类型为**抽象数据类型** 或者 ADT, 因为“具体”的类型本身不可存取，我们只能通过这些运算存取类型，所以这些运算为类型提供了一种抽象。图 16.1 是这种抽象的图示，它表明，这种抽象不仅是 存储器类型的自然表示，而且具有下列优点

- 类型说明 (StoreSig) 构成类型的用户和类型的实现人员之间的一个清晰定义结口，称之为 ADT 的 Signature。用户和实现人员只需在接口上取得一致便可独立进行工作，因此，这是将一个复杂问题分解成简单问题的另一种方式，也是分而治之方法的一个方面。
- 我们可以**修改**Store 的实现，而对用户不产生任何影响。与此相对的是实现对于用户是可见的，特别地，如果实现使用了代数类型，如果对类型进行了任何修改，那么使用模式匹配定义的函数都必须修改。这些需修改的函数不仅包括 Signature 中的函数，而且包括用户使用模式匹配定义的函数。

我们将在本章看到这些优点，下面首先介绍 Haskell 抽象数据类型机制。

§16.2 Haskell 抽象数据类型机制

在第十五章介绍 Haskell 模块系统时，我们看到两种输出一个数据类型的方法，不妨称此类型为 Data。如果将 Data(..) 包含在模块的输出列表中，则类型的构造符同类型一起输出; 如果将 Data 放在输出列表中，则类型的构造符没有输出，我们只能利用 Signature 的运算对类型进行操作.

对于类型 Store，模型首如下

```
module Store (Store, initial, value, update) where
```

它表示，我们只能通过这里提供的 3 个函数存取类型 Store。本书沿用在模块首用注释形式说明输出函数的类型的习惯。例如，给模块 Store 添加类型注释后为

```
module Store
  (Store,
  initial,    -- Store
  value,      -- Store -> Var -> Int
  update      -- Store -> Var -> Int -> Store
  ) where
```

接下来，模块必须提供类型 Store 的定义以及三个函数的定义。

如果所实现的类型是一个数据类型 (data)，那么抽象数据类型的实现也就完成了。但是，我们的例子使用了二元组的列表`[(Int, Var)]`来刻画类型 Store，所以我们必须定义一个新的数据类型，称之为 Store

```
data Store = Sto [(Int, Var)]
```

类型 Store 有一个构造符 Sto，它是将一个列表转换为 Store 元素的函数，也可将其视为将一个列表“打包”，使之成为 Store 的元素。

现在我们需要定义 Store 类型上的函数 initial，value 和 update。一种方法是先定义`[(Int Var)]`上的类似函数，然后将其改造成 Store 上的函数

```
init :: [(Int, Var)]
init = []

val :: [(Int, Var)] -> Var -> Int
val [] v           = 0
val ((n,w):sto) v
  | v==w           = n
  | otherwise      = val sto v

upd :: [(Int, Var)] -> Var -> Int -> [(Int, Var)]
upd sto v n = (n,v):sto
```

初始 存储器init 由空列表表示；v 的值通过在列表中寻找到第一个二元组`(n, v)`求得，存储器的修改通过在列表首添加一个新二元组`(Int Var)`完成。这些函数需要转换到类型Store 上，使得参数和结果形如`Sto xs`，其中`xs :: [(Int Var)]`。故定义成为

```
initial :: Store
initial = Sto []

value :: Store -> Var -> Int
value (Sto []) v           = 0
value (Sto ((n,w):sto)) v
  | v==w           = n
  | otherwise      = val (Sto sto) v

update :: Store -> Var -> Int -> Store
update (Sto sto) v n = Sto ((n,v):sto)
```

我们看到，定义的模式是相似的，只是需要把左边的参数和右边的结果用 Sto“包起来”。我们将在 16.8 节介绍抽象数据类型 Set 时看到“打包”函数的一般机制。

假如我们打破抽象壁垒，认为 Store 的元素形如`Sto xs`，结果会怎么样呢？如果在输入 Store 的模块中输入

```
initial == Sto []
```

我们将得到类型错误信息

```
ERROR: Undefined constructor function "Sto"
```

虽然 initial 是用`Sto []`实现的，但是在模块 Store 之外是不可见的。

newtype 结构

在定义 Store 时，我们也可以使用如下定义

```
newtype Store = Sto [(Int, Var)]
```

它的作用等同于只有一个构造符的 data 类型，但是它的实现更高效。

我们也可以如下定义 Store

```
newtype Store = Store [(Int, Var)]
```

即使用与类型名同名的构造符名。不幸的是，在本书写作的时候，Hugs 系统会使得这样的构造符可见，尽管模块首说明构造符是不可见的。用 data 定义的类型，当构造符名同类型名相同时，会出现同样的现象。

类型分类：显示值和相等

对于抽象数据类型，我们也可以说明它们属于特定的类型分类，如 Show 和 Eq

```
instance Eq Store where
  (Sto sto1) == (Sto sto2) = (sto1 == sto2)


instance Show Store where
   show (Sto sto)  = show sto
```

但是，要注意，一旦有这样的说明，这些特例便不能隐蔽；如果一个函数使用了这些特例说明中的函数，那么即使这个函数未列入模块的输出列表，它仍然在调用 Store 的模块中可见。当然，我们可以选择不定义这些特例，因此不为 Store 提供相等运算和 Show 函数

用函数描述 Store

Store 的另一种实现方法是使用变量到整数的函数类型

```
newtype Store = Sto (Var -> Int)

initial :: Store
initial = Sto (\v -> 0)

value :: Store -> Var -> Int
value (Sto sto) v = sto v

update :: Store -> Var -> Int -> Store
update (Sto sto) v n = Sto (\w -> if v==w then n else sto w)
```

在这种实现中

- initial 将每个变量映射到 0

- 要查找一个变量 v 的值，可以将函数 sto 作用于 v;
- 对于 uptate，函数在不做修改的变量上的返回值与 sto 相同。

习题

16.1 试用按变量名有序的列表实现Store。这种实现相对于原列表实现有何优点或者缺点。

16.2 对于Store的列表类型实现，试定义 存储器上的相等函数：如果两个 存储器在任意一个变量下的值均相同，则定义这两个 存储器相等。你能在Store的第二个实现上定义这样的相等函数吗？如果不能，试修改Store的实现，然后定义相等函数。

16.3 假设一个变量在 存储器中不存在值，则查询该变量的结果是一个错误。解释如何修改Store的signature和两个实现。你需要使用类型Maybe a.

16.4 查询在 存储器中没有值的变量时，也可以不返回错误，而是采取扩充signature，提供一个测试一个变量在某个 存储器中是否有值的函数。试解释如何对两个实现做这样的修改。

16.5 假设要实现Store上第四个运算：

```
setAll :: Int -> Store
```

使得 存储器setAll n 的所有变量值均为n。你能对例子中两个实现均实现这个运算吗？如果可以，试给出实现，如果不可以，试说明为什么。

16.6 为第五章的图书馆数据库设计一个ADT。

§16.3 队列

一个队列是一个“先进先出”结构，如果 Flo 和 Eddie 先后加入一空队列，那么第一个出队的是 Flo。作为一个抽象数据类型，我们希望不仅能够在队列上添加元素和删除元素，而且有空队列

```
module Queue
   ( Queue,
     emptyQ,   -- Queue a
     isEmptyQ, -- Queue a -> Bool
     addQ,     -- a -> Queue a -> Queue a
     remQ,     -- Queue a -> (a, Queue a)
   ) where
```

函数 remQ 返回一个二元组: 被删除的元素以及队列的剩余部分，条件是原队列不空。否则，返回错误信息。

列表可用于描述队列：添加在列表尾进行，删除在列表首进行。

```
newtype Queue a = Qu [a]


emptyQ = Qu []
```

```
isEmptyQ (Qu []) = True
isEmptyQ _       = False
addQ x (Qu xs)  = Qu (xs++[x])
remQ q@(Qu xs)
  | not (isEmptyQ q)   = (head xs, Qu (tail xs))
  | otherwise          = error "remQ"
```

定义 remQ 使用了一种新的模式匹配。我们使用了模式`q@(Qu xs)`来匹配输入，其中 `@` 读作" 为"。变量 q 匹配整个输入，而且当输入同时匹配`Qu xs`时，xs 能够给出构成队列的列表。使用这样的匹配可以直接存取整个输入以及输入的组成部分。如果不使用以上模式，定义将是

```
remQ (Qu xs)
 | not (isEmptyQ (Qu xs)) = (head xs, Qu (tail xs))
 | otherwise        = error "remQ"
```

在此，我们要在 xs 上重新构造原来的队列。

另一种实现方法是将元素加到列表首，而不是列表尾；函数 emptyQ 和 isEmyptyQ 的定义不需修改，其他函数定义如下

```
addQ x (Qu xs) = Qu (x:xs)


remQ q@(Qu xs)
  | not (isEmptyQ q) = (last xs, Qu (init xs)
  | otherwise        = error "remQ"
```

其中预定义函数 last 和 init 分别返回列表的最后一个元素和除最后一个元素的剩余列表。

尽管我们还没有介绍如何估算计算的代价（第十九章内容），但是，可以看出，在每一个实现中，有些运算“便宜”，而另一些运算“昂贵”。“便宜” 的函数可以在一步完成计算，如第一个实现中的 remQ 和第二个实现中的 addQ；但是，” 昂贵” 函数必须扫描整个列表才能完成计算，所以其花费与列表长度成正比。

是否有办法使两个运算均” 便宜” 呢? 方法是将队列分成两个列表，使得添加和删除均在列表的头上进行。图 16.2 表示这样的一个队列运算过程，初始队列包含元素 7、5、2 和 3。随后是删除元素，添加元素 0，然后再删除两个元素。在每种情况下，队列由两个列表表示，左边的列表向左增长，右边的列表向右增长。

函数 remQ 从左边的列表首删除元素，函数 addQ 在右边列表首添加元素。当左边列表为空时，右边列表的元素必须转移到左边列表。转移元素代价较大，因为我们必须将整个列表倒置，不过，这样的转移并不是在每次删除时发生。下面是 Haskell 实现

```
data Queue a = Qu [a] [a]
```

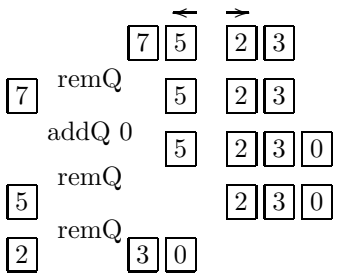


图 16.2:

```
emptyQ = Qu [] []

isEmptyQ (Qu [] []) = True
isEmptyQ _          = False

addQ x (Qu xs ys) = Qu xs (x:ys)

remQ (Qu (x:xs) ys) = (x, Qu xs ys)
remQ (Qu [] (y:ys)) = remQ (Qu (reverse (y:ys)) [])
remQ (Qu [] [])     = error "remQ"
```

正如我们对类型 Store 所做的评论一样, 从抽象数据类型观点来看, 以上实现的操作与前两种实现的操作没有区别。另一方面, 如上所述, 双列表实现较单列表实现效率高, 有关数据结构有效实现的近期工作, 请参见 Okasaki(1998)。

习题

16.7 计算下列式子

```
"abcde" ++ "f"
init "abcdef"
last "abcdef"
```

其中

```
init x = take (length x-1) x
last x = x !! (length x -1)
```

16.8 试解释 *3* 队列模型在下列队列运算序列下的变化情况：`add 2, add 1, remove item, add 3, remove item, add 1, add 4, remove item, remove item.`

16.9 一个双端队列 (deque) 允许在两端添加元素或者删除元素。试给出抽象数据类型`Deque a`的signature和两个不同的实现。

16.10 在一个唯一队列中一个元素只出现一次 (最先到达者)。给出这种队列的抽象数据类型定义和实现。

16.11 在优先队列中每个元素有一个数值优先权。在删除元素时，优先权最大的元素出队。如果这样的元素有多个，则最先到达的元素出队。给出优先队列的抽象数据类型定义与实现。

16.12 (较难) 解释如何用优先队列实现第十五章的Huffman编码系统。

§16.4 设计

本节讨论 Haskell 抽象数据类型的设计以及抽象数据类型机制在程序设计中的作用。

一般原则

在构造一个系统时，类型的选择是很基本的，而且对随后的设计与实现产生很大的影响。在早期使用抽象数据类型我们便有希望找到问题中出现的类型的“自然”表示。设计抽象数据类型分三个阶段

- 首先确认并**命名**系统中的类型;
- 其次对于每个类型给出**非正式描述**;
- 在上述描述的基础上, 写出每个抽象数据类型的**Signature**。

如何确定 signature 包含哪些运算呢？当然这要依具体情况而定。但是，对抽象数据类型的运算，我们可以考虑一些一般的问题。

- 我们能构造类型的对象吗？例如，在类型 Queue a 中，我们可以创建对象 emptyQ；在类型 Set 中，我们可以构造一个函数，它将用一个元素构造包含此元素的单元素集合。如果不存在这样的对象或者函数，这样的类型定义必有问题。
- 我们能检查对象的种类吗？例如, 在树 ADT 中我们可能希望检则一个对象是树叶还是含根的树。
- 如果需要，我们能抽取对象的组成部分吗？例如，我们能提取队列 Queue a 的头吗？
- 我们能对象做变换吗？例如，我们能将一个列表置逆吗？或者能否在队列上添加一个元素。
- 我们能否结合对象？例如，我们可能希望将两棵树合并为一棵树。
- 我们能否将对象拆分？例如，可否求一个数值列表的大小和或者求一个对象的大小。

并非所有这些问题适用于每种情况，但是，类型上的大部分运算属于其中的某一个范畴。例如，下列二叉树 Signature 的所有运算属于如上所列范畴：

```
module Tree
  (Tree,
   nil,        -- Tree a
   isNil,      -- Tree a -> Bool
   isNode,     -- Tree a -> Bool
```

```
    leftSub,    -- Tree a -> Tree a
    rightSub,   -- Tree a -> Tree a
    treeVal,    -- Tree a -> a
    insTree,    -- Ord a => a -> Tree a -> Tree a
    delete,     -- Ord a => a -> Tree a -> Tree a
    minTree,    -- Ord a => Tree a -> Maybe a
  ) where
```

其中的 Signature 可以包含其他函数，例如，我们可以添加一个函数 size，此函数可以由其他函数定义

```
size :: Tree a -> Int
size t
  | isNil t    = 0
  | otherwise  = 1 + size (leftSub t) + size (rightSub t)
```

函数 size 的定义独立于类型实现，因此，如果 Tree a 的实现发生变化，size 的实现不需修改，这是不将 size 写入 Signature 的很好理由，这也是检测 Signature 的方法：Signature 中的每个运算都是必要的吗？我们将在今后继续讨论这一点和树类型。下节是一个较大的实例。

习题

16.13 类型Tree a的signature中所有运算都是必需的吗？那些运算可以由其他运算实现？

16.14 为第五章图书馆数据库的抽象类型设计一个signature.

16.15 为 *10.8* 节的索引抽象类型设计一个signature.

§16.5 仿真

我们在 4.5 节介绍了一个仿真例子，并定义了代数类型 Inmess 和 Qutmess。为了方便解释，假设系统时间用分钟表示。

对象 Inmess No 表示无客户到达，Yes 34 12 表示客户在第 34 分钟到达，需要 12 分钟服务时间。元素`Outmess Discharge 34 27 12`表示一个客户在第 34 分钟到达，在得到 12 分钟服务之前等待了 27 分钟。

本节的目的是设计一个简单排队仿真的 ADT。首先考虑一个队列的情况。我们将称之为 QueueState, 其描述如下：

队列上有两上主要运算，第一个是在队列上添加一个元素，即添加一个 Inmess。第二个是用一分钟时间处理队列；其作用是给队列的头元素（如果存在）一分钟服务时间，结果有两种可能，头元素服务结束，此时一个 Qutemss 被生成，或者头元素需要进一步的处理。此外，我们需要一个空队列，队列长

度的信息以及判断队列是否空的测试。 由以上描述可得到下列 Signatue 说明

```
module QueueState
  ( QueueState,
    addMessage,       -- Inmess -> QueueState -> QueueState
    queueStep,        -- QueueState -> (QueueState, [Outmess])
    queueStart,       -- QueueState
    queueLength,      -- QueueState -> Int
    queueEmpty,       -- QueueState -> Bool
  ) where
```

函数 queueStep 返回一个二元组：经过一步处理的队列和一个 Outmess 列表。这里使用了 Outmess 的列表，而没有使用 Outmess，这样在没有输出的情况下可以返回空列表。

类型 QueueState 允许我们描述所有客户由一个处理器（银行职员）提供服务的情形。如何描述不只一个队列的情况呢？我们将称之为一个服务器 (server)，并用 ADT ServerState 描述。

一个服务器由一些队列构成，这些队列可以用整数 0、1 等表示。假定系统每分钟接收到一个 Inmess，即每分钟最多有一个客户到达。服务器上有三个主要操作。第一，我们能够给某个队列添加一个 Inmess。第二，服务器的一步处理由每个队列的一步处理构成，并生成一个 Outmess 列表，因为每一个队列会生成一个 Outmess。最后，一步仿真由一个服务器步和将 Inmess 分配到服务器中最短队列中的操作构成。其它三个必要的操作包括：我们需要一个由一些空队列构成的初始服务器；我们需要知道服务器中队列的个数以及服务器中的最短队列。 由上述描述，我们可以得到下列 Signature

```
module ServerState
  ( ServerState,
    addToQueue,      -- Int -> Inmess -> ServerState -> ServerState
    serverStep,      -- ServerState -> (ServerState, [Outmess])
    simulationStep,  -- ServerState -> Inmess -> (ServerState, [Outmess])
    serverStart,     -- ServerState
    serverSize,      -- ServerState -> Int
    shortestQueue,   -- ServerState -> Int
  ) where
```

下一节我们讨论如何实现这两个 ADT。需要注意的是，这些 ADT 的用户可以开始程序设计，因为用户所需要的信息均已包含在抽象数据类型的 Signature 中。

习题

16.16 在抽象数据类型QueueState和ServerState中是否存在冗余运算。

16.17 设计一个round-robin仿真，其中第一个元素入队列 *0*，第二个元素入队列 *1*，如此下去，待第 m 个元素入最后一个队列后，下一个元素从队列 *0* 开始入队。

§16.6 实现仿真

本节给出一个队列 ADT 和一个服务器 ADT 的实现，其中 ServerState 的实现将建立在 QueueState 的基础上，这也表明两上实现是相互独立的，即修改 QueueState 的实现并不影响 ServerState 的实现。

队列

我们在前一节设计了 ADT 的界面，如何继续进行 ADT 的实现呢？首先我们应该检查类型 QueueState 的描述。这种描述意味着类型包含什么样的信息呢？

- 我们需要处理 Inmess 构成的队列。这个队列可用一个列表表示，队列的头元素为当前处理的对象。
- 我们需要记录系统在某个时刻处理头元素所花的时间；
- 在 Outemss 中我们需要给出正在处理的某个对象的等待时间。已知对象的到达时间和需要处理的时间，如果当前时间已知，则可以计算出等待时间。

为此，我们定义

```
data QueueState = QS Time Service [Inmess]
                  deriiving (Eq, Show)
```

其中第一个域表示当前时间，第二个域表示当前对象得到的服务时间，第三个是队列本身。下面考虑各个运算。添加信息：将添加的信息置于信息列表尾部

```
addMessage :: Inmess -> QueueState -> QueueState

addMessage im (QS time serv ml) = QS time serv (ml++[im])
```

最复杂的定义是 queueStep。如前所述，在处理一个对象时，有两种主要情况

```
queueStep :: QueueState -> (QueueState, [Outmess])
queueStep (QS time servSoFar (Yes arr serv : inRest))
  | servSoFar < serv
       = (QS (time+1) (servSoFar +1) (Yes arr serv : inRest), [])
  | otherwise
       = (QS (time+1) 0 inRest, [Discharge arr (time-serv-arr) serv])
```

第一种情况，当所得到的服务时间（servSoFar）小于要求的时间（serv）时，服务还未完成。为此，时间加 1，服务加 1，不产生输出信息。

另一种情况是服务结束，队列的新状态是`QS (time+1) 0 inRest`。在这种状态，时间加一个单位，处理时间置 0，并且列表的头元素被删除。同时，输出信息被生成，其中等待时间为当前时间减去到达时间和服务时间。

如果没有对象可以处理，则只需将当前时间加 1，不生成任何输出。

```
queueStep (QS time serv []) = (QS (time+1) serv [], [])
```

注意，输入信息为 No 的情况在这里没有考虑，因为这样的信息被服务器过滤掉了，我们将在后面考虑这种情形。

其他三个函数定义如下

```
queueStart :: QueueState
queueStart = QS 0 0 []

queueLength :: QueueState -> Int
queueLenght (QS _ _ q) = length q

queueEmpty :: QueueState -> Bool
queueEmpty (QS _  _q) = (q == [])
```

至此我们完成了队列的实现。

显然，有多种不同的实现可供选择。我们可以选择将处理的元素与队列分离开来，或者使用一个 ADT 作为队列，而非使用一个“具体”的列表。

服务器

服务器由一组队列构成，并由整数 0 开始命名；我们将使用一个队列的列表表示

```
newtype ServerState = SS [QueueState]
                      deriving (Eq, Show)
```

注意，上述 ADT 的实现建立在另一个 ADT 上，这是 ADT 实现常见的方法。下面讨论各个函数的实现。

在队列上添加一个元素使用抽象类型 QueueState 的函数 addMessage。

```
addToQueue :: Int -> Inmess -> ServerState -> ServerState

addToQueue n im (SS st)
  = SS (take n st++[newQueueState]++drop (n+1) st)
    where
    newQueueState = addMessage im (st!!n)
```

服务器的一步由其中各队列的一步构成，并将各队列产生的输出信息合并在一起。

```
serverStep :: ServerState -> (ServerState , [Outmess])
serverStep (SS [])
  = (SS (q':qs'), mess++messes)
    where
    (q',mess)         = queueStep q
    (SS qs', messes)  = serverStep (SS qs)
```

在进行一步仿真时，我们令服务器进行一步，然后将表示到达的输入信息加到最短队列上。

```
simulationStep
   :: ServerState -> Inmess -> (ServerState, [Outmess])
simulationStep servSt im
  = (addNewObject im servSt1, outmess)
  where
  (servSt1, outmess) = serverStep servSt
```

将信息加到最短队列的操作由函数 addNewObject 完成，此函数不含在 Signature 中，其原因是它可以用操作 addToQueue 和 shortestQueue 来定义

```
addNewObject :: Inmess -> ServerState -> ServerState
addNewObject No servSt = servSt
addNewObject (Yes arr wait) servSt
  = addToQueue (shortestQueue servSt) (Yes arr wait) servSt
```

如前所述，正是在这个函数中输入信息 No 没有传到队列上。

Signature 中的其他三个函数是标准的

```
serverStart :: ServerState
serverStart = SS (replicate numQueues queueStart)
```

其中 numQueues 是待定义的常量，标准函数 relicate 应用于一个整数 n 和一个对象 x 时，返回 x 的 n 个拷贝构成的列表。

```
serverSize :: ServerState -> Int
serverSize (SS xs) = length xs
```

在寻找最短队列时，我们使用类型 QueueState 中的函数 queueLength

```
shortestQueue :: ServerState -> Int
shortestQueue (SS [q]) = 0
shortestQueue (SS (q:qs))
  | (queueLength (qs!!short) <= queueLength q) = short+1
  | otherwise
    where
    short = shortestQueue (SS qs)
```

由此完成了两个仿真 ADT 的实现。这个例子的目的在于说明分段设计的优点。首先给出类型上运算的非形式描述，然后是类型的 Signature 的描述，最

后是 Signature 的实现。用这种方式分解问题使得每个阶段的问题更容易解决。这个例子也说明类型的实现可以是互不依赖的；因为 ServerState 只使用 QueueState 的抽象数据类型运算，我们可以重新实现 QueueState 的抽象数据类型运算，这对服务器状态不产生任何影响。

习题

16.18 计算下列表达式

```
queueStep (QS 12 3 [Yes 8 4])
queueStep (QS 13 4 [Yes 8 4])
queueStep (QS 14 0 [])
```

16.19 假设

```
serverSt1 = SS [(QS 13 4 [Yes 8 4]), (QS 13 3 [Yes 8 4])]
```

试计算

```
serverStep serverSt1
simulationStep (Yes 13 10) serverSt1
```

16.20 解释为什么不能用函数类型(Int -> QueueState)表示ServerState的类型。设计此类型的一个扩充，使其能表达服务器状态，并实现此类型上signature的运算。

16.21 对于本节的ADT实现，队列和服务器signature的运算是否有冗余？

16.22 如果你还没有做round-robin仿真问题，现在设计一个。

16.23 使用抽象数据类型ServerState实现round-robin仿真。

16.24 实现一个不同的round-robin仿真，它修改了类型ServerState的实现本身。

§16.7 查找树

一个二叉查找树是类型为Tree a的对象，而且其中的元素是有序的。二叉树可以用下列代数类型 Tree 表示

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

怎么样的树是有序的呢？称树(Node val t1 t2)是有序的，如果

- t1中的所有值小于 val;
- t2 中的所有值均大于 val;
- t1和t2本身是有序的。

同时定义树 Nil 是有序的。

例如，查找树可用于表示集合。如何构造查找树类型呢？代数类型 Tree a 不能担当此任，因为它包含无序树，如Node 2 (Node 3 Nil Nil) Nil 。

解决这个问题的办法是只使用建立有序树和保持有序性的运算来构造类型 Tree a 的元素。我们通过抽象数据类型确保只能使用这些“授权”的运算。

查找树的抽象代数类型

我们在 16.4 节讨论了抽象代数类型 Tree 的 Signature，在这里重复写下来

```
module Tree
  (Tree,
   nil,        -- Tree a
   isNil,      -- Tree a -> Bool
   isNode,     -- Tree a -> Bool
   leftSub,    -- Tree a -> Tree a
   rightSub,   -- Tree a -> Tree a
   treeVal,    -- Tree a -> a
   insTree,    -- Ord a => a -> Tree a -> Tree a
   delete,     -- Ord a = >a -> Tree a -> Tree a
   minTree     -- Ord a => Tree a -> Maybe a
  ) where
```

如前所述，类型的实现是

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

区分不同种类树和提取树的组成部分的运算如下：

```
nil :: Tree a

nil = Nil

isNil :: Tree a -> Bool
isNil Nil = True
isNil _   = False

isNode :: Tree a -> Bool
isNode Nil = False
isNode _   = True

leftSub, rightSub :: Tree a -> Tree a

leftSub Nil            = error "leftSub"
leftSub (Node _ t1 _)  = t1

rightSub Nil            = error "rightSub"
rightSub (Node v t1 t2) = t2
```

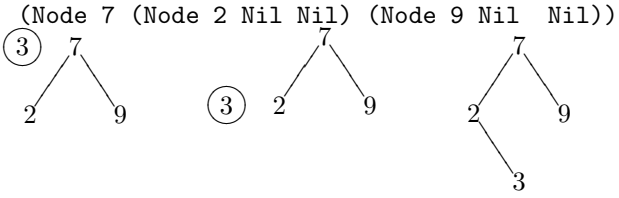
```
treeVal  :: Tree a -> a

treeVal Nil          = error "treeVal"
treeVal (Node v _ _) = v
```

图 16.3 包含插入、删除和连接运算的定义。函数 join 用于连接两棵树，其中第一个参数树上的元素均小于第二个参数树上的元素；这个运算在 delete 中被调用。函数 join 没有被输出，因为这样的运算应用于任意一对二叉查找树时会产生非查找树。

注意，函数`insTree`, `delete`, `minTree`和`join`的类型包含上下文`Ord a`。由第十二章可知，这些函数只能应用于带有运算`<=`的类型。从这些函数的定义中可以看出，它们确实使用了比较运算，由查找树的定义也不难理解这些运算使用有序性。下面考虑图 16.3 中的运算。

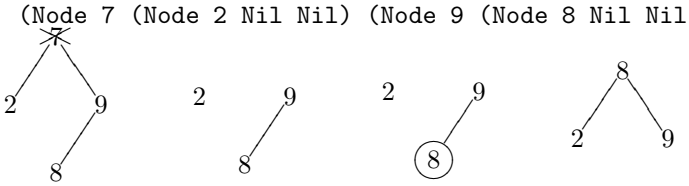
如果插入的元素已经存在，则不进行任何操作; 如果插入的元素小于根结点的值，则插入到左子树上; 如果插入元素大于根结点的值，则插入到右子树上，下图表示插入 3 的过程。

```
(Node 7 (Node 2 Nil Nil) (Node 9 Nil  Nil))
```



如果删除的元素小于（或者大于）根结点的值，则删除在左子树（或右子树）上进行。如果删除的值等于根结点的值，而且根结点的左子树或右子树为 Nil，则可以直接删除该结点，并返回非空子树。当根结点的左右子树均不空时，情况比较复杂。此时，必须将两个子树合并在一起，并且保持有序。要合并两棵非空子树 t1 和 t2，其中 t1 小于 t2，我们选择 t2 的最小元素 mini 作为根结点的值。左子树为 t1，右子树为 t2 中删除 mini 后的树。下图表示删除 7 的过程。

```
(Node 7 (Node 2 Nil Nil) (Node 9 (Node 8 Nil Nil) Nil))
```



函数 minTree 返回类型为`Maybe a`的值，因为空树 Nil 没有最小值。因此，在 join 的 where 子句中必须去除构造符 Just。

修改实现

给定一个查找树，我们可能需要求它的第 n 个元素，

```
indexT:: Int-> Tree a -> a
indexT n t
```

```
insTree :: Ord a => a -> Tree a -> Tree a

insTree val Nil = (Node val Nil Nil)
insTree val (Node v t1 t2)
 | v==val      = Node v t1 t2
 | val > v     = Node v t1 (insTree val t2)
 | val < v     = Node v (insTree val t1) t2

delete :: Ord a => a -> Tree a -> Tree a

delete val (Node v t1 t2)
 | val < v     = Node v (delete val t1) t2
 | val > v     = Node v t1 (delete val t2)
 | isNil t2    = t1
 | isNil t1    = t2
 | otherwise   = join t1 t2

minTree :: Ord a => Tree a -> Maybe a

minTree t
 | isNil t     = Nothing
 | isNil t1    = Just v
 | otherwise   = minTree t1
     where
     t1 = leftSub t
     v  = treeVal t

--The join function is an auxiliary, used in delete
-- join is not exported.

join :: Ord a => Tree a -> Tree a -> Tree a

join t1 t2
  = Node mini t1 newt
    where
    (Just mini) = minTree t2
    newt        = delete mini t2
```

图 16.3: 搜索树上的运算

```
  | inNil t    = error "indexT"
  | n < st1    = indexT n t1
  | n == st1   = v
  | otherwise  = indexT (n-st1-1) t2
    where
    v   = treeVal t
    t1  = leftSub t
    t2  = rightSub t
    st1 = size t1
```

其中 size 的定义如下

```
size :: Tree a -> Int
size t
  | inNil t    = 0
  | otherwise  = 1 + size (leftSub t) + size (rightSub t)
```

如果我们经常性地对树的元素索引, 则需要重复求树的 size。

我们可以通过修改 Tree 的实现来提高 size 运算的效率, 方法是添加一个域用于保存树的规模

```
data Stree a = Nil | Node a Int (Stree a) (Stree a)
```

实现中哪些定义需要修改呢?

- 我们将不得不重新定义 Signature 中的所有运算, 因为这些运算使用类型的定义。例如, 插入函数的新定义如下

  ```
  insTree val Nil = (Node val 1 Nil Nil)
  insTree val (Node v n t1 t2)
   | v==val    = Node v n t1 t2
   | val > v   = Node v (1+size t1 + size nt2) t1 nt2
   | val < v   = Node v (1+size nt1 + size t2) nt1 t2
     where
     nt1 = insTree val t1
     nt2 = insTree val t2
  ```

- 我们必须将 size 加到 Signature 中, 并如下重新定义, 以使用树中存储的规模值

  ```
  size Nil            = 0
  size (Node _ n _ _) = n
  ```

其他的定义不需要修改, 特别是 indexT 的定义 (indexT) 不需要修改。这是使用抽象数据类型而不使用模式匹配的重要原因。如果 (indexT) 的参数使用了模式匹配, 那么它的定义也必须随着其依赖的类型的变化而变化。如本章开始所述, 使用 ADT 使得程序更容易修改。

总之，应该说这些查找树构成一簇类型的模型，因为我们可以对其修改以使其带有不同的信息。例如，我们可以附加一个元素出现次数的信息。在插入一个元素时，相应的计数加 1，删除时，相应的计数减 1。事实上，任何信息均可存储在结点处，插入、删除以及其他运算使用元素的顺序来组织树，而不依赖于存储在结点处的信息。例如，我们可以在结点处存储一个词及其索引信息，由此可以重新实现 10.8 节的索引系统。

习题

16.25 解释如何测试查找树上函数的实现。你可能需要在类型的signature中添加一个打印树的函数。

16.26 定义下列函数

```
successor :: Ord a => a -> Tree a -> Maybe a
closest   :: Int -> Tree Int -> Int
```

一棵树t中v的后继 *(*successor*)* 是t中大于v的最小值；v在t中的最近值是与v的差最小的值。可以假定closest只应用于非空树，所以它总是返回一个值。

16.27 在类型Stree上定义Tree a的signature中的函数。

16.28 为了加速maxTree和其他函数的计算，你可以设想将每个结点的子树的最大值和最小值存储在该结点。重新定义signature中的函数以处理这些最大值和最小值，并利用这些额外信息重新定义函数minTree, maxTree和successor。

16.29 假如在查找树中需要记录一个元素出现的次数，那么类型的signature会受到什么影响？如何实现这些运算？哪些先前定义可以被重用？

16.30 使用改进的查找树，而不使用列表，重新实现 *10.8* 节的索引系统。

16.31 设计一个多态抽象数据类型

```
Tree a b c
```

使得每个结点包含一个类型a的元素，而且树关于此元素有序；每个结点包含一个类型b的值，类似于计数，或者一个索引列表。利用这个多态类型应该可以实现计数树和索引树。

§16.8 集合

一个有限集由一个特定类型的一些元素构成。一个有限集与一个列表既有相同之处，又有区别。列表的例子包括

```
[Joe,Sue,Ben]       [Ben,Sue,Joe]
[Joe,Sue,Sue,Ben]   [Joe,Sue,Ben,Sue]
```

以上列表各不相同。一个列表不仅说明它包含哪些元素，而且这些元素出现的次序以及每个元素出现的次数（元素的多重性）都是重要的。

在许多情形下，顺序和多重性是没有意义的。假设我们谈论的是参加生日聚会的人的集合，我们只关心这些人的姓名；一个人或者参加了聚会，或者没有，所以多重性没有意义，而且人名的顺序也是无关紧要的。换言之，我

们想知道的是参加聚会的人的**集合**。在上例中，这个集合由 Joe，Sue 和 Ben 组成。

如列表，队列和树等一样，集合也可以用许多方式组合：这些将集合组合的运算构成抽象数据类型的 Signature。前面讨论的查找树提供了作用于有序集元素上的运算。例如，我们可以求一个集合 S 中元素 e 的后继是什么？

本节主要讨论集合上的运算。集合的 Signature 如下所示。我们将在实现运算的同时解释其目的。

```
module Set
 (Set ,
  empty              , -- Set a
  sing               , -- a -> Set a
  memSet             , -- Ord a => Set a -> a -> Bool
  union,inter,diff   , -- Ord a => Set a -> Set a -> Set a
  eqSet              , -- Eq a  => Set a -> Set a -> Bool
  subSet             , -- Ord a => Set a -> Set a -> Bool
  makeSet            , -- Ord a => [a] -> Set a
  mapSet             , -- Ord b => (a -> b) -> Set a -> Set b
  filterSet          , -- (a -> Bool) -> Set a -> Set a
  foldSet            , -- (a -> a -> a) -> a -> Set a -> a
  showSet            , -- (a -> String) -> Set a -> String
  card                 -- Set a -> Int
  ) where
```

集合有多种可能的 Signature，其中有些 Signature 对元素类型做了一定的假设。为了测试一个元素是否属于一个集合，我们要求元素的类型属于分类 Eq；事实上，我们在这里假设元素的类型属于有序分类, 因此某些运算的类型说明包含 Ord a 和 Ord b 等上下文。

实现类型和运算

我们将用无重复元素的有序列表表示集合。

```
newtype Set a = SetI [a]
```

`Set a`上的主要运算定义见图 16.4 和 16.5。文件的开始输入了库 List，但是，因为文件包含一个 union 的定义，所以在输入 List 时需要隐蔽其中的同名函数

```
import List hiding (union)
```

另外，在文件的开始我们给出了类型的特例说明。因为在模块的首部没有特例的显式说明，所以在文件的开始将其列出。

下面我们分别解释图 16.4 和 16.5 中的函数。在解释中我们将使用花括号 `'{','}'` 表示集合。注意，这不是 Haskell 的记法。

空集合 {} 由空列表表示，单元素集合 {x} 由单元素列表表示。

集合元素的属于关系用函数 memSet 表示。注意，定义使用了集合元素的有序性。考虑非空列表的三种情况：情况 (memSet.1)，列表的头元素 x 小于查找的元素 y，故应在列表的尾部 xs 继续查找 y；对于情况 (memSet.2)，查找 y 成功；对于情况 (memSet.3)，头元素 x 大于要查找的元素 y，因为列表是从小到大有序的，故 y 不可能在列表中存在。如果使用任意列表表示集合，则此定义无效。

函数 union，inter 和 diff 分别返回两个集合的并、交和差。集合的并由两个集合的元素共同构成，集合的交由同时属于两个集合的元素构成。diff 的定义作为练习留给读者。例如

```
union {Joe,Sue} {Sue,Ben} = {Joe,Sue,Ben}
inter {Joe,Sue} {Sue,Ben} = {Sue}
diff  {Joe,Sue} {Sue,Ben} = {Joe}
```

在定义这些函数时，我们利用了参数的有序性质。此外，我们先定义了“裸”列表上的函数，然后再将这些函数“打包”。例如，在定义 union 时，先定义有序列表上的函数，

```
uni :: Ord a => [a] -> -> [a]
```

然后将其转换成 Set 上的函数。

```
union    :: Ord a => Set a -> Set a -> Set a
union (SetI xs) (SetI ys) = SetI (uni xs ys)
```

注意，花括号不是 Haskell 的表示法，我们可以将其视为下面的简写

```
{e1, ..., en} = makeSet [e1, ..., en]
```

函数 subSet 用于测试第一个参数是否第二个参数的子集；一个集合 x 是另一个集合 y 的子集，如果 x 的所有元素都是 y 的元素。

两个集合相等，如果相应的有序列表相等，如 eqSet 的定义所述。注意，定义 Set a 上的相等，需要 a 上的相等运算。函数 eqSet 在 Signature 中输出，但是，我们也说明 Set a 为 Eq 的一个特例，并将==定义为 eqSet

```
instance Eq a => Eq (Set a) where
  (==) = eqSet
```

ADT 的相等一般地不一定是底层类型上的相等。如果我们使用了任意列表表示集合，则相等测试会更复杂，因为 [1, 2] 和 [2, 1, 2, 2] 表示同一个集合。

我们还将列表的顺序作为 Set 上的顺序输出

```
instance Ord a => Ord (Set a) where
  (<=) = leqSet
```

子集顺序没有定义为<=，因为习惯上 Ord 中的<=是全序，即对所有元素 x 和 y，或者 x⌊=y，或者 y⌊=x。子集顺序不是全序，但是，列表上的字典序是全序。习题中列出了一些比较的例子。

```
import List hiding ( union )

instance Eq a => Eq (Set a) where
  (==) = eqSet

instance Ord a => Ord (Set a) where
  (<=) = subSet

newtype Set a = SetI [a]

empty               :: Set a
empty  = SetI []

sing                :: a -> Set a
sing x = SetI [x]

memSet              :: Ord a => Set a -> a -> Bool
memSet (SetI []) y    = False
memSet (SetI (x:xs)) y
  | x<y     = memSet (SetI xs) y                    (memSet.1)
  | x==y    = True                                  (memSet.2)
  | otherwise   = False                             (memSet.3)

union    :: Ord a => Set a -> Set a -> Set a
union (SetI xs) (SetI ys) = SetI (uni xs ys)

uni    :: Ord a => [a] -> [a] -> [a]
uni [] ys        = ys
uni xs []        = xs
uni (x:xs) (y:ys)
  | x<y     = x : uni xs (y:ys)
  | x==y    = x : uni xs ys
  | otherwise   = y : uni (x:xs) ys

inter    :: Ord a => Set a -> Set a -> Set a
inter (SetI xs) (SetI ys) = SetI (int xs ys)

int    :: Ord a => [a] -> [a] -> [a]
int [] ys = []
int xs [] = []
int (x:xs) (y:ys)
  | x<y     = int xs (y:ys)
  | x==y    = x : int xs ys
  | otherwise   = int (x:xs) ys
```

图 16.4: 抽象数据类型集合上的运算, 第一部分

```
subSet (SetI xs) (SetI ys) = subS xs ys

subS        :: Ord a => [a] -> [a] -> Bool
subS [] ys = True
subS xs [] = False
subS (x:xs) (y:ys)
  | x<y     = False
  | x==y    = subS xs ys
  | x>y     = subS (x:xs) ys

eqSet        :: Eq a => Set a -> Set a -> Bool
eqSet (SetI xs) (SetI ys) = (xs == ys)
--
makeSet             :: Ord a => [a] -> Set a
makeSet = SetI . remDups . sort
          where
          remDups []     = []
          remDups [x]    = [x]
          remDups (x:y:xs)
            | x < y     = x : remDups (y:xs)
            | otherwise = remDups (y:xs)

mapSet              :: Ord b => (a -> b) -> Set a -> Set b
mapSet f (SetI xs) = makeSet (map f xs)

filterSet           :: (a -> Bool) -> Set a -> Set a
filterSet p (SetI xs) = SetI (filter p xs)

foldSet             :: (a -> a -> a) -> a -> Set a -> a
foldSet f x (SetI xs)  = (foldr f x xs)

showSet             :: (a->String) -> Set a -> String
showSet f (SetI xs) = concat (map ((++"\n") . f) xs)

card                :: Set a -> Int
card (SetI xs)     = length xs
```

图 16.5: 抽象数据类型集合上的运算, 第二部分

函数 makeSet 将任意列表转化为集合：先将列表排序，然后删除重复元素，最后用 SetI 包装。sort 的定义由库 List 中输入。

函数 mapSet, filterSet 和 foldSet 的作用类似于 map, filter 和 foldr，只是其定义域是集合。后两个函数的定义与 filter 和 foldr 基本相同，mapSet 定义中需要删除重复元素。

函数showSet f (SetI xs) 给出一个集合的可打印形式，每行一个元素，其中 f 给出每个元素的可打印形式。

```
showSet f (SetI xs) = conct  (map ((++"\n") . f ) xs)
```

一个集合的基数是它的成员的个数。函数 card 定义为列表的长度。

在下一节我们将在 Set 库的基础上，建立关系和图的函数库。

习题

16.32 比较下列集合对在顺序<=和subSet下的关系：

```
[3]        [3,4]
[2,3]      [3,4]
[2,9]      [2,7,9]
```

16.33 定义函数diff使得diff s1 s2是由属于s1但不属于s2的元素构成的集合。

16.34 定义下列表示对称差的函数

```
symmDiff:: Ord a => Set a -> Set a -> Set a
```

两个集合的对称差由属于一个集合但是不属于另一个集合的元素构成。例如，

```
symmDiff {Joe,Sue} {Sue,Ben} = {Joe,Ben}
```

在定义symmDiff时你能使用函数diff吗？

16.35 你能定义下列函数吗？

```
powerSet :: Ord a => Set a -> Set (Set a)
```

此函数返回一个集合的所有子集。你能仅使用抽象数据类型的运算而不使用抽象数据类型的具体实现定义此函数吗？

16.36 如何利用抽象数据类型的运算定义下列函数？

```
setUnion :: Ord a => Set (Set a) -> Set a
setInter :: Ord a => Set (Set a) -> Set a
```

16.37 无穷集合（例如，数的无穷集合）能用有序列表表示吗？例如，你能判断两个无穷列表是否相等吗？

16.38 抽象数据类型Set a有多种表述方法。其中包括任意列表（没有有序和无重复元素限制）和布尔值函数，即类型为a -> Bool的元素。试用上述两种表示法实现Set a。

16.39 利用查找树实现抽象数据类型集合Set。

16.40 试用有序列表实现查找树抽象数据类型，并比较两种实现的差异。

§16.9 关系和图

我们现在使用集合 ADT 来实现关系，并将关系作为图来研究。

关系

一个二元关系将一个集合的某些元素相关联。一个家庭关系可以用关系 isParent 来表示：Ben 和 Sue，Ben 和 Leo，以及 Sue 和 Joe 具有这种关系。换句话说，这个关系关联着二元组 (Ben,Sue)，(Ben,Leo) 和 (Sue,Joe)，所以这个关系可表示为集合

```
isParent = {(Ben, Sue), (Ben, Leo), (Sue, Joe)}
```

一般地，我们定义

```
type Relation a = Set (a,a)
```

这个定义表明所有的集合运算也是关系上的运算。我们可以使用 memSet 测试两个元素是否具有一个关系；两个的关系的并，如 isParent 和 isSibling 的并表示或者是父亲或者是亲属的关系等。

假定已有关系 isParent，下面讨论家庭关系的两个例子。我们首先定义函数 addChildren，它将一些人以及他们的孩子作为一个集合返回；然后定义关系 isAncestor。这些函数的代码见图 16.6。

例 1. 我们将采用由底向上的方法，首先求与一个给定元素相关的所有元素。例如，求 Ben 的所有孩子，我们需要求出所有第一个分量为 Ben 的二元组，然后返回它们的第二个分量。完成这个任务的函数是 image，而且 Ben 的所有孩子将是

```
image isParent Ben = {Sue, Leo}
```

那么，如何求与一个集合中的元素关联的所有元素呢？我们可以先分别求每个元素的 image，然后求这些集合的并。一组集合的并可以通过折叠二元运算 union 完成。

```
unionSet{s1, ..., sn}
   = s1 ∪...∪sn
   = s1 `union` ... `union` sn
```

最后，如何将集合中所有人的孩子添加到集合上呢？我们先求这些人的集合在 isParent 下的图像，然后将这两个集合并列一起，其实现是函数 addChildren。

例 2. 第二个任务是求关系 isAncestor。此问题的一般陈述是：求一个关系的传递闭包。这个任务由函数 tClosure 完成，见图 16.6。其方法是在一个关系上加上祖父关系，曾祖父关系等，直至没有新的关系生成。稍后我们详细解释传递闭包。

如何求曾祖父关系 isGrandParent 呢？下列两个二元组的匹配表明 Ben 是 Joe 的祖父。

```
(Ben,Sue)    (Sue,Joe)
```

```
image :: Ord a => Relation a -> a -> Set a
image rel val = mapSet snd (filterSet ((==val).fst) rel)

setImage :: Ord a => Relation a -> Set a -> Set a
setImage rel = unionSet . mapSet (image rel)

unionSet :: Ord a => Set (Set a) -> Set a
unionSet = foldSet union empty

addImage :: Ord a => Relation a -> Set a -> Set a
addImage rel st = st `union` setImage rel st

type People = String

isParent :: Relation People

isParent = isParent   --  dummy definition
                      --  needs to be replaced

addChildren :: Set People -> Set People
addChildren = addImage isParent

compose :: Ord a => Relation a -> Relation a -> Relation a

compose rel1 rel2
  =  mapSet outer (filterSet equals (setProduct rel1 rel2))
     where
     equals ((a,b),(c,d)) = (b==c)
     outer  ((a,b),(c,d)) = (a,d)

setProduct :: (Ord a,Ord b) => Set a -> Set b -> Set (a,b)
setProduct st1 st2 = unionSet (mapSet (adjoin st1) st2)

adjoin :: (Ord a,Ord b) => Set a -> b -> Set (a,b)
adjoin st el = mapSet (addEl el) st
               where
               addEl el el' = (el',el)

tClosure :: Ord a => Relation a -> Relation a

tClosure rel = limit addGen rel
               where
               addGen rel' = rel' `union` compose rel' rel

limit             :: Eq a => (a -> a) -> a -> a
limit f xs
```

我们称此为 isParent 与其本身的关系复合。一般地

```
isGrandParent
   = isParent `compose` isParent
   = [(Ben,Joe)]
```

在定义 compose 时，我们使用了 setProduct 求两个集合的积，它由第一个集合的每个元素与第二个集合的每个元素构成的二元组构成。例如，

```
setProduct[Ben, Suzie] [Sue, Joe]
  = { (Ben,Sue),(Ben,Joe),(Suzie,Sue),(Suzie,Joe) }
```

函数 setProduct 使用了函数 adjoin，其功能为构造由一个集合的每个元素与一个给定元素形成的二元组集合。例如,

```
adjoin Joe [Ben,Sue] = { (Ben, Joe),(Sue,Joe) }
```

一个关系 rel 是 **传递的**，如果对于 rel 中所有的 (a, b) 和 (b, c)，(a, c) 也属于 rel。一个关系的 **传递闭包** 是包含 rel 并且传递的最小关系。关系 rel 的传递闭包用`tClosure rel`表示，其计算方法是使用 compose 不断地添加 rel 的后代，直至没有新的二元组可添加为止。为此，我们使用一个多态高阶函数 limit。`limit f x`给出下列序列的 **极限**

```
x, f x, f (f x), f (f (f x)), ...
```

如果一个序列的极限存在，则极限值是序列最终所取的值。极限值是序列中第一个其后继等于它本身的元素。例如，Ben 是 Sue 的父亲，Sue 是 Joe 的母亲，Joe 本身无子女。现在定义

```
addChildren :: Set Person -> Set Person
```

它将在一个集合上添加此集合的所有元素的子女，例如

```
addChildren {Joe,Ben} = {Joe,Sue, Ben}
```

下面是计算一个函数在一个集合上极限的例子

```
limit addChildren Ben
  ?? {Ben} == {Ben,Sue} ⇝ False
⇝ limit addChildren Ben,Sue
  ?? {Ben,Sue} == {Ben,Joe,Sue} ⇝ False
⇝ limit addChildren Ben,Joe,Sue
  ?? {Ben,Joe,Sue} == {Ben,Joe,Sue} ⇝ True
⇝ verb+ Ben,Joe,Sue+
```

简化上下文

图 16.6 给出类型分类上下文简化的例子。函数 adjoin 要求类型 a 和类型 b 上具有顺序关系。Haskell 包括特例说明

```
instance (Ord a, Ord b) => Ord (a,b) ...        (pair)
```

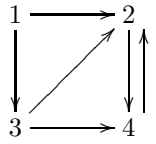
所以，Ord a 和 Ord b 可以确保 adjoin 调用 mapSet 时的要求 Ord(a, b)。类似地，在定义 compose 时需要在类型`((a,a),(a,a))`上具有顺序，Ord a 可确保这一点，因为`(pair)`可以导出`((a,a),(a,a))`上有序。

图

一个关系还可以视为一个有向图。例如，下列关系

```
graph1 = {(1,2), (1,3), (3,2), (3,4), (4,2), (2,4)}
```

可以看成下图



其中 a 到 b 的箭头表示 (a, b) 属于此关系。那么关系的传递闭包表示什么呢？两个点 a 和 b 在关系`tClosure graph1`中相关，如果在图中有一条从 a 到 b 的**路径**。例如，二元组 (1, 4) 属于闭包，因为存在一条从 1 经过 3，2，然后到达 4 的路径，而二元组 (2, 1) 不属于闭包，因为在图 graph1 中不存在从 2 到 1 的路径。

强连通图

寻找一个图的强连通分支是许多应用中要解决的问题，包括网络和编译器。每个图可以分解成几个分量，在每个分量中，从任意一个结点均可通过一条路径到达同一分量中的其他结点。图 graph1 的分量是`1，3`和`2,4`。我们可以分两步求图的分量：

- 首先构造连接同一分量结点的关系，然后
- 构造这个关系生成的分量（或者等价类）。

如果 (x, y) 和 (y, x) 均属于关系的闭包，则存在 x 到 y 以及 y 到 x 的通路，故可定义

```
connect :: Ord a => Relation a -> Relation a
connect rel = clos `inter` solc
              where
              clos = tClosure rel
              solc = inverse clos

inverse :: Ord a => Relation a -> Relation a
inverse = mapSet swap
          where
          swap (x,y) = (y,x)
```

那么如何构造由关系 graph1 形成的分量呢？我们从下列集合出发

```
{{1},  {2},  {3},  {4}}
```

然后不断地给集合中的每个类添加它在此关系下的图像，直至达到一个不动点。一般地，这个过程可定义如下

```
classes :: Ord a => Relation a -> Set (Set a)
```

```
classes rel
  = limit (addImages rel) start
    where
    start = mapSet sing (eles rel)
```

其中的辅助函数为

```
eles :: Ord a => Relation a -> Set a
eles rel = mapSet fst rel `union` mapSet snd rel

addImages :: Ord a => Relation a -> Set (Set a) -> Set (Set a)
addImages rel = mapSet (addImage rel)
```

图中的搜索

许多算法需要在图的结点中进行搜索；比如求从一个结点到另一个结点的最短路径，或者计算两个结点之间路径的数量。深度优先和广度优先是两种通用的搜索模式。在深度优先搜索中，我们从一个结点开始，先搜索此结点的第一个子结点以下的所有结点，然后转到下一个子结点。在广度优先搜索中，我们先搜索一个结点的所有子结点，然后搜索子结点的子结点，等等。例如，在图 graph1 中从结点 1 开始搜索，深度优先搜索的顺序是 [1, 2, 4, 3]，广度优先的顺序是 [1, 2, 3, 4]。这些例子表明，搜索可以表示为变换

```
breadthFirst :: Ord a => Relation a -> a -> [a]
depthFirst   :: Ord a => Relation a -> a -> [a]
```

例如，`breadthFirst graph1 1 = 1,2,3,4`。在这些函数中列表的使用很关键：我们不仅关心一个结点下的结点（用 tClosure 实现），而且关心他们发生的顺序。

在以上两种搜索中，关键的一步是求一个结点的所有未被访问过的子结点。为此，我们定义

```
newDescs :: Ord a => Relation a -> Set a -> a -> Set a
newDescs rel st v = image rel v `diff` st
```

此函数返回关系 rel 中 v 的所有不属于 st 的子结点。这里有一个问题：这个函数的结果是一个集合，不是列表，但是，我们希望这些元素具有一定的顺序。一种解决方法是在抽象数据类型 Set 上添加一个函数，

```
flatten :: Set a -> [a]
flatten (SetI xs) = xs              (setList)
```

这里假定集合是用有序列表实现的。另一种方法是定义下列函数

```
minSet :: Set a -> Maybe a
```

这个函数返回一个非空集的最小元素。这个函数可用于将一个集合转换为一个列表，而不需要破坏抽象数据类型的抽象性。我们假定一个类型为 (setList) 的函数的存在性，暂不讨论它的定义。现在我们可以定义

```
findDescs :: Ord a => Relation a -> [a] -> a -> [a]
findDescs rel xs v = flatten (newDescs rel (makeSet xs) v)
```

广度优先搜索

广度优先搜索需要重复使用函数 findDescs 直至找到它的极限。前面定义的函数 limit 可用于求极限，所以我们可定义

```
breadthFirst :: Ord a => Relation a -> a -> [a]
breadthFirst rel val
    = limit step start
      where
      start = [val]
      step xs = xs ++ nub (concat (map (findDescs rel xs) xs))
```

函数 step 完成以下运算：

- 首先求 xs 中元素的子结点而且这些子结点不属于 xs。这个操作可用将`(findDescs rel xs)`映射到 xs 上完成。
- 将列表的列表合并成一个列表。
- 因为一个结点可以是几个结点的子结点，所以列表中可能有重复元素，所以必须将重复元素删除。这个工作可以用库函数`nub :: Eq a => [a] -> [a]`完成，它将删除列表中每个元素的重复出现。

深度优先搜索

如何进行深度优先搜索呢？我们首先将问题推广为

```
depthSearch :: Ord a => Relation a -> a -> [a] -> [a]
depthFirst rel v = depthSearch rel v []
```

其中第三个参数表示已经访问过的结点列表，所以没有出现在函数 depthFirst 调用中。

```
depthSearch rel v used
    = v : depthList rel (findDescs rel used' v) used'
      where
      used' = v:used
```

其中辅助函数 depthList 求出一个结点列表的所有子结点

```
depthList :: Ord a => Relation a -> [a] -> [a] -> [a]

depthList rel [] used = []

depthList rel (val:rest) used
  = next ++ depthList rel rest (used++next)
    where
    next = if elem val used
```

```
        then []
        else depthSearch rel val used
```

此定义含有两个等式，第一个等式定义没有结点需要搜索的情况。在第二个等式中，解由两部分构成：

- 函数 next 给出图中从 val 可到达的结点。如果 val 属于 used，则这部分为空，否则调用 depthSearch。
- 在列表的尾部调用 depthList，并将 next 加到已访问结点列表中。

以上两定义使用了 **相互递归**，即两个等式相互调用。当然可以使用一个函数完成以上两个函数的功能，但上述定义更自然。

习题

16.41 试计算

```
classes (connect graph1)
classes (connect graph2)
```

其中 graph2 = graph1 ∪ 4, 3.

16.42 试计算下列式子

```
breadthFirst graph2 1
depthFirst graph2 1
```

其中graph2在前一习题中定义。

16.43 利用搜索模型定义图中一个结点到另一个结点的最短路径函数

```
distance :: Eq a => Relation a -> a -> a -> Int
```

例如

```
distance graph1 1 4 = 2
distance graph1 4 1 = 0
```

其中 *0* 表示两点间不存在路径或者两结点重合。

16.44 每个边上带有一个数值**权**的图称为带权图。设计一个带权图类型，并定义宽度优先和深度优先搜索函数。函数返回一个二元组列表，其中每个二元组由一个结点和搜索开始结点到该结点的最短距离构成。

16.45 一个异种关系是不同类型元素间的关系。例如，一个人与其年龄的关系。试为这种关系设计一个类型。能否修改Relation a的函数使其适用于你定义的类型。

§16.10 评论

本节讨论引入 ADT 后的有关问题。首先，我们还没有谈到 ADT 上函数的验证问题，这是因为这种验证仍然是定理的证明，关于实现的定理证明方法完全同前所述。对于抽象数据类型成立的定理恰好是符合 Signature 中函数上类型约束的定理。例如，对于一个队列，我们可以证明

```
remQ (addQ x emptyQ) = (x, emptyQ)
```

方法是证明队列的实现满足相应的等式。但是，下列等式不成立

```
emptyQ = Qu []
```

这是因为它破坏了 ADT 的信息封闭性。

其次，注意到集合的实现需要我们证明某些性质，通常称之为 **证明的必要条件**。我们假定集合使用了无重复元素的有序列表实现；我们应该证明这个实现上的每个运算保持这个性质。

最后，注意分类和抽象数据类型均使用了 signature，所以有必要讨论一下它们的异同。

- 它们的目的不同：`signature`在 ADT 中用于信息隐蔽和组织程序；在分类中用于过载命名，以使得同一个名可用于一个分类的不同类型。
- ADT 的 signature 与一个单一的实现类型关联，这个实现类型可以是单态的，也可以是多态的。另一方面，分类的 signature 与多个特例相关联；事实上，这正是引入分类的关键所在。
- ADT 的 signature 中的函数是存取 ADT 类型的唯一方法。而在分类上没有这样的信息隐蔽：一个类型作为一个分类的成员，它至少必须提供 signature 中的函数。
- ADT 可以是多态的，例如，我们可以实现一个多态的搜索树类型。分类将单个类型，而非多态的类型簇分类；第十八章将介绍的构造符分类是多态类型簇的分类，是分类的扩展。

小结

本章的抽象数据类型有三个相关的重要性质。

- 它们提供了一个类型的自然表示，避免了过多的具体化。一个抽象数据类型恰好包含了与此类型自然相关的运算。
- 一个抽象数据类型的 signature 是用户和实现者之间的稳固界面：一个系统的实现可以在界面的两边独立地进行。
- 如果一个类型的实现需要修改，只有 signature 中的运算需要修改；使用 signature 运算的任何函数均无需改动。在搜索树的例子中，我们通过增加规模信息后类型实现的修改说明了这一点。

我们看到了各种 ADT 开发的例子。最重要的是，我们提议的三阶段设计模式和仿真类型的实例：首先命名类型，然后给类型以非形式的描述，最后设计 signature。接下来我们便可以独立地完成 signature 中运算的实现。

设计 signature 的困难之一是确定它是否包含了所有相关的运算；我们给出一个 signature 应该包含的运算种类的列表，通过检查这个列表可以选择 signature 应该包含的定义。

第十七章　惰性计算

我们曾讲过, Haskell 计算步骤的顺序不会影响计算结果, 它只可能影响计算是否会得出一个结果。本章介绍 Haskell 的**情性计算**策略。惰性计算名符其实：一个惰性计算器在计算一个函数的值时，只有计算过程需要参数的值时，惰性计算器才去计算它的值；而且，如果一个参数是有结构的（例如，一个列表），那么惰性计算器只计算参数中所需要的部分。

惰性计算影响着我们书写程序的方式。因为一个中间列表是按需要**生成的**，所以使用中间列表在计算上代价不一定很高。我们将给出一系列例子，包括一个语法分析实例研究。

在构造语法分析器时，我们构造了一个多态、高阶函数构成的工具包，这些工具可以灵活地结合和扩充，并且用于语言的各种处理。函数程序语言的一个突出特点便是它能够提供定义这样的构件结合的方法。

我们还将扩展列表概括记法，虽然扩展并不能让我们书写新程序，但是它可以使大量列表处理程序更容易表达和理解，特别是那些首先生成解，然后再测试这些解的程序。

惰性计算的另一个作用是描述无穷结构。计算无穷结构的时间也是无穷的，但是使用惰性计算时，只有无穷结构的一部分需要用于计算。任何递归类型都包含无穷结构；我们将主要讨论列表类型，因为它是使用最广泛的无穷结构。

在介绍完各种例子之后，例如素数以及随机数的无穷列表，我们讨论无穷列表在程序设计中的重要作用，并将看到处理无穷列表的程序也可以看作产生数据流和消耗数据流的过程。基于这种思想，我们讨论如何完成模拟实例研究。

本章最后讨论在惰性计算和无穷列表的情况下的程序验证；这一节只是本领域的一个简介，但是其中指出了有关详细内容的参考资料。

17.1 节和 17.2 节是本章的基本内容，读者可以选择是否阅读其后的各节内容，这些内容不影响今后各章的学习。

§17.1　惰性计算

Haskell 计算的主要方法是函数应用，其基本思想很简单：要计算函数 f 应用于它的参数 $a_1, a_2, ..., a_k$，只需将表达式 a_i 代入函数定义中相应的变量。例如，如果

```
f x y = x + y
```

则

```
f (9-3) (f 34 3)
⇝ (9-3)+(f 34 3)
```

因为我们用(9-3)代替了 x，用(f 34 3)代替了 y，表达式(f 34 3)和(9-3)在传给函数之前并未进行计算。

在这种情况下，继续计算时需要计算"+" 的参数，由此得

```
⇝ 6+(34+3)
⇝ 6+37
⇝ 43
```

在这个例子中，两个参数最终都进行了计算，但情况并非总是这样的。如果我们定义

```
g x y = x+12
```

则

```
g (9-3) (g 34 3)
⇝ (9-3)+12
⇝ 6+12
⇝ 18
```

在这里(9-3)被代入 x，但是因为 y 在定义等式的右边没有出现，所以参数(g 34 3)在结果中不出现，因此并未被计算。我们在这里看到了惰性计算的第一个优点 — 不需要的参数将不被计算。以上例子过于简单，如果不需要第二个参数，为什么要写在那里？一个更实际的例子是

```
switch :: Int -> a -> a -> a
switch n x y
  | n >0        = x
  | otherwise   = y
```

如果 n 是正整数，则结果是 x；否则结果是 y。定义或者使用参数 x 或者使用参数 y，在第一种情况 y 不被计算，在第二种情况，x 不被计算。第三个例子是

```
h x y = x+x
```

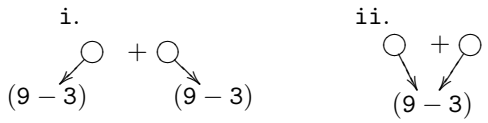
由此得

```
h (9-3) (h 34 3)              (h-eval)
⇝ (9-3)+(9-3)
```

在这里参数(9-3)似乎需要计算两次，因为在代入时它重复出现。惰性计算保证一个**多次出现的参数不会被重复计算**。这个过程可以通过下列对应的并行计算步骤模拟

```
h (9-3) 17
⇝ (9-3)+(9-3)
⇝ 6+6
⇝ 12
```

Haskell 系统的实现没有重复计算，因为计算是在图上进行的，而不是在代表表达式的树上进行。例如，(h-eval) 的计算不使用像 (i) 中重复参数的模式，而是像图 (ii) 所示，加号的两边使用同一个表达式。



最后一个例子是下列模式匹配函数

```
pm (x,y) = x+1
```

将其应用于二元组 (3+2, 4-17) 的计算过程为

```
pm (3+2, 4-17)
⇝ (3+2)+1
⇝ 6
```

在这里参数只有一部分被计算。二元组的第二个分量没有被计算，因为计算过程中不需要它。以上是对惰性计算的非正式介绍，可以将其归纳为下列三点：

- 一个函数的参数只有在计算过程需要它时才被计算。
- 一个参数不必完全计算：只有需要的部分参加计算。
- 一个参数最多被计算一次。这是通过把表达式用图表示，然后在图上进行计算实现的。

下面我们将正式地解释惰性计算的计算规则。

§17.2 计算规则与惰性计算

我们在 3.7 节看到，一个函数的定义由一些条件等式构成。每个条件等式可以含有多个子句，而且可以有 where 引导的多个局部定义。每个等式左边是函数名应用于一些模式。

```
f p1 p2 ... pk
  | g1        = e1
  | g2        = e2
  ...
  | otherwise = er
    where
    v1 a1  ... = r1
    ...
f q1 q2 ... qk
  = ...
```

f a1 ... ak的计算包含三个方面。

计算 — 模式匹配

为了确定使用哪一个等式，我们需要计算参数。这些参数并不需要完全计算，只需计算到足以辩识它们是否和相应的模式匹配。如果它们和模式 p1 至 pk 匹配，则使用第一个等式；否则检查是否和第二个等式匹配，这期间或许需要做进一步的参数计算。重复这个过程，直至找到一个与它们匹配的等式或者没有与它们匹配的等式（此时系统会产生 program error）。例如，给定如下定义

```
f :: [Int] -> [Int] -> Int
f [] ys         = 0                              (f.1)
f (x:xs) []     = 0                              (f.2)
f (x:xs) (y:ys) = x+y                            (f.3)
```

`f [1 .. 3] [ 1 .. 3]`的计算过程如下：

```
 f [1 .. 3] [ 1 .. 3]      (1)
 ⇝ f (1:[2..3]) [1..3]     (2)
 ⇝ f (1:[2..3]) (1:[2..3]) (3)
 ⇝ 1+1                     (4)
```

在阶段(1)，没有足够的信息决定参数是否与`(f.1)`匹配。计算一步后到达(2)，此时可看出参数与`(f.1)`不匹配。(2) 的第一个参数匹配`(f.2)`的第一个模式，为此我们检查第二个参数。进一步计算得到(3)，它表明参数与`(f.2)`不匹配，而是与`(f.3)`匹配，故得 (4)。

计算 — 守卫

为了解释方便，我们假设第一个条件等式与参数匹配。此时条件等式中的模式`p1`到`pk`被表达式`a1`至`ak`代替，下一步必须确定条件等式中的哪一个子句适用。依次计算守卫，直至找到一个其值为 True；相应的子句便是计算要使用的定义。如果我们有下列函数定义

```
f :: Int -> Int -> Int -> Int
f m n p
  | m>=n && m>= p   = m
  | n>=m && n>=p    = n
  | otherwise       = p
```

则有

```
f (2+3) (4-1) (3+9)
  ?? (2+3)>=(4-1) && (2+3)>=(3+9)
  ??   ⇝ 5>=3 && 5>=(3+9)
  ??   ⇝ True && 5>=(3+9)
  ??   ⇝ 5>=(3+9)
  ??   ⇝ 5>=12
  ??   ⇝ False
  ??  3>=5 && 3 >=12
  ??   ⇝ False && 3>=12
  ??   ⇝ False
  ??  otherwise  ⇝ True
⇝ 12
```

请读者找出上述计算过程中哪些部分是共享的。

计算 — 局部定义

由 where 引导的子句中的值是按需计算的；一个值的计算只有需要时才会开始。给定下列定义

```
f :: Int -> Int -> Int
f m n
  | notNil xs      = front xs
  | otherwise      = n
    where
    xs = [m .. n]

front (x:y:zs) = x+y
front [x]      = x

notNil []    = False
notNil (_:_) = True
```

表达式`f 3 5`的计算过程如下

```
f 3 5
  ??     notNil xs
  ?? |   where
  ?? |   xs = [3..5]
  ?? |      ⇝ 3:[4..5]                                    (1)
  ??  ⇝ notNil (3:[4..5])
  ??  ⇝ True
⇝ front xs
|        where
|        xs = 3:[4..5]
|            ⇝ 3:4:[5]                                    (2)
⇝ 3+4                                                     (3)
⇝ 7
```

在计算`notNil xs`时，xs 的计算开始，一步计算后得到 (1)，它表示这个条件是真的。计算`front xs`需要 xs 的更多信息，所以进一步计算得到 (2)。此时参数与 front 定义模式匹配成功，由此得到 (3)，最后得到结果 7。

运算符和其他构造表达式的运算

上面介绍了计算函数应用的三个方面。下面介绍关于预定义运算符的计算。如果我们写出这些运算的 Haskell 定义，例如

```
True   && x = x
Flase  && x = False
```

那么它们的计算也遵循 Haskell 定义规则。例如，从左到右的计算顺序，如果'&&'的第一个参数是 False，则它的第二个参数不必计算。这与许多其他的程序语言不同，在这些语言中 and 函数会计算两个参数的值。

其他的运算则依情况而定。例如算术运算，加法的计算需要计算两个参数，但是列表上的相等在比较`[]`和`(x:xs)`时可返回 False，而不必计算 x 和 xs。一般地，Haskell 语言的实现使得明显不必要的计算不会发生。

我们知道，if ··· then ··· else，case，let 和 lamdda 表达式可用于构造表达式。它们的计算遵循函数应用的规则。特别地，if ··· then ··· else 被视为守卫来计算，case 被视为模式匹配，let 被视为 where 子句，lambda 表达式的计算如上述命名函数`f`的应用一样。

最后，我们讨论在应用顺序上的选择。

计算顺序

在描述 Haskell 的计算时，除去参数不会重复计算这一点外，还需说明当有多个函数应用选择时，函数应用的顺序如何进行。

- 计算**由外向里**进行。如下面的情形

`f e1 (f2 e2 17)`

其中一个应用中包含另一个应用，此时，应该选择外层的应用`f1 e1 (f2 e2 17)`做计算。

- 否则，计算应该**从左向右**进行。在下列表达式中 `f1 e2` + `f2 e2`

下划线表达式均被计算，左边的`f1 e1`先计算。

以上这些规则足以描述惰性计算如何进行。在今后几节，我们将讨论惰性计算给函数程序设计带来的影响。

§17.3 再谈列表概括

虽然列表概括并未在 Haskell 中增加新程序，但是它可以使我们以一种全新的、更清晰的方式也书写程序。在 5.5 节的基础上，列表概括允许我们在一个表达式中使用多个 map 和 filter。使用这些函数的结合可以书写生成和测试这些元素的组合，最后返回结果。在本节，我们首先回顾列表概括的语法，然出给出一些代表性的例子，最后是一些较大的例子。

语法

一个列表概括具有格式`[e | q1, ..., qk]`，其中每个 q_i 具有下列形式之一

- 可以是一个生成器，形如`p<-lExp`，其中 P 是一个模式，lExp 是列表类型的表达式；
- 可以是一个**测试**，即一个布尔型表达式`bExpr`

在 q_i 中出现的表达式 lExp 或 bExp 可以包含模式 q_1 到 q_{i-1} 中出现的变量。

简单例子

多个生成器可用于结合两个或者更多的列表

```
pairs :: [a] -> [b] -> [(a, b)]
pairs xs ys = [(x,y) | x<- xs, y<- ys]
```

这个例子很重要，因为它表示了选择 x 和 y 的方式

```
pairs [1,2,3] [4,5]
⇝ [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

其中 xs 的第一个元素 1 赋给 x，然后，对于这个固定的值，选取 ys 中所有可能的值。对于 xs 的其余元素，即 2 和 3，重复这个过程。

这种选择不是随意的，因为对于下列的式子

```
triangle :: Int -> [(Int, Int)]
triangle n = [(x,y) | x <- [1 .. n], y <- [1 .. x]]
```

第二个生在器`y <- [1 .. x]`依赖于第一个生成器生成的 x 值。

```
triangle 3
⇝ [(1,1), (2,1), (2,2), (3,1), (3,2), (3,3)]
```

对于 x 的第一个选择 1, y 的值从[1 .. 1]中选出, 对于 x 的第二个值, y 从 [1 .. 2] 中选取, 依次类推。

三个正整数构成一个 Pythagorean 三元组, 如果前两个数的平方和等于第三个数的平方。我们可以写出所有这些三元组的列表, 其中每个分量以 n 为界:

```
pyTriple n
  = [(x,y,z) | x <- [2 .. n], y <- [x+1 .. n],
               z <- [y+1 .. n], x*x + y*y = z*z]
```

```
pyTriple 100
⇝ [(3,4,5),(5,12,13),(6,8,10),...,(65,72,97)]
```

其中测试结合了三个生成器产生的元素。

列表概括的计算

如何描述列表概括的结果呢? 一种方法是将它翻译成应用 map, filter 和 concat 的表达式。我们在这里介绍一种不同的方法, 即通过直接计算表达式获得结果。

我们先介绍一个非常有帮助的记法。我们用 e[f/x] 表示将表达式 e 中变量 x 的所有出现用 f 代替, 即在表达式 e 中 x 被 f 代换。如果 p 是一个模式, 我们使用 e[f/p] 表示将 p 中的变量用 f 的适当部分代换。例如

```
[(x,y) | x<- xs]{[2,3]/xs} = [(x,y) | x<- [2,3]]
```

```
(x+sum xs){(2,[3,4])/(x,xs)} = 2 + sum [3,4]
```

因为当 (2, [3, 4]) 与 (x, xs) 匹配时, 2 与 x 匹配, [3, 4] 与 xs 匹配。

我们现在解释列表概括。记法看起来可能显得繁琐, 但意义是清晰的。生成器 v <- [$a_1, \cdots, a_n$] 的作用是将 a_1 到 a_n 依次赋给 v。在列表概括计算中, 将表达式中的变量 v 用相应的值代换

```
[e | v <- [a1,···,an],q2,···,qk]
⇝   [ e{a1/v} |q2{a1/v},···,qk{a1/v}]
    ++    ... ++
    [ e{an/v} |q2{an/v},···,qk{an/v}]
```

例如

```
  [x+y | x <- [1,2], isEven x, y <- [x .. 2*x]]
⇝ [1+y | isEven 1, y <- [1 .. 2*1]] ++
    [2+y | isEven 2, y <- [2 .. 2*2]]
```

其中 x 分别用 1 和 2 代换。测试规则很简单

```
 [ e | True,q2,···,qk]
⇝ [ e |q2,···,qk]

 [ e | False,q2,···,qk]
⇝ []
```

所以上例结果是

```
⇝ [1+y | False, y<- [1 .. 2*1]] ++
    [2+y | True, y <- [2 .. 2*2]]
⇝ [2+y | y <- [2,3,4]]
⇝ [2+2 | ] ++ [2+3 | ] ++ [2+4 | ]
```

当不存在约束时

```
[e | ] = [ e ]
```

最后的结果是

```
 [x+y | x<- [1,2], isEven x, y <- [x .. 2*x]]
⇝ [4,5,6]
```

下面是更多的例子。

```
 triangle 3
⇝ [(x,y) | x<- [1 .. 3], y<- [1 .. x]]
⇝ [(1,y) | y <- [1 ..1]] ++
  [(2,y) | y <- [1 .. 2]] ++
  [(3,y) | y <- [1 .. 3]] ⇝ [(1,1) |]++
  [(2,1) |] ++ [(2,2) |] ++
  [(3,1) |] ++ [(3,2) |] ++ [(3,3) |]
⇝ [(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]
```

另一个例子含有下列测试

```
 [m*m | m<- [1 .. 10], m*m <50]
⇝ [1*1 | 1*1<50] ++ [2*2 |2*2 | 2*2<50] ++ ...
    [7*7 |7*7<50] ++ [8*8 | 8*8<50] ++ ...
⇝ [1 | True] ++ [4 | True] ++ ...
    [49| True] ++ [64| False] ++ ...
⇝ [1,4,...,49]
```

下面是使用列表概括的两个较大的例子。

列表置换

一个列表的置换是列表元素的重新排列。函数 perms 返回一个列表的所有置换构成的列表。

```
perms :: Eq a => [a] -> [[a]]
```

空列表的置换是它本身。如果 xs 不空，一个置换可以这样获得：从 xs 中选取一个元素 x，将 x 置于 xs\\[x] 的置换之前（运算 '\\' 返回两个列表的

差：xs\\ys 是将 ys 中的所有元素从 xs 中删除后的列表)。由此获得下列定义

```
perms []   = [[]]
perms xs   = [x:ps | x <- xs, ps <- perms (xs\\[x])]
```

例如，求单元素列表置换的计算过程

```
  perms [2]
 ⇝ [x:ps | x<- [2], ps <- perms []]
 ⇝ [x:ps | x<- [2], ps <- [[]] ]
 ⇝ [2:ps | ps <- [[]] ]
 ⇝ [2:[] | ]
 ⇝ [[2]]
```

求两个元素列表的置换过程

```
  perms [2,3]
 ⇝ [x:ps | x <- [2,3], ps <- perms([2,3]\\[x])]
 ⇝ [2:ps | ps <- perms [3] ] ++ [3:ps | ps <- perms [2] ]
 ⇝ [2:[3] ] ++ [3: [2]]
 ⇝ [[2,3],[3,2]]
```

最后，求三个元素列表的置换过程

```
  perms [1,2,3]
  [x:ps | x<- [1,2,3],ps <- perms([1,2,3]\\[x]) ]
 ⇝ [1:ps| ps <- perms [2,3]] ++ ... ++ [3:ps| ps <- perms [1,2] ]
 ⇝ [1:ps|ps <- [[2,3],[3,2]] ++ ... ++ [3:ps | ps <- [[1,2],[2,1]]]
 ⇝ [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

列表置换的另一种方法: 列表(x:xs)的置换可以通过将 x 插入到 xs 的置换的不同位置获得, 插入位置可以通过将一个列表分析成两个子列表获得。

```
perm :: [a] -> [[a]]
perm []     =  [[]]
perm (x:xs) = [ps++[x]++qs | rs <- perm xs,
                             (ps,qs) <- splits rs ]
```

要得到列表 xs 的所有折分，关键是形如 (y:ys) 列表的折分：或者在 (y:ys) 的前面折分，或者在 ys 中折分：

```
splits :: [a] -> [([a], [a])]


splits []     = [([],[])]
splits (y:ys) = ([],y:ys):[(y:ps,qs) | (ps,qs) <- splits ys]
```

注意到 perms 的类型要求 a 必须属于 Eq 类, 这是因为运算 \\定义在类型 [a] 上。但是，在 perm 上无此限制，因为我们使用了计算置换的不同方法。

向量与矩阵

本节介绍适用于各种目的的实数向量和矩阵的一种模型。一个向量是一个实数序列，如 [2.1, 3.0, 4.0]

```
type Vector = [Float]
```

两个向量的矢量积（假定它们的长度相同）是对应元素乘积之和。

```
scalarProduct [2.0,3.1] [4.1,5.0]
⇝ 2.0*4.1 + 3.1*5.0
⇝ 23.7
```

定义矢量积的第一次尝试可能是

```
mul xs ys = sum [x*y | x <- xs, y<- ys]
```

但是，由此定义得到

```
mul [2.0,3.1] [4.1,5.0]
⇝ sum [8.2,10.0,12.71,15.5]
⇝ 46.41
```

因为定义中使用了二元组的所有可能结合。为了使对应的元素相乘，我们先使用拉链函数将对应元素组成对：

```
scalarProduct :: Vector -> Vector -> Float
scalarProduct xs ys = sum [x*y | (x,y) <- zip xs ys]
```

计算前例表明这个结果是我们所需要的。（我们也可以使用 zipWith 定义scalarProduct。）

一个矩阵可以看成是一个行向量的列表，也可以看成是一个列向量的列表，如

$$\begin{pmatrix} 2.0 & 3.0 & 4.0 \\ 5.0 & 6.0 & -1.0 \end{pmatrix}$$

我们选择用行向量的列表表示矩阵，

```
type Matrix = [Vector]
```

如上面的例子可表示为

```
[[2.3,3.0,4.0],[5.0,6.0,-1.0]]
```

两个矩阵 M 和 P 的乘积可以通过计算 M 的行与 P 的列向量的矢量积求得。

$$\begin{pmatrix} 2.0 & 3.0 & 4.0 \\ 5.0 & 6.0 & -1.0 \end{pmatrix} \times \begin{pmatrix} 1.0 & 0.0 \\ 1.0 & 1.0 \\ 0.0 & -1.0 \end{pmatrix} = \begin{pmatrix} 5.0 & -1.0 \\ 11.0 & 7.0 \end{pmatrix}$$

为此，我们定义

```
matrixProduct :: Matrix -> Matrix -> Matrix
matrixProduct m p
  = [ [ scalarProduct r c | c <- column p] | r <- m]
```

其中函数 colum 将矩阵表示为列向量的列表:

```
column :: Matrix -> Matrix
column y = [ [x!!j | z <- y] | j <- [0 .. s] ]
           where
           s = length (head y) - 1
```

表达式[z!!j | z <- y]在 y 的每一行选取第 j 个元素; 结果正是 y 的第 j 列。length (head y) 是 y 中一行的长度, 所以下标 j 的取值范围是 0 至 s = length (head y)-1。columns 函数的另一个版本是库 List.Hs 中的 transpose。

生成器中的假模式

有些模式是假的, 即匹配可能失败。如果在 '<-' 左边使用了假模式, 其作用是只过滤那些匹配的元素。例如,

```
[x | (x:xs) <- [[],[2],[],[4,5]] ] ⇝ [2,4]
```

包含假模式的生成器的规则与上面的解释类似, 只是在进行模式代换前, 列表的元素要用模式过滤。细节见习题。

习题

17.1 计算下列表达式

```
[x+y | x <- [1 .. 4], y <- [2 .. 4], x>y]
```

17.2 使用列表概括定义下列返回一个列表的所有子列表和所有子序列的函数

```
subList, subSequences :: [a] -> [[a]]
```

一个列表的子列表是删除某些元素后的列表; 一个列表的子序列是列表中连续的一段。例如, [2,4]和[3,4]是列表[2,3,4]的子列表, 但是, 只有[3,4]是[2,3,4]的子序列。

17.3 计算下列表达式

```
perm [2]
perm [2,3]
perm [1,2,3]
matrixProduct [[2.0,3.0,4.0],[5.0,6.0,-1.0]]
              [[1.0,0.0],[1.0,1.0],[0.0,-1.0]]
```

17.4 使用zipWith定义scalarProduct。

17.5 定义求方阵行列式的函数; 如果行列式不等于零, 定义求行列式逆的函数。

17.6 计算列表概括的规则可以按照分情况[]和(x:xs)重新定义。完成下列式子, 给出这两个规则

```
[e | v <- [] , q2, ..., qk] ⇝ ...
[e | v <- (x:xs) , q2, ..., qk] ⇝ ...
```

17.7 给出计算生成器的精确规则，其中生成器包含一个假模式，如(x:xs) <- lExp。你可能需要定义辅助函数。

17.8 使用下列式子可以将列表概括转换成包含map, filter和concat的式子

```
[x | x<-xs]              = xs
[f x | x<-xs]            = map f xs
[e | x<-xs, p x, ...]    = [e | x<-filter p xs, ...]
[e | x<-xs, y<-ys, ...] = concat [[e | y<-ys, ...] | x<-xs]
```

使用这些式子翻译下列表达式

```
[m*m | m<-[1 .. 10]]
[m*m | m<-[1 .. 10], m*m<50]
[x+y | x<-[1 .. 4], y<-[2 .. 4], x>y]
[x:p | x<-xs, p<-perm (xs\\[x])]
```

你可能需要定义一些辅助函数。

§17.4 数据导向编程

一个程序所处理的数据结构将按需要生成，而且有可能永远不会明确地表达出来。这使得我们可以使用 **数据导向的编程** 方式来构造和处理复杂的数据结构。例如，求 1 至 n 的 4 次方之和。数据导向的解为

- 建立数的列表 [1 .. n];
- 求每个数的 4 次幂：[1,16,...,n^4]
- 求以上列表元素之和。

这样的程序为

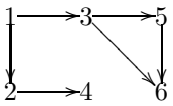
```
sumFourthPowers n = sum (map (^4) [1 .. n])
```

这个程序的计算过程是如何进行的呢？

```
  sumFourthPowers n
⇝ sum (map (^4) [1 .. n])
⇝ sum (map (^4) (1:[2 .. n]))
⇝ sum ((^4) 1 : map (^4) [2 .. n])
⇝ (1^4) + sum (map (^4) [2 .. n])
⇝ 1+ sum (map (^4) [2 .. n])
⇝ ...
⇝ 1 + (16 + sum (map (^4) [3 .. n]))
⇝ ...
⇝ 1 + (16 + (81 + ... +n^4))
```

可以看出，在这个过程中没有生成任何中间列表。当列表的头生成时，程序计算它的 4 次方，然后累加到所求的和上。

例 17.1 (求列表中的最小元素) 一个更有说服力的例子是求一个数值列表中的最小值。一种解法是对列表排序，然后取头元素。




```
graphEx = makeSet [(1,2),(1,3),(2,4),(3,5),(5,6),(3,6)]
```

在这个过程中，我们不需要将整个列表排好序，事实上，我们可以得到

```
iSort [8,6,1,7,5]
⇝ ins 8 (ins 6 (ins 1 (ins 7 (ins 5 []))))
⇝ ins 8 (ins 6 (ins 1 (ins 7 [5])))
⇝ ins 8 (ins 6 (ins 1 (5:  ins 7 [])))
⇝ ins 8 (ins 6 (1:(5:  ins 7 [])))
⇝ ins 8 (1:(ins 6 ((5:  ins 7 []))))
⇝ 1:  ins 8 (ins 6 (1:(5:  ins 7 [])))
```

从计算过程中划线部分可以看出，ins 的每次应用会计算列表一大部分的最小值，因为 ins 的结果的首元素由一步求得。在这种情况下，我们不必计算尾部即可求得到列表的头元素，因此，求最小值可以用(head.iSort)求得。

例 17.2 (求图中的路径) 一个图可看作类型Relation a（见 *16.9* 节）的一个对象。如何在图中找一个结点到另一个结点的路径呢？例如，在下图中，由 *1* 到 *4* 的路径是 *[1, 2, 4]*。

我们将解决下列问题: 求 x 到 y 的所有路径组成的列表；原问题可以用取列表的头元素来解决。注意，因为结果是列表，算法允许返回空列表，即 x 到 y 没有路径。这种适用于许多场合的方法称为 **成功的列表** 技术；算法返回的不是一个解，或者在无解的情况下返回一个错误信息，而是返回解的列表，无解时返回空列表。这种方法也允许返回多个解。下面我们会看到这样的例子。

如何解决上述问题呢？我们假定图是无环的，即不存在从一个结点到它本身的回路。

- x 到 x 的唯一路径是 [x]；
- 一个由 x 到 y 的路径的第一步是由 x 到它的邻接点 z，路径的其余部分是由 z 到 y 的路径。

为此，我们可以寻找从 x 出发经过 x 的每个邻接点 z 到达 y 的路径。

```
routes :: Ord a => Relation a -> a -> [[a]]
routes rel x y
  | x==y       = [[x]]
  | otherwise  = [x:r| z <- nbhrs rel x,
                       r <- routes rel z y]
```

函数nbhrs的定义如下

```
nbhrs :: Ord a => Relation a -> a -> [a]
nbhrs rel x = flatten (image rel x)
```

其中flatten将一个集合转换为一个列表。下面看几个例子，为了阅读的方便，我们用routes'表示routes graphEx，nbhrs'表示nbhrs graphEx。

```
routes' 1 4
⇝ [1:r | z <- nbhrs' 1, r <- routes' z 4]
⇝ [1:r | z <- [2,3], r <- routes' z 4]
⇝ [1:r | r <- routes' 2 4] ++
      [1:r | r <- routes' 3 4]            †
⇝ [1:r | r <- [2:s | w <- nbhrs' 2, s<- routes' w 4]] ++ ...
⇝ [1:r | r <- [2:s | w <- [4], s <- routes' w 4]] ++ ...
⇝ [1:r | r <- [2:s | s <- routes' 4 4 ]] ++ ...            ‡
⇝ [1:r | r <- [2:s | s <- [[4]] ]] ++ ...
⇝ [1:r | r <- [ [2,4] ]] ++ ...
⇝ [[1,2,4]] ++ ...
```

此列表的表头表示通过寻找 1 的第一个邻接点 2，以及 2 的第一个邻接点 4 即可求得 1 到 4 的路径。在这种情况，寻找路径直接给出了一个解。但是，情况并非总是这样的。例如

```
routes' 1 6 = ...
⇝ [1:r | r <- routes' 2 6] ++
    [1:r | r <- routes' 3 6]            †
⇝ ...
⇝ [1:r | r <- [3:s | s <- routes' 4 6] ] ++
[1:r | r <- routes' 3 6]            ‡
```

计算中的对应点标以 † 和 ‡。寻找由 4 到 6 的路径将失败，因为 4 没有邻接点，因此

```
⇝ [] ++ [1:r | r <- routes' 3 6] = ...
⇝ [1:r | r <- [3:s | s <- routes' 5 6] ] ++ ...
⇝ [[1,3,5,6]] ++ ...
```

这个算法的作用是当寻找失败时回溯: 不存在由 1 经过 2 到达 6 的路径，所以我们检查了经过 3 的可能路径。但是，只有在第一种可能性穷尽的情况下才进行回塑，所以，惰性计算确保这种寻找所有路径的方法也是求一条路径的一种有效方法。

我们假定图是无环的，所以寻找路径不会陷入循环。我们可以对算法稍加修改，使其在寻找路径时不会形成环，因此算法可用于任意图。方法是给函数增加一个列表参数，表示那些不被重复访问的结点，这样便得到下列算法

```
routesC :: Ord a => Relation a -> a -> a -> [a] -> [[a]]
routesC rel x y avoid
  | x==y      = [[x]]
  | otherwise = [x:r | z <- nbhrs rel x \\ avoid,
                       r <- routesC rel z y (x:avoid) ]
```

在递归调用时我们做了两处修改。

- 在寻找x的邻接点时，只考虑那些不属于列表avoid的结点；
- 在寻找x到y的路径时，排除访问列表avoid中的元素和结点x本身。

在图rel中寻找从x到y的路径可以表示为routesC rel x y []。

习题

17.9 定义graphEx2为

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

试计算routes graphEx2 1 4。使用下列定义重新计算上式

```
routes rel x y
  | x==y       = [[x]]
  | otherwise  = [x:r | z <- nbhrs rel x,
                        r <- routes rel z y,
                        not (elem x z) ]
```

试解释为什么这个定义不适用于带圈的图。最后，计算routesC graphEx 1 4 []。

§17.5 实例：分析表达式

我们在 14.2 节定义了算术表达式的类型Expr，后来在第 222 页对此进行了修改

```
data Expr = Lit Int | Var Var | Op Ops Expr Expr
data Ops  = Add | Sub | Mul | Div | Mod
```

并且说明如何使用函数 eval 计算这些表达式的值。第十六章讨论了如何用抽象数据类型 Store 表示变量所代表的值。使用这些成份可以构造一个简单的算术表达式计算器。但是，输入一个表达式是很困难，因为我们必须输入类型 Expr 的元素，例如，2 和 3 的和必须用下式表示

```
Op Add (Lit 2) (Lit 3)                    (exp)
```

要使得输入更合理，我们需要定义函数 show 的逆：它将把文本行"(2+3)" 转换为表达式 (exp)。

类型 Expr 的语法分析器是类型分类 Read（见 12.4 节）的函数 read 的一个实现。注意，Expr 的导出函数 read 只能读取形如"Op Add (Lit 2)(Lit)"的串，而不是形如"(2+3)"的紧凑语法分析器输入格式。

语法分析器的类型：Parse

在构造语法分析函数库时，我们首先建立表示语法分析器的类型。语法分析的问题是从一个类型 a 的对象的列表中抽取出另一个类型 b 的对象。对

于我们的例子，输入是串，如"(2+3)"，输出为 Expr 的元素。为此，我们可能定义分析器的类型为

```
type Parse1 a b = [a] -> b
```

假定 bracket 和 number 是类型为 Parse 的分析器，并且分别识别括号和数，则有

```
bracket "(xyz"  ⇝ "("
number "234" ⇝ 2 or 23 or 234?
bracket "234" ⇝ no result?
```

一个明显的问题是一个分析器可能返回多个结果，如第二种情况，或者没有结果，如最后一种情形。为此，我们定义下列类型

```
type Parse2 a b = [a] -> [b]
```

其中返回的是结果的列表。在这样的定义下，前述分析结果成为

```
bracket "(xyz"  ⇝ ['(']
number "234" ⇝ [2, 23, 234]
bracket "234" ⇝ []
```

在这里，空列表示无结果，多个结果表示有多种可能的分析方法。事实上，我们又在使用“成功列表”技术。

上述类型仍然存在一个问题。假如我们先找一个括号，再找一个数，那么后者的输入应该是哪一部分呢？我们需要记录一次分析成功后剩余的输入。为此，我们定义下列类型

```
type Parse a b = [a] -> [(b, [a])]
```

此时，上面的例子的结果成为

```
bracket "(xyz"  ⇝ [('(', "xyz")]
number "234" ⇝ [(2,"34"), (23, "4"), (234, "")]
bracket "234" ⇝ []
```

输出列表的每个元素表示一次成功分析。例如，number "234" 有 3 个成功的分析结果。第一个结果是 2，未分析的部分为"34"。

标准引导库用于定义 Read 分类的类型`ReadS b`是`Parse a b`的特殊情形，其中 a 成为类型 Char, [a] 成为 String。

一些基本分析器

在定义分析器类型之后，我们可以定义一些基本分析器。这些基本分析器和分析器组合函数在图 17.1 列出。下面介绍这些函数。

第一个分析器总是失败，不接收任何输入，输出列表为空:

```
none :: Parse a b
none val inp = []
```

我们也可以不读取任何输入，直接以成功返回，返回的结果是函数的参数

```
succeed :: b -> Parse a b
succeed val inp = [(val,inp)]
```

```
infixr 5 >*>

type Parse a b = [a] -> [(b,[a])]

none :: Parse a b
none val inp = []

succeed :: b -> Parse a b
succeed val inp = [(val,inp)]

token :: Eq a => a -> Parse a a
token t = spot (==t)

spot :: (a -> Bool) -> Parse a a
spot p (x:xs)
  | p x       = [(x,xs)]
  | otherwise = []
spot p []     = []

alt :: Parse a b -> Parse a b -> Parse a b
alt p1 p2 = p1 inp ++ p2 inp

(>*>) :: Parse a b -> Parse a c -> Parse a (b, c)
(>*>) p1 p2 inp
  = [((y,z), rem2) | (y, rem1) <- p1 inp, (z, rem2) <- p2 rem1]

build :: Parse a b -> (b -> c) -> Parse a c
build p f inp = [ (f x, rem) | (x, rem) <- p inp]

list :: Parse a b -> Parse a [b]
list p = (succeed []) `alt`
         ((p >*> list p) `build` (uncurry (:)))
```

图 17.1: 主要语法分析函数

更常用的是能够识别一个对象或者符号的分析器:

```
token :: Eq a => a -> Parse a a
token t (x:xs)
  | t==x      = [(t,xs)]
  | otherwise = []
token t []    = []
```

更一般地，我们可以识别满足某种性质的对象，其中性质用一个布尔值函数表示：

```
spot :: (a -> Bool) -> Parse a a
spot p (x:xs)
  | p x       = [(x,xs)]
  | otherwise = []
spot p []     = []
```

我们可以利用这些分析器识别一个字符，如左括号, 或者一个数字：

```
bracket = token '('
dig     = spot isDigit
```

我们还可以利用 spot 定义 token

```
token t = spot (==t)
```

如果我们想建立更复杂结构的分析器，我们需要将这些分析器结合起来。例如，识别由多个数字组成的数。

分析器的结合

我们将建立一个高阶多态函数库，然后利用这些库函数构造表达式的分析器。首先考虑分析器的结合方式。

对于表达式一例，一个表达式或者是一个原子，或者是一个变量，或者是由运算符构成的表达式。我们希望由这三种表达式分析器构造一个表达式分析器。为此，我们使用函数 alt

```
alt :: Parse a b -> Parse a b -> Parse a b
alt p1 p2 inp = p1 inp ++ p2 inp
```

这个分析器将 p1 和 p2 分析的结果合并成一个列表，所以每个分析器的成功分析构成两者结合分析器的一个成功分析。例如

```
bracket `alt` dig "234"
⇝ [] ++ [(2, "34")]
```

其中分析器 bracket 失败，但 dig 成功，所以两者的结合分析成功。

对于第二个分析器结合函数，仍然考虑表达式例子。在识别由运算符构成的表达式时，我们先识别一个括号，然后识别一个数。如何将两个分析器结合在一起，使得第二个分析器应用于第一个分析器剩余的输入呢？

我们将这样的函数定义为一个满足结合律的中缀运算符，因为它将用于顺序连接一系列分析器

```
infixr 5 >*>

(>*>) :: Parse a b -> Parse a c -> Pare a (b,c)
(>*>) p1 p2 inp
  = [((y,z),rem2) | (y,rem1) <- p1 inp, (z,rem2) <- p2 rem1]
```

其中 (y,rem1) 遍历 p1 在 inp 上的所有可能结果。对于每个 (y, rem1), 将 p2 应用于 rem1, 即 p1 未消耗的输入。两次成功分析的结果 y 和 x 作为一个二元组返回。

例如, 假定 number 识别非空数字序列, 试计算`(number >*> bracket) "24("`的结果。将 number 应用于串`"24("`有两个结果

```
number "24(" ⇝ [(2,"4("), (24,"(")]
```

所以, (y,rem1) 有两种情况

```
(number >*> bracket) "24("
⇝ [((y,z), rem2) | (y,rem1) <- [(2,"4("),(24,"(")],
                    (z,rem2) <- bracket rem1]
⇝ [((2,z),rem2) | (z,rem2) <- bracket "4("] ++
    [((24,z),rem2) | (z,rem2) <- bracket "("]
```

因为 `bracket "4(" ⇝ []`, 故有

```
⇝ [] ++ [((24,z),rem2) | (z,rem2) <- bracket "("]
```

以及

```
bracket "(" ⇝ [('(',"")]
```

故最后结果为

```
⇝ [((24,z),rem2) | (z,rem2) <- [('(',"")] ]
⇝ [ ((24,'('), "")]
```

这表示最终有一次分析成功, 结果是一个数 24 后接一个左括号。

最后一个运算是改变一个分析器返回的结果, 或者在此结果上构造另一个对象。考虑返回一个数字列表的分析器 digList。能否返回这个列表所代表的数字呢? 我们可以在结果上应用转换函数

```
build :: Parse a b -> (b -> c) -> Parse a c

build p r inp = [(f x, rem) | (x, rem) <- p inp ]
```

例如

```
(digList `build` digsToNum) "21a3"
⇝ [ (digsToNum x, rem) | (x,rem) <- digList "21a3"]
⇝ [ (digsToNum x, rem) | (x,rem) <- [("2","1a3"), ("21","a3")]]
⇝ [ (digsToNum "2", "1a3"), (digsToNum "21","a3")]
⇝ [ (2,"1a3"), (21, "a3")]
```

使用上述三个运算或结合算子`alt`，`>*>`和 build 以及上节所介绍的基本分析器，我们可以定义所有的语法分析器。

例如，给定识别一个对象的分析器，我们来定义一个对象列表的分析器。列表分两种情形

- 列表空，此时由分析器`succeed []`完成；
- 列表不空，即由一个对象和一个对象列表构成。这样的列表可以由`p >*> list`识别；然后我们需要将结果二元组 (x, xs) 转换为列表 (x: xs)。这个任务可以将 build 应用于 (:) 的非 Curry 形式完成，即

```
list :: Parse a b -> Parse a [b]
list p = (succeed []) `alt`
          ((p >*> list p) 'build' (uncurry (:)))
```

习题

17.10 定义下列函数

```
neList   :: Parse a b -> Parse a [b]
optional :: Parse a b -> Parse a [b]
```

使得`neList p`能够识别一个对象的非空列表，其中每个对象由`p`识别；`optional p`可选择地识别一个对象，即它可以识别一个对象，也可以立即返回。

17.11 定义函数

```
nTimes :: Int -> Parse a b -> Parse a [b]
```

使得`nTimes n p`能够识别`n`个由`p`识别的对象。

表达式分析器

现在我们可以定义表达式的语法分析器。表达式有三种形式

- 原子，如`67`，`89`，其中 '~' 表示负数运算；
- 变量：'a'至'z'；
- 二元运算符的应用，包括 `+`，`*`，`-`，`/`，`%`，其中 `%` 表示 mod，`/` 表示整数除法。复合表达式必须加括号，如`(23+(34-45))`，其中不允许有空格。

语法分析器由对应于三种表达式的三个分析器构成

```
parser :: Parse Char Expr
parser = litParse `alt` varParse `alt` opExpParse
```

最简单的是

```
varParse :: Parse Char Expr
varParse = spot isVar `build` Var

isVar :: Char -> Bool
isVar x = ('a'<= x && x <= 'z')
```

(其中构造符函数 Var 用作从字符到类型 Expr 的转换函数。)

一个包含运算符的表达式由一个运算符和两个表达式构成，整个表达式介于一对括号之间：

```
opExprParse
  = (token '(' >*>
     parser    >*>
     spot isOP >*>
     parser    >*>
     token ')')
     `build` makeExpr
```

其中转换函数将一个嵌套二元组序列转换为表达式，例如

```
('(', (Lit 23, ('+', (Var 'x', ')'))))
```

被转换为表达式`Op Add (Lit 23) (Var 'x')`，故有

```
makeExpr (_, (e1, (bop, (e2, _)))) = Op (charToOp bop) e1 e2
```

函数 isOp 和 charToOp 的定义留作练习。最后，我们定义原子的分析器。一个数由一个非空数字列表构成，前面可以带有符号' '。利用前一节习题定义的函数可以定义下列分析器：

```
litParse
  = ((optional (token '~')) >*>
      (neList (spot isDigit)))
      'build' (charlistToExpr . uncurry (++))
```

将一个字符序列转换为整数原子的函数留作练习。

习题

17.12 定义分析表达式中使用的函数

```
isOp ::     Char -> Bool
charToOp :: Char -> Ops
```

17.13 定义下列函数

```
charlistToExp :: [Char] -> Expr
```

使得

```
charlistToExp "234" ⇝ Lit 234
charlistToExp " 98" ⇝ Lit (-98)
```

17.14 将一个值expr赋予变量var的计算器指令表示为

```
var:expr
```

试为这种命令设计一个分析器。

17.15 如果允许整数与分数相加，如何修改数的分析器。

17.16 如果允许多于一个字符的变量名，如何修改变量的分析器。

17.17 试说明如何修改你的分析器使其忽略表达式中的空格和制表符。(提示：一种方法是使用简单的预处理。)

17.18 (注意，本练习适用于熟悉Backus-Naur语法范式的读者。) 下列语法描述了无括号并且允许乘除法运算优先的表达式

```
Expr   ::= Int | Var | (Expr Ops Expr) |
           Lexpr Mop Mexpr | Mexpr Aop Expr
Lexpr  ::= Int | Var | (Expr Ops Expr)
Mexpr  ::= Int | Var | (Expr Ops Expr) | Lexpr Mop Mexpr
Mop    ::= '*' | '/' | '%'
Aop    ::= '+' | '-'
Ops    ::= Mop | Aop
```

试为此语法设计一个Haskell语法分析器，并讨论运算'-'的结合性。

顶层分析器

上节定义的分析器 parser 具有类型

```
[Char] -> [ (Expr, [Char])]
```

我们需要一个将串转换为它所表示的表达式的函数。为此定义下列函数

```
topLevel :: Parse a b -> [a] -> b
topLevel p inp
  = case results of
      [] -> error "parse unsuccessful"
      _  -> head results
    where
    results = [found | (found,[]) <- p inp]
```

语法分析 parse p inp 成功的条件是结果中至少包含一个元素（第二种情况），并且所有的输入已消耗完（即测试模式与(found, [])匹配）。此时，第一个值found被返回；否则输入有错误。

我们可以定义的计算器命令类型如下

```
data Command = Eval Expr | Assign Var Expr | Null
```

其目的是

- 计算表达式的值；
- 给变量赋值；
- 空操作

如果赋值命令形如 var:expr，则不难为此类型设计一个分析器

```
commandParse :: Parse Char Command
```

在今后谈到计算器时，我们假定这个分析器已有定义。

结论

分析器类型 Parse a b 的下列函数可用于直接构造递归语法分析器

```
none    :: Parse a b
succeed :: b -> Parse a b
token   :: Eq a => a -> Parse a a
spot    :: (a -> Bool) -> Parse a a
```

```
alt      :: Parse a b -> Parse a b -> Parse a b
(>*>)    :: Parse a b -> Parse a c -> Parse a (b, c)
build    :: Parse a b -> (b -> c) -> Parse a c
topLevel :: Parse a b -> [a] -> b
```

进一步对这些函数进行探讨是很有意义的。

- 类型 Parse a b 是一个函数类型，所以，所有的分析器结合函数是高阶函数。
- 因为函数是多态的，所以不需要明确输入或者输出的类型。在表达式例子中，输入限于字符串，但是输入可以是任意其他类型的符号；例如，这些符号可以是词，分析结果是句子。

  更重要的是，利用同样的结合函数可以返回任意类型的对象。在以上例子中，我们返回字符、表达式、二元组以及列表。
- 惰性计算在这里也起着重要作用。我们构造的可能分析结果是按需要在测试不同分支时生成的。这些分析器将通过不同的选择回朔，直至产生一个成功的分析。

习题

17.19 定义一个分析器以识别数字列表构成的串，如"[2,-3,45]"。

17.20 定义一个识别简单英语句子的分析器，简单句由主语、谓语和宾语组成。你需要提供一些单词，如"cat","dog"等以及识别一个串的分析器。你还需要定义一个函数

```
tokenList :: Eq a => [a] -> Parse a [a]
```

例如， `tokenList "Hello" "Hello Sailor"` $\rightsquigarrow$ `[("Hello", " Sailor")]`

17.21 定义下列函数

```
spotWhile :: (a -> Bool) -> Parse a [a]
```

其中函数参数测试输入类型元素的性质，函数返回输入元素满足性质的最长前缀。例如，

```
spotWhile digit "234abc" ⇝ [("234","abc")]
spotWhile digit "abc234" ⇝ [([],"abc234")]
```

§17.6 无穷列表

惰性计算的一个重要作用是可以用语言来描述**无穷**结构。无穷结构的完全计算需要的时间也是无穷的，但是，使用惰性计算，我们可以只计算数据结构的一部分，而非全部。任何递归类型包含无穷对象，我们在这里将主要讨论列表，因为列表是最常用的无穷结构。

本节将介绍许多例子，从最简单的单行定义到仿真实例中用到的随机数的生成。最简单的无穷列表定义是常数列表，如

```
ones = 1 : ones
```

在 Haskell 系统中计算这个列表将生成无穷个 1 的序列。在 Hugs 中可用 Ctrl-C 来中断，在 Hugs 的 Windows 界面上可按 stop 键中断。在这两个系统中都将产生下列结果

```
[1, 1, 1, ,1, 1, ,1, 1, 1, 1^C{Interrupted!}
```

我们可以计算函数应用于 ones。例如

```
addFirstTwo :: [Int] -> Int
addFirstTwo (x:y:zs) = x+y
```

将上述函数应用于 ones 得到

```
addFirstTwo ones
⇝ addFirstTwo (1:ones)
⇝ addFirstTwo (1:1:ones)
⇝ 1+1
⇝ 2
```

系统中预定义了列表`[n .. ]`, `[n,m ..]`, 例如

```
[3 .. ]   = [3,4,5,6,...
[3,5 .. ] = [3,5,7,9,...
```

我们可以自己定义这些列表函数:

```
from :: Int -> [Int]
from n   = n : from (n-1)

fromStep :: Int -> Int [Int]
fromStep n m = n : fromStep (n+m) m
```

例如,

```
fromStep 3 2
⇝ 3:fromStep 5 2
⇝ 3:5:fromStep 7 2
⇝ ...
```

这些函数也可定义在 Enum 的任意特例上，详见 Prelude.hs。

列表概括也可用于定义无穷列表。所有 Pythagorean 三元组的列表可以如下生成

```
pythagTriples =
  [(x,y,z) | x <- [2 .. ], y <- [2 .. x-1],
             z <- [2 .. y-1], x*x + y*y = z*z]
pythagTriples
=[(3,4,5),(6,8,10),(5,12,13),(9,12,15),(8,15,17),...
```

一个整数的所有幂可如下定义

```
powers :: Int -> [Int]
powers n = [n^x | x <- [0 .. ]]
```

```
2 3 4 5 6 7 8 9 10 11 12 13
2 3   5   7   9    11    13
2 3   5   7        11    13
```

图 17.2: Eratosthenes 筛法

这是引导库函数 iterate 的一个特例, iterate 生成如下无穷列表

```
[ x, f x, ...,f^n x, ...
```

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

生成素数

一个大于 1 的正整数是素数, 如果它只能被它本身和 1 整除。两千多年前产生的 Eratosthenes 筛法通过删除筛法生成的素数的所有倍数产生所有素数。筛法中留下来的数全部是素数。筛法过程见图 17.2。

我们从以 2 为首元素的列表出发。首元素为 2, 我们从列表中删除 2 的所有倍数, 结果列表的首元素 3 是素数, 因为它在 2 的筛选中没有被删除。然后在列表的尾部删除 3 的倍数, 并将此过程一直重复下去。用 Haskell 定义此函数

```
primes :: [Int]
primes         = sieve [2 .. ]
sieve (x:xs)   = x : sieve [y | y <- xs, y `mod` x >0]
```

其中我们通过计算y `mod` x是否为 0 决定 y 是否为 x 的倍数。primes 的计算过程如下

```
primes
⇝ sieve [2 ..  ]
⇝ 2 :  sieve [y | y <- [3 ..  ], y `mod` 2 >0]
⇝ 2 :  sieve (3:[y | y <- [4 ..  ], y `mod` 2 >0])
⇝ 2:3:sieve ([z| z <- [y| y<- [4 ..], y `mod` 2>0],
                    z `mod` 3 >0]
⇝ ...
⇝ a:3:sieve [z | z <- [5,7,9,...],z `mod` 3 >0]
⇝ ...
⇝ 2:3:sieve [5,7,11,...]
⇝ ...
```

我们是否可以用 primes 测试一个数是否素数呢? 如果我们计算 member primes 7, 结果为 True, 而计算 member primes 6 无结果。这是因为我们需要测试无穷多个元素来确定 6 是否在列表中。关键问题是 member 函数不能利

用 primes 是有序的性质。为此, 我们定义函数 memberOrd

```
memberOrd :: Ord a => [a] -> a -> bool
memberOrd (x:xs) n
  | x<n        = memberOrd xs n
  | x==n       = True
  | otherwise  = False
```

这里的区别在于最后一种情况：如果列表的首元素 (x) 大于我们搜索的元素 (n), 则 n 不可能是有序列表的一个元素。再次测试得

```
memberOrd [2,3,5,7,11,...]  6
⇝ memberOrd [3,5,7,11,...]  6
⇝ memberOrd [5,7,11,...]  5
⇝ [7,11,...]  6
⇝ False
```

随机数的生成

许多计算机系统需要生成一个接一个的“随机”数。下面将看到的排队问题便是一个例子。

Haskell 程序不能产生真正的随机数序列。但是, 我们可以生成自然数的**伪随机数**序列, 并且所有随机数小于某个值 modulus。这种 **线性同余** 方法从一个种子出发, 然后使用下列方法由序列的前一个值生成下一个值

```
nextRand :: Int -> Int
nextRand n = (multiplier*n + increment) `mod` modulus
```

迭代这个函数可以得到 (伪) 随机序列

```
randomSequence :: Int -> [Int]
randomSequence = iterate nextRand
```

给定下列值

```
seed       = 17489
multiplier = 25173
increment  = 1849
modulus    = 65536
```

由 randomSequene seed 生成的序列为

```
[17489,59134,9327,52468,43805,8378,...
```

序列中的数 (介于 0 和 65535 之间) 发生的频率均相同。如何使这些数介于 a 和 b 之间呢 (包括 a 和 b)? 我们需要缩小这个序列中的数, 可以用一个 map 实现

```
scaleSequence :: Int -> Int -> [Int] -> [Int]
scaleSequence s t
  = map scale
    where
```

```
scale n = n `div` denom + n
range   = t-s+1
denom   = modulus `div` range
```

原来的区域 0 至 modulus-1 被分成 range 个块，每个块有相同的长度。s 赋给第一块的值，s+1 赋给第二块，以此类推。在仿真一例中，我们将生成每个顾客所需要的服务时间。例如，假定服务时间介于 1 至 6 分钟，但是，它们发生的概率不同。

等待时间	1	2	3	4	5	6
概率	0.2	0.25	0.25	0.15	0.1	0.05

我们需要用一个函数将这样的分布转换为无穷列表转换器。一旦有了一个作用于个别值上的函数，我们便可将其映射到列表上。

列表 a 的对象的分布可以用类型`[(a, Float)]`表示，其中的数值之和为 1。在单个值上的转换函数具有类型

```
makeFunction :: [(a, Float)] -> (Float -> a)
```

函数将把 0 至 65535 间的数转换为类型 a 的元素。其基本思想是将前面的列表转换为下表

等待时间	1	2	3	...
区间始点	0	(m*0.2)+1	(m*0.45)+1	...
区间终点	m*0.2	m*0.45	m*0.7	...

其中 m 表示 modulus。函数的定义如下

```
makeFunction dist = makeFun dist 0.0

makeFun ((ob,p):dis) nLast rand
  | nNext >= rand && rand > nLast  = ob
  | otherwise
      = makefun dist nNext rand
        where
        nNext = p*fromInt modulus + nLast
```

函数 makeFun 有一个额外参数，用于表示在 0 至 madulus-1 之间目前搜索到的位置，其初值为 0。函数 fromInt 将 Int 转换为对应的 Float。

一个随机数列的转换可表示为

```
map (makeFunction dist)
```

等待时间的随机分布序列为

```
map (makeFunction dist . fromInt) (randomSequence seed)
 = [2,5,1,4,3,1,2,5,4,2,2,2,1,3,2,5,...
```

其中 6 将第一次出现在位置 35。另一个随机数生成器在库 Random.hs 中定义。

注 11 (无穷列表生成器) 列表概括pythagTriples2将生成所有的Pythagorean三

元组，但结果没有任何输出。

```
pythagTriples2 =
  = [(x,y,z) | x <- [2 ..],
               y <- [x+1 .. ],
               z <- [y+1 .. ],
               x*x + y*y = z*z ]
```

问题在于选择元素的顺序。x的第一个选择是2，y为3；此时z有无穷个选择：4、5等。因此，我们没有机会选择其它的x和y值。解决的方法有两种，首先重新定义函数，如pythagTriples，使其中只有一个无穷列表。另一种方法是定义一个由两个无穷列表生成一个二元组无穷列表的函数：

```
infiniteProduct :: [a] -> [b] -> [(a,b)]
```

这个函数的定义留作练习。使用这个函数，可以修改pythagTriples2使其生成所有的Pythagorean三元组。

习题

17.22 定义阶乘和Fibonacci数列的无穷列表

```
factorial = [1,1,2,6,24,120,720,...]
fibonacci = [0,1,1,2,3,5,8,13,...]
```

17.23 定义下列函数

```
factors :: Int -> [Int]
```

函数返回一个正整数的所有因子的列表。例如，

```
factors 12 = [1,2,3,4,6,12]
```

利用此函数（或者其他函数）定义只含有因子 *2*, *3* 和 *5* 的整数序列，此序列也称为Hamming数列

```
hamming = [1,2,3,5,6,8,9,10,12,15,...]
```

17.24 定义求所有和的函数

```
runningSums :: [Int] -> [Int]
```

对于输入 $[a_0, a_1, a_2, \cdots$，函数返回列表 $[0, a_0, a_0+a_1, a_0+a_1+a_2, \cdots$。

17.25 类似于上题，定义函数infiniteProduct，并利用它给出pythagTriples2的正确定义。

§17.7 为什么使用无穷列表

Haskell 支持无穷列表和其他无穷结构。在上一节我们看到了一些很复杂的列表，如素数列表和随机数列表。那么这些无穷列表的定义是否仅仅是兴趣所至呢？我们将说明无穷结构在函数程序设计中的两个重要意义。

首先，一个程序的无穷版本可以更抽象，因而更容易编写。如寻找使用 Eratosthenes 筛选第 n 个素数的问题。如果使用有穷列表，我们需要预先知

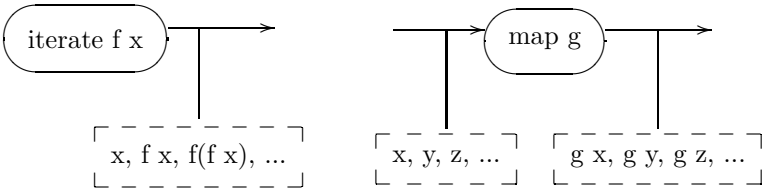


图 17.3: 一个生成器和一个转换器

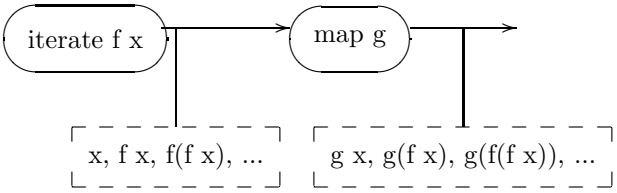


图 17.4: 连接起来的进程

道产生第 n 个素数需要检查多长的列表；如果使用无穷列表，这种顾虑是不必要的，在计算进行过程中只有列表的一部分会生成。

类似地，由 randomSequence seed 提供了随机数的无穷资源：我们可以在列表中得到任意多个随机数。(这就象计算机中的虚拟存储。通常预测一个程序所需的内存是可能的，但过于复杂；虚拟内存使得程序员不必顾虑内存问题，而去专注于其他任务。)

第二个重要意义可以从随机数的产生中看出。我们使用 iterate 生成无穷列表，然后将 map 应用于这个列表；这个过程如图 17.3 所示。

这些运算可看成一个列表生成器和一个列表转换器。虚线框里表示其值，这些成分可以链接起来构成更复杂的组合，如图 17.4。这种方法以更有趣的方式模块化了一个分布的生成。我们将生成与转换分离开来，由此可以独立地修改每个部分。认识到这种无穷列表作为进程的联接后，我们可以使用其他的结合方式，特别地，可以编写使用递归的进程式程序。

在上一节的练习中，有一个题目是求列表 [0,a_0,a_1,a_2,... 的动态累加和

$$[0, a_0, a_0 + a_1, a_0 + a_1 + a_2, ...$$

给定 a_k 之前元素的累加和，我们可以将输入 a_{k+1} 加上去，得到下一个累加和。这就象我们把和输入进程，使其与 a_{k+1} 相加。

这恰好是进程网络的效果，如图 17.5 所示，其中虚线框内的值表示沿着箭头传递的值。输出 out 的第一个值是 0，out 的其他值可以通过 iList 的下一个值与 out 的前一个和相加求得。Haskell 程序如下：函数在输入 iList 上的输出是 out，它由 0 后接 zipWith (+) 的输出构成，而 zipWith(+) 的输入为 iList 和 out。换言之，

```
listSums :: [Int] -> [Int]
```

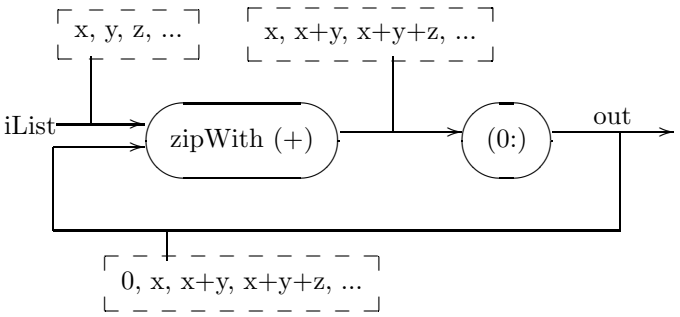


图 17.5: 一个计算列表动态累加和的进程

```
listsums iList = out
                 where
                 out = 0 : zipWith (+) iList out
```

其中 zipWith 的定义如下

```
zipWith f (x:xs) (y:ys) = f x y : zipWith f xy ys
zipWith f _      _      = []
```

部分运算 (0:) 将 0 置于一个列表的前面。下面是一个例子

```
listSums [1 ..  ]
⇝ out
⇝ 0:zipWith (+) [1 ..  ]  out
⇝ zipWith (+) [1 ..  ]  (0:...)                    (1)
⇝ 0:1+0:zipWith (+) [2 ..  ]  (1+0:...)              (2)
⇝ 1:2+1:zipWith (+) [3 ..  ]  (2+1:...) ⇝ ...
```

在计算过程中，我们用不完全列表 (0:...) 代替换了 (1) 行中的 out。类似地，在 (2) 中我们用 (1+0: ⋯) 代替 out 的尾。函数 listSums 是一个更通用函数 scanl1' 的特例

```
scanl1' :: (a -> b -> b) -> b -> [a] -> [b]
scanl1' f st iList
  = out
    where
    out = st:zipWith f iList out
```

函数 listSums 可表示为 scanll' (+) 0，动态地对一个列表的首部进行排序的函数是`sorts = scanll' ins []`，其中 ins 在列表的适当位置插入一个元素。阶乘的列表由`scanll' (*) 1 [1 ..  ]`求得，而且利用这种模式，任何原始递归函数可以用类似的方式描述。

上述定义与引导库函数 scanl 略有不同。我们选择这个定义是为了使它更接近于图 17.5 中动态的进程网络。

习题

17.26 利用scanl1'给出列表 [2^n | n <- [0 ..]] 的进程网络定义。(提示：参照阶乘一例。)

17.27 如何在无穷列表中选取某些元素？例如，如何计算一个列表中正整数的所有和？

17.28 如何合并两个有序的无穷列表？如何在结果中删除重复元素？例如，合并 *2* 的所有幂构成的列表和 *3* 的所有幂构成的列表。

17.29 利用进程网络定义Fibonacci数列和Hamming数列(参见第 *319* 页)。你也许会发现合并函数有助于解决后一个问题。

§17.8 实例：仿真

现在我们将把有关队列仿真的各部分组织起来

- 在 14.5 节，我们设计了代数类型 Inmess 和 Outmess
- 在 16.6 节，我们引进了抽象类型 QueueState 和 ServerState
- 在 17.6 节，我们说明如何根据 1 和 6 之间的时间分布生成等待时间的无穷伪随机数列表。

如 14.5 节所述，仿真的顶层是一个由一系列输入信息到一系列输出信息的函数，故有

```
doSimulation :: ServerState -> [Inmess] -> [Outmess]
```

其中第一个参数是仿真开始时服务器的状态。在 16.5 节我们定义了完成一步仿真的函数

```
simulationStep :: ServerState ->
                  Inmess ->
                  (SeverState, [Outmess])
```

对于当前的服务器状态，当前 (分钟) 到达的输入信息，函数输出一分钟处理后的状态，以及在这一分钟输出的信息列表。(在同一时刻，每个队列都有可能释放一个客户，正如同一时刻可能没有任何一个队列释放客户。)

仿真的输出将由第一分钟产生的输出信息，以及仿真在新的状态上继续产生的输出信息构成：

```
doSimulation servSt (im:messes)
  = outmess ++ doSimulation servStNext messes
    where
    (servStNext, outmess) = simulationStep servSt im
```

如何生成一个输入序列呢？由 17.6 节我们可以得到时间序列

```
    randomTimes
      = map (makeFunction dist, fromInt) (randomSequenceseed)
       ⇝ [2,5,1,4,3,1,2,5,...
```

因为每分钟有一个客户到达，所以我们生成的输入信息为

```
simulationInput
   = zipWith Yes [1 .. ] randomTimes
    ⇝ [Yes 1 2, Yes 2 5, Yes 3 1, Yes 4 4, Yes 5 3, ...
```

在这样的输入上运行 4 个队列（设 numQueue 为 4）的输出是什么呢？输出将如下开始

```
doSimulation serverStart simulationInput
⇝ [Discharge 1 0 2, Discharge 3 0 1, Discharge 6 0 1,
    Discharge 2 0 5, Discharge 5 0 3, Discharge 4 0 4,
    Discharge 7 2 2, ...
```

前 6 个输入被无延迟处理，但第 7 个需要等待 2 个时间单位。

由 SimulationInput 表示的无穷输入显然会生成无穷个输出信息。我们可以取输入的一个近似值

```
simulationInput2 = take 50 simulationInput ++ noes
noes = No : noes
```

在前 50 分钟，每分钟有一个顾客到达，之后再无顾客到达。这样将生成 50 条输出信息，其定义为

```
take 50 (doSimulation serverStart simulationInput2)
```

实验

我们现在可以对不同的数据进行实验，如不同的分布和队列数目。对于一个有穷的 Outmess，总的等待时间为

```
totalWait :: [Outmess] -> Int
totalWait = sum . map waitTime
            where
            waitTime (Discharge _ w _) = w
```

对于 simulationInput2 总的等待时间是 29，对于 3 个队列等待时间可升至 287，对于 5 个队列可降为 0。请读者实验 16.5 节中的 round robin 仿真练习

一个更大规模的项目是多个雇员服务单个队列顾客。一种方法是在`serverState`上增加一个输入队列，此队列将为其他队列提供输入；当一个小队列空时，一个元素离开输入队列。这样的模式应当避免选择队列错误造成的不必要等待时间。仿真表明这种策略将减少等待时间，但是，如果服务时间较短，减少的等待时间没有我们所想象的多。

§17.9 再谈证明

在总结了惰性计算对 Haskell 类型的影响后，我们来探讨它对程序推理的影响。把列表作为一个代表性例子，我们来说明如何证明无穷列表的性质

和所有列表的性质，而不仅仅是我们在第 8，10 和 14 章看到的，有穷列表的性质。

本节不是验证的完整介绍，我们将在最后提出一些进一步阅读资料。

无定义

几乎在每个程序设计语言中都可以编写一个不停机的程序，Haskell 也不例外。我们称这种程序的值无定义，因为程序的计算无结果。最简单的无定义结果的表达式是

```
undef :: a
undef = undef                          (undef.1)
```

它给出所有类型的一个非终止值或无定义值，当然，我们也可能不经意写下一个无定义的程序。例如，在定义阶乘时写下

```
fak n = (n+1) * fak n
```

fak n 的值与 undef 相同，因为它们都不能终止。

注意术语“无定义”有两个不同的含义。命名**undef**由 (undef.1) 定义；它的**值**是无定义值，它表示一个非终止计算的结果（因此这个结果不能定义）。

这些无定义值的存在对列表类型有影响。例如，我们可以定义列表

```
list1 = 2:3:undef
```

此列表有一个合适定义的头 2 和尾`3:undef`。类似地，这个尾有头元素 3，但其尾部无定义。因此，列表 [Int] 类型包含了如 list1 这样的**不完全**列表。即列表的部分是有定义的，部分无定义。

当然，因为存在无定义的整数，所以列表 [Int] 也包含下面的列表：

```
list2 = undef:[2,3]
list3 = undef:4:undef
```

其中包含了无定义的值，因此也是不完全列表。注意，在 list3 中，第一个 undef 的类型是 Int，第二个 undef 的类型为 [Int]。

将一个函数应用于 undef 的结果是什么呢？根据计算规则，常数函数满足

```
const 17 undef ⇝ 17
```

如果应用于 undef 的函数需要进行模式匹配，则结果也是 undef，因为模式匹配需要检查 undef 的结构，这个过程永远不会终止。例如，对于第八章使用的函数

```
sum undef         ⇝ undef         (sum.u)
doubleAll undef   ⇝ undef   (doubleAll.u)
```

在前面书写证明时，我们曾说明，证明只适用于有定义的情形。

一个整数是有定义的，如果它不等于 undef；一个列表是有定义的，如果它是有穷个有定义值的列表。利用这种方式不难说明任意代数类型的值是否有定义。

一个有穷列表可能包含无定义值。注意到以前的证明中，我们只说明结论对于有定义的列表成立，即对有定义值的有穷列表成立。

再谈列表归纳

如上所述，因为在每个列表类型中存在一个未定义的列表 undef，所以归纳原理的归纳基包含两种情况。

结构归纳证明

为了证明性质 P(xs) 对于所有的有穷列表和部分列表 (统称 fp 列表) 成立，我们需要证明下列三点：

归纳基 证明 P([]) 和 P(undef) 成立

归纳步 假定 P(xs) 成立，证明 P(x:xs) 成立

在第八章，我们证明了对于任意有穷列表 xs,ys 和 zs 下列等式成立：

```
sum (doubleAll xs) = 2* sum xs                    (sum-double)
xs ++ (ys ++ zs)   = (xs ++ ys) ++ zs               (assoc++)
reverse (xs ++ ys) = reverse ys ++ reverse undef   (reverse++)
```

欲证明上述等式对于所有的 fp 列表成立，我们需要证明下列等式

```
sum (doubleAll undef)    = 2* sum undef             (sum-double.u)
undef ++ (ys ++ zs)      = (undef ++ ys) ++ zs        (assoc++.u)
reverse (undef ++ ys) = reverse ys ++ reverse undef  (reverse++.u)
```

并且归纳步对于所有的 fp 列表成立。由 (sum.u) 和 (doubleAll.u) 可以推断等式 (sum-double.u) 成立，所以，(sum-double) 对于所有的 fp 列表成立。类似地，我们可以证明 (assoc++.u)。更有趣的是等式 (reverse++.u) 的证明。回顾 reverse 的定义

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

因为定义需要在第一个参数上进行模式匹配，所以当参数是无定义值时，结果也是无定义的：

```
reverse undef = undef
```

在 (reverse++.u) 中令 ys 为一个定义的列表，如 [2,3]，则有

```
reverse (undef ++ [2,3])
  = reverse undef
  = undef


reverse [2,3] ++ reverse undef
  = [3,2] ++ undef
```

这已经说明 (reverse++.u) 不成立，所以，我们不能证明 (reverse++) 对于所有的 fp 列表成立。事实上，上例说明 (reverse++) 不成立。

无穷列表

除 fp 列表之外，列表类型包含无穷成员。如何证明无穷列表的性质呢？打印无穷列表计算结果的讨论给予我们一些启示。经过一段时间后，我们通常使用 Ctrl-C 中断无穷列表的计算，我们可以把屏幕上输出的列表视为无穷列表的近似值。

如果我们看到的结果是 $a_0, a_1, a_2, \cdots, a_n$，我们可以认为近似值是列表

$$a_0, a_1, a_2, \cdots, a_n : \texttt{undef}$$

因为我们没有关于 a_n 之后的元素的信息。

更严格地说，下列部分列表是无穷列表 $a_0, a_1, a_2, \cdots, a_n, \cdots]$ 的逼近：

$$\texttt{undef}, a_0 : \texttt{undef}, a_0, a_1 : \texttt{undef}, a_0, a_1, a_2 : \texttt{undef},$$

如果两个列表 xs 和 ys 的所有等长近似值均相等，即对于所有的自然数 n, `take n xs = take n ys`，则称 xs 和 ys 相等。

无穷列表的相等

一个表是无穷的，如果对于所有的自然数 n，`take n xs` $\neq$ `take (n+1) xs`。两个无穷列表 xs 和 ys 相等，如果对于所有的自然数 n，`xs!!n = ys!!n`。

举例：两个阶乘列表

本例来自 17.7 节基于进程的程序。设 fac 是如下定义的阶乘函数

```
fac :: Int -> Int
fac 0 = 1                                    (fac.1)
fac n = n * fac (n-1)                        (fac.2)
```

定义阶乘无穷列表的一种方法是

```
facMap = map fac [0 .. ]                     (facMap.1)
```

另一个基于进程的解是

```
facs = 1 : zipWith (*) [1 .. ] facs          (facs.1)
```

假定这些列表是无穷的（显然是的），我们必须证明对于所有的自然数 n

```
facMap!!n = facs!!n                          (facMap.!!)
```

证明 我们假定对于所有的自然数 n

```
(map f xs)!!n        = f (xs!!n)                 (map.!!)
(zipWith g xs ys)!!n = g (xs!!n) (ys!!n)      (zipWith.!!)
```

稍后我们再证明上述等式成立。

我们用归纳法证明 (facMap.!!)：先直接证明等式对于 0 成立，然后假定等式对于 (n-1) 成立，证明等式对于 n 也成立。

归纳基 我们先证明 n 为 0 的情况。检查等式的左边

```
facMap!!0
  = (map fac [0 .. ])!!0                    by (facMap.1)
  = fac ([0 .. ]!!0)                        by (map.!!)
  = fac 0                                 by def of [0 .. ],!!
  = 1                                       by (fac.1)
```

再检查等式的右边

```
fac!!0
  = (1 : zipWith (*) [1.. ] facs)!!0        by (facs.1)
  = 1                                       by def of !!
```

故归纳基成立。

归纳步 假定下面的归纳假设然后证明 (facMap.!!) 成立

```
facMap!!(n-1) = facs!!(n-1)                 (hyp)
```

计算 (facMap.!!) 的左边

```
facMap!!n
  = (map fac [0 ..])!!n             by (facMap.1)
  = fac ([0 .. ]!!n)                by (map.!!)
  = fac n                           by defn of [0 ..], !!
  = n * fac (n-1)                   by (fac.2)
```

不难看出, `facMap!!(n-1) = fac (n-1)`, 故有

```
= n * (facMap!!(n-1))
```

再计算 (facMap.!!) 的右边

```
facs!!n
  = (1:zipWith (*) [1 .. ] facs)!!n         by (facs.1)
  = (zipWith (*) [1 .. ] facs)!!(n-1)       by def of !!
  = (*) ([1 .. ]!!(n-1))(facs!!(n-1))       by (zipWith.!!)
  = ([1 .. ]!!(n-1))*(facs!!(n-1))          by def of (*)
  = n * (facs!!(n-1))                       by defn of [1 .. ], !!
  = n * (facMap!!(n-1))                     by (hyp)
```

最后使用归纳假设完成证明。 □

无穷列表的证明

什么样的结论对于所有 fp 列表成立时对于所有列表也成立? 如果一个结论对于所有 fp 列表成立, 则它对无穷列表的所有逼近也成立。对于某些性质, 由性质对于所有逼近成立足以推出它对于所有无穷列表成立。特别地, 等式便是这样的性质。例如, 我们可以推断, 对于所有列表 xs:

```
(map f . map g) xs = map (f . g) xs
```

因此, 由函数的外延原则

```
map f . map g = map (f . g)
```

我们证明过的许多对于有穷列表成立的等式可以推广到 fp 列表，因此可以推广到所有列表。本节的习题中列出了一些这样的等式。

进一步的读物

上面所述的技术只是无穷列表和无穷数据结构上证明的简单介绍。我们在这里不能给出更深入的全面介绍，请读者参考 Paulson (1987)。阶乘列表例子的另一种证明方法参见 Thompson (1999)，其中给出了函数程序证明的概述。

习题

17.30 证明对于任意fp列表ys和zs

```
undef ++ (ys ++ zs) = (undef ++ ys) ++ zs
```

从而推出结合律在所有的列表上成立。

17.31 如果rev xs定义为shunt xs []*(见 8.7 节)*，证明

```
rev (rev undef) = undef                    (rev-rev.1)
```

在第八章我们证明了对于任意的有穷列表xs下列等式成立

```
rev (rev xs) = xs                          (rev-rev.2)
```

为什么我们不能由(rev-rev.1)和(rev-rev.2)推出等式rev (rev xs) = xs对于所有的fp列表成立。

17.32 证明对于任意自然数m，n和函数f :: Int -> a，下列等式成立

```
(map f [m .. ])!!n = f (m+n)
```

提示：在归纳证明中需要选择合适的变量。

17.33 证明下列列表是无穷的

```
facMap = map fac [0 .. ]
facs = 1 : zipWith (*) [1 .. ] facs
```

17.34 如果我们定义下列索引函数

```
(x:_)!!0  = x
(_:xs)!!n = xs!!(n-1)
[]!!n     = error "Indexing"
```

证明对于所有的函数f，fp列表xs和自然数n

```
(map f xs)!! n = f (xs!!n)
```

从而推断上述结论对于所有列表成立。对于函数zipWith证明类似的结果。

17.35 证明下列函数等式成立

```
filter p . map f     = map f . filter (p . f)
filter p . filter q  = filter (q &&& p)
concat . map (map f) = map f . concat
```

其中算子 &&& 的定义如下

```
(q &&& p) x = q x && p x
```

小结

Haskell 表达式的惰性计算意味着，我们可以用一种独特的方式编写程序。在程序运行中建立的数据结构只有在需要的时候生成，如求 4 次幂的和的例子。在求图中通路的例子中，我们只需考虑图中通路穿过的部分。在这些例子中以及许多其他情况下，惰性计算的优点使得程序的目的更明了，运行更有效。

我们再次讨论了列表概括语法，它使得列表处理程序更容易表达；在求路径和语法分析例子中我们看到了这个优点。

本章探讨的设计原则涉及惰性列表的应用：如果一个函数可能返回多个结果，我们可以用列表示之。使用惰性计算，这些结果只有在需要时一个个生成。同时，我们可以用空列表 [] 表示无结果。“成功的列表”方法可用于许多场合。

在探讨上述原则、高阶函数、多态和列表概括的过程中，我们给出了一个语法分析函数库，并将其应用于算术表达式类型 Expr。这表明现代函数程序设计语言的优势之一是，它尤其适应于这种类型的通用库的构建。

本章探讨了无穷列表的多个目的：

- 在求素数和随机数无穷列表中，我们提供了一个无限的资源：在构造程序时不必预先知道我们需要多少资源；这种抽象使得程序设计更简单明了。
- 无穷列表在函数程序设计中提供了一种基于进程程序设计的机制。

本章最后讨论了如何证明不完全列表和无穷列表的性质：我们对两种情况均给出了原则、例子以及反例。

第十八章　交互式程序设计

目前为止，本书所写的程序都是自成一体的。但是，大部分大型程序都与“外部世界”有交互作用。这种交互作用的形式有多种。

- 一个程序可能从一个终端读数据，也可能把数据写到终端，如 Hugs 解释器本身；
- 一个邮件系统可以读文件和写文件，还可以读写标准的终端设备；
- 一个操作系统在并行运行其它程序的同时，还控制着如打印机，CD-ROM 和终端等设备。

本章讨论如何在 Haskell 中开发最简单的读写终端的程序。我们描述的模式将成为更复杂的交互程序的基础，如邮件系统和操作系统。

我们先介绍在过去为什么对于函数程序员 I/O 曾经是一个问题。Haskell 的解决方案是引进类型 `IO a`，我们可以将它的成员看成这样的**程序**：做一些输入/输出，然后返回一个类型 a 的值。这些程序包括简单的运算，如读写信息，以及利用 do 结构把一些 IO 程序顺序连接成一个复杂程序。

我们将介绍一些交互式程序例子，其中包括一个交互式计算器，并讨论标准引导库和函数库中更一般的 I/O 程序。类型`IO a`的顺序特性不仅仅是 I/O 所特有的。在本章的第二部分，我们介绍更一般的 monad，IO 类型只是 monad 的一个特例，其他例子包括副作用，错误处理和非确定性。

我们将看到，monad 不仅提供了函数世界和命令式世界的界面，而且提供了在函数程序中处理这些作用的有力组织机制。这一点可以通过一个例子说明：如果我们使用 monad 方式编程，那么树上的两个不同程序将具有相同的顶层结构。

§18.1　为什么 I/O 是一个问题

一个函数程序由一些定义构成，如

```
val :: Int
val = 42


function :: Int -> Int
function n = val + n
```

这些定义的作用是给每个名一个确定的值；在上例中，val 的值是一个整数，function 的值是一个由整数到整数的函数。那么一个输入或者输出动作如何适应这种模型呢？一种方法（例如，标准 ML（Milner et al-1997））是使用下面的运算

```
inputInt :: Int
```

其作用是从输入读一个整数，所读入的值成为 inputInt 的值。每次计算`inputInt`时，它都获得一个新的值，所以，根据我们所述的模型，inputInt 并不是一个固定

的整数。在我们的语言中使用这样的运算，似乎并不会引起太大的问题，但是，下面的例子说明，它对我们的函数程序模型有两个重要的影响:

```
inputDiff = inputInt - inputInt                    (inputDiff.1)
```

- 假设第一个输入是 4，第二个是 3。对于 (-) 的参数的不同计算次序，`inputDiff`将有不同的结果：1 或者 -1；
- 更为严重的是，(inputDiff.1) 破坏了我们的推理模型。在我们的推理模型中，我们期望一个值减去它自身的结果是 0，但在这里情况则不一样。其原因恰恰是，一个表达式的意义不再仅仅由构成它的组成部分决定，因为我们不知道 inputInt 在一个程序中出现的位置，便不能确定它的意义；如在上例中，inputInt 在 inputDiff 中的第一个出现和第二出现有不同的值。

如第二点所述，如果我们采用这样的方法，那么理解一个程序会变得更加困难。这是因为程序中的任何定义都可能受到 I/O 运算的影响。一个例子是函数

```
funny :: Int -> Int
funny n = inputInt + n
```

由定义可看出它对 I/O 的依赖，但是，任何函数都可能受到类似的潜在影响。

正因为如此，I/O 相当长一段时间对于函数程序员是个棘手的问题，而且人们在寻找 I/O 的正确模型方面做过多种尝试。事实上，早期的 Haskell 也做过两次尝试。有关函数式 I/O 的历史和概述参见 Gordon (1994)。

本章描述 monadic 方法。这种方法被证明为可扩展到“外部世界”的其他交互方式的健壮模型。Monadic I/O 的基本思想是，控制构造具有 I/O 操作的程序的方式，特别是限制 I/O 运算影响函数的方式。这将是下一节的内容。

§18.2 输入/输出

在考虑**输入/输出**或者**I/O**时，最好将它们看成**顺序发生的动作**。例如，先读一些输入，然后在此基础上再读一些输入，或者产生输出。

Haskell 提供类型 `IO a`，其成员称为**类型 a 的 I/O 动作**或者**类型为 a 的 I/O 程序**。一个属于`IO a`的对象是一个程序：它将做一些 I/O，然后返回类型 a 的值。Haskell 包含一些原始 I/O 程序，以及顺序连接 I/O 程序的机制。一种理解 `IO a`类型的方法是：它们在 Haskell 之上提供了一种书写 I/O 程序的命令式程序语言，而且没有破坏 Haskell 函数模型。理解`IO a`工作原理的最好方式是阅读标准引导库中 `IO a`的代表性例子。在下一节我们介绍如何使用 do 记法把这些程序组织成更复杂的 I/O 程序。

读输入

从标准输入读一行文本的操作需要进行 I/O 操作，然后返回刚刚读入的

文本行。根据以上解释，这种操作应该是类型 IO String 的对象。Haskell 的预定义函数 getLine 从标准输入读入一行

```
getLine :: IO String
```

类似地，getChar 从标准输入读入一个字符

```
getChar :: IO Char
```

包含一个元素的类型

Haskell 包含类型 ()，其中只包含一个元素，也记作 `()`。这个类型的值并不传递有用信息，所以这样的类型并不常用。但是，在进行 I/O 时它是有用的，因为有些 IO 程序的意义在于它们完成的 I/O 动作，而不是它们返回的结果。这种 IO 程序的类型将具有类型

```
IO ()
```

而且将返回值 `()`作为其结果。

输出串

输出串`"Hello, World!"`的操作将完成某些 I/O，但是并不需要返回任何值。因此，它的类型为`IO ()`。一般的输出文本操作是一个函数，其参数是输出的串，结果是完成输出串的 I/O 对象：

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

用这个函数可以编写显示`"Hello, World"`程序。

```
helloWorld :: IO ()
helloWorld = putStr "Hello, World!"
```

我们可以使用 putStr 定义输出一行字符串的函数

```
putStrLn :: String -> IO ()
putStrLn = putStr . (++"\n")
```

这个函数的作用是在传给 putStr 的参数的尾部加上换行符。

输出其他值

Haskell 引导库提供了 Show 类，其中的函数 show 可用于显示许多类型的值

```
show :: Show a => a -> String
```

例如，我们可以定义标准引导库的一般输出函数

```
print :: Show a => a -> IO ()
print = putStrLn . show
```

返回一个值：return

假如我们想写一个 I/O 动作，它不进行任何 I/O 操作，但是返回一个值，今后我们会看到这种程序的例子。这种操作可以用预定义函数 return 实现

```
return :: a -> IO a
```

`return x`的作用是，不做任何 I/O，只返回值 x。

运行 I/O 程序

我们已经编写了一个简单的 I/O 程序，即 helloWold，那么如何运行这个程序呢？我们可以在 Hugs 中计算它：

```
Main> helloWorld
Hello, World!
Main>...
```

严格地说，Haskell 程序的主函数`main`的定义应该具有类型 `IO a`，`a`是一个类型。在 Hugs 中，当我们计算类型 b 的表达式 e 时，其结果被`print`转变为一个`IO ()`对象输出。

以上是关于引导库中的基本 I/O 函数以及它们如何运行的简介。下一节我们讨论如何将这些程序顺序运行，以及如何使用输入程序 (如 getLine 等) 读入的值。

§18.3 do 记法

do 记法是一种很灵活的机制，它可用于

- 顺序连接 I/O 程序；
- “捕捉”IO 动作返回的值，并将其传给随后的程序。

以上思想使 do 记法看上去象一个包含一系列命令和赋值的命令式程序，尽管这种类比不完全。我们将在下一节看到，由 IO 类型建立的 I/O 模型是我们所熟知的，只是其表示方式不同。

顺序连接 I/O 动作

do 记法的一个目的是顺序连接 I/O 动作，我们将通过一些例子说明如何使用 do 记法。

例 18.1 首先看引导库中putStrLn的定义。putStrLn str的作用是：先将串str输出，然后输出一个换行符。这两个动作可以实现如下：

```
putStrLn :: String -> IO ()
putstrLn str = do putStr str
                  putStr "\n"
```

由此看出，do的作用是将一些 *IO* 动作顺序连接成一个动作。do的语法遵循边界规则。do可以有任意多个参数或动作。以下是更多的例子。

例 18.2 我们可以用一个 *I/O* 程序将一个串输出 *4* 次。程序的第一个版本是

```
put4times:: String -> IO ()
put4times str
   = do putStrLn str
        putStrLn str
```

```
        putStrLn str
        putStrLn str
```

例 18.3 我们可以将输出次数作为参数，而不必将一个动作书写多次：

```
putNtimes :: Int -> String -> IO ()
putNtimes n str
  = if n <= 1
       then putStrLn str
       else do putStrLn str
               putNtimes (n-1) str
```

利用这个定义，前一个例子可写作

```
put4times = putNtimes 4
```

例 18.4 目前我们只看到输出程序，我们也可以用输入作为动作序列的一部分。例如，我们可以读入两行输入，然后输出"Two lines read." *:*

```
read2times :: IO ()
read2times
  = do getLine
       getLine
       putStrLn "Two lines read."
```

类似于上例，不难编写一个输入任意行的程序。

捕捉所读的值

如 18.1 节所述，在处理输入的方式上必须格外小心。运算inputInt :: Int 功能太强，不适合函数程序模型。但是，我们需要一种处理输入值的机制，这便是 do 记法的第二个目的；一个输入的结果只能用于 do 表达式内，这种限制避免了 I/O 动作“污染”整个程序。以下例子将说明 do 在这方面的用法。

例 18.5 前面的例子读入两行，但对于getLine的结果未做任何事情。如何在 *I/O* 程序中使用读入的这些行呢？在一个do程序中，我们可以命名类型为 IO a动作的结果。下列程序读一行，然后输出所读入的行

```
getNput :: IO ()
getNput = do line <- getLine
             putStrLn line
```

其中'line <-'用于命名getLine的结果。如果你熟悉命令式程序设计，你可以将这样的动作看作给一个变量赋值*:*

```
line := getLine
```

但是，要记住 *Haskell* 程序中的名和命令式程序中的变量间有重大区别。基本的区别是，每个'var <-'创建一个新变量var，所以 *Haskell* 允许“**单一**赋值”，而不是像大多数现代命令式语言中的”可修改赋值”。我们将在 *18.4* 节

进一步讨论它们的区别。

例 18.6 我们不一定原封不动地输出读入的行。例如，可以定义下面的程序

```
reverse2lines :: IO ()
reverse2lines
   = do line1 <- getLine
        line2 <- getLine
        putStrLn (reverse line2)
        putStrLn (reverse line1)
```

在上例中，程序读入两行，然后将它们逆转，并接相反顺序输出。

do 表达式中的局部定义

var <-getLine命名 getLine 的结果，故其作用类似于一个定义。前一程序可以如下书写。

例 18.7 上例可以使用局部变量编写

```
reverse2lines :: IO ()
reverse2lines
   = do line1 <- getLine
        line2 <- getLine
        let rev1 = reverse line1
        let rev2 = reverse line2
        putStrLn rev2
        putStrLn rev1
```

输入其他值

Haskell 定义了一个 Read 类，其中包含函数

```
read :: Read a => String -> a
```

这个函数可用于将一个串转换为它所表达的某个类型的值。

例 18.8 假设我们要编写一个读入一个整数的程序。要从一行输入读一个整数，可以使用

```
do line <- getLine
```

然后我们需要将此动作与一个 *I/O* 动作顺序连接，将line解释为整数返回。我们可以用下式将line转换为整数

```
read line :: Int
```

我们需要的是返回这个值的类型为IO Int的动作，这便是前一节介绍的return的目的。所以，读入一个整数的程序可写作

```
getInt：：IO Int
getInt = do line <- getLine
            return (read line :: Int)
```

小结

do 表达式提供了一种顺序程序设计的方法。我们可以使用 do 顺序连接简单的 I/O 程序，从而构造复杂的 I/O 交互程序。表示法'<-'可用于命名一个动作返回的值，并且在随后的 I/O 程序中使用这个值。我们还可以使用 let 定义命名中间计算结果，使得程序更易读。

下一节我们介绍如何编写带循环的 I/O 程序，例如，读取输入的所有行。我们将看到，这样的程序可以通过递归地定义一个循环来实现。我们还将讨论'<-'与通常赋值运算的不同之处。

习题

18.1 编写一个I/O程序：读入一行输入并测试输入是否一个回文。程序应该提示用户输入，并在测试结束后输出适当的信息。

18.2 编写一个I/O程序：分别读入两个整数，然后返回它们的和。程序应该提示输入并解释输出。

18.3 编写一个I/O程序：先读入一个正整数，设为 n，然后读入 n 个整数，并输出它们的和。程序应该提示输入并解释输出。

§18.4 迭代与递归

本节介绍如何构造带有循环的程序，其中包含如何构造一般的 while 循环。我们还将看到变量与 do 表达式中命名的区别。

while 循环

假设我们想在一个条件成立时重复一个 IO () 动作。条件将依赖于 I/O 系统，所以其类型为 IO Bool。一个例子是函数库 IO.hs 中用于测试输入尾的函数

```
isEOF :: IO Bool
```

由上述讨论可知，while 循环应该具有下列类型

```
while :: IO Bool -> IO () -> IO ()
```

其定义如下

```
while test action
   = do res <- test
        if res then do action
                       while test action
               else return ()
```

定义中包含两个 IO 动作。第一个完成测试 test，其布尔型结果被命名为 res。第二个动作依赖于 res；如果 res 为 True，则执行下列动作

```
do action
   while test action
```

这表示在条件为真的情况下，先执行动作 action，然后重复循环。另一方面，如果条件为 False，则程序应该“不做任何动作”。空 I/O 动作为

```
return ()
```

因为它返回类型()的唯一值，而且并没有执行任何 I/O。

将输入拷贝到输出

考虑 while 循环应用的一个实例。假如我们想把输入逐行拷贝到输出，可以使用 while 循环完成：

```
while -- not end of file
      -- read and write a line
```

如何测试文件的末尾呢？我们可以测试 isEOF，并且返回这个结果的相反值：

```
do res <- isEOF
   return (not res)
```

如果仍有输入可读，我们需要完成什么动作呢？如前所述，读一行然后写一行

```
do line <- getLine
   putStrLn line
```

综合以上的分析，最后的程序为

```
copyInputToOutput :: IO ()
copyInputToOutput
  = while (do res <- isEof
              return (not res))
          (do line <- getLine
              putStrLn line)
```

要注意的是，程序中的括号是不可省略的。

一个重要例子

假设我们想拷贝一些输入行直至遇到一个空行。第一次尝试如下

```
goUntilEmpty :: IO ()
goUntilEmpty
  = do line <- getLine                          (1)
       while (return (line /= []))              (2)
             (do putStrLn line                  (3)
                 line <- getLine                (4)
                 return ())                     (5)
```

其中命名了程序行以便于讨论。上述程序的功能如下：(1) 读入一行并命名为 line；当此行不空时，即 (2) 的测试return (line/=[])为真时，输出 (3) 的 line，然后在 (4) 读另一行并命名为 line。

这个程序的功能是重复输出输入的第一行；也就是在(1)读入并在(2)，(3)中使用的 line。在 (4) 中我们建立了一个**新变量**line，把它记作读入的值，但

这并不是对原变量 line 的重新赋值，所以 (2) 中的测试和 (3) 中的输出都仍然指的是第一行的输入值。这便是这些变量与命令式语言中变量的不同之处。如果我们把

```
line <- getLine
```

看作一个赋值`line := getLine`, 那么这是对一个变量的一次性赋值，不能修改：每次`line <-...`的出现均建立一个新变量 line。换句话说，变量的值不会改变。

如何编写满足上述要求的正确程序呢？关键是使用递归。

```
goUntilEmpty :: IO ()
goUntilEmpty
  = do line <- getLine                    (1)
       if (line == [])                    (2)
          then return ()                  (3)
          else (do putStrLn line          (4)
                   goUntilEmpty)          (5)
```

程序的功能是：读一行 (1)，如果这一行为空 (2)，则 IO 动作返回() (3)。另一方面，如果读入的行不空，则输出 (4)。然后重新执行整个程序，下一次读入的行称为 line，如果 line 不空，则重复 (2)、(4) 和 (5)。

求一个整数序列之和

假如我们要编写一个交互式程序，每次读一个整数，直至输入的整数为 0，然后计算这些整数之和。程序将具有下列类型

```
sumInts :: IO Int
```

程序中有两种情况：如果读入的整数为 0，则结果必为 0；否则，将读入的整数加到其余整数之和上，后者可调用 sumInts 得到。由此得下面的程序

```
sumInts
  = do n <- getInt
       if n == 0
          then return 0
          else (do m <- sumInts
                   return (n+m))
```

其中使用了前面定义的函数 getInt，它从一个输入行读入一个整数。有趣的是上述程序与下列递归的比较

```
sum [] = 0
sum (n:ns)
  = n + sum ns
```

或者如下修改的 sum

```
sum [] = 0
```

```
sum (n:ns)
 = let m = sum ns
   in (n+m)
```

我们还可以将 sumInts “包装” 起来; 解释它的目的并且最后输出其和。

```
sumInteract :: IO ()
sumInteract
  = do putStrLn "Enter integers one per line"
       putStrLn "These will be summed until zero is entered"
       sum <- sumInts
       putStr "The sum was "
       print sum
```

习题

18.4 编写一个程序, 重复读入输入行并检查它是否一个文回, 直至读入空行。程序应该向用户清楚地解释, 输入是什么, 输出是什么。

18.5 编写一个程序, 重复读取整数 (每行一个整数) 直至输入的数为 *0*, 然后输出读入整数的有序序列。在这种情况下, 哪一种排序算法最适合?

18.6 定义下列函数

```
mapF :: (a -> b) -> IO a -> IO b
```

其功能是将一个函数应用于一个交互程序的结果。使用 *do* 记法。

18.7 定义下列函数

```
repeat :: IO Bool -> IO () -> IO ()
```

使得repeat test oper的功能是重复oper直至条件test为True。

18.8 定义while的推广,

```
whileG :: (a -> IO Bool) -> (a -> IO a) -> (a -> IO a)
```

其中条件和运算定义在类型a上。

18.9 编写下列程序: 读入一个数, 如n, 然后读入n个数, 并输出它们的平均值。如果可能, 请使用whileG。

18.10 修改前一习题的答案, 如果在读入n个数之前到达文件尾, 则输出相应的信息。

18.11 定义下列函数

```
accumulate :: [IO a] -> IO [a]
```

程序完成一系列的动作, 并将其结果存储于一个列表中, 再定义下列函数

```
sequence :: [IO a] -> IO ()
```

程序顺序完成一系列的交互动作, 并舍弃返回的结果。最后, 说明如何完成一系列值的传递, 即将一个程序的结果传给下一个程序:

```
seqList :: [a -> IO a] -> a -> IO a
```

函数在空列表上的结果个什么?

§18.5 计算器

计算器的各组成部分分布在本书的三个地方：

- 在 14.2 节我们引入表达式的代数类型 Expr，并在 17.5 节修改为

```
data Expr = Lit Int | Var Var | Op Ops Expr Expr
data Ops  = Add | Sub | Mul | Div | Mod
type Var  = Char
```

在讨论了变量的值的存储之后，我们如下修改了表达式的计算

- 在第十六章我们介绍了抽象类型 Store，用于描述变量的当前值。抽象数据类型的 signature 是

```
initial :: Store
value   :: Store -> Var -> Int
update  :: Store -> Var -> Int -> Store
```

- 在 17.5 节我们讨论了如何对表达式和命令做语法分析，其中

```
data Command = Eval Expr | Assign Var Expr | Null
```

并定义了将一行输入转换为一个 Command 的函数，例如

```
commLine "(3+x)"   = (Eval (Op Add (Lit 3) (Var 'x')))
commLine "x:(3+x)" = (Assign 'x' (Op Add (Lit 3) (Var 'x')))
commLine ""        = Null
```

计算表达式的函数定义为

```
eval :: Expr -> Store -> Int

eval (Lit n) st   = n
eval (Var v) st   = value st v
eval (Op op e1 e2) st
  = opValue op v1 v2
    where
    v1 = eval e1 st
    v2 = eval e2 st
```

其中类型为`Ops -> Int -> Int -> Int`的函数 opValue 用于解释每个运算符，如 Add 解释为相应的函数 (+)。

一个命令的功能是什么呢？一个表达式应该返回表达式在当前状态下的值；一个赋值应改变当前状态；一个空命令不做任何事情。为此，我们定义一个返回一个值和一个状态的函数

```
command :: Command -> Store -> (Int, Store)

command Null st     = (0, st)
command (Eval e) st = (eval e st, st)
```

```
command (Assign v e) st
  = (val, newSt)
    where
    val   = eval e st
    newSt = update st v val
```

计算器的一步计算将从一个初始 Store 开始，读入一个输入行，计算相应的命令，输出某个结果并返回一个修改的 Store，

```
calcStep :: Store -> IO Store

calcStep st
  = do line <- getLine
       let comm         = commLine line        (1)
           (val, newSt) = command comm st      (2)
       print val
       return newSt
```

在 calcStep 的定义 (1) 和 (2) 中，我们看到在 do 表达式中使用 let 的例子。其中 (1) 给出对 line 进行语法分析后的值 comm

```
let comm   = commLine line
```

这个值随后被应用于 (2) 中

```
(val, newSt) = command comm st
```

(2) 同时给出读入的表达式的值 val 和新的状态 newSt。注意，let 引导了多行定义，它在"print" 前结束。在 let 之后，val 被输出，新的状态 newSt 是整个交互程序的结果。

计算器的一系列计算步骤定义如下

```
calcSteps :: Store -> IO ()
calcSteps st
  = while notEOF
          (do newSt <- calcStep st
              calcSteps newSt)
```

其中循环检测 notEOF 如下定义

```
notEOF :: IO Bool
notEOF = do res <- isEOF
            return (not res)
```

计算器的主程序由 calcSteps 应用于初始状态构成

```
mainCalc :: IO ()
mainCalc = calcSteps initial
```

在随后的习题中，我们将讨论计算器的各种扩充和修改。

习题

18.12 如何给计算器的输出加上初始信息和最后结果信息。

18.13 如果输入计算器的命令是非法的，那么函数topLevel将生成一个错误信息，并终止计算。试讨论如何给topLevel增加一个参数，在出错的情况下，计算不会终止。

18.14 如何修改系统，使得系统可以使用任意长度由字母开始，并且由字母和数字构成的变量名。

18.15 如何扩展计算器，使它不仅可以处理整数，还可以处理浮点数。

18.16 讨论如何修改计算器，使其接收多行输入一个命令。你需要决定用户如何告知系统，一个命令由多行组成，例如，在底部加上 /\。如何修改交互程序以适应这样的变化。另一种方法是用户不需给出多行信号；你能使程序做到这一点吗？

18.17 如何修改程序使其允许输入命令中含空格，例如

```
"   x : (2\t+3)     "
```

语法分析结果为

```
(Assign 'x' (Op Add (Lit 2) (Lit 3)))
```

§18.6 进一步的 I/O

本节概述 Haskell I/O 的进一步特色。

文件 I/O

到目前为止的 I/O 只涉及读标准输入和写到标准输出。Haskell I/O 模型还支持读文件、写文件和追加文件，它们分别由下列函数完成：

```
readFile   :: FilePath - >IO String
writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

其中

```
type FilePath = String
```

即文件用一个适当的字符串说明。

错误

I/O 程序可以产生错误，其类型是依赖于系统的数据类型 IOError。下列函数对于给定的错误，建立一个失败的 I/O 动作

```
ioError :: IOError -> IO a
```

程序

```
catch :: IO a -> (IOError -> IO a) -> IO a
```

捕获第一个参数产生的错误，然后用第二个参数处理。第二个参数对每个可能的 IOError 给出一个句柄，是 `IO a`类型的动作。有关错误处理的更详细信息参见 I/O 库 IO.hs 的文档。

输入和输出为惰性列表

在早期惰性函数程序设计中，流行的 I/O 程序观点是将输入和输出看作 String，即字符的列表。在这样的模型中，一个 I/O 程序是一个函数

```
listIOprog :: String -> String
```

显然，这种观点对于首先读入所有输出，然后产生输出的“批”程序是有意义的。事实上，它也适用于惰性语言中输入和输出交替进行的交互式程序。这是因为，在惰性语言中，我们可以在参数 – 交互式输入 – 完全计算之前开始打印计算结果 – 交互式程序的输出。例如，在这种模型下重复地将输入行逆转可写作

```
unlines . map reverse . lines
```

这种方法的缺点是不能推广。输入和输出的交替方式经常难以预测：输出经常出现得迟，有时会出现得早；Haskell 的 IO 方法避免了这些问题。不过，我们仍然可以通过下列函数使用这种模式

```
getContents :: IO String
```

这是一个读取标准输入的原始操作，可用于下列程序

```
interact :: (String -> String) -> IO ()
interact f = do s <- getContents
                putStr (f s)
```

IO.hs 库

有关文件处理的更多函数参见 IO.hs 库。IO.hs 的设计基于上述原理，详情见函数库文档。

习题

18.18 写出goUntilEmpty和sumInts在文件上的版本。

18.19 写出goUntilEmpty和sumInts的惰性列表版本。

18.20 (较难) 编写计算器的惰性列表版本。

§18.7 再谈 do 记法

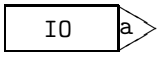
我们看到，类型 `IO a`上带有一些预定义操作，包括

```
return  :: a -> IO a
putStr  :: String -> IO ()
getLine :: IO String
```

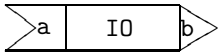
我们还可以用 do 顺序连接 IO a的程序。本节讨论 do 的工作原理，我们将看到，IO 是一个更一般概念的特例。理解 do 的关键是运算(>>=)，常读作“然后”。此运算用于顺序连接两个操作，并将前一个操作的结果作为参数传给第二个操作

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

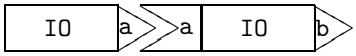
这个运算的作用是什么呢？它将一个 IO a的元素



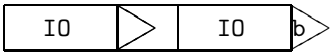
与一个类型a->IO b 的函数结合



上述两个程序结合后，前一个程序的结果传给了第二个程序



将两者合到一起的结果是：实现一些 IO 操作，然后返回一个类型 b 的值，即 IO b的对象。



这些与 do 有何关系呢？看下面的例子。试想下列程序的作用

```
addOneInt :: IO ()
addOneInt
  = do line <- getLine
       putStrLn (show (1+read line :: Int))
```

getLine 返回的值记为 line，然后 line 用于随后的操作中。应用(>>=)时，我们必须使用一个期待类型 String 参数的函数，所以可写作

```
addOneInt
  = getLine >>= \line ->
    putStrLn (show (1+read line :: Int))
```

其中 \x->e 表示参数为 x，结果为 e 的函数，这里把参数也称作 line。更复杂的例子可用类似的方式将 do 转换为顺序连接(>>=)。我们今后将继续使用 do，但是，要记住的是，do 建立在顺序连接 I/O 程序的运算(>>=)之上。

习题

18.21 使用(>>=)代替do重复实现前面的部分例子和习题。

§18.8 函数程序设计中的 monads

随着函数程序设计研究的进展和经验的积累，人们发现了一些很优美和表达力很丰富的程序设计方式，其中包括适用于很多领域的 monad 式的程序

设计。本节包括对这种程序设计方式的简介，有关这种方式以及进一步的技术参见 Jeuring 和 Meijer (1995)。我们已经看到，monad 的特点是它使得操作发生的顺序显式化。do 结构所基于的顺序结合运算(>>=)是顺序连接一般 monad 的运算

```
(>>=) :: m a -> (a -> m b) -> m b
```

什么时候需要这样的顺序连接运算呢? 考虑下列的数值表达式

```
e - f
```

因为这只是一个表达式，我们可以用任意顺序计算 '-' 的参数 e 和 f，或者同时计算 e 和 f。但是，假如表达式 e 和 f 引发 I/O 操作，或者改变某些内存，那么我们需要说明计算发生的顺序，因为不同的顺序产生不同的结果。一个简单例子是 18.1 节讨论的例子

```
inputInt - inputInt
```

如果先输入 7，再输入 4，则从左至右计算的结果是 3，而从右至左的计算结果是 -3；并行计算的结果不可预测。

如何得到一种明确的顺序呢? 下列运算明确表示 e 在 f 之前输入

```
do e <- getInt
   f <- getInt
   return (e-f)
```

其中getInt::IO Int完成整数输入。这种明确的顺序是许多具有副作用的程序设计语言的特点。monad 方法的基本特点是它可以把这种副作用揉合到纯函数程序设计中。下面介绍 monad 的概念。

什么是 monad?

一个 monad 是基于一个多态类型构造符 m 的一簇类型m a，并且具有函数 return, (>>=), (>>)和 fail:

```
class Monad m where
 (>>=)  :: m a -> (a -> m b) -> m b
 return :: a -> m a
 (>>)   :: m a -> m b -> m b
 fail   :: String -> m a
```

这是一个 **构造符类** 的例子，它是由 **类型构造符**m 构造的一个分类。一个类型构造符是一个由类型到类型的函数。

Monad 的定义包含>>和 fail 的缺省定义

```
m >> k = m >>= \_ -> k
fail s = error s
```

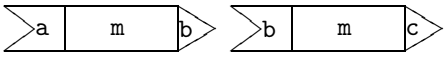
从上述定义看出，>>的作用类似于>>=，只是由第一个参数返回的结果被舍弃了，而没有传给第二个参数。要真正成为一个 monad，函数 return 和(>>=)应该满足一些简单性质：

- 运算 return x 只返回值 x，没有任何计算作用，例如在 IO monad 的情形，它不涉及任何输入和输出。
- 由>>=构造的顺序与括号的方式没有关系；
- fail s 的值对应于失败的计算，并给出一个错误信息 s。

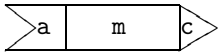
使用导出运算符 `>@>` 时, 这些规则的描述显得更清楚：

```
(>@>) :: Monad m => (a -> m b) ->
                    (b -> m c) ->
                    (a -> m c)

f >@> g = \x -> (f x) >>= g
```

这个运算符是对象的复合，是函数复合的推广[1]。将下列对象复合



其结果为



注意，return 也具有这种形式，因为它的类型是`a->m a`。

现在我们可以正式地给出 monad 应该满足的规则。首先，return 对于运算 `>@>` 是一个单位元：

```
return >@> f = f                                (M1)
f >@> return = f                                (M2)
```

其次，运算符 `>@>` 是可结合的。

```
(f >@> g) >@> h = f >@> (g >@> h)               (M3)
```

当然，我们无法在 Monad 的 Haskell 定义中要求这些规则 (M1) - (M3) 成立。我们也可以用 do 来陈述上述规则，因为

```
m >>= f = do x <- m
             f x
```

前两个规则成为

```
do y <- return x  = f x
   f y
do x <- m         = m
   return x
```

第三条规则隐含在 do 结构是可结合的性质中。

monad 的例子

我们曾经讲过，一个 monad `m a`可以看作一种计算的表示，即`m a`的元素是一个完成某些动作、然后返回类型 a 的一个值的计算。下面介绍一些例子，

[1]在范畴论中这种复合称为 **Kleisli 复合**

并解释它们的计算意义。

恒等 monad

最简单的 monad 是将一个类型映射到它本身的恒等 monad, 其定义如下

```
m >>= f = f m
return  = id
```

在这种解释下, >@> 成为函数的向前复合, >.>, 它确实满足结合律, 而且 id 是它的单位元。一个未定义的计算与任何计算顺序连接结果是未定义的。在计算意义上, 这个 monad 代表平凡的状态, 不进行任何动作并立即返回值。

输入/输出 monad

我们已经在 18.2 节看到 IO monad 的例子。

列表 monad

我们可以用列表构造一个 monad

```
instance Monad [] where
  xs >>= f = concat (map f xs)
  return x = [x]
  fail s   = []
```

列表 monad 的计算解释是非确定性计算：[a] 的一个元素表示潜在的非确定性计算的所有结果。在这里, return 给出一个结果, >>=将函数 f 应用于 xs 的所有可能结果, 然后将结果连接在一起, 形成最后可能结果的列表。fail s 的值对应于非确定性计算无结果, 或者说, 不能给出结果。

Maybe monad

Monad 的另一个例子是“可能”类型Maybe a, 它的成员包含'Just' 类型 a 的元素和 Nothing, 也就是说, 可能是类型a的元素:

```
instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing  >>= k = Nothing
  return         = Just
  fail s         = Nothing
```

它的计算意义是可能产生一个结果的计算, 但也可能产生一个错误, 详见 14.4 节。

语法分析 monad

第五个例子是语法分析, 我们将说明Parse a是一个 monad。为此, 我们将用一个新的构造符将 SParse 把Parse a包裹在一个数据类型中

```
data SParse a b = SParse (Parse a b)
```

```
instance Monad (SParse a) where
  return x = SParse (succeed x)
  fail s   = SParse none
  (SParse pr) >>= f
    = SParse (\st -> concat [sparse (f x) rest | (x,rest) <- pr
    st])

sparse :: SParse a b -> Parse a b
sparse (SParse pr) = pr
```

(>>=)的定义类似于(>*>)：一趟语法分析由一个语法分析器实现，输入的剩余部分传给第二个语法分析器f。设第一次分析的结果为x，则分析器(f x)被应用于剩余的输入部分rest。

状态 monad

本章稍后将介绍一个状态 monad，`State a b`。这个类型的一个运算在返回一个类型 b 的值之前可以改变状态（类型 a）。

monad 的组合

Monad 可以组合起来完成更复杂的功能，例如，可以构造完成 I/O 和处理状态值的计算。有关详情参见 Liang, Hudak 和 Jones (1995)。

一些标准函数

我们可以定义 monad 上的一些标准函数，它们的类型与 List 上函数的类型相似

```
mapF  :: Monad m => (a -> b) -> m a -> m b
joinM :: Monad m => m (m a) -> m a
```

其定义为

```
mapF f m = do x <- m
              return (f x)
joinM m = do x <- m
             x
```

这些函数在列表上称为 map 和 concat。map 和 concat 的许多性质可以搬到这些函数上来。例如，使用 (M1) 至 (M3) 可以证明下式对任意 f 和 g 成立

```
mapF (f . g) = mapF f . mapF g          (M4)
```

习题

18.22 证明集合，树和下列类型均为Monad*:*

```
data Error  a = OK a | Error String
```

18.23 证明(M1)至(M3)对于monad Id，[]和Maybe以及前一习题中的monad均成立。

18.24 利用(M1)–(M3)证明性质(M4)。

18.25 利用monad规则证明下列性质：

```
joinM . return = joinM . mapF return
joinM . return = id
```

18.26 你能在列表上定义一个不同于本节定义的monad结构吗？证明你的定义满足性质(M1)–(M3)。

18.27 利用列表概括定义列表上的函数map和join。试比较这些定义与本节中用do定义的mapF和joinF。

18.28 使用do（基于>>=，而非>*>）和build重新实现计算器的语法分析器，并比较这两种方法。

§18.9 树上的 monadic 计算

本节介绍如何使得下列类型上的计算具有 monad 结构

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

我们先看一个简单例子，然后再看一个更实际的例子。我们将看到，使用 monad 方式时，两个解的最外层结构是一个样的。这个结构将引导我们完成第二个例子的实现。

这些例子说明，monad 提供了一种构造程序的重要方法，因为它鼓励程序员**分层次**考虑问题。一个计算的外层结构使用 monad 组织，但其细节留待 monad 实现。在 monad 内部是计算如何进行的具体描述，如维护状态或完成 I/O（或同时进行）；monad 的顺序运算将保证数据以适当的方式在程序中不同的部分之间传递。

当需要修改计算的细节时，这种分层次方法很快显示出它的优点：通常只需要在最外层做最少的修改即可完成对 monad 实现的计算的修改。这与非 monad 计算形成鲜明对比，在后一种情形下，数据的表示是可见的，因此通常需要对整个程序进行修改。

求整数树之和

假设我们要求一个整数树的和

```
sTree :: Tree Int -> Int
```

直接的递归解是

```
sTree Nil            = 0
sTree (Node n t1 t2) = n + sTree t1 + sTree t2
```

上述解没有说明计算和的顺序：我们可以依次计算 STree t1 和 STree t2 或者并行计算两者。如何用 monad 来解这个问题呢？

```
sumTree :: Tree Int -> St Int
```

其中 St 是我们将要定义的 monad。在 Nil 的情况

```
sumTree Nil = return 0
```

而在 Node 的情况，我们按如下顺序计算各部分

```
sumTree (Node n t1 t2)
  = do num <- return n
       s1 <- sumTree t1
       s2 <- sumtree t2
       return (num+s1+s2)
```

这个定义是如何组织的呢？我们使用 do 顺序连接各个运算。首先返回值 n，命名为 num；然后计算 sumTree t1 和 sumTree t2，分别用 s1 和 s2 计它们的结果；最后返回结果，即和 num+s1+s2。因为我们在这里做的是数值的计算，没有完成任何 I/O 或者有副作用的运算，我们将 St 定义为前面提到的恒等 monad Id，其定义为

```
data Id a = Id a

instance Monad Id where
  return         = Id
  (>>=) (Id x) f = f x
```

因此，我们有

```
sumTree :: Tree Int -> Id Int
```

定义 (sumTree) 与其它的命令式程序及其相似，因为 do 实现顺序执行，`j <-...`给 j 赋一个值。使用命令式语言书写的程序可能是

```
num := n ;
s1 := sumTree t1 ;
s2 := sumTree t2 ;
return (num + s1 +s2) ;
```

其中'num:='对应于<-, do 就像分号';'一样，将一系列命令顺序连接起来。要得到一个类型`Tree Int->Int`的函数，可以使用一个提取函数的复合

```
extract . sumTree
```

其中 extract 丢掉 Id x 元素的包装，返回 x:

```
extract :: Id a -> a
extract (Id x) = x
```

下一节我们将解决一个更复杂的问题，但是使用了相同的 monad 结构。

在树的计算中使用状态 monad

在上节定义 sumTree 的经验之上，我们来解决一个更困难的问题。我们希望定义一个函数

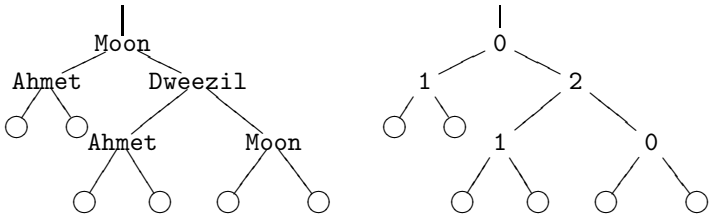


图 18.1: 使用整数代替树上的元素

```
numTree :: Eq a => Tree a -> Tree Int
```

它将任意树转换为一个整数树，即原来树上的元素被自然数代替。例如，图 18.1 表示这样的一个例子。同一个元素的所有出现应该用同一个自然数代替，而且当我们遇到一个未访问的元素时，必须找出一个" 新的" 数来代替它。

如何定义这个函数呢？我们给出函数的类型

```
numberTree :: Eq a => Tree a -> State a (Tree Int)
```

其中 monad State a 将包含足够的信息使我们正确地替换树中的元素。程序的结构如下

```
numberTree Nil = return Nil
numberTree (Node x t1 t2)                          (numberTree)
   = do num <- numberNode x
        nt1 <- numberTree t1
        nt2 <- numberTree t2
        return (Node num nt1 nt2)
```

这个程序的结构与第 351 页中 (sumTree) 的结构完全一样；我们在组成树的成分上完成相应的运算，然后将它们结合起来形成结果(`Node num nt1 nt2`)。

要计算上述结果我们还需要做什么呢？我们需要定义 monad `State a`以及代换结点元素的函数

```
numberTree :: Eq a => a -> State a Int
```

我们现在必须考虑如何实现这个monad。因为它记录了哪些元素与哪些数关联，所以我们称之为 State。我们可以用一个表格来描述它

```
type Table a = [a]
```

例如, [True, False] 表示 True 与 0 关联，False 与 1 关联。

那么状态 monad State 是什么呢？它由下列函数相成

```
data State a b = State (Table a -> (Table a, b))
```

如果将上面的构造符去掉，它表示将运算前的状态转移到运算后的状态，并且返回一个结果。换句话说，我们返回类型 b 的值，但是由于副作用可能改变了类型 Table a 的状态值。下面我们定义 monad 的两个运算

```
instance Monad (State a) where
  return x = State (\tab -> (tab, x))
```

即对于 return 函数，只需保持状态不变，并且直接返回值。

如何顺序连接运算呢？我们的目的是执行 st，将它的返回值传给 f，然后完成相应的运算。更具体地说，要完成 st，我们将表 tab 传给 st，其结果是一个新的状态，newTab 和值 y。将 y 传给 f，结果是类型 State a b 的一个对象 trans，然后在新状态 newTab 上完成 trans

```
(State st) >>= f
  = State (\tab -> let
                    (newTab, y)      = st tab
                    (State trans)    = f y
                   in trans newTab)
```

可以看出，运算确实是从一个状态到下一个状态顺序进行的。剩下的工作是定义函数 numberNode，其定义为

```
numberNode :: Eq a => a -> State a Int
numberNode x = State (nNode x)

nNode :: Eq a => a -> (Table a -> (Table a, Int))
nNode x table
  | elem x table   = (table,    lookup x table)
  | otherwise      = (table++[x], length table)
```

如果 x 是 table 的一个元素，我们返回它在表中的位置，由 lookup 求得；如果 x 不是 table 的元素，我们将其加到表的末尾，并返回它的位置，即 length table。Lookup 的定义是标准的

```
lookup :: Eq a => a -> Table a -> Int
```

留给读者作为练习。

我们已经完成下列函数的定义

```
numberTree :: Eq a => Tree a -> State a (Tree Int)
```

但是，仍然有一个问题需要解决。如果对于某个树 exampleTree 计算

```
numberTree exampleTree
```

我们得到下列类型的一个对象

```
State a (Tree Int)
```

为了提取结果，我们必须定义函数

```
extract :: State a b -> b
```

函数必须从某个初始状态开始，完成计算，并返回类型 b 的结果。其定义为

```
extract :: State a b -> b
extract (State st) = snd (st [])
```

其中 st 被应用于初始状态，其结果是一个二元组，类型为 b 的第二个分量便是我们求的结果。最后得到下列解

```
numTree :: Eq a => Tree a -> tree Int
```

```
numTree = extract . numberTree
```

综上所述，我们看到树上的复杂计算（numberTree）如何组织成与一个简单计算（sumTree）完全一样的结构。对于树类型，其优点是明显的，但是对于更复杂的类型，如果我们要完成一个具有复杂副作用的计算，那么 monad 结构是很基本的。

习题

18.29 说明如何在一个列表中寻找一个元素的位置：

```
lookup :: Eq a => a -> Table a -> Int
```

或许定义下列函数会有帮助

```
look :: Eq a => a -> Table a -> Int -> Int
```

其中的整数参数表示列表中当前的偏移量。

18.30 说明如何使用State式monad将一棵树中的元素用随机数代换。使用 *17.6* 节的技术生成随机数。

18.31 我们可以用monad以各种方式扩展计算器。试考虑如何做下列扩展

- 当遇到零作除数时添加异常处理或者输出信息；
- 统计计算中的步数；
- 综合上述两者。

小结

从我们所介绍的例子可以看出，使用 monad 来组织计算有下列三方面的特点：

- 一种非常方便的书写顺序程序的方法，它非常类似于命令式程序的书写方式；
- 抽象的优点：我们可以改变其中的 monad 而保持计算的结构不变；
- 最后，一旦我们定义了 monad，各种性质可以自动推导出来。例如，(M4) 是 monad 性质 (M1) – (M3) 的推论。

我们看到，许多计算效果，如 IO，状态，错误（由 Maybe 类型实现）和非确定性 (由列表 monad 实现) 均可以由 monad 实现。这表明，一方面我们可以在函数程序语言中描述这些计算效果，另一方面，我们可以使用 monad 建立纯函数语言 Haskell 与有副作用的系统之间的接口；这种方法是非常有用的，它使得 Haskell 程序可以调用其他语言的函数。事实上，Haskell 程序员可以利用这种方法与 C 和 Java 程序员协同工作。在最后一章，我们将提供有关使用 monad 编程的进一步工作的一些参考资料出处。

第十九章　时间和空间行为

本章不讨论程序计算的值，以及计算这些值的方法；我们感兴趣的是程序的**效率**，而不是程序的正确性。

我们先讨论一般地如何度量复杂度，然后讨论如何度量函数程序的时间和空间性能。我们将计算一系列函数的时间复杂度，最后讨论 Set 抽象数据类型的各种实现。

惰性函数程序的空间行为比较复杂：我们将证明有些程序使用的空间比我们预料的少，有的比我们预料的多。然后我们讨论列表上折叠函数的空间性能，并引入从左边开始折叠的函数 foldl'，这个折叠函数的空间效率较高，但是它的参数需要使用**严格**运算符。与此相对应的是，一般地，foldr 的惰性折叠效率更高。

在许多算法中，朴素的实现会导致部分解的重复计算，因此导致实现的低效率。在本章的最后一节，我们介绍如何使用惰性计算通过**记忆**部分结果以得到算法更高效的实现。

§19.1　函数的复杂度

如果我们想度量函数的性能，一种方法是检查函数在不同输入值上计算时所消耗的时间和空间。例如，给定自然数上的一个函数 fred，一个自然数 n，统计计算`fred n`的值的计算步数。由此得到一个函数，称之为 stepsFred，然后我们可以讨论这个函数的复杂性如何。

一种估计函数复杂度的方法是检查当参数增长时，它的值增长的速度。这种方法的思想是：一个函数的基本性能在它的值增长到很大时会显示出来。我们先看一个例子。对于下列函数，随着 n 的增大，函数值会增长多快？

```
f n = 2*n^2 + 4*n + 13
```

这个函数有三项组成：

- 常数 13
- 4*n 以及
- $2*n^2$。（注意：我们在这里使用了常用的数学计法 n^2，而不是 Haskell 计法 `n^2`。）

当 n 增大时，这些项如何变化呢？

- 常数项 13 保持不变；
- 4*n 像直线一样增长；但是
- 平方项 $2*n^2$ 增长保持得最快。

对于“较大的”n 值，平方项大于其他项，所以我们说 f 的**阶**是 n^2，记作 $O(n^2)$。对于此例，当 n 大于等于 3 时，平方项将占主导地位；在定义阶时我们将会给

出“大于”的确切含义。规则的主要思想是，当把除增长最快以外的项去除，并且忽略它的常数倍后，阶将把函数的性能分类。本节将进一步说明这个规则的确切含义，但是，这里的解释应足以理解本章的其余内容。n^2是数学家表示函数“输入 n，输出 n^2”的记法，也是通常描述复杂度的记法。所以我们在这里也使用这种记法。在 Haskell 程序中，这种函数或者记作 `\n->n^2`，或者记作部分运算 `(^2)`。

在本节的剩余部分我们将给出阶的确切定义，然后讨论各种例子，并将其置于度量复杂性的尺度上。

大 O 和大 Θ 表示法十十上界

一个函数 `f :: Int->Int` 的阶是 O(g)，读作“大 O”，如果存在正整数 m 和 d 使得对所有的 $n \geq m$,

$$f\ n \leq\ d*(g\ n)$$

定义表示，当 n 足够大时 ($n \geq m$),f 的值不大于函数 g 的常数倍。例如，上述 f 是 $O(n^2)$，因为当 n 大于等于 1 时，

$$2*n^2+4*n+13 \leq 2*n^2+4*n^2+13*n^2 = 19*n^2$$

所以，定义中的 m 可取 1，d 可取 19。注意，这种度量给出一种可能过高估计的上界；同理可推得 f 是 $O(n^{17})$。在大多数情况，我们考虑的是上确界。表达 g 是 f 的确界的方法是，除 f 的阶是 O(g) 外，g 的阶是 O(f)；此时称 f 是 Θ(g)，读作“Theta g”。对于上例，f 是 $\Theta(n^2)$。

度量尺度

如果 f 为 O(g)，但 g 不是 O(f)，则记作 $f \ll g$；如果 f 是 O(g)，同时 g 是 O(f)，则记作 $f \equiv g$。

我们现在给出度量函数复杂度的尺度。常数是 $O(n^0)$，它的增长较 $O(n^1)$ 阶的**线性**函数慢，线性函数的增长又低于 $O(n^2)$ 阶的**平方**函数等等，而所有的幂函数 n^k 以指数函数为上界，如 2^n。

$n^0 \ll n^1 \ll n^2 \ll \cdots \ll n^k \ll \cdots \ll 2^n \ll \cdots$

在上面的尺度上应该再增加两个点。对数函数 log 的增长速度低于 n 的任何正数幂，函数 n 与 $\log_2 n$的乘积，即 $n\log_2 n$，界于线性函数和平方函数之间。

计数

以下的许多讨论将涉及计数。我们先来看几个一般例子。我们将会在以后讨论函数的性能时遇到这些函数。

例 19.1 第一个问题是：给定一个列表，经过多少次平分后子列表的长度变成1?

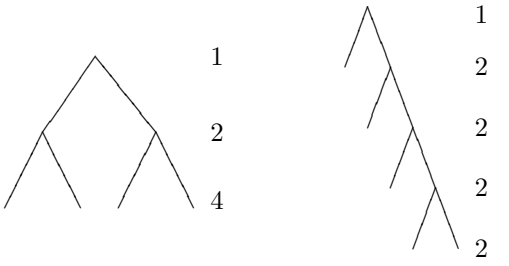


图 19.1: 计算树中的结点数

如果列表的长度是 n, 则第一次平分后每个子表的长度为 n/2, p 次平分后每个子表的长度是 $n/(2^p)$。在下列条件下次数小于或等于 1：

$$(2^p) \geq n > (2^{(p-1)})$$

对上式每边取对数 $\log_2$:

$$p \geq \log_2 n > p - 1$$

因此，用列表长度表示的步数函数是 $\Theta(\log_2 n)$。

例 19.2 第二个问题是有关树的计数。

一棵树称为**平衡的**，如果它的所有分支具有相同的长度。假定一棵平衡树所有的分支长度为 b, 试问树上的结点数是多少？在树的第一层有一个结点，第二层有两个，第 k 层有 $2^{(k-1)}$, 所以，结点总数为

$$1 + 2 + 4 + ... + 2^{(k-1)} + ... + 2^b = 2^{(b+1)} - 1$$

如图 19.1 所示。

设平衡二叉树的分支长度为 b, 则其结点数为 $\Theta(2^b)$; 所以，结点数为 n 的平衡二叉树的分支长度为 $\Theta(\log_2 n)$。如果一棵树是不平衡的，则树的最长分支的长度可以与树的结点数同阶，如图 19.1 例所示。

例 19.3 最后一个统计问题是和的计算。

如果我们每天得到一件礼物，n 天后共得到 n 件礼物；如果每天得到 n 件礼物，则 n 天后共得到 n^2件礼物；如果第一天得到一件，第二天得到两件，如此类推，那么 n 天后共得到多少件礼物呢？换句话说，列表`[1 .. n]`的和是多少呢？将列表之和按照从前到后和从后到前如下书写

```
1    + 2     + 3     + ... + (n-1) + n +
n    + (n-1) + (n-2) + ... + 2     + 1
```

将上式垂直相加得到

```
(n+1) + (n+1) + (n+1) + ... + (n+1) + (n+1)
```

其中 (n+1) 出现 n 次，故

```
sum [1 .. n] = n*(n+1) `div` 2
```

故此和的阶是 $\Theta(n^2)$, 即平方阶。类似地可以得到，平方和的阶是 $\Theta(n^3)$，等等。

习题

19.1 说明上一节的例子

$$\texttt{f n = 2} * \texttt{n}^2 \texttt{ + 4*n + 13}$$

是 $\Theta(\texttt{n}^2)$.

19.2 对于下列 n 值

```
0 1 2 3 4 5 10 50 100 500 1000 10000 100000 10000000
```

列表计算函数 $\texttt{n}^0$, $\texttt{log}_2\texttt{n}$, $\texttt{n}^1$, $\texttt{nlog}_2\texttt{n}$, $\texttt{n}^2$, $\texttt{n}^3$以及 $2^{\texttt{n}}$.

19.3 给出 d, m 和 c, 证明下列函数具有所示的复杂度

```
f1 n =  0.1 * n^5 + 31*n^3 + 1000    O(n^6)
f2 n = 0.7 * n^5 + 13*n^2 + 1000     Θ(n^5)
```

19.4 证明对于任意k>0, $\texttt{n}^{\texttt{k}} \ll 2^{\texttt{n}}$。通过两边取对数，证明 $\texttt{log}_2\texttt{n} \ll \texttt{n}^{\texttt{k}}$。

19.5 证明

$$\texttt{log} \equiv \texttt{ln} \equiv \texttt{log}_2$$

事实上，任何底的对数都具有相同的增长率。

19.6 函数fib定义如下

```
fib 0 = 0
fib 1 = 1
fib m = fib (m-2) + fib (m-1)
```

证明：对于所有的k, $\texttt{n}^{\texttt{k}} \ll$ `fib n`

19.7 证明 $\ll$ 具有传递性，即如果 $f \ll g$, $g \ll h$, 则 $f \ll h$。证明 $\equiv$ 是一个等价关系。

19.8 证明：如果f是 $O(\texttt{g})$, 则f的任何常数倍也是同阶的。如果f1和f2是 $O(\texttt{g})$, 则它们的和以及差也是同阶的。将 O 替换成 Θ 上述结论还成立吗？

19.9 假设f1是 $O(\texttt{n}^{\texttt{k1}})$, f2是 $O(\texttt{n}^{\texttt{k2}})$, 证明它们的乘积`f n = f1 n * f2 n`是 $O(\texttt{n}^{(\texttt{k1}+\texttt{k2})})$。

19.10 利用归纳法证明下列等式

$$\begin{aligned}
1 + 2 + 4 + ... + 2^{\texttt{n}} &= 2^{(\texttt{n}+1)} - 1 \\
1 + 2 + ... + \texttt{n} &= \texttt{n} * (\texttt{n} + 1)\texttt{`div`}2 \\
1^2 + 2^2 + ... + \texttt{n}^2 &= \texttt{n} * (\texttt{n} + 1) * (2 * \texttt{n} + 1)\texttt{`div`}6 \\
1^3 + 2^3 + ... + \texttt{n}^3 &= (\texttt{n} * (\texttt{n} + 1)\texttt{`div`}2)^2
\end{aligned}$$

§19.2 计算的复杂度

如何度量我们定义的函数的复杂度呢？一种方法是使用 Haskell 的实现获得有关计算的某些诊断信息。在 Hugs 下，使用设置:`set +s`可以得到这样的信息。虽然如此，我们希望得到有关计算的更清晰画面。我们选择分析计算的过程。这里有三个供选择的度量：

- 计算一个结果所花的时间用惰性计算中的步数度量；
- 计算使用的空间可以用两种方式度量。首先，成功完成计算所需空间存在一个下界。在计算过程中，所计算的表达式会扩展或者缩小，我们需要足够多的空间存放计算过程中的最大表达式，常称之为计算的驻区，我们将称之为空间复杂度。
- 我们也可以度量计算所使用的总空间，它以某种方式反映了计算所占的总区域，这对函数语言的实现者有意义，但对于一般用户前两个是最重要的度量。

那么如何度量一个函数的复杂程度呢？

复杂度度量

对于一个函数 f，将以上描述的时间复杂度和空间复杂度视为 f 的输入规模的函数，这些函数分别称为 f 的时间复杂度和空间复杂度。一个数的规模是数本身，一个列表的规模是它的长度，而一颗树的规模是它所含的结点数。下面是一些计算复杂度的例子。

例 19.4

首先看 fac 一例。

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

它的求值过程如下

```
fac n
⇝ ...
⇝ n * ((n-1) * ...  * (2 * (1 * 1)) ...)      (facMax)
⇝ n * ((n-1) * ...  * (2 * 1) ...)
⇝ n * ((n-1) * ...  * 2 ...)
⇝ ...
⇝ n!
```

上述计算中包含2*n+1步，而且最大的表达式 (facMax) 包括 n 个乘法符号。所求，fac 的时间和空间复杂度均为 $\Theta(n^1)$，即线性的。

例 19.5 再看插入排序的例子。

```
iSort []     = []
iSort (x:xs)  = ins x (iSort xs)

ins x [] = [x]
ins x (y:ys)
  | (x<=y)    = x:y:ys
  | otherwise = y:ins x ys
```

一般的插入过程为

```
iSort [a1, a2, ···, an−1, an]
⇝ ins a1 (iSort[a2, ···, an−1, an])
⇝ ...
⇝ ins a1 (ins a2 ···(ins an−1 (ins an [])) ···))
```

然后是 n 个 ins 的计算。函数 ins 具有什么样的行为呢？观察一般的插入

```
ins a [a1, a2, ···, an−1, an]
```

其中假设 $[a_1, a_2, \cdots, a_{n-1}, a_n]$ 是有序的。这里有三种可能性：

- 在最好的情况下，即 $a \leq a_1$ 时，计算需要 1 步；
- 在最坏的情况下，即 $a \geq a_n$ 时，计算需要 n 步；
- 在平均情况下，计算需要n/2步。

以上结果对 iSort 意味着什么呢?

- 在最好的情况下，每个 ins 需要 1 步，所以，在这种情况下 iSort 需要 n 步，即 O(n)。
- 另一方面，在最坏情况下，第一个 ins 需要 1 步，第二个 ins 需要 2 步，依次类推。利用 19.1 节的统计公式，在这种情况下 iSort 需要 $O(n^2)$ 步。
- 在平均情况下，ins 的总步数为

  ```
  1/2 + 2/2 + ... + (n-1)/2 + n/2
  ```

  此和的阶也是 $O(n^2)$。

由此看出，在大多数情况下，算法需要平方阶的时间，但是在一些特殊情况下，当对一个（几乎）有序列表排序时，时间复杂度是列表长度的线性阶。在所有的情况下，空间复杂度是线性阶的。

例 19.6

用 ++ 求两个列表连接的过程如下

```
[a1, a2, ···, an−1, an] ++ x
⇝ a1 :([a2, ···, an−1, an] ++ x)
⇝ a1 :( a2 :[a3, ···, an−1, an ] ++ x)
⇝ ...  n-3步...
⇝ a1 :( a2 : ··· :( an :x)··· )
```

可见时间复杂度是第一个列表长度的线性阶。

例 19.7

排序算法 quickSort 的定义如下

```
qSort []     = []
qSort (x:xs) = qSort [z|z <- xs, z<=x] ++ [x] ++ qSrot [z|z<-xs,z>x]
```

当列表有序并且不包含重复元素时，计算过程如下：

qSort [$a_1, a_2, ..., a_{n-1}, a_n$]
$\rightsquigarrow \cdots$ n 步 $\cdots$
$\rightsquigarrow$ [] ++ [a_1] ++ qSort [$a_2, ..., a_{n-1}, a_n$]
$\rightsquigarrow \cdots$ n-1 步 $\cdots$
$\rightsquigarrow a_1$:([] ++ [a_2] ++ qSort [$a_3, ..., a_n$])
$\rightsquigarrow \cdots$
$\rightsquigarrow a_1$:(a_2:(a_3:$\cdots a_n$:[])
$\rightsquigarrow$ [$a_1, a_2, ..., a_{n-1}, a_n$]

因为计算步数是1+2+...+n, 所以在这种情况下的复杂度是平方阶的。在平均情况下, 排序过程如下

qSort [$a_1, a_2, ..., a_{n-1}, a_n$]
$\rightsquigarrow$ qSort [$b_1, \cdots, b_{n/2}$] ++ [a_1] ++ qSort [$c_1, \cdots, c_{n/2}$]

其中列表被分为两半。将列表折分成两半以及将其连接在一起均需要 O(n) 步。如 19.1 节所述, 将列表分为单元素列表需要 $\log_2 n$次步计算, 由此得, 在平均情况下, quickSort 需要 O($n\log_2 n$) 次, 但是在最坏情况下 (已经有序的列表) 需要平方阶步。

对数阶是分治法算法的一个特征：我们将问题分解为两个更小的问题, 然后解决这些小问题并将部分问题的解组合成原问题的解, 其结果是一个相对有效的算法。分治法能够在 O($\log_2 n$) 步 (而不是 O(n) 步) 化简到最简单的情况。

习题

19.11 估算下面定义的两个置逆函数的复杂度

```
rev1 []       = []
rev1 (x:xs)   = rev1 xs ++ [x]

rev2           = shunt []
shunt xs []    = xs
shunt xs (y:ys) = shunt (y:xs) ys
```

19.12 乘法可以用不断相加来实现

```
mult n 0 = 0
mult n m = mult n (m-1) + n
```

下面是俄罗斯乘法的定义

```
russ n 0 = 0
rss n m
  | (m `mod` 2 == 0) = russ (n+n) (m `div` 2)
  | otherwise        = russ (n+n) (m `div` 2) + n
```

试估算这两种乘法算法的时间复杂度。

19.13 估算Fibonacci函数的时间复杂度。

19.14 证明下面定义的归并排序算法的最坏情况复杂度为 $O(\mathtt{n}\log_2\mathtt{n})$

```
mSort xs
  | (len < 2)     = xs
  | otherwise     = mer (mSort (take m xs)) (mSort (drop m xs))
    where
    len = length xs
    m = len `div` 2

mer (x:xs) (y:ys)
  | (x<=y)     = x:mer xs (y:ys)
  | otherwise  = y:mer (x:xs) ys
mer (x:xs) []  = (x:xs)
mer []     ys  = ys
```

§19.3 集合的实现

在 16.8 节我们介绍了抽象数据类型 Set，并给出用无重复元素的有序序列表示集合的实现。我们也可以用允许重复元素的列表实现集合。

```
type Set a   = [a]
empty        = []
memSet       = member
inter xs ys  = filter (member xs) ys
union        = (++)
subSet xs ys = and (map (member ys) xs)
eqSet xs ys  = subSet xs ys && subSet ys xs
makeSet      = id
mapSet       = map
```

我们还可以在 16.7 节查找树的基础上实现集合。我们将这些实现的时间复杂度列表如中：

	列表	有序列表	查找树（平均）
memSet	$O(\mathtt{n}^1)$	$O(\mathtt{n}^1)$	$O(\mathtt{log_2n})$
sunSet	$O(\mathtt{n}^2)$	$O(\mathtt{n}^1)$	$O(\mathtt{nlog_2n})$
inter	$O(\mathtt{n}^2)$	$O(\mathtt{n}^1)$	$O(\mathtt{nlog_2n})$
makeSet	$O(\mathtt{n}^0)$	$O(\mathtt{nlog_2n})$	$O(\mathtt{nlog_2n})$
mapSet	$O(\mathtt{n}^1)$	$O(\mathtt{nlog_2n})$	$O(\mathtt{nlog_2n})$

由表中看出，不存在明显的好或者坏的选择。我们应该根据所使用的运算来选择不同的实现。这是提供抽象数据类型的另一个原因：它使得用户可以根据自己的需要选择不同的实现，而无需改变用户程序。

习题

19.15 对于集合的两个列表实现，确认上表的时间复杂度。

19.16 对于查找树的实现，实现运算subSet、inter、makeSet和mapSet，并估算你的实现的复杂度。

19.17 使用无重复列表实现集合，并估算你实现的函数的时间复杂度。

§19.4 空间行为

估算计算一个结果所需的空间的原则是度量计算过程中生成的最大表达式。如果计算的结果是数或者布尔值，则这样的度量是精确的；如果计算的结果是一个数据结构，如列表，则这样的度量是不精确的。

惰性计算

在 17.1 节我们对惰性计算的解释是部分结果尽可能快地被打印输出。部分结果一旦输出便不再占用空间。在统计空间复杂度时，我们必须明白这一点。以列表 [m..n] 为例，其定义为

```
[m .. n]
  | n>=m      = m:[m+1..n]
  | otherwise = []
```

计算 [1..n] 得到

```
[1 ..  n]
        ?? n>=1
⇝ 1:[1+1 ..  n]
        ??  n>=2
⇝ 1:2 :[2+1 ..  n]
⇝ ...
⇝ 1:2:3:...:n:[]
```

其中下划线部分可以输出。度量空间复杂度时考虑未划下划线部分，它们占用的空间是常数，所以空间复杂度为 $O(n^0)$。上述计算约需要 2*n 步，所以时间复杂度是线性的。

用 where 子句存储值

考虑下列例子

```
exam1 = [1 .. n] ++ [1 .. n]
```

计算上式的时间复杂度为 O(n)，空间复杂度为 O(1)，但是我们必须计算表达式 [1..n] 两次。假如我们计算

```
exam2 = list ++ list
        where
        list = [1 .. n]
```

其效果是计算表达式 [1..n] 一次，将它的值存储起来以便于两次使用。不幸的是，这也意味着计算 list 以后，将整个列表存储，使空间复杂度成为 O(n)。

这是一个普遍现象。如果我们用 where 存储一个值，我们必须付出它占用的空间的代价：如果空间足够用，这样做是很合理的；如果空间不够用，一个计算会因为缺乏足够的空间而失败。

上述问题有时会变得更糟。例如

```
exam3 = [1  .. n] + [last [1 .. n]]
exam4 = list ++ [last list]
        where
        list=[1 .. n]
```

其中 last 返回一个非空列表的最后一个元素。exam3 所需的空间是 O(1)，而exam4所需的空间是 O(n)。因为尽管我们只需要 list 的最后一个值，但是必须保存整个 list 的值。这种保存一个大结构，而只需要其中一小部分的特性称为 dragging problem。在上例中问题很明显，但在大型系统中很难找出dragging problem的根源。

上述例子的经验是，虽然避免重复计算一个简单值总是明智的，但存储一个复合值，如列表和元组，会增加程序的空间开销。

节省空间?

如 19.2 节所见，朴素的阶乘算法需要构造表达式

```
n * ((n-1) * ... * (1*1) ...)
```

故其空间复杂度为 O(n^1)。我们也可以在构造表达式过程中完成乘积的计算，即使用下列函数

```
newFac :: Int -> I n t
newFac n = aFac n 1

aFac 0 p = p
aFac n p = aFac (n-1) (p*n)
```

阶乘的计算可以表示成aFac n 1。以下是计算过程

```
newFac n
⇝ aFac (n-1) (1*n)
      ??  (n-1)==0 ⇝ False
⇝ aFac (n-2) (1*n*(n-1))
⇝ ...
⇝ aFac 0 (1*n*(n-1)*(n-2)*...*2*1)
⇝ (1*n*(n-1)*(n-2)*...*2*1)          (needVal)
```

可以看出，这个程序具有同样的效果：程序仍然构造了一个未计算的大表达式。其原因是，只有在计算进行到(needVal)这一步时才能决定这个表达式是否有被计算的必要。

如何改变上述状况呢？我们应该强制每个乘积的计算及时完成，方法是增加一个测试（在下一节介绍另外一种方法）

```
aFac n p
  | p==p       = aFac (n-1) (p*n)
```

利用这个新定义，阶乘 4 的计算过程如下

```
aFac 4 1
⇝ aFac (4-1) (1*4)
    ?? (4-1)==0       ⇝ False
    ?? (1*4)==(1*4)   ⇝ True         (eqTest)
⇝ aFac (3-1) (4*3)
    ?? (3-1)==0       ⇝ False
    ?? (4*3)==(4*3)   ⇝ True         (eqTest)
⇝ aFac (2-1) (12*2)
⇝ ...
⇝ aFac 0 (24*1)
⇝ (24*1)
⇝ 24
```

其中(eqTest)表示测试p==p的进行，所以乘积的计算被执行。由此看出，这样的程序具有更好的（常数）空间复杂度。

习题

19.18 估算下列函数的空间复杂度

```
sumSquares :: Int -> Int
sumSquares n = sumList (map sq [1 .. n])

sumList = foldr (+) 0
sq n = n*n
```

19.19 估算第七章文字处理函数的复杂度。

§19.5 再谈折叠

第九章介绍了将一个运算或函数折叠入列表的模式。本节讨论两个标准折叠函数的复杂度，以及在程序设计中如何选择它们。首先，我们定义表达一个函数需要计算它的参数的概念。这个概念对于充分理解折叠是很关键的。

严格性

称一个函数关于一个参数是严格的，如果当此参数无定义时，函数也无定义。例如，(+) 关于两个参数均是严格的，而 (&&) 仅对于第一个参数是严格的。后者的定义如下

```
True  && x = x
False && x = False                          (andFalse)
```

第一个参数上的模式匹配迫使函数对于第一个参数是严格的，但等式 (andFalse) 表示当第二个参数为 undef 时，函数仍有可能得到一个结果，故函数对于第二个参数不是严格的。如果一个函数对于一个参数不是严格的，则称之为对此参数 **非严格的** 或者 **惰性的**。

从右边折叠

折叠的定义如下

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


foldr f st []     = st
foldr f st (x:xs) = f x (foldr f st xs)
```

此函数具有广泛的适用性。列表上的插入排序可定义如下

```
iSort = foldr ins []
```

而且列表上的任何原始递归函数可以利用 foldr 定义。

如果用中缀形式表示函数应用，则有

$$\texttt{foldr f st [} a_1, \cdots, a_n \texttt{]} \rightsquigarrow a_1 \texttt{ `f` (} a_2 \texttt{ `f` ... `f` (} a_{n-1} \texttt{ `f` (} a_n \texttt{ `f` st))...)} \quad \text{(foldr)}$$

由此看出'r' 的含义：初始值 st 出现在列表元素的右边，且括号是右结合的。由 (foldr) 可看出，如果 f 关于第二个参数是惰性的，给定列表的头元素，则上式仍有可能给出结果。例如，map 可以定义如下

```
map f = foldr ((:).f) []
```

计算 `map (+2) [1..n]` 的过程如下：

```
foldr ((:).(+2)) [] [1 ..  n]
⇝ ((:).(+2)) 1 (foldr ((:).(+2)) [] [2 ..  n])
⇝ 1+2 :  (foldr ((:).(+2)) [] [2 ..  n])
⇝ 3 :  (foldr ((:).(+2)) [] [2 ..  n])
⇝ ...
```

如 19.4 节所见，因为列表的元素在计算时被输出，所以其空间复杂度为 $O(n^0)$。将一个严格运算折叠入列表结果会怎么样呢？19.2 节中的 fac 可以重新定义为

```
fac n = foldr (*) 1 [1 .. n]
```

可以看出其结果是时间复杂度为 $O(n^1)$，这是因为括号向右结合，在整个表达式形成之后才可以计算乘积。为此我们定义一个从左边折叠的函数。

从左折叠

我们可以定义从左折叠

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f st []      = st
foldl f st (x:xs)  = foldl f (f st x) xs
```

其计算过程如下

$$\texttt{foldl f st [} a_1,\cdots,a_n \texttt{]} \rightsquigarrow \texttt{(...((st `f` } a_1 \texttt{) `f` } a_2 \texttt{) `f` ... `f` } a_{n-1} \texttt{) `f` } a_n \quad \texttt{(foldl)}$$

我们可以用 foldl 计算阶乘的例子，其结果为

```
foldl (*) 1 [1 ..  n]
⇝ foldl (*) (1*1) [2 ..  n]
⇝ ...
⇝ foldl (*) (...((1*1)*2)*...*n) []
⇝ (...((1*1)*2)*...*n)
```

如 19.4 节所述，这里的困难是 foldr 关于第二个参数是非严格的。使用标准函数 seq

```
seq :: a -> b -> b
```

我们可以使其在第二个参数上是严格的。`seq x y` 的结果是在返回 y 之前先计算 x。函数 seq 是多态的，故可应用于任意类型。如果我们定义

```
strict :: (a -> b) -> a -> b
strict f x = seq x (f x)
```

则 `strict f` 是 f 的**严格**版本，即在返回结果`f x`之前先计算 x。因此，我们可以定义 foldr 的严格版 foldr'

```
foldl' :: (a -> b -> a ) -> a -> [b] -> a
foldl' f st []     = st
foldl' f st (x:xs) = strict (foldl' f) (f st x) xs
```

再次计算上述表达式：

```
foldl' (*) 1 [1 ..  n]
⇝ foldl' (*) 1 [2 ..  n]
⇝ foldl' (*) 2 [3 ..  n]
⇝ foldl' (*) 6 [4 ..  n]
⇝ ...
```

显然，这个计算的空间复杂度是 $O(n^0)$。我们从这些例子中可以得出什么结论呢？

折叠的设计

当我们折叠一个严格函数时，使用 foldr 将形成一个列表规模的表达式，所以使用 foldr' 总是值得的。这些例子包括 (+), (*) 等。

我们看到，使用 foldr 定义 map 时，我们可以在构造整个列表参数之前输出结果。如果我们在 map 中使用 foldr' ，那么我们在输出任何结果之前不得不遍历整个列表，因为 foldr' 的计算遵循下列模式

```
foldl' f st1  xs1
⇝ foldl' f  st2  xs2
⇝ ...
⇝ foldl' f  stk  xsk
⇝ ...
⇝ foldl' f  stn []
⇝ stn
```

所以，在 map 的情况，foldr 是显然的选择。

一个更有趣的例子是判定一个列表中所有元素均为 True。当然，我们需要折叠 `(&&)`。但是，应该使用 foldr 还是 foldr' 呢？后者使用常数空间，但需要检查整个列表。因为 `(&&)` 关于第二个参数是惰性的，我们不必检查由列表的剩余部分返回的值。例如，

```
 foldr (&&) True (map (==2) [2 .. n])
⇝  (2==2) && (foldr (&&) True (map (==2) [3 .. n]))
⇝  True && (foldr (&&) True (map (==2) [3 .. n]))
⇝  foldr (&&) True (map (==2) [3 .. n])
⇝  (3==2) && (foldr (&&) True (map (==2) [4 .. n]))
⇝  False && (foldr (&&) True (map (==2) [4 .. n]))
⇝ False
```

这个版本使用常数空间，并且不必检查整个列表，显然是最好的选择。除 (+) 和 (*) 这些例子外，还有许多其他例子使用 foldl' 是较好的选择，其中包括

- 倒置一个列表。使用 foldr 时我们必须在列表 x 的尾部加上一个元素 a。运算`x++[a]`关于 x 是严格的。而'cons' 运算 (:) 对于它的列表参数是惰性的。
- 将一个数字列表如"7364" 转换为一个数，对于首段"736" 及最后字符'4' 的转换都是严格的。

因为 foldl' 在给出任何结果之前消耗整个列表，所以在编写交互式系统中使用无穷列表或者部分列表的函数中不可以使用 foldl' 。

```
fibP 3
= (y,x+y)
  where
  (x,y) = fibP 2
        = (y1,x1+y1)
          where
          (x1,y1) = fibP 1
                  = (y2,x2+y2)
                    where
                   (x2,y2) = fibP 0
                           = (0,1)
                  = (1,1)
        = (1,2)
= (2,3)
```

图 19.2: 计算fibP 3

习题

19.20 使用foldr和foldl'定义将列表逆转的函数和将数字列表转换为数的函数，并通过计算的方式比较它们的效率。

19.21 定义插入排序时，使用foldr或者foldl'更好？试说明理由。

19.22 foldr和foldl'的结果间有何关系？

19.23 当折叠具有下列性质的函数时foldr和foldl'有何关系？

满足结合律：　　a `f` (b `f` c) = (a `f` b) `f` c

具有单位元st：　　st `f` a = a = a `f` st

满足交换率：　　a `f` b = b `f` a

假如上述性质同时成立，情况又如何？

§19.6 避免重复计算：记忆

本节讨论在求表达式值的过程中避免重复计算的一般方法，特别是在使用递归解的情况，子问题的解可以重复使用。

我们继续讨论 Fibonacci 函数。

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-2) + fib (n-1)
```

这个定义的效率非常低。计算fib n需要调用fib (n-2)和fib (n-1)，而计算后者又需要调用fib (n-2)，而且每次调用fib (n-2)需要调用两次fib

(n-3)。fib的时间复杂度大于任何幂函数。如何避免这种重复计算呢？我们介绍两种改进定义的方法；第一种方法使每个调用返回一个复杂数据结构，第二种利用一个无穷列表记录函数的所有值。

首先，我们注意到函数在 n 处的值需要前面的两个值；为此我们让结果返回两个值。

```
fibP :: Int -> (Int, Int)
fibP 0 = (0,1)
fibP n = (y,x+y)
         where
         (x,y) = fibP (n-1)
```

图 19.2 是一个求值的例子，其中局部变量 x 和 y 的不同出现使用了 x1, y1 等，虽然这些不是必需的，但可使得变量的不同出现更清晰。

另外一种方法是直接定义 Fibonacci 值的列表 fibs。以上函数 fib 的值现在成为这个列表中某一点的值：

```
fibx :: [Int]
fibs!!0    = 0
fibs!!1    = 1
fibs(n+1)  = fibs!!n + fibs!!(n+1)
```

上述定义给出列表的一种*描述*，但这种描述是不可运行的。前两个描述说明fibs=0:1:rest，第三个等式说明 rest 是什么。fibs 的第 (n+2) 个元素是 rest 的第 n 个元素；同理，fibs 的第 (n+1) 个元素是 (tail fibs) 的第 n 个元素。因此，对于每个 n

```
rest!!n = fibs!!n + (tail fibs)!!n
```

这表明每个元素是两个列表相应元素之和，即

```
rest = zipWith (+) fibs (tail fibs)
```

。由此得到计算 Fibonacci 数列的**过程网**。

```
fibs :: [Int]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

这个过程的时间复杂度是线性的，空间复杂度为常数。注意前一个二元组解的时间复杂度和空间复杂度均为线性的；因为在返回结果之前，我们需要完成嵌套调用 fibP。

动态程序设计

本节的例子说明一种称为**动态编程**(**dynamic programming**) 的解题方法。使用动态编程时我们将一个问题分解为一些子问题，但是，正如 Fibonacci 例子所示，这些子问题一般地不是独立的。一个朴素解将包含许多冗余，在动态编程中我们通过建立子问题解的表来消除冗余。

考虑求两个列表的最大公共子序列的长度，子序列的元素不必是相邻的。

在下面的例子中

```
[2,1,4,5,2,3,5,2,4,3]    [1,7,5,3,2]
```

最大公共序列为 [1,5,3,2]，其长度为 4。这个问题并不单单是一个“练习”；这个问题的解可用于寻找两个文件的公共文本行，这也是 Unix 程序 diff 的基础。程序 diff 可用于比较存储于不同文件的不同版本的程序。

图 19.3 给出一个朴素解。有趣的是定义中的第三个等式。当列表的第一个元素相同时，这个元素必然包含在最大子序列中，所以我们将此元素加到列表尾的解上。问题在于列表的第一个元素不相同的情形。我们或者去掉 x，或者去掉 y，然后求这两个可能情况下的最大子列表，当然，这里是冗余计算的根源 – 在算法中每一种情况均可能重复计算`mLen xs ys`。如何避免这种情形呢？我们将把这些结果存储在一个表中，这个表可用列表的列表表示。一旦一个结果已存储于列表，我们便不需再计算它。

我们将解重新写作 maxLen 作为中间步骤。maxLen 使用了列表下标，使得

```
maxLen xs ys u v
```

表示列表`take u xs`和`take v ys`的最长公共子列表。函数的定义在图 19.3 中给出。这个定义是 mLen 的定义的直接改写。

我们要定义一个表`maxTab xs ys`，使得

```
(maxTab  xs ys)!!u!!v= maxLen xs ys u v
```

这个要求被 (maxLen.1) 至 (maxLen.4) 具体化。最简单的情况是 (maxLen.1)，即对于所有的 v

```
(maxTab xs ys)!!0!!v = 0
```

换言之，

```
(maxTab xs ys)!!0 = [0,0..]
```

所以，

```
result = [0,0..] : ...
```

等式 (maxLen.2) 至 (maxLen.4) 说明由 maxTax!! i 和 i 来定义 maxTab!!(i+1)，所以我们定义

```
maxTab xs ys
  = result
    where
    result = [0,0 .. ] : zipWith f [0 .. ] result
```

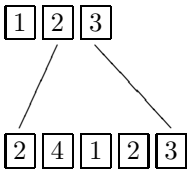
其中`f::  Int->[Int]->[Int]`是将 i 和前一个值 maxTab!!(i+1) 映射到 maxTab!!(i+1) 的函数。现在我们来定义 maxTab!!(i+1)，在定义中记作 ans。

等式 (maxLen.2) 说明 maxTab!!(i+1) 的第一个元素为 0，函数 g 将 maxTab!!(i+1)!!j 和 j 映射到 maxTab!!(i+1)!!(j+1)，其中我们可以使用命名为 prev 的值 maxTab!!i。利用这些观察，g 的定义便是 (maxLen.3) 和 (maxLen.4) 的直译：

```
ans   = 0 : zipWith g [0 .. ] ans
```

```
mLen :: Eq a => [a] -> [a] -> Int

mLen xs []        = 0
mLen [] ys        = 0
mLen (x:xs) (y:ys)
  | x==y        = 1 + mLen xs ys
  | otherwise   = max (mLen xs (y:ys)) (mLen (x:xs) ys)

maxLen :: Eq a => [a] -> [a] -> Int -> Int -> Int

maxLen xs ys 0 j = 0                                        (maxLen.1)
maxLen xs ys i 0 = 0                                        (maxLen.2)
maxLen xs ys i j
  | xs!!(i-1) == ys!!(j-1)  = (maxLen xs ys (i-1) (j-1)) + 1
                                                            (maxLen.3)
  | otherwise               = max (maxLen xs ys i (j-1))
                                  (maxLen xs ys (i-1) j)
                                                            (maxLen.4)

maxTab ::  Eq a => [a] -> [a] -> [[Int]]

maxTab xs ys
  = result
    where
    result = [0,0 .. ] : zipWith f [0 .. ] result
    f i prev
        = ans
          where
          ans   = 0 : zipWith g [0 .. ] ans
          g j v
            | xs!!i == ys!!j      = prev!!j + 1
            | otherwise           = max v (prev!!(j+1))
```

图 19.3: 求最大公共之序列算法




```
g j v
   | xs!!i == ys!!j      = prev!!j + 1
   | otherwise           = max v (prev!!(j+1))
```

最外层的定义为

```
maxTab xs ys !! (length xs) !! (length ys)
```

而且计算这个式的时间和空间均为线性的。

Haskell 提供了一个数组模块。利用数组许多算法的实现更有效，包括上例的算法。有关数组的细节参见模块 Array.hs 及其文档。

贪婪算法

用贪婪法解决动态程序设计问题的过程是，在构造最优解的过程中，选择在局部范围看起来最好的子问题解。在上例求公共序列的问题中，我们可以考虑一次扫描两个列表，不断寻找两个列表中的相同元素；对于任意 n，在比较下标为 n 的元素前，我们搜索所有下标小于 n 的下标队。在下面的例子中，贪婪法给出的解为 [2,3]，

这并不是最优的解，子序列 [1,2,3] 被遗漏了，因为我们第一次选择的相同元素为 2。这个局部选择并不包含在最优解中，不过，算法的运行效率是比较合理的。

对于局部选择总是整体解的一部分的情形，使用贪婪法可以找到最优解。我们见到的例子包括

- 第七章的文本行拆分算法，它在文本行对齐时使得字间的空格总数达到最小；
- 第十五章的 Huffman 码能给出文件的最短编码。在构造编码的过程中，我们并没有 Huffman 码的所有子集，而是从局部的最优解一步步构造而成。

习题

19.24 试实现求最大公共子序列的贪婪解，并说明在列表`[1,2,3]`和`[2,4,1,2,3]`贪婪法的运行如上述所述。

19.25 你能按照`fibP`的思想，通过返回一个复杂的（有穷）数据结构而不仅仅是一个值的方法改进求最大公共子序列的解吗？

19.26 我们在 *14.5* 节讨论了求两个串的"编辑距离"，并给出一个动态程序设计解。使用本节介绍的技术，说明你可以给出本算法的一个高效解，并给

出一个贪婪解。试比较这两个解。

19.27 在本节例子的基础上，设计一个输出两个文件的差别的程序。输出格式可以是一列匹配的行号，或者参照Unix程序diff的输出。

小结

本章探讨了惰性函数程序的效率。我们看到，许多函数的时间复杂度分析都比较容易。空间复杂度的分析更困难一些，但是我们说明了如何使用计算来估算惰性程序的空间消耗。

函数 foldl 的引入使空间的利用成为关注的焦点。严格函数和惰性函数的区分使得我们能够分析两种折叠的不同运行表现。

我们最后讨论了利用惰性无穷表记录结果并用于重用；其中从朴素的算法到高效算法的转换是系统的，也可以应用于其他领域。

本章是函数程序行为研究的简介，更多的信息，特别是有关函数数据结构的信息可参见 OKasaki(1998)。

第二十章　结论

本书涵盖了惰性语言 Haskell 函数程序设计的基本内容。书中通过下列方式说明了如何构造程序：一方面在介绍语言的新特点时我们给出大量例子，另一方面，我们提出在程序设计阶段如何设计程序的建议。我们强调了程序设计阶段是明显地介于精确地说明问题和用 Haskell 书写问题的解之间的一个阶段。

函数程序设计的威力

函数程序的模型建立在很高的抽象层。在这个抽象层，程序员专注于值之间的关系，并将其嵌入函数定义之中。与此相对的是关注事物之间关系的细节的低层模型观。例如，在 Haskell 中，列表不过是值，而在 C 和 C++ 中它们是由指针建立的数据结构，即使在 Java 中也很难建立列表的适当抽象模型。书中叙述了这种高层方法的一系列结果：

- 高阶函数与多态的结合支持通用目的函数库的构建。Haskell 引导库和列表库中包含了这样的列表函数。例如，map 函数

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

  它刻画了将一个变换应用于列表中每一个元素的模式，这种模式在列表的很多应用中被重用。

  支持重用的机制还有利用重载的类型分类。例如，使用重载定义函数

  ```
  elem :: Eq a => a-> [a] ->Bool
  ```

  其功能是利用重载的相等函数测试列表的属于关系。
- 函数的定义是表示它所定义函数的性质的等式。例如，由 map 和函数复合'.' 的定义可以证明，对于所有的函数 f 和 g

  ```
  map (f . g) = map f . map g
  ```

  证明为用户提供了程序在所有参数上会如何运行的保障，与此相对的是测试，后者只能给出程序在一个（希望）具有代表性的输入子集上如何运行的直接信息。
- 数据结构可以用直接递归的方式引进，如树和队列等，而无需考虑它们的表示。书写算法与非正式描述算法可以在同一层次进行，而在更传统的算法设计与编写中，我们必须考虑数据表示和算法实现的很多细节。

本书只能对函数程序设计这种既丰富又发达的题目一个介绍。最后一章的剩余部分将讨论函数程序设计的其他方面，同时指出网上的其他资源，有关的书籍和论文，以供进一步学习参考。

进一步的 Haskell

本书的目的是使用 Haskell 语言介绍函数程序设计思想。书中包括语言

的各个重要方面，但并非所有的方面。省略的内容包括带标记域的数据类型，类似于其他语言的记录或者结构；用于表示数据结构构造符在部分或全部参数上严格的严格记号；有关 Read 分类和数值类型及其分类的细节。

有关以上内容的信息参见 Haskell 语言报告 (Peyton Jones and Hughes 1998)。Hudak, Fasel 和 Perterson(1997) 所著的“渐进介绍”也包括有关内容的有用信息，同时提供了一个经验丰富的程序员对 Haskell 的看法。以上内容，以及其他 Haskell 资源可以在 Haskell 的网页上得到：`http://www.haskell.org`。

本书讨论了标准引导库的许多重要函数，但没有讨论 Peyton Jones 和 Hughes 所建档的函数库的内容。这些库可分为两类。首先是实用函数库，如 List.hs 包含许多列表处理函数。这些库可以由用户根据需要选择调入。

另一类库是语言的扩展库，包括数据库 Array.hs，文件的创建和处理的程序 Directory.hs 以及系统连接的库 System.hs。这些库包含在所有 Haskell 实现中，每种实现还带有特殊的扩展，通常以库模块的形式供调入使用。

未来的 Haskell

Haskell 最初是在 1987 年定义的，此后经过了修改和扩展。本书是以 Haskell 98 为基础的。Haskell 是一个包含了实验和测试过的特色的稳定系统。函数程序设计研究的进展表明，像 Haskell 这样的语言不会永远停止不前。在本书写作的时候，在多方面有重大修改和扩展的 Haskell 2 的设计正在酝酿之中。但是，未来系统很可能将支持本书所讲的特点。Haskell 的主页包含 Haskell 状况的最新可靠信息。

扩展 Haskell

如本书所述，Haskell 是一个通用的高级程序设计语言。许多实际应用需要程序处理计算机图形，修改机器的状态，或者并行处理等，Haskell 目前不直接支持这些应用。但是，Haskell 实现的扩展支持以上这些任务。Haskell 的大量应用和扩展的有关信息参见下列主页

`http://www.haskell.org/libraries.htlm` (libraries)

或者参见有关实现的文档，详情参见 Haskell 主页和附录 E。

语言通常不是独立使用的，所以它们与外部库和程序设计语言的联系很重要。这些接口问题的讨论参见 Finne(1998) 和 Meijer(1998)。人们已经开发了很多 Haskell 的图形用户接口，详情见 (library) 主页。`http://www.cse.ori.edu/~erik/Personal/cgi.htm`

其他函数程序设计语言

Haskell 是一种惰性、强类型函数程序设计语言; Miranda(Turner 1986; Thompson 1995) 是另一个这样的语言。本书的第十七章对惰性做了明确的说明，并讨论了 Haskell 与应用广泛的标准 ML(Milner et al. 1997; Appel 1993) 的共同特性。SML 是最知名的、应用广泛的严格强类型函数语言。有关它的

介绍见 Paulson(1996)。我们可以在严格的语言中描述惰性计算，而且 Haskell 提供了严格计算的方法，所以这两个学派是非常接近的。

一种尽可能避免变量的函数程序设计方式是由 Baskus(1978) 提出的。Bird 和 de Moor(1997) 最近的著作强调了上述编程方式在支持程序变换方面的优势，同时提供了扩展函数式的“关系”式程序设计。

LISP 是最早的函数语言，但不同于 Haskell 和 SML 的是，LISP 不是强类型语言。Abelson, Sussman 和 Sussman 撰写了一篇使用 LISP 的 Scheme 方言进行程序设计的优秀介绍论文。另一种类似于 LISP 的命令式语言，并用于开发电话交换系统和其他实时应用的语言是 Erlang (Armstrong, Virding 和 Williams 1993)。

有关函数程序设计语言在大规模项目中的最新应用的综述见 Runciman 和 Wakeling(1995) 以及 Hartel 和 Plasmeijer（19956）的著作，Haskell 主页也包含有关的信息。

在过去的十年中，惰性函数语言的实现技术有了长足的进展。有关内容参见两本著作 (Peyton Jones 1987, Peyton Jones 和 Lester 1992)。

函数程序设计的发展方向

本书介绍使用强类型、惰性语言的现代函数程序设计。随着这个领域的发展，新的技术和方法不断涌现。两个高级函数程序设计夏季讨论班的文集 (Jeuring 和 Meijer 1995, Launchbury, Meijer 和 Sheard 1996) 是学习以上内容的好材料。

关于函数语言在教学中的应用参见 Hartel 和 Plasmeijer 编辑的“教学中的函数语言会议文集”。函数程序设计的研究成果参见”Journal of functional programming”, `http://www.dcs.gla.ac.uk/jfp/`, 还有函数程序设计国际年会以及前述网页上的其他会议。

在计算这样的领域预测将来的发展方向是困难的，不过，很显然，函数程序设计的一个富有成果的方向是构造大型系统的一个组成部分。Fran 系统 (Elliott 和 Hudak 1997) 使用一个函数语言描述动画，这些动画最终由 C++ 的库完成。函数系统的出现，使得“或者函数式，或者非函数式，而不是两者兼有”不是一个选择，这意味着函数语言在程序员技术工具箱中占有一席之地，而且证明它们与面向对象和其他语言可以并驾齐驱。这种函数式与其他语言的联合的例子有 Haskell 与 Com 成功连接的系统 (Finne et al, 1998) 和 ActiveHaskell 系统，有关的细节见 (libraries) 网站。

另一个研究方向是函数语言的强化类型系统，它使得程序员只能编写停机程序。初看起来，这样会排除太多的实用程序，但是，使用“其数据”(Turner 1995) 和相关类型 – 结果的类型依赖于一个参数的值 (Augustsson 1998) – 我们似乎可以定义这样的实用语言，这些语言的优点是对程序的推理变得更直接，而且程序员可以更多地在程序中表达一个程序会如何运行的直觉。本书

已经显示，一个强类型语言可以在编译时捕捉许多错误，强化类型系统对此会有更大的帮助。

第三个课题是为函数程序的开发者提供支持工具。正如在惰性计算的讨论中所看到的，要预测惰性程序的空间行为常常是很困难的。这方面的有趣工作参见 Runciman 和 Röjemo (1996)。

以上这些只是函数语言的三个可能方向。显然，它们提供了一种多产的程序设计方法，并且在未来的几年后将持续成为计算机科学的重要成员。

附录一　函数式，命令式和 OO 程序设计

我们在本附录中将 Haskell 程序设计与更传统的命令式语言如 Pascal 和 C, 以及面象对象 (OO) 的语言如 C++ 和 Java 等程序设计做比较。

值与状态

考虑求自然数 0 至某个数的平方和的例子。一个函数程序直接描述所求的值

```
sumSquares :: Int -> Int
sumSquares 0 = 0
sumSquares n = n*n + sumSquares (n-1)
```

这些等式描述了对于一个给定的自然数相应的平方和是什么。首先它是一个直接的描述; 第二, 它表示对于一个非零数 n, 相应的和等于 n-1 的和加上 n 的平方。

解决这个问题的典型命令式程序可能如下

```
s := 0;
i := 1;
while i<n do begin
    i := i+1
    s := i*i + s;
end {while}
```

所求的和是变量 s 的最后值, 象计数变量 i 一样, s 的值在程序运行中是不停地变化的。程序的效果只能跟随程序中的命令对这些变量所做的修改序列来观察, 而函数程序可以读作定义平方和的一系列方程。函数程序的意义是**显式的**, 然而命令式程序的效果从程序本身看不是明显的, 它的效果是整体的。

一个更鲜明的算法是完全透明的: " 求平方和, 构造 1 到 n 的列表, 将其中的每个数平方, 然后求其和"。这个程序既不使用命令式例子中的控制流, 也不使用上例 sumSquares 中的递归, 可以写作如下

```
newSumSq :: Int -> Int
newSumSq n = sum (map square [1 .. n])
```

其中 `square x = x*x`, 运算 map 将它的第一个参数应用于列表中的每一个元素, 最后 sum 求出一个列表中数的和。读者在本书中可以找到更多这种**数据导引**程序设计的例子。

函数与变量

函数式与命令式之间的重要区别是某些术语的含义。“函数” 和 “变量” 均有不同的解释。

如前所述, 在函数程序中一个函数只是返回一个依赖于输入值的东西。在

命令式和面向对象的语言中, 如 Pacal, C, C++ 和 Java, 一个函数是完全不同的。它将返回一个依赖于它的参数的值, 同时, 它一般会修改变量的值。它不是一个纯粹的函数, 而是当它停止时返回一个值的过程。

在一个函数程序中, 一个变量代表任意的或者未知的值。在一个等式中一个变量的每一次出现有相同的解释。这里的变量类似于逻辑公式中的变量, 或者类似于数学中下列等式中的变量

$$a^2 - b^2 = (a - b)(a + b)$$

在任何时候, a 的三次出现都具有相同的值。同理, 在下到表达式中

```
sumSquares n = n*n + sumSquares (n-1)
```

n 的所有出现具有相同的值。例如,

```
sumSquares = 7*7 + sumsquares (7-1)
```

关键的是" 函数程序中的变量不变化"。

另一方面, 在命令式程序中一个变量的值在其生命周期中会变化。在上例的平方和中, 变量 s 将相继取值 0, 1, 5, ... 等。也就是说, 命令式程序中的变量确实随时间而变化。

程序验证

或许函数程序和命令式程序之间的最重要区别在于逻辑上。一个函数式定义既是一个程序, 也是描述函数**性质**的逻辑等式。函数程序是**自我描述的**。利用函数的定义, 我们可以推导出函数的其他性质。

举一个简单的例子, 对于所有的 `n>0`, 下列性质成立

```
sumSquares n > 0
```

首先,

```
sumSquares 1
= 1*1 + sumSquares 0
= 1*1 + 0
= 1
```

此时性质成立。一般地, 当 n 大于 0 时

```
sumSquares n = n*n + sumSquares (n-1)
```

这里 `n*n` 是正的, 而且如果 `sumSquares (n-1)` 是正的, 那么它们的和也是正的。我们可以用数学归纳法将此证明形式化。本书包含很多在数, 列表和树等数据结构上归纳证明的例子。

命令式程序的验证也是可能的, 但是命令式程序不象函数程序一样是自我描述的。要描述一个命令式程序的效果, 如上述求平方和的例子, 我们需要给程序加上一个逻辑公式或者断言, 以描述程序运行中各个点的状态。这些方法不仅不直接, 而且更困难, 程序验证对于 Pascal 和 C 这样的实用语言是非常困难的。程序验证的另一个方面是程序变换, 即将一个程序转变为与

其等效的程序，但是运行效率更高。这样的程序变换对于传统的命令式程序也是困难的。

记录和元组

第五章引进了 Haskell 的元组类型。特别是我们看到了下列定义

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

与 Pascal 中一个记录的说明相比较

```
type Person = record
  name  : String
  phone : String
  age   : Integer
end;
```

其中有三个需要命名的域。在 Haskell 中，元组的域可以通过模式匹配来访问，但是，如有需要，也可以定义类似的选择函数。

```
name :: Person -> String
name (n, p, a) = n
```

如果 `per :: Person`，则 `name per :: String`，类似地，在 Paskell 中如果 r 是类型为 Person 的变量，则 r.name 是一个串变量。

Haskell 98 包含命名域的记录，它们更象 Pascal 的记录。有关细节请参阅 Haskell 报告 (Peyton Jones and Hughes 1998)。

列表和指针

Haskell 包含内置的列表类型，其他类型如树等可以直接定义。因为在 Haskell 中不需要考虑 Pascal 中的指针值或者存储分配 (new 和 dispose)，所以我们可以把 Pascal 中的由指针实现的链表看作列表的实现。确实，我们可以把 Haskell 程序视为 Pascal 列表程序的设计。如果我们定义

```
type list = ^node;
type node = record
  head : value;
  tail : list
end;
```

那么我们有下列的对应表，其中 Haskell 函数 head 和 tail 给出一个列表的头和尾，

```
    []          nil
    head ys     ys^.head
    tail ys     ys^.tal
    (x:xs)      cons(x,xs)
```

函数 cons 在 Pascal 中的定义如下

```
function cons(y:value;ys:list):list
```

```
  var xs:list;
  begin
    new(xs);
    xs^.head := y;
    xs^.tail := ys;
    cons := xs
  end;
```

如果定义下列函数

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

这个函数可以很直接地转变为 Pascal 程序

```
function sumList(xs:list):integer;
  begin
    if xs=nil
      then sumList : = 0
      else sunList := xs^.head+ sumList(xs^.tail)
  end;
```

第二个例子是

```
doubleAll []     = []
doubleAll (x:xs) = (2*x) : doubleAll xs
```

对应的 Pascal 定义如下，其中使用了 cons

```
function doubleAll(xs:list):list;
  begin
    if sx=nil
       then doulbeAll := nil
       else doubleAll := cons(2*xs^.head, doubleAll(xs^.tail))
  end;
```

如果定义下列函数

```
function head(xs:list):value;
  begin
    head := xs^.head
  end;
```

```
function tail(xs:list):list;
   begin
     tail := xs^.tail
   end;
```

那么对应关系更清楚

```
function doubleAll(xs:list):list;
  begin
    if xs=nil
      then doubleAll := nil
      else doubleAll := cons(2*head(xs), doubleAll(tail(xs)))
```

```
  end;
```

即使我们使用命令式语言，函数式程序仍然是有用的，这里提供了一个有力的证据：函数程序设计语言是命令式语言的高层设计语言。做出这样的划分对于我们找到命令式程序有极大的帮助 – 我们可以将设计与低层的实现分离，使得每个程序更小，更简单因而更容易解决。

高阶函数

传统命令式语言使用高阶函数的空间很有限。Pascal, Java 和 C 允许函数作为参数，只要它们本身不是高阶函数，但是不允许返回的结果为函数。在 C++ 中可以通过重载函数应用符返回表示函数的对象。这是 C++ 标准模板库通用性之所在，但是，要实现 map 和 filter 需要使用语言更高级的特性。

控制结构如 if-then-else 与高阶函数有相似性, 因为它们可以将命令 c1,c2 等组会成新命令, 如

```
if b then c1 else c2    while b do c1
```

这就类似于 map 将函数映射到函数一样。反过来, 我们可以把 Haskell 的高阶函数看作可以自己定义的**控制结构**。这或许可以解释我们构造高阶函数库的原因: 它们是编制特别系统的控制结构。包括在本书中的例子除列表函数外, 还有构造语法分析器的库 (17.5 节) 和交互式 I/O 程序 (第十八章)。

多态

多态在许多命令式语言中也是很难表达的。例如, 在 Pascal 中我们最多可以使用文本编辑器将一种列表函数通过拷贝和修改使其成为处理另一种类型列表的函数。当然，我们在修改过程中必须特别小心以保持修改的一致性。

Haskell 的多态是公认的**通用**多态: 同一段” 通用” 码可用于一批类型。一个简单例子是求列表逆的函数。

Haskell 的分类支持所谓的” 特别”(ad hoc) 多态，或者命令式语言中所称的” 多态”, 它表示不同的程序在不同的类型实现了同一个运算。一个例子是带有相等运算的类型的 Eq 分类: 判定相等的方法在不同的类型上是完全不一样的。另一种解读分类的方法是将其视为在不同类型上实现的**界面**; 在这样的解释下, 分类与面向对象语言如 Java 的界面相似。

如书中听讲, 在 Haskell 中多态是使程序可重用的机制之一, 这对于高级命令式是否成之立, 目前尚不清楚。

定义类型与分类

Haskell 的代数类型机制 (见第十四章) 包含了各种传统的类型定义。枚举类型可以用构造符均为 0 元 (无参数) 的代数类型定义; 变长记录可定义为有多个构造符的代数类型; 通常由指针实现的**递归**类型成为递归代数类型。

如我们对列表解释的一样，树等类型上的 Haskell 程序可以看作命令式语言中处理这些类型的指针实现的程序的设计。

第十六章介绍的抽象数据类型与 Modula-2 等的抽象数据类型非常相像; 我们建议的使用抽象数据类型的设计方法反映出现代命令式语言如 Ada**基于对象**的特色。

Haskell 分类系统也有面向对象的特色，如 14.6 节所见。重要的是要注意到，Haskell 的分类在某些方面与 C++ 的类是非常不一样的。Haskell 的分类是由类型组成的，而每个类型本身有它自己的成员; C++ 的类像一个类型, 它包含了对象。由于这个原因，C++ 面向对象的许多特色可以看成是 Haskell 类型设计的结果。

列表概括

列表概括提供了列表上迭代的方便记法: 类似于在数组下标上进行的 for 循环。例如，求由 xs 和 ys 构成的所有二元组之和可记作

```
[a+b| a <- xs, b <- ys]
```

迭代的顺序为: 选定 xs 中的一个值 a, 然后让 b 遍历 ys 的所有可能值; 选取 xs 中的下一个值，重复上述过程，直至耗尽 xs 的值。下面的嵌套 for 循环具有同样的功效

```
for i := 0 to xLen-1 do
  for j:=0 to yLen-1 do                (twoFor)
     write ( x[i]+y[j] )
```

其中固定 i 后让 j 循环。

在 for 循环中，我们必须遍历下标; 而列表生成器直接遍历值。列表的下标由下式给出

```
[0 .. length xs -1]
```

所以，与 (twoFor) 类似的 Haskell 记法为

```
[ xs!!i + ys!!j | i <- [0 ..length xs -1],
                | j <- [0 .. length ys -1]]
```

惰性计算

惰性计算与命令式语言不能较好地结合。例如，在 Pascal 中可以定义下列函数

```
function succ(x : interger):integer;
begin
  y     := y+1;
  succ  := x+1
end;
```

此函数在其参数上加 1，但同时有将 y 增加 1 的**副作用**。如果我们计算 f(y, succ(z))，我们不能预测其结果。

- 如果 f 先计算它的第二个参数，y 的值在被传给 f 之前增加了 1;

- 另一方面，如果 f 首先需要它的第一个参数 (或许根本不需要它的第二个参数)，那么即使 y 的值在函数调用之前增加了，传给 f 的值将不会被增加。

一般地，即使预测最简单的程序的行为也是不可能的。因为计算一个表达式可以改变系统的状态，计算表达式的顺序决定一个程序的总体效果，所以惰性实现的行为可能 (以不可预测的方式) 不同于标准的实现。

状态，无穷列表和 Monad

17.6 节介绍了无穷列表，其中的第一个例子是随机数无穷列表。这个列表可提供给一个需要随机数的函数，由于惰性计算策略的使用，这些数只有在需要时才会生成。

如果用命令式语言实现这个列表，我们很可能会用一个变量存储最后产生的随机数，并且在每次需要随机数时更新这个变量。我们可以把无穷列表视为可提供变量所取的所有值的单一结构，因此我们不需要记录状态，这是命令式观点的一个**抽象**。

结论

显然，在函数式和命令式之间既有区别又有对等。一个系统的函数式观点常常是高层的，即使我们最后要得到一个命令式解，一个函数式的设计或者**原型**仍然是非常有用的。

我们看到 Monad在函数式语言中可用于提供命令式特点的接口。Haskell 的许多实现都提供了这样的机制，因此在没有失去函数式语言的纯粹性的同时，将两个是重要的程序设计天地的优点结合了起来。其他语言如 Standard ML(Milner, Tofer and Happer 1990) 结合了函数式和命令式，但是，这些系统往往失去了纯函数性质。

有趣的是现代函数式程序设计语言中的思想对 Java 扩展设计的影响。Java 的主要缺陷之一是缺乏一种通用机制; Pizza 语言 (Odersky and Wadler 1997) 中加入了这种机制以及 Haskell 式的模式匹配, 而旦 Pizza 是通用扩展 Java, GJ, `www.cs.bell-labs.com/who/walder/pizza/gj/`, 的先行者。

附录二　名词解释

附录三　Haskell 运算符

下表按结合力递减序列出 Haskell 的运算符。有关结合性和结合力的讨论参见 3.7 节。

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

本书还定义了下列运算符

9	`>.>`		
5			`>*>`

运算符由下列符号构成

```
! # $ % & * + . / < = > ? \ ^ | : - ~
```

但不能由 : 开始; : 是中缀构造符的开始符。用户可定义 - 和 !, 但是它们在某些情形下有特珠的含义, 建议不使用它们。最后，有些符号的组含是保留的，请勿使用，如 `.. :: => = @ \ | ^ <- ->`。

若要改变某个运算符的结合性与结合力, 如 `&&&`, 可做如下说明

```
infixl 7 &&&
```

它表明 `&&&` 的结合力为 7, 并且是左结合的。我们还可以用 infix 说明一个运算符是非结合的, 用 infixr 说明它是右结合的。结合力的缺省值为 9。这些说明也可用于反引号括起来的函数名, 例如

```
infix 0 ‘poodle‘
```

附录四　理解程序

本附录将帮助读者面对不熟悉函数定义时如何理解函数程序。对于一个定义, 我们有很多事情可做, 我们将对此做一一介绍。给定一个函数程序, 如

```
mapWhile :: (a -> b) -> (a -> Bool) -> [a] -> [b]

mapWhile f p []     = []                          (mapWhile.1)
mapWhile f p (x:xs)
   | p x            = f x : mapWhile f p xs       (mapWhile.2)
   | otherwise      = []                          (mapWhile.3)
```

我们可以用许多互补的方式理解它。我们可以阅读程序本身, 应用程序计算一些例子, 证明程序的性质, 还可以估算它的时间复杂度和空间复杂度。

阅读程序

除注释外, 程序本身是它的最重要文档。

类型说明给出输入和输出类型的信息：对于 `mapWhile`, 我们必须提供三个参数:

- 一个函数, 如 f, 其类型为 `a -> b`;
- 类型 a 的对象的性质, 它是一个由类型 a 到 Bool 的函数; 以及
- a 上对象的列表。

它的输出是类型 b(即 f 的输出类型) 上的列表。

函数定义本身用于给出 `mapWhile` 的值, 但是也可以将其读作对程序的描述。

- 在 `[]` 上的结果为 `[]`;
- 在一个非空列表上, 如果头元素 x 具有性质 p, 则根据 `(mapWhile.2)`, `f x` 便是结果的头元素, 结果的尾部由函数在 xs 上的递归调用求得;
- 如果 x 不具有性质 p, 则结果为 `[]`。

这个函数定义包含了程序的运行的完整描述, 但是, 我们还可以用特殊的例子演示程序。

利用程序做计算

通过计算具体的例子, 我们可以获得程序功能后的更具体的印象。例如,

mapWhile (2+) (>7) [8,12,7,13,16]
⇝ 2+8 : mapWhile (2+) (>7) [12,7,13,16]　　by `(mapWhile.2)`
⇝ 10 : 2+12 : mapWhile (2+) (>7) [7,13,16]　　by `(mapWhile.2)`
⇝ 10 : 14 : []　　by `(mapWhile.3)`
⇝ [10,14]

其他例子包括

mapWhile (2+) (>2) [8,12,7,13,16] ⇝ [10,14,9,15,18]
mapWhile (2+) (>2) [] ⇝ []

注意在这些例子中我们使用了 mapWhile 的下列类型

```
(Int -> Int) -> (Int -> Bool) -> [Int] -> [Int]
```

这是多态类型中变量 a 和 b 用 Int 代替后的一个特例。

对程序进行推理

通过证明一个程序可能具有的性质, 我们可以对程序获得更深入的理解。对于 mapWhile, 我们或许可以证明, 对于任意的 f, p 和有穷列表

```
mapWhile f p xs              = map f (takeWhile p xs)   (mapWhile.4)
mapWhile f (const True) xs   = map f xs                 (mapWhile.5)
mapWhile id p xs             = takeWhile p xs           (mapWhile.6)
```

事实上, `(mapWhile.5)` 和 `(mapWhile.6)` 是 `(mapWhile.4)` 的推论。

程序的行为

不难看出, 程序在最坏情况下的时间复杂度为线性的 `O(n)`, 假定列表参数长度为 n, f 和 p 的时间复杂度为 `O(1)`。

空间复杂度显得更有趣; 因为一旦一个列表的头元素产生便可将其输出, 故所需的空间是常数, 如下面下划线部分所示

mapWhile (2+) (>7) [8,12,7,13,16]
⇝ 2+8 : mapWhile (2+) (>7) [12.7.13.16]
⇝ 10 : 2+12 : mapWhile (2+) (>7) [7,13,16]
⇝ 10 : 14 : []
⇝ [10,14]

从此开始

以上的每一种观点都使我们对程序的行为有不同的理解, 但是, 当我们面对一个不熟悉的定义时, 可以通过计算各种小例子来理解程序的功能。如果我们面对的是很多函数, 可以采取自底向上的方法, 将一个计算建立在另一个之上, 逐步理解整个程序。

最重要的是认识到, 我们可以从计算代表性的例子开始理解程序。

附录五　Haskell 的实现

Haskell 已经有多个实现。本书讨论的解释器 Hugs 是由英国的诺丁汉大学和美国的耶鲁大学联合完成的。编译器是在英国的格拉斯哥大学和瑞典的查尔莫斯理工大学实现的。

Hugs

Hugs 可在下述网站得到

`http://www.haskell.org/hugs/`

有关 Uinx 版本, 请参阅安装手册。

注 12　下载 Hugs 的 Windows 版本 如果你想下载标准安装版本, 你应该下载下列两个文件之一

`selfinstall.exe`

`selfinstall.zip`

这些文件建立适当的注册项目和有关设置等。如果你下载**二进制文件**, 你必须自己处理这些设置。

你可以选择 Hugs 的缺省编辑器。最简单的办法是运行 WinHugs 并改变相应的设置。这些设置对 Hugs 和 WinHugs 均有效。由下列网站可得到免费的程序员文件编辑器:

`http://www/lancs.ac.uk/people/cpaap/pfe/`

对于 Macintosh 系统, 请参见 Hans Aberg 为 Power Macintosh OS 制做的文件

`ftp://haskell.org/pub/haskell/hugs/mac/hugs-Hune98-MacPPC-binary.sea.bin`

Hugs 的优点是编译周期快, 但是其运行速度不能与各种编译器的速度相比。

其他 Haskell 系统

其他系统包括在格拉斯哥大学开发的 Glasgow Haskell compiler (ghc)

`http://www.dcs.gla.ac.uk/fp/software/ghc/`

以及在查尔莫斯大学开发的 HBC/HBI 系统

`http://www/cs/chalmers.se/~augustss/hbc/hbc.html`

NHC13 是为实验实现而开发的" 简易" 编译器

`http://www.cs.york.ac.uk/fp/nhc13/`

进一步的信息

有关以上系统及其它实现的最新信息请参阅 Haskell 主页

`http://www.haskell.org/implementations.html`

附录六　Hugs 错误

本附录总结了 Haskell 程序设计中常见的错误以及 Hugs 系统给出的相应错误信息。

我们书写的程序常常包含错误。当遇到错误时, 系统或者停止并给出**错误信息**, 或者继续运行, 但给出**警告信息**, 告诉我们有不正常现象发生, 也许是错误。本附录选取了一些错误信息, 这些错误信息一方面是常见的, 另一方面也是需要解释的。下面的信息是不言自明的：

```
Program error: {head []}
```

错误信息大致划分成不同的领域。语法错误是由病态程序造成的, 而类型错误是因对象应用于错误的类型引起的。事实上, 一个病态的表达式通常表现为一个类型错误, 而不是语法错误, 所以, 两者间的界限并不是很楚的。

语法错误

一个 Haskell 系统设法将我们的输入与语言的语法匹配。一般地, 如果有错误发生, 那么我们输入了*不期望*(unexpected) 的东西。健入 `'2==3)'` 将产生下列错误信息

```
ERROR: Syntax error in input (unexpected ')')
```

如果一个定义不完整, 例如

```
fun x
fun 2 = 34
```

我们得到下列信息

```
Syntax error in declaration (unexpected ';')
```

其中 `';'` 是一个定义结束的标志 – 这个错误信息告诉我们一个定义在不期望的地方结束了, 因为 `fun x` 没有相应的右式。

在 where 子句中定义类型会得到下面的错误信息

```
Syntax error in declaration (unexpected keyword "type")
```

模式的语法较完整表达式的语法有更多的限制, 所以我们有下面的错误信息

```
Reppeated variable "x" in pattern
```

它表示变量 x在模式中使用了两次。

在说明常量时可能会犯错误：浮点数可能太大, 由 ASCII 码定义的字符越界:

```
Inf.0
ERROR: Decimal character escape out of range
```

并非每个字符串可用于命名; 有些词在 Haskell 中为关健字或者保留字, 如果用作标识符将出错。关键字包括

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

特殊标识符 as, qualified 和 hiding 在一定的上下文中有特别的含义, 但是可以用作标识符。

最后一个对命名的限制是构造符名和类型名必须以大写字母开始; 没有其他标识符可以如此, 所以, 如果我们定义一个名为 Montana 的函数, 将得到类似于下列的错误信息

```
Undefined constructor function "Montana"
```

类型错误

我们已经看到类型错误的一个典型例子: 在 Hugs 提示符下键入 `'c' && True` 会得到下列信息

```
ERROR: Type error in application
*** expression        : 'c' && True
*** term              : 'c'
*** type              : Char
*** does not match    : Bool
```

这是因为在需要 Bool 型的位置使用了一个 Char。其他的类型错误, 如

```
True + 4
```

会产生下列的错误信息

```
ERROR: Bool is not an instance of class "Num"
```

这个错误信息是由分类机制产生的: 系统试图将 Bool 视为数值类型 Num 分类的一个特例, 因为 `+` 是 Num 的一个运算。错误产生的原因是系统找不到 Bool 是 Num 分类的一个特例的说明。

如前所述, 语法错误也可以产生类型错误。将 `abs (-2)` 写成 `abs -2` 会产生下列错误信息

```
ERROR: a -> a is not an instance of class "Num"
```

因为语法分析器将表达式分析成 `abs::a->a` 减去 2, 而运算 `'-'` 要求其参数属于 Num, 而不是类型为 `a->a` 的函数。其他的类型错误是因为混淆 `':'` 和 `'++'` 的作用而引起的, 例如, `2++[2]` 和 `[2]:[2]`。

我们总是结出定义的类型说明, 其优点是系统很快能发现定义是否与其宣称的类型一致。例如,

```
myCheck :: Int -> Bool
myCheck n = ord n == 6
```

此定义会产生下列错误信息

```
ERROR: "error.hs" (line 8): Type error in function binding
*** term           : myCheck
*** type           : Char -> Bool
*** does not match : Int -> Bool
```

如果没有类型说明, 系统将接收此定义, 只有在使用定义时系统才给出错误信息。最后一个与类型有关的错误是由类似于下列的定义引起的

```
type Fred = (Fred, Int)                          (Fred)
```

一个递归的类型同义词, 其错误信息为

```
ERROR "error.hs" (line 11): Recursive type synonym "Fred"
```

(Fred) 的效果可以用下列代数类型描述

```
data Fred = Node Fred Int
```

定义引入构造符 Node 以识别这个类型的对象。

程序错误

当完成了一个语法和类型正确的脚本后, 在求一个系统可接受的表达式的值时, 错误仍然会产生。

第一类错误源于定义未包含所有的情况。如果一个函数的定义如下

```
bat [] = 45
```

将此函数应用于 `[34]`, 我们会得到下列信息

```
Program error: {bat {dict} [Num_Int_fromInt 34]}
```

它表示在这里计算不能继续进行下去, 因为函数的定义没有包括非空列表的情况。类似的错误也可由预定义函数引起, 如 head。

其他的错误是因为**算术约束条件**不满足引起的。这些约束条件包括列表下标越界, 除数为 0, 在应该使用整数的地方使用了分数, 以及浮点数运算越界; 这些错误信息都具有相同的形式

```
Program error: PreludeList.!!: index too large
Program error: [primDivInt 3 0]
```

Haskell 采用按需计算的策略, 所以, 如果一个脚本使用了一个没有定义的名则不构成错误, 只有在计算中需要这个名的值时才会产生下列信息

```
ERROR: Undefined variable "cat"
```

模块错误

语句 module 和 import 可以造成各种错误信息: 文件可能不存在或者有错误; 命名可能被重复调入, 或者输入的一个别名引起了命名冲突。这些错误信息是易于理解的。

系统信息

对某些命令和中断的响应, 系统将产生一些信息, 包括目前任务的中断

```
^C{Interrupted!}
```

以及计算所消耗的空间超出可利用的范围。解决后一个问题的办法是增加堆的大小。使用命令 `:set` 可显示堆的大小以及系统的其他参数的设置。此命令给出的信息显示如何改变堆的大小和系统的其他参数。

如果设置参数时选择了 `+s`, 系统将输出下列形式的诊断信息

```
(2 reductions, 8 cells)
```

其中的化简 (reductions) 数目表示计算步数, 单元 (cells) 数表示空间消耗总量。

如第十九所述, 一个函数的空间复杂度可以由计算它的最小堆的大小来度量, 不过系统没有给出这样的直接度量。

索 引

白盒测试, 70
保留字, 56
边界规则, 55
编辑距离, 230
变量, 22
标识符, 20, 56
标准函数库, 37
标准引导库, 37
布尔, 43
部分地应用, 157
部分运算 (operator sections), 160
参数, 16
测试, 27
测试组, 70
插入, 115
成功的列表, 304
程序, 15
程序变换, 175
程序错误, 41
程序设计语言, 15
抽象数据类型, 244, 260
传递闭包, 285
传递的, 285
代数 (数据) 类型, 211
递归步, 65
递归基, 64
定义, 20
动态编程 (dynamic programming), 370
动态捆绑, 196
多继承, 190
多态, 75, 87
多元数组, 75
多元组类型, 75
惰性的, 366
二元, 58
反思, 179
反引号, 45
仿真, 231
非 Curry(uncurrying), 165
非退化的, 53
非严格的, 366
分情况, 43
符号测试, 127
符号计算, 126
浮点, 43
浮点数, 51
高阶, 141
高阶的, 89
构造符, 56, 110, 211
构造符类, 346
构造符模式, 111
构造函数, 214
管道 (pipeline), 170
规格说明, 70
函数, 16
函数程序, 20
函数程序设计, 16
函数的合成, 23
函数的应用, 16
黑盒测试, 70
回顾, 179
基本类型, 43
极限, 285
计算, 28
继承, 189
检测, 199
建模, 16
脚本, 31
结果, 16
结合力, 57
结合性, 56
局部, 95, 100
科学计法, 51

可视, 243
快速排序, 118
类型, 18, 20
类型变量, 87
类型抽象, 23
类型错误, 27
类型分类 (type class), 184
类型构造符, 346
类型检测, 27
类型说明, 22
量词, 129
列表, 24, 75, 76
列表上的原始递归, 112
幂等的 (idempotent), 173
名, 20
模板, 66
模块, 37
模式 (pattern), 77
模式匹配, 44
模式匹配 (pattern matching), 77
凝聚力, 57
前缀码, 247
嵌套注释, 31
软件, 15
上下文 (context), 41, 185
设计, 59
生成, 81
生成器, 81
实际参数, 22
实现, 15
守卫, 43, 47
受限的, 245
输出, 16
输入, 16, 37
数据导向的编程, 303
特例, 87
特例 (instances), 184
提示符, 35
条件, 43
通配符, 109
同义词, 76
外延的 (extensional), 172
未知量, 22
文学型, 31
无定义, 128
线性同余, 317
相互递归, 289
项 (term), 41
信息隐蔽, 246
形式参数, 22
选择函数, 77
一般递归, 68
一致的 (conformal), 101
引用点, 98
隐藏, 56
硬件, 15
语法, 43
原始递归, 65
原子, 44
约束, 27
约束满足, 202
运算, 23
蕴涵, 129
真值表, 43
证明, 26
证明的必要条件, 290
值, 20
中缀, 45, 56
终止, 127
终止计算, 70
种类, 16
重新命名, 56
重载, 47
逐步, 28
注释, 31
转换, 27

自变量, 16
自底向上, 65
自顶向下, 65
自上而下, 61
字典序, 116
字符, 24, 50
最广公共特例, 203
最广类型, 87
最通用类型, 87
作用域, 102

Curry 形式 (curried form), 164

Hugs, 19

Kleisli 复合, 347