第3章

列表程序设计

程序处理的输入和输出往往是同类型的一系列数据，这种数据通常称为列表，其类型也称为列表类型。因此，列表类型是一种非常重要的数据类型。本章首先介绍列表类型的构造方法以及列表函数的定义方法，然后通过几个实例进一步说明列表类型上的程序设计方法。

3.1 列表的构造

当处理的输入数据或者输出数据是同类型的一系列元素时，这些数据的类型通常被视为列表。因此，在定义有关函数时，需要用列表表达函数的输入和输出。本节介绍列表的构造方法，包括基本的列表构造函数以及根据一个列表构造另一个列表的列表概括法。

3.1.1 构造函数


构造函数

一个类型的**构造函数** (constructor) 是构造该类型的元素的基本方法。例如，Bool 类型的构造函数（0 元函数）有 True 和 False，它们给出 Bool 的所有元素。

一个类型的构造函数是定义该类型上其他函数的基础。例如，定义 Bool 上的一元否定运算 neg：

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

因为 Bool 只有两个构造函数，或者两种模式，所以定义用两个等式列出所有可能的输入值。

再例如，对于任意类型 a 和 b[①]，二元组类型 (a, b) 的构造函数是 (,)。

```
Prelude> :t (,)
(,) :: a -> b -> (a, b)
```

① 习惯上用 a、b、c 等表示类型变量。

二元组类型 (a,b) 的任何元素 (x,y) 都是由中缀二元运算 (,) 应用于 x 和 y 得到的，其中 x 和 y 分别是类型 a 和类型 b 的元素。例如，

```
Prelude> (,) 1.2 34
(1.2,34)
Prelude> (,) "Qiao" 60
("Qiao",60)
```

如果要定义一个函数 fst，该函数返回二元组的第一个分类，则可以如下定义：

```
fst :: (a, b) -> a
fst (x, y) = x
```

定义只用一个等式即可，因为 (a,b) 类型的任何元素都具有**模式** (x,y)。

列表的构造函数是构造列表的基本方法。

对于任意类型 a，列表类型 [a] 的元素用如下方式生成。

（1）`[]` 是 [a] 的空列表。

（2）如果 x 是类型 a 的元素，xs 是类型 [a] 的列表，那么 x:xs 是类型 [a] 的列表，x 称为该列表的**首元素**，xs 称为列表的**尾列表**。

`[]` 和 `(:)` 称为列表的**构造函数**，这表明列表的所有元素都是由这两个函数构造出来的。任何列表有两种**模式**——`[]` 或者 `x:xs`。

在解释器下可以查看 (:) 的类型：

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

例如，[Int] 类型的列表 [1,2,3] 表示首元素为 1，尾部为列表 [2,3]，因此它表示可以用 1:[2,3] 构造；同样，列表 [2,3] 可以用 2:[3] 构造，列表 [3] 可以用 `3:[]` 构造。事实上，[1,2,3] 是列表 `1:(2:(3:[]))` 的简写，表示该列表可以使用列表的两个构造函数逐步构造出来：

`[]`使用构造函数 `[]` (3.1)

`3:[]`基于 3.1和构造函数 `(:)` (3.2)

`2:(3:[])`基于 3.2和构造函数 `(:)` (3.3)

`1:(2:(3:[]))`基于 3.3和构造函数 `(:)` (3.4)

列表的这两种模式为定义列表上的函数提供了基础。

例如，定义一个函数 isEmpty，判断任意列表是否为空：

```
isEmpty :: [a] -> Bool
isEmpty [] = True
isEmpty (x:xs) = False
```

因为列表只有两种模式，因此定义中使用两个等式列出所有可能的情况。

注 1　在函数 isEmpty 定义的第二个等式中，左面的输入列表 x:xs 外围括号不可少，否则会出现类型错误。这是因为表达式 isEmpty (x:xs) 中包含两个运算：isEmpty 后面空格表示函数的应用和构造函数 (:)，而函数应用的优先级最高，因此，省略括号后的表达式相当于 (isEmpty x):xs，将 isEmpty 应用于类型 a 的元素是类型错误，因为按照定义，isEmpty 只能应用于类型 [a] 的列表。

例 3.1　定义求列表长度的函数 length。输入是任意类型的列表，输出是列表中的元素个数。因为列表不外乎有两种模式：[] 或者 x:xs。对于前者，其元素个数为 0；对于后者，可以先计算列表尾 xs 的元素个数，然后加 1。因此，有下列递归定义：

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

length 函数定义使用了递归，两个等式分别表示了递归基和递归步。注意，递归步中右边递归调用的参数 xs 小于等式左面的参数 x:xs。这是保证列表递归函数有穷步终止计算的关键。

length 是标准库函数，其类型包含类型变量 a，即该函数可应用于任何类型的列表：

```
Prelude> length [1.0,3.2,2]
3
Prelude> length [True, True,False]
3
Prelude> length ["Hello","Haskell"]
2
```

这种包含类型变量的类型称为**多态类型**，具有多态类型的函数称为**多态函数**。标准库提供了许多列表上的多态函数，例如，返回列表首元素的函数 head，返回列表尾部的函数 tail。

例 3.2　标准库函数 head 和 tail 的定义。

```
head :: [a] -> a
head (x:xs) = x

tail :: [a] -> [a]
tail (x:xs)  = xs
```

注 2　head 和 tail 这两个函数都只对非空列表有定义，因此，定义只列出一个输入为非空的模式。如果将这两个函数应用于空列表，则会引发异常错误。

例 3.3　定义一个函数 mySum，输入是一个整数列表，输出是列表中的元素之和。

函数的定义仍然对输入列表分情况考虑。如果列表为空，则结果为 0；如果列表不为空，例如 x:xs，则问题可化为计算尾部列表 xs 的元素累加和，这个计算可以通过递归完成，因此有下面定义：

```
mySum :: [Int] -> Int
mySum [] = 0
mySum (x:xs) = x + mySum xs
```

例 3.4 定义一个函数，输入一个非负整数 n 和一个整数 x，输出是 n 个 x 构成的列表。例如，输入 n=3，x=1，输出是列表 [1,1,1]。

(1) 如果输入 n=0，那么输出就是空列表。

(2) 如果 n>0，则结果列表是非空列表，其首元素是 x，其尾部是 n-1 个 x 构成的列表，后者可以用递归计算。

由此得到下列递归定义:

```
duplicate :: Int -> Int ->[Int]
duplicate 0 x = []
duplicate n x = x : duplicate (n-1) x
```

注 3 这个定义需要构造结果列表，两个等式分别使用了列表的两个构造函数，特别是第二个等式使用递归和构造函数 (:) 表达了结果列表。

注意，函数 duplicate 的第二个输入可以是任意类型的元素，以上定义仍然适用。为此，修改定义中第二个输入的类型，用一个类型变量代替，可以定义一个多态的 duplicate:

```
duplicate :: Int -> a ->[a]
duplicate 0 x = []
duplicate n x = x : duplicate (n-1) x
```

3.1.2 列表概括


列表概括

列表是一种非常重要的类型。我们经常需要用列表来表达数据，除了使用构造函数构造列表外，常常需要基于一个列表去构造另外一个列表。例如，对列表的每个元素做某种运算，从而生成了一个新的列表。例如，对于一个整数列表 [1,2,3] 的每一个元素加倍，从而得到列表 [2,4,6]。

Haskell 为这类运算提供了一种叫作列表概括的构造方法。如果 xs 是任意的整数列表，则表达式 `[2*x|x<-xs]` 表示将 xs 的每个元素加倍后的列表。这里 `x<-xs` 称为**生成器**，表示 x 取遍 xs 的每个元素。例如，

```
Prelude> xs = [1,2,3]
Prelude> [2*x | x <- xs]
[2,4,6]
```

一般地，如果 xs 是一个列表，那么表达式 `[e | x <- xs]` 就是一个列表，其中 e 是一个表达式，如上例中 e 表示 2*x，通常 e 包含了 x，表示将 xs 的每个元素 x 逐个代入 e 得到的元素构成的列表。

这种构造列表的方法称为**列表概括** (list comprehension)。

每当需要对一个列表的每个元素进行某种运算时，都可以考虑使用列表概括。

例 3.5　定义一个函数 triplePlus1，输入是一个整数列表，输出是对输入每个元素乘 3 加 1 后的列表。

不难给出函数的递归定义。因为这里结果列表是对输入列表进行统一的运算而得，因此可以用列表概括定义：

```
triplePlus1 :: [Int] -> [Int]
triplePlus1 xs = [3*x + 1 | x <- xs]
```

例 3.6　定义一个函数，输入是姓名和年龄二元组列表，输出是年龄的列表。例如，输入是 [("Wang Bin", 20), ("Gao Xin", 18), ("Alice",18)], 则输出是 [20,18,18]。

因为输出列表仍然是对输入列表进行同一个运算所得，因此可以用列表概括表示该函数：

```
getAge :: [(String, Int)] -> [Int]
getAge xs = [snd x |x <- xs]
```

列表概括生成器中的 x 可以是变量，也可以是模式。例如，以上例子中，列表 xs 的每个元素是二元组，因此也可以如下书写定义：

```
getAge :: [(String, Int)] -> [Int]
getAge xs = [y |(x,y) <- xs]
```

列表概括也可以表示对列表元素有选择地进行操作，只要在生成器之后加一个逗号，然后加一个类型为 Bool 的表达式 test 表示条件：`[e |x <- xs, test]`，表示只将 xs 中满足条件 test 的元素 x 代入表达式 e，不满足条件 test 的元素忽略。例如，`[x | x <- [1..10], mod x 2 ==0]` 表达了 [1..10] 中偶数构成的列表 [2,4,6,8,10]。

例 3.7　定义一个函数 triplePlus2，输入是一个整数列表，输出是对输入列表中每个奇数乘 3 加 1 后的列表，偶数元素忽略。例如，输入 [1,2,3]，输出 [4,10]。

这里结果列表仍然是对输入列表进行统一运算而得，因此可以用列表概括定义：

```
triplePlus2 xs = [3*x + 1 | x <- xs, mod x 2 == 1]
```

例 3.8　例如，定义一个函数，输入是姓名和年龄二元组列表，输出是年龄小于或等于 18 的人名列表。例如，输入是 [("Wang Bin", 20), ("Gao Xin", 18), ("Alice",18)], 则输出是 ["Gao Xin", "Alice"]。可以用列表概括表示该函数的输出：

```
getName :: [(String, Int)] -> [Int]
getName xs = [x |(x, y) <- xs, y <= 18]
```

列表概括的一般形式如下：

```
[e | x <- xs, test]
```

其中，xs 是一个列表，e 是表达式，x 是变量或者模式，test 是条件，也称为测试，它可以是类型为 Bool 的任意表达式，例如可以包含布尔运算 (&&) 和 (||) 等。测试也可以用逗号分隔开的多个条件表示，表示这些条件同时成立。

例 3.9 定义一个函数 factors，计算一个正整数的所有大于 1 的因子。

该函数的输入是一个正整数，输出就是这个正整数的所有大于 1 的因子，可以用列表表示。例如，10 的所有大于 1 的因子是 [2,5,10]。因此，函数的类型可以确定：

```
factors :: Int -> [Int]
```

结果类型是列表，构造列表的基本方法是用列表构造函数和列表概括。因为 n 的因子属于列表 [2..n]，因此考虑基于该列表构造因子的列表：对列表 [2..n] 的每个元素 i，检查 i 是不是 n 的因子。由此得到列表概括定义：

```
factors :: Int -> [Int]
factors n = [i | i <- [2..n], n `mod` i == 0]
```

例如，

```
*Main> factors 2020
[2,4,5,10,20,101,202,404,505,1010,2020]
*Main> factors 2021
[43,47,2021]
*Main> factors 2029
[2029]
```

根据解释器中计算结果可以判断，2029 是素数，2021 不是素数。

例 3.10 定义一个函数 isPrime，判断一个正整数是不是素数。

一个素数是因子只有 1 和它本身的正整数，如 2、3、5 等。可以利用前一个函数 factors，先计算一个正整数 n 的所有大于 1 的因子，然后检查其因子是否只有 n 本身，由此判断它是不是素数。下面是函数 isPrime 的定义：

```
isPrime :: Int -> Bool
isPrime n = factors n == [n]
```

这里假定输入大于或等于 2。例如，

```
*Main> isPrime 2021
False
*Main> isPrime 2029
True
```

3.2 图书借阅管理

列表是常用的类型，特别是列表与多元组一起使用具有非常强的表达能力。本节以一个简单的图书馆借阅管理程序为例，说明多元组、列表以及列表概括的综合应用。

过去在没有计算机的年代，读者借书时需要填写借阅卡并保存在图书馆，还书时撤销相应的借阅卡。现在的任务是将这些借阅卡信息存储在计算机上，并实现计算机管理，包括增加、删除和查询借阅卡信息。

3.2.1　借阅卡的数据及其类型

这种借阅卡信息包括哪些内容呢？显然，借阅卡应该记录借阅者以及书名。为了简单起见，假定借阅卡只记录这两个数据。这样，借阅卡的数据就是表示借阅者和书名的二元组。借阅者和书名均可以用字符串表示，例如，`("Alice", "A river runs through it")` 表示 Alice 借阅了 *A river runs through it*。因此，借阅卡的数据类型为 (String, String)。

另外，需要存储所有借阅卡数据，用二元组的列表表示，其类型为 [(String, String)]。例如，下面是一个包含多个借阅卡的数据的例子：

```
exampleData = [("Alice", "Haskell:The Craft of Functional Programming"),
               ("Alice", "A river runs through it"),
               ("Gates", "Haskell Functional Programming"),
               ("Gates", "Python Programming") ]
```

为了便于阅读，用保留字 type 给一个类型冠以另外一个**类型别名**：

```
type Borrower = String
type Book = String
type Card = (Borrower, Book)
type DataBase = [Card]
```

这里 Borrower 实际上是 String 的别名，DataBase 实际上是列表类型 [(String, String)]。注意，类型别名要以大写字母开头。

对于这样一个数据库 DataBase，可以查阅某人借阅的图书。借了几本书，并实现借书和还书的管理。接下来需要确定这些函数的类型，并分别给出其定义。

3.2.2　图书查询


图书借阅管理

查询某人借阅的图书函数 books，输入是某读者名以及当前保存了所有借阅卡数据的 DataBase，输出是该读者借阅的所有图书，所以，输出类型就是 Book 的列表 [Book]，函数 books 的类型为

```
books :: DataBase -> Borrower -> [Book]
```

可以通过查看 DataBase 每个卡片数据的第一个分量是不是给定的读者，并记录对应的第二个分量定义函数 books，因此可以用列表概括定义：

```
books :: DataBase -> Borrower -> [Book]
books db person = [b |(p,b) <- db, p == person]
```

例如，可以在解释器下运行：

```
*Main> books exampleData "Alice"
["Haskell:The Craft of Functional Programming",
 "A river runs through it"]
```

```
*Main> books exampleData "Bob"
[]
*Main>
```

查询显示 Bob 的借阅结果为 `[]`，表示 Bob 没有借阅任何图书。

3.2.3 借阅管理

借阅管理实现借书和还书操作。首先考虑借书操作。

借书函数 makeLoan 的类型是什么呢？借书时需要将新的借阅卡数据添加到现有的数据库，所以该函数的输入既包括借阅者和图书名，也包括当前的数据库。输出类型是体现了借阅完成后的新数据库，即在现有数据库上添加了新的借阅卡数据的新数据库，因此，借书函数 makeLoan 的类型为

```
makeLoan :: DataBase -> Borrower -> Book -> DataBase
```

注意，这里数据库既是输入也是输出，第一个 DataBase 表示借书前的数据库，最后一个 DataBase 表示借书完成后的数据库。

借书函数的结果是新的数据库，即在输入数据列表中添加了新的借阅二元组数据的列表，因此可以直接用构造函数 (:) 表示结果列表：

```
makeLoan :: DataBase -> Borrower -> Book -> DataBase
makeLoan db person book = (person, book) : db
```

例如，在当前的数据库 exampleData 中 Alice 借阅了 *Python Programming*：

```
*Main> makeLoan exampleData "Alice" "Python Programming"
[("Alice","Python Programming"),
 ("Alice","Haskell:The Craft of Functional Programming"),
 ("Alice","A river runs through it"),
 ("Gates","Haskell Functional Programming"),
 ("Gates","Python Programming")]
```

结果返回在原 exampleData 上添加了一条新记录 `("Alice","Python Programming")` 的列表。

注 4 函数 makeLoan 的结果列表也可用列表函数 ++ 构造，它将两个列表合并成一个列表：

```
makeLoan :: DataBase -> Borrower -> Book -> DataBase
makeLoan db person book = db ++ [(person, book)]
```

同样，还书应该是在现有的数据库中把相应的读者和借阅卡删除，结果是新的数据库，因此，其类型同借书函数的类型相同。

我们可以遍历列表中每一条借阅卡数据，检查该元素是否等于还书读者和书名构成的二元组，如果相等，则删除该元素；否则保留。因此，可以用列表概括表示：

```
returnLoan :: DataBase -> Borrower -> Book -> DataBase
returnLoan db person book = [(p,b) |(p,b) <- db,
                                    (p,b) /= (person,book)]
```

例如，在当前的数据库 exampleData 中 Gates 归还了 *Python Porgramming*：

```
*Main> returnLoan exampleData "Gates" "Python Programming"
[("Alice","Haskell:The Craft of Functional Programming"),
 ("Alice","A river runs through it"),
 ("Gates","Haskell Functional Programming")]
```

结果显示在 exampleData 中删除了二元组 `("Gates","Python Programming")` 的数据库。

注 5　类型别名为理解函数的语义及其函数的应用提供了方便。如果不使用别名，那么 returnLoan 的类型为

```
returnLoan :: DataBase -> String -> String -> DataBase
```

这里第二个和第三个输入类型都是 String，不便理解哪个表示读者，哪个表示书名。

3.3　超市购物清单

本节讨论超市购物清单打印问题：列出顾客购买商品的品名、单价、数量和总价信息。

假如一顾客购买苹果 2.5kg，单价 5.99 元/公斤，购买面包两个，单价 3.50 元/个，超市需要为顾客打印图 3.1所示的超市购物清单。

Name	Quantity	Price	Amount
Apple	2.50	5.90	14.97
Bread	2.00	3.50	7.00
Total	.....		21.97

图 3.1　超市购物清单

本节将为客户购买物品设计数据类型，并按图 3.1所示方式在屏幕上打印。

3.3.1　数据类型的设计

顾客购买的每件商品可用一个三元组表示，如（苹果，2.5kg，单价 5.99 元/公斤），为此，可以定义如下类型分别表示商品名、数量和单价：

```
type Name    = String     -- 商品名
type Quantity = Float     -- 数量
type Price   = Float      -- 单价
```

每件商品可用三元组类型表示，例如用 Item 表示一件商品的类型，则顾客购买的所有商品可用 Item 的列表表示：

```
type Item = (Name, Quantity, Price)    -- 一件商品
type Items = [Item]                    -- 多件商品
```

例如，某顾客购买的商品为列表：

```
customer1 :: Items
customer1 = [("Apple", 2.5, 5.99), ("Bread", 2, 3.5)]
```

本节的任务是设计一个函数，将类型为 Items 的数据打印在屏幕上。

3.3.2 屏幕打印函数

将数据显示到屏幕上，使用以下函数：

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

这个函数的输入类型为字符串 String，输出类型为 IO ()，关于这种类型将在第 7 章进一步讨论。例如，

```
Prelude> putStrLn "Hello World!"
Hello World!
```

如果要将数值类型打印在屏幕上，首先需要用 show 函数将数值转换为 String，然后再打印。例如，

```
Prelude> show 5.99
"5.99"
Prelude> putStrLn (show 5.99)
5.99
```

如果要将一个三元组打印在屏幕上，需要先将三元组转换为一个字符串，然后打印。例如，将三元组 ("Apple", 2.5, 5.99) 转换为一个字符串，然后打印：

```
Prelude> "Apple" ++" " ++ show 2.5 ++" " ++ show 5.99
"Apple 2.5 5.99"
Prelude> putStrLn ("Apple"++" "++ show 2.5 ++" "++ show 5.99)
Apple 2.5 5.99
```

注意，在转换为字符串时使用了字符串连接运算 (++)，并在两个数据之间添加了空格。

将多个串显示在屏幕不同行，需要使用函数 unlines，将字符串列表转换为一个字符串。例如，

```
Prelude> :t unlines
unlines :: [String] -> String
Prelude> unlines ["Hello World!", "Hello China!"]
"Hello World!\nHello China!\n"
Prelude> putStrLn (unlines ["Hello World!", "Hello China!"])
Hello World!
Hello China!
```

注意，unlines 在字符串列表的每个串后面添加了换行符 ('\n')，并将它们连接成一个串。

3.3.3　打印清单函数

解决复杂问题的基本方法是分步进行。例如，将三元组 ("Apple", 2.5, 5.99) 打印在屏幕上分两步完成：

（1）将三元组 ("Apple", 2.5, 5.99) 转换为一个字符串"Apple 2.5 5.99"。

（2）用 putStrLn 将字符串"Apple 2.5 5.99" 打印在屏幕上。

将类型为 Items 的数据打印在屏幕上，也分多步进行。以 customer1 为例，将其打印在屏幕上可以经过下列步骤完成。

（1）设计将一件商品数据转换为 String，并加上该商品的总价的函数，如将 ("Apple", 2.5, 5.99) 转换为"Apple 2.5 5.99 14.97"。

（2）设计函数将客户购买的商品列表转换为对应串的列表，如将 [("Apple", 2.5, 5.99), ("Bread", 2, 3.5)] 转换为 ["Apple 2.5 5.99 14.97", "Bread 2 3.5 7.0"]。考虑用列表概括完成。

（3）在上一步生成的列表上添加表头和表尾的串，形成满足格式要求的串的列表，如 ["Haskell Store", "Name Price Quantity Amount", "Apple 2.5 5.99 14.97", "Bread 2 3.5 7.0", "Total .....21.97"]。注意，这里需要设计一个能够统计总价的函数。

（4）将函数 unlines :: [String] -> String 应用于以上串的列表，得到一个符合打印格式要求的字符串。

（5）最后用 putStrLn 将以上满足格式要求的串输出。

下面分步完成各个函数的实现。

将表示一件商品的三元组 (n, a, p) 转换为字符串，包括该商品的总价：

```
formatItem :: Item -> String
formatItem (n, a, p) = n ++ spaces ++ showPre a ++ spaces ++
                       showPre p ++ spaces ++ showPre (a * p)
    where spaces = "   "
```

注意，这里使用了局部定义，用 spaces 表示适当长度的空格串。另外，为了将浮点数显示为确定的两位小数，这里使用了模块 Printf 的函数 printf[①]，保留两位小数：

① 使用 printf 需要导入 Text.Printf：import Text.Printf。

```
showPre :: Float -> String
showPre x = printf "%.2f" x
```

例如，

```
*HaskellStore> formatItem ("Apple", 2.5, 5.99)
"Apple    2.50    5.99    14.97"
```

将客户购买商品列表转换为字符串列表，可以用列表概括表示为 [formatItem x | x <- xs]。例如，

```
*HaskellStore> [formatItem x | x <- customer1]
["Apple    2.50    5.99    14.97","Bread    2.00    3.50    7.00"]
```

接下来定义实现第 (3) 步的函数，在前一个字符串列表上添加表头串和表尾串：

```
formatItems :: Items -> [String]
formatItems xs = header ++ [formatItem x | x <- xs] ++ [tail]
    where
    header = ["Haskell Store",
                        "Name    Quantity    Price    Amount"]
    tail = "Total .......... " ++ showPre (total xs)
```

其中，total 计算商品的总价：

```
total :: Items -> Float
total [] = 0
total ((n, a, p) : is) = a * p + total is
```

最后，将函数 unlines 和 putStrLn 依次应用于格式化后的字符串即可完成屏幕打印：

```
printItems :: Items -> IO ()
printItems is = putStrLn (unlines (formatItems is))
```

例如，

```
*HaskellStore> printItems customer1
Haskell Store
Name    Quantity    Price    Amount
Apple    2.50    5.99    14.97
Bread    2.00    3.50    7.00
Total ................ 21.97
```

3.4 一个简单图形库

本节实现一个简单的黑白字符图形模块[①]，给出图形的类型定义、图形的拼接和翻转运算定义。

① 本节例子来自 John Hughes 教授的“函数程序设计基础”课程。

考虑黑白字符艺术图形 (ASCII Art)，如图 3.2所示，这种图形可以视为由多行字符串构成的列表。对于这样的图形，可以进行上下翻转，左右翻转，两个图形左右并排，或者两个图形上下并列等操作。如图 3.3表示图 3.2左右翻转后的图形。

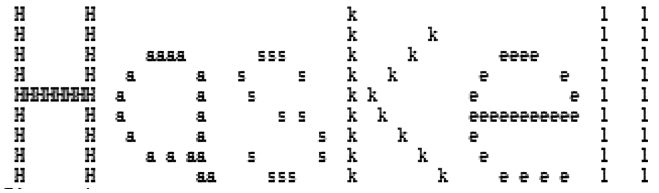


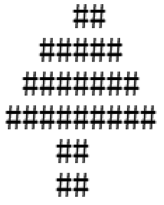
图 3.2　一个简单的 Haskell 字符图

图 3.3　Haskell 字符图的左右翻转

3.4.1　图形的类型

图形库的设计

图形是一种数据。首先需要确定用什么样的数据来表示图形，这种数据的类型是什么。这里考虑简单的黑白图形，即图形由黑点和白点构成。图 3.4 表示一棵树状黑白图形及其表示。树状黑白图可以看作 6 行 9 列黑白点构成的图，因此，该图可以用 6 个字符串表示，分别对应图形的 6 行黑白点，其中黑色点用字符 # 表示，白色点用空格表示。



（a）树状图

```
["    ##    ",
 "  #####   ",
 " #######  ",
 "######### ",
 "    ##    ",
 "    ##    "]
```

（b）树的表示

图 3.4　图形及其表示

由此定义图形的类型为字符串的列表：

```
type Line = [Char]
-- 一个字符画由多行字符串构成
type Picture = [Line]
```

注意，字符串类型 String 是字符列表类型 [Char] 的别名。

3.4.2 图形的显示

图 3.4（a）是一棵黑白树在屏幕上的显示，而图 3.4（b）是一棵黑白树的内部表示。为此，需要设计一个将图的表示显示在屏幕上的函数。假设将该函数命名为 printPicture，那么将该函数应用于一个黑白字符图形的表示可以在屏幕上显示相应的图。例如，

```
*MyPicture> printPicture tree
    ##
  #####
 #######
#########
   ##
   ##
```

这里 tree 是图 3.4中树的表示：

```
tree :: Picture
tree = ["    ##    ",
        "  #####   ",
        " #######  ",
        "######### ",
        "   ##     ",
        "   ##     "]
```

因此，函数 printPicture 的类型与打印超市清单函数 printItems（见 3.3.3节）类似：

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

表示图形的字符串列表中，每个串对应图形的一行，因此这些字符串应该依次打印在屏幕上，每一字符串占一行。因此，该函数的实现同函数 printItems，需要将函数 unlines 应用于字符串列表，然后使用 putStrLn 打印：

```
printPicture :: Picture -> IO ()
printPicture pic = putStr (unlines pic)
```

3.4.3 图形上的运算

本节考虑几种简单的图形处理函数：上下翻转、左右翻转、上下拼接和左右拼接。

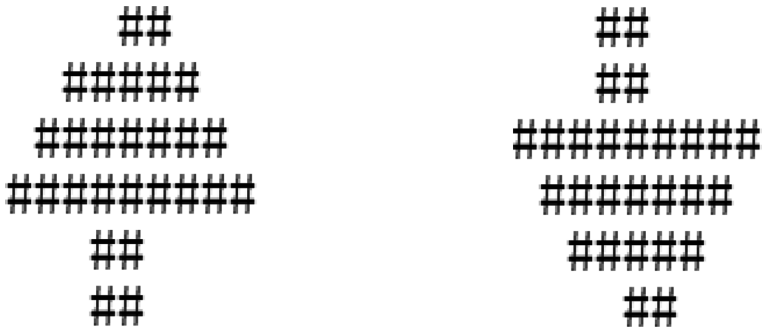
图 3.5显示一棵树及其上下翻转后的效果。

用 flipV 表示实现如图 3.5所示的上下翻转函数，则函数的输入和输出均为 Picture：

```
flipV :: Picture -> Picture
```

如果将 flipV 应用于图 3.5（a）的表示，则结果应为图 3.5（b）的表示，如图 3.6所示。

一个图形上下翻转后，新的图形的第一行是原图形的最后一行，第二行是原图形的倒数第二行，以此类推，也就是新图形的表示是原图形表示字符串列表的逆，因此定义上下翻转函数为



（a）一棵树　　（b）上下翻转后的树

图 3.5　图形的上下翻转

```
tree = ["    ##    ",
        "  #####   ",
        " ####### ",
        "######### ",
        "   ##     ",
        "   ##     "]
```

```
["   ##     ",
 "   ##     ",
 "######### ",
 " ####### ",
 "  #####   ",
 "    ##    "]
```

（a）一棵树的表示tree　　（b） flipV tree：上下翻转后的树表示

图 3.6　图形表示的上下翻转

```
flipV :: Picture -> Picture
flipV pic = reverse pic
```

其中，reverse :: [a] -> [a] 是预定义函数，即对列表取逆。例如，

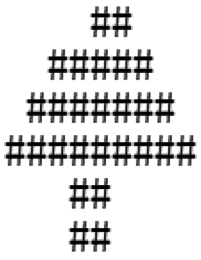
```
Prelude> reverse [1,2,3]
[3,2,1]
Prelude> reverse ["Hello","Haskell"]
["Haskell","Hello"]
Prelude> reverse "Haskell"
"lleksaH"
```

图 3.7表示字符图水平翻转效果及其表示。

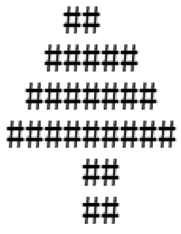
可以看出，一个图形左右翻转后，新图形的第一行是原图形表示第一行的逆，第二行是原图形第二行的逆，以此类推。因此，可以通过对输入列表的每个字符串依次逆转实现左右翻转。这种操作可以用列表概括表示，因此，左右翻转可以定义如下：

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

注意，reverse 是一个多态函数，可应用任意类型的列表。在 flipV 定义中，reverse 应用于字符串列表，而这里 reverse 应用于字符串，但是，字符串仍然是列表，因为 String 是 [Char] 的别名。



（a）字符图tree



（b）水平翻转后的字符图tree

```
tree = ["    ##    ",
        "  #####   ",
        " #######  ",
        "######### ",
        "    ##    ",
        "    ##    "]
```

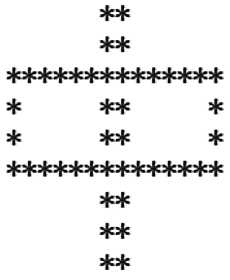
（c）tree的表示

```
["    ##    ",
 "   #####  "
 "  #######  "
 " ######### "
 "     ##    "
 "     ##    "]
```

（d）水平翻转的表示flipH tree

图 3.7　字符图水平翻转效果及其表示

接下来实现图形的拼接运算。图 3.8显示两个字符图及其表示。



（a）字符图“中”

（b）字符图“山”

```
zhong = ["       **       ",
         "       **       ",
         " ************** ",
         " *     **     * ",
         " *     **     * ",
         " ************** ",
         "       **       ",
         "       **       ",
         "       **       "]
```

（c）“中”的表示zhong

```
shan = ["                ",
        "       *        ",
        "       **       ",
        " **    **    ** ",
        " **    **    ** ",
        " **    **    ** ",
        " ************** ",
        " *            * ",
        "                "]
```

（d）“山”的表示shan

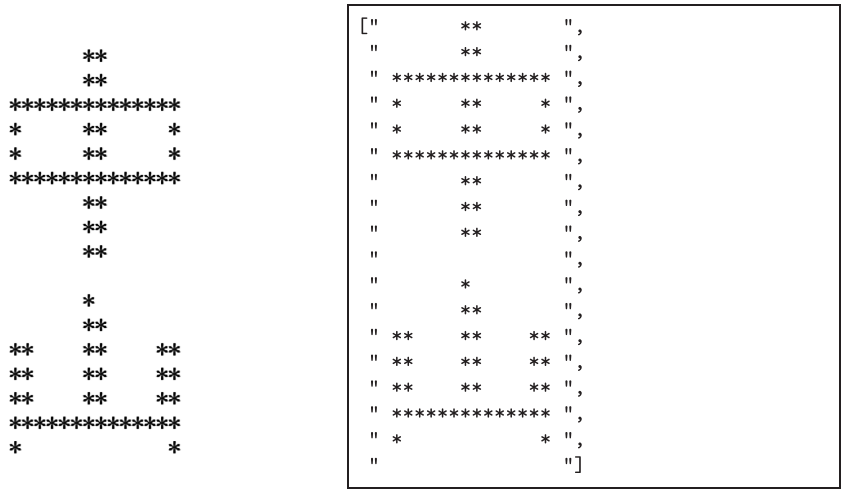
图 3.8　两个字符图及其表示

假设用 above 表示将两个图形上下拼接的函数，则 above 具有以下类型：

```
above :: Picture -> Picture -> Picture
```

图 3.9显示两个图形 zhong 和 shan 上下拼接的效果及其表示。

可见，两个图形的上下拼接后得到的新图形，就是将两个图形表示的列表串接起来，因此，

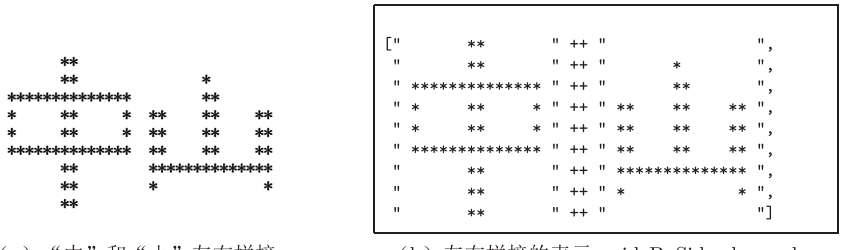


（a）“中”和“山”上下拼接的图形　　（b）上下拼接above zhong shan的表示

图 3.9　两个图形的上下拼接效果及其表示

```
above :: Picture -> Picture -> Picture
above p q = p ++ q
```

图 3.10显示两个图形的左右拼接效果及其表示。



（a）“中”和“山”左右拼接　　（b）左右拼接的表示：sideBySide zhong shan

图 3.10　字符图的左右拼接

两个图形左右并列起来的图形，其第一行是两个图形第一行的串接，第二行是两个图形第二行的串接，以此类推。为了将两个拼接图形的第一行串接，第二行串接，直至最后一行串接，需要使用一个函数 `zip :: [a] -> [b] -> [(a, b)]`，它将两个列表的对应元素配对，返回二元组的列表。例如，

```
Prelude> zip [1,2,3] [4,5,6]
[(1,4),(2,5),(3,6)]
Prelude> zip ["11","22","33"] ["44","55","66"]
[("11","44"),("22","55"),("33","66")]
```

因此，实现 zhong 和 shan 的左右拼接，可以先构造二元组列表 zip zhong shan，然后逐个将该列表中二元组的两个字符串分量用 (++) 串接成一个字符串，故左右拼接可用列表概括定义如下：

```
sideBySide :: Picture -> Picture -> Picture
sideBySide p q = [line1 ++ line2 | (line1, line2) <- zip p q]
```

3.4.4 图形模块及其应用

图形库的应用

将以上的字符图形类型定义及其图形处理函数存储在一个脚本中，由此形成一个简单的可供其他用户使用的图形函数库：

```
module Pictures(
    Picture,
    printPicture, -- :: Picture -> IO ()
    flipV,        -- :: Picture -> Picture
    flipH,        -- :: Picture -> Picture
    above,        -- :: Picture -> Picture -> Picture
    sideBySide, -- :: Picture -> Picture -> Picture
    tree, zhong, shan -- :: Picture
    ) where

type Line = [Char]
type Picture = [[Char]]
-- 以下是函数的定义，这里略去
```

注意，该模块的模块名为 Pictures，其中括号中列出模块输出的类型和函数。另外，该模块需要存储在脚本 Pictures.hs 中。

现在用户可以利用模块 Pictures 进行图形处理或者构造新的图形。例如，用户可以利用 Pictures 输出的 tree 构造一个三棵树平行构成的图形 threeTrees（见图 3.11）：

```
module MyPicture where
import Pictures
threeTrees :: Picture
threeTrees = tree `sideBySide` (tree `sideBySide` tree)
```

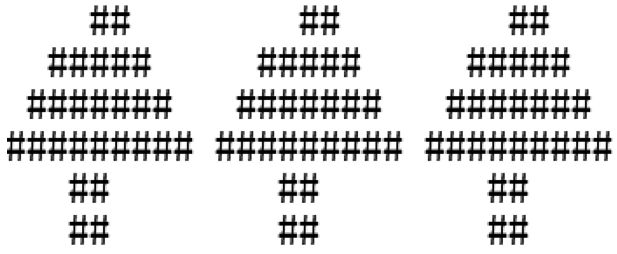


图 3.11 利用模块 Pictures 构造新的图形

用户还可以基于 Pictures 的运算构造其他图形运算。例如，基于一个图形 pic 构造一个拼接图形：在图形 pic 的右侧拼接 pic 的左右翻转图形，在下面分别拼接它们的上下翻转图形。例如，将该运算命名为 square，那么将 square 应用于 tree 可以得到如图 3.12所示效果。square 的定义留作习题。

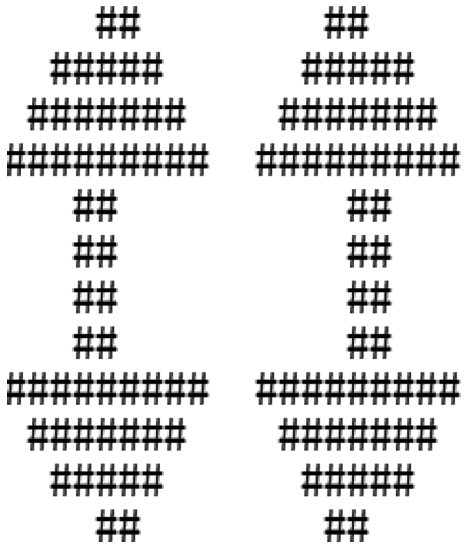


图 3.12　运算 square 应用于 tree 的图形

3.5　习题

1. 定义一个函数 isIn，第一个输入是整数 x，第二个输入是整数列表 ys，输出结果是一个布尔值：如果 x 在 ys 中出现，则结果是 True，否则结果是 False。

2. 定义一个函数 sumSquare，输入是一个整数列表，输出是输入列表中所有整数平方的和。请分别给出函数的递归定义和列表概括定义。

3. 定义一个函数，输入是一个正整数 n，输出是 $1^2+2^2+\cdots+n^2$。

提示：可以基于列表 [1..n]，然后使用前一个习题定义的函数求和。

4. 给定任意正整数 n，求不超过 n 的所有素数。

5. 列表概括可以使用多个产生式。请计算下列列表概括表达式的值：

(1) `[(x, y) | x <- [1..n], y<- [1..n]]`

(2) `[(x, y) | x <- [1..n], y<- [x..n]]`

(3) `[(x, y, z) | x <- [1..n], y<- [x..n], z<- [y..n]]`

6. 一个整数三元组 (x,y,z) 如果满足 $x^2+y^2=z^2$，则称 (x,y,z) 为毕达哥拉斯三元组。试定义一个函数

```
triads :: Int -> [(Int, Int, Int)]
```

对于任意正整数 n，triads n 给出所有分量介于 `[1..n]` 区间的毕达哥拉斯三元组。例如，

```
> triads 5
[(3,4,5), (4,3,5)]
```

7. 给出 3.4.4节所述函数 square 的定义。

8. 在 3.4.4节模块 Pictures 中添加一个将图形顺时针旋转 90° 的运算。

9. 编写一个显示放大字符串的程序。假设字符串由字母（a,b,...,z）组成，字母不分大小写。设计下列函数：

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

例如，运行

```
>sayit "Hello"
```

屏幕得到如图 3.13所示的显示效果。

```
H   H  EEEEE  L      L       OOO
H   H  E      L      L      O   O
HHHHH  EEEEE  L      L      O   O
H   H  E      L      L      O   O
H   H  EEEEE  LLLLL  LLLLL   OOO
```

图 3.13 "Hello" 的放大效果

为此，将该函数的实现分成下列几步。

（1）设计每个字母的图形，例如，字母'H'或者'h'的图形表示如图 3.14所示。然后构造一个字母图形的列表，并按照字母顺序排列：

```
pic_a2z = [pic_a, pic_b, ..., pic_z]
```

注意，这里“...”需要依次填写其他字母的图形名。

```
pic_h = [" H   H",
         " H   H",
         " HHHHH",
         " H   H",
         " H   H"]
```

图 3.14 字符'H'('h') 的图形表示

（2）构造字母到其对应图形的函数：

```
letter2pic :: Char -> Picture
```

例如，letter2pic 'h' 或者 letter2pic 'H' 均得到 pic_h。提示：可以考虑用字母的相对次序取得列表相应位置元素。模块 Data.Char 给出有关函数。

（3）构造字符串到其对应图形的列表函数：

```
string2pics :: String -> [Picture]
```