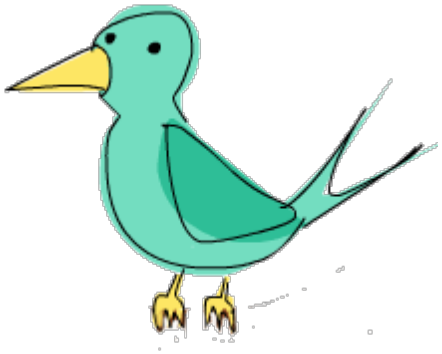
HASKELL 趣学指南

第一章 简介

关于这个教程

欢迎来到**haskell趣学指南**！阅读此文表明你正要学haskell。很好，来对地方了，先容我简单介绍一下这个教程。

编写这个教程,一方面是为了巩固我自己对haskell的理解，另一方面也是希望能够分享我的经验，给初学者提供一定帮助。网上现有的haskell教程已经汗牛充栋，在我学习的时候就曾参阅过许多教程与文章，它们讲解问题的思路各不相同，综合的阅读使得我最终能够整理起知识的碎片并正确地理解。所以说，编写这个教程也是创造另一个学习资源的尝试，给读者增加一个选择的余地。



本教程主要是面向已经有命令式编程经验(C, C++, Java, Python ...) 、却未曾接触过函数式编程 (Haskell, ML, OCaml ...)的读者。还没有编程基础？没关系，像你这样的聪明小伙一定能够学会haskell！

若在学习中遇到什么地方不明白，freenode上的#haskell频道是提问的绝佳去处。那儿的人们友善，耐心且照顾新人。

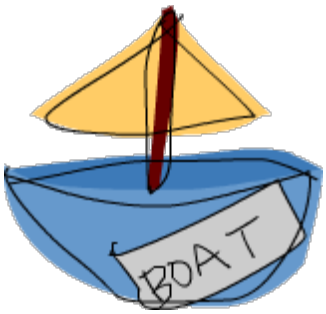
在我掌握haskell之前的学习曾经失败过两次，它看起来太不可思议，难以理解。不过随后突然灵光一闪，马上就开窍了，往后的学习也就变得游刃有余。我想说的就是：haskell很棒，如果你喜欢编程，那就得好好学学--尽管在咋一看它可能会显得很别扭--它迫使你换个脑筋思考，很有趣！

好，下一节。。。

那么，haskell是啥？

haskell是一门**纯函数式编程语言**。在命令式语言中执行操作需要给电脑安排一组命令，随着命令的执行，状态就会随之发生改变。例如你给变量a赋值为5，而随后做了其它一些事情之后a就可能变成的其它值。有控制流程，你就可以重复执行操作。然而在函数式编程语言中，你不是像命令式语言那样命令电脑“要做什么”，而是通过用函数来描述出问题“是什么”，如“*阶乘是指从1到某数间所有数字的乘积*”。变量一旦赋值，就不可以更改了，你已经说了a就是5，就不能再另说a是别的什么数。做人不能食言，对不？所以说，函数式编程语言中的函数能做的唯一事情就是求值，因而没有副作用。一开始会觉得这很受限，不过好处也正源于此：若以同样的参数调用同一函数两次，得到的结果总是相同。这被称作“*引用透明*”。如此一来编译器就可以理解程序的行为，你也很容易就能验证一个函数的正确性，继而可以将一些简单的函数组合成更复杂的函数。

haskell是**惰性**的。也就是说若非特殊指明，函数在真正需要结果以前不会被求值。再加上引用透明，你就可以把程序仅看作是数据的一系列变形。如此一来就有了很多有趣的特性，如无限长度的数据结构。假设你有一个List: `xs = [1,2,3,4,5,6,7,8]`，还有一个函数 `doubleMe`，它可以将一个List中的所有元素都乘以二，返回一个新的List。若是在命令式语言中，把一个List乘以8，执行 `doubleMe(doubleMe(doubleMe(xs)))`，得遍历三遍xs才会得到结果。而在惰性语言中，调用doubleMe时并不会立即求值，它会说“嗯嗯，待会儿再做！”。不过一旦要看结果，第一个doubleMe就会对第二个说“给我结果，快！”第二个doubleMe就会把同样的话传给第三个doubleMe，第三个doubleMe只能将1乘以2得2后交给第二个，第二个再乘以2得4交给第一个，最终得到第一个元素8。也就是说，这一切只需要遍历一次list即可，而且仅在你真正需要结果时才会执行。惰性语言中的计算只是一组初始数据和变换公式。



haskell 是**静态类型**的。当你编译程序时，编译器需要明确哪个是数字，哪个是字符串。这就意味着很大一部分错误都可以在编译时被发现，若试图将一个数字和字符串相加，编译器就会报错。haskell拥有一套强大的类型系统，支持自动类型推导。这一来你就不需要在每段代码上都标明它的类型，像计算a=5+4，你就不需另告诉编译器“a是一个数值”，它可以自己推导出来。类型推导可以让你的程序更加简练。假设有个二元函数是将两个数值相加，你就无需声明其类型，这个函数可以对一切可以相加的值进行计算。

haskell采纳了很多高级概念，因而它的代码**优雅且简练**。与同层次的命令式语言相比，haskell的代码往往会更短，更短就意味着更容易理解，bug也就更少。

haskell的研发工作始于1987年，当时是一个学会的精英分子（很多PhD哦）聚到一块儿，商量着要设计一门牛X的语言。03年，《 Haskell Report 》发布，标志着稳定版本的最终确定。

你需要...

一个编辑器和一个编译器。你可能已经安装了最喜欢的编辑器，在此不加赘述。如今最常用的haskell编译器是GHC和hugs，在本教程中我们将使用ghc。安装的细节就不消多说了，在windows下只要下载一个installer然后一路next最后重启一下（貌似不需要重启，译者注）即可；在基于debain的linux系统下一个

```
apt-get install ghc6 libghc6-mtl-dev
```

看着玩就是了；我没mac电脑，不过听说你如果安装了macPort，就可以通过

```
sudo port install ghc
```

来获得ghc。嗯，应该可以用那古怪的单键鼠标搞haskell吧，我拿不准。

GHC既可以解释执行haskell脚本（通常是以.hs作为后缀），也可以编译。它还有个交互模式，你可以在里面调用脚本里定义的函数，即时得到结果。 对于学习而言，这可比每次修改都编译执行要方便的多。想进入交互模式，只要打开控制台输入ghci即可。假设你在myfunctions.hs里定义了一些函数，在ghci中输入`:l myfunctions.hs`)载后就可以调用了。一旦修改了这个.hs文件的内容，再次执行`:l myfunctions.hs`或者与之等价的:r，都可以重新装载该文件。我本人通常就是在.hs文件中定义几个函数，再到ghci装载，调试，再修改再装载。这也正是我们往后的基本流程。

第二章 入门

各就各位，预备！

好的，出发！如果你就是那种从不看说明书的不良人士，我推荐你还是回头看一下简介的最后一节。那里面讲了这个教程中你需要用到的工具及基本用法。我们首先要做的就是进入ghc的交互模式，接着就可以调几个函数小体验一把haskell了。打开控制台，输入ghci，你会看到如下欢迎信息

```
GHCi, version 6.8.2: http://www.haskell.org/ghc/
:? for help  Loading package base ... linking ... done.
Prelude>
```

恭喜，您已经进入了ghci！目前它的命令行提示是`prelude>`，不过它在你装载什么东西后会变的比较长。免得碍眼，我们输入个: `set prompt "ghci> "` 把它改成`ghci>`。

如下是一些简单的运算

```
ghci> 2 + 15  17
ghci> 49 * 100  4900
ghci> 1892 - 1472  420
ghci> 5 / 2  2.5
ghci>
```

很简单。也可以在一行中使用多个运算符，按照运算符优先级执行计算，使用括号可以更改优先级次序。

```
ghci> (50 * 100) - 4999
1
ghci> 50 * 100 - 4999
1
ghci> 50 * (100 - 4999)
-244950
```

很酷么？嗯，我承认不。处理负数时会有个小陷阱：执行`5 * -3`会使ghci报错。所以说，使用负数时最好将其置于括号之中，像`5*(-3)`就不会有问题。

逻辑运算也同样直白，你也许知道，&&指逻辑与，||指逻辑或，not指逻辑否。

```
ghci> True && False
False
ghci> True && True
True
ghci> False || True
True
ghci> not False
True
ghci> not (True && True)
False
```

相等性可以这样判定

```
ghci> 5 == 5
True
ghci> 1 == 0
False
ghci> 5 /= 5
False
ghci> 5 /= 4
True
ghci> "hello" == "hello"
True
```

执行5+"llama"或者5==True会怎样？好的，一个大大的报错等着你。

```
No instance for (Num [Char])
arising from a use of `+' at :1:0-9
Possible fix: add an instance declaration for (Num [Char])
In the expression: 5 + "llama"
In the definition of `it': it = 5 + "llama"
```

Yikes！ghci 提示说"llama"并不是数值类型，所以它不知道该怎样才能给它加上5。即便是“four”甚至是“4”也不可以，haskel不拿它当数值。执行True==5, ghci就会提示类型不匹配。+运算符要求两端都是数值，而==运算符仅对两个可比较的值可用。这就要求他们的类型都必须一致，苹果和橙子就无法做比

较。我们会在后面深入地理解类型的概念。Note: `5+4.0` 是可以执行的，5既可以做被看做整数也可以被看做浮点数，但4.0则不能被看做整数。

也许你并未察觉，不过从始至终我们一直都在使用函数。*就是一个将两个数相乘的函数，就像三明治一样，用两个参数将它夹在中央，这被称作中缀函数。而其他大多数不能与数夹在一起的函数则被称作前缀函数。绝大部分函数都是前缀函数，在接下来我们就不多做甄别。大多数命令式编程语言中的函数调用形式通常就是函数名，括号，由逗号分隔的参数表。而在haskell中，函数调用的形式是函数名，空格，空格分隔的参数表。简单据个例子，我们调用haskell中最无聊的函数：

```
ghci> succ 8
9
```

succ函数返回一个数的后继（successor, 在这里就是8后面那个数，也就是9。译者注）。如你所见，通过空格将函数与参数分隔。调用多个参数的函数也是同样容易，min和max接受两个可比较大小的参数，并返回较大或者较小的那个数。

```
ghci> min 9 10
9
ghci> min 3.4 3.2
3.2
ghci> max 100 101
101
```

函数调用拥有最高的优先级，如下两句是等效的

```
ghci> succ 9 + max 5 4 + 1
16
ghci> (succ 9) + (max 5 4) + 1
16
```

若要取9乘10的后继，`succ 9*10` 是不行的，程序会先取9的后继，然后再乘以10得100。正确的写法应该是 `succ(9*10)` ，得91。如果某函数有两个参数，也可以用 ` 符号将它括起，以中缀函数的形式调用它。例如取两个整数相除所得商的div函数,div 92 10可得9，但这种形式不容易理解：究竟是哪个数是除数，哪个数被除？使用中缀函数的形式 92 `div` 10 就更清晰了。从命令式编程走过来的人们往往会觉得函数调用与括号密不可分，在C中，调用函数必加括号，就像foo(),bar(1),或者 `baz(3,"haha")` 。而在haskell中，函数的调用必使用空格，例如 `bar (bar 3)` ，它并不表示以bar和3两个参数去调用bar，而是以bar 3所得的

结果作为参数去调用bar。在C中，就相当于 bar(bar(3)) 。

启蒙：你的第一个函数

在前一节中我们简单介绍了函数的调用，现在让我们编写我们自己的函数！打开你最喜欢的编辑器，输入如下代码，它的功能就是将一个数字乘以2.

```
doubleMe x = x + x
```

函数的声明与它的调用形式大体相同，都是先函数名，后跟由空格分隔的参数表。但在声明中一定要在 = 后面定义函数的行为。

保存为baby.hs或任意名称，然后转至保存的位置，打开ghci，执行:l baby.hs。这样我们的函数就装载成功，可以调用了。

```
ghci> :l baby
[1 of 1] Compiling Main             ( baby.hs, interpreted )
Ok, modules loaded: Main.
ghci> doubleMe 9
18
ghci> doubleMe 8.3
16.6
```

+运算符对整数和浮点都可用（实际上所有有数字特征的值都可以），所以我们的函数可以处理一切数值。声明一个包含两个参数的函数如下：

```
doubleUs x y = x*2 + y*2
```

很简单。将其写成doubleUs x y = x + x + y + y也可以。测试一下（记住要保存为baby.hs并到ghci下边执行:l baby.hs）

```
ghci> doubleUs 4 9  26
ghci> doubleUs 2.3 34.2  73.0
ghci> doubleUs 28 88 + doubleMe 123
478
```

你可以在其他函数中调用你编写的函数，如此一来我们可以将doubleMe函数改为：

```
doubleUs x y = doubleMe x + doubleMe y
```

这种情形在haskell下边十分常见：编写一些简单的函数，然后将其组合，形成一个较为复杂的函数，这样可以减少重复工作。设想若是哪天有个数学家验证说2应该是3，我们只需要将doubleMe改为x+x+x即可，由于doubleUs调用到doubleMe，于是整个程序便进入了2即是3的古怪世界。

haskell中的函数并没有顺序，所以先声明doubleUs还是先声明doubleMe都是同样的。如下，我们编写一个函数，它将小于100的数都乘以2，因为大于100的数都已经足够大了！

```
doubleSmallNumber x = if x > 100
                      then x
                      else  x*2
```

接下来介绍haskell的if语句。你也许会觉得和其他语言很像，不过存在一些不同。haskell中if语句的else部分是不可省略。在命令式语言中，你可以通过if语句来跳过一段代码，而在haskell中，每个函数和表达式都要返回一个结果。对于这点我觉得将if语句置于一行之中会更易理解。haskell 中的if语句的另一个特点就是它其实是个表达式，表达式就是返回一个值的一段代码：5是个表达式，它返回5；4+8是个表达式；x+y也是个表达式，它返 回x+y的结果。正由于else是强制的，if语句一定会返回某个值，所以说if语句也是个表达式。如果要给刚刚定义的函数的结果都加上1，可以如此修改：

```
doubleSmallNumber' x = (if x > 100 then x else x*2) + 1
```

若是去掉括号，那就会只在小于100的时候加1。注意函数名最后的那个单引号，它没有任何特殊含义，只是一个函数名的合法字符罢了。通常，我们使用单引号来区分一个稍经修改但差别不大的函数。定义这样的函数也是可以的：

```
conanO'Brien = "It's a-me, Conan O'Brien!"
```

在这里有两点需要注意。首先就是我们没有大写conan的首字母，因为首字母大写的函数是不允许的，稍后我们将讨论其原因；另外就是这个函数并没有任何参数。没有参数的函数通常被称作“定义”（或者“名字”），一旦定义，conanO'Brien就与字符串"It's a-me, Conan O'Brien!"完全等价，且它的值不可以修改。

List入门

在Haskell中，List就像现实世界中的购物单一样重要。它是最常用的数据结构，并且十分强大，灵活地使用它可以解决很多问题。本节我们将对List，字符串和list comprehension有个初步了解。 在Haskell中，List是一种单类型的数据结构，可以用来存储多个类型相同的元素。我们可以在里面装一组数字或者一组字符，但不能把字符和数字装在一起。

> **Note**:在ghci下，我们可以使用let关键字来定义一个常量。在ghci下执行 `let a =1` 与在脚本中编写a=1是等价的。

```
ghci> let lostNumbers = [4,8,15,16,23,48]
ghci> lostNumbers
[4,8,15,16,23,48]
```

如你所见，一个List由方括号括起，其中的元素用逗号分隔开来。若试图写 `[1,2,'a',3,'b','c',4]` 这样的List，Haskell就会报出这几个字符不是数字的错误。字符串实际上就是一组字符的List，"Hello"只是 `['h','e','l','l','o']` 的语法糖而已。所以我们可以使用处理List的函数来对字符串进行操作。 将两个List合并是很常见的操作，这可以通过++运算符实现。

```
ghci> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
ghci> "hello" ++ " " ++ "world"
"hello world"
ghci> ['w','o'] ++ ['o','t']
"woot"
```

在使用++运算符处理长字符串时要格外小心(对长List也是同样)，Haskell会遍历整个的List(++符号左边的那个)。在处理较短的字符串时问题还不大，但要是在一个5000万长度的List上追加元素，那可得执行好一会儿了。所以说，用:运算符往一个List前端插入元素会是更好的选择。

```
ghci> 'A':" SMALL CAT"
"A SMALL CAT"
ghci> 5:[1,2,3,4,5]
[5,1,2,3,4,5]
```

:运算符可以连接一个元素到一个List或者字符串之中，而++运算符则是连接两个List。若要使用++运算符连接单个元素到一个List之中，就用方括号把它括起使之成为单个元素的List。 `[1,2,3]` 实际上是 `1:2:3:[]` 的语法糖。 `[]` 表示一个空List,若要从前端插入3，它就成了 `[3]` ,再插入2，它就成了 `[2,3]` ，以此类推。

> **Note**: `[],[[]],[[],[],[]]` 是不同的。第一个是一个空的List，第二个是含有一个空List的List，第三个是含有

> 三个空List的List。

若是要按照索引取得List中的元素，可以使用!!运算符，索引的下标为0。

```
ghci> "Steve Buscemi" !! 6
'B'
ghci> [9.4,33.2,96.2,11.2,23.25] !! 1
33.2
```

但你若是试图在一个只含有4个元素的List中取它的第6个元素，就会报错。要小心！

List同样也可以用来装List，甚至是List的List的List：

```
ghci> let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b ++ [[1,1,1,1]]
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3],[1,1,1,1]]
ghci> [6,6,6]:b
[[6,6,6],[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b !! 2
[1,2,2,3,4]
```

List中的List可以是不同长度，但必须得是相同的类型。如不可以在List中混合放置字符和数组相同，混合放置数值和字符的List也是同样不可以的。当List内装有可比较的元素时，使用 > 和 >=可以比较List的大小。它会先比较第一个元素，若它们的值相等，则比较下一个，以此类推。

```
ghci> [3,2,1] > [2,1,0]
True
ghci> [3,2,1] > [2,10,100]
True
ghci> [3,4,2] > [3,4]
True
ghci> [3,4,2] > [2,4]
True
ghci> [3,4,2] == [3,4,2]
True
```

还可以对List做啥？如下是几个常用的函数:

head返回一个List的头部，也就是List的首个元素。

```
ghci> head [5,4,3,2,1]
5
```

tail返回一个LIst的尾部，也就是List除去头部之后的部分。

```
ghci> tail [5,4,3,2,1]
[4,3,2,1]
```

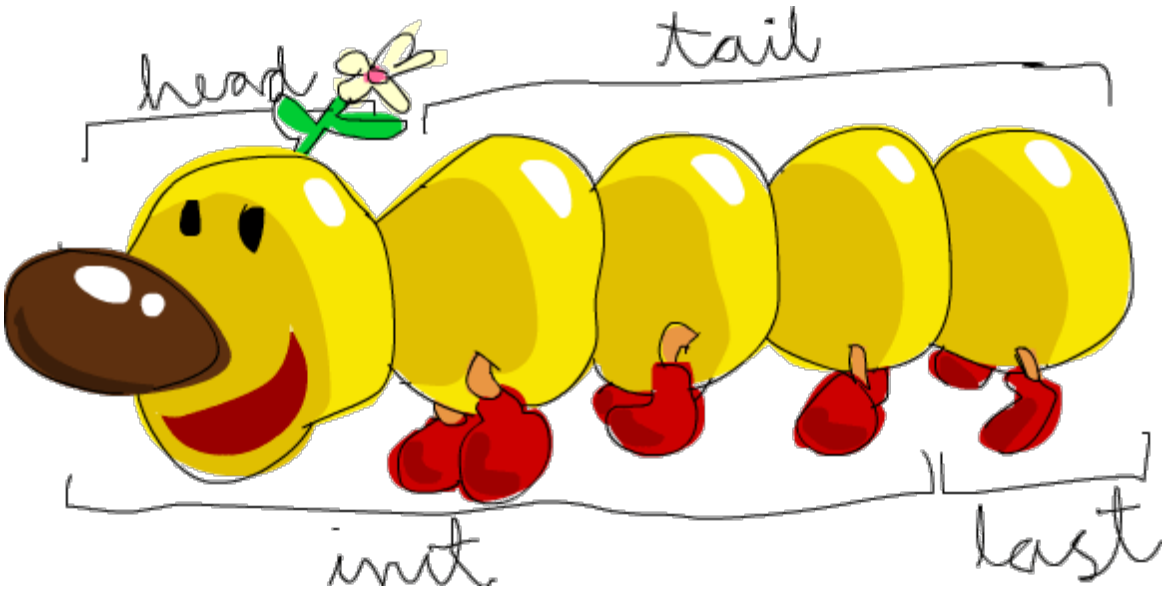
last返回一个LIst的最后一个元素。

```
ghci> last [5,4,3,2,1]
1
```

init返回一个LIst出去最后一个元素的部分。

```
ghci> init [5,4,3,2,1]
[5,4,3,2]
```

如果我们把List想象为一头怪兽，那这就是它的样子：



试一下，若是取一个空List的head又会怎样？

```
ghci> head []
*** Exception: Prelude.head: empty list
```

omg，它翻脸了！怪兽压根就不存在，head又从何而来？在使用head，tail，last和init时要小心别用到空的List上，这个错误不会在编译时被捕获。所以说做些工作以防止从空List中取值会是个好的做法。

length返回一个List的长度。

```
ghci> length [5,4,3,2,1]
5
```

null检查一个List是否为空。如果是，则返回True，否则返回False。应当避免使用xs==[]之类的语句来判断List是否为空，使用null会更好。

```
ghci> null [1,2,3]
False
ghci> null []
True
```

reverse将一个List反转

```
ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]
```

take返回一个List的前几个元素，看：

```
ghci> take 3 [5,4,3,2,1]
[5,4,3]
ghci> take 1 [3,9,3]
[3]
ghci> take 5 [1,2]
[1,2]
ghci> take 0 [6,6,6]
[]
```

如上，若是图取超过List长度的元素个数，只能得到原List。若take 0个元素，则会得到一个空List！drop与take的用法大体相同，它会删除一个List中的前几个元素。

```
ghci> drop 3 [8,4,2,1,5,6]
[1,5,6]
ghci> drop 0 [1,2,3,4]
[1,2,3,4]
ghci> drop 100 [1,2,3,4]
[]
```

maximum返回一个List中最大的那个元素。miniimun返回最小的。

```
ghci> minimum [8,4,2,1,5,6]
1
ghci> maximum [1,9,2,3,4]
9
```

sum返回一个List中所有元素的和。product返回一个List中所有元素的积。

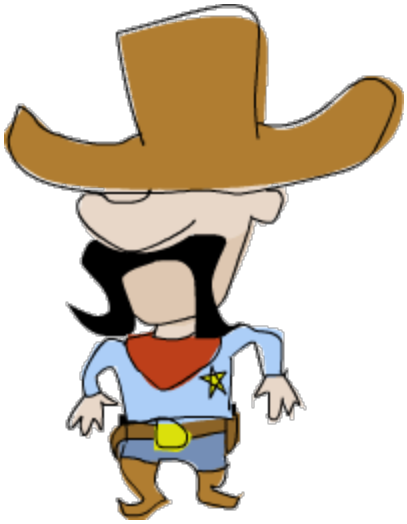
```
ghci> sum [5,2,1,6,3,2,5,7]
31
ghci> product [6,2,1,2]
24
ghci> product [1,2,5,6,7,9,2,0]
0
```

elem判断一个元素是否在包含于一个List，通常以中缀函数的形式调用它。

```
ghci> 4 `elem` [3,4,5,6]
True
ghci> 10 `elem` [3,4,5,6]
False
```

这就是几个基本的List操作函数，我们会在往后的一节中了解更多的函数。

德州区间



该怎样得到一个包含1到20之间所有数的List呢?我们完全可以用手把它全打出来，但显而易见，这并不是完美人士的方案，他们都用区间（Range）。Range是构造List方法之一，而其中的值必须是可枚举的，像1、2、3、4...字符同样也可以枚举，字母表就是A..Z所有字符的枚举。而名字就不可以枚举了，"john"后面是谁？我不知道。

要得到包含1到20中所有自然数的List，只要 `[1..20]` 即可，这与用手写 `[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]` 是完全等价的。其实用手写一两个还不是什么大事，但若是手写一个非常长的List那就一定是笨得可以了。

```
ghci> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> ['K'..'Z']
"KLMNOPQRSTUVWXYZ"
```

Range很cool，允许你申明一个步长。要得到1到20间所有的偶数或者3的倍数该怎样？

```
ghci> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
ghci> [3,6..20]
[3,6,9,12,15,18]
```

仅需用逗号将前两个元素隔开，再标上上限即可。尽管Range很聪明，但它恐怕还满足不了一些人对它的期许。你就不能通过`[1,2,4..100]`这样的语句来获得所有2的幂。一方面是因为步长只能标明一次，另一方面就是仅凭前几项，数组的后项是不能确定的。要得到20到1的List，`[20..1]`是不可以的。必须得`[20,19..1]`。在Range中使用浮点数要格外小心！出于定义的原因，浮点数并不精确。若是使用浮点数的话，你就会得到如下的糟糕结果

```
ghci> [0.1, 0.3 .. 1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
```

我的建议就是避免在Range中使用浮点数。

你也可以不标明Range的上限，从而得到一个无限长度的List。在后面我们会讲解关于无限List的更多细节。取前24个13的倍数该怎样？恩，你完全可以`[13,26..24*13]`，但有更好的方法：`take 24 [13,26..]`。

由于Haskell是惰性的，它不会对无限长度的List求值，否则会没完没了的。它会等着，看你会从它那儿取多少。在这里它见你只要24个元素，便欣然交差。如下是几个生成无限List的函数cycle接受一个List做参数并返回一个无限List。如果你只是想看一下它的运算结果而已，它会运行个没完的。所以应该在某处划好范围。

```
ghci> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
ghci> take 12 (cycle "LOL ")
"LOL LOL LOL "
```

repeat接受一个值作参数，并返回一个仅包含该值的无限List。这与用cycle处理单元素List差不多。

```
ghci> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
```

其实，你若只是想得到包含相同元素的List，使用replicate会更简单，如`replicate 3 10`，得`[10,10,10]`。

我是List Comprehension

学过数学的你对集合的comprehension（Set Comprehension）概念一定不会陌生。通过它，可以从既有的集合中按照规则产生一个新集合。前十个偶数的set comprehension可以表示为 $S = \{ 2 \cdot x \mid x \in \mathbb{N},\ x \leq 10 \}$，竖线左端的部分是输出函数，x是变量，N是输入集合。在haskell下，我们可以通过类似 `take 10 [2,4..]` 的代码来实现。但若是把简单的乘2改成更复杂的函数操作该怎么办呢？用list comprehension，它与set comprehension十分的相似，用它取前十个偶数轻而易举。这个list comprehension可以表示为：

```
ghci> [x*2 | x  [1..10]]
[2,4,6,8,10,12,14,16,18,20]
```

如你所见，结果正确。给这个comprehension再添个限制条件（predicate），它与前面的条件由一个逗号分隔。在这里，我们要求只取乘以2后大于等于12的元素。

```
ghci> [x*2 | x  [1..10], x*2 >= 12]
[12,14,16,18,20]
```

cool，灵了。若是取50到100间所有除7的余数为3的元素该怎么办？简单：

```
ghci> [ x | x  [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]
```

成功！从一个List中筛选出符合特定限制条件的操作也可以称为过滤（flitering）。即取一组数并且按照一定的限制条件过滤它们。再举个例子 吧，假如我们想要一个comprehension，它能够使list中所有大于10的奇数变为“BANG”，小于10的奇数变为“BOOM”，其他则统统 扔掉。方便重用起见，我们将这个comprehension置于一个函数之中。

```
boomBangs xs = [ if x 10 then "BOOM!" else "BANG!" | x  xs, odd x]
```

这个comprehension的最后部分就是限制条件，使用odd函数判断是否为奇数：返回True，就是奇数，该List中的元素才被包含。

```
ghci> boomBangs [7..13]
["BOOM!","BOOM!","BANG!","BANG!"]
```

也可以加多个限制条件。若要达到10到20间所有不等于13，15或19的数，可以这样：

```
ghci> [ x | x  [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20]
```

除了多个限制条件之外，从多个List中取元素也是可以的。这样的话comprehension会把所有的元素组合交付给我们的输出函数。在不过滤的前提 下，取自两个长度为4的集合的comprehension会产生一个长度为16的List。假设有两个List， `[2,5,10]` 和 `[8,10,11]` ， 要取它们所有组合的积，可以这样：

```
ghci> [ x*y | x  [2,5,10], y  [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

意料之中，得到的新List长度为9。若只取乘积为50的结果该如何？

```
ghci> [ x*y | x [2,5,10], y  [8,10,11], x*y > 50]
[55,80,100,110]
```

取个包含一组名词和形容词的List comprehension吧，写诗的话也许用得着。

```
ghci> let nouns = ["hobo","frog","pope"]
ghci> let adjectives = ["lazy","grouchy","scheming"]
ghci> [adjective ++ " " ++ noun | adjective  adjectives, noun  nouns]
["lazy hobo","lazy frog","lazy pope","grouchy hobo","grouchy frog", "grouchy pope","scheming hobo"
,
"scheming frog","scheming pope"]
```

明白！让我们编写自己的length函数吧！就叫做length'!

```
length' xs = sum [1 | _  xs]
```

表示我们并不关心从List中取什么值，与其弄个永远不用的变量，不如直接一个。这个函数将一个List中所有元素置换为1，并且使其相加求和。得到的结果便是我们的List长度。友情提示：字符串也是List，完全可以使用list comprehension来处理字符串。如下是个除去字符串中所有非大写字母的函数：

```
removeNonUppercase st = [ c | c  st, c `elem` ['A'..'Z']]
```

测试一下：

```
ghci> removeNonUppercase "Hahaha! Ahahaha!"
"HA"
ghci> removeNonUppercase "IdontLIKEFROGS"
"ILIKEFROGS"
```

在这里，限制条件做了所有的工作。它说：只有在 ['A'..'Z'] 之间的字符才可以被包含。

若操作含有List的List，使用嵌套的List comprehension也是可以的。假设有个包含许多数值的List的List，让我们在不拆开它的前提下除去其中的所有奇数：

```
ghci> let xxs = [[1,3,5,2,3,1,2,4,5],[1,2,3,4,5,6,7,8,9],[1,2,4,2,1,6,3,1,3,2,3,6]]
ghci> [ [ x | x  xs, even x ] | xs  xxs]
[[2,2,4],[2,4,6,8],[2,4,2,6,2,6]]
```

将List Comprehension分成多行也是可以的。若非在GHCI之下，还是将List Comprehension分成多行好，尤其是需要嵌套的时候。

Tuple

从某种意义上讲，Tuple(元组)很像List--都是将多个值存入一个个体的容器。但它们却有着本质的不同，一组数字的List就是一组数字，它们的类型相 同，且不关心其中包含元素的数量。而Tuple则要求你对需要组合的数据的数目非常的明确，它的类型取决于其中项的数目与其各自的类型。Tuple中的项 由括号括起，并由逗号隔开。

另外的不同之处就是Tuple中的项不必为同一类型，在Tuple里可以存入多类型项的组合。

动脑筋，在haskell中表示二维向量该如何？使用List是一种方法，它倒也工作良好。若要将一组向量置于一个List中来表示平面图形又该怎样？我们可以写类似 [[1,2],[8,11],[4,5]] 的代码来实现。但问题在于，[[1,2],[8,11,5],[4,5]] 也是同样合法的，因为其中元素的类型都相同。尽管这样并不靠谱，但编译时并不会报错。然而一个长度为2的Tuple（也可以称作序对，Pair），是一个独立的类 型，这便意味着一个包含一组序对的List不能再加入一个三元组，所以说把原先的方括号改为圆括号使用Tuple会 更好: [(1,2),(8,11),(4,5)] 。若试图表示这样的图形： [(1,2),(8,11,5),(4,5)] ，就会报出以下的错误：

```
Couldn't match expected type `(t, t1)'
against inferred type `(t2, t3, t4)'
In the expression: (8, 11, 5)
In the expression: [(1, 2), (8, 11, 5), (4, 5)]
In the definition of `it': it = [(1, 2), (8, 11, 5), (4, 5)]
```

这告诉我们说程序在试图将序对和三元组置于同一List中，而这是不允许的。同样 [(1,2),("one",2)] 这样的List也不行，因为 其中的第一个Tuple是一对数字，而第二个Tuple却成了一个字符串和一个数字。Tuple

可以用来储存多个数据，如，我们要表示一个人的名字与年 龄，可以使用这样的Tuple: `("Christopher", "Walken", 55)` 。从这个例子里也可以看出，Tuple中也可以存储List。

使用Tuple前应当事先明确一条数据中应该由多少个项。每个不同长度的Tuple都是独立的类型，所以你就不可以写个函数来给它追加元素。而唯一能做的，就是通过函数来给一个List追加序对，三元组或是四元组等内容。

可以有单元素的List，但Tuple不行。想想看，单元素的Tuple本身就只有一个值，对我们又有啥意义？不靠谱。

同List相同，只要其中的项是可比较的，Tuple也可以比较大小，只是你不可以像比较不同长度的List那样比较不同长度的Tuple。如下是两个有用的序对操作函数：

fst返回一个序对的首项。

```
ghci> fst (8,11)
8
ghci> fst ("Wow", False)
"Wow"
```

snd返回序对的尾项。

```
ghci> snd (8,11)
11
ghci> snd ("Wow", False)
False
```

> **Note**：这两个函数仅对序对有效，而不能应用于三元组，四元组和五元组之上。稍后，我们将过一遍从Tuple中取数据的所有方式。

有个函数很cool，它就是zip。它可以用来生成一组序对(Pair)的List。它取两个List，然后将它们交叉配对，形成一组序对的List。它很简单，却很实用，尤其是你需要组合或是遍历两个List时。如下是个例子：

```
ghci> zip [1,2,3,4,5] [5,5,5,5,5]
[(1,5),(2,5),(3,5),(4,5),(5,5)]
ghci> zip [1 .. 5] ["one", "two", "three", "four", "five"]
[(1,"one"),(2,"two"),(3,"three"),(4,"four"),(5,"five")]
```

它把元素配对并返回一个新的List。第一个元素配第一个，第二个元素配第二个..以此类推。注意，由于序对中可以含有不同的类型，zip函数可能会将不同类型的序对组合在一起。若是两个不同长度的List会怎么样？

```
ghci> zip [5,3,2,6,2,7,2,5,4,6,6] ["im","a","turtle"]
[(5,"im"),(3,"a"),(2,"turtle")]
```

较长的那个会在中间断开，去匹配较短的那个。由于haskell是惰性的，使用zip同时处理有限和无限的List也是可以的：

```
ghci> zip [1..] ["apple", "orange", "cherry", "mango"]
[(1,"apple"),(2,"orange"),(3,"cherry"),(4,"mango")]
```

接下来考虑一个同时应用到List和Tuple的问题：如何取得所有三边长度皆为整数且小于等于10，周长为24的直角三角形？首先，把所有三遍长度小于等于10的三角形都列出来：

```
ghci> let triangles = [ (a,b,c) | c  [1..10], b  [1..10], a  [1..10] ]
```

刚才我们是从三个List中取值，并且通过输出函数将其组合为一个三元组。只要在ghci下边调用triangle，你就会得到所有三边都小于等于 10的三角形。我们接下来给它添加一个限制条件，令其必须为直角三角形。同时也考虑上b边要短于斜边，a边要短于b边情况：

```
ghci> let rightTriangles = [ (a,b,c) | c  [1..10], b  [1..c], a  [1..b], a^2 + b^2 == c^2]
```

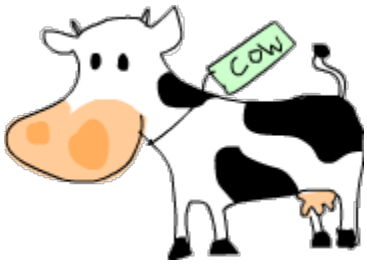
已经差不多了。最后修改函数，告诉它只要周长为24的三角形。

```
ghci> let rightTriangles' = [ (a,b,c) | c  [1..10], b  [1..c], a  [1..b], a^2 + b^2 == c^2, a+b+c == 24]
ghci> rightTriangles'
[(6,8,10)]
```

得到正确结果！这便是函数式编程的一般思路：先取一个初始的集合并将其变形，执行过滤条件，最终取得正确的结果。

第三章 类型和类型类

相信类型



在前面我们谈到Haskell是静态类型的，在编译时每个表达式的类型都已明确，这就提高了代码的安全性。若代码中让布尔值与数字相除，就不会通过编译。这样的好处就是与其让程序在运行时崩溃，不如在编译时捕获可能的错误。Haskell中万物皆有类型，因此在执行编译之时编译器可以大有所为。

与java和pascal不同，haskell支持类型推导。写下一个数字，你就没必要另告诉haskell说“它是个数字”，它自己能推导出来。这样我们就不必在每个函数或表达式上都标明其类型了。在前面我们只简单涉及一下haskell的类型方面的知识，但是理解这一类型系统对于haskell 的学习是至关重要的。

类型是每个表达式都有的某种标签，它标明了这一表达式所属的范畴。例如，表达式True是boolean型，"hello"是个字符串，等等。

可以使用ghci来检测表达式的类型。使用:t命令后跟任何可用的表达式，即可得到该表达式的类型，先试一下：

```
ghci> :t 'a'
'a' :: Char
ghci> :t True
True :: Bool
ghci> :t "HELLO!"
"HELLO!" :: [Char]
ghci> :t (True, 'a')
(True, 'a') :: (Bool, Char)
ghci> :t 4 == 5
4 == 5 :: Bool
```

可以看出，`:t` 命令处理一个表达式的输出结果为表达式后跟 `::` 及其类型，`::` 读作“它的类型为”。凡是明确的类型，其首字母必为大写。`'a'`,如它的样子，是 `Char` 类型，易知是个字符（character）。`True` 是

`Bool` 类型，也靠谱。不过这又是啥，检测 `"hello"` 得一个 `[Char]` ？这方括号表示一个List，所以我们可以将其读作“一组字符的List”。而与List不同，每个Tuple都是独立的类型，于是 `(True,"a")` 的类型是 `(Bool,Char)` ，而 `('a','b','c')` 的类型为 `(Char,Char,Char)` 。 `4==5` 一定返回 False，所以它的类型为 Bool。

同样，函数也有类型。编写函数时，给它一个明确的类型声明是个好习惯，比较短的函数就不用多此一举了。还记得前面那个过滤大写字母的List Comprehension吗？给它加上类型声明便是这个样子：

```
removeNonUppercase :: [Char] -> [Char]
removeNonUppercase st = [ c | c  st, c `elem` ['A'..'Z']]
```

`removeNonUppercase` 的类型为 `[Char]->[Char]` ，从它的参数和返回值的类型上可以看出，它将一个字符串映射为另一个字符串。 `[Char]` 与 `String` 是等价的，但使用String会更清晰： `removeNonUppercase :: String -> String` 。编译器会自动检测出它的类型，我们还是标明了它的类型声明。要是多个参数的函数该怎样？如下便是一个将三个整数相加的简单函数。

```
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

参数之间由->分隔，而与返回值之间并无特殊差异。返回值是最后一项，参数就是前三项。稍后，我们将讲解为何只用->而不是 Int,Int,Int->Int 之类“更好看”的方式来分隔参数。

如果你打算给你编写的函数加上个类型声明却拿不准它的类型是啥，只要先不写类型声明，把函数体写出来，再使用:t命令测一下即可。函数也是表达式，所以:t对函数也是同样可用的。

如下是几个常见的类型：

Int表示整数。7可以是Int，但7.2不可以。Int是有界的，也就是说它由上限和下限。对32位的机器而言，上限一般是214748364，下限是-214748364。

Integer表示...厄...也是整数，但它是无界的。这就意味着可以用它存放非常非常大的数，我是说非常大。它的效率不如Int高。

```
factorial :: Integer -> Integer
factorial n = product [1..n]
```

```
ghci> factorial 50
30414093201713378043612608166064768844377641568960512000000000000
```

Float表示单精度的浮点数。

```
circumference :: Float -> Float
circumference r = 2 * pi * r
```

ghci> circumference 4.0

25.132742

Double表示双精度的浮点数。

```
circumference' :: Double -> Double
circumference' r = 2 * pi * r

ghci> circumference' 4.0
25.132741228718345
```

Bool表示布尔值，它只有两种值：True和False。

Char表示一个字符。一个字符由单引号括起，一组字符的List即为字符串。

Tuple的类型取决于它的长度及其中项的类型。注意，空Tuple同样也是个类型，它只有一种值：`()`。

类型变量

你觉得head函数的类型是啥？它可以取任意类型的List的首项，是怎么做到的呢？我们查一下！

```
ghci> :t head
head :: [a] -> a
```

嗯! a是啥？类型吗？想想我们在前面说过，凡是类型其首字母必大写，所以它不会是个类型。它是个类型变量，意味着a可以是任意的类型。这一点与其他语言中的泛型(generic)很相似，但在haskell中要更为强大。它可以让我们轻而易举地写出类型无关的函数。使用到类型变量的函数被称作“多态函数”，head函数的类型声明里标明了它可以取任意类型的List并返回其中的第一个元素。

在命名上，类型变量使用多个字符是合法的，不过约定俗成，通常都是使用单个字符，如a,b,c,d...

还记得fst？我们查一下它的类型：

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

可以看到fst取一个包含两个类型的Tuple作参数，并以第一个项的类型作为返回值。这便是fst可以处理一

个含有两种类型项的pair的原因。注意，a和b是不同的类型变量，但它们不一定非得是不同的类型，它只是标明了首项的类型与返回值的类型相同。

类型类101

类型定义行为的接口，如果一个类型属于某类型类，那它必实现了该类型类所描述的行为。很多从OOP走过来的人们往往会把类型类当成面向对象语言中的类而感到疑惑，厄，它们不是一回事。易于理解起见，你可以把它看做是java中接口（interface）的类似物。

==函数的类型声明是怎样的？

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

> **Note**:判断相等的==运算符是函数，+-*/之类的运算符也是同样。在默认条件下，它们多为中缀函数。若要检查它的类型，就必须得用括号括起使之作为另一个函数，或者说以前缀函数的形式调用它。

有意思。在这里我们见到个新东西：=>符号。它左边的部分叫做类型约束。我们可以这样阅读这段类型声明：“相等函数取两个相同类型的值作为参数并返回一个布尔值，而这两个参数的类型同在Eq类之中（即类型约束）”

Eq这一类型类提供了判断相等性的接口，凡是可比较相等性的类型必属于Eq类。

```
ghci> 5 == 5
True
ghci> 5 /= 5
False
ghci> 'a' == 'a'
True
ghci> "Ho Ho" == "Ho Ho"
True
ghci> 3.432 == 3.432
True
```

elem函数的类型为: `(Eq a)=>a->[a]->Bool` 。这是它在检测值是否存在于一个list时使用到了==的缘故。

几个基本的类型类：

Eq包含可判断相等性的类型。提供实现的函数是==和/=。所以，只要一个函数有Eq类的类型限制，那么它就必定在定义中用到了==和/=。刚才说了，除函数意外的所有类型都属于Eq，所以它们都可以判断相等性。

Ord包含可比较大小的类型。除了函数以外，我们目前所谈到的所有类型都属于Ord类。Ord包中包含了,=之类用于比较大小的函数。compare函数取两个Ord类中的相同类型的值作参数，返回比较的结果。这个结果是如下三种类型之一：GT,LT,EQ。

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

类型若要成为Ord的成员，必先加入Eq家族。

```
ghci> "Abrakadabra" < "Zebra"
True
ghci> "Abrakadabra" `compare` "Zebra"
LT
ghci> 5 >= 2
True
ghci> 5 `compare` 3
GT
```

Show的成员为可用字符串表示的类型。目前为止，除函数以外的所有类型都是Show的成员。操作Show类型类，最常用的函数表示show。它可以取任一Show的成员类型并将其转为字符串。

```
ghci> show 3
"3"
ghci> show 5.334
"5.334"
ghci> show True
"True"
```

Read是与Show相反的类型类。read函数可以将一个字符串转为Read的某成员类型。

```
ghci> read "True" || False
True
ghci> read "8.2" + 3.8
12.0
ghci> read "5" - 2
3
ghci> read "[1,2,3,4]" ++ [3]
[1,2,3,4,3]
```

一切良好，如上的所有类型都属于这一类型类。尝试read "4"又会怎样？

```
ghci> read "4"
< interactive >:1:0:
    Ambiguous type variable `a' in the constraint:
      `Read a' arising from a use of `read' at :1:0-7
    Probable fix: add a type signature that fixes these type variable(s)
```

ghci跟我们说它搞不清楚我们想要的是什么样的返回值。注意调用read后跟的那部分，ghci通过它来辨认其类型。若要一个boolean值，他就 知道必须得返回一个Bool类型的值。但在这里它只知道我们要的类型属于Read类型类，而不能明确到底是哪个。看一下read函数的类型声明吧：

```
ghci> :t read
read :: (Read a) => String -> a
```

看？它的返回值属于Read类型类，但我们若用不到这个值，它就永远都不会得知该表达式的类型。所以我们需要在一个表达式后跟 `::` 的**类型注释**，以明确其类型。如下：

```
ghci> read "5" :: Int
5
ghci> read "5" :: Float
5.0
ghci> (read "5" :: Float) * 4
20.0
ghci> read "[1,2,3,4]" :: [Int]
[1,2,3,4]
ghci> read "(3, 'a')" :: (Int, Char)
(3, 'a')
```

编译器可以辨认出大部分表达式的类型，但遇到 `read "5"` 的时候它就搞不清楚究竟该是Int还是Float了。只有经过运算，haskell才会明确其类型；同时由于haskell是静态的，它还必须得在 编译前搞清楚所有值的类型。所以我们就最好提前给它打声招呼：“嘿，这个表达式应该是这个类型，省的你认不出来！”

Enum的成员都是连续的类型--也就是可枚举。Enum类存在的主要好处就在于我们可以在Range中用到它的成员类型：每个值都有后继子(successer)和前置子(predecesor)，分别可以通过succ函数和pred函数得到。该类型类包含的类型有：`()`，`Bool`，`Char`，`Ordering`，`Int`，`Integer`，`Float` 和 `Double`。

```
ghci> ['a'..'e']
"abcde"
ghci> [LT .. GT]
[LT,EQ,GT]
ghci> [3 .. 5]
[3,4,5]
ghci> succ 'B'
'C'
```

Bounded的成员都有一个上限和下限。

```
ghci> minBound :: Int
-2147483648
ghci> maxBound :: Char
'\1114111'
ghci> maxBound :: Bool
True
ghci> minBound :: Bool
False
```

`minBound` 和 `maxBound` 函数很有趣，它们的类型都是 `(Bounded a) => a` 。可以说，它们都是多态常量。

如果其中的项都属于 `Bounded` 类型类，那么该Tuple也属于 `Bounded`

```
ghci> maxBound :: (Bool, Int, Char)
(True,2147483647,'\1114111')
```

Num是表示数字的类型类，它的成员类型都具有数字的特征。检查一个数字的类型：

```
ghci> :t 20
20 :: (Num t) => t
```

看样子所有的数字都是多态常量，它可以作为所有 `Num` 类型类中的成员类型。以上便是 `Num` 类型类中包含的所有类型，检测*运算符的类型，可以发现它可以处理一切的数字：

```
ghci> :t (*)
(*) :: (Num a) => a -> a -> a
```

它只取两个相同类型的参数。所以 `(5 :: Int) * (6 :: Integer)` 会引发一个类型错误，而 `5 * (6 :: Integer)` 就不会有问题。

类型只有亲近 `Show` 和 `Eq` ，才可以加入 `Num` 。

Integral同样是表示数字的类型类。Num包含所有的数字：实数和整数。而Intgral仅包含整数，其中的成员类型有Int和Integer。

Floating仅包含浮点类型：Float和Double。

有个函数在处理数字时会非常有用，它便是 `fromIntegral` 。其类型声明为：`fromIntegral :: (Num b, Integral a) => a -> b` 。从中可以看出，它取一个整数做参数并返回一个更加通用的数字，这在同时处理整数和浮点时会尤为有用。举例来说， `length` 函数的类型声明为：`length :: [a] -> Int` ，而非更通用的形式，如 `(Num b) => length :: [a] -> b` 。这应该时历史原因吧，反正我觉得挺蠢。如果取了一个List长度的值再给它加3.2就会报错，因为这是将浮点数和整数相加。面对这种情况，我们就用 `fromIntegral (length [1,2,3,4]) + 3.2` 来解决。

注意到，`fromIntegral`的类型声明中用到了多个类型约束。如你所见，只要将多个类型约束放到括号里用逗号隔开即可。

第四章 函数的语法

模式匹配

本章讲的就是haskell那套酷酷的语法结构，先从模式匹配开始。模式匹配通过检查数据的特定结构来检查其是否匹配，并按模式从中取得数据。

在定义函数时，你可以为不同的模式分别定义函数体，这就让代码更加简洁易读。你可以匹配一切数据类型---数字，字符，List，元组，等等。我们弄个简单函数，让它检查我们传给它的数字是不是7。

```
lucky :: (Integral a) => a -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"
```

在调用 lucky 时，模式会从上至下进行检查，一旦有匹配，那对应的函数体就被应用了。这个模式中的唯一匹配是参数为7，如果不是7，就转到下一个模式，它匹配一切数值并将其绑定为x。这个函数完全可以使用if实现，不过我们若要个分辨1到5中的数字，而无视其它数的函数该怎么办？要是没有模式匹配的话，那可得好大一棵if-else树了！

```
sayMe :: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

注意下，如果我们把最后匹配一切的那个模式挪到最前，它的结果就全都是 "Not between 1 and 5" 了。因为它自己匹配了一切数字，不给后面的模式留机会。

记得前面实现的那个阶乘函数么？当时是把 n 的阶乘定义成了 product [1..n] 。也可以写出像数学那样的递归实现，先说明0的阶乘是1，再说明每个正整数的阶乘都是这个数与它前驱(predecessor)对应的阶乘的积。如下便是翻译到haskell的样子：

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

这就是我们定义的第一个递归函数。递归在haskell中十分重要，我们会在后面深入理解。如果拿一个数（如3）调用factorial函数，这就是接下来的计算步骤：先计算 3*factorial 2 ， factorial 2 等于 2*factorial 1 ，也就是 3*(2*(factorial 1)) 。 factorial 1 等于 1*factorial 0 ，好，得 3*(2*(1*factorial 0)) ，递归在这里到头了，嗯---我们在万能匹配前面有定义，0的阶乘是1.于是最终的结果等于 3*(2*(1*1)) 。若是把第二个模式放在前面，它就会捕获包括0在内的一切数字，这一来我们的计算就永远都不会停止了。这便是为什么说模式的顺序是如此重要：它总是优先匹配最符合的那个，最后才是那个万能的。

模式匹配也会失败。假如这个函数：

```
charName :: Char -> String
charName 'a' = "Albert"
charName 'b' = "Broseph"
charName 'c' = "Cecil"
```

拿个它没有考虑到的字符去调用它，你就会看到这个：

```
ghci> charName 'a'
"Albert"
ghci> charName 'b'
"Broseph"
ghci> charName 'h'
"*** Exception: tut.hs:(53,0)-(55,21): Non-exhaustive patterns in function charName
```

它告诉我们说，这个模式不够全面。因此，在定义模式时，一定要留一个万能匹配的模式，这样我们的程

序就不会为了不可预料的输入而崩溃了。

对Tuple同样可以使用模式匹配。写个函数，将二维空间中的向量相加该如何？将它们的x项和y项分别相加就是了。如果不了解模式匹配，我们很可能会写出这样的代码：

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors a b = (fst a + fst b, snd a + snd b)
```

嗯，可以运行。但有更好的方法，上模式匹配：

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

there we go！好多了！注意，它已经是个万能的匹配了。两个addVector的类型都是`addVectors:: (Num a) => (a,a) -> (a,a) -> (a,a)`，我们就能够保证，两个参数都是序对(Pair)了。

fst和snd可以从序对中取出元素。三元组(Tripple)呢？嗯，没现成的函数，得自己动手：

```
first :: (a, b, c) -> a
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

third :: (a, b, c) -> c
third (_, _, z) = z
```

这里的_就和List Comprehension中一样。表示我们不关心这部分的具体内容。

说到List Comprehension，我想起来在List Comprehension中也能用模式匹配：

```
ghci> let xs = [(1,3), (4,3), (2,4), (5,3), (5,6), (3,1)]
ghci> [a+b | (a,b)  xs]
[4,7,6,8,11,4]
```

一旦模式匹配失败，它就简单挪到下个元素。

对list本身也可以使用模式匹配。你可以用`[]`或`:`来匹配它。因为`[1,2,3]`本质就是`1:2:3:[]`的语法糖。你也可以使用前一种形式，像`x:xs`这样的模式可以将list的头部绑定为x，尾部绑定为xs。如果这list只有一个元素，那么xs就是一个空list。

> **Note**：x:xs这模式的应用非常广泛，尤其是递归函数。不过它只能匹配长度大于等于1的list。

如果你要把list的前三个元素都绑定到变量中，可以使用类似`x:y:z:xs`这样的形式。它只能匹配长度大于等于3的list。

我们已经知道了对list做模式匹配的方法，就实现个我们自己的head函数。

```
head' :: [a] -> a
head' [] = error "Can't call head on an empty list, dummy!"
head' (x:_) = x
```

看看管不管用：

```
ghci> head' [4,5,6]
4
ghci> head' "Hello"
'H'
```

漂亮！注意下，你若要绑定多个变量（用_也是如此），我们必须用括号将其括起。同时注意下我们用的这个error函数，它可以生成一个运行时错误，用参数中的字符串表示对错误的描述。它会直接导致程序崩溃，因此应谨慎使用。可是对一个空list取head真的不靠谱哇。

弄个简单函数，让它用非标准的英语给我们展示list的前几项。

```
tell :: (Show a) => [a] -> String
tell [] = "The list is empty"
tell (x:[]) = "The list has one element: " ++ show x
tell (x:y:[]) = "The list has two elements: " ++ show x ++ " and " ++ show y
tell (x:y:_) = "This list is long. The first two elements are: " ++ show x ++ " and " ++ show y
```

这个函数顾及了空list，单元素list，双元素list以及较长的list，所以这个函数很安全。`(x:[])` 与 `(x:y:[])` 也可以写作 `[x]` 和 `[x,y]`（有了语法糖，我们不必多加括号）。不过 `(x:y:_)` 这样的模式就不行了，因为它匹配的list长度不固定。

我们曾用List Comprehension实现过自己的length函数，现在用模式匹配和递归重新实现它：

```
length' :: (Num b) => [a] -> b
length' [] = 0
length' (_:xs) = 1 + length' xs
```

这与先前写的那个factorial函数很相似。先定义好未知输入的结果---空list，这也叫作边界条件。再在第二个模式中将这List分割为头部和尾部。说，List的长度就是其尾部的长度加1。匹配头部用的_，因为我们并不关心它的值。同时也应明确，我们顾及了List所有可能的模式：第一个模式匹配空list，第二个匹配任意的非空list。

看下拿 `"ham"` 调用 `length'` 会怎样。首先它会检查它是否为空List。显然不是，于是进入下一模式。它匹配了第二个模式，把它分割为头部和尾部并无视掉头部的值，得长度就是 `1+length' "am"`。ok。以此类推，`"am"` 的 length 就是 `1+length' "m"`。好，现在我们有了 `1+(1+length' "m")`。`length' "m"` 即 `1+length ""`（也就是 `1+length' []`）。根据定义，`length' []` 等于 `0`。最后得 `1+(1+(1+0))`。

再实现 sum 。我们知道空list的和是0，就把它定义为一个模式。我们也知道一个list的和就是头部加上尾部的和的和。写下来就成了：

```
sum' :: (Num a) => [a] -> a
sum' [] = 0
sum' (x:xs) = x + sum' xs
```

还有个东西叫做as模式，就是将一个名字和@置于模式前，可以在按模式分割什么东西时仍保留对其整体的引用。如这个模式 xs@(x:y:ys) ，它会匹配出与 x:y:ys 对应的东西，同时你也可以方便地通过xs得到整个list，而不必在函数体中重复 x:y:ys 。看下这个quick and dirty的例子：

```
capital :: String -> String
capital "" = "Empty string, whoops!"
capital all@(x:xs) = "The first letter of " ++ all ++ " is " ++ [x]
```

```
ghci> capital "Dracula"
"The first letter of Dracula is D"
```

我们使用as模式通常就是为了在较大的模式中保留对整体的引用，从而减少重复性的工作。

还有——你不可以在模式匹配中使用 ++ 。若有个模式是 (xs++ys) ，那么这个List该从什么地方分开呢？不靠谱吧。而 (xs++[x,y,z]) 或只一个 (xs++[x]) 或许还能说的过去，不过出于list的本质，这样写也是不可以的。

注意，门卫!

模式用来检查一个值是否合适并从中取值，而门卫（guard）则用来检查一个值的某项属性是否为真。咋一听有点像是if语句，实际上也正是如此。不过处理多个条件分支时门卫的可读性要高些，并且与模式匹配契合的很好。

在讲解它的语法前，我们先看一个用到门卫的函数。它会依据你的BMI值（body mass index，身体质量指数）来不同程度地侮辱你。BMI值即为体重除以身高的平方。如果小于18.5，就是太瘦；如果在18.5到25之间，就是正常；25到30之间，超重；如果超过30，肥胖。这就是那个函数（我们目前暂不为您计算bmi，它只是直接取一个emi值）。

```
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
    | bmi  18.5 = "You're underweight, you emo, you!"
    | bmi  25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi  30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise   = "You're a whale, congratulations!"
```

门卫由跟在函数名及参数后面的竖线标志，通常他们都是靠右一个缩进排成一列。一个门卫就是一个布尔表达式，如果为真，就使用其对应的函数体。如果为假，就送去见下一个门卫，如之继续。如果我们用24.3调用这个函数，它就会先检查它是否小于等于18.5，显然不是，于是见下一个门卫。24.3小于25.0，因此通过了第二个门卫的检查，就返回第二个字符串。

在这里则是相当的简洁，不过不难想象这在命令式语言中又会是怎样的一棵if-else树。由于if-else的大树比较杂乱，若是出现问题会很难发现，门卫对此则十分清楚。

最后的那个门卫往往都是`otherwise`，它的定义就是简单一个`otherwise = True`，捕获一切。这与模式很相像，只是模式检查的是匹配，而它们检查的是布尔表达式。如果一个函数的所有门卫都没有通过（而且没有提供otherwise作万能匹配），就转入下一模式。这便是门卫与模式契合的地方。如果始终没有找到合适的门卫或模式，就会发生一个错误。

当然，门卫可以在含有任意数量参数的函数中使用。省得用户在使用这函数之前每次都自己计算bmi。我们修改下这个函数，让它取身高体重为我们计算。

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | weight / height ^ 2  18.5 = "You're underweight, you emo, you!"
    | weight / height ^ 2  25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | weight / height ^ 2  30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise                 = "You're a whale, congratulations!"
```

看看我胖不胖.....

```
ghci> bmiTell 85 1.90
"You're supposedly normal. Pffft, I bet you're ugly!"
```

Yay!我不胖！不过haskell依然说我很猥琐...什么道理...

注意下，函数名和参数的后面并没有=。许多新人容易搞出语法错误，就是因为在后面加上了=。

另一个简单的例子：实现个自己的`max`函数。应该还记得，它是取两个可比较的值，返回较大的那个。

```
max' :: (Ord a) => a -> a -> a
max' a b
    | a > b     = a
    | otherwise = b
```

门卫也可以堆一行里面。这样的可读性会差些，因而是不被鼓励的。即使是较短的函数也是如此，仅仅出于演示，我们可以这样重写max'：

```
max' :: (Ord a) => a -> a -> a
max' a b | a > b = a | otherwise = b
```

Ugh！一点都不好读！继续进发，用门卫实现我们自己的compare函数：

```
myCompare :: (Ord a) => a -> a -> Ordering
a `myCompare` b
    | a > b     = GT
    | a == b    = EQ
    | otherwise = LT
```

```
ghci> 3 `myCompare` 2
GT
```

> **Note**：通过反单引号，我们不仅可以以中缀形式调用函数，也可以在定义函数的时候使用它。有时这样会更易读。

Where?

前一节中我们写了这个bmi计算函数：

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | weight / height ^ 2  18.5 = "You're underweight, you emo, you!"
    | weight / height ^ 2  25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | weight / height ^ 2  30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise                 = "You're a whale, congratulations!"
```

注意，我们重复了3次。我们重复了3次。程序员的字典里不应该有“重复”这个词。既然发现有重复，那么给它一个名字来代替这三个表达式会更好些。嗯，我们可以这样修改：

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi  18.5 = "You're underweight, you emo, you!"
    | bmi  25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi  30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise   = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
```

我们的where关键字跟在门卫后面（最好是与竖线缩进一致），可以定义多个名字和函数。这些名字对每个门卫都是可见的，这一来就避免了重复。如果我们打算换种方式计算bmi，只需进行一次修改就行了。

通过命名，我们提升了代码的可读性，并且由于bmi只计算了一次，函数的执行效率也有所提升。我们可以再做下修改：

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi  skinny = "You're underweight, you emo, you!"
    | bmi  normal = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi  fat    = "You're fat! Lose some weight, fatty!"
    | otherwise     = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
          skinny = 18.5
          normal = 25.0
          fat = 30.0
```

函数在*where*绑定中定义的名字只对本函数可见，因此我们不必担心它会污染其他函数的命名空间。注意，其中的名字都是一列垂直排开，如果不这样规范，haskell就搞不清楚它们在哪个地方了。

*where*绑定不会在多个模式中共享。如果你在一个函数的多个模式中重复用到同一名字，就应该把它置于全局定义之中。

*where*绑定也可以使用**模式匹配**！前面那段代码可以改成：

```
...
where bmi = weight / height ^ 2
      (skinny, normal, fat) = (18.5, 25.0, 30.0)
```

我们再搞个简单函数，让它告诉我们姓名的首字母：

```
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
    where (f:_) = firstname
          (l:_) = lastname
```

我们完全按可以在函数的参数上直接使用模式匹配（这样更短更简洁），在这里只是为了演示在where语句中同样可以使用模式匹配：

*where*绑定可以定义名字，也可以定义函数。保持健康的编程风格，我们搞个计算一组bmi的函数：

```
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi w h | (w, h)
    where bmi weight height = weight / height ^ 2
```

这就全了！在这里将`bmi`搞成一个函数，是因为我们不能依据参数直接进行计算，而必须先从传入函数的list中取出每个序对并计算对应的值。

*where*绑定还可以嵌套。有个已被广泛接受的理念，就是一个函数应该有几个辅助函数。而每个辅助函数

也可以通过where拥有各自的辅助函数。

Let it be

let绑定与where绑定很相似。where绑定是在函数底部定义名字，对包括所有门卫在内的整个函数可见。let绑定则是个表达式，允许你在任何位置定义局部变量，而对不同的门卫不可见。正如haskell中所有赋值结构一样，let绑定也可以使用模式匹配。看下它的实际应用！这是个依据半径和高度求圆柱体表面积的函数：

```
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r ^2
    in  sideArea + 2 * topArea
```

let的格式为 let [bindings] in [expressions] 。在*let*中绑定的名字仅对in部分可见。*let*里面定义的名字也得对齐到一列。不难看出，这用*where*绑定也可以做到。那么它俩有什么区别呢？看起来无非就是，*let*把绑定放在语句前面而*where*放在后面嘛。

不同之处在于，*let*绑定本身是个表达式，而*where*绑定则是个语法结构。还记得前面我们讲if语句时提到它是个表达式，因而可以随处安放？

```
ghci> [if 5 > 3 then "Woo" else "Boo", if 'a' > 'b' then "Foo" else "Bar"]
["Woo", "Bar"]
ghci> 4 * (if 10 > 5 then 10 else 0) + 2
42
```

用*let*绑定也可以实现：

```
ghci> 4 * (let a = 9 in a + 1) + 2
42
```

*let*也可以定义局部函数：

```
ghci> [let square x = x * x in (square 5, square 3, square 2)]
[(25,9,4)]
```

若要在一行中绑定多个名字，再将它们排成一列显然是不可以的。不过可以用分号将其分开。

```
ghci> (let a = 100; b = 200; c = 300 in a*b*c, let foo="Hey "; bar = "there!" in foo ++ bar)
(6000000,"Hey there!")
```

最后那个绑定后面的分号不是必须的，不过加上也没关系。如我们前面所说，你可以在let绑定中使用模式匹配。这在从Tuple取值之类的操作中很方便。

```
ghci> (let (a,b,c) = (1,2,3) in a+b+c) * 100
600
```

你也可以把let绑定放到List Comprehension中。我们重写下那个计算bmi值的函数，用个let替换掉原先的where。

```
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi | (w, h)  xs, let bmi = w / h ^ 2]
```

List Comprehension中let绑定的样子和限制条件差不多，只不过它做的不是过滤，而是绑定名字。let中绑定的名字在输出函数及限制条件中都可见。这一来我们就可以让我们的函数只返回胖子的bmi值：

```
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi | (w, h)  xs, let bmi = w / h ^ 2, bmi >= 25.0]
```

在 `(w, h) <- xs` 这里无法使用 `bmi` 这名字，因为它在let绑定的前面。

在List Comprehension中我们忽略了let绑定的in部分，因为名字的可见性已经预先定义好了。不过，把一个*let...in*放到限制条件中也是可以的，这样名字只对这个限制条件可见。在GHCi中in部分也可以省略，名字的定义就在整个交互中可见。

```
ghci> let zoot x y z = x * y + z
ghci> zoot 3 9 2
29
ghci> let boot x y z = x * y + z in boot 3 4 2
14
ghci> boot
>:1:0: Not in scope: `boot'
```

你说既然*let*已经这么好了，还要where干嘛呢？嗯，let是个表达式，定义域限制的相当小，因此不能在多个门卫中使用。一些朋友更喜欢*where*，因为它是跟在函数体后面，把主函数体距离类型声明近一些会更易读。

case表达式

有命令式编程(*C, C++, Java, etc*)的经验的同学一定会有所了解，很多命令式语言都提供了*case*语句。就是取一个变量，按照对变量的判断选择对应的代码块。其中可能会存在一个万能匹配以处理未预料的情况。

haskell取了这一概念融合其中。如其名，case表达式就是，嗯，一种表达式。跟*if..else*和*let*一样的表达式。用它可以对变量的不同情况分别求值，还可以使用模式匹配。Hmm，取一个变量，对它模式匹配，执行对应的代码块。好像在哪儿听过？啊，就是函数定义时参数的模式匹配！好吧，模式匹配本质上不过就是case语句的语法糖而已。这两段代码就是完全等价的：

```
head' :: [a] -> a
head' [] = error "No head for empty lists!"
head' (x:_) = x
```

```
head' :: [a] -> a
head' xs = case xs of [] -> error "No head for empty lists!"
                      (x:_) -> x
```

看得出，*case*表达式的语法十分简单：

```
case expression of pattern -> result
                   pattern -> result
                   pattern -> result
                   ...
```

*expression*匹配合适的模式。如料，第一个模式若匹配，就执行第一个代码块；否则就交给下一个模式。如果到最后依然没有匹配的模式，就会产生一个运行时错误。

函数参数的模式匹配只能在定义函数时使用，而case表达式可以用在任何地方。例如：

```
describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of [] -> "empty."
                                               [x] -> "a singleton list."
                                               xs -> "a longer list."
```

这在表达式中作模式匹配很方便，由于模式匹配本质上就是case表达式的语法糖，那么写成这样也是等价

的：

```
describeList :: [a] -> String
describeList xs = "The list is " ++ what xs
    where what [] = "empty."
          what [x] = "a singleton list."
          what xs = "a longer list."
```

第五章 递归

你好，递归！

前面的章节中我们简要谈了一下递归。而在本章，我们会深入地了解到它为何在haskell中是如此重要，能够以递归思想写出简洁优雅的代码。

如果你还不明白什么是递归，就读这个句子。哈哈！玩笑而已！递归实际上是定义函数以调用自身的方式。在数学定义中，递归随处可见，如斐波那契数列（fibonacci）。它先是定义两个非递归的数：*F(0)=0,F(1)=1*，表示斐波那契数列的前两个数为0和 1。然后就是对其他自然数，其斐波那契数就是它前面两个数字的和，即*F(N)=F(N-1)+F(N-2)*。这样一来，*F(3*就是*F(2)+F(1)*，进一步便是*(F(1)+F(0))+F(1)*。已经下探到了前面定义的非递归斐波那契数，可以放心地说*F(3*就是2了。在递归定义中声明的一两个非递归的值（如*F(0*和*F(1)*）也可以称作**边界条件**，这对递归函数的正确求值至关重要。要是前面没有定义*F(0*和*F(1*的话，它下探到0之后就会进一步到负数，你就永远都得不到结果了。一不留神它就算到了*F(-2000)=F(-2001)+F(-2002)*，并且永远都算不到头！

递归在haskell中至关重要。命令式语言要求你提供求解的步骤，haskell则倾向于让你提供问题的描述。这便是haskell没有while或for循环的原因，递归是我们的替代方案。

麦克西米不可思议

`maximum` 函数取一组可排序的List（属于 Ord类型类）做参数，并返回其中的最大值。想想，在命令式风格中这一函数该怎么实现。很可能你会设一个变量来存储当前的最大值，然后用循环遍历该 List，若存在比这个值更大的元素，则修改变量为这一元素的值。到最后，变量的值就是运算结果。唔！描述如此简单的算法还颇费了点口舌呢！

现在看看递归的思路是如何：我们先定下一个边缘条件，即处理单个元素的List时，返回该元素。如果该

List的头部大于尾部的最大值，我们就可以假定较长 的List的最大值就是它的头部。而尾部若存在比它更大的元素，它就是尾部的最大值。就这么简单！现在，我们在haskell中实现它

```
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs)
    | x > maxTail = x
    | otherwise = maxTail
    where maxTail = maximum' xs
```

如你所见，模式匹配与递归简直就是天造地设！大多数命令式语言中都没有模式匹配，于是你就得造一堆if-else来测试边界条件。而在这里，我们仅需要使用 模式将其表示出来。第一个模式说，如果该List为空，崩溃！就该这样，一个空List的最大值能是啥？我不知道。第二个模式也表示一个边缘条件，它说，如果这个List仅包含单个元素，就返回该元素的值。

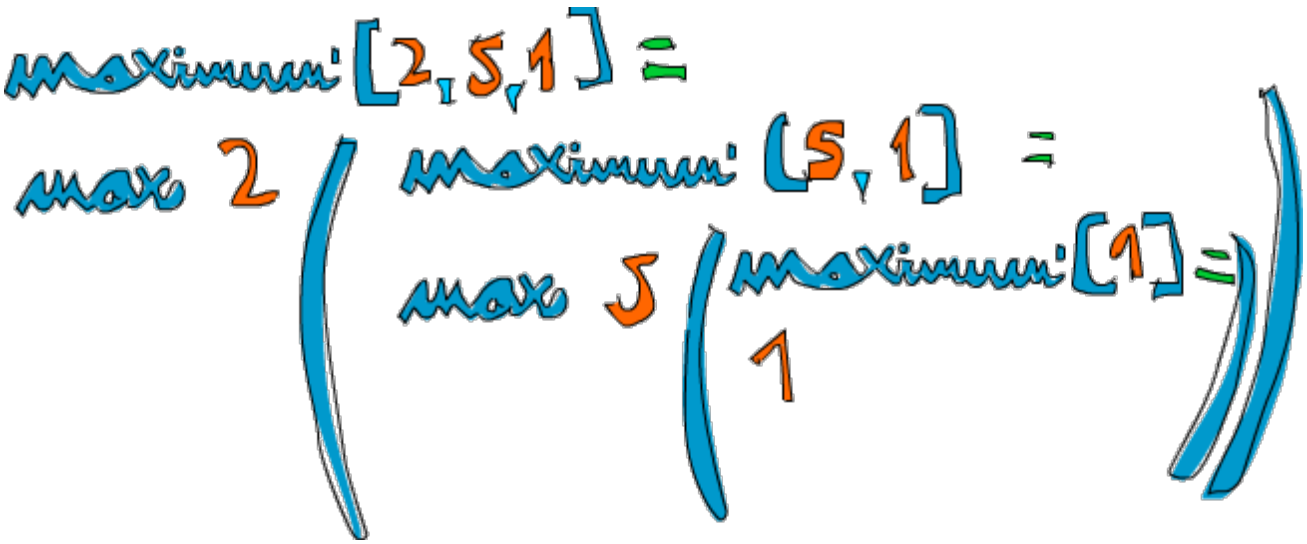
现在是第三个模式，执行动作的地方。 通过模式匹配，可以取得一个List的头部和尾部。这在使用递归处理List时是十分常见的。出于习惯，我们用个where语句来表示 `maxTail` 作为该List中尾部的最大值，然后检查头部是否大于尾部的最大值。若是，返回头部；若非，返回尾部的最大值。

我们取个List `[2,5,1]` 做例子来看看它的工作原理。当调用 `maximum'` 处理它时，前两个模式不会被匹配，而第三个模式匹配了它并将其分为 `2` 与 `[5,1]` 。 where子句再取 `[5,1]` 的最大值。于是再次与第三个模式匹配，并将 `[5,1]` 分割为 `5` 和 `[1]` 。继续，where子句取 `[1]` 的最大值，这时终于到了边缘条件！返回 `1` 。进一步，将 `5` 与 `[1]` 中的最大值做比较，易得5，现在我们就得到了 `[5,1]` 的最大值。再进一步，将 `2` 与 `[5,1]` 中的最大值相比较，可得 `5` 更大，最终得 `5` 。

改用 `max` 函数会使代码更加清晰。如果你还记得， `max` 函数取两个值做参数并返回其中较大的值。如下便是用 `max` 函数重写的 `maximun'`

```
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs) = max x (maximum' xs)
```

太漂亮了！一个List的最大值就是它的首个元素与它尾部中最大值相比较所得的结果，简明扼要。



几个递归函数

现在我们已经了解了递归的思路,接下来就使用递归来实现几个函数. 先实现下 `replicate` 函数, 它取一个 `Int` 值和一个元素做参数, 返回一个包含多个重复元素的List, 如 `replicate 3 5` 返回 `[5,5,5]` . 考虑一下, 我觉得它的边界条件应该是负数. 如果要replicate重复某元素零次, 那就是空List. 负数也是同样, 不靠谱.

```
replicate' :: (Num i, Ord i) => i -> a -> [a]
replicate' n x
    | n  0    = []
    | otherwise = x:replicate' (n-1) x
```

在这里我们使用了门卫而非模式匹配, 是因为这里做的是布尔判断. 如果n小于0就返回一个空List, 否则, 返回以x作首个元素并后接重复n-1次x的List. 最后, (n-1)的那部分就会令函数抵达边缘条件.

> **Note**: Num不是Ord的子集, 表示数字不一定得拘泥于排序, 这就是在做加减法比较时要将Num与Ord类型约束区别开来的原因.

接下来实现 `take` 函数, 它可以从一个List取出一定数量的元素. 如 `take 3 [5,4,3,2,1]` ,得 `[5,4,3]` . 若要取零或负数个的话就会得到一个空List. 同样, 若是从一个空List中取值, 它会得到一个空List. 注意, 这儿有两个边界条件, 写出来:

```
take' :: (Num i, Ord i) => i -> [a] -> [a]
take' n _
    | n  0    = []
take' _ []     = []
take' n (x:xs) = x : take' (n-1) xs
```

首个模式辨认若为0或负数, 返回空List. 同时注意这里用了一个门卫却没有指定otherwise部分, 这就表示n若大于0, 会转入下一模式. 第二个模式指明了若试图从一个空List中取值, 则返回空List. 第三个模式将List分割为头部和尾部, 然后表明从一个list中取多个元素等同于令x作头部后接从尾部取n-1个元素所得的List. 假如我们要从 `[4,3,2,1]` 中取3个元素, 试着从纸上写出它的推导过程

`reverse` 函数简单地反转一个List, 动脑筋想一下它的边界条件! 该怎样呢? 想想...是空List! 空List的反转结果还是它自己. Okay , 接下来该怎么办? 好的, 你猜的出来. 若将一个List分割为头部与尾部, 那它反转的结果就是反转后的尾部与头部相连所得的List.

```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]
```

继续进发!

haskell支持无限List，所以我们的递归就不必添加边界条件。这样一来，它可以对某值计算个没完, 也可以产生一个无限的数据结构，如无限List。而无限List的好处就在于我们可以在任意位置将它断开.

`repeat` 函数取一个元素作参数, 返回一个仅包含该元素的无限List. 它的递归实现简单的很, 看:

```
repeat' :: a -> [a]
repeat' x = x:repeat' x
```

调用 `repeat 3` 会得到一个以3为头部并无限数量的3为尾部的List, 可以说 `repeat 3` 运行起来就是 `3:repeat 3` , 然后 `3:3:3:3` 等等. 若执行 `repeat 3` , 那它的运算永远都不会停止。而 `take 5 (repeat 3)` 就可以得到5个3, 与 `replicate 5 3` 差不多.

zip取两个List作参数并将其捆在一起。 `zip [1,2,3] [2,3]` 返回 `[(1,2),(2,3)]` , 它会把较长的List从中间断开, 以匹配较短的List. 用zip处理一个List与空List又会怎样? 嗯, 会得一个空List, 这便是我们的限制条件, 由于zip取两个参数, 所以要有两个边缘条件

```
zip' :: [a] -> [b] -> [(a,b)]
zip' _ [] = []
zip' [] _ = []
zip' (x:xs) (y:ys) = (x,y):zip' xs ys
```

前两个模式表示两个List中若存在空List, 则返回空List. 第三个模式表示将两个List捆绑的行为, 即将其头部配对并后跟捆绑的尾部. 用zip处理 [1,2,3] 与 ['a','b'] 的话, 就会在 [3] 与 [] 时触及边界条件, 得到 (1,'a'):(2,'b'):[] 的结果,与 [(1,'a'),(2,'b')] 等价.

再实现一个标准库函数--elem! 它取一个元素与一个List作参数, 并检测该元素是否包含于此List. 而边缘条件就与大多数情况相同, 空List. 大家都知道空List中不包含任何元素, 便不必再做任何判断

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' a [] = False
elem' a (x:xs)
    | a == x    = True
    | otherwise = a `elem'` xs
```

简单直接. 若头部不是该元素, 就检测尾部, 若为空List就返回False

排序，要快！

假定我们有一个可排序的List,其中元素的类型为Ord类型类的成员. 现在我们要给它排序! 有个排序算法非常的酷, 就是快速排序(*quick sort*), 睿智的排序方法. 尽管它在命令式语言中也不过10行, 但在haskell下边要更短,更漂亮, 俨然已经成了haskell的招牌了. 嗯, 我们在这里也实现一下. 或许会显得很俗气, 因为每个人都用它来展示haskell究竟有多优雅!

它的类型声明应为 quicksort :: (Ord a) => [a] -> [a] , 没啥奇怪的. 边界条件呢? 如料，空List。排过序的空List还是空List。接下来便是算法的定义：**排过序的List就是令所有小于等于头部的元素在先(它们已经排过了序), 后跟大于头部的元素(它们同样已经拍过了序)。** 注意定义中有两次排序，所以就得递归两次！同时也需要注意算法定义的动词为"是"什么而非"做"这个,"做"那个,再"做"那个...这便是函数式编程之美！如何才能从List中取得比头部小的那些元素呢？List Comprehension。好，动手写出这个函数！

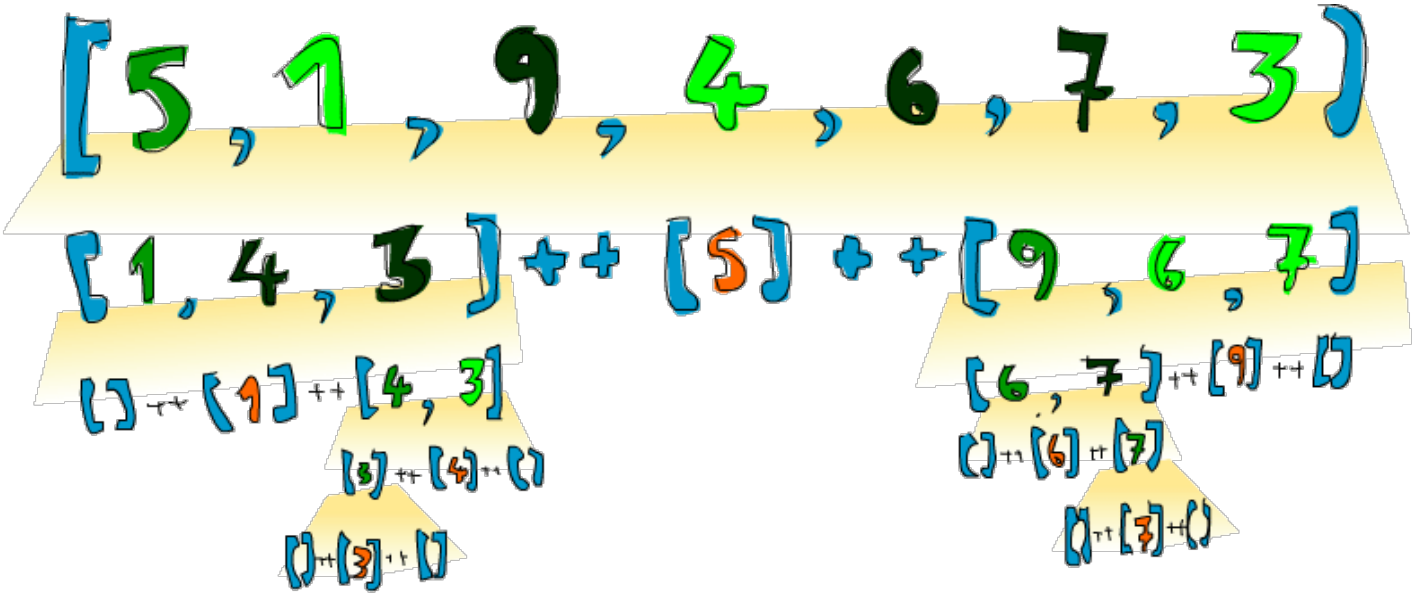
```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
  let smallerSorted = quicksort [a | a  xs, a  x]
      biggerSorted = quicksort [a | a  xs, a > x]
  in smallerSorted ++ [x] ++ biggerSorted
```

小小的测试一下, 看看结果是否正确~

```
ghci> quicksort [10,2,5,3,1,6,7,4,2,3,4,8,9]
[1,2,2,3,3,4,4,5,6,7,8,9,10]
ghci> quicksort "the quick brown fox jumps over the lazy dog"
" abcdeeefghhijklmnoooopqrrsttuuvwxyz"
```

booyah! 如我所说的一样! 若给 `[5,1,9,4,6,7,3]` 排序，这个算法就会取出它的头部，即5。 将其至于分别比它大和比它小的两个List中间，得 `[1,4,3] ++ [5] ++ [9,6,7]` ，我们便知道了当排序结束之时，5会在第四位，因为有3个数比它小每，也有三个数比它大。好的，接着排 `[1,4,3]` 与 `[9,6,7]` ，结果就出来了！对它们的排序也是使用同样的函数，将它们分成许多小块，最终到达临界条件，即空List经排序依然为空，有个插图：

橙色的部分表示已定位并不再移动的元素。从左到右看，便是一个排过序的List。在这里我们将所有元素与head作比较，而实际上就快速排序算法而言，选择任意元素都是可以的。被选择的元素就被称作锚（*pivot*），以方便模式匹配。小于锚的元素都在浅绿的部分，大于锚都在深绿部分，这个黄黄的坡就表示了快速排序的执行方式：



递归地思考

我们已经递不少归了，也许你已经发觉了其中的固定模式：先定义一个边界条件，再定义个函数，让它从一堆元素中取一个并做点事情后，把余下的元素重新交给这个函数。 这一模式对List、Tree等数据结构都是适用的。例如，sum函数就是一个List头部与其尾部的sum的和。一个List的积便是该List的头与其尾部

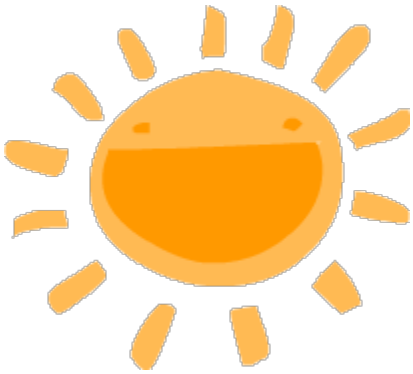
的积相乘的积，一个List的长度就是1与其尾部长度的和. 等等

再者就是边界条件。一般而言，边界条件就是为避免程序出错而设置的保护措施，处理List时的边界条件大部分都是空List，而处理Tree时的边界条件就是没有子元素的节点。

处理数字时也与之相似。函数一般都得接受一个值并修改它。早些时候我们编写过一个计算斐波纳契的函数，它便是某数与它减一的斐波纳契数的积。让它乘以零就不行了， 斐波纳契数又都是非负数，边界条件便可以定为1，即乘法的单位元。 因为任何数乘以1的结果还是这个数。而在sum中，加法的单位元就是0。在快速排序中，边界条件和单位元都是空List，因为任一List与空List相加的结果依然是原List。

使用递归来解决问题时应当先考虑递归会在什么样的条件下不可用, 然后再找出它的边界条件和单位元, 考虑参数应该在何时切开(如对List使用模式匹配), 以及在何处执行递归.

第六章 高阶函数



haskell中的函数可以作为参数和返回值传来传去，这样的函数就被称作高阶函数。高阶函数可不只是某简单特性而已，它贯穿于haskell的方方面面。要拒绝循环与状态的改变而通过定义问题"是什么"来解决问题，高阶函数必不可少。它们是编码的得力工具。

柯里函数

本质上，haskell的所有函数都只有一个参数，那么我们先前编那么多含有多个参数的函数又是怎么回事? 呵，小伎俩! 所有多个参数的函数都是柯里函数。 什么意思呢? 取一个例子最好理解，就拿我们的好朋友 `max` 函数说事吧。它看起来像是取两个参数，返回较大的那个数。 实际上，执行 `max 4 5` 时，它会首先返回一个取一个参数的函数，其返回值不是4就是该参数，取决于谁大。 然后，以5为参数调用它，并取得最终结果。 这听着挺绕口的，不过这一概念十分的酷! 如下的两个调用是等价的：

```
ghci> max 4 5
5
ghci> (max 4) 5
5
```

把空格放到两个东西之间，称作**函数调用**。它有点像个运算符，并拥有最高的优先级。 看看max函数的类型: `max :: (Ord a) => a -> a -> a` 。 也可以写作: `max :: (Ord a) => a -> (a -> a)` 。 可以读作max取一个参数a，并返回一个函数(就是那个 `->`)，这个函数取一个a类型的参数，返回一个a。 这便是为何只用箭头来分隔参数和返回值类型。

这样的好处又是如何? 简言之，我们若以不全的参数来调用某函数，就可以得到一个**不全调用**的函数。 如果你高兴，构造新函数就可以如此便捷，将其传给另一个函数也是同样方便。

看下这个函数，简单至极:

```
multThree :: (Num a) => a -> a -> a -> a
multThree x y z = x * y * z
```

我们若执行 `mulThree 3 5 9` 或 `((mulThree 3) 5) 9` ，它背后是如何运作呢？ 首先，按照空格分隔，把 `3` 交给 `mulThree` 。 这返回一个返回函数的函数。 然后把 `5` 交给它，返回一个取一个参数并使之乘以 `15` 的函数。 最后把 `9` 交给这一函数，返回 `135` 。 想想，这个函数的类型也可以写作 `multThree :: (Num a) => a -> (a -> (a -> a))` ， `->` 前面的东西就是函数取的参数，后面的东西就是其返回值。 所以说，我们的函数取一个 `a` ，并返回一个类型为 `(Num a) => a -> (a -> a)` 的函数，类似，这一函数返回一个取一个 `a` ，返回一个类型为 `(Num a) => a -> a` 的函数。 而最后的这个函数就只取一个 `a` 并返回一个 `a` ，如下:

```
ghci> let multTwoWithNine = multThree 9
ghci> multTwoWithNine 2 3
54
ghci> let multWithEighteen = multTwoWithNine 2
ghci> multWithEighteen 10
180
```

前面提到，以不全的参数调用函数可以方便地创造新的函数。例如，搞个取一数与100比较大小的函数该如何? 大可这样:

```
compareWithHundred :: (Num a，Ord a) => a -> Ordering
compareWithHundred x = compare 100 x
```

用99调用它，就可以得到一个 `GT` 。 简单。 注意下在等号两边都有x。 想想 `compare 100` 会返回什么？ 一个取一数与100比较的函数。 Wow，这不正是我们想要的? 这样重写:

```
compareWithHundred :: (Num a，Ord a) => a -> Ordering
compareWithHundred = compare 100
```

类型声明依然相同，因为 `compare 100` 返回函数。 compare的类型为 `(Ord a) => a -> (a -> Ordering)` ，用100调用它后返回的函数类型为 `(Num a，Ord a) => a -> Ordering` ，同时由于100还是 `Num` 类型类的实例，所以还得另留一个类约束。

Yo! 你得保证已经弄明白了柯里函数与不全调用的原理，它们很重要！

中缀函数也可以不全调用，用括号把它和一边的参数括在一起就行了。 这返回一个取一参数并将其补到缺少的那一端的函数。 一个简单函数如下:

```
divideByTen :: (Floating a) => a -> a
divideByTen = (/10)
```

调用 `divideByTen 200` 就是 `(/10) 200` ，和 `200 / 10` 等价。

一个检查字符是否为大写的函数:

```
isUpperAlphanum :: Char -> Bool
isUpperAlphanum = (`elem` ['A'..'Z'])
```

唯一的例外就是 `-` 运算符，按照前面提到的定义， `(-4)` 理应返回一个并将参数减4的函数，而实际上，处于计算上的方便， `(-4)` 表示负 `4` 。 若你一定要弄个将参数减4的函数，就用 `subtract` 好了，像这样 `(subtract 4)` .

若不用*let*给它命名或传到另一函数中，在ghci中直接执行 `multThree 3 4` 会怎样?

```
ghci> multThree 3 4
:1:0:
No instance for (Show (t -> t))
arising from a use of `print' at :1:0-12
Possible fix: add an instance declaration for (Show (t -> t))
In the expression: print it
In a 'do' expression: print it
```

ghci说，这一表达式返回了一个 `a -> a` 类型的函数，但它不知道该如何显示它。 函数不是 `Show` 类型类

的实例，所以我们不能得到表示一函数内容的字符串。 若在ghci中计算 `1+1` ，它会首先计算得 `2` ，然后调用 `show 2` 得到该数值的字符串表示，即"2"，再输出到屏幕.

是时候了，来点高阶函数！

haskell中的函数可以取另一个函数做参数，也可以返回函数。 举个例子，我们弄个取一个函数并调用它两次的函数.

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

首先注意这类型声明。 在此之前我们很少用到括号，因为 `(->)` 是自然的右结合，不过在这里括号是必须的。 它标明了首个参数是个参数与返回值类型都是a的函数，第二个参数与返回值的类型也都是a。 我们可以用柯里函数的思路来理解这一函数，不过免得自寻烦恼，我们姑且直接把它看作是取两个参数返回一个值，其首个参数是个类型为 `(a->a)` 的函数,第二个参数是个 `a` 。 该函数的类型可以是 `(Int->Int)` ，也可以是 `(String->String)` ，但第二个参数必须与之一致。

> **Note**: 现在开始我们会直说某函数含有多个参数(除非它真的只有一个参数)。 以简洁之名，我们会说(a->a->a)取两个参数，尽管我们知道它在背后做的手脚.

这个函数是相当的简单，就拿参数f当函数，用x调用它得到的结果再去调用它。 也就可以这样玩:

```
ghci> applyTwice (+3) 10
16
ghci> applyTwice (++ " HAHA") "HEY"
"HEY HAHA HAHA"
ghci> applyTwice ("HAHA " ++) "HEY"
"HAHA HAHA HEY"
ghci> applyTwice (multThree 2 2) 9
144
ghci> applyTwice (3:) [1]
[3,3,1]
```

看，不全调用多神奇! 如果有个函数要我们给它传个一元函数，大可以不全调用一个函数让它剩一个参数，再把它交出去。

接下来我们用高阶函数的编程思想来实现个标准库中的函数，它就是 zipWith 。 它取一个函数和两个List做参数，并把两个List交到一起(使相应的元素去调用该函数)。 如下就是我们的实现:

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

看下这个类型声明，它的首个参数是个函数，取两个参数处理交叉，其类型不必相同，不过相同也没关系。 第二三个参数都是List，返回值也是个List。 第一个List中元素的类型必须是a，因为这个处理交叉的函数的第一个参数是a。 第二个List中元素的类型必为b，因为这个处理交叉的函数第二个参数的类型是b。 返回的List中元素类型为c。 如果一个函数说取一个类型为 a->b->c 的函数做参数，传给它个 a->a->c 类型的也是可以的，但反过来就不行了。 可以记下，若在使用高阶函数的时候不清楚其类型为何，就先忽略掉它的类型声明，再到ghci下用 :t 命令来看下haskell的类型推导.

这函数的行为与普通的 zip 很相似，边界条件也是相同，只不过多了个参数，即处理元素交叉的函数。它关不着边界条件什么事儿，所以我们就只留一个_ 。后一个模式的函数体与zip也很像，只不过这里是 f x y 而非 (x,y) 。 只要足够通用，一个简单的高阶函数可以在不同的场合反复使用。 如下便是我们 zipWith' 函数本领的冰山一角:

```
ghci> zipWith' (+) [4,2,5,6] [2,6,2,3]
[6,8,7,9]
ghci> zipWith' max [6,3,2,1] [7,3,1,5]
[7,3,2,5]
ghci> zipWith' (++) ["foo " , "bar " , "baz "] ["fighters" , "hoppers" , "aldrin"]
["foo fighters","bar hoppers","baz aldrin"]
ghci> zipWith' (*) (replicate 5 2) [1..]
[2,4,6,8,10]
ghci> zipWith' (zipWith' (*)) [[1,2,3],[3,5,6],[2,3,4]] [[3,2,2],[3,4,5],[5,4,3]]
[[3,4,6],[9,20,30],[10,12,12]]
```

如你所见，一个简单的高阶函数就可以玩出很多花样。 命令式语言使用for、while、赋值、状态检测来实现功能，再包起来留个接口，使之像个函数一样调用。而函数式语言使用高阶函数来抽象出常见的模式，像成对遍历并处理两个List或从中筛掉自己不需要的结果。

接下来实现标准库中的另一个函数flip，flip简单地取一个函数作参数并返回一个相似的函数，只是它们的两个参数倒了个。

```
flip' :: (a -> b -> c) -> (b -> a -> c)
flip' f = g
    where g x y = f y x
```

从这类型声明中可以看出，它取一个函数，其参数类型分别为a和b，而它返回的函数的参数类型为b和a。由于函数默认都是柯里化的， -> 为右结合，这里的第二对括号其实并无必要，

`(a -> b -> c) -> (b -> a -> c)` 与 `(a -> b -> c) -> (b -> (a -> c))` 等价,也与 `(a -> b -> c) -> b -> a -> c` 等价。 前面我们写了 `g x y = f y x` ，既然这样可行，那么 `f y x = g x y` 不也一样? 这一来我们可以改成更简单的写法:

```
flip' :: (a -> b -> c) -> b -> a -> c
flip' f y x = f x y
```

在这里我们就利用了柯里函数的优势，只要调用 `flip' f` 而不带 `y` 和 `x` ，它就会返回一个俩参数倒个的函数。 `flip` 处理的函数往往都是用来传给其他函数调用，于是我们可以发挥柯里函数的优势，预先想好发生完全调用的情景并处理好返回值.

```
ghci> flip' zip [1,2,3,4,5] "hello"
[('h',1),('e',2),('l',3),('l',4),('o',5)]
ghci> zipWith (flip' div) [2,2..] [10,8,6,4,2]
[5,4,3,2,1]
```

map 与 filter

map取一个函数和List做参数，遍历该List的每个元素来调用该函数产生一个新的List。 看下它的类型声明和实现:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

从这类型声明中可以看出，它取一个取 `a` 返回 `b` 的函数和一组a的List，并返回一组b。 这就是haskell的有趣之处：有时只看类型声明就能对函数的行为猜个大致。 `map` 函数多才多艺，有一百万种用法。 如下是其中一小部分:

```
ghci> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
ghci> map (++ "!") ["BIFF", "BANG", "POW"]
["BIFF!","BANG!","POW!"]
ghci> map (replicate 3) [3..6]
[[3,3,3],[4,4,4],[5,5,5],[6,6,6]]
ghci> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]
[[1,4],[9,16,25,36],[49,64]]
ghci> map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]
[1,3,6,2,2]
```

你可能会发现，以上的所有代码都可以用List Comprehension来替代。 `map (+3) [1,5,3,1,6]` 与 `[x+3 | x <- [1,5,3,1,6]` 完全等价。

filter函数取一个限制条件和一个List，返回该List中所有符合该条件的元素。 它的类型声明及实现大致如

下:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

很简单。 只要 `p x` 所得的结果为真，就将这一元素加入新List，否则就无视之。几个使用范例:

```
ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
ghci> filter (==3) [1,2,3,4,5]
[3]
ghci> filter even [1..10]
[2,4,6,8,10]
ghci> let notNull x = not (null x) in filter notNull [[1,2,3],[],[3,4,5],[2,2],[],[],[]]
[[1,2,3],[3,4,5],[2,2]]
ghci> filter (`elem` ['a'..'z']) "u LaUgH aT mE BeCaUsE I aM diFfeRent"
"uagameasadifeent"
ghci> filter (`elem` ['A'..'Z']) "i lauGh At You BecAuse u r aLL the Same"
"GAYBALLS"
```

同样，以上都可以用List Comprehension的限制条件来实现。 并没有教条规定你必须在什么情况下用 `map` 和 `filter` 还是*List Comprehension*，选择权归你，看谁舒服用谁就是。 如果有多个限制条件，只能连着套好几个filter或用&&等逻辑函数的组合之，这时就不如list comprehension来得爽了。

还记得上一章的那个quicksort函数么? 我们用到了List Comprehension来过滤大于或小于锚的元素。 换做filter也可以实现，而且更加易读：

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort (filter (<x) xs)
        biggerSorted = quicksort (filter (>x) xs)
    in  smallerSorted ++ [x] ++ biggerSorted
```

map和filter是每个函数式程序员的面包黄油（呃，map和filter还是List Comprehension并不重要）。 想想前面我们如何解决给定周长寻找合适直角三角形的问题的? 在命令式编程中，我们可以套上三个循环逐个测试当前的组合是否满足条件，若满足，就打印到屏幕或其他类似的输出。 而在函数式编程中，这行就都交给map和filter。 你弄个取一参数的函数，把它交给map过一遍List，再filter之找到合适的结果。 感

谢haskell的惰性，即便是你多次map一个list也只会遍历一遍该list，要找出小于100000的数中最大的3829的倍数，只需过滤结果所在的list就行了.

要找出**小于100000的3829的所有倍数**，我们应当过滤一个已知结果所在的list.

```
largestDivisible :: (Integral a) => a
largestDivisible = head (filter p [100000,99999..])
    where p x = x `mod` 3829 == 0
```

首先， 取一个降序的小于100000所有数的List，然后按照限制条件过滤它。 由于这个List是降序的，所以结果List中的首个元素就是最大的那个数。 惰性再次行动! 由于我们只取这结果List的首个元素，所以它并不关心这List是有限还是无限的，在找到首个合适的结果处运算就停止了。

接下来，我们就要**找出所有小于10000的奇数的平方和**，得先提下takeWhile函数，它取一个限制条件和List作参数，然后从头开始遍历这一List，并返回符合限制条件的元素。 而一旦遇到不符合条件的元素，它就停止了。 如果我们要取出字符串 `"elephants know how to party"` 中的首个单词，可以 `takeWhile (/=' ') "elephants know how to party"` ，返回 `"elephants"` 。 okay，要求所有小于10000的奇数的平方的和，首先就用 `(^2)` 函数map掉这个无限的List `[1..]` 。然后过滤之，只取奇数就是了。 在大于10000处将它断开，最后前面的所有元素加到一起。 这一切连写函数都不用，在ghci下直接搞定.

```
ghci> sum (takeWhile (10000) (filter odd (map (^2) [1..])))
166650
```

不错! 先从几个初始数据(表示所有自然数的无限list)，再map它，filter它，切它，直到它符合我们的要求，再将其加起来。 这用list comprehension也是可以的，而哪种方式就全看你的个人口味.

```
ghci> sum (takeWhile (10000) [m | m  [n^2 | n  [1..]], odd m])
166650
```

感谢haskell的惰性特质，这一切才得以实现。 我们之所以可以map或filter一个无限list，是因为它的操作不会被立即执行，而是拖延一下。 只有我们要求haskell交给我们sum的结果的时候，sum函数才会跟takeWhile说，它要这些数。 `takeWhile` 就再去要求filter和map行动起来，并在遇到大于等于10000时候停止.

下个问题与Collatz序列有关，取一个自然数，若为偶数就除以2。 若为奇数就乘以3再加1。 再用相同的方式处理所得的结果，得到一组数字构成的的链。 它有个性质，无论任何以任何数字开始，最终的结果都会归1。 所以若拿13当作起始数，就可以得到这样一个序列*13，40，20，10，5，16，8，4，2，1*。 13*3+1得40，40除2得20，如是继续，得到一个10个元素的链。

好的，我们想知道的是: 以1到100之间的所有数作为起始数，会有多少个链的长度大于15?

```
chain :: (Integral a) => a -> [a]
chain 1 = [1]
chain n
    | even n =  n:chain (n `div` 2)
    | odd n  =  n:chain (n*3 + 1)
```

该链止于1，这便是边界条件。 标准的递归函数:

```
ghci> chain 10
[10,5,16,8,4,2,1]
ghci> chain 1
[1]
ghci> chain 30
[30,15,46,23,70,35,106,53,160,80,40,20,10,5,16,8,4,2,1]
```

yay! 貌似工作良好。 现在由这个函数来告诉我们结果:

```
numLongChains :: Int
numLongChains = length (filter isLong (map chain [1..100]))
    where isLong xs = length xs > 15
```

我们把chain函数map到 `[1..100]` ，得到一组链的list，然后用个限制条件过滤长度大于15的链。 过滤完毕后就可以得出结果list中的元素个数.

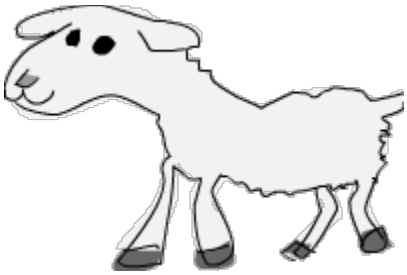
> **Note**: 这函数的类型为 `numLongChains :: Int` 。 这是由于历史原因， `length` 返回一个 `Int` 而非 `Num` 的成员类型，若要得到一个更通用的 `Num a` ，我们可以使用 `fromInterval` 函数来处理所得结果.

用map，我们可以写出类似 `map (*) [0..]` 之类的代码。 如果只是为了例证柯里函数和不全调用的函数是真正的值及其原理，那就是你可以把函数传递或把函数装在list中(只是你还不能将它们转换为字符串)。 迄今为止，我们还只是map单参数的函数到list，如 `map (*2) [0..]` 可得一组类型为 `(Num a) => [a]` 的list，而 `map (*) [0..]` 也是完全没问题的。 `*` 的类型为 `(Num a) -> a -> a -> a` ，用单个参数调用二元函数会返回一个一元函数。 如果用 `*` 来map 一个 `[0..]` 的list，就会得到一组一元函数组成的list，即 `(Num a) => [a->a]` 。 `map (*) [0..]` 所得的结果写起来大约就是 `[(*0),(*1),(*2)..]`

```
ghci> let listOfFuns = map (*) [0..]
ghci> (listOfFuns !! 4) 5
20
```

取所得list的第四个元素可得一函数，与 `(*4)` 等价。 然后用 `5` 调用它，与 `(* 4) 5` 或 `4*5` 都是等价的.

lambda



lambda就是匿名函数。有些时候我们需要传给高阶函数一个函数，而这函数我们只会用这一次，这就弄个特定功能的lambda。编写lambda，就写个 `\`（因为它看起来像是希腊字母的lambda--如果你斜视的厉害），后面是用空格分隔的参数，`->` 后面就是函数体。通常我们都是用括号将其括起，要不然它就会占据整个右边部分。

向上5英寸左右，你会看到我们在 numLongChain 函数中用where语句声明了个 isLong 函数传递给了filter。好的，用lambda代替它。

```
numLongChains :: Int
numLongChains = length (filter (\xs -> length xs > 15) (map chain [1..100]))
```

lambda是个表达式，因此我们可以任意传递。表达式 `(\xs -> length xs > 15)` 返回一个函数，它可以告诉我们一个list的长度是否大于15。

不熟悉柯里函数与不全调用的人们往往会写出很多lambda，而实际上大部分都是没必要的。例如，表达式 `map (+3) [1,6,3,2]` 与 `map (\x -> x+3) [1,6,3,2]` 等价，`(+3)` 和 `(\x -> x+3)` 都是给一个数加上3。不用说，在这种情况下不用lambda要清爽的多。

和普通函数一样，lambda也可以取多个参数。

```
ghci> zipWith (\a b -> (a * 30 + 3) / b) [5,4,3,2,1] [1,2,3,4,5]
[153.0,61.5,31.0,15.75,6.6]
```

同普通函数一样，你也可以在lambda中使用模式匹配，只是你无法为一个参数设置多个模式，如 `[]` 和 `(x:xs)`。lambda的模式匹配若失败，就会引发一个运行时错误，所以慎用！

```
ghci> map (\(a,b) -> a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)]
[3,8,9,8,7]
```

一般情况下，lambda都是括在括号中，除非我们想要后面的整个语句都作为lambda的函数体。很有趣，由于有柯里化，如下的两段是等价的：

```
addThree :: (Num a) => a -> a -> a -> a
addThree x y z = x + y + z
```

```
addThree :: (Num a) => a -> a -> a -> a
addThree = \x -> \y -> \z -> x + y + z
```

这样的函数声明与函数体中都有`->`，这一来类型声明的写法就很明白了。当然第一段代码更易读，不过第二个函数使得柯里化更容易理解。

有些时候用这种语句写还是挺酷的，我觉得这应该是最易读的flip函数实现了：

```
flip' :: (a -> b -> c) -> b -> a -> c
flip' f = \x y -> f y x
```

尽管这与`flip' f x y = f y x`等价，但它可以更明白地表示出它会产生一个新的函数。flip常用来处理一个函数，再将返回的新函数传递给map或filter。所以如此使用lambda可以更明确地表现出返回值是个函数，可以用来传递给其他函数作参数。

折叠纸鹤

回到当初我们学习递归的情景。我们会发现处理list的许多函数都有固定的模式，通常我们会将边界条件设置为空list，再引入(x:xs)模式，对单个元素和余下的list做些事情。这一模式是如此常见，因此haskell引入了一组函数来使之简化，也就是fold。它们与map有点像，只是它们返回的是单个值。

一个fold取一个二元函数，一个初始值（我喜欢管它叫累加值）和一个需要fold（折叠）的list。这个二元函数有两个参数，即累加值和list的首项（或尾项），返回值是新的累加值。然后，以新的累加值和新的list首项调用该函数，如是继续。到list遍历完毕时，只剩下一个累加值，也就是最终的结果。

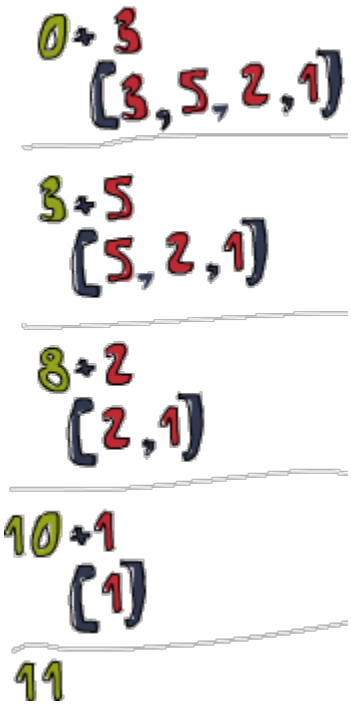
首先看下foldl函数，也叫做左折叠。它从list的左端开始折叠，用初始值和list的头部调用这二元函数，得一新的累加值，并用新的累加值与list的下一个元素调用二元函数。如是继续。

我们再实现下sum，这次用fold替代那复杂的递归：

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs
```

测试下，一二三~

```
ghci> sum' [3,5,2,1]
11
```



我们深入看下fold的执行过程：`\acc x-> acc + x`是个二元函数，`0`是初始值，`xs`是待折叠的list。一开始，累加值为`0`，当前项为`3`，调用二元函数`0+3`得`3`，作新的累加值。接着来，累加值为`3`，当前项为`5`，得新累加值`8`。再往后，累加值为`8`，当前项为`2`，得新累加值`10`。最后累加值为`10`，当前项为`1`，得`11`。恭喜，你完成了一次折叠(fold)！

左边的这个图表示了折叠的执行过程，一步又一步（一天又一天!）。浅棕色的数字都是累加值，你可以从中看出list是如何从左端一点点加到累加值上的。唔对对对！如果我们考虑到函数的柯里化，可以写出更简单的实现：

```
sum' :: (Num a) => [a] -> a
sum' = foldl (+) 0
```

这个lambda函数`(\acc x -> acc + x)`与(+)等价。我们可以把xs等一应参数省略掉，反正调用`foldl (+) 0`会返回一个取list作参数的函数。通常，如果你的函数类似`foo a = bar b a`，大可改为`foo = bar b`。有柯里化嘛。

呼呼，进入右折叠前我们再实现个用到左折叠的函数。大家肯定都知道elem是检查某元素是否属于某list的函数吧，我就不再提了（唔，刚提了）。用左折叠实现它:

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' y ys = foldl (\acc x -> if x == y then True else acc) False ys
```

好好好，这里我们有什么？起始值与累加值都是布尔值。在处理fold时，累加值与最终结果的类型总是相同的。如果你不知道怎样对待起始值，那我告诉你，我们先假设它不存在，以False开始。我们要是fold一个空list，结果就是False。然后我们检查当前元素是否为我们寻找的，如果是，就令累加值为True，如果否，就保留原值不变。若False，及表明当前元素不是。若True，就表明已经找到了。

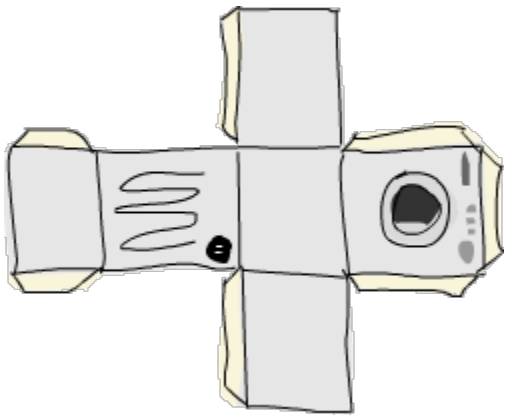
右折叠foldr的行为与左折叠相似，只是累加值是从list的右边开始。同样，左折叠的二元函数取累加值作首个参数，当前值为第二个参数（即 `\acc x -> ...` ），而右折叠的二元函数参数的顺序正好相反（即 `\x acc -> ...` ）。这倒也正常，毕竟是从右端开始折叠。

累加值可以是任何类型，可以是数值，布尔值，甚至一个新的list。我们可以用右fold实现map函数，累加值就是个list。将map处理过的元素一个一个连到一起。很容易想到，起始值就是空list。

```
map' :: (a -> b) -> [a] -> [b]
map' f xs = foldr (\x acc -> f x : acc) [] xs
```

如果我们用 `(+3)` 来映射 `[1,2,3]` ，它就会先到达list的右端，我们取最后那个元素，也就是 `3` 来调用 `(+3)` ，得 `6` 。追加 `(:)` 到累加值上， `6:[]` 得 `[6]` 并成为新的累加值。用 `2` 调用 `(+3)` ，得 `5` ，追加到累加值，于是累加值成了 `[5,6]` 。再对 `1` 调用 `(+3)` ，并将结果4追加到累加值，最终得结果 `[4,5,6]` 。

当然，我们也完全可以用左折叠来实现它， `map' f xs = foldl (\acc x -> acc ++ [f x]) [] xs` 就行了。不过问题是，使用 `(++)` 往list后面追加元素的效率要比使用 `(:)` 低得多。所以在生成新list的时候人们一般都是使用右折叠。



反转一个list，既也可以通过右折叠，也可以通过左折叠。有时甚至不需要管它们的分别，如sum函数的左右折叠实现都是十分相似。不过有个大的不同，那就是右折叠可以处理无限长度的数据结构，而左折叠不可以。将无限list从中断开执行左折叠是可以的，不过若是向右，就永远到不了头了。

所有遍历list中元素并据此返回一个值的操作都可以交给fold实现。无论何时需要遍历list并返回某值，都可以尝试下fold。因此，fold的地位可以说与map和filter并驾齐驱，同为函数式编程中最常用的函数之一。

foldl1与foldr1的行为与 `foldl` 和 `foldr` 相似，只是你无需明确提供初始值。他们假定list的首个（或末尾）

元素作为起始值，并从旁边的元素开始折叠。这一来，`sum`函数大可这样实现：`sum = foldl1 (+)`。这里待折叠的list中至少要有一个元素，若使用空list就会产生一个运行时错误。不过foldl和foldr与空list相处的就很好。所以在使用fold前，应该先想下它会不会遇到空list，如果不会遇到，大可放心使用`foldr1`和`foldl1`。

为了体会fold的威力，我们就用它实现几个库函数：

```
maximum' :: (Ord a) => [a] -> a
maximum' = foldr1 (\x acc -> if x > acc then x else acc)

reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []

product' :: (Num a) => [a] -> a
product' = foldr1 (*)

filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x acc -> if p x then x : acc else acc) []

head' :: [a] -> a
head' = foldr1 (\x _ -> x)

last' :: [a] -> a
last' = foldl1 (\_ x -> x)
```

仅靠模式匹配就可以实现head函数和last函数，而且效率也很高。这里只是为了演示，用fold的实现方法。我觉得我们这个reverse'定义的相当聪明，用一个空list做初始值，并向左展开list，从左追加到累加值，最后得到一个反转的新list。`\acc x -> x : acc`有点像`:`函数，只是参数顺序相反。所以我们可以改成`foldl (flip (:)) []`。

有个理解折叠的思路：假设我们有个二元函数f，起始值z，如果从右折叠`[3,4,5,6]`，实际上执行的就是`f 3 (f 4 (f 5 (f 6 z)))`。f会被list的尾项和累加值调用，所得的结果会作为新的累加值传入下一个调用。假设f是`(+)`，起始值z是`0`，那么就是`3 + (4 + (5 + (6 + 0)))`，或等价的前缀形式：`(+) 3 ((+) 4 ((+) 5 ((+) 6 0)))`。相似，左折叠一个list，以g为二元函数，z为累加值，它就与`g (g (g (g z 3) 4) 5) 6`等价。如果用`flip (:)`作二元函数，`[]`为累加值（看得出，我们是要反转一个list），这就与`flip (:) (flip (:) (flip (:) (flip (:) [] 3) 4) 5) 6`等价。显而易见，执行该表达式的结果为`[6,5,4,3]`。

scanl和scanr与`foldl`和`foldr`相似，只是它们会记录下累加值的所有状态到一个list。也有scanl1和scanr1。

```
ghci> scanl (+) 0 [3,5,2,1]
[0,3,8,10,11]
ghci> scanr (+) 0 [3,5,2,1]
[11,8,3,1,0]
ghci> scanl1 (\acc x -> if x > acc then x else acc) [3,4,5,3,7,9,2,1]
[3,4,5,5,7,9,9,9]
ghci> scanl (flip (:)) [] [3,2,1]
[[],[3],[2,3],[1,2,3]]
```

当使用scanl时，最终结果就是list的最后一个元素。而在scanr中则是第一个。

```
sqrtSums :: Int
sqrtSums = length (takeWhile (1000) (scanl1 (+) (map sqrt [1..]))) + 1
```

```
ghci> sqrtSums
131
ghci> sum (map sqrt [1..131])
1005.0942035344083
ghci> sum (map sqrt [1..130])
993.6486803921487
```

`scan` 可以用来跟踪fold函数的执行过程。想想这个问题，**取所有自然数的平方根的和，寻找在何处超过1000**？先 `map sqrt [1..]` ，然后用个fold来求它们的和。但在这里我们想知道求和的过程，所以使用 `scan` ， `scan` 完毕时就可以得到小于1000的所有和。所得结果list的第一个元素为1，第二个就是1+根2，第三个就是1+根2+根3。若有x个和小于1000，那结果就是x+1。

有$的函数调用

好的，接下来看看$函数。它也叫作**函数调用符**。先看下它的定义：

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

什么鬼东西？这没啥意义的操作符？它只是个函数调用符罢了？好吧，不全是，但差不多。普通的函数调用符有最高的优先级，而 `$` 的优先级则最低。用空格的函数调用符是左结合的，如f a b c与((f a) b) c等价，而$则是右结合的。

听着不错。但有什么用？它可以减少我们代码中括号的数目。试想有这个表达式：

`sum (map sqrt [1..130])` 。由于低优先级的$，我们可以将其改为 `sum $ map sqrt [1..130]` ，可以省敲不少键！ `sqrt 3 + 4 + 9` 会怎样？这会得到9，4和根3的和。若要取 `(3+4+9)` 的平方根，就得 `sqrt (3+4+9)` 或用 `$` ： `sqrt $ 3+4+9` 。因为 `$` 有最低的优先级，所以你可以把$看作是在右面写一对括号的等价形式。

`sum (filter (> 10) (map (*2) [2..10]))` 该如何？嗯，$是右结合， `f (g (z x))` 与 `f $ g $ z x` 等价。所以我么可以将 `sum (filter (> 10) (map (*2) [2..10])` 重写为 `sum $ filter (> 10) $ map (*2) [2..10]` 。

除了减少括号外，$还可以将数据作为函数使用。例如映射一个函数调用符到一组函数组成的list：

```
ghci> map ($ 3) [(4+),(10*),(^2),sqrt]
[7.0,30.0,9.0,1.7320508075688772]
```

函数组合

在数学中，函数组合是这样定义的：$(f \circ g)(x) = f(g(x))$，表示组合两个函数成为一个函数。以x调用这一函数，就与用x调用g再用所得的结果调用f等价。

haskell中的函数组合与之很像，即.函数。其定义为：

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

注意下这类型声明， `f` 的参数类型必须与 `g` 的返回类型相同。所以得到的组合函数的参数类型与 `g` 相同，返回类型与 `f` 相同。表达式 `negate . (*3)` 返回一个求一数字乘以3后的负数的函数。

函数组合的用处之一就是生成新函数，并传递给其它函数。当然我们可以用lambda实现，但大多数情况下，使用函数组合无疑更直白。假设我们有一组由数字组成的list，要将其全部转为负数，很容易就想到应先取其绝对值，再取负数，像这样：

```
ghci> map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

注意下这个lambda与那函数组合是多么的相像。用函数组合，我们可以将代码改为：

```
ghci> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

漂亮！函数组合是右结合的，我们同时组合多个函数。表达式 `f (g (z x))` 与 `(f . g . z) x` 等价。按照这个思路，我们可以将

```
ghci> map (\xs -> negate (sum (tail xs))) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

改为：

```
ghci> map (negate . sum . tail) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

不过含多个参数的函数该怎么办？好，我们可以使用不全调用使每个函数都只剩下一个参数。`sum (replicate 5 (max 6.7 8.9))` 可以重写为 `(sum . replicate 5 . max 6.7) 8.9` 或 `sum . replicate 5 . max 6.7 $ 8.9` 。在这里会产生一个函数，它取与 `max 6.7` 同样的参数，并使用结果调用 `replicate 5` 再用 `sum` 求和。最后用 `8.9` 调用该函数。不过一般你可以这么读，用8.9调用 `max 6.7` ，然后使它 `replicate 5` ，再sum之。如果你打算用函数组合来替掉那堆括号，可以先在最靠近参数的函数后面加一个 `$` ，接着就用 `.` 组合其所有函数调用，而不用管最后那个参数。如果有这样一段代码：`replicate 100 (product (map (*3) (zipWith max [1,2,3,4,5] [4,5,6,7,8])))` ，可以改为：`replicate 100 . product . map (*3) . zipWith max [1,2,3,4,5] $ [4,5,6,7,8]` 。如果表达式以3个括号结尾，就表示你可以将其修改为函数组合的形式。

函数组合的另一用途就是定义point free style（也称作pointless style）的函数。就拿我们之前写的函数作例子：

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (+) 0 xs
```

等号的两端都有个xs。由于有柯里化（Currying），我们可以省掉两端的xs。 `foldl (+) 0` 返回的就是一个取一list作参数的函数，我们把它修改为 `sum' = foldl (+) 0` ，这就是point free style。下面这个函数又该如何改成point free style呢？

```
fn x = ceiling (negate (tan (cos (max 50 x))))
```

像刚才那样简单去掉两端的x是不行的，函数体中x的右边还有括号。cos (max 50)是有错误的，你不能求一个函数的余弦。我们的解决方法就是，使用函数组合。

```
fn = ceiling . negate . tan . cos . max 50
```

漂亮！point free style会令你去思考函数的组合方式，而非数据的传递方式，更加简洁直白。你可以将一组简单的函数组合在一起，使之形成一个复杂的函数。不过函数若过于复杂，再使用point free style往往会适得其反，因此构造较长的函数组合链是不被鼓励的（虽然我本人热衷于函数组合）。更好的解决方法，就是使用let语句给中间的运算结果绑定一个名字，或者说把问题分解成几个小问题再组合到一起。这样一来我们代码的读者就可以轻松些，不必要纠结那巨长的函数组合链了。

在map和filter那节中，我们求了小于10000的所有奇数的平方的和。如下就是将其置于一个函数中的样子：

```
oddSquareSum :: Integer
oddSquareSum = sum (takeWhile (10000) (filter odd (map (^2) [1..])))
```

身为函数组合狂人，我可能会这么写：

```
oddSquareSum :: Integer
oddSquareSum = sum . takeWhile (10000) . filter odd . map (^2) $ [1..]
```

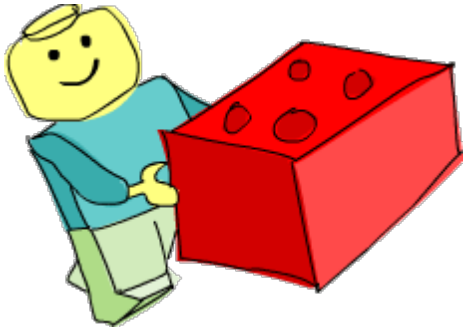
不过若是给别人看，我可能就这么写了：

```
oddSquareSum :: Integer
oddSquareSum =
    let oddSquares = filter odd $ map (^2) [1..]
        belowLimit = takeWhile (10000) oddSquares
    in  sum belowLimit
```

这段代码可赢不了代码花样大赛，不过我们的读者可能会觉得它比函数组合链更好看。

第七章 模块

装载模块



haskell中的模块是含有一组相关的函数，类型和类型类的组合。而haskell程序的本质便是从主模块中引用其它模块并调用其中的函数来执行操作。这样可以把代码分成多块，只要一个模块足够的独立，它里面的函数便可以被不同的程序反复重用。这就让不同的代码各司其职，提高了代码的健壮性。

haskell的标准库就是一组模块，每个模块都含有一组功能相近或相关的函数和类型。有处理List的模块，有处理并发的模块，也有处理复数的模块，等等。目前为止我们谈及的所有函数,类型以及类型类都是Prelude模块的一部分，它默认自动装载。在本章，我们看一下几个常用的模块，在开始浏览其中的函数之前，我们先得知道如何装载模块.

在haskell中，装载模块的语法为*import*，这必须得在函数的定义之前，所以一般都是将它置于代码的顶部。无疑，一段代码中可以装载很多模块，只要将import语句分行写开即可。装载Data.List试下，它里面有很多实用的List处理函数.

执行 `import Data.List` ，这样一来 `Data.List` 中包含的所有函数就都进入了全局命名空间。也就是说，你可以在代码的任意位置调用这些函数. `Data.List` 模块中有个 `nub` 函数，它可以筛掉一个List中的所有重复元素。用点号将 `length` 和 `nub` 组合: `length . nub` ，即可得到一个与 `(\xs -> length (nub xs))` 等价的函数。

```
import Data.List

numUniques :: (Eq a) => [a] -> Int
numUniques = length . nub
```

你也可以在GHCi中装载模块，若要调用 Data.List 中的函数，就这样:

```
ghci> :m Data.List
```

若要在GHci中装载多个模块，不必多次 :m 命令，一下就可以全部搞定:

```
ghci> :m Data.List Data.Map Data.Set
```

而你的程序中若已经有包含的代码，就不必再用 :m 了.

如果你只用得到某模块的两个函数，大可仅包含它俩。若仅装载 Data.List 模块 nub 和 sort ，就这样:

```
import Data.List (nub，sort)
```

也可以只包含除去某函数之外的其它函数，这在避免多个模块中函数的命名冲突很有用。假设我们的代码中已经有了一个叫做 nub 的函数，而装入 Data.List 模块时就要把它里面的nub除掉.

```
import Data.List hiding (nub)
```

避免命名冲突还有个方法，便是 qualified import ， Data.Map 模块提供一了一个按键索值的数据结构，它里面有几个和Prelude模块重名的函数。如 filter 和 null ，装入Data.Map模块之后再调用 filter，haskell就不知道它究竟是哪个函数。如下便是解决的方法:

```
import qualified Data.Map
```

这样一来，再调用 Data.Map 中的 filter 函数，就必须得 Data.Map.filter ，而 filter 依然是为我们熟悉喜爱的样子。但是要在每个函数前面都加 个Data.Map 实在是太烦人了! 那就给它起个别名，让它短些:

```
import qualified Data.Map as M
```

好，再调用 Data.Map 模块的 filter 函数的话仅需 M.filter 就行了

要浏览所有的标准库模块，参考这个手册。翻阅标准库中的模块和函数是提升个人haskell水平的重要途径。你也可以各个模块的源代码，这对haskell的深入学习及掌握都是大有好处的.

检索函数或搜寻函数位置就用Hoogle，相当了不起的Haskell搜索引擎! 你可以用函数名，模块名甚至类型声明来作为检索的条件.

Data.List

显而易见，Data.List是关于List操作的模块，它提供了一组非常有用的List处理函数。在前面我们已经见过了其中的几个函数(如map和filter)，这是Prelude模块出于方便起见，导出了几个Data.List里的函数。因

为这几个函数是直接引用自Data.List，所以就无需使用qulified import。在下面，我们来看看几个以前没见过的函数:

intersperse取一个元素与List作参数，并将该元素置于List中每对元素的中间。如下是个例子:

```
ghci> intersperse '.' "MONKEY"
"M.O.N.K.E.Y"
ghci> intersperse 0 [1,2,3,4,5,6]
[1,0,2,0,3,0,4,0,5,0,6]
```

intercalate取两个List作参数。它会将第一个List交叉插入第二个List中间，并返回一个List.

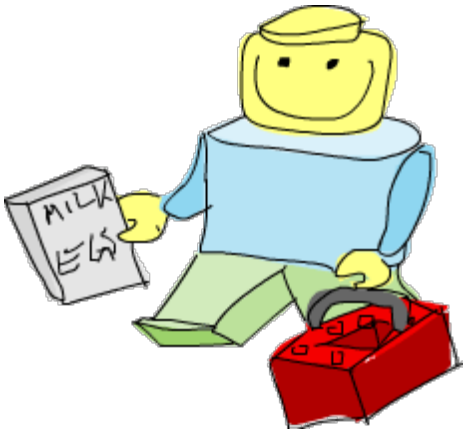
```
ghci> intercalate " " ["hey","there","guys"]
"hey there guys"
ghci> intercalate [0,0,0] [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,0,0,0,4,5,6,0,0,0,7,8,9]
```

transpose函数可以反转一组List的List。你若把一组List的List看作是个2D的矩阵，那`transpose`的操作就是将其列为行。

```
ghci> transpose [[1,2,3],[4,5,6],[7,8,9]]
[[1,4,7],[2,5,8],[3,6,9]]
ghci> transpose ["hey","there","guys"]
["htg","ehu","yey","rs","e"]
```

假如有两个多项式*3x2*+ 5x + 9，*10x3 + 9*和*8x3 + 5x2 + x - 1*，将其相加，我们可以列三个List: `[0,3,5,9]`，`[10,0,0,9]` 和 `[8,5,1,-1]` 来表示。再用如下的方法取得结果.

```
ghci> map sum $ transpose [[0,3,5,9],[10,0,0,9],[8,5,1,-1]]
[18,8,6,17]
```



使用`transpose`处理这三个List之后，三次幂就倒了第一行，二次幂到了第二行，以此类推。在用sum函数将其映射，即可得到正确的结果。

foldl'和foldl1'是它们各自惰性实现的严格版本。在用fold处理较大的List时，经常会遇到堆栈溢出的问

题。而这罪魁祸首就是fold的惰性: 在执行fold时，累加器的值并不会被立即更新，而是做一个"在必要时会取得所需的结果"的承诺。每过一遍累加器，这一行为就重复一次。而所有的这堆"承诺"最终就会塞满你的堆栈。严格的fold就不会有这一问题，它们不会作"承诺"，而是直接计算中间值的结果并继续执行下去。如果用惰性fold时经常遇到溢出错误，就应换用它们的严格版。

concat把一组List连接为一个List。

```
ghci> concat ["foo","bar","car"]
"foobarcar"
ghci> concat [[3,4,5],[2,3,4],[2,1,1]]
[3,4,5,2,3,4,2,1,1]
```

它相当于移除一级嵌套。若要彻底地连接其中的元素，你得`concat`它两次才行.

concatMap函数与map一个List之后再concat它等价.

```
ghci> concatMap (replicate 4) [1..3]
[1,1,1,1,2,2,2,2,3,3,3,3]
```

and取一组布尔值List作参数。只有其中的值全为True的情况下才会返回True。

```
ghci> and $ map (>4) [5,6,7,8]
True
ghci> and $ map (==4) [4,4,4,3,4]
False
```

or与`and`相似，一组布尔值List中若存在一个True它就返回True.

```
ghci> or $ map (==4) [2,3,4,5,6,1]
True
ghci> or $ map (>4) [1,2,3]
False
```

any和all取一个限制条件和一组布尔值List作参数，检查是否该List的某个元素或每个元素都符合该条件。通常较map一个List到and或or而言，使用any或all会更多些。

```
ghci> any (==4) [2,3,5,6,1,4]
True
ghci> all (>4) [6,9,10]
True
ghci> all (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
False
ghci> any (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
True
```

iterate取一个函数和一个值作参数。它会用该值去调用该函数并用所得的结果再次调用该函数，产生一个

无限的List.

```
ghci> take 10 $ iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512]
ghci> take 3 $ iterate (++ "haha") "haha"
["haha","hahahaha","hahahahahaha"]
```

splitAt取一个List和数值作参数，将该List在特定的位置断开。返回一个包含两个List的二元组.

```
ghci> splitAt 3 "heyman"
("hey","man")
ghci> splitAt 100 "heyman"
("heyman","")
ghci> splitAt (-3) "heyman"
("","heyman")
ghci> let (a,b) = splitAt 3 "foobar" in b ++ a
"barfoo"
```

takeWhile这一函数十分的实用。它从一个List中取元素，一旦遇到不符合条件的某元素就停止.

```
ghci> takeWhile (>3) [6,5,4,3,2,1,2,3,4,5,4,3,2,1]
[6,5,4]
ghci> takeWhile (/=' ') "This is a sentence"
"This"
```

如果要求所有三次方小于1000的数的和，用filter来过滤 `map (^3) [1..]` 所得结果中所有小于1000的数是不行的。因为对无限List执行的filter永远都不会停止。你已经知道了这个List是单增的，但haskell不知道。所以应该这样：

```
ghci> sum $ takeWhile (10000) $ map (^3) [1..]
53361
```

用 `(^3)` 处理一个无限List，而一旦出现了大于10000的元素这个List就被切断了，sum到一起也就轻而易举.

dropWhile与此相似，不过它是扔掉符合条件的元素。一旦限制条件返回False，它就返回List的余下部分。方便实用!

```
ghci> dropWhile (/=' ') "This is a sentence"
" is a sentence"
ghci> dropWhile (3) [1,2,2,2,3,4,5,4,3,2,1]
[3,4,5,4,3,2,1]
```

给一Tuple组成的List，这Tuple的首相表示股票价格，第二三四项分别表示年,月,日。我们想知道它是在哪天首次突破$1000的!

```
ghci> let stock = [(994.4,2008,9,1),(995.2,2008,9,2),(999.2,2008,9,3),(1001.4,2008,9,4),(998.3,2008,9,5)]

ghci> head (dropWhile (\(val,y,m,d) -> val 1000) stock)
(1001.4,2008,9,4)
```

span与 takeWhile 有点像，只是它返回两个List。第一个List与同参数调用takeWhile所得的结果相同，第二个List就是原List中余下的部分。

```
ghci> let (fw，rest) = span (/=' ') "This is a sentence" in "First word:" ++ fw ++ "，the rest:" ++ rest
"First word: This，the rest: is a sentence"
```

span是在条件首次为False时断开list，而break则是在条件首次为True时断开 List。break p 与 span (not 。p) 是等价的.

```
ghci> break (==4) [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
ghci> span (/=4) [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
```

break返回的第二个List就会以第一个符合条件的元素开头。

sort可以排序一个List，因为只有能够作比较的元素才可以被排序，所以这一List的元素必须是Ord类型类的实例类型。

```
ghci> sort [8,5,3,2,1,6,4,2]
[1,2,2,3,4,5,6,8]
ghci> sort "This will be sorted soon"
" Tbdeehiillnooorssstw"
```

group取一个List作参数，并将其中相邻并相等的元素各自归类，组成一个个子List.

```
ghci> group [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[[1,1,1,1],[2,2,2,2],[3,3],[2,2,2],[5],[6],[7]]
```

若在group一个List之前给它排序就可以得到每个元素在该List中的出现次数。

```
ghci> map (\l@(x:xs) -> (x,length l)) . group . sort $ [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[(1,4),(2,7),(3,2),(5,1),(6,1),(7,1)]
```

inits和tails与 init 和 tail 相似，只是它们会递归地调用自身直到什么都不剩，看:

```
ghci> inits "w00t"
["","w","w0","w00","w00t"]
ghci> tails "w00t"
["w00t","00t","0t","t",""]
ghci> let w = "w00t" in zip (inits w) (tails w)
[("","w00t"),("w","00t"),("w0","0t"),("w00","t"),("w00t","")]
```

我们用fold实现一个搜索子List的函数:

```
search :: (Eq a) => [a] -> [a] -> Bool
search needle haystack =
  let nlen = length needle
  in foldl (\acc x -> if take nlen x == needle then True else acc) False (tails haystack)
```

首先，对搜索的List调用tails，然后遍历每个List来检查它是不是我们想要的.

```
ghci> "cat" `isInfixOf` "im a cat burglar"
True
ghci> "Cat" `isInfixOf` "im a cat burglar"
False
ghci> "cats" `isInfixOf` "im a cat burglar"
False
```

由此我们便实现了一个类似isIndexOf的函数，isInfixOf从一个List中搜索一个子List，若该List包含子List，则返回True.

isPrefixOf与isSuffixOf分别检查一个List是否以某子List开头或者结尾.

```
ghci> "hey" `isPrefixOf` "hey there!"
True
ghci> "hey" `isPrefixOf` "oh hey there!"
False
ghci> "there!" `isSuffixOf` "oh hey there!"
True
ghci> "there!" `isSuffixOf` "oh hey there"
False
```

elem与notElem检查一个List是否包含某元素.

partition取一个限制条件和List作参数，返回两个List，第一个List中包含所有符合条件的元素，而第二个List中包含余下的.

```
ghci> partition (`elem` ['A'..'Z']) "BOBsidneyMORGANeddy"
("BOBMORGAN","sidneyeddy")
ghci> partition (>3) [1,3,5,6,3,2,1,0,3,7]
([5,6,7],[1,3,3,2,1,0,3])
```

了解 span 和 break 的差异是很重要的.

```
ghci> span (`elem` ['A'..'Z']) "BOBsidneyMORGANeddy"
("BOB","sidneyMORGANeddy")
```

span和break会在遇到第一个符合或不符合条件的元素处断开，而 partition 则会遍历整个List。

find取一个List和限制条件作参数，并返回首个符合该条件的元素，而这个元素是个Maybe值。在下章，我们将深入地探讨相关的算法和数据结构，但在这里你只需了解Maybe值是Just something或Nothing就够了。与一个List可以为空也可以包含多个元素相似，一个Maybe可以为空，也可以是单一元素。同样与List类似，一个Int型的List可以写作Int，Maybe有个Int型可以写作Maybe Int。先试一下find函数再说.

```
ghci> find (>4) [1,2,3,4,5,6]
Just 5
ghci> find (>9) [1,2,3,4,5,6]
Nothing
ghci> :t find
find :: (a -> Bool) -> [a] -> Maybe a
```

注意一下 find 的类型，它的返回结果为 Maybe a‥，这与 a的写法有点像，只是Maybe型的值只能为空或者单一元素，而List可以为空,一个元素，也可以是多个元素.

想想前面那段找股票的代码， head (dropWhile (\(val,y,m,d) -> val < 1000) stock) 。但head并不安全!如果我们的股票没涨过$1000会怎样? dropWhile 会返回一个空List，而对空List取head就会引发一个错误。把它改成 find (\(val,y,m,d) -> val > 1000) stock 就安全多啦，若存在合适的结果就得到它,像 Just (1001.4,2008,9,4) ，若不存在合适的元素(即我们的股票没有涨到过$1000)，就会得到一个Nothing.

elemIndex与 elem 相似，只是它返回的不是布尔值，它只是'可能'(Maybe)返回我们找的元素的索引，若这一元素不存在，就返回 Nothing 。

```
ghci> :t elemIndex
elemIndex :: (Eq a) => a -> [a] -> Maybe Int
ghci> 4 `elemIndex` [1,2,3,4,5,6]
Just 3
ghci> 10 `elemIndex` [1,2,3,4,5,6]
Nothing
```

elemIndices与 elemIndex 相似，只不过它返回的是List，就不需要Maybe了。因为不存在用空List就可以表示，这就与Nothing相似了.

```
ghci> ' ' `elemIndices` "Where are the spaces?"
[5,9,13]
```

findIndex与 find 相似，但它返回的是可能存在的首个符合该条件元素的索引。findIndices会返回所有符

合条件的索引.

```
ghci> findIndex (==4) [5,3,2,1,6,4]
Just 5
ghci> findIndex (==7) [5,3,2,1,6,4]
Nothing
ghci> findIndices (`elem` ['A'..'Z']) "Where Are The Caps?"
[0,6,10,14]
```

在前面，我们讲过了zip和zipWidth，它们只能将两个List组到一个二元组数或二参函数中，但若要组三个List该怎么办? 好说~有 zip3，zip4 ,,,,和 zipWith3，zipWidth4 ...直到7。这看起来像是个hack，但工作良好。连着组8个List的情况很少遇到。还有个聪明办法可以组起无限多个List，但限于我们目前的水平，就先不谈了.

```
ghci> zipWith3 (\x y z -> x + y + z) [1,2,3] [4,5,2,2] [2,2,3]
[7,9,8]
ghci> zip4 [2,3,3] [2,2,2] [5,5,3] [2,2,2]
[(2,2,5,2),(3,2,5,2),(3,2,3,2)]
```

与普通的 zip 操作相似，以返回的List中长度最短的那个为准.

在处理来自文件或其它地方的输入时，lines会非常有用。它取一个字符串作参数。并返回由其中的每一行组成的List.

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

'\n'表示unix下的换行符，在haskell的字符中，反斜杠表示特殊字符.

unlines是lines的反函数，它取一组字符串的List，并将其通过'\n'合并到一块.

```
ghci> unlines ["first line"，"second line"，"third line"]
"first line\nsecond line\nthird line\n"
```

words和unwords可以把一个字符串分为一组单词或执行相反的操作，很有用.

```
ghci> words "hey these are the words in this sentence"
["hey","these","are","the","words","in","this","sentence"]
ghci> words "hey these are the words in this\nsentence"
["hey","these","are","the","words","in","this","sentence"]
ghci> unwords ["hey","there","mate"]
"hey there mate"
```

我们前面讲到了nub，它可以将一个List中的重复元素全部筛掉，使该List的每个元素都如雪花般独一无二，'nub'的含义就是'一小块'或'一部分'，用在这里觉得很古怪。我觉得，在函数的命名上应该用更确切的

词语，而避免使用老掉牙的过时词汇.

```
ghci> nub [1,2,3,4,3,2,1,2,3,4,3,2,1]
[1,2,3,4]
ghci> nub "Lots of words and stuff"
"Lots fwrdanu"
```

delete取一个元素和List作参数，会删掉该List中首次出现的这一元素.

```
ghci> delete 'h' "hey there ghang!"
"ey there ghang!"
ghci> delete 'h' 。delete 'h' $ "hey there ghang!"
"ey tere ghang!"
ghci> delete 'h' 。delete 'h' 。delete 'h' $ "hey there ghang!"
"ey tere gang!"
```

\表示List的差集操作，这与集合的差集很相似，它会除掉左边List中所有存在于右边List中的元素.

```
ghci> [1..10] \\ [2,5,9]
[1,3,4,6,7,8,10]
ghci> "Im a big baby" \\ "big"
"Im a baby"
```

union与集合的并集也是很相似，它返回两个List的并集，即遍历第二个List若存在某元素不属于第一个List，则追加到第一个List。看，第二个List中的重复元素就都没了!

```
ghci> "hey man" `union` "man what's up"
"hey manwt'sup"
ghci> [1..7] `union` [5..10]
[1,2,3,4,5,6,7,8,9,10]
```

intersection相当于集合的交集。它返回两个List的相同部分.

```
ghci> [1..7] `intersect` [5..10]
[5,6,7]
```

insert可以将一个元素插入一个可排序的List，并将其置于首个大于它的元素之前，如果使用insert来给一个排过序的List插入元素，返回的结果依然是排序的.

```
ghci> insert 4 [1,2,3,5,6,7]
[1,2,3,4,5,6,7]
ghci> insert 'g' $ ['a'..'f'] ++ ['h'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> insert 3 [1,2,4,3,2,1]
[1,2,3,4,3,2,1]
```

length，take，drop，splitAt和replace之类的函数有个共同点。那就是它们的参数中都有个Int值，我觉得使用Intergal或Num类型类会更好，但出于历史原因，修改这些会破坏掉许多既有的代码。在Data.List中包含了更通用的替代版，
如:genericLength，genericTake，genericDrop，genericSplitAt，genericIndex 和 genericReplicate。length的类型声明为 `length :: [a] -> Int` ，而我们若要像这样求它的平均值，`let xs = [1..6] in sum xs / length xs` ，就会得到一个类型错误，因为 `/` 运算符不能对Int型使用! 而 `genericLength` 的类型声明则 为genericLength :: (Num a) => [b] -> a ，Num既可以是整数又可以是浮点数， `let xs = [1..6] in sum xs / genericLength xs` 这样再求平均数就不会有问题了.

`nub` , `delete` , `union` , `intsect` 和 `group` 函数也有各自的通用替代版 `nubBy` ， `deleteBy` ， `unionBy` ， `intersectBy` 和 `groupBy` ，它们的区别就是前一组函数使用(==)来测试是否相等，而带By的那组则取一个函数作参数来判定相等性，group就与groupBy (==)等价.

假如有个记录某函数在每秒的值的List，而我们要按照它小于零或者大于零的交界处将其分为一组子List。如果用 `group` ，它只能将相邻并相等的元素组到一起，而在这里我们的标准是它是否为负数。groupBy登场! 它取一个含两个参数的函数作为参数来判定相等性.

```
ghci> let values = [-4.3，-2.4，-1.2，0.4，2.3，5.9，10.5，29.1，5.3，-2.4，-14.5，2.9，2.3]
ghci> groupBy (\x y -> (x > 0) == (y > 0)) values
[[-4.3,-2.4,-1.2],[0.4,2.3,5.9,10.5,29.1,5.3],[-2.4,-14.5],[2.9,2.3]]
```

这样一来我们就可以很清楚地看出哪部分是正数，哪部分是负数，这个判断相等性的函数会在两个元素同时大于零或同时小于零时返回True。也可以写作 `\x y -> (x > 0) && (y > 0) || (x <= 0) && (y <= 0)` 。但我觉得第一个写法的可读性更高。Data.Function中还有个on函数可以让它的表达更清晰，其定义如下:

```
on :: (b -> b -> c) -> (a -> b) -> a -> a -> c
f `on` g = \x y -> f (g x) (g y)
```

执行 `(==) on (> 0)` 得到的函数就与 `\x y -> (x > 0) == (y > 0)` 基本等价。 `on` 与带By的函数在一起会非常好用，你可以这样写:

```
ghci> groupBy ((==) `on` (> 0)) values
[[-4.3,-2.4,-1.2],[0.4,2.3,5.9,10.5,29.1,5.3],[-2.4,-14.5],[2.9,2.3]]
```

可读性很高! 你可以大声念出来: 按照元素是否大于零，给它分类！

同样， `sort` ， `insert` ， `maximum` 和 `min` 都有各自的通用版本。如 `groupBy` 类似，sortBy，insertBy，maximumBy和minimumBy都取一个函数来比较两个元素的大小。像sortBy的类型声明为: `sortBy :: (a -> a -> Ordering) -> [a] -> [a]` 。前面提过，Ordering类型可以有三个值, `LT` ， `EQ` 和 `GT` 。 `compare` 取两个Ord类型类的元素作参数，所以 `sort` 与 `sortBy compare` 等价.

List是可以比较大小的，且比较的依据就是其中元素的大小。如果按照其子List的长度为标准当如何? 很

好，你可能已经猜到了，sortBy函数.

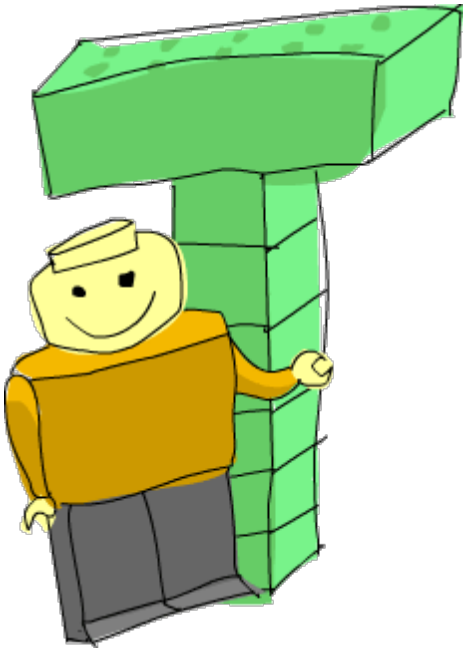
```
ghci> let xs = [[5,4,5,4,4],[1,2,3],[3,5,4,3],[],[2],[2,2]]
ghci> sortBy (compare `on` length) xs
[[],[2],[2,2],[1,2,3],[3,5,4,3],[5,4,5,4,4]]
```

太绝了! `compare on length` ，乖乖，这简直就是英文! 如果你搞不清楚on在这里的原理，就可以认为它与 `\x y -> length x compare length y` 等价。通常，与带By的函数打交道时，若要判断相等性，则 `(==) on something` 。若要判定大小，则 `compare on something` .

Data.Char

如其名， `Data.Char` 模块包含了一组用于处理字符的函数。由于字符串的本质就是一组字符的List，所以往往会在filter或是map字符串时用到它.

`Data.Char` 模块中含有一系列用于判定字符范围的函数，如下:



isControl判断一个字符是否是控制字符。

isSpace判断一个字符是否是空格字符，包括空格，tab，换行符等.

isLower判断一个字符是否为小写.

isUper判断一个字符是否为大写。

isAlpha判断一个字符是否为字母.

isAlphaNum判断一个字符是否为字母或数字.

isPrint判断一个字符是否是可打印的.

isDigit判断一个字符是否为数字.

isOctDigit判断一个字符是否为八进制数字.

isHexDigit判断一个字符是否为十六进制数字.

isLetter判断一个字符是否为字母.

isMark判断是否为unicode注音字符，你如果是法国人就会经常用到的.

isNumber判断一个字符是否为数字.

isPunctuation判断一个字符是否为标点符号.

isSymbol判断一个字符是否为货币符号.

isSeperater判断一个字符是否为unicode空格或分隔符.

isAscii判断一个字符是否在unicode字母表的前128位。

isLatin1判断一个字符是否在unicode字母表的前256位.

isAsciiUpper判断一个字符是否为大写的ascii字符.

isAsciiLower判断一个字符是否为小写的ascii字符.

以上所有判断函数的类型声明皆为 `Char -> Bool` ，用到它们的绝大多数情况都无非就是过滤字符串或类似操作。假设我们在写个程序，它需要一个由字符和数字组成的用户名。要实现对用户名的检验，我们可以结合使用 `Data.List` 模块的all函数与 `Data.Char` 的判断函数.

```
ghci> all isAlphaNum "bobby283"
True
ghci> all isAlphaNum "eddy the fish!"
False
```

Kewl~ 免得你忘记， `all` 函数取一个判断函数和一个List做参数，若该List的所有元素都符合条件，就返回 `True` .

也可以使用 `isSpace` 来实现 `Data.List` 的 `words` 函数.

```
ghci> words "hey guys its me"
["hey","guys","its","me"]
ghci> groupBy ((==) `on` isSpace) "hey guys its me"
["hey"," ","guys"," ","its"," ","me"]
ghci>
```

Hmm，不错，有点 `words` 的样子了。只是还有空格在里面，恩，该怎么办? 我知道，用 `filter` 滤掉它们!

啊哈.

`Data.Char` 中也含有与 `Ordering` 相似的类型。 `Ordering` 可以有两个值， `LT` ， `GT` 和 `EQ` 。这就是个

枚举，它表示了两个元素作比较可能的结果.GeneralCategory类型也是个枚举，它表示了一个字符可能所在的分类。而得到一个字符所在分类的主要方法就是使用 `generalCategory` 函数.它的类型为: `generalCategory :: Char -> GeneralCategory` 。那31个分类就不在此一一列出了，试下这个函数先:

```
ghci> generalCategory ' '
Space
ghci> generalCategory 'A'
UppercaseLetter
ghci> generalCategory 'a'
LowercaseLetter
ghci> generalCategory '.'
OtherPunctuation
ghci> generalCategory '9'
DecimalNumber
ghci> map generalCategory " \t\nA9?|"
[Space,Control,Control,UppercaseLetter,DecimalNumber,OtherPunctuation,MathSymbol]
```

由于 `GeneralCategory` 类型是Eq类型类的一部分，使用类似 `generalCategory c == Space` 的代码也是可以的.

toUpper将一个字符转为大写字母，若该字符不是小写字母，就按原值返回.

toLower将一个字符转为小写字母，若该字符不是大写字母，就按原值返回.

toTitle将一个字符转为title-case，对大多数字符而言，title-case就是大写.

digitToInt将一个字符转为Int值，而这一字符必须得在 `'1'..'9','a'..'f'` 或 `'A'..'F'` 的范围之内.

```
ghci> map digitToInt "34538"
[3,4,5,3,8]
ghci> map digitToInt "FF85AB"
[15,15,8,5,10,11]
```

`intToDigit` 是 `digitToInt` 的反函数。它取一个 `0` 到 `15` 的 `Int` 值作参数，并返回一个小写的字符.

```
ghci> intToDigit 15
'f'
ghci> intToDigit 5
'5'
```

ord与char函数可以将字符与其对应的数字相互转换.

```
ghci> ord 'a'
97
ghci> chr 97
'a'
ghci> map ord "abcdefgh"
[97,98,99,100,101,102,103,104]
```

两个字符的`ord`值之差就是它们在unicode字符表上的距离.

*Caesar ciphar*是加密的基础算法，它将消息中的每个字符都按照特定的字母表进行替换。它的实现非常简单，我们这里就先不管字母表了.

```
encode :: Int -> String -> String
encode shift msg =
  let ords = map ord msg
  shifted = map (+ shift) ords
  in map chr shifted
```

先将一个字符串转为一组数字，然后给它加上某数，再转回去。如果你是标准的组合牛仔，大可将函数写为: `map (chr 。(+ shift) 。ord) msg` 。试一下它的效果:

```
ghci> encode 3 "Heeeeey"
"Khhhhh|"
ghci> encode 4 "Heeeeey"
"Liiiii}"
ghci> encode 1 "abcd"
"bcde"
ghci> encode 5 "Marry Christmas! Ho ho ho!"
"Rfww~%Hmwnxyrfx&%Mt%mt%mt&"
```

不错。再简单地将它转成一组数字，减去某数后再转回来就是解密了.

```
decode :: Int -> String -> String
decode shift msg = encode (negate shift) msg
```

```
ghci> encode 3 "Im a little teapot"
"Lp#d#olwwoh#whdsrw"
ghci> decode 3 "Lp#d#olwwoh#whdsrw"
"Im a little teapot"
ghci> decode 5 . encode 5 $ "This is a sentence"
"This is a sentence"
```

Data.Map

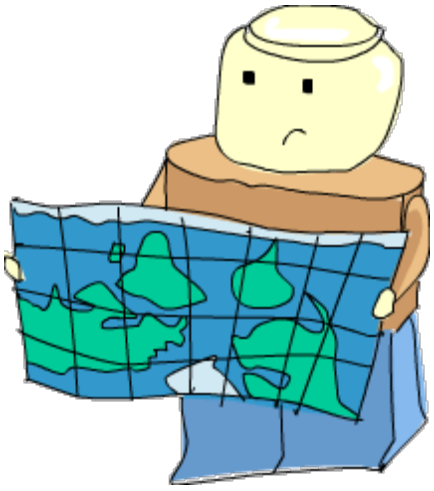
关联列表(也叫做字典)是按照键值对排列而没有特定顺序的一种List。例如，我们用关联列表储存电话号

码，号码就是值，人名就是键。我们并不关心它们的存储顺序，只要能按人名得到正确的号码就好.在haskell中表示关联列表的最简单方法就是弄一个二元组的List，而这二元组就首项为键，后项为值。如下便是个表示电话号码的关联列表:

```
phoneBook = [("betty","555-2938") ,
             ("bonnie","452-2928") ,
             ("patsy","493-2928") ,
             ("lucille","205-2928") ,
             ("wendy","939-8282") ,
             ("penny","853-2492") ]
```

不理这貌似古怪的缩进，它就是一组二元组的List而已。话说对关联列表最常见的操作就是按键索值，我们就写个函数来实现它。

```
findKey :: (Eq k) => k -> [(k,v)] -> v
findKey key xs = snd . head . filter (\(k,v) -> key == k) $ xs
```



简洁漂亮。这个函数取一个键和List做参数，过滤这一List仅保留键匹配的项，并返回首个键值对。但若该关联列表中不存在这个键那会怎样? 哼，那就会在试图从空List中取head时引发一个运行时错误。无论如何也不能让程序就这么轻易地崩溃吧，所以就应该用Maybe类型。如果没找到相应的键，就返回Nothing。而找到了就返回 `Just something` 。而这something就是键对应的值。

```
findKey :: (Eq k) => k -> [(k,v)] -> Maybe v
findKey key [] = Nothing findKey key ((k,v):xs) =
    if key == k then
        Just v
    else
        findKey key xs
```

看这类型声明，它取一个可判断相等性的键和一个关联列表做参数，可能(Maybe)得到一个值。听起来不错.这便是个标准的处理List的递归函数，边界条件，分割List，递归调用，都有了 -- 经典的fold模式。

看看用fold怎样实现吧。

```
findKey :: (Eq k) => k -> [(k,v)] -> Maybe v
findKey key = foldr (\(k,v) acc -> if key == k then Just v else acc) Nothing
```

> **Note**: 通常，使用fold来替代类似的递归函数会更好些。用fold的代码让人一目了然，而看明白递归则得多花点脑子。

```
ghci> findKey "penny" phoneBook
Just "853-2492"
ghci> findKey "betty" phoneBook
Just "555-2938"
ghci> findKey "wilma" phoneBook
Nothing
```

如魔咒般灵验! 只要我们有这姑娘的号码就Just可以得到，否则就是Nothing.方才我们实现的函数便是Data.List模块的lookup，如果要按键去寻找相应的值，它就必须得遍历整个List，直到找到为止。而 `Data.Map` 模块提供了更高效的方式(通过树实现)，并提供了一组好用的函数。从现在开始，我们扔掉关联列表，改用map.由于 `Data.Map` 中的一些函数与Prelude和 `Data.List` 模块存在命名冲突，所以我们使用 qualified import 。 `import qualified Data.Map as Map` 在代码中加上这句，并load到ghci中.继续前进，看看 `Data.Map` 是如何的一座宝库!

如下便是其中函数的一瞥:

fromList取一个关联列表，返回一个与之等价的map。

```
ghci> Map.fromList [("betty","555-2938"),("bonnie","452-2928"),("lucille","205-2928")]
fromList [("betty","555-2938"),("bonnie","452-2928"),("lucille","205-2928")]
ghci> Map.fromList [(1,2),(3,4),(3,2),(5,5)]
fromList [(1,2),(3,2),(5,5)]
```

若其中存在重复的键,就将其忽略。如下即 `fromList` 的类型声明。

```
Map.fromList :: (Ord k) => [(k，v)] -> Map.Map k v
```

这表示它取一组键值对的List，并返回一个将k映射为v的map。注意一下，当使用普通的关联列表时，只需要键的可判断相等性就行了。而在这里，它还必须得是可排序的。这在Data.Map模块中是强制的。因为它会按照某顺序将其组织在一棵树中.在处理键值对时，只要键的类型属于Ord类型类，就应该尽量使用Data.Map.empty返回一个空map.

```
ghci> Map.empty
fromList []
```

insert取一个键，一个值和一个map做参数，给这个map插入新的键值对，并返回一个新的map。

```
ghci> Map.empty
fromList []
ghci> Map.insert 3 100
Map.empty fromList [(3,100)]
ghci> Map.insert 5 600 (Map.insert 4 200 ( Map.insert 3 100  Map.empty))
fromList [(3,100),(4,200),(5,600)]
ghci> Map.insert 5 600 。Map.insert 4 200 . Map.insert 3 100 $ Map.empty
fromList [(3,100),(4,200),(5,600)]
```

通过 empty ， insert 与 fold ，我们可以编写出自己的 fromList 。

```
fromList' :: (Ord k) => [(k,v)] -> Map.Map k v
fromList' = foldr (\(k,v) acc -> Map.insert k v acc) Map.empty
```

多直白的fold！ 从一个空的map开始，然后从右折叠，随着遍历不断地往map中插入新的键值对.

null检查一个map是否为空.

```
ghci> Map.null Map.empty
True
ghci> Map.null $ Map.fromList [(2,3),(5,5)]
False
```

size返回一个map的大小。

```
ghci> Map.size Map.empty
0
ghci> Map.size $ Map.fromList [(2,4),(3,3),(4,2),(5,4),(6,4)]
5
```

singleton取一个键值对做参数,并返回一个只含有一个映射的map.

```
ghci> Map.singleton 3 9
fromList [(3,9)]
ghci> Map.insert 5 9 $ Map.singleton 3 9
fromList [(3,9),(5,9)]
```

lookup与 Data.List 的 lookup 很像,只是它的作用对象是map，如果它找到键对应的值。就返回 Just something ，否则返回 Nothing 。

member是个判断函数，它取一个键与map做参数，并返回该键是否存在于该map。

```
ghci> Map.member 3 $ Map.fromList [(3,6),(4,3),(6,9)]
True
ghci> Map.member 3 $ Map.fromList [(2,5),(4,5)]
False
```

map与filter与其对应的List版本很相似:

```
ghci> Map.map (*100) $ Map.fromList [(1,1),(2,4),(3,9)]
fromList [(1,100),(2,400),(3,900)]
ghci> Map.filter isUpper $ Map.fromList [(1,'a'),(2,'A'),(3,'b'),(4,'B')]
fromList [(2,'A'),(4,'B')]
```

toList是fromList的反函数。

```
ghci> Map.toList .Map.insert 9 2 $ Map.singleton 4 3
[(4,3),(9,2)]
```

keys与elems各自返回一组由键或值组成的List，keys与`map fst 。Map.toList`等价，`elems`与`map snd 。Map.toList`等价. `fromListWith`是个很酷的小函数，它与fromList很像，只是它不会直接忽略掉重复键，而是交给一个函数来处理它们。假设一个姑娘可以有多个号码，而我们有个像这样的关联列表:

```
phoneBook =
    [("betty","555-2938"),
    ("betty","342-2492"),
    ("bonnie","452-2928"),
    ("patsy","493-2928"),
    ("patsy","943-2929"),
    ("patsy","827-9162"),
    ("lucille","205-2928"),
    ("wendy","939-8282"),
    ("penny","853-2492"),
    ("penny","555-2111")]
```

如果用`fromList`来生成map，我们会丢掉许多号码! 如下才是正确的做法:

```
phoneBookToMap :: (Ord k) => [(k，String)] -> Map.Map k String phoneBookToMap xs = Map.fromLi
stWith (\number1 number2 -> number1 ++ "，" ++ number2) xs ghci> Map.lookup "patsy" $ phone
BookToMap phoneBook
"827-9162，943-2929，493-2928"
ghci> Map.lookup "wendy" $ phoneBookToMap phoneBook
"939-8282"
ghci> Map.lookup "betty" $ phoneBookToMap phoneBook
"342-2492，555-2938"
```

一旦出现重复键，这个函数会将不同的值组在一起，同样，也可以默认地将每个值放到一个单元素的List

中，再用 `++` 将他们都连接在一起。

```
phoneBookToMap :: (Ord k) => [(k，a)] -> Map.Map k [a]
phoneBookToMap xs = Map.fromListWith (++) $ map (\(k,v) -> (k,[v])) xs
ghci> Map.lookup "patsy" $ phoneBookToMap phoneBook
["827-9162","943-2929","493-2928"]
```

很简洁! 它还有别的玩法，例如在遇到重复元素时，单选最大的那个值.

```
ghci> Map.fromListWith max [(2,3),(2,5),(2,100),(3,29),(3,22),(3,11),(4,22),(4,15)]
fromList [(2,100),(3,29),(4,22)]
```

或是将相同键的值都加在一起.

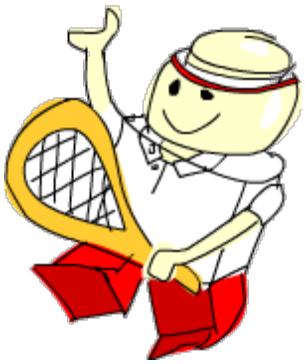
```
ghci> Map.fromListWith (+) [(2,3),(2,5),(2,100),(3,29),(3,22),(3,11),(4,22),(4,15)]
fromList [(2,108),(3,62),(4,37)]
```

insertWith之于 `insert` ，恰如 fromListWith 之于 fromList 。它会将一个键值对插入一个map之中，而该map若已经包含这个键，就问问这个函数该怎么办。

```
ghci> Map.insertWith (+) 3 100 $ Map.fromList [(3,4),(5,103),(6,339)]
fromList [(3,104),(5,103),(6,339)]
```

`Data.Map` 里面还有不少函数，这个文档中的列表就很全了.

Data.Set



`Data.Set` 模块提供了对数学中集合的处理。集合既像List也像Map: 它里面的每个元素都是唯一的，且内部的数据由一棵树来组织(这和Data.Map模块的map很像)， 必须得是可排序的。同样是插入,删除,判断从属关系之类的操作，使用集合要比List快得多。对一个集合而言，最常见的操作莫过于并集，判断从属或是将集合转为List

由于Data.Set模块与Prelude模块和Data.List模块中存在大量的命名冲突，所以我们使用*qualified import*

将import语句至于代码之中:

```
import qualified Data.Set as Set
```

然后在GHci中装载

假定我们有两个字符串，要找出同时存在于两个字符串的字符

```
text1 = "I just had an anime dream。Anime..。Reality..。Are they so different?"
text2 = "The old man left his garbage can out and now his trash is all over my lawn!"
```

fromList函数同你想的一样，它取一个List作参数并将其转为一个集合

```
ghci> let set1 = Set.fromList text1
ghci> let set2 = Set.fromList text2
ghci> set1
fromList " .?AIRadefhijlmnorstuy"
ghci> set2
fromList " !Tabcdefghilmnorstuvwy"
```

如你所见，所有的元素都被排了序。而且每个元素都是唯一的。现在我们取它的交集看看它们共同包含的元素:

```
ghci> Set.intersection set1 set2
fromList " adefhilmnorstuy"
```

使用 `difference` 函数可以得到存在于第一个集合但不在第二个集合的元素

```
ghci> Set.difference set1 set2
fromList ".?AIRj"
ghci> Set.difference set2 set1
fromList "!Tbcgvw"
```

也可以使用 `union` 得到两个集合的并集

```
ghci> Set.union set1 set2
fromList " !.?AIRTabcdefghijlmnorstuvwy"
```

`null` , `size` , `member` , `empty` , `singleton` , `insert` , `delete` 这几个函数就跟你想的差不多啦

```
ghci> Set.null Set.empty
True
ghci> Set.null $ Set.fromList [3,4,5,5,4,3]
False
ghci> Set.size $ Set.fromList [3,4,5,3,4,5]
3
ghci> Set.singleton 9
fromList [9]
ghci> Set.insert 4 $ Set.fromList [9,3,8,1]
fromList [1,3,4,8,9]
ghci> Set.insert 8 $ Set.fromList [5..10]
fromList [5,6,7,8,9,10]
ghci> Set.delete 4 $ Set.fromList [3,4,5,4,3,4,5]
fromList [3,5]
```

也可以判断子集与真子集，如果集合A中的元素都属于集合B，那么A就是B的子集 如果A中的元素都属于B且B的元素比A多，那A就是B的真子集

```
ghci> Set.fromList [2,3,4] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
True
ghci> Set.fromList [1,2,3,4,5] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
True
ghci> Set.fromList [1,2,3,4,5] `Set.isProperSubsetOf` Set.fromList [1,2,3,4,5]
False
ghci> Set.fromList [2,3,4,8] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
False
```

对集合也可以执行 `map` 和 `filter`

```
ghci> Set.fromList [2,3,4] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
True
ghci> Set.fromList [1,2,3,4,5] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
True
ghci> Set.fromList [1,2,3,4,5] `Set.isProperSubsetOf` Set.fromList [1,2,3,4,5]
False
ghci> Set.fromList [2,3,4,8] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
False
```

集合有一常见用途，那就是先 `fromList` 删掉重复元素后再 `toList` 转回去。尽管Data.List模块的 `nub` 函数完全可以完成这一工作，但在对付大List时则会明显的力不从心。使用集合则会快很多， `nub` 函数只需List中的元素属于Eq类型类就行了，而若要使用集合，它必须得属于Ord类型类

```
ghci> Set.filter odd $ Set.fromList [3,4,5,6,7,2,3,4]
fromList [3,5,7]
ghci> Set.map (+1) $ Set.fromList [3,4,5,6,7,2,3,4]
fromList [3,4,5,6,7,8]
```

在处理较大的List时，`setNub`要比`nub`快，但也可以从中看出，`nub`保留了List中元素的原有顺序，而`setNub`不。

构造自己的模块

我们已经见识过了几个很酷的模块，但怎样才能构造自己的模块呢？几乎所有的编程语言都允许你将代码分成多个文件，haskell也不例外。在编程时，将功能相近的函数和类型至于同一模块中会是个很好的习惯。这样一来，你就可以轻松地一个import来重用其中的函数.

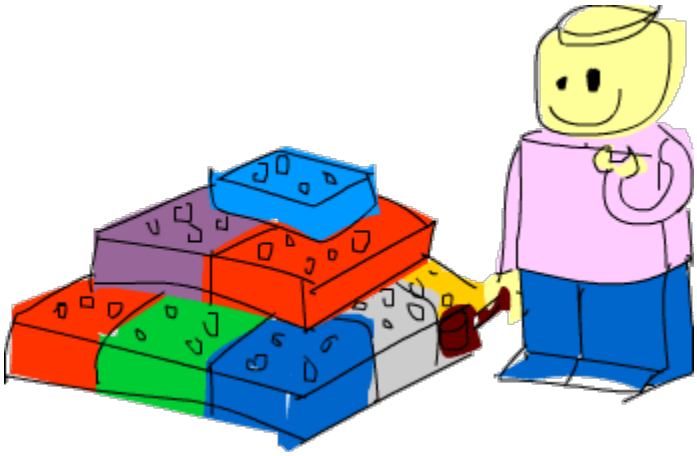
接下来我们将构造一个由计算机几何图形体积和面积组成的模块，先从新建一个`Geometry.hs`的文件开始.

在模块的开头定义模块的名称，如果文件名叫做`Geometry.hs`那它的名字就得是*Geometry*。在声明出它含有的函数名之后就可以编写函数的实现啦，就这样写:

```
module Geometry
( sphereVolume
 , sphereArea
 , cubeVolume
 , cubeArea
 , cuboidArea
 , cuboidVolume
) where
```

如你所见，我们提供了对球体,立方体和立方体的面积和体积的解法。继续进发，定义函数体:

```
module Geometry
( sphereVolume
 , sphereArea
 , cubeVolume
 , cubeArea
 , cuboidArea
 , cuboidVolume
) where

sphereVolume :: Float -> Float
sphereVolume radius = (4.0 / 3.0) * pi * (radius ^ 3)

sphereArea :: Float -> Float
sphereArea radius = 4 * pi * (radius ^ 2)

cubeVolume :: Float -> Float
cubeVolume side = cuboidVolume side side side

cubeArea :: Float -> Float
cubeArea side = cuboidArea side side side

cuboidVolume :: Float -> Float -> Float -> Float
cuboidVolume a b c = rectangleArea a b * c

cuboidArea :: Float -> Float -> Float -> Float
cuboidArea a b c = rectangleArea a b * 2 + rectangleArea a c * 2 + rectangleArea c b * 2

rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```



标准的几何公式。有几个地方需要注意一下，由于立方体只是长方体的特殊形式，所以在求它面积和体积的时候我们就将它当作是边长相等的长方体。在这里还定义了一个helper函数，`rectangleArea` 它可以通过长方体的两条边计算出长方体的面积。它仅仅是简单的相乘而已，分量不大。但请注意我们可以在这一模块中调用这个函数，而它不会被导出! 因为我们这个模块只与三维图形打交道.

当构造一个模块的时 候，我们通常只会导出那些行为相近的函数，而其内部的实现则是隐蔽的。如果有人用到了Geometry模块，就不需要关心它的内部实现是如何。我们作为编写者，完全可以随意修改这些函数甚至将其删掉，没有人会注意到里面的变动，因为我们并不把它们导出.

要使用我们的模块，只需:

```
import Geometry
```

将Geometry.hs文件至于用到它的程序文件的同一目录之下.

模块也可以按照分层的结构来组织，每个模块都可以含有多个子模块。而子模块还可以有自己的子模块。我们可以把Geometry分成三个子模块，而一个模块对应各自的图形对象.

首先，建立一个Geometry文件夹，注意首字母要大写，在里面新建三个文件

如下就是各个文件的内容:

sphere.hs

```
module Geometry.Sphere
( volume
 , area
) where

volume :: Float -> Float
volume radius = (4.0 / 3.0) * pi * (radius ^ 3)

area :: Float -> Float
area radius = 4 * pi * (radius ^ 2)
```

cuboid.hs

```
module Geometry.Cuboid
( volume
 , area
) where

volume :: Float -> Float -> Float -> Float
volume a b c = rectangleArea a b * c

area :: Float -> Float -> Float -> Float
area a b c = rectangleArea a b * 2 + rectangleArea a c * 2 + rectangleArea c b * 2

rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

cube.hs

```
module Geometry.Cube
( volume
 , area
) where

import qualified Geometry.Cuboid as Cuboid

volume :: Float -> Float
volume side = Cuboid.volume side side side

area :: Float -> Float
area side = Cuboid.area side side side
```

好的! 先是Geometry.Sphere。注意，我们将它置于Geometry文件夹之中并将它的名字定为Geometry.Sphere。对Cuboid也是同样，也注意下，在三个模块中我们定义了许多名称相同的函数，因为所在模块不同，所以不会产生命名冲突。若要在Geometry.Cube使用Geometry.Cuboid中的函数，就不能直接import Geometry.Cuboid，而必须得qualified import。因为它们中间的函数名完全相同.

```
import Geometry.Sphere
```

然后，调用 area 和 volume ，就可以得到球体的面积和体积，而若要用到两个或更多此类模块，就必须得 qualified import 来避免重名。所以就得这样写:

```
import qualified Geometry.Sphere as Sphere
import qualified Geometry.Cuboid as Cuboid
import qualified Geometry.Cube as Cube
```

然后就可以调用 Sphere.area ， Sphere.volume ， Cuboid.area 了，而每个函数都只计算其对应物体的面积和体积.

以后你若发现自己的代码体积庞大且函数众多，就应该试着找找目的相近的函数能否装入各自的模块，也方便日后的重用.

第八章 构造我们自己的类型和类型类

数据类型入门

在前面的章节中，我们谈了一些Haskell内置的类型和类型类。而在本章，我们将学习构造类型和类型类的方法。

我们以已经见识了许多数据类型，如 `Bool` 、 `Int` 、 `Char` 、 `Maybe` 等等，不过该怎样构造自己的数据类型呢？好问题，使用data关键字是一种方法。我们看看 `Bool` 在标准库中的定义：

```
data Bool = False | True
```

data表示我们要定义一个新的数据类型。=的左端标明类型的名称即 `Bool` ， `=` 的右端就是**值构造子**（*Value Constructor*），它们明确了该类型可能的值。 `|` 读作“或”，所以可以这样阅读该声明： `Bool` 类型的值可以是True或False。类型名和值构造子的首字母必大写。

相似，我们可以假想 `Int` 类型的声明：

```
data Int = -2147483648 | -2147483647 | ... | -1 | 0 | 1 | 2 | ... | 2147483647
```

首位两个值构造子分别表示了Int类型可能的最小值和最大值，这些省略号表示我们省去了中间大段的数字。当然，真实的声明不是这个样子的，这样写只是为了便于理解。

我们想想Haskell中图形的表示方法。表示圆可以用一个元组，如 `(43.1,55.0,10.4)` ，前两项表示圆心的位置，末项表示半径。听着不错，不过三维向量或其它什么东西也可能是这种形式！更好的方法就是自己构

造一个表示图形的类型。假定图形可以是圆（Circle）或长方形（Rectangle）：

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
```

这是啥，想想？`Circle`的值构造子有三个项，都是Float。可见我们在定义值构造子时，可以在后面跟几个类型表示它包含值的类型。在这里，前两项表示圆心的坐标，尾项表示半径。`Rectangle`的值构造子取四个Float项，前两项表示其左上角的坐标，后两项表示右下角的坐标。

谈到“项”(field)，其实应为“参数”(parameters)。值构造子的本质是个函数，可以返回一个类型的值。我们看下这两个值构造子的类型声明：

```
ghci> :t Circle
Circle :: Float -> Float -> Float -> Shape
ghci> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
```

Cool，这么说值构造子就跟普通函数并无二致咯，谁想得到？我们写个函数计算图形面积：

```
surface :: Shape -> Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

值得一提的是，它的类型声明表示了该函数取一个Shape值并返回一个Float值。写`Circle -> Float`是不可以的，因为Circle并非类型，真正的类型应该是Shape。这与不能写`True->False`的道理是一样的。再就是，我们使用的模式匹配针对的都是值构造子。之前我们匹配过`[]`、`False`或`5`，它们都是不包含参数的值构造子。

我们只关心圆的半径，因此不需理会表示坐标的前两项：

```
ghci> surface $ Circle 10 20 10
314.15927
ghci> surface $ Rectangle 0 0 100 100
10000.0
```

Yay，it works！不过我们若尝试输出`Circle 10 20`到控制台，就会得到一个错误。这是因为Haskell还不知道该类型的字符串表示方法。想想，当我们往控制台输出值的时候，Haskell会先调用show函数得到这个值的字符串表示才会输出。因此要让我们的Shape类型成为Show类型类的成员。可以这样修改：

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float deriving (Show)
```

先不去深究**deriving**（派生），可以先这样理解：若在data声明的后面加上`deriving (Show)`，那Haskell就会自动将该类型至于Show类型类之中。好了，由于值构造子是个函数，因此我们可以拿它交给map，拿它不全调用，以及普通函数能做的一切。

```
ghci> Circle 10 20 5
Circle 10.0 20.0 5.0
ghci> Rectangle 50 230 60 90
Rectangle 50.0 230.0 60.0 90.0
```

我们若要取一组不同半径的同心圆，可以这样：

```
ghci> map (Circle 10 20) [4,5,6,6]
[Circle 10.0 20.0 4.0,Circle 10.0 20.0 5.0,Circle 10.0 20.0 6.0,Circle 10.0 20.0 6.0]
```

我们的类型还可以更好。增加加一个表示二维空间中点的类型，可以让我们的Shape更加容易理解：

```
data Point = Point Float Float deriving (Show)
data Shape = Circle Point Float | Rectangle Point Point deriving (Show)
```

注意下Point的定义，它的类型与值构造子用了相同的名字。没啥特殊含义，实际上，在一个类型含有唯一值构造子时这种重名是很常见的。好的，如今我们的Circle含有两个项，一个是Point类型，一个是Float类型，好作区分。Rectangle也是同样，我们得修改surface函数以适应类型定义的变动。

```
surface :: Shape -> Float
surface (Circle _ r) = pi * r ^ 2
surface (Rectangle (Point x1 y1) (Point x2 y2)) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

唯一需要修改的地方就是模式。在Circle的模式中，我们无视了整个Point。而在Rectangle的模式中，我们用了一个嵌套的模式来取得Point中的项。若出于某原因而需要整个Point，那么直接匹配就是了。

```
ghci> surface (Rectangle (Point 0 0) (Point 100 100))
10000.0
ghci> surface (Circle (Point 0 0) 24)
1809.5574
```

表示移动一个图形的函数该怎么写？ 它应当取一个Shape和表示位移的两个数，返回一个位于新位置的图形。

```
nudge :: Shape -> Float -> Float -> Shape
nudge (Circle (Point x y) r) a b = Circle (Point (x+a) (y+b)) r
nudge (Rectangle (Point x1 y1) (Point x2 y2)) a b = Rectangle (Point (x1+a) (y1+b)) (Point (x2+a) (y2+
b))
```

很直白，我们给这一Shape的点加上位移的量。

```
ghci> nudge (Circle (Point 34 34) 10) 5 10
Circle (Point 39.0 44.0) 10.0
```

如果不想直接处理Point，我们可以搞个辅助函数(auxilliary function)，初始从原点创建图形，再移动它们。

```
baseCircle :: Float -> Shape
baseCircle r = Circle (Point 0 0) r

baseRect :: Float -> Float -> Shape
baseRect width height = Rectangle (Point 0 0) (Point width height)
```

```
ghci> nudge (baseRect 40 100) 60 23
Rectangle (Point 60.0 23.0) (Point 100.0 123.0)
```

毫无疑问，你可以把你的数据类型导出到模块中。只要把你的类型与要导出的函数写到一起就是了。再在后面跟个括号，列出要导出的值构造子，用逗号隔开。如要导出所有的值构造子，那就写个..。

若要将这里定义的所有函数和类型都导出到一个模块中，可以这样：

```
module Shapes
( Point(..)
, Shape(..)
, surface
, nudge
, baseCircle
, baseRect
) where
```

一个Shape (..)，我们就导出了 Shape 的所有值构造子。这一来无论谁导入我们的模块，都可以用 Rectangle 和 Circle 值构造子来构造Shape了。这与写 Shape(Rectangle,Circle) 等价。

我们可以选择不导出任何Shape的值构造子，这一来使用我们模块的人就只能用辅助函数 baseCircle 和 baseRect 来得到 Shape 了。 Data.Map 就是这一套，没有 Map.Map [(1,2),(3,4)] ，因为它没有导出任何一个值构造子。但你可以用，像 Map.fromList 这样的辅助函数得到map。应该记住，值构造子只是函数而已，如果不导出它们，就拒绝了使用我们模块的人调用它们。但可以使用其他返回该类型的函数，来取得这一类型的值。

不导出数据类型的值构造子隐藏了他们的内部实现，令类型的抽象度更高。同时，我们模块的使用者也就无法使用该值构造子进行模式匹配了。

Record Syntax

OK，我们需要一个数据类型来描述一个人，得包含他的姓、名、年龄、身高、体重、电话号码以及最爱的冰激淋。我不知你的想法，不过我觉得要了解一个人，这些资料就够了。就这样，实现出来！

```
data Person = Person String String Int Float String String deriving (Show)
```

O~Kay，第一项是名，第二项是姓，第三项是年龄，等等。我们造一个人：

```
ghci> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
ghci> guy
Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

貌似很酷，就是难读了点儿。弄个函数得人的某项资料又该如何？如姓的函数，名的函数，等等。好吧，我们只能这样：

```
firstName :: Person -> String
firstName (Person firstname _ _ _ _) = firstname

lastName :: Person -> String
lastName (Person _ lastname _ _ _) = lastname

age :: Person -> Int
age (Person _ _ age _ _ _) = age

height :: Person -> Float
height (Person _ _ _ height _ _) = height

phoneNumber :: Person -> String
phoneNumber (Person _ _ _ _ number _) = number

flavor :: Person -> String
flavor (Person _ _ _ _ _ flavor) = flavor
```

唔，我可不愿写这样的代码！虽然it works，但也太无聊了哇。

```
ghci> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
ghci> firstName guy
"Buddy"
ghci> height guy
184.2
ghci> flavor guy
"Chocolate"
```

你可能会说，一定有更好的方法！呃，抱歉，没有。

开个玩笑，其实有的，哈哈哈~Haskell的发明者都是天才，早就料到了此类情形。他们引入了一个特殊的类型，也就是刚才提到的更好的方法--**Record Syntax**。

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     , flavor :: String
                     } deriving (Show)
```

与原先让那些项一个挨一个的空格隔开不同，这里用了花括号{}。先写出项的名字，如firstName，后跟两个冒号（也叫Raamayim Nekudotayim，哈哈~(译者不知道什么意思~囧)），标明其类型，返回的数据类型仍与以前相同。这样的好处就是，可以用函数从中直接按项取值。通过Record Syntax，haskell就自动生成了这些函数：`firstName`，`lastName`，`age`，`height`，`phoneNumber`和`flavor`。

```
ghci> :t flavor
flavor :: Person -> String
ghci> :t firstName
firstName :: Person -> String
```

还有个好处，就是若派生(deriving)到`Show`类型类，它的显示是不同的。假如我们有个类型表示一辆车，要包含生产商、型号以及出场年份：

```
data Car = Car String String Int deriving (Show)
```

```
ghci> Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967
```

若用Record Syntax，就可以得到像这样的新车：

```
data Car = Car {company :: String, model :: String, year :: Int} deriving (Show)
```

```
ghci> Car {company="Ford", model="Mustang", year=1967}
Car {company = "Ford", model = "Mustang", year = 1967}
```

这一来在造车时我们就不必关心各项的顺序了。

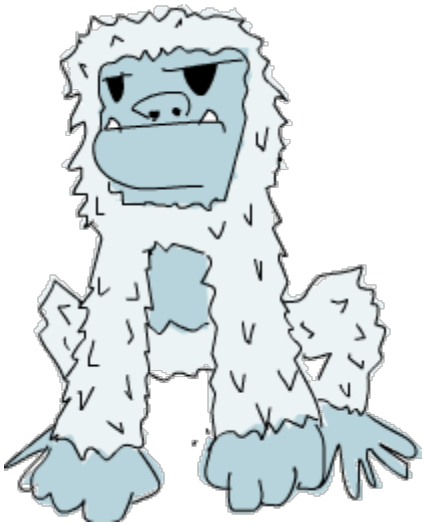
表示三维向量之类简单数据，`Vector = Vector Int Int Int`就足够明白了。但一个值构造子中若含有很多个项且不易区分，如一个人或者一辆车啥的，就应该使用Record Syntax。

类型参数

值构造子可以取几个参数产生一个新值，如Car的构造子是取三个参数返回一个Car。与之相似，类型构造子可以取类型作参数，产生新的类型。这咋一听貌似有点深奥，不过实际上并不复杂。如果你对C++的模

板有了解，就会看到很多相似的地方。我们看一个熟悉的类型，好对类型参数有个大致印象：

```
data Maybe a = Nothing | Just a
```



这里的a就是个类型参数。也正因为有了它，Maybe就成为了一个类型构造子。在它的值不是Nothing时，它的类型构造子可以搞出Maybe Int，Maybe String等等诸多类型。但只一个Maybe是不行的，因为它不是类型，而是类型构造子。要成为真正的类型，必须得把它需要的类型参数全部填满。

所以，如果拿 `Char` 作参数交给 `Maybe` ，就可以得到一个 `Maybe Char` 的类型。如， `Just 'a'` 的类型就是 `Maybe Char` 。

你可能并未察觉，在遇见Maybe之前我们早就接触到类型参数了。它便是List类型。这里面有点语法糖，List类型实际上就是取一个参数来生成一个特定类型，这类型可以是Int，Char也可以是String，但不会跟在[]的后面。

把玩一下 `Maybe` ！

```
ghci> Just "Haha"
Just "Haha"
ghci> Just 84
Just 84
ghci> :t Just "Haha"
Just "Haha" :: Maybe [Char]
ghci> :t Just 84
Just 84 :: (Num t) => Maybe t
ghci> :t Nothing
Nothing :: Maybe a
ghci> Just 10 :: Maybe Double
Just 10.0
```

类型参数很实用。有了它，我们就可以按照我们的需要构造出不同的类型。若执行 `:t Just "Haha"` ，类型推导引擎就会认出它是个 `Maybe [Char]` ，由于 `Just a` 里的 `a` 是个字符串，那么 `Maybe a` 里的 `a` 一定也是个字符串。

注意下，`Nothing`的类型为`Maybe a`。它是多态的，若有函数取`Maybe Int`类型的参数，就一概可以传给它一个Nothing，因为Nothing中不包含任何值。`Maybe a`类型可以有`Maybe Int`的行为，正如`5`可以是`Int`也可以是`Double`。与之相似，空List的类型是`[a]`，可以与一切List打交道。因此，我们可以`[1,2,3]++[]`，也可以`["ha","ha,","ha"]++[]`。

类型参数有很多好处，但前提是用对了地方才行。一般都是不关心类型里面的内容，如Maybe a。一个类型的行为若有点像是容器，那么使用类型参数会是个不错的选择。我们完全可以把我们的Car类型从

```
data Car = Car { company :: String
               , model :: String
               , year :: Int
               } deriving (Show)
```

改成：

```
data Car a b c = Car { company :: a
                     , model :: b
                     , year :: c
                      } deriving (Show)
```

但是，这样我们又得到了什么好处？回答很可能是，一无所得。因为我们只定义了处理`Car String String Int`类型的函数，像以前，我们还可以弄个简单函数来描述车的属性。

```
tellCar :: Car -> String
tellCar (Car {company = c, model = m, year = y}) = "This " ++ c ++ " " ++ m ++ " was made in " ++ s
how y
```

```
ghci> let stang = Car {company="Ford", model="Mustang", year=1967}
ghci> tellCar stang  "This Ford Mustang was made in 1967"
```

可爱的小函数！它的类型声明得很漂亮，而且工作良好。好，如果改成Car a b c又会怎样？

```
tellCar :: (Show a) => Car String String a -> String
tellCar (Car {company = c, model = m, year = y}) = "This " ++ c ++ " " ++ m ++ " was made in " ++ s
how y
```

我们只能强制性地给这个函数安一个(Show a) => Car String String a 的类型约束。看得出来，这要繁复得多。而唯一的好处貌似就是，我们可以使用Show类型类的实例来作a的类型。

```
ghci> tellCar (Car "Ford" "Mustang" 1967)
"This Ford Mustang was made in 1967"
ghci> tellCar (Car "Ford" "Mustang" "nineteen sixty seven")
"This Ford Mustang was made in \"nineteen sixty seven\""
ghci> :t Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967 :: (Num t) => Car [Char] [Char] t
ghci> :t Car "Ford" "Mustang" "nineteen sixty seven"
Car "Ford" "Mustang" "nineteen sixty seven" :: Car [Char] [Char] [Char]
```

其实在现实生活中，使用Car String String Int在大多数情况下已经满够了。所以给Car类型加类型参数貌似并没有什么必要。通常我们都是都是在一个类型中包含的类型并不影响它的行为时才引入类型参数。一组什么东西组成的List就是一个List，它不关心里面东西的类型是啥，然而总是工作良好。若取一组数字的和，我们可以在后面的函数体中明确是一组数字的List。Maybe与之相似，它表示可以有什么东西可以没有，而不必关心这东西是啥。

我们之前还遇见过一个类型参数的应用，就是Data.Map中的Map k v。k表示Map中键的类型，v表示值的类型。这是个好例子，map中类型参数的使用允许我们能够用一个类型索引另一个类型，只要键的类型在Ord类型类就行。如果叫我们自己定义一个map类型，可以在data声明中加上一个类型类的约束。

```
data (Ord k) => Map k v = ...
```

然而haskell中有一个严格的约定，那就是永远不要在data声明中添加类型约束。为啥？嗯，因为这样没好处，反而得写更多不必要的类型约束。Map k v要是有Ord k的约束，那就相当于假定每个map的相关函数都认为k是可排序的。若不给数据类型加约束，我们就不必给那些不关心键是否可排序的函数另加约束了。这类函数的一个例子就是toList，它只是把一个map转换为关联List罢了，类型声明为 `toList :: Map k v -> [(k, v)]` 。要是加上类型约束，就只能是 `toList :: (Ord k) =>Map k a -> [(k,v)]` ，明显没必要嘛。

所以说，永远不要在data声明中加类型约束---即便看起来没问题。免得在函数声明中写出过多无畏的类型约束。

我们实现个表示三维向量的类型，再给它加几个处理函数。我么那就给它个类型参数，虽然大多数情况都是数值型，不过这一来它就支持了多种数值类型。

```
data Vector a = Vector a a a deriving (Show)
vplus :: (Num t) => Vector t -> Vector t -> Vector t
(Vector i j k) `vplus` (Vector l m n) = Vector (i+l) (j+m) (k+n)
vectMult :: (Num t) => Vector t -> t -> Vector t
(Vector i j k) `vectMult` m = Vector (i*m) (j*m) (k*m)
scalarMult :: (Num t) => Vector t -> Vector t -> t
(Vector i j k) `scalarMult` (Vector l m n) = i*l + j*m + k*n
```

vplus用来相加两个向量，即将其所有对应的项相加。`scalarMult`用来求两个向量的标量积，`vectMult`求一个向量和一个标量的积。这些函数可以处理`Vector Int`，`Vector Integer`，`Vector Float`等等类型，只要`Vector a`里的这个`a`在Num类型类中就行。同样，如果你看下这些函数的类型声明就会发现，它们只能处理相同类型的向量，其中包含的数字类型必须与另一个向量一致。注意，我们并没有在data声明中添加Num的类约束。反正无论怎么着都是给函数加约束。

再度重申，类型构造子和值构造子的区分是相当重要的。在声明数据类型时，等号=左端的那个是类型构造子，右端的（中间可能有|分隔）都是值构造子。拿`Vector t t t -> Vector t t t -> t`作函数的类型就会产生一个错误，因为在类型声明中只能写类型，而Vector的类型构造子只有个参数，它的值构造子才是有三个。我们就慢慢耍：

```
ghci> Vector 3 5 8 `vplus` Vector 9 2 8
Vector 12 7 16
ghci> Vector 3 5 8 `vplus` Vector 9 2 8 `vplus` Vector 0 2 3
Vector 12 9 19
ghci> Vector 3 9 7 `vectMult` 10
Vector 30 90 70
ghci> Vector 4 9 5 `scalarMult` Vector 9.0 2.0 4.0
74.0
ghci> Vector 2 9 3 `vectMult` (Vector 4 9 5 `scalarMult` Vector 9 2 4)
Vector 148 666 222
```

派生实例

在typeclass 101那节里面，我们了解了typeclass的基础内容。里面提到，类型类就是定义了某些行为的接口。例如，Int类型是Eq类型类的一个实例，Eq类就定义了判定相等性的行为。Int值可以判断相等性，所以Int就是Eq类型类的成员。它的真正威力体现在作为Eq接口的函数中，即==和/=。只要一个类型是Eq类型类的成员，我们就可以使用==函数来处理这一类型。这便是为何`4==4`和`"foo"/="bar"`这样的表达式都需要作类型检查。

我们也曾提到，人们很容易把类型类与Java，python，C++等语言的类混淆。很多人对此都倍感不解，在原先那些语言中，类就像是蓝图，我们可以根据它来创造对象、保存状态并执行操作。而类型类更像是接口，我们不是靠它构造数据，而是给既有的数据类型描述行为。什么东西若可以判定相等性，我们就可以让它成为Eq类型类的实例。什么东西若可以比较大小，那就可以让它成为Ord类型类的实例。

在下一节，我们将看一下如何手工实现类型类中定义函数来构造实例。现在呢，我们先了解下Haskell是如何自动生成这几个类型类的实例，`Eq`，`Ord`，`Enum`，`Bounded`，`Show`，`Read`。只要我们在构造类型时在后面加个deriving（派生）关键字，Haskell就可以自动地给我们的类型加上这些行为。

看这个数据类型：

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     }
```

这描述了一个人。我们先假定世界上没有重名重姓又同龄的人存在，好，假如有两个record，有没有可能是描述同一个人呢？当然可能，我么可以判定姓名年龄的相等性，来判断它俩是否相等。这一来，让这个类型成为Eq的成员就很靠谱了。直接derive这个实例：

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving (Eq)
```

在一个类型派生为Eq的实例后，就可以直接使用==或/=来判断它们的相等性了。Haskell会先看下这两个值的值构造子是否一致（这里只是单值构造子），再用==来检查其中的所有数据（必须都是Eq的成员）是否一致。在这里只有String和Int，所以是没有问题的。测试下我们的Eq实例：

```
ghci> let mikeD = Person {firstName = "Michael", lastName = "Diamond", age = 43}
ghci> let adRock = Person {firstName = "Adam", lastName = "Horovitz", age = 41}
ghci> let mca = Person {firstName = "Adam", lastName = "Yauch", age = 44}
ghci> mca == adRock
False
ghci> mikeD == adRock
False
ghci> mikeD == mikeD
True
ghci> mikeD == Person {firstName = "Michael", lastName = "Diamond", age = 43}
True
```

自然，Person如今已经成为了Eq的成员，我们就可以将其应用于所有在类型声明中用到Eq类约束的函数了，如elem。

```
ghci> let beastieBoys = [mca, adRock, mikeD]
ghci> mikeD `elem` beastieBoys
True
```

Show和Read类型类处理可与字符串相互转换的东西。同Eq相似，如果一个类型的构造子含有参数，那所有参数的类型必须都得属于Show或Read才能让该类型成为其实例。就让我们的Person也成为Read和Show的一员吧。

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving (Eq, Show, Read)
```

然后就可以输出一个Person到控制台了。

```
ghci> let mikeD = Person {firstName = "Michael", lastName = "Diamond", age = 43}
ghci> mikeD
Person {firstName = "Michael", lastName = "Diamond", age = 43}
ghci> "mikeD is: " ++ show mikeD
"mikeD is: Person {firstName = \"Michael\", lastName = \"Diamond\", age = 43}"
```

如果我们还没让Person类型作为Show的成员就尝试输出它，haskell就会向我们抱怨，说它不知道该怎么把它表示成一个字符串。不过现在既然已经派生成为了Show的一个实例，它就知道了。

Read几乎就是与Show相对的类型类，show是将一个值转换成字符串，而read则是将一个字符串转成某类型的值。还记得，使用read函数时我们必须得用类型注释注明想要的类型，否则haskell就不会知道如何转换。

```
ghci> read "Person {firstName =\"Michael\", lastName =\"Diamond\", age = 43}" :: Person
Person {firstName = "Michael", lastName = "Diamond", age = 43}
```

如果我们read的结果会在后面用到参与计算，Haskell就可以推导出是一个Person的行为，不加注释也是可以的。

```
ghci> read "Person {firstName =\"Michael\", lastName =\"Diamond\", age = 43}" == mikeD
True
```

也可以read带参数的类型，但必须填满所有的参数。因此 `read "Just 't'" :: Maybe a` 是不可以的，`read "Just 't'" :: Maybe Char` 才对。

很容易想象Ord类派生实例的行为。首先，判断两个值构造子是否一致，如果是，再判断它们的参数，前提是它们的参数都得是Ord的实例。Bool类型可以有两种值，False和True。为了了解在比较中程序的行为，我们可以这样想象：

```
data Bool = False | True deriving (Ord)
```

由于值构造子False安排在True的前面，我们可以认为True比False大。

```
ghci> True `compare` False
GT
ghci> True > False
True
ghci> True
False
```

在Maybe a数据类型中，值构造子Nothing在Just值构造子前面，所以一个 `Nothing` 总要比 `Just something` 的值小。即便这个 `something` 是 `100000000` 也是如此。

```
ghci> Nothing
True
ghci> Nothing > Just (-49999)
False
ghci> Just 3 `compare` Just 2
GT
ghci> Just 100 > Just 50
True
```

不过类似Just (**3), Just(**2)之类的代码是不可以的。因为(**3)和(**2)都是函数，而函数不是Ord类的成员。

作枚举，使用数字类型就能轻易做到。不过使用Enmu和Bounded类型类会更好，看下这个类型：

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

所有的值构造子都是nullary的（也就是没有参数），每个东西都有前置子和后继子，我们可以让它成为Enmu类型类的成员。同样，每个东西都有可能的最小值和最大值，我们也可以让它成为Bounded类型类

的成员。在这里，我们就同时将它搞成其它可派生类型类的实例。再看看我们能拿它做啥：

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
         deriving (Eq, Ord, Show, Read, Bounded, Enum)
```

由于它是Show和Read类型类的成员，我们可以将这个类型的值与字符串相互转换。

```
ghci> Wednesday
Wednesday
ghci> show Wednesday
"Wednesday"
ghci> read "Saturday" :: Day
Saturday
```

由于它是Eq与Ord的成员，因此我们可以拿Day作比较。

```
ghci> Saturday == Sunday
False
ghci> Saturday == Saturday
True
ghci> Saturday > Friday
True
ghci> Monday `compare` Wednesday
LT
```

它也是Bounded的成员，因此有最早和最晚的一天。

```
ghci> minBound :: Day
Monday
ghci> maxBound :: Day
Sunday
```

它也是Enmu的实例，可以得到前一天和后一天，并且可以对此使用List的区间。

```
ghci> succ Monday
Tuesday
ghci> pred Saturday
Friday
ghci> [Thursday .. Sunday]
[Thursday,Friday,Saturday,Sunday]
ghci> [minBound .. maxBound] :: [Day]
[Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday]
```

那是相当的棒。

类型别名

在前面我们提到在写类型名的时候，`[Char]`和`String`等价，可以互换。这就是由类型别名实现的。类型别名实际上什么也没做，只是给类型提供了不同的名字，让我们的代码更容易理解。这就是`[Char]`的别名`String`的由来。

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

我们已经介绍过了type关键字，这个关键字有一定误导性，它并不是用来创造新类（这是data关键字做的事情），而是给一个既有类型提供一个别名。

如果我们随便搞个函数toUpperString或其他什么名字，将一个字符串变成大写，可以用这样的类型声明`toUpperString :: [Char] -> [Char]`，也可以这样`toUpperString :: String -> String`，二者在本质上是完全相同的。后者要更易读些。

在前面Data.Map那部分，我们用了一个关联List来表示`phoneBook`，之后才改成的Map。我们已经发现了，一个关联List就是一组键值对组成的List。再看下我们phoneBook的样子：

```
phoneBook :: [(String,String)]
phoneBook =
    [("betty","555-2938")
    ,("bonnie","452-2928")
    ,("patsy","493-2928")
    ,("lucille","205-2928")
    ,("wendy","939-8282")
    ,("penny","853-2492")
    ]
```

可以看出，phoneBook的类型就是`[(String,String)]`，这表示一个关联List仅是String到String的映射关系。我们就弄个类型别名，好让它类型声明中能够表达更多信息。

```
type PhoneBook = [(String,String)]
```

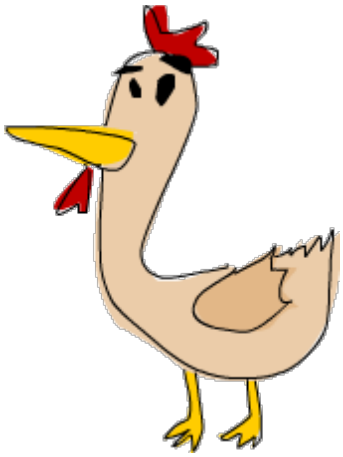
现在我们phoneBook的类型声明就可以是phoneBook :: PhoneBook了。再给字符串加上别名：

```
type PhoneNumber = String
type Name = String
type PhoneBook = [(Name,PhoneNumber)]
```

Haskell程序员给String加别名是为了让函数中字符串的表达方式及用途更加明确。

好的，我们实现了一个函数，它可以取一名字和号码检查它是否存在于电话本。现在可以给它加一个相当好看明了的类型声明：

```
inPhoneBook :: Name -> PhoneNumber -> PhoneBook -> Bool
inPhoneBook name pnumber pbook = (name,pnumber) `elem` pbook
```



如果不用类型别名，我们函数的类型声明就只能是 `String -> String -> [(String ,String)] -> Bool` 了。在这里使用类型别名是为了让类型声明更加易读，但你也不必拘泥于它。引入类型别名的动机既非单纯表示我们函数中的既有类型，也不是为了替换掉那些重复率高的长名字类型(如 `[(String,String)]`)，而是为了让类型对事物的描述更加明确。

类型别名也是可以有参数的，如果你想搞个类型来表示关联List，但依然要它保持通用，好让它可以使用任意类型作key和value，我们可以这样：

```
type AssocList k v = [(k,v)]
```

好的，现在一个从关联List中按键索值的函数类型可以定义为 `(Eq k) => k -> AssocList k v -> Maybe v. AssocList i` 。AssocList是个取两个类型做参数生成一个具体类型的类型构造子，如 `Assoc Int String` 等等。

> **Fronzie说**：Hey！当我提到具体类型，那我就是说它是完全调用的，就像 `Map Int String` 。要不就是多态函数中的 `[a]` 或 `(Ord a) => Maybe a` 之类。有时我和孩子们会说“Maybe类型”，但我们的意思并不是按字面来，傻瓜都知道Maybe是类型构造子嘛。只要用一个明确的类型调用Maybe，如 `Maybe String` 可得一个具体类型。你知道，只有具体类型才可以储存值。

我们可以用不全调用来得到新的函数，同样也可以使用不全调用得到新的类型构造子。同函数一样，用不全的类型参数调用类型构造子就可以得到一个不全调用的类型构造子，如果我们要一个表示从整数到某东西间映射关系的类型，我们可以这样：

```
type IntMap v = Map Int v
```

也可以这样：

```
type IntMap = Map Int
```

无论怎样，IntMap的类型构造子都是取一个参数，而它就是这整数指向的类型。

Oh yeah，如果要你去实现它，很可能会用个qualified import来导入Data.Map。这时，类型构造子前面

必须得加上模块名。所以应该写个 `type IntMap = Map.Map Int`

你得保证真正弄明白了类型构造子和值构造子的区别。我们有了个叫IntMap或者AssocList的别名并不意味着我们可以执行类似 AssocList [(1,2),(4,5),(7,9)] 的代码，而是可以用不同的名字来表示原先的List，就像 [(1,2),(4,5),(7,9)] :: AssocList Int Int 让它里面的类型都是Int。而像处理普通的二元组构成的那种List处理它也是可以的。类型别名（类型依然不变），只可以在Haskell的类型部分中使用，像定义新类型或类型声明或类型注释中跟在::后面的部分。

另一个很酷的二参类型就是 `Either a b` 了，它大约是这样定义的：

```
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)
```

它有两个值构造子。如果用了Left，那它内容的类型就是a；用了Right，那它内容的类型就是b。我们可以用它来将可能是两种类型的值封装起来，从里面取值时就同时提供Left和Right的模式匹配。

```
ghci> Right 20
Right 20
ghci> Left "w00t"
Left "w00t"
ghci> :t Right 'a'
Right 'a' :: Either a Char
ghci> :t Left True
Left True :: Either Bool b
```

到现在为止，Maybe是最常见的表示可能失败的计算的类型了。但有时Maybe也并不是十分的好用，因为Nothing中包含的信息还是太少。要是我们不关心函数失败的原因，它还是不错的。就像Data.Map的lookup只有在搜寻的项不在map时才会失败，对此我们一清二楚。但我们若想知道函数失败的原因，那还得使用Either a b，用a来表示可能的错误的类型，用b来表示一个成功运算的类型。从现在开始，错误一律用Left值构造子，而结果一律用Right。

一个例子：有个学校提供了不少壁橱，好给学生们地方放他们的Gun'N'Rose海报。每个壁橱都有个密码，哪个学生想用个壁橱，就告诉管理员壁橱的号码，管理员就会告诉他壁橱的密码。但如果这个壁橱已经让别人用了，管理员就不能告诉他密码了，得换一个壁橱。我们就用Data.Map的一个map来表示这些壁橱，把一个号码映射到一个表示壁橱占用情况及密码的二元组里。

```
import qualified Data.Map as Map

data LockerState = Taken | Free deriving (Show, Eq)

type Code = String

type LockerMap = Map.Map Int (LockerState, Code)
```

很简单，我们引入了一个新的类型来表示壁橱的占用情况。并为壁橱密码及按号码找壁橱的map分别设置了一个别名。好，现在我们实现这个按号码找壁橱的函数，就用 Either String Code 类型表示我们的结

果，因为lookup可能会以两种原因失败。厨子已经让别人用了或者压根就没有这个橱子。如果lookup失败，就用字符串表明失败的原因。

```
lockerLookup :: Int -> LockerMap -> Either String Code
lockerLookup lockerNumber map =
    case Map.lookup lockerNumber map of
        Nothing -> Left $ "Locker number " ++ show lockerNumber ++ " doesn't exist!"
        Just (state, code) -> if state /= Taken
                              then Right code
                              else Left $ "Locker " ++ show lockerNumber ++ " is already taken!"
```

我们在这里个map中执行一次普通的 `lookup` ，如果得到一个 `Nothing` ，就返回一个 `Left String` 的值，告诉他压根就没这个号码的橱子。如果找到了，就再检查下，看这橱子是不是已经让别人用了，如果是，就返回个 `Left String` 说它已经让别人用了。否则就返回个Right Code的值，通过它来告诉学生壁橱的密码。它实际上就是个 `Right String` ，我们引入了个类型别名让它这类型声明更好看。

如下是个map的例子：

```
lockers :: LockerMap
lockers = Map.fromList
    [(100,(Taken,"ZD39I"))
    ,(101,(Free,"JAH3I"))
    ,(103,(Free,"IQSA9"))
    ,(105,(Free,"QOTSA"))
    ,(109,(Taken,"893JJ"))
    ,(110,(Taken,"99292"))
    ]
```

现在从里面lookup某个橱子号..

```
ghci> lockerLookup 101 lockers
Right "JAH3I"
ghci> lockerLookup 100 lockers
Left "Locker 100 is already taken!"
ghci> lockerLookup 102 lockers
Left "Locker number 102 doesn't exist!"
ghci> lockerLookup 110 lockers
Left "Locker 110 is already taken!"
ghci> lockerLookup 105 lockers
Right "QOTSA"
```

我们完全可以用 `Maybe a` 来表示它的结果，但这样一来我们就对得不到密码的原因不得而知了。而在这里，我们的新类型可以告诉我们失败的原因。