计算概论A—实验班

函数式程序设计

Functional Programming

胡振江，张 伟

北京大学 信息科学技术学院 计算机科学技术系

2021年09～2021年12月

初见Haskell

使用 Haskell 语言定义函数

在这一节中，我们采用 Haskell 语言，对上一章中给出的若干函数示例进行重定义，并结合这些重定义对 Haskell 语言的相关细节进行说明.

逻辑运算函数

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

❖函数名为什么是 not'，而不是 not

- Prelude 模块中已经存在了not
- 为了避免歧义，用 not’

❖如果坚持使用 not，有什么问题?

- 当在一个模块中加载了两个模块，且这两个模块中存在同名的函数时，为了避免歧义，在使用这个同名函数时，需要加上 模块名 和 点 作为前缀

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

- `not` 与其类型 `Bool -> Bool` 之间的分隔字符是两个连续冒号 `::`. 为什么用两个连续冒号，用一个冒号不是更简洁吗? 原因很简单：在 Haskell 程序中，一个冒号另有他用. 具体用途，请耐心等待一段时间.

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

- 函数类型中表示两个集合之间映射关系的符号是两个连续字符 `->`．为什么不用符号 $\to$ 呢，这个符号不是更形象吗？这个问题说来话长．话说在很久很久以前，计算机上只能表示 128 个字符[3]．不幸的是，字符 $\to$ 不在其中．但是，那个时候程序员已经在辛苦地编写程序了．由此，也导致了一个潜规则的形成：大部分的程序语言所采用的关键词，都会选择由最原始的那 128 个字符中的一个或若干个构成．
- 在 Haskell 程序中，布尔类型（也即：由两个布尔值构成的集合）被命名为 `Bool`．为什么不用符号 $\mathbb{B}$ 呢，这个符号不是更简洁吗？关于此类问题的答案，请自行推理．

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

```
not' True  = False
```

- 这条语句中出现了两个单词：`True` 和 `False`．其中，前者指代布尔值集合 `Bool` 中的*真*值元素；后者指代布尔值集合 `Bool` 中的*假*值元素．也即：`Bool` ≐ {`True, False`}
- `not True` 的含义是将函数 `not` 作用在值 `True` 上．用程序员的语言，即：用 `True` 这个实际参数去调用函数 `not`．这里，你可以看到 Haskell 语言的一个重要设计决策：调用函数时，不需要把参数放在一对圆括号中．这一点与大多数主流的程序设计语言是不同的．为什么要采用这样的设计决策呢？你可以回顾一下在上一章中引入的 *qsort* 函数，其中层层叠叠的圆括号嵌套让程序的易理解性变得非常的差．把这些圆括号去掉，会让程序变得更易理解吗？谁知道呢，走着瞧呗！
- 整个语句 `not True = False` 的含义是：将函数调用 `not True` 的返回值定义为 `False`．这种定义函数的方式在 Haskell 中被称为*模式匹配*（Pattern Matching）．在这条语句中，模式匹配具体化为：如果对 `not` 函数的一次调用被匹配为 `not True`，则这次函数调用的返回值被定义为 `False`[4]．

```
not'' :: Bool -> Bool
not'' x = if x == True then False else True
```

conditional expression
分支表达式

- `not x` 中的 `x` 在此处表示函数 not 的一个形式参数（parameter）. 顾名思义，所谓形式参数，就是一种形式上的参数. 当真正发生函数调用的时候，形式参数会被实际传入的参数（argument）所替换.

- `not x =` 的右侧是一个分支表达式（conditional expression）. 其中出现的三个单词 `if`、`then` 和 `else` 是 Haskell 语言的三个关键词. 所谓关键词，就是程序设计语言中具有特殊含义的单词；这些单词通常不能被另做他用. 这个分支表达式所表达的含义为：如果 `x` 等于 `True`，则这个表达式返回 `False`；否则，返回 `True`.

```
not'' :: Bool -> Bool
not'' x = if x == True then False else True
```

- 在 Haskell 语言中，一个等号 = 和两个连续等号 == 具有不同的含义．前者表示“定义为”，即：把等号左侧表达式的值定义为等号右侧表达式的值．后者是一个逻辑运算操作符．

guarded equations
条件方程组

```
not''' :: Bool -> Bool
not''' x | x == True  = False
         | x == False = True
```

```
not'''' :: Bool -> Bool
not'''' x | x         = False
          | otherwise = True
```

```
and' :: Bool -> Bool -> Bool
and' True  True  = True
and' True  False = False
and' False True  = False
and' False False = False
```

- 函数调用/应用时的圆括号没有了；这一点我们刚刚讲过.
- 函数类型声明中的圆括号也没有了. Haskell 语言规定, 在函数类型声明中, 操作符 `->` 具有*右*结合性. 因此, `Bool -> Bool -> Bool` 等价于 `Bool -> (Bool -> Bool)`.

这种定义具有明显的不简洁性. 我们可以利用*模式匹配*提供的*通配符*机制，让这个定义变得更加简洁. 请看如下定义:

```haskell
and'' :: Bool -> Bool -> Bool
and'' True True = True
and'' _    _    = False
```

作业 01

关于逻辑与函数，你还能想到其他定义方式吗？请用 Haskell 语言写出至少三种其他定义方式.

```
and''' :: (Bool, Bool) -> Bool
and''' (True,  True) = True
and''' (_   ,  _   ) = False
```

$$
\begin{aligned}
\mathit{and'} &: \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\
\mathit{and'}(\mathrm{T},\mathrm{T}) &\doteq \mathrm{T} \\
\mathit{and'}(\mathrm{T},\mathrm{F}) &\doteq \mathrm{F} \\
\mathit{and'}(\mathrm{F},\mathrm{T}) &\doteq \mathrm{F} \\
\mathit{and'}(\mathrm{F},\mathrm{F}) &\doteq \mathrm{F}
\end{aligned}
$$

整数的算术运算

在书写算术运算相关的数学方程时，我们通常不会采用函数的形式进行书写，而是采用更为直观的算术运算操作符（operator）．例如，给定两个整数 `a,b`，如果要将这两个数相加，我们通常不会书写 `plus(a,b)`，而会书写为 `a+b`．这里，`+` 就是一个操作符，实现将左右两个操作数相加的功能．对于大部分人类个体而言，采用操作符书写的算术运算更加直观．为了继承这种直观性，Haskell 语言提供了算数运算的常用操作符．对于整数上的加、减、乘、指数这四种算数运算，对应的操作符分别为：`+`，`-`，`*`，`^`．这些操作符具有两个操作数，所以也被称为二元操作符．

下面，我们采用操作符对操作符对应的函数进行定义，目的是展示这些操作符的使用方式.

```
plus :: Integer -> Integer -> Integer
plus x y = x + y

minus :: Integer -> Integer -> Integer
minus x y = x - y

mult :: Integer -> Integer -> Integer
mult x y = x * y

expn :: Integer -> Integer -> Integer
expn x y = x ^ y
```

Haskell 语言提供了一种语法机制，可以将任意一个二元操作符变换为对应的函数，即：把一个二元操作符放在一对圆括号中．例如，表达式 `x + y`，可以等价地书写为 `(+) x y`．也就是说，`(+)` 是一个函数，其定义如下：

```
(+) :: Integer -> Integer -> Integer
(+) x y = x + y
```

事情还没有完．我们还可以把一个操作数和一个操作符放在一对圆括号中，得到一个新的函数．例如，`(x +)` 和 `(+ y)` 也是两个合法的表达式，分别表示两个函数．这两个函数的定义如下：

```
(x +) :: Integer -> Integer
(x +) y = x + y

(+ y) :: Integer -> Integer
(+ y) x = x + y
```

Haskell 语言还提供了把一个合适的函数转换为对应的二元操作符的语法机制，即：把这个函数放在一对 `` 符号中．例如，表达式 `div x y`，可以等价地书写为 `x `div` y`.

最后，还存在一件奇怪的事情：为什么没有介绍整数的除运算呢？这个事情稍微有那么一点点复杂．两个整数相除，你是想得到一个整数，还是想得到一个更准确的带小数部分的数值呢？对于这两种情况，Haskell 语言分别提供了对应的函数和操作符．例如，对于 `5 ÷ 2`，如果想得到一个整数值，则可以书写为 `div 5 2` 或者 `5 `div` 2`；如果想得到一个带小数部分的数值，则可以书写为 `5 / 2` 或者 `(/) 5 2`．其中，`div` 和 `/` 分别是 `Prelude` 模块中定义的一个函数和一个操作符．

作业 02 请用目前介绍的 Haskell 语言知识，给出函数 `div` 的一种或多种定义． `div :: Integer -> Integer -> Integer`

- 不用关注效率
- 如果你认为这个问题无解或很难，请给出必要的说明（为什么无解或主要困难在哪里）

自然数相关的函数

```
fact :: Natural -> Natural
fact 0  = 1
fact n  = n * fact (n-1)
```

- Natural 是在 Haskell 语言的模块 Numeric.Natural 中定义的一种自然数类型，可以表示任意精度的自然数．在缺省情况下，该模块不会被自动加载．因此，为了使用 Natural 类型，首先需要把这个类型加载到当前程序中．可以使用下面这条语句实现对该类型的加载

```
import Numeric.Natural (Natural)
```

需要注意的是，该语句仅仅会把模块 Numeric.Natural 中定义的元素 Natural 加载到当前模块中，而不会加载该模块中定义的其他元素．如需加载其他元素，还需要把这些元素名添加到上面语句的圆括号中（元素名之间用逗号作为间隔符）．如果要加载一个模块中定义的所有元素，直接把模块名放在 import 后面即可（不用在模块名后面书写任何字符）．

```
fact :: Natural -> Natural
fact 0  = 1
fact n  = n * fact (n-1)
```

- 第二行和第三行代码采用了模式匹配的方式对函数进行定义. 具体可以阐述为：对于一个自然数 `n`，如果 `n == 0`，则将表达式 `fact n` 定义为自然数 `1`；否则，将表达式 `fact n` 定义为表达式 `n * fact (n-1)`.
- 你可能会疑惑：上面的定义中明明没有出现“如果…，则…；否则，…”这样的结构. 是的，你观察地很仔细. 这里涉及到一个潜规则：在进行模式匹配时，按照程序语句的顺序依次匹配. 在上面的定义中，首先尝试匹配的是 `fact 0`. 如果匹配成功，则得到定义中等号右侧的值；然后，匹配活动就停止了. 只有当匹配 `fact 0` 失败后，才会匹配 `fact n`. 在这里，`n` 是一个通配符[8]，可以匹配到任何自然数.

- 第三行语句定义符号右侧的表达式，反应出 Haskell 语言对于运算优先级的一些规定．在 Haskell 语言中，函数调用/应用具有最高的优先级．因此，表达式 `n * fact (n-1)` 等价于 `n * (fact (n-1))`；但显然前者更加简洁．类似地，表达式 `n * fact n - 1` 等价于 `n * (fact n) - 1`.
- 第三行语句还涉及了另外一个概念：函数的递归调用．在这里，在定义 `fact n` 的过程中，我们使用到了 `fact (n-1)`．但是，这并不是循环定义，因此并不存在逻辑上的冲突．例如，当需要对表达式 `fact (n-1)` 进行求值时，我们又会将其转换为对 `fact (n-2)` 的求值．这样一直走下去，直到遇到 `fact 0`，它的值被定义为自然数 `1`．此时，就不需要再继续向前走了．

```
fact :: Natural -> Natural
fact 0  = 1
fact n  = n * fact (n-1)
```

作业 03

关于阶乘函数，你还能想到其他定义方式吗？请分别用条件方程组和分支表达式写出阶乘函数的定义．

```
fib :: Natural -> Natural

fib 0 = 0

fib 1 = 1

fib n = fib (n-1) + fib (n-2)
```

这个定义中没有涉及新的细节信息. 这个定义虽然简洁，但在实际运行时，效率很低.

```
foldn :: (a -> a) -> a -> Natural -> a
foldn h c 0 = c
foldn h c n = h (foldn h c (n-1))
```

- 在函数的类型声明中出现了一个小写字母 `a`，它表示的是一个类型变量（type variable）. 在实际调用 `foldn` 函数时，Haskell 会根据实际传入的参数的类型，确定类型变量 `a` 的具体取值.
- 如果传入 `foldn` 的参数不合法（也即，不满足 `foldn` 的类型要求），Haskell 会报出错误. 例如，如果传入 `foldn` 的第一个参数是一个类型为 `Natural -> Bool` 的函数，那么这个函数就不是一个合法的值：`a -> a` 这个类型声明要求传入的函数具有相同的定义域和值域；类型为 `Natural -> Bool` 的函数显然不满足这种类型要求.

```
foldn :: (a -> a) -> a -> Natural -> a
foldn h c 0 = c
foldn h c n = h (foldn h c (n-1))
```

- 那么，如何确定函数类型声明中出现的一个名称是类型还是类型变量呢？Haskell 在语法层次上给出了一种简单有效的解决方案：如果这个名称的首字符是小写字母，则它表示一个类型变量；如果首字符是大写字母，则表示一个类型.

```
foldn :: (a -> a) -> a -> Natural -> a
foldn h c 0 = c
foldn h c n = h (foldn h c (n-1))
```

- 在第三行语句定义符号的右侧出现了一个奇怪的事情．在前文中，我们刚刚讲过，在 Haskell 语言中调用函数时，不需要在参数前后加上圆括号；那么，这里在调用函数 `h` 时，为什么又在参数前后加上圆括号了呢？这里的括号不是包围函数参数的圆括号，而是用于修改运算优先级的圆括号．如果把 `h` 后面的圆括号删除，会得到表达式 `h foldn h c (n-1)`；其中出现了两个连续的函数调用．Haskell 语言规定：函数调用具有*左*结合性．因此，这个表达式等价于 `(h foldn) h c (n-1)`；这显然不是我们所期望的．

```
foldn :: (a -> a) -> a -> Natural -> a
foldn h c 0 = c
foldn h c n = h (foldn h c (n-1))
```

- 如果你对表达式 `h (foldn h c (n-1))` 中 `h` 后面的这一对圆括号很反感，Haskell 语言还提供了一个二元操作符 `$`，可以帮你避免这种反感．`$` 是 Haskell 语言中具有最低优先级的二元操作符，且具有*右*结合性；除此之外，这个操作符没有任何其他功能．使用这个操作符，上述表达式可以改写为 `h $ foldn h c (n-1)`．进一步，利用 `$` 具有的*右*结合性，我们可以消除仍然存在的一对圆括号，即：`h $ foldn h c $ n - 1`．

```
f :: (Natural, Natural) -> (Natural, Natural)
f (m, n) = (m + 1, (m + 1) * n)
```

```
fact' :: Natural -> Natural
fact' = outr.(foldn f (0,1))
```

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

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

在 `fact'` 的定义中，我们使用了 Haskell 语言提供的点操作符．这个操作符实现了函数组合（见定义 1.1.3）的功能．例如，给定两个函数 `f,g` 以及一个合法的表达式 `f (g x)`．使用点操作符，这个表达式可以改写为 `(f.g) x`．需要注意的是，表达式 `(f.g) x` 中的这对圆括号不能忽略．函数调用具有最高优先级；因此，忽略括号后形成的表达式 `f.g x` 等价于 `f.(g x)`，这可能不是你所期望的．但是，你可能也注意到了，把表达式 `f (g x)` 等价地改写为 `(f.g) x`，非但没有让我们消除这对圆括号，还多出了一个点操作符．如果你对此耿耿于怀，还可以这样改写: `f.g $ x`．

最后，请看斐波那契函数 fib' 的定义：

```
g :: (Natural, Natural) -> (Natural, Natural)
g (m, n) = (n, m + n)

fib' :: Natural -> Natural
fib' = outl.(foldn g (0,1))
```

序列以及序列上的fold函数

在 Haskell 语言中，给定一个类型 `a`，`[a]` 表示一个新的数据类型，其中包含了所有由 0 到多个 `a` 中的元素形成的序列．可以将 `[]` 理解为一个函数：该函数接收一个类型，返回一个对应的序列类型．

下面，我们以整数类型 `Integer` 为例，展示 Haskell 语言对序列的若干表示方式：

- `[]` 表示空序列，即：由 0 个整数构成的序列．
- `[1]` 表示由一个整数 1 构成的序列．这个序列也可以表示为 `1:[]`．这里，`:` 是一个二元操作符，其功能是把左操作数添加到右操作数（一个序列）的左侧，从而返回一个新的序列．
- `[1,2,2,3,3,3]` 表示由 6 个自然数形成的序列．类似地，这个序列也可以表示为 `1:2:2:3:3:3:[]`．因此可知，操作符 `:` 具有右结合性．当然，如果你愿意，也可以将这个序列表示为 `1:2:2:[3,3,3]`．

```
len :: [a] -> Natural
len []     = 0
len (n:ns) = 1 + len ns
```

```
rev  :: [a] -> [a]
revm :: [a] -> [a] -> [a]
rev          = revm []

revm xs []     = xs
revm xs (y:ys) = revm (y:xs) ys
```

```
concat' :: [a] -> [a] -> [a]
concat' []     ns = ns
concat' (m:ms) ns = m : concat' ms ns
```

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' p []                  = []
filter' p (n:ns) | p(n)       = n : filter' p ns
                 | otherwise =     filter' p ns
```

首先，请看 foldlr 函数的定义：

```
foldlr :: (a -> b -> b) -> b -> [a] -> b
foldlr h c []     = c
foldlr h c (x:xs) = h x (foldlr h c xs)
```

然后，请看 foldll 函数的定义：

```
foldll :: (a -> b -> b) -> b -> [a] -> b
foldll h c []     = c
foldll h c (x:xs) = foldll h (h x c) xs
```

对前面4个函数的重定义

```
len' :: [a] -> Natural
len' = foldlr h 0
h :: a -> Natural -> Natural
h x n = n + 1

rev' :: [a] -> [a]
rev' = foldll (:) []
```

```
concat'' :: [a] -> [a] -> [a]
concat'' xs ys = foldlr (:) ys xs

filter'' :: (a -> Bool) -> [a] -> [a]
filter'' p = foldlr (k p) []
k :: (a -> Bool) -> a -> [a] -> [a]
k p x | p x         = (x:)
      | otherwise = id'
id' :: a -> a
id' x = x
```

一种快速排序算法

```
qsort :: [Integer] -> [Integer]
qsort []     = []
qsort (n:ns) = concat' (qsort $ filter (<  n) ns)
               $       n:(qsort $ filter (>= n) ns)
```

Haskell 还提供了一些语法机制，可以让上述 `qsort` 函数的定义更加结构化．一种是 `let ... in ...` 表达式．请看下面的函数定义：

```
qsort' :: [Integer] -> [Integer]
qsort' []     = []
qsort' (n:ns) = let smaller = qsort' $ filter (<  n) ns
                    larger  = qsort' $ filter (>= n) ns
                in  concat' smaller (n:larger)
```

我们还可以通过 where 子句对 qsort 函数进行另一种形式的改写. 请看下面的函数定义:

```
qsort'' :: [Integer] -> [Integer]
qsort'' []     = []
qsort'' (n:ns) = concat' smaller (n:larger)
         where smaller = qsort'' $ filter (<  n) ns
               larger  = qsort'' $ filter (>= n) ns
```

- where 子句不是一个表达式. 在上面的程序中, where 子句挂载到定义 qsort (n:ns) 上. 在这个子句中，可以引用 n 和 ns 这两个变量. 同时, 在 qsort (n:ns) 定义的右侧到关键词 where 的这个区域，都可以引用 where 中定义的变量. where 子句中定义的变量，通常具有更大作用范围. 例如，在下面这个示意性的函数定义中:

```
f x y | cond1 x y = g z
      | cond2 x y = h z
      | otherwise = k z
     where z = p x y
```

我们在 where 子句中定义了一个变量 z，而在条件方程组的任何地方都可以访问到变量 z.

你的感觉如何？我们用了一些朝三暮四的把戏（规定一些语法规则），把原本很难理解的一个函数定义（见公式 1.47）变得似乎没有那么难于理解了.

一个好的程序设计语言应该具有一个基本性质：用这种语言书写出的程序具有很好的易理解性. 同时，你应该注意到：易理解性不是程序单方面的事情，而是与试图理解程序的主体紧密相关. 在我们的上下文中，你必须深刻理解函数式思维的特点，才有可能很容易看懂使用函数式思维书写的程序，也才有可能书写出体现函数式思维的优雅程序.

标识符和操作符的命名规则

在上一节给出的 Haskell 函数定义中，可以观察到一个现象：在很多时候，我们需要为程序中出现的各种成分命名．所谓命名，就是给一个东西起一个名称．命名的作用在于可以通过名称引用到一个已定义的程序元素．

- 例如，当我们定义了一个有名称的函数，那么，就可以通过函数名对一个函数进行调用．
- 又例如，我们通过模式匹配匹配到特定的变量，并且给这个变量命名后，那么在定义符号的右侧，就可以通过变量名引用到这些变量．

在 Haskell 语言中，这些名称被分为两大类：标识符（Identifier）和操作符（Operator symbol）．

标识符的命名规则

- 标识符由 1 或多个字符构成
- 标识符的首字符只能是一个字母（letter）. 具体而言，字母包括 ASCII 编码表中的所有字母（也即：所有的小写和大写英文字母），以及 Unicode 编码表中的所有小写和大写字母.
- 除首字符外，标识符中的其他字符（如果存在）只能是字母、数字、英文下划线、或者英文单引号.
- 最后，标识符不能与 Haskell 语言的保留词重名. 这些关键词包括：`case`、`class`、`data`、`default`、`deriving`、`do`、`else`、`foreign`、`if`、`import`、`in`、`infix`、`infixl`、`infixr`、`instance`、`let`、`module`、`newtype`、`of`、`then`、`type`、`where`、`_`.

标识符的命名规则

根据命名的程序元素的不同，Haskell 还对标识符的首字符进行了进一步的限制．有一些程序元素，其标识符的首字符只能是大写字母；其他程序元素，其标识符的首字符只能是小写字母．具体信息，我们会在介绍这些程序元素时进行说明．目前已经涉及的程序元素包括：函数、变量、以及类型变量的名称的首字符是小写字母；类型的名称的首字符是大写字母．

操作符的命名规则

- 操作符由 1 或多个符号（Symbol）构成．具体而言，符号包括 ASCII 编码表中的所有符号，以及 Unicode 编码表中的绝大多数符号．ASCII 编码表中的符号包括：`!`、`#`、`$`、`%`、`&`、`*`、`+`、`.`、`/`、`<`、`=`、`>`、`?`、`@`、`\`、`^`、`|`、`-`、`~`、`:`．Unicode 编码表中的可用符号或不可用符号我们就不一一列举了．如果你有兴趣，可自行查找相关资料．
- 用户自定义的操作符不能与 Haskell 语言的保留操作符重名．这些保留操作符包括：`..`、`:`、`::`、`=`、`\`、`|`、`<-`、`->`、`@`、`~`、`=>`．

Haskell 语言进一步将操作符划分为两类：以冒号 `:` 为首字符的操作符、其他操作符．这两类操作符的含义在后文中合适的地方会进行说明．

Hello, World!

在很多程序设计语言的教科书中，都会包含一个程序．这个程序的功能是在控制台打印出一个字符串 `Hello, World!`．我们也从这样一个简单的程序出发，开始接下来的 Haskell 语言探索之旅．

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

上面这个程序就是 Haskell 语言中的 `Hello, World!` 程序．如果你把上述程序书写在一个文本文件中，然后在合适的环境中运行这个程序[10]，你就会在控制台看到程序输出的一个字符串．

按照 Haskell 语言规范，这个程序省略了一些语句，以至于看起来有些奇怪. 加上这些被省略的语句，会得到如下更为完整的程序.

```
module Main(main) where
  import Prelude

  main :: IO ()
  main = do
    putStrLn "Hello, World!"
```

```
module Main(main) where
  import Prelude


  main :: IO ()
  main = do
    putStrLn "Hello, World!"
```

这个程序的第一行 `module Main(main) where` 是一个模块声明语句．其中，`module` 和 `where` 是 Haskell 语言的两个关键词．这个语句的效果是：声明当前文件即将定义一个名称为 `Main` 的模块，而且声明这个模块对外输且仅对外输出一个名称为 `main` 的程序元素（至于这个程序元素的具体定义是什么，可以在后面的语句中看到）．这行语句的最后是关键词 `where`．这个关键词我们在前面已经看到过：可以在其后定义若干前文中出现但未被定义的程序元素．

关于模块，Haskell 语言规范给出了如下信息

- 一个 Haskell 程序由 1 或多个模块构成．每一个模块定义在一个单独的文件中．
- 一个 Haskell 程序中必须包含一个名称为 `Main` 的模块．这个模块必须输出一个名称为 `main` 的程序元素．这个元素的类型必须是 `IO t`：其中，`t` 是一个类型变量；`IO` 是在 `Prelude` 模块中定义的一种程序元素，用于封装 IO 相关的计算．一个 Haskell 程序的运行过程就是对 `Main` 模块中的 `main` 元素进行求值的过程；而且，最终获得的值会被抛弃．

- 模块的名称满足如下两种条件之一：(1) 一个首字符为大写字母的标识符(例如,`Mymodule`);(2) 二个或多个首字符为大写字母的标识符通过点符号连接在一起(例如,`This.Is.Mymodule`).
- 如果一个模块在设计时就已经确定不会被其他的模块所引用，那么，这么模块可以放在任意具有合法名称的文件中．通常，一个 Haskell 程序的 `Main` 模块不会被其他模块所引用．因此，可以给 `Main` 模块所在的文件起一个不同于 `Main` 的名称．但是，将 `Main` 模块的文件名设定为 `Main`，不失为一个好的选择.

- 如果一个模块在设计时被确定会被其他模块所引用，那么这个模块所在文件的名称必须满足如下条件．如果模块的名称是一个标识符，那么，这个模块所在文件的名称必须等于模块名．如果模块的名称是由两个或多个标识符通过点符号连接形成，那么，这个模块所在文件的名称必须等于模块名中最后一个标识符的名称．同时，这个模块名最后一个标识符之前的所有标识符分别对应到文件系统中的一个文件夹 (Directory)；且相邻标识符对应的文件夹存在嵌套关系；且这个模块所在的文件必须存放在倒数第二个标识符对应的文件夹下[11]．例如，对模块 `This.Is.Mymodule`而言，如果它会被其他模块引用，那么，它必须存放在文件夹 `This` 下的子文件夹 `Is` 中一个名称为 `Mymodule` 的文件中．

- 在上述关于文件名的论述中，文件名指的是不包含扩展名的文件名[12]．一个 Hasekell 模块文件，按照程序的书写方式不同，具有两种不同的文件扩展名．一种是 hs：目前为止出现的所有 Haskell 程序的书写方式，都属于这种类型．另一种是 lhs，全称为 literate Haskell．这种书写方式的更多信息见下文．

```
module Main(main) where
  import Prelude


  main :: IO ()
  main = do
    putStrLn "Hello, World!"
```

上面程序的第二行 `import Prelude` 是一个模块加载语句. 其效果是把 `Prelude` 模块对外输出的所有程序元素加载到当前模块中.

- Haskell 语言规范规定，如果在一个模块的源文件中没有通过 `import` 语句引入 `Prelude` 模块的任何输出元素，那么，就等价于该模块中存在一个 `import Prelude` 语句.
- 但是,如果模块中已经存在一条 `import` 语句,其引入了 `Prelude` 模块中的任何一个或多个输出元素，那么，该模块就不存在一个隐式的 `import Prelude` 语句.

- 例如，如果模块中存在这样一条语句：

```
import Prelude(Integer, (+),(-))
```

这条语句的效果是：把 Prelude 模块对外输出的一个类型 Integer 和两个操作符 +, - 加载到当前模块中；Prelude 模块对外输入的其他任何元素都不会被加载到当前模块中．因此，无需再画蛇添足地把 import Prelude 语句添加到当前模块中．

因此，下面的 Haskell 模块定义不是一个合法的 Haskell 程序.

```
module Main(main) where
  import Prelude(Integer, (+), (-))

  main :: IO ()
  main = do
    putStrLn "Hello, World!"
```

原因在于：在这个模块定义中，出现了两个未定义的程序元素 `IO` 和 `putStrLn`. 把这两个元素添加到 `import Prelude` 后面的一对圆括号中，才能得到一个合法的模块定义.

```
module Main(main) where
  import Prelude


  main :: IO ()
  main = do
    putStrLn "Hello, World!"
```

小麻雀的第三行语句是 `main :: IO()`. 该语句的效果是: 声明程序元素 `main` 的类型是 `IO ()`. 其中, `()` 表示一个元组（Tuple）类型, 且该类型仅包含 1 个值. 可以说, 类型 `()` 中不包含任何有意义的值. 这也说明, `main` 元素所封装的 IO 计算过程返回值没有什么实际的含义.

```
module Main(main) where
  import Prelude


  main :: IO ()
  main = do
    putStrLn "Hello, World!"
```

小麻雀的最后两行定义了 `main` 元素所封装的计算过程．其中，只包含一个 IO 动作，即：在控制台打印出字符串 `Hello, World!`．当然，如果你愿意，可以添加更多的 IO 动作．例如，可以在上面的程序后添加一行语句：`putStrLn "Hello, World! AGAIN"`．此时，`main` 所封装的 IO 计算过程中就包含了两个顺序执行的 IO 动作．

```
module Main(main) where
  import Prelude


  main :: IO ()
  main = do
    putStrLn "Hello, World!"
```

天生好奇的你，一定发现了一个非常奇怪的地方：do 是何方神圣? 其实呢，只要不是黑白色盲，基本上都能看到这一点．其实呢，上面两句话只是为了缓解一下尴尬．对于绝大部份程序设计语言而言，IO 操作都不是什么大不了的事情．但是，对于 Haskell 这类函数式语言而言，IO 操作是一种非常不和谐的存在．因为在函数式思维所依赖的数学概念中，并不存在一个或多个概念能够对 IO 操作进行有效建模．

```
module Main(main) where
  import Prelude


  main :: IO ()
  main = do
    putStrLn "Hello, World!"
```

但是，如果想要开发一个可以和用户动态交互的软件，IO 操作是不可避免的．为此，Haskell 语言的设计者们想到了一个对策：把 IO 操作放在特殊的数据类型中封装起来．于是，`Prelude` 模块中就定义了一个程序元素 `IO`．这个元素的具体定义，我们在后文中会详细说明．

```
module Main(main) where
  import Prelude


  main :: IO ()
  main = do
    putStrLn "Hello, World!"
```

让我们把关注点再次聚焦到 do 上．在没有介绍更多的相关知识之前，我们无法给出 do 的准确定义．下面，我们采用自然语言对它的效果进行描述．在比喻的意义上，do 和 where 由一些相似之处．在 where 之后，可以给出 1 或多个变量的定义．而在 do 之后，可以给出 1 或多个 IO 操作，其表达的含义是：按照出现顺序，依次执行（do）这些 IO 操作．关于 do 的解释，暂且先到这里．

```
main :: IO ()
main = do
  putStrLn "Hello, World!"
```

小麻雀的 do 后面给出了一个 IO 操作：putStrLn "Hello, World!"。其中，putStrLn 是在 Prelude 模块中定义的一个封装了 IO 操作的函数，其类型为 String -> IO ()。这里，String 是 Prelude 模块中定义的一种字符串类型。从 putStrLn 函数的类型可知，该函数接收一个字符串类型的值，然后进行内部的 IO 操作，最后返回 0 元组类型中存在的唯一一个值 ()。此处的 IO 操作就表现为对 putStrLn 函数的一次调用。另外，这个函数调用还反映出 Haskell 语言的一个语法规则：在声明一个字符串字面量（Literal）时，需要在这个字符串的前后加上双引号。这种设计决策的原因之一是：如果不在字符串前后加上双引号，则字符串的内容有可能会与标识符或操作符重名，从而导致歧义。

在结束本节的内容之前，我们对上面的 Hello, World! 程序进行一点扩展，让它变的更有交互性．请看下面的程序[13]：

```
00 module Main(main) where
01  import Prelude
02
03  main :: IO()
04  main = do
05    putStrLn "Please input your name:"
06    name <- getLine
07    putStrLn $ "Hello, " ++ name
08    putStrLn "Please input an integer:"
09    str1 <- getLine
10    putStrLn "Please input another integer:"
11    str2 <- getLine
12    let int1 = (read str1 :: Integer)
13    let int2 = (read str2 :: Integer)
14    putStrLn $ str1 ++ " + " ++ str2 ++ " = "
15                    ++ (show $ int1 + int2)
```

这个程序实现了如下的交互行为：

1. 程序在控制台打印出一个字符串，提示用户输入姓名；
2. 用户在控制台输入一个字符串作为其姓名；
3. 程序接收到用户输入的姓名，然后打印出欢迎信息；
4. 程序提示用户输入一个整数；
5. 用户输入一个整数字符串；
6. 程序提示用户输入另一个整数；
7. 用户输入另一个整数字符串；
8. 程序将用户输入的整数字符串转换成对应的整数；
9. 程序打印出这两个整数的和．

在结束本节的内容之前，我们对上面的 Hello, World! 程序进行一点扩展，让它变的更有交互性．请看下面的程序[13]：

```
00 module Main(main) where
01  import Prelude
02
03  main :: IO()
04  main = do
05    putStrLn "Please input your name:"
06    name <- getLine
07    putStrLn $ "Hello, " ++ name
08    putStrLn "Please input an integer:"
09    str1 <- getLine
10    putStrLn "Please input another integer:"
11    str2 <- getLine
12    let int1 = (read str1 :: Integer)
13    let int2 = (read str2 :: Integer)
14    putStrLn $ str1 ++ " + " ++ str2 ++ " = "
15                    ++ (show $ int1 + int2)
```

- 第 6 行 `name <- getLine` 这种语法结构在前文没有出现过．其中出现了 2 个陌生的成分：`getLine` 和 `<-`．
  - `getLine` 是在 `Prelude` 模块中定义的一个程序元素，其类型为 `IO String`．从这个类型可知，`getLine` 封装了特定的 IO 操作，并最终形成一个 `String` 类型的值．具体而言，`getLine` 实现的 IO 操作是从控制台读取用户输入的一行字符串．虽然 `getLine` 看起来像是一个函数，但它真的不是一个函数．
  - `<-` 看起来像是一个二元操作符，但它真的不是一个二元操作符．在比喻的意义上，它和前面程序中出现的*定义为*符号 `=` 有些类似．`=` 效果是把左侧表达式的值定义为右侧表达式的值．`<-` 是与关键词 `do` 绑定的一种符号．在第 6 行程序中，其效果是把左侧表达式的值定义为右侧 IO 操作 `getLine :: IO String` 最终形成的那个字符串．

在结束本节的内容之前，我们对上面的 Hello, World! 程序进行一点扩展，让它变的更有交互性．请看下面的程序[13]：

```
00 module Main(main) where
01  import Prelude
02
03  main :: IO()
04  main = do
05    putStrLn "Please input your name:"
06    name <- getLine
07    putStrLn $ "Hello, " ++ name
08    putStrLn "Please input an integer:"
09    str1 <- getLine
10    putStrLn "Please input another integer:"
11    str2 <- getLine
12    let int1 = (read str1 :: Integer)
13    let int2 = (read str2 :: Integer)
14    putStrLn $ str1 ++ " + " ++ str2 ++ " = "
15                    ++ (show $ int1 + int2)
```

- 第 12 行程序 `let int1 = (read str1 :: Integer)` 看起来既熟悉又陌生：我们看到了熟悉的 `let`，但是却没有看到它的好伙伴 `in`．但是，这里的 `let` 就是以前我们看到的 `let`，如假包换，用于定义在后文中可以被引用的变量；只不过在 IO 这样一类特殊的上下文中，为了程序的简洁性，它隐藏了一些东西．具体原因和细节一言难尽，暂且放下它．

在结束本节的内容之前，我们对上面的 Hello, World! 程序进行一点扩展，让它变的更有交互性．请看下面的程序[13]：

```
00 module Main(main) where
01  import Prelude
02
03  main :: IO()
04  main = do
05    putStrLn "Please input your name:"
06    name <- getLine
07    putStrLn $ "Hello, " ++ name
08    putStrLn "Please input an integer:"
09    str1 <- getLine
10    putStrLn "Please input another integer:"
11    str2 <- getLine
12    let int1 = (read str1 :: Integer)
13    let int2 = (read str2 :: Integer)
14    putStrLn $ str1 ++ " + " ++ str2 ++ " = "
15                    ++ (show $ int1 + int2)
```

- 表达式 `read str1 :: Integer` 看起来有些奇怪．`read` 是在 `Prelude` 模块中声明的一个函数，它的类型大概[14]是 `String -> a`．这个函数实现的功能就是把一个字符串变换为类型为 `a` 的一个值．这是，`a` 是一个类型变量．在调用 `read` 函数的时候，需要将 `a` 绑定到一个具体的类型上．因此，在调用 `read` 时，如果不能够从上下文推断出 `a` 的类型，我们需要在函数调用后加上 `:: Type`，显式绑定 `a` 为类型 `Type`[15]；此处，你需要把 `Type` 替换为当前模块中可以访问到的一个合适的类型（例如：`Integer`）．

在结束本节的内容之前，我们对上面的 Hello, World! 程序进行一点扩展，让它变的更有交互性．请看下面的程序[13]：

```
00 module Main(main) where
01  import Prelude
02
03  main :: IO()
04  main = do
05    putStrLn "Please input your name:"
06    name <- getLine
07    putStrLn $ "Hello, " ++ name
08    putStrLn "Please input an integer:"
09    str1 <- getLine
10    putStrLn "Please input another integer:"
11    str2 <- getLine
12    let int1 = (read str1 :: Integer)
13    let int2 = (read str2 :: Integer)
14    putStrLn $ str1 ++ " + " ++ str2 ++ " = "
15                    ++ (show $ int1 + int2)
```

- 第 15 行程序中的 show 是在 Prelude 模块中声明的一个函数，其类型为 a -> String．这个函数的作用与 read 恰好是相反的：它把一个类型为 a 的值转换为一个字符串．

掌握了 Haskell 语言输出/输出相关的操作，再加上前面介绍的 Haskell 程序设计知识，你应该可以做很多事情了．但是，非常遗憾的是，这些程序目前还不能动．不用担心，想让程序动起来，就是分分钟的事情．

Haskell 程序的编译、运行、管理

当你用自然语言写了一本小说，你可以把它发表在互联网上；然后，读者们就可以快乐地阅读你小说了．当你用 Haskell 语言写了一个程序，你也可以把它发表在互联网中的某个代码托管网站[16]上；然后，程序员们就可以阅读你的程序了．

但是，与采用自然语言书写的小说相比，采用程序设计语言书写的程序还有另外一类读者：计算机．计算机需要理解程序表达的计算过程，并在各类硬件和软件资源的支持下，执行程序所表达的计算过程．因此，对于一种程序设计语言的发明者们而言，定义语言的语法形式，仅仅是万里长征的第一步．为了让程序能够运行在计算机上，还需要提供一系列的软件支撑工具．这些工具又被称为程序设计语言的工具链（toolchain）．

在本节中，我们主要介绍 Haskell 语言工具链中的三个基本工具：

- GHC：全称为 Glasgow Haskell Compiler，是一种得到广泛使用的 Haskell 语言编译器，能够把合法的 Haskell 程序变换为计算机可执行的机器指令序列．
- GHCi：GHC 的一种交互式程序运行环境．程序员可以在其中输入任意合法的 Haskell 表达式，然后 GHCi 对表达式进行求值，并输出求值的结果．
- Stack：一种常用的 Haskell 软件开发项目管理工具．

工具的安装

安装程序设计语言的编译器等系统软件，通常不是一个具有良好体验的过程．其中会涉及各种看似琐碎但却直接影响安装结果的细节和前置条件．当你耗费九牛二虎之力安装成功后，在很长一段时间内可能都不会再使用到安装过程中积累的成功经验；于是，这些经验就会慢慢被遗忘．许多年之后，你又会再一次经历这种探索并遗忘的过程．

Haskell 语言上述工具的安装也不是一件容易的事情．在最原始的层次上，你可以下载到这些工具的源代码，在本机对源代码进行编译，生成可执行文件，并进行相关的设置．对于程序设计语言的初学者而言，不建议采用这样的方式进行安装．费时费力，而且不一定会成功．

Haskell 语言的官方网站提供了一种按操作系统进行安装的教程.在浏览器中打开如下链接[17]:

```
https://www.haskell.org/platform/
```

这个链接对应的页面会自动检测你当前使用的操作系统，并在页面的下部展示针对这个操作系统的安装说明．典型的操作系统包括: Mac OS X、Windows、以及 Linux 的各种发行版本.

下面，我们以 Mac OS X 操作系统为例，展示如何进行上述工具的安装．上面这个页面给出的安装说明是一段文字：

The recommended way to install the components of the mac platform is using ghcup to install ghc and cabal-install, and following the instructions at haskellstack.org to install stack.

程序员写的东西一般很简洁．翻译成中文，大概包括两点信息：

1. 请使用 ghcup 安装 ghc 和 cabel-install[18]．
2. 请按照网站 haskellstack.org 上的说明安装 stack．

在按照上述步骤安装之前，建议你先安装 Mac OS X 上的官方软件开发工具 Xcode（如果已经安装了，则需要将其升级为最新版本）[19]：Xcode 会帮助你在本机上建立一个 C/C++ 语言的编译环境，为后续的安装过程提供便利．安装 Xcode 的方式很简单：打开 Mac OS X 上的 App Store，找到 Xcode，然后点击安装按钮．Xcode 是一个免费的应用．Xcode 安装文件的体积有点大，大概 11GB 左右；所以要耐心等待一段时间．

安装 ghcup 及相关工具

点击安装说明中 ghcup 这个超链接，浏览器跳转到一个新的页面．
这个页面告诉你，你需要在终端（Terminal）[20]中运行如下命令：

```
curl --proto '=https' --tlsv1.2 -sSf https://get-ghcup.haskell.org | sh
```

所谓“运行如下命令”，就是把上述字符串拷贝到终端中光标所在位置，然后回车．在此之前，需要确保本机可以访问互联网，因为安装过程需要从互联网上下载相关的安装文件．如果一切顺利，安装过程就开始了．你需要按照终端上的提示逐步进行操作．

首先，安装程序给出如下提示信息：

Welcome to Haskell!

This script will download and install the following binaries:

* ghcup - The Haskell toolchain installer
* ghc - The Glasgow Haskell Compiler
* cabal - The Cabal build tool for managing Haskell software
* stack - (optional) A cross-platform program for developing Haskell projects
* hls - (optional) A language server for developers to integrate with their editor/IDE

Press ENTER to proceed or ctrl-c to abort.

按下回车键，继续进行安装．然后，安装程序会提示你输入若干安装选项．如果没有意外，一路按回车键即可．

Note that this script can be re-run at any given time.

接下来，安装程序会从互联网上下载安装文件，体积为 12.7M．耐心等待一会．下载完成后，安装程序会给出一段提示信息：

System requirements Note: On OS X, in the course of running ghcup you will be given a dialog box to install the command line tools. Accept and the requirements will be installed for you. You will then need to run the command again.

Press ENTER to proceed or ctrl-c to abort. Installation may take a while.

它告诉你：下面的安装过程可能会弹出一个对话框，请你同意其中的安装请求；然后，你需要重新运行上面的命令安装 ghcup．接下来，按下回车键，继续下面的安装活动．

如果成功安装，安装会给出如下信息：

```
==============================================

OK! /Users/Saturn/.bashrc has been modified. Restart your terminal
for the changes to take effect, or type "source /Users/Saturn/.ghcup/env"
to apply them in your current terminal session.

==============================================

All done!

To start a simple repl, run: ghci

To start a new haskell project in the current directory, run: cabal init
--interactive

To install other GHC versions and tools, run: ghcup tui
```

- 你需要重启终端程序，或在其中运行一个命令，从而使得安装过程中的一些配置可以生效.
- 使用 `ghci` 命令可以启动 Haskell 程序的一个交互式运行环境.
- 使用 `cabal init --interactive` 命令可以快速建立一个 Haskell 软件开发项目. 请不要进行这方面的尝试，因为下面即将介绍的 stack 工具会做的更好.
- 使用 `ghcup tui` 命令[22]，可以启动一个终端环境下的用户界面，在其中，可以对 Haskell 的相关工具进行安装和管理. 例如，你可选择安装其他版本的 GHC 编译器，并设定在终端中默认使用的编译器版本.

安装成功后，在本机的终端中，会增加三个新的可用命令：ghcup、ghc、ghci．后面两个命令分别是 Haskell 程序的编译器和交互式运行环境，我们稍后会对它们进行说明．第一个命令 ghcup 是 Haskell 工具链安装工具（Haskell toolchain installer）．关于 ghcup 命令的详细使用说明，可访问其官方链接：

```
https://gitlab.haskell.org/haskell/ghcup-hs
```

安装 stack

在浏览器中打开如下链接:

```
https://www.haskellstack.org/
```

按照指示，我们需要在终端中运行如下命令进行 stack 的安装：

```
curl -sSL https://get.haskellstack.org/ | sh
```

关于 stack 的使用方式，我们稍后会进行简要说明.

如果你已经成功的安装了 ghcup 和 stack，真心地祝贺你．走到这一步，太不容易了!

ghc 的使用

命令 ghc 的主要作用就是把一个合法的 Haskell 程序转换/编译为当前计算机上的可运行程序. 我们以上文介绍的 Hello, World! 程序为例, 简要说明 ghc 的使用方式.

首先,我们使用一个顺手的程序源文件编辑器[23],把 Hello, World! 程序的源代码书写到一个文件中. 下面展示了这个文件中的源代码:

```
-- This is my first Haskell program
module Main(main) where

main :: IO()
main = do
  putStrLn "Hello, World!"
```

源代码中的第一行是一种新的语法成分: 注释 (Comment). 注释的主要目的是在程序合适的地方添加一些文字说明, 方便自己或他人快速理解程序的功能或相关信息. Haskell 中的注释分为两类: 单行注释、多行注释. 单行注释以两个连续的符号 --开始, 作用范围一直到当前行的末尾. 多行注释[24]起始于两个连续字符 {-, 终止于两个连续字符 -}.

把存放了上述源代码的文件命名为 `Main.hs`；然后，把这个文件放在计算机文件系统的一个文件夹下．接下来，打开计算机上的**终端** (Terminal) 应用[25]；把终端的当前目录设置为存放了 `Main.hs` 的那个文件夹[26]．接下来，进入正题．在终端中输入如下命令，并回车．

```
ghc Main.hs
```

如果一切顺利，你会在终端中看到如下信息：

```
> ghc Main.hs
[1 of 1] Compiling Main  ( Main.hs, Main.o )
Linking Main ...
>
```

这表明，Main.hs已经被成功地编译为可执行程序．在终端中输入 ls 命令并回车，终端又出现了一些新的信息：

```
> ghc Main.hs
[1 of 1] Compiling Main  ( Main.hs, Main.o )
Linking Main ...
> ls
Main    Main.hi     Main.hs   Main.o
>
```

可以看到，在当前文件夹下，多出了三个新的文件．其中，文件 Main[27] 就是编译器最终输出的可执行文件．然后，在终端中输入 ./Main 命令并回车，就会执行刚刚编译形成的可执行文件．此时，终端中的信息变为：

```
> ghc Main.hs
[1 of 1] Compiling Main  ( Main.hs, Main.o )
Linking Main ...
> ls
Main    Main.hi     Main.hs   Main.o
> ./Main
Hello, World!
>
```

是的，你要求程序输出字符串 `Hello, World!`，它也确实做到了．对于第一次接触程序设计语言的同学而言，这是一个具有历史意义的时刻．这是人类的一大步，却只是个体的一小步．许多年之后，面对未名湖边随风摇曳的垂柳，你将会回想起，费尽千辛万苦终于成功运行了这个无聊程序的那个遥远的夜晚．

动手练一练

请把前文介绍的那个更有交互性的 Haskell 程序用 ghc 命令编译为可执行程序，运行该程序，观察程序和你的交互过程.

关于 ghc 命令的详细使用说明，可访问其官方链接[28]:

```
https://downloads.haskell.org/ghc/latest/docs/html/users_guide/ghc.html
```

28: 没事不要打开这个链接. 打开了也看懂. 你需要在学习过编译原理相关的知识后，再来看一看.

ghci 的使用

命令 `ghci` 的作用是启动 Haskell 程序的一种交互式运行环境．在终端应用中输入这个命令，回车，就能进入这个交互式环境．你大概会在终端中看到如下信息：

```
> ghci
GHCi, version 8.10.7: https://www.haskell.org/ghc/  :? for help
Prelude>
```

ghci 环境是一种命令行环境．你可以在其中输入合法的 Haskell 程序表达式，环境会输出求值的结果．例如，如果输入表达式 `1 + 2`，环境会输出 `3`；如果输入表达式 `reverse [1,2,3]`，环境会输出 `[3,2,1]`．与 Haskell 模块默认加载 `Prelude` 模块类似，ghci 环境也会自动加载 `Prelude` 模块；因此，该模块中定义的各种程序元素，在 ghci 环境中都可以访问到．

你也可以在 ghci 环境中输入各种预置的命令．例如，输入 `:quit`，会退出当前的环境；输入 `:?`，则会列出该环境支持的全部命令．这些命令有一个特点：以冒号 `:` 作为首字符．另外，对于大部分命令而言，如果冒号后的内容是一个具有多个字母的单词，只输入首字母，也可以触发相同的命令．例如，命令 `:quit` 也可以简写为 `:q`．

对于很多程序员而言，ghci 环境缺省显示的命令行提示符 `Prelude>` 实在是太丑了．通常，他/她们会使用命令 `:set prompt "ghci> "` 将其修改为 `ghci>`．现在，这个世界看起来就很美了．唉，真的很难 get 到程序员的审美点．

在 ghci 环境中，你也可以手动加载其他 Haskell 模块．例如，打开终端应用，将当前目录设置为上述 Main.hs 文件所在的文件夹，然后进入 ghci 环境，在其中输入命令 `:load Main.hs`；不出意外，ghci 环境会告诉你它已经成功地加载了这个模块．在 ghci 环境中输入 main，然后环境会尝试对 main 进行求值，进而触发其中的封装的 IO 操作：打印出字符串 `Hello, World!`．对应的 ghci 环境快照如下所示：

```
ghci> :load Main.hs
[1 of 1] Compiling Main   (Main.hs, interpreted)
Ok, one module loaded.
ghci> main
Hello, World!
ghci>
```

动手练一练

请把前文介绍的快速排序函数 `qsort` 封装在一个 Haskell 模块中；在 ghci 环境中加载这个模块；然后，在 ghci 环境中对 `qsort` 函数的正确性进行测试（即：把这个函数作用到若干序列数据上，观察函数的返回值是否符合预期）.

关于 `ghci` 命令的详细使用说明，请访问其官方链接:

`https://downloads.haskell.org/ghc/latest/docs/html/users_guide/ghci.html`

stack 的使用

上述两个命令 ghc 和 ghci 适合做一些小打小闹的事情，比如：学习 Haskell 语言、编写一个小规模的 Haskell 程序等．其中，ghci 可以作为一种入门级的程序调试环境．

真实的软件开发实践可能与你的想象并不相同，正如很多人对大学生活的想象一样．软件开发是一种面向群体的活动．也就是说，通常会有一个规模或大或小的开发者群体参与到软件开发中．有同学可能会说:“我就一个人开发一个复杂的软件应用，不可以吗?”当然可以啊，正如一个建筑工人也可以独立建造一个摩天大楼一样，只要给他/她足够的时间．因为有一个群体参与其中，软件开发还面临各种复杂的管理问题，包括：人力资源的管理、需求管理、软件制品的管理、编译环境的管理、开发进度的管理等．工欲善其事，必先利其器．需要采用合适的工具应对这些管理问题．

stack 的使用

stack 是一种面向 Haskell 程序开发的构建管理工具．其管理内容覆盖代码组织方式、编译器版本及编译参数、外部依赖关系、测试等方面．下面，我们基于 stack 的官方使用说明[30]，对它进行简要的介绍．

stack new

使用 stack new 命令，可以创建一个具有特定名称的软件开发项目，其中包含一个 Haskell 包（package）[31]．Package 这个概念在 Haskell 语言规范中并不存在，但在实际软件开发中得到了广泛应用．在逻辑上，一个 package 包含一组相关的 Haskell 模块．例如，你可以把一个完整的 Haskell 程序打包为一个 package，其中包含一个 Main 模块、若干个被 Main 模块加载的自定义模块、以及相关的测试模块．一个 package 具有一个全局唯一的名称．package 的名称由若干个单词通过连字符 - 连结在一起；每一个单词由若干字母或数字组成，且至少包含一个字母．稍等片刻，我们会介绍 stack 中的 package 管理相关的信息．

举例而言,如果我们要在一个特定的文件夹下创建一个名称为 helloworld 的项目，且指定该项目使用的模版是 new-template，那么，我们可以这么做．首先，打开终端应用，将当前目录设定为项目所在的文件夹，然后运行如下命令（确保你的计算机可以访问互联网）：

```
stack new helloworld new-template
```

如果一切顺利，当前目录中应该多了一个名称为 `helloworld` 的文件夹．从此之后，我们创建的项目的所有信息都会被存放在这个文件夹中[32]．使用命令 `cd helloworld` 进入这个文件夹．使用命令 `ls` 查看这个文件夹中的内容．其中的内容具有如下层次结构：

```
.
├── ChangeLog.md
├── LICENSE
├── README.md
├── Setup.hs
├── app
│   └── Main.hs
├── helloworld.cabal
├── package.yaml
├── src
│   └── Lib.hs
├── stack.yaml
├── stack.yaml.lock
└── test
    └── Spec.hs
```

上述文件包含了一个完整的 Haskell 程序．为了运行这个程序，我们需要首先对他进行构建（build）[33]．

stack build

使用命令 `stack build` 对当前项目进行构建．如果你是第一次使用这个命令，还会自动从互联网上下载 GHC 编译器．虽然我们已经在系统中安装了 GHC 编译器，stack 工具仍然会下载一个供自己单独使用的 GHC 编译器[34]．

构建的结果存放在当前目录下一个名称为 `.stack-work` 的隐藏文件夹中．你可以用 `cd .stack-work` 进入到这个文件夹中查看相关信息；如果你这样做了，在继续下面的活动前，使用命令 `cd ..` 返回到上层目录．

stack exec

使用命令 `stack exec helloworld-exe` 运行上述构建活动输出的可运行程序. 此时, 可以看到如下的快照:

```
> stack exec helloworld-exe
someFunc
>
```

这个 Haskell 程序实现的功能很简单: 在终端打印出一个字符串.

stack test

使用命令 stack test 可以触发对当前项目的测试．测试是任何软件开发项目不可缺少的一个环节．stack 已经帮助我们建立了一个空的测试程序．我们需要根据项目的实际内容向其中填写相应的测试代码．例如，如果你自己编写了一个排序函数，为了确保功能的正确性，你需要在若干种具有代表性的数据上测试排序函数的输出是否符合你的预期．只要把这些测试数据按照规定的方式写在特定的文件中，stack test 命令就会自动执行对应的测试活动，并给出测试结果．

stack 的创建的文件

- 其中的 `LICENSE`、`README.md`、`ChangeLog.md` 三个文件，不会参与到编译活动中，因此不会对生成的可执行程序产生直接影响．第一个文件声明了当前项目版权相关的信息，这是一个文本文件．第二个文件是对当前项目的简要说明；扩展名 `md` 表明该文件是采用 markdown 语法书写的一个文本文件．第三个记录了当前项目在不同版本中发生的变更情况．
- 其中的 `helloworld.cabal` 和 `Setup.hs` 两个文件，是更底层的构建工具 `cabal` 相关的两个文件；我们无需去手工修改它们．所以，也不用关注它们．

- 文件 `stack.yaml` 记录了项目级别的一些参数设置. 扩展名 `yaml` 表示这是一个采用 YAML[35] 语法书写的文本文件. 这个文件目前仅包含两个配置项: 一个配置项为 `packages`, 它的值为点符号 `.`, 表示当前项目中仅包含一个 package, 它就训在于文件 `stack.yaml` 所在文件夹中; 另一个配置项为 `resolver`, 它的值为一个 URL, 指向互联网上的一个 yaml 文件, 其中指明了当前项目使用的 GHC 版本以及一些可用的外部 package.

```
32    packages:
33    - .
```

```
20  resolver:
21    url: https://raw.githubusercontent.com/commercialhaskell/stackage-snapshots/master/lts/18/9.yaml
```

- 文件 `package.yaml` 记录了当前 package 的很多配置项. 我们不再一一列举. 其中, 可以看到很多信息: 当前项目的可执行文件放在文件夹 `app` 中, 对应的 `main` 元素位于文件 `Main.hs` 中; 当前项目包含的模块放在文件夹 `src` 中; 当前项目的测试程序放在文件夹 `test` 中, 对应的 `main` 元素位于文件 `Spec.hs` 中.

```yaml
name:                helloworld
version:             0.1.0.0
github:              "githubuser/helloworld"
license:             BSD3
author:              "Author name here"
maintainer:          "example@example.com"
copyright:           "2021 Author name here"
```

```yaml
library:
  source-dirs: src
```

```yaml
executables:
  helloworld-exe:
    main:                Main.hs
    source-dirs:         app
    ghc-options:
```

```yaml
tests:
  helloworld-test:
    main:                Spec.hs
    source-dirs:         test
```

stack 为我们创建的三个 hs 文件

文件 app/Main.hs 的内容如下所示:

```
00 module Main where
01
02  import Lib
03
04  main :: IO ()
05  main = someFunc
```

模块 Lib 存放在 src/Lib.hs 中，其内容如下所示:

```
00 module Lib
01    ( someFunc
02    ) where
03
04 someFunc :: IO ()
05 someFunc = putStrLn "someFunc"
```

对于这个模块定义，实在无话可说.

文件 `test/Spec.hs` 的内容如下所示:

```
00 main :: IO ()
00 main = putStrLn "Test suite not yet implemented"
```

继续无话.

stack 的使用

我想，你大概明白了 `stack new helloworld new-template` 做了啥吧？它用一个预定义的项目模版帮我们创建了一个 Haskell 程序的骨架以及编译和运行环境．而所有的这一切，stack 都为我们进行了很好的封装，使得我们只需要使用 stack 提供的几个命令就能对一个软件开发项目进行便捷的管理．

动手练一练

请使用 stack 创建一个名为 `qsort` 的项目．然后，在 `src/Lib.hs` 添加并输出前面介绍的 `qsort` 函数;在 `app/Main.hs` 中加载 `Lib` 模块，随便找几个待排序的序列数据，用`qsort` 函数对它们进行排序，打印出排序的结果)．

基于 stack 的 package 管理

有人说，他站在了巨人的肩膀上，看到了很远的地方．此言确实不虚，在软件开发中也是如此．

在真实的软件开发项目中，很少有开发者从零开始编写所有的软件代码，而总是尽可能复用其他开发者已经开发完成的功能模块．例如，前面我们看到的 `Prelude` 模块就是 Haskell 语言自身提供的一个模块．除此之外，Haskell 语言还提供了一些其他模块；具体信息可参见 Haskell 语言官方规范．Haskell 语言也提供了 `import` 语句来支持对其他模块的复用．

但是，事情到此并没有结束．开发者群体是一个乐于分享的群体：有很多程序员耗费了大量的精力，开发出很多高质量的软件模块，然后把这些模块放在互联网，供其他开发者免费使用；然后，其他开发者在前人开发的模块的基础上又开发出新的模块，并共享到开发者群体中；长此以往，就形成了一种欣欣向荣的生态系统．在这个生态系统中，丰富多样的软件模块不断涌现，持续演化，就像自然界生态系统所展现出的物种的多样性和持续演化那样．

这种乐于分享的特点在 Haskell 开发者群体中也是存在的，也在此基础上形成了欣欣向荣的生态系统．在这个生态系统中，开发者分享工作成果的基本单位是 package，也即：一个开发者把一组相关的 Haskell 模块封装为一个 package，然后将其发布到互联网上．

你可能会问：分享工作成果的基本单位为什么不能是模块呢？其实，你把一个模块单独封装为一个 package 也是可以的．在更一般意义上，不以模块作为基本发布单位的主要原因如下：

- 模块不存在版本的概念．在软件开发生态系统中，演化是一种常态．缺失了版本的概念，使得我们不能对同一个模块的不同版本进行有效管理．
- 在很多场景下，模块过于细粒度．例如，如果你要对外发布一个复杂的 Haskell 应用程序，以模块为基本单元显然是不合适的．
- 当你对外发布一个模块时，为了使得其他开发者对于这个模块的质量有足够的信息，你可能还需要将该模块的测试数据和程序一起对外发布．此时，将一个模块以及附带的测试模块打包为一个 package，具合理性．

基于 stack 的 package 管理

首先注意一点：使用 `stack new` 命令创建的 Haskell 软件开发项目，其中就包含了一或多个 package．这些 package 的存放目录记录在 `stack.yaml` 文件配置项 `packages` 的值中．例如，在我们上面使用 `stack new` 命令创建的 helloworld 项目中，`packages` 下面只包含一个值，即：点符号．这表明，在项目所在的文件夹中存在一个 package．

在 stack 管理的软件开发项目中，每一个 package 的相关信息记录在一个名为 `package.yaml` 的文件中. 在这个文件中, 除了包含关于当前 package 的名称、版本、版权声明、开发者等基本信息外, 还包含一个重要的配置项 `dependencies`. 其中记录了当前 package 依赖的所有其他 package 的名称与版本信息. 例如，在上面 helloworld 项目包含的唯一一个 package 的 `package.yaml` 文件中，配置项 `dependencies` 包含一个值: `base >= 4.7 && < 5`. 这表明，当前 package 依赖于一个名称为 base 的 package，且要求 base 的版本在区间 $[4.7, 5)$ 中[36]. 紧接着的一个问题是：如何获得这个名称为 base 的特定版本的 package 呢?

```
22    dependencies:
23    - base >= 4.7 && < 5
```

37: https://hackage.haskell.org/

Haskell 开发者社区维护了一个 package 仓库[37]，并将其命名为 Hackage．任何一个开发者都可以向这个仓库中发布自己开发的 package，也可以从这个仓库中下载特定名称和特定版本的 package．当你访问这个仓库，搜索名称为 base 的 package；在这个 package 页面上，你可以看到它的所有版本，以及每一个版本中包含的所有模块．穿行在长长的模块列表中，你会看到两个熟悉的名字：`Prelude` 和 `Numeric.Natural`．这两个模块已经包含在了 base 中，因此，在你自己程序中，就可以使用 `import` 语句加载这两个模块了．

你可以在一个 `package.yaml` 文件的 `dependencies` 配置项中添加更多的 package 名称以及对应的版本需求. 然后, 在使用 `stack build` 命令时，stack 就会自动到 Hackage 仓库中下载对应 package. 如果你不相信，就试试下面的练习吧.

动手练一练

Hackage 中有一个名称为 random 的 package，其中包含一个名称为 `System.Random` 的模块，这个模块中定义了一个名称为 `randomIO` 的类型．然后，在 `do` 后面的代码块中，使用下面的语句，

```
rnd <- randomIO :: IO Int
```

就能得到一个随机生成的整数．

请你使用 stack 创建一个名称为 random-num 的项目，在 `package.yaml` 文件的 `dependencies` 下添加一个值：`random == 1.2.0`．这个值的含义是：当前 package 依赖一个名称为 random、版本为 1.2.0 的 package．然后，在当前项目中实现在终端打印出一个随机数的功能．请特别注意，当你使用 `stack new` 命令后，终端的输出信息．

需要指出的是，在主流的程序设计语言开发社区中，都存在类似的 package 管理方式，即：一个被开发者广泛认同的 package 仓库、一个配套的命令行工具（从仓库自动下载 package）．这是在互联网时代形成的群体软件开发模式，可能会陪伴你很长的时间．选择一个开发者社区，选择一个有价值的软件开发项目，努力成为项目的核心贡献者，你会收获很多很多．

关于 stack，暂且讲到这里吧．有兴趣的同学可自行阅读相关材料．

Haskell 程序的书写

Haskell 源程序的书写风格

对于学习过 C、C++、或 Java 语言的同学而言，可能会觉得 Haskell 程序的书写有些奇怪．在这三种语言中，源程序中会出现大量的分号 ；和花括号对 {}．前者的作用是作为一条语句的终结符；后者的作用是把几条语句封装为一个代码块（Code Block）．但是，在前文出现的 Haskell 程序中，从来没有看到过花括号和分号．

其实，你误解 Haskell 了．Haskell 语言规定，在 `where`、`let`、`do`、`of` 四个关键词后面需要放置一个代码块．在代码块的书写上，Haskell 提供了两种书写风格．第一种即是我们在前文中看到的书写风格：利用代码行的缩进表示语句的结束或代码块的结束．第二种即是类似 C、C++、或 Java 语言的书写风格：利用分号表示语句的结束，利用花括号对封装代码块．在官方规范中，这两种风格分别被命名为 layout-insensitive 和 layout-sensitive．

layout-sensitive

```
00 module Main(main) where
01  import Prelude
02
03  main :: IO()
04  main = do
05    putStrLn "Please input your name:"
06    name <- getLine
07    putStrLn $ "Hello, " ++ name
08    putStrLn "Please input an integer:"
09    str1 <- getLine
10    putStrLn "Please input another integer:"
11    str2 <- getLine
12    let int1 = (read str1 :: Integer)
13    let int2 = (read str2 :: Integer)
14    putStrLn $ str1 ++ " + " ++ str2 ++ " = "
15                       ++ (show $ int1 + int2)
```

layout-insensitive

```
00 module Main(main) where {
01   import Prelude;
02
03   main :: IO ();
04   main = do {
05     putStrLn "Please input your name:";
06     name <- getLine;
07     putStrLn $ "Hello, " ++ name;
08     putStrLn "Please input an integer:";
09     str1 <- getLine;
10     putStrLn "Please input another integer:";
11     str2 <- getLine;
12     let {int1 = (read str1 :: Integer);};
13     let {int2 = (read str2 :: Integer);};
14     putStrLn $ str1 ++ " + " ++ str2 ++ " = "
15                        ++ (show $ int1 + int2);
16   }
17 }
```

在采用 layout-sensitive 风格书写程序时，需要如何确定一行 代码的缩进

请记住三条朦胧的规则

1.相同缩进，开始一条新语句

2.更多缩进，继续上一条语句

3.更少缩进，代码块结束

Haskell 源文件的书写方式

源程序需要存放在对应的源文件中．在这种对应关系中，存在两种不同的书写方式．第一种书写方式就是我们已经看到的：把 layout-sensitive 或 layout-insensitive 风格的程序直接存放到文件中；此时，文件的扩展名为 hs．另一种书写方式，其文件的扩展名为 lhs．在这种书写方式中，注释和其他源代码的地位发生了调换：书写注释时，不需要使用前缀 -- 或起始/终止字符串 {- / -}；书写其他源代码时，每一行开始必须添加符号 >．

```
00 -- This is my first Haskell program
01 -- Stored in file: Main.hs
02 module Main(main) where
03
04   main :: IO()
05   main = do
06     putStrLn "Hello, World!"
07 -- This is the end of my first Haskell program
```

```
00 This is my first Haskell program
01 Stored in file: Main.lhs
02
03 > module Main(main) where
04 >
05 >  main :: IO()
06 >  main = do
07 >    putStrLn "Hello, World!"
08
09 This is the end of my first Haskell program
```

lhs 文件的书写存在一个硬性的条件：以符号 > 开始的代码行与注释之间至少存在一个空行.

为什么要发明这种书写方式呢？这个问题，你自己慢慢体会吧.

本章内容回顾

2.1 使用 Haskell 语言定义函数 . . . . .
2.1.1 逻辑运算函数 . . . . . .
2.1.2 整数的算术运算 . . . . . .
2.1.3 自然数相关的函数 . . . . . .
2.1.4 自然数上的 fold 函数 . . . .
2.1.5 序列以及序列上的 fold 函数
2.1.6 一种快速排序算法 . . . . . .
2.2 标识符和操作符的命名规则 . . . . .
2.3 Hello, World! . . . . . .
2.4 Haskell 程序的编译、运行与管理 . .
2.4.1 工具的安装 . . . . . .
2.4.2 ghc . . . . . .
2.4.3 ghci . . . . . .
2.4.4 stack . . . . . .
2.5 Haskell 程序的书写 . . . . . .
2.5.1 Haskell 源程序的书写风格 .
2.5.2 Haskell 源文件的书写方式 .

作业 04

小明同学学习了这么多Haskell语言的知识后，觉得很累；于是，他想用Haskell语言编写一个简单的命令行游戏让自己放松一下。这个游戏描述如下：

A. 系统随机生成一个1～100之间的整数，记为 x
B. 在命令行中提示用户输入一个整数
C. 接收用户输入的整数，记为 x’
D. 如果 x’ < x，提示用户他/她输入的值比真实值小，跳转到 B
E. 如果 x’ > x，提示用户他/她输入的值比真实值大，跳转到 B
F. 如果 x’ == x，提示用户他/她成功了，游戏结束。

小明同学太累了，所以想请你帮他写一个这样的程序。你觉得这个事情可行吗?

1. 请尝试编写一个这样的程序。
2. 如果你发现这个事情有困难，请告诉我们：A. 你的求解思路是什么（多种思路也可以）？ B. 在按照一个思路前进的过程中，遇到了什么困难，使得你无法继续走下去。

初见Haskell

暂且先到这里吧