第 03 章：初见 Haskell

1. 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

逻辑非 (not) 函数的第 1 种定义方式：

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

相关信息说明如下：

- Haskell 缺省加载的模块 Prelude 中已经存在了 not 函数的定义
  - 为了测试你写的这个函数是否正确，最简便的方式把这个函数换一个名字
- 在 Haskell 中：
  - `Bool` 布尔类型
  - `True False` 布尔类型的两个值
  - `not :: Bool -> Bool` not 的类型是 Bool -> Bool
    - 问题：为什么不用单个冒号(:)作为类型声明符？
    - 回答：在 Haskell 中，单冒号另有它同；
      目前看来，“采用双冒号作为类型声明符”这个设计决策并不好
  - `not True = False` 把 not True 定义为 False
  - `not False = True` 把 not False 定义为 True

not 函数的第 2 种定义方式 (conditional expression；条件表达式)：

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

相关信息说明如下：

- 在 Haskell 语言中，= 和 == 具有完全不同的含义
  - = 表示“定义为”，即：将其左侧的表达式定义为右侧的表达式
  - == 是一个逻辑运算符，计算左右两侧表达式的值是否相等

not 函数的第 3 种定义方式（guarded equations）：

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

或者：

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

逻辑与（and）函数的第 1 种定义方式（pattern matching）：

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

相关信息说明如下：

- 在 Haskell 中，运算符 -> 满足右结合律
  - 也即：Bool -> Bool -> Bool 等价于 Bool -> (Bool -> Bool)
  - 思考：(Bool -> Bool) -> Bool 这个类型的含义是什么？
- 这个定义具有显而易见的繁琐感；因此，请看下面一种更简洁的定义

逻辑与（and）函数的第 2 种定义方式（pattern matching + wildcard）：

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

相关信息说明如下：

- 一个单独的下划线 `(_)` 是一个 `wildcard`，表示这里有一个值，但我们选择无视它

> **作业 01**
> 关于逻辑与 `(and)` 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。

更传统的一种逻辑与函数，其定义如下：

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

✧ 整数的算数运算

- 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符（`operator`）
  - 例如，我们一般不会书写 `plus(a, b)`，而是直接书写 `a + b`
- `Haskell` 提供了常用的算术运算符：
  1. `+` 加运算符
  2. `-` 减运算符
  3. `*` 乘运算符
  4. `^` 指数运算符

✧ 用算术运算符定义对应的算术运算函数

```
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
```

相关信息说明如下：

- `Integer` Prelude 模块中存在的一种整数类型，可表示任意整数值

✧ 把 二元运算符 转换为 对应的函数

Haskell 提供了一种语法，可方便地实现这种转换，

即：把一个二元操作符放置在一对圆括号中

例如，对于两个整数 x y ：

- `x + y` 等价于 `(+) x y`
- `(+)` 是一个函数，类型为 `Integer -> Integer -> Integer`

在此基础上，Haskell 还提供了一种增强语法

例如，对于两个整数 x y ：

- `(x +) y` 等价于 `x + y`
- `(+ y) x` 等价于 `x + y`

✧ 把 函数 转换为 二元运算符

Haskell 提供了一种语法，可方便地实现这种转换，

即：把一个函数放置在一对反引号 (`...`) 之间

例如，对于两个整数 x y ：

- `div x y` 等价于 `x `div` y`

✧ 整数的除运算 (Haskell 的 Prelude 中存在两种除运算)

1. 如果你需要的是整除运算，使用函数 `div`
2. 如果你需要的是结果带有小数点的除运算，使用运算符 `/`

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
```

说明：

- 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 困难在哪里？）

✧ 自然数相关的函数

✧ 自然数类型

- Haskell 语言标准库的模块 Numeric.Natural 中定义了一种自然数类型 Natural，可表示任意自然数
- 但 Haskell 缺省加载的 Prelude 模块中，不包含 Numeric.Natural 模块中的元素
- 为了在程序中使用 Natural 类型，需要在程序文件的开始处添加如下语句：

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

  其含义是：把 Numeric.Natural 模块中的元素 Natural 引入到当前文件中

✧ 阶乘函数

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

相关信息说明如下：

- 这是一个采用 pattern matching 进行定义的函数，其含义如下：
  - 对于一个自然数 n，
    - 如果 n == 0，则 fact n = 1
    - 否则，fact n = n * fact (n - 1)
- 在程序运行过程中，如果要评估表达式 fact x 的值，会按照定义中给出顺序进行模式匹配。具体而言：
  - 首先检查 x 是否可以匹配到 0 上：如果匹配成功，则将 fact x 评估为 0；

否则，继续匹配

- 继续匹配 `x` 是否可以匹配到 通配符 `n` 上：因为 `n` 是通配符，可以匹配到任何自然数上，因此，这次匹配一定成功，从而将 `fact x` 评估为 `x * fact (x - 1)`

- 在 Haskell 中，function application（函数调用）具有最高优先级

作业 03

关于阶乘函数，你还能想到其他定义方式吗？请分别使用 “guarded equations” 和 “conditional expression” 写出阶乘函数的定义。

✧ 斐波那契函数

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

说明：

- 这个定义很简洁，但运行效率很低

✧ 自然数上的 `fold` 函数

```
fold :: (t -> t) -> t -> Natural -> t
fold h  c  0 = c
fold h  c  n = h (fold h c (n - 1))
```

说明：

- 在 `fold` 的类型中，小写字母 `t` 表示一个类型变量，`Natural` 表示一个具体类型

如何确定函数类型声明中出现的一个名称
是一个具体类型，还是一个类型变量呢？

对于该问题
Haskell 在语法层次上给出了一种简单有效的解决方案

- 如果名称的 首字符 是 小写字母，则表示 类型变量
- 如果名称的 首字符 是 大写字母，则表示 具体类型

- 在表达式 `h (fold h c (n - 1))` 中出现的圆括号，不是主流编程语言

(C/C++/Java/Rust) 中函数调用时用于传参的圆括号，而是用于调整运算顺序的圆括号。

- 如果不加这些括号，则 `h fold h c n - 1`

  等价于 `((((h fold) h) c) n) - 1`

- 在 Haskell 中：

  1. 函数调用具有最高优先级（比运算符的优先级高）
  2. 函数调用满足左结合律

- 如果你对这种调整运算顺序的括号很反感，Haskell 提供了另一种方案：

  二元运算符 `$`

  该运算符：具有最低优先级；满足右结合率

  其作用是：把其左侧的函数作用到右侧的元素上

  因此，如下三个表达式等价：

```
h  (fold h c  (n - 1))
h $ fold h c  (n - 1)
h $ fold h c $ n - 1
```

✧ 阶乘函数

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

snd :: (a, b) -> b
snd _ y = y

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

fact :: Natural -> Natural
fact = snd . (fold f (0, 1))
```

说明：

- dot 运算符 (.) 具有函数组合的功能
  - 给定两个函数 `f g` 以及一个合法的表达式 `f (g x)`
  - 则 `f (g x)` 等价于 `(f . g) x`

- 显然可知：
  - `(f . g) x` 等价于 `f . g $ x`
  - `f . g x` 等价于 `f . (g x)`

✧ 斐波那契函数

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

fib :: Natural -> Natural
fib = fst . (fold g (0, 1))
```

✧ Haskell 中的序列类型

- 给定一个类型 a，其对用的序列类型书写为 [a]
- 序列的基本表示符号（以 [Natural] 为例）
  - 空序列：[]
  - 包含一个自然数 1 的序列：[1] 或者 1 : []
    - 注意：这里的单冒号 (:) 是一个二元运算符
  - 包含三个自然数 1 2 3 的序列：[1, 2, 3] 或者 1 : 2 : 3 : []

- 序列上的若干函数

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

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
```

```
rev :: [a] -> [a]
rev = revm [] where
  revm :: [a] -> [a] -> [a]
  revm xs [] = xs
  revm xs (y:ys) = revm (y:xs) ys
```

✧ 序列类型上的 fold 函数

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

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

✧ 使用 fold 函数对前面 4 个函数重新定义

重定义	原定义

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

`concat :: [a] -> [a] -> [a]`
`concat xs ys = foldr (:) ys xs`
	`concat :: [a] -> [a] -> [a]`
	`concat [] ns = ns`
	`concat (m:ms) ns = m : concat ms ns`

`filter :: (a -> Bool) -> [a] -> [a]`
`filter p = foldr (k p) [] where`
`  k :: (a -> Bool) -> a -> [a] -> [a]`
`  k p x | p x = (x:)`
`  | otherwise = id`
`  id :: a -> a`
	`filter :: (a -> Bool) -> [a] -> [a]`
	`filter p [] = []`
	`filter p (n:ns)`
	`  | p n  =  n : filter p ns`
	`  | otherwise  =  filter p ns`

`id x = x`	

`rev :: [a] -> [a]`
`rev = foldl (flip (:)) []`
	`rev :: [a] -> [a]`
	`rev = revm [] where`
	`  revm :: [a] -> [a] -> [a]`
	`  revm xs [] = xs`
	`  revm xs (y:ys) = revm (y:xs) ys`

✧ 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 qsort 的定义更加结构化。

一种语法机制是： `let ... in ...` 表达式

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

说明：

○ 在 `in` 后面的这个表达式中，可以访问 `let ... in` 之间定义的变量

另一种语法机制是： `where` 子句

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

✧ `let in` 表达式 VS `where` 子句

○ 在大部分情况下，两者没有本质的区别，仅仅反映了不同的表现形式

○ 在一些情况下，where 子句定义的变量具有更大的作用范围

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

说明：

- 在 where 子句中定义的一个变量 z，可在 guarded equations 的任何地方访问
- let in 表达式不具有这样的能力

<table>
<tr>
<td rowspan="2">  </td>
<td>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）
把一个非常难于理解的算法变得更加容易理解了</td>
<td rowspan="2">  </td>
</tr>
<tr>
<td>我再观察一下；
如果你继续搞这些朝三暮四的小把戏，我就准备 ... 了</td>
</tr>
</table>

✧ 好的程序设计语言应该具有一种基本性质：

- 用这种语言写出的程序具有易理解性

✧ 但是，程序的易理解性不仅仅是程序自身的性质，

而是与试图理解程序的主体（对，就是你）具有密切的关系

- 你必须深刻理解函数式思维的特点，

  才有可能轻松理解函数式程序，也才能写出体现函数式思维的优雅程序

2. 标识符和运算符的命名规则

✧ 在很多情况下，我们需要为程序中定义的元素命名

- 所谓命名，就是给一个东西赋予一个具有区分作用的名称
- 命名的作用：通过名称引用到所指向的那个程序元素

✧ Haskell 中的名称，分为两大类：

1. 标识符 (Identifier)
2. 运算符 (Operator Symbol)

✧ 标识符的命名规则：

1. 由 1 或多个字符顺序构成
2. 首字符只能是一个字母 (letter)，具体包括
   - ASCII 编码表中的所有字母 (即：所有英文大小写字母)
   - Unicode 字符集中的所有字母
3. 其他字符只能是 字母 / 数字 / 英文下划线 / 英文单引号
4. 不能与 Haskell 的保留词重名
   - 这些保留词包括：`case class data default deriving do else foreign if import in infix infixl infixr instance let module newtype of then type where _`
5. 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
   a. 一些程序元素，其 标识符首字符 只能是 大写字母
   b. 其他程序元素，其 标识符首字符 只能是 小写字母

> 目前已经涉及到的程序元素包括：
> - 函数 / 变量 / 类型变量：名称首字符必须是小写字母
> - 类型：名称首字符必须是大写字母
>
> 更多信息，会在涉及到相关程序元素时按需说明

✧ 运算符的命名规则：

1. 由 1 或多个符号 (symbol) 顺序构成
   - ASCII 编码表中的所有符号：`! # $ % & * + . / < = > ? @ \ ^ | - ~ :`
   - Unicode 字符集中的大部分符号：...
2. 不能与 Haskell 的保留运算符重名
   - `..  :  ::  =  \  |  <-  ->  @  ~  =>`
3. Haskell 进一步将运算符分为两类
   - (1) 以英文冒号(:)为首字符的运算符；(2) 其他运算符
   - 更新信息，会按需说明

3. Hello, World!

✧ Haskell 中的 Hello, World! 程序

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

说明：

- 这是一个合法的 Haskell 程序
- 但是，在符合 Haskell 规范的前提下，它隐藏了一些代码，以至于看起来略显奇怪
- 恢复这些被删除代码后，会得到如下更为完整的程序：

```
01  module Main(main) where
02  import Prelude
03
04  main :: IO ()
05  main = do
06    putStrLn "Hello, World!"
```

说明：

- 这个程序定义了一个模块 (module)
- 这个模块的名称为 Main
- 这个模块对外输出了 (export) 一个名为 main 的元素
- where 后对 Main 模块中存在的元素进行了定义

	为什么 main 的类型不是函数呢？ C/C++/Java/Rust 语言的 main 都是函数	
	因为 main 本来就不是函数 你在哪本数学书上见到过 main 这样的函数？ 哪有函数一言不合就向控制台输出字符串的呢？	
	既然如此，C/C++等语言为什么把 main 作为函数呢	
	我想，它们大概是为了让自己显得很 NB 任何东西，能和数学沾上关系，就会显得高大上了 例如，有的公司举办了全球数学竞赛，又不公布成绩，...	

✧ 关于模块 (Module)，Haskell 语言规范中给出了如下信息：

1. 一个 Haskell 程序由 1 或多个模块构成，且每一个模块定义在一个单独的文件中
2. 一个 Haskell 程序必须包含一个名为 Main 的模块

- Main 模块必须输出一个名为 main 的程序元素
- main 元素的类型必须是 IO t，其中
  - t 是一个类型变量；

    在定义一个具体的 main 元素时，需将其替换为一个具体类型
  - IO 是 Prelude 中定义的一个元素，用于封装 IO 运算
- 一个 Haskell 程序的运行就是对 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

3. 模块的名称必须满足如下两个条件之一
   - 1 个以大写字母开头的标识符
     - 例如：Mymodule
   - 2 或多个以大写字母开头的标识符，通过 dot 字符 (.) 连接在一起
     - 例如：This.Is.Mymodule
4. 若一个模块在设计时就已经确定不会被其他模块所引用

   那么，该模块可以放在任意一个具有合法名称的文件中
   - 通常，Haskell 程序的 Main 模块不会被其他模块所引用

     因此，可以把 Main 模块放在任意一个文件中
   - 但是，将 Main 模块所在文件名设定为 Main，不失为一个好选择
5. 若一个模块可能会被其他模块所引用，

   那么，该模块所在的文件必须满足如下条件：
   - 若模块名是一个标识符，则：模块所在文件的名称必须与模块名相同
   - 若模块名是多个标识符通过 dot 字符连接在一起，则：
     1. 模块所在文件的名称必须与模块名中最后的标识符相同

        例如，模块 This.Is.Mymodule 必须放置在一个名为 Mymodule 的文件中

        (这里的文件名，不包含文件的扩展名)
     2. 模块名中前面的每一个标识符及其之间的顺序关系，必须与文件系统的目录名以及目录之间的包含关系存在一一对应

        例如，模块 This.Is.Mymodule 必须放置在

        源文件目录下的 This/Is/Mymodule 这个文件中

```
01
02 module Main(main) where
03 import Prelude
```

```
04
05  main :: IO ()
    main = do
06
      putStrLn "Hello, World!"
```

说明：

- 第 2 行代码是一个模块加载声明，其含义是：
  - 把 Prelude 模块对外输出的所有程序元素加载到当前模块中
- Haskell 规定：
  - 若模块源码中不存在 import Prelude，则表明其缺省存在
    - 效果：你在程序中可以直接使用 Prelude 输出的所有元素
  - 若模块源码中存在 import Prelude 或其变体，则从 Prelude 中按需加载元素
    - 例如：如果模块中存在 import Prelude(Integer, (+), (-))

      则表明只需把 Prelude 对外输出的 类型 Integer、加运算符、减运算符 三个元素加载到当前模块中；Prelude 对外输出的其他元素无需加载
    - 例如：下面的 Haskell 模块定义不合法

      ```
      01  module Main(main) where
      02  import Prelude(Integer, (+), (-))
      03
      04  main :: IO ()
      05  main = do
      06    putStrLn "Hello, World!"
      ```

      原因：
      - 在这个模块定义中，出现了两个未定义的程序元素：IO、putStrLn
      - 把它们加入到 import Prelude 后的圆括号中，才是一个合法的模块定义

```
01  module Main(main) where
02  import Prelude
03
04  main :: IO ()
05  main = do
06    putStrLn "Hello, World!"
```

说明：

- 第 4 行代码声明 main 的类型为 IO ()
  - () 是一个类型，称为 0 元组 (0-tuple) 类型
  - IO 是一个类型构造器 (type constructor)
  - IO () 是一个类型，其中封装了 IO 运算，且该运算会返回一个 0 元组

```
01  module Main(main) where
02  import Prelude
03
04  main :: IO ()
05  main = do
06    putStrLn "Hello, World!"
```

说明：

- 第 5~6 行代码定义了 main 中封装的 IO 运算
  - 这个 IO 运算中仅包含了一个 IO action，即：在控制台输出一串字符
  - 如果你愿意，可以继续添加一个 IO action:

    ```
    putStrLn "Hello, World! AGAIN"
    ```

    注意：应保持与第 6 行具有相同的缩进

    此时，main 中就封装了两个顺序执行的 IO actions

do 是什么梗？

一言难尽啊

- ✧ 在没有介绍更多的相关知识之前，无法给出 do 的准确定义
- ✧ 简而言之：do 是一种语法糖 (syntax sugar)
  - 在函数的世界里，没有“顺序执行”这个概念（这句话其实有些含糊）
  - 但是，可以用一些机制去仿真“顺序执行”
  - do 的作用就是把这些机制封装起来，让程序具有更好的易理解性

```
01
    module Main(main) where
02
```

```
03  import Prelude
04
05  main :: IO ()
06  main = do
      putStrLn "Hello, World!"
```

说明：

- putStrLn 是 Prelude 模块输出的一个程序元素，定义如下：

```
putStrLn :: String -> IO ()
putStrLn s  =  do     putStr s
                      putStr "\n"
```

- 一个具有更多交互性的程序

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

说明：

- 第 07 行：`name <- getLine`

  getLine 是 Prelude 模块输出的一个元素，其定义如下：

```
getLine :: IO String
getLine =  do     c <- getChar
                      if c == '\n' then return "" else
```

```
                                    do s <- getLine
                                           return (c:s)
```

符号 `<-` 是与 do 语法绑定的一种语法

在这行代码中，`<-` 的效果是把 `getLine` 返回的 `IO String` 类型的值中的那个 `String` 类型的值赋给 `name`

- 第 13 行：`let int1 = (read str1 :: Integer)`

  这一行代码看来既熟悉又陌生

  - 我们看到了熟悉的 `let`，却没有看到它的好伙伴 `in`
  - 这个 `let` 就是 `let in` 中的 `let`，用于定义在后面被使用的变量；

    但是，`in` 被语法糖隐藏了 (时机合适时，再介绍细节)

  `read str1 :: Integer`

  - `read` 是 `Prelude` 输出的一个函数
  - `read` 的类型：大约是 `String -> a`
  - 这一行代码的功能：把一个字符串映射/转换为一个整数值
  - 在调用 `read` 时，若无法从上下文中推断出 `a` 对应的具体类型，

    则需在其后面放置 `:: X`，显式说明 `a` 的具体类型为 `X`

- 第 15 行中的 `show $ int1 + int2`
  - `show` 是 `Prelude` 输出的一个函数
  - `show` 的类型：大约是 `a -> String`
  - `show` 的功能：把一个值映射/转换为一个字符串

	掌握了 Haskell 语言 IO 相关的操作 再加上前面介绍的 Haskell 相关知识 你应该可以做很多事情了	
	非常遗憾的是，这些程序目前还不能运行	
	不用太担心，想让程序运行，分分钟的事	
	分分钟是多久呢？	

4. Haskell 程序的编译、运行、管理

✧ 当你用自然语言写了一本小说，可以把它发表在互联网上；
然后，读者们就可以快乐地阅读这本小说了

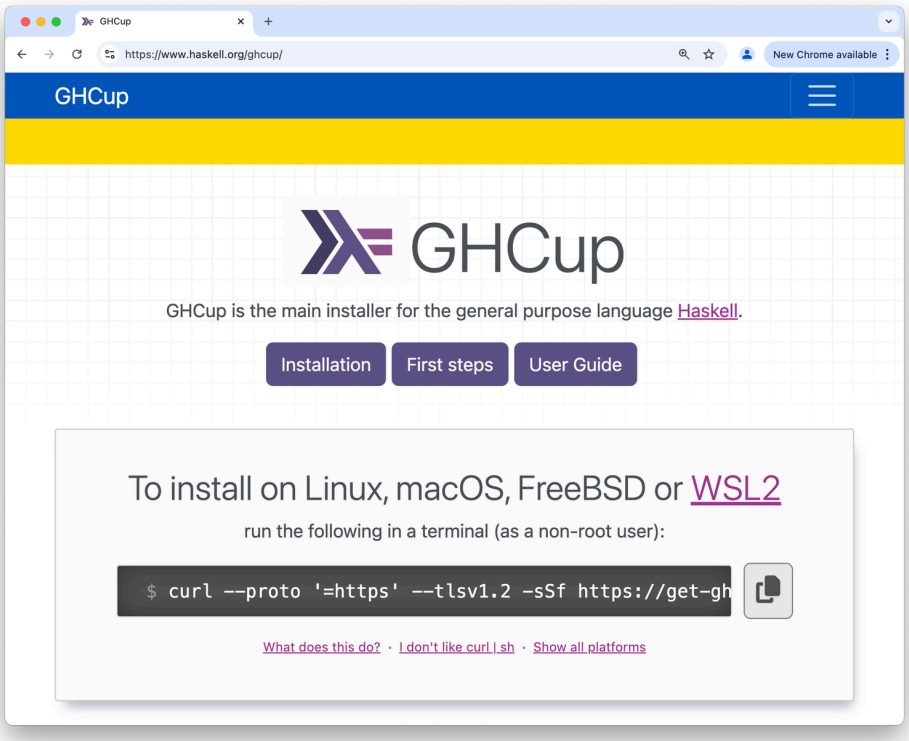
✧ 当你用 Haskell 语言写了一个程序，也可以把它发表在互联网的某个代码托管网站上；
然后，程序员们就可以阅读这个程序了

✧ 与小说不同的是，程序还有另外一类读者：计算机

- 计算机需要理解程序，
  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
- 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
- 为了让程序能够在硬件上运行，还需提供一系列软件支撑工具
- 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：
>
> 1. GHC (Glasgow Haskell Compiler)：一种得到广泛使用的 Haskell 语言编译器，能够把合法的 Haskell 程序变换计算机可执行的机器指令序列
> 2. GHCi：Haskell 程序的一种交互式 (interactive) 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
> 3. Stack：一种常用的 Haskell 软件项目构建管理工具

✧ 通过 ghcup 安装 Haskell 工具链；进入页面 https://www.haskell.org/ghcup/
按照说明，在自己的计算机上安装 Haskell 工具链
安装过程中总会遇到各种问题；遇到问题，莫慌张，主动寻求助教或其他同学的帮助



✧ ghc 的使用

> ghc 是 Haskell 语言的一种编译器 (compiler)
> 作用：把一个合法的 Haskell 程序转换/编译为在当前计算机上可运行的二进制程序

Step 1：把下面的程序放置到某个文件夹下的 Main.hs 文件

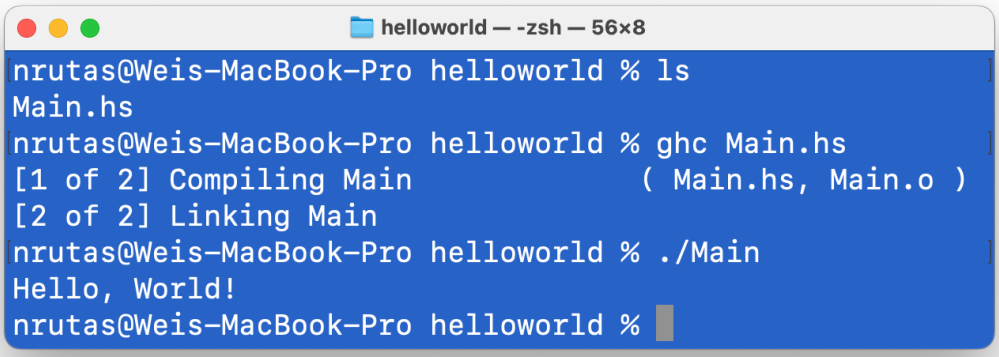
```
01  -- This is my first Haskell program
02  module Main(main) where
03  import Prelude
04
05  main :: IO ()
06  main = do
07    putStrLn "Hello, World!"
```

○ 第一行为程序注释 (comment)

Haskell 的注释分为两类：

1. 单行注释：以两个连续连字符 -- 开始的一行文字
2. 多行注释：以 {- 开始、以 -} 结束的所行文字

Step 2：打开终端（Terminal）应用，把当前目录设置为 Main.hs 所在的文件夹



- 对于第一次接触程序设计语言的同学，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这个无聊程序的那个遥远的夜晚

动手练一练 01

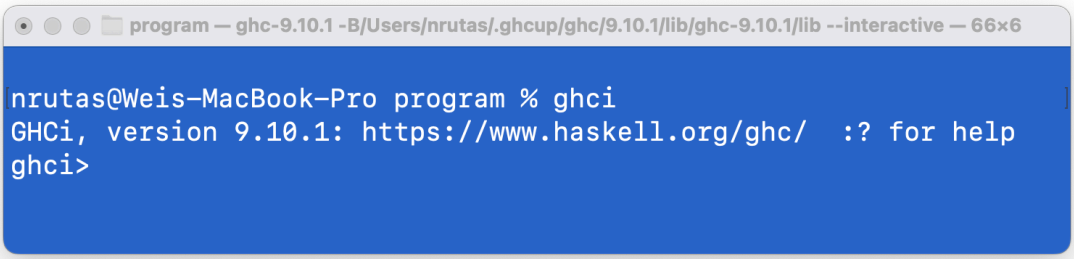
请把前文介绍的那个更有交互性的 Haskell 程序，用 ghc 命令编译为可执行程序，运行该程序，观察程序和你的交互过程。

关于 ghc 的详细使用说明，可访问其官方网站：https://www.haskell.org/ghc/

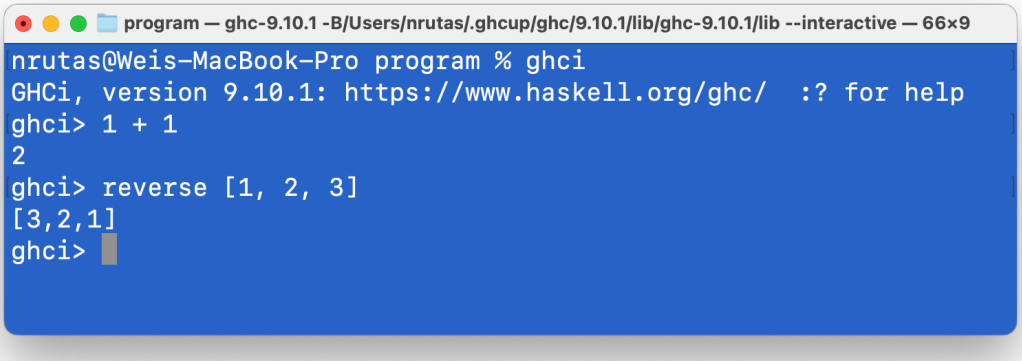
- 没事不要打开这个链接；打开了也看不懂
  你需要在学习过编译原理相关的知识后，再来看一看

✧ ghci 的使用

> ghci 是 Haskell 程序的一种交互式运行环境
> ghci 默认加载 Prelude 模块；因此，可直接使用该模块输出的元素



你可以在其中输出合法的 Haskell 表达式；ghci 会输出求值结果



ghci 中的常用命令：

- :?　　列出 ghci 支持的所有命令
- :quit　或　:q　　退出当前 ghci 环境
- :load 模块文件名　　把一个指定的模块加载到当前环境中
- :reload　　重新加载那些已经加载的模块（这些模块可能被修改了）

:load 命令使用示例

- 打开终端（Terminal）应用，把当前目录设置为 Main.hs 所在的文件夹

```
helloworld — ghc-9.10.1 -B/Users/nrutas/.ghcup/ghc/9.10.1/lib/ghc-9.10.1/lib --interactive — 71×12

nrutas@Weis-MacBook-Pro helloworld % ls
Main    Main.hi Main.hs Main.o
nrutas@Weis-MacBook-Pro helloworld % ghci
GHCi, version 9.10.1: https://www.haskell.org/ghc/  :? for help
ghci> :load Main.hs
[1 of 2] Compiling Main             ( Main.hs, interpreted )
Ok, one module loaded.
ghci> main
Hello, World!
ghci>
```

动手练一练 02

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` 中加载这个模块；

然后，在 `ghci` 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）

✧ `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

- `ghc`、`ghci` 适合做一些小打小闹的事情
  - 例如，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是一种面向群体的智力密集型活动

	我就一个人开发一个复杂的软件应用，不可以吗？	
	可以，一个建筑工人也可以独立建造一栋摩天大楼 只要给他足够的时间	
	群体软件开发还面临各种复杂的管理问题 包括：人力资源管理、需求管理、软件制品管理、编译环境管理、开发进度管理等 工欲善其事，必先利其器	

	需要采用合适的工具应对这些问题	

下面，我们基于 stack 的官方使用说明，对它进行简要的介绍

stack new 命令

- 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)
- Package 概念在语言规范中并不存在，但在实践中得到广泛应用

  在逻辑上，一个 package 包含若干相关的 Haskell 模块

  例如：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

  每个单词由若干字母或数字组成，且至少包含一个字母

如果要在一个特定的文件夹下创建一个名称为 foo 的项目，可以这么做：

1. 打开终端应用，将当前目录设定为项目所在的文件夹
2. 运行如下命令 (确保你的计算机处于联网状态)

   ```
   stack new foo
   ```

如果一切顺利，当前文件夹下会存在一个名为 foo 的文件夹

- foo 项目的所有信息都会被放在这个文件夹中

使用 cd foo 命令进入这个文件夹，可以看到其中存在的信息：

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

```
└── Spec.hs
```

stack build 命令

- 在终端的 foo 目录下输入命令 **stack build** 对 foo 项目进行构建

stack exec 命令

- 在终端的 foo 目录下输入命令 **stack exec foo-exe**
  - 然后，你就会看到 foo 项目的缺省行为
  - 如果你觉得 foo-exe 这个名字不好，可以在配置文件中修改成另外一个

stack test 命令

- 在终端的 foo 目录下输入命令 **stack test**，可以触发对当前项目的测试
  - 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 在 foo 项目中创建的文件：

- 三个文件：LICENSE / README.md / CHANGELOG.md

  这三个文件不会参与到编译活动中，不会对构建过程产生影响

  1. LICENSE　　声明当前项目版权相关的信息
  2. README.md　　对当前项目的简要说明
  3. CHANGELOG.md　　记录项目在不同版本中发生的变更情况

- 两个文件：helloworld.cabal / Setup.hs

  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，不用关注它们

- 一个文件：stack.yaml

  其中记录了两条信息：

```
packages:
- .
```

- 当前项目中包含一个 package，它就存在于 stack.yaml 所在的文件夹中

```
snapshot:
  url: https://raw.githubusercontent.com/commercialhaskell/stackage-snapshots/master/lts/22/33.yaml
```

- 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

- 一个文件：package.yaml

```
name:                foo
version:             0.1.0.0
github:              "githubuser/foo"
license:             BSD-3-Clause
author:              "Author name here"
maintainer:          "example@example.com"
copyright:           "2024 Author name here"

extra-source-files:
- README.md
- CHANGELOG.md

dependencies:
- base >= 4.7 && < 5

ghc-options:
- -Wall
- -Wcompat
- -Widentities
- -Wincomplete-record-updates
- -Wincomplete-uni-patterns
- -Wmissing-export-lists
- -Wmissing-home-modules
- -Wpartial-fields
- -Wredundant-constraints

library:
  source-dirs: src       # lib 源文件放在 package.yaml 所在文件夹下的子文件夹 src 中

executables:
  foo-exe:                    # stack exec 命令后跟的那个名字；可以被修改为其他名称
    main:           Main.hs   # main 元素定义在 Main.hs 中
    source-dirs:    app       # foo-exe 的源文件放在 app 文件夹中
```

```
    ghc-options:
    - -threaded
    - -rtsopts
    - -with-rtsopts=-N
    dependencies:
    - foo

tests:
  foo-test:
    main:                Spec.hs
    source-dirs:        test
    ghc-options:
    - -threaded
    - -rtsopts
    - -with-rtsopts=-N
    dependencies:
    - foo
```

- 三个 hs 文件：

1. app/Main.hs

```
module Main (main) where

import Lib

main :: IO ()
main = someFunc
```

2. src/Lib.hs

```
module Lib
    ( someFunc
    ) where

someFunc :: IO ()
someFunc = putStrLn "someFunc"
```

3. test/Spec.hs

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

我想，你大概明白 stack new foo 做了啥吧？ 它帮我们创建了一个 Haskell 程序的骨架以及编译和运行环境 所有的这一切，stack 都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 03

请使用 `stack` 创建一个名为 `qsort` 的项目。

然后：

1. 在 `src/Lib.hs` 中添加并输出前面介绍的 `qsort` 函数；
2. 在 `app/Main.hs` 中加载 `Lib` 模块，

   找几个待排序的数据，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

  开发者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` 还提供了一些其他模块；具体参见 `Haskell` 语言规范

  `Haskell` 也提供了 `import` 语句来支持对其他模块的复用

- 但是，事情并没有到此结束
- 软件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，不以模块作基本发布单位的主要原因如下：

1. 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.
2. 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
3. 当你对外发布一个模块时，为了使得其他开发者对该模块的质量具有足够的信心，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一个 package，具合理性.

- 首先注意一点：使用 stack new 创建的项目，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 foo 项目中，packages 下面只包含一个值，即：点符号 (.)

  这表明，在 foo 所在的文件夹 foo 中存在一个 package

- 在 stack 管理的项目中，每一个 package 的管理信息记录在一个名为 package.yaml 的文件中.

  该文件，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；它记录了当前 package 依赖的所有其他 package 的名称与版本信息

  例如，在 foo 项目包含的唯一一个 package 的 package.yaml 文件中，配置项 dependencies 信息如下：

```
dependencies:
- 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 的版本号在区间 [4.7, 5) 中。

那么，一个问题是：如何获得这个名称为 base 的特定版本的 package 呢？

- Haskell 社区维护了一个在线的 package 仓库，并将其命名为 Hackage

  https://hackage.haskell.org/

  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
- 你可以在 Hackage 中搜索名称为 base 的 package

  然后，在 base 的页面上，可以看到它的所有版本，和每一版本包含的所有模块

  在长长的模块列表中，会看到两个熟悉的名字：Prelude 和 Numeric.Natural

  因为 base 包含了这两个模块，且当前的项目依赖于 base，所以，在当前项目中，就可以使用 import 语句加载这两个模块了

- 你可以在 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，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 04

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 randomIO 的元素。

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。

请使用 stack 创建一个名称为 random-num 的项目，

并在 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 random、版本 1.2.0 的 package
- 在 Mac 的 M 系列芯片上，可能需要把这个值修改为 random >= 1.2 && < 2

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、一个配套的构建管理工具（负责从仓库自动下载 package）。

  这是在互联网时代形成的一种群体软件开发模式，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.

5. Haskell 程序的书写

- Haskell 源程序的书写风格
  - 对于学习过 C / C++ / Java 语言的同学而言，可能会觉得 Haskell 程序的书写有些奇怪
    - 在传统语言中，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，在前面出现的 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 这四个关键词后，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
  1. Layout-insensitive （布局无关）

     我们在前面看到的书写风格：利用代码行的缩进表示语句的结束或代码块的结束
  2. Layout-sensitive （布局相关）

     类似 C / C++ / Java 的书写风格：利用分号表示语句的结束，利用一对花括号形成一个代码块

- 下面是前面已经出现的一个采用 layout-insensitive 风格书写的源程序

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

- 让我们把它改写为 layout-sensitive 的风格

```
01 module Main(main) where {
02   import Prelude;
03
```

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

你更喜欢哪一种风格呢？ 由繁入简易，由简返繁难；回不去咯！	

- 问：在采用 layout-sensitive 风格书写程序时，如何确定一行代码的缩进？
- 答：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 => 结束一个代码块

✧ Haskell 源程序的书写方式

Haskell 提供了两种源程序的书写方式

1. 文件扩展名为 hs 的书写方式

   把 layout-insensitive/sensitive 风格的程序放置在扩展名为 hs 的文件中

2. 文件扩展名为 lhs 的书写方式

   注释 与 其他代码 的地位发生了变化

   - 书写注释时，不需要使用前缀 -- 或起始/终止字符串 {- / -};

- 书写其他代码时，每一行开始必须添加符号 `>`

`hs / lhs` 对比：

`-- This is my first Haskell program`
`-- Stored in file: Main.hs`
 	
`module Main(main) where`
 	
`main :: IO ()`
`main = do`
`  putStrLn "Hello, World!"`
 	
`-- This is the end.`
	`This is my first Haskell program`
	`Stored in file: Main.lhs`
	
	`> module Main(main) where`
	`>`
	`> main :: I0 ()`
	`> main = do`
	`>  putstrun "Hello, World!"`
	
	`This is the end.`

- 注意：`lhs` 文件的书写存在一个硬性的要求

  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 lhs 这种书写方式呢？

这个问题，你自己慢慢体会吧

不重要

作业 04

小明同学学习了这么多 Haskell 语言的知识后，感觉很累。

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