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 计算机学院

2022年09～12月

第2.2章：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?
  - 当一个模块中加载了两个模块，且这两个模块存在同名函数时，直接通过函数名访问该同名函数，会存在歧义
  - 为避免歧义，可在使用该函数时，加上 模块名 和 点 作为前缀

问：这里为什么是两个冒号
答：一个冒号另有它用

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

符号	含义
Bool	Haskell中的布尔类型
True	布尔类型中的真值
False	布尔类型中的假值

代码行	含义
第1行	函数not'的类型声明
第2,3行	函数not'的具体定义
第2行	该函数将True映射为False
第3行	该函数将False映射为True

not函数：另一种定义方式

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

分支表达式
conditional expression

在Haskell中，= 和 == 具有完全不同的含义	
=	“定义为”：将左侧表达式的值定义为右侧表达式的值
==	一个逻辑运算符

not函数：又一种定义方式

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

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

guarded equations
条件方程组

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

Haskell规定：在函数类型声明中，->具有右结合性

因此 `Bool -> Bool -> Bool`

等价于 `Bool -> (Bool -> Bool)`

这个定义具有显而易见的繁琐感

```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符（Operator）
- 例如：我们通常不会书写 plus(a, b)，而会书写 a + b.
- Haskell 提供了常用的算数运算符：
  - `+` ： 加运算符
  - `-` ： 减运算符
  - `*` ： 乘运算符
  - `^` ： 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
```

符号	含义
Integer	Prelude模块中存在的一种整数类型 可表示任意精度整数

二元运算符 → 对应的函数

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 等价于（+）x y

其中，函数（+）的定义如下：

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

二元运算符 → 对应的函数

还可以把一个值和一个运算符
同时放在一对圆括号中，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，其定义如下：

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

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

函数 → 二元运算符

把函数放在一对``符号中

例如：表达式 `div x y`，等价于 `x `div` y`

为什么不介绍除运算符呢?

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
Prelude模块分别提供了对应的函数 div 和操作符 /

例如:
表达式 div 5 2 或 5 `div` 2 的值为 2
表达式 5 / 2 或 (/) 5 2 的值是一个小数

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

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

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

符号	含义
Natural	Haskell的模块Numeric.Natural中 定义的一种自然数类型 可表示任意精度的自然数

缺省情况下，该模块中的任何成分
都不会被加载到当前程序中

为了在程序中使用Natural类型
可在程序开始处添加如下语句

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

- 若要同时加载Numeric.Natural中的其他元素x，可声明为: import Numeric.Natural (Natural, x)
- 若要加载Numeric.Natural中的全部元素，可声明为： import Numeric.Natural

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

❖第2和3行语句采用模式匹配（pattern matching）进行函数定义

- 对于一个自然数 n，如果 n == 0，fact n = 1
- 否则，fact n = n * fact (n - 1)

❖若程序运行时需要评估表达式 fact x 的值，会按照定义的顺序

- 首先匹配 fact 0，若成功，则评估完成；若失败，
- 继续匹配 fact n：因为n是一个通配符，可匹配到任意自然数，则匹配一定成功，…

在Haskell语言中，函数应用具有最高优先级

```
1  fact :: Natural -> Natural
2  fact 0  = 1
3  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.

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

符号	含义
第1行中的小写字母a	▸ 一个类型变量（type variable） ▸ 在调用foldn函数时，会根据实际参数的类型，确定a表示的具体类型 ▸ 如果实际参数的类型无法满足foldn的类型要求，则报错 • 例如，当传入的第一个参数的类型为Natural -> Bool，会报错

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

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

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

前面似乎提到，Haskell中的函数调用，参数前后不需要放括号
为什么这里又出现括号了呢?

这里的括号，作用是调整运算顺序

```
h foldn h c (n-1)
```

等价于

在Haskell语言中：1. 函数调用具有最高优先级
2. 函数调用具有左结合性

```
(h foldn) h c (n-1)
```

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

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：二元操作符 $

$	Haskell中具有最低优先级的操作符，且具有右结合性
	除此之外，没有任何其他效果

等价于

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

等价于

```
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
```

dot运算符	▸ 实现函数组合（function composition）的功能 - 给定函数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

$$(f.g)\ x \xrightarrow{\text{等价于}} f.g\ \$\ x$$

$$f.g\ x \xrightarrow{\text{等价于}} f.(g\ x)$$

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]表示一个新的类型：
其中包含了所有由0到多个a中的元素形成的序列

❖其实，在类型[a]这种表示方式中，[]也是一个函数

- 函数[]接收一个类型，返回另一个类型
- 也即：函数[]将一个类型映射为另一个类型

以整数类型为例展示Haskell中序列类型数据基本表示方式	[]	▸ 空序列，即：由0个整数形成的序列
	[1], 1:[]	▸ 由一个整数1形成的序列 ▸ :是一个二元运算符
	[1,2,3]	▸ 由多个整数形成的序列
	1:2:3:[]	

```
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)
```

- 在in后面的这个表达式中，可访问let ... in之间定义的变量
- let ... in之间定义的变量，只能被in后的那个表达式所访问

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子句挂载到定义qsort''(n:ns)上
- 在 qsort''(n:ns)=右侧 到 where关键词之间的区域, 都可以访问where子句中定义的变量

let in 表达式 VS where 子句

在很多情况下，两者没有本质的不同，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
```

在这种情况下
let in 就不太适用了

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）
把一个非常难于理解的算法变得更加容易理解了

- 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，而与试图理解程序的主体有密切的关系
  - 例如，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

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	标识符（Identifier）
	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） – 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	其它字符只能是字母、数字、英文下划线、或英文单引号
4	不能与Haskell的保留词重名 – 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

- 一些程序元素，其标识符首字符只能是大写字母
- 其他程序元素，其标识符首字符只能是小写字母

目前已经涉及的程序元素，包括：

- 函数、变量、及类型变量：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，会在介绍到相关程序元素时再进行说明

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的保留操作符重名** .. : :: = \ | <- -> @ ~ =>

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
- 具体含义在合适的时机再进行说明

Hello, World!

Haskell中的 Hello, World! 程序

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

- 在遵从Haskell语言规范的前提下，这个程序省略了一些语句，以至于看起来略显奇怪

```
module Main(main) where
  import Prelude


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

- 恢复这些被省略的语句，会得到左侧这个更完整的程序

```
1  module Main(main) where
2    import Prelude
3
4    main :: IO ()
5    main = do
6      putStrLn "Hello, World!"
```

因为它原本
就不是函数啊

为什么main的
类型不是函数呢

这个程序声明了一个模块

- 这个模块的名称为Main
- 这个模块对外输出了一个名为main的程序元素
- where子句后对该模块包含的程序元素进行了定义

关于模块，Haskell 语言规范给出了如下信息

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称为Main的模块：** ▸ Main模块必须输出一个名称为main的程序元素 ▸ main元素的类型必须是IO t，其中： – t：类型变量；在声明main的类型时需传入一个实际类型 – IO：Prelude中定义的一个程序元素，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.

关于模块，Haskell 语言规范给出了如下信息

3	**模块的名称必须满足如下两个条件之一：** ▸ 一个以大写字母开头的标识符 – 例如：Mymodule ▸ 两个或多个以大写字母开头的标识符通过点符号连接在一起 – 例如：This.Is.Mymodule
4.1	**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

关于模块，Haskell 语言规范给出了如下信息

4.2

若一个模块可能会被其他模块所引用，那么，该模块所在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.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
  2. 模块名中最后标识之前的所有标识符分别对应到文件系统的一个文件夹，且相邻标识符对应的文件夹之间存在嵌套关系
  3. 模块所在文件存放在模块名倒数第二个标识符对应的文件夹下
- 例如，模块This.Is.Mymodule必须存放在…/This/Is/Mymodel文件中

✳这里所指的模块文件名称并不包含文件的扩展名

```
1  module Main(main) where
2    import Prelude
3
4    main :: IO ()
5    main = do
6      putStrLn "Hello, World!"
```

第2行代码是一个模块加载语句，其含义是：

- 若把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：

- 若模块源码中不存在 import Prelude 语句，则缺省存在该语句
- 若模块源码中存在以 import Prelude 为前缀的语句，则不存在该语句

- 例如，如果模块中存在这样一条语句：

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

该语句的效果是：

- 把Prelude模块对外输出的 Integer、+、– 加载到当前模块
- 但Prelude模块对外输出的其它元素，则不会被加载到当前模块

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
- 把它们添加到import Prelude后的圆括号中，才构成合法的模块定义

```
1  module Main(main) where
2    import Prelude
3
4    main :: IO ()
5    main = do
6      putStrLn "Hello, World!"
```

第4行语句声明：程序元素main的类型为 IO ()

- ()：零元组（0-tuple）类型
- IO：一个类型构造器（type constructor）
- IO ()：一个封装了IO运算的类型，且该运算会返回一个零元组

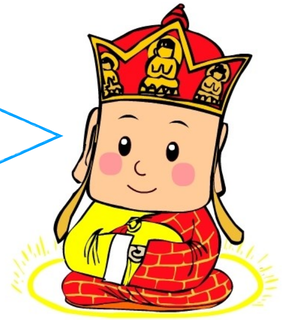
```
1  module Main(main) where
2    import Prelude
3
4    main :: IO ()
5    main = do
6      putStrLn "Hello, World!"
```

第5、6行代码定义了main中封装的IO运算

- 其中，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"` --应与第6行具有相同缩进
- 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



```
1  module Main(main) where
2    import Prelude
3
4    main :: IO ()
5    main = do
6      putStrLn "Hello, World!"
```

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

```
1  module Main(main) where
2    import Prelude
3
4    main :: IO ()
5    main = do
6      putStrLn "Hello, World!"
```

putStrLn是Prelude模块输出一个程序元素，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 ++ " = "
16       ++ (show $ int1 + int2)
```

getLine :Prelude模块输出一个程序元素

- 定义如下:

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

<- :与do绑定的一种语法符号

- 在左侧第07行中，其效果是：把getLine得到的IO String值中的那个String值赋给name

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 ++ " = "
16       ++ (show $ int1 + int2)
```

第13行代码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
- 这个let就是前面看到的let，用于定义在后面被使用的变量；具体细节，时机合适时再谈

`read str1 :: Integer`

- `read`: Prelude输出的一个函数
- `read`的类型大约是`String -> a`
- 效果:从字符串中读取一个整数值
- 在调用`read`时，若无法从上下文推断出a指代的类型，则需在其后放置 `:: Type`，显式指定a代表的类型为`Type`

一个具有更多交互性的程序

```haskell
module Main(main) where
  import Prelude

  main :: IO()
  main = do
    putStrLn "Please input your name:"
    name <- getLine
    putStrLn $ "Hello, " ++ name
    putStrLn "Please input an integer:"
    str1 <- getLine
    putStrLn "Please input another integer:"
    str2 <- getLine
    let int1 = (read str1 :: Integer)
    let int2 = (read str2 :: Integer)
    putStrLn $ str1 ++ " + " ++ str2 ++ " = "
      ++ (show $ int1 + int2)
```

```haskell
show $ int1 + int2
```

- show: Prelude输出的一个函数
- read的类型大约是a -> String
- 效果:把一个值映射为一个字符串

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动

不用太担心，想让程序动，分分钟的事

分分钟是多久呢

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

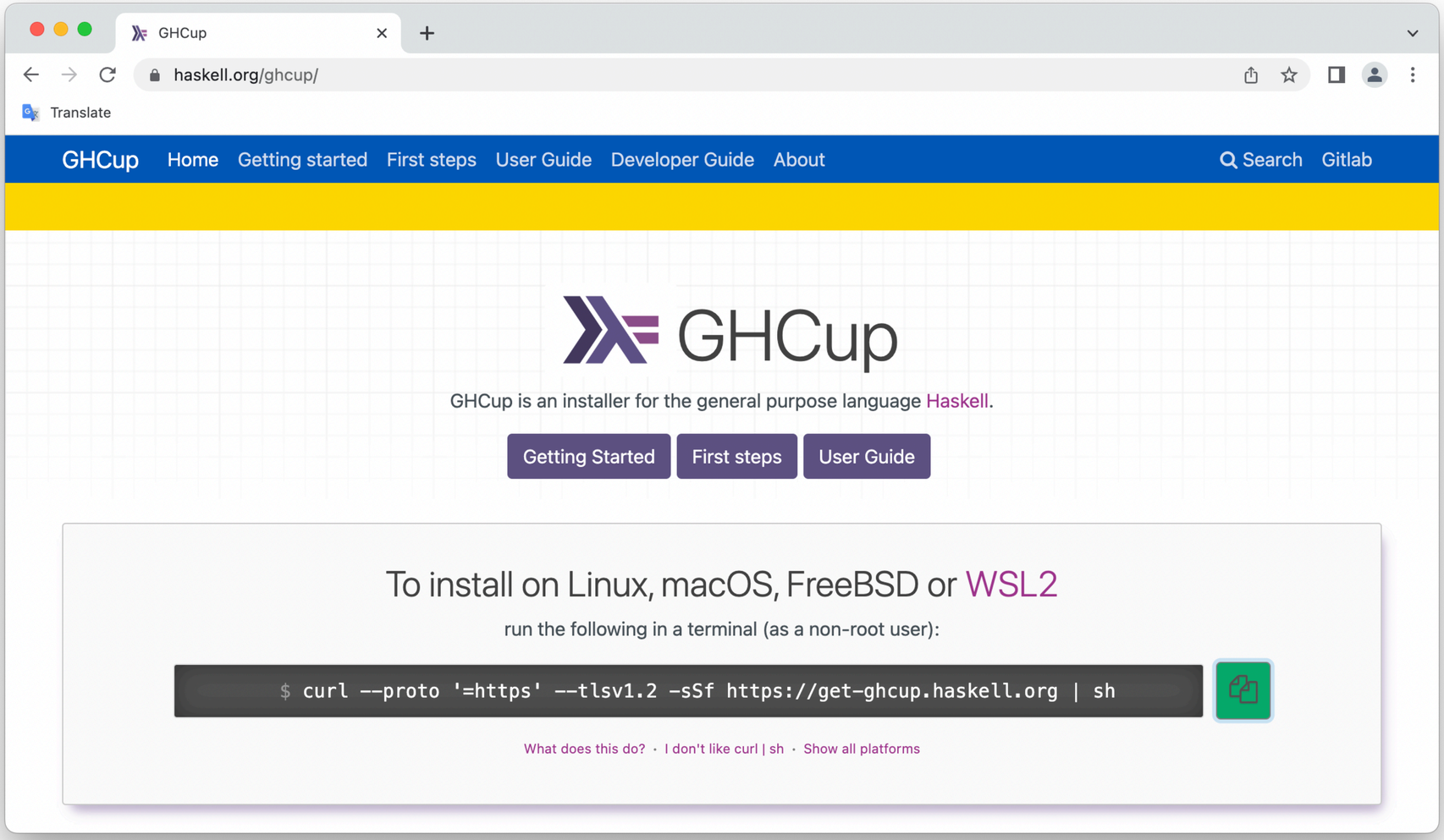
- 计算机需要理解程序，并在各类硬件和软件资源的支持下，执行程序所表达的计算过程
- 对于一种程序设计语言的发明者们而言，定义语言的语法形式，仅仅是万里长征的第一步
- 为了让程序能够在硬件上运行，还需提供一系列软件支撑工具
- 这些工具又被称为：程序设计语言的**工具链（toolchain）**

在本节中，我们主要介绍 Haskell 语言工具链中的三个基本工具：

- 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．

通过 ghcup 安装Haskell工具链

ghcup - The Haskell toolchain installer



在终端（Terminal）程序中运行如下命令

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

- 所谓“运行这个命令”，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

第一步：欢迎页面

```
nrutas@Weis-MacBook-Pro ~ % curl --proto '=https' --tlsv1.2 -sSf https://get-ghcup.haskell.org | sh

Welcome to Haskell!

This script can 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

ghcup installs only into the following directory,
which can be removed anytime:
  /Users/nrutas/.ghcup

Press ENTER to proceed or ctrl-c to abort.
Note that this script can be re-run at any given time.

------------------------------------------------------------------------
```

第二步：选择安装哪些工具

```
-----------------------------------------------------------------------------

Detected zsh shell on your system...
Do you want ghcup to automatically add the required PATH variable to "/Users/nrutas/.zshrc"?

[P] Yes, prepend  [A] Yes, append  [N] No  [?] Help (default is "P").


-----------------------------------------------------------------------------
Do you want to install haskell-language-server (HLS)?
HLS is a language-server that provides IDE-like functionality
and can integrate with different editors, such as Vim, Emacs, VS Code, Atom, ...
Also see https://haskell-language-server.readthedocs.io/en/stable/

[Y] Yes  [N] No  [?] Help (default is "N").

Y
-----------------------------------------------------------------------------
Do you want to install stack?
Stack is a haskell build tool similar to cabal t
Also see https://docs.haskellstack.org/

[Y] Yes  [N] No  [?] Help (default is "N").

Y
```

然后，就是漫长的下载过程

其中可能还会遇到有些链接无法打开的问题

但是，…，终于还是安装成功了

第嗯步：提示安装成功的页面

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

OK! /Users/nrutas/.zshrc has been modified. Restart your terminal for the changes to take effect,
or type "source /Users/nrutas/.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

If you are new to Haskell, check out https://www.haskell.org/ghcup/steps/
```

需重启终端程序，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

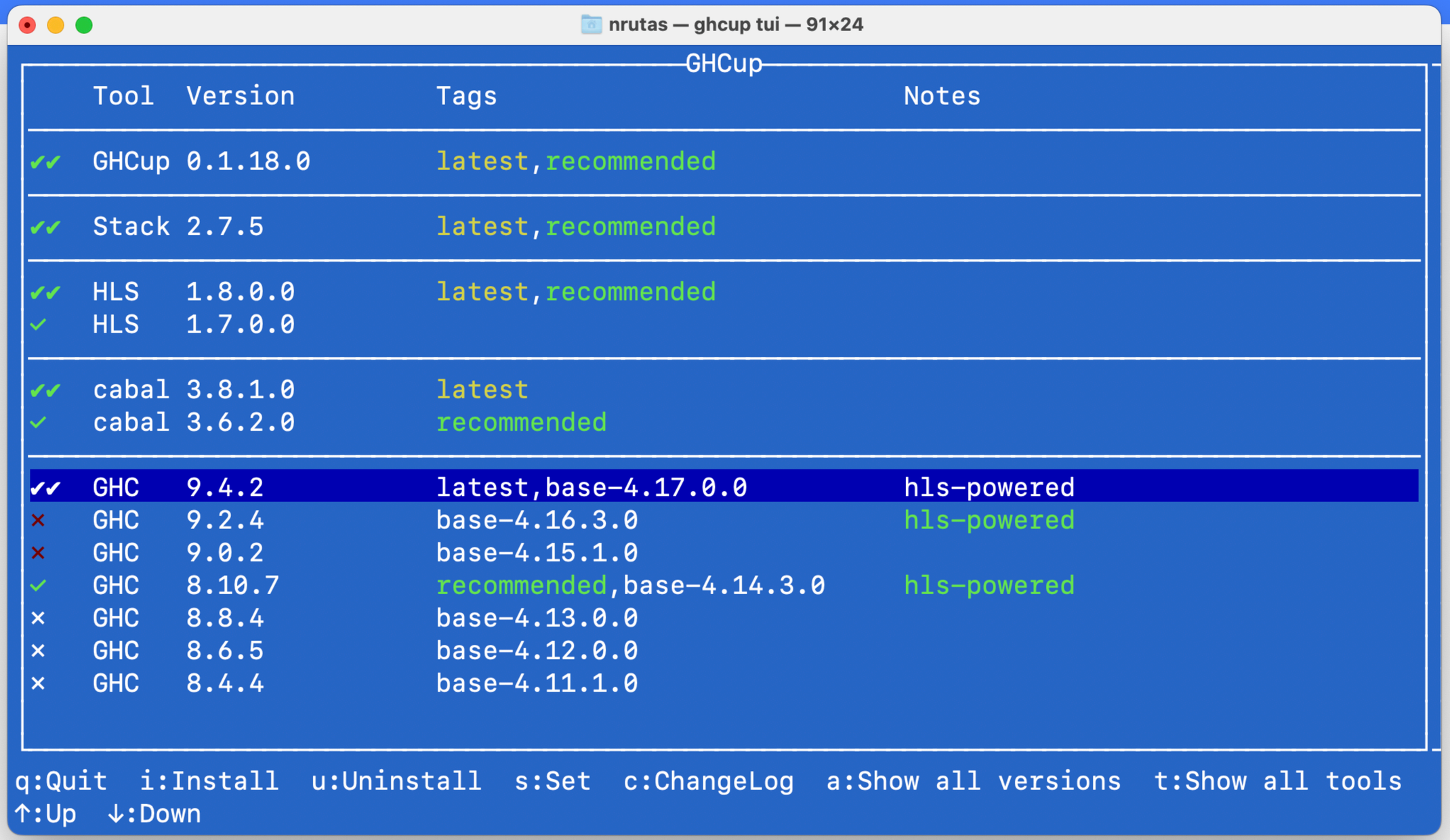
安装成功后，在本机的终端中，会增加三个新的可用命令：ghcup、ghc、ghci．后面两个命令分别是 Haskell 程序的编译器和交互式运行环境，我们稍后会对它们进行说明．第一个命令 ghcup 是 Haskell 工具链安装工具（Haskell toolchain installer）．关于 ghcup 命令的详细使用说明，可访问其官方链接：

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

通过 ghcup tui 命令对Haskell工具链进行管理

如果你的CPU是Apple M1系列
把这些工具的版本都拉到最高吧

我计算机上的截图


nrutas — ghcup tui — 91×24
GHCup
Tool Version Tags Notes
✔✔ GHCup 0.1.18.0 latest,recommended
✔✔ Stack 2.7.5 latest,recommended
✔✔ HLS 1.8.0.0 latest,recommended
✓ HLS 1.7.0.0
✔✔ cabal 3.8.1.0 latest
✓ cabal 3.6.2.0 recommended
✔✔ GHC 9.4.2 latest,base-4.17.0.0 hls-powered
✗ GHC 9.2.4 base-4.16.3.0 hls-powered
✗ GHC 9.0.2 base-4.15.1.0
✓ GHC 8.10.7 recommended,base-4.14.3.0 hls-powered
✗ GHC 8.8.4 base-4.13.0.0
✗ GHC 8.6.5 base-4.12.0.0
✗ GHC 8.4.4 base-4.11.1.0
q:Quit i:Install u:Uninstall s:Set c:ChangeLog a:Show all versions t:Show all tools
↑:Up ↓:Down

ghc 的使用

- ghc: 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

把该程序存放到某个文件夹F下的Main.hs文件中

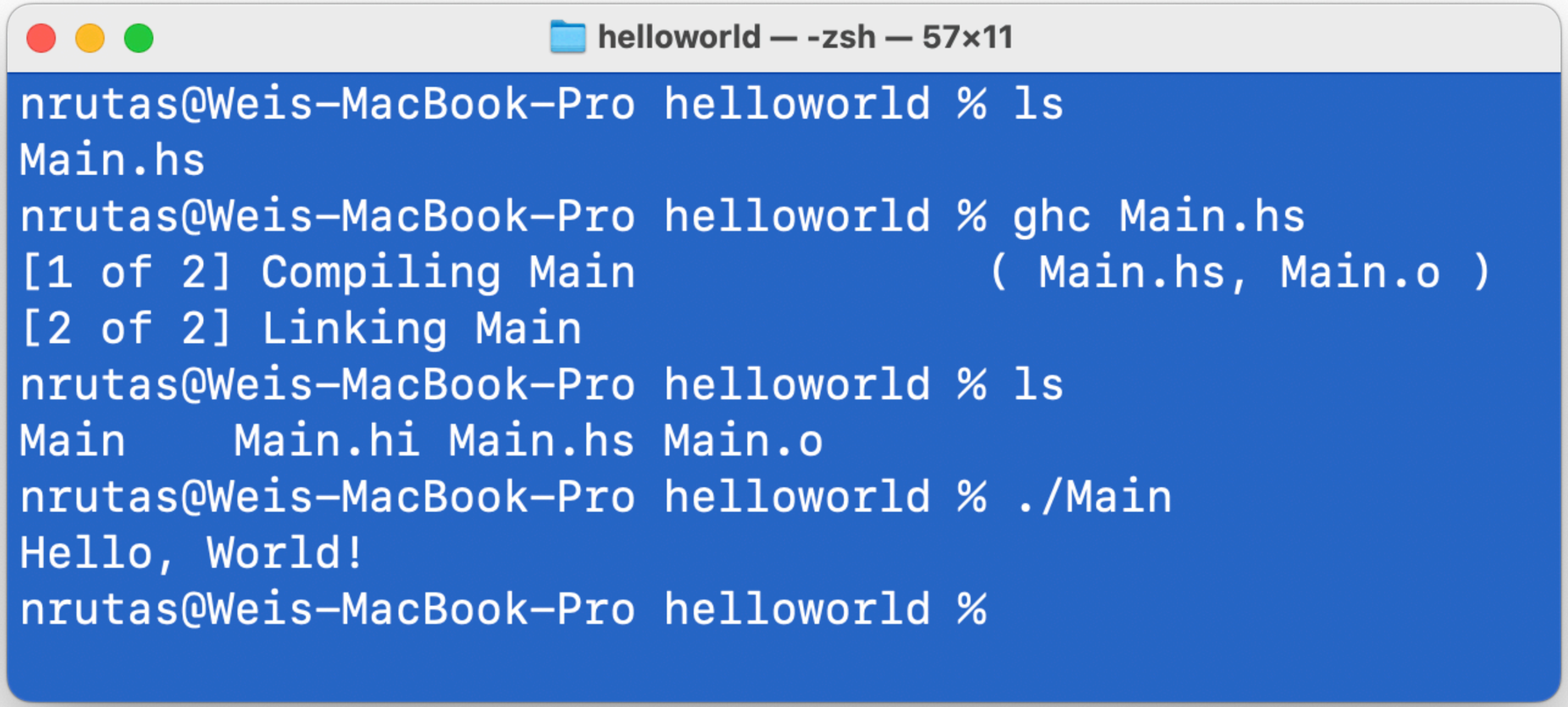
```
01 -- This is my first Haskell program
02 module Main(main) where
03   import Prelude
04
05   main :: IO ()
06   main = do
07     putStrLn "Hello, World!"
```

以左边程序为例，展示ghc的使用

第一行：程序注释（comment）

- Haskell的注释分为两类
  - 单行注释：以--开始的一行文字
  - 多行注释：以{-开始、以-}结束的多行文字

打开终端程序，把当前文件夹设为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

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，可访问其官方链接：
https://downloads.haskell.org/ghc/latest/docs/html/users_guide/ghc.html

- 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

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

- 你可以在其中输入合法的Haskell表达式，环境会输出求值结果
  - 例如：输入表达式 `1 + 2`，环境会输出 `3`
  - 例如：输入表达式 `reverse [1,2,3]`，环境输出 `[3,2,1]`

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
:set prompt "ghci> "	把ghci环境的命令行提示符修改为指定的字符串
: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

打开终端程序，把当前文件夹设为Main.hs所在的文件夹

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

- ghc、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

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

- 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helloworld的项目，且指定该项目使用的模版是new-template，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，当前文件夹下会存在一个名称为helloworld的文件夹：
  - 项目的所有信息都会被放在这个文件夹中
- 使用 cd helloworld 命令进入这个文件夹，会发现其中存在的信息如右图所示

```
.
├── 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
```

为了运行这个程序，需要首先对它进行构建（build）

stack build

- 使用命令 stack build 对当前项目进行构建
  - 如果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仍然会下载一个供自己单独使用的GHC编译器

- 构建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
  - 如果这样做了，在继续下面的活动前，使用 cd .. 返回上层目录

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

三个文件 LICENSE README.md ChangeLog.md	不会参与到编译活动中，不会对构建过程产生影响 ▶ 第一个文件：声明当前项目版权相关的信息 ▶ 第二个文件：对当前项目的简要说明 ▶ 第三个文件：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

stack 的创建的文件：stack.yaml

文件stack.yaml记录了项目级别的一些参数设置

其中，存在两个需要注意的配置项

```
packages:
- .
```

- 表示当前项目中仅包含一个package，它就存在于文件stack.yaml 所在文件夹中

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

- 默认情况下，其值为一个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

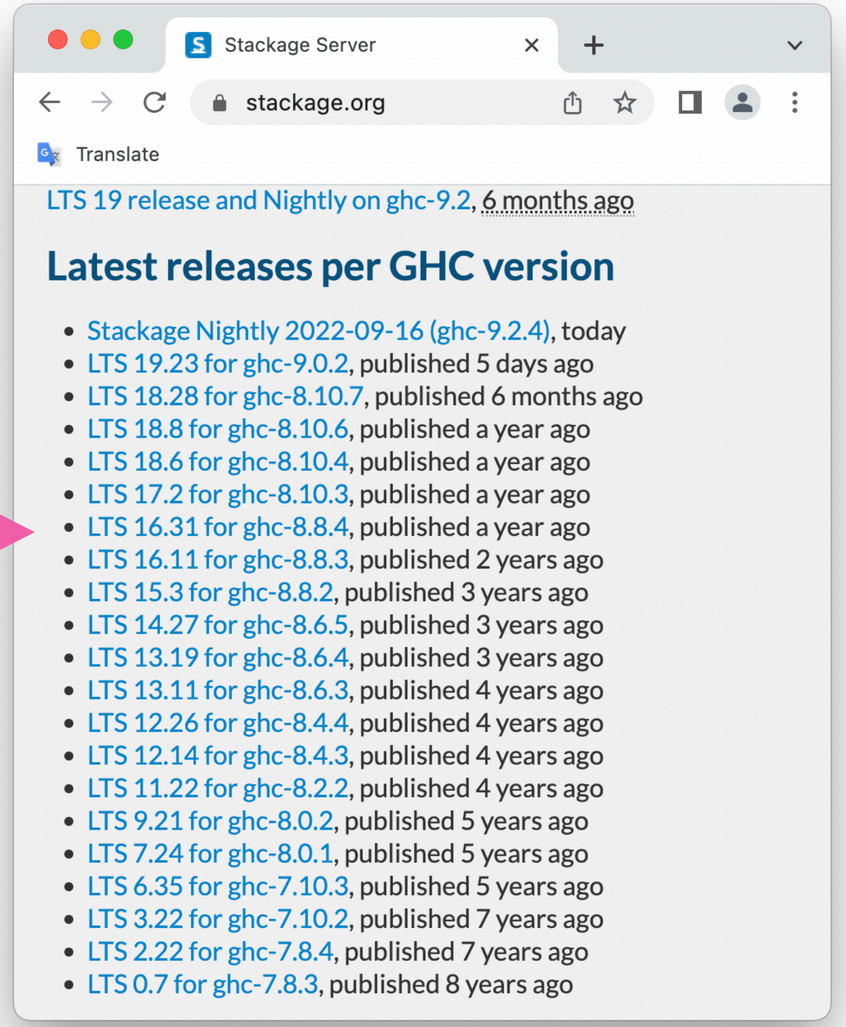
Apple M1芯片需要对resolver进行修改

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

把上面的两行文字
修改成下面的样子

```
resolver: nightly-2022-09-15
```

或者到stackage.org上，找到一个
支持ghc-9.2.4或以上版本的resolver



stack 的创建的文件：package.yaml

- 文件package.yaml记录了当前package的很多配置项，例如：
  - 可执行文件放在文件夹app中，main在文件Main.hs中
  - 包含的模块放在文件夹src中
  - 测试程序放在文件夹test中，main在文件Spec.hs中

```
1   name:                helloworld
2   version:             0.1.0.0
3   github:              "githubuser/helloworld"
4   license:             BSD3
5   author:              "Author name here"
6   maintainer:          "example@example.com"
7   copyright:           "2021 Author name here"
```

```
25  library:
26    source-dirs: src
```

```
28  executables:
29    helloworld-exe:
30      main:                Main.hs
31      source-dirs:         app
32      ghc-options:
```

```
39  tests:
40    helloworld-test:
41      main:                Spec.hs
42      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．

为了兼容M1芯片，请把这个值修改为

`random >= 1.2 && < 2`

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

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

第2.2章：初见Haskell

暂且先到这里吧