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!"
```

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具有合法名称的一个文件中** ▸ 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模块放在任意具有合法名称的文件中 ▸ 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

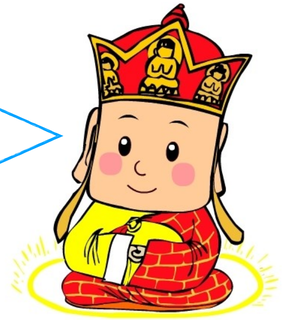
```
1  module Main(main) where
2    import Prelude
3
4    main :: IO ()
5    main = do
6      putStrLn "Hello, World!"
```

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
- 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

第2.2章：初见Haskell

未完待续