Haskell

1. 第 02 章：初见函数式思维

1.1. 两句很有哲理的话

1. 工欲善其事，必先利其器。
2. To a man with a hammer, everything looks like a nail.

✧ 思维方式是一种工具

✧ 不能被思维方式束缚

1.2. 函数式思维 是一种什么样的思维方式

✧ 使用 “数学中的函数” 作为 求解信息处理问题的基本成分。

✧ “使用方式”包括：

- 从零开始，定义一些基本函数
- 把已有的函数组装起来，形成新的函数

1.3. 简要回顾：数学中的函数

定义（函数）

给定定义域 X、值域 Y，称两者之间的关系 $f \subseteq X \times Y$ 是一个函数，当且仅当如下条件成立：

- 对 X 中的任何元素 x，存在且仅存唯一一个元素 $y \in Y$，满足 $(x, y) \in f$

函数相关的表示符号：

- $X \times Y$
  - 一个集合，其定义为 $\{ (x,y) \mid x \in X, y \in Y \}$
- X -> Y
  - 一个由函数构成的集合，包含且仅包含所有从 X 到 Y 的函数
- f : X -> Y
  - 基本等价于 f ∈ X -> Y，表示 f 是一个从 X 到 Y 的函数

- 也称：f 是一个类型为 X -> Y 的函数
- 在一般意义上，若 x ∈ X，也称 x 的类型为 X，记为 x : X

○ f(x)

- 函数 f 中与定义域中的元素 x 相对应的那个值
- 显然可知 f(x) : Y

常用的集合符号：

○ ℕ : Set 自然数集合/类型

○ ℤ : Set 整数集合/类型

○ ℚ : Set 有理数集合/类型

○ ℝ : Set 实数集合/类型

○ 𝔹 : Set = {true, flse} 布尔集合/类型；true 表示"真"，flse 表示"假"

定义（函数的组合）

给定函数 f : X -> Y 和 g : Y -> Z，两者的组合，记为 g * f，是一个函数，定义如下：

```
g * f : X -> Z
(g * f)(x) = g(f(x))
```

1.4. 为什么在函数的基础上，可以形成一种思维方式

✧ 函数可以建模 变换 和 因果关系

✧ 信息处理问题，本质上是一种信息的变换问题

✧ 在面向特定领域问题的软件应用中，大量涉及对物理世界中因果关系的仿真

1.5. 几个简单的函数

✧ 逻辑非 函数

```
not : 𝔹 -> 𝔹
not = { true |-> flse, flse |-> true }


not : 𝔹 -> 𝔹
not(true) = flse
```

```
not(flse) = true
```

✧ 逻辑与 函数

```
and : 𝔹 × 𝔹 -> 𝔹
and(true, true) = true
and(_, _) = flse

and : 𝔹 -> (𝔹 -> 𝔹)
and(true)(true) = true
and(_)(_) = flse
```

✧ 为了定义关于自然数 ℕ 的一些函数，我们首先给出 ℕ 的定义

✧ 自然数类型 ℕ 的定义（参考了 Haskell 定义类型的语法）

```
data ℕ = 0 | succ ℕ
```

这是一种递归定义，其含义如下：

1. 0 是 ℕ 中的一个元素
2. 如果 n 是 ℕ 中的一个元素，那么 succ n 也是 ℕ 中的一个元素
3. ℕ 中不存在其他不符合上述规则的元素

在这种定义下：1 == succ 0，2 == succ 1 == succ (succ 0)，...

✧ 自然数的加运算函数

```
plus : ℕ -> (ℕ -> ℕ)
plus(m)(0) = n
plus(m)(succ n) = succ (plus(m)(n))
```

加运算函数示例：

```
  plus(3)(4)
 = plus(3)(succ 3)
 = succ(plus(3)(3))
 = succ(plus(3)(succ 2))
```

```
= succ(succ(plus(3)(2)))
= succ(succ(plus(3)(succ 1)))
= succ(succ(succ(plus(3)(1))))
= succ(succ(succ(plus(3)(succ 0))))
= succ(succ(succ(succ(plus(3)(0)))))
= succ(succ(succ(succ(3))))
= (succ * succ * succ * succ)(3)
```

> 不要被上面这种看似复杂的定义所困扰。
> 它只不过用递归定义的方式表达了一件很简单的事情：
>
> ```
>     plus(m)(n) = (succ * succ * ... * succ)(m)
>                                 -- the compostion of n succ--
> ```

✧ 自然数的乘运算函数

```
mult : ℕ -> (ℕ -> ℕ)
mult(m)(0) = 0
mult(m)(succ n) = plus(m)(mult(m)(n))
```

乘运算函数示例：

```
  mult(3)(4)
= mult(3)(succ 3)
= plus(3)(mult(3)(3))
= plus(3)(mult(3)(succ 2))
= plus(3)(plus(3)(mult(3)(2)))
= plus(3)(plus(3)(mult(3)(succ 1)))
= plus(3)(plus(3)(plus(3)(mult(3)(1))))
= plus(3)(plus(3)(plus(3)(mult(3)(succ 0))))
= plus(3)(plus(3)(plus(3)(plus(3)(mult(3)(0)))))
= plus(3)(plus(3)(plus(3)(plus(3)(0))))
= (plus(3) * plus(3) * plus(3) * plus(3))(0)
```

不要被上面这种看似复杂的定义所困扰。

它只不过用递归定义的方式表达了一件很简单的事情：

```
mult(m)(n) = (plus(m) * plus(m) * ... * plus(m))(0)
                    ----- the compostion of n plus(m) -----
```

✧ 自然数的指数运算函数

```
expn : ℕ -> (ℕ -> ℕ)
expn(m)(0) = 1
expn(m)(succ n) = mult(m)(expn(m)(n))
```

指数运算函数示例：

```
  expn(3)(4)
= expn(3)(succ 3)
= mult(3)(expn(3)(3))
= mult(3)(expn(3)(succ 2))
= mult(3)(mult(3)(expn(3)(2)))
= mult(3)(mult(3)(expn(3)(succ 1)))
= mult(3)(mult(3)(mult(3)(expn(3)(1))))
= mult(3)(mult(3)(mult(3)(expn(3)(succ 0))))
= mult(3)(mult(3)(mult(3)(mult(3)(expn(3)(0)))))
= mult(3)(mult(3)(mult(3)(mult(3)(1))))
= (mult(3) * mult(3) * mult(3) * mult(3))(1)
```

不要被上面这种看似复杂的定义所困扰。

它只不过用递归定义的方式表达了一件很简单的事情：

```
expn(m)(n) = (mult(m) * mult(m) * ... * mult(m))(1)
                    ----- the compostion of n mult(m) -----
```

 小和尚	总是 n 个相同函数的组合；能不能有些新东西呢？	 老和尚
	何必让自己这么累；这样划水不挺好嘛！	

✧ 阶乘函数

```
fact : ℕ -> ℕ
fact(0) = 1
fact(succ n) = mult(succ n)(fact(n))
```

指数运算函数示例：

```
  fact(3)
= fact(succ 2)
= mult(succ 2)(fact(2))
= mult(succ 2)(fact(succ 1))
= mult(succ 2)(mult(succ 1)(fact(1)))
= mult(succ 2)(mult(succ 1)(fact(succ 0)))
= mult(succ 2)(mult(succ 1)(mult(succ 0)(fact(0))))
= mult(succ 2)(mult(succ 1)(mult(succ 0)(1)))
= (mult(succ 2) * mult(succ 1) * mult(succ 0))(1)
```

> 不要被上面这种看似复杂的定义所困扰。
> 它只不过用递归定义的方式表达了一件很简单的事情：
>
> ```
>     fact(n) = (mult(n) * mult(n - 1) * ... * mult(1))(1)
>                         ------- the compostion of n mult(_) -------
> ```

老和尚：看，是不是有那么一点点新东西了😊

✧ 斐波那契函数

```
fib : ℕ -> ℕ
```

```
fib(0) = 0
fib(1) = 1
fib(succ (succ n)) = plus(fib(n))(fib(succ n))
```

指数运算函数示例：

```
  fib(5)
= plus(fib(3))(fib(4))
= plus(plus(fib(1))(fib(2)))(plus(fib(2))(fib(3)))
= plus(plus(1)(plus(fib(0))(fib(1))))
  (plus(plus(fib(0))(fib(1)))(plus(fib(1))(fib(2))))
= plus(plus(1)(plus(0)(1)))(plus(plus(0)(1))(plus(1)(plus(fib(0))(fib(1)))))
= plus(plus(1)(plus(0)(1)))(plus(plus(0)(1))(plus(1)(plus(0)(1))))
```

小和尚：这下好了，没有规律了，看你怎么圆过来😝

1.6. 自然数上的 fold 函数

- plus / mult / expn 这三个函数之间存在共性
- 这种共性可以被封装在一个函数中

```
fold : (t -> t) -> (t -> (ℕ -> t))
fold(h)(c)(0) = c
fold(h)(c)(succ n) = h(fold(h)(c)(n))
```

说明：

- fold 的类型中出现了一个小写字母 t

  它是一个类型变量：可以用一个具体的类型替换 t，从而得到一个具体的 fold 类型

  这种语法来源于 Haskell

可知：

- h : t -> t,
- c : t

- 给定 h : t -> t, c : t，令 f = fold(h)(c)，则可知：

```
f(0) = c
```

```
f(succ n) = h(f(n))
```

如果不理解这个定义的含义，请看如下解释：

- 给定一个自然数 n，可知：

```
n = (succ * succ * ... * succ)(0)
         --the composition of n succ --
```

- 已知 f = fold(h)(c)，则可知：

```
f(n) = ( h   *   h   *   ...   *   h )(c)
          ----the composition of n h----
```

- 也即：
  - f(n) 把 n 中的 0 替换为 c，把 n 中的每一个 succ 替换为 h
  - n 和 f(n) 是同构的，即：两者具有相同的结构

✧ 使用 fold 函数，可以对 plus / mult / expn 这三个函数进行深刻的定义

```
plus : ℕ -> (ℕ -> ℕ)
plus(m) = fold(succ)(m)

mult : ℕ -> (ℕ -> ℕ)
mult(m) = fold(plus(m))(0)

expn : ℕ -> (ℕ -> ℕ)
expn(m) = fold(mult(m))(1)
```

示例：

```
                ------ the composition of n succ ------
          n          =  (succ          *  succ          * ... *  succ
       )(0)
                         |              |              |        |
plus(m)(n)     =  (succ          *  succ          * ... *  succ
   )(m)
                         |              |              |        |
mult(m)(n)     =  (plus(m)       *  plus(m)       * ... *  plus(m))(0)
                         |              |              |        |
```

```
expn(m)(n)    =  (mult(m)      *  mult(m)       * ... *  mult(m))(1)
```

✧ 使用 fold 函数，可以对 fact / fib 这两个函数进行深刻的定义

在此之前，首先引入两个辅助函数：

```
  fst : a × b -> a     // 注意：这里的 a b 都是类型变量
  fst(x, _) =  x

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

fact 函数的定义：

```
  fact : ℕ -> ℕ
  fact = snd * (fold(f)(0, 1))
where
  f : ℕ × ℕ -> ℕ × ℕ
  f(m, n) = (m + 1, (m + 1) * n)
```

```
                       ---the composition of n succ---
       n        =      (succ      *  succ     * ... *  succ    )(0)
                         |          |          |       |
 fact(n)        = snd((f          *   f       * ... *   f
    )(0,1))
                        --- the composition of n f ---
```

fib 函数的定义：

```
  fib : ℕ -> ℕ
  fib = fst * (fold(g)(0, 1))
where
  g : ℕ × ℕ -> ℕ × ℕ
  g(m, n) = (n, m + n)
```

```
                    ---the composition of n succ---
      n        =     (succ      *  succ    * ... *  succ    )(0)
                      |          |          |      |
 fib(n)        = fst((g         *   g      * ... *   g
   )(0,1))
                     --- the composition of n g ---
```

1.7. List 类型

✧ 在信息处理问题中，经常涉及一组按照某种顺序排列的数据；我们将这类数据称 List 或 List 类型的数据。

- 例如：对于排序问题而言，待排序的数据通常是采用 List 的方式进行组织的；排序的结果自然也是以序列的方式返回的.

✧ List 的符号表示

- 给定一个类型 a，我们使用 [a] 表示一个关于类型 a 的 List 类型。
  - 该类型的每一个元素是一个由 0 或多个 a 类型的元素形成的一个序列。
- 下面，我们以 [ℕ] 为例，对本章后面使用的关于 List 的若干表示方式进行说明
  - [] 表示由 0 个自然数形成的一个 list；显然，这是一个 empty list
  - [1] 或 1 ≺ [] 表示一个仅包含自然数 1 的 list
  - [1,2,2,3,3,3] 或 1 ≺ 2 ≺ 2 ≺ 3 ≺ 3 ≺ 3 ≺ [] 表示由 6 个自然数形成的一个 list

✧ List 类型的定义

data [a] = [] | a ≺ [a]

这是一种递归定义，其含义如下：

1. [] 是 [a] 中的一个元素
2. 如果 x 是类型 a 的一个元素，且 l 是类型 [a] 的一个元素，
   那么 x ≺ l 也是类型 [a] 的一个元素
3. [a] 中不存在其他不符合上述规则的元素

✧ List 相关的函数

```
cons : a -> ([a] -> [a])
cons(n)(ns) = n ≺ ns

len : [a] -> ℕ
len([]) = 0
len(n ≺ ns) = 1 + len(ns)

rev : [a] -> [a]
rev = revm([])
where
  revm : [a] ->([a] -> [a])
  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 -> 𝔹) -> ([a] -> [a])
filter(p)([]) = []
filter(p)(n ≺ ns) = concat(ifelse(p(n))([n])([]))(filter(p)(ns))
where
  ifelse : 𝔹 -> (a -> (a -> a))
  ifelse(true)(x)(_) = x
  ifelse(flse)(_)(y) = y
```

1.8. List 上的 fold 函数

✧ 如果我的理解没有错误，在任何类型上都存在 fold 函数。【这个观点待确认】

✧ 无论如何，ℕ 类型上存在 fold 函数，而且存在两个。

我们将这两个 fold 函数分别命名为 foldl 和 foldr

foldr 函数的定义

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

如果不理解这个定义，请看如下解释：

- 给定 xs : [a]。不失一般性，令 xs = x_n ≺ x_{n-1} ≺ ... ≺ x_1 ≺ []，可知：

  xs = (cons(x_n) * cons(x_{n-1}) * ... * cons(x_1))([])

  ------the composition of n cons(_)-----

- 已知 f = foldr(h)(c)，则可知：

  f(xs) = (h(x_n) * h(x_{n-1}) * ... * h(x_1))(c)

  ------the composition of n h(_)-------

- 也即：
  - f(xs) 把 xs 中的 [] 替换为 c，把 xs 中的每一个 cons 替换为 h
  - xs 和 f(xs) 是同构的，即：两者具有相同的结构

foldl 函数的定义

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

为了对这个定义进行进一步的解释，引入一个工具函数：

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

如果不理解 foldl 的定义，请看如下解释：

- 给定 xs : [a]。不失一般性，令 $xs = x_n \prec x_{n-1} \prec \ldots \prec x_1 \prec []$，可知：

  $xs = (cons(x_n) * cons(x_{n-1}) * \ldots * cons(x_1))([])$

  ------the composition of n cons(_)-----

- 已知 f = foldl(h)(c)，令 h' = flip(h)，则可知：

  $f(xs) = (h'(x_1) * h'(x_2) * \ldots * h'(x_n))(c)$

  ------the composition of n h'(_)-------

- 也即：
  - f(xs) 把 xs 中的 [] 替换为 c，把 xs 中的每一个 cons 替换为 h'，还顺便逆序了一下。
  - 这时，xs 和 f(xs) 是否具有相同的结构呢？

上面的解释不够直观，请再看如下解释：

- 给定 xs : [a]。不失一般性，令 $xs = x_n \prec x_{n-1} \prec \ldots \prec x_1 \prec []$，可知：

  $xs = x_n \prec x_{n-1} \prec \ldots \prec x_1 \prec []$

- 已知 f = foldl(h)(c)，令 u 𝕙 v = h(u)(v)，且运算符 𝕙 具有左结合律，可知

  $f(xs) = c \; 𝕙 \; x_n \; 𝕙 \; x_{n-1} \prec \ldots \prec x_1 \prec []$

1.9. 使用 fold 函数，重定义 List 相关的函数

✧ len 函数

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

```
               -----   the composition of n cons   -----
       xs    = (cons(x_n) * cons(x_{n-1}) * ... * cons(x_1))([])
                         |              |          |            |     |
   len(xs)   = (    h(x_n) *     h(x_{n-1})      * ... *      h(x_1))(0)
               -----   the composition of n h   --------
```

✧ rev 函数

```
rev : [a] -> [a]
rev = foldl(≻)([])
where
  l ≻ x = x ≺ l
```

```
     xs   =            x_n  ≺  x_{n-1}  ≺  ...  ≺  x_1  ≺  []
                  |       |        |         |       |
  rev(xs) = []  ≻  x_n  ≻  x_{n-1}  ≻  ...  ≻  x_1
```

✧ concat 函数

```
concat : [a] -> ([a] -> [a])
concat(xs)(ys) = foldr(cons)(ys)(xs)
```

```
        xs            = (cons(x_n) * cons(x_{n-1}) * ... * cons(x_1))([])
                          |             |              |      |
    |
   concat(xs)(ys)  = (cons(x_n) * cons(x_{n-1}) * ... * cons(x_1))(ys)
```

✧ filter 函数

```
filter : (a -> 𝔹) -> ([a] -> [a])
filter(p) = foldr(f(p))([])
where
```

```
f : (a -> 𝔹) -> (a -> ([a] -> [a]))
f(p)(x) = ifelse(p(x))(cons(x))(id)
id : a -> a
id x = x
```

```
              xs    = (cons(x_n) * cons(x_{n-1}) * ... * cons(x_1))([])

filter(f(p))(xs)    = (f(p)(x_n) * f(p)(x_{n-1}) * ... * f(p)(x_1))([])
```

1.10. 一种排序算法

✧ 快速排序算法

```
qsort : [ℕ] -> [ℕ]
qsort([]) = []
qsort(n ≺ ns) = concat(concat(filter(lt(n))(ns))([n]))
                          (qsort(filter(ge(n))(ns)))
where
  lt : ℕ -> (ℕ -> 𝔹)
  lt(n)(m) = if m < n then true else flse
  ge : ℕ -> (ℕ -> 𝔹)
  ge(n)(m) = not(lt(n)(m))
```

<table>
<tr><td rowspan="3">
小和尚</td><td>如果这就是用 FP 书写的算法，此生绝不学 FP！</td><td rowspan="3">
老和尚</td></tr>
<tr><td>好孩子，如果给你三生三世的财富，学否？</td></tr>
<tr><td></td></tr>
</table>

✧ 内容 与 形式

- 这是一个关于“内容”与“形式”两者之间关系的问题
  - 内容：对自然数序列进行排序的一种方法
  - 形式：表现这种排序方法的形式
- 进一步而言，这个问题可以表述为：
  - “形式”小于“内容”：内容是很好的，但形式实在是太糟糕了
- 如果你能体会到这一点，你会发现：这个问题的严重程度并不像表面上看起来的那样
- 为什么这么说呢？因为，本质（内容）毕竟还是很好的

✧ 重走长征路

- 在某种意义上，我们正在“重走长征路”
- 在很多年以前，科研工作者们就已经意识到了这个问题
- 在这个问题的驱使下，他/她们设计了各种各样的函数式程序设计语言
- 我们即将介绍的 `Haskell` 语言，就是这些函数式程序设计语言的集大成者
- 不过，目前看来，`Haskell` 语言正在逐渐老去：一鲸落，万物生！

1.11. 剧透：采用 Haskell 语言编写的 qsort 算法

```
qsort :: Ord a => [a] -> [a]
qsort []          = []
qsort (p:xs)      = qsort lt ++ [p] ++ qsort ge
  where
    lt = filter (< p) xs
    ge = filter (>= p) xs
```

> 这一章没有作业