# Learn Me a Haskell

Jacob Bishop

2023-07-01

# 1 Introduction

WIP

# 2 Starting Out

## 第一个函数

在 `./test/` 文件夹下创建一个 `baby.hs` 的文件，写入：

```
doubleMe x = x + x
```

使用 `ghci` 加载该文件（在本项目根目录时使用 `:l tests/baby` ）：

```
ghci> :l baby
[1 of 1] Compiling Main             ( baby.hs, interpreted )
Ok, modules loaded: Main.
ghci> doubleMe 9
18
ghci> doubleMe 8.3
16.6
```

一个带有 `if` 的函数：

```
doubleSmallNumber x =
  if x > 100
    then x
    else x * 2
```

Haskell 中的 `if` 声明是一个 `表达式`，那么 `else` 是强制性的，因为表达式一定要有所返回。因此加上述函数可以改写为：

```
doubleSmallNumber' x = (if x > 100 then x else x * 2) + 1
```

这里的 `'` 符号是 Haskell 中的有效字符，且在 Haskell 中并没有特殊的意义，因此可以用作于函数名。通常情况下，使用 `'` 代表着一个函数（非懒加载的函数）的严格版本，或是一个有细微变化的函数或者变量。又因为 `'` 是一个有效字符，那么可以创建以下函数：

```
conanO'Brien = "It's a-me, Conan O'Brien!"
```

这里又有两点值得注意的地方。首先，函数名不能以大写开头，稍后会进行说明；其次，该函数并没有任何入参。当一个函数没有入参，我们通常称其为一个 `定义` *definition*，因为一旦定义了它便不能修改其名称，以及其返回。

## list 的介绍

Haskell 中的 list 是 **同质的 homogenous** 数据结构。

> **Note**
>
> 在 `GHCI` 中可以使用 `let` 关键字定义一个名称。换言之，`GHCI` 中的 `let a = 1` 等同于脚本中的 `a = 1` 。

通常使用 `++` 操作符将两个数组进行合并:

```
ghci> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
ghci> "hello" ++ " " ++ "world"
"hello world"
ghci> ['w','o'] ++ ['o','t']
"woot"
```

可以使用 `:` 操作符将元素直接添加至数组头部:

```
ghci> 'A':" SMALL CAT"
"A SMALL CAT"
ghci> 5:[1,2,3,4,5]
[5,1,2,3,4,5]
```

实际上，`[1, 2, 3]` 是 `1:2:3:[]` 的语法糖，其中 `[]` 为一个空数组。如果头部追加 `3` ，`[]` 就变成了 `[3]` ，再次进行头部追加 `2` ，则变为 `[2, 3]` ，以此类推。

如果希望通过索引获取数组中的元素，那么可以使用 `!!` 操作符:

```
ghci> "Steve Buscemi" !! 6
'B'
ghci> [9.4,33.2,96.2,11.2,23.25] !! 1
33.2
```

超出索引时则会报错。

数组还可以通过操作符 `<` ，`<=` ，`==` ，`>` 以及 `>=` 操作符来进行比较，而比较的方式则是顺序比较。当进行头部比较元素相等时，再进行下一个元素进行比较。

数组的四种基础操作 `head` ，`tail` ，`last` 以及 `init` :

```
ghci> head [5,4,3,2,1]
5
ghci> tail [5,4,3,2,1]
[4,3,2,1]
ghci> last [5,4,3,2,1]
1
ghci> init [5,4,3,2,1]
[5,4,3,2]
```

当使用上述四种操作时，需要注意是否应用于空数组，这样的错误在编译期并不能被发现。

其它的操作:

1. `length` 获取数组长度;
2. `null` 检查数组是否为空;
3. `reverse` 翻转数组;
4. `take` 获取数组的头几个元素的数组;

5. `drop` 移除数组的头几个元素，并返回剩余元素的数组；
6. `maximum` 获取最大值；
7. `minimum` 获取最小值；
8. `sum` 求和；
9. `product` 求积；
10. `elem` 元素是否存在于数组中。

```
ghci> length [5,4,3,2,1]
5

ghci> null [1,2,3]
False
ghci> null []
True

ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]

ghci> take 3 [5,4,3,2,1]
[5,4,3]
ghci> take 1 [3,9,3]
[3]
ghci> take 5 [1,2]
[1,2]
ghci> take 0 [6,6,6]
[]

ghci> drop 3 [8,4,2,1,5,6]
[1,5,6]
ghci> drop 0 [1,2,3,4]
[1,2,3,4]
ghci> drop 100 [1,2,3,4]
[]

ghci> minimum [8,4,2,1,5,6]
1
ghci> maximum [1,9,2,3,4]
9

ghci> sum [5,2,1,6,3,2,5,7]
31
ghci> product [6,2,1,2]
24
ghci> product [1,2,5,6,7,9,2,0]
```

```
0

ghci> 4 `elem` [3,4,5,6]
True
ghci> 10 `elem` [3,4,5,6]
False
```

### Texas 排列

```
ghci> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> ['K'..'Z']
"KLMNOPQRSTUVWXYZ"
```

带有 step 的排列：

```
ghci> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
ghci> [3,6..20]
[3,6,9,12,15,18]
```

而对于浮点数的排列需要注意精度问题：

```
ghci> [0.1, 0.3 .. 1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
```

以下是若干用于生产无限长度数组的函数：

`cycle` 循环周期：

```
ghci> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
ghci> take 12 (cycle "LOL ")
"LOL LOL LOL "
```

`repeat` 重复：

```
ghci> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
```

另外就是 `replicate` 函数可以重复单个元素：

```
ghci> replicate 3 10
[10,10,10]
```

### 列表表达式

数学里的 集合表达式 *set comprehensions* 例如 $S = 2 \cdot x | x \in \mathbb{N}, x \leq 10$；Haskell 中的列表表达式，例如 1 至 10 数组中每个元素乘以 2：

```
ghci> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
```

为列表表达式添加条件（或称谓语 predicate）：

```
ghci> [x*2 | x <- [1..10], x*2 >= 12]
[12,14,16,18,20]
ghci> [ x | x <- [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]
```

将列表表达式置于一个函数中便于复用：

```
ghci> boomBangs xs = [ if x < 10 then "BOOM!" else "BANG!" | x <- xs, odd x]
ghci> boomBangs [7..13]
["BOOM!","BOOM!","BANG!","BANG!"]
```

多个谓语也是可以的：

```
ghci> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20]
```

除此之外，还可以处理若干数组：

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

当然也可以加上谓语：

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]
[55,80,100,110]
```

那么对于字符串也可以使用列表表达式：

```
ghci> let nouns = ["hobo","frog","pope"]
ghci> let adjectives = ["lazy","grouchy","scheming"]
ghci> [adjective ++ " " ++ noun | adjective <- adjectives, noun <- nouns]
["lazy hobo","lazy frog","lazy pope","grouchy hobo","grouchy frog",
"grouchy pope","scheming hobo","scheming frog","scheming pope"]
```

现在让我们编写一个自己的 `length` ，命名 `lenght'` （这里的 `_` 意为无需使用的变量）：

```
length' xs = sum [1 | _ <- xs]
```

由于字符串是数组，因此我们可以使用列表表达式处理并生产字符串。以下是一个移除所有字符但保留大写字符的函数：

```
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
removeUppercase st = [ c | c <- st, c `notElem` ['A'..'Z']]
```

## 元组

在某种程度上，元组类似于数组 – 存储若干值至单个变量上。然而有一些基础的差异：数组长度可以无限，元组长度固定；数组中元素类型是同质的，而元组则可以是异质的 heterogenous。

对于对元组（当且仅当包含两个元素）有以下操作：

`fst` 获取对元组的第一个元素：

```
ghci> fst (8,11)
8
ghci> fst ("Wow", False)
"Wow"
```

`snd` 获取对元组的第二个元素：

```
ghci> snd (8,11)
11
ghci> snd ("Wow", False)
False
```

另外一个有意思的函数则是 `zip` ，它可以将两个数组按对拼接成对元组的数组

```
ghci> zip [1,2,3,4,5] [5,5,5,5,5]
[(1,5),(2,5),(3,5),(4,5),(5,5)]
ghci> zip [1 .. 5] ["one", "two", "three", "four", "five"]
[(1,"one"),(2,"two"),(3,"three"),(4,"four"),(5,"five")]
```

当两个数组的长度不一时，`zip` 则按最短的那个进行对齐，长的数组剩余部分则被丢弃，这是因为 Haskell 是懒加载的缘故。

```
ghci> zip [5,3,2,6,2,7,2,5,4,6,6] ["im","a","turtle"]
[(5,"im"),(3,"a"),(2,"turtle")]
ghci> zip [1..] ["apple", "orange", "cherry", "mango"]
[(1,"apple"),(2,"orange"),(3,"cherry"),(4,"mango")]
```

# 3 Types and Typeclasses

WIP

# 4 Syntax in Functions

WIP

# 5 Recursion

WIP

# 6 Higher Order Functions

WIP

# 7 Modules

WIP

# 8 Making Our Own Types and Typeclasses

WIP

# 9 Input and Output

WIP

# 10 Functionally Solving Problems

WIP

# 11 Functors, Applicative Functors and Monoids

WIP

# 12 A Fistful of Monads

WIP

# 13 For a Few Monads More

WIP

# 14 Zippers

WIP