4 Haskell 类型

4.1 数据类型的查看与表示

4.1.1 查看数据类型

```
ghci>:t 'a'
'a' :: Char
ghci>:t 3
3 :: Num a => a
ghci>:t 2.5
2.5 :: Fractional a => a
ghci>:t (1,2)
(1,2) :: (Num t1, Num t) => (t, t1)
ghci>:t [1,2]
[1,2] :: Num t => [t]
ghci>:t "string"
"string" :: [Char]
```

4.1.2 函数类型的表示

一个特别的地方在函数类型

```
ghci>:t head
head :: [a] -> a
```

上面的head函数，类型是[a] -> a，意思是参数是List类型，其中List元素是a类型，返回的值是一个a类型。 这里的a叫做 **类型变元(Type variable)**,它是个变量，取值可以是整数，字符，浮点数等等具体类型 另外，对于多个参数的函数,例如map

```
ghci>:t map
map :: (a -> b) -> [a] -> [b]
```

这表示map有两个参数，第一个参数是一个函数，类型是a->b,即这个函数的类型是接受一个a类型的参数，返回一个b 类型的结果。第二个参数是一个List,元素类型是a。返回结果是一个List,元素类型是b。

之前我们定义一个函数时并没有指定一个函数的类型，这和C,C++,Java等语言定义函数时是不同的。之所以可以这样， 是因为Haskell的类型系统可以推断出函数的类型，所以即使你没给出函数的类型，在你输入一个错误的参数时， Haskell仍然会给出错误提示。

当然，我们也可以给出函数的类型，方法如下：

```
charReverse :: [Char] -> [Char]
charReverse [] = []
charReverse (x:xs) = charReverse xs ++ [x]
ghci>:t charReverse
charReverse :: [Char] -> [Char]
```

```
ghci>charReverse "asdf"
"fdsa"
```

4.1.3 基本数据类型

- Int：代表有范围的整数
- Integer: 代表没有范围的整数
- Float,Double,Bool,Char: 你懂的
- Tuple:　之前介绍过

4.2 类型类(Type class)

4.2.1 为什么要有类型类

我们看这样一个函数: ==,它有什么类型呢？检查一下。

```
ghci>:t (==)
(==) :: Eq a => a -> a -> Bool
```

这个函数的类型是a -> a -> Bool.即接收两个相同类型的参数，返回Bool类型的值，例如

```
ghci>3 == 4
False
ghci>3.1 == 3.2
False
ghci>'a' == 'b'
False
```

我们发现，这个函数并没有给出参数的具体类型，只说明了两个参数类型必须一样，那么是不是只满足这一点就可以用==函数来判断相等了呢？当然不是。不然以后我们定义了新 类型，不告诉==什么才叫相等，==就能判断新类型相等了，那不是过于神奇了嘛。所以，必须 对==的参数类型加以限制，这就是前面那个Eq a 的作用了！ 这里的Eq就是一个类型类(Type class),它并不是一个类，倒更像一个接口，Eq a对类型a 作了限制，表示类型a必须得是Eq的一个实例(instance)才行。而成为Eq的实例， 需要实现一些函数，这些函数告诉了==满足什么样的条件说明这两个元素是相等的。 所以，并不是所有类型的元素都可以做==的参数的，必须得是Eq实例的类型才可以。

4.2.2 常见类型类

- Eq:一个类型成为它的实例，就可以用==比较这个类型的两个元素是不是相等了。

```
ghci>:t (==)
(==) :: Eq a => a -> a -> Bool
ghci>3 == 4
False
```

- Ord:一个类型成为它的实例，就可以用>比较这个类型的两个元素的大小了。

```
ghci>:t (>)
(>) :: Ord a => a -> a -> Bool
ghci>3 > 4
False
ghci>3 == 4
False
ghci>3 < 4
True
ghci>3 /= 4
True
ghci>:t compare
compare :: Ord a => a -> a -> Ordering
ghci>3 `compare` 4
LT
ghci>3 `compare` 3
EQ
ghci>4 `compare` 3
GT
```

- Show:一个类型成为它的实例，ghci就知道如何来显示它了。

```
ghci>show 3
"3"
ghci>show 4
"4"
ghci>show ['a','b']
"\"ab\""
ghci>show "ab"
"\"ab\""
```

- Read:一个类型成为它的实例，ghci就可以把读取的东西当成这种类型的了。

```
ghci>:t read
read :: Read a => String -> a
ghci>read "5" :: Int
5
ghci>read "5" :: Float
5.0
```

- Enum:一个类型成为它的实例，这个类型的元素就是可以依序列举出来。
- Bounded:一个类型成为它的实例，这个类型的元素有上界和下界。
- Num:一个类型成为它的实例，这个类型的元素有着和数字相似的行为。比如Int,Integer,Float,Double 都是Num的实例(instance)。
- Floating:Float和Double是它的实例。
- Integral:Int和Integer是它的实例。

4.3 定义和使用新的类型

4.3.1 定义一个新一数据类型

先看看在标准库里，数据类型是如何定义的

```
data Bool = False | True
data Int = -2147483648 | -2147483647 | ... | -1 | 0 | 1 | 2 | ... |
2147483647
```

我们看到了，数据类型可以使用 **data** 关键字来定义。 下面我们定义自己的数据类型。 假如我们要定义形状这个数据类型，形状可能是圆，也可能是矩形，我们可以这样定义

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
```

这里的 **Circle** 和 **Rectangle** 我们称做值构造符(value constructor).这是什么意思呢？ 我们看一下它们的类型：

```
ghci>:t Circle
Circle :: Float -> Float -> Float -> Shape
ghci>:t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
```

这么一看，居然和函数一样。为什么要这样表示呢？ Shape是一个类型，一个类型总要有确定的值。就像Int类型有1,2,3这样的值。 那么怎么表示Shape的值呢？Shape的值无法用一个统一的形式来表示，因为圆需要三个值可以确定， 矩形则需要四个值。所以就引入值构造符(value constructor)这个概念，让不同形式的值统一起来。 例如：

```
ghci>:t (Circle 1 2 3)
(Circle 1 2 3) :: Shape
ghci>:t (Rectangle 1 2 3 4)
(Rectangle 1 2 3 4) :: Shape
```

(Circle 1 2 3)和(Rectangle 1 2 3 4)都是Shape类型的值。

4.3.2 使用新数据类型

在定义了新类型后，就可以用这个新类型写一些函数了。

```
area :: Shape -> Float
area (Circle _ _ r) = pi * r * r
area (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

```
ghci>area (Rectangle 1 2 3 4)
4.0
ghci>area (Circle 1 2 3)
28.274334
```

当然，我们也可以再定义一个数据类型Point,再用Point构成Shape

```
data Point = Point Float Float
data Shape = Circle Point Float | Rectangle Point Point
```

另外，如果我们在ghci里直接输入Circle 1 2 3是无法正常显示的，那么如何显示他们呢？之前讲的类型类就有用了，要想显示，必须成为Show的实例，如何成为Show的实例呢？只需要使用deriving

```
data Shape = Circle Float Float Float | Rectangle Float Float Float
Float
           deriving (Show)
ghci>Circle 1 2 3
Circle 1.0 2.0 3.0
```

虽然我们没有具体定义如何成为Show的实例，即显示的方式，但是总算可以正常显示了， 这里是使用了默认的显示方式。

另外，在使用新数据类型时，还涉及一个问题，如果a = Circle 1 2 3.那么我们如何通过 a 得到里面的值1 2 3呢？我们可以通过写三个函数来得到：

```
mrx (Circle x _ _ ) = x
mry (Circle _ y _ ) = y
mr (Circle _ _ vr ) = vr
ghci>let a = (Circle 1 2 3)
ghci>mrx a
1.0
ghci>mry a
```

```
2.0
ghci>mr a
3.0
```

但是这种方法实在太麻烦了，所以Haskell提供了另外一种方式。

```
data Shape = Circle {rx :: Float, ry :: Float, r :: Float} | Rectangle
Float Float Float Float
           deriving (Show)
ghci>let a = (Circle 1 2 3)
ghci>rx a
1.0
ghci>ry a
2.0
ghci>r a
3.0
```

这种方式在定义数据类型的同时，就定义了取得相应值的函数。

4.3.3 类别名(Type Synonyms)

可以通过下面的方式为[Char]起一个别名String,这样可以使类型的名字有较好的自我描述能力。

```
type String = [Char]
```

4.3.4 递归定义+类型变元(Type variable)

下面我们来定义一棵树，树的定义是递归的，我们可以这样定义

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a) deriving (Show)
```

这棵树是空树或者是一个结点值＋左子树＋右子树的形式。 这里的a是结点值的类型，即type variable,它可以代表Int,Float等等类型。如果不引入类型变元 的概念，那岂不是要为

每种类型都定义一种树？ 其中的Node是值构造符，它把结点值＋左子树＋右子树结合在了一起，构成一个Tree类型的值。

4.3.5 让新类型成为类型类(Type class)的实例

之前我们说明类型类的概念，比如Eq是一个类型类,新类型要想使用==函数，必须是 Eq的一个实例(instance)，实例化的具体过程我们将在这里介绍。 假如定义了一种新类型

```
data TrafficLight = Red | Yellow | Green
```

此时，这种新类型必然无法使用==判断是否相等，因为==不知道何为相等，所以我们需要 告诉==何为相等，告诉的过程就是新类型成为Eq实例的过程。

```
instance Eq TrafficLight where
  Red == Red = True
  Yellow == Yellow = True
  Green == Green = True
  _ == _ = False
```

这就是实例化的过程，它告诉了==什么叫相等，什么叫不相等。

```
ghci>Red == Red
True
ghci>Red == Yellow
False
```