Haskell 3-4

Misc

- **Imperative languages** 命令式语言

函数语法

- `parttern matching` 可以为一个函数不同的pattern定义不同的执行逻辑，如下，如果传入的参数是一个数字，且是7，就输出幸运，否则不是．这样做的好处是可读性更强(可以代替 if，else 判断)．Pattern是从上到下检查的．第一个符合的pattern被采用，后面的符合的patten也不会被检查．

```
lucky :: (Integral a) => a -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"
```

- 递归调用的定义相当于pattern定义的一种特殊形式，递归调用的出口要写在最前面．
- 如果输入的参数不能被任何一个pattern match，那么会抛出异常．
- `error` 这个函数，相当于在抛异常前打个log．然后让程序RE．
- `pattern match in list comprehensions` 可以在集合的filter里用pattern matching．例如下面的code．所谓的pattern matching 就是如果不用一个a,b去表示pair的两个数，就不能把他们的简洁的表达为`a+b`，所以这里的a,b就是pattern matching．

```
ghci> let xs = [(1,3), (4,3), (2,4), (5,3), (5,6), (3,1)]
ghci> [a+b | (a,b) <- xs]
[4,7,6,8,11,4]
```

- 如果一个pattern matching失败了，自动跳过，继续下一个．
- list 本身也可以被pattern matching．所有list都是由`[]`和一个个元素，通过`:`运算符连接起来的．`[1, 2, 3]`和`1:2:3:[]`是等价的．可以用`x:xs` 匹配长度大于等于1的list，`x:y:xs` 长度大于等于2．
- **Notice that if you want to bind to several variables (even if one of them is just _ and doesn't actually bind at all), we have to surround them in parentheses**，前面的a+b的匹配，相当于pair里的tie操作，把表达式和值绑定起来，这个操作一定要用`()` 包起来，否则会报错．然后如果pattern matching的时候其实有一些部分是我们不需要的，一般用`_`忽略，这里`_`就是没有被绑定的．看下面的例子．这

里的`x:[]`和`x:y:[]`就是pattern matching，所以一定要有`()`，如果写成`[x]`和`[x,y]`就不需要加括号了．但是最后的`x:y:_`没有不用pattern matching的表示，所以一定得有括号．

```
tell :: (Show a) => [a] -> String
tell [] = "The list is empty"
tell (x:[]) = "The list has one element: " ++ show x
tell (x:y:[]) = "The list has two elements: " ++ show x ++ " and " ++ show y

tell (x:y:_) = "This list is long. The first two elements are: " ++ show x +
+ " and " ++ show y
```

- `:`这个表达式，如果表示连接一个list，左边一定是一个元素，右边一定是一个list，所以`(_:xs)`这个pattern matching没有歧义，`_`只能match一个元素，而不能match一个list．
- 可以用`@`给一个pattern取一个名字(patterns)．语法是`name@(pattern)`．例如:`listLongerThan1@(x:_)`，以后`listLongerThan1`就和`(x:_)`没有区别了．**Normally we use as patterns to avoid repeating ourselves when matching against a bigger pattern when we have to use the whole thing again in the function body.** patterns的作用是在定义函数的时候避免重复输入一个较长的pattern．
- `++` 不能用在pattern matching．也就是不能`xs ++ [x, y, z]` 去作为一个pattern，匹配一个至少有三个元素的list．这是规定．
- `guards` 是一种和pattern matching类似的函数中的语法，但是他对输入的参数做一个bool表达式的判断，如果真,就命中．否则继续下一个`guard`．
- `otherwise`是`guard`中最后一个guard，用于捕捉所有的没有被前面的guard捕捉的case．`otherwise`前面也有一个`|`．如果没有`otherwise`，没有被捕捉到的话，处理方式是抛出异常．
- `guards` 的语法是函数定义的时候，先给声明，然后定义的第一行是函数名加参数，空格分隔，**然后没有`=`** 直接换行，后面是一些guard．每个guard以`|`开头，加上一个`bool`类型的断言，然后是`=`，然后是函数返回值．
- `where` 语法，可以在where中定义一些变量，计算公式，然后在guard等其他的断言中使用这些变量．如下面的Code．在一个函数的定义范围内定义的这些变量都是只对这个函数可见的．还有一个好处是，变量的只只会被算一次．

```haskell
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= skinny = "You're underweight, you emo, you!"
    | bmi <= normal = "You're supposedly normal. Pffft, I bet you're ugly!"

    | bmi <= fat    = "You're fat! Lose some weight, fatty!"
    | otherwise     = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
          skinny = 18.5
          normal = 25.0
          fat = 30.0
```

- `where`的缩进和pipeline的缩进相同，其中用到的变量要有相同的缩进层次，如果没有，Haskell就不知道是不是这个函数定义里面的了．所以haskell和python类似，也是缩进相关的．
- 可以在`where`语句中用pattern matching的语法．(pattern matching 就像一个绑定工具，把输入的参数的某些信息绑定到你想用的,你自己定义的变量上去)．

```haskell
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
    where (f:_) = firstname
          (l:_) = lastname
```

+`let binding in expression` 和`where`的作用类似，只不过写的顺序不同．先写变量的binding(类似定义的样子)，然后在写应用的表达式．注意如果有多个变量要binding，那么也要有同级规范的缩进．但是和`where`不同的地方是，`let ... in ...`本身是个表达式．但是`where`只是一个语法结构(syntactic constructs)．也就是`in expression`这里的表达式会算出一个值来返回．如下面的代码．

```haskell
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r ^2
    in  sideArea + 2 * topArea
```

- 感觉这个`let`有点bug，不太符合Haskell的类型推断的思想，因为它相当与在小范围里定义了一个函数(或者类似宏的东西)，这是没有声明类型的，也就是传错了参数，会RE．如下面的代码．

```haskell
ghci> [let square x = x * x in (square 5, square 3, square 2)]
[(25,9,4)]
```

+有时候`let`里面的binding不能做到缩进统一，只能写在一行，那么就得用`;`隔开.
+`Case Expression` case语句是一个对表达式的结果进行pattern matching的方式，方便我们对一个计算出来的结果进行匹配. (原来的pattern matching只能在函数定义的时候使用，有了case语句，使用场景大大增加)语法如下：

```
case expression of pattern -> result
                   pattern -> result
                   pattern -> result
                   ...
```