Haskell - P31

functor laws

1st: if fmap the id function over a functor, the functor that we get back should be the same as the original functor

也就是说当 fmap 函数将 id 函数作用于 f a 时，得到的返回值也是 f a

即有，fmap id = id，即 fmap id (f a) = id (f a) = f a

而 id :: a -> a，且有 id = \x -> x

即 id is identity function, just returns its parameter unmodified

证明如考虑应用于 Maybe 类型的情况

注意有 instance Functor Maybe where

```
fmap f (Just x) = Just (f x)
fmap _ Nothing  = Nothing
```

(trivial)

可见当 fmap id Nothing 时，总是有 fmap 返回 Nothing，则 fmap id = id 平凡地为真

而当 fmap id 作用于 (Just x) 时，根据定义有返回 Just (id x)

又 id x → x，即 fmap id (Just x) → Just x

于是可知 fmap id = id 为真

2nd: composing two functions and then fmapping the resulting function over a functor should be the same as first fmapping the righe function of (.) over the functor and then fmapping the left one of (.)

即有 fmap (f.g) = (fmap f).(fmap g)

或者 fmap (f.g) (f a) = (fmap f).(fmap g) $ (f a)

证明如考虑作用于Maybe类型的情况：

当 fmap (f.g) Nothing 时，平凡地返回 Nothing，

同时 fmap f (fmap g Nothing) → fmap f Nothing → Nothing

而当 fmap (f.g) (Just x) 时，根据定义返回 Just (f (g x))

同时 fmap f (fmap g (Just x)) → fmap f (Just (g x))
→ Just (f (g x))

于是可知 fmap (f.g) = (fmap f).(fmap g)

特别注意：有 instance of the Functor typeclass 但是并不是一个 functor (not obey the laws)

如 data CMaybe a = CNothing | CJust Int a deriving (Show)

```
instance Functor CMaybe where
  fmap _ CNothing = CNothing
  fmap f (CJust c x) = CJust (c+1) (f x)
```

注意：对于 1st fmap id (CJust 0 1) → CJust 1 1
id (CJust 0 1) → CJust 0 1

对于 2nd fmap ((+1).(*100)) (CJust 0 1) → CJust 1 101
fmap (+1) (fmap (*100) (CJust 0 1)) → CJust 2 101

特别注意 CMaybe 虽然是 typeclass Functor 的成员，但并不是一个合法的 functor

Haskell - P32

在 Haskell 中，函数是 curried by default 的，即函数每次传入一个参数并返回一个值或函数
即 如 函数 f :: a -> b -> c，虽然直观见上是传入类型为 a, b 的两个参数并返回类型为 c 的结果
但实际上是 首先函数 (a -> b -> c) 传入类型为 a 的参数，返回类型为 (b -> c) 的函数
再有函数 (b -> c) 传入类型为 b 的参数，返回类型为 c 的结果
所以在 Haskell 中 f x y 和 (f x) y 是等价的

对于 typeclass Functor 的成员 f，可以用 fmap :: (a -> b) -> (f a) -> (f b)
如果传入的函数需要传入多于一个参数，则将传入函数视为 a -> r 的形式
如 :t fmap (+) (Just 1) :: (Num a) => Maybe (a -> a)
:t fmap (\x y z -> x + y/z) [1,2,3,4] :: (Fractional a) => [a -> a -> a]
:t fmap compare "Maybe" :: [Char -> Ordering]
注意 fmap 直接提取最外层的 Functor 的成员 class
如 :t fmap compare (Just "Maybe") :: Maybe ([Char] -> Ordering)
也就是说 by mapping "multi-parameter" function over functor
get functor that contains function inside it
而且可以 fmap 以函数为参数的函数 在包含函数的 functor 上
如 let a = fmap (^) [1,2,3] -> a :: (Integral b, Num a) => [b -> a]
fmap (\f -> f 4) a -> [1, 16, 81]，即 [1^4, 2^4, 3^4]

但是注意：fmap 传入的参数函数必须是 normal function，而不能是 function inside functor
即 fmap 无法 map 一个 function inside functor (如 Just (*3))，到 functor value (如 Just (2)) 上

Applicative
为一个 typeclass，是 Functor 的加强版，即 applicative functors are beefed up functors
有定义为 class (Functor f) => Applicative (f :: * -> *) where
pure :: a -> (f a)
(<*>) :: (f (a -> b)) -> (f a) -> (f b)
] typeclass Applicative 主要支持的两个函数
(*>) :: (f a) -> (f b) -> (f b)
(<*) :: (f a) -> (f b) -> (f a)
GHC.Base.liftA2 :: (a -> b -> c) -> (f a) -> (f b) -> (f c)
{-# MINIMAL pure, ((<*>) | liftA2) #-}
由定义可知，如果一个 type constructor (f :: * -> *) 是 typeclass Applicative 的成员
则首先必须是 typeclass Functor 的成员，满足 class constraint (Functor f) =>
所以 对于 typeclass Applicative 的成员 f，也支持函数 fmap

Haskell - P33

pure :: a -> (f a)，指 typeclass Applicative 支持的第一个方法.
其中 a 为一个任意类型的 value，而于是 Applicative 的一个成员
pure 的作用是将传入的类型为 a 的值，装入 Applicative functor 之中
返回一个类型为 (f a) 的值
或者说 pure 描述了如何将一个给定类型的 value 装入一个给定的 Applicative functor
注意这个过程与 value constructor 生成给定类型值的过程类似
所以 pure 的定义可以直接使用或调用 constructor

<*> :: (f (a->b)) -> (f a) -> (f b)，注意这个定义与 fmap :: (a->b) -> (f a) -> (f b) 类似
于是可知 <*> 运算符是 fmap 函数的一个加强版本
<*> 传入两个参数，对于一个给定的 Applicative functor 的成员 f
一个装在 f 中类型为 a->b 的函数，和一个装在 f 中的类型为 a 的值
<*> 从 f 中提取(extract)函数，并 fmap 至 (f a) 参数上
并返回一个 (f b) 类型的值

pure 和 <*> 的具体行为是在定义一个 type 为 typeclass Applicative 的成员的定义的

```
instance Applicative Maybe where
    pure = Just                          ] 定义 pure 的行为
    Nothing <*> _ = Nothing              ] 定义 <*> 的行为
    (Just f) <*> x = fmap f x
```

对于 pure = Just，可以发现 pure 被直接定义为 Maybe 的 constructor，即 Just
也就是 pure x = Just x
对于 <*>，当试图从 Nothing 中提取函数 f 时，会直接返回 Nothing，而非返回 (f a)
当从 (Just f) 中提取到函数 f 时，则 fmap 函数到 f a 上

注意：:i <*> 有 infixl 4 <*>，即 <*> 是左结合的
对于普通函数，可以通过 function arg1 arg2 arg3 传入多个参数
<*> 也可以应用于对 Applicative functor 中的多参数函数传入多个参数
形如 pure f <*> x <*> y <*> z <*> ...
这里使用 pure f 的意义是 pure 会根据 type inference 的结果来将 f 装入 functor
基于 functor law 的规则，pure f <*> x 与 fmap f x 是等价的
所以 pure f <*> x <*> y <*> ... 可写作 fmap f x <*> y <*> ...
如 pure (+) <*> [1,2,3] <*> [3,4,5] ⟶ [4,5,6,5,6,7,6,7,8]
fmap (+) [1,2,3] <*> [3,4,5] ⟶ [4,5,6,5,6,7,6,7,8]

Haskell - P34

<$> :: (Functor f) => (a -> b) -> (f a) -> (f b)] 即<$>运算符实际上等价于fmap函数

f <$> x = fmap f x

所以 调用函数f在3个 Applicative functor上可以写作

f <$> x <*> y <*> z ，其中 f :: a->b ， x, y, z :: (Functor f) => (f a)

于是有 pure f <*> x <*> y <*> z ，fmap f x <*> y <*> z ，f <$> x <*> y <*> z 是等价的

即如 (++) <$> (Just "abc") <*> (Just "XYZ") ⟶ Just "abcXYZ"

pure (++) <*> (Just "abc") <*> (Just "XYZ") ⟶ Just "abcXYZ"

fmap (++) (Just "abc") <*> (Just "XYZ") ⟶ Just "abcXYZ"

形成相同的 Just ("abc"++) :: Maybe ([Char] -> [Char])

注意：在函数声明(function declaration)中的f是类型变量(type variable)

(Functor f)为一个class constraint，表示替换f的type constructor必须是Functor typeclass

在函数定义(function body)中的f是一个类型为 a->b 的函数

fmap f x 将函数f作用于类型为(f a)的参数x内部的值上

由此可知函数声明与函数定义中相同的变量名可以指向不同的对象

[] 是类型List的type constructor，也是Applicative functor的成员

```
instance Applicative [] where
  pure x = [x]
  fs <*> xs = [f x | f <- fs, x <- xs]
```

] 由于fs, xs都是list，所以可以直接使用到表生成器

注意 pure函数的行为取决于 default context，或者说，一个包含值的最小上下文

in other words, a minimal context that still yields that value

对于List类型，minimal context是空列表[]，

empty list represents the lack of a value, so it can't hold in itself the value used pure on

但是对于Maybe类型，minimal context是Nothing

但其 represents the lack of a value instead of a value

所以对于Maybe类型，pure的实现是通过Just

注意：<*> :: [a->b] -> [a] -> [b] 中利用了列表生成式(list comprehension)

所以每次应用<*>运算符都会构成一次新的列表

如 [(+), (*)] <*> [1,2] <*> [3,4] ⟶ [4,5,5,6,3,4,6,8]

[(+1), (+2), (*1), (*2)]

(++) <$> ["ha", "he", "hi"] <*> [".", "!", "?"]

⟶ ["ha.", "ha!", "ha?", "he.", "he!", "he?", "hi.", "hi!", "hi?"]