函数式语言程序设计

Applicative Functor

Functor 抽象的局限性

```
-- Functor 抽象给了我们操作『盒子』里的数据的能力
fmap :: Functor f => (a -> b) -> f a -> f b

-- 我们可否把这个能力扩展到多个盒子里的数据呢?
fmapII :: Functor f => (a -> b -> c) -> f a -> f b -> f c
fmapII = ???

fmapII f x y = (fmap f x) ??? y
```

How to compose?

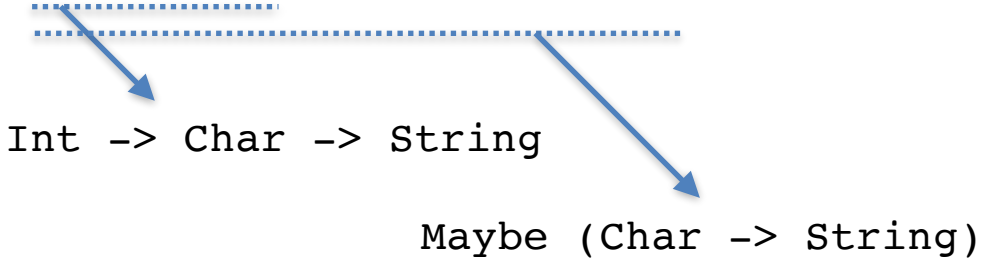
```
replicate :: Int -> a -> [a]
-- replicate 3 'x' === "xxx"

sometimes :: Maybe Int
maybeChar :: Maybe Char

replicate ??? sometimes maybeChar

replicateMaybes :: Maybe Int -> Maybe a -> Maybe [a]
replicateMaybes (Just x) (Just y) = Just (replicate x y)
replicateMaybes _        _        = Nothing
```

How to compose?

```
liftMaybe2 :: (a -> b -> c)
           -> Maybe a -> Maybe b -> Maybe c
liftMaybe2 f (Just x) (Just y) = Just (f x y)
liftMaybe2 _ _        _        = Nothing

-- liftMaybe replicate sometimes maybeChar
-- liftMaybe (+) (Just 0) (Just 1)

liftMaybe3 :: (a -> b -> c -> d)
           -> Maybe a -> Maybe b -> Maybe c -> Maybe d
liftMaybe3 = ???
liftMaybe3' :: (a -> b -> c -> d)
            -> Maybe a -> Maybe b -> c -> Maybe d
liftMaybe3' = ???
```

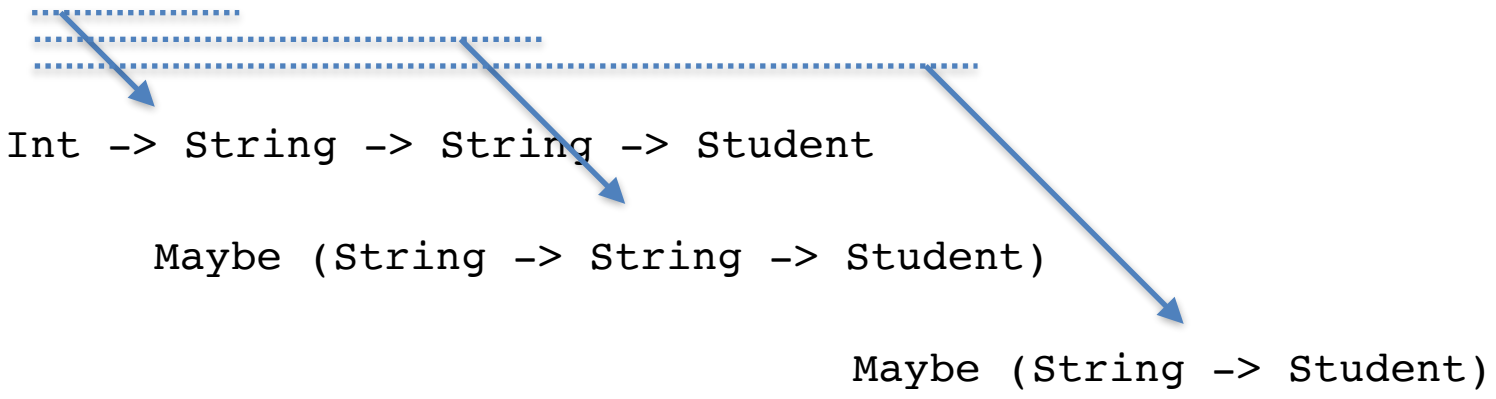
How to compose?

```
(<*>) :: Maybe (b -> c) -> Maybe b -> Maybe c
Just g <*> Just y = Just (g y)
_      <*> _      = Nothing
infixl 4 <*>

<$> = fmap
infixl 4 <$>

replicate <$> sometimes <*> maybeChar
```



How to compose?

```
data Student = Student
  { id :: Int, name :: String, department :: String }

Student <$> Just 0 <*> Just "Jeo" <*> Nothing
```



Introduce Applicative

```
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
    liftA2 :: (a -> b - > c) -> f a -> f b -> f c

    liftA2 f fa fb = f <$> fa <*> fb

    (<*>) = liftA2 id

instance Applicative Maybe where
    pure = Just
    Just f <*> Just x = Just (f x)
    _      <*> _      = Nothing
```

Compose lists

```
-- 现在考虑如何将函数列表 [b -> c] 和参数列表 [b] 组合起来

applyList :: [b -> c] -> [b] -> [c]
applyList (f:fs) ys = map f ys ++ applyList fs ys
applyList _      _  = []

applyList' :: [b -> c] -> [b] -> [c]
applyList' (f:fs) (y:ys) = f y : applyList' fs ys
applyList' _      _      = []
-- applyList' = zipWith ($)
```

Compose lists

```
instance Applicative [] where
    pure x = [x]
    f:fs <*> ys = map f ys ++ (fs <*> ys)
    _    <*> _  = []


newtype ZipList a = ZipList { getZipList :: [a] }

instance Applicative ZipList where
    pure x = ZipList (repeat x)  -- ZipList [x, x, x ...]
    ZipList fs <*> ZipList ys =
        ZipList (zipWith ($) fs ys)
    liftA2 f (ZipList xs) (ZipList ys) =
        ZipList (zipWith f xs ys)
```

Applicative Laws

```
-- identity
pure id <*> v = v

-- composition
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)

-- homomorphism
pure f <*> pure x = pure (f x)

-- interchange
u <*> pure y = pure ($ y) <*> u
```

The Reader Applicative

```
-- (->) r 代表的是需要 r 类型参数的计算过程
-- r -> a -> b 可以通过 r 得到 a -> b 的计算过程
-- r -> a 可以通过 r 得到 a

instance Applicative (->) r where
    -- pure :: a -> (r -> a)
    pure x = \ _ -> x
    -- (<*>) :: (r -> a -> b) -> (r -> a) -> (r -> b)
    f <*> x = \ r -> f r (x r)
```