函数式语言程序设计

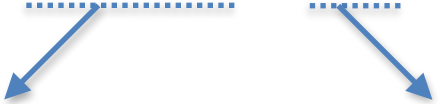
Monad is a Magic Word!

组合计算的方式

```
-- | 函数组合
(.) :: (b -> c) -> (a -> b) -> a -> c

-- | 0优先级的函数应用
($) :: (a -> b) -> a -> b

-- | 帮助把函数应用在函子里的数据上
(<$>) :: (a -> b) -> f a -> f b

-- | 把函子里的函数应用在函子里的数据上
(<*>) :: f (a -> b) -> f a -> f b
```

『上下文』的组合

```
instance Monoid w => Applicative (w,) where
    pure x = (mempty, x)
    (w1, f) <*> (w2, x) = (w1 <> w2, f x)
```



函子上下文的组合　　函子里的函数应用

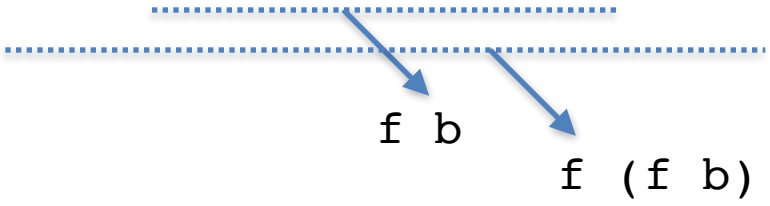
```
f = ("function need three params: ", \ x y z -> x+y+z)
x1 = ("x1 is 3, ", 3)
x2 = ("x2 is 4, ", 4)
x3 = ("x3 is 5, ", 5)

-- f <*> x1 <*> x2 <*> x3
-- ("function need three params: x1 is 3, x2 is 4, x3 is 5, ",12)
```

组合『上下文』？

```
-- data Maybe a = Just a | Nothing

Just f <*> Just x = Just (f x)  -- Just, Just -> Just
Nothing <*> _     = Nothing     -- Nothing, _ -> Nothing
_ <*> Nothing     = Nothing     -- _, Nothing -> Nothing


[(*10), (*100)] <*> [1,2,3] == [10,20,30,100,200,300]
```

组合『上下文』？

```
-- Maybe (Maybe a) -> Maybe Int
Just (Just 3) -> Just 3
Just Nothing -> Nothing

-- [[a]] -> [a]
[[1,2,3], [4,5,6]] -> [1,2,3,4,5,6]

join :: ??? f => f (f a) -> f a

(<*>) :: ??? f => f (a -> b) -> f a -> f b
ff <*> fx = join $ fmap (\ f -> fmap f fx) ff
```



组合『上下文』？

```
class Applicative m => Monad m where
    return :: a -> m a
    return = pure

    join :: m (m a) -> m a
    (>>=) :: m a -> (a -> m b) -> m b

instance Monad Maybe where
    -- join :: Maybe (Maybe a) -> Maybe a
    join (Just (Just x)) = Just x
    -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
    Just x >>= f = f x
    _      >>= _ = Nothing
```

组合『上下文』？

```
instance Monad [] where
    (>>=) :: [a] -> (a -> [b]) -> [b]
    xs >>= fs = concat (fs <$> xs)

join :: Monad m => m (m a) -> m a
join mma = mma >>= id

join [[1,2,3], [3,4,5]]
-- [[1,2,3],[3,4,5]] >>= id
-- concat (id <$> [[1,2,3],[3,4,5]])
-- concat [[1,2,3],[3,4,5]]
-- [1,2,3,4,5,6]
```

组合『上下文』？

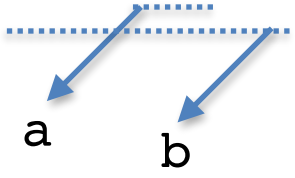
```
instance Monoid w => Monad (w,) where
    (>>=) :: Monoid w => (w, a) -> (a -> (w, b)) -> (w, b)
    (w1, x) >>= f = let (w2, y) = f x
                    in (w1 <> w2, y)
```

计算过程中产生的上下文

```
("Arg is 2, ", 2) >>= (\ x -> ("We plus it by 3, ", x+3))
       >>= (\ x -> ("Then we time it by 10.", 10*x))

-- ("First Arg is 2, We plus it by 3, ",5)
       >>= (\ x -> ("Then we time it by 10.", 10*x)
```

全局常量的传递

```
instance Monad (-> r) where
    (>>=) :: (r -> a) -> (a -> r -> b) -> r -> b
    f >>= g = \ r -> g (f r) r
```




```
(+1) >>= (*) >>= (-) $ 3
-- (3+1) * 3 - 3
-- 9
```

do语法糖

```
f :: Double -> Maybe Double
f x = do
    pure x
    y <- if x /= 0 then pure (100/x)
                   else Nothing
    pure (y * 10)

f x =
    pure x
        >>= (\ x -> if x /= 0 then pure (100/x)
                              else Nothing)
        >>= (\ y -> pure (y * 10))
```

do语法糖

```
g :: [Int]
g = do
    x <- [1,2,3]
    y <- [4,5,6]
    pure (x*y)

g =
   [1,2,3] >>= \ x ->
       [4,5,6] >>= \ y ->
           pure (x*y)
```

The Reader Monad

```
-- | 使用 newtype 封装 r -> a 类型的函数
newtype Reader r a = Reader { runReader :: r -> a }

instance Monad Reader where
    Reader f >>= Reader g = Reader (\ r -> g (f r) r)

runReader (Reader (+1) >>= (\ x -> Reader (x*))
                       >>= (\ y -> Reader (y-))) 3
-- runReader (Reader (\ r -> (r+1)*r)
                       >>= (\ y -> Reader (y-))) 3
-- runReader (Reader (\ r -> (r+1)*r) - r)) 3
-- (3+1)*3-3
-- 9
```

do语法糖

```
f :: Reader Double Double
f = do
    x <- Reader (+1)
    y <- Reader (x*)
    z <- Reader (y-)
    pure z

-- runReader f 3
-- (3+1)*3-3
-- 9
```