Haskell系列教程 III

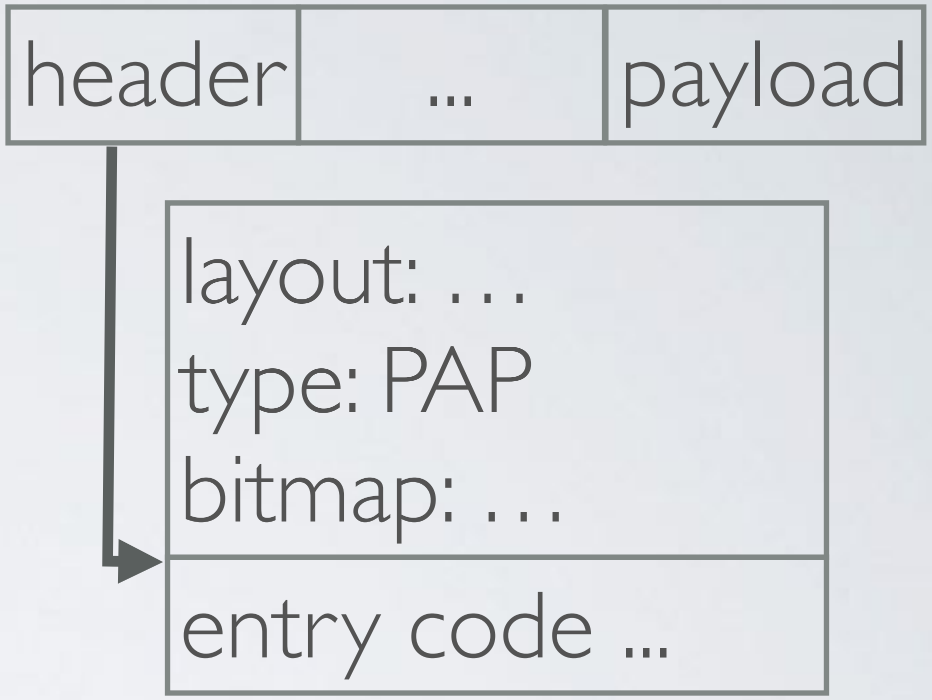
by 韩冬@滴滴FP

- Functor的定律
- 关于->的一些有趣的事实
- 文本解析 OOP版本 FP版本
- 控制结构
- IO 更多控制结构

Functor的定律

```
fmap :: Functor f => (a -> b) -> f a -> f b

-- fmap一定要保留functor的内部结构
fmap id  ==  id
fmap (f . g)  ==  fmap f . fmap g

fmap id (Just 3) == Just 3
fmap id Nothing == Nothing

fmap (even . (+1)) [1..]
== fmap even . fmap (+1) $ [1..]

-- 中缀版本fmap，$的升格版本，左结合
(<$>) :: (a -> b) -> f a -> f b
(<$>) = fmap
infixl 4 <$>
f <$> (g <$> (h <$> x)) == f . g . h <$> x
```

Reader




```
(->) r a -- r -> a的另一种写法
(->) r指的是接受r类型参数的函数
```

```
instance Functor ((->) r)
    -- fmap :: (a -> b) -> ((->) r a) -> ((->) r b)
    -- fmap :: (a -> b) -> (r -> a) -> (r -> b)
    fmap = (.)
```

```
fmap even (+1) :: Num a => a -> Bool    -- even . (+1)
fmap id (+1) == id (+1)
fmap not . fmap even $ (+1) == fmap (not . even) (+1)
```

Contravariant

```
class Contravariant f where
    contramap :: (a -> b) -> f b -> f a

newtype Op a b = Op { getOp :: b -> a }

instance Contravariant (Op r) where
    -- contramap :: (a -> b) -> Op b r -> Op a r
                                (b -> r)  (a -> r)
    contramap f g = Op (getOp g . f)

contramap (+1) even 1
-- fmap even (+1) 1
```

Endo

```
-- Endomorphisms自映射，这里指同一个类型之间的函数
newtype Endo a = Endo { appEndo :: a -> a }

data Monoid Endo where
    mempty = Endo id
    Endo f `mappend` Endo g = Endo f . g

Endo id <> Endo f == f
Endo f <> Endo id == f
(Endo f <> Endo g) <> Endo h
== Endo f <> (Endo g <> Endo h)

mconcat (map Endo [(+1), (*2), (^3)]) `appEndo` 3
-- 55
```

Functor的局限

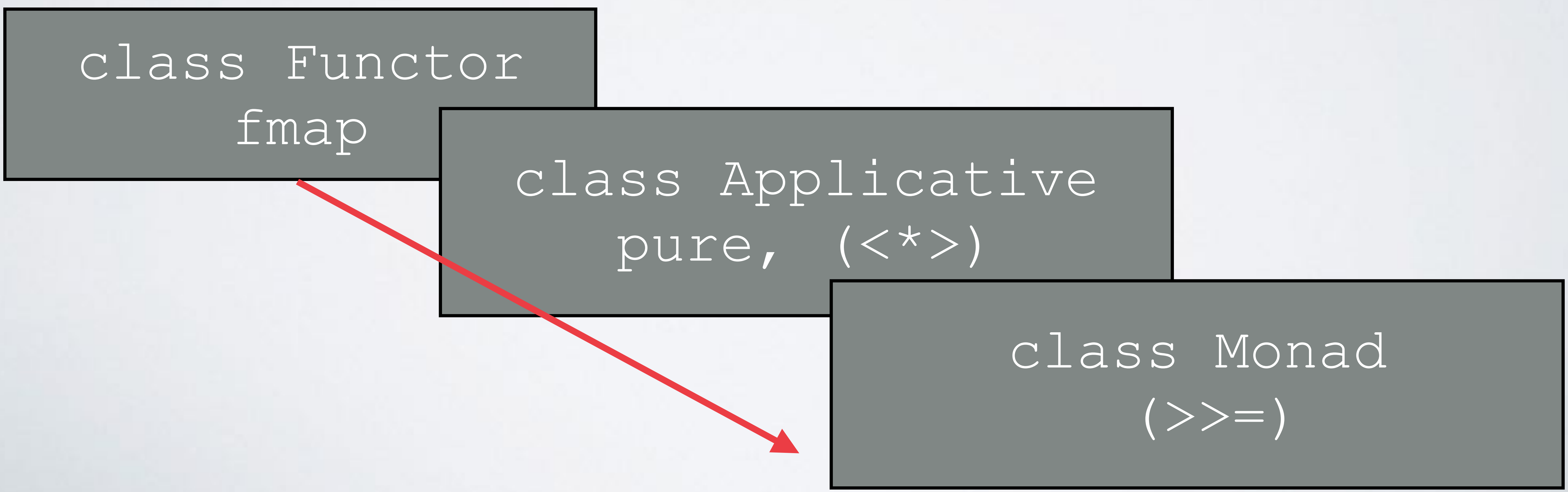
```
-- 读取环境变量，可能失败
env :: Maybe String -- env <- lookupEnv "PORT"

-- 解析数字，可能失败
readMaybe :: String -> Maybe Int

port = fmap readMaybe env ::  -- Maybe (Maybe Int)

-- 假设我们现在要使用到port
withPort :: Int -> ...
withPort ??? port

-- fmap (fmap withPort)      port
                       Maybe (Maybe Int)
```

Monad

```
class Monad m where
    return :: a -> m a  -- 赋予a一个“最简单”的函子包裹
    infixl 1 >>=
    (>>=) :: m a -> (a -> m b) -> m b
```

上一部分计算的结果　新的计算　合成后的计算结果

```
-------------------------------------------------
fmap :: Monad m => (a -> b) -> m a -> m b
fmap f ma = ma >>= return . f
```



Monad

```
instance Monad Maybe where
    -- return :: a -> Maybe a
    return x = Just x
    -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
    ma >>= f = case ma of Just a -> f a
                          Nothing -> Nothing
(>>=) :: Maybe String
      -> (String -> Maybe Int)
      -> Maybe Int
env >>= readMaybe :: Maybe Int
env >>= readMaybe >>= withPort


env >>= \ e ->
    readMaybe e >>= \ p ->
        return (p+1)
```

```
( do   -- 开启do语法糖
    e <- env
    p <- readMaybe e
    return (p+1)
) :: Maybe Int
```

Monad

```
instance Monad [] where
    -- return :: a -> [a]
    return x = [x]
    -- (>>=) :: [a] -> (a -> [b]) -> [b]
    x >>= f = mconcat (f <$> x)

"abc" >>= \ x ->
    "efg" >>= \ y ->
        [x,y]
```

```
do
    x <- "abc"
    y <- "efg"
    [x,y]
```

```
-- "aeafagbebfbgcecfcg"

"abc" >>= \ x -> "efg" >>= \ y -> [x,y]
mconcat [ "efg" >>= \ y -> ['a',y]
        , "efg" >>= \ y -> ['b',y]
        , "efg" >>= \ y -> ['c',y]
        ]
mconcat [ mconcat ["ae", "af", "ag"]
        , mconcat ["be", "bf", "bg"]
        , mconcat ["ce", "cf", "cg"]
        ]
```

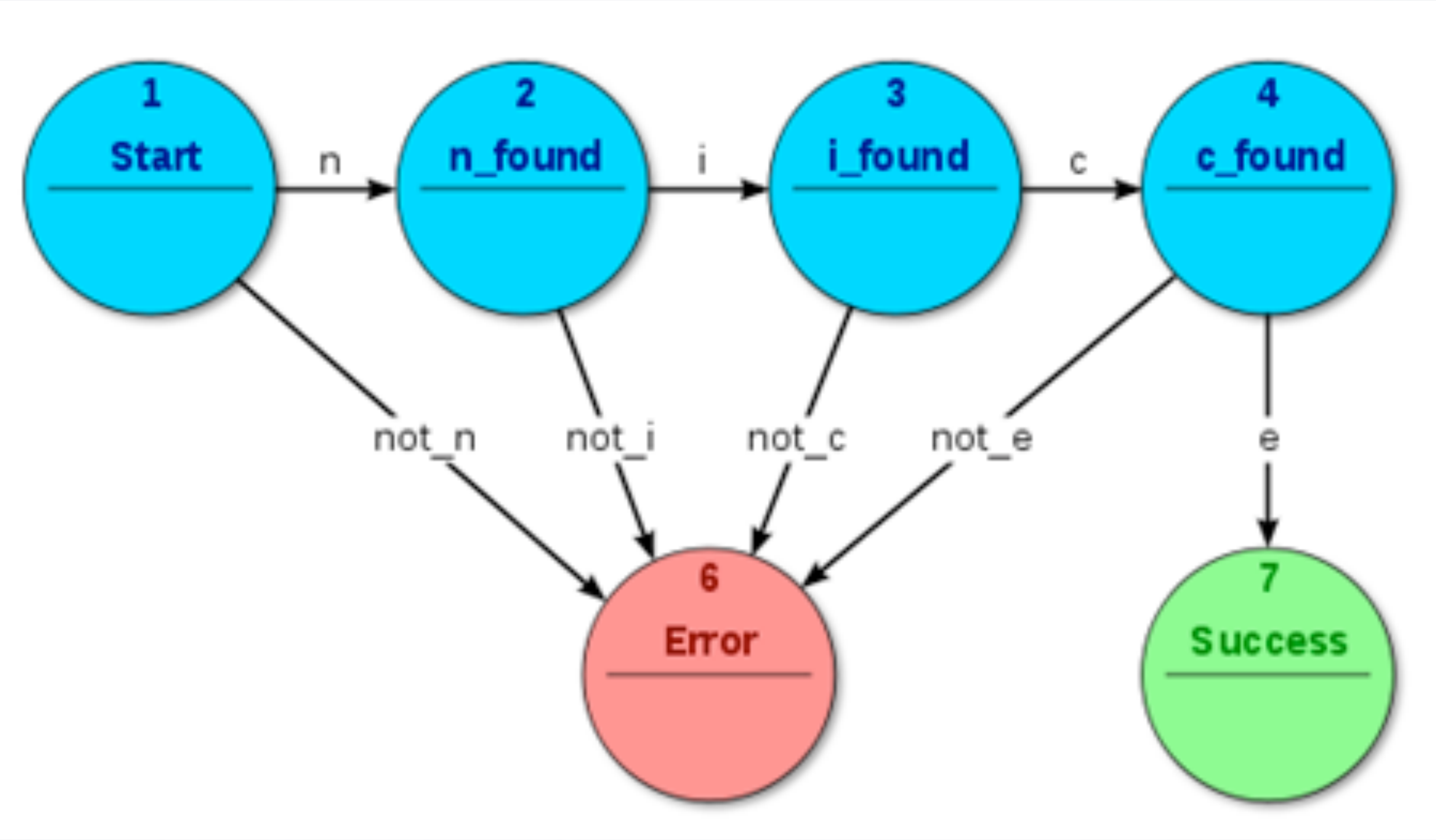
Monad的定律

```
左单位元:   return a >>= f ≡ f a
右单位元:   m >>= return ≡  m
结合律:  (m >>= f) >>= g ≡ m >>= (\x -> f x >>= g)

instance Monad ((->) r) where
    -- return :: a -> (r -> a)
    return x = const x
    -- (>>=) :: (r -> a) -> (a -> r -> b) -> (r -> b)
    fa >>= fb = \ r -> fb (fa r) r

return 1 >>= (+)   == \ x -> (+) (const 1 x) x
                   == \ x -> 1 + x
                   == (+1)

(+1) >>= return    == (+1) >>= \x -> const x
                   == \ x -> const (x + 1) x
                   == \ x -> x + 1
                   == (+1)
```

Monad的定律

```
(m >>= f) >>= g ≡ m >>= (\ x -> f x >>= g)

((+1) >>= (*)) >>= (^) = \ x -> (x + 1) * x  >>= (^)
                       = \ x -> ((x + 1) * x) ^ x

(+1) >>= (\ x -> (x*) >>= (^))
                       = (+1) >>= (\ y -> (x * y) ^ y)
                       = \ y -> ((y + 1) * y) ^ y

f = do
    a <- (+1)
    b <- (*2)
    c <- (^3)
    return (a + b + c)

f == \x -> (x+1) + (x*2) + (x^3)
```

文本解析



1
Start
2
n_found
3
i_found
4
c_found
n
i
c
not_n
not_i
not_c
not_e
e
6
Error
7
Success

文本解析

从左向右扫描输入的字符串，提取符合需要的数据，遇到不符合的情况报错。假设我们使用一门OOP语言X++(纯属虚构)，你可能会这么做：

```
class Parser
  constructor: func(buffer){
    this.buffer = buffer
  }
  char: func(c){
    if (this.buffer.length == 0){ throw "input ends"}
    if (this.buffer[0] == c){ this.buffer = this.buffer[1..]; }
    else { throw ("unexpected " + this.buffer[0]) }
  }
  digit: func(){
    if (this.buffer.length == 0){ throw "input ends"}
    if (this.buffer[0] >='0' || this.buffer[0] <= '9'){
      this.buffer = this.buffer[1..];
      return this.buffer[0] - '0';
    }
    else { throw "not a digit" }
  }
  ...
```

文本解析

```
class Parser
    char: ...
    digit: ...
    space: ...

// 扩展方式1，继承
class MyParser extends Parser
    bracketDigit: func(){
        parent.char('[')
        d = parent.digit()
        parent.char(']')
        return d;
    }

// 扩展方式2，MonkeyPatch
p = new Parser(buf)
p.bracketDigit = func() { ... }
```

问题

问题1：this丢失

```
class MyParser extends Parser
    bracket: func(p){             // p.bracket(p.digit)
        parent.char('[')
        d = p()  ??? where is this in p
        parent.char(']')
        return d;
    }
```

问题2：多继承?

```
class FooParser
    helper: ...
    foo: ...
class BarParser
    helper: ...
    bar: ...
class MyParser
    foobar:  ??? which helper
```

文本解析

```
data () = ()  -- 没有信息量的数据类型

digit :: String -> (Int, String)
digit []     = error "input ends"
digit (x:xs) =
  if x >= '0' || x <= '9' then (fromEnum x - 97, xs)
                          else error "not a number"
--------------------------------------------------
char :: Char -> String -> ((), String)
char _ [] = error "input ends"
char c (x:xs) =
  if c == x then ((), xs)
            else error ("unexpected " ++ [x])
--------------------------------------------------
bracketDigit :: String -> (Int, String)
bracketDigit input = let (_, input') = char '[' input
                         (d, input'') = digit input'
                         (_, input''') = char ']' input''
                     in (d, input''')
```

文本解析

```
newtype Parser a = Parser
    { runParser :: String -> (a, String) }

digitP :: Parser Int
digitP = Parser digit
charP :: Char -> Parser ()
charP c = Parser $ char c

instance Monad Parser where
    -- return :: a -> Parser a
    --           a -> String -> (a, String)
    return x = Parser $ \ input -> (x, input)

    -- (>>=) :: Parser a -> (a -> Parser b) -> Parser b
    -- (>>=) :: (String -> (a, String))
    --       -> (a -> String -> (b, String))
    --       -> String -> (b, String)
    (Parser f) >>= g = Parser $ \ input ->
                           let (a, input') = f input
                           in runParser (g a) input'
```

文本解析

```
bracketDigit :: Parser Int
bracketDigit = charP '[' >>= \ _ ->
                   digitP >>= \ d ->
                       charP ']' >>= \ _ ->
                           return d
```

```
(>>) :: m a -> m b -> m b
ma >> mb = ma >>= \ _ -> mb
```

```
Just 3 >> Just 4       == Just 4
Nothing >>= Just 4     == Nothing
```

```
bracketDigit :: Parser Int
bracketDigit = charP '[' >>
               digitP >>= \ d ->
               charP ']' >>
               return d
```

```
do
    charP '['
    d <- digitP
    charP ']'
    return d
```

文本解析

```
-- 解析任何Parser的括号版本
bracket :: Parser a -> Parser a
bracket p = do charP '['
               x <- p
               charP ']'
               return x

bracketDigit = bracket digitP
runParser bracketDigit "[8]"  -- 8
runParser bracketDigit "[!]"  -- error "not a number"

doubleBracketDigit = bracket (bracket digitP)
-- do charP '['
      charP '['
      x <- p
      charP ']'
      charP ']'
      return x
```

单子:计算模式的抽象

instance Monad Maybe

判断左侧的值是否为Nothing

•Nothing返回Nothing ===> Maybe b

•Just a则解包后传递给a -> Maybe b ===> Maybe b

instance Monad []

把新的计算a -> [b]应用到左侧[a]中的每一个元素

把[[b]]相连 ===> [b]

instance Monad ((->) r)

创建一个新的函数\ r -> ...

r经过左侧的r -> a得到a

a和r交给右侧的a -> r -> b ===> r -> b

instance Monad Parser

创建一个新的解析函数\ input -> ...

input经过左侧的Parser a得到a和剩余的input'

a和input'交给右侧的a -> Parser b ===> String -> (b, String)

优雅地报错

```haskell
data Either a b = Left a | Right b
-- 常常使用Either String a来标记可能失败的值，String记录原因

newtype Parser a = Parser
  { runParser :: String -> (Either String a, String) }

instance Monad Parser where
    -- return :: a -> Parser a
    return x = Parser $ \ input -> (Right x, input)
    -- (>>=) :: Parser a -> (a -> Parser b) -> Parser b
    (Parser f) >>= g = Parser $ \ input ->
         let (a, input') = f input
         in case a of
             -- 传递错误
             Left err -> (Left err, input')
             -- 传递计算结果
             Right a' -> runParser (g a') input'
```

优雅地报错

```
digitP :: Parser Int
digitP = Parser digit
  where
    digit []       = (Left "input ends", [])
    digit input@(x:xs) =
        if x >= '0' || x <= '9' then
        then (Right (fromEnum x - 97), xs))
        else (Left "not a number", input)

bracket :: Parser a -> Parser a
bracket p = do charP '['
               x <- p
               charP ']'
               return x

runParser (bracket digitP) "[!]"
-- (Left "not a number", "!]")
```

控制结构

```
replicateM :: Monad m => Int -> m a -> m [a]
replicateM 0 _  = return []
replicateM n ma = do a  <- ma
                     as <- replicateM (n - 1) ma
                     return (a:as)
--以[]为例
replicateM :: Int -> [a] -> [[a]]
replicateM 0 _ = [[]]
replicateM n ls = do a  <- ls
                     as <- replicateList (n - 1) ls
                     return (a:as)


replicateM 1 [1,2,3]
-- [[1],[2],[3]]
replicateM 2 [1,2,3]
-- [[1,1],[1,2],[1,3],[2,1],[2,2],[2,3],[3,1],[3,2],
[3,3]]
...
```

控制结构

```
--以Parser为例
replicateM :: Int -> Parser a -> Parser [a]
replicateM 0 _ = Parser []
replicateM n p = do a  <- p
                    as <- replicateList (n - 1) p
                    return (a:as)

p = replicateM 3 digit
runParser p "1234567"
-- Right ([1,2,3], "4567")
runParser p "1?"
-- Left ("not a number", "?")

toDecimal :: [Int] -> Int
toDecimal = foldl' (\acc x -> acc * 10 + x) 0

runParser (toDecimal <$> replicateM 3 digit) "1234567"
-- Right (123, "4567")
```

控制结构

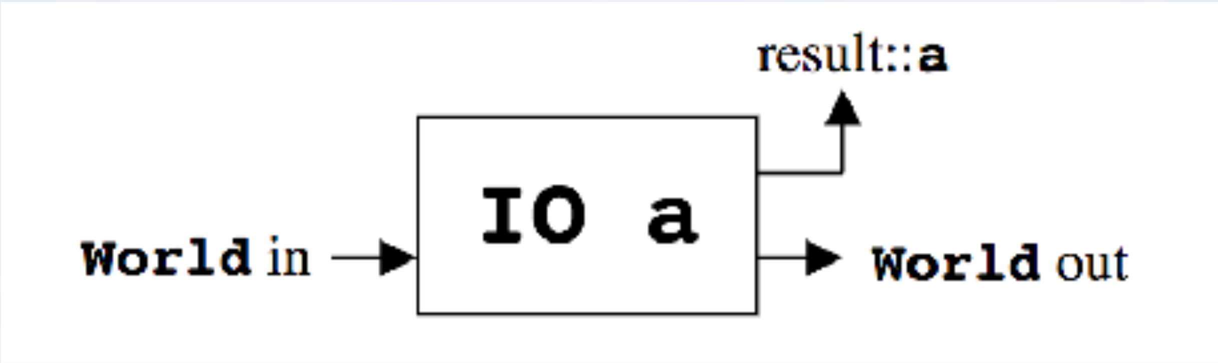
```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM _ [] = return []
mapM f (x:xs) = do b  <- f x
                   bs <- mapM f xs
                   return (b:bs)

mapM readMaybe ["123", "456", "closed"] :: Maybe [Int]
-- Nothing
map readMaybe ["123", "456", "closed"] :: [Maybe Int]
-- [Just 123, Just 456, Nothing]
----------------------------------------
mapM readMaybe ["123", "456", "closed"]
-- do b <- readMaybe "123"
      bs <- mapM readMaybe ["456", "closed"]
      return (b:bs)
...
-- do b1 <- readMaybe "123"
      b2 <- readMaybe "456"
      b3 <- readMaybe 'closed'
      return (b1:b2:b3:[])
```

控制结构

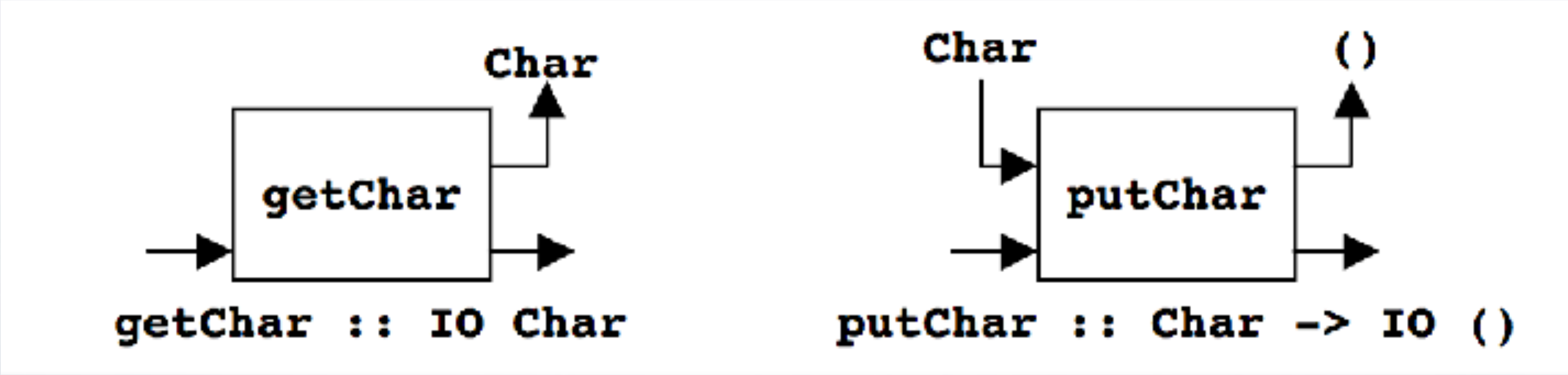
```
mapM charP "data"
-- do b <- charP 'd'
      bs <- mapM charP "ata"
      return (b:bs)

runParser (mapM charP "data") "data X = X"
-- (Right [(),(),(),()], " X = X")
runParser (mapM charP "data") "date 2016-11-01"
-- (Left "unexpected e", "e 2016-11-01")

mapM_ :: (a -> m b) -> [a] -> m ()
mapM_ _ [] = return ()
mapM_ f (x:xs) = f x >> mapM f xs

stringP = mapM_ charP :: Parser ()

runParser (stringP "data") "data X = X"
-- (Right (), " X = X")
```

IO - Tackling The Awkward Squad

```
main :: String -> String          -- 并行处理? ? ?
main :: [Response] -> [Request]   -- 不可扩展? ? ?

type IO a = World -> (a, World)
```



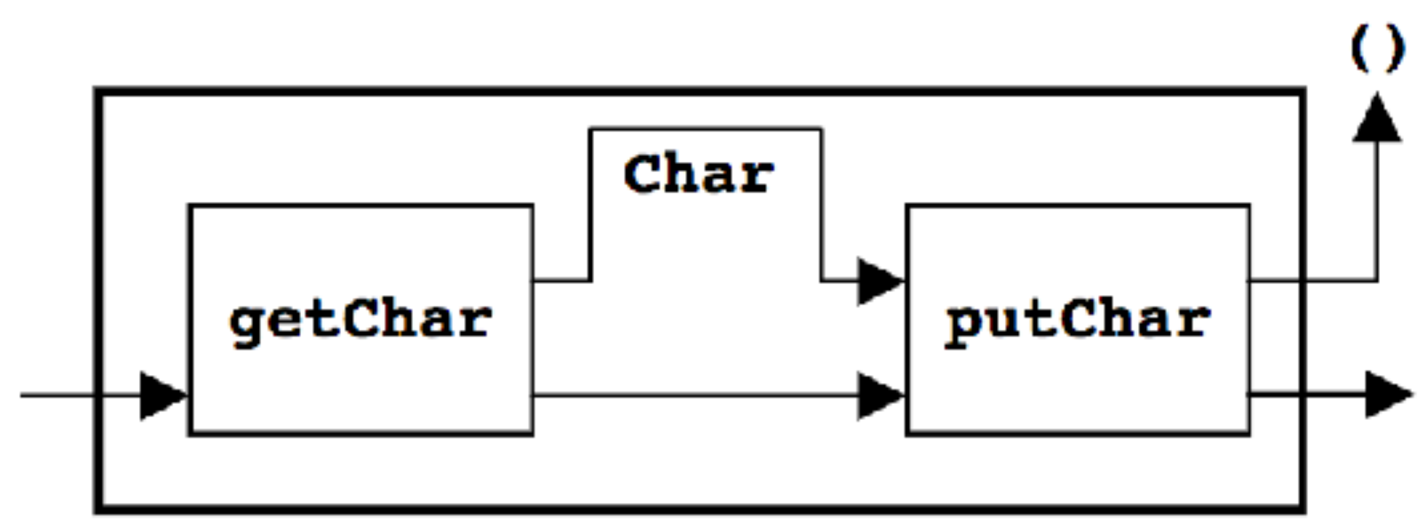

```
getChar :: IO Char
-- getChar :: World -> (Char, World)
putChar :: Char -> IO ()
-- putChar :: Char -> World -> ((), World)
```



IO Monad

```haskell
newtype IO a = IO { runIO :: World -> (a, World) }

instance Monad IO where
    ioA >> = f = IO $ \ w -> let (a, w') = runIO ioA w
                             in runIO (f a) w'

main :: IO ()  -- 在程序启动时传入一个World
main = do c <- getChar
          putChar c
```

World



getChar >>= putChar

更多控制结构

```
mapM_ putChar "abcd"
-- do putChar 'a'
      mapM_ putChar "bcd"
-- do putChar 'a'
      putChar 'b'
      mapM_ putChar "cd"
...

putStr :: String -> IO ()
putStr = mapM_ charP

forM = flip mapM :: [a] -> (a -> m b) -> m [b]
forM_ = flip mapM_ :: [a] -> (a -> m b) -> m ()

main = do
    ...
    forM_ [1..10] $ \ n ->
        ...
    ...
```

更多控制结构

```
when :: Bool -> m a -> m ()
when True ma = ma >> return ()
when False _ = return ()

unless = when . not

forever :: m a -> m b
forever ma = ma >> forever ma

filterM / foldM / zipWithM / ...
whileM / untilM / whileJust / unfoldM / andM / orM ...
----------------------------------------
when (x > 3) $ do
    ...

forever $ do
    ...
```