

Monad Law

Identity:

- return a >>= k = k a
- m >>= return = m

Associativity:

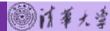
- $m >>= (\x -> k \x >>= h) = (m >>= k) >>= h$
- (f >=> g) >=> k = f >=> (g >=> k)

$$(>=>)$$
 :: Monad m => $(a -> m b) -> (b -> m c) -> a -> m c$ f >=> g = $\ x -> f x >>= g$



状态机的组合

```
newtype State s a = \{ \text{runState } :: s \rightarrow (s, a) \}
instance Functor (State s) where
     -- fmap :: (a -> b) -> State s a -> State s b
                   (a \rightarrow b) \rightarrow (s \rightarrow (s, a)) \rightarrow (s \rightarrow (s, b))
     fmap f m1 = State $ \setminus s0 \rightarrow let (s1, a) = runState m1 s0
                                         in (s1, f a)
instance Applicative (State s) where
     -- pure :: a -> State s a
                   a -> s -> (s, a)
     pure x = State (\ s \rightarrow (s, x))
     (<*>) :: State s (a -> b) -> State s a -> State s b
                (s \rightarrow (s, a \rightarrow b)) \rightarrow (s \rightarrow (s, a)) \rightarrow (s \rightarrow (s, b))
     mf < *> mg = State $ \setminus s0 -> let (s1, f) = runState mf s0
                                              (s2, x) = runState mg s1
                                         in (s2, fx)
```



状态机的组合

```
State s (State s a) ===> State s a
(s \rightarrow (s, (s \rightarrow (s, a)))
join :: (State s (State s a)) -> State s a
join f = State $ \setminus s0 \rightarrow
                let (s1, g) = runState f s0
                in runState q s1
instance Monad (State s) where
     (>>=) :: State s a -> (a -> State s b) -> State s b
                s \rightarrow (s, a) \rightarrow (a \rightarrow s \rightarrow (s, b)) \rightarrow (s \rightarrow (s, b))
    f >>= g = State $ \setminus s0 -> let (s1, a) = f s0
                                      in q a s1
```



状态机的组合

```
get :: State s s
modify :: (s \rightarrow s) \rightarrow State s ()
foo :: State Int (Int,Int,Int)
foo = do
   s1 <- get
   modify (+1)
   s2 <- get
   modify (+1)
   s3 <- get
   pure (s1, s2, s3)
-- runState foo 0 == (0,1,2)
```

```
-- 设计一个简单的数字解析器,可以解析科学记数法的数字
"314e3" ===> Just 314000
"abc" ===> Nothing
class Parser
   String input;
   digits: function () {
       Int result = 0;
       for (;this.input.length > 0;) {
           if (this.input[0] >= '0' && this.input[0] <= '9') {
               result = result * 10 + (this.input[0] - '0');
               this.input = this.input[1..]
           } else throw "not a digit!"
       return result;
   char: ...
   sci: ...
```

```
newtype Parser a = Parser
    { runParser :: String -> (String, Maybe a) }
digits :: Parser Int
digits = Parser $ \ input ->
    let r = takeWhile isDigit input
    in if null r
        then ([], Nothing)
        else (drop (length r) input, Just $
            foldl' (\acc a \rightarrow acc*10+(fromEnum a-48)) 0 r)
  where isDigit x = x \ge 0' \& x \le 9'
runParser digits "123e5" == ("e5", Just 123)
```



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

char :: Char -> Parser ()
char c = Parser $ \ input -> case input of
    (x:xs) | x == c -> (xs, Just ())
    _ -> (input, Nothing)

runParser (char 'e') "e5" == ("5", Just ())
runParser (char 'e') "abc" == ("abc", Nothing)
```



```
newtype Parser a = Parser
{ runParser :: String -> (String, Maybe a) }
instance Functor Parser where
fmap f (Parser p) = Parser $ \ input ->
let (input', ma) = p input
in (input', f <$> ma)
```



```
newtype Parser a = Parser
    { runParser :: String -> (String, Maybe a) }
instance Applicative Parser where
    pure x = Parser $ \ input -> (input, Just x)
    (<*>) = ap
instance Monad Parser where
    Parser pa >>= f = Parser $ \ input ->
        case pa input of
             (input', Just a) -> runParser (f a) input'
             (input', Nothing) -> (input', Nothing)
```



```
sci :: Parser Int
sci = do
    base <- digits</pre>
    char 'e'
    exp <- digits</pre>
    pure (base * 10^exp)
-- digits >>= \ base ->
       char 'e' >>
           digits >>= \ exp ->
                pure (base * 10^exp)
(>>) :: Monad m => m a -> m b -> m b
ma >> mb = ma >>= \ -> mb
```

Monad与控制结构

```
mapM :: (a -> m b) -> [a] -> m [b]
mapM f (x:xs) = do
     y < -f x
     ys <- mapM f xs
     return (y:ys)
mapM :: (a -> m b) -> [a] -> m ()
mapM f (x:xs) = f x >> mapM f xs
string :: String -> Parser ()
string str = mapM char str
-- runParser (string "hello") "hello world"
-- (" world", Just ())
```

Monad与控制结构

```
replicateM :: Int -> m a -> m [a]
replicateM n f
    | n > 0 = do
       x < - f
        xs <- replicateM (n-1) f
        return (x:xs)
    otherwise = return []
replicateM :: Int -> m a -> m ()
replicateM n f
     n > 0 = f >> replicateM_ (n-1) f
     otherwise = return ()
-- replicateM 3 [1,2,3] == [[1,1,1],[1,1,2],[1,1,3],[1,2,1],[1,2,2],
[1,2,3],[1,3,1],[1,3,2],[1,3,3],[2,1,1],[2,1,2],[2,1,3],[2,2,1],[2,2,2],
[2,2,3],[2,3,1],[2,3,2],[2,3,3],[3,1,1],[3,1,2],[3,1,3],[3,2,1],[3,2,2],
[3,2,3],[3,3,1],[3,3,2],[3,3,3]]
```

