Lecture 7 – Monad

- A program can be viewed as a function from values to computations (i.e. Monadic function).

```haskell
program :: a -> m b
```

- A computation is assigned a constructed type.

```haskell
m a

-- where m is a type constructor such as
--           Maybe, [], Tree, Map, and IO
```

- Operators for creating and composing functions from values to computations.

```haskell
return :: a -> m a
-- 'return' lifts a value into a computation

(<=<) :: (b -> m c) -> (a -> m b) -> (a -> m c)
-- f <=< g composes f and g
```

Category

- A category C consists of a class of objects `obj(C)` and a class of morphisms `hom(C)` between the objects.
- Set is a category, where objects are sets and morphisms are functions.

```
f :: a -> b
g :: b -> c
h :: c -> d

id : b -> b

h . (g . f)  =  (h . g) . f -- associativity

    id . f   =  f           -- left identity
     g . id  =  g           -- right identity
```

Category

- A category C consists of a class of objects `obj(C)` and a class of morphisms `hom(C)` between the objects.
- The natural category for interpreting programs is the Kleisli category, which is the category of monadic functions.

```
f :: a -> m b
g :: b -> m c
h :: c -> m d

return :: b -> m b

  h <=< (g <=< f)  =  (h <=< g) <=< f -- associativity

return <=< f       =  f               -- left identity
     g <=< return  =  g               -- right identity
```

Function and Monadic function

- Monadic function generalizes ordinary function.

```
-- identity
    id :: b -> b
return :: b -> m b

-- composition
   (.) :: (b -> c)   -> (a -> b)   -> (a -> c)
 (<=<) :: (b -> m c) -> (a -> m b) -> (a -> m c)

-- application
   ($) :: (a -> b)   -> a   -> b   -- (g  .  f) x = g $ f x
 (=<<) :: (a -> m b) -> m a -> m b -- (g <=< f) x = g =<< f x

-- inverse application
   (&) :: a   -> (a -> b)   -> b  -- import Data.Function
 (>>=) :: m a -> (a -> m b) -> m b
```

Monad class

- Monad type class defines an identity function and a bind function, though only bind function is required.

```
class Application m => Monad (m :: * -> *) where
  return :: a -> m                          -- identity
  return =  pure

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

   (>>)   :: m a -> m b -> m b            -- sequence
  m >> k =  m >>= \_ -> k
```

Maybe Monad

- Maybe type constructor is an instance of the Monad class

```
instance Monad Maybe where
    return x = Just x

    Nothing >>= f   =   Nothing
     Just x >>= f   =   f x

instance Applicative Maybe where
    pure x = Just x

    Just f  <*> Just a   = Just $ f a
    Nothing <*> _        = Nothing
          _ <*> Nothing = Nothing

instance Functor Maybe where
    fmap _ Nothing   = Nothing
    fmap f (Just x) = Just $ f x
```

Define a Log type

- A Log type allows us to record log strings during computation.

```
newtype Log a = Log { runLog :: (a, String) }

logger :: String -> Log ()

logger x = Log ((), x)
```

- A `Log a` value is essentially a tuple – `(a, String)`.
- To add a log string, we create a pair of the type – `Log ()` using the logger function.

Make Log type Functor

- fmap only applies f to the value inside Log while leaving the log string undisturbed.

```
instance Functor Log where

  fmap f m = Log (f a, l)

      where (a, l) = runLog m
```

- runLog unwraps the pair inside the Log type.

Make Log type Applicative

- pure simply creates a Log with empty string.

```haskell
instance Applicative Log where
  pure a = Log (a, "")
  f <*> v = Log (f_a f_v, l_f ++ "\n" ++ l_v)
      where (f_a, l_f) = runLog f
            (f_v, l_v) = runLog v
```

- <*> unwraps both Logs, applies their values, and appends their log strings.

Make Log type Monad

- 'return' function is just 'pure'

```
instance Monad Log where
  return a = Log (a, "") -- the same as 'pure'

  m >>= k = Log (b, l ++ "\n" ++ l')

    where (a, l)  = runLog m

          (b, l') = runLog $ k a
```

- >>= binds a Log monad with a function k that returns another Log monad.
- <*> can be implemented with >>=.

```
m_f <*> m_v = m_f >>= \f ->
              m_v >>= \v ->
              return $ f v
```

Using Log type to log computation

- calc function logs the start, the result, and the end of the computation, and then returns the final result in a Log monad.

```
calc :: Integer -> Log Integer

calc x = logger "start" >>
            let y = (sum [1..x])
            in logger ("sum from 1 to " ++ show x ++
                           " is: " ++ show y) >>
               logger "done" >>
               return y
```

Using Log type to log computation

- do notation is easier to read than using >>= and ‘return’.

```haskell
calc' :: Integer -> Log Integer

calc' x = do
           logger "start"
           let y = sum [1..x]
           logger ("sum from 1 to " ++ show x ++
                   " is: " ++ show y)
           logger "done"
           return y
```

Using Log type to log computation

- use 'runLog' to extract the value and the logged string from a Log monad.

```haskell
main :: IO ()
main = do
          let (y, l) = runLog $ calc 10
          let (y', l') = runLog $ calc' 20
          putStrLn l'
          putStrLn l
          putStrLn $ show (y', y)

-- start
-- sum from 1 to 10 is: 55
-- done

-- start
-- sum from 1 to 20 is: 210
-- done

-- (210,55)
```