

Lecture 9 – State Monad

- ▶ State monad maintains a mutable state so that the computation can get or update the state.

```
1 newtype State s a = State { runState :: s -> (a, s) }
```

- ▶ Reader monad maintains an environment that contains read-only values that can be read by the computation.

```
1 newtype Reader r a = Reader { runReader :: r -> a }
```

- ▶ Writer monad maintains a log to record information during the computation.

```
1 newtype Writer w a = Writer { runWriter :: (a, w) }
```

State as a Functor

► `newtype State s a = State { runState :: s -> (a, s) }`

► Define State as an instance of Functor

```
1 instance Functor (State s) where
2
3     fmap :: (a -> b) -> State a -> State b
4
5     fmap f p = State $ \s -> (f a, s')
6                     where (a, s') = runState p s
```

State as Applicative

▶ `newtype State s a = State { runState :: s -> (a, s) }`

▶ Define State as an instance of Applicative

```
1 instance Applicative (State s) where
2
3   pure    :: a -> State s a
4
5   pure x   = State $ \s -> (x, s)
6
7   (<*>)    :: State s (a->b) -> State s a -> State s b
8
9   p <*> q = State $ \s -> (f a, s2)
10                      where (f, s1) = runState p s
11                      (a, s2) = runState q s1
```

State as a Monad

▶ `newtype State s a = State { runState :: s -> (a, s) }`

▶ Define State as an instance of Monad

```
1 instance Monad (State s) where
2
3   return :: a -> State s a
4
5   return x = State $ \s -> (x, s)
6
7   (>>=) :: State s a -> (a -> State s b) -> State s b
8
9   p >>= k = State $ \s0 -> (x2, s2)
10                      where (x1, s1) = runState p      $ s0
11                      (x2, s2) = runState (k x1) $ s1
```

MonadState

▶ `newtype State s a = State { runState :: s -> (a, s) }`

▶ MonadState class defines interface for State Monad

```
1  -- s is the state type
2  class Monad m => MonadState s (m :: * -> *) | m -> s where
3      -- retrieve state
4      get :: m s
5
6      -- set new state
7      put :: s -> m ()
8
9      -- create a state monad
10     state :: (s -> (a, s)) -> m a
11
12     {-# MINIMAL state | get, put #-}
```

State as a MonadState

▶ `newtype State s a = State { runState :: s -> (a, s) }`

▶ MonadState class defines interface for State Monad

```
1 -- s is the state type
2 instance MonadState s (State s) where
3   -- retrieve state
4   get :: m s
5   get = State $ \s -> (s, s)
6
7   -- set new state
8   put :: s -> m ()
9   put s = State $ \_ -> ((), s)
10
11  -- create a state monad
12  state :: (s -> (a, s)) -> m a
13  state = State
```

Pretty printer example

```
1 data Term = Const Integer      | Var String
2           | Plus Term Term     | Times Term Term
3           | LE Term Term       | IF Term Term Term
4           | App Term Term      | Fn (String, Term)
5           | Fun (String, String, Term)
```

► Pretty printer for functions.

```
1 pp0 :: Term -> String -> String
2
3 pp0 (IF t0 t1 t2) space =
4   let t = pp0 t1 (space ++ "\t")
5       e = pp0 t2 (space ++ "\t")
6   in space ++ "if " ++ show t0 ++ "\n" ++ space ++
7       "then\n" ++ t ++ "\n" ++ space ++
8       "else\n" ++ e
9
10 pp0 (Fun (f, x, t)) space =
11   let body = pp0 t (space ++ "\t")
12   in space ++ "fun " ++ f ++ " " ++ x ++ " =\n" ++ body
13
14 pp0 x space = space ++ show x
```

Pretty printer with line numbers

```
1 data Term = Const Integer      | Var String
2           | Plus Term Term     | Times Term Term
3           | LE Term Term       | IF Term Term Term
4           | App Term Term      | Fn (String, Term)
5           | Fun (String, String, Term)
```

► Line number is a mutable state.

```
1 pp2 :: Int -> Term -> String -> (String, Int)
2
3 pp2 line (IF t0 t1 t2) space =
4     let (t, line1) = pp2 (line+2) t1 (space ++ "\t")
5         (e, line2) = pp2 (line1+1) t2 (space ++ "\t")
6     in (show line      ++ space ++ "if " ++ show t0 ++ "\n" ++
7         show (line+1) ++ space ++ "then\n" ++ t ++ "\n" ++
8         show (line1)  ++ space ++ "else\n" ++ e, line2)
9
10 pp2 line (Fun (f, x, t)) space =
11     let (body, line') = pp2 (line+1) t (space ++ "\t")
12     in (show line ++ space ++ "fun " ++ f ++ " " ++ x ++
13         " =\n" ++ body, line')
14
15 pp2 line x space = (show line ++ space ++ show x, line+1)
```


Pretty printer with line numbers

- ▶ Line number is a mutable state.

```
1 pp2 :: Int -> Term -> String -> (String, Int)
2
3 main :: IO ()
4 main = do
5     let fact = Fun ("fact", "x",
6                     IF (LE (Var "x") (Const 1))
7                         (Const 1)
8                         (Times (Var "x")
9                               (App (Var "fact")
10                                   (Plus (Var "x")
11                                         (Const (-1))))))
12     putStrLn $ fst $ pp2 1 fact " "
13
14 -- 1 fun fact x =
15 -- 2     if (x <= 1)
16 -- 3     then
17 -- 4         1
18 -- 5     else
19 -- 6         (x * (fact (x + -1)))
```

Pretty printer using State Monad

```
1 pp2' :: Term -> String -> State Int String
2
3 pp2' (IF t0 t1 t2) space = do
4   -- print then part
5   line <- get      -- read line number for if expression
6   put $ line+2     -- increment line number for then part
7   t <- pp2' t1 (space ++ "\t")
8
9   -- print else part
10  line1 <- get     -- read line number after then part
11  put $ line1+1    -- increment line number for else part
12  e <- pp2' t2 (space ++ "\t")
13
14  -- print if/then/else
15  return $ (show line) ++ space ++ "if " ++ show t0 ++ "\n" ++
16           (show $ line+1) ++ space ++ "then\n" ++ t ++ "\n" ++
17           (show $ line1) ++ space ++ "else\n" ++ e
```

Pretty printer using State Monad

```
1 pp2' :: Term -> String -> State Int String
2
3 pp2' (Fun (f, x, t)) space = do
4
5     -- print function body
6     line <- get      -- read line number for the function
7     put $ line+1    -- increment line number for the body
8     body <- pp2' t (space ++ "\t")
9
10    -- print the function
11    return $ (show line) ++ space ++ "fun " ++ f ++ " " ++ x ++
12            " =\n" ++ body
```

Pretty printer using State Monad

```
1 pp2' :: Term -> String -> State Int String
2
3 pp2' x space = do
4   line <- get      -- read line number for the expression
5   put $ line+1    -- increment line number
6
7   -- print the expression
8   return $ (show line) ++ space ++ show x
```

Pretty printer with line numbers

► run state with initial line number 1

```
1 pp2 :: Int -> Term -> String -> (String, Int)
2
3 main :: IO ()
4 main = do
5     let fact = Fun ("fact", "x",
6                     IF (LE (Var "x") (Const 1))
7                         (Const 1)
8                         (Times (Var "x")
9                               (App (Var "fact")
10                                   (Plus (Var "x")
11                                         (Const (-1))))))
12     putStrLn $ fst $ (runState $ pp2' fact " ") 1
13
14 -- 1 fun fact x =
15 -- 2     if (x <= 1)
16 -- 3     then
17 -- 4         1
18 -- 5     else
19 -- 6         (x * (fact (x + -1)))
```

State Transformer

- ▶ `(StateT s)` transforms a monad `m` into a State Monad `(StateT s m)`

```
1 newtype StateT s (m :: * -> *) a
2   = StateT {runStateT :: s -> m (a, s)}
```

- ▶ State is defined with `StateT`.

```
1 type State s = StateT s Identity :: * -> *
```

- ▶ Identity monad is really just identity.

```
1 module Data.Functor.Identity
2
3 newtype Identity a = Identity {runIdentity :: a}
```

Pretty printer using StateT

```
1 pp3 :: Term -> StateT Int (ReaderT String (Writer String)) ()
2
3 pp3 (IF t0 t1 t2) = do
4   line <- get      -- get line number
5   space <- ask     -- get space
6   tell $ (show line) ++ space ++ "if " ++ show t0 -- print if
7
8   -- print then
9   tell $ "\n" ++ (show $ line+1) ++ space ++ "then\n"
10  put $ line+2 -- set line number
11  tab $ pp3 t1 -- print then part
12  line1 <- get -- get line number
13
14  -- print else
15  tell $ "\n" ++ (show $ line1) ++ space ++ "else\n"
16  put $ line1+1 -- set line number
17  tab $ pp3 t2 -- print else part
18
19  where tab = local (\s -> s ++ "\t")
```

Pretty printer using StateT

```
1 pp3 :: Term -> StateT Int (ReaderT String (Writer String)) ()
2
3 pp3 (Fun (f, x, t)) = do
4   line <- get      -- get line number
5   space <- ask     -- get space
6
7   -- print function
8   tell $ (show line) ++ space ++
9         "fun " ++ f ++ " " ++ x ++ " =\n"
10
11   put $ line+1     -- set line number
12   tab $ pp3 t      -- print function body
13
14 where tab = local (\s -> s ++ "\t")
```


Pretty printer using StateT

```
1 pp3 :: Term -> StateT Int (ReaderT String (Writer String)) ()
2
3 pp3 x = do
4   line <- get      -- get line number
5   space <- ask     -- get space
6
7   -- print the expression
8   tell $ (show line) ++ space ++ show x
9
10  put $ line+1     -- set line number
```

Pretty printer with line numbers

► run state with initial line number 1

```
1 pp3 :: Term -> StateT Int (ReaderT String (Writer String)) ()
2
3 main :: IO ()
4 main = do
5     let fact = Fun ("fact", "x",
6                     IF (LE (Var "x") (Const 1))
7                         (Const 1)
8                         (Times (Var "x")
9                               (App (Var "fact")
10                                    (Plus (Var "x")
11                                            (Const (-1))))))
12     -- 1. run StateT, 2. run ReaderT, 3. run Writer,
13     putStrLn $ snd $ runWriter $
14         runReaderT
15             (runStateT (pp3 fact) 1) " "
16 -- 1 fun fact x =
17 -- 2     if (x <= 1)
18 -- 3     then
19 -- 4         1
20 -- 5     else
21 -- 6         (x * (fact (x + -1)))
```