Lecture 8 – Reader, Writer, and State Monad

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

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

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

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

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

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

Example with an interpreter

- Implement an interpreter for a nano language.

```
1  -- abstract syntax
2  data Term = Const Integer
3            | Var String
4            | Plus Term Term
5            | Times Term Term
6            | LE Term Term
7            | IF Term Term Term
8            | App Term Term
9            | Fn (String, Term)
10           | Fun (String, String, Term) deriving (Show)
```

- Concrete syntax.

$$
\begin{array}{rcl}
t & ::= & n \mid x \mid t + t \mid t * t \mid t \leq t \\
& \mid & \text{if } t_0 \text{ then } t_1 \text{ else } t_2 \\
& \mid & t_1 \ t_2 \\
& \mid & \text{fn } x \Rightarrow t \\
& \mid & \text{fun } f \ x \ = \ t
\end{array}
$$

Example with an interpreter

- Implement an interpreter for a nano language.

```
-- abstract syntax
data Term = Const Integer
          | Var String
          | Plus Term Term
          | Times Term Term
          | LE Term Term
          | IF Term Term Term
          | App Term Term
          | Fn (String, Term)
          | Fun (String, String, Term) deriving (Show)
```

- Concrete syntax.

```
a = Plus (Const 10) (Const 20)}                  -- 10 + 20
f = Fn ("x", Plus(Var "x") (Const 5))} -- fn x => x + 5
t = App f a                        -- (fn x => x + 5) (10 + 20)
```

Example with an interpreter

- Implement an interpreter for a nano language.

```
-- abstract syntax
data Term = Const Integer
          | Var String
          | Plus Term Term
          | Times Term Term
          | LE Term Term
          | IF Term Term Term
          | App Term Term
          | Fn (String, Term)
          | Fun (String, String, Term) deriving (Show)
```

- Concrete syntax.

```
let fact = Fun ("fact", "x",
              IF (LE (Var "x") (Const 1))
                 (Const 1)
                 (Times (Var "x")
                        (App (Var "fact")
                             (Plus (Var "x") (Const (-1))))))
-- fun fact x = if x <= 1 then 1 else x * fact (x - 1)
```

Example with an interpreter

- Implement an interpreter for a nano language.

```
1  -- abstract syntax
2  data Term = Const Integer
3            | Var String
4            | Plus Term Term
5            | Times Term Term
6            | LE Term Term
7            | IF Term Term Term
8            | App Term Term
9            | Fn (String, Term)
10           | Fun (String, String, Term) deriving (Show)
```

- Define a type for variable context and a type for results.

```
1  type Context = [(String, Val)]
2
3  data Val = IntVal Integer
4           | BoolVal Bool
5           | FVal (Maybe String, String, Term) Context
6             deriving (Show)
```

Example with an interpreter

- Define a type for variable context and a type for results.

```
type Context = [(String, Val)]

data Val = IntVal Integer
         | BoolVal Bool
         | FVal (Maybe String, String, Term) Context
             deriving (Show)
```

- Evaluate simple terms

```
lookup :: String -> Context -> Maybe Val

lookup x [] = Nothing          -- lookup variable in context
lookup x ((y, v) : xs)
  | x == y     = Just v
  | otherwise = lookup x xs

eval :: Term -> Context -> Maybe Val

eval (Var x) ctx = lookup x ctx          -- evaluate variable
```

Example with an interpreter

- Define a type for variable context and a type for results.

```haskell
type Context = [(String, Val)]

data Val = IntVal Integer
         | BoolVal Bool
         | FVal (Maybe String, String, Term) Context
             deriving (Show)
```

- Evaluate simple terms

```haskell
eval :: Term -> Context -> Maybe Val

eval (Const x) _ = Just x     -- evaluate constant

eval (Plus t1 t2) ctx = do    -- evaluate plus
    v1 <- eval t1 ctx
    case v1 of                -- t1 has to be an int
      (IntVal c1) -> do
         v2 <- eval t2 ctx
         case v2 of           -- t2 has to be an int
            (IntVal c2) -> Just $ IntVal (c1+c2)
            _ -> Nothing
      _ -> Nothing
```

Example with an interpreter

- Recap Maybe Monad

```haskell
instance Monad Maybe where
   return a = Just a

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

instance MonadFail Maybe where
   fail _ = Nothing -- called if <- fails to pattern match
```

- Evaluate simple terms

```haskell
eval :: Term -> Context -> Maybe Val

eval (Const x) _ = Just x     -- evaluate constant

eval (Plus t1 t2) ctx = do    -- evaluate plus
    v1 <- eval t1 ctx
    case v1 of                -- t1 has to be an int
      (IntVal c1) -> do
          v2 <- eval t2 ctx
          case v2 of          -- t2 has to be an int
             (IntVal c2) -> Just $ IntVal (c1+c2)
             _ -> Nothing
      _ -> Nothing
```

Example with an interpreter

- Recap Maybe Monad

```haskell
instance Monad Maybe where
   return a = Just a

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

instance MonadFail Maybe where
   fail _ = Nothing -- called if <- fails to pattern match
```

- Simplify case analysis based on default 'fail' behavior

```haskell
eval :: Term -> Context -> Maybe Val

eval (Const x) _ = Just x     -- evaluate constant

eval (Plus t1 t2) ctx = do
   (IntVal c1) <- eval t1 ctx -- if fails, return Nothing
   (IntVal c2) <- eval t2 ctx -- if fails, return Nothing
   return $ IntVal (c1 + c2)  -- pattern matching succeeds
```

Example with an interpreter

- Evaluate simple terms

```
1  eval :: Term -> Context -> Maybe Val
2
3  eval (Times t1 t2) ctx = do      -- evaluate times
4     (IntVal c1) <- eval t1 ctx
5     (IntVal c2) <- eval t2 ctx
6     return $ IntVal (c1 * c2)
7
8  eval (Plus t1 t2) ctx = do       -- evaluate plus
9     (IntVal c1) <- eval t1 ctx
10    (IntVal c2) <- eval t2 ctx
11    return $ IntVal (c1 + c2)
12
13 eval (LE t1 t2) ctx = do         -- evaluate less/than/equal
14    (IntVal c1) <- eval t1 ctx
15    (IntVal c2) <- eval t2 ctx
16    return $ BoolVal (c1 <= c2)
17
18 eval (IF t0 t1 t2) ctx = do      -- evaluate if/then/else
19    (BoolVal b) <- eval t0 ctx
20    if b then eval t1 ctx else eval t2 ctx
```

Example with an interpreter

- Evaluate function value and function call.

```
1  eval :: Term -> Context -> Maybe Val
2
3  eval (App t1 t2) ctx = do
4      -- evaluate function value
5      fun@(FVal (f, x, t0) ctx0) <- eval t1 ctx
6
7      -- evaluate argument value
8      arg <- eval t2 ctx
9
10     -- check whether the function is named
11     let ctx = case f of Just name -> [(name, fun)]
12                         Nothing -> []
13
14     -- evaluate function body with new context
15     eval t0 $ ctx ++ (x, arg) : ctx0
16
17 -- anonymous function
18 eval (Fn (x, t))      ctx = return $ FVal (Nothing, x, t) ctx
19 -- named function
20 eval (Fun (f, x, t)) ctx = return $ FVal (Just f, x, t) ctx
```

Example with an interpreter

- Do some tests

```
eval :: Term -> Context -> Maybe Val

main :: IO ()
main = do
          let x = Plus (Const 10) (Const 20)
          let f = Fn ("x", Plus(Var "x") (Const 5))
          print $ eval (App f x) []    -- Just (IntVal 35)

          let fact = Fun ("fact", "x",
                          IF (LE (Var "x") (Const 1))
                             (Const 1)
                             (Times (Var "x")
                                    (App (Var "fact")
                                         (Plus (Var "x")
                                               (Const (-1))))))
          let f10 = App fact (Const 10)
          print $ eval f10 []          -- Just (IntVal 3628800)
```

Example with an interpreter

- Avoid passing context parameter with reader monad (hand-rolled version).

```
1  newtype Result r a =  Result {runResult :: r -> Maybe a}
2
3  instance Functor (Result r) where
4     fmap f (Result g) = Result (\ctx -> fmap f $ g ctx)
5
6  instance Applicative (Result r) where
7     pure x = Result (\_ -> pure x)
8     Result f <*> Result a = Result (\ctx -> f ctx <*> a ctx)
9
10 instance Monad (Result r) where
11    Result f >>= k = Result $ \ctx -> do
12                                        a <- f ctx
13                                        (runResult $ k a) ctx
14
15 instance Fail.MonadFail (Result r) where
16    fail s = Result $ \_ -> Fail.fail s
```

Example with an interpreter

- eval function no longer takes context explicitly.

```haskell
newtype Result r a =  Result {runResult :: r -> Maybe a}

eval :: Term -> Result Context Val

eval (Const a) = return $ IntVal a

eval (Var s) = Result $ \ctx -> lookup s ctx
```

Example with an interpreter

- eval function no longer takes context explicitly.

```
1  newtype Result r a =  Result {runResult :: r -> Maybe a}
2
3  eval :: Term -> Result Context Val
4
5  eval (Times t1 t2) = do
6     (IntVal c1) <- eval t1
7     (IntVal c2) <- eval t2
8     return $ IntVal (c1 * c2)
9
10 eval (Plus t1 t2) = do
11    (IntVal c1) <- eval t1
12    (IntVal c2) <- eval t2
13    return $ IntVal (c1 + c2)
14
15 eval (LE t1 t2) = do
16    (IntVal c1) <- eval t1
17    (IntVal c2) <- eval t2
18    return $ BoolVal (c1 <= c2)
19
20 eval (IF t0 t1 t2) = do
21    (BoolVal b) <- eval t0
22    if b then eval t1 else eval t2
```

Example with an interpreter

- eval function no longer takes context explicitly.

```haskell
newtype Result r a =  Result {runResult :: r -> Maybe a}

eval :: Term -> Result Context Val

eval (App t1 t2) = do
    fun@(FVal (f, x, t0) ctx0) <- eval t1 -- eval function

    arg <- eval t2                        -- eval argument

    -- check whether the function is named
    let ctx = case f of Just name -> [(name, fun)]
                        Nothing   -> []

    -- evaluate function body with new context
    Result $ \_ ->
              runResult (eval t0) $ ctx ++ (x, arg) : ctx0

eval (Fn (x,t)) = Result (\ctx ->
                            return $ FVal (Nothing, x, t) ctx)

eval (Fun (f,x,t)) = Result (\ctx ->
                            return $ FVal (Just f, x, t) ctx)
```

Example with an interpreter

- Do some tests

```
eval :: Term -> Context -> Maybe Val

main :: IO ()
main = do
          let x = Plus (Const 10) (Const 20)
          let f = Fn ("x", Plus(Var "x") (Const 5))
          print $ runResult (eval (App f x)) []
          -- Just (IntVal 35)

          let fact = Fun ("fact", "x",
                          IF (LE (Var "x") (Const 1))
                             (Const 1)
                             (Times (Var "x")
                                    (App (Var "fact")
                                         (Plus (Var "x")
                                               (Const (-1))))))
          let f10 = App fact (Const 10)
          print $ runResult (eval f10) []
          -- Just (IntVal 3628800)
```

MonadReader

- MonadReader defines the standard Reader interface.

```haskell
newtype Result r a =  Result {runResult :: r -> Maybe a}

class Monad m => MonadReader r (m :: * -> *) | m -> r where
  ask :: m r           -- extract the current context

  -- replace the context with a new one and use it locally
  local :: (r -> r) -> m a -> m a

  -- run a function with the current context
  reader :: (r -> a) -> m a
  {-# MINIMAL (ask | reader), local #-}
```

- To make the Result type a proper Reader, we need to define its MonadReader instance.

MonadReader

- MonadReader defines the standard Reader interface.

```haskell
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}

newtype Result r a =  Result {runResult :: r -> Maybe a}

class Monad m => MonadReader r (m :: * -> *) | m -> r where
     ask :: m r
   local :: (r -> r) -> m a -> m a
  reader :: (r -> a) -> m a

instance MonadReader r (Result r) where
         ask = Result $ \ctx -> return ctx
  local f m = Result $ \ctx -> (runResult m) (f ctx)
   reader f = Result $ \ctx -> return (f ctx)

-- lift a Maybe value to a Result value.
lift' :: Maybe a -> Result r a
lift' mb = Result $ \_ -> mb
```

MonadReader

- Using standard Reader interface

```
1  class Monad m => MonadReader r (m :: * -> *) | m -> r where
2    ask :: m r
3    local :: (r -> r) -> m a -> m a
4
5  lift' :: Maybe a -> Result r a
6
7  -- use 'ask' to extract context and 'lift' to promote Maybe
8  eval (Var s) = ask >>=
9                          \ctx ->
10                                 lift' $ lookup s ctx
```

MonadReader

- Using standard Reader interface

```
1  class Monad m => MonadReader r (m :: * -> *) | m -> r where
2    ask :: m r
3    local :: (r -> r) -> m a -> m a
4
5  eval (App t1 t2) = do
6      fun@(FVal (f, x, t0) ctx0) <- eval t1
7
8      arg <- eval t2
9
10     let ctx = case f of Just name -> [(name, fun)]
11                         Nothing   -> []
12
13     -- use 'local' to make local changes to the context
14     local (\_ -> ctx ++ (x, arg) : ctx0)
15           (eval t0)
```

MonadReader

- Using standard Reader interface

```haskell
class Monad m => MonadReader r (m :: * -> *) | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a

-- use 'ask' to extract context
eval (Fn (x,t)) =
    ask >>=
            \ctx ->
                  return $ FVal (Nothing, x, t) ctx

eval (Fun (f,x,t)) =
    ask >>=
            \ctx ->
                  return $ FVal (Just f, x, t) ctx
```

Reader Transformer

- Monad can be layered using Monad transformers.

```haskell
-- T1, T2, T3 are transformers
-- M' is the new monad built from M
type M' a = T1 (T2 (T3 M)) a
```

- ReaderT transforms another monad into a Reader monad.

```haskell
class Monad m => MonadReader r (m :: * -> *) | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a

newtype ReaderT r (m :: * -> *) a =
                -- m is a monad (& a type operator)
                ReaderT {runReaderT :: r -> m a}
```

Reader Transformer

- (ReaderT r m) is a Monad
- (ReaderT r m) is a also MonadReader

```haskell
class Monad m => MonadReader r (m :: * -> *) | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a

newtype ReaderT r (m :: * -> *) a =
                ReaderT {runReaderT :: r -> m a}

-- (ReaderT r m) is an instance of Monad
instance Monad m => Monad (ReaderT r m)

-- (ReaderT r m) is an instance of MonadReader
instance Monad m => MonadReader r (ReaderT r m)
```

Reader Transformer

- (ReaderT r) is a Monad transformer.
- Any (m a) value can be lifted into a (ReaderT r m a) value

```
1  newtype ReaderT r (m :: * -> *) a =
2                  ReaderT {runReaderT :: r -> m a}
3
4  -- (ReaderT r) is an instance of MonadTrans
5  instance MonadTrans (ReaderT r)
6
7  -- lift any type (m a) into (t m a)
8  lift :: (MonadTrans t, Monad m) => m a -> t m a
9
10 lift $ Just (IntVal 1) :: ReaderT Context Maybe Val
```

Reader Transformer

- No need to 'hand-roll' a Reader.

```haskell
class Monad m => MonadReader r (m :: * -> *) | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a

newtype ReaderT r (m :: * -> *) a =
                ReaderT {runReaderT :: r -> m a}

-- (ReaderT Context) transforms Maybe
--    into a (Reader Context) + Maybe Monad
eval :: Term -> ReaderT Context Maybe Val

eval (Const a) = return $ IntVal a

-- ask and lift come for free
eval (Var s) = ask >>=
                        \ctx ->
                                lift $ lookup s ctx
```

Reader Transformer

- No need to 'hand-roll' a Reader.

```haskell
class Monad m => MonadReader r (m :: * -> *) | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a

newtype ReaderT r (m :: * -> *) a =
                ReaderT {runReaderT :: r -> m a}

-- (ReaderT Context) transforms Maybe
--    into a (Reader Context) + Maybe Monad
eval :: Term -> ReaderT Context Maybe Val

eval (App t1 t2) =
  do
    fun@(FVal (f, x, t0) ctx0) <- eval t1
    arg <- eval t2
    let ctx = case f of Just name -> [(name, fun)]
                        Nothing -> []

    -- local is free too
    local (\_ -> ctx ++ (x, arg) : ctx0)
          (eval t0)
```

Reader Transformer

- No need to 'hand-roll' a Reader.

```haskell
class Monad m => MonadReader r (m :: * -> *) | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a

newtype ReaderT r (m :: * -> *) a =
                ReaderT {runReaderT :: r -> m a}

-- (ReaderT Context) transforms Maybe
--    into a (Reader Context) + Maybe Monad
eval :: Term -> ReaderT Context Maybe Val

-- everything remains the same
eval (Fn (x, t)) =
       ask >>=
              \ctx ->
                     return $ FVal (Nothing, x, t) ctx
eval (Fun (f, x, t)) =
       ask >>=
              \ctx ->
                     return $ FVal (Just f, x, t) ctx
```

Exception handling with Either Monad

- Either represents a binary choice – but left is usually bad

```haskell
data Either a b = Left a | Right b

instance Functor (Either e) where
        fmap _ (Left a) = Left a
        fmap f (Right a) = Right (f a)

instance Monad (Either e) where
        return = Right
        Right m >>= k = k m
        Left e  >>= _ = Left e

instance Applicative (Either e) where
        pure = Right
        a <*> b = do x <- a; y <- b; return (x y)
```

Exception handling with Either Monad

- throwError and catchError using Either monad

```haskell
data Either a b = Left a | Right b

class Monad m => MonadError e (m :: * -> *) | m -> e where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a

instance MonadError e (Either e) where
    throwError              = Left

    Left  l `catchError` h = h l

    Right r `catchError` _ = Right r
```

Exception handling with Either Monad

- throwError and catchError using Either monad

```haskell
data Either a b = Left a | Right b

-- customized error type
data EvalError = VariableNotFound String
               | NotAnInt Val
               | NotABool Val
               | NotAFun Val deriving (Show)

-- (ReaderT Context) transforms (Either EvalError) monad
eval :: Term -> ReaderT Context (Either EvalError) Val
```

Exception handling with Either Monad

- throwError and catchError using Either monad

```haskell
data Either a b = Left a | Right b

data EvalError = VariableNotFound String
               | NotAnInt Val
               | NotABool Val
               | NotAFun Val deriving (Show)

eval :: Term -> ReaderT Context (Either EvalError) Val

-- throwError if variable 's' is not found
eval (Var s) = do
        ctx <- ask
        case lookup s ctx of
            Just a -> return a
            Nothing -> throwError $ VariableNotFound s
```

Exception handling with Either Monad

- throwError and catchError using Either monad

```haskell
data Either a b = Left a | Right b

data EvalError = VariableNotFound String
               | NotAnInt Val
               | NotABool Val
               | NotAFun Val deriving (Show)

eval :: Term -> ReaderT Context (Either EvalError) Val

-- throwError if 'fun' is not a function
eval (App t1 t2) =
  do
    fun <- eval t1
    case fun of
      (FVal (f, x, t0) ctx0) -> do
          arg <- eval t2
          let ctx = case f of Just name -> [(name, fun)]
                              Nothing   -> []
          local (\_ -> ctx ++ (x, arg) : ctx0) $ eval t0
      _ -> throwError $ NotAFun fun
```

Exception handling with Either Monad

- throwError and catchError using Either monad

```haskell
data Either a b = Left a | Right b

data EvalError = VariableNotFound String
               | NotAnInt Val
               | NotABool Val
               | NotAFun Val deriving (Show)

eval :: Term -> ReaderT Context (Either EvalError) Val

-- throwError if either operand of + is not an int
eval (Plus t1 t2) = do
   v1 <- eval t1
   v2 <- eval t2
   case (v1, v2) of
     (IntVal c1, IntVal c2) -> return $ IntVal (c1 + c2)
     (_, IntVal _) -> throwError $ NotAnInt v1
     (_, _) -> throwError $ NotAnInt v2
```

Exception handling with Either Monad

- throwError and catchError using Either monad

```haskell
data Either a b = Left a | Right b

data EvalError = VariableNotFound String
               | NotAnInt Val
               | NotABool Val
               | NotAFun Val deriving (Show)

eval :: Term -> ReaderT Context (Either EvalError) Val

-- throwError if condition of branch is not a bool
eval (IF t0 t1 t2) = do
   v <- eval t0
   case v of
        (BoolVal b) -> if b then eval t1 else eval t2
        _ -> throwError $ NotABool v
```

Exception handling with Either Monad

- run ReaderT

```
1  main :: IO ()
2  main = do
3            let x = Plus (Const 10) (Const 20)
4            let f = Fn ("x", Plus(Var "y") (Const 5))
5            let fact = Fun ("fact", "x",
6                            IF (LE (Var "x") (Const 1))
7                               (Const 1)
8                               (Times (Var "x") (App (Var "fact")
9                         )
10           let f10 = App fact (Const 10)
11           print $ runReaderT (eval f10) []
12 -- Right (IntVal 3628800)
13           let e0 = App f x
14           let e1 = App x (Const 10)
15           let e2 = Plus f (Const 10)
16           print $ runReaderT (eval e0) []
17 -- Left (VariableNotFound "y")
18           print $ runReaderT (eval e1) []
19 -- Left (NotAFun (IntVal 30))
20           print $ runReaderT (eval e2) []
21 -- Left (NotAnInt (FVal (Nothing,"x",Plus (Var "y") (Const 5))
22 --                           []))
```

Writer Monad

- A Writer Monad pairs a value ‘a’ with a monoid ‘w’ to record output during computation.

```haskell
class (Monoid w, Monad m) =>
      MonadWriter w (m :: * -> *) | m -> w where

  -- creates a writer action with value 'a' and output 'w'
  writer :: (a, w) -> m a

  -- creates a writer action with output 'w'
  tell :: w -> m ()

  -- executes the action and return '(a, w)'
  listen :: m a -> m (a, w)

  -- executes an action to return a value and a function &
  -- returns the value and applies the function to the output
  pass :: m (a, w -> w) -> m a
  {-# MINIMAL (writer | tell), listen, pass #-}
```

Writer Monad

- A Writer Monad pairs a value ‘a’ with a monoid ‘w’ to record output during computation.

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

-- output type 'w' has to be a monoid
instance (Monoid w) => Monad (Writer w) where

    -- returned value is associated with empty monoid
    return x = Writer (x, mempty)


    -- bind appends output v with v'
    (Writer (x,v)) >>= f = let
                               (Writer (y, v')) = f x
                           in
                               Writer (y, v `mappend` v')
```

Pretty print example

- Print a function in readable format using tabs.

```
data Term = Const Integer
          | Plus Term Term   | Times Term Term
          | LE Term Term     | IF Term Term Term
          | Var String
          | App Term Term
          | Fn (String, Term)
          | Fun (String, String, Term)

pp0 :: Term -> String -> String

pp0 (IF t0 t1 t2) space =
   let t = pp0 t1 (space ++ "\t")
       e = pp0 t2 (space ++ "\t")
   in space ++ "if " ++ show t0 ++ "\n" ++ space ++
               "then\n" ++ t ++ "\n" ++ space ++
               "else\n" ++ e

pp0 (Fun (f, x, t)) space =
   let body = pp0 t (space ++ "\t")
   in "fun " ++ f ++ " " ++ x ++ " =\n" ++ body

pp0 x space = space ++ show x
```

Pretty print example

- Print a function in readable format using tabs.

```
1  pp0 :: Term -> String -> 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
13           putStrLn $ pp0 fact ""
14
15 -- fun fact x =
16 --       if (x <= 1)
17 --       then
18 --                 1
19 --       else
20 --                 (x * (fact (x + -1)))
```

Pretty print example

- Use Writer Monad to collect output.

```
1  data Term = Const Integer
2            | Plus Term Term   | Times Term Term
3            | LE Term Term     | IF Term Term Term
4            | Var String
5            | App Term Term
6            | Fn (String, Term)
7            | Fun (String, String, Term)
8
9  pp1 :: Term -> String -> Writer String ()
10
11 pp1 (IF t0 t1 t2) space = do
12    tell $ space ++ "if " ++ show t0 ++ "\n"
13    tell $ "\n" ++ space ++ "then\n"
14    pp1 t1 $ space ++ "\t"
15    tell $ "\n" ++ space ++ "else\n"
16    pp1 t2 $ space ++ "\t"
17
18 pp1 (Fun (f, x, t)) space = do
19    tell $ "fun " ++ f ++ " " ++ x ++ " =\n"
20    pp1 t $ space ++ "\t"
21
22 pp1 x space = tell $ space ++ show x
```

Pretty print example

- Run Writer Monad to obtain output (the right of the tuple)

```
1  pp1 :: Term -> 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
13           putStrLn $ snd $ runWriter $ pp1 fact ""
14
15 -- fun fact x =
16 --       if (x <= 1)
17 --       then
18 --                 1
19 --       else
20 --                 (x * (fact (x + -1)))
```

Pretty print example

- Use Reader Monad to remember tabs

```
1  tab :: MonadReader String m => m a -> m a
2  tab = local (\s -> s ++ "\t")
3
4  pp :: Term -> ReaderT String (Writer String) ()
5
6  pp (IF t0 t1 t2) = do
7      space <- ask
8      tell $ space ++ "if " ++ show t0
9      tell $ "\n" ++ space ++ "then\n"
10     tab $ pp t1
11     tell $ "\n" ++ space ++ "else\n"
12     tab $ pp t2
13
14 pp (Fun (f, x, t)) = do
15     space <- ask
16     tell $ "fun " ++ f ++ " " ++ x ++ " =\n"
17     tab $ pp t
18
19 pp x = do
20     space <- ask
21     tell $ space ++ show x
```

Pretty print example

- Run ReaderT with initial string and then run Writer.

```
pp :: Term -> ReaderT String (Writer String) ()

main :: IO ()
main = do
          let fact = Fun ("fact", "x",
                          IF (LE (Var "x") (Const 1))
                             (Const 1)
                             (Times (Var "x")
                                    (App (Var "fact")
                                         (Plus (Var "x")
                                               (Const (-1))))))

          putStrLn $ snd $ runWriter $ runReaderT (pp fact) ""

-- fun fact x =
--       if (x <= 1)
--       then
--                1
--       else
--                (x * (fact (x + -1)))
```

Extend interpreter with Writer Monad

- Implement Show instance for Val type

```
instance Show Val where
  show (IntVal x) = show x
  show (BoolVal b) = show b
  show (FVal (Just f, x, t) _) =  f
  show (FVal (Nothing, x, t) _) =
       "(" ++ x ++ " => " ++ show t ++ ")"

-- Use WriterT transformer to wrap (Either EvalError) monad
eval :: Term -> ReaderT Context
                        (WriterT [String]
                                 (Either EvalError)) Val
```

- Writer's output is a list of strings.

Extend interpreter with Writer Monad

- Use Writer Monad to log the trace of function calls.

```haskell
-- Use WriterT transformer to wrap (Either EvalError) monad
eval :: Term -> ReaderT Context
                        (WriterT [String]
                                 (Either EvalError)) Val

eval (App t1 t2) =
  do
    fun <- eval t1
    case fun of
      (FVal (f, x, t0) ctx0) -> do
          arg <- eval t2

          let ctx = case f of Just name -> [(name, fun)]
                              Nothing   -> []

          -- record the function call in writer monad
          tell $ [show fun ++ " " ++ (show arg)]

          local (\_ -> ctx ++ (x, arg) : ctx0) $ eval t0

      _ -> throwError $ NotAFun fun
```

Extend interpreter with Writer Monad

- Run eval function with error handling and logging.

```haskell
runEval :: Term -> String

runEval t = y

  where -- unwrap the value
    WriterT (Right (y, _)) = runReaderT x []

    x =  -- extract value and writer output 'w'
      do (a, w) <- listen $ eval t

         -- make a string out of the log list
         let w' = mconcat $ map (\x -> "\t" ++ x ++ "\n") w

         -- return the answer + log as string value
         return $ "answers: " ++ show a ++
                  "\nwith call trace: \n" ++ w'

      -- catch any evaluation error and return it as string
      `catchError` (\e -> return $ show e ++ "\n")
```

Extend interpreter with Writer Monad

```
eval :: Term -> ReaderT Context
                        (WriterT [String]
                                 (Either EvalError)) Val
runEval :: Term -> String

main :: IO ()
main = do
          -- define 'fact'
          putStrLn $ runEval (App fact (Const 10))

answers: 3628800
with call trace:
         fact 10
         fact 9
         fact 8
         fact 7
         fact 6
         fact 5
         fact 4
         fact 3
         fact 2
         fact 1
```