# Monad

A design pattern for composable effectful computations

Extends values with structure to model effects.
$t$ becomes $t_{\text{effect}}$ when $t$ is in the "effect" monad.

Allows effects to be combined when working with values in the monad. Like function application + effect accumulation:
$t_{\text{effect}}$ -> ($t$ -> $t'_{\text{effect}}$) -> $t'_{\text{effect}}$

Abstraction layer for the effect machinery so that code is cleaner and effects can be controlled separately from computation.

# Monads in Haskell

```
class Monad m where
    -- Sequentially compose two actions, passing
    -- any value produced by the first as an
    -- argument to the second.
    (>>=) :: m a -> (a -> m b) -> m b
    -- Inject a value into the monadic type.
    return :: a -> m a
```

Laws:
left-identity: (return x) >>= f == f x
right-identity: mv >>= return == mv
associativity: (mv >>= f) >>= g == mv >>= (\x -> f x >> g)

# Effect: Partiality

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
    return x = Just x
    Nothing >>= _ = Nothing
    Just x  >>= f = f x
```

# Example: Partiality

```
divide n d | d != 0 = return (n/d)
           | d == 0 = Nothing

-- compute a / (b / c)
myfun a b c =
  (divide b c) >>= (\d -> divide a d)

-- sample execution
myfun 30 14 7 ==> Just 15
myfun 30  0 7 ==> Nothing
myfun 30 14 0 ==> Nothing
```

# Effect: Failure

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

instance Monad (Either e) where
    return x = Right x
    Left err >>= _ = Left err
    Right x  >>= f = f x

fail err = Left err
```

# Example: Failure

```
lookup_by_name n db =
  case (find n db) of
    (Just entry) -> return entry
    Nothing      -> fail "name not found"

get_email_addr entry =
  if (has_email entry)
  then return (email_addr entry)
  else fail "no email address"

name_to_email name db =
  (lookup_by_name name db) >>= get_email_addr
```

# Effect: State

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

instance Monad (State st) where
    return x = State $ \s -> (x,s)
    mv >>= f  = State $ \s ->
      let (x, s') = runState mv s
      in runState (f x) s'

get = State $ \s -> (s,s)
put s = State $ \_ -> ((),s)
```

# Example: State

```
-- build a histogram of arguments to a function
use_and_record fn x =
  get >>= \hist ->
  put (incr hist x) >>= \_ ->
  return (fn x)


-- with “do notation”
use_and_record fn x =
  do hist ← get
     let new_hist = incr hist x
     put new_hist
     return (fn x)
```

# The Monad Design Pattern

```
-- define data or newtype for modeling the effect
???


instance Monad ??? where
    -- add effect structure to pure value
    return x = ???
    -- apply f to value from mv and
    -- combine effects of mv and f's result
    mv >>= f  = ???


-- monad-specific functions for injecting and
-- working with effects
???
```

# Monad Functions

```
-- sequentially compose effects, discarding the
-- value produced by the first argument
>> :: (Monad m) => m a -> m b -> mb
mv1 >> mv2 = mv1 >>= (\_ -> mv2)


-- lift a function on values into a monad
liftM :: (Monad m) -> (a -> b) -> (m a -> m b)
liftM f mv = do { x1 ← mv; return (f x1) }


-- evaluate each action in sequence from left
-- to right, collecting the results
sequence :: (Monad m) => [m a] -> m [a]
sequence ms = foldr k (return []) ms
  where k mv mv' =
    do { x <- mv; xs <- mv'; return (x:xs) }
```

many more: mapM, filterM, foldM, replicateM, liftM2, liftM3, ...

# Monad Transformers

Often you want to use multiple effects in combination, e.g. state and errors.

A monad transformer adds an effect to a base monad.

Multiple monad transformers can be used to layer on multiple effects. The combination of transformers and the base monad is called a "transformer stack".

# StateT

```
newtype StateT st m a =
    StateT { runStateT :: st -> m (a,st) }

instance (Monad m) => Monad (StateT st m) where
    return x = StateT $ \s -> return (x, s)
    mv >>= f  = StateT $ \s -> do
        (x, s') <- runStateT mv s
        runStateT (f x) s'

getT = StateT $ \s -> return (s,s)
putT s = StateT $ \_ -> return ((),s)
```

# ErrorT

```
newtype ErrorT e m a =
   ErrorT { runErrorT :: m (Either e a) }

instance (Monad m) => Monad (ErrorT e m) where
    return x = ErrorT $ return (Right x)
    mv >>= f  = ErrorT $ do
        a <- runErrorT mv
        case a of
            Left  err -> return (Left err)
            Right x   -> runErrorT (f x)

failT err = ErrorT $ return (Left err)
```

# Working with a Transformer Stack

Order matters!

Contrast `ErrorT e (State st)` vs. `StateT st (Error e)`.

# Working with a Transformer Stack

## Order matters!

Contrast `ErrorT e (State st)` vs. `StateT st (Error e)`.

`ErrorT e (State st)` **==>** `st -> (Either e a, st)`

Failure does not produce a value, but still gives a valid state!

# Working with a Transformer Stack

## Order matters!

Contrast `ErrorT e (State st)` vs. `StateT st (Error e)`.

`ErrorT e (State st)` ==> `st -> (Either e a, st)`

Failure does not produce a value, but still gives a valid state!

`StateT st (Error e)` ==> `st -> Either e (a, st)`

Failure does not produce a value or a valid state!

# Lifting Operations

When working with transformer stacks it is often required to lift values or operations in a lower monad into the combined monad, or to recover a value in a lower monad from a value in the combined monad.

In Haskell, there is a type class operation for monad transformers to lift a value from the lower monad:

```
lift :: (Monad m) => m a -> t m a
```

To lift through multiple levels, the operation can be composed multiple times (e.g., `lift . lift . lift`).

To move down the stack the various run functions are used to “peel the onion” back to the desired monad level.