

# Functional Programming

---

WS 2019/20

Torsten Grust  
University of Tübingen

## Monads (“*The Programmable Semicolon*”)

---

Function composition via `(.)` is *the* FP idiom that constructs larger programs. In a sense, `(.)` serves the role of `;` (statement sequencing) in imperative PLs.

- **Sequencing functions:**

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

```
(>=>) :: (a -> b) -> (b -> c) -> (a -> c)
f >=> g = g . f
```

- `f1>=> f2 >=> ... >=> fn` composes the `fi` in left-to-right order (think UNIX pipes).
- `(>=>, id)` forms a monoid: `>=>` is associative with identity `id`.

## Sequencing Partial Functions ( $a \rightarrow \text{Maybe } b$ )

---

- If the  $f_i$  in  $f_1 \Rightarrow f_2 \Rightarrow \dots \Rightarrow f_n$  are potentially failing (“partial”) functions, the composition should fail once the first function fails.
  - Return `Nothing` as soon as first function in sequence returns `Nothing`:

```
(=>) :: (a -> Maybe b) -> (b -> Maybe c) -> (a -> Maybe c)
f => g = \x -> case f x of
             Nothing -> Nothing
             Just y   -> g y
```

## Sequencing Exception-Generating Functions ( $a \rightarrow \text{Exc } b$ )

---

- Result of computations that may either throw an exception or return a regular value  $b$  (assumes suitable representation type for errors/exceptions, *e.g.*, `type Error = String`):

```
type Exc b = Either Error b
```

- In a composition, exceptions are propagated once occurred:

```
(>=>) :: (a -> Exc b) -> (b -> Exc c) -> (a -> Exc c)  
f >=> g = \x ->
```

## Sequencing Non-Deterministic Functions (**a -> NonDet b**)

---

- A non-deterministic function may respond with *any answer* in a list of possible of answers:

```
type NonDet b = [b]
```

- In a composition of non-deterministic functions, treat all possible outcomes alike:

```
(>=>) :: (a -> NonDet b) -> (b -> NonDet c) -> (a -> NonDet c)  
f >=> g = \x -> concat $ map g $ f x
```

or

```
f >=> g = \x -> concat [ g y | y <- f x ]
```

## Sequencing Stateful Functions ( $a \rightarrow \text{State} \rightarrow (\text{State}, b)$ )

---

- Perform regular computation (to yield a value of type  $b$ ) and transform the current state of the system/program.
  - **State transformers** (assumes state representation  $\text{State}$ ):

```
type ST b = State -> (State, b)
```

- Stateful functions thus have type  $a \rightarrow \text{ST } b$ .
- Composition (with state transformation  $s_0 \mapsto s_1 \mapsto s_2$ ):

```
(>=>) :: (a -> ST b) -> (b -> ST c) -> (a -> ST c)
f >=> g = \x s0 -> let (s1, y) = f x s0 in
                  let (s2, z) = g y s1 in      } simplifies to
                  (s2, z)                       } g y s1
```

## Sequencing Side-Effecting Functions ( $a \rightarrow \text{World} \rightarrow (\text{World}, b)$ )

---

- Modeling **side effects**: functions consume current “world” state, return new world state along with result:

```
type World = "\_(ツ)_/" -- abstract (defined by GHC runtime)
type IO b = World -> (World, b)
```

- Values of type  $\text{IO } b$  are **actions** that
  1. *when performed* have a side-effect on the world, and then
  2. return a value of type  $b$ .
- Haskell built-ins with side-effects:
  - `print :: Show a => a -> IO ()`, `putStrLn :: String -> IO ()`
  - `getLine :: IO String`
  - `readFile :: FilePath -> IO String`
  - `(>=>) :: (a -> IO b) -> (b -> IO c) -> (a -> IO c)`

## Type Class Monad

---

- These sequencing functions follow an obvious pattern:

```
(>=>) :: (a -> b) -> (b -> c) -> (a -> c)
(>=>) :: (a -> Maybe b) -> (b -> Maybe c) -> (a -> Maybe c)
(>=>) :: (a -> Exc b) -> (b -> Exc c) -> (a -> Exc c)
(>=>) :: (a -> NonDet b) -> (b -> NonDet c) -> (a -> NonDet c)
(>=>) :: (a -> ST b) -> (b -> ST c) -> (a -> ST c)
(>=>) :: (a -> IO b) -> (b -> IO c) -> (a -> IO c)

(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
```

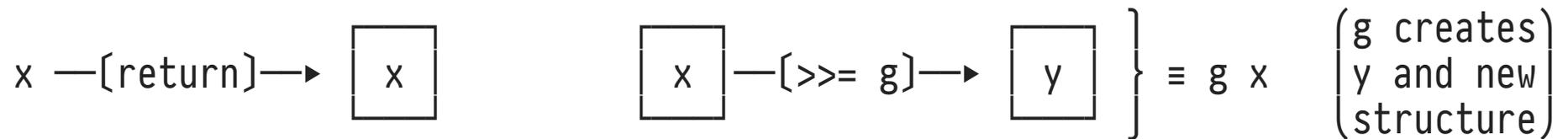
- To instantiate `(>=>)` for different type constructors `m`, build a type class whose instances are **type constructors** (i.e. `m` is of kind `* -> *`).

## Type Class `Monad` (Or: Why Haskell's Logo is $\gg=$ )

---

- Type class `Monad` ( $m$  has kind  $* \rightarrow *$ ):

```
class Applicative m => Monad m where  
  return :: a -> m a           -- lifting  
  (>>=)  :: m a -> (a -> m b) -> m b  -- sequencing ("bind")
```



- NB:** Once a value is inside the monadic  $m$ -structure, class `Monad` does not let it “escape” again.

## (>>=) vs. (>=>)

---

- **Bind** (>>=) is simpler/does less than (>=>):
  - `f >=> g` involves *two* applications (of `f` and `g`). Instead, `s >>= g` receives structure `s`, extracts its value `x`, and routes `x` to `g` (*one* application).
  - Nevertheless, both are closely related.  
(>=>) may be expressed in terms of (>>=):

```
(=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
f => g = \x -> f x >>= g
                └──
                :: m b
```

- (>=>) is also known as **Kleisli composition**.

## Lifting via `return`

---

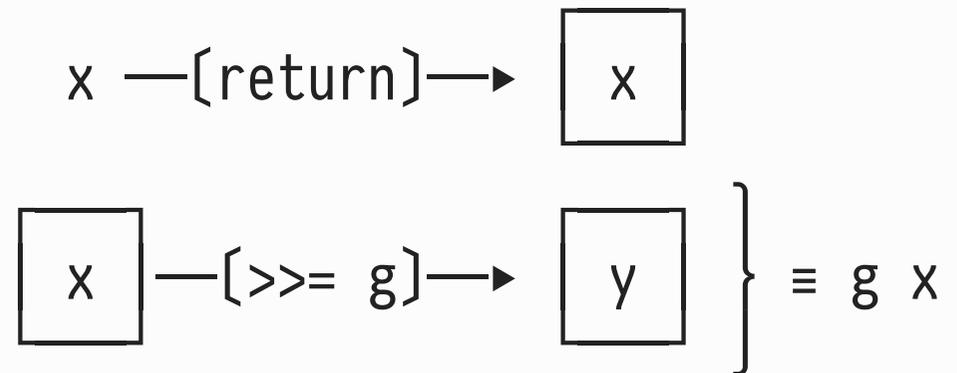
- **Lifting** creates a trivial context/structure (recall `pure`):

Monad	lifting ( <code>return x</code> )
Maybe	Just x
Exc	Right x
NonDet	[x]
ST	\s -> (s,x)

- **Example:** Make `Maybe` an instance of `Monad`:

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

```
Nothing >>= g = Nothing
Just x   >>= g = g x
```



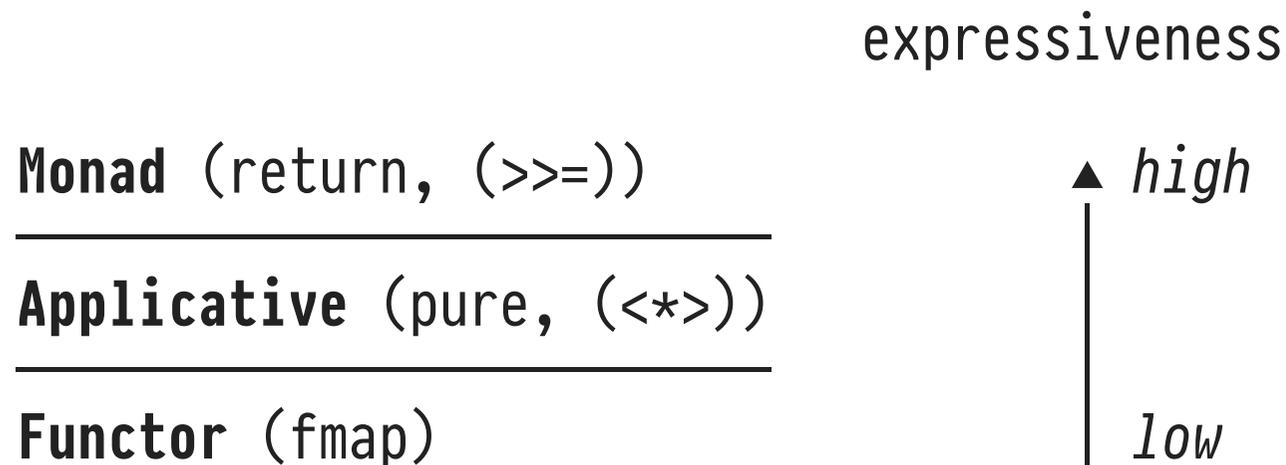
## The **Functor**/**Applicative**/**Monad** Tower

---

- The type classes **Functor**, **Applicative**, and **Monad** are obviously closely related:

```
class Functor m      => Applicative m where ...  
class Applicative m => Monad m      where ...
```

- **Monad**, **Applicative**, **Functor** form a “tower of abstractions”:



## A Unique Ability of Monad

---

- **Monad** is truly more expressive than the abstractions we have studied so far.

Monadic bind `>>=` can create entirely **new contexts/structure**:

These functions emit values (of type `b`) to be embedded in existing/combined structures

<code>(<math>\\$</math>)</code>	<code>::</code>	$(a \rightarrow b) \rightarrow a$	<code>-&gt;</code>	<code>a</code>	<code>-&gt;</code>	<code>b</code>	
<code>(&lt;<math>\\$</math>&gt;)</code>	<code>::</code>	<b>Functor</b> <code>m =&gt;</code>	$(a \rightarrow b) \rightarrow m a$	<code>-&gt;</code>	<code>m a</code>	<code>-&gt;</code>	<code>m b</code>
<code>(&lt;*&gt;)</code>	<code>::</code>	<b>Applicative</b> <code>m =&gt; m</code>	$(a \rightarrow b) \rightarrow m a$	<code>-&gt;</code>	<code>m a</code>	<code>-&gt;</code>	<code>m b</code>

<code>(&gt;&gt;=)</code>	<code>::</code>	<b>Monad</b> <code>m =&gt;</code>	<code>m a</code>	<code>-&gt;</code>	$(a \rightarrow m b)$	<code>-&gt;</code>	<code>m b</code>
--------------------------	-----------------	-----------------------------------	------------------	--------------------	-----------------------	--------------------	------------------

This function can generate a new context `m`!

## Expressiveness: Monad > Applicative > Functor

---

- A *Monad*  $m$  can do whatever an *Applicative* can.

Proof: Express *Applicative*'s operations in terms of *Monad*'s:

```
pure x     $\stackrel{\text{def}}{=}$     return x

                :: (a -> b)                :: a                :: b
                |                          |                          |
u <*> v     $\stackrel{\text{def}}{=}$     u >>= \f -> v >>= \x -> return (f x)
                |                          |                          |
                :: m (a -> b)                :: m a                :: m b
```

- Given this and  $\text{fmap } f \ v = \text{pure } f \ <*> \ v$ ,  $m$  also is a *Functor*:

```
fmap f v     $\stackrel{\text{def}}{=}$     return f >>= \f -> v >>= \x -> return (f x)
```

## A Dose of Syntactic Sugar: **do**-Notation

---

- Long chains of  $e_1 \gg= \lambda x \rightarrow e_2$  are typical for monadic code, yet hard to read.
- *Imperative* programming languages have introduced established notation for programs that
  1. perform computation  $e_1$ , name the result  $x$ , and then
  2. perform computation  $e_2$  (which may refer to variable  $x$ ):

```
x := e1;  
e2;          /* e2 may use x */
```

- Haskell offers **syntactic sugar** that can mimic such sequential programs (but indeed is much more general): **do-notation**.

## do-Notation

---

- Haskell's `do`-notation ( $e$ : expression of type  $m\ a$ ,  $es$ : sequence of `;`-separated expressions  $e_1; e_2; \dots; e_n$ ):

```
do { e }           ≡ e
do { x <- e; es }  ≡ e >>= \x -> do { es }
do { e; es }       ≡ e >>= \_ -> do { es }
do { let v = x; es } ≡ let v = x in do { es }  -- a pure let
```

- May use 2D layout instead of `{ ... }` and `;`.
-  Syntactically mimics imperative statements, but still is an expression that computes a monadic value: `do { ... } :: m b`.

## Example ①: Interactively Copy a File (do-Notation)

---

- Using `do`-notation leads to almost Python-like  syntax:

```
copyFile :: FilePath -> FilePath -> IO Int
```

```
copyFile from to = do  
  content <- readFile from  
  writeFile to content  
  return (length content)
```

```
main :: IO ()
```

```
main = do  
  putStrLn "Which file do you want to copy?"  
  from <- getLine  
  putStrLn "Where do you want to copy it to?"  
  to <- getLine  
  n <- copyFile from to  
  putStrLn ("Copied " ++ show n ++ " bytes.")
```

## Example ②: Interactively Copy a File (>>= Explicit)

---

```
copyFile :: FilePath -> FilePath -> IO Int
```

```
copyFile from to =
```

```
  readFile from          >>= \content ->
```

```
  writeFile to content >>= \_          ->      -- ← may use (>>)
```

```
  return (length content)
```

```
main :: IO ()
```

```
main =
```

```
  putStrLn "Which file do you want to copy?" >>= \_ ->
```

```
  getLine >>= \from ->
```

```
  putStrLn "Where do you want to copy it to?" >>= \_ ->
```

```
  getLine >>= \to ->
```

```
  copyFile from to >>= \n ->
```

```
  putStrLn ("Copied " ++ show n ++ " bytes.")
```

## ! Do Not Let **do** Fool You

---

- This is **not** imperative programming in Haskell. Beware of actually thinking in , , **JS**, ... terms.
- **Example:** Is this correct Haskell? What is the output?

```
f :: IO Integer
f = do
  putStrLn "Running f ..."
  return 1893
  return 1904

main :: IO ()
main = do
  x <- f
  putStrLn (show x)
```

## Input/Output

---

- But how can I/O possibly be pure? Reading a string with `getLine` may well yield different results on each execution.

### 💡 Key Idea:

Decouple the **pure construction** of I/O actions from their **effectful execution**.

1. Construct (complex) I/O action of type `IO a` using `getLine`, `putStrLn`, `>>=`, ... **No I/O or side effects happened yet.**
2. Perform constructed `IO a` action: **side effects will occur** and value of type `a` is returned. *Only* the `IO a` action returned by function `main` is performed by Haskell's runtime system.

## Roll Your Own I/O DSL

---

- GHC's internal representation of IO actions is opaque. Likewise, the internals of the Haskell runtime—the interpreter for these IO actions—are hidden.
- To better see what is going on, build our own versions of both in terms of a simple **IO action DSL**:
  1. The *constructors* build IO actions (e.g., printing/reading strings). Supply a `Monad` instance for the constructors to use `do`-notation to conveniently sequence complex IO actions.
  2. The *observer* `runIO` will interpret the resulting IO actions and perform the actual input/output side effects.

## Monad Laws

---

- Haskell presently cannot enforce (or even check) this, but `return` and `>=>` (or `>=>`) are expected to behave “sanely”:

<code>return &gt;=&gt; g</code>	<code>≡</code>	<code>g</code>	(left identity)
<code>f &gt;=&gt; return</code>	<code>≡</code>	<code>f</code>	(right identity)
<code>(f &gt;=&gt; g) &gt;=&gt; h</code>	<code>≡</code>	<code>f &gt;=&gt; (g &gt;=&gt; h)</code>	(associativity)

- $\Rightarrow$  (`>=>`, `return`) forms a monoid over the carrier of functions `a -> m b`.

*“A monad is just a monoid in the category of endofunctors.  
What's the problem?”* — James Iry

- Category is *Hask* (objects: Haskell types, arrows: functions)
- `m` is the endofunctor: `m b` is a Haskell type again

## Merging Monads: Monad Transformers

---

The monads we have seen so far are specialists in representing computations with a *single* type of side effect:

Monad	represents computation that...
Maybe a	may yield an <code>a</code> or return no result at all
Either e a	throws an exception <code>e</code> or succeeds with <code>a</code>
NonDet a ([a])	yields one of many possible <code>a</code>
ST a	reads/modifies current state, then returns <code>a</code>
IO a	performs input/output, then returns <code>a</code>
Reader e a	reads environment <code>e</code> , then returns <code>a</code>

- Can we build a monad to represent computations that perform I/O **and** may potentially throw an exception?
  - Do we need to start from scratch?
  - Or can we **combine the Either and IO monads** systematically?

## Either + IO: ① Verifying E-Mail Addresses

---

- Start with a computation that “verifies” a given e-mail address (does it contain exactly one '@')?

```
data LoginError = InvalidEmail
  deriving Show
```

```
checkEmail :: String -> Either LoginError String
checkEmail email = case splitOn "@" email of
    [_name, _domain] -> Right email
    _                  -> Left InvalidEmail
```

- Case `Left e` represents an exception that subsequent computations need to handle ( [monad-trans-1.hs](#)).

## Either + IO: ② Adding I/O

---

- Go interactive: ask for e-mail address, then verify it:

```
askForEmail :: IO (Either LoginError String)
askForEmail = do
  putStrLn "Enter e-mail address:"
  email <- getLine
  return (checkEmail email)
```

- As long as exception-generating computation and I/O do not interact, we are doing fine.
  - Here: perform all I/O, *then* verify e-mail (and we're done).
- If, instead, exception-generating computation and I/O mix, the code quickly gets nested and messier ( [monad-trans-2.hs](#)).

## **Either + IO:** ③ Build a Custom Combined Monad

---

We are operating in the **combined** (or: **stacked**) monad  
`IO (Either LoginError String)`. Generalized:

```
-- A computation that performs I/O and then either  
-- ① fails with an exception (of type e) or  
-- ② succeeds and returns with a result (of type a):
```

```
newtype EitherIO e a = EitherIO { runEitherIO :: IO (Either e a) }
```

- Recall that this `newtype` declaration defines two type conversions (at no runtime cost):

```
EitherIO      :: IO (Either e a) -> EitherIO e a  
runEitherIO   :: EitherIO e a    -> IO (Either e a)
```

## Either + IO: ③ Build a Custom Combined Monad

---

- The essence (behavior) of the combined monad:

**instance Monad (EitherIO e) where**

trivial (= no) I/O, computation succeeds

**return x = EitherIO \$**  **(return<sub>io</sub> . Right) x**

---

**x >>= f = EitherIO \$**  **runEitherIO x >>=<sub>io</sub> either (return<sub>io</sub> . Left)**  
 **(runEitherIO . f)**  
perform I/O,  
then yield **Either e a**  **2** computation succeeded,  
continue with f

**1** computation failed,  
trivial I/O

## ④ Lifting Values Into the Combined `EitherIO` Monad

---

-  To run existing code in the new `EitherIO` monad, we need to convert/lift `IO a` (and `Either e a`) values into `EitherIO e a` values. Otherwise, type errors ( $\Leftarrow$ ) occur:

```
askForEmail :: EitherIO LoginError String  
askForEmail = do  
  putStrLn "Enter e-mail address:"  $\Leftarrow^1$  :: IO ()  
  email <- getLine  $\Leftarrow^1$  :: IO String  
  return (checkEmail email)  $\Leftarrow^2$  :: IO (Either Lo...r String)
```

- $\Leftarrow^1$ : Has type `IO a` but we need `EitherIO e a`.
- $\Leftarrow^2$ : Has type `IO (Either e a)` but we need `EitherIO e a`.

## ④ Lifting Values Into the Combined `EitherIO` Monad

---

- Both **liftings** (`Either e a → EitherIO e a ← IO a`) are readily defined and help to write `EitherIO` computations concisely:

```
liftEither :: Either e a -> EitherIO e a
liftEither x = EitherIO (return x)
                        |
                        :: IO (Either e a)

liftIO :: IO a -> EitherIO e a
liftIO x = EitherIO (fmap Right x)
                |
                :: IO (Either e a) [recall: IO is a functor]
```

- That is all we need to rewrite our login/password application code ( `monad-trans-3.hs`).

## 5 Throwing and Catching Exceptions

---

- In `userLogin`, `liftEither (Left exc)` is used to raise (or “throw”) exception `exc`. Capturing that idiom:

```
throw :: e -> EitherIO e a
throw exc = liftEither (Left exc)
```

- As a counterpart to `throw`, “catch” exceptions thrown by a computation and invoke a **handler** to cope with the exception:

```
computation that may
  throw an exception
  |
catch :: EitherIO e a -> (e -> EitherIO e a) -> EitherIO e a
  |
  handler (receives
  exception as input)
```

## ⑤ Throwing and Catching Exceptions

---

- Exception handlers allow to model complex application behavior.

### Example:

① Attempt user login.

② Success?  $\left\{ \begin{array}{l} \text{Yes, logged in.} \\ \text{First failure?} \end{array} \right. \left\{ \begin{array}{l} \text{Yes, offer 2nd attempt. Goto ①.} \\ \text{No, abort. Print error message.} \end{array} \right.$

- Implementation ( [monad-trans-4.hs](#)):

```
loginDialogue :: EitherIO LoginError String
```

```
loginDialogue = do
```

```
  email <- userLogin `catch` wrongPassword `catch` abortLogin
```

```
  liftIO $ putStrLn ("Logged in as " ++ email)
```

```
  return email
```

## ⑥ Generalizing From `IO` to Any Monad `m`

---

- Abstract from `IO` and combine `Either` with any monad `m`. We arrive at the monad transformer `EitherT e m a`:

```
-- Run a computation in monad m and then either  
-- ① fail with an exception (of type e) or  
-- ② succeed and return with a result (of type a):
```

```
newtype EitherT e m a = EitherT { runEitherT :: m (Either e a) }
```

- See the Haskell `transformers` library for a collection of such monad transformers.
  - Example: An implementation of `EitherT` is found in `Control.Monad.Trans.ExceptT`.

## ⑥ Generalizing From IO to Any Monad $m$

**instance Monad  $m \Rightarrow$  Monad (EitherT  $e$   $m$ ) where**

trivial  $m$ -computation, computation succeeds

$\text{return } x = \mathbf{EitherT} \ \$ \ (\text{return}_m \ . \ \mathbf{Right}) \ x$

$x \gg= f = \mathbf{EitherT} \ \$ \ \underbrace{\text{runEitherT } x \gg=_m}_{\text{perform } m\text{-computation } x, \text{ then yield } \mathbf{Either} \ e \ a} \ \text{either} \ \underbrace{\begin{array}{l} \mathbf{1} \text{ computation failed,} \\ \text{trivial } m\text{-computation} \\ \text{(return}_m \ . \ \mathbf{Left}) \\ \text{(runEitherT } \cdot \ f) \end{array}}_{\mathbf{2} \text{ computation succeeded,}} \ \text{continue with } f$

- See  [monad-trans-5.hs](#).