Enter the Monad

Monoids

- the essense of a monoid is to take things of the same type (category) and return another thing of the same type (category) via a 'binary operation' called the combination function.
- it is a set of things, and rules for combining these things. we add complexity through composition.
  - f: A => A => A (combination function)
  - x: A
  - y: A
  - z: A
  - the combination function must be associative and have a nuetral / id value

$$f(x, f(y, z)) = f(f(x, y), z)$$

$$f(id, x) = x$$

Monoid Clock example

```
        _____
     _.'_____`._
   .'•-'  12 `-.`.
  /,' 11      1 `.\
 // 10      /   2 \\
;;         /       ::
|| 9  ---0      3 ||
::                 ;;
 \\ 8           4 //
  \`. 7       5 ,'/
   '.`-.__6__•-'.'
    ((-._____•-))
    _))       ((_
   '--'       '--'
```

```
Set S = {1, ... , 12}
function to add Time: (x + y) % 12
10 hours from 6 ⇒ (6 + 10) % 12 = 4
represent this combination function as (+)
this makes (+) an endomorphism as it maps elements
from our set S to elements in our set S (closure)
```

- here (+) is a closed binary monoidal function with respect to our Monoid structure

$$associativity$$
$$(x(+)y)(+)z = x(+)(y(+)z)$$
$$identity$$
$$x(+)12 = x$$
$$12(+)x = x$$

Functors

- functors can be thought of as describing the structure of mappable things, they bridge categories while maintaining their structure
- a functor from a category to itself is an **endofunctor**
- going from a basic map to a functor

```scala
def map[A, B](fa: List[A], f: A ⇒ B): List[B]  // Locked into the container of 'List'

trait Functor[F[_]] {
    def map[A, B](fa: F[A])(f: A ⇒ B): F[B]    // generalized over container F
}
```

- these must obey identity and composition
  - `fa.map(x ⇒ x) = fa` , mapping the identity function leaves the container (`fa`) unchanged
  - mapping function f and g sequentially should be equivalent to mapping the composition of f and g
  - $fa.map(f).map(g) = fa.map(f \cdot g)$

Monads

- monads are monoids in the category of endofunctors
  - category of endofunctors
    - all the functors of a category that map back to that category
  - we can think of it as a monoid with a bind, and identity function.
    - the identity is sometimes called pure, return, id
    - the **bind** acts as the **binary combination operation** of the monoid
    - the **id function** acts as the **identity element** of the monoid
  - the most important idea behind monads is composition. viewing them a monoid of functors
    - monoid set: the set of the monoid is represetned by all functors from some category back to that category
    - monoid properties: id element of the monoid is id function, compose function is bind

Monad Type Signature

- monoidal functions
  - `x : a`, `f: a → M a`, `g: a → M a`, `h: a → M a`
  - `M a` is a *type constructor* with arbitrary side effects
  - functions from `a → M a` live in a monad
  - the "data" `M a` lives in a monad
- the identity can be represented as
  - `id : a → M a`
- monad bind, represented by the infix (`>>=`)
  - the purpose of bind is to ensure functions are composable
    - `>>= : M a → (a → M b) → M b`
  - expression: `(f a) >>= λ a → (g a)`
  - type signature: M a · a -> Ma
  - confusing as it lacks symmetry

Symmetry and generalizing the 'data'

- we can rewrite the bind operation to show it is just composition
  - `λ a → (f a) >>= λ a → (g a)`
  - a -> M a · a -> M a
- generalizing the 'data' wrapped in the monad
  - `g: a → M b`
  - `f: b → M c`
  - `λ a → (g a) >>= λ b → (f b)`
    - a -> **M b -> (b -> M c) -> M c**
    - which can be thought of as **a -> M c**
    - this shows the bind is just about composition we take `a` through a series of transformations and end up with `M c`
      - `a → Operations and side effects → M c`

Ocaml example of Monad

```ocaml
(*
 * the `Option` in ocaml has a Monadic structure but doesn't have the needed
 * operations (return or id & bind) defined. here we define them.
 *)
module Maybe : Monad = struct
  type 'a t = 'a option

  let return x = Some x

  let (>>=) m f =   (* f: a -> Option b *)
    match m with    (* 'unwrap' the monad *)
    | None -> None
    | Some x -> f x (* apply f to the unwrapped value *)
end
```

Monad Laws

- Monads follow 3 laws, some monads used in programming can break the laws but are 'monads' in spirit
  - an example would be a monad that produces random values

1. left identity
2. right identity
3. associativity

Left Identity

```
-- Left Identity
instance : Monad List where
  pure := List.pure -- singleton
  bind := List.bind -- flatmap

def a := ["apple", "orange"]

#eval a >>= pure         -- ["apples", "orange"]
#eval a >>= pure = a     -- true
```

Right Identity

```lean
-- Right Identity
instance : Monad Option where
  pure := Option.some
  bind := Option.bind

def h (x : Nat) : Option Nat := some (x + 1)
def z := 5

#eval pure z >>= h          -- 6
#eval pure z >>= h = h z    -- true
```

Associativity

```
-- Associativity
-- (x : m a)
-- (f : a → m B)
-- (g : B → m y)
x >>= f >>= g = x >>= (λ x ⇒ f x >>= g)
```