Algebraic Thinking

Christopher R. Genovese

Department of Statistics & Data Science

Tue 18 Nov 2025
Session #22

Plan

Revisiting Functors and Friends

Plan

Revisiting Functors and Friends

Detailed Examples

Plan

Revisiting Functors and Friends

Detailed Examples

Case Study: Parsing

Announcements

- Documents repo: Lecture notes, fpc update
- **Reading**:
  - ATT
  - Fun with FP
    - cube composer
- **Homework**:
  - **sym-spell** assignment due now.
  - **ATTN** assignment due now.
  - addition activity

Plan

Revisiting Functors and Friends

Detailed Examples

Case Study: Parsing

Recap: Functors

```
trait Functor (f : Type -> Type) where
    map : (a -> b) -> f a -> f b
```

A **functor** is a type constructor with an associated function that ***lifts*** a transformation of values to a transformation of "containers." (We could call it `lift`!)

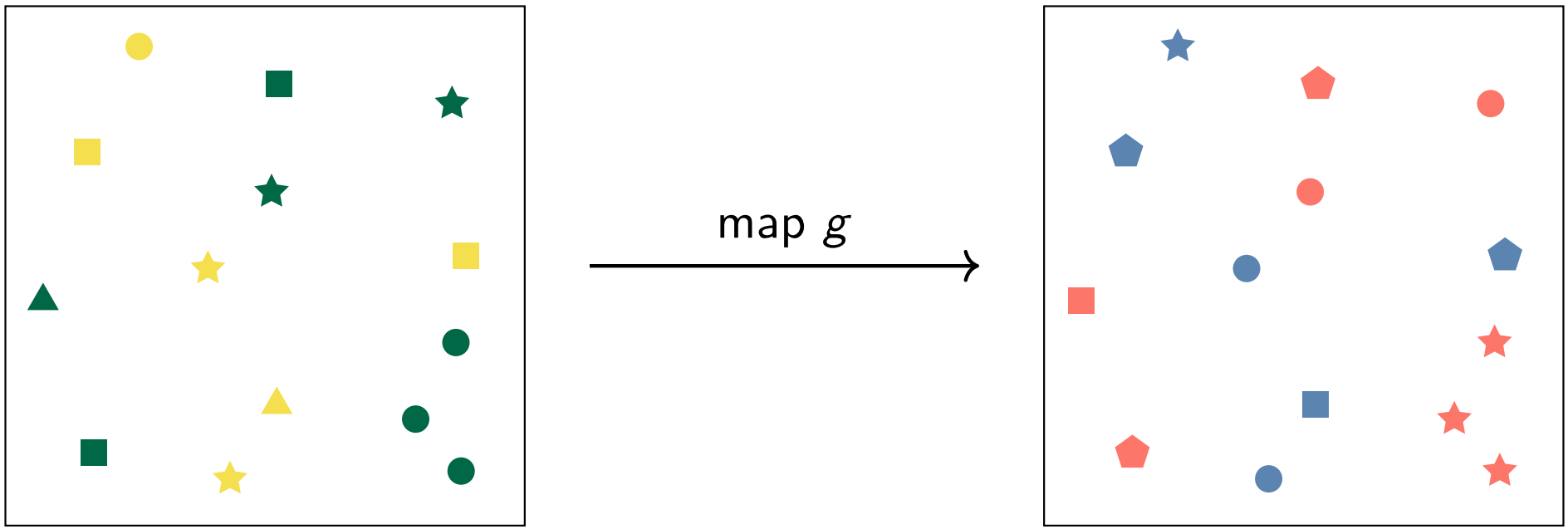
Functors satisfy two important **laws**:

1. `map id == id`
2. `(map g) . (map f) == map (g . f)`

The first law says that lifting the identity gives you the identity. The second law says that composing the lift of g and the lift of f is the same as lifting $g \circ f$. This is equivalent to `compose (map f) (map g) == map (compose f g)`.

Recap: Functors as a Computational Context

We can view functors as a ***computational context*** where we can transform the "results" inside it while preserving the context's "shape."




```
trait Functor (f : Type -> Type) where
    map : (a -> b) -> f a -> f b

laws Functor where
    map id == id
    map g . map h == map (g . h)
```

Recap Functors: Basic Examples

```
trait Functor (f : Type -> Type) where
    map : (a -> b) -> f a -> f b
```

Some fundamental examples:

Recap Functors: Basic Examples

```
trait Functor (f : Type -> Type) where
    map : (a -> b) -> f a -> f b
```

Some fundamental examples:

```
data Maybe : Type -> Type where
  None : Maybe a
  Some : a -> Maybe a

implements Functor Maybe where
  map : (a -> b) -> Maybe a -> Maybe b
  map f None = None
  map f (Some x) = Some (f x)
```

Recap Functors: Basic Examples

```
trait Functor (f : Type -> Type) where
    map : (a -> b) -> f a -> f b
```

Some fundamental examples:

```
data List (a : Type) where
  [] : List a
  (::) : a -> List a -> List a

implements Functor List where
  map : (a -> b) -> List a -> List b
  map f [] = []
  map f (x :: rest) = (f x) :: (map f rest)
```

Recap Functors: Basic Examples

```
trait Functor (f : Type -> Type) where
    map : (a -> b) -> f a -> f b
```

Some fundamental examples:

```
implements Functor ((->) r) where
  map : (a -> b) -> (r -> a) -> (r -> b)

  map f g = f . g          -- (.) is composition

  -- or equivalently
  map f g x = f (g x)
```

Recap Functors: Basic Examples

```
trait Functor (f : Type -> Type) where
    map : (a -> b) -> f a -> f b
```

Some fundamental examples:

```
data BinaryTree a = Node (BinaryTree a) a (BinaryTree a)

implements Functor BinaryTree where
  map : (a -> b) -> BinaryTree a -> BinaryTree b

  map f (Node left x right) = Node (map f left) (f x) (map f right)
```

Recap Functors: Basic Examples

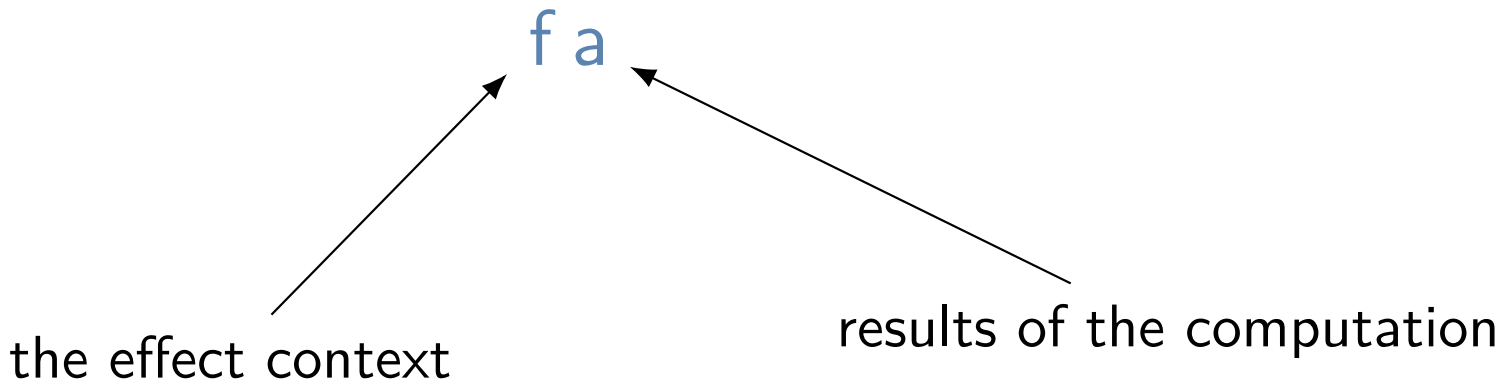
```
trait Functor (f : Type -> Type) where
    map : (a -> b) -> f a -> f b
```

Some fundamental examples:

```
data Tree a = record Node { value : a
                          , children : List (Tree a)
                          }


implements Functor Tree where
  map : (a -> b) -> Tree a -> Tree b

  map f tree = Node { value = f(tree.value)
                    , children = map (map f) (tree.children)
                    }
```

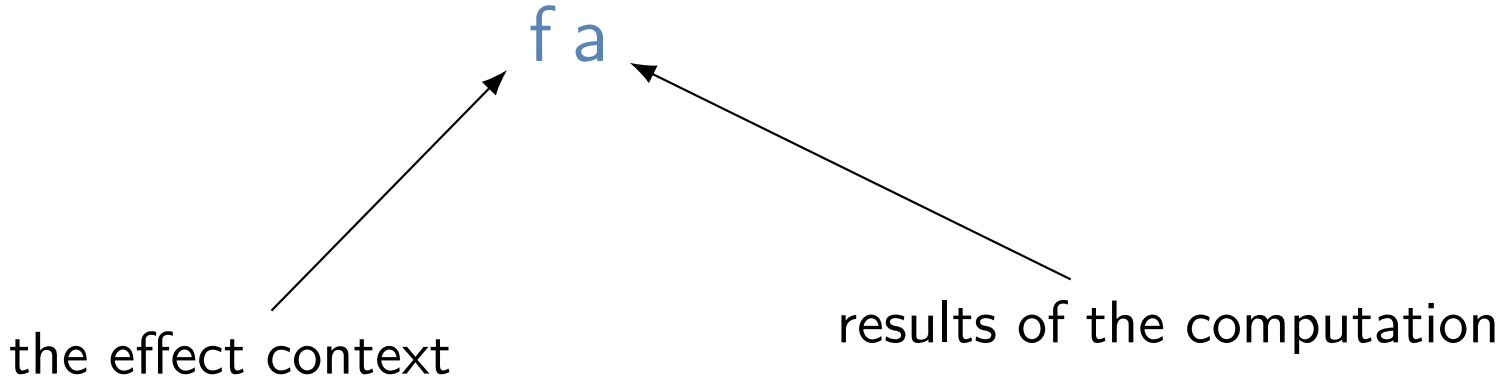
Recap: Effects

Effects refer to ordinary computations augmented with some extra capabilities. We represent effects with types `f : Type -> Type` that *lift* **calculations to actions**.



Recap: Effects

Effects refer to ordinary computations augmented with some extra capabilities. We represent effects with types `f : Type -> Type` that *lift* **calculations to actions**.



There are many different, commonly-used effects. These can all be expressed as Functors, but in practice we need more power to use them than `map` alone can give.

Recap: Effects

`List a`
(non-determinism)

`Pair c a`
(conjunction)

`Maybe a`
(partiality)

`IO a`
(input/output)

`Either e a`
(disjunction)

`Reader r a`
(environment)

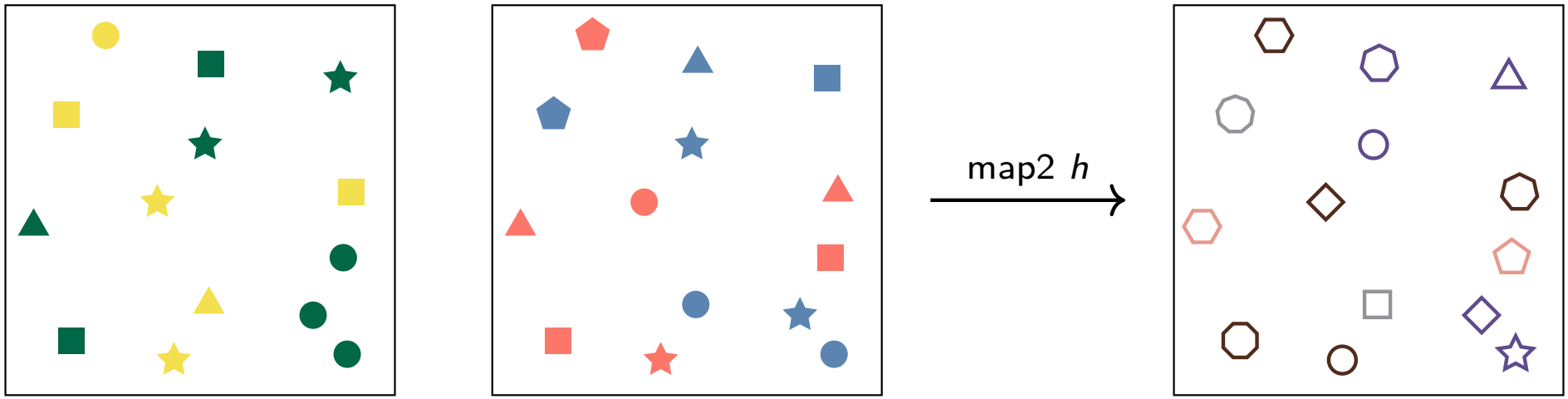
`...`
many more

`Random g a`
(randomness)

`Writer w a`
(logging)

`State s a`
(updating state)

Recap: Applicative Functors



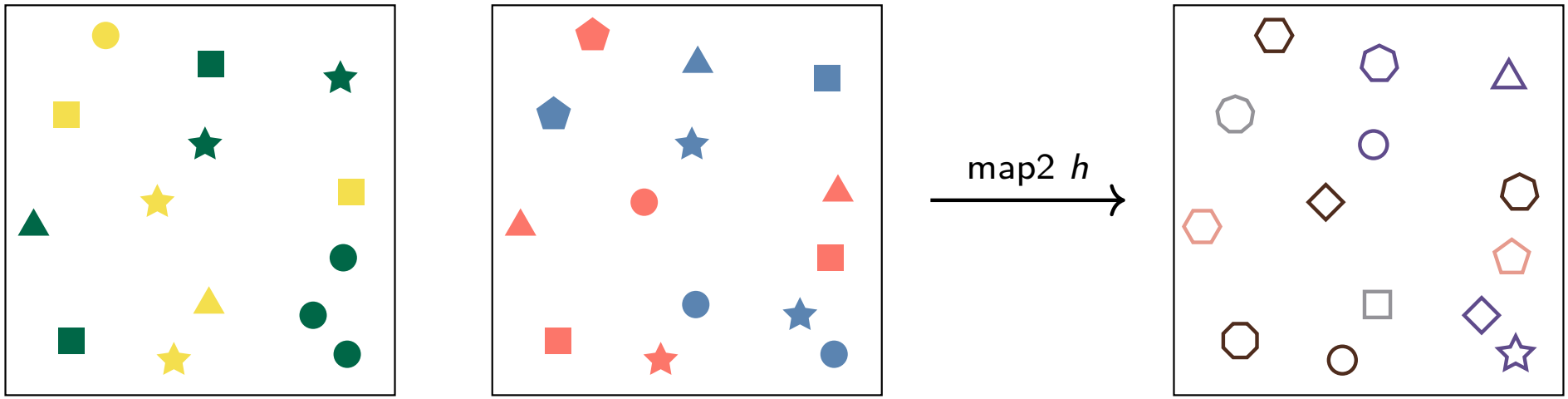

```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c     -- lift2 := map2 h
    ap   : f (a -> b) -> f a -> f b
    unit : f Unit                                -- Unit equiv ()
    combine : f a -> f b -> f (a, b)
```

Can derive pairs pure and map2, pure and ap, and unit and combine from each other.

```
laws Applicative where
    combine unit a  ~= a ~= combine a unit
    combine a (combine b c) ~= combine (combine a b) c
    combine (map g fa) (map h fb) == bimap g h (combine fa fb)
```

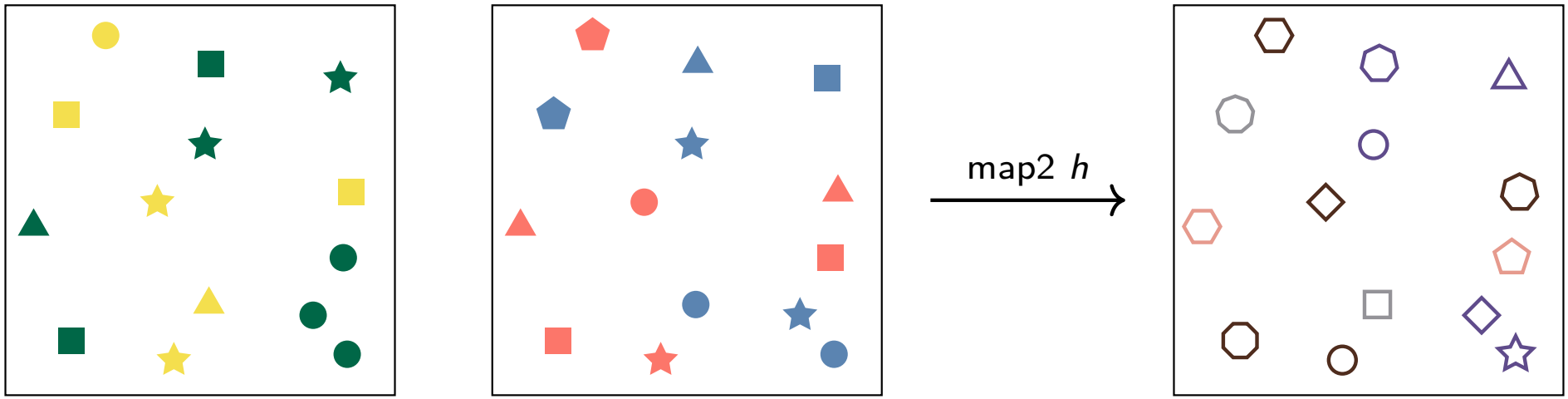
The laws can also be stated in terms of pure and ap or pure and map2.

Recap: Applicative Functors




```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c     -- lift2 := map2 h
    ap   : f (a -> b) -> f a -> f b
    unit : f Unit                                -- Unit equiv ()
    combine : f a -> f b -> f (a, b)

implements Applicative Maybe where
    pure : a -> Maybe a
    pure = Some

    map2 : (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
    map2 f (Some x) (Some y) = Some (f x y)
    map2 f _ _ = None

map2 (+) (Some 10) (Some 90) == Some 100
```

Recap: Applicative Functors




```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c     -- lift2 := map2 h
    ap   : f (a -> b) -> f a -> f b
    unit : f Unit                                -- Unit equiv ()
    combine : f a -> f b -> f (a, b)

implements Applicative List where
    pure : a -> List a
    pure x = [x]

    map2 : (a -> b -> c) -> List a -> List b -> List c
    map2 f xs ys = [f x y for x <- xs, y <- ys]

map2 (+) [1, 2, 3] [10, 11] == [11, 12, 12, 13, 13, 14]
```

Recap: Monads

A **monad** is a strategy for structuring, composing, and sequencing computations augmented with additional context.

Monads are Applicative Functors with even more power and expressiveness. They give us a way to compose *programs* accounting for their effects and returned values.

```
trait Applicative m => Monad (m : Type -> Type) where
    bind : m a -> (a -> m b) -> m b
    join : m (m a) -> m a

    -- derived method, look familiar?
    kleisli : (b -> m c) -> (a -> m b) -> (a -> m c)
```

Both `bind` and `join` can be defined in terms of the other.

```
laws Monad where
  bind (pure x) f == f x
  bind m pure == m
  bind (bind m f) g == bind m (\x -> bind (f x) g)
```

Recap: Monads

A **monad** is a strategy for structuring, composing, and sequencing computations augmented with additional context.

Monads are Applicative Functors with even more power and expressiveness. They give us a way to compose *programs* accounting for their effects and returned values.

```
trait Applicative m => Monad (m : Type -> Type) where
    bind : m a -> (a -> m b) -> m b
    join : m (m a) -> m a

    -- derived method, look familiar?
    kleisli : (b -> m c) -> (a -> m b) -> (a -> m c)
```

Laws easier to express (and more familiar!) in terms of `kleisli`:

```
kleisli pure f == f
kleisli f pure == f
kleisli (kleisli f g) h == kleisli f (kleisli g h)
```

Recap: Monads

A **monad** is a strategy for structuring, composing, and sequencing computations augmented with additional context.

Monads are Applicative Functors with even more power and expressiveness. They give us a way to compose *programs* accounting for their effects and returned values.

```
trait Applicative m => Monad (m : Type -> Type) where
    bind : m a -> (a -> m b) -> m b
    join : m (m a) -> m a

    -- derived method, look familiar?
    kleisli : (b -> m c) -> (a -> m b) -> (a -> m c)

implements Monad Maybe where
    bind : Maybe a -> (a -> Maybe b) -> Maybe b
    bind None _ = None
    bind (Some x) f = f x

    join : Maybe (Maybe a) -> Maybe a
    join None = None
```

Syntactic Sugar: do-Notation

Computations in a monadic context become much easier to use if we can describe them more easily. The **do-notation** is a syntactic sugar that converts code into applications of `bind`. `fp-concepts` has do notation as well, but it is constrained by the syntactic requirements of Python.

Here, `indexOf : List Int -> Int -> Maybe Int`.

```
stuff : Maybe (List Int)
stuff = Some [10, 20, 30, 40]

got20a : Maybe (List Int)
got20a = do
  x <- stuff
  ind <- indexOf x 20
  pure ind

-- for instance:
indexOf xs x = go 0 xs
  where
    go _ [] = Nothing
    go n (a : xs)
      | a == x    = Some n
      | otherwise = go (n + 1) as
```

```
stuff: Maybe[List[int]] = Some([10, 20, 30, 40])

@do(Maybe)    # defines got20a : Maybe[List[int]]
def got20a():
    xs = yield Some([10, 20, 30, 40])
    ind = yield index_of(20, xs)
    return ind

# for instance:
def index_of(x: int, xs: List[int]) -> Maybe[int]:
    try:
        ind = xs.index(x)
        return Some(ind)
    except ValueError:
        return Nothing()
```

Syntactic Sugar: do-Notation

Computations in a monadic context become much easier to use if we can describe them more easily. The **do-notation** is a syntactic sugar that converts code into applications of `bind`. `fp-concepts` has do notation as well, but it is constrained by the syntactic requirements of Python.

Here, `indexOf : List Int -> Int -> Maybe Int`.

```
stuff : Maybe (List Int)
stuff = Some [10, 20, 30, 40]

got20b : Maybe (List Int)
got20b = do
  x <- stuff
  ind <- indexOf x 25
  pure ind

-- for instance:
indexOf xs x = go 0 xs
  where
    go _ [] = Nothing
    go n (a : xs)
      | a == x    = Some n
      | otherwise = go (n + 1) as
```

```
stuff: Maybe[List[int]] = Some([10, 20, 30, 40])

@do(Maybe)    # defines got20b : Maybe[List[int]]
def got20b():
    xs = yield Some([10, 20, 30, 40])
    ind = yield index_of(25, xs)
    return ind

# for instance:
def index_of(x: int, xs: List[int]) -> Maybe[int]:
    try:
        ind = xs.index(x)
        return Some(ind)
    except ValueError:
        return Nothing()
```

Recap: Folds, Traversals, and Filters

Contexts that can be reduced to a summary value one piece at a time are ***foldable***:

```
trait Foldable (f : Type -> Type) where
    foldM : Monoid m => (a -> m) -> f a -> m
    fold  : (a -> b -> a) -> a -> f b -> a
```

Contexts in which elements can be removed are ***filterable***:

```
trait Functor f => Filterable (f : Type -> Type) where
    mapMaybe : (a -> Maybe b) -> f a -> f b
```

Contexts that can be transformed to one of the same *shape* by executing an *effectful* function one element at a time are ***traversable***:

```
trait (Functor t, Foldable t) => Traversable (t : Type -> Type) where
    traverse : Applicative f => (a -> f b) -> t a -> f (t b)
    sequence : Applicative f => t (f a) -> f (t a)
```

Plan

Revisiting Functors and Friends

Detailed Examples

Case Study: Parsing

Reader, Writer, and State, Oh My!

Consider two tasks related to a binary tree: counting the leaf nodes and relabeling all the nodes sequentially.

We can use traversals and optics to do both tasks easily, but consider the case where we were going to take a direct approach.

Reader, Writer, and State, Oh My!

```
def gen(b: int):
    return ( b
           , b + 2 if b < 32 and b % 2 == 0 else Tip
           , b + 1 if b < 32 and b % 3 == 0 else Tip
           )

bt = binary_tree(gen, seed=6)
```

Reader, Writer, and State, Oh My!

```
def number_of_leaves[A](t: BinaryTree[A] | Tip_) -> int:
    match t:
        case Tip_():
            return 0

        case BinaryTree(left_subtree, _, right_subtree):
            if left_subtree == Tip and right_subtree == Tip:
                return 1
            return number_of_leaves(left_subtree) + number_of_leaves(right_subtree)

        case _:
            return 0

# number_of_leaves(bt)
# #=> 6  Good!
```

Reader, Writer, and State, Oh My!

```
def relabel_sequentially[A](
    t: BinaryTree[A] | Tip_,
    n=0
) -> tuple[BinaryTree[tuple[int, A]] | Tip_, int]:
    match t:
        case Tip_():
            return (Tip, n)

        case BinaryTree(left_subtree, value, right_subtree):
            left_prime, p = relabel_sequentially(left_subtree, n + 1)
            right_prime, r = relabel_sequentially(right_subtree, p)
            return (BinaryTree.make((n, value), left_prime, right_prime), r)

        case _:
            return (t, 0)
```

Reader, Writer, and State, Oh My!

The first of these is just a standard traversal, but we've had to write a specialized function for it.

The second is tricky and it **conflates the plumbing** of a simple traversal with the "business logic" of the calculation.

The return type reflects this.

It would be nice to decouple these things.

Reader, Writer, and State, Oh My!

These are computations with an underlying state.

```
newtype State s v = State (s -> (v, s))
```

This captures a pattern that provides a (monadic) computational context for transparently using state.

```
@effect(State)
def relabel_statefully(t):
    if t == Tip:
        return t

    n = yield State.get
    _ = yield State.put(n + 1)

    left, value, right = t.node_contents
    lp = yield relabel_statefully(left)
    rp = yield relabel_statefully(right)

    return BinaryTree.make((n, value), lp, rp)
```

Reader, Writer, and State, Oh My!

These are computations with an underlying state.

```haskell
newtype State s v = State (s -> (v, s))
```

This captures a pattern that provides a (monadic) computational context for transparently using state.

```python
@effect(State)
def count_leaves_statefully(t):
    if t == Tip:
        return ()

    if BinaryTree.is_leaf(t):
        _ = yield State.modify(lambda n: n + 1)
        return ()

    left, _, right = t.node_contents
    _ = yield count_leaves_statefully(left)
    _ = yield count_leaves_statefully(right)

    return ()
```

Reader, Writer, and State, Oh My!

```
def map[C](self, g: Callable[[A], C]) -> State[S, C]:
  def g_state(s):
      a, s_prime = self._state(s)
      return (g(a), s_prime)
  return State(g_state)

@classmethod
def pure(cls, a):
    return cls(lambda s: (a, s))

def map2[B, C](self, g: Callable[[A, B], C], fb: State[S, B]) -> State[S, C]:
    def g_state(s):
        a, s1 = self._state(s)
        b, s2 = fb._state(s1)
        return (g(a, b), s2)
    return State(g_state)

def bind[B](self, g: Callable[[A], State[S, B]]) -> State[S, B]:
    def bind_state(s):
        a, s1 = self._state(s)
        st = g(a)
        return st._state(s1)
    return State(bind_state)
```

Reader, Writer, and State, Oh My!

Computations that have access to data in an environment.

```
newtype Reader env val = Reader (env -> val)
```

Example:

```
@effect(Reader)
def line(x):
    a = yield ask_for('slope')
    b = yield ask_for('intercept')

    return a * x + b

line(4).run({'slope': 9, 'intercept': 1000, 'delta': 0})
#=> 1036
```

Probability Distributions

Let's construct a monadic context for working with probability distributions on finite sets. (Assume a predicate `equal: a -> a -> Bool` for the value types. We can handle that but will ignore it for now.)

```
class FinDist[A](Monad):
    def __init__(
        self,
        prob_val_pairs: List[Pair[float, A]],
        equal: Callable[[A, A], bool] = eq
    ):
        self._pvs = self._normalize(self._squash(prob_val_pairs)
        self._equal = equal
        super().__init__()

    def map[B](self, g: Callable[[A], B]) -> FinDist[B]:
        return FinDist(self._pvs.map(lift(g)))

    @classmethod
    def pure(cls, a):
        ...

    def map2(self, g, fb):
        ...

    def bind(self, g):
        ...

    @classmethod
    def __do__(cls, make_generator, is_generator):
        ...
```

Probability Distributions

We can define factories for common distributions, and the types give us combinators for the primary operations.

```
die_roll = uniform(1, 2, 3, 4, 5, 6)
two_dice = ap(pair, die_roll, die_roll)

@do(FinDist)
def sum_of_two_rolls():
    x, p = yield die_roll
    y, q = yield die_roll
    return pair(x + y, p * q)

def on_slice(a):
    return uniform(a - 1, a, a + 1)

@do(FinDist)
def mixture():
    x, p = yield uniform(1, 2, 3, 4)
    y, q = yield on_slice(x)
    return pair(pair(x, y), p * q))

marginal_x = mixture.map(itemgetter(0))
marginal_y = mixture.map(itemgetter(1))
```

Reader, Writer, and State, Oh My!

Input/Output effects can be captured this way, giving pure functions that represent those effects.

The effects are enacted at runtime.

What do these do? (Recall that `traverse : (a -> f b) -> t a -> f (t a)` for Applicative `f` and Traversable `t`.)

```
traverse(lambda x: IO(lambda: as_io(print(f"At node {x}"), x + 10)),
         rt, IO)

traverse(lambda x: IO(
    lambda: as_io(print(f"At {'even' if x % 2 == 0 else 'odd'} node {x}"),
                  x + 10)),
         rt, IO)

io_actions = List.of(
    IO(lambda: as_io(print("foo"), 10)), IO(lambda: as_io(print("bar"), 20)
    IO(lambda: as_io(print("zap"), print("ok"), 30)),
    IO(lambda: as_io(print("wow"), print("***"), 40))
)
tmp = sequence(io_actions, IO)
tmp2 = IO.run(tmp)
```

Modeling Pokemon*

The goal of the game is "knock out" six of your opponent's Pokemon.

- You do this with **attacks**.
- Attacks cost **energy**.
- Attacks do **damage**.
- The **hp** field describes how much damage a character can take.

*Borrowed shamelessly from Alejandro Serrano

Modeling Pokemon*

Each Pokemon card comes with

1. Name (e.g., Pikachu)
2. Kind (e.g., ⚡)
3. HP (e.g., 70)
4. Various Attacks
5. $\cdots$ (ignore the rest for now)

*Borrowed shamelessly from Alejandro Serrano

Modeling Cards


```
data Card = PokemonCard { name     : String
                        , kind     : Energy
                        , hp       : Natural
                        , attacks  : List Attack
                        }
          | EnergyCard { kind : Energy
                       , amount : Natural
                       }
          | ...
```

Energy

There are ten types of energy in the game:

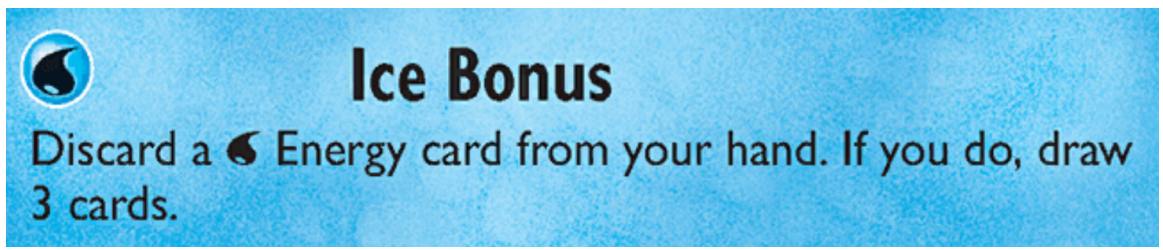
9 are “regular” and “colorless”  acts as a wildcard.

```
data Energy = Colorless
            | Grass | Fire | Water
            | Lightning | Fighting | Psychic
            | Darkness | Metal | Dragon
```

gives 2 lightning energy and 1 of any other kind.

Attacks

Attacks have name, cost (of each kind), and damage, but ***they do more than damage***.



Attacks

Attacks are actions that can depend on state (e.g., cards in hand, coin flips).

These can include randomization, drawing cards, discarding cards, and assorted conditionals and loops.

How do we model Actions?

Attacks (cont'd)

```
type Hand = List Card
data FlipOutcome = Tails | Heads

data Action
  = FlipCoin (FlipOutcome -> Action)
  | DrawCard (Maybe Card -> Action)
  | QueryHand (Hand -> Action)
  | ...
  | Damage Natural
```

`Damage` is the final action in the sequence, the other forms generate a "value" that is used to determine the next action in the sequence.

Attacks (cont'd)


```
surpriseAttackAction
  = FlipCoin $ \case Heads -> Damage 30
                     Tails -> Damage 0
```

Attacks (cont'd)


```
ironTailAction = go 0
  where go total = FlipCoin $ \case Heads -> go (total + 30)
                                    Tails -> Damage total
```

Attacks (cont'd)

During the game, we *interpret* this domain-specific language to generate the actions

```
interpretRandom : Action -> IO Natural
interpretRandom (Damage d) = pure d
interpretRandom (FlipCoin f) = do
    outcome <- flipCoin
    interpretRandom (f outcome)
```

A Useful Pattern

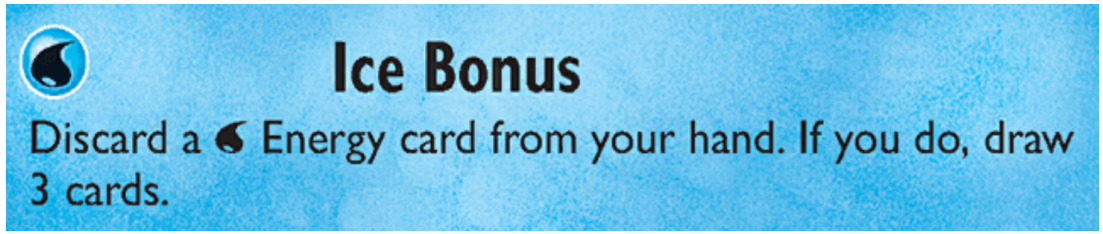
We can *separate* the two parts of this design: the operation of sequencing and the concrete instructions that represent the generated values.

```
data Action a where
  FlipCoin  : Action
  DrawCard  : Action
  QueryHand : Action

data Program : (Type -> Type) -> Type -> Type where
  Done : result -> Program instr result
  Then : instr result
         -> (result -> Program instr result')
         -> Program instr result'

-- ^^ This is a monad ^^

-- Create a program from a single instruction/action
perform : instr r -> Program instr r
perform action = action `Then` Done
```

A Useful Pattern (cont'd)


```
ironTailAction : Program Action Natural
ironTailAction = do
  outcome <- perform FlipCoin
  case outcome of
    Tails -> pure 0
    Heads -> map (\x -> 30 + x) ironTailAction

-- Alternate elegant form using monadic looping
ironTailAction : Program Action Natural
ironTailAction = do
  flips <- repeatWhileM (== Heads) (perform FlipCoin)
  pure (30 * length flips)
```

Exercise



How would we implement this card's actions?

Interpreting the Language

We can now interpret the language in any desired computational context.

```
interpret : Monad m
          => (forall a. instr a -> m a)
          -> Program instr r
          -> m r
interpret f = go
  where
    go (Done d) = pure d
    go (action `Then` next) = do
      x <- f action
      go (next x)
```

Plan

Revisiting Functors and Friends

Detailed Examples

Case Study: Parsing

What is a Parser?

A **parser** takes unstructured data as input and returns a structured representation of the data as output.

Classic examples:

- a programming language as text is converted to a *syntax tree* describing the code
- a text file in CSV format converted to a data frame,
- a series of network packets containing JSON converted to a nested record/dict.

Parsers and the operations that act on them to give *meaning* to the structured representations (e.g., compilers) have wide and frequent application.

Here, we will use parsers as a mechanism to think about programming approaches, patterns, and concepts.

We will build context-sensitive parsers out of smaller modular units called ***combinators***.

Starter Parsers

We want a parser that reads a string and gives us a character and a parser that reads a string and gives us a natural number. How can we represent these as types?

Starter Parsers

We want a parser that reads a string and gives us a character and a parser that reads a string and gives us a natural number. How can we represent these as types?

Try:

```
char : String -> Maybe Char
natural : String -> Maybe Natural
```

Why is Maybe here? What other basic operations might we want? Think about how we might process data or programs? How might we get two natural numbers in a CSV file? How might we read an alphabetic word? What operations do we need?

Examples: `splitAtChar`, `twoThings`

Parsing

Starting with a parser `char` to (maybe) read a character from a string, of type `String -> Maybe Char`. But this leaves us unable to do anything else.

Parsing

We can extend this to type `String -> Pair String (Maybe Char)` and similarly for the parser `natural` of type `String -> Pair String (Maybe Natural)`.

What does the `String` represent?

This suggests a type

```
type Parser a = String -> Pair String (Maybe a)
```

We can generalize this further (parser state).

Can we compose parsers?

Parsing

```
type Parser a = String -> Pair String (Maybe a)
```

Can we compose parsers? **Yes!**

```
both : Parser a -> Parser b -> Parser (Pair a b)
```

```
def both(parser1: Parser[a], parser2: Parser[b]) -> Parser[Pair[a, b]]:
    def do_both(input1: str) -> Pair[str, Maybe[Pair[a, b]]]:
        input2, mc = parser1(input1)
        if not mc:
            return (input, None_())
        input3, mn = parser2(input2)
        if not mn:
            return (input, None_())
        return (input3, map2(pair, mc, mn))
    return do_both
```

This can be made more elegant with monadic computation!

Next Steps

We will build up to a fully functional parser algebra, starting from simple parsers, and adding power to the abstraction as we go.

Things to look for:

Components

1. Carrier type (`f a -> a`, TBD)
2. Introduction Forms
3. Combinators
4. Elimination Forms
5. Laws

Features:

1. Composable
2. Lawful
3. Polymorphic
4. Uses the Least Powerful Abstraction that works
5. Specifies an interface/contract
6. Open

Next steps (cont'd)

Let's begin with a couple more starter parsers to sharpen the ideas and we'll build from there.

```
void : Parser Unit                          -- Parser that always fails
empty : Parser Unit                         -- Parser that always succeeds
char : Char -> Parser Char                  -- Parser that accepts only given
chars_in : Iterable Char -> Parser Char     -- Parser that accepts one of give
natural : Parser Natural                    -- Parser for natural numbers
```

And a few combinators (among many)

```
pipe : Parser a -> (a ->  Parser b) -> Parser b
use : Parser a -> b -> Parser b             -- uses given value when parser su
alt : Parser a -> Parser b -> Parser (Either a b)
seq : Parser a -> Parser b -> Parser (Pair a b)
optional: Parser a -> a -> Parser a
```

The End