Functional Thinking, Part 1 Prep

Christopher R. Genovese

Department of Statistics & Data Science

Thu 09 Oct 2025
Session #14

Plan

Recap and Debrief

Plan

Plan

Plan

Plan

Announcements

- Questions?
- **Reading**:
  - Hughes, Why Functional Programming Matters
  - Commentary on Hughes with Python examples
  - Starting sequence on categories
    - What is Category Theory?
    - Definitions and Examples
    - What is a Functor? Part 1
    - What is a Functor? Part 2
    - Fibonacci Functor
    - Natural Transformations
  - Backus, Can Programming be Liberated from the Von Neumann Architecture
- **Homework**:
  - **classification-tree-basic** assignment due Thu 23 Oct.
  - Push outstanding mini-assignments when complete

Plan

FP Intro Activities

Choose one of the tasks below at an appropriate challenge level. **Do not use loops or any imperative constructs**.

1. Cow Proximity

   Suppose that cows of the same breed get into an argument with each other if they are standing too close together. Two cows of the same breed are "crowded" if their positions within a line of n cows differ by no more than k, where $1 \leq k < n$.

   Given a value of k and a sequence representing the breed IDs of a line of cows, compute the maximum breed ID of a pair of crowded cows.

   If there are no crowded cows, return -1.

2. Write a function that sums a list of numbers.

3. Write a function that takes a list/vector of integers and returns only those elements for which the *preceding* element is negative.

4. Write a function that takes in a string and returns true if all square `[ ]`, round `( )`, and curly `{ }` delimiters are properly paired and legally nested, or returns false otherwise. For example, `[(a)]{[b]}` is legally nested, but `{a}([b)]` is not.

   Hint: What data structure can you use to track the delimiters you've seen so far?

FP Intro Activities

Choose one of the tasks below at an appropriate challenge level. **Do not use loops or any imperative constructs**.

5. Write a function `roman` that parses a Roman-numeral string and returns the number it represents. You can assume that the input is well-formed, in upper case, and adheres to the "subtractive principle" (link). You need only handle positive integers up to MMMCMXCIX (3999), the largest number representable with ordinary letters. You can also define constant objects or expressions (e.g., a dictionary to map characters like X, I, V, etc. to their corresponding value).

6. Write a function `chain` that takes as its argument a list of functions, and returns a new function that applies each function in turn to its argument. (This requires a language with first-class functions, like R or Python.)

   For example,

```python
import math

positive_root = chain([abs, math.sqrt])

positive_root(-4)   #=> 2.0
```

FP Intro Activities

Choose one of the tasks below at an appropriate challenge level. **Do not use loops or any imperative constructs**.

7. Write a function `partial` that takes a function and several arguments and returns a function that takes additional arguments and calls the original function with all the arguments in order. For example,

```
foo <- function(x, y, z) { x + y + z }
bar <- partial(foo, 2)

bar(3, 4) #=> 2 + 3 + 4 = 9
```

Note: You don't actually need `map`, `reduce`, or `filter` for this, but it's still good to know how to do it.

Higher-Order Functions

Examples of some commonly used higher-order functions. Below, we use `f : Type -> Type` to represent a container/collection of values of a specified type, e.g., `f = List`, `f = Vector n`, `f = Maybe`, or `f = RoseTree`.

- `map : (a -> b) -> f a -> f b`
  The `map` operation calls a function on every element of a collection and returns a collection of the same type and shape with the results.
- `filter : (a -> Bool) -> f a -> f a`
  The `filter` operation uses a predicate to select elements of a collection, keeping the elements for which the predicate is true.
- `mapMaybe : (a -> Maybe b) -> f a -> f b`
  The `mapMaybe` operation combines `map` and `filter`, keeping the transformed values `v` for which the given function returns `Some v`.
- `fold : (acc -> elt -> acc) -> acc -> f elt -> acc`
  The `fold` operation implements the fold pattern.

Plan

Recap and Debrief

A Surprising Example

A First Look at Functors

More Examples

Word Search Fusion (if time allows)

Advent of Code Challenge: Part 1

You've managed to sneak in to the prototype suit manufacturing lab. The Elves are making decent progress, but are still struggling with the suit's size reduction capabilities.

While the very latest in 1518 alchemical technology might have solved their problem eventually, you can do better. You scan the chemical composition of the suit's material and discover that it is formed by extremely long polymers (one of which is available as your puzzle input).

The polymer is formed by smaller units which, when triggered, react with each other such that two adjacent units of the same type and opposite polarity are destroyed. Units' types are represented by letters; units' polarity is represented by capitalization. For instance, r and R are units with the same type but opposite polarity, whereas r and s are entirely different types and do not react.

For example:

- In `aA`, `a` and `A` react, leaving nothing behind.
- In `abBA`, `bB` destroys itself, leaving `aA`. As above, this then destroys itself, leaving nothing.
- In `abAB`, no two adjacent units are of the same type, and so nothing happens.
- In `aabAAB`, even though `aa` and `AA` are of the same type, their polarities match, and so nothing happens.

Now, consider a larger example, `dabAcCaCBAcCcaDA`.

```
dabAcCaCBAcCcaDA  The first 'cC' is removed.
dabAaCBAcCcaDA    This creates 'Aa', which is removed.
dabCBAcCcaDA      Either 'cC' or 'Cc' are removed (the result is the same).
dabCBAcaDA        No further actions can be taken.
```

After all possible reactions, the resulting polymer contains 10 units.

How many units remain after fully reacting the polymer you scanned?

Exploiting Algebraic Structure

How would you approach this? Let's sketch it out.

Exploiting Algebraic Structure

Thinking functionally leads us to consider this in a different context. We can interpret the data *algebraically* and exploit that structure.

```
foldMap : (Foldable t, Monoid m) => (a -> m) -> t a -> m

-- This maps Component -> Structure
inject : Char -> FreeGroup Char
inject c
    | isAlpha c and isLowerCase c = free c
    | isAlpha c and isUpperCase c = invert (free (toLower c))
    | otherwise                   = unit  -- group identity

-- This maps Data -> Structure
represent : List Char -> FreeGroup Char
represent = foldMap inject

solution = length . toList . represent
```

Exploiting Algebraic Structure

```
newtype FreeGroup a = FreeGroup (List (Either a a))

toList : FreeGroup a -> List (Either a a)
toList (FreeGroup xs) = xs

...
```

We will see shortly how easy all this is.

Advent of Code Challenge: Part 2

Time to improve the polymer.

One of the unit types is causing problems; it's preventing the polymer from collapsing as much as it should. Your goal is to figure out which unit type is causing the most problems, remove all instances of it (regardless of polarity), fully react the remaining polymer, and measure its length.

For example, again using the polymer `dabAcCaCBAcCcaDA` from above:

- Removing all A/a units produces `dbcCCBcCcD`. Fully reacting this polymer produces `dbCBcD`, which has length 6.
- Removing all B/b units produces `daAcCaCAcCcaDA`. Fully reacting this polymer produces `daCAcaDA`, which has length 8.
- Removing all C/c units produces `dabAaBAaDA`. Fully reacting this polymer produces `daDA`, which has length 4.
- Removing all D/d units produces `abAcCaCBAcCcaA`. Fully reacting this polymer produces `abCBAc`, which has length 6.

In this example, removing all `C/c` units was best, producing the answer 4.

What is the length of the shortest polymer you can produce by removing all units of exactly one type and fully reacting the result?

Plan

Introduction to 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`!)

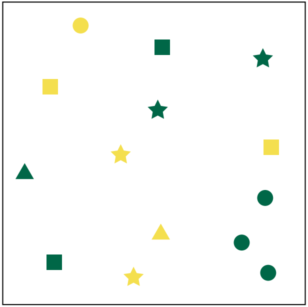
Functors satisfy two important **laws**:

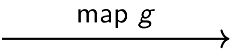
1. `map id == id`
2. `(map g) . (map f) == map (g . f)`

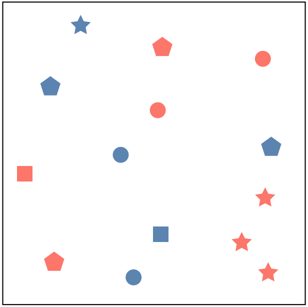
The first law says that lifting the identity gives you the identity. The second law says that composing the lift of g and the lift of f is the same as lifting $g \circ f$. This is equivalent to `compose (map f) (map g) == map (compose f g)`.

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)
```

Functors (cont'd)

What is the shape of a ______?

1. Maybe
2. List
3. Pair
4. Function `r -> __` (aka Reader r)
5. Tree
6. Dict

Functors (cont'd): Basic Examples

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

Some fundamental examples:

Functors (cont'd): 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 =
  map f (Some x) =
```

Functors (cont'd): 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)
```

Functors (cont'd): 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) =
```

Functors (cont'd): 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)
```

Functors (cont'd): 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 =

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

Functors (cont'd): 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)
```

Functors (cont'd): 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) =
```

Functors (cont'd): 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)
```

Rose Trees

We will frequently use a more general tree structure, *rose trees*:

```
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 =
```

Rose Trees

We will frequently use a more general tree structure, *rose trees*:

```
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 f (tree.children)
                    }
```

What does this do?

```
map (map (__ + 1)) [Some 10, None, Some -1, None, None, Some 99]
```

What does this do?

```
tree = Node {
  value=10
  , children=[ Node { value=20, children=[] }
             , Node { value=30, children=[ Node {value=40, children=[]} ] }
             , Node { value=50, children=[ Node {value=60, children=[]}
                                         , Node {value=70, children=[]}
                                         ]
                    }
             ]
}
map (__ * 2) tree
```

Plan

Minimax Analysis of Game Trees

Consider a traditional, two-player perfect-information game like tic-tac-toe or chess. We can analyze a game by looking at the "game tree" and scoring positions heuristically.

Assume we have types `Player` and `Position`. For instance,

```
data Player = X | O
data Position = TopLeft | TopMid | TopRight | ... | BotRight
```

We will let these be generic types as they can apply to any game.

To keep things simple, we'll start by ignoring the player, assuming that player can be determined from a position.

Minimax Analysis of Game Trees (cont'd)

We have a function

```
moves : Position -> List Position
```

and define (using rose trees)

```
propagate : (a -> List a) -> a -> Tree a
propagate f x = Node { value = x
                     , children = map (propagate f) (f x)
                     }
gameTree : Position -> Tree Position
gameTree = propagate moves
```

Minimax Analysis of Game Trees (cont'd)

We could handle the player as follows.

```
next : Player -> Player
moves : Player -> Position -> List Position
```

defining

```
propagate : (a -> List a) -> (a -> List a) -> a -> Tree a
propagate f1 f2 x = Node { value = x
                         , children = map (propagate f2 f1) (f1 x)
                         }
gameTree : Player -> Position -> Tree Position
gameTree player = propagate (moves player) (moves (next player))
```

But we'll keep to the simple version in what follows.

Minimax Analysis of Game Trees (cont'd)

Now imagine that we have some heuristic static evaluator for some position:

```
static : Position -> Number
```

Assume that the results are negative when they favor one player and positive when they favor another. This is a local guess that we will refine by analyzing the game tree. Note that `map static : Tree Position -> Tree Number`.

To extend our static analyzer, we lookahead in the tree, taking account of the best (greedy) move at each stage.

```
maximize : Tree Number -> Number
maximize (Node v []) = v
maximize (Node v sub) = max (map minimize sub)

minimize : Tree Number -> Number
minimize (Node v []) = v
minimize (Node v sub) = min (map maximize sub)

evaluate : Position -> Number
evaluate = maximize . map static . gameTree
```

This is fine, but it might not terminate. (Why?) And it can take a long time in any case.

Minimax Analysis of Game Trees (cont'd)

We need to prune the tree.

```
prune : Natural -> Tree a -> Tree a
prune 0 (Node v cs) =
prune n (Node v cs) =
```

Minimax Analysis of Game Trees (cont'd)

We need to prune the tree.

```
prune : Natural -> Tree a -> Tree a
prune 0 (Node v cs) = Node v []
prune n (Node v cs) = Node v (map (prune (n - 1)) cs)
```

Minimax Analysis of Game Trees (cont'd)

We need to prune the tree.

```
prune : Natural -> Tree a -> Tree a
prune 0 (Node v cs) = Node v []
prune n (Node v cs) = Node v (map (prune (n - 1)) cs)
```

Now we have a more realistic evaluation

```
evaluate : Position -> Number
evaluate = maximize . map static . prune 4 . gameTree
```

which gives a lookahead of 4 moves.

We are using *lazy evaluation* here. This evaluates positions only as *demanded* by `maximize` so the whole tree is never in memory.

We can optimize this further.

Newton-Raphson Square Roots

Consider the recurrence relation $a_{n+1} = (a_n + x/a_n)/2$ for $x > 0$ and $a_0 = x$. As n increases, $a_n \to \sqrt{x}$.

We might compute with this typically with

```
def sqrt(x, tolerance=1e-7):
    u = x
    v = x + 2.0 * tolerance
    while abs(u - v) > tolerance:
        v = u
        u = 0.5 * (u + x / u)
    return u
```

We will put this in a more modular style with ingredients that can be reused for other problems.

Newton-Raphson Square Roots (cont'd)

```
next : Real -> Real -> Real
next x a = (a + x / a) / 2

iterate f x = x :: (iterate f (f x))     -- a lazy sequence

iterate (next x) init    -- lazy sequence of sqrt approximations

within : Real -> List Real -> Real
within tol (a0 :: (a1 :: rest))
   | (abs(a0 - a1) <= tol) = a1
   | otherwise             = within tol (a1 :: rest)

relative : Real -> List Real -> Real
relative tol (a0 :: (a1 :: rest))
   | (abs(a0/a1 - 1) <= tol) = a1
   | otherwise               = relative tol (a1 :: rest)

a_sqrt init tol   = within tol (iterate (next x) init)
r_sqrt init tol x = relative tol (iterate (next x) init)
```

These same primitives apply to give us other approximations, e.g., numerical differentiation, integration, ...

Numerical Derivatives

```
divDiff f x h = (f (x + h) - f x) / h
halve x = x / 2
derivative f x h0 = map (divDiff f x) (iterate halve h0)

within tol (derivative f x h0)

-- sharpen error terms that look like a + b h^n
sharpen n (a :: (b :: rest))
  = ((b * (2**n) - a) / (2**n - 1)) :: (sharpen n (b :: rest))

order (a :: (b :: (c :: rest)))
  = round (log2 ((a - c) / (b - c) - 1))

improve seq = sharpen (order seq) seq

within tol (improve (derivative f x h0))

within tol (improve (improve (derivative f x h0)))
```

- ... can improve even further.

Plan

The End