The ***Tyranny*** of **Structurelessness**

How more ***meaningful*** code
can make your project more
resilient & maintainable

I have regarded it as the highest goal of programming language design to enable ***good ideas*** to be ***elegantly expressed***

Tony Hoare, Turing Award Lecture 1980

Give me the ***right word***
and the ***right accent***
and I will ***move the world***

Joseph Conrad on Archimedes Lever

Brooklyn Zelenka

@expede

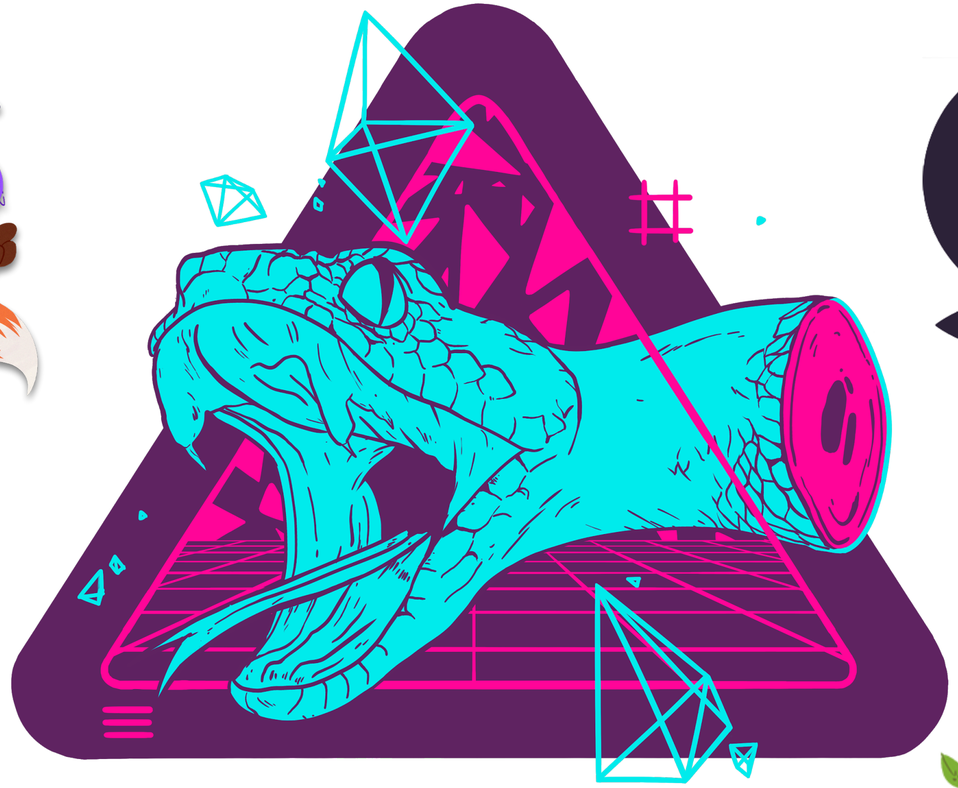
fission

Brooklyn Zelenka

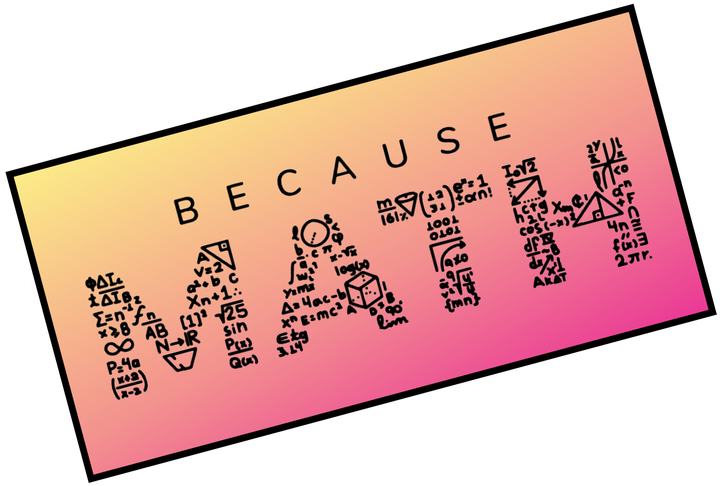
@expede

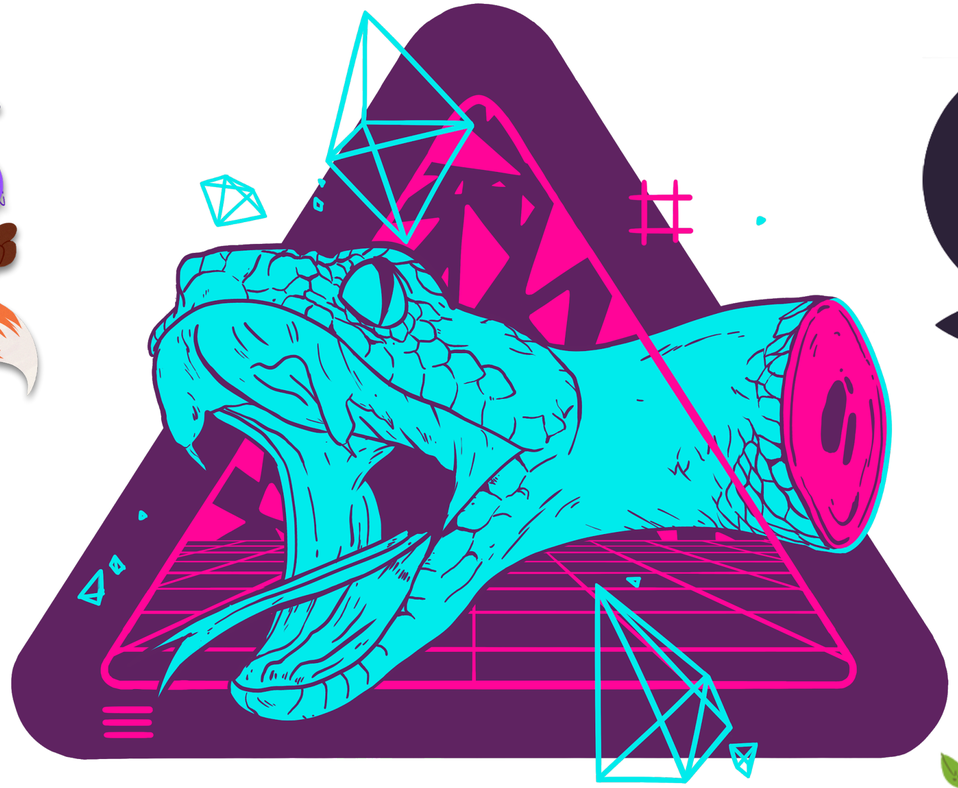
- CTO at Fission — https://fission.codes
  - WebNative
  - Making backends obsolete 🤫
- PLT, VMs, Distributed Systems, ETH Core
- Founder of the Vancouver FP meetup
- Witchcraft, Quark, Algae, Exceptional, and others
  - Exceptional (Elixir) → Rescue (Haskell)
  - Witchcraft (Elixir) ← Prelude (Haskell)

fission

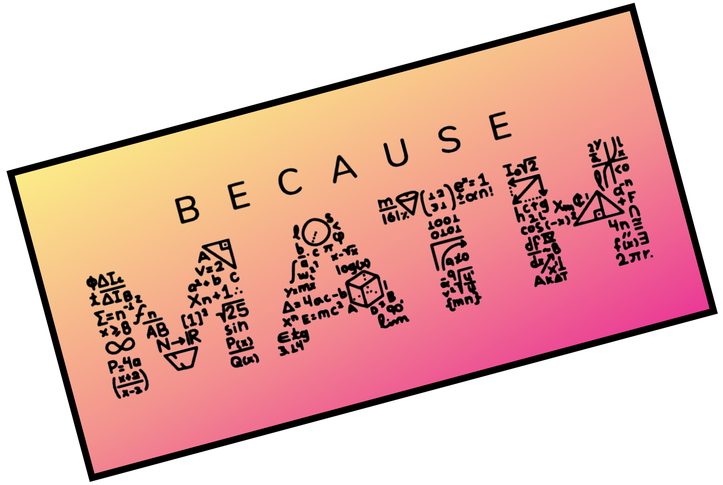


SCREAMING_SNAKE_CASE





SCREAMING_SNAKE_CASE



PING ME AND WE'LL MAIL SOME

Stickers!

This Talk is About...

This Talk is About...

- An approach to programming (broadly)
- Some observations about Elixir specifically
- A vision for the future of the ecosystem
- If you were at CodeBEAM BR, this talk generalizes some of the same ideas

This is the

Big Idea

Big Idea

One Liner

🛸 Work at a higher level 🔭

Big Idea

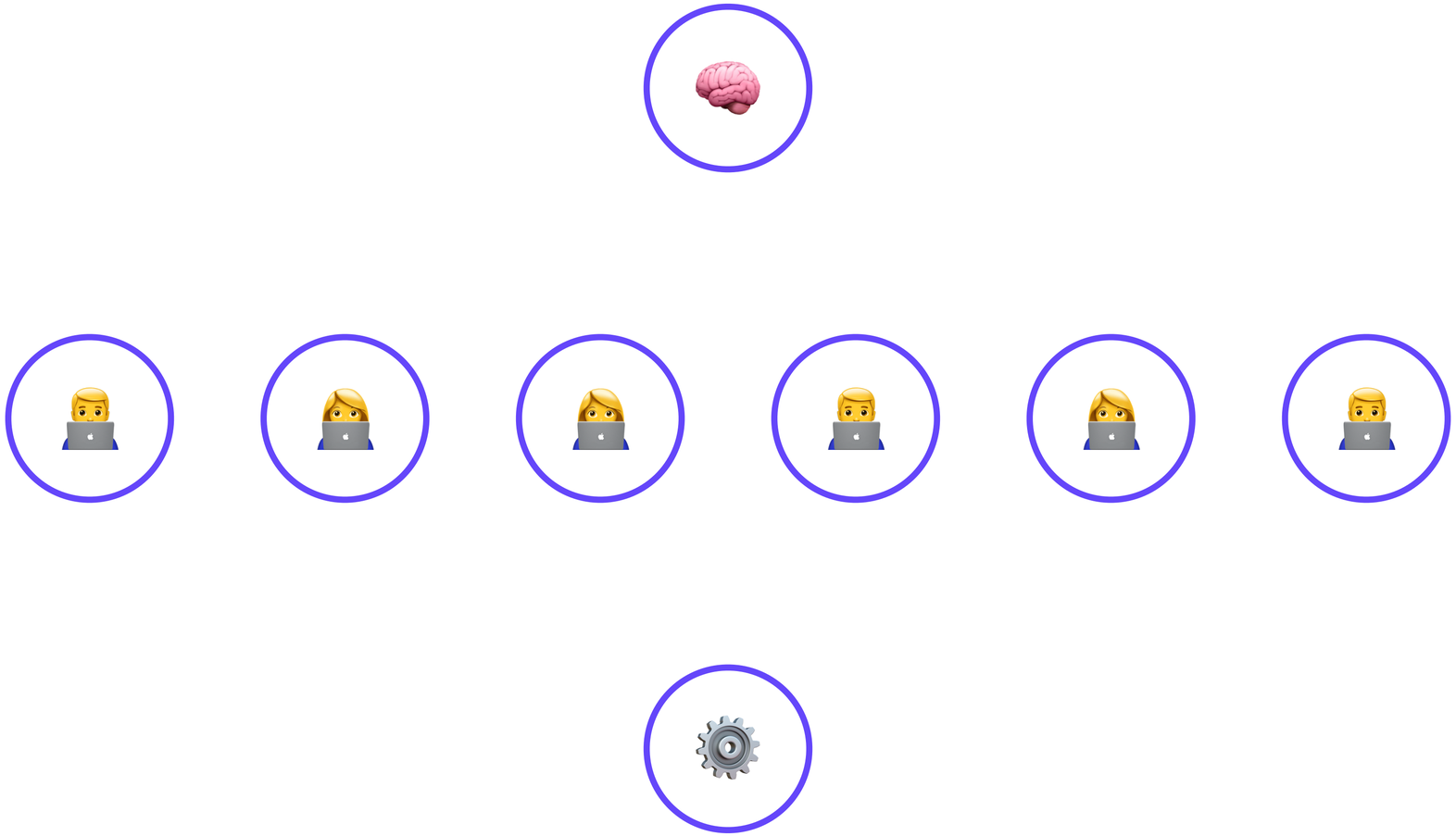
Language Design Reflects Intended Use

Big Idea

Who's Org Looks Like This?

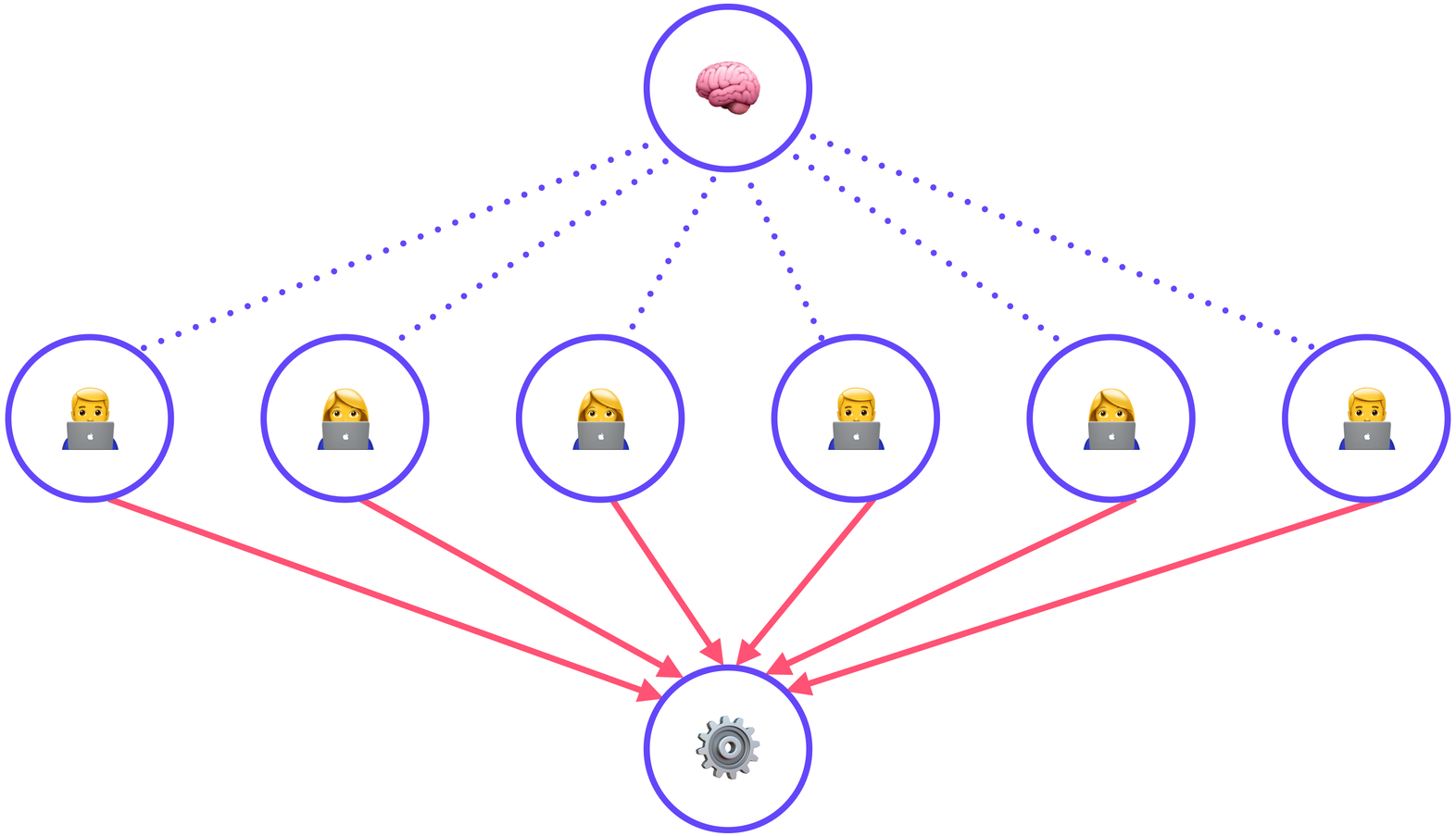
Big Idea

Who's Org Looks Like This?



Big Idea

Who's Org Looks Like This?

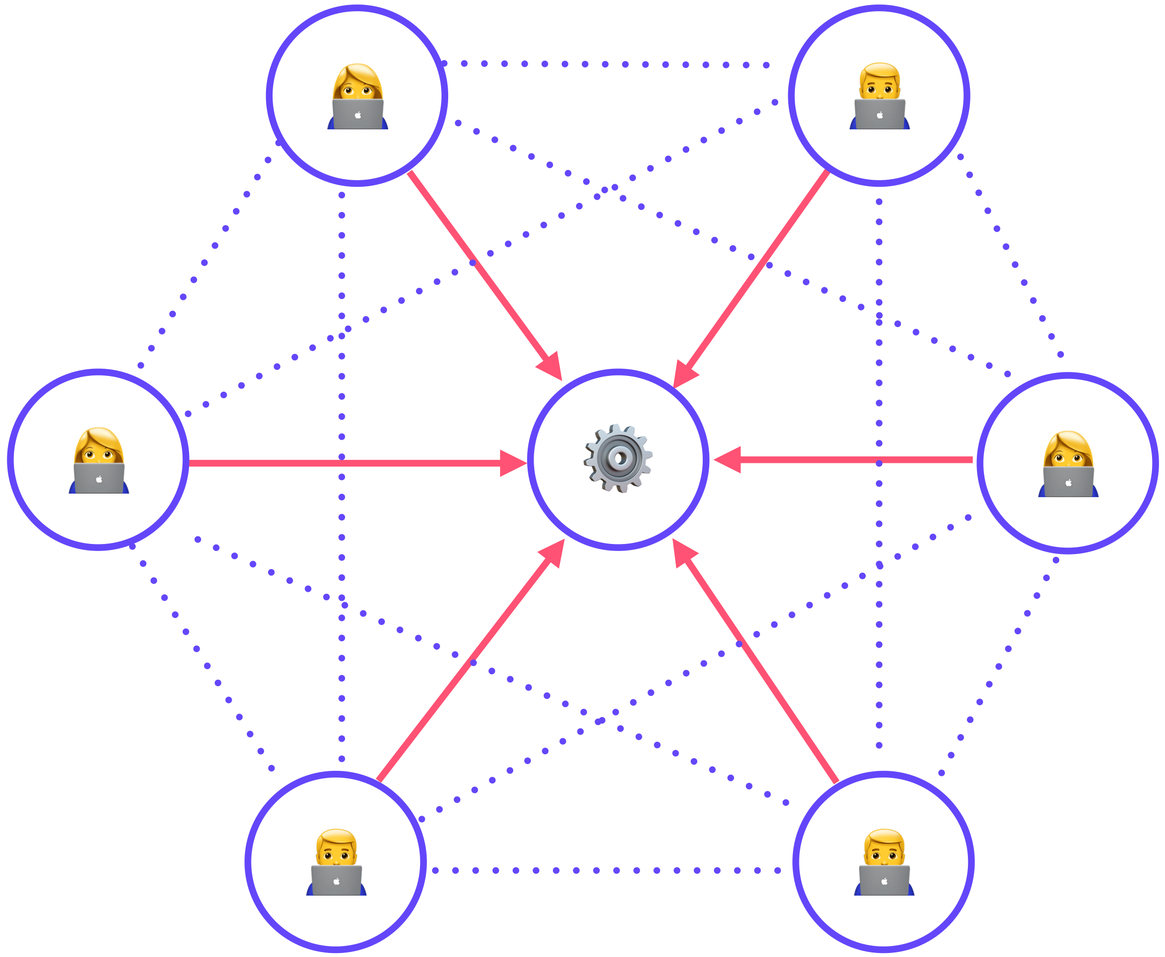


Big Idea

Who's Org Looks Like This?

Big Idea

Who's Org Looks Like This?



Big Idea

Forward Thinking

We want more type of features over time.
As a result, complexity grows at an exponential rate.

How do you make Elixir code
more flexible and easier to reason about at scale?

Do you think that the patterns we use today are
the best possible patterns for software?

How will you write code in 2025, 2030, and 2050?

Big Idea

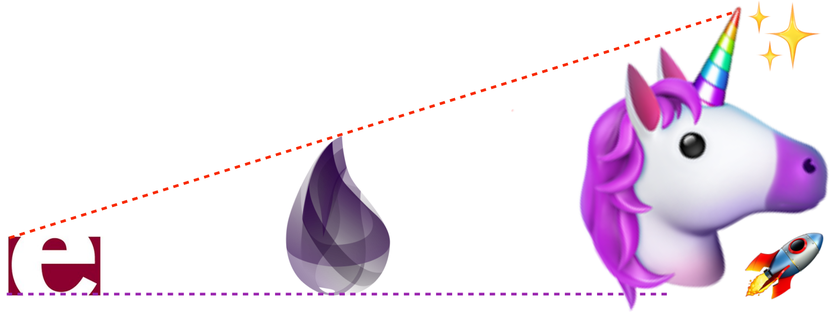
Core Evolution

We need to evolve our approach:
focus on domain and structure!

Big Idea

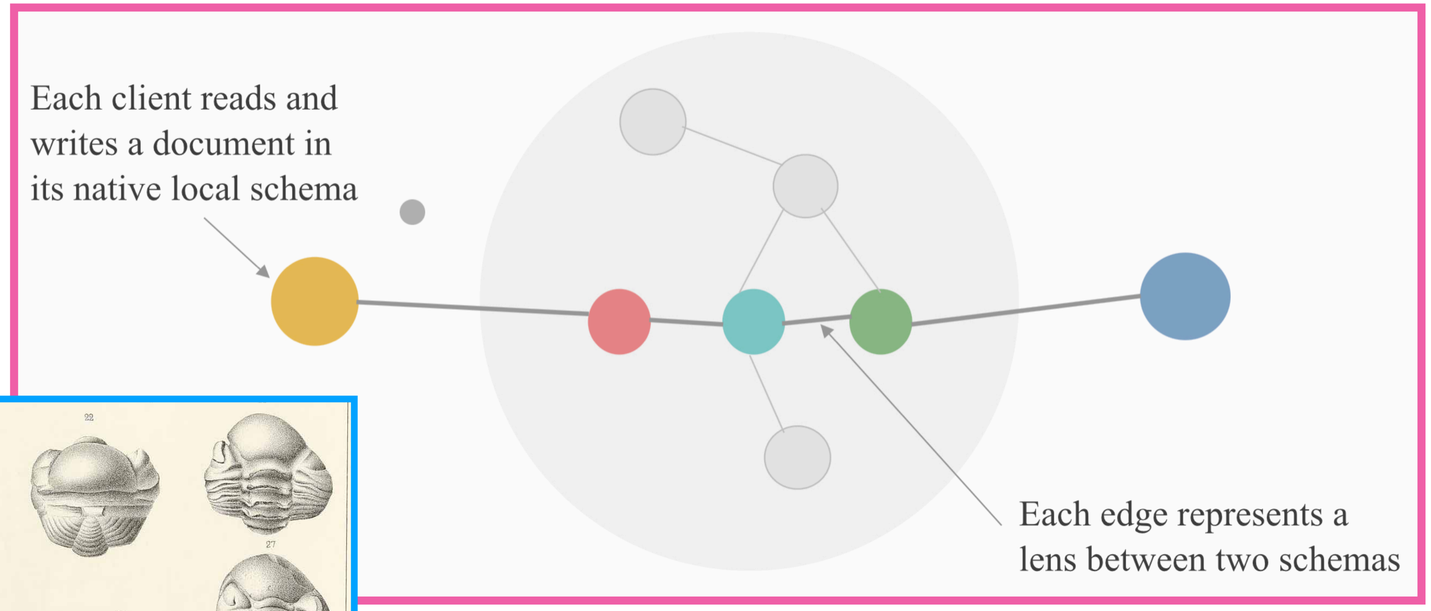
Core Evolution

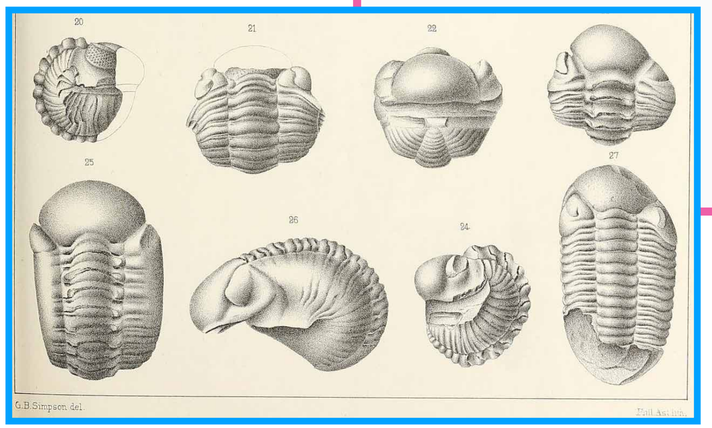
We need to evolve our approach:
focus on domain and structure!



Big Idea

Structural Example: Schema Lenses





www.inkandswitch.com/cambria.html

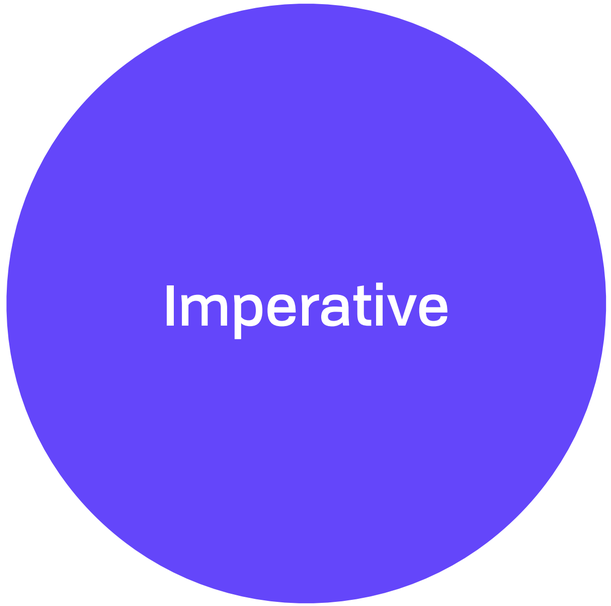
In the Large

In the Large

Code You Used to Write

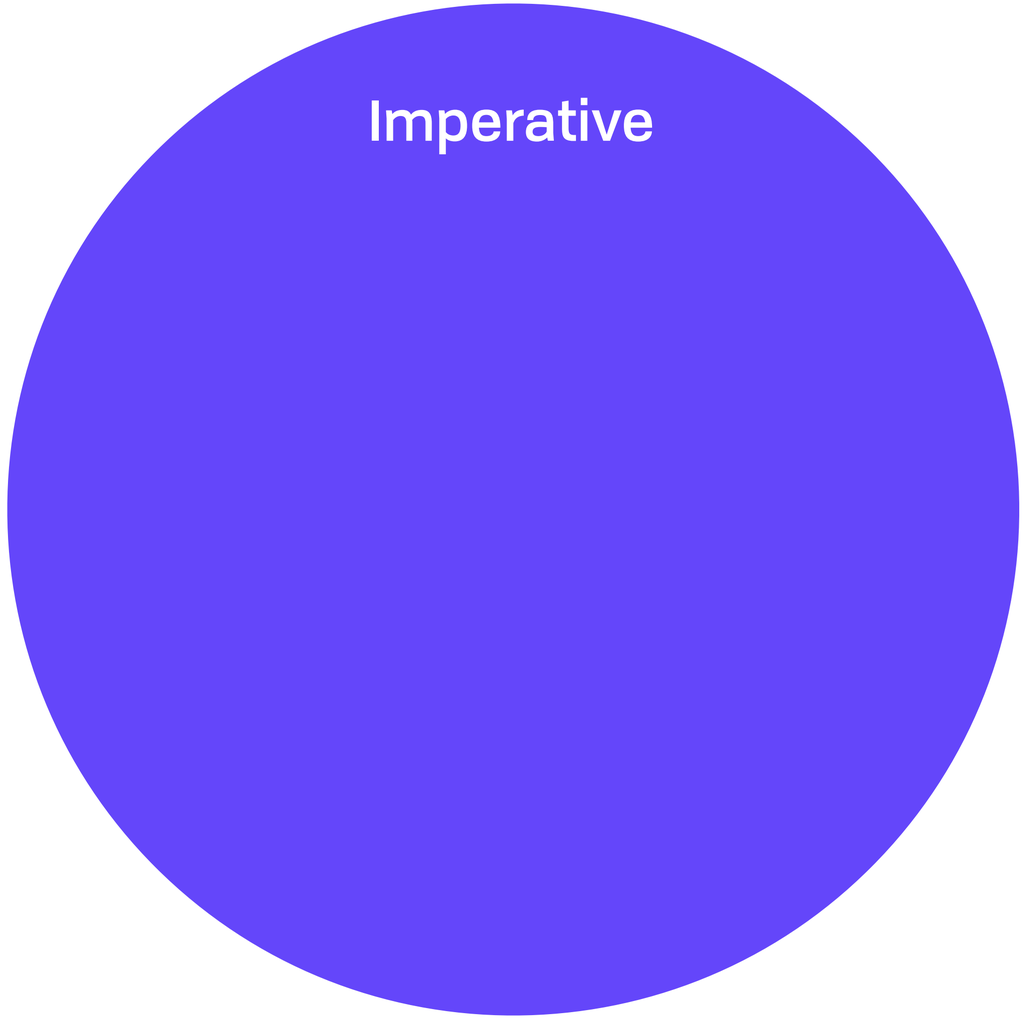
In the Large

Code You Used to Write



In the Large

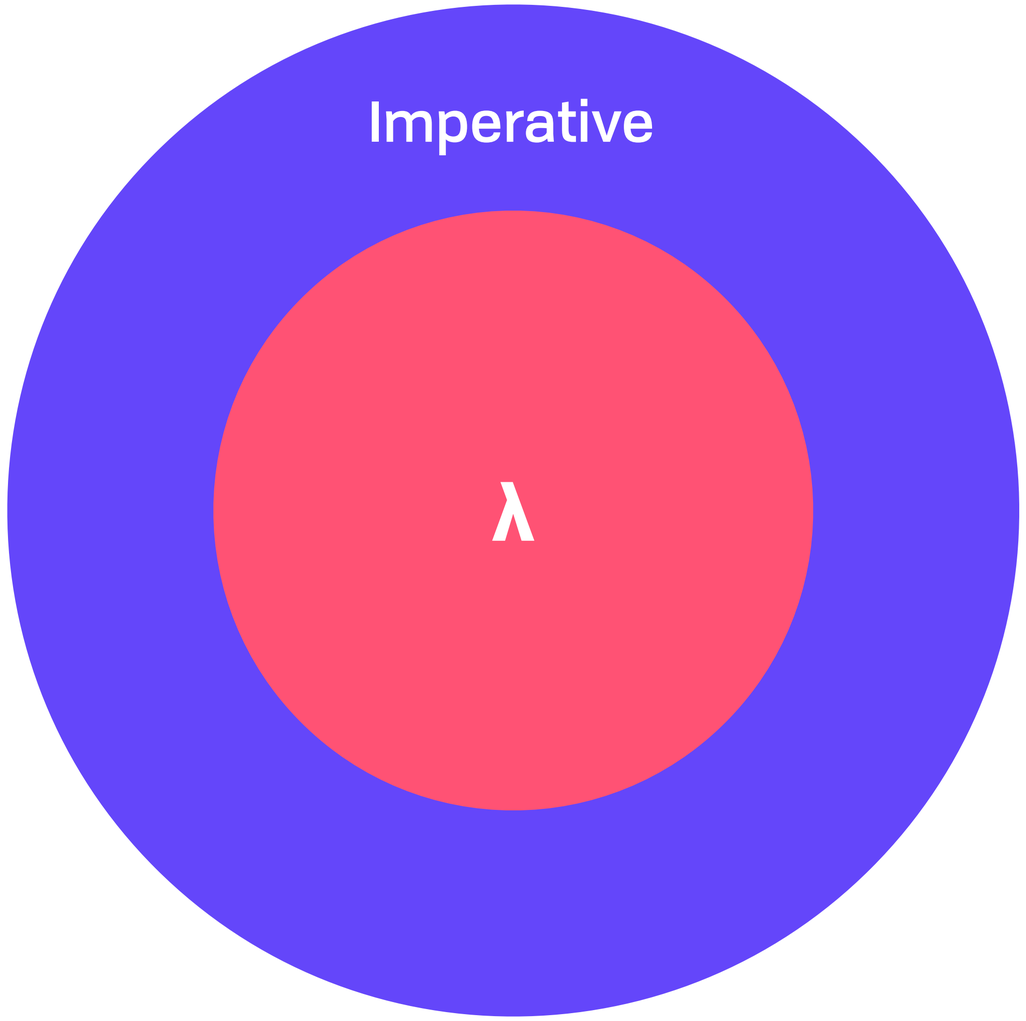
"Good" Elixir



* Functional core, imperative shell

In the Large

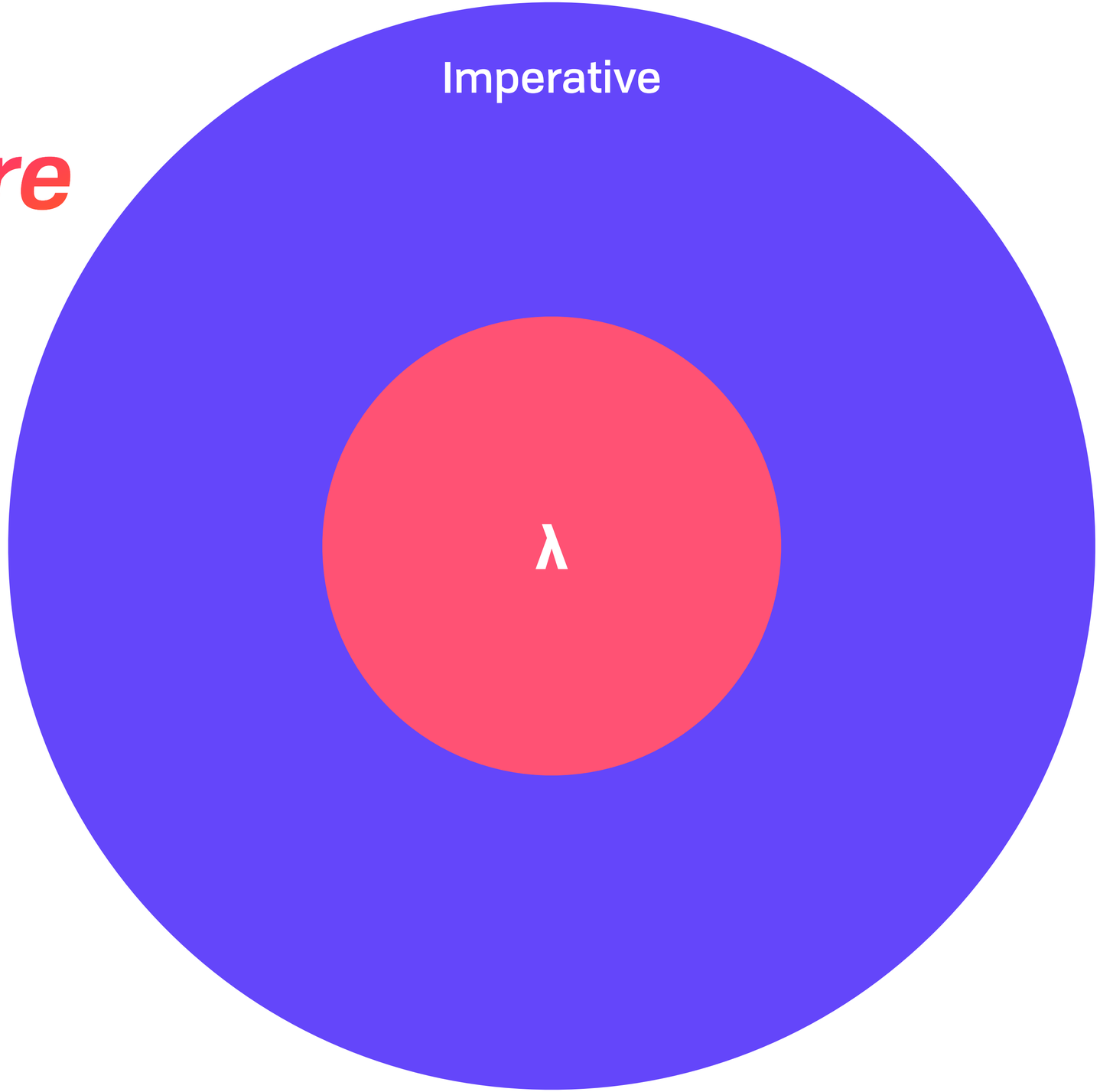
"Good" Elixir



* Functional core, imperative shell

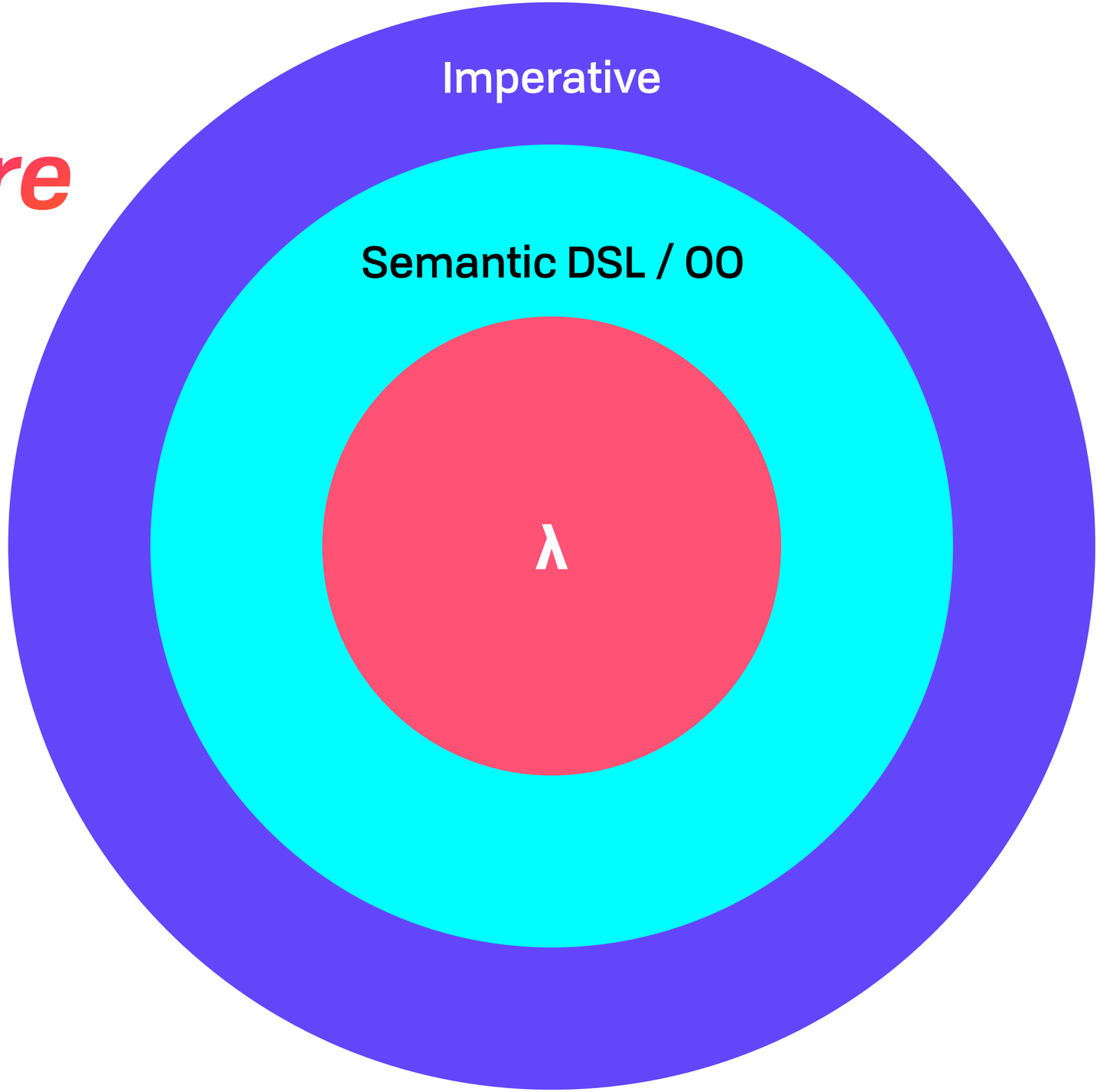
In the Large

3LA Future



In the Large

3LA Future



In the Large

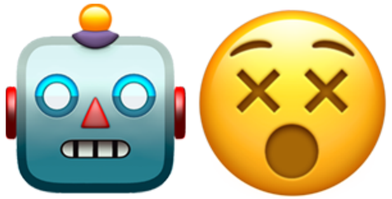
Prop & Model Test

```
defmodule ListTest do
  use ExUnit.Case, async: true
  use ExUnitProperties

  property "++ is associative" do
    check all list_a <- list_of(term()),
              list_b <- list_of(term()),
              list_c <- list_of(term()) do

      ab_c = (list_a ++ list_b) ++ list_c
      a_bc = list_a  ++ (list_b ++ list_c)
      assert ab_c == a_bc
    end
  end
end
```

GOTO Considered Harmful



The quality of programmers is a *decreasing function* of the density of GOTO statements in the programs they produce

Edsger Dijkstra

GOTO Considered Harmful

What's So Bad About Having Control? 🦶🔫

GOTO Considered Harmful

What's So Bad About Having Control? 🦶🔫

- GOTOs

GOTO Considered Harmful

What's So Bad About Having Control? 🦶🔫

- GOTOs
- Low level instruction

GOTO Considered Harmful

What's So Bad About Having Control? 🦶🔫

- GOTOs
- Low level instruction
- Literally how the machine is going to see it

What's So Bad About Having Control? 🦶🔫

- GOTOs
- Low level instruction
- Literally how the machine is going to see it
- Extremely flexible

What's So Bad About Having Control? 🦶🔫

- GOTOs
- Low level instruction
- Literally how the machine is going to see it
- Extremely flexible
- Highly concrete

GOTO Considered Harmful

What's So Bad About Having Control? 🦶🔫

- GOTOs
- Low level instruction
- Literally how the machine is going to see it
- Extremely flexible
- Highly concrete
- Huge number of implicit states

GOTO Considered Harmful

What's So Bad About Having Control? 🦶🔫

- GOTOs
- Low level instruction
- Literally how the machine is going to see it
- Extremely flexible
- Highly concrete
- Huge number of implicit states

Line 1

Line 2

Line 3

Line 4

Line 5 — GOTO

Line 6

GOTO Considered Harmful

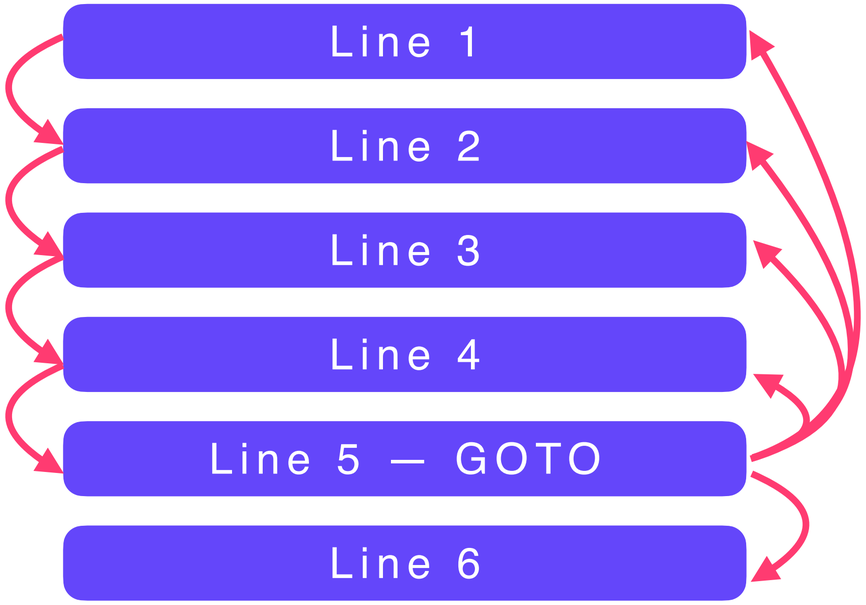
What's So Bad About Having Control? 🦶🔫

- GOTOs
- Low level instruction
- Literally how the machine is going to see it
- Extremely flexible
- Highly concrete
- Huge number of implicit states

GOTO Considered Harmful

What's So Bad About Having Control? 🦶🔫

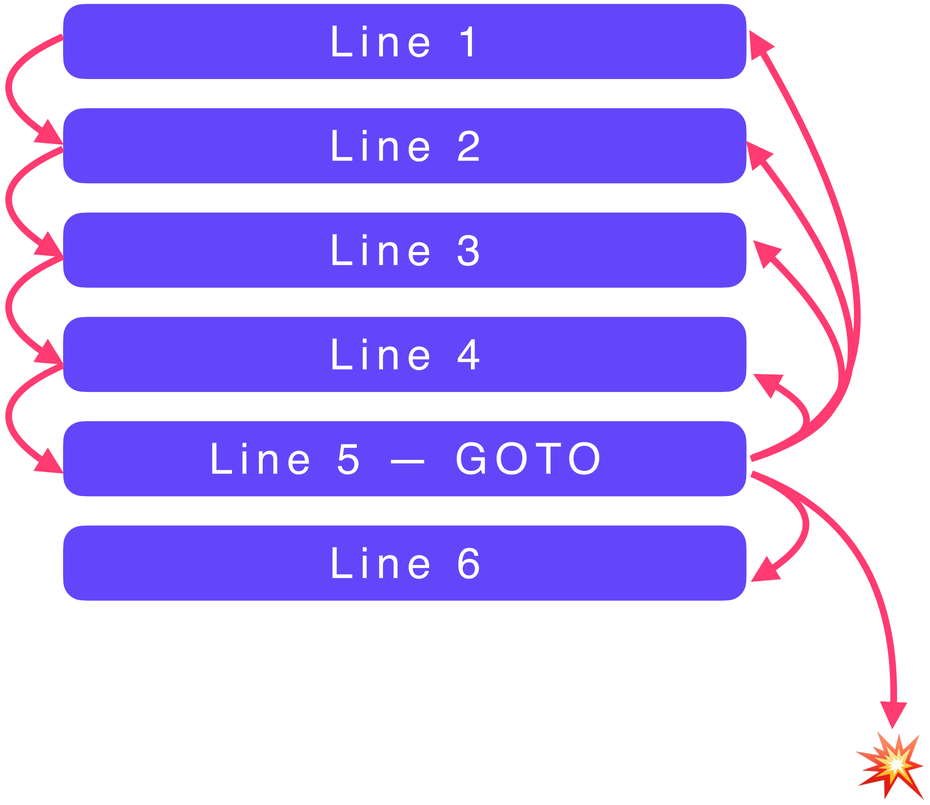
- GOTOs
- Low level instruction
- Literally how the machine is going to see it
- Extremely flexible
- Highly concrete
- Huge number of implicit states



GOTO Considered Harmful

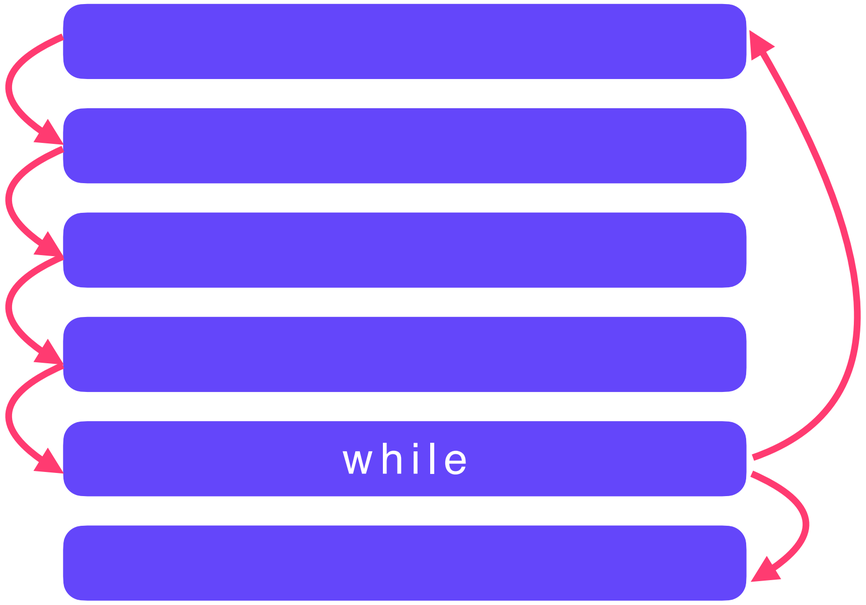
What's So Bad About Having Control? 🦶🔫

- GOTOs
- Low level instruction
- Literally how the machine is going to see it
- Extremely flexible
- Highly concrete
- Huge number of implicit states



GOTO Considered Harmful

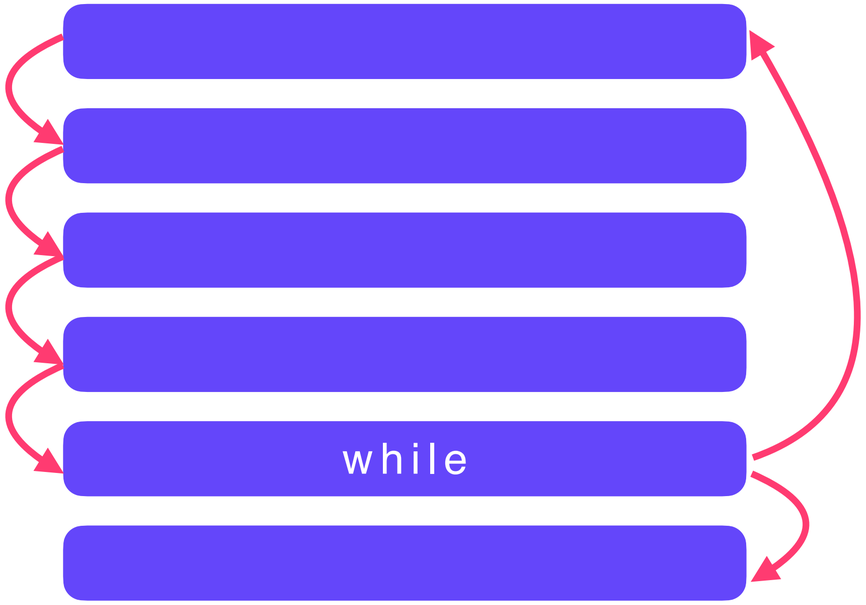
Structured Programming



GOTO Considered Harmful

Structured Programming

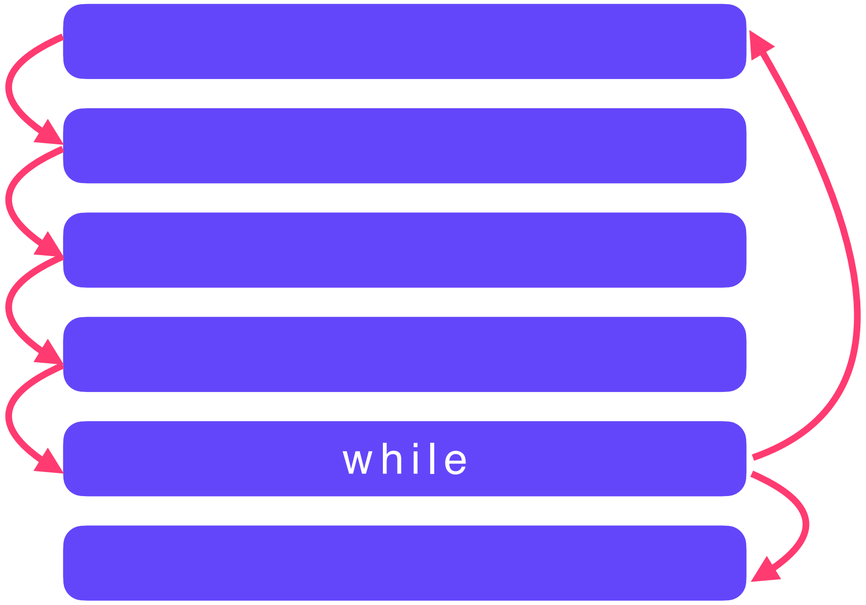
- Subroutines



GOTO Considered Harmful

Structured Programming

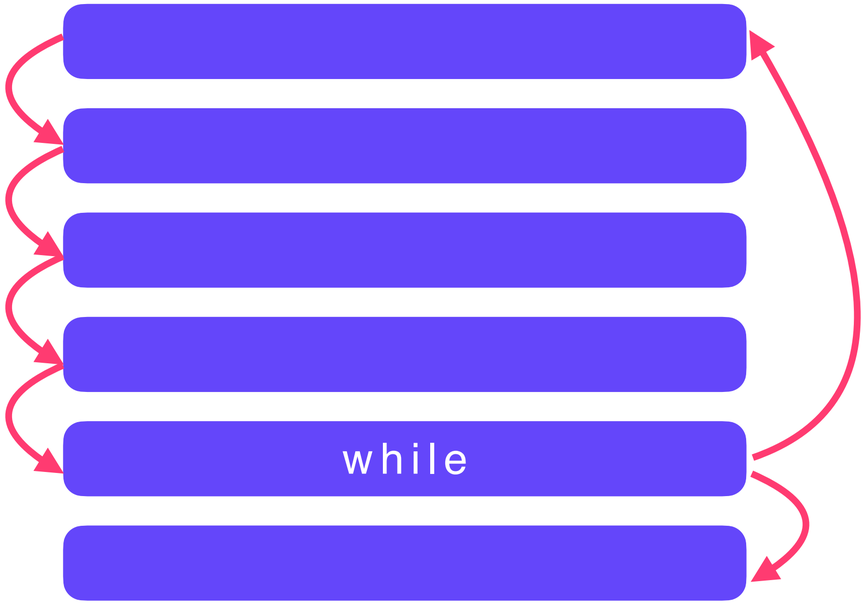
- Subroutines
- Loops



GOTO Considered Harmful

Structured Programming

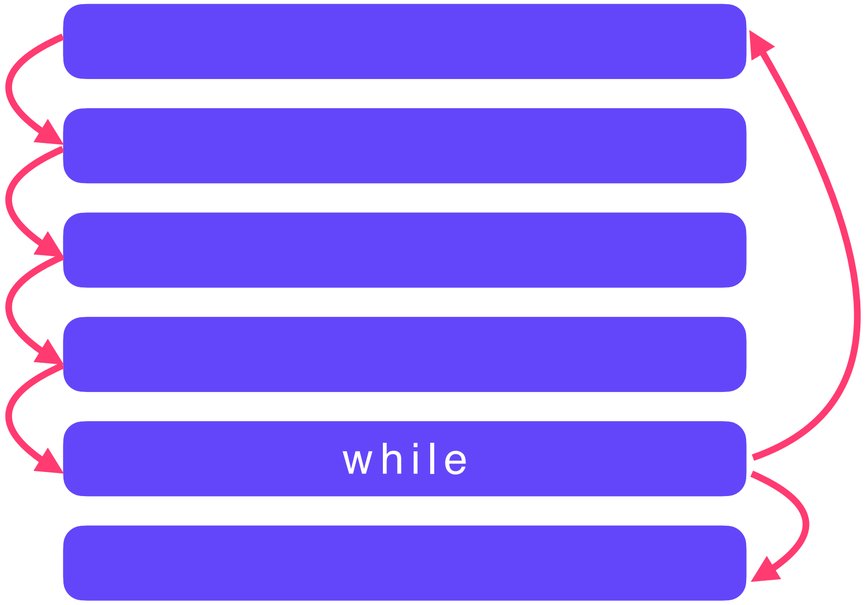
- Subroutines
- Loops
- Switch/branching



GOTO Considered Harmful

Structured Programming

- Subroutines
- Loops
- Switch/branching
- Named routines



GOTO Considered Harmful

The Next Generation 🚀

GOTO Considered Harmful

The Next Generation 🚀

- Objects, Actors, Protocols

GOTO Considered Harmful

The Next Generation 🚀

- Objects, Actors, Protocols
- Map, Reduce, Filter

GOTO Considered Harmful

The Next Generation 🚀

- Objects, Actors, Protocols
- Map, Reduce, Filter
- Functor, Applicative, Monad

GOTO Considered Harmful

The Next Generation 🚀

- Objects, Actors, Protocols
- Map, Reduce, Filter
- Functor, Applicative, Monad
- Constraint Solvers

GOTO Considered Harmful

Tradeoffs

GOTO Considered Harmful

Tradeoffs

- Exchange granular control for structure

GOTO Considered Harmful

Tradeoffs

- Exchange granular control for structure
- **Meaning over mechanics**

GOTO Considered Harmful

Tradeoffs

- Exchange granular control for structure
- **Meaning over mechanics**
- More human than machine

GOTO Considered Harmful

Tradeoffs

- Exchange granular control for structure
- **Meaning over mechanics**
- More human than machine
- *Safer!*

GOTO Considered Harmful

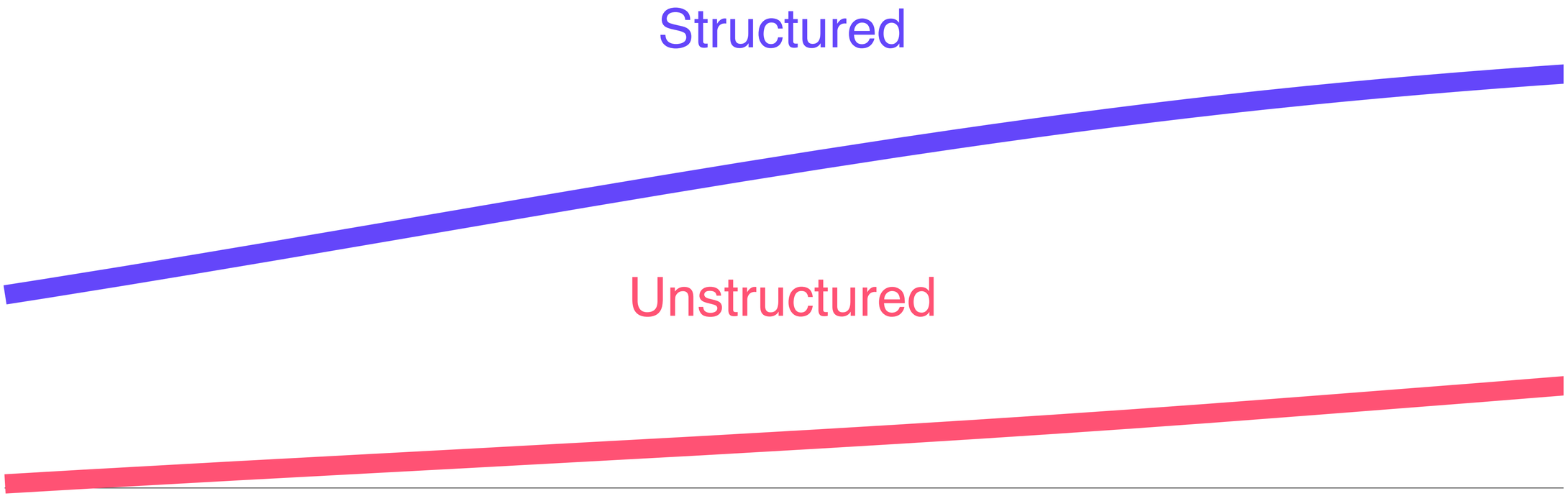
Tradeoffs

- Exchange granular control for structure
- **Meaning over mechanics**
- More human than machine
- *Safer!*

- Spectrum
  - Turing Tarpit
  - Church Chasm
  - Haskell Fan Fiction

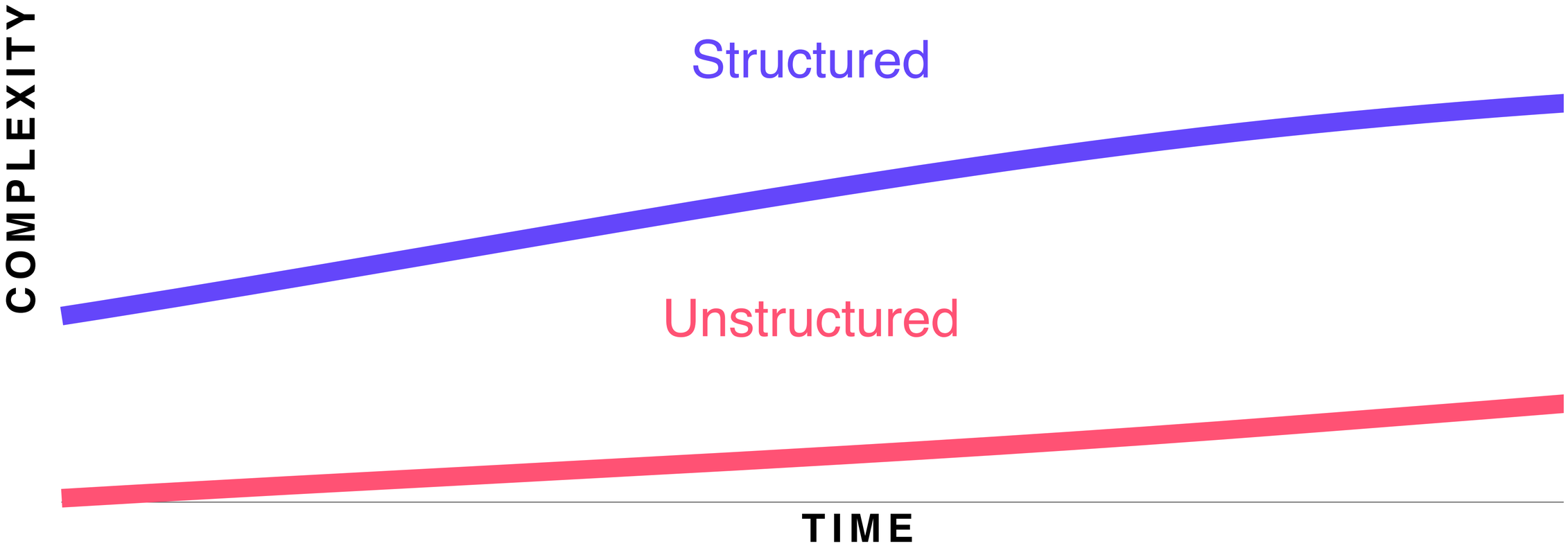
GOTO Considered Harmful

Payoff



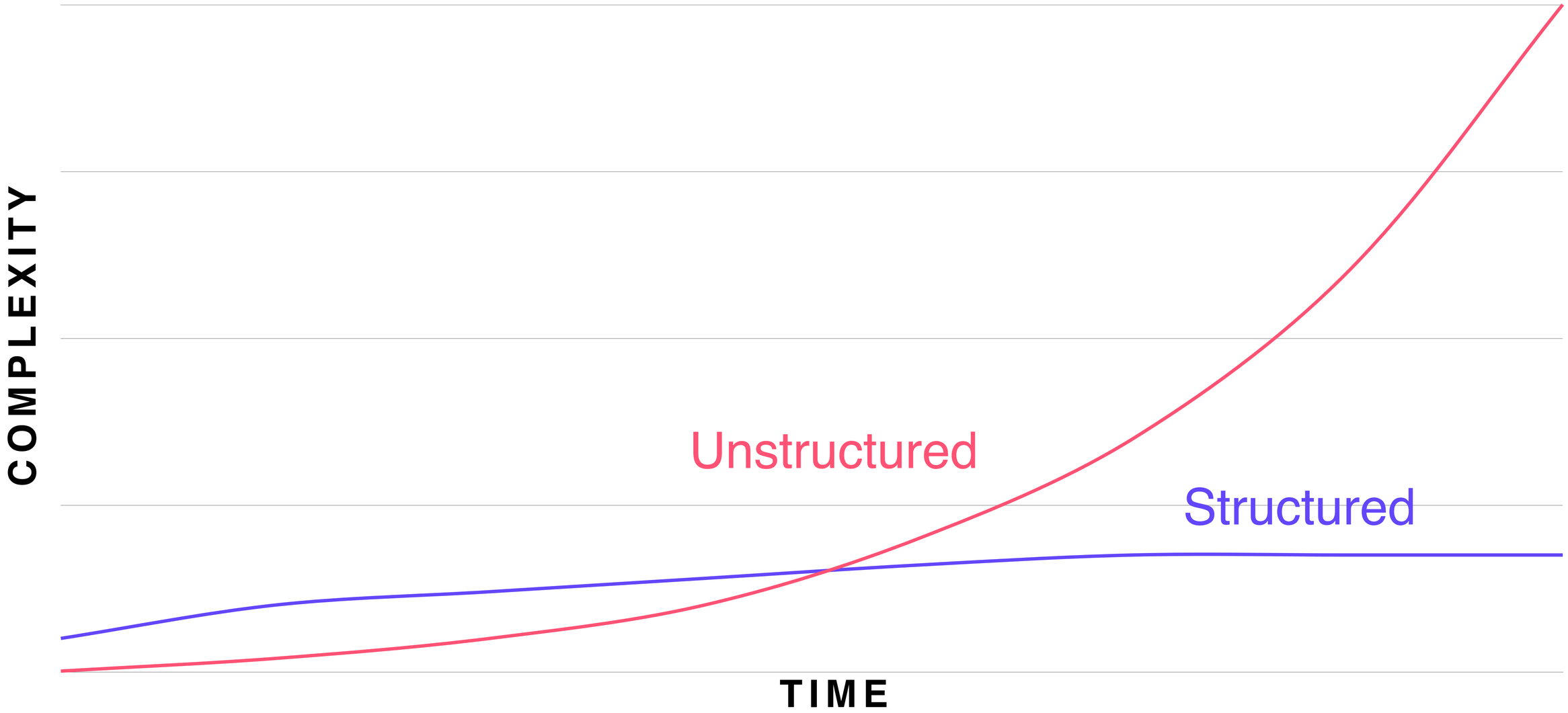
GOTO Considered Harmful

Payoff



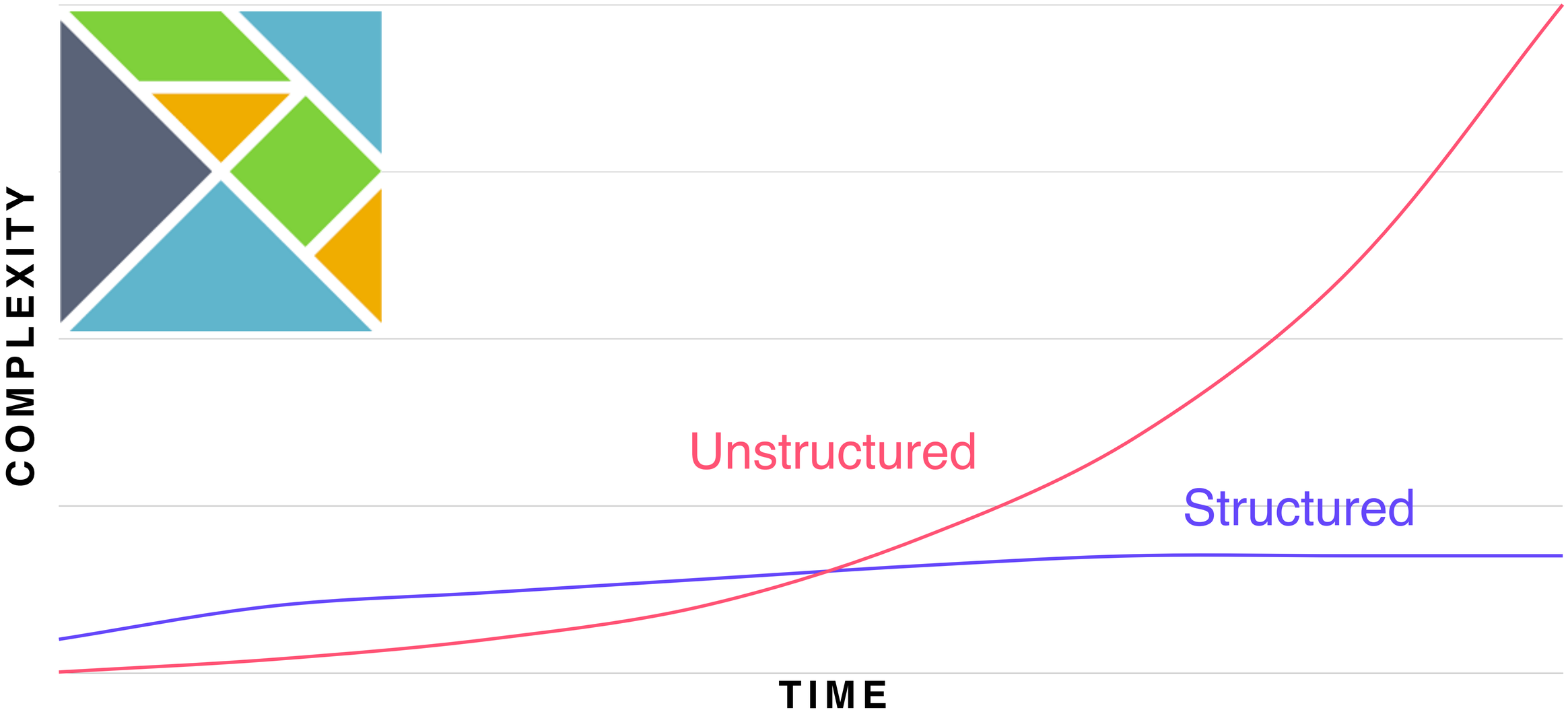
GOTO Considered Harmful

Payoff



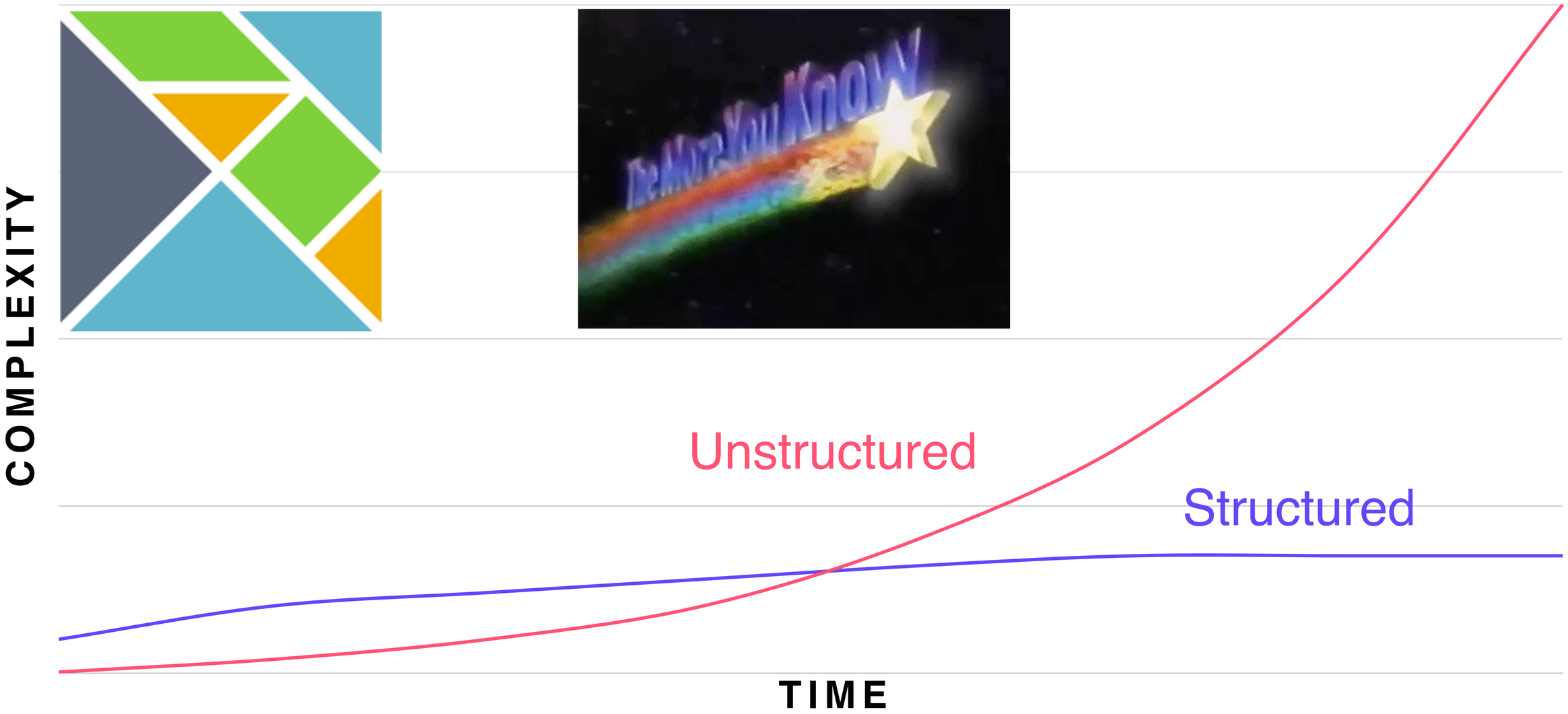
GOTO Considered Harmful

Payoff



GOTO Considered Harmful

Payoff

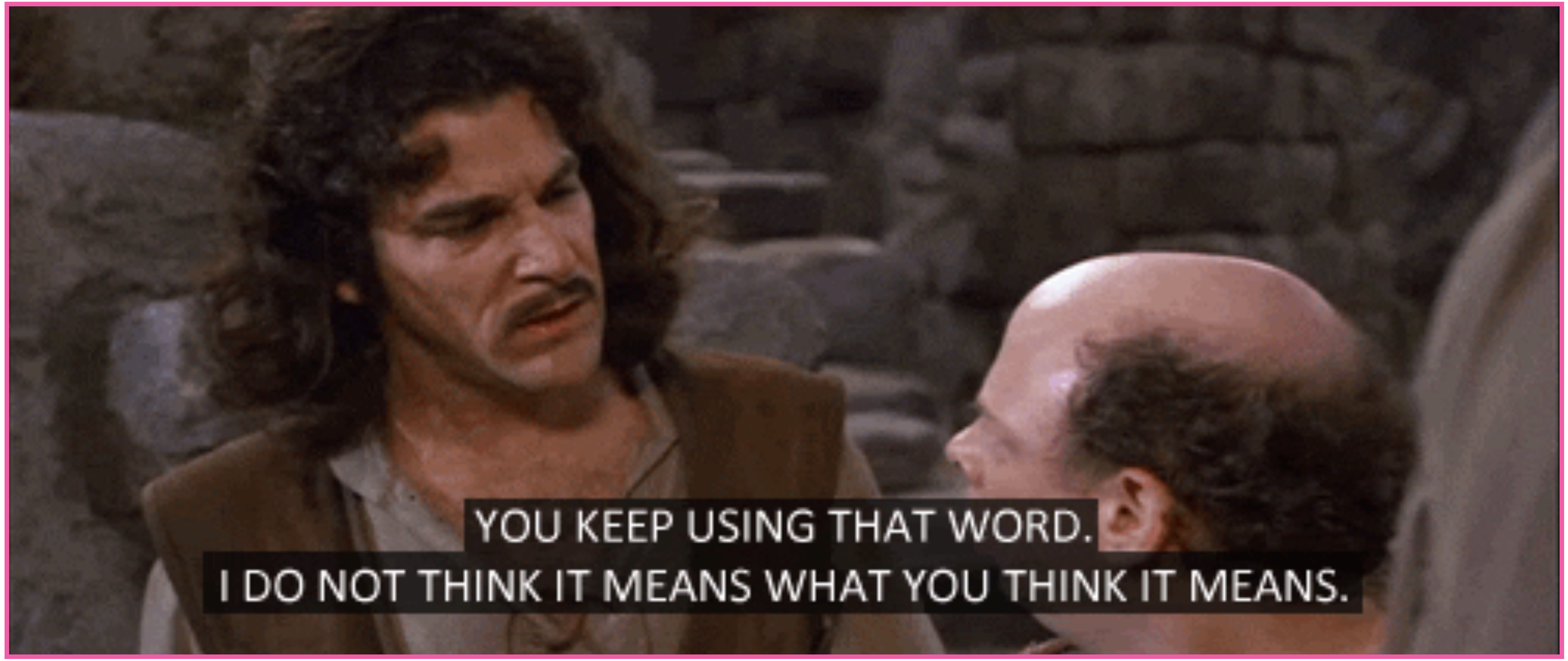


On Complexity

🌱🌸🌾

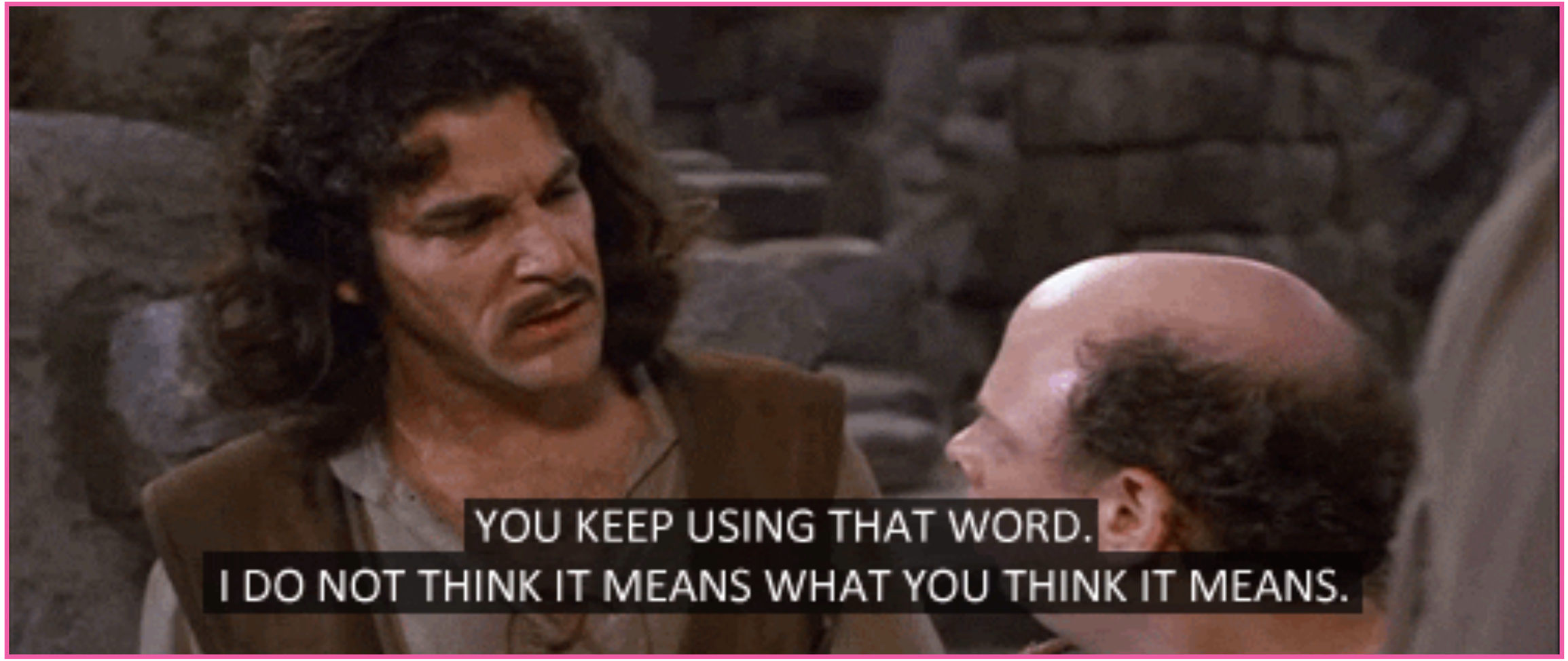
On Complexity

Overused



On Complexity

Overused



On Complexity

The Bad Kind ☠️

On Complexity

The Bad Kind ☠️

- Probably pretty familiar with this

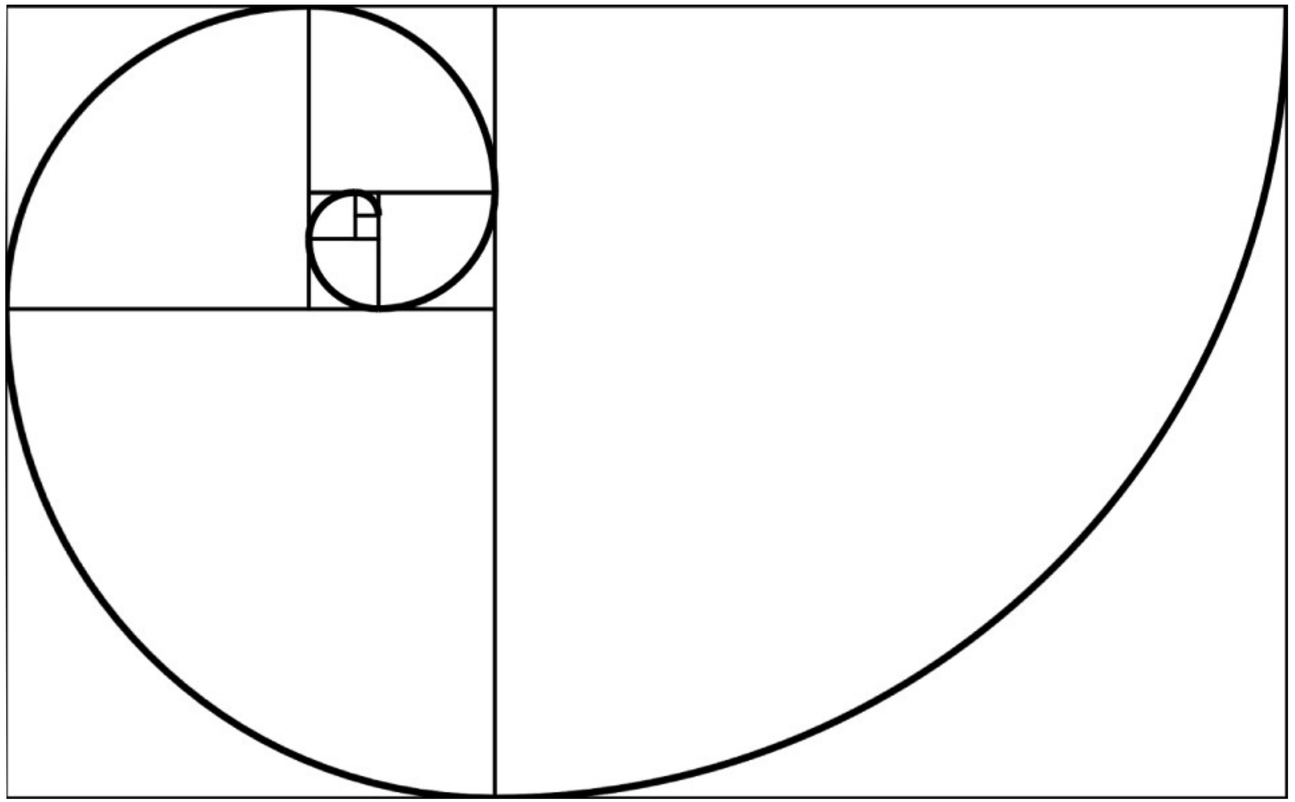
On Complexity

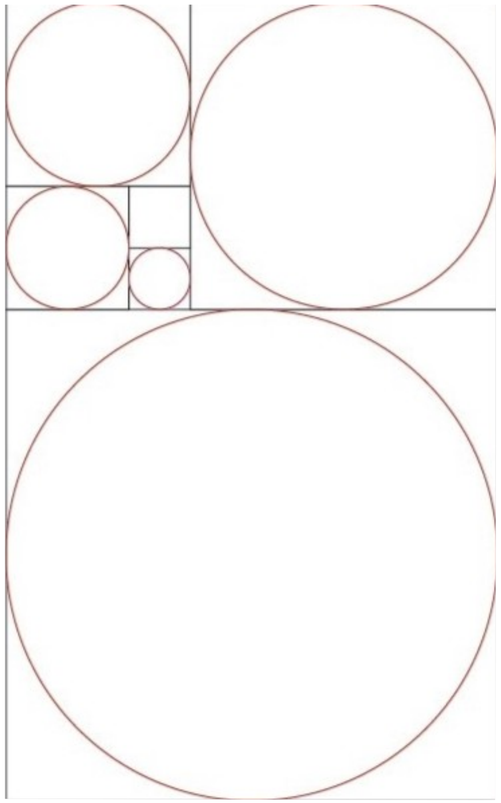
The Bad Kind ☠️

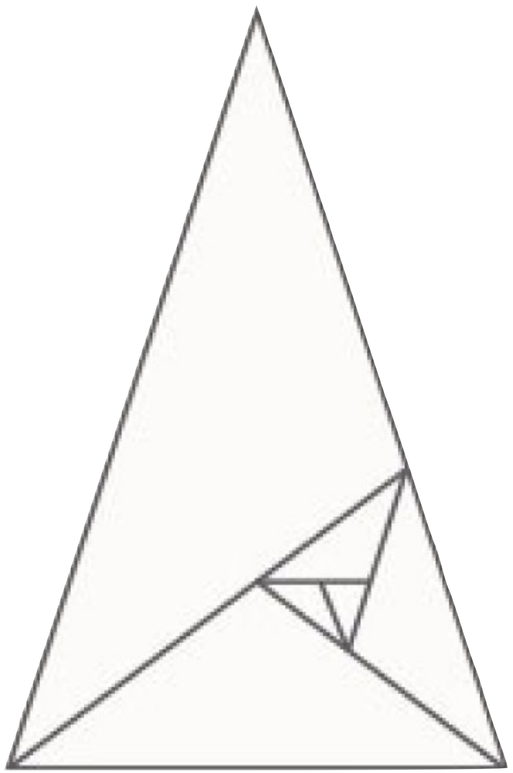
- Probably pretty familiar with this
- Euphemism for:
  - Complicated
  - Inconsistent
  - No plan
  - “Unstructured mess”

On Complexity

The Good Kind: Deep







What do these have in common?

$(a+b)/a \sim a / b$

On Complexity

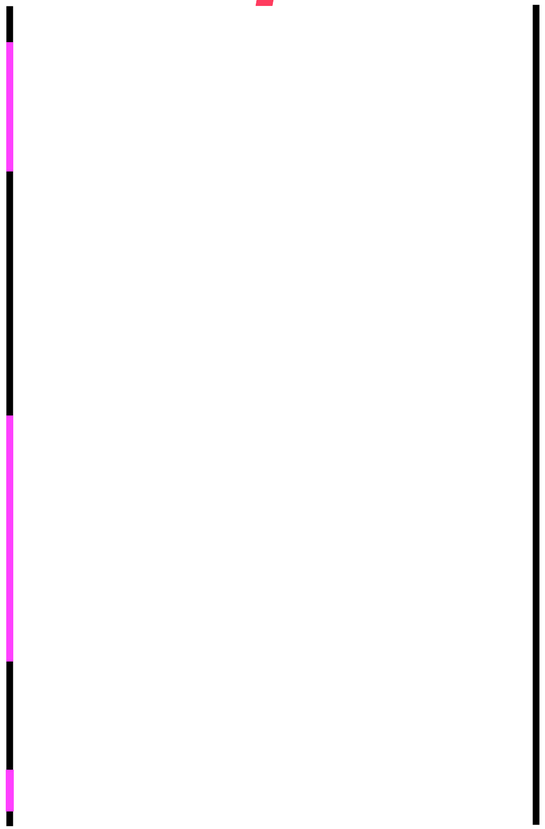
Orthogonal Complecting

On Complexity

Orthogonal Complecting

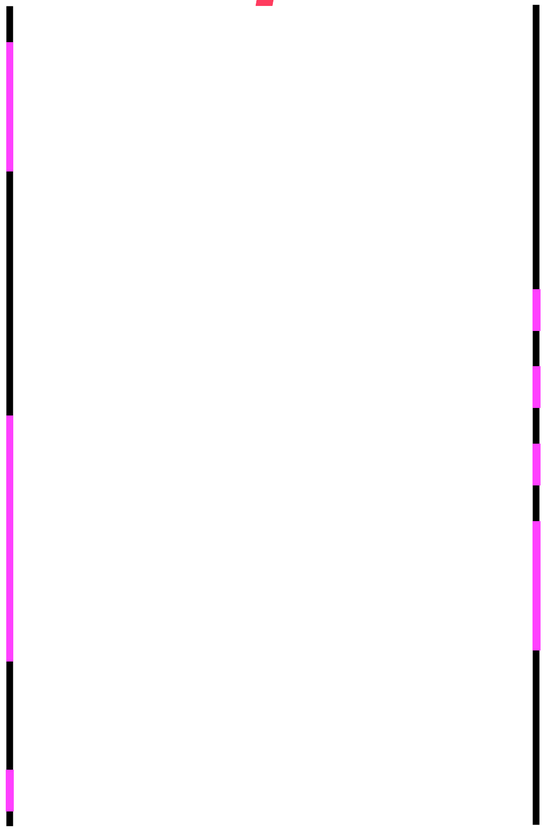
On Complexity

Orthogonal Complecting



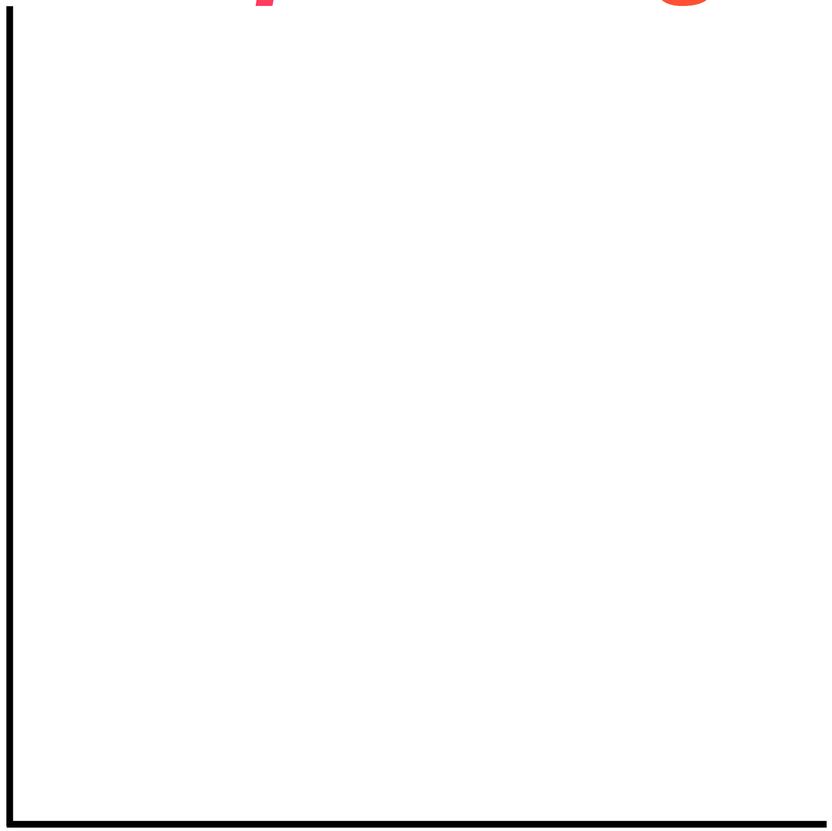
On Complexity

Orthogonal Complecting



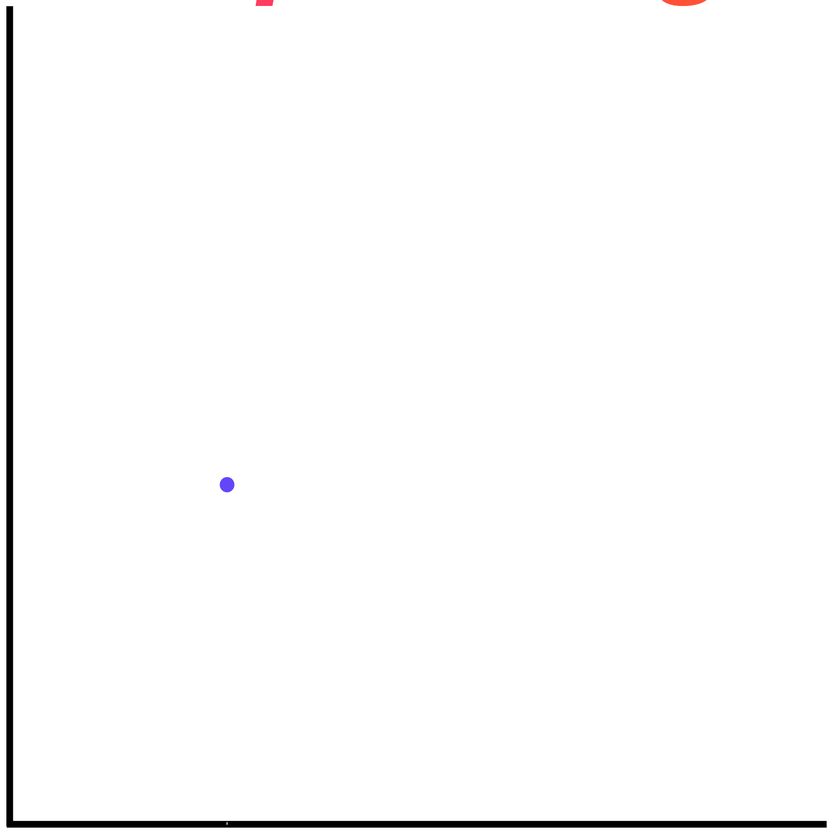
On Complexity

Orthogonal Complecting



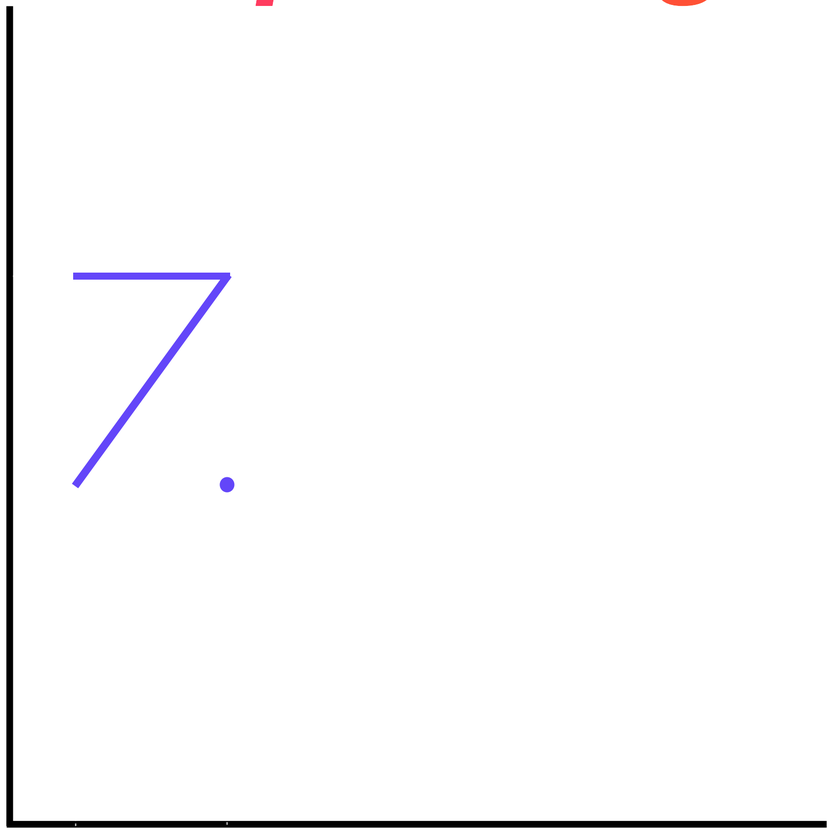
On Complexity

Orthogonal Complecting



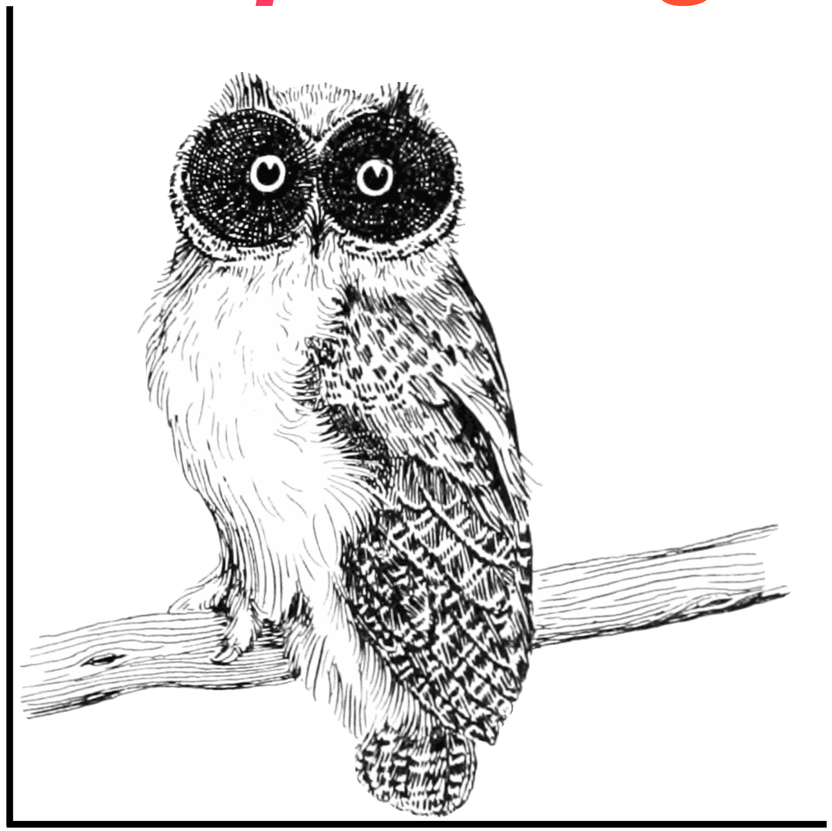
On Complexity

Orthogonal Complecting



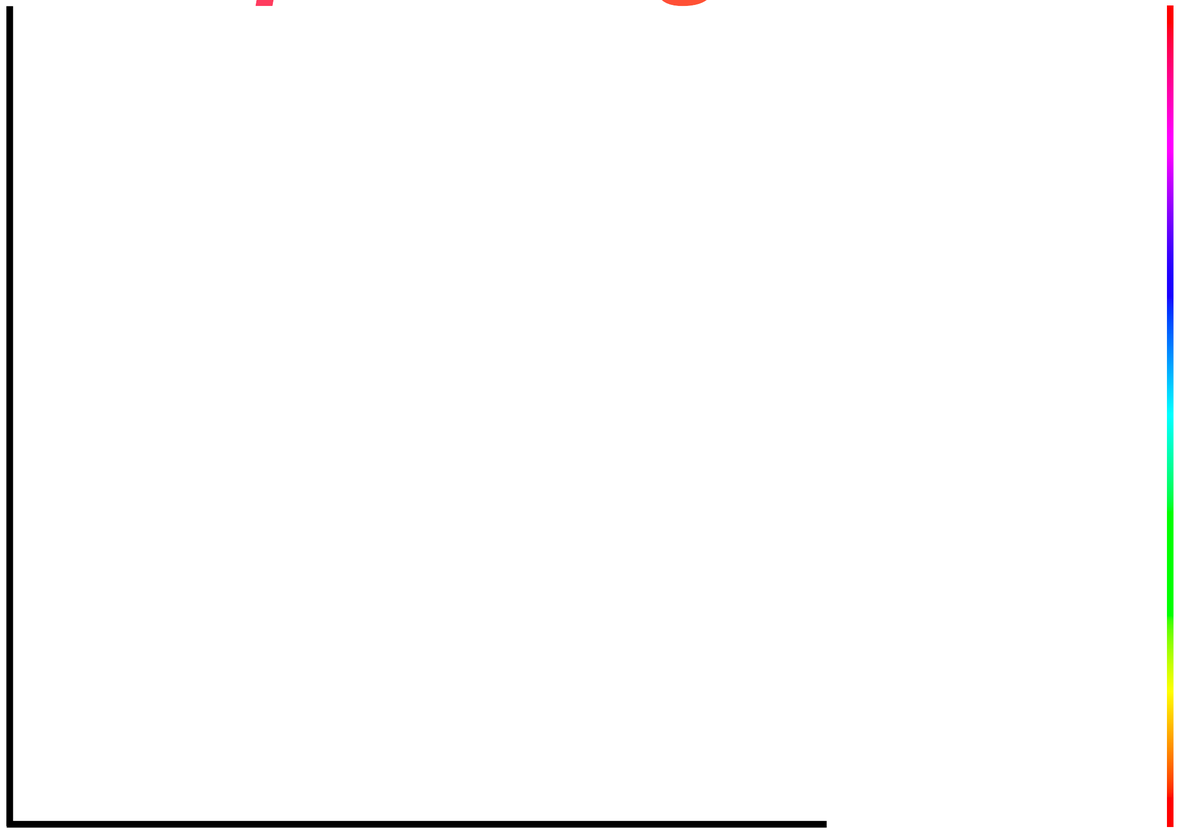
On Complexity

Orthogonal Complecting



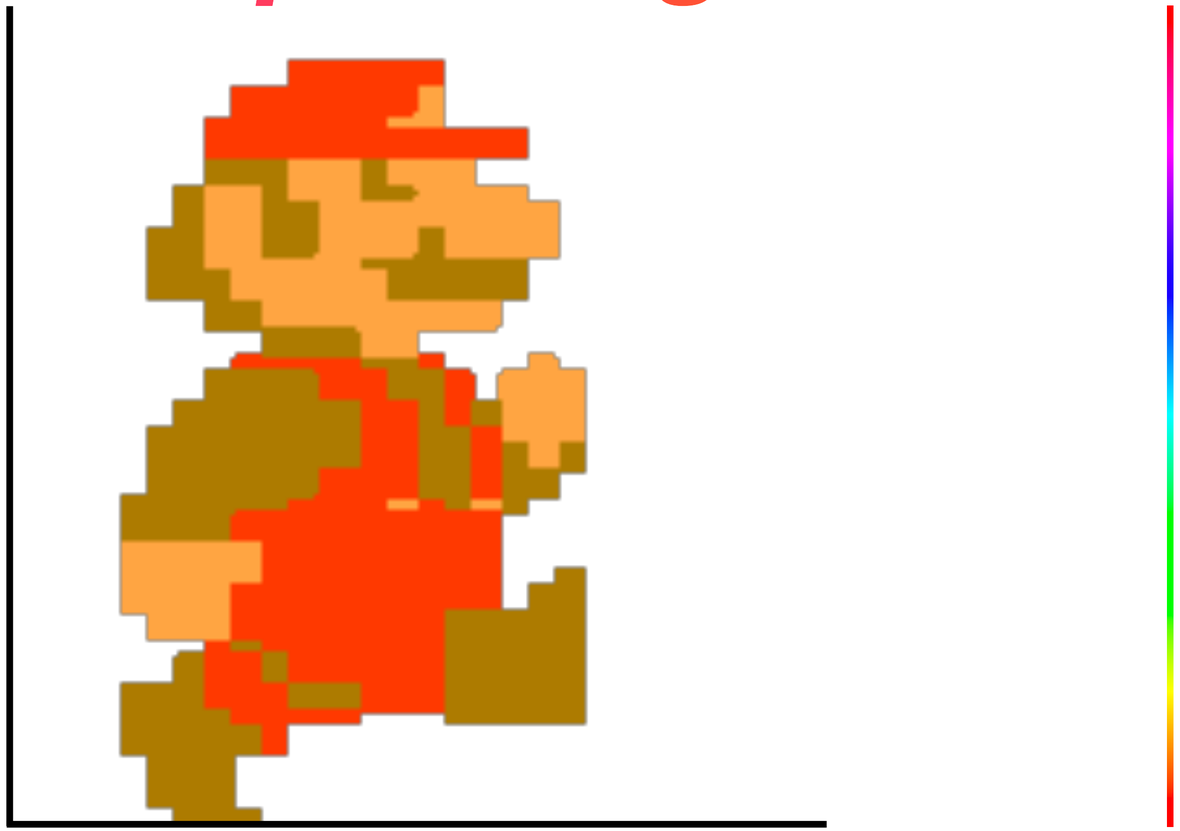
On Complexity

Orthogonal Complecting



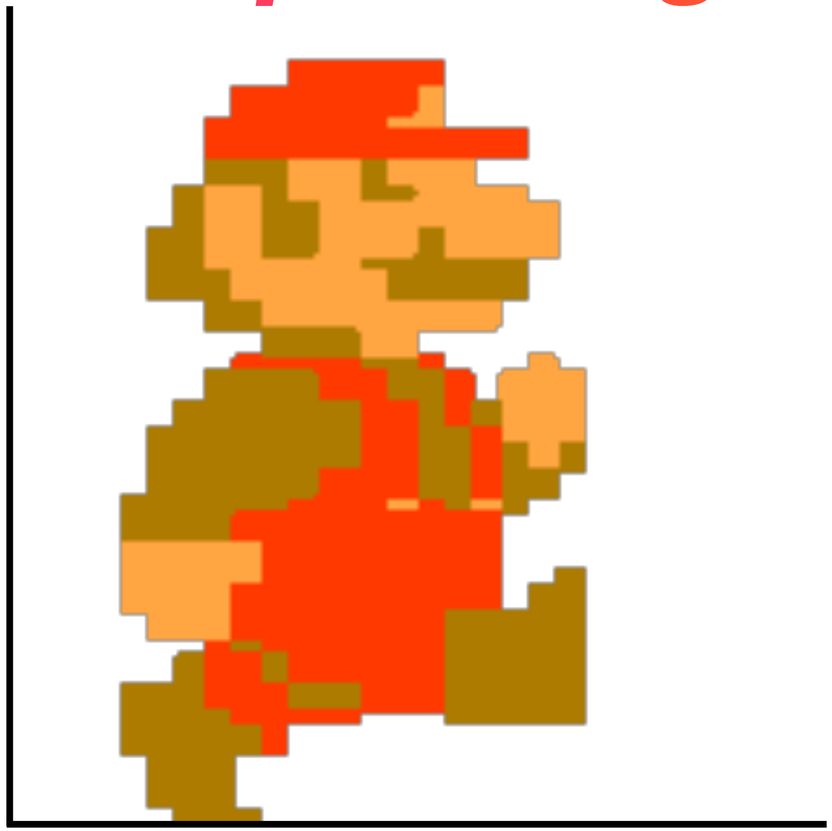
On Complexity

Orthogonal Complecting



On Complexity

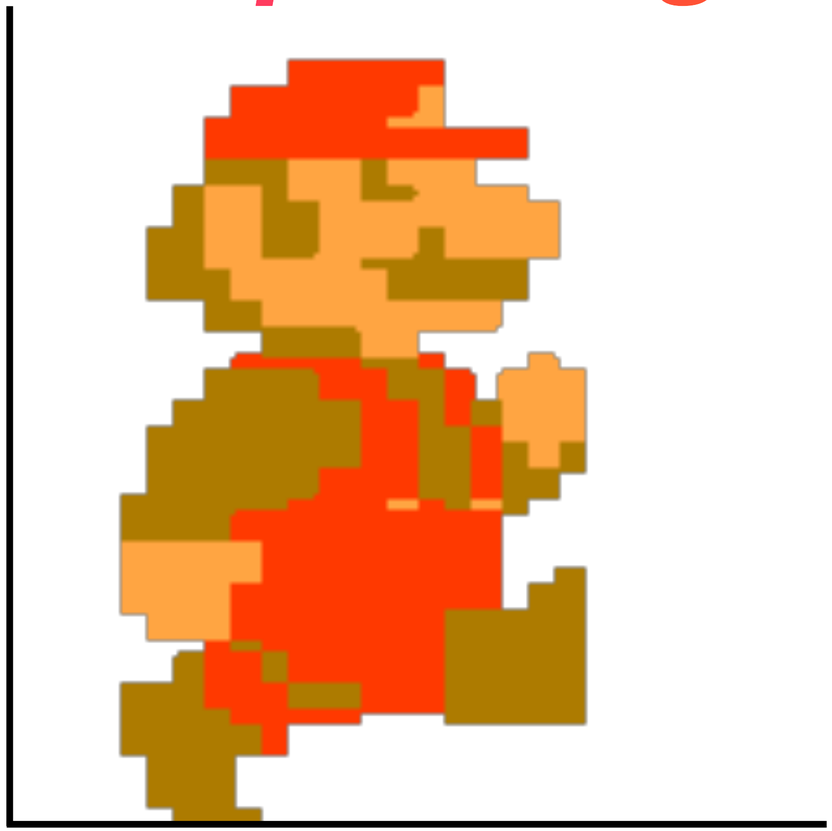
Orthogonal Complecting



🕐

On Complexity

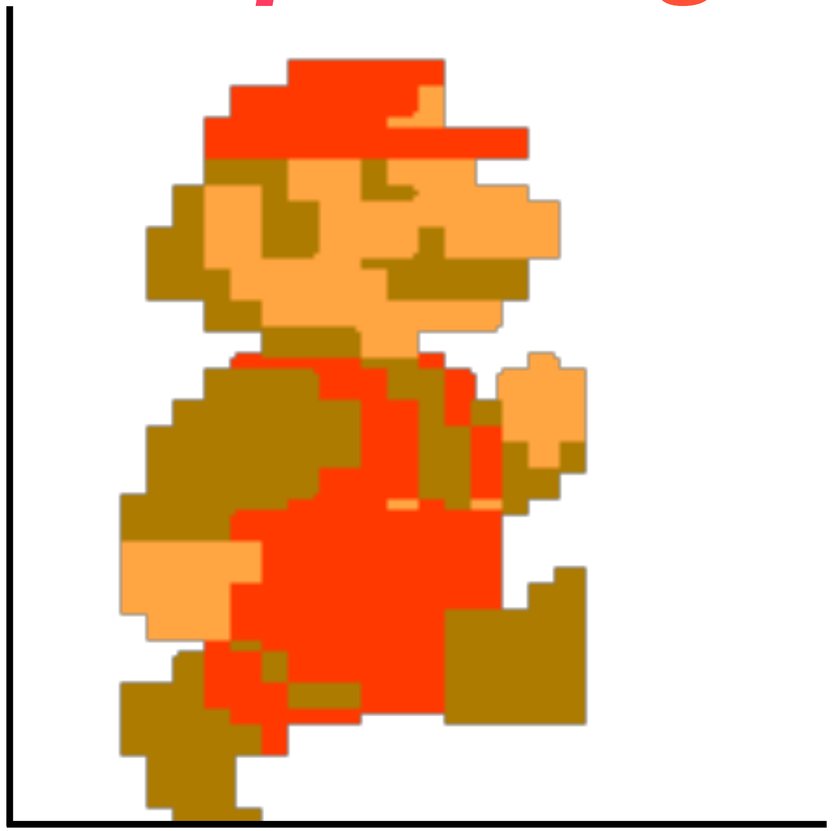
Orthogonal Complecting



Structures: 4

On Complexity

Orthogonal Complecting



Structures: 4

Results: effectively limitless

On Complexity

Complex != Complicated

On Complexity

Complex != Complicated

- **Complex** — interconnected parts

On Complexity

Complex != Complicated

- **Complex** — interconnected parts
- **Complicated** — difficult to understand

Abstraction & DSLs

The Power of Words

The Power of Words

- Restrict your vocabulary to your domain
  - ...the hard part is deciding on that vocabulary

The Power of Words

- Restrict your vocabulary to your domain
  - ...the hard part is deciding on that vocabulary
- Technical debt is lack of understanding
  - https://daverupert.com/2020/11/technical-debt-as-a-lack-of-understanding/

On Complexity

The Actor Abyss

On Complexity

The Actor Abyss

- Each step is very simple

On Complexity

The Actor Abyss

- Each step is very simple
- Reasoning about dynamic organisms is hard
  - Remember to (re)store your data
    - e.g. crash recovery
  - Called collaborator may not be there

On Complexity

The Actor Abyss

- Each step is very simple
- Reasoning about dynamic organisms is hard
  - Remember to (re)store your data
    - e.g. crash recovery
  - Called collaborator may not be there
- Complexity grows faster than linear

On Complexity

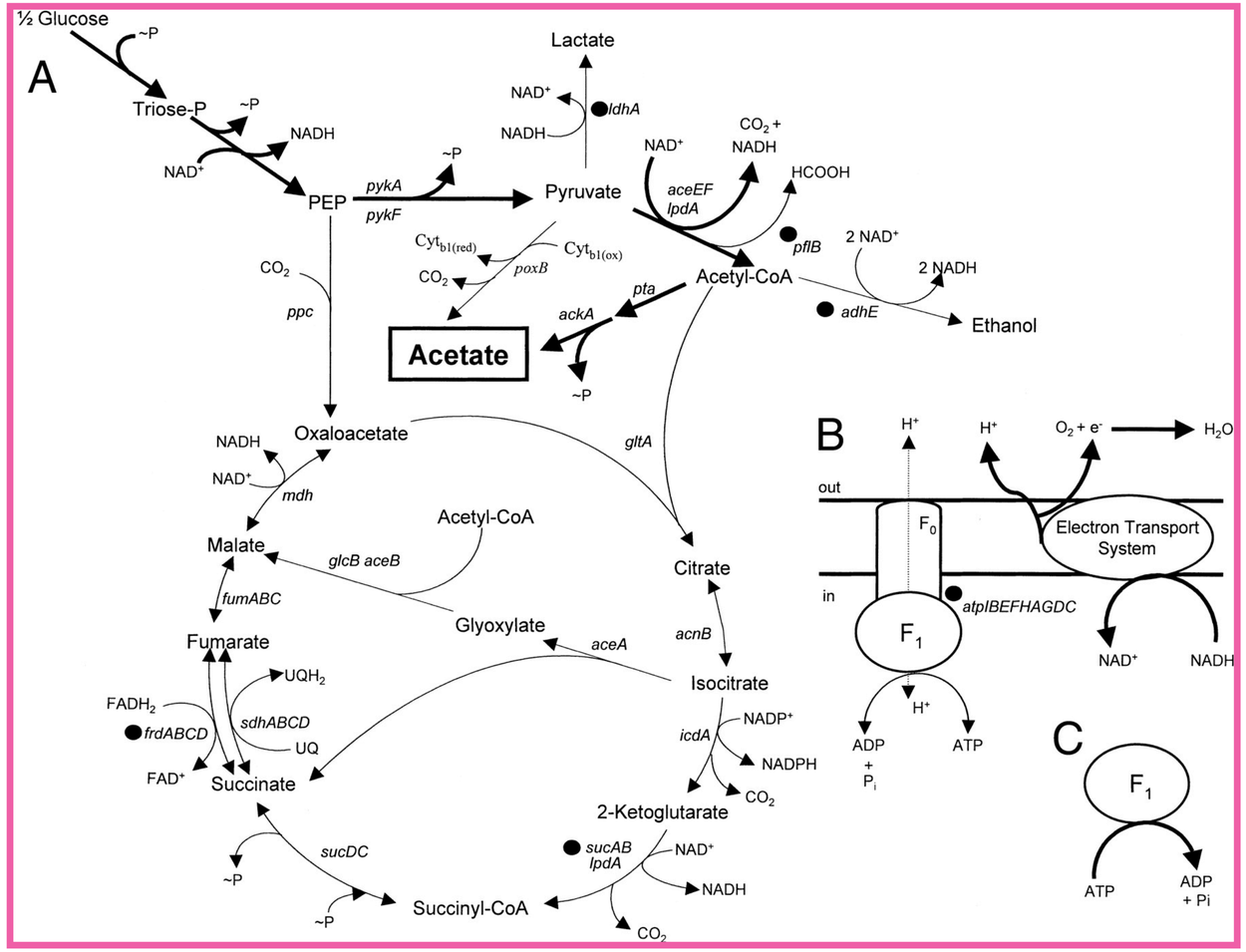
The Actor Abyss

- Each step is very simple
- Reasoning about dynamic organisms is hard
  - Remember to (re)store your data
    - e.g. crash recovery
  - Called collaborator may not be there
- Complexity grows faster than linear
- Find common factors — your abstraction

On Complexity

The Actor Abyss

- Each step is very simple
- Reasoning about dynamic organisms is hard
  - Remember to (re)store your data
    - e.g. crash recovery
  - Called collaborator may not be there
- Complexity grows faster than linear
- Find common factors — your abstraction



Fighting GenSoup

🚫🍲⚔️

Fighting GenSoup

Good Interfaces != Good Abstractions

Fighting GenSoup

Good Interfaces != Good Abstractions

- `GenServer` & co are actually pretty low level
  - Please add some semantics 🙏

Fighting GenSoup

Good Interfaces != Good Abstractions

- `GenServer` & co are actually pretty low level
  - Please add some semantics 🙏
- Don't reinvent the wheel every time 🎡

Good Interfaces != Good Abstractions

- `GenServer` & co are actually pretty low level
  - Please add some semantics 🙏
- Don't reinvent the wheel every time 🎡
- Let's look at a very common example

Fighting GenSoup

Abstraction

```
defprotocol KeyValue do
  def init(proxy)
  def get(db, value)
  def set(db, key, value)
end
```

Fighting GenSoup

Simple Case: Map

```
defimpl KeyValue, for: Map do
  def init(_), do: %{}
  def get(db, value), do: Map.get(db, value, :not_found)
  def set(db, key, value), do: Map.put(db, key, value)
end
```

Fighting GenSoup

Async Case: Part I (defstruct)

```
defmodule ProcDB do
  use Agent

  defstruct [:pid]

  # Works with any inner data type!
  def start_link(starter), do: Agent.start_link(fn -> starter end)

  def get(pid, key) do
    Agent.get(pid, fn state -> KeyValue.get(state, key) end)
  end

  def set(pid, key, value) do
    Agent.update(pid, fn state -> KeyValue.set(state, key, value) end)
  end
end
```

Fighting GenSoup

Async Case: Part II (`defimpl`)

```
defimpl KeyValue, for: %ProcDB do
  def init(_) do
    {:ok, pid} = ProcDB.startLink()
    %MyDB{pid: pid}
  end

  def get(%ProcDB{pid: pid}, key), do: ProcDB.get(pid, key)
  def set(%ProcDB{pid: pid}, key, value), do: ProcDB.set(pid, key, value)
end
```

Fighting GenSoup

What Did We Get?

Fighting GenSoup

What Did We Get?

- Common interface

Fighting GenSoup

What Did We Get?

- Common interface
- Encapsulate the detail

Fighting GenSoup

What Did We Get?

- Common interface
- Encapsulate the detail
- Don’t have to think about mechanics anymore 🙅🔧

Fighting GenSoup

Abstraction = Focus & Essence

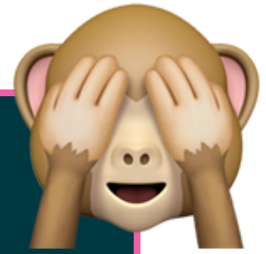
Fighting GenSoup

Abstraction = Focus & Essence

```
defprotocol KeyValue do
  def init(proxy)
  def get(db, value)
  def set(db, key, value)
end
```

Fighting GenSoup

Abstraction = Focus & Essence



```
defprotocol KeyValue do
  def init(proxy)
  def get(db, value)
  def set(db, key, value)
end
```

Abstraction & DSLs

Not Getting Trapped in the Details

Abstraction & DSLs

Commonalities

```
list = [
          "a",
          "b",
          "c"
        ]
```

```
tuples = [
            {0, "a"}
            {1, "b"}
            {2, "c"}
          ]
```

```
map = %{
          0 => "a",
          1 => "b"
          2 => "c"
        }
```

Abstraction & DSLs

Commonalities

- They clearly have a similar structure

```
list = [
          "a",
          "b",
          "c"
        ]
```

```
tuples = [
            {0, "a"}
            {1, "b"}
            {2, "c"}
          ]
```

```
map = %{
          0 => "a",
          1 => "b"
          2 => "c"
        }
```

Abstraction & DSLs

Commonalities

- They clearly have a similar structure
- NOT equally expressive
  - `Enumerable`

```
list = [
          "a",
          "b",
          "c"
        ]
```

```
tuples = [
            {0, "a"}
            {1, "b"}
            {2, "c"}
          ]
```

```
map = %{
          0 => "a",
          1 => "b"
          2 => "c"
        }
```

Abstraction & DSLs

Commonalities

- They clearly have a similar structure
- NOT equally expressive
  - `Enumerable`
- Always converted to `List`
  - `Witchcraft.Functor`

```
list = [
          "a",
          "b",
          "c"
        ]
```

```
tuples = [
            {0, "a"}
            {1, "b"}
            {2, "c"}
          ]
```

```
map = %{
          0 => "a",
          1 => "b"
          2 => "c"
        }
```

Abstraction & DSLs

Commonalities

Commonalities

- Different, but also have similar structure
  - Not very pipeable because 2 paths
  - …lots of duplicate code

Commonalities

- Different, but also have similar structure
  - Not very pipeable because 2 paths
  - …lots of duplicate code
- Why limit to only to two ways?

Abstraction & DSLs

Commonalities

```
def seq_fun(input) do
  position = input / 5

  one_more = input + 1
  bang = inspect(one_more) <> "!"
  string = "#{one_more}#{bang}"

  String.at(string, round(position))
end
```

- Different, but also have similar structure
  - Not very pipeable because 2 paths
  - …lots of duplicate code
- Why limit to only to two ways?

Abstraction & DSLs

Commonalities

```
def seq_fun(input) do
  position = input / 5

  one_more = input + 1
  bang = inspect(one_more) <> "!"
  string = "#{one_more}#{bang}"

  String.at(string, round(position))
end
```

- Different, but also have similar structure
  - Not very pipeable because 2 paths
  - …lots of duplicate code
- Why limit to only to two ways?

```
def par_fun do
  position = Task.async(fn -> input / 5 end)
  string = Task.async(fn ->
    one_more = input + 1
    bang = inspect(one_more)  <> "!"
    "#{one_more}#{bang}"
  end)

  String.at(Task.await(string), round(Task.await(position)))
end
```

Abstraction & DSLs

Start From Rules 📋

Abstraction & DSLs

Start From Rules 📋

- Describe **what** the overall solution looks like

Abstraction & DSLs

Start From Rules 📋

- Describe **what** the overall solution looks like
- Choose **how** it gets run contextually

Abstraction & DSLs

2-Phase

Abstraction & DSLs

2-Phase

- Always a two-phase process

Abstraction & DSLs

2-Phase

- Always a two-phase process
- Abstract, then concrete

Abstraction & DSLs

2-Phase

- Always a two-phase process
- Abstract, then concrete
- Do concretion at application boundary

Abstraction & DSLs

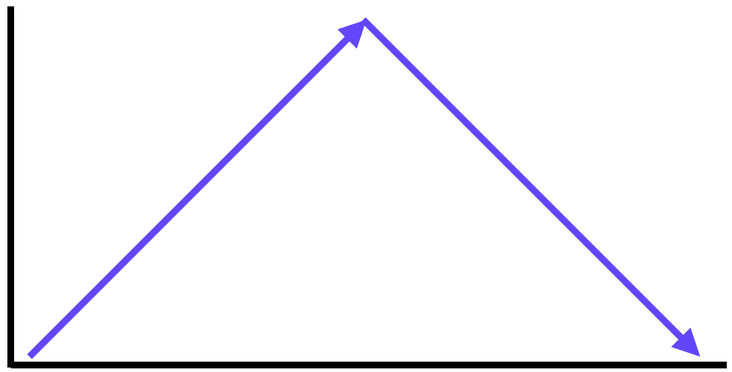
2-Phase

- Always a two-phase process
- Abstract, then concrete
- Do concretion at application boundary

Abstraction & DSLs

2-Phase

- Always a two-phase process
- Abstract, then concrete
- Do concretion at application boundary



Abstraction & DSLs

Improving Kernel

Improving Kernel

- Fallback keys

Improving `Kernel`

- Fallback keys
- Bang-functions

Improving `Kernel` *with Fallback Keys*

Improving `Kernel` *with Fallback Keys*

- Insight:
  - Composition is at the heart of modularity
  - Orthogonality is at the heart of composition

Improving `Kernel` *with Fallback Keys*

- Insight:
  - Composition is at the heart of modularity
  - Orthogonality is at the heart of composition
- Let's abstract default values!
  - More focused (does one thing)
  - More general (works everywhere)
  - Ad hoc function extension

Abstraction & DSLs

Improving `Kernel` *with Fallback Keys*

- Insight:
  - Composition is at the heart of modularity
  - Orthogonality is at the heart of composition
- Let's abstract default values!
  - More focused (does one thing)
  - More general (works everywhere)
  - Ad hoc function extension

```
def get(map, key, default \\ nil)
```

Abstraction & DSLs

Improving `Kernel` *with Fallback Keys*

- Insight:
  - Composition is at the heart of modularity
  - Orthogonality is at the heart of composition
- Let's abstract default values!
  - More focused (does one thing)
  - More general (works everywhere)
  - Ad hoc function extension

```
def get(map, key, default \\ nil)

%{a: 1} |> Map.get(:b, 4)
#=> 4
```

Abstraction & DSLs

Improving `Kernel` *with Fallback Keys*

- Insight:
  - Composition is at the heart of modularity
  - Orthogonality is at the heart of composition
- Let's abstract default values!
  - More focused (does one thing)
  - More general (works everywhere)
  - Ad hoc function extension

```
def get(map, key, default \\ nil)

%{a: 1} |> Map.get(:b, 4)
#=> 4

def fallback(nil, default), do: default
def fallback(val, _), do: value
```

Abstraction & DSLs

Improving `Kernel` *with Fallback Keys*

- Insight:
  - Composition is at the heart of modularity
  - Orthogonality is at the heart of composition
- Let's abstract default values!
  - More focused (does one thing)
  - More general (works everywhere)
  - Ad hoc function extension

```
def get(map, key, default \\ nil)

%{a: 1} |> Map.get(:b, 4)
#=> 4

def fallback(nil, default), do: default
def fallback(val, _), do: value

%{a: 1} |> Map.get(:b) |> fallback(4)
#=> 4
```

Abstraction & DSLs

Improving `Kernel` *with Fallback Keys*

- Insight:
  - Composition is at the heart of modularity
  - Orthogonality is at the heart of composition
- Let's abstract default values!
  - More focused (does one thing)
  - More general (works everywhere)
  - Ad hoc function extension

```
def get(map, key, default \\ nil)

%{a: 1} |> Map.get(:b, 4)
#=> 4

def fallback(nil, default), do: default
def fallback(val, _), do: value

%{a: 1} |> Map.get(:b) |> fallback(4)
#=> 4

[] |> List.first() |> fallback(:empty)
#=> :empty
```

Improving `Kernel` *with(out?) Bang Functions*

Improving `Kernel` *with(out?) Bang Functions*

Get `foo!/*` from `foo/*`

Improving `Kernel` *with(out?) Bang Functions*

```
Map.fetch!(%{a: 1}, :b, fun)
#=> ** (KeyError) key :b not found in: %{a: 1}
```

Get `foo! /*` from `foo /*`

Improving `Kernel` *with(out?) Bang Functions*

```
Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}
```

Get `foo! /*` from `foo /*`

Improving `Kernel` *with(out?) Bang Functions*

```
Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}
```

Get `foo!/*` from `foo/*`

Abstraction & DSLs

Improving `Kernel` *with(out?) Bang Functions*

```
Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}
```

Get `foo!` /* from `foo`/*

Improving `Kernel` *with(out?) Bang Functions*

```
Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}

ensure!(x)
#=> ** (KeyError) key :b not found in: %{a: 1}
```

Get `foo!/*` from `foo/*`

Improving `Kernel` *with(out?) Bang Functions*

Get `foo! /*` from `foo /*`

```
Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}

ensure!(x)
#=> ** (KeyError) key :b not found in: %{a: 1}

value = SafeMap.fetch(%{a: 1}, :a)
#=> 1
```

Improving `Kernel` *with(out?) Bang Functions*

Get `foo! /*` from `foo /*`

```
Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}

ensure!(x)
#=> ** (KeyError) key :b not found in: %{a: 1}

value = SafeMap.fetch(%{a: 1}, :a)
#=> 1

value ~> (&(&1 + 1))
#=> 2

error ~> (&(&1 + 1))
#=> %KeyError{key: :b, message: "..."}
```

Abstraction & DSLs

Improving `Kernel` *with(out?) Bang Functions*

Get `foo!/*` from `foo/*`

```
Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}

ensure!(x)
#=> ** (KeyError) key :b not found in: %{a: 1}

value = SafeMap.fetch(%{a: 1}, :a)
#=> 1

value ~> (&(&1 + 1))
#=> 2

error ~> (&(&1 + 1))
#=> %KeyError{key: :b, message: "..."}

error >>> (&(&1 + 1))
#=> ** (KeyError) key :b not found in: %{a: 1}
```

Abstraction & DSLs

Improving `Kernel` *with(out?) Bang Functions*

```
Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}

ensure!(x)
#=> ** (KeyError) key :b not found in: %{a: 1}

value = SafeMap.fetch(%{a: 1}, :a)
#=> 1

value ~> (&(&1 + 1))
#=> 2

error ~> (&(&1 + 1))
#=> %KeyError{key: :b, message: "..."}

error >>> (&(&1 + 1))
#=> ** (KeyError) key :b not found in: %{a: 1}
```

Works everywhere

Get `foo!` /* from `foo`/*

Abstraction & DSLs

Improving `Kernel` *with(out?) Bang Functions*

```
Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}

ensure!(x)
#=> ** (KeyError) key :b not found in: %{a: 1}

value = SafeMap.fetch(%{a: 1}, :a)
#=> 1

value ~> (&(&1 + 1))
#=> 2

error ~> (&(&1 + 1))
#=> %KeyError{key: :b, message: "..."}

error >>> (&(&1 + 1))
#=> ** (KeyError) key :b not found in: %{a: 1}
```

Works everywhere

Any data

Get `foo!` /* from `foo`/*

Abstraction & DSLs

Improving `Kernel` *with(out?) Bang Functions*

Get `foo! /* from foo/*`

```
Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}

ensure!(x)
#=> ** (KeyError) key :b not found in: %{a: 1}

value = SafeMap.fetch(%{a: 1}, :a)
#=> 1

value ~> (&(&1 + 1))
#=> 2

error ~> (&(&1 + 1))
#=> %KeyError{key: :b, message: "..."}

error >>> (&(&1 + 1))
#=> ** (KeyError) key :b not found in: %{a: 1}
```

Works everywhere

Any data

Any exception struct

Abstraction & DSLs

Improving `Kernel` *with(out?) Bang Functions*

Get `foo!` /* from `foo`/*

```
Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}

ensure!(x)
#=> ** (KeyError) key :b not found in: %{a: 1}

value = SafeMap.fetch(%{a: 1}, :a)
#=> 1

value ~> (&(&1 + 1))
#=> 2

error ~> (&(&1 + 1))
#=> %KeyError{key: :b, message: "..."}

error >>> (&(&1 + 1))
#=> ** (KeyError) key :b not found in: %{a: 1}
```

Works everywhere

Any data

Any exception struct

Your choice o flow (e.g. pipes!)

Abstraction & DSLs

Improving `Kernel` *with(out?) Bang Functions*

Get `foo!` /* from `foo` /*

```
Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}

ensure!(x)
#=> ** (KeyError) key :b not found in: %{a: 1}

value = SafeMap.fetch(%{a: 1}, :a)
#=> 1

value ~> (&(&1 + 1))
#=> 2

error ~> (&(&1 + 1))
#=> %KeyError{key: :b, message: "..."}

error >>> (&(&1 + 1))
#=> ** (KeyError) key :b not found in: %{a: 1}
```

Works everywhere

Any data

Any exception struct

Your choice o flow (e.g. pipes!)

Super easy to test

Abstraction & DSLs

Improving `Kernel` *with(out?) Bang Functions*

Get `foo!` /* from `foo` /*

```
Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}

ensure!(x)
#=> ** (KeyError) key :b not found in: %{a: 1}

value = SafeMap.fetch(%{a: 1}, :a)
#=> 1

value ~> (&(&1 + 1))
#=> 2

error ~> (&(&1 + 1))
#=> %KeyError{key: :b, message: "..."}

error >>> (&(&1 + 1))
#=> ** (KeyError) key :b not found in: %{a: 1}
```

Works everywhere

Any data

Any exception struct

Your choice o flow (e.g. pipes!)

Super easy to test

Abstraction & DSLs

Improving `Kernel` *with(out?) Bang Functions*

Get `foo!` /* from `foo` /*

```
Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}

ensure!(x)
#=> ** (KeyError) key :b not found in: %{a: 1}

value = SafeMap.fetch(%{a: 1}, :a)
#=> 1

value ~> (&(&1 + 1))
#=> 2

error ~> (&(&1 + 1))
#=> %KeyError{key: :b, message: "..."}

error >>> (&(&1 + 1))
#=> ** (KeyError) key :b not found in: %{a: 1}
```

Works everywhere

Any data

Any exception struct

Your choice o flow (e.g. pipes!)

Super easy to test

BONUS

Abstraction & DSLs

Improving `Kernel` *with(out?) Bang Functions*

Get `foo!` /* from `foo`/*

```
Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}

ensure!(x)
#=> ** (KeyError) key :b not found in: %{a: 1}

value = SafeMap.fetch(%{a: 1}, :a)
#=> 1

value ~> (&(&1 + 1))
#=> 2

error ~> (&(&1 + 1))
#=> %KeyError{key: :b, message: "..."}

error >>> (&(&1 + 1))
#=> ** (KeyError) key :b not found in: %{a: 1}
```

Works everywhere

Any data

Any exception struct

Your choice o flow (e.g. pipes!)

Super easy to test

BONUS

Fix `nil` blindness,

Abstraction & DSLs

Improving `Kernel` *with(out?) Bang Functions*

Get `foo!` /* from `foo`/*

```
Map.fetch!(%{a: 1}, :b)
#=> ** (KeyError) key :b not found in: %{a: 1}

use Exceptional

error = SafeMap.fetch(%{a: 1}, :b)
#=> %KeyError{key: :b, message: "..."}

ensure!(x)
#=> ** (KeyError) key :b not found in: %{a: 1}

value = SafeMap.fetch(%{a: 1}, :a)
#=> 1

value ~> (&(&1 + 1))
#=> 2

error ~> (&(&1 + 1))
#=> %KeyError{key: :b, message: "..."}

error >>> (&(&1 + 1))
#=> ** (KeyError) key :b not found in: %{a: 1}
```

Works everywhere

Any data

Any exception struct

Your choice o flow (e.g. pipes!)

Super easy to test

BONUS

Fix `nil` blindness,
the "billion dollar mistake"

A Note on Metaphor

Abstraction & DSLs

A Note on Metaphor

- Concept: Flow-ability is very core to Elixir's ethos
  - **`Kernel.|>/2`**

Abstraction & DSLs

A Note on Metaphor

- Concept: Flow-ability is very core to Elixir's ethos
  - **`Kernel.|>/2`**
- Consistent flow metaphor / punning on existing metaphor
  - Exceptional: **`~>/2`** and **`>>>/2`**

Because it's easier now

Abstraction & DSLs

A Note on Metaphor

- Concept: Flow-ability is very core to Elixir's ethos
  - **`Kernel.|>/2`**
- Consistent flow metaphor / punning on existing metaphor
  - Exceptional: **`~>/2`** and **`>>>/2`**

```
[1,2,3]
|> hypothetical_returns_exception()
~> Enum.map(fn x -> x + 1 end)
~> Enum.sum()
>>> fn x -> x + 1 end.()
```

Abstraction & DSLs

A Note on Metaphor

- Concept: Flow-ability is very core to Elixir's ethos
  - **`Kernel.|>/2`**
- Consistent flow metaphor / punning on existing metaphor
  - Exceptional: **`~>/2`** and **`>>>/2`**

```
[1,2,3]
|> hypothetical_returns_exception()
~> Enum.map(fn x -> x + 1 end)
~> Enum.sum()
>>> fn x -> x + 1 end.()
```

Abstraction & DSLs

A Note on Metaphor

- Concept: Flow-ability is very core to Elixir's ethos
  - **`Kernel.|>/2`**
- Consistent flow metaphor / punning on existing metaphor
  - Exceptional: **`~>/2`** and **`>>>/2`**

```
[1,2,3]
|> hypothetical_returns_exception()
~> Enum.map(fn x -> x + 1 end)
~> Enum.sum()
>>> fn x -> x + 1 end.()
```

Abstraction & DSLs

A Note on Metaphor

- Concept: Flow-ability is very core to Elixir's ethos
  - **`Kernel.|>/2`**
- Consistent flow metaphor / punning on existing metaphor
  - Exceptional: **`~>/2`** and **`>>>/2`**

```
[1,2,3]
|> hypothetical_returns_exception()
~> Enum.map(fn x -> x + 1 end)
~> Enum.sum()
>>> fn x -> x + 1 end.()
```

Abstraction & DSLs

What's Gained

Abstraction & DSLs

What's Gained

- Clear

Abstraction & DSLs

What's Gained

- Clear
- Composable

Abstraction & DSLs

What's Gained

- Clear
- Composable
- Greater reuse ♻️

Abstraction & DSLs

What's Gained

- Clear
- Composable
- Greater reuse ♻️
- User choice

Abstraction & DSLs

What's Gained

- Clear
- Composable
- Greater reuse ♻️
- User choice
- Increased testability
  - Simple example: `is_exception?/1`

Abstraction & DSLs

What's Gained

- Clear
- Composable
- Greater reuse ♻️
- User choice
- Increased testability
  - Simple example: `is_exception?/1`
- Could still add protocol to get even more power

Abstraction & DSLs

Storytelling 📖

```
conn
|> route()
|> parse()
|> model()
|> view()
|> render()
```

Abstraction & DSLs

Storytelling 📖

- Your code read like a story

```
conn
|> route()
|> parse()
|> model()
|> view()
|> render()
```

Abstraction & DSLs

Storytelling 📖

- Your code read like a story
- We even see this in high-level goals of (e.g.) Phoenix

```
conn
|> route()
|> parse()
|> model()
|> view()
|> render()
```

Abstraction & DSLs

Storytelling 📖

- Your code read like a story
- We even see this in high-level goals of (e.g.) Phoenix
- Go make some DSLs!

```
conn
|> route()
|> parse()
|> model()
|> view()
|> render()
```

Abstraction & DSLs

How to Eat the Elephant 🍴

Abstraction & DSLs

How to Eat the Elephant 🍴

- By feature?

Abstraction & DSLs

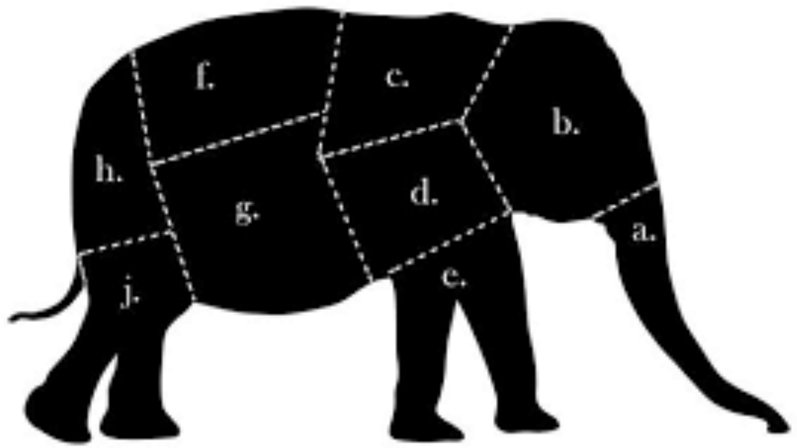
How to Eat the Elephant 🍴

- By feature?
- By behaviour?

Abstraction & DSLs

How to Eat the Elephant 🍴

- By feature?
- By behaviour?
- By structure / properties!



Intermission Puzzle

Intermission Puzzle

What Do The Following Have In Common?

Intermission Puzzle

What Do The Following Have In Common?

- Async/await (or Task, if you prefer)

Intermission Puzzle

What Do The Following Have In Common?

- Async/await (or Task, if you prefer)
- throw/catch

Intermission Puzzle

What Do The Following Have In Common?

- Async/await (or Task, if you prefer)
- throw/catch
- with blocks

Intermission Puzzle

What Do The Following Have In Common?

- Async/await (or Task, if you prefer)
- throw/catch
- with blocks
- SQL queries — LINQ

Intermission Puzzle

What Do The Following Have In Common?

- Async/await (or Task, if you prefer)
- throw/catch
- with blocks
- SQL queries — LINQ
- JSON parsing

Intermission Puzzle

What Do The Following Have In Common?

- Async/await (or Task, if you prefer)
- throw/catch
- with blocks
- SQL queries — LINQ
- JSON parsing
- "Warm fuzzy thing"

Structure

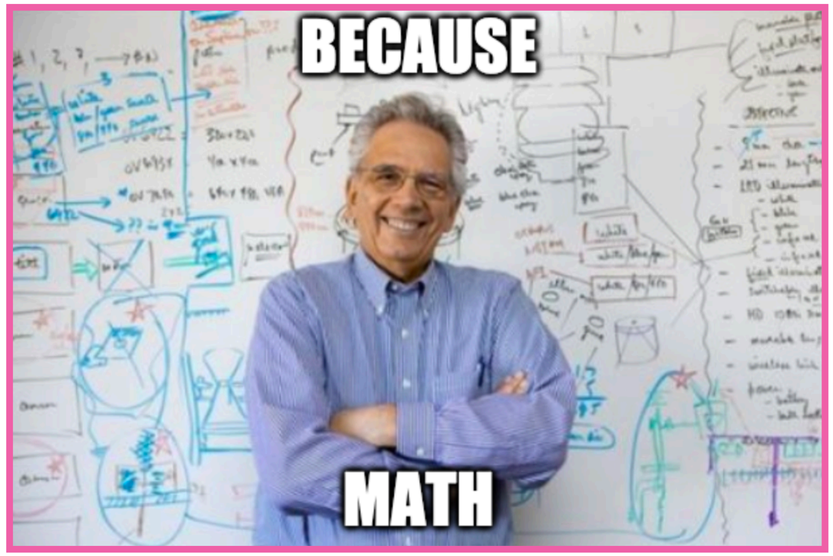
One of these things is like all the others

Structure

There Are Only Three Right Answers

1

2



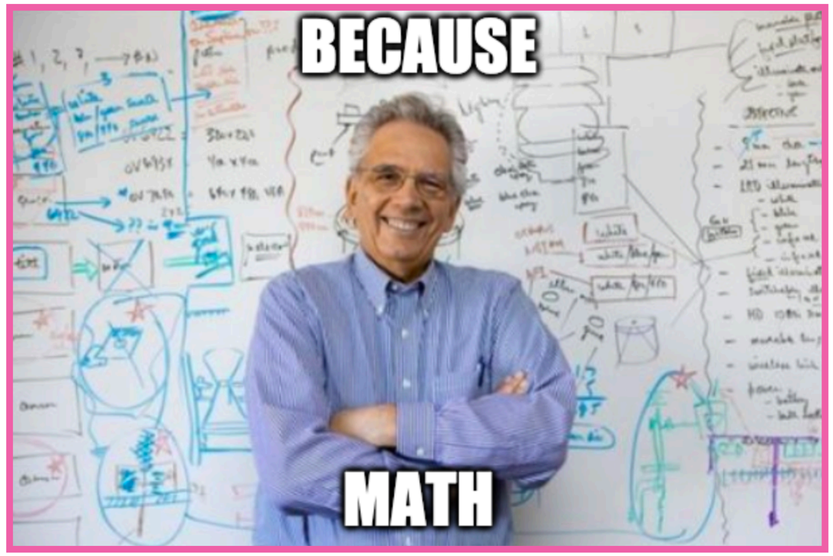
3

Structure

There Are Only Three Right Answers

1

2



3

Structure

Associativity

```
(a • b) • c == a • (b • c)

AKA

concat(concat(a, b), c) == concat(a, concat(b, c))
```

Structure

Associativity

```
(a • b) • c == a • (b • c)

AKA

concat(concat(a, b), c) == concat(a, concat(b, c))
```

- Not a data structure

Structure

Associativity

```
(a • b) • c == a • (b • c)

AKA

concat(concat(a, b), c) == concat(a, concat(b, c))
```

- Not a data structure
- Not a function

Structure

Associativity

```
(a • b) • c == a • (b • c)

AKA

concat(concat(a, b), c) == concat(a, concat(b, c))
```

- Not a data structure
- Not a function
- An interface & rules!

Structure

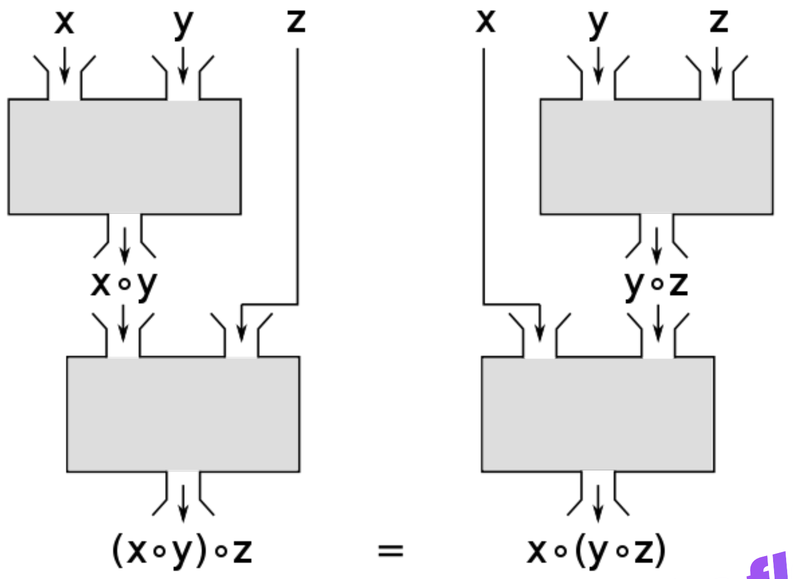
Associativity

```
(a • b) • c == a • (b • c)

AKA

concat(concat(a, b), c) == concat(a, concat(b, c))
```

- Not a data structure
- Not a function
- An interface & rules!



(Note the flow metaphor)

Structure

A Semigroup On...

Structure

A Semigroup On...

```
defprotocol Semigroup do
  def concat(a, b)
end

defimpl Semigroup, for: Integer do
  def concat(a, b), do: a + b
end

defimpl Semigroup, for: List do
  def concat(xs, ys), do: xs ++ ys
end
```

Structure

An Unlawful Counterexample 🚨

```
 1.0 / (2.0  / 3.0) == 1.5
(1.0 /  2.0) / 3.0  == 0.1666...
```

Structure

How to Enforce Properties

Structure

How to Enforce Properties

- A structure of structures
  - Keep it in your brain
  - Manually prop test
  - Enforce with `TypeClass`

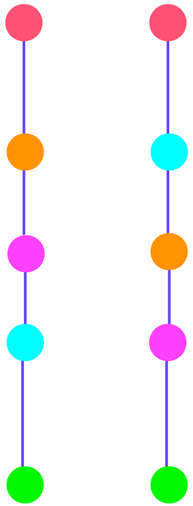
Let's Do Something

Wild

(Power Up Pipes)

Power Up

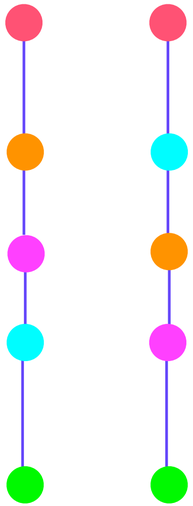
Explicit Assumptions



Power Up

Explicit Assumptions

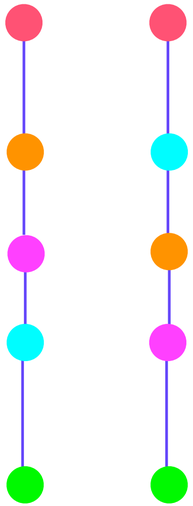
- Parallel pipes!
  - Concurrency = partial order
  - Monotonic
  - All loops must be linearized



Power Up

Explicit Assumptions

- Parallel pipes!
  - Concurrency = partial order
  - Monotonic
  - All loops must be linearized
- Properties
  - Serial composition
  - Parallel composition
  - Explicit evaluation strategy



Power Up

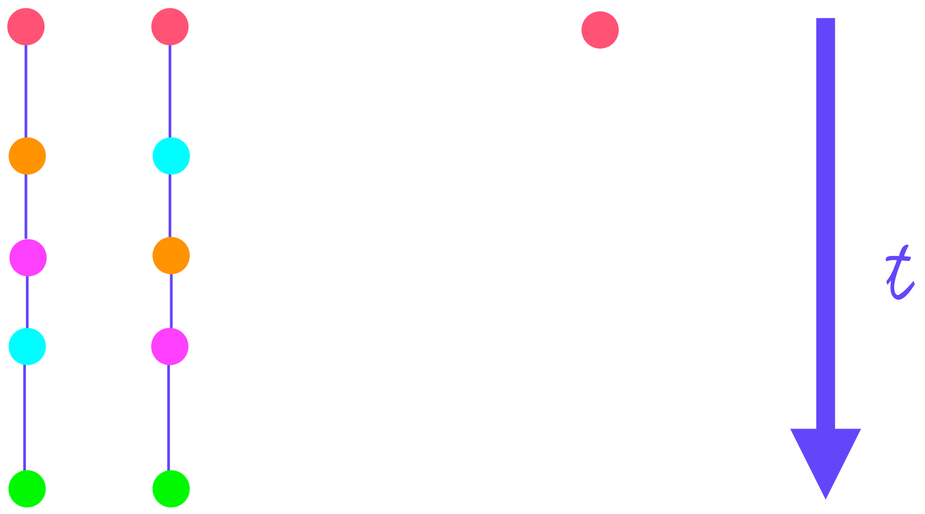
Explicit Assumptions

- Parallel pipes!
  - Concurrency = partial order
  - Monotonic
  - All loops must be linearized
- Properties
  - Serial composition
  - Parallel composition
  - Explicit evaluation strategy

Power Up

Explicit Assumptions

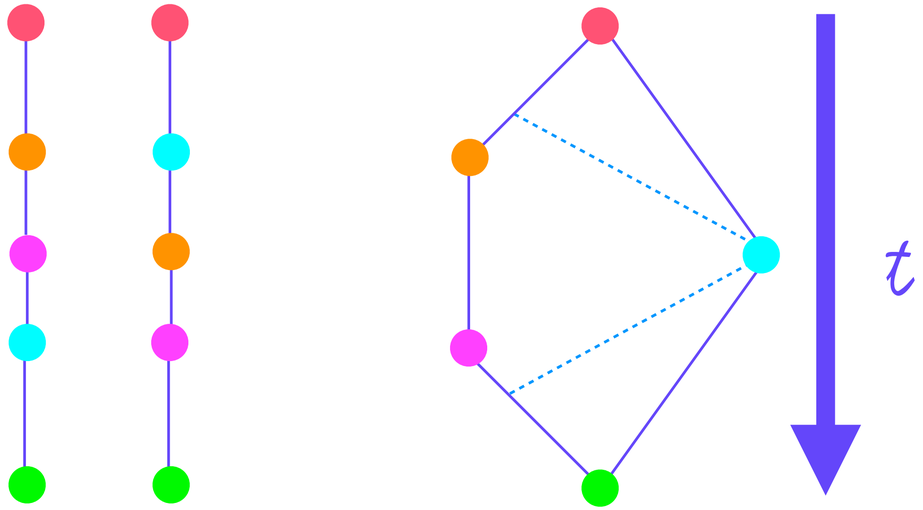
- Parallel pipes!
  - Concurrency = partial order
  - Monotonic
  - All loops must be linearized
- Properties
  - Serial composition
  - Parallel composition
  - Explicit evaluation strategy



Power Up

Explicit Assumptions

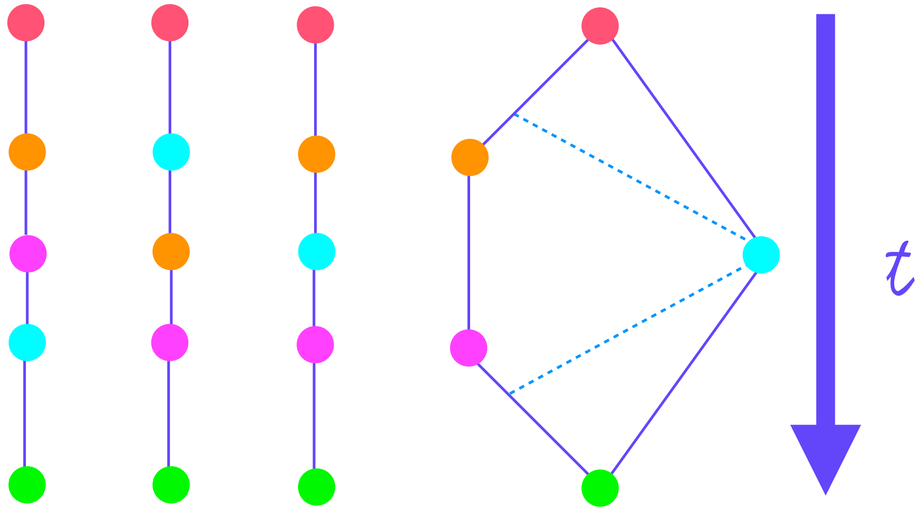
- Parallel pipes!
  - Concurrency = partial order
  - Monotonic
  - All loops must be linearized
- Properties
  - Serial composition
  - Parallel composition
  - Explicit evaluation strategy



Power Up

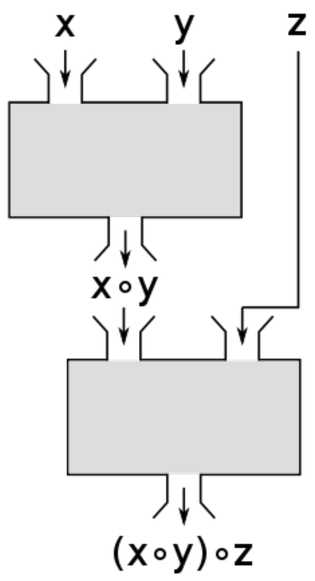
Explicit Assumptions

- Parallel pipes!
  - Concurrency = partial order
  - Monotonic
  - All loops must be linearized
- Properties
  - Serial composition
  - Parallel composition
  - Explicit evaluation strategy



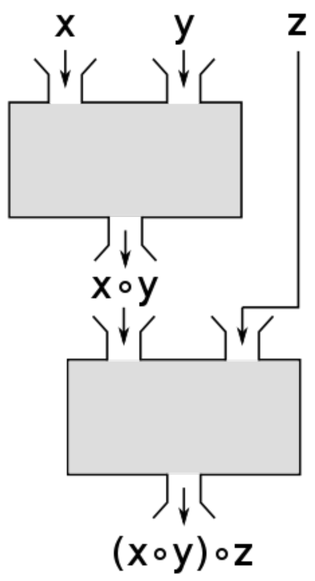
Power Up

Pipes++



Power Up

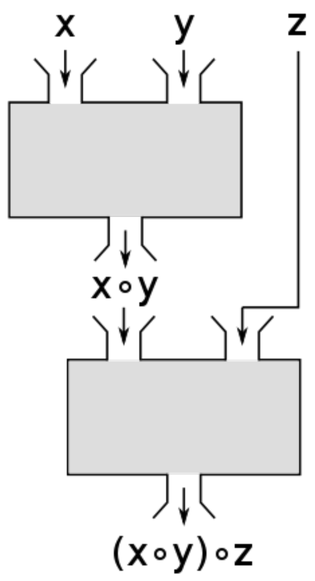
Pipes++




```
arrow_diagram =
  fanout(fn x -> x / 5 end, fn y -> y + 1 end
  |                         <~> fanout(&inspect/1, fn z -> z end) |
  |                         <~> unsplit(fn(a, b) -> "#{b}#{a}" end))
  | <~> unsplit(&String.at(&2, round(&1)) end)
```

Power Up

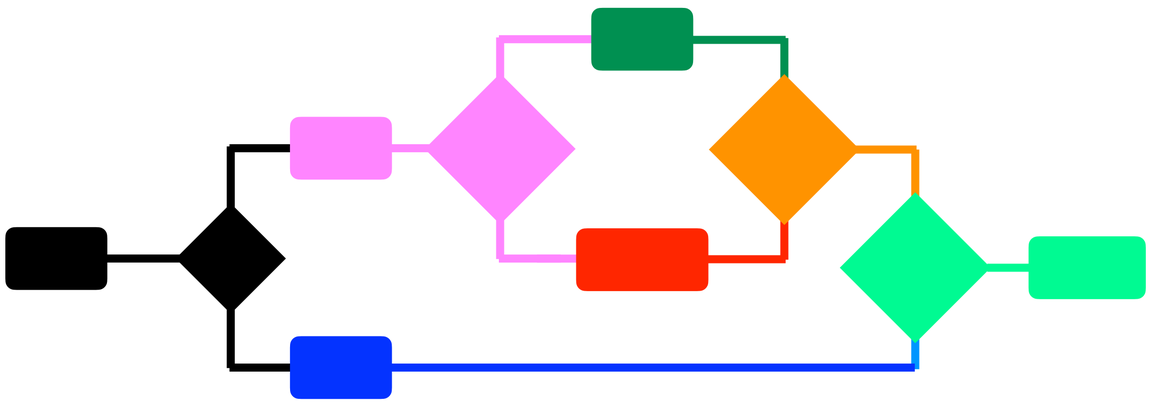
Pipes++

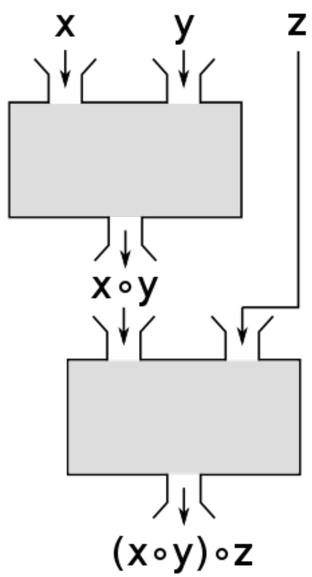



```
arrow_diagram =
  fanout(fn x -> x / 5 end, fn y -> y + 1 end
  |                         <~> fanout(&inspect/1, fn z -> z end)
  |                         <~> unsplit(fn(a, b) -> "#{b}#{a}" end))
  | <~> unsplit(&String.at(&2, round(&1)) end)
```

Power Up

Pipes++






```
arrow_diagram =
  fanout(fn x -> x / 5 end, fn y -> y + 1 end
                            <~> fanout(&inspect/1, fn z -> z end)
                            <~> unsplit(fn(a, b) -> "#{b}#{a}" end))
  <~> unsplit(&String.at(&2, round(&1)) end)
```

Power Up

How?!

Power Up

How?!

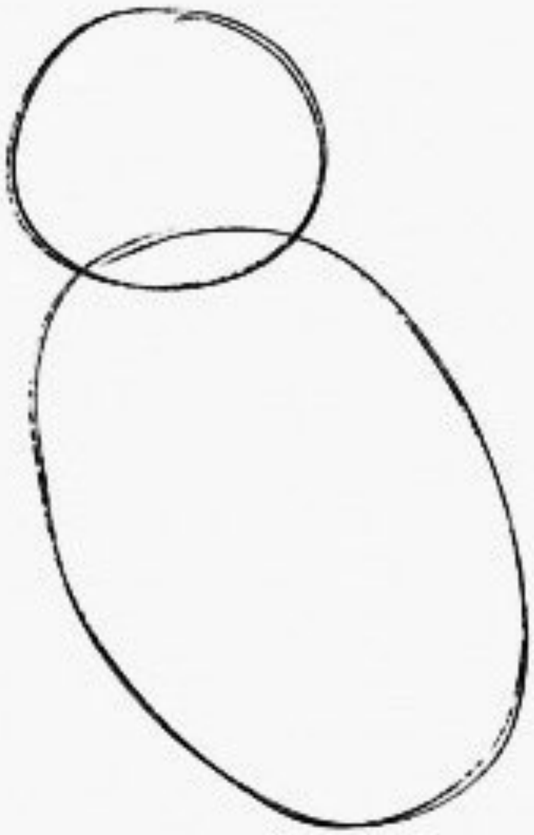


Fig 1. Draw two circles

Fig 2. Draw the rest of the damn Owl

Power Up

Cleanup

```
split do
  fn x -> x / 5 end

  fn y -> y + 1 end |> into(split do
    &inspect/1
    fn z -> z end
  end)
  |> unsplit(fn (a, b) -> "#{b}#{a}" end)
end
|> unsplit(&String.at(&2, round(&1)))
```

Power Up

Cleanup

```
split do
  fn x -> x / 5 end

  fn y -> y + 1 end |> into(split do
    &inspect/1
    fn z -> z end
  end)
  |> unsplit(fn (a, b) -> "#{b}#{a}" end)
end
|> unsplit(&String.at(&2, round(&1)))
```

Power Up

Carrier Data

```
%Unsplit{
  split: %Split{
    left: fn x -> x / 5 end,
    right: %Tree{
      node: fn y -> y + 1 end,
      left: %Unsplit{
        node: %Split{
          left: &inspect/1,
          right: fn z -> z end
        },
        with: fn (left, right) -> "#{right}#{left}" end
      }
    }
  },
  with: &String.at(&2, round(&1))
}
```

Power Up

Carrier Data

```
%Unsplit{
  split: %Split{
    left: fn x -> x / 5 end,
    right: %Tree{
      node: fn y -> y + 1 end,
      left: %Unsplit{
        node: %Split{
          left: &inspect/1,
          right: fn z -> z end
        },
        with: fn (left, right) -> "#{right}#{left}" end
      }
    }
  },
  with: &String.at(&2, round(&1))
}
```

Power Up

Carrier Data

```
%Unsplit{
  split: %Split{
    left: fn x -> x / 5 end,
    right: %Tree{
      node: fn y -> y + 1 end,
      left: %Unsplit{
        node: %Split{
          left: &inspect/1,
          right: fn z -> z end
        },
        with: fn (left, right) -> "#{right}#{left}" end
      }
    }
  },
  with: &String.at(&2, round(&1))
}
```

Power Up

Carrier Data

```
%Unsplit{
  split: %Split{
    left: fn x -> x / 5 end,
    right: %Tree{
      node: fn y -> y + 1 end,
      left: %Unsplit{
        node: %Split{
          left: &inspect/1,
          right: fn z -> z end
        },
        with: fn (left, right) -> "#{right}#{left}" end
      }
    }
  },
  with: &String.at(&2, round(&1))
}
```

Power Up

Carrier Data

```
%Unsplit{
  split: %Split{
    left: fn x -> x / 5 end,
    right: %Tree{
      node: fn y -> y + 1 end,
      left: %Unsplit{
        node: %Split{
          left: &inspect/1,
          right: fn z -> z end
        },
        with: fn (left, right) -> "#{right}#{left}" end
      }
    }
  },
  with: &String.at(&2, round(&1))
}
```

Power Up

Carrier Data

```
defprotocol Dataflow do
  def split(input, path_a, path_b)
  def unsplit(pre_spilt, by: combine)
end
```

Power Up

Base Case

```
defprotocol Dataflow do
  def split(input, path_a, path_b)
  def unsplit(pre_spilt, by: combine)
end
```

Power Up

Base Case

```
defprotocol Dataflow do
  def split(input, path_a, path_b)
  def unsplit(pre_spilt, by: combine)
end
```

```
defimpl Dataflow, for: Any do
  def split(input, path_a, path_b) do
    {path_a(input), path_b(input)}
  end

  def unsplit({a, b}, by: combine), do: combine(a, b)
end
```

Power Up

Base Case

```
split 45 do
  fn x -> x / 5 end

  fn y -> y + 1 end
  |> split(do
    &inspect/1
    fn z -> z end
  end)
  |> unsplit(by: fn (a, b) -> "#{b}#{a}" end)
end
|> unsplit(by: &String.at(&2, round(&1)))
```

```
defprotocol Dataflow do
  def split(input, path_a, path_b)
  def unsplit(pre_spilt, by: combine)
end
```

```
defimpl Dataflow, for: Any do
  def split(input, path_a, path_b) do
    {path_a(input), path_b(input)}
  end

  def unsplit({a, b}, by: combine), do: combine(a, b)
end
```

Power Up

Async

```
defprotocol Dataflow do
  def split(input, path_a, path_b)
  def unsplit(pre_spilt, by: combine)
end
```

Power Up

Async

```
defprotocol Dataflow do
  def split(input, path_a, path_b)
  def unsplit(pre_spilt, by: combine)
end
```

```
defmodule Async do
  defstruct :value

  def asyncify(input) do
    %Async{value: input}
  end

  def syncify(%{value: value}), do: value
end
```

Power Up

Async

```
defprotocol Dataflow do
  def split(input, path_a, path_b)
  def unsplit(pre_spilt, by: combine)
end
```

```
defmodule Async do
  defstruct :value

  def asyncify(input) do
    %Async{value: input}
  end

  def syncify(%{value: value}), do: value
end
```

```
defimpl Dataflow, for: Async do
  def split(%{value: input}, path_a, path_b) do
    %Async{
      value: {
        Task.async(path_a(input)),
        Task.async(path_b(input))
      }
    }
  end

  def unsplit({a, b}, by: combine) do
    %Async{value: combine(Task.await(a), Task.await(b))}
  end
end
```

Power Up

Async

```
45
|> asyncify()
|> split do
  fn x -> x / 5 end

  fn y -> y + 1 end |> split(do
    &inspect/1
    fn z -> z end
  end)
  |> unsplit(fn (a, b) -> "#{b}#{a}" end)
end
|> unsplit(&String.at(&2, round(&1)))
|> syncify()
```

```
defprotocol Dataflow do
  def split(input, path_a, path_b)
  def unsplit(pre_spilt, by: combine)
end
```

```
defmodule Async do
  defstruct :value

  def asyncify(input) do
    %Async{value: input}
  end

  def syncify(%{value: value}), do: value
end
```

```
defimpl Dataflow, for: Async do
  def split(%{value: input}, path_a, path_b) do
    %Async{
      value: {
        Task.async(path_a(input)),
        Task.async(path_b(input))
      }
    }
  end

  def unsplit({a, b}, by: combine) do
    %Async{value: combine(Task.await(a), Task.await(b))}
  end
end
```

Power Up

Upshot

Power Up

Upshot

- Higher ***semantic density*** (meaning > mechanics)

Power Up

Upshot

- Higher ***semantic density*** (meaning > mechanics)
- Declarative, ***configurable*** data flow 🤯

Power Up

Upshot

- Higher ***semantic density*** (meaning > mechanics)
- Declarative, ***configurable*** data flow 🤯
- Extremely extensible
  - `defimpl Dataflow, for: %Stream{}`
  - `defimpl Dataflow, for: %Distributed{}`
  - `defimpl Dataflow, for: %Broadway{}`

Power Up

Upshot

- Higher ***semantic density*** (meaning > mechanics)
- Declarative, ***configurable*** data flow 🤯
- Extremely extensible
  - `defimpl Dataflow, for: %Stream{}`
  - `defimpl Dataflow, for: %Distributed{}`
  - `defimpl Dataflow, for: %Broadway{}`
- Model-testable

Power Up

Upshot

- Higher ***semantic density*** (meaning > mechanics)
- Declarative, ***configurable*** data flow 🤯
- Extremely extensible
  - `defimpl Dataflow, for: %Stream{}`
  - `defimpl Dataflow, for: %Distributed{}`
  - `defimpl Dataflow, for: %Broadway{}`
- Model-testable
- Composable with other pipes and ***change evaluation strategies***

Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident.

Data structures, not algorithms, are central to programming.

Rob Pike, 5 Rules of Programming

A Call for

Libraries

A Call for Libraries

Summary

A Call for Libraries

Summary

- Can plug into / extend

A Call for Libraries

Summary

- Can plug into / extend
- Single-threaded context

A Call for Libraries

Summary

- Can plug into / extend
- Single-threaded context
- Distributed context

A Call for Libraries

Summary

- Can plug into / extend
- Single-threaded context
- Distributed context
- Dynamic hybrid contexts

A Call for Libraries

Extend Railroad Programming

```
def unreliable() do
  exploding()
  dangerous()
  bad()
  mightFail()
rescue
  err -> handleOrReport(err)
end
```

A Call for Libraries

Extend Railroad Programming

```
def unreliable() do
  exploding()
  dangerous()
  bad()
  mightFail()
rescue
  err -> handleOrReport(err)
end
```

Happy Path (Continue)

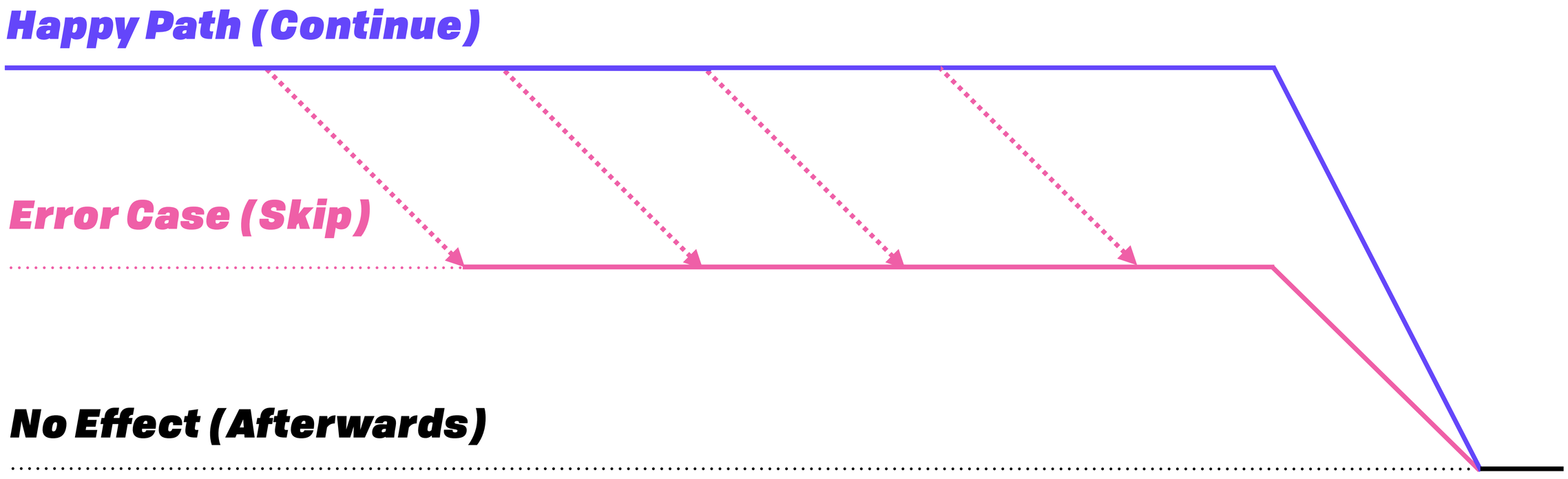
Error Case (Skip)

No Effect (Afterwards)

A Call for Libraries

Extend Railroad Programming

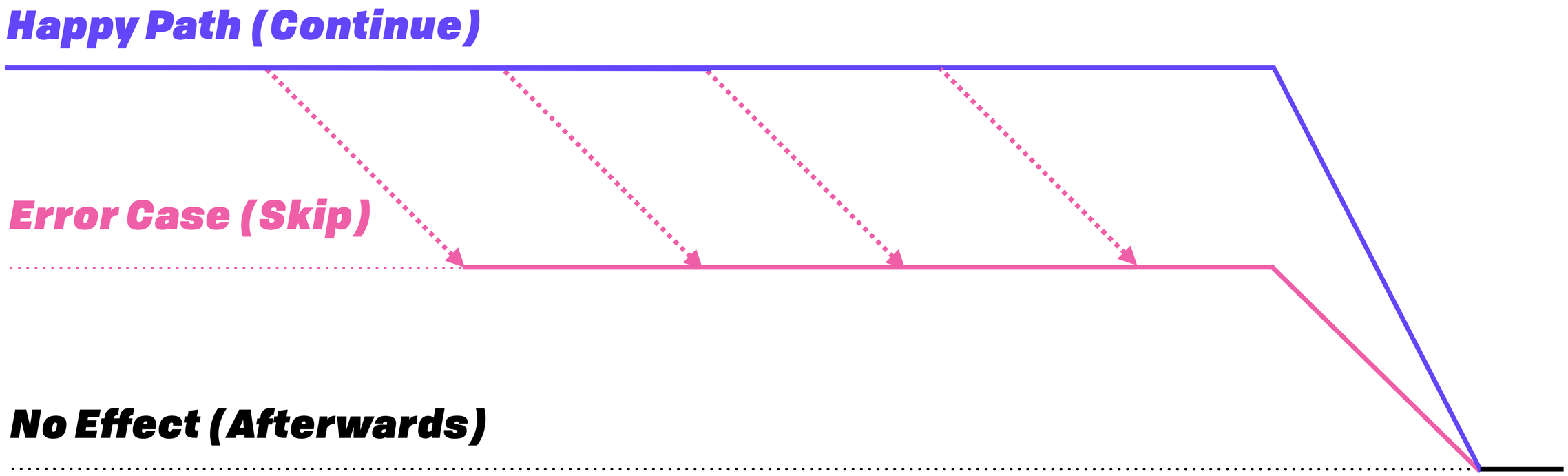
```
def unreliable() do
  exploding()
  dangerous()
  bad()
  mightFail()
rescue
  err -> handleOrReport(err)
end
```



A Call for Libraries

Extend Railroad Programming

```
def unreliable() do
  exploding()
  dangerous()
  bad()
  mightFail()
rescue
  err -> handleOrReport(err)
end
```



A Call for Libraries

Surprising Number of Factors

```
def foo(val) do
  Task.async(fn ->
    IO.inspect(val)

    bar = fn (inner_val) ->
      Task.async(fn ->
        IO.inspect(inner_val)
      end)
    end

    bar(val + 2)
  end)
end

foo(42)
```

A Call for Libraries

Surprising Number of Factors

```
def foo(val) do
  Task.async(fn ->
    IO.inspect(val)

    bar = fn (inner_val) ->
      Task.async(fn ->
        IO.inspect(inner_val)
      end)
    end

    bar(val + 2)
  end)
end

foo(42)
```

Log

Program

A Call for Libraries

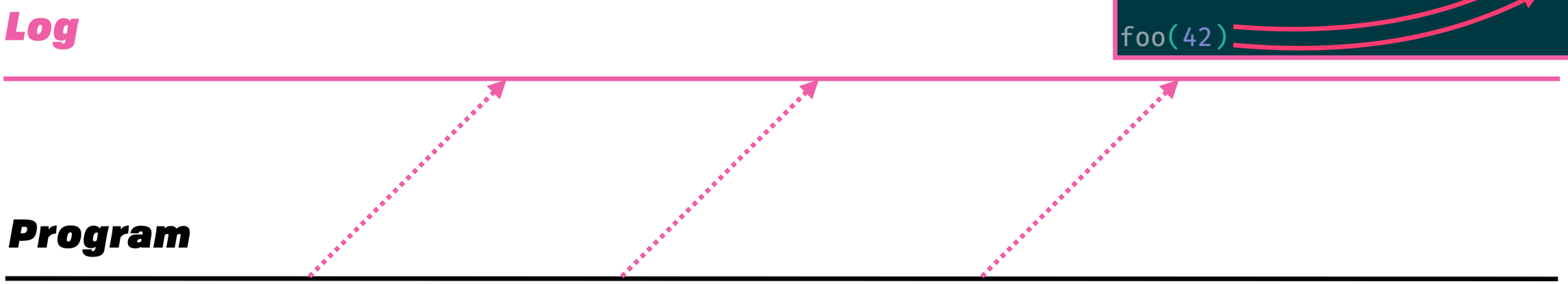
Surprising Number of Factors

```
def foo(val) do
  Task.async(fn ->
    IO.inspect(val)

    bar = fn (inner_val) ->
      Task.async(fn ->
        IO.inspect(inner_val)
      end)
    end

    bar(val + 2)
  end)
end

foo(42)
```



Summary

Summary

Keep In Mind...

Summary

Keep In Mind...

- Protocols-for-DDD

Summary

Keep In Mind...

- Protocols-for-DDD
- Add a semantic layer

Summary

Keep In Mind...

- Protocols-for-DDD
- Add a semantic layer
- How do you locally test your distributed system? Look at the properties!

Summary

Keep In Mind...

- Protocols-for-DDD
- Add a semantic layer
- How do you locally test your distributed system? Look at the properties!
- Under which conditions does your code work? What are your assumptions?

Summary

Keep In Mind...

- Protocols-for-DDD
- Add a semantic layer
- How do you locally test your distributed system? Look at the properties!
- Under which conditions does your code work? What are your assumptions?
- Prop testing is useful for structured abstractions

Summary

Keep In Mind...

- Protocols-for-DDD
- Add a semantic layer
- How do you locally test your distributed system? Look at the properties!
- Under which conditions does your code work? What are your assumptions?
- Prop testing is useful for structured abstractions
- You should be able to code half-asleep

https://fission.codes
https://talk.fission.codes
https://tools.fission.codes

🇧🇷 Thank You, Elixir Brasil 🎉

brooklyn@fission.codes
github.com/expede
@expede

LIBRARY PRINCIPLES 📚 GENERALITY

SWEET SPOTS 🍭

SWEET SPOTS 🍭

Generality

- Low information
- Few assumptions
- Many use cases

SWEET SPOTS 🍭

Generality

- Low information
- Few assumptions
- Many use cases

Power

- High information
- Can make many assumptions
- Tailored to few use cases

SWEET SPOTS 🍭

Generality

- Low information
- Few assumptions
- Many use cases

Power

- High information
- Can make many assumptions
- Tailored to few use cases

SWEET SPOTS 🍭

Generality

- Low information
- Few assumptions
- Many use cases

Power

- High information
- Can make many assumptions
- Tailored to few use cases

SWEET SPOTS 🍭

Generality

- Low information
- Few assumptions
- Many use cases

Power

- High information
- Can make many assumptions
- Tailored to few use cases

SWEET SPOTS 🍭

Generality

- Low information
- Few assumptions
- Many use cases

Power

- High information
- Can make many assumptions
- Tailored to few use cases

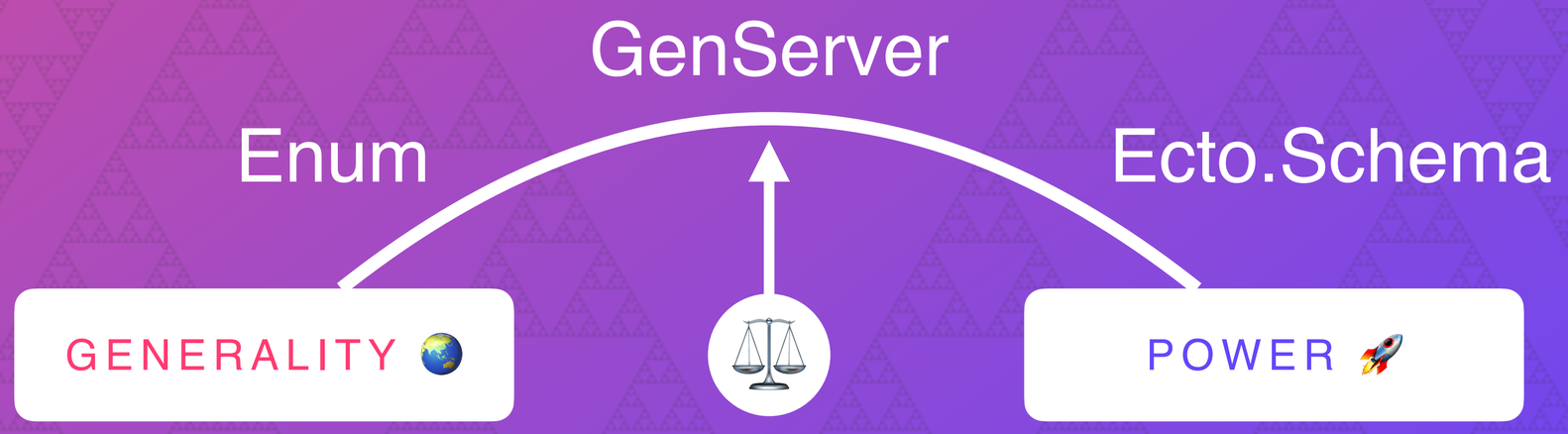
SWEET SPOTS 🍭

Generality

- Low information
- Few assumptions
- Many use cases

Power

- High information
- Can make many assumptions
- Tailored to few use cases



SWEET SPOTS 🍭

Generality

- Low information
- Few assumptions
- Many use cases

Power

- High information
- Can make many assumptions
- Tailored to few use cases