ILLUMINATING THE

DARKEST

OF MAGICS

✨ TOWARDS A DISCIPLINE OF LIBRARY DESIGN 📚

Like punning, programming is a **play on words**

ALAN J. PERLIS

Experience is the name everyone gives to their **mistakes**

OSCAR WILDE, “LADY WINDERMERE'S FAN“

META

TALK ABOUT TALK

META

BROOKLYN ZELENKA

@EXPEDE

- CEO & Chief Scientist at FISSION
- 🔗 https://tools.fission.codes
- Ethereum Core Dev
- 📜 Authored a bunch of standards
- Previously Elixir consultant & trainer
- Witchcraft, Algae, Quark, Exceptional
- Open Sourceress
- PLT Enthusiast
- λ Founded VanFP

META

BROOKLYN ZELENK

@EXPEDE

- CEO & Chief Scientist at FISSION
- 🔗 https://tools.fission.codes
- Ethereum Core Dev
- 📜 Authored a bunch of standards
- Previously Elixir consultant & trainer
- Witchcraft, Algae, Quark, Exceptional
- Open Sourceress
- PLT Enthusiast
- λ Founded VanFP

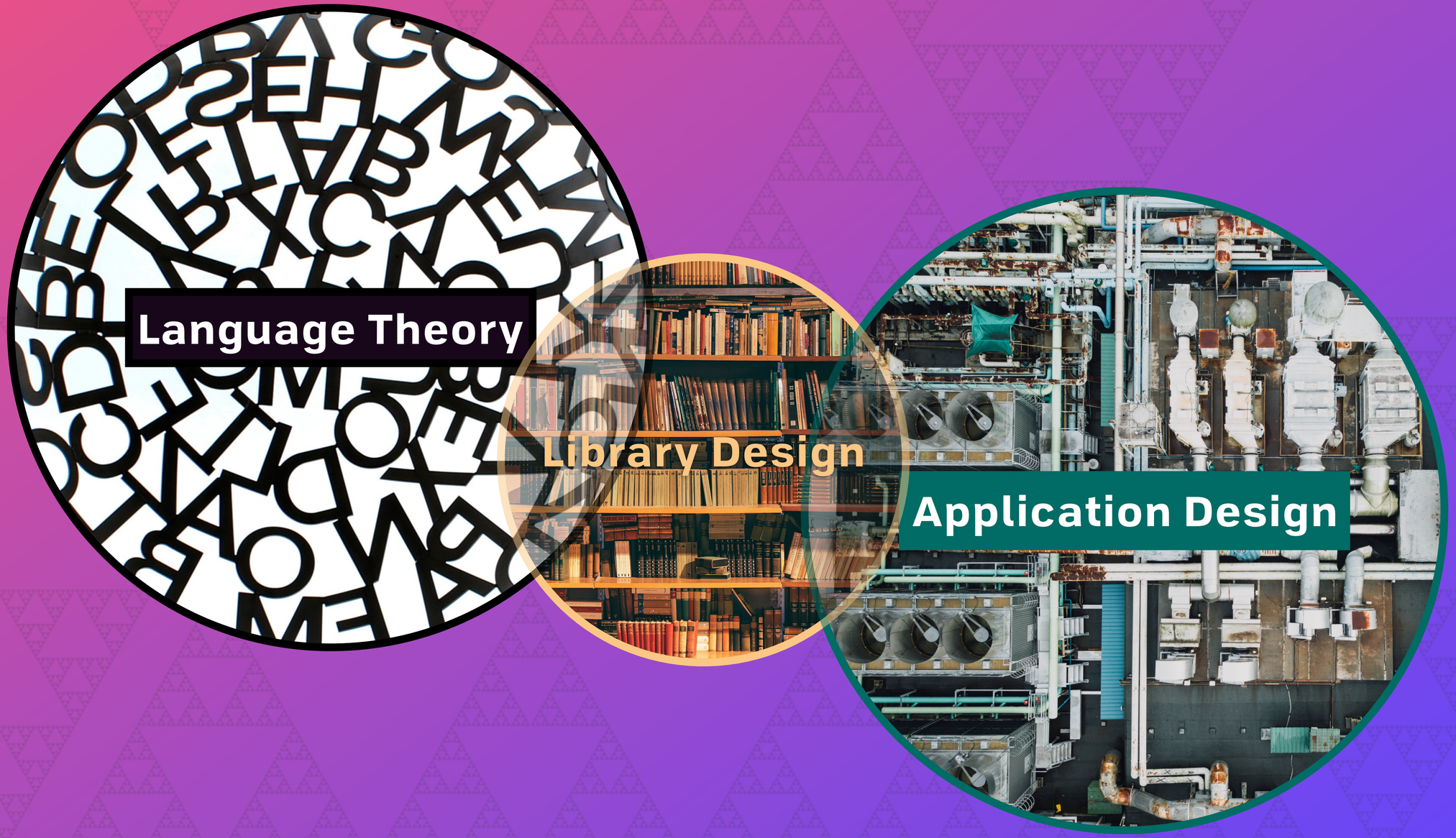
META

STRUCTURE 🏛️

- Top-down: general to concrete
  1. Modes of Thought
  2. Library Principles
  3. Library Pragmatics
- Goals
  - Turn intuitive design into explicit choices
  - Learn from my ~~mistakes~~ experiences!

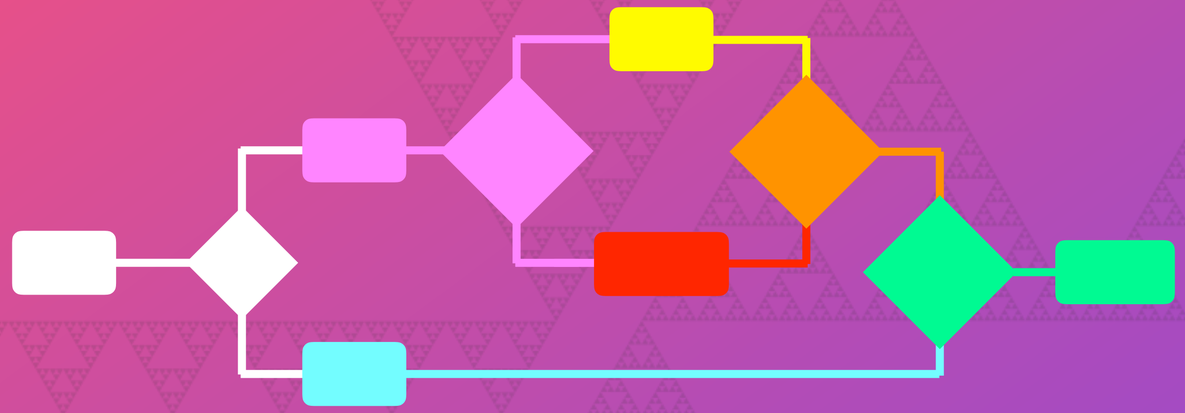
META

WHY FOCUS ON LIBRARIES? 📚



META

HIGH LEVEL CODE ILLUSTRATIONS ONLY



```
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)                     |
                                                                   | |
```

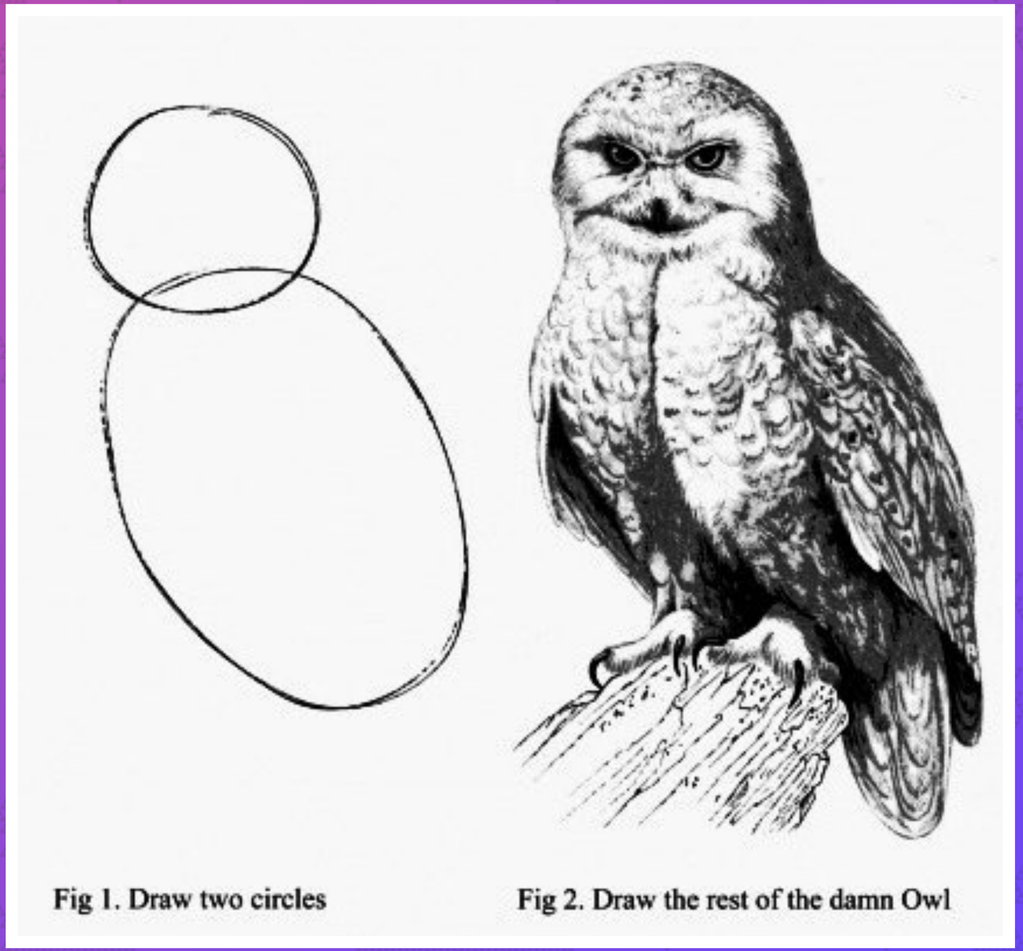


Fig 1. Draw two circles

Fig 2. Draw the rest of the damn Owl

PART I

MODES OF THOUGHT

SETTING THE STAGE 🎭

The limits of my **language** mean the limits of my **world**

LUDWIG WITTGENSTEIN

MODES OF THOUGHT

TOOLS > BIOLOGY ⛺🔥🚀

- Possibly use our minds *less than our ancestors*
- Better mental tools = lower cognitive load 🏹🔥
  - Metric > Imperial
  - Arabic numerals > Roman numerals
  - 24-hours and 360-degrees have nice divisors

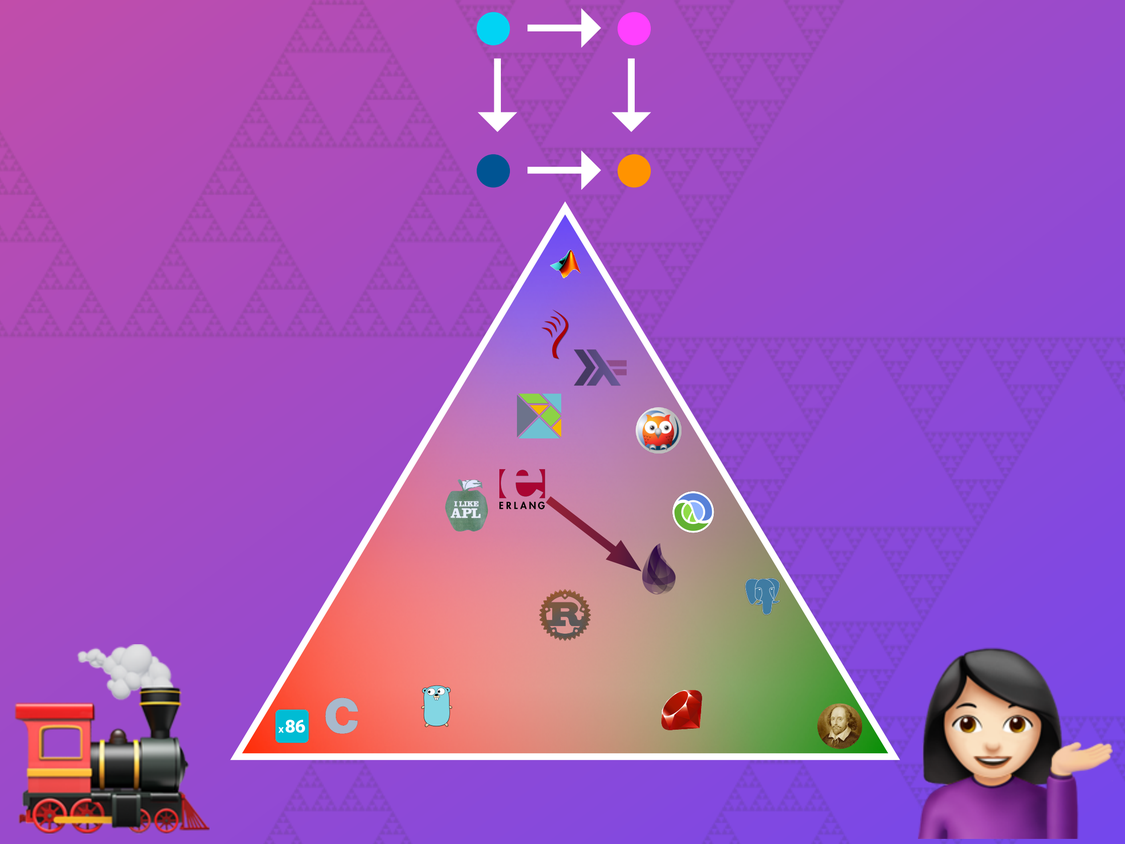
Flavours 🍨🍧

1. Algorithms ("how")
   - 🖥️ Much of undergrad CS
   - ⚙️ Mechanistic
   - ➗ e.g. Long division
2. Abstraction ("what")
   - 👋 A lot of language & UX
   - 📐 e.g. Space & geometry
   - 💭 e.g. Analogy & metaphor
   - 📚 e.g. Narrative

MODES OF THOUGHT

MODES OF THOUGHT ⚙️💭

- Operational / Mechanistic
- Denotational / Equational
- Humanist / Linguistic / HCI



MODES OF THOUGHT

WHAT'S SO **GREAT** ABOUT FP ANYWAY? 🥳

- Modularity
- Composition
- Can describe general solutions via abstraction!
- Make the large feel like the small & fewer patterns to memorize 🙌
- Maleable code

MODES OF THOUGHT

OPERATIONAL VS DENOTATIONAL THINKING

- `Enum.map/2` can be seen through the lens of…
  - Collections
    - A cleaner `for` loop (operational)
    - Run a function on every item in a structure
  - Programs
    - Transforms one program into a new one 😐

MODES OF THOUGHT

FUNCTIONAL ALCHEMY 

```elixir
@spec long_form(integer()) :: String.t()
def long_form(int) do
  case int do
    0 -> "zero"
    1 -> "one"
    2 -> "two"
    #...
```

```
@spec long_form(integer()) :: String.t()
```

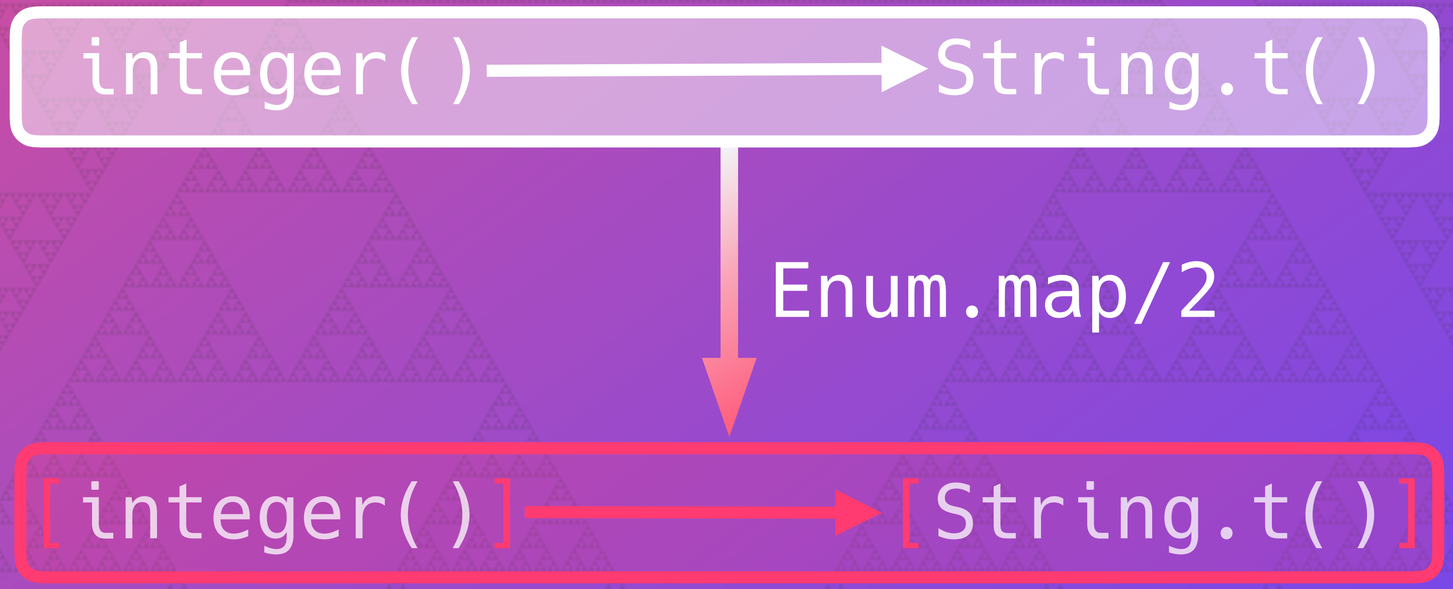
Enum.map/2

```
@spec long_forms([integer()]) :: [String.t()]
```

MODES OF THOUGHT

FUNCTIONAL ALCHEMY ⚗️

```elixir
@spec long_form(integer()) :: String.t()
def long_form(int) do
  case int do
    0 -> "zero"
    1 -> "one"
    2 -> "two"
    #...
```



MODES OF THOUGHT

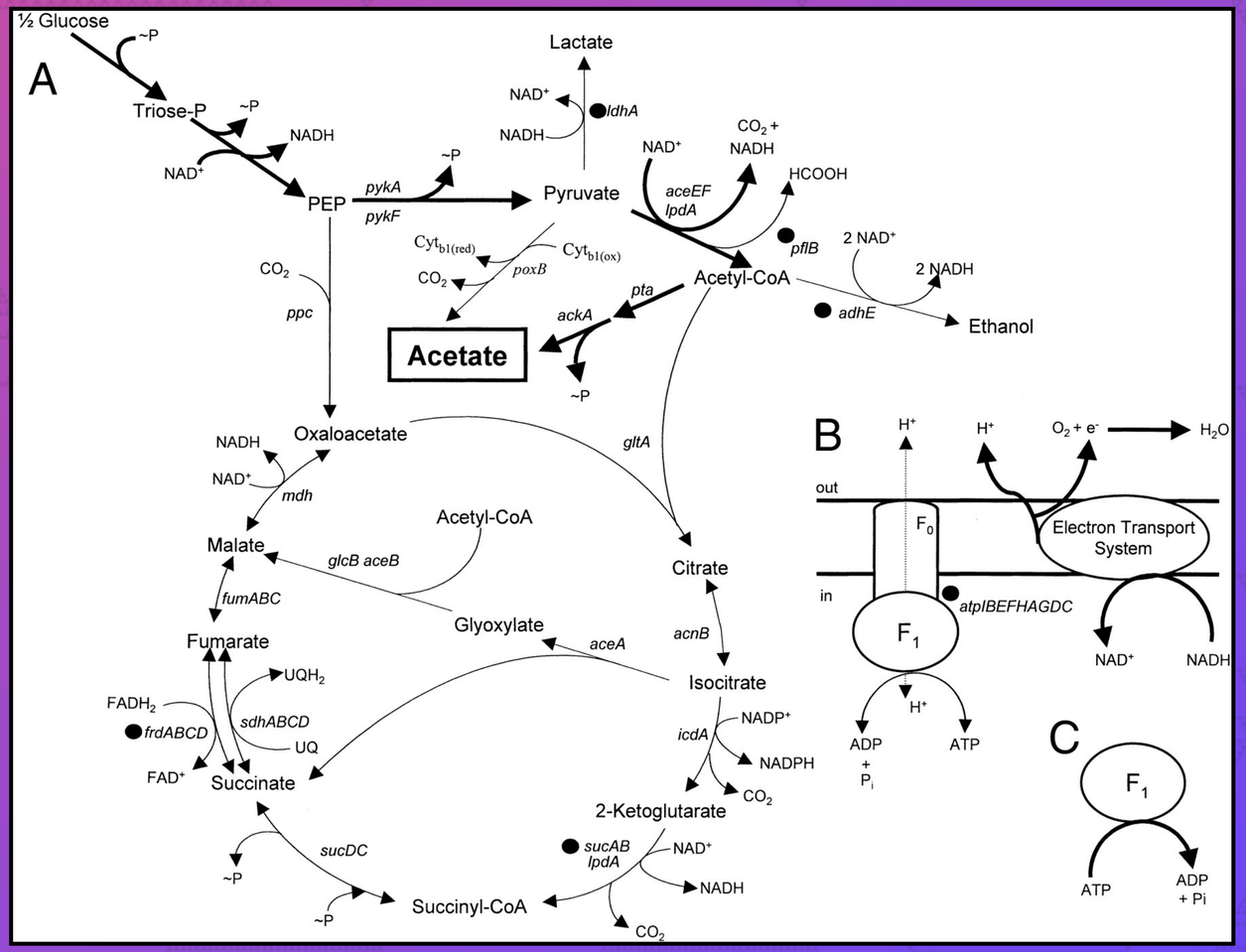
WHAT'S SO **TERRIBLE** ABOUT FP ANYWAY? 😱

- The Church Chasm
  - If you have a hammer, everything looks like a nail
  - "Use anonymous functions for *everything*! Aren't closures amazing?!"
  - Can focus on mechanics rather than concepts
  - Explicit state = lots of explicit handling, "pass the world", &c
- A **tendency** towards cleverness

MODES OF THOUGHT

ACTOR ABYSS

- Each step is very simple
- Reasoning about dynamic organisms is **hard**
  - *Remember* to store your data for crash recovery
  - Called collaborator may not be there
- Exactly why frameworks & `GenStage` are great!
  - Abstract this all away for many use cases
  - More concrete than arrows & better matches the Elixir ethos
- Complexity grows faster than linear
- **Takeaway**: provide structured abstractions



MODES OF THOUGHT

{:tarpit, “tradeoffs”}

- Surface simplicity has a cost
  - Complex in the large
  - Restricted domain in the small

PART II

LIBRARY PRINCIPLES 📚

LIB LONG AND PROSPER 🖖

The utility of a language as a tool of thought
increases with the **range of topics** it can treat,
but **decreases** with the amount of vocabulary
and the **complexity** of grammatical rules
which the user must keep in mind

KENNETH IVERSON

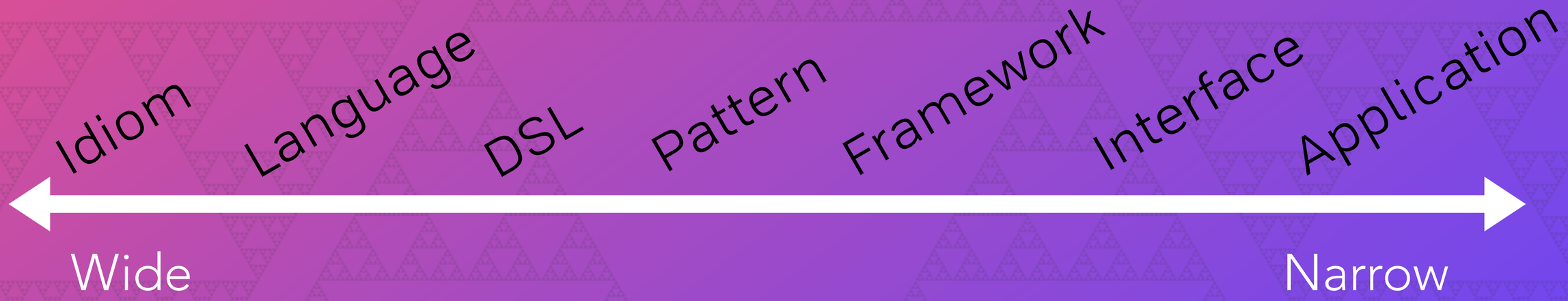
LIBRARY PRINCIPLES 📚

CLASSIFICATION 🔬

Category	Example
Data Representation / Interface	Poison, ExAws
Domain Specific Language (DSL)	Ecto, Algae, Absinthe
Reusable Patterns	Enum, Quark, Exceptional, Witchcraft
Feature Emulation	TypeClass, Credo, Dialyxir
Framework / System	Phoenix, Nerves

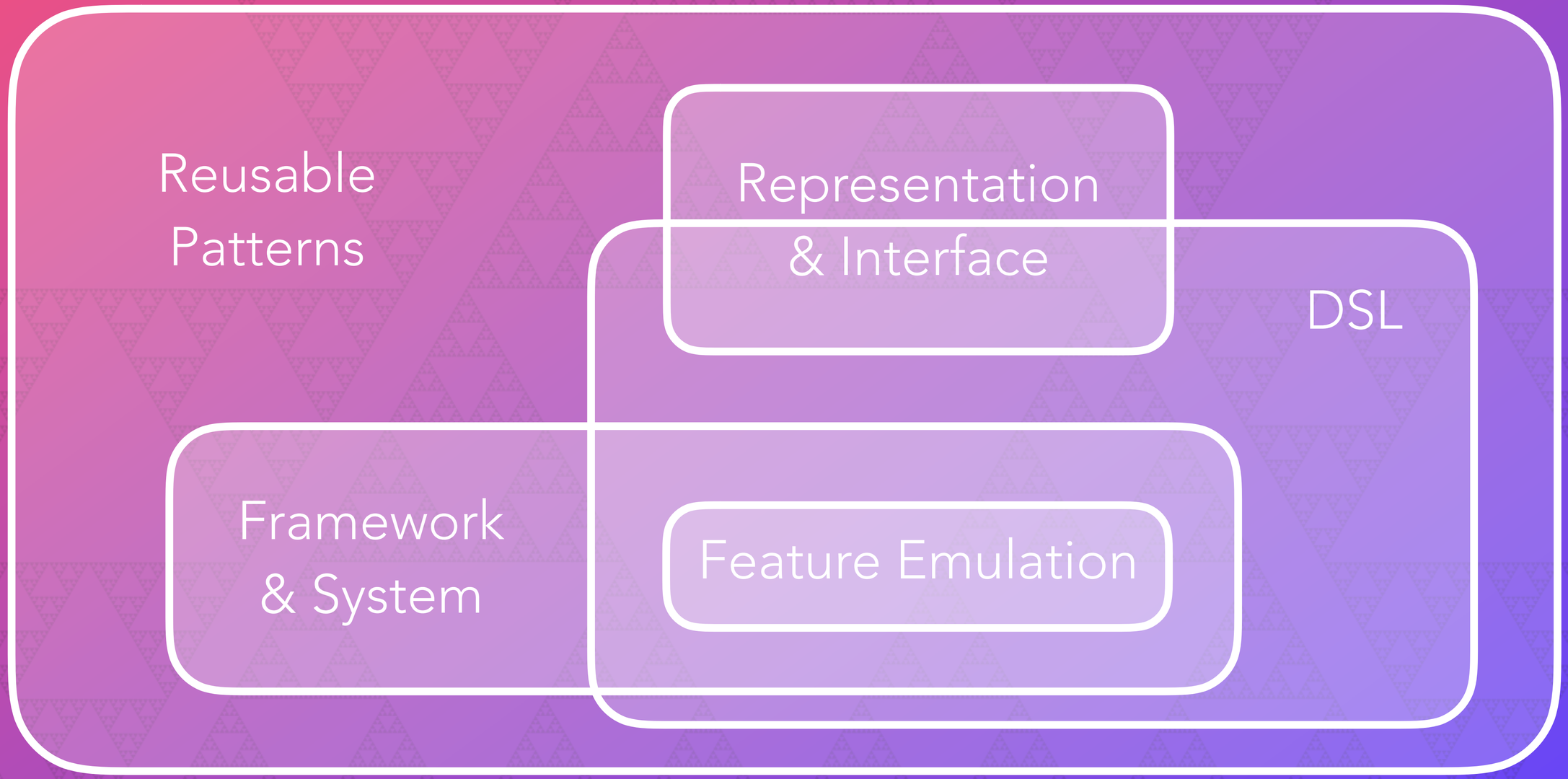
LIBRARY PRINCIPLES 📚

SPECTRUM 🌈



LIBRARY PRINCIPLES 📚

RELATIONSHIPS 👭



LIBRARY PRINCIPLES 📚

RELATIONSHIPS 👭

Reusable
Patterns

DSL

Feature Emulation

LIBRARY PRINCIPLES 📚

I, LIBRARY 🤖

- Ethos
- Generality
- Orthogonality
- Independence
- Completeness
- Extensibility

LIBRARY PRINCIPLES 📚

ETHOS

THE **WISDOM** TO BALANCE CONSISTENCY, CHANGE, AND RESTRAINT 📚

A library must remain **consistent**
with the character of the host language
and broadest community standards.

LIBRARY PRINCIPLES 📚 ETHOS

CHARACTER MOTIVATION 🥕

- Two styles of Scala
  1. Java without the braces
  2. Haskell fan fiction
- “I’d need to convert my entire project for this lib to make sense” 👎
- Seen in production with Ruby, Swift, JS, and more
- It doesn’t have to be this way 🚀

LIBRARY PRINCIPLES 📚 ETHOS

A MATTER OF CHARACTER

- **Ethos** — noun. (/ˈiːθɒs/ or US: /ˈiːθoʊs/)
  The fundamental character or spirit of a culture; the underlying sentiment that informs the beliefs, customs, or practices of a group or society; dominant assumptions of a people or period
- Like all interesting things, it fuzzy
- Feel as close to the host language as possible
- But also blending in features from a target

LIBRARY PRINCIPLES 📚 ETHOS

A “STRONG PERSONALITY” 🤨

- Strong character: C, Haskell, Rust, Elm, Golang
- Erlang designed for practicality (telecom switches)
- Elixir has no clear *language-level* raison d’être
- It’s like tofu: very easy to mix flavours in!

LIBRARY PRINCIPLES 📚 ETHOS

THE PIPES STRATEGY

- A break in ethos from Erlang
- Fits Elixir’s strategy
  - Clean
  - Easy to follow
  - Similar to OO’s fluent interfaces
- Helps to **tell a story**

- Tradeoffs
  - Operational/mechanistic mode of thought
    - Cleans up a lot of code
    - Focus or “zoom” effect 🔭
    - Zooming doesn’t really apply to actors
- **Takeaway**: consider you metaphors

LIBRARY PRINCIPLES 📚 ETHOS

CASE STUDY: `Exceptional`'S PIPES

- **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.()
```

EDGE CASE — GUEST LANGUAGE SYNTAXES

- Which language is this?
- Does it matter?

```
monad [] do
  a <- [1, 2, 3]
  b <- [4, 5, 6]
  return(a * b)
end
```

LIBRARY PRINCIPLES 📚

GENERALITY

THE **FLEXIBILITY** TO WORK ANYWHERE

Each entity must be useful
in as **many scenarios** as possible,
but **no more**.

LIBRARY PRINCIPLES 📚 GENERALITY

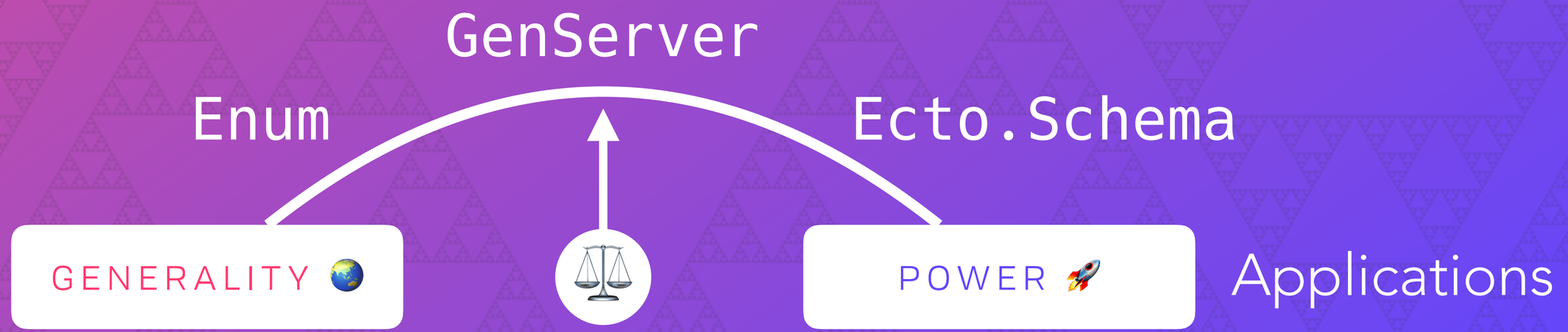
SWEET SPOTS 🍭

Generality

- Low information
- Few assumptions
- Many use cases

Power

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



LIBRARY PRINCIPLES 📚 GENERALITY

…TOO FAR? 😬

Witchcraft.Category

A category is some collection of objects and relationships (morphisms) between them.

This idea is captured by the idea of an identity function for objects, and the ability to compose relationships between objects. In most cases, these are very straightforward and composition and identity are the standard functions from the `Quark` package or similar.

Type Class

An instance of `Witchcraft.Category` must also implement `Witchcraft.Semigroupoid`, and define `Witchcraft.Category.identity/1`.

```
Semigroupoid  [compose/2, apply/2]
     ↓
 Category     [identity/1]
```

LIBRARY PRINCIPLES 📚

INDEPENDENCE

THE **COURAGE** TO VENTURE ALONE ⚔️

Each unit of code must be able to be **taken on its own**.
It must not break nor alter the runtime semantics
of either enclosed or exterior code.

INDEPENDENCE DAY 🛸

- Similar to referential transparency
  - Especially important with FP assumptions
- Each chunk of code (horizontal or vertical) should be comprehensible on its own
- **Most common issues**:
  - Behaviour depending on implicit or hidden state
  - Macro rewriting magic

LIBRARY PRINCIPLES 📚 INDEPENDENCE

CASE STUDY: Algae

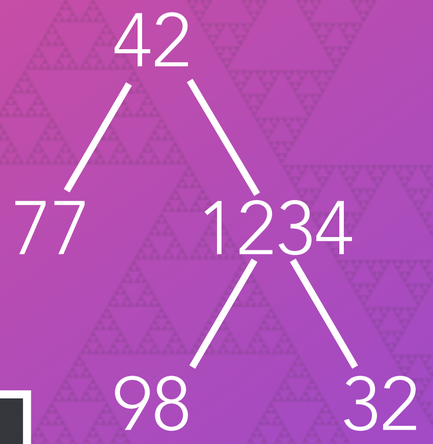
All the features!

```
defmodule Algae.Tree.BinarySearch do
  alias __MODULE__, as: BST

  defsum do
    defdata Empty :: none()

    defdata Node do
      node  :: any()
      left  :: BST.t() \\ Empty.new()
      right :: BST.t() \\ Empty.new()
    end
  end
end
```

LIBRARY PRINCIPLES 📚 INDEPENDENCE

CASE STUDY: Algae

All the features!




```elixir
BST.Node.new(
  42,
  BST.Node.new(77),
  BST.Node.new(
    1234,
    BST.Node.new(98),
    BST.Node.new(32)
  )
)
```

```elixir
defmodule Algae.Tree.BinarySearch do
  alias __MODULE__, as: BST

  defsum do
    defdata Empty :: none()

    defdata Node do
      node  :: any()
      left  :: BST.t() \\ Empty.new()
      right :: BST.t() \\ Empty.new()
    end
  end
end
```

```elixir
%Algae.Tree.BinarySearch.Node{
  node: 42,
  left: %Algae.Tree.BinarySearch.Node{
    node:  77,
    left:  %Algae.Tree.BinarySearch.Empty{},
    right: %Algae.Tree.BinarySearch.Empty{}
  },
  right: %Algae.Tree.BinarySearch.Node{
    node:  1234,
    left:  %Algae.Tree.BinarySearch.Node{
      node:  98,
      left:  %Algae.Tree.BinarySearch.Empty{},
      right: %Algae.Tree.BinarySearch.Empty{}
    },
    right: %Algae.Tree.BinarySearch.Node{
      node:  32,
      left:  %Algae.Tree.BinarySearch.Empty{},
      right: %Algae.Tree.BinarySearch.Empty{}
    }
  }
}
```

LIBRARY PRINCIPLES 📚 INDEPENDENCE

CASE STUDY: `Algae`

- **Concept**: Bootstrap structs into ADTs
- **Takeaways:**
  - Composition
  - Orthogonality
  - Structured abstraction
  - Turns into vanilla Elixir
- Removes
  - Nothing
- Adds
  - Structure DSL
  - Types with automatic default values
  - Manual default values
  - Constructor helpers (e.g. `Foo.new`)

LIBRARY PRINCIPLES 📚

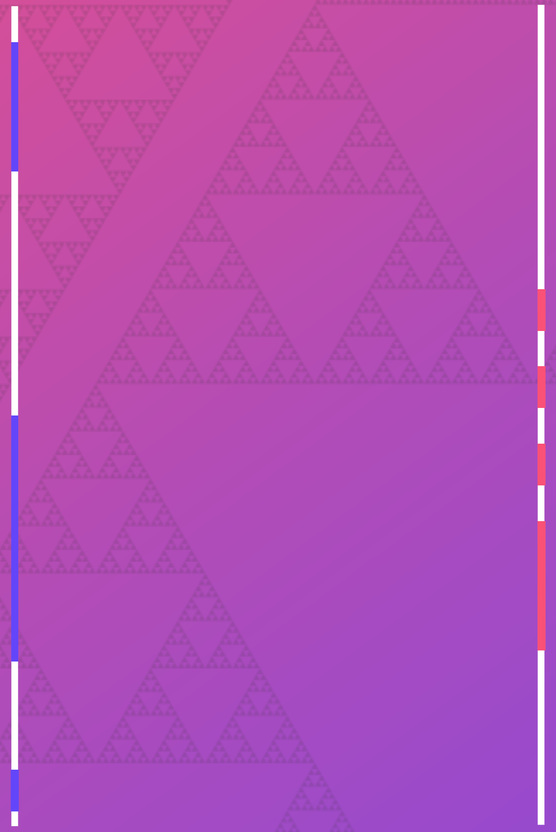
ORTHOGONALITY

THE **ZEN** OF UNDERSTATEMENT

Each entity must be **the best at a specific task**.
It must **not conflict** with other functions.

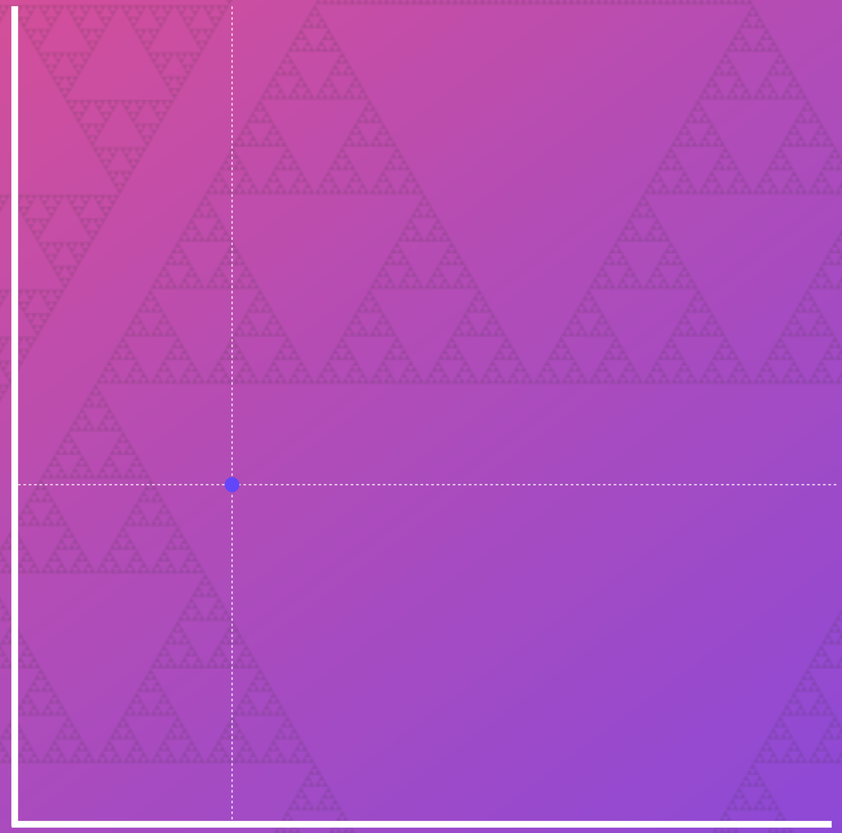
LIBRARY PRINCIPLES 📚 ORTHOGONALITY

GEOMETRIC METAPHOR

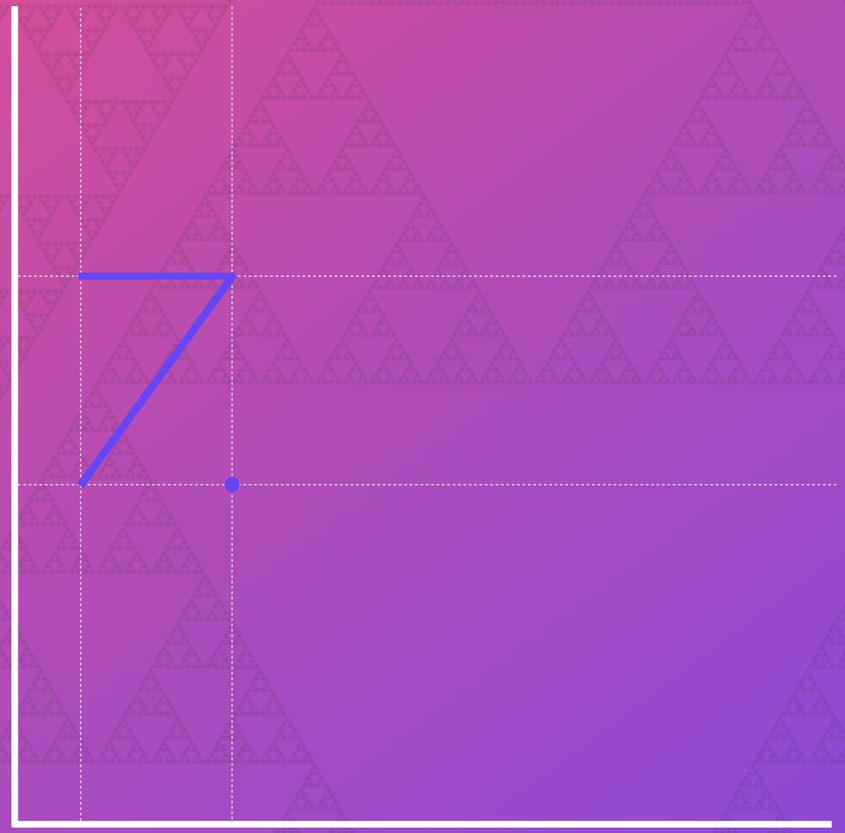


GEOMETRIC METAPHOR

GEOMETRIC METAPHOR



GEOMETRIC METAPHOR

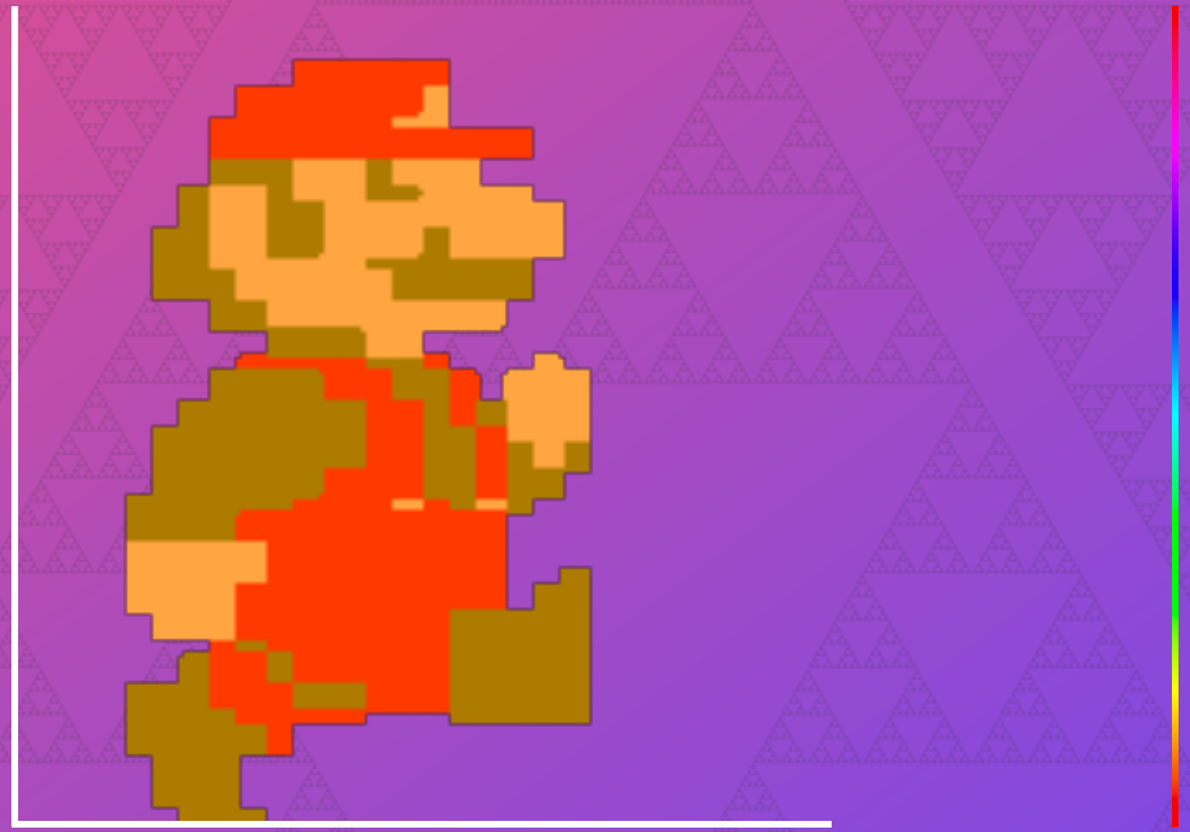


LIBRARY PRINCIPLES 📚 ORTHOGONALITY

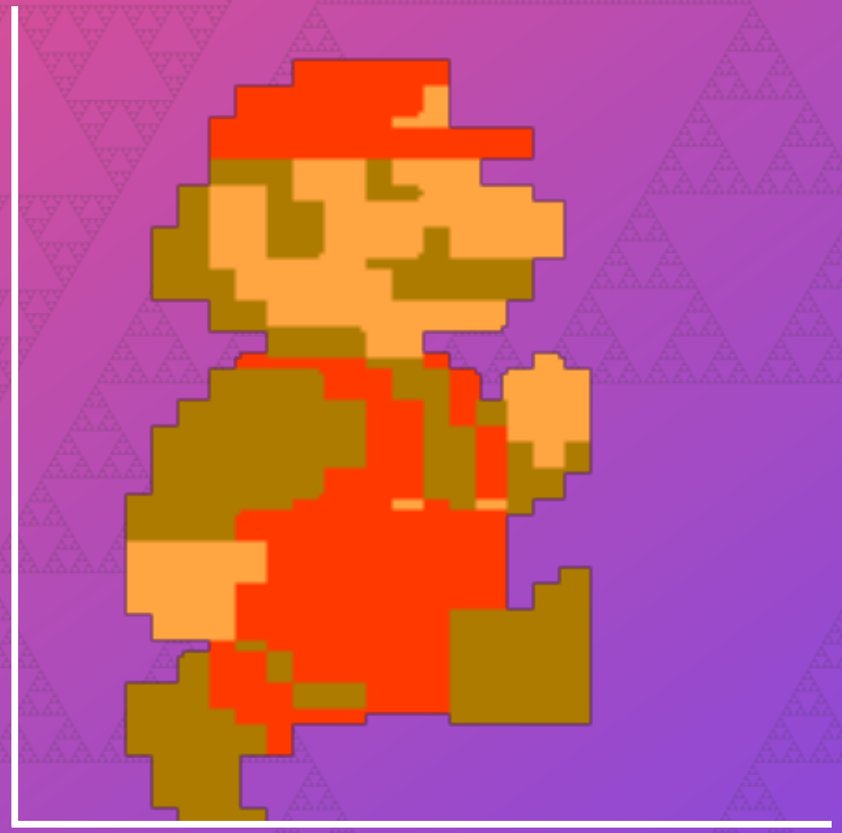
GEOMETRIC METAPHOR

GEOMETRIC METAPHOR

GEOMETRIC METAPHOR

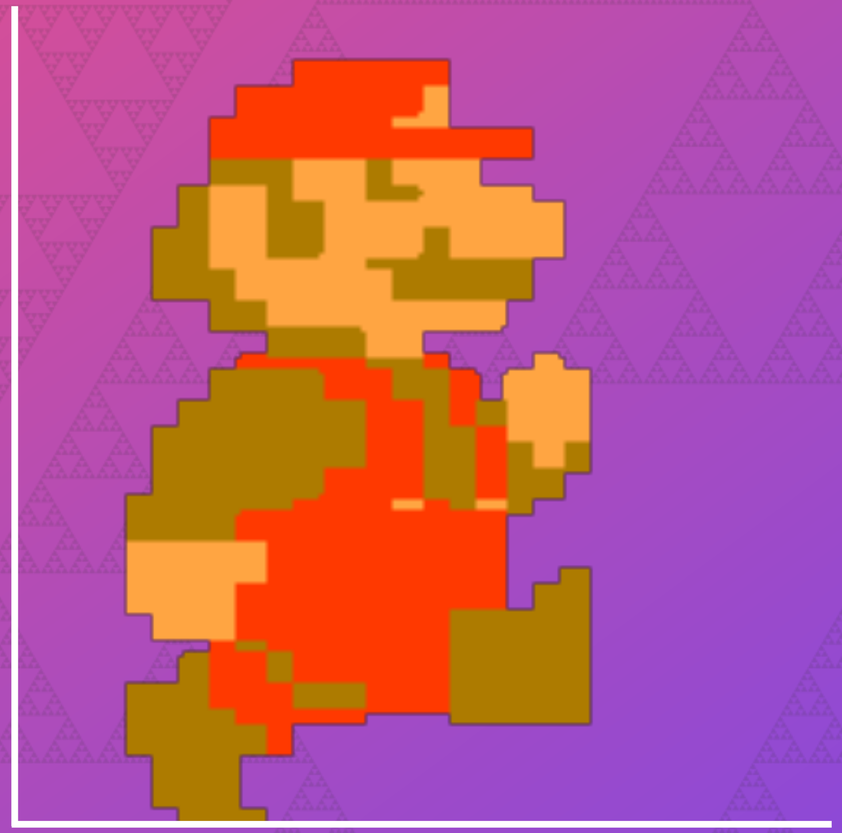


GEOMETRIC METAPHOR



LIBRARY PRINCIPLES 📚 ORTHOGONALITY

GEOMETRIC METAPHOR



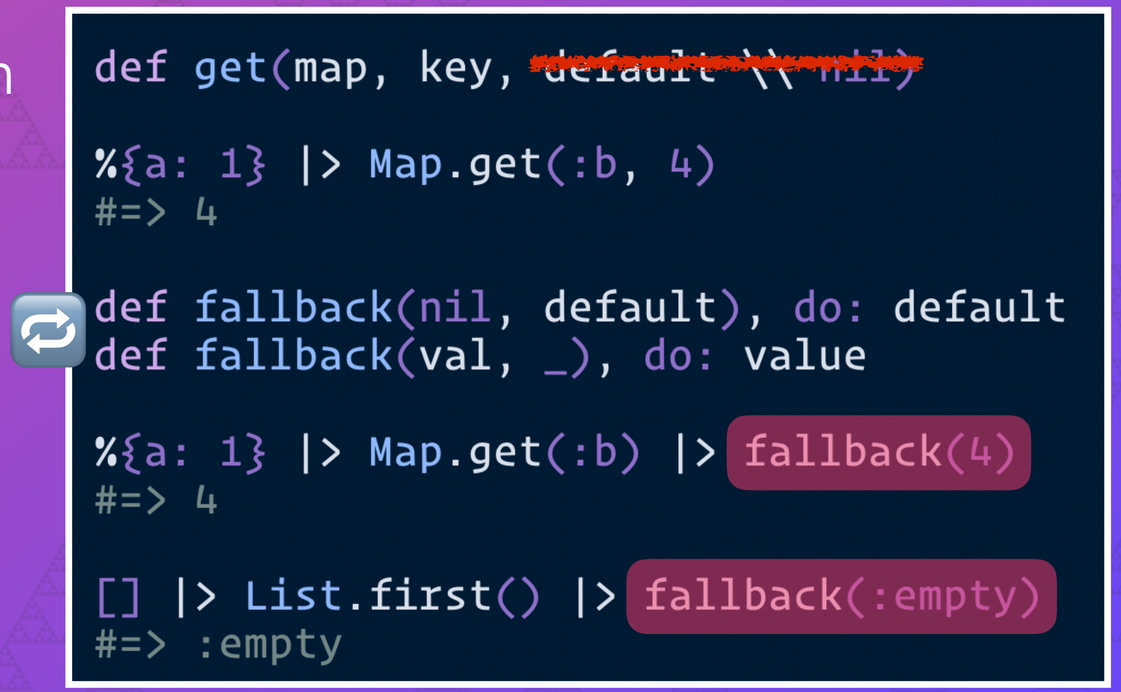
Components: 4

Results: effectively limitless

LIBRARY PRINCIPLES 📚 ORTHOGONALITY

MORE THAN THE SUM OF ITS PARTS 🧩

- 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
- Can we take this further?



LIBRARY PRINCIPLES 📚 ORTHOGONALITY

AN **Exceptional** EXAMPLE

Abstracted out `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 error struct

Any flow (esp. pipes)

Super easy to test

BONUS

Disambiguate between `nil` value and actual errors

LIBRARY PRINCIPLES 📚

COMPLETENESS

THE **POWER** OF HARMONIOUS TOOLS 💪

A library must introduce
or integrate its concept(s)
as **completely as possible**.

LIBRARY PRINCIPLES 📚 COMPLETENESS

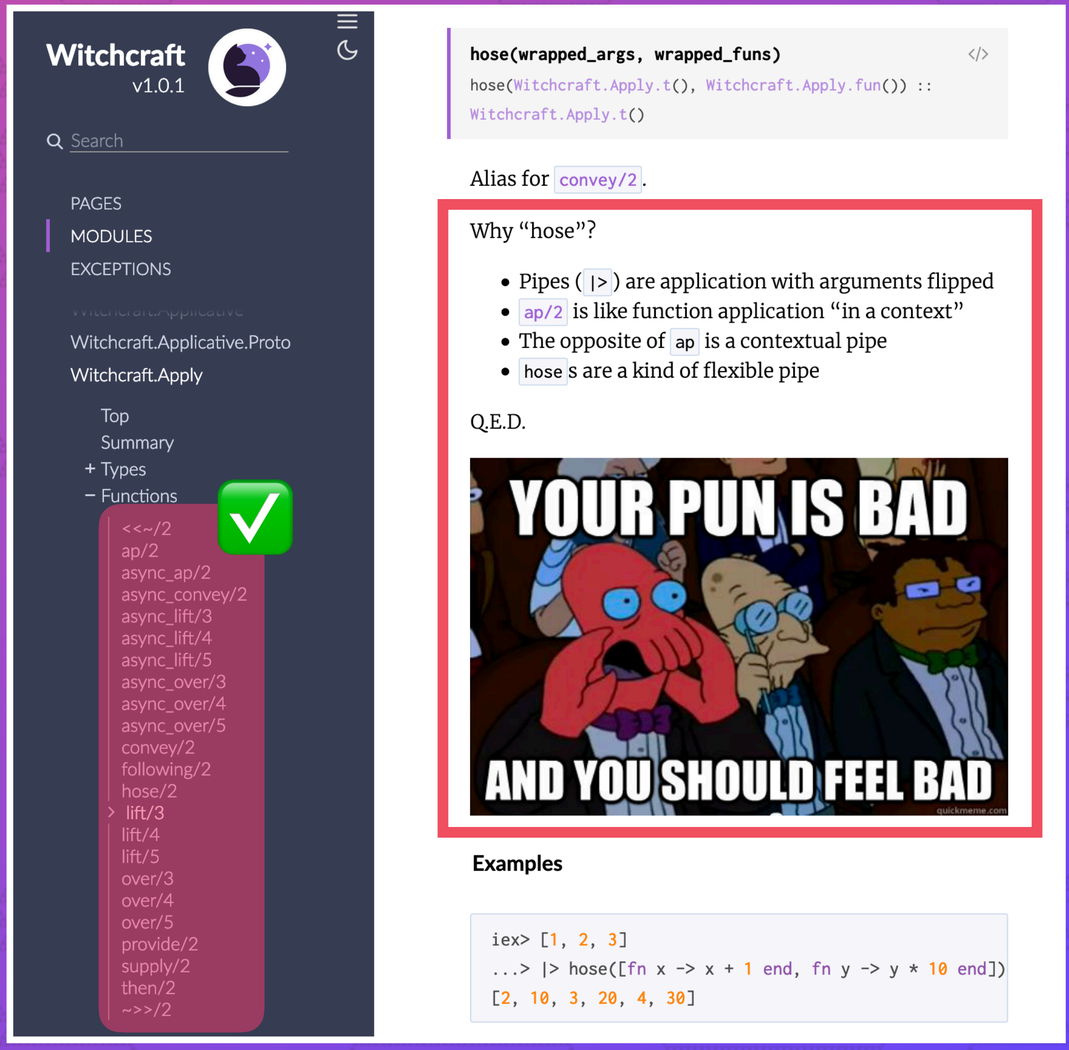
CASE STUDY: `Witchcraft`

- Can go **pretty far** with completeness
- Update, clean, and power up `Kernel` functions variants
  - Curried, async, flipped
  - What about orthogonality? 😭

LIBRARY PRINCIPLES 📚 COMPLETENESS

KEEPIN' IT WITCHY 🧙

- Ported things with immediate synergy
- Renamed many functions for clarity
  - Sometimes very difficult
- Operator `>>>` aliases
- Changed **lots** of argument orders
  - `:pipable |> everything_is()`



LIBRARY PRINCIPLES 📚 COMPLETENESS

KEEPIN' IT WITCHY 🧙

- Ported things with immediate synergy
- Renamed many functions for clarity
  - Sometimes very difficult
- Operator `>>>` aliases
- Changed **lots** of argument orders
  - `:pipable |> everything_is()`

HASKELL PRELUDE	WITCHCRAFT
`flip ($)`	`|>/2` (`Kernel`)
`.`	`<|>/2`
`<<<`	`<|>/2`
`>>>`	`<~>/2`
`<>`	`<>/2`
`<$>`	`<~/2`
`flip (<$>)`	`~>/2`
`fmap`	`lift/2`
`liftA`	`lift/2`
`liftA2`	`lift/3`
`liftA3`	`lift/4`
`liftM`	`lift/2`
`liftM2`	`lift/3`
`liftM3`	`lift/4`
`ap`	`ap/2`
`<*>`	`<<~/2`
`<**>`	`~>>/2`
`*>`	`then/2`
`<*`	`following/2`
`pure`	`of/2`
`return`	`of/2`
`>>`	`then/2`
`>>=`	`>>>/2`
`=<<`	`<<</2`
`***`	`^^^/2`
`&&&`	`&&&/2`

LIBRARY PRINCIPLES 📚

EXTENSIBILITY

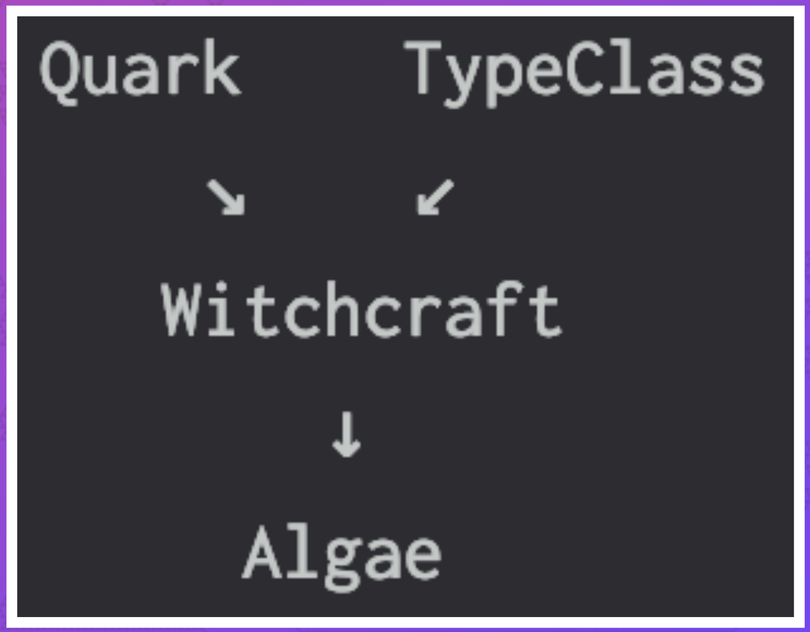
THE **FREEDOM** TO DO MORE

A library should provide the ability
to **adapt its functionality to new contexts**
without altering the library itself.

LIBRARY PRINCIPLES 📚 EXTENSIBILITY

LIBRARY-LEVEL EXTENSION

- `Witchcraft` lib hierarchy was a very good idea
- Wish I had gone further
  - e.g. `Algae`'s DSL vs instances
- Balance how much each lib should do
- Extension as usage example
- Market as more reusable 😉



PART III

LIBRARY PRAGMATICS

FAILURE MODES & LOW HANGING FRUIT 🥝

The art of programming is the art
of **organizing complexity**,
of **mastering multitude**,
and **avoiding** its bastard **chaos**
as effectively as possible

EDSGER DIJKSTRA

LIBRARY PRAGMATICS

YOUR INNER MAD SCIENTIST 👩‍🔬

If you feel like this

you've probably created a monster

LIBRARY PRAGMATICS

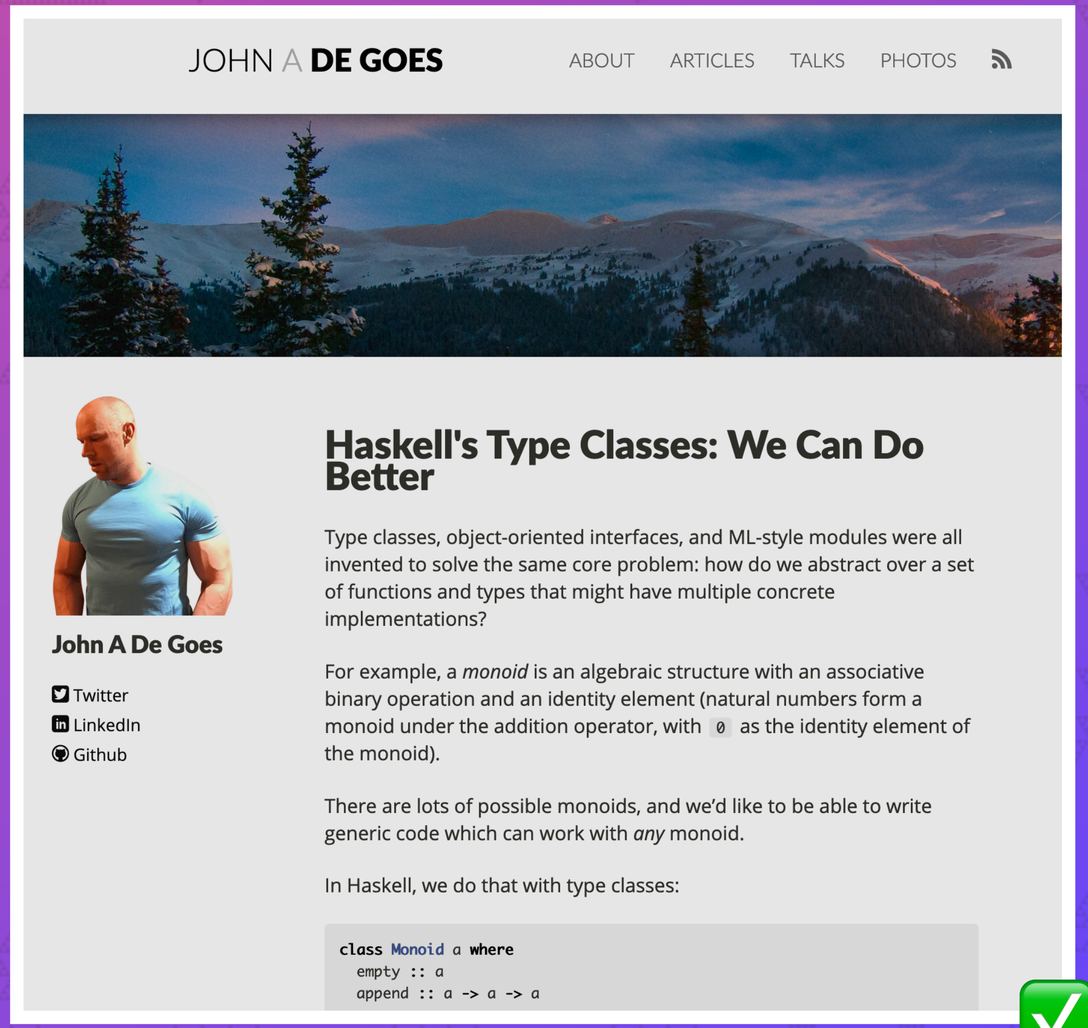
MIND TRAPS 🕳️🧠

- More dangerous in libraries than application code
  - Great definitional power comes with great responsibility to your users!
- Identify the way you tend to fail
  - e.g. My failure mode is to *enforce* correctness on others
- KISS principle 😘
  - Common trap: getting too cute with function names and metaphors
  - Names should be descriptive of what they do, not just be clever
- Other traps include: no tests, incomplete scope, not general enough, &c

LIBRARY PRAGMATICS

DO NOT FORCE PRACTICES ON USERS 🙅

- Overall very happy with `TypeClass`
  - But contains my biggest regret 😖
- Enforced prop testing at compile time
  - Slow
  - Detracts people from using the lib 😭
- Wanted to encourage good practice
- Please give user choice
- Generate test helpers



✅

LIBRARY PRAGMATICS

ALL THE OPERATORS 🤔🙅‍♀️✅

- Super duper handy to have 🎉
- Limited number of operators in Elixir
- Operator-space collisions 💥
- Handy if you're user has one lib per application, but you don't know
- Always provide a named variant
  - `Operator` 😉

```
@operator :<~>
def multiply(a, b), do: a * b
```

LIBRARY PRAGMATICS

THE *OPTION* TO DISAMBIGUATE ✅

```
foo
~> fn(x) -> x * 10 end
~> &IO.inspect/1
```

What does this do?

```
foo
|> Functor.lift(fn(x) -> x * 10 end)
|> Functor.lift(IO.inspect/1)
```

```
foo
|> exception_or_continue(fn(x) -> x * 10 end)
|> exception_or_continue(IO.inspect/1)
```

LIBRARY PRAGMATICS

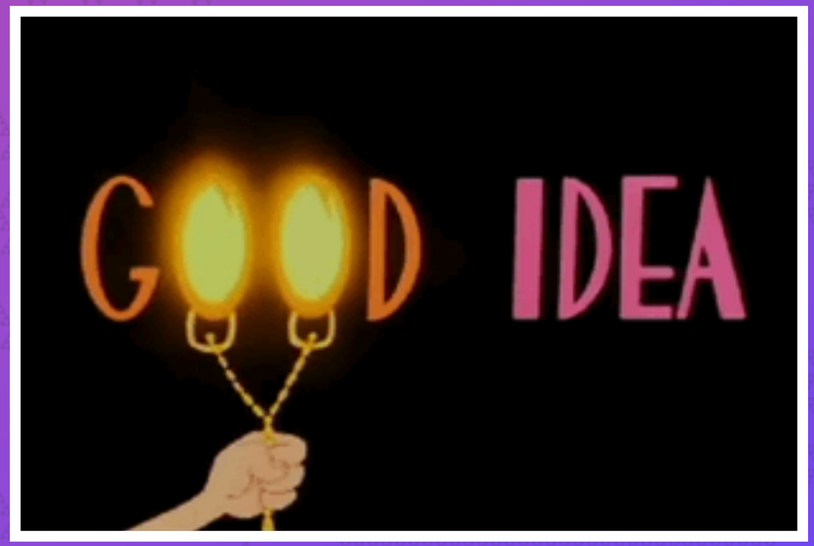
FAKE AS MUCH AS POSSIBLE

- Started out by wanting to do everything by hand 🤦
- Save yourself and others the trouble:
  - Excellent idea to just fake things by bootstrapping
  - Vanilla Elixir!
- `Algae`: “just” a vanilla struct DSL
- `TypeClass`: “just” vanilla protocols (plus some enforced checking)
- **I repeat**: it’s all vanilla Elixir underneath the hood

LIBRARY PRAGMATICS

SMOKE & MIRRORS 🧙

- Leverage what's already there!
  - Stays compatible with entire ecosystem
  - Less work for you 😉



LIBRARY PRAGMATICS

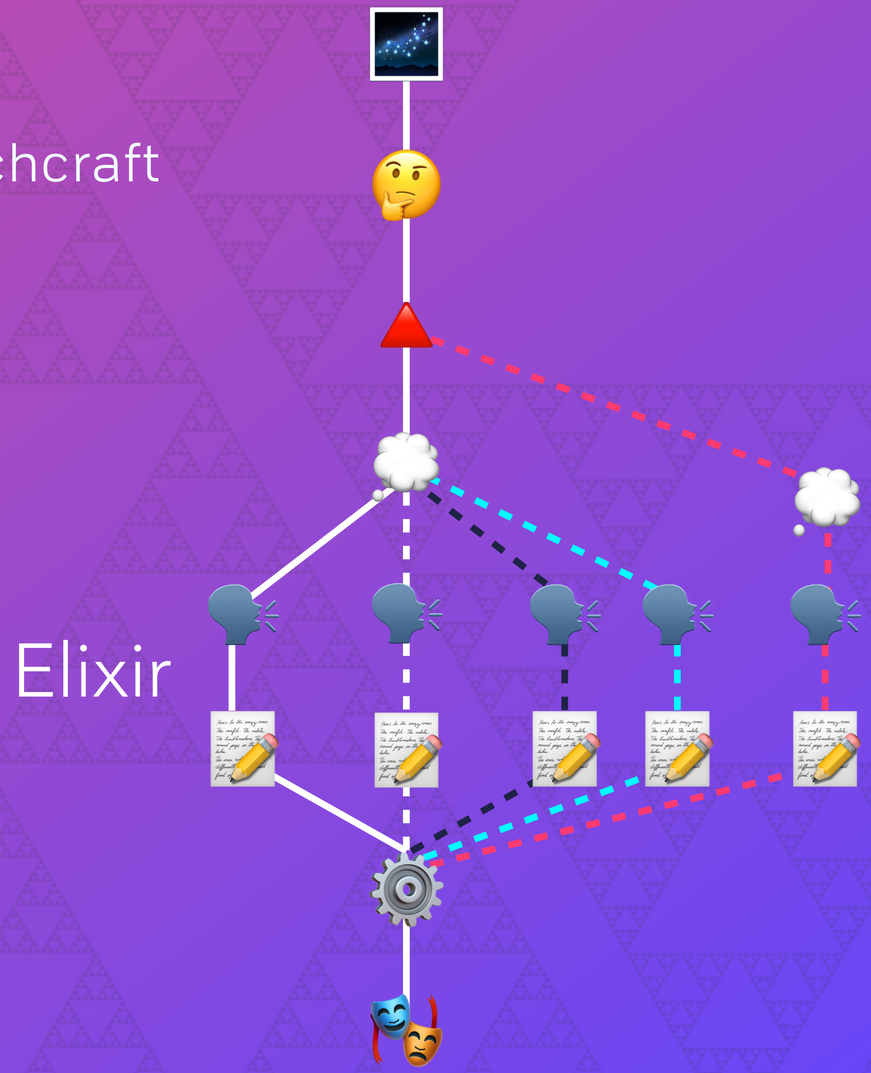
GREAT ARTISTS STEAL 👩‍🎨

- Don’t reinvent the wheel: steal!
- Cat Theory → Haskell & PureScript & Elm &c → Witchcraft
- Clean up the type class hierarchy
  - Haskell
  - Purescript
  - Fantasy Land Specification
  - and others
- Some pipings and naming
  - Elm

LIBRARY PRAGMATICS

GREAT ARTISTS STEAL 👩‍🎨

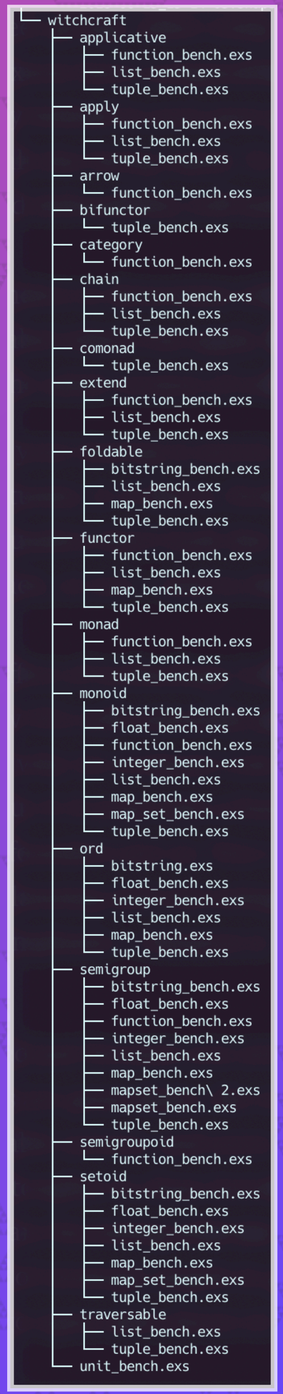
- Don’t reinvent the wheel: steal!
- Cat Theory → Haskell & PureScript & Elm &c → Witchcraft
- Clean up the type class hierarchy
  - Haskell
  - Purescript
  - Fantasy Land Specification
  - and others
- Some pipings and naming
  - Elm



LIBRARY PRAGMATICS

BENCH ALL THE THINGS 🏋️

- Sometimes truly surprising results (both good and bad)
- Bench different data types and **sizes**
- Run analyses against application-specific functions or `Kernel`
  - TL;DR `Kernel` functions are really fast!
    - Optimized & battle tested
    - Use in `defimpl`s whenever possible



LIBRARY PRAGMATICS

TEXT DIAGRAMS > IMAGES > ONLY WORDS > NOTHING

Witchcraft.Monad

Very similar to `Chain`, `Monad` provides a way to link actions, and a way to bring plain values into the correct context (`Applicative`).

This allows us to view actions in a full framework along the lines of functor and applicative:

```
          data ---------------- function ----------------------------> result
            |                     |                                      |
 of(Container, data)          of/2, or similar                 of(Container, result)
            ↓                     ↓                                      ↓
%Container<data> --- (data -> %Container<updated_data>) ---> %Container<updated_data>
```

As you can see, the linking function may just be `of` now that we have that.

For a nice, illustrated introduction, see Functors, Applicatives, And Monads In Pictures.

Having `of` also lets us enhance do-notation with a convenuenct `return` function (see `monad/2`)

Type Class

An instance of `Witchcraft.Monad` must also implement `Witchcraft.Applicative` and `Wicthcraft.Chainable`.

```
                  Functor     [map/2]
                    ↓
                  Apply       [convey/2]
                  ↓   ↓
[of/2]  Applicative   Chain   [chain/2]
                  ↓   ↓
                  Monad
                   [_]
```

```
                                Witchcraft.Monad

Very similar to Chain, Monad provides a way to link actions, and a way to bring
plain values into the correct context (Applicative).

This allows us to view actions in a full framework along the lines of functor
and applicative:

              data ---------------- function ----------------------------> result
                |                     |                                      |
     of(Container, data)          of/2, or similar                 of(Container, result)
                ↓                     ↓                                      ↓
    %Container<data> --- (data -> %Container<updated_data>) ---> %Container<updated_data>

As you can see, the linking function may just be of now that we have that.

For a nice, illustrated introduction, see Functors, Applicatives, And Monads In
Pictures
(http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html).

Having of also lets us enhance do-notation with a convenuenct return function
(see monad/2)

## Type Class

An instance of Witchcraft.Monad must also implement Witchcraft.Applicative and
Wicthcraft.Chainable.

                    Functor     [map/2]
                      ↓
                    Apply       [convey/2]
                    ↓   ↓
    [of/2]  Applicative   Chain   [chain/2]
                    ↓   ↓
                    Monad
                     [_]
```

LIBRARY PRAGMATICS

PRODUCT THINKING: **ROI**

- People use libs because of they make their lives better
- Tradeoffs suck: have as few as possible
- Low effort / high return (for the user)
- Make it as easy as possible to set up
  - e.g. `use Exceptional`
  - Doesn’t conflict with other libs, `Kernel`, &c

FINAL THOUGHTS

🎁 WRAPPING UP 🤔

FINAL THOUGHTS

BIG THEMES

- Standard libs are not sacred
  - But all lib code should feel like it belongs there!
  - Just because you **can** doesn't mean you **should**
- Great artists steal
- Library-as-language-extension (think "version x.1")
- Focus on cohesion
- Love your code ❤️

THANK YOU, PRAGUE

HELLO@BROOKLYNZELENKA.COM
GITHUB.COM/EXPEDE
@EXPEDE