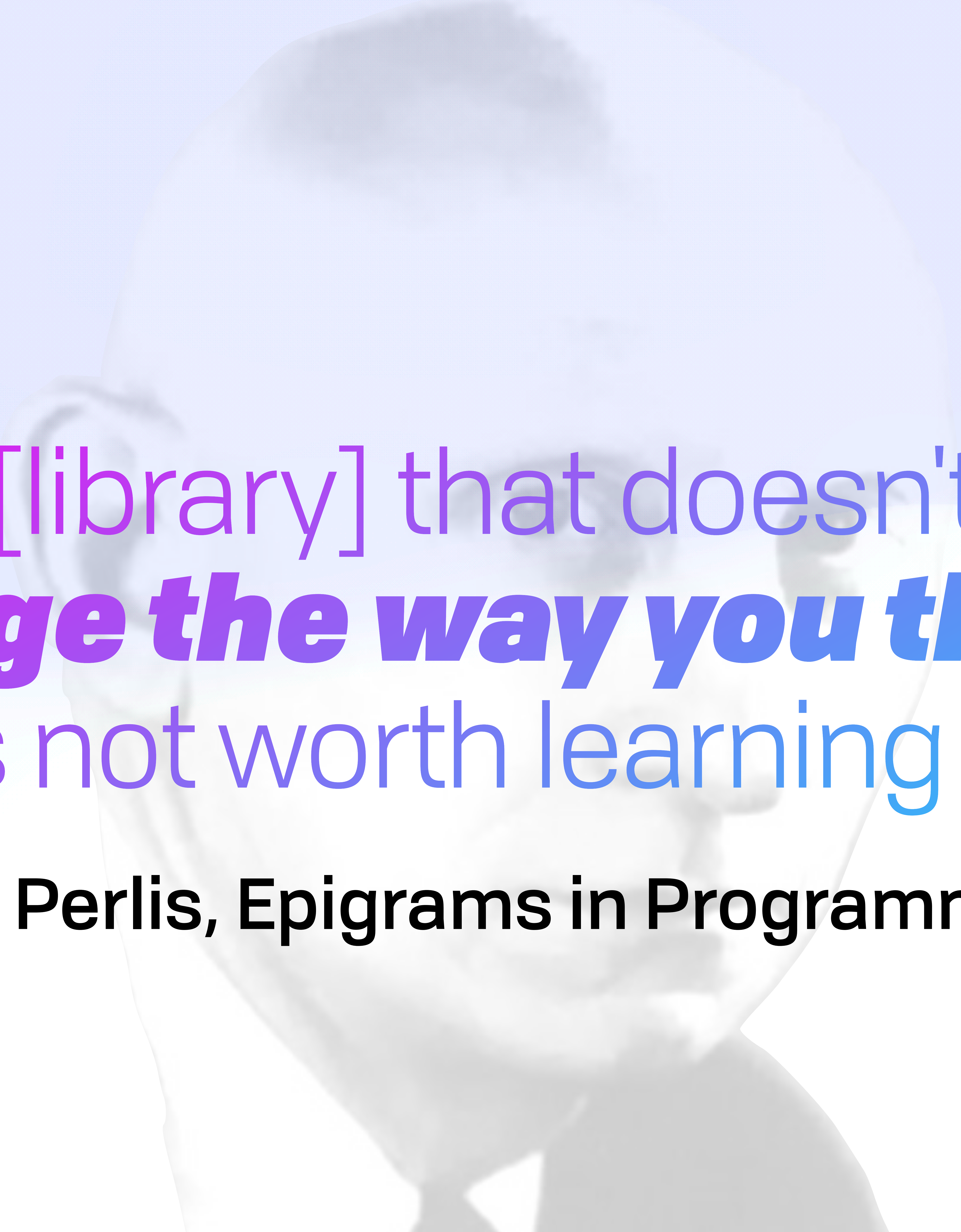
Witchcraft Retrospective

Lessons From Bringing Monads to Elixir



A [library] that doesn't ***change the way you think*** is not worth learning

— Alan Perlis, Epigrams in Programming

Witchcraft

Brooklyn Zelenka @expede

Witchcraft

Brooklyn Zelenka @expede

github.com/expede

Witchcraft

Brooklyn Zelenka @expede

- Indie researcher
- Local-First, P2P
    - Esp. E2EE, capabilities, CRDTs, VMs
- Author of Quark, Algae, Witchcraft / "Haskell Fan Fiction"
    - Exceptional and others
    - Had a ton of fun writing these libraries 🔮🙌💜
- Founded the Vancouver Functional Programming Meetup

github.com/expede

Witchcraft

Brooklyn Zelenka @expede

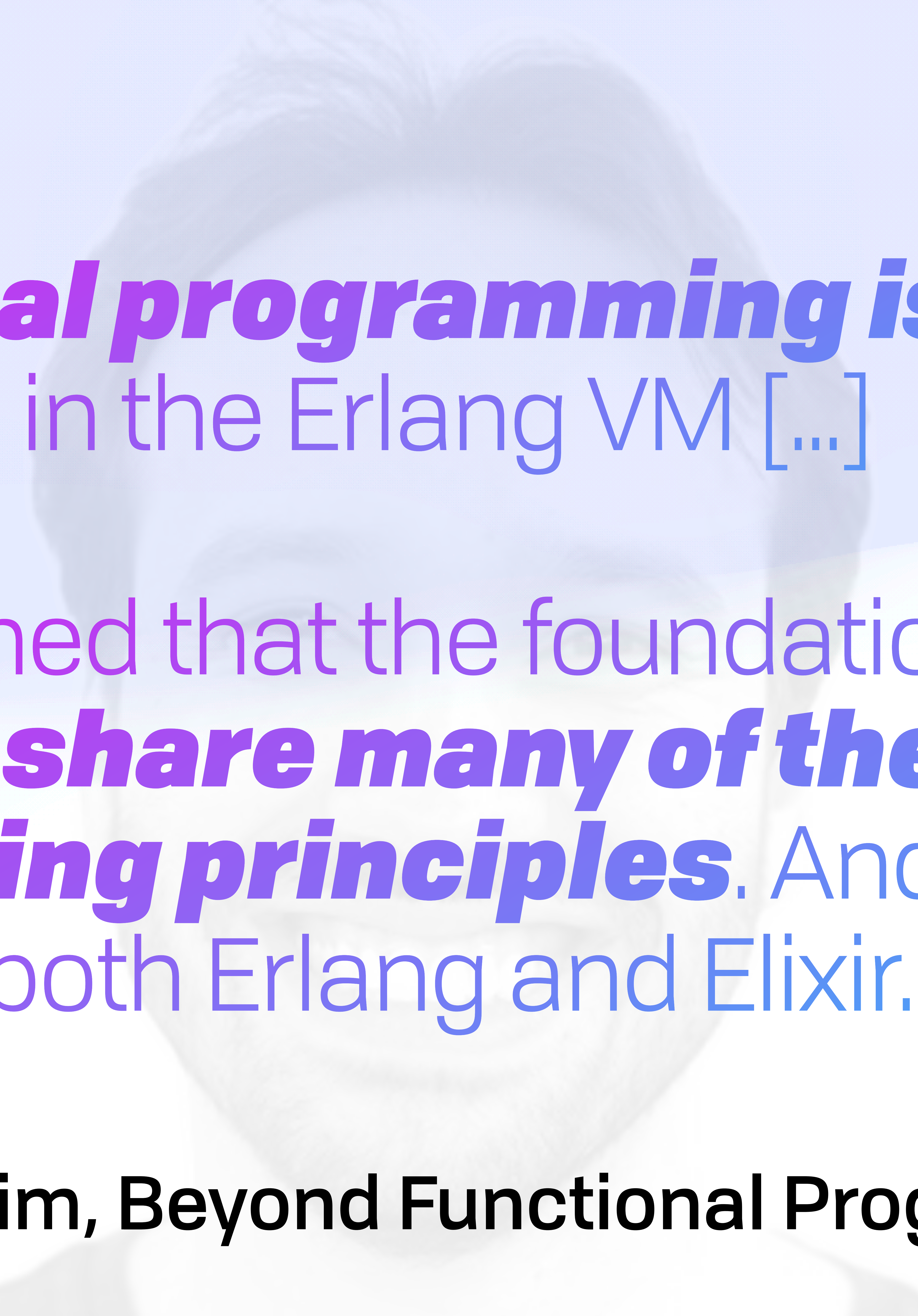
- Indie researcher
- Local-First, P2P
  - Esp. E2EE, capabilities, CRDTs, VMs
- Author of Quark, Algae, Witchcraft / "Haskell Fan Fiction"
  - Exceptional and others
  - Had a ton of fun writing these libraries 🔮🙌💜
- Founded the Vancouver Functional Programming Meetup

Vancouver
Functional
Programmers

github.com/expede

What is Witchcraft?

Strange Brew



> [...] ***functional programming is not a goal*** in the Erlang VM [...]
>
> It just happened that the foundation for writing such systems ***share many of the functional programming principles***. And it reflects in both Erlang and Elixir.

— José Valim, Beyond Functional Programming

[...] ***functional programming is not a goal***
in the Erlang VM [...]

It just happened that the foundation for writing such sys ***here many of the functional*** ***pro inciples***. And it reflects in rlang and Elixir.

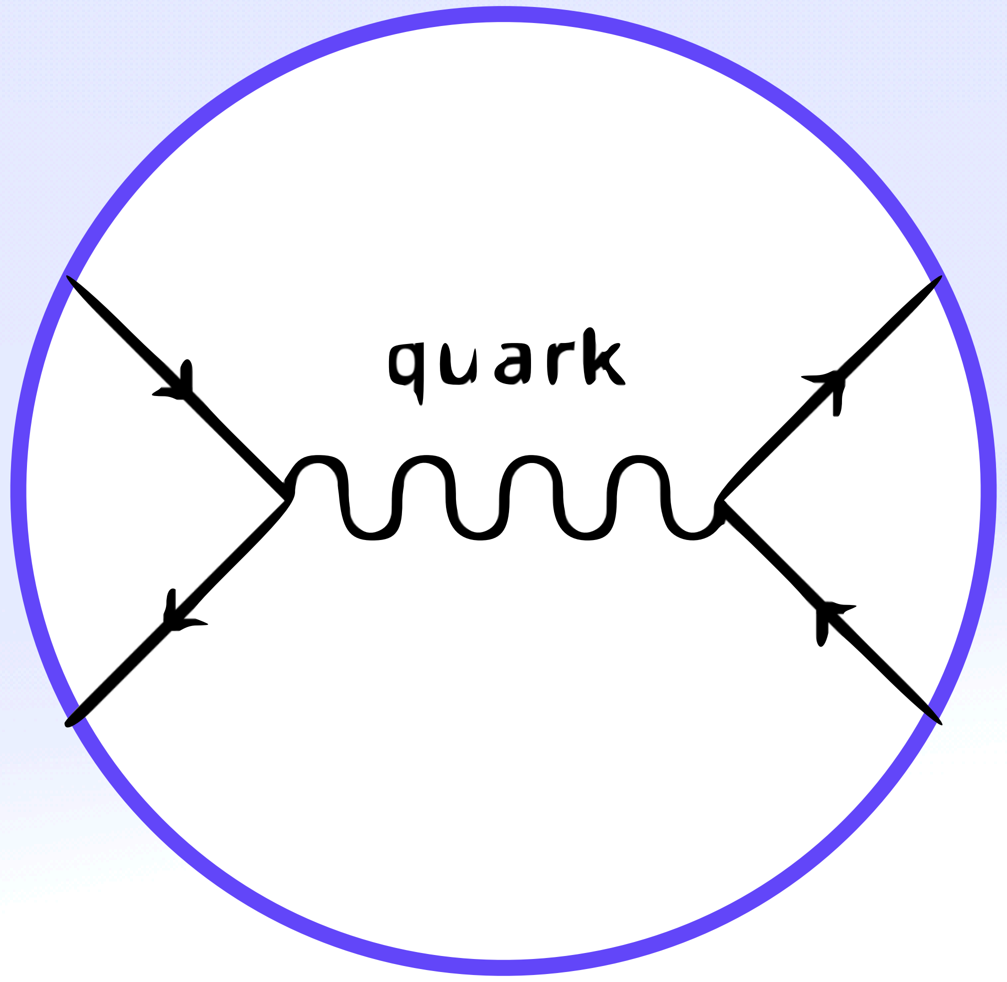
— José Valim, Beyond Functional Programming

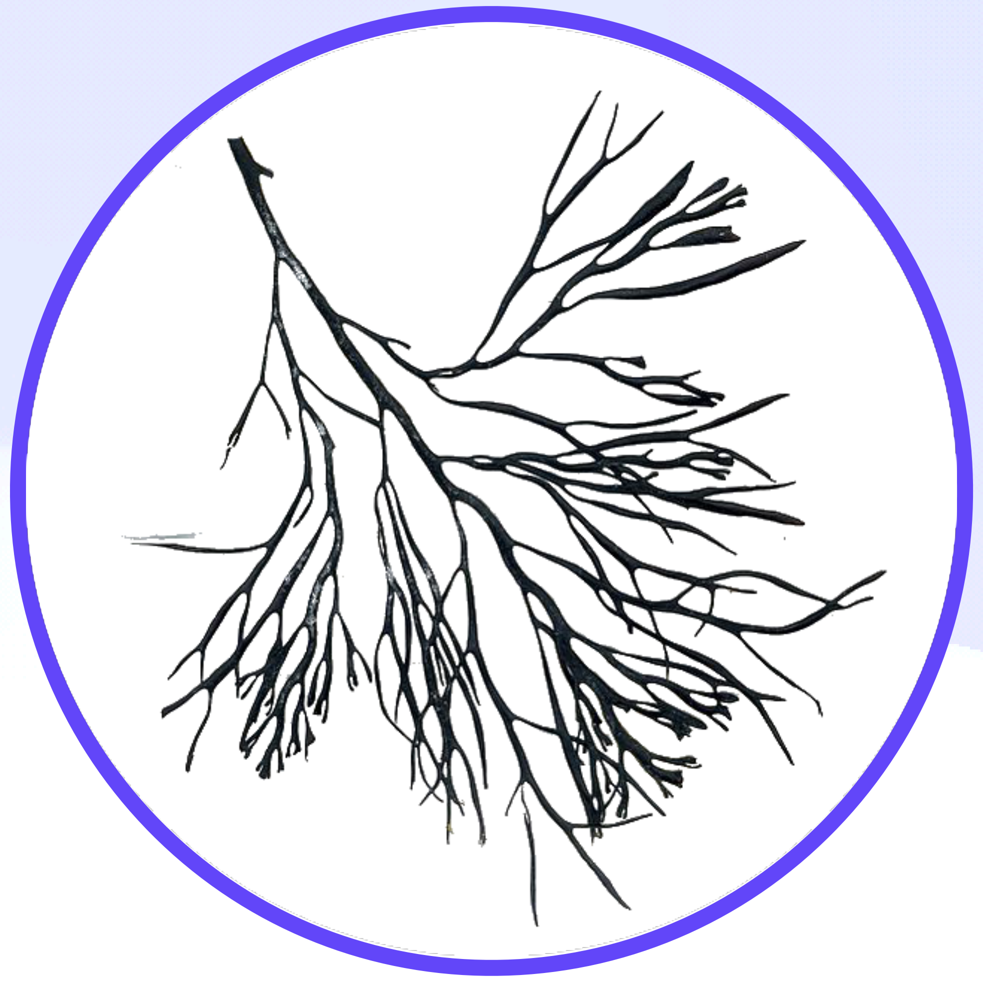
Strange Brew

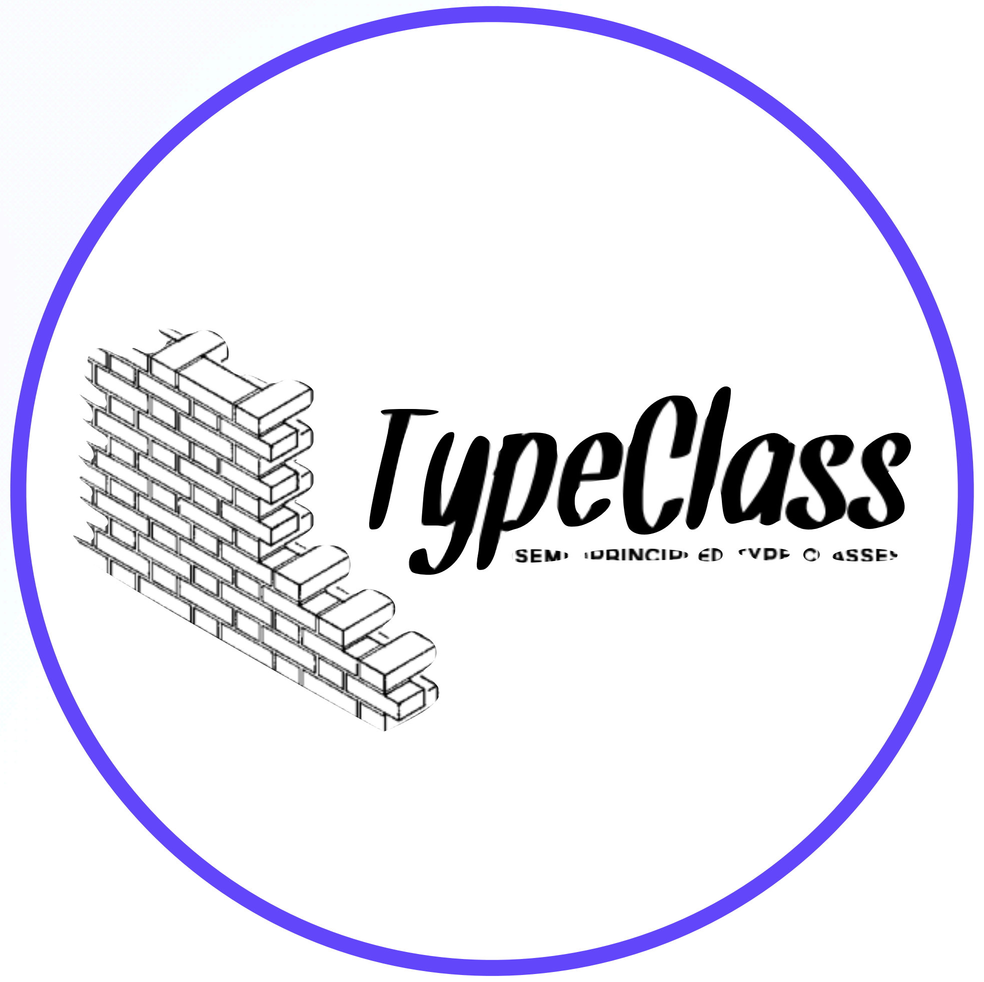
"How Hard Could It Be?"

Strange Brew

"How Hard Could It Be?"







Strange Brew

Motivation

- Elixir is awesome, but misses some (built-in) FP features
- Heavily side effectful, low on equational reasoning
- Simply typed, with bolt-on static analysis
- **At the time** there was no syntax for railroad exceptions
- VanFP skill swap: leaning Alchemists leaning Haskell & Haskellers learning Elixir

Strange Brew

What Is?

- Started (Quark) around 2014, bulk of the code ~2015
- "Classic" & denotational FP idioms
- Production use from nearly day one (lots web, but also apparently at least one bank?)

Strange Brew

Strategy

- **Stay as *idiomatic* as possible(!!!)**
- Fill feature gaps
    - e.g. protocol inheritance, lack of types
- Start small (`compose`) and grow (`Arrow.fanout`)
- Keep it simple™, except when you really need something
    - Use functions wherever possible
    - **Heavily** abuse macros as needed

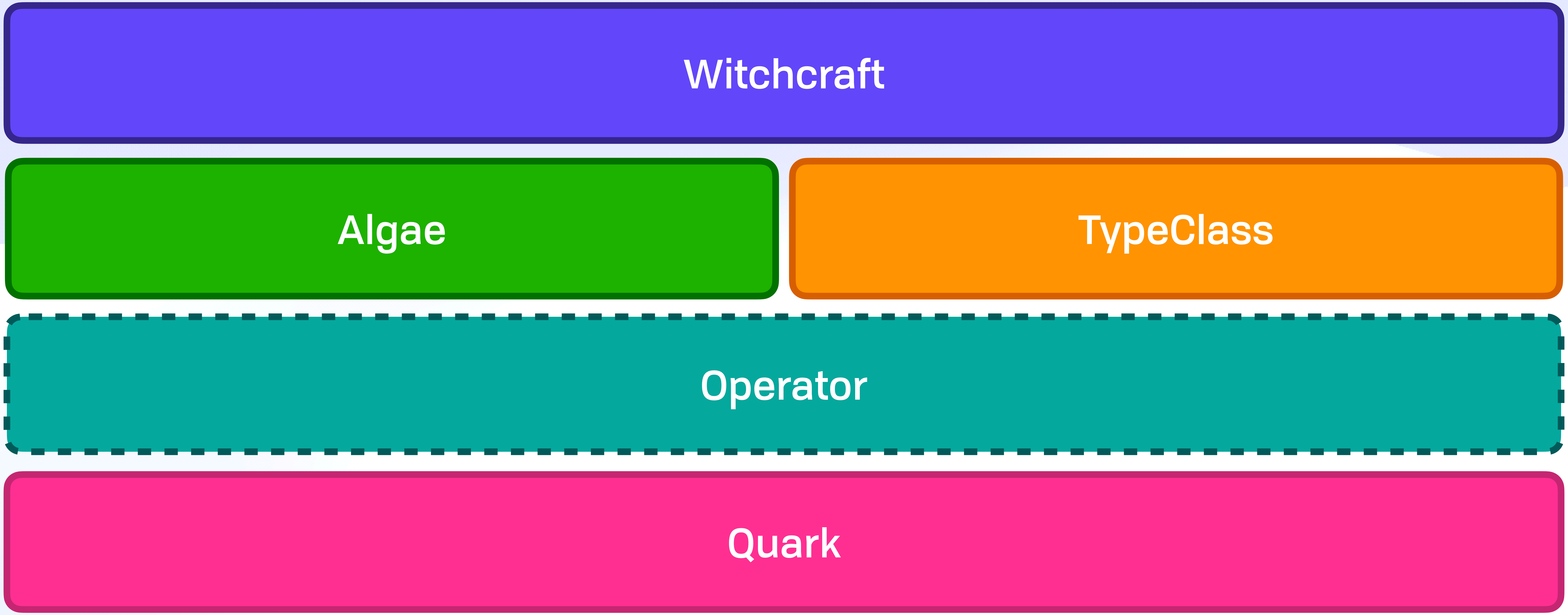
Strange Brew

Design Principles

- Compatibility with Elixir ecosystem
- Consistency with mental models
- Portability from other ecosystems
- Pedagogy and approachability

Strange Brew

Stack



Witchcraft

Algae

TypeClass

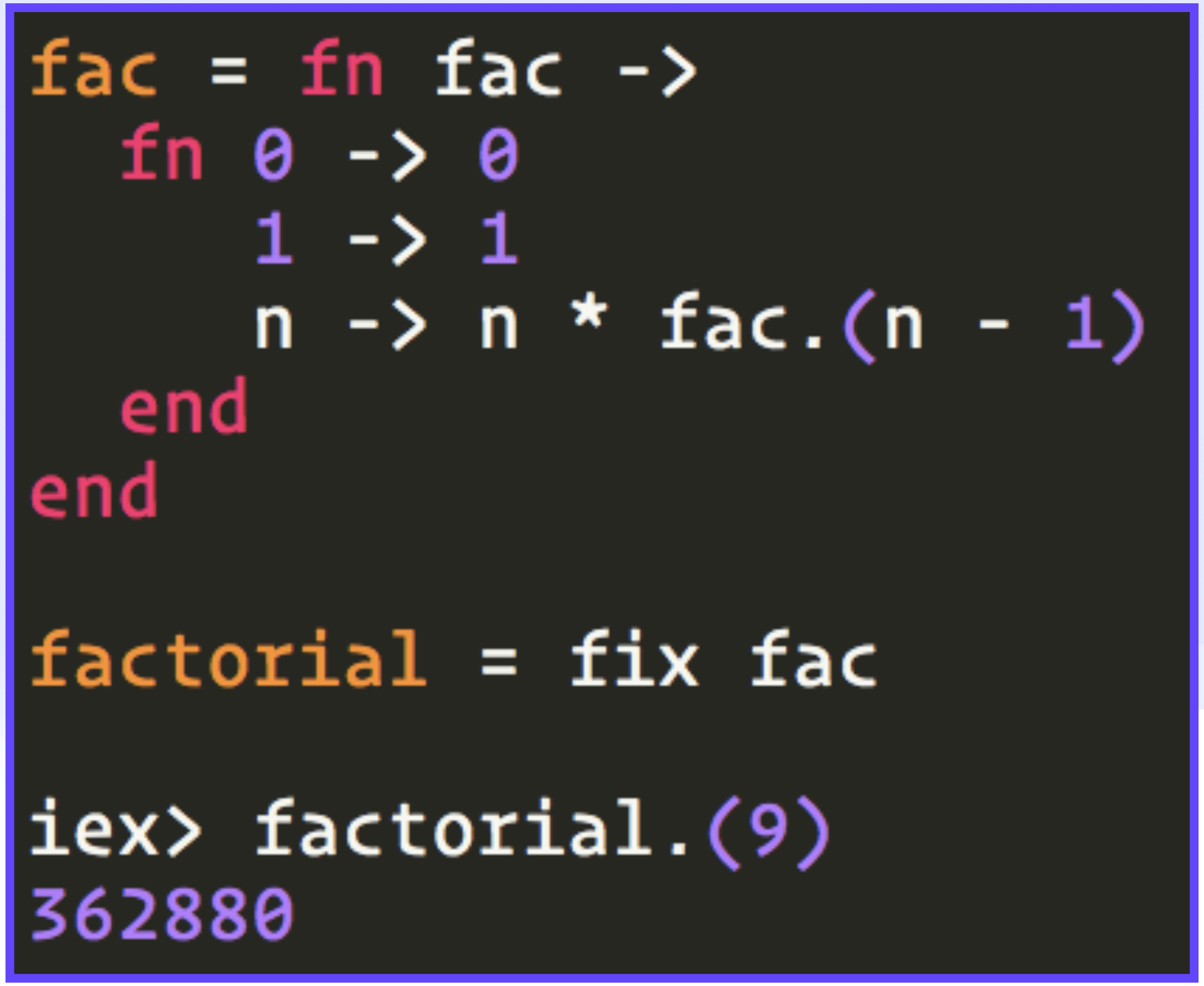
Operator

Quark

Strange Brew

Quark

- "The basics"
  - SKI, fixed points
  - Automatic partial application (defpartial, defcurry)
- "Classic" combinators
  - `id, flip, const`, and so on
  - Point-free style using <~>
- Useful! Ended up inside of e.g. `Exceptional`

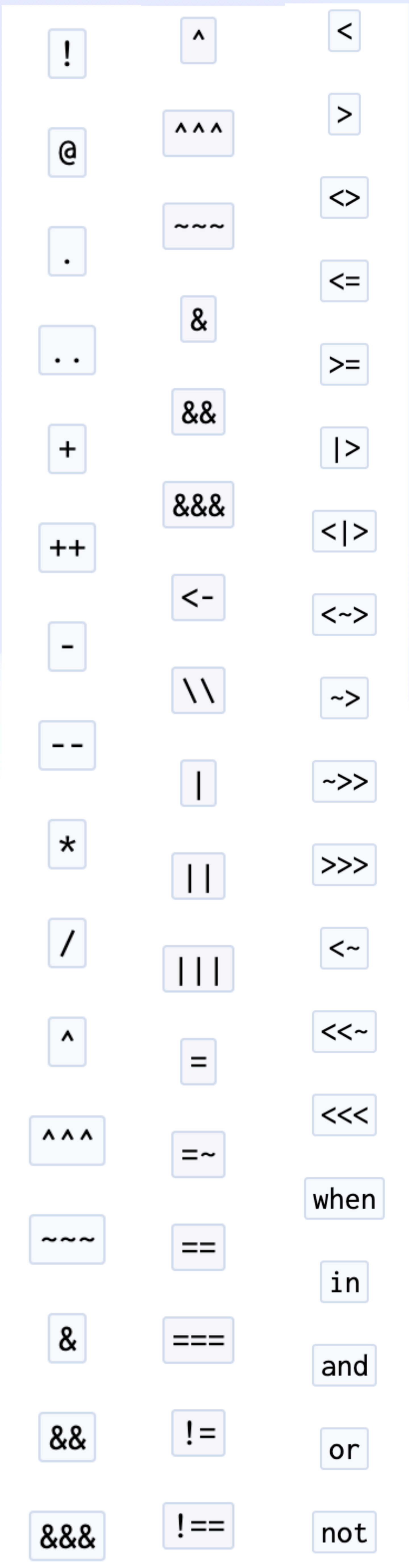


```
fac = fn fac ->
  fn 0 -> 0
     1 -> 1
     n -> n * fac.(n - 1)
  end
end

factorial = fix fac

iex> factorial.(9)
362880
```

Strange Brew

Operator

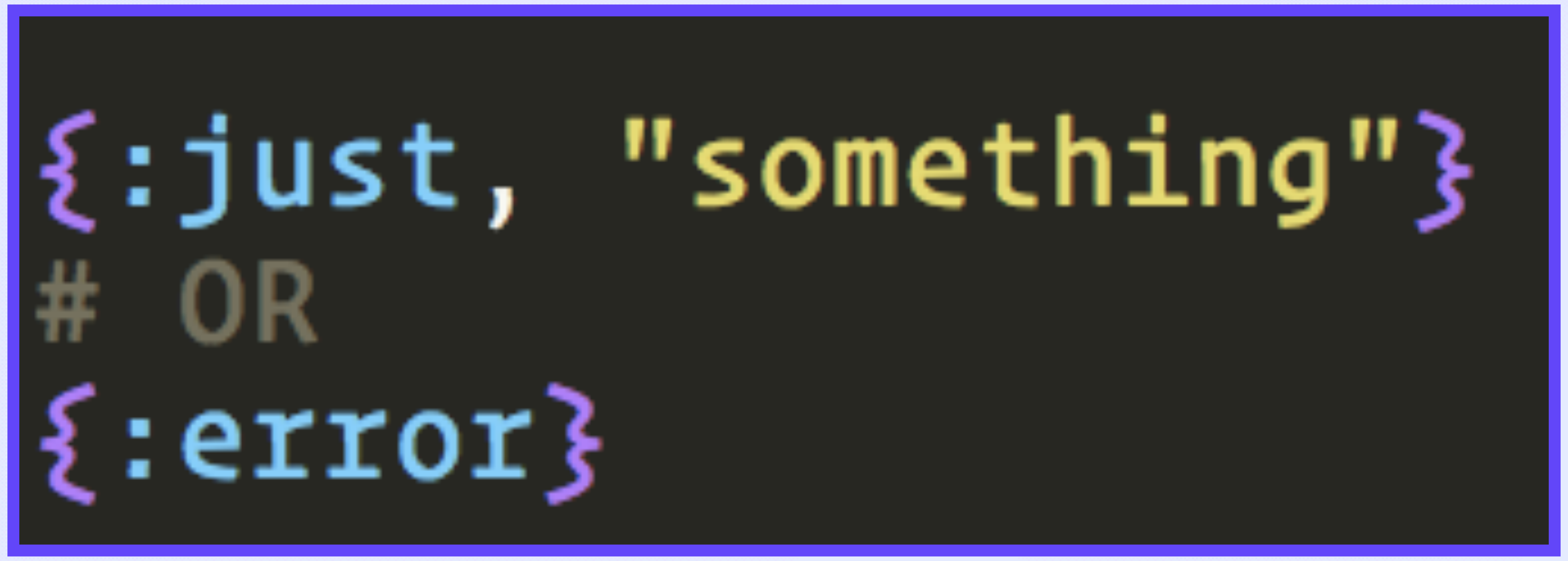
- Elixir has fixed set of operators
- Enforces best practice: always have a named variant
- Pipes get around this in most cases, but nice to have



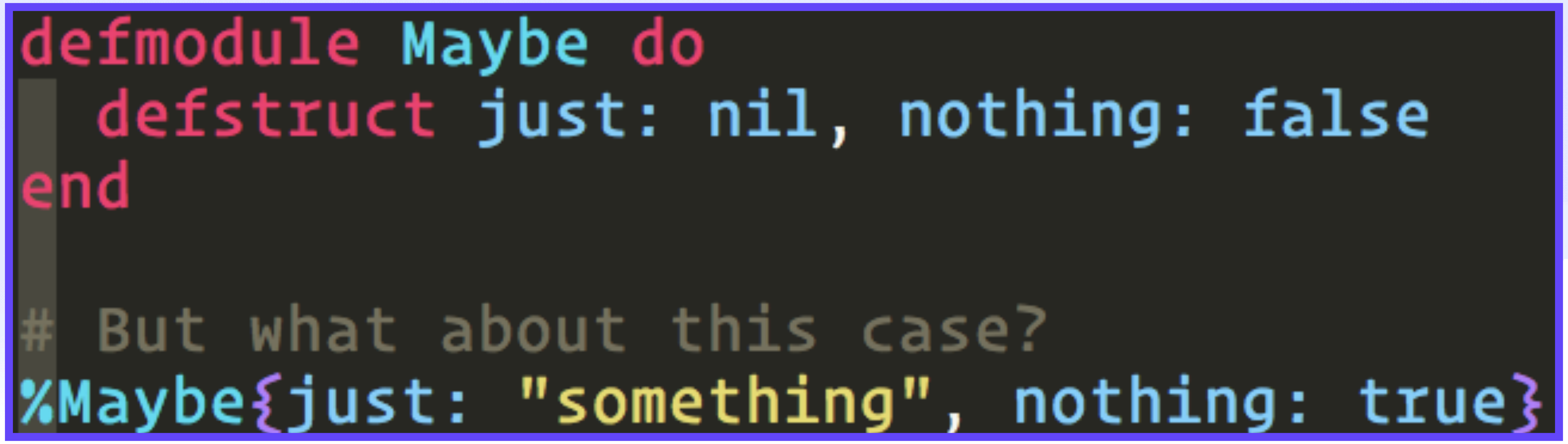
```
!
@
.
..
+
++
-
--
*
/
^
^^^
~~~
&
&&
&&&
^
^^^
~~~
&
&&
&&&
<-
\\
|
||
|||
=
=~
==
===
!=
!==
<
>
<>
<=
>=
|>
<|>
<~>
~>
~>>
>>>
<~
<<~
<<<
when
in
and
or
not
```

Strange Brew

Algae



```
{:just, "something"}
# OR
{:error}
```

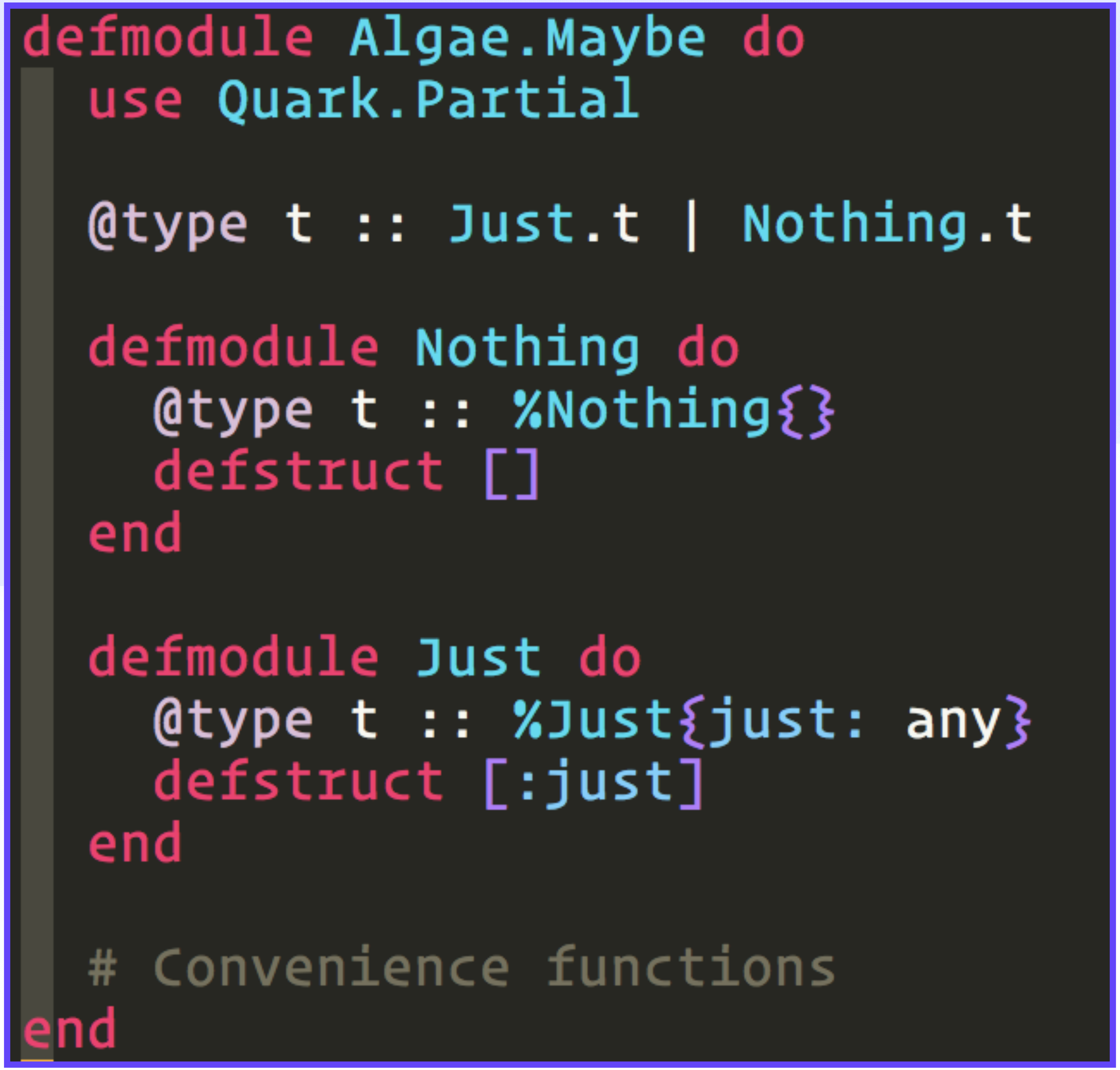


```
defmodule Maybe do
  defstruct just: nil, nothing: false
end

# But what about this case?
%Maybe{just: "something", nothing: true}
```

- **Algae**braic data types
- Coproducts, nesting
- Probably the most broadly useful library (until first-class types ship)
- Since it doesn't really require you to change the style / idioms

Strange Brew

Algae

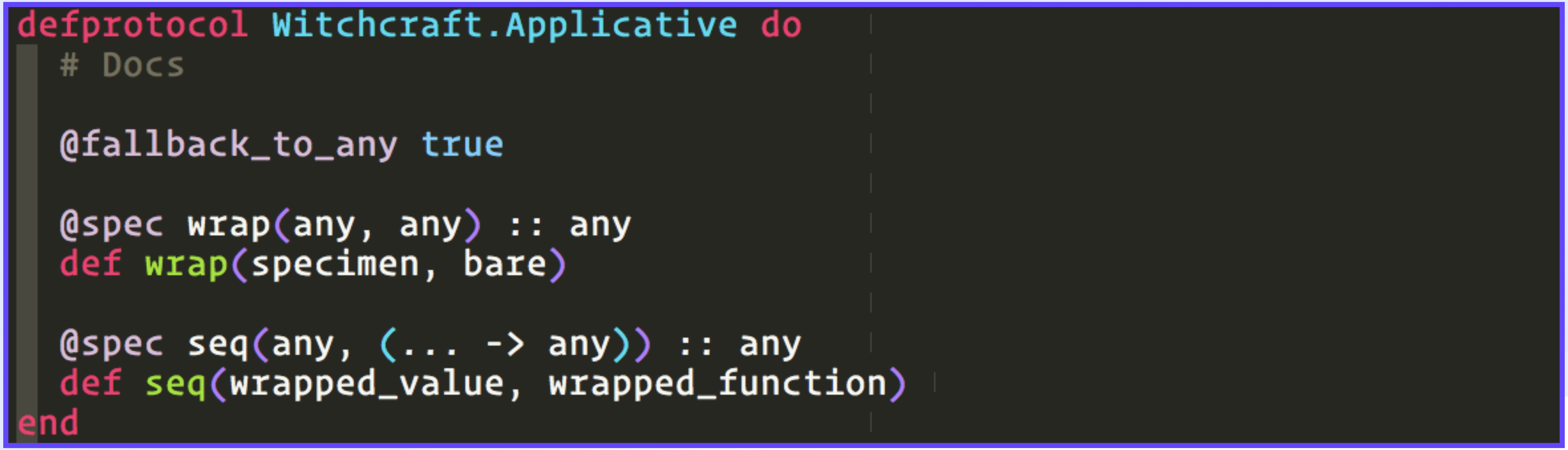


```
defmodule Algae.Maybe do
  use Quark.Partial

  @type t :: Just.t | Nothing.t

  defmodule Nothing do
    @type t :: %Nothing{}
    defstruct []
  end

  defmodule Just do
    @type t :: %Just{just: any}
    defstruct [:just]
  end

  # Convenience functions
end
```

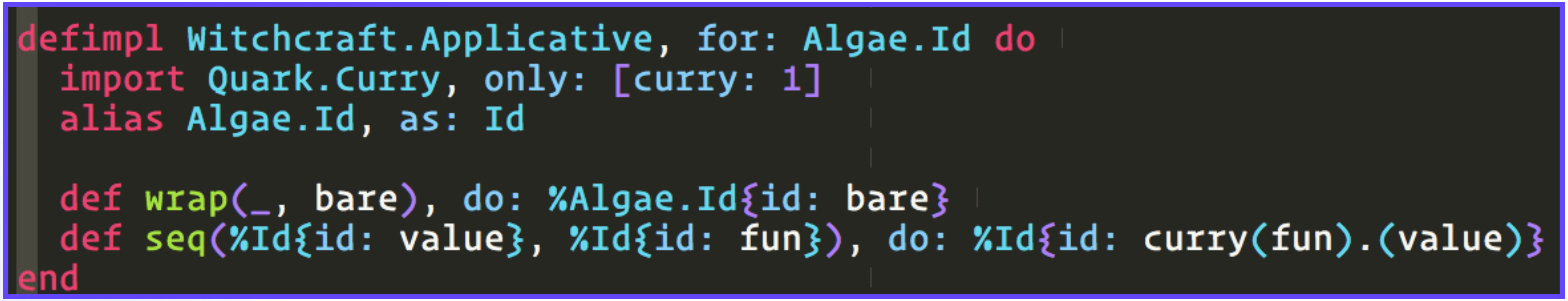
Strange Brew

TypeClass

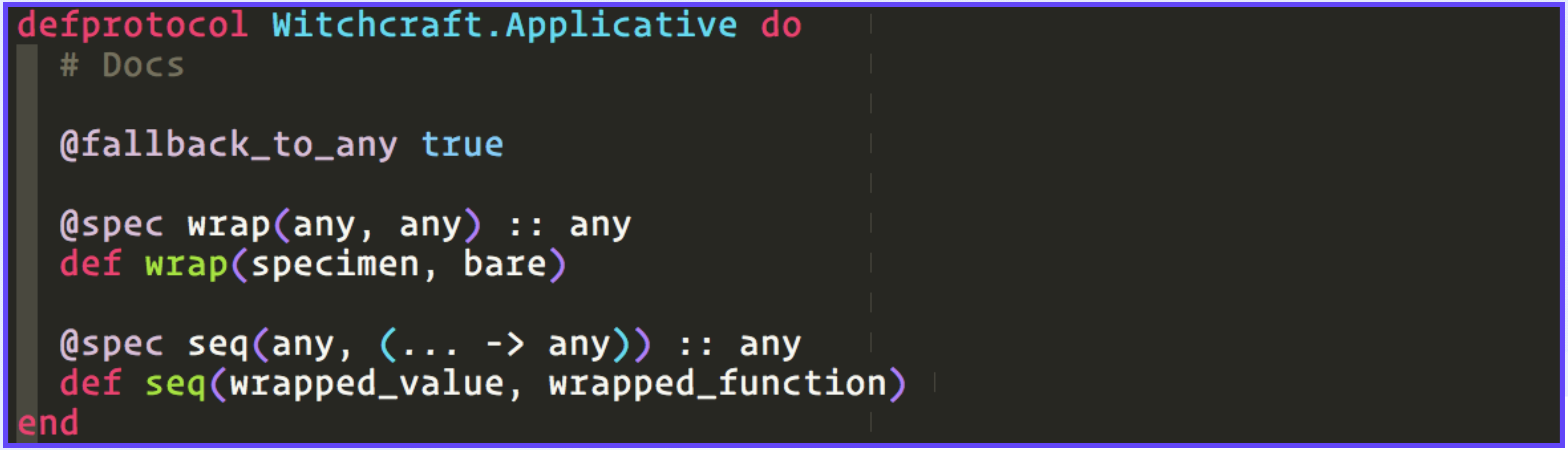
- Look, it was their fault for putting macros into the language 😉
- Elixir has protocols, but no constraint implication
- TypeClasses can be "unprincipled", so strong arm use of prop tests (even with a type system, writing e.g. dependently typed proofs can be a whole thing)
  - Lesson: Defaults are super powerful
  - We'll talk about why this was a mistake (that could have been easily fixed) later
- Actually used this in a production setting when taking over a project from a team that was months behind



```elixir
defprotocol Witchcraft.Applicative do
  # Docs

  @fallback_to_any true

  @spec wrap(any, any) :: any
  def wrap(specimen, bare)

  @spec seq(any, (... -> any)) :: any
  def seq(wrapped_value, wrapped_function)
end
```

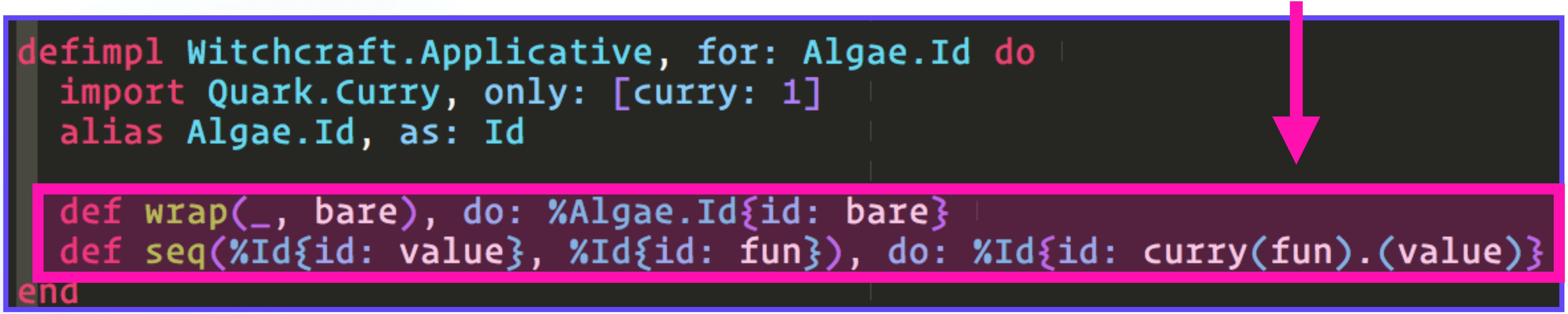


```elixir
defimpl Witchcraft.Applicative, for: Algae.Id do
  import Quark.Curry, only: [curry: 1]
  alias Algae.Id, as: Id

  def wrap(_, bare), do: %Algae.Id{id: bare}
  def seq(%Id{id: value}, %Id{id: fun}), do: %Id{id: curry(fun).(value)}
end
```

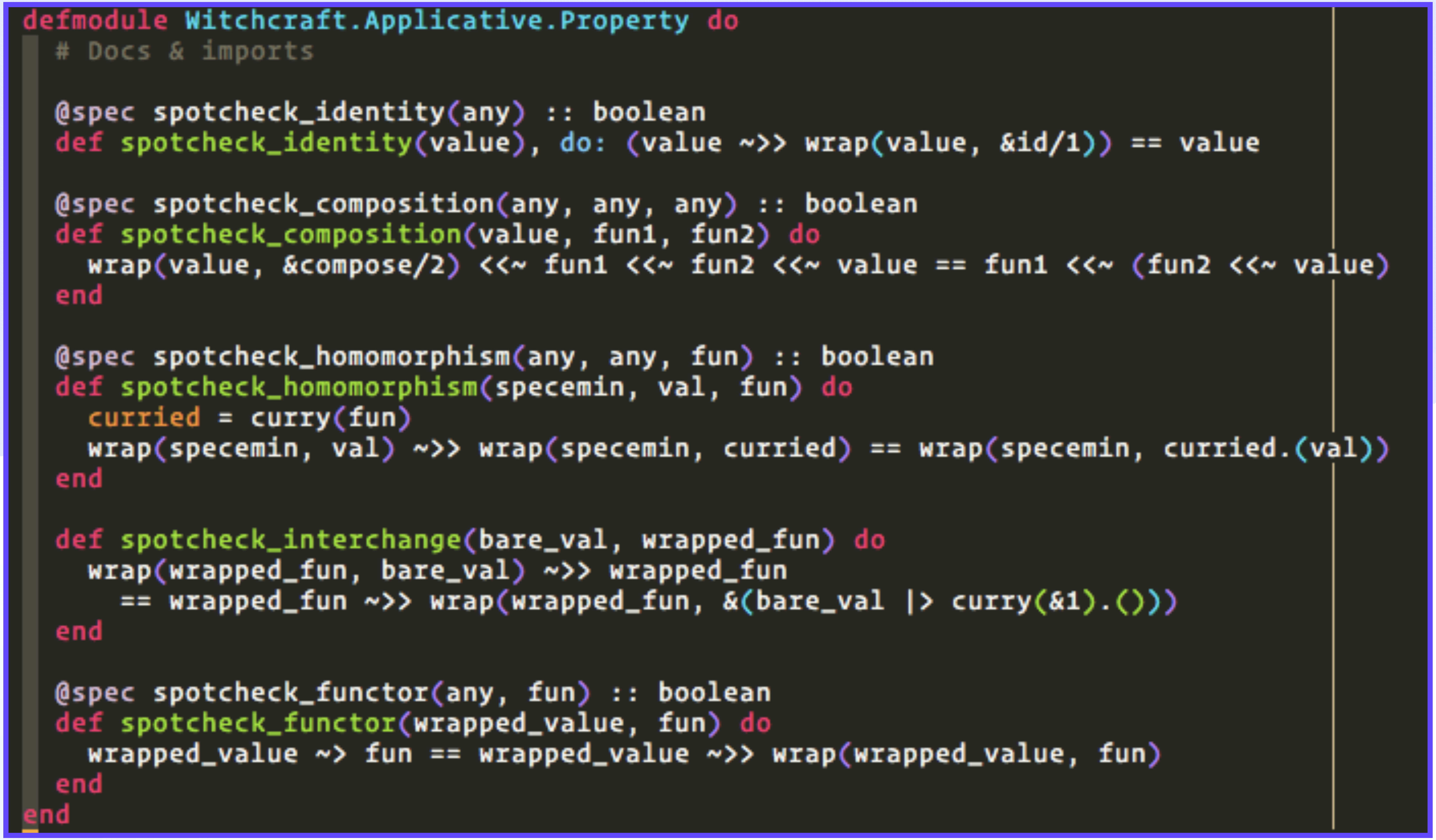


```
defprotocol Witchcraft.Applicative do
  # Docs

  @fallback_to_any true

  @spec wrap(any, any) :: any
  def wrap(specimen, bare)

  @spec seq(any, (... -> any)) :: any
  def seq(wrapped_value, wrapped_function)
end
```

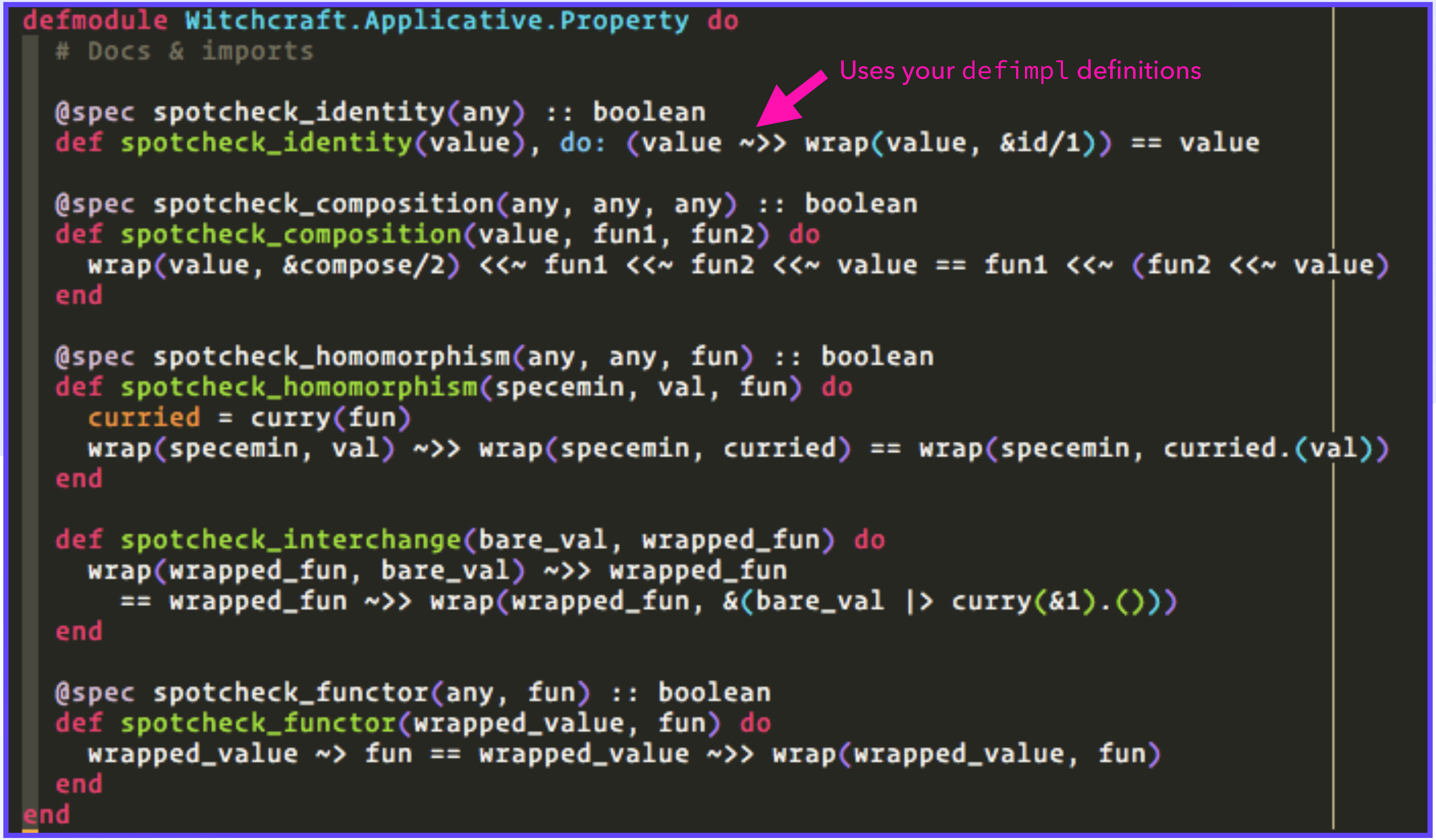
The only custom code for this data type
(low effort)



```
defimpl Witchcraft.Applicative, for: Algae.Id do
  import Quark.Curry, only: [curry: 1]
  alias Algae.Id, as: Id

  def wrap(_, bare), do: %Algae.Id{id: bare}
  def seq(%Id{id: value}, %Id{id: fun}), do: %Id{id: curry(fun).(value)}
end
```

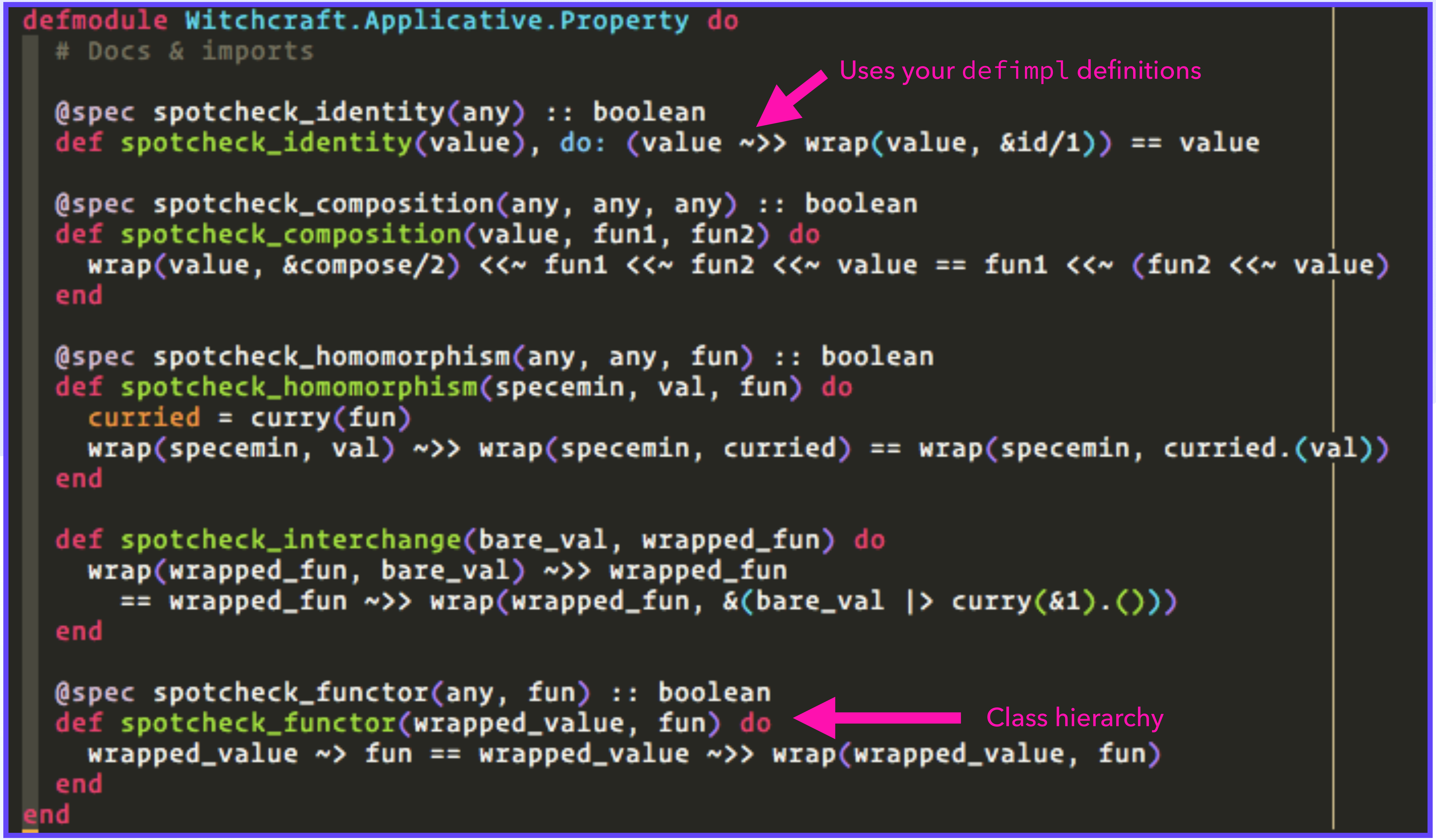


```
defmodule Witchcraft.Applicative.Property do
  # Docs & imports

  @spec spotcheck_identity(any) :: boolean
  def spotcheck_identity(value), do: (value ~>> wrap(value, &id/1)) == value

  @spec spotcheck_composition(any, any, any) :: boolean
  def spotcheck_composition(value, fun1, fun2) do
    wrap(value, &compose/2) <<~ fun1 <<~ fun2 <<~ value == fun1 <<~ (fun2 <<~ value)
  end

  @spec spotcheck_homomorphism(any, any, fun) :: boolean
  def spotcheck_homomorphism(specemin, val, fun) do
    curried = curry(fun)
    wrap(specemin, val) ~>> wrap(specemin, curried) == wrap(specemin, curried.(val))
  end

  def spotcheck_interchange(bare_val, wrapped_fun) do
    wrap(wrapped_fun, bare_val) ~>> wrapped_fun
      == wrapped_fun ~>> wrap(wrapped_fun, &(bare_val |> curry(&1).()))
  end

  @spec spotcheck_functor(any, fun) :: boolean
  def spotcheck_functor(wrapped_value, fun) do
    wrapped_value ~> fun == wrapped_value ~>> wrap(wrapped_value, fun)
  end
end
```



```elixir
defmodule Witchcraft.Applicative.Property do
  # Docs & imports

  @spec spotcheck_identity(any) :: boolean
  def spotcheck_identity(value), do: (value ~>> wrap(value, &id/1)) == value

  @spec spotcheck_composition(any, any, any) :: boolean
  def spotcheck_composition(value, fun1, fun2) do
    wrap(value, &compose/2) <<~ fun1 <<~ fun2 <<~ value == fun1 <<~ (fun2 <<~ value)
  end

  @spec spotcheck_homomorphism(any, any, fun) :: boolean
  def spotcheck_homomorphism(specemin, val, fun) do
    curried = curry(fun)
    wrap(specemin, val) ~>> wrap(specemin, curried) == wrap(specemin, curried.(val))
  end

  def spotcheck_interchange(bare_val, wrapped_fun) do
    wrap(wrapped_fun, bare_val) ~>> wrapped_fun
      == wrapped_fun ~>> wrap(wrapped_fun, &(bare_val |> curry(&1).()))
  end

  @spec spotcheck_functor(any, fun) :: boolean
  def spotcheck_functor(wrapped_value, fun) do
    wrapped_value ~> fun == wrapped_value ~>> wrap(wrapped_value, fun)
  end
end
```

Uses your `defimpl` definitions



```
defmodule Witchcraft.Applicative.Property do
  # Docs & imports

  @spec spotcheck_identity(any) :: boolean
  def spotcheck_identity(value), do: (value ~>> wrap(value, &id/1)) == value

  @spec spotcheck_composition(any, any, any) :: boolean
  def spotcheck_composition(value, fun1, fun2) do
    wrap(value, &compose/2) <<~ fun1 <<~ fun2 <<~ value == fun1 <<~ (fun2 <<~ value)
  end

  @spec spotcheck_homomorphism(any, any, fun) :: boolean
  def spotcheck_homomorphism(specemin, val, fun) do
    curried = curry(fun)
    wrap(specemin, val) ~>> wrap(specemin, curried) == wrap(specemin, curried.(val))
  end

  def spotcheck_interchange(bare_val, wrapped_fun) do
    wrap(wrapped_fun, bare_val) ~>> wrapped_fun
      == wrapped_fun ~>> wrap(wrapped_fun, &(bare_val |> curry(&1).()))
  end

  @spec spotcheck_functor(any, fun) :: boolean
  def spotcheck_functor(wrapped_value, fun) do
    wrapped_value ~> fun == wrapped_value ~>> wrap(wrapped_value, fun)
  end
end
```

Uses your `defimpl` definitions

Class hierarchy

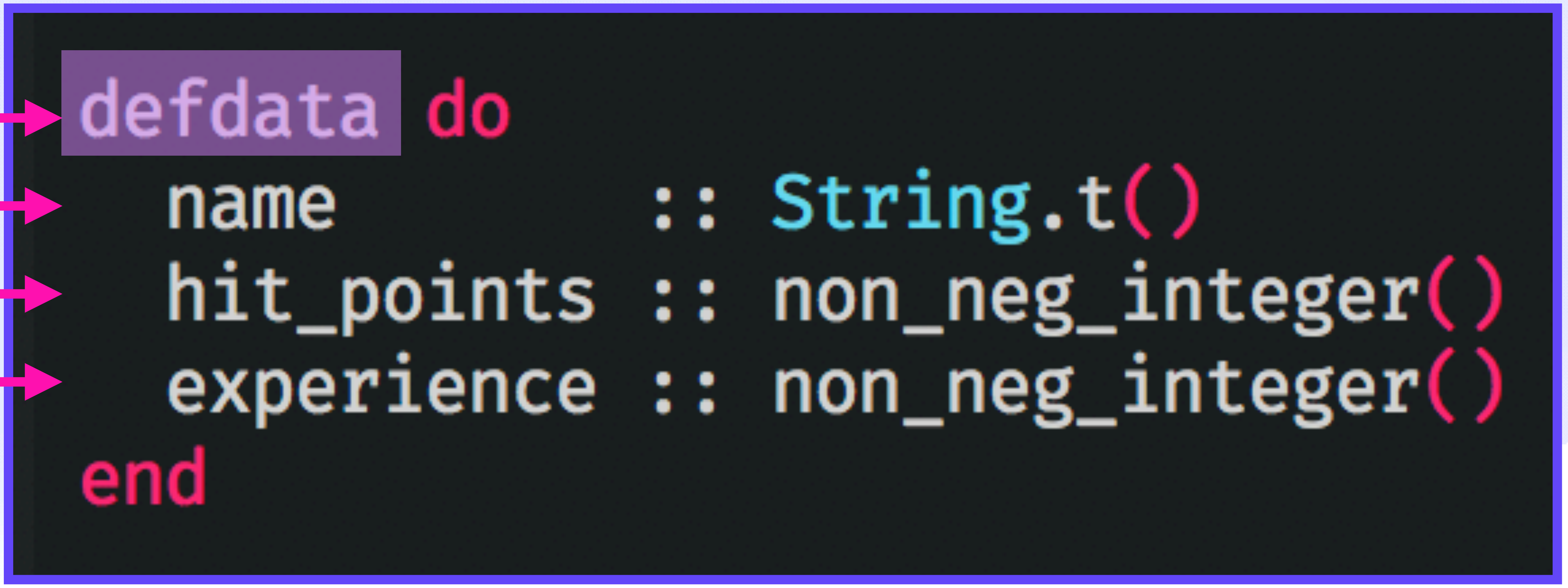
Strange Brew

Algae

`defdata`

All of these fields

Roughly “and”

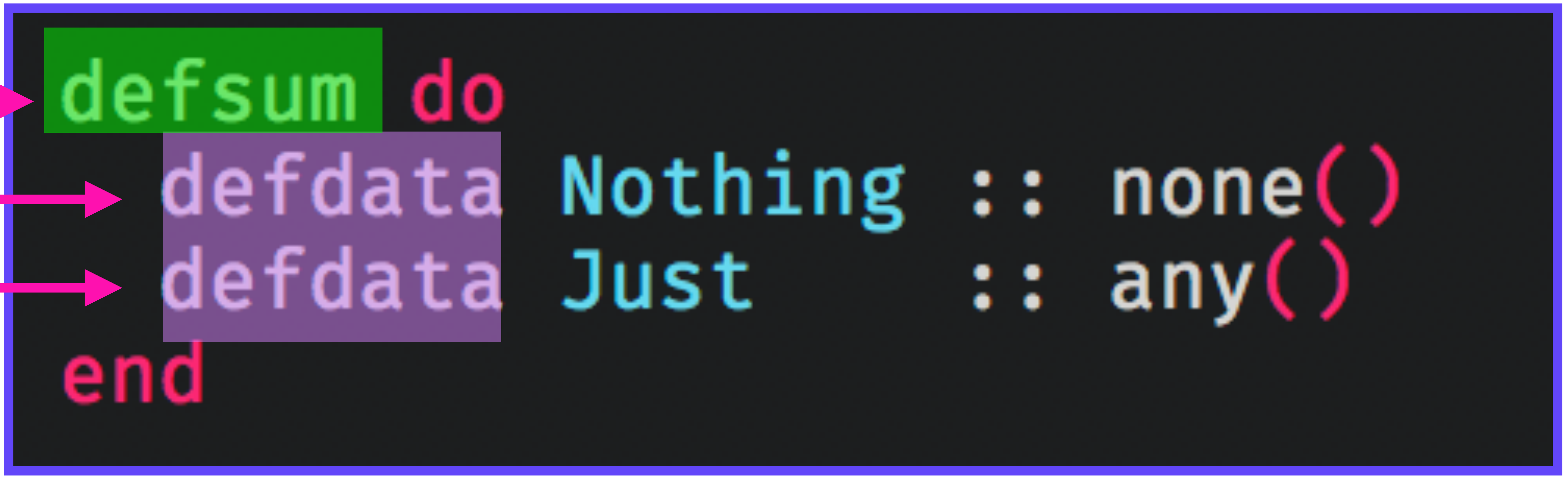


```
defdata do
  name        :: String.t()
  hit_points  :: non_neg_integer()
  experience  :: non_neg_integer()
end
```

`defsum`

One of these structs

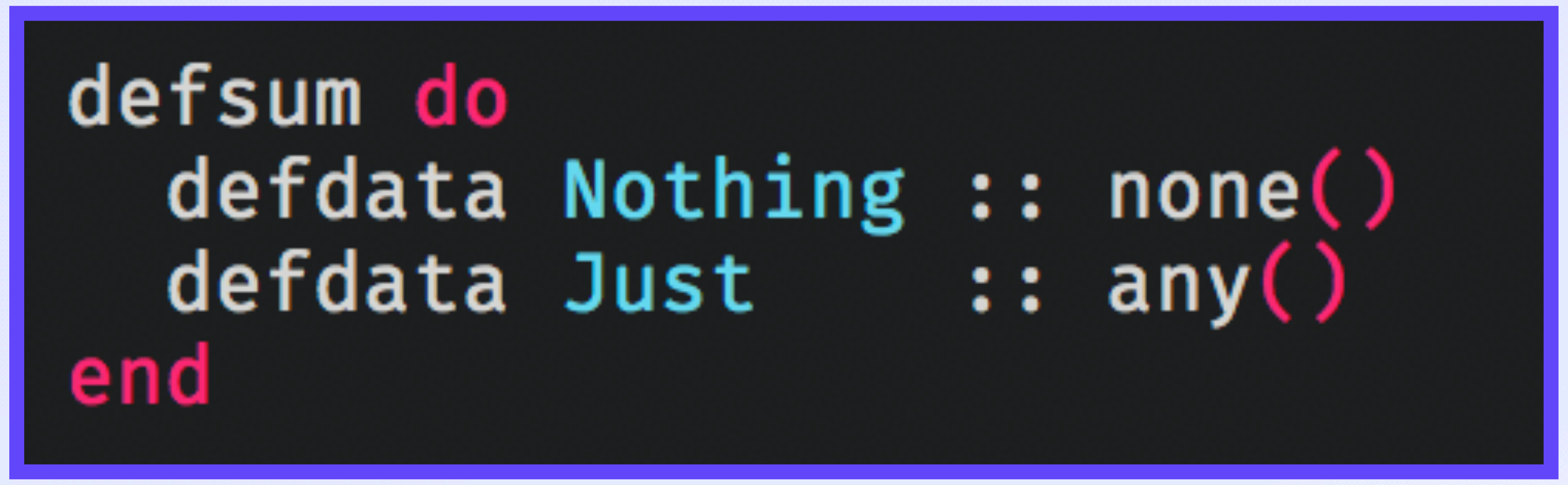
Roughly “or”



```
defsum do
  defdata Nothing :: none()
  defdata Just    :: any()
end
```

Strange Brew

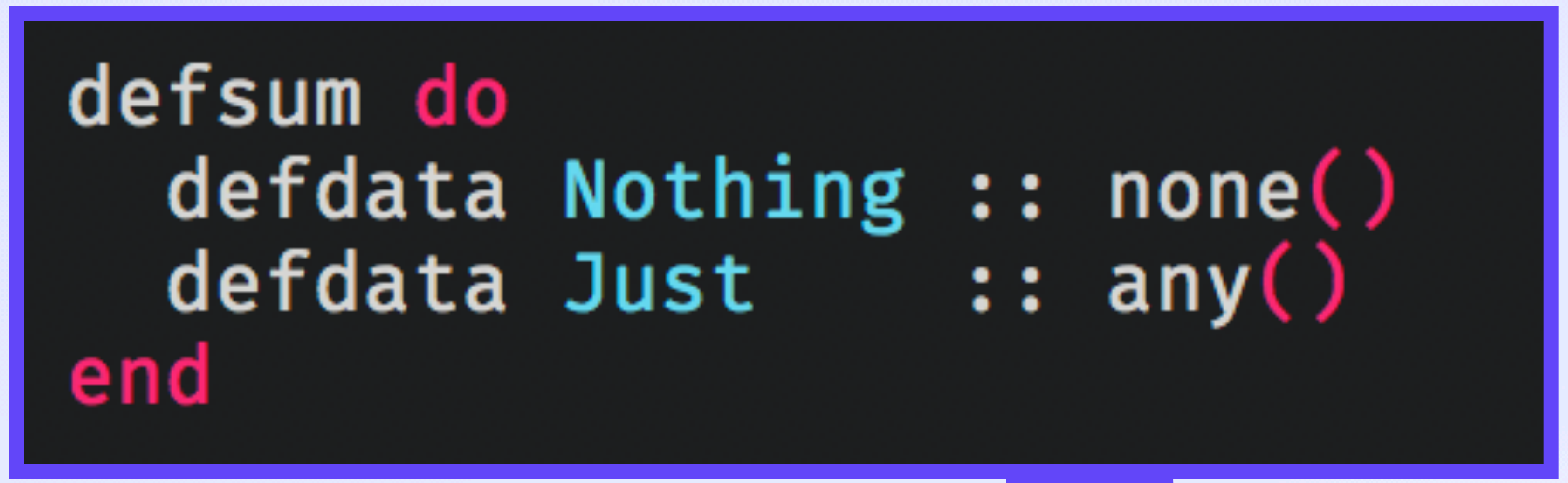
Algae



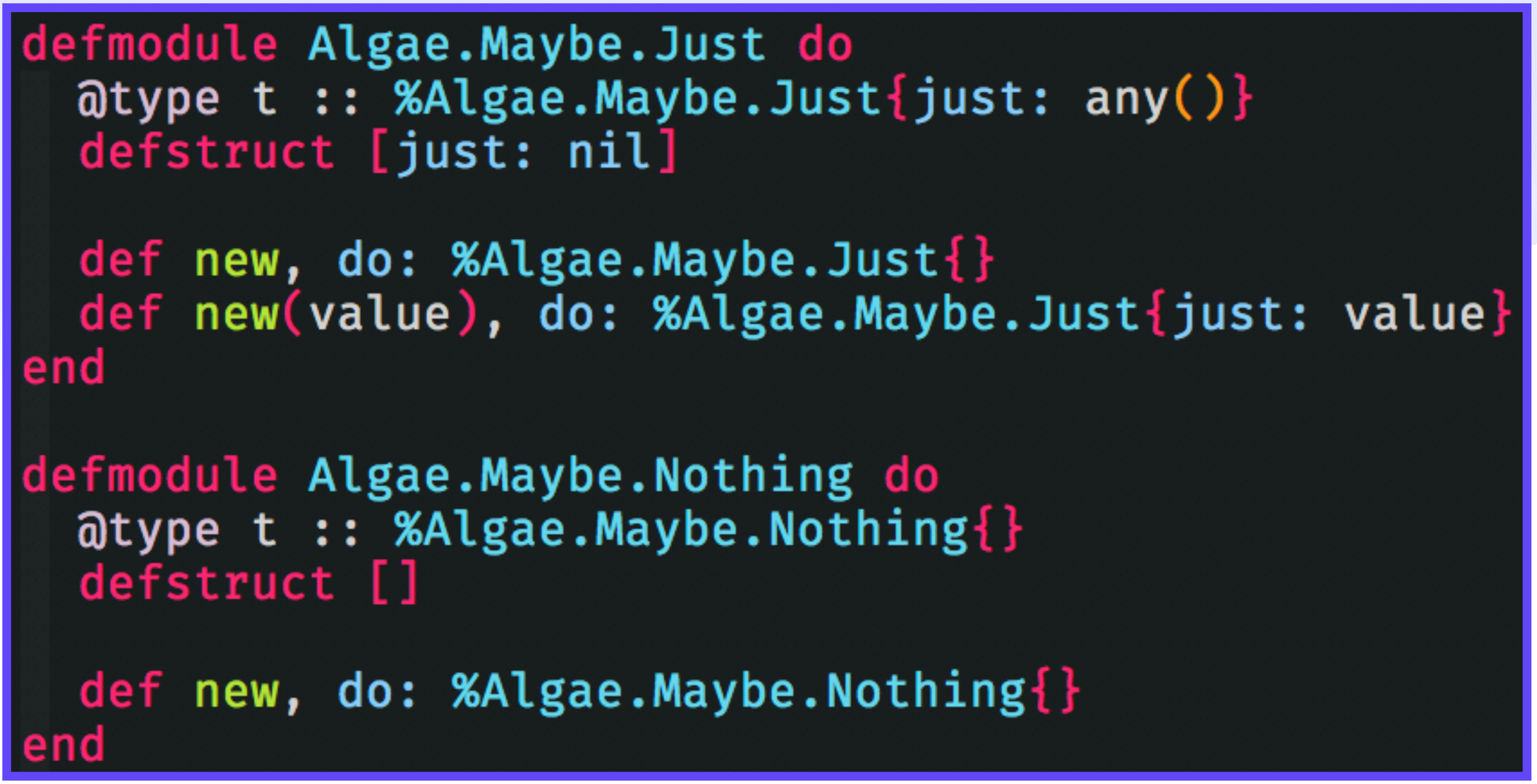
```
defsum do
  defdata Nothing :: none()
  defdata Just    :: any()
end
```

Strange Brew

Algae



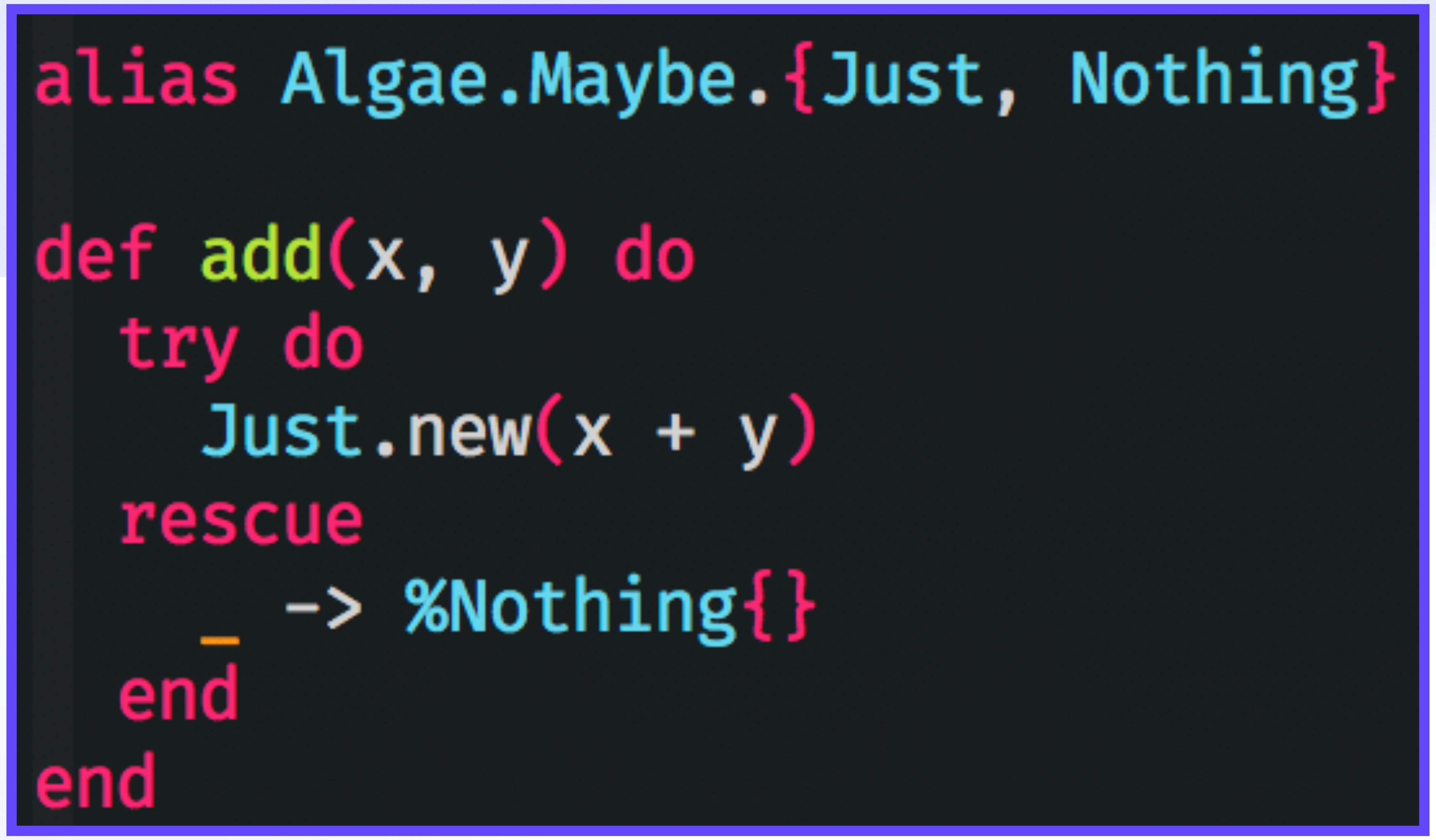
```
defsum do
  defdata Nothing :: none()
  defdata Just    :: any()
end
```



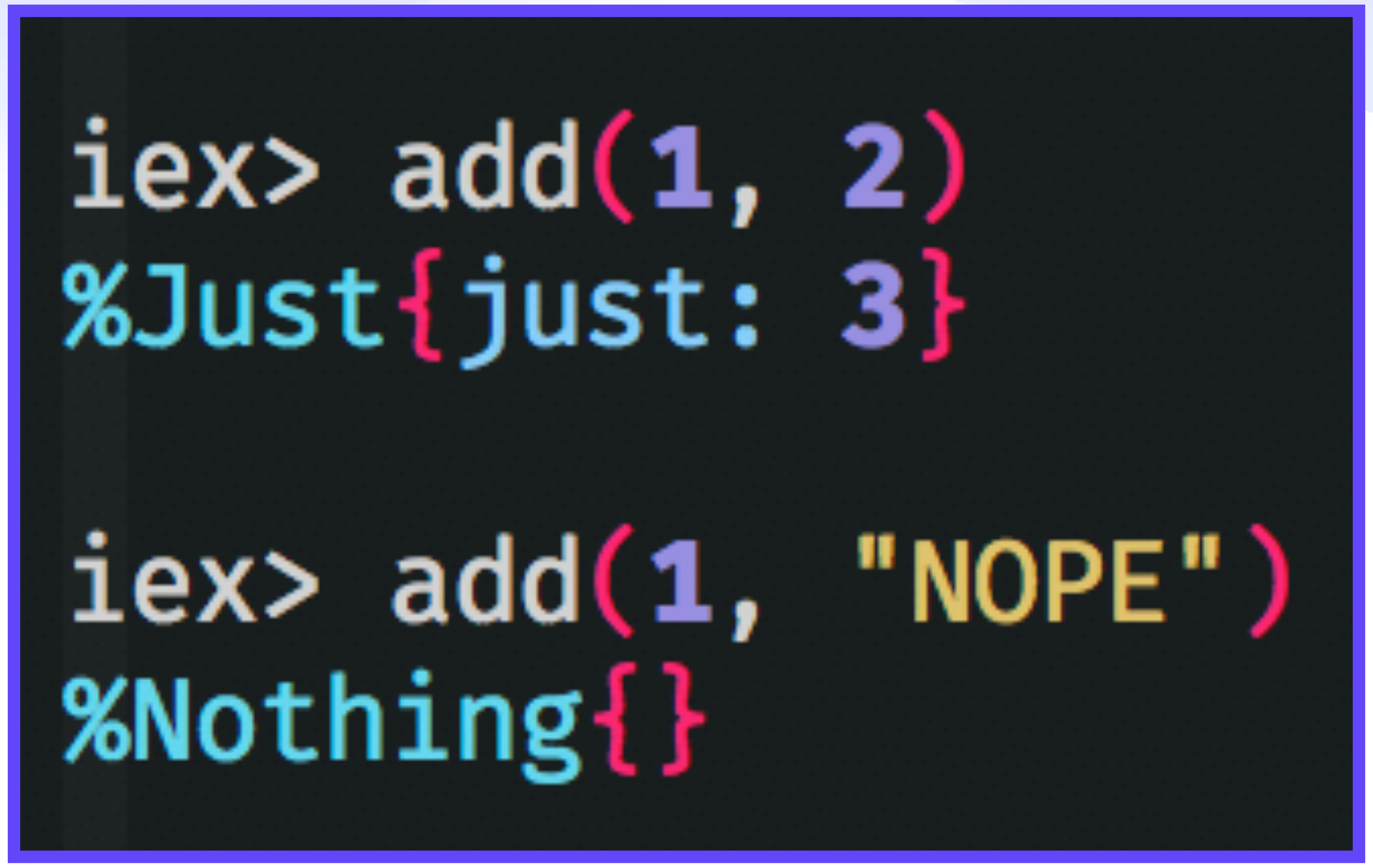
```
defmodule Algae.Maybe.Just do
  @type t :: %Algae.Maybe.Just{just: any()}
  defstruct [just: nil]

  def new, do: %Algae.Maybe.Just{}
  def new(value), do: %Algae.Maybe.Just{just: value}
end

defmodule Algae.Maybe.Nothing do
  @type t :: %Algae.Maybe.Nothing{}
  defstruct []

  def new, do: %Algae.Maybe.Nothing{}
end
```

Strange Brew

Algae



```
alias Algae.Maybe.{Just, Nothing}

def add(x, y) do
  try do
    Just.new(x + y)
  rescue
    _ -> %Nothing{}
  end
end
```



```
iex> add(1, 2)
%Just{just: 3}

iex> add(1, "NOPE")
%Nothing{}
```

Strange Brew

Witchcraft

- "The main show"
- No way to enforce purity 😭
- Async variants
    - `map` and `async_map`
    - One way of thinking about Elixir is that it's implicitly in IO, or at best Async
        - Why implicit asyncs instead of a monad?
- Differences from Haskell
    - Pipe order is different

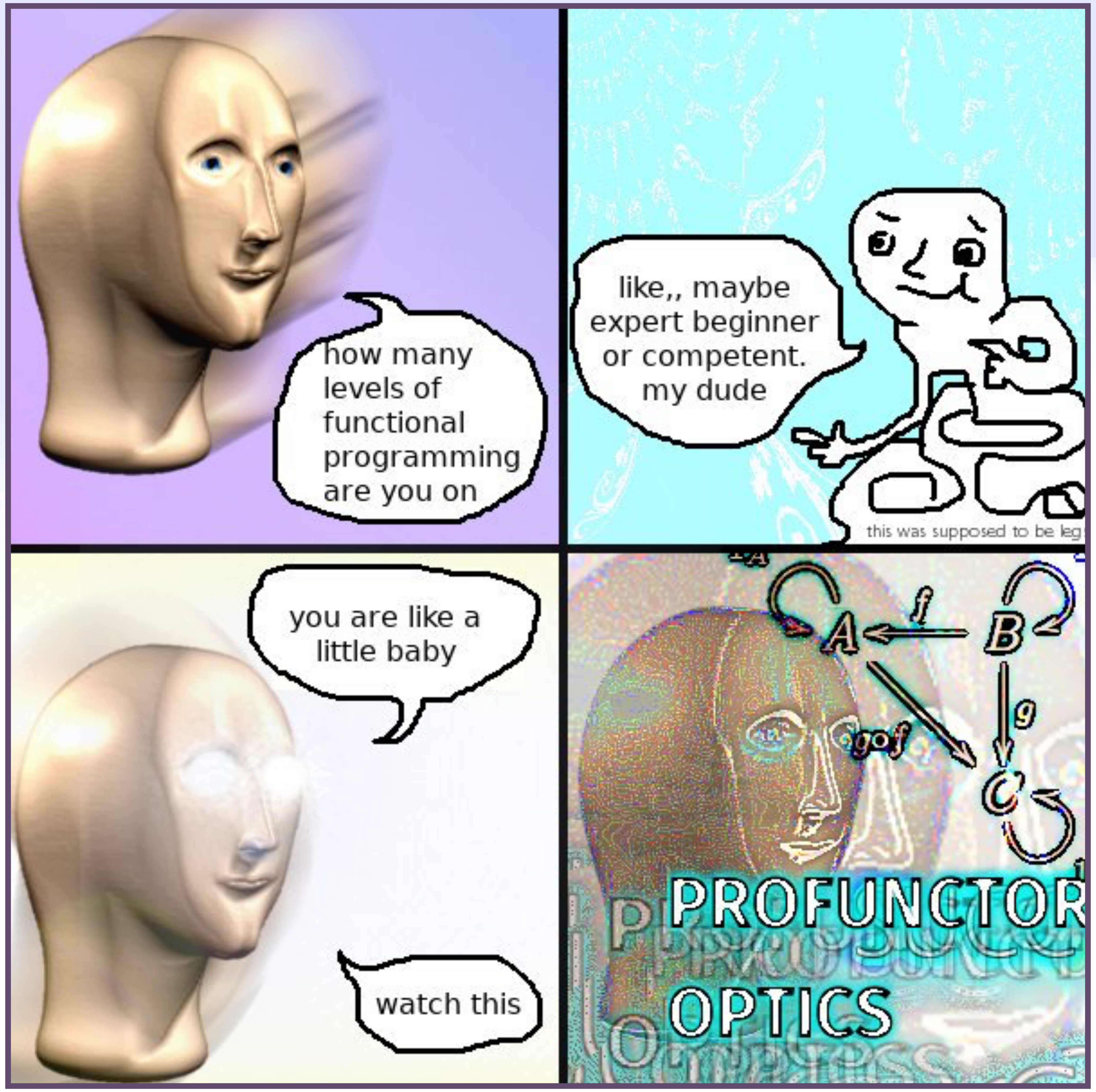
More than Syntax… but also Syntax

Consistency & Ethos

🚰

Consistency & Ethos

What We're Trying to Avoid

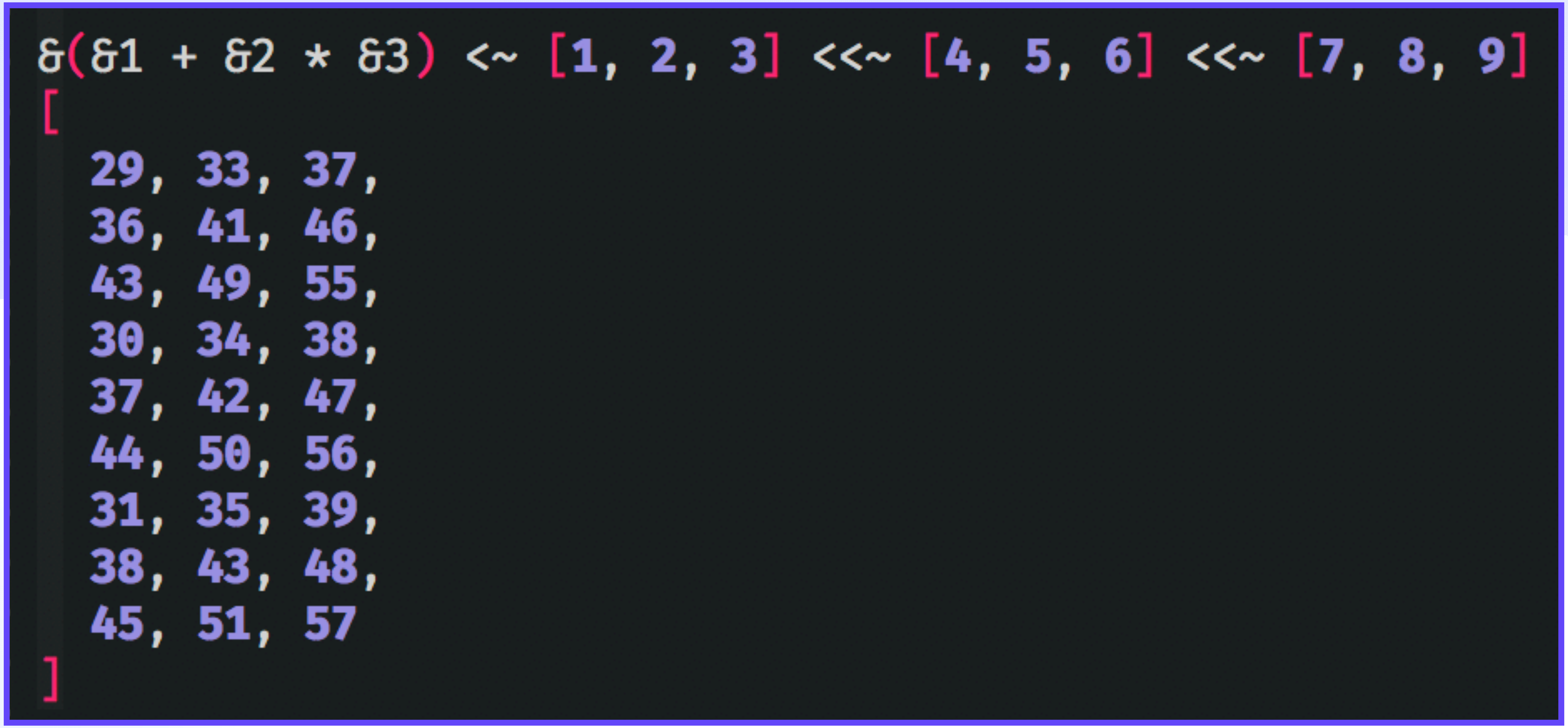

how many levels of functional programming are you on
like,, maybe expert beginner or competent. my dude
this was supposed to be leg
you are like a little baby
watch this
A
B
C
f
g
g∘f
PROFUNCTOR
OPTICS

Strange Brew

Pipes

Strange Brew

Pipes

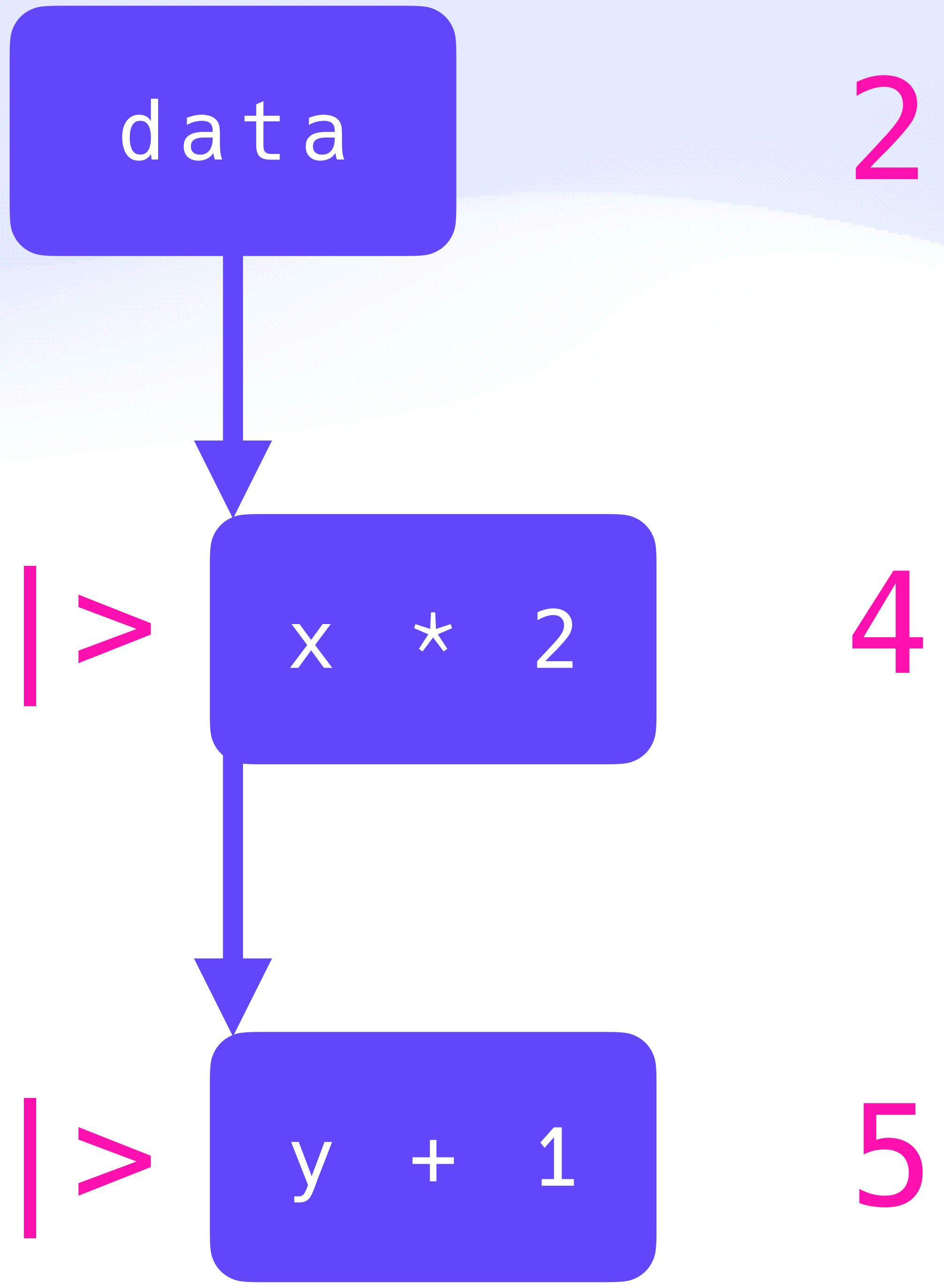


```
&(&1 + &2 * &3) <~ [1, 2, 3] <<~ [4, 5, 6] <<~ [7, 8, 9]
[
  29, 33, 37,
  36, 41, 46,
  43, 49, 55,
  30, 34, 38,
  37, 42, 47,
  44, 50, 56,
  31, 35, 39,
  38, 43, 48,
  45, 51, 57
]
```

Consistency & Ethos

Dataflow & Directionality

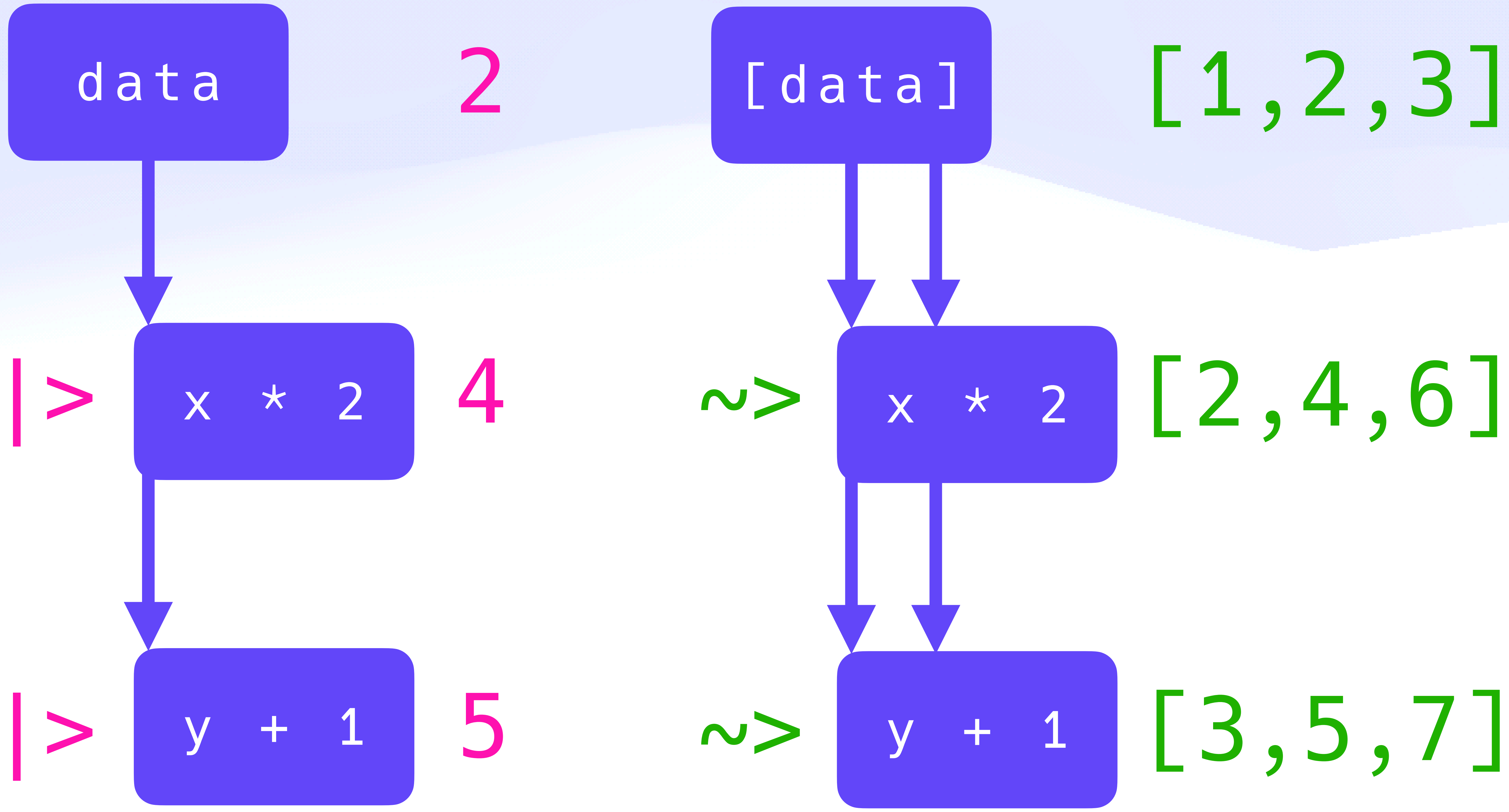
- Let's bootstrap people's intuitions!
- Elixir prefers diagrammatic ordering
- Important to maintain consistency with rest of language!
- Pipes are generally awesome
- Want to maintain this awesomeness
- What if we just gave the pipe operator superpowers?



Consistency & Ethos

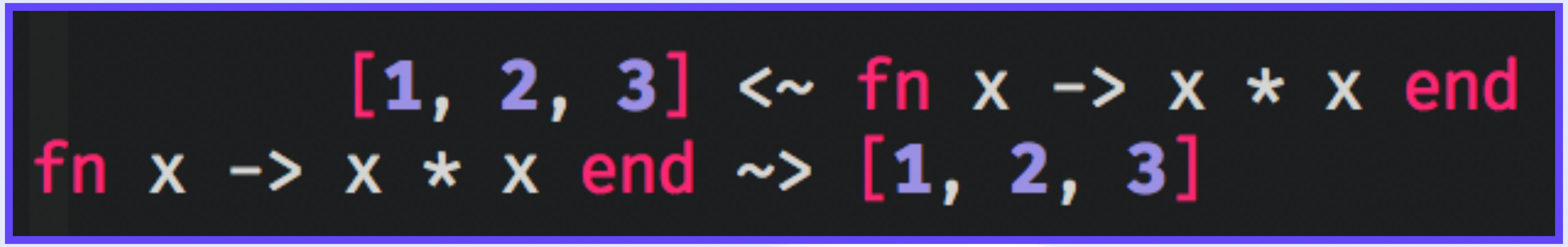
Giving Pipes Superpowers

- Witchcraft operators follow same flow
- Data on flows through pointed direction
- Just like pipes
- |> becomes ~> (curried `map/2`)



Consistency & Ethos

Dataflow & Directionality

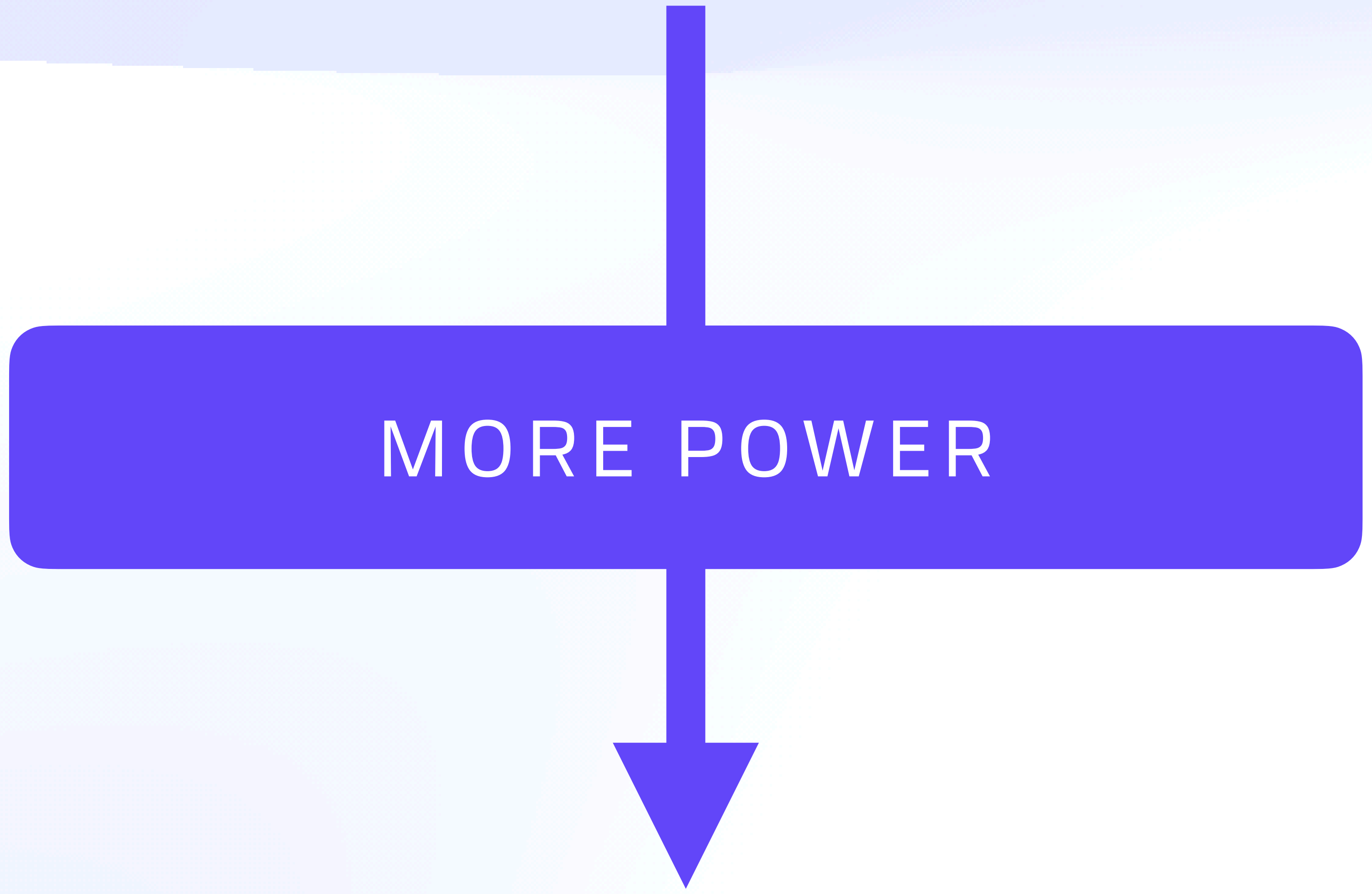


```
      [1, 2, 3] <~ fn x -> x * x end
fn x -> x * x end ~> [1, 2, 3]
```

- Operators follow same flow

- Data on flows through arrow direction

- |>
- ~>
- ~>>
- >>>



MORE POWER

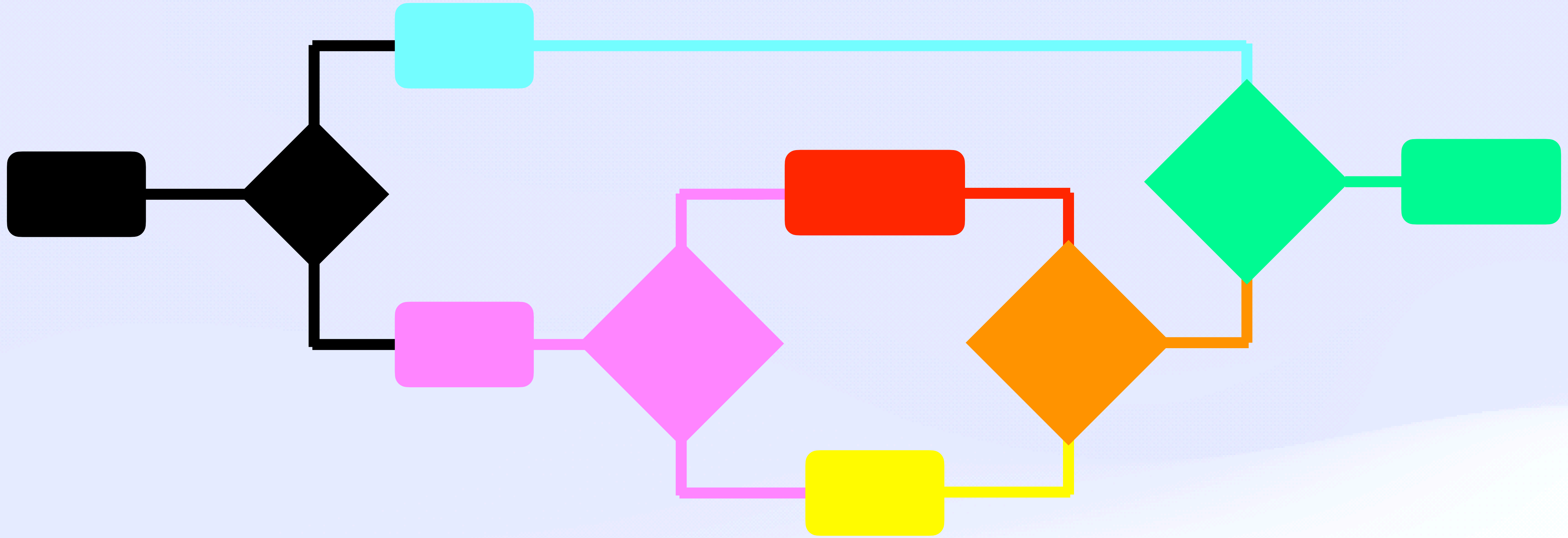
(_)	apply/2
<~	map/2
<<~	ap/2
<<<	chain/2

Consistency & Ethos

Arrows

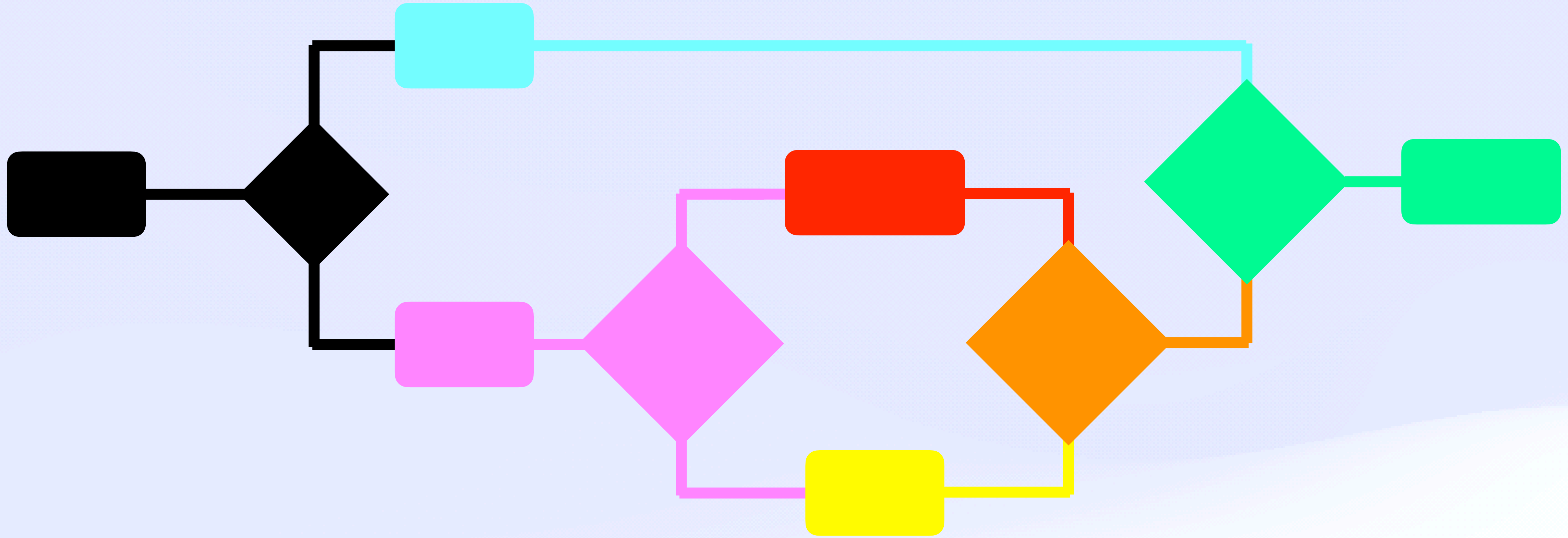
Consistency & Ethos

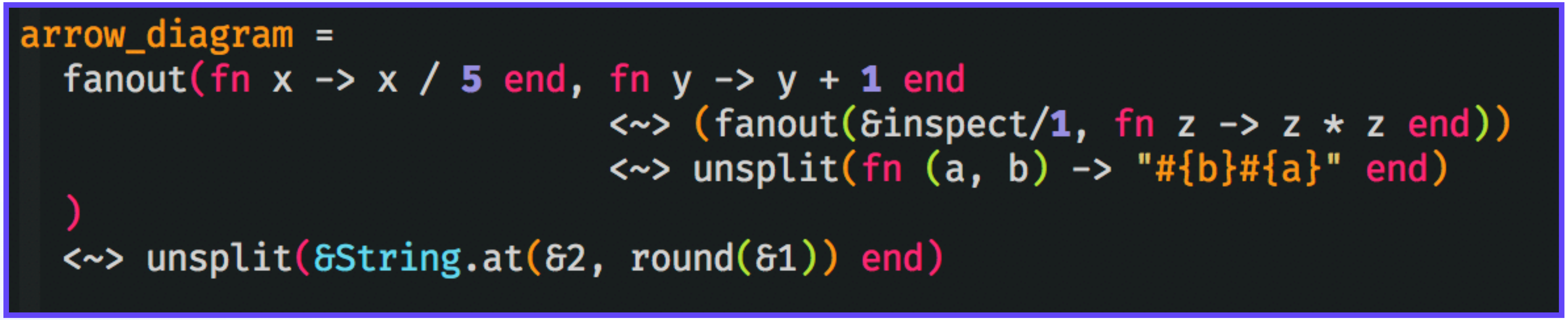
Arrows



Consistency & Ethos

Arrows

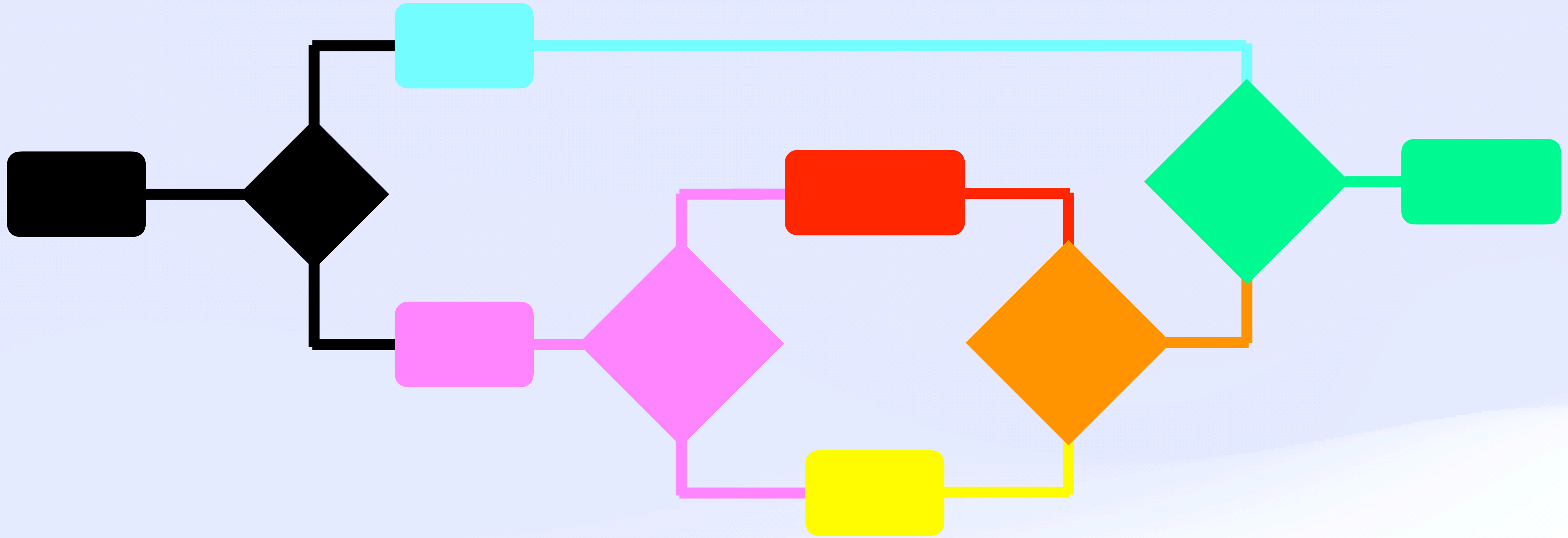


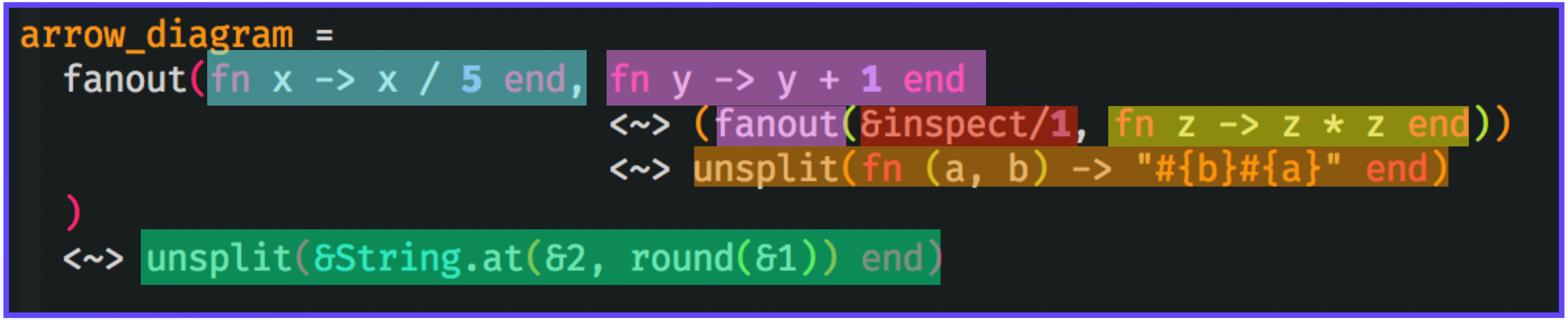


```
arrow_diagram =
  fanout(fn x -> x / 5 end, fn y -> y + 1 end
                            <~> (fanout(&inspect/1, fn z -> z * z end))
                            <~> unsplit(fn (a, b) -> "#{b}#{a}" end)
  )
  <~> unsplit(&String.at(&2, round(&1)) end)
```

Consistency & Ethos

Arrows



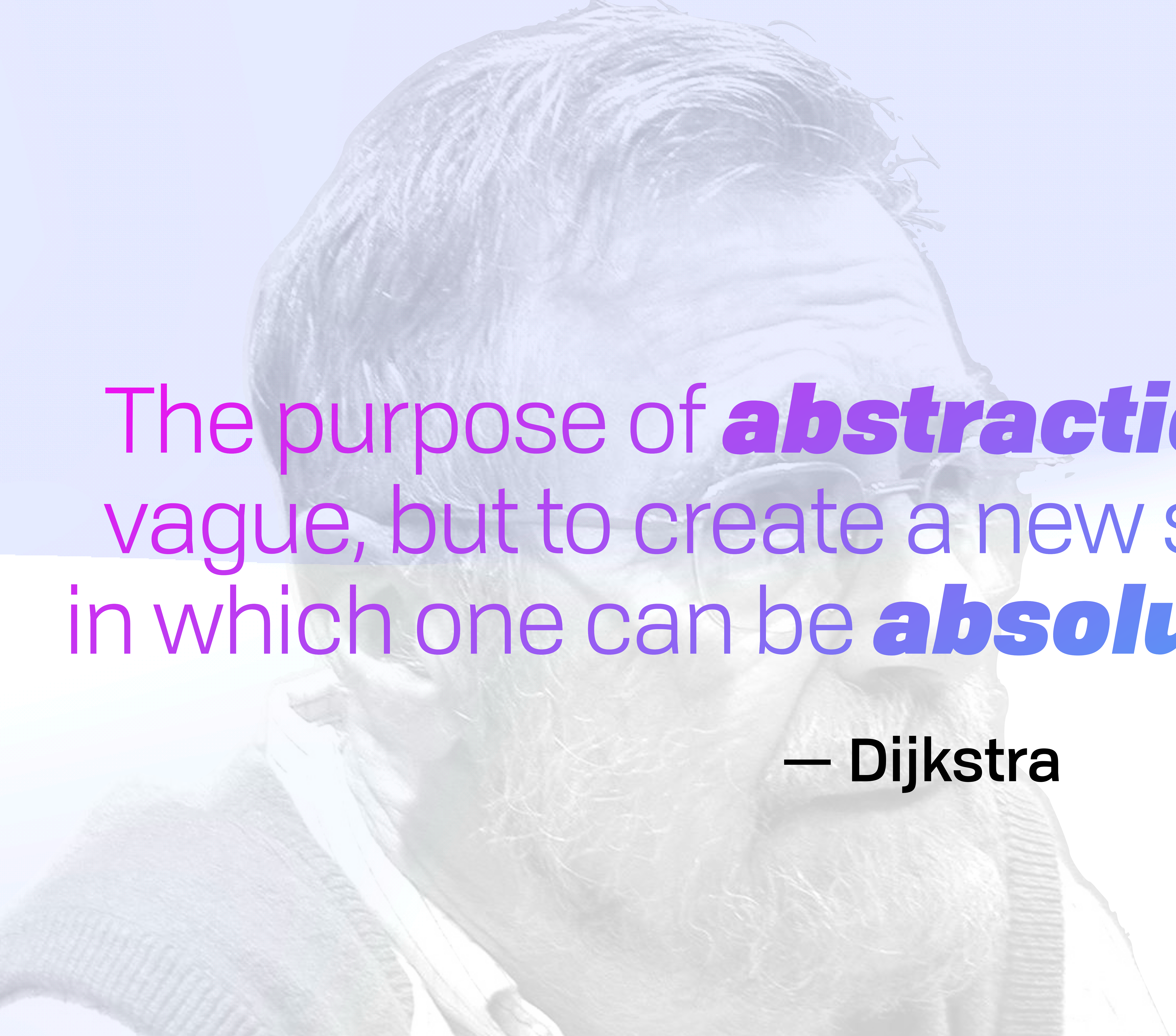


```
arrow_diagram =
  fanout(fn x -> x / 5 end, fn y -> y + 1 end
                            <~> (fanout(&inspect/1, fn z -> z * z end))
                            <~> unsplit(fn (a, b) -> "#{b}#{a}" end)
  )
  <~> unsplit(&String.at(&2, round(&1)) end)
```

Functional & Principled

Functor Tower

🗼

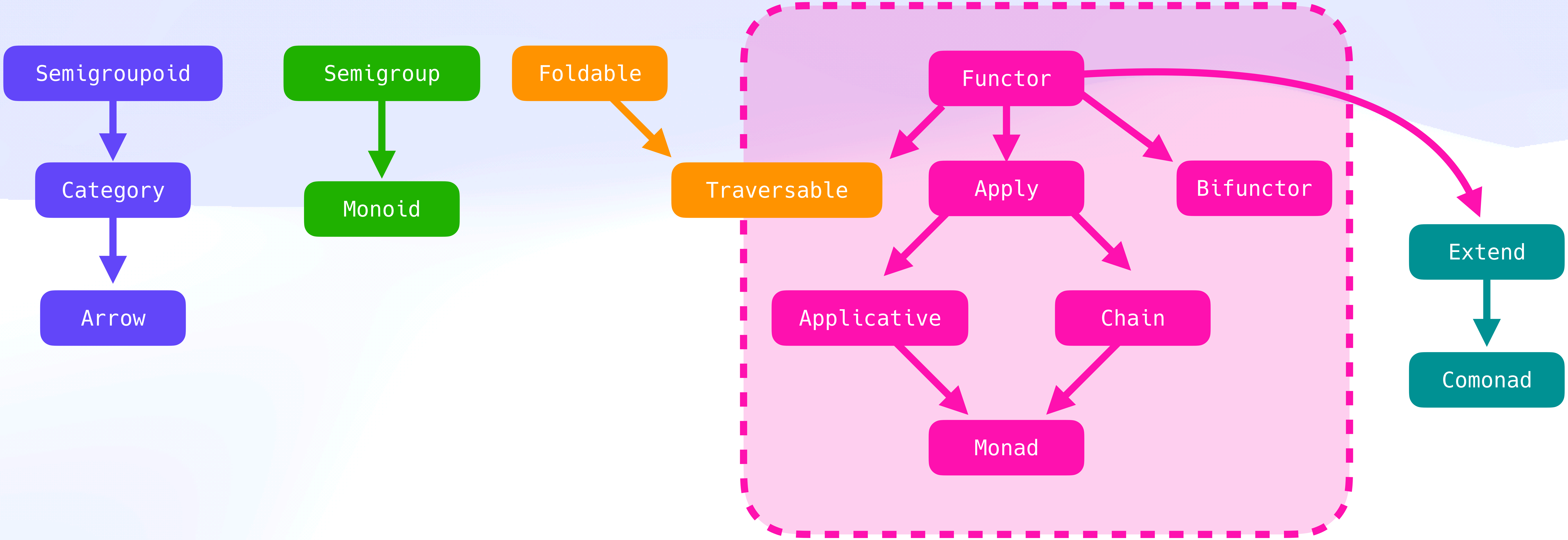


The purpose of ***abstraction*** is not to be vague, but to create a new semantic level in which one can be ***absolutely precise***

— Dijkstra

Design Patterns

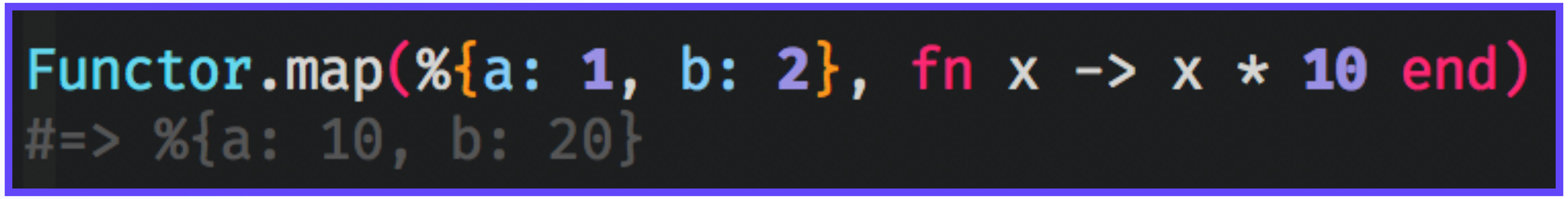
Witchcraft v1.0 Hierarchy



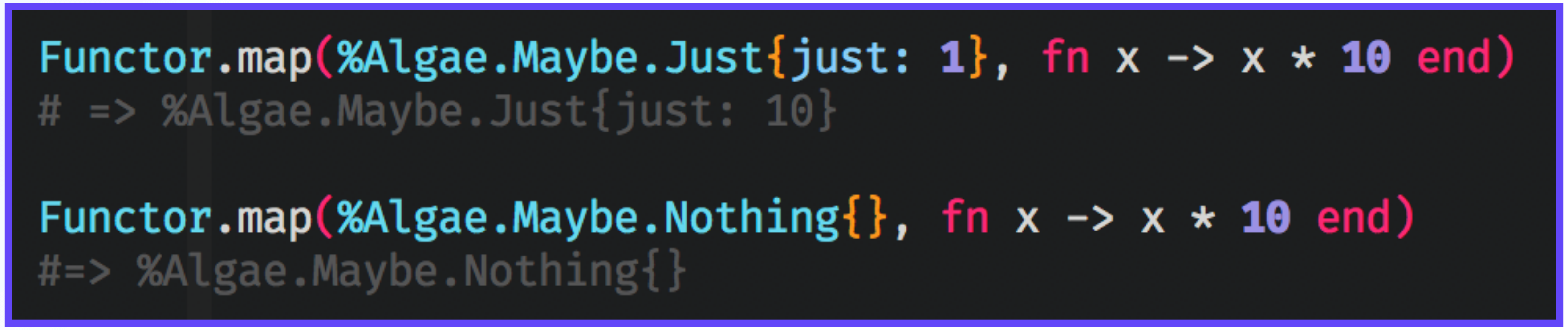
Consistency & Ethos

Functor

- Provides `map/2` (~>), but different from `Enum`
- Always returns the same type of data
- No more manual `Enum.map(…)|> Enum.into(…)`



```
Functor.map(%{a: 1, b: 2}, fn x -> x * 10 end)
#=> %{a: 10, b: 20}
```

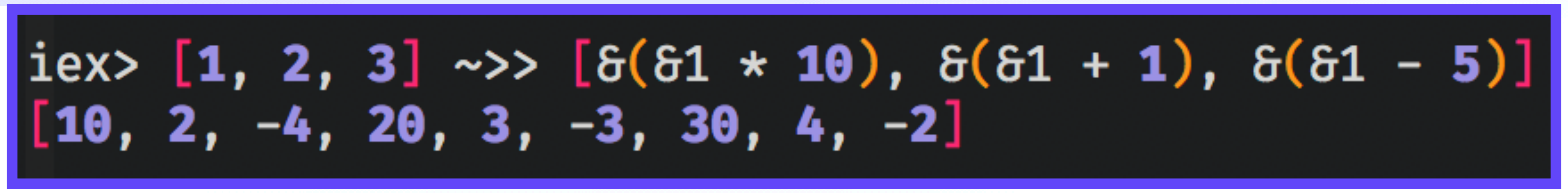


```
Functor.map(%Algae.Maybe.Just{just: 1}, fn x -> x * 10 end)
# => %Algae.Maybe.Just{just: 10}

Functor.map(%Algae.Maybe.Nothing{}, fn x -> x * 10 end)
#=> %Algae.Maybe.Nothing{}
```

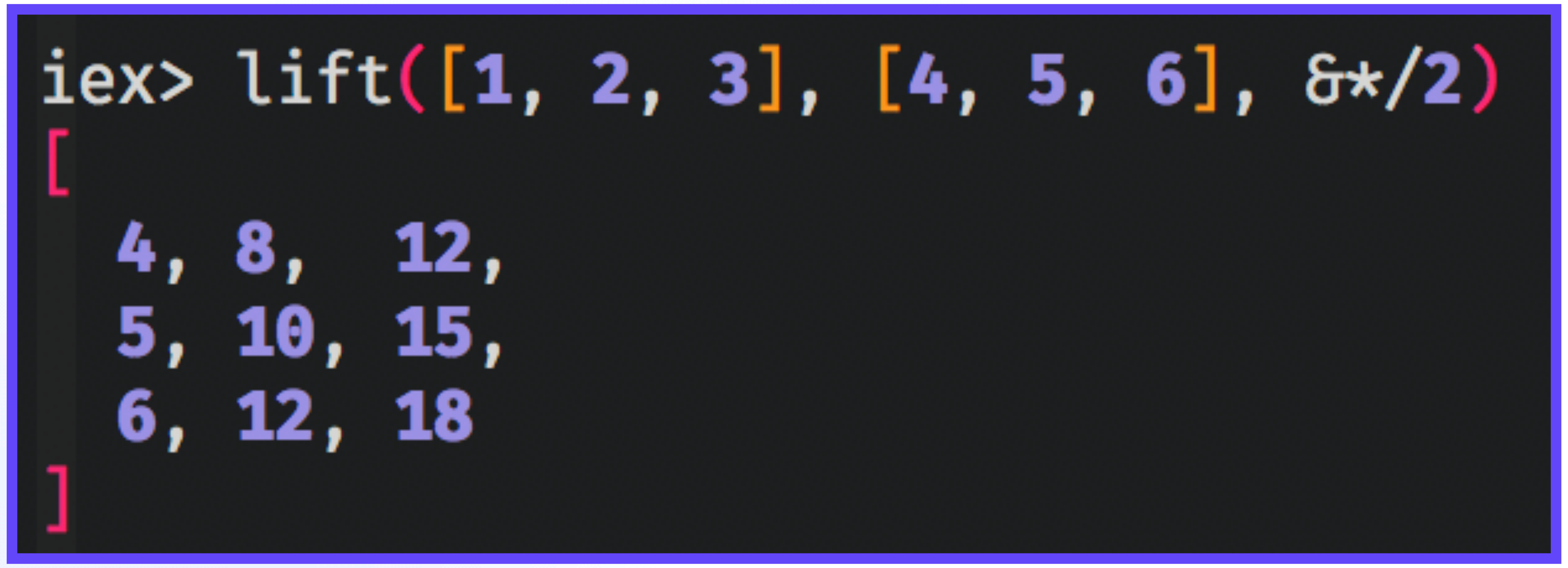
Consistency & Ethos

Apply

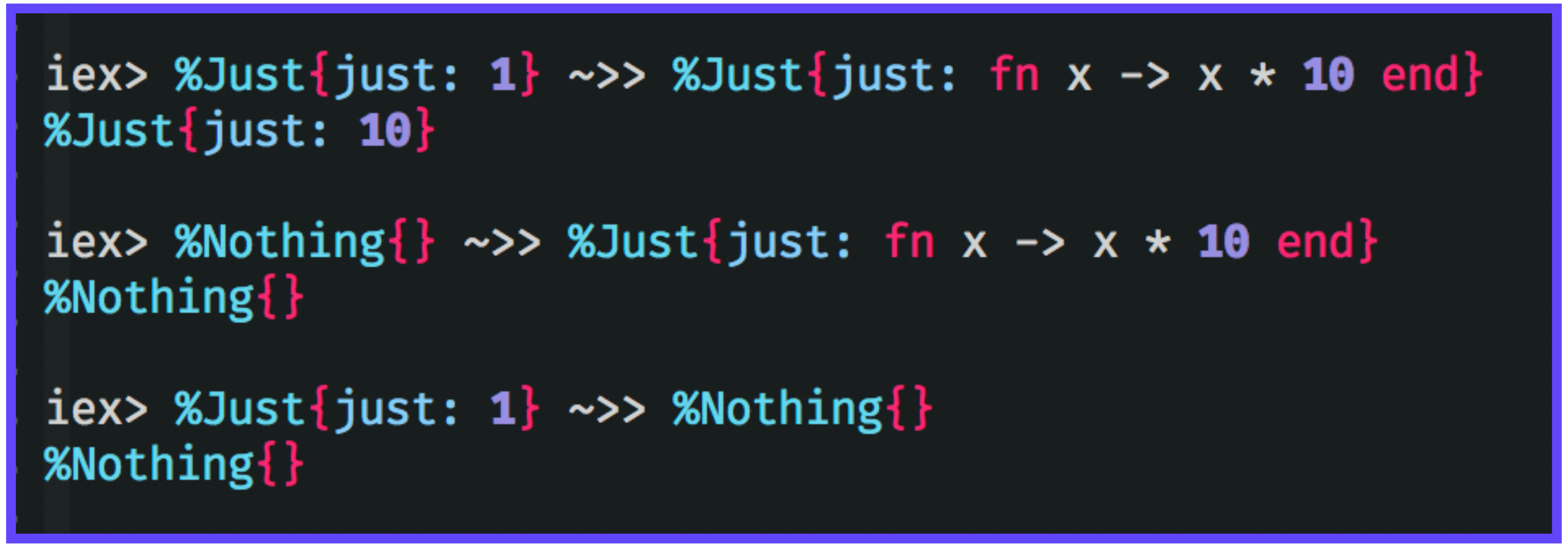
- Provides `convey/2` and `ap/2`
- Embellishes basic function application
- Specific embellishment changes per data type



```
iex> [1, 2, 3] ~>> [&(&1 * 10), &(&1 + 1), &(&1 - 5)]
[10, 2, -4, 20, 3, -3, 30, 4, -2]
```



```
iex> lift([1, 2, 3], [4, 5, 6], &*/2)
[
  4, 8,  12,
  5, 10, 15,
  6, 12, 18
]
```

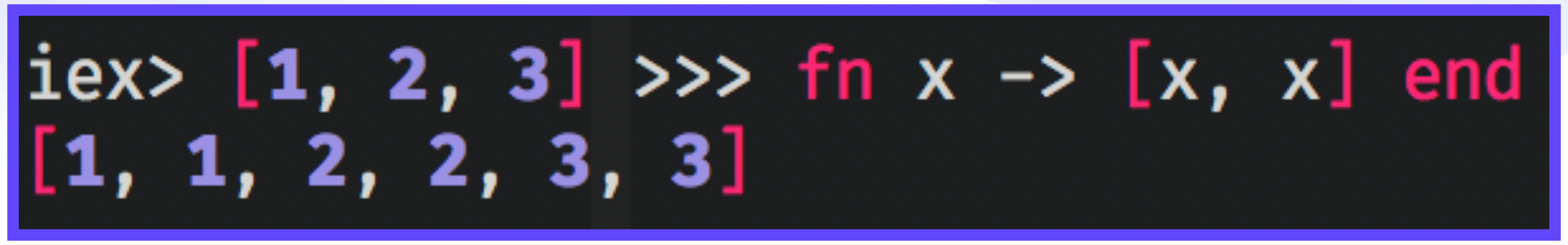


```
iex> %Just{just: 1} ~>> %Just{just: fn x -> x * 10 end}
%Just{just: 10}

iex> %Nothing{} ~>> %Just{just: fn x -> x * 10 end}
%Nothing{}

iex> %Just{just: 1} ~>> %Nothing{}
%Nothing{}
```

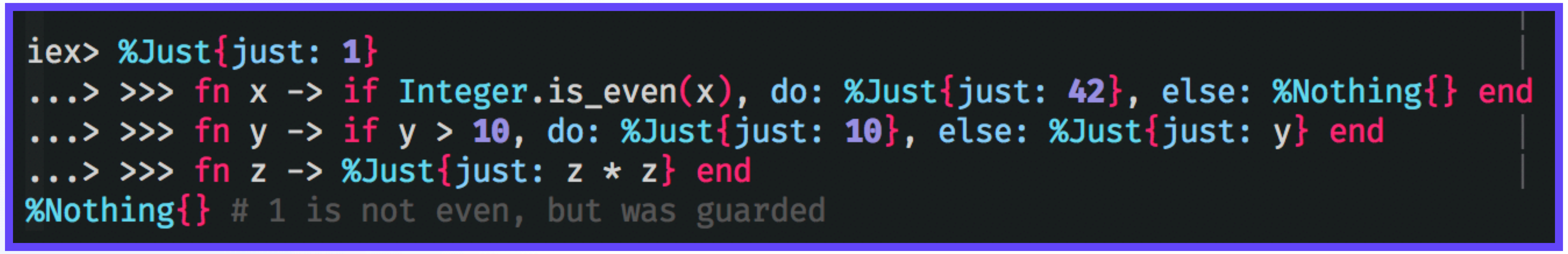
Consistency & Ethos

Chain: Functions to Actions

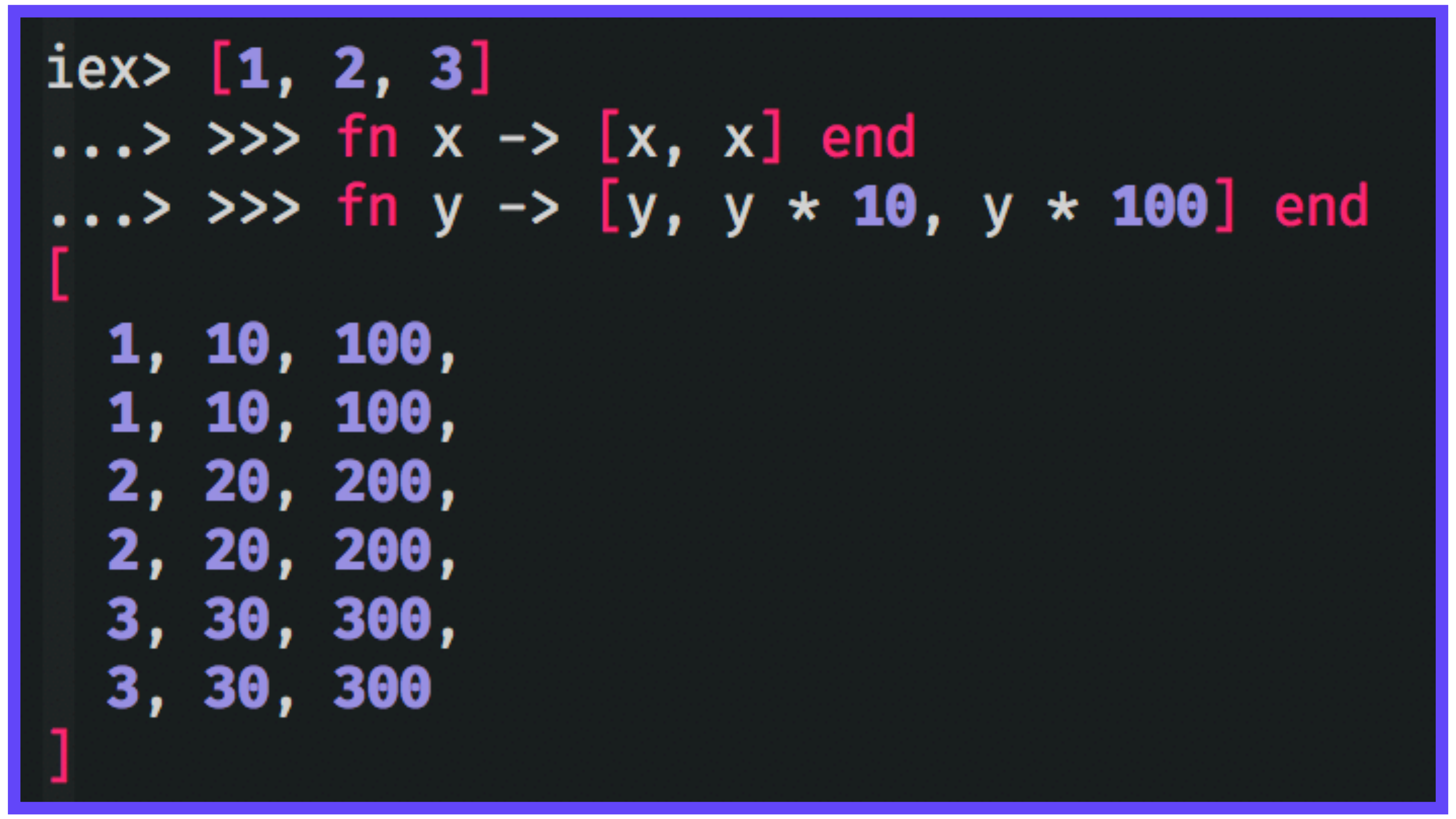
- Like Apply & Applicative, but with a special “linking” function
- Take raw value
- Do something to it
- Put the result into the original datatype
- Makes it easy to chain functions in a context



```
iex> [1, 2, 3] >>> fn x -> [x, x] end
[1, 1, 2, 2, 3, 3]
```



```
iex> %Just{just: 1}
...> >>> fn x -> if Integer.is_even(x), do: %Just{just: 42}, else: %Nothing{} end
...> >>> fn y -> if y > 10, do: %Just{just: 10}, else: %Just{just: y} end
...> >>> fn z -> %Just{just: z * z} end
%Nothing{} # 1 is not even, but was guarded
```



```
iex> [1, 2, 3]
...> >>> fn x -> [x, x] end
...> >>> fn y -> [y, y * 10, y * 100] end
[
  1, 10, 100,
  1, 10, 100,
  2, 20, 200,
  2, 20, 200,
  3, 30, 300,
  3, 30, 300
]
```

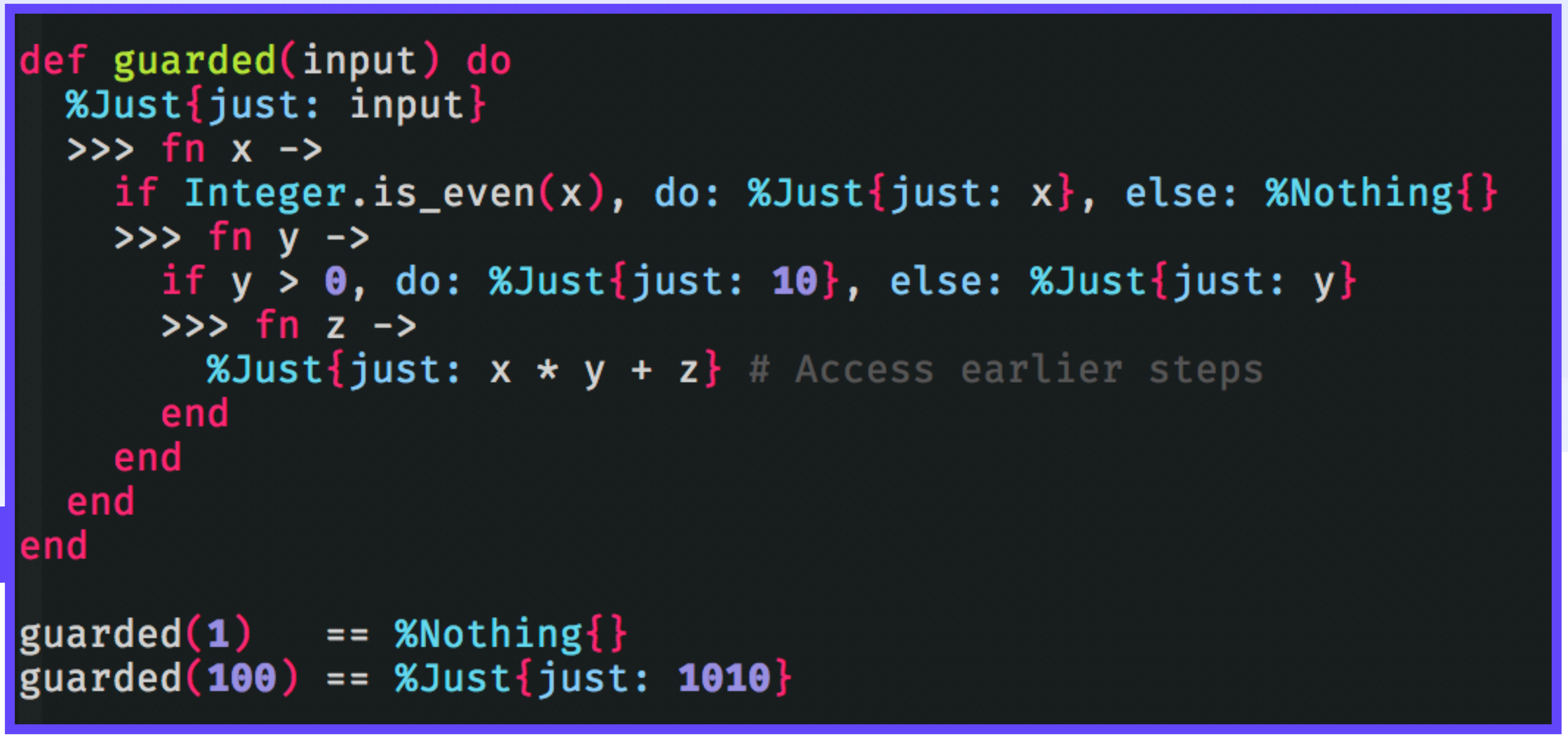
Consistency & Ethos

Chaining With do-***Notation***

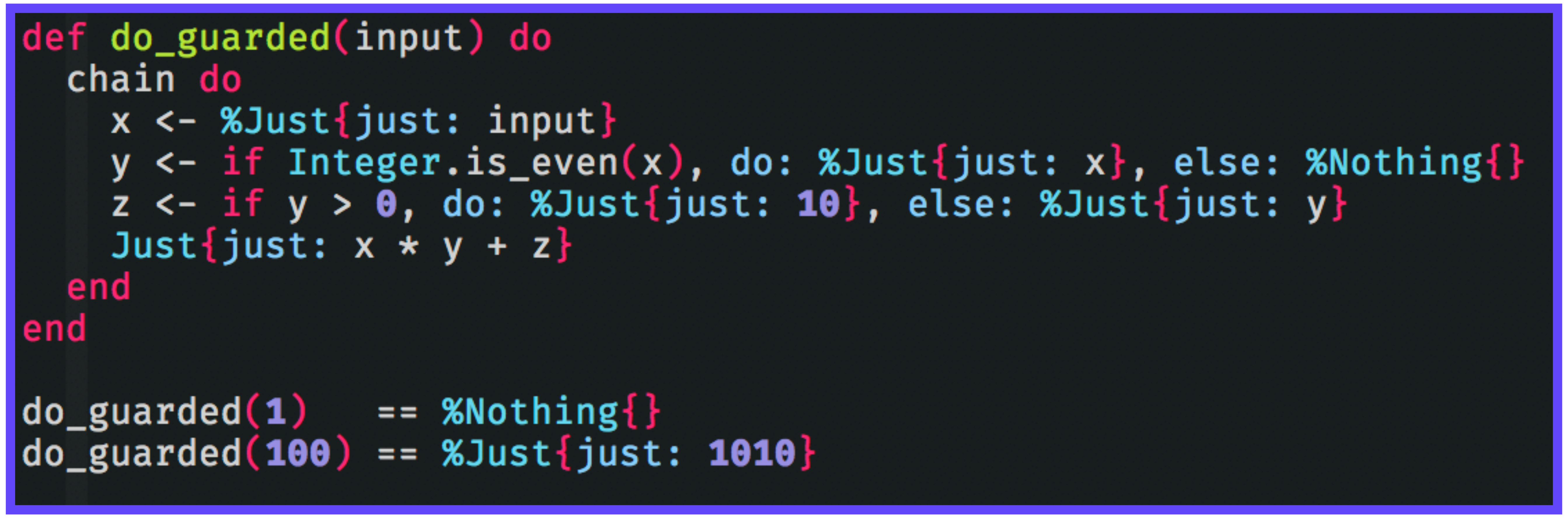
- Macro to “linearize” chains

- Gives us back an operational feel

- Great DSLs (seen shortly)



```
def guarded(input) do
  %Just{just: input}
  >>> fn x ->
    if Integer.is_even(x), do: %Just{just: x}, else: %Nothing{}
    >>> fn y ->
      if y > 0, do: %Just{just: 10}, else: %Just{just: y}
      >>> fn z ->
        %Just{just: x * y + z} # Access earlier steps
      end
    end
  end
end

guarded(1)   == %Nothing{}
guarded(100) == %Just{just: 1010}
```



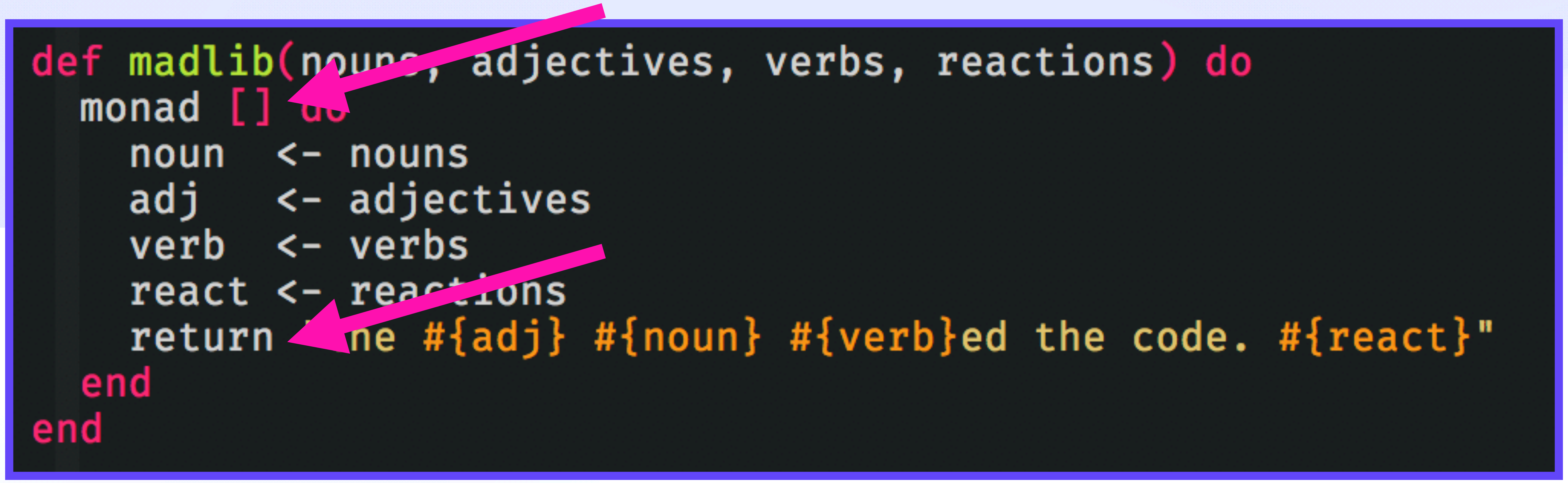
```
def do_guarded(input) do
  chain do
    x <- %Just{just: input}
    y <- if Integer.is_even(x), do: %Just{just: x}, else: %Nothing{}
    z <- if y > 0, do: %Just{just: 10}, else: %Just{just: y}
    Just{just: x * y + z}
  end
end

do_guarded(1)   == %Nothing{}
do_guarded(100) == %Just{just: 1010}
```

Consistency & Ethos

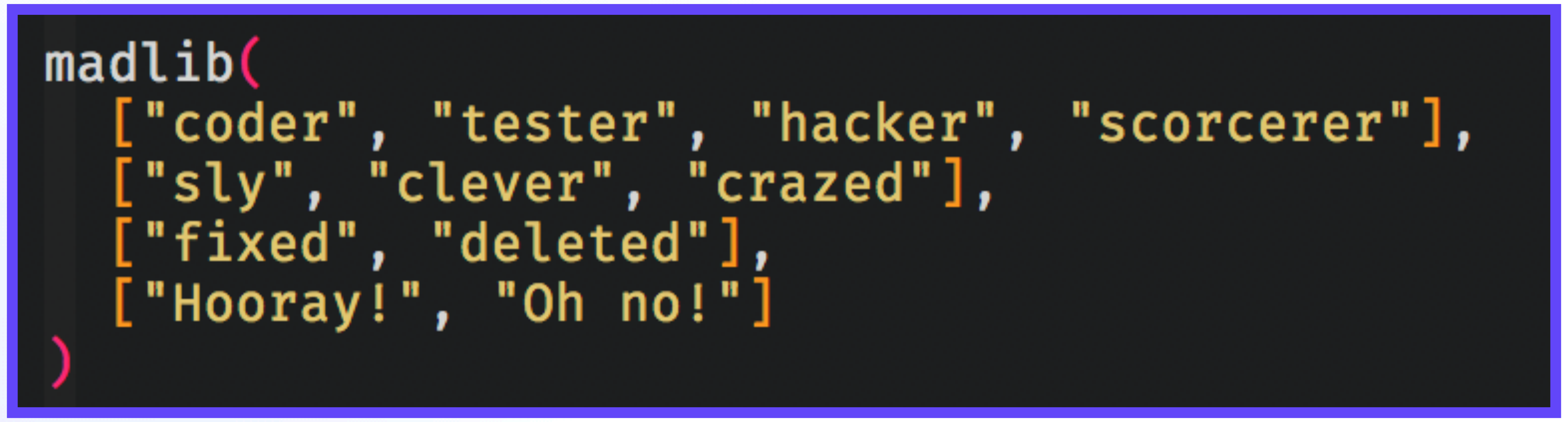
Monadic do-Notation

- Need to specify the data type

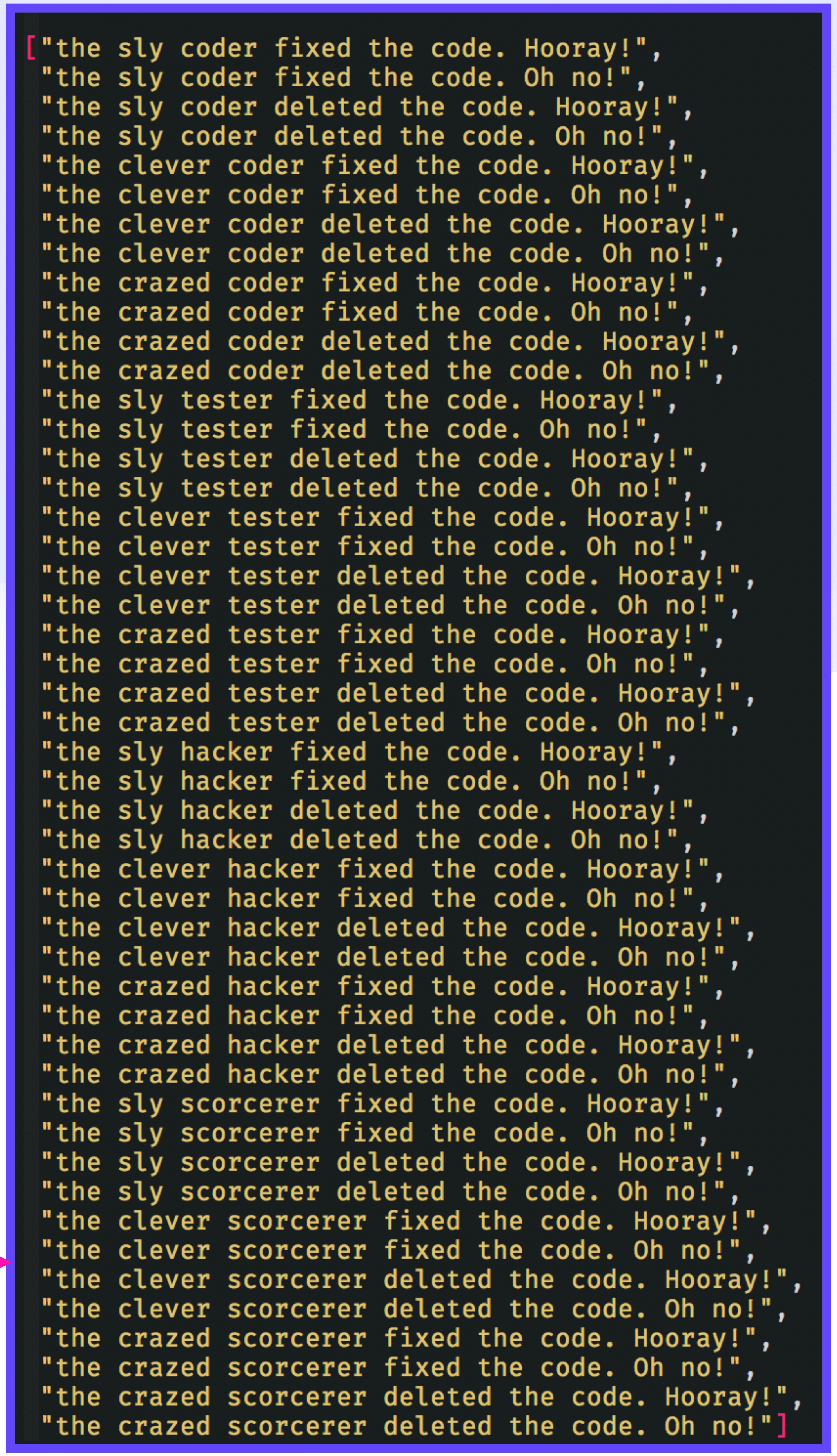
- Just add return (specialized Applicative.of/2)



```
def madlib(nouns, adjectives, verbs, reactions) do
  monad [] do
    noun  <- nouns
    adj   <- adjectives
    verb  <- verbs
    react <- reactions
    return "the #{adj} #{noun} #{verb}ed the code. #{react}"
  end
end
```



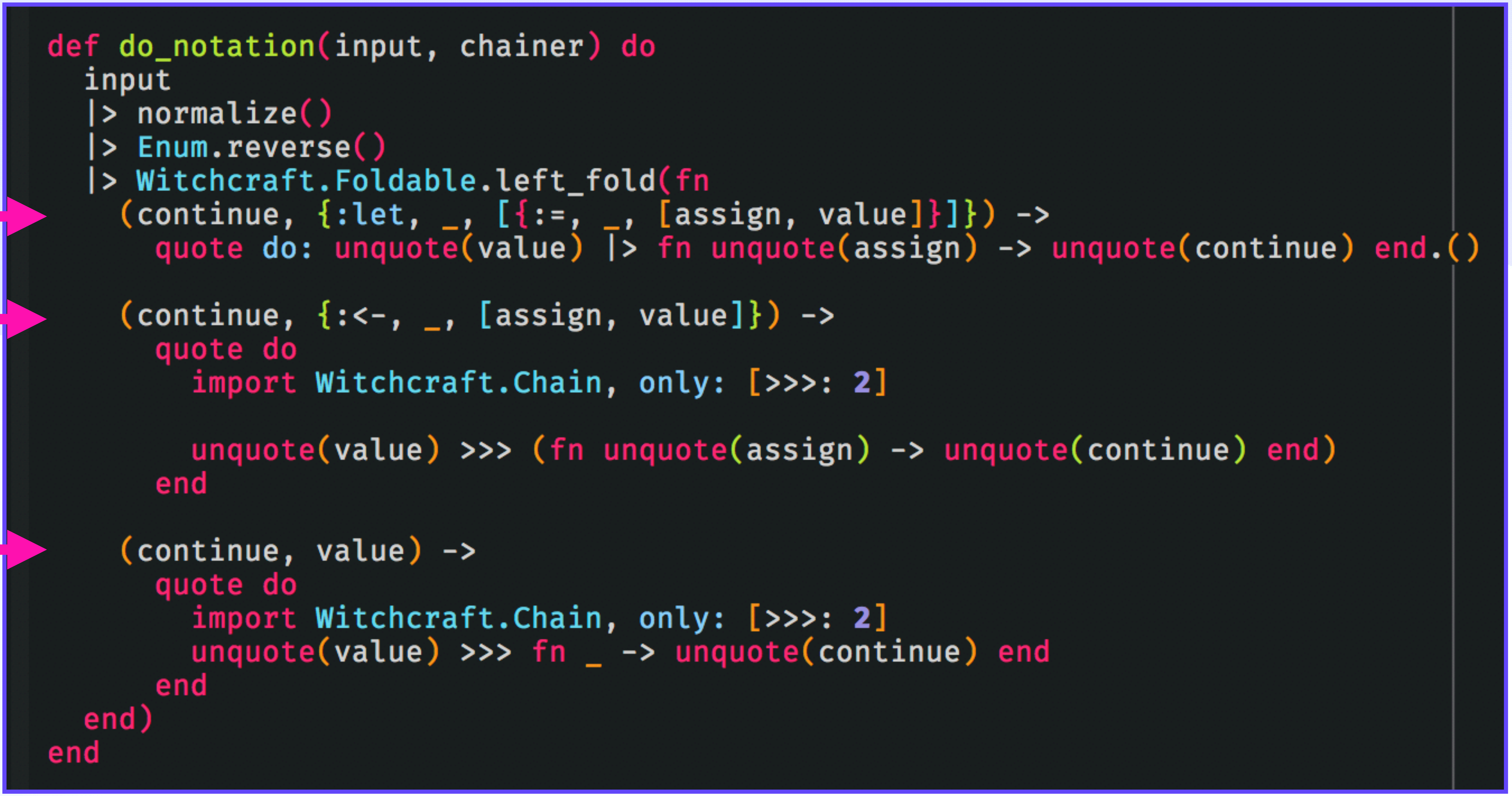
```
madlib(
  ["coder", "tester", "hacker", "scorcerer"],
  ["sly", "clever", "crazed"],
  ["fixed", "deleted"],
  ["Hooray!", "Oh no!"]
)
```



```
["the sly coder fixed the code. Hooray!",
 "the sly coder fixed the code. Oh no!",
 "the sly coder deleted the code. Hooray!",
 "the sly coder deleted the code. Oh no!",
 "the clever coder fixed the code. Hooray!",
 "the clever coder fixed the code. Oh no!",
 "the clever coder deleted the code. Hooray!",
 "the clever coder deleted the code. Oh no!",
 "the crazed coder fixed the code. Hooray!",
 "the crazed coder fixed the code. Oh no!",
 "the crazed coder deleted the code. Hooray!",
 "the crazed coder deleted the code. Oh no!",
 "the sly tester fixed the code. Hooray!",
 "the sly tester fixed the code. Oh no!",
 "the sly tester deleted the code. Hooray!",
 "the sly tester deleted the code. Oh no!",
 "the clever tester fixed the code. Hooray!",
 "the clever tester fixed the code. Oh no!",
 "the clever tester deleted the code. Hooray!",
 "the clever tester deleted the code. Oh no!",
 "the crazed tester fixed the code. Hooray!",
 "the crazed tester fixed the code. Oh no!",
 "the crazed tester deleted the code. Hooray!",
 "the crazed tester deleted the code. Oh no!",
 "the sly hacker fixed the code. Hooray!",
 "the sly hacker fixed the code. Oh no!",
 "the sly hacker deleted the code. Hooray!",
 "the sly hacker deleted the code. Oh no!",
 "the clever hacker fixed the code. Hooray!",
 "the clever hacker fixed the code. Oh no!",
 "the clever hacker deleted the code. Hooray!",
 "the clever hacker deleted the code. Oh no!",
 "the crazed hacker fixed the code. Hooray!",
 "the crazed hacker fixed the code. Oh no!",
 "the crazed hacker deleted the code. Hooray!",
 "the crazed hacker deleted the code. Oh no!",
 "the sly scorcerer fixed the code. Hooray!",
 "the sly scorcerer fixed the code. Oh no!",
 "the sly scorcerer deleted the code. Hooray!",
 "the sly scorcerer deleted the code. Oh no!",
 "the clever scorcerer fixed the code. Hooray!",
 "the clever scorcerer fixed the code. Oh no!",
 "the clever scorcerer deleted the code. Hooray!",
 "the clever scorcerer deleted the code. Oh no!",
 "the crazed scorcerer fixed the code. Hooray!",
 "the crazed scorcerer fixed the code. Oh no!",
 "the crazed scorcerer deleted the code. Hooray!",
 "the crazed scorcerer deleted the code. Oh no!"]
```

Consistency & Ethos

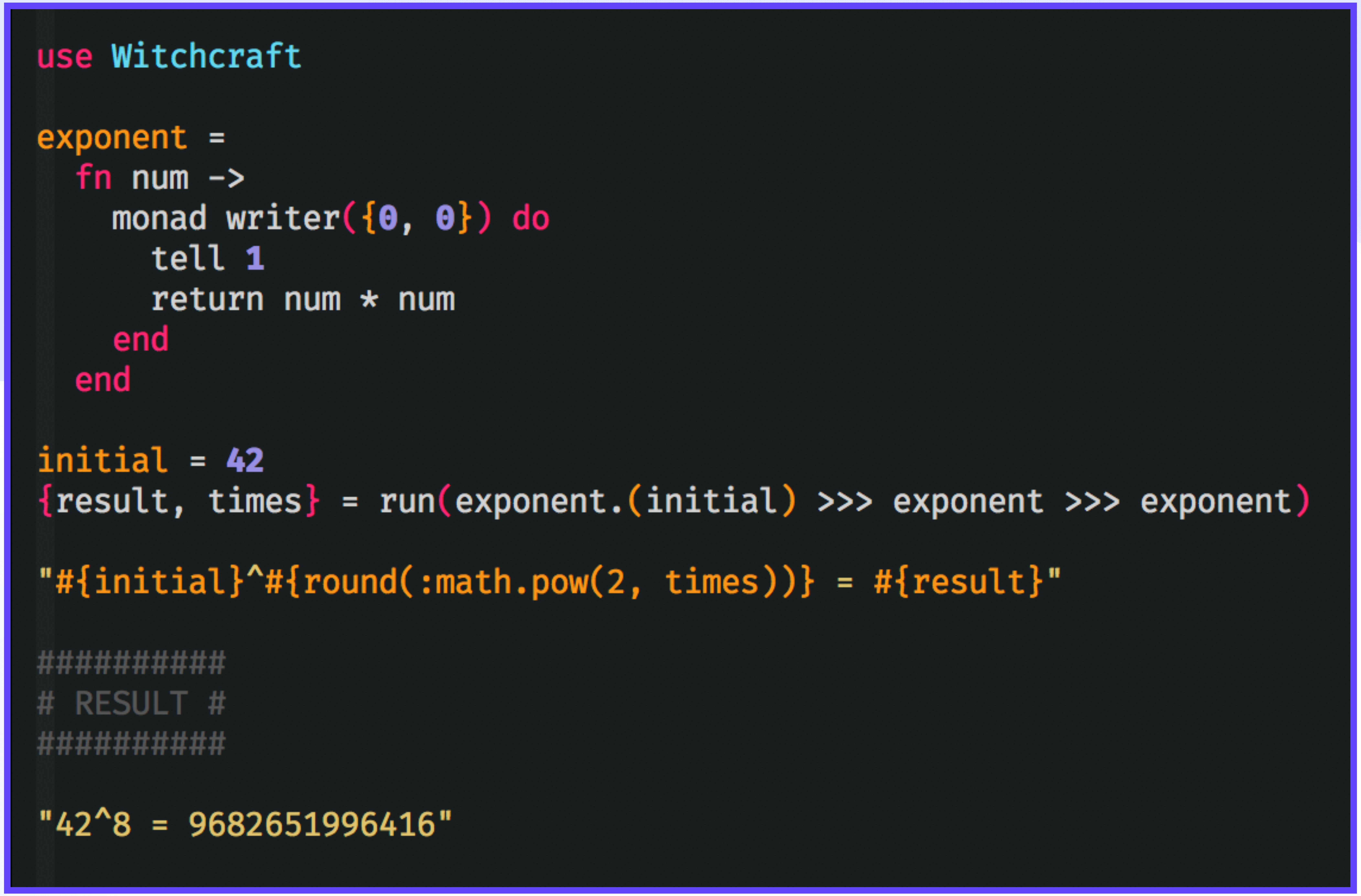
do-*Notation Implementation*



```
def do_notation(input, chainer) do
  input
  |> normalize()
  |> Enum.reverse()
  |> Witchcraft.Foldable.left_fold(fn
    (continue, {:let, _, [{:=, _, [assign, value]}]}) ->
      quote do: unquote(value) |> fn unquote(assign) -> unquote(continue) end.()

    (continue, {:<-, _, [assign, value]}) ->
      quote do
        import Witchcraft.Chain, only: [>>>: 2]

        unquote(value) >>> (fn unquote(assign) -> unquote(continue) end)
      end

    (continue, value) ->
      quote do
        import Witchcraft.Chain, only: [>>>: 2]
        unquote(value) >>> fn _ -> unquote(continue) end
      end
  end)
end
```

Consistency & Ethos

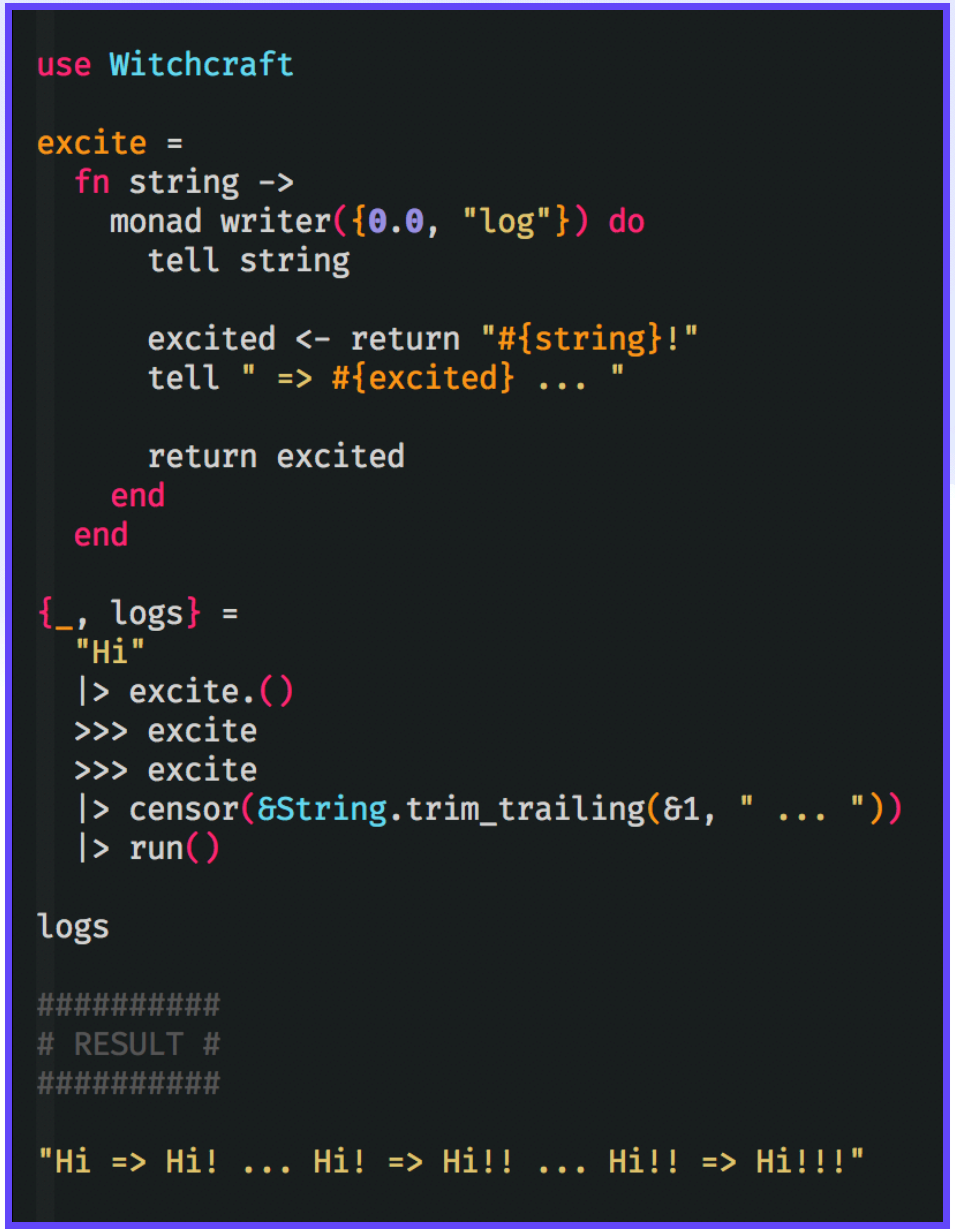
Writer Monad



```
use Witchcraft

exponent =
  fn num ->
    monad writer({0, 0}) do
      tell 1
      return num * num
    end
  end

initial = 42
{result, times} = run(exponent.(initial) >>> exponent >>> exponent)

"#{initial}^#{round(:math.pow(2, times))} = #{result}"

##########
# RESULT #
##########

"42^8 = 9682651996416"
```

Consistency & Ethos

Writer Monad



```
use Witchcraft

excite =
  fn string ->
    monad writer({0.0, "log"}) do
      tell string

      excited <- return "#{string}!"
      tell " => #{excited} ... "

      return excited
    end
  end

{_, logs} =
  "Hi"
  |> excite.()
  >>> excite
  >>> excite
  |> censor(&String.trim_trailing(&1, " ... "))
  |> run()

logs

##########
# RESULT #
##########

"Hi => Hi! ... Hi! => Hi!! ... Hi!! => Hi!!!"
```

Some more uncategorised observations

Lessons Learned

⏮

Lessons Learned

Build It & They Will Come

- ...but often difficult to coordinate timing, esp. with maintainer burnout
- Get more people involved very early — earlier than you think you need to!
- Actively hand off credit to others
- A "prior art" section in READMEs diffuses many conflicts
- All things come down to people and governance

Lessons Learned

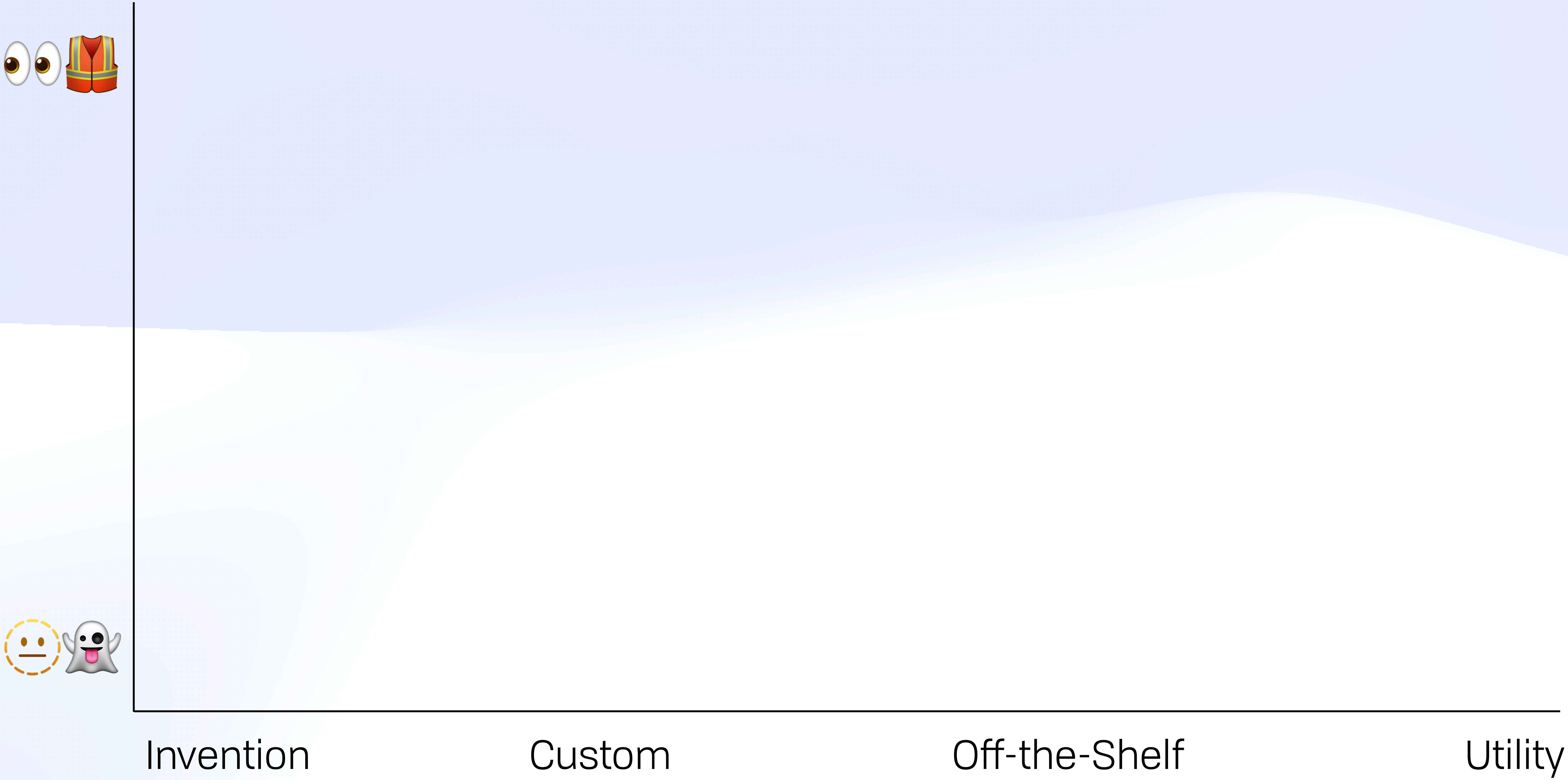
Wardley Stages

Lessons Learned

Wardley Stages

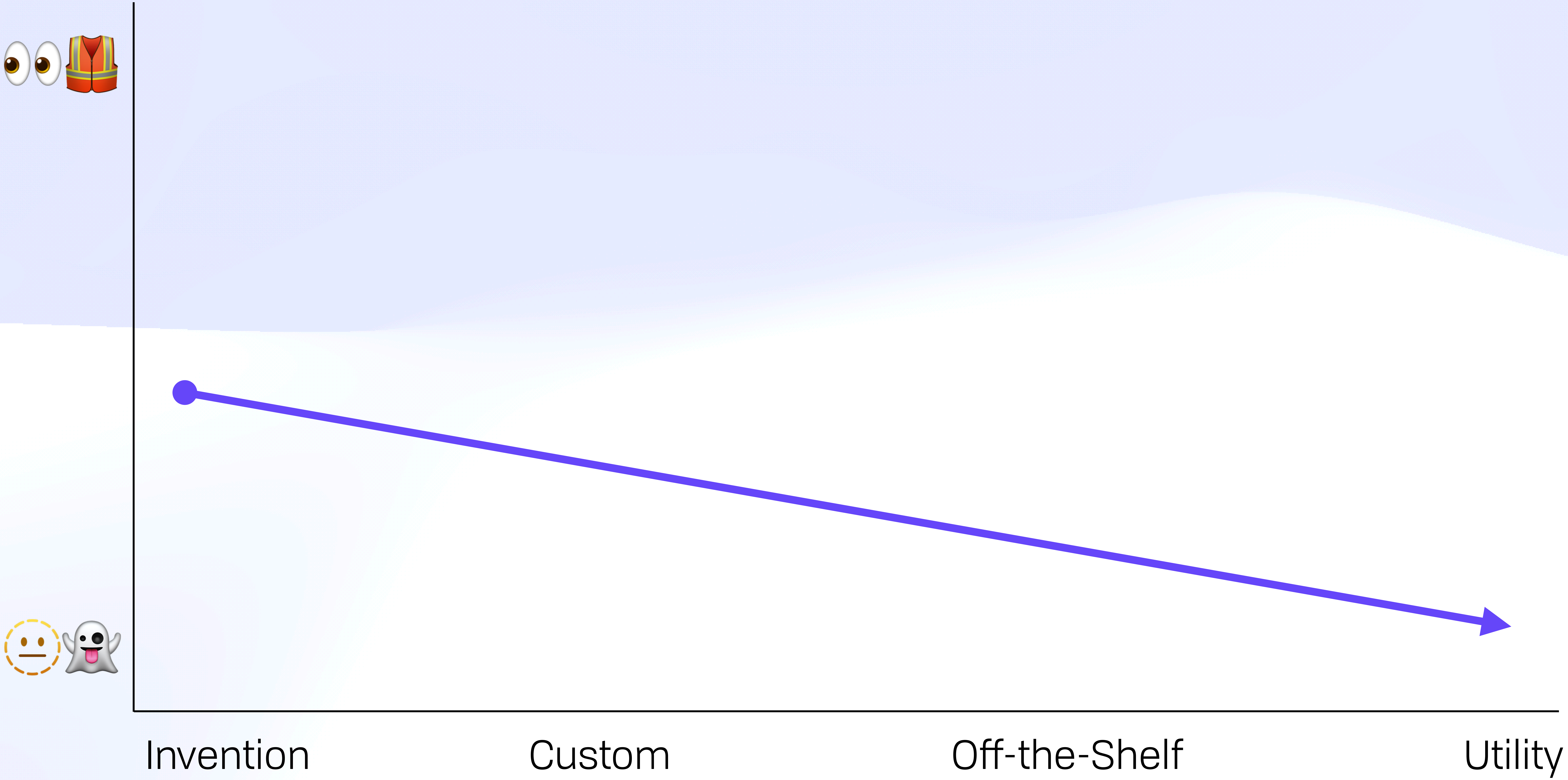
Lessons Learned

Wardley Stages



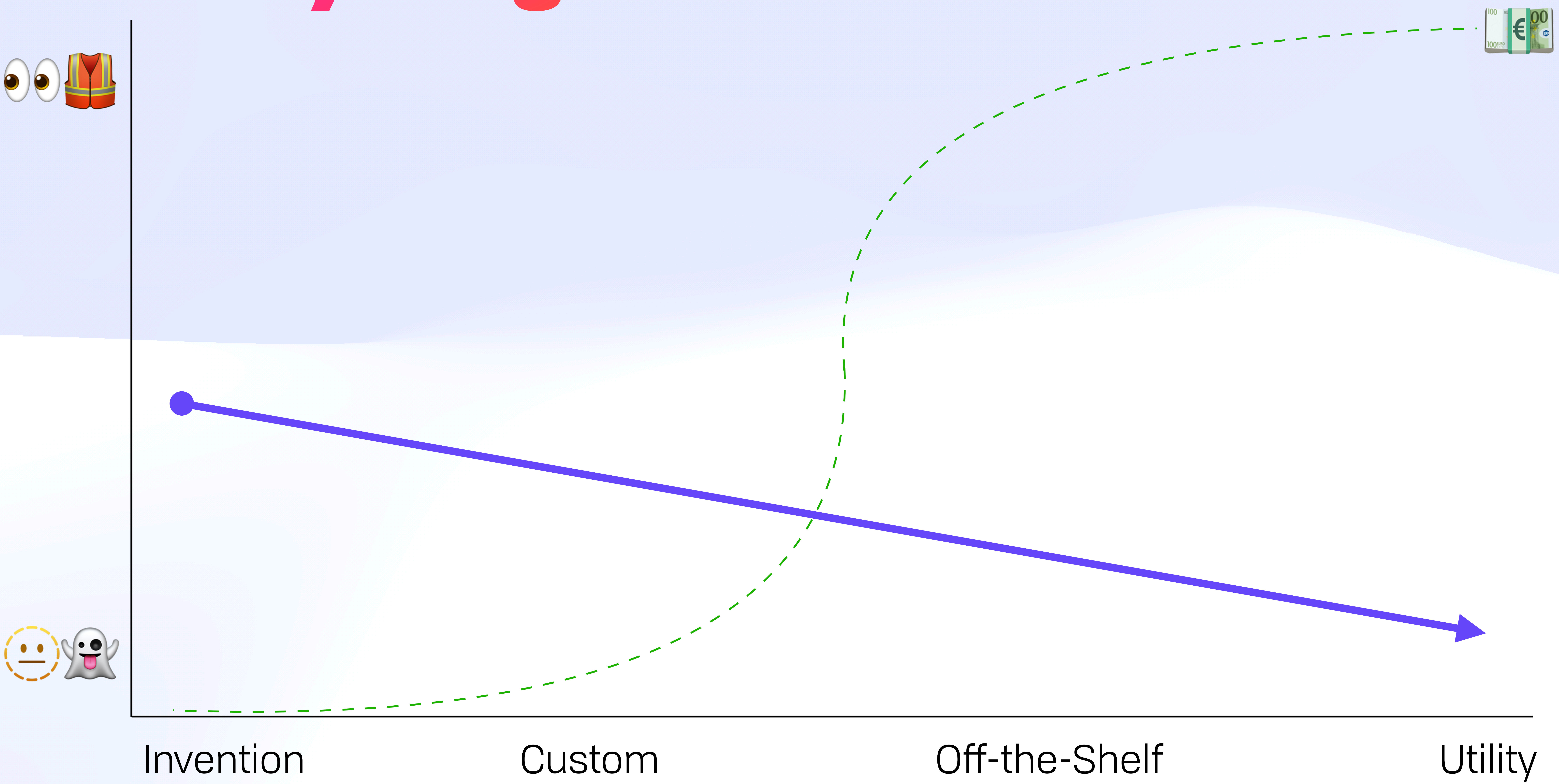
Lessons Learned

Wardley Stages



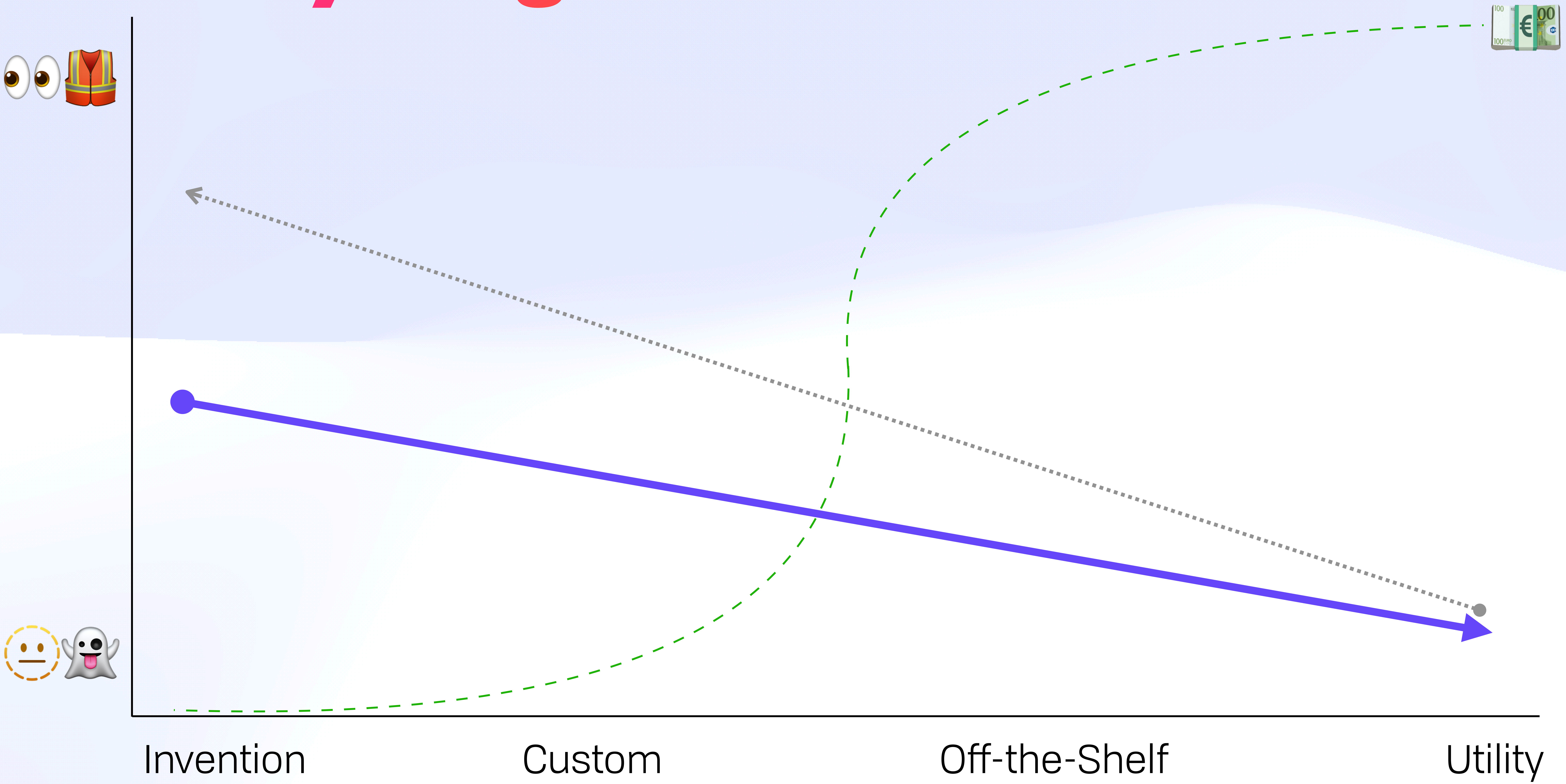
Lessons Learned

Wardley Stages



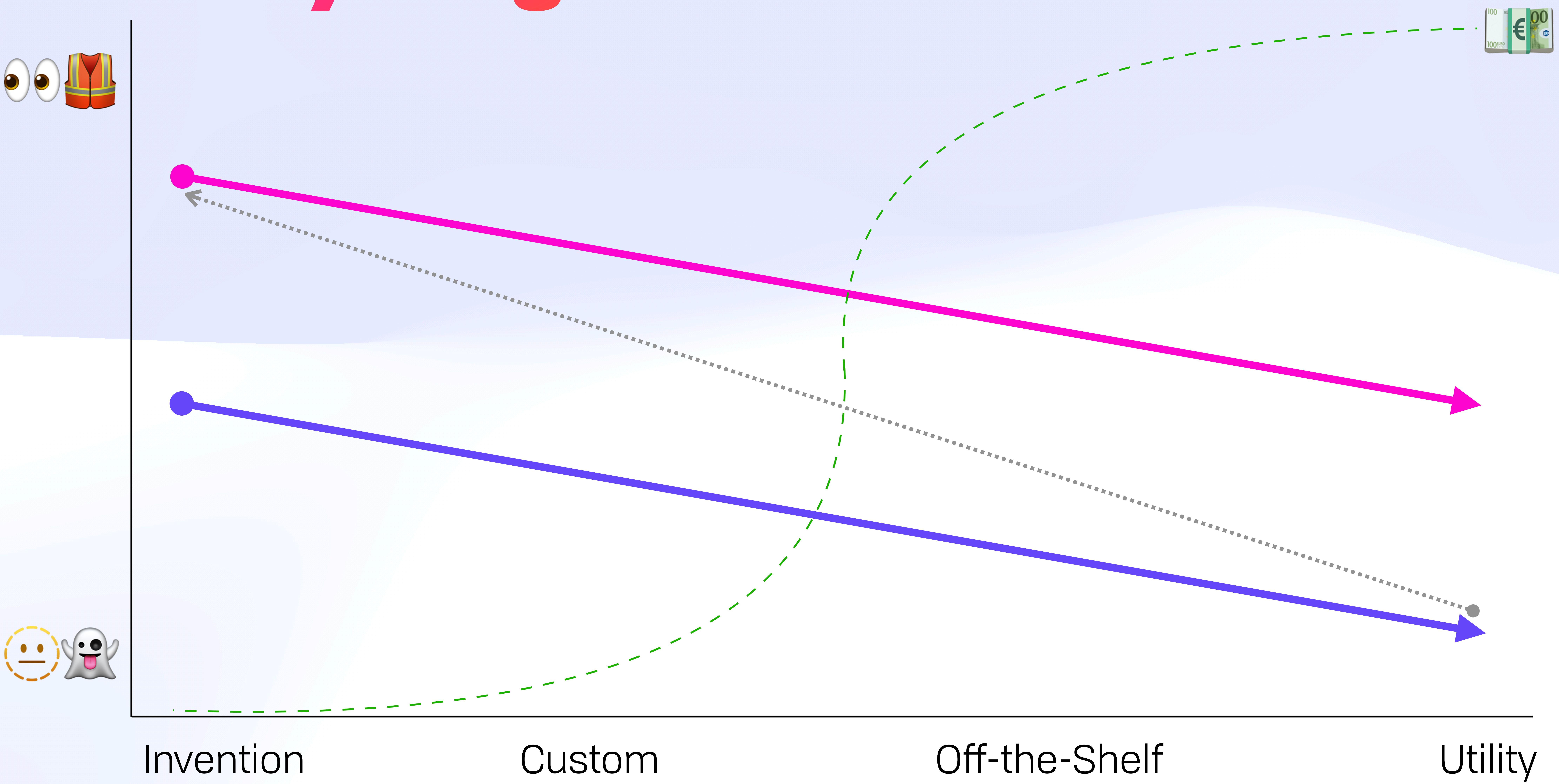
Lessons Learned

Wardley Stages



Lessons Learned

Wardley Stages



Lessons Learned

Approach in 2024?

- Turnstile is pretty great
    - Type Systems as Macros
- (Co)effect systems, capabilities
- Branding was always important, but even more now

🎉 Thank You, Fun Prog Sweden 🇸🇪

📝 notes.brooklynzelenka.com

📧 hello@brooklynzelenka.com

🦋 bsky.app/profile/expede.wtf

🐘 @expede@octodon.social