



# THE *TYRANNY* OF STRUCTURELESSNESS

HOW MORE MEANINGFUL CODE CAN MAKE YOUR PROJECT MORE RESILIENT & MAINTAINABLE

*VIRTUAL EDITION*



# THE TYRANNY OF STRUCTURELESSNESS

For a number of years I have been familiar with the observation that **the quality of programmers is a decreasing function of the density of GOTO statements** in the programs they produce



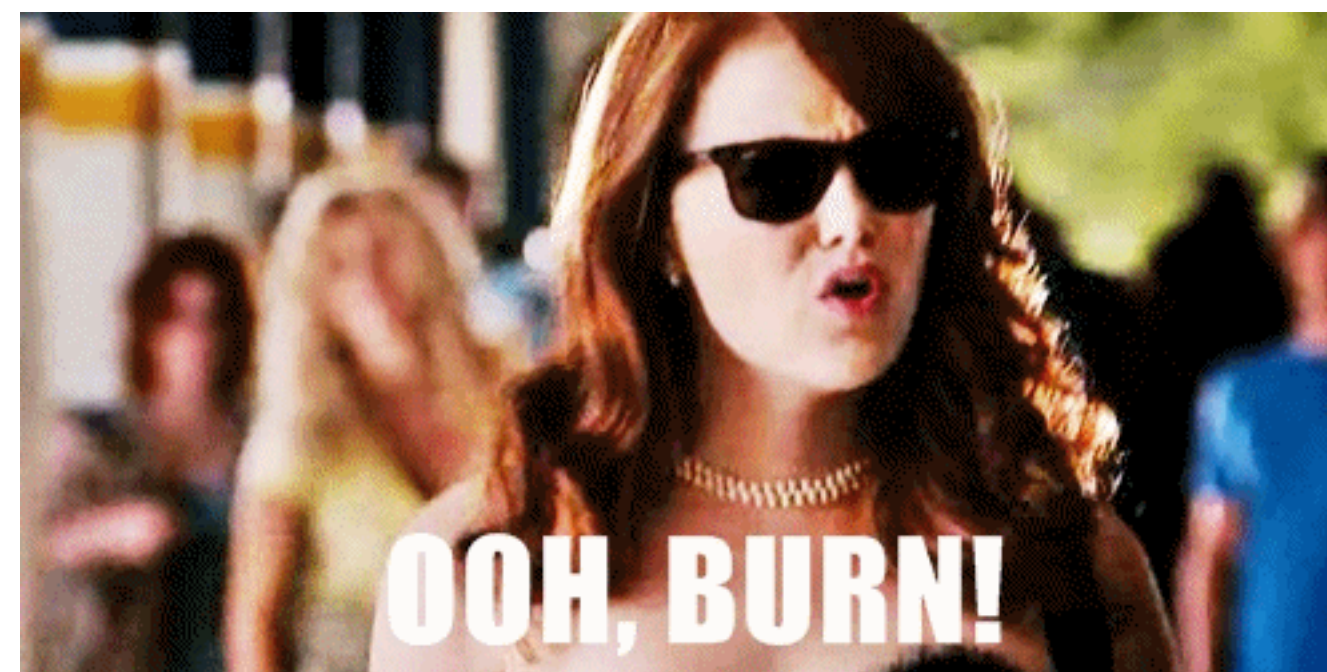
*EDSGER DIJKSTRA*

# THE TYRANNY OF STRUCTURELESSNESS

For a number of years I have been familiar with the observation that **the quality of programmers is a decreasing function of the density of GOTO statements** in the programs they produce



*EDSGER DIJKSTRA*



# THE TYRANNY OF STRUCTURELESSNESS

What's she on about? Elixir doesn't have GOTOs...



*THIS AUDIENCE*



THE TYRANNY OF STRUCTURELESSNESS  
BROOKLYN ZELENIKA, @EXPEDE





# THE TYRANNY OF STRUCTURELESSNESS

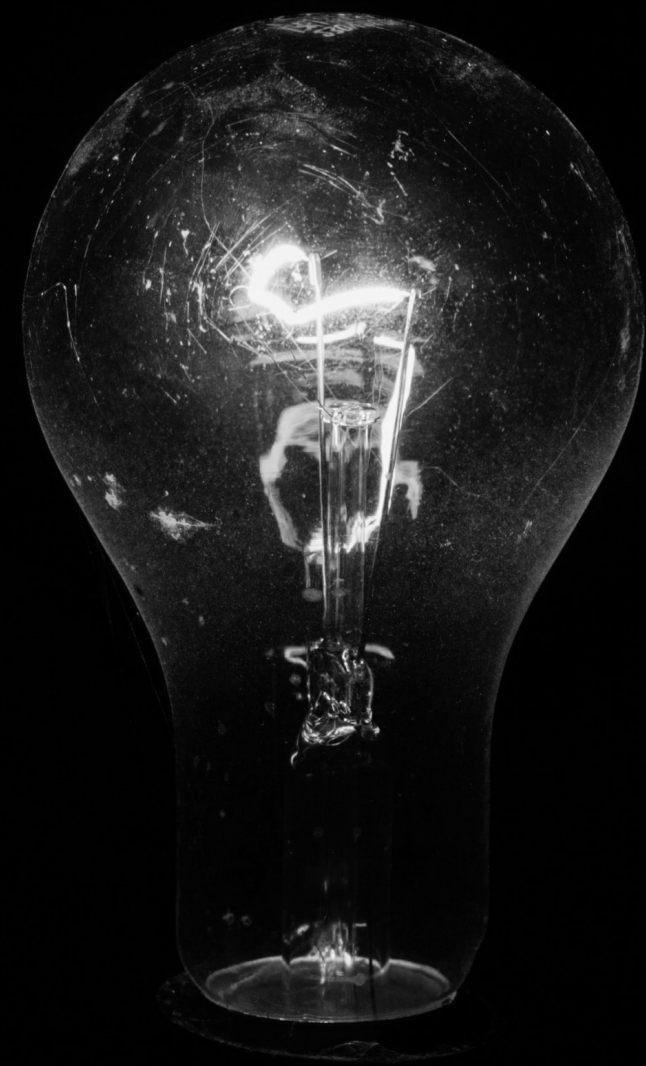
## BROOKLYN ZELENIKA, @EXPEDE

- Cofounder/CTO at Fission
  - <https://fission.codes>
  - Make DevOps & Backend obsolete 🐱
  - Spending a *lot* of time with IPFS, DIDs, CRDTs
  - Want to hear more? *Berlin FP online meetup June 2*
- PLT & VM enthusiast
- Prev. Ethereum Core Developer
- Primary author of Witchcraft Suite & Exceptional
- This is a version of CodeBEAM Amsterdam 2019 keynote
  - Whova app for Q&A afterwards

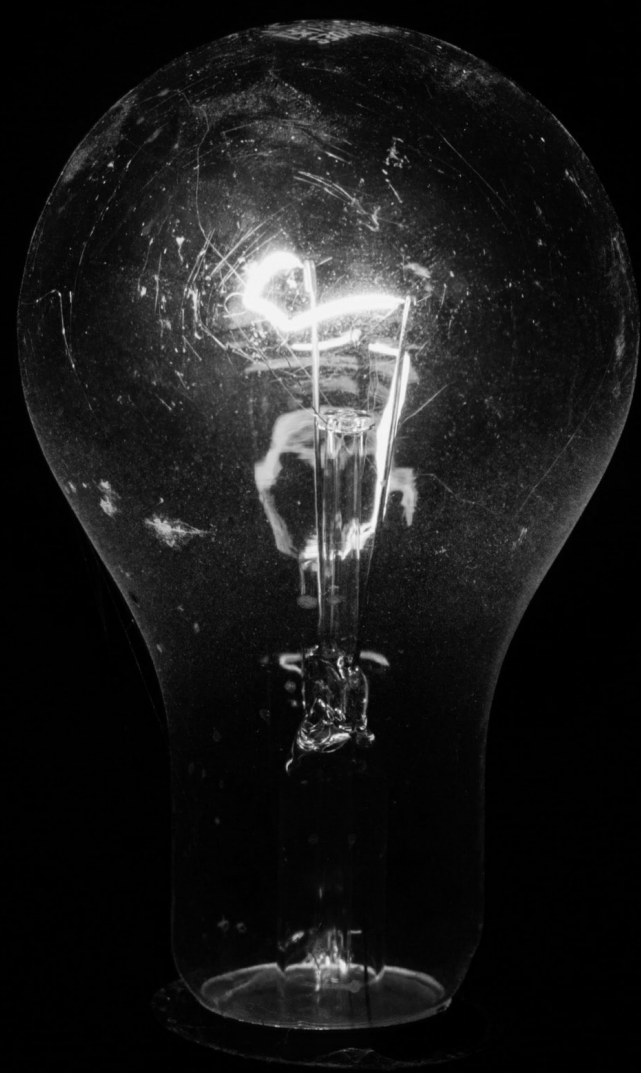




# THE **BIG** IDEA



# THE **BIG** IDEA





THE BIG IDEA  
ONE-LINER

# THE BIG IDEA ONE-LINER

 Work at a higher level 



THE BIG IDEA

LANGUAGE DESIGN REFLECTS INTENDED USE

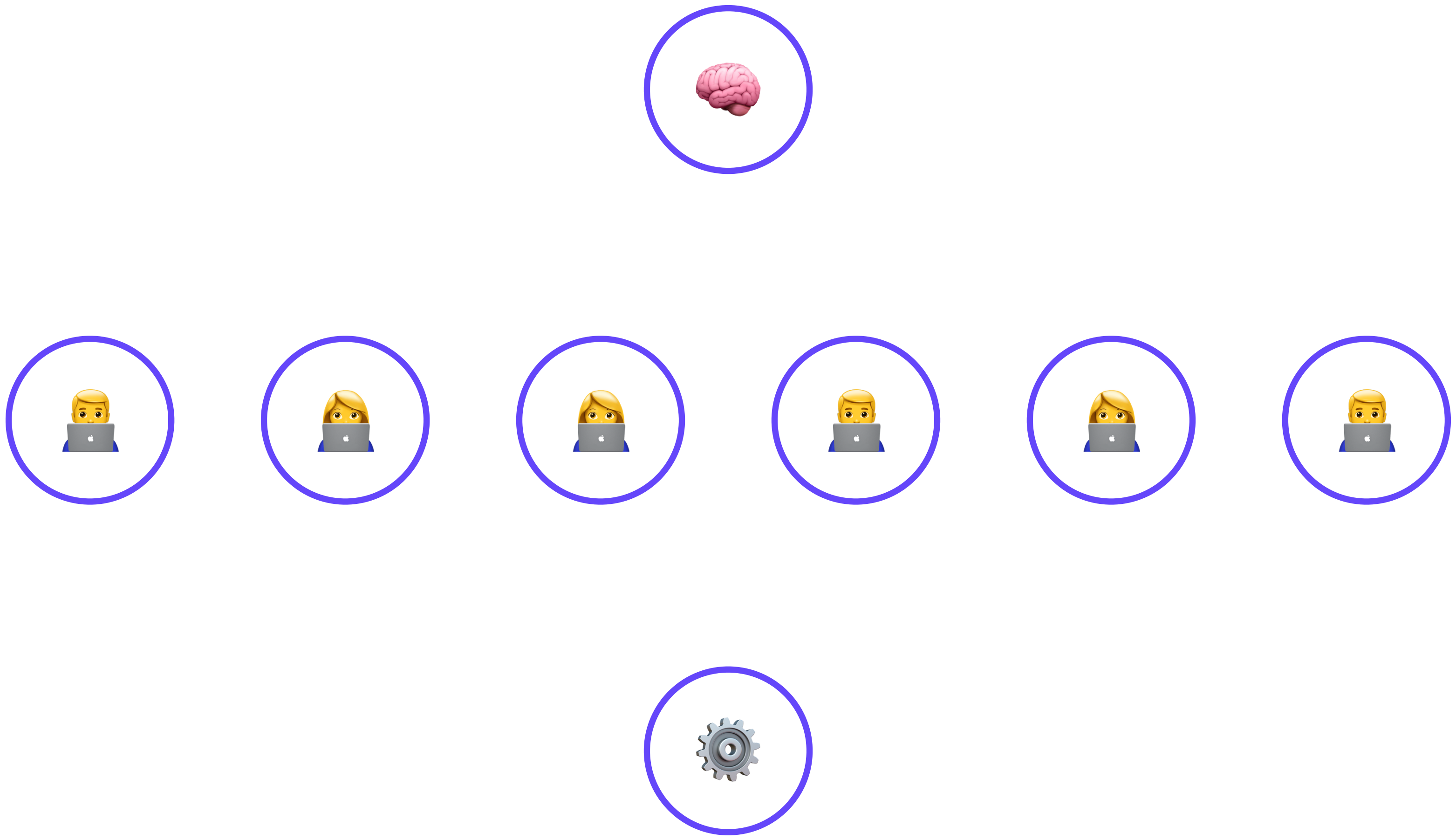


THE BIG IDEA

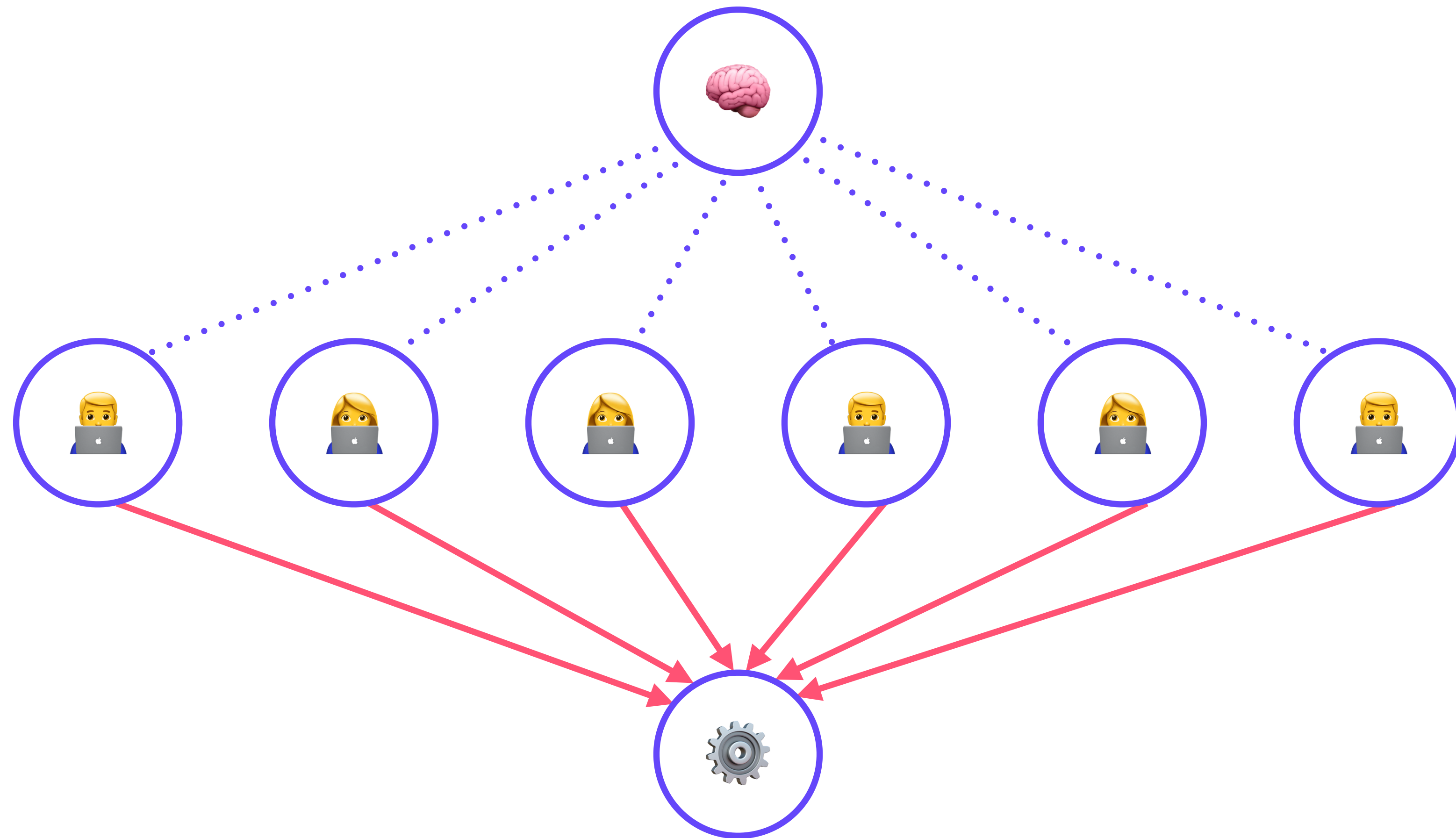
WHO'S ORG LOOKS LIKE THIS?



THE BIG IDEA  
WHO'S ORG LOOKS LIKE THIS?



THE BIG IDEA  
WHO'S ORG LOOKS LIKE THIS?

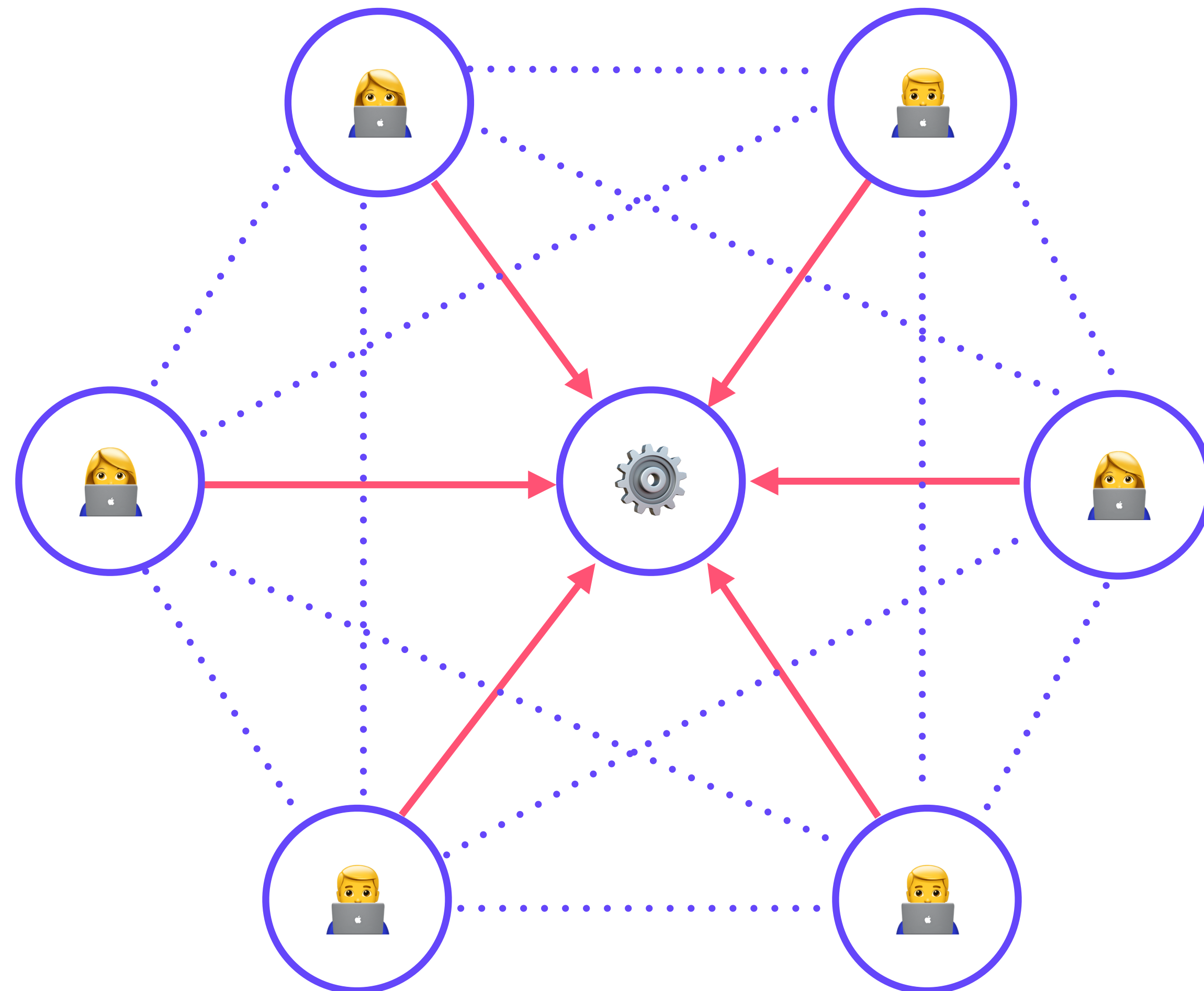


THE BIG IDEA  
HOW ABOUT THIS?





THE BIG IDEA  
HOW ABOUT THIS?



THE BIG IDEA  
QUESTIONS

# THE BIG IDEA QUESTIONS

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



# THE BIG IDEA QUESTIONS

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?**

# THE BIG IDEA QUESTIONS

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?

# THE BIG IDEA QUESTIONS

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?

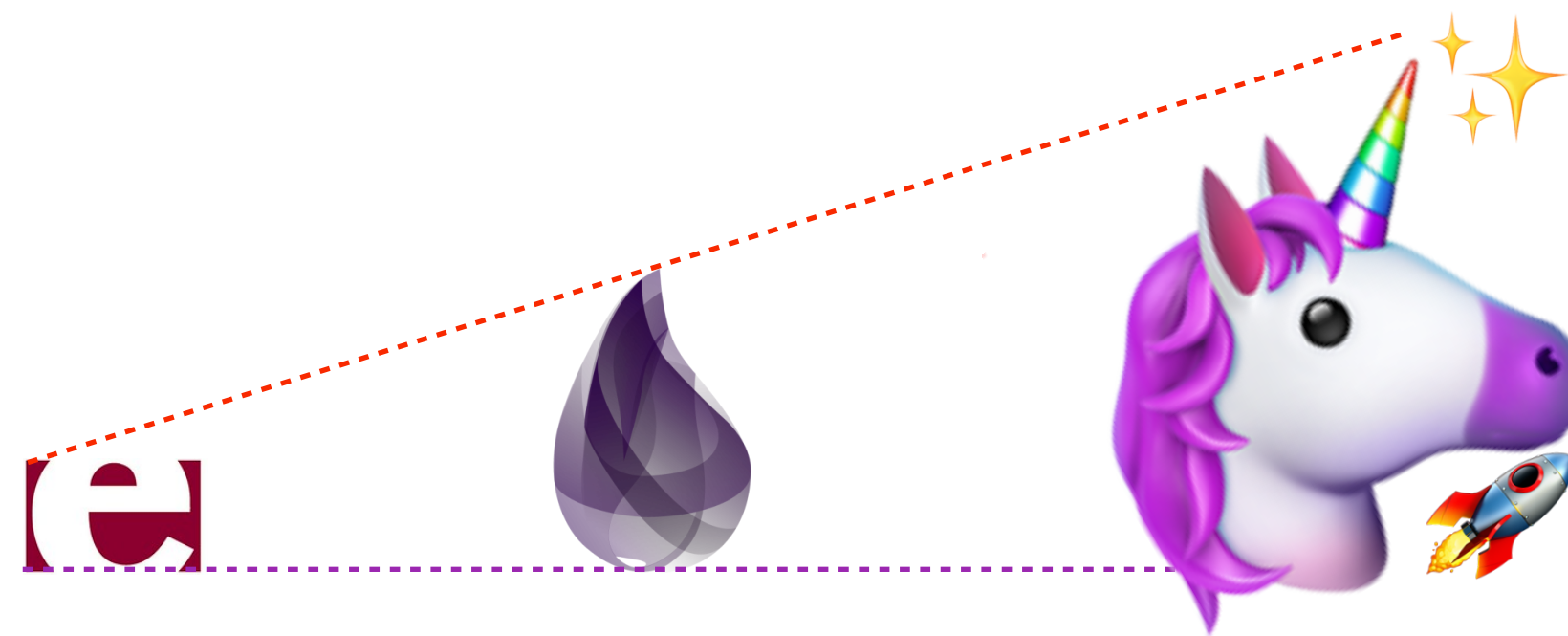
# THE BIG IDEA CORE

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



# THE BIG IDEA CORE

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





IN THE LARGE



IN THE LARGE





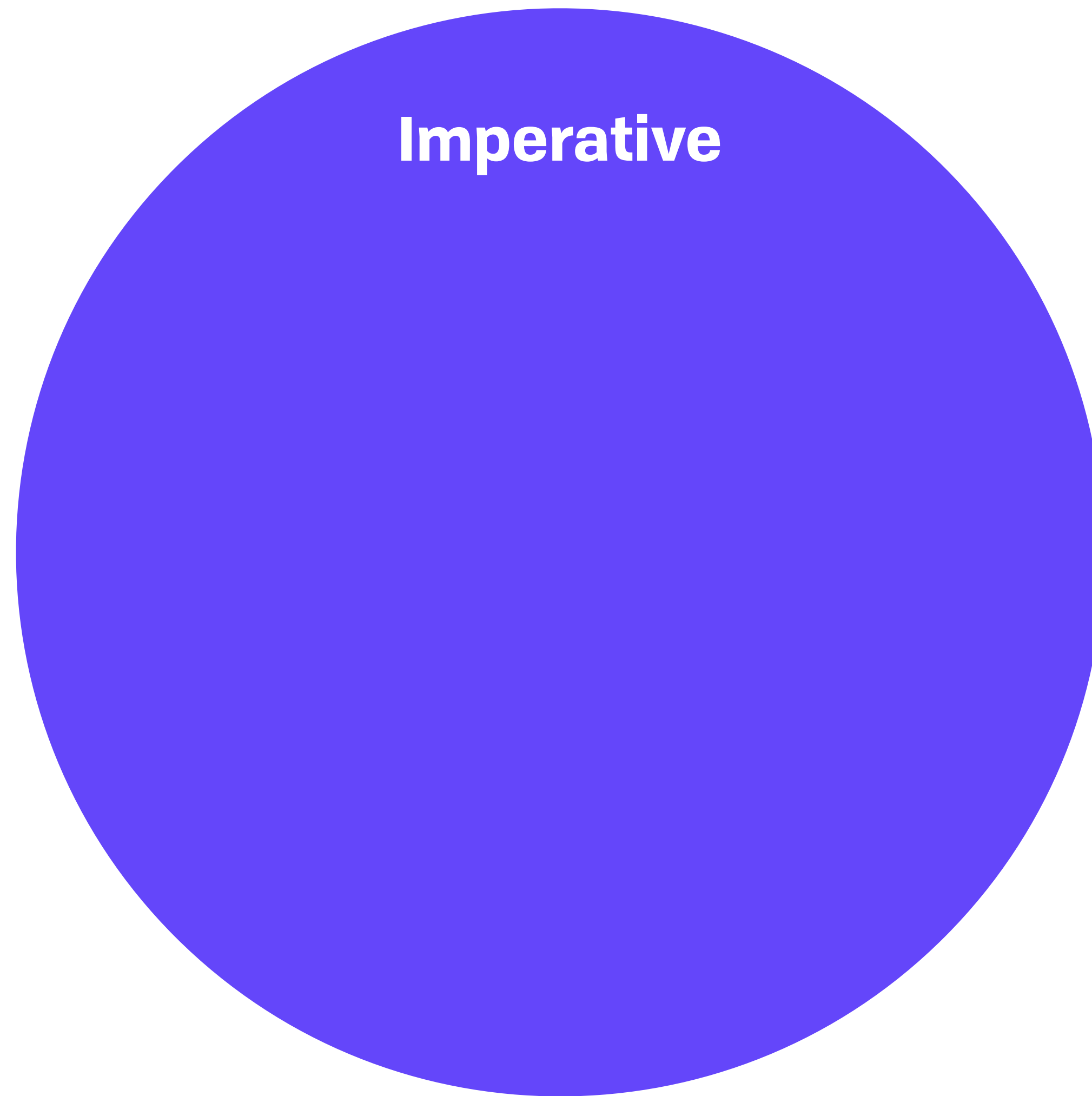
IN THE LARGE  
CODE YOU USED TO WRITE

IN THE LARGE  
CODE YOU USED TO WRITE



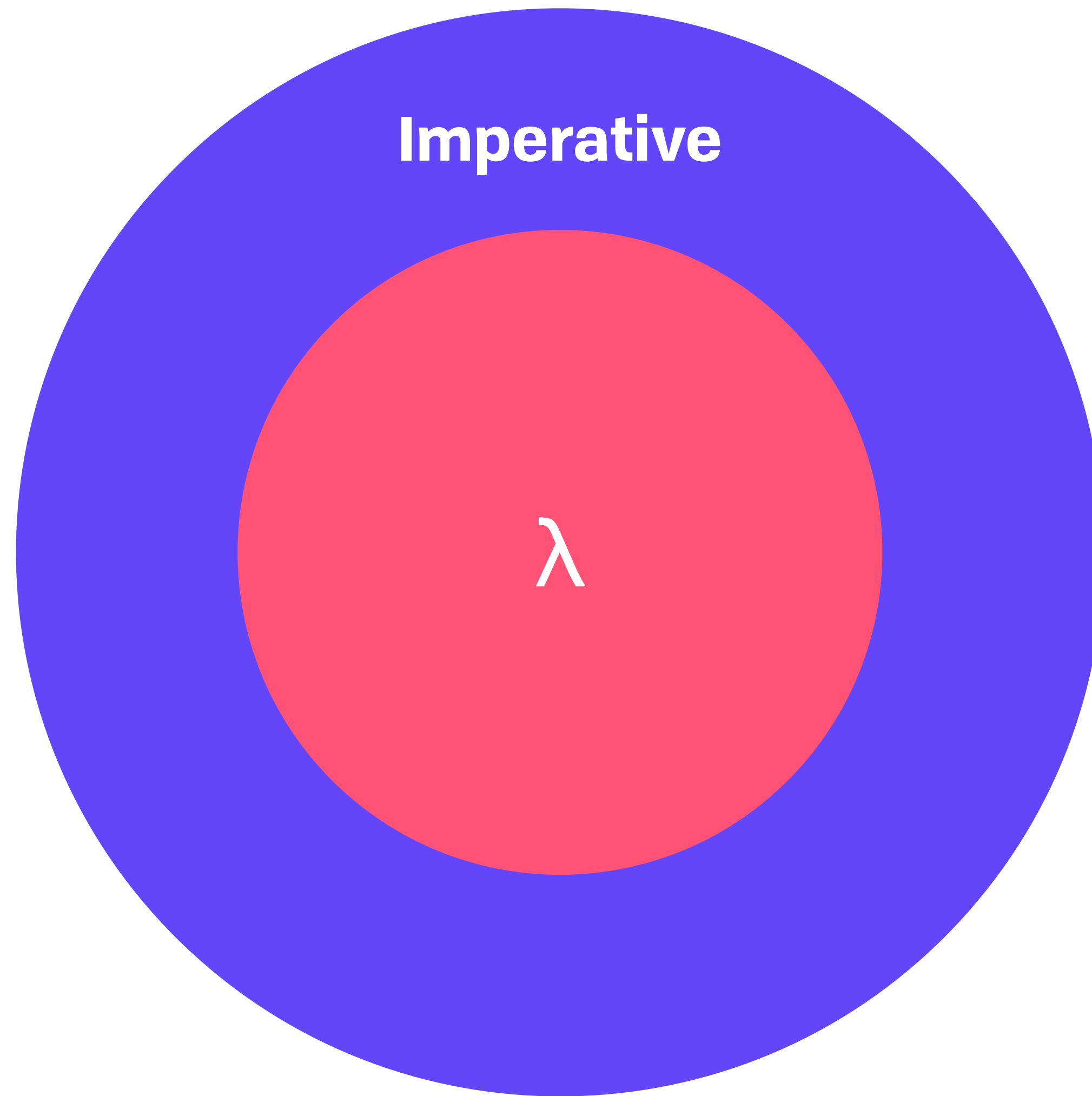
**Imperative**

IN THE LARGE  
“GOOD” ELIXIR



\* Functional core,  
imperative shell

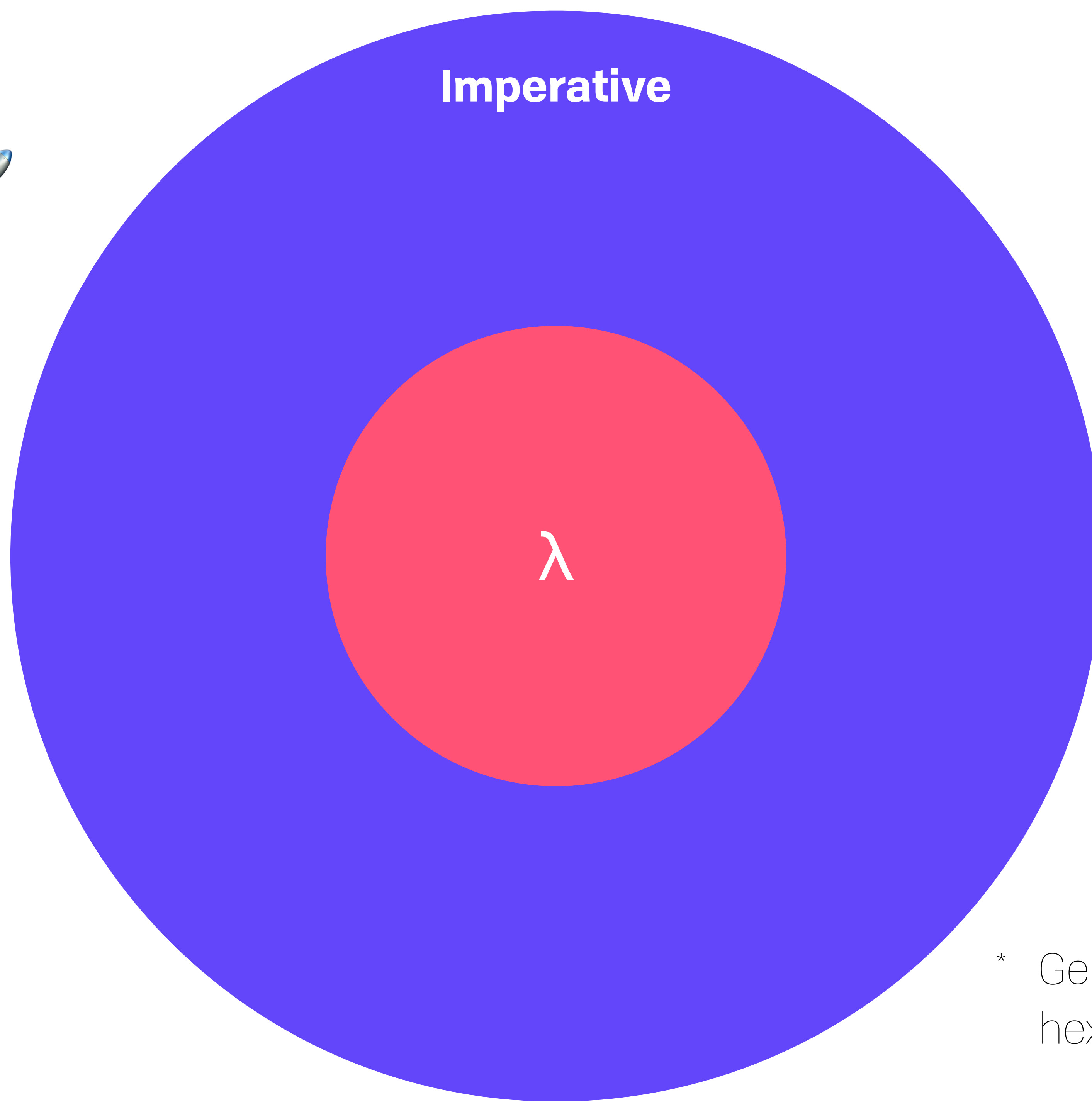
IN THE LARGE  
“GOOD” ELIXIR



\* Functional core,  
imperative shell

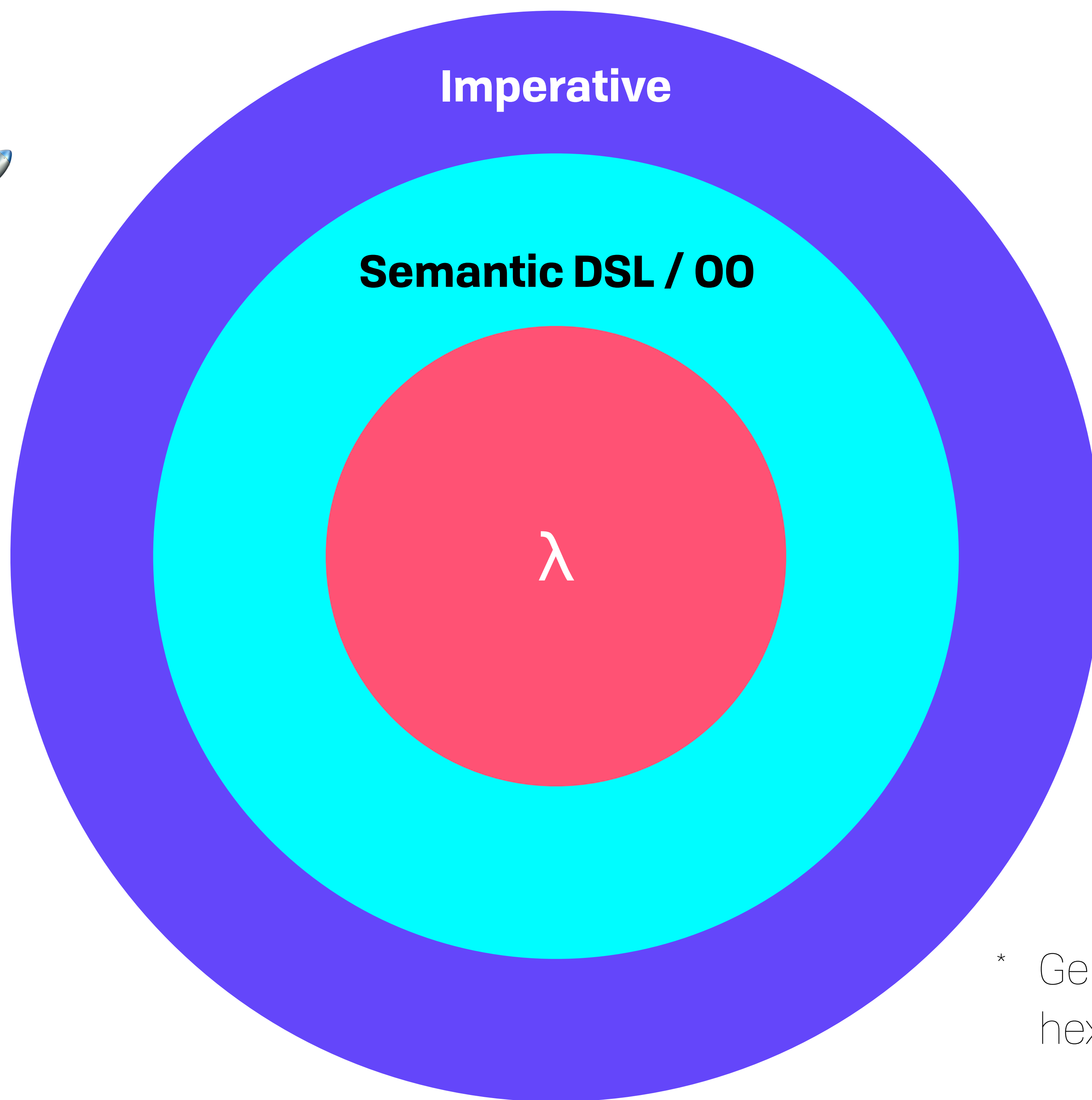


IN THE LARGE  
3-LAYER 🚀



\* Generalization of  
hexagonal/12-factor

IN THE LARGE  
3-LAYER 🚀



\* Generalization of  
hexagonal/12-factor

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
```




A black and white photograph of a concrete wall. The wall features diagonal stripes, alternating between dark and light sections. Several horizontal metal rebar rods are visible, embedded in the concrete. A white rectangular box is superimposed over the center of the image, containing the text "GOTO CONSIDERED HARMFUL" in a thin, black, sans-serif font.

GOTO CONSIDERED  
HARMFUL



GOTOS CONSIDERED HARMFUL

WHAT'S SO BAD ABOUT HAVING CONTROL?  



GOTOS 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

GOTOS 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

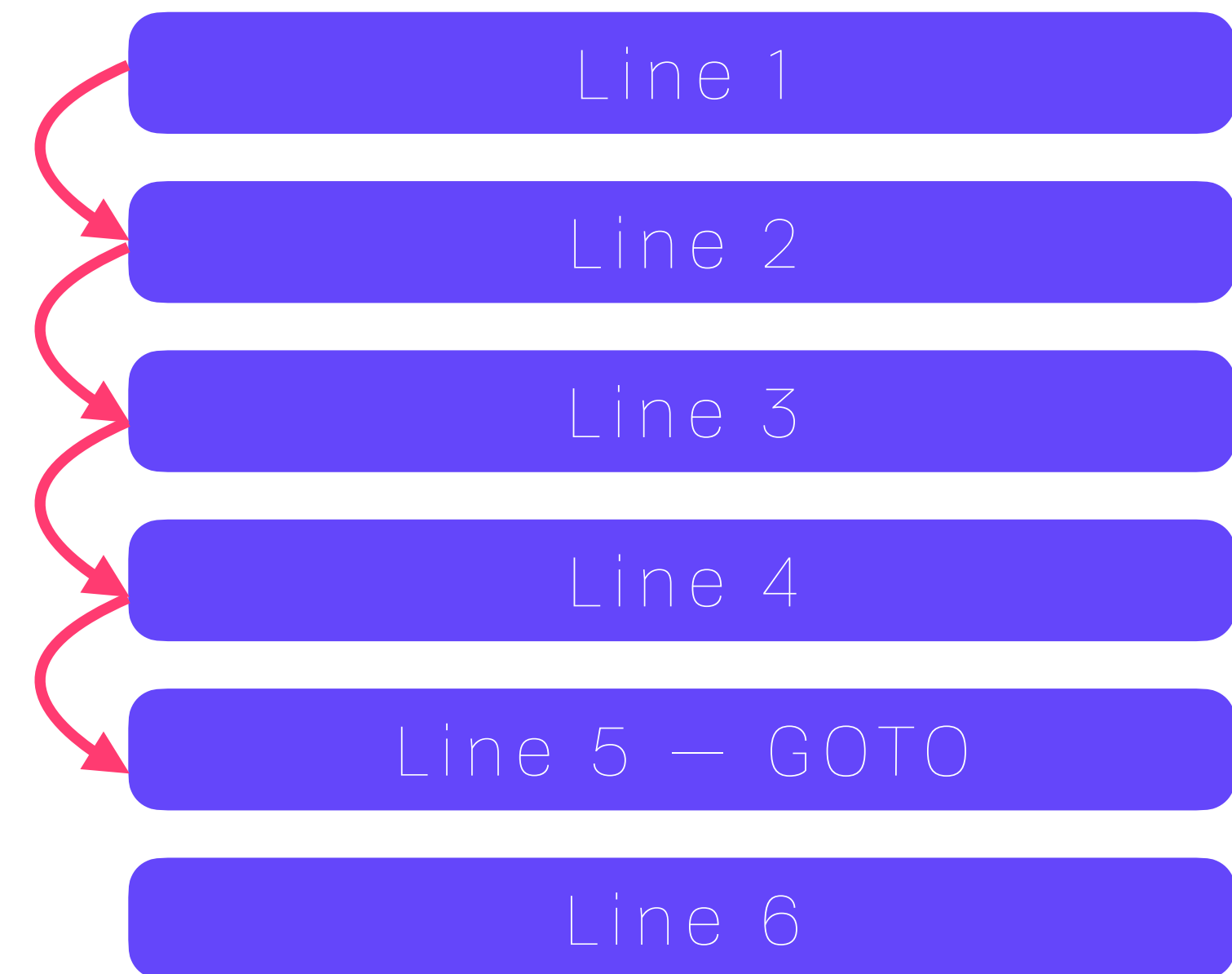
Line 5 — GOTO

Line 6

GOTOS CONSIDERED HARMFUL

WHAT'S SO BAD ABOUT HAVING CONTROL? 🦶🔫

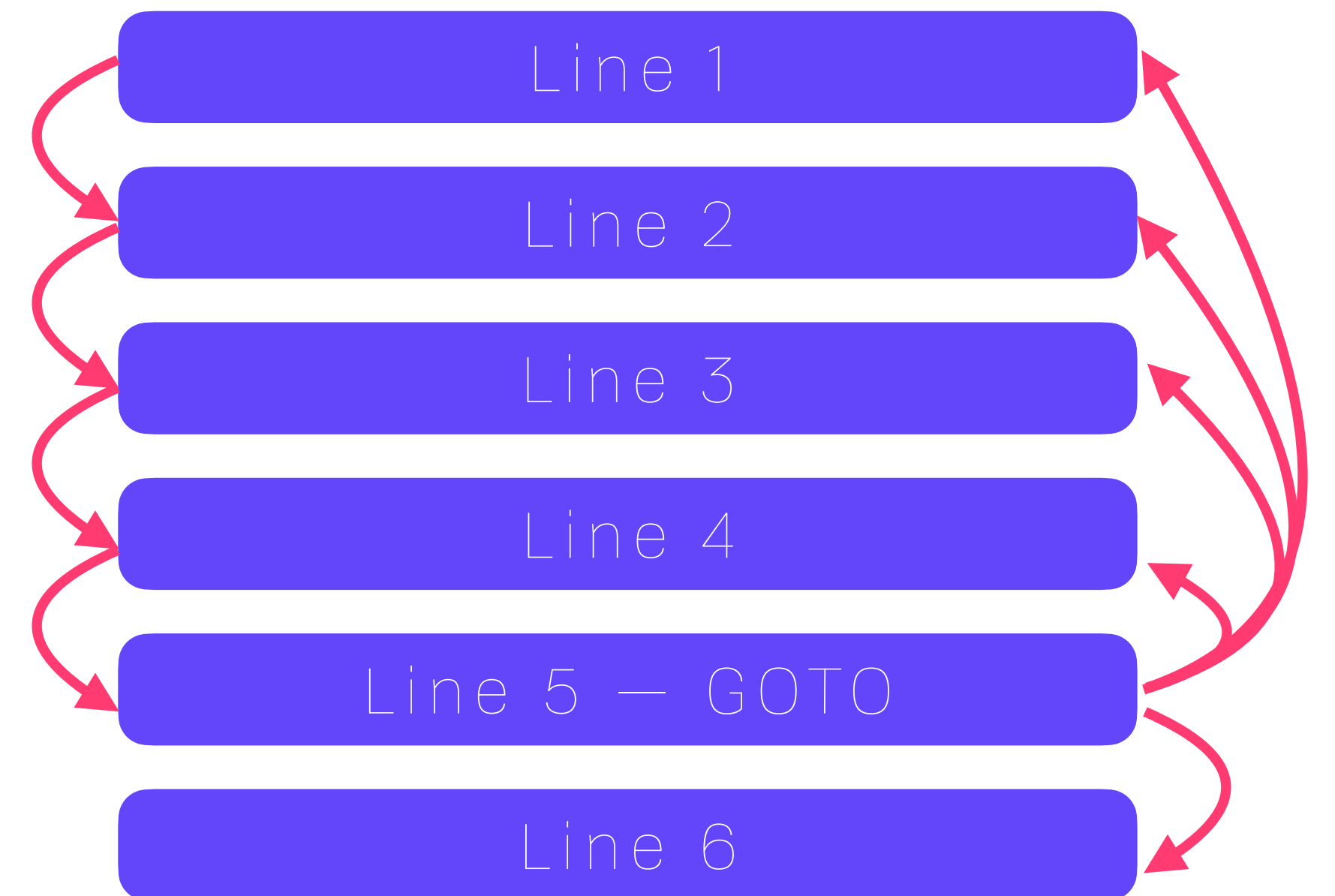
- GOTOs
- Low level instruction
- Literally how the machine is going to see it
- *Extremely* flexible
- Highly concrete
- *Huge* number of **implicit** states



GOTOS CONSIDERED HARMFUL

WHAT'S SO BAD ABOUT HAVING CONTROL? 🦶🔫

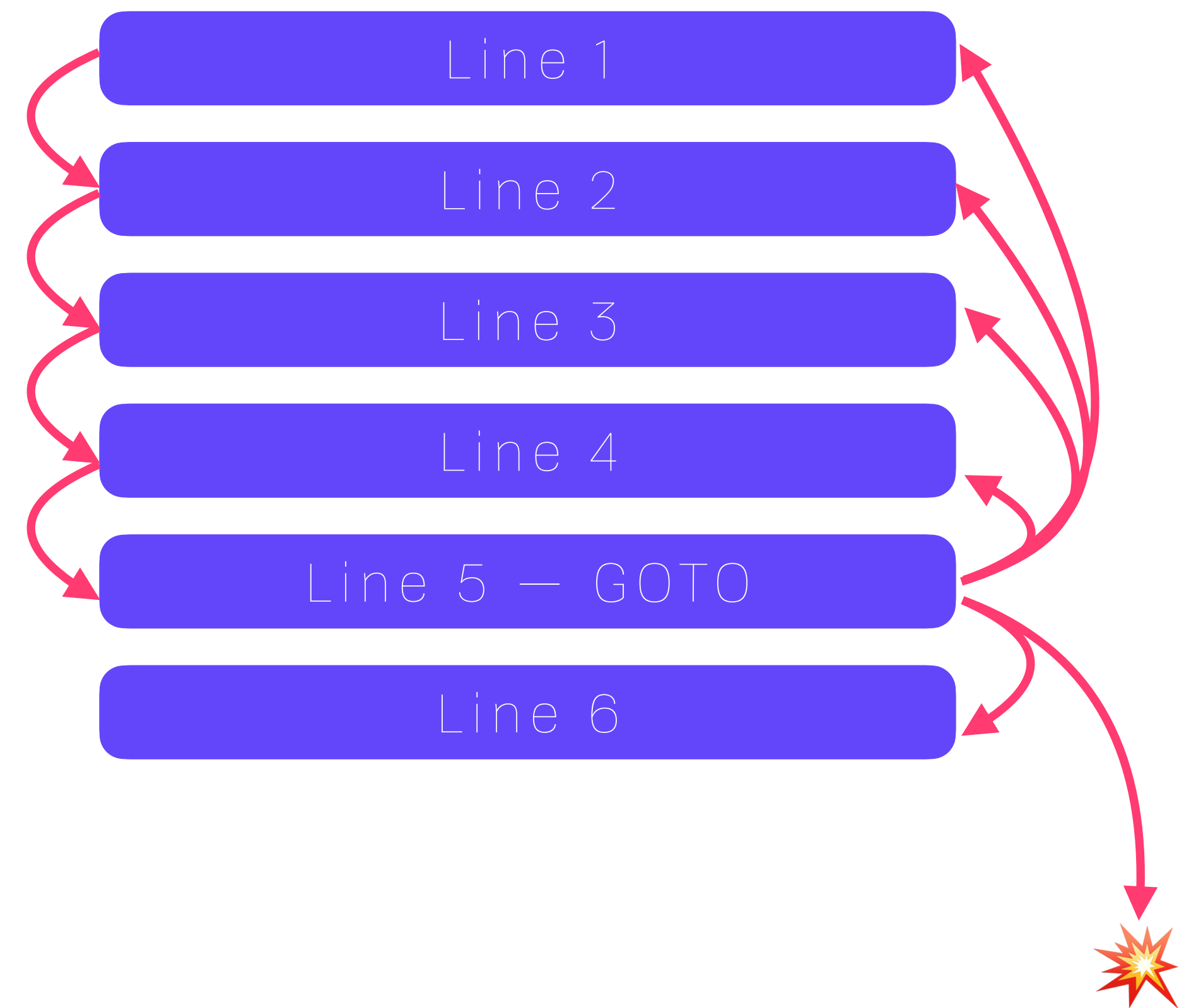
- GOTOs
- Low level instruction
- Literally how the machine is going to see it
- *Extremely* flexible
- Highly concrete
- *Huge* number of **implicit** states



GOTOS CONSIDERED HARMFUL

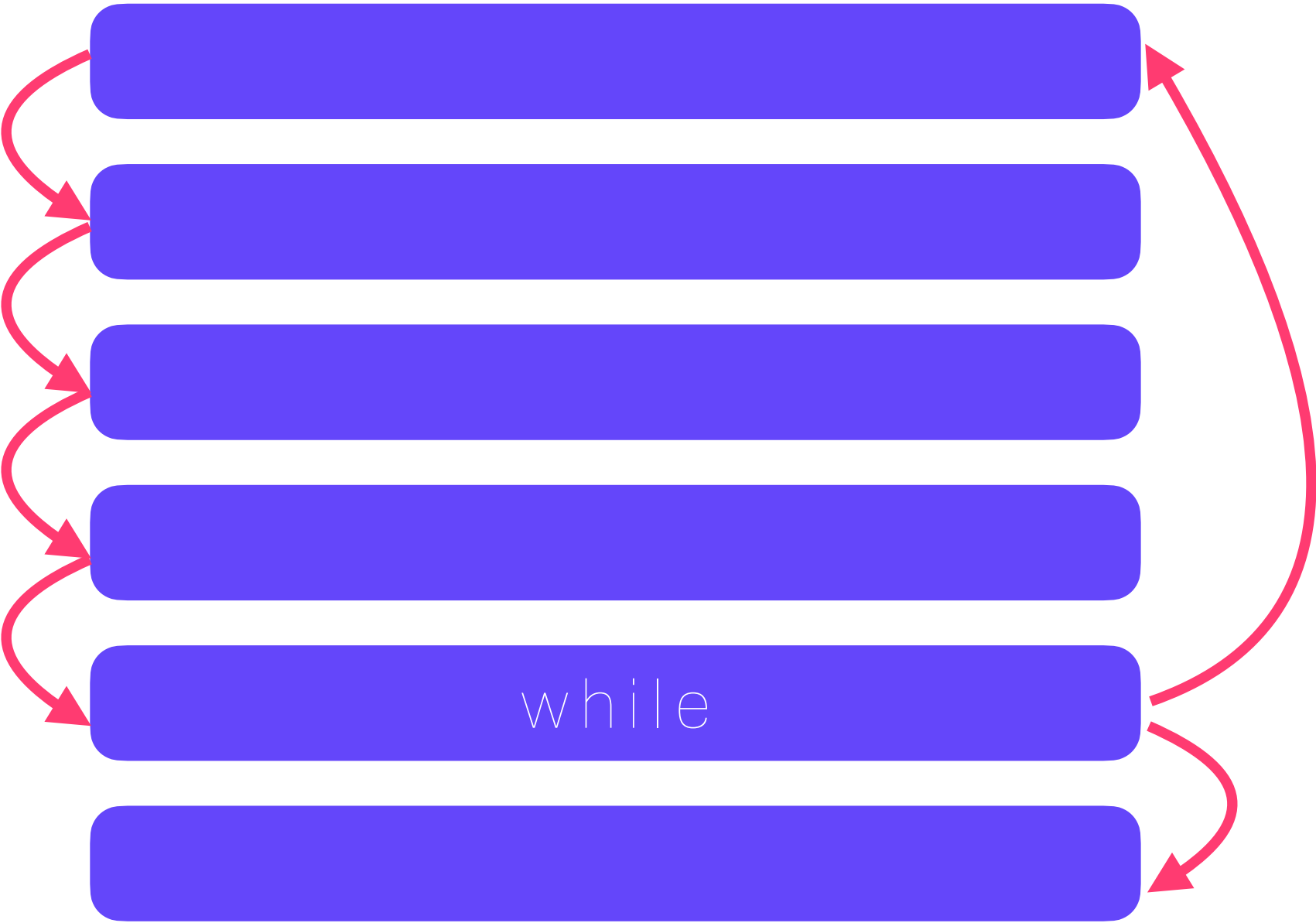
WHAT'S SO BAD ABOUT HAVING CONTROL? 🦶🔫

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





GOTOS CONSIDERED HARMFUL  
STRUCTURED PROGRAMMING



# GOTOS CONSIDERED HARMFUL STRUCTURED PROGRAMMING

- Subroutines
- Loops
- Switch/branching
- Named routines



GOTOS CONSIDERED HARMFUL  
THE NEXT GENERATION 🚀

# GOTOS CONSIDERED HARMFUL THE NEXT GENERATION 🚀

- Functions
- Map
- Reduce
- Filter
- Constraint solvers

GOTOS CONSIDERED HARMFUL  
TRADEOFFS

# GOTOS CONSIDERED HARMFUL TRADEOFFS

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



# GOTOS CONSIDERED HARMFUL TRADEOFFS

- Exchange granular control for structure
- **Meaning over mechanics**
- More human than machine
- *Safer!*
- Spectrum
- Turing Tarpit
- Church Chasm
- Haskell Fan Fiction

COMPLEXITY

ACTOR ABYSS 🕳️

# COMPLEXITY

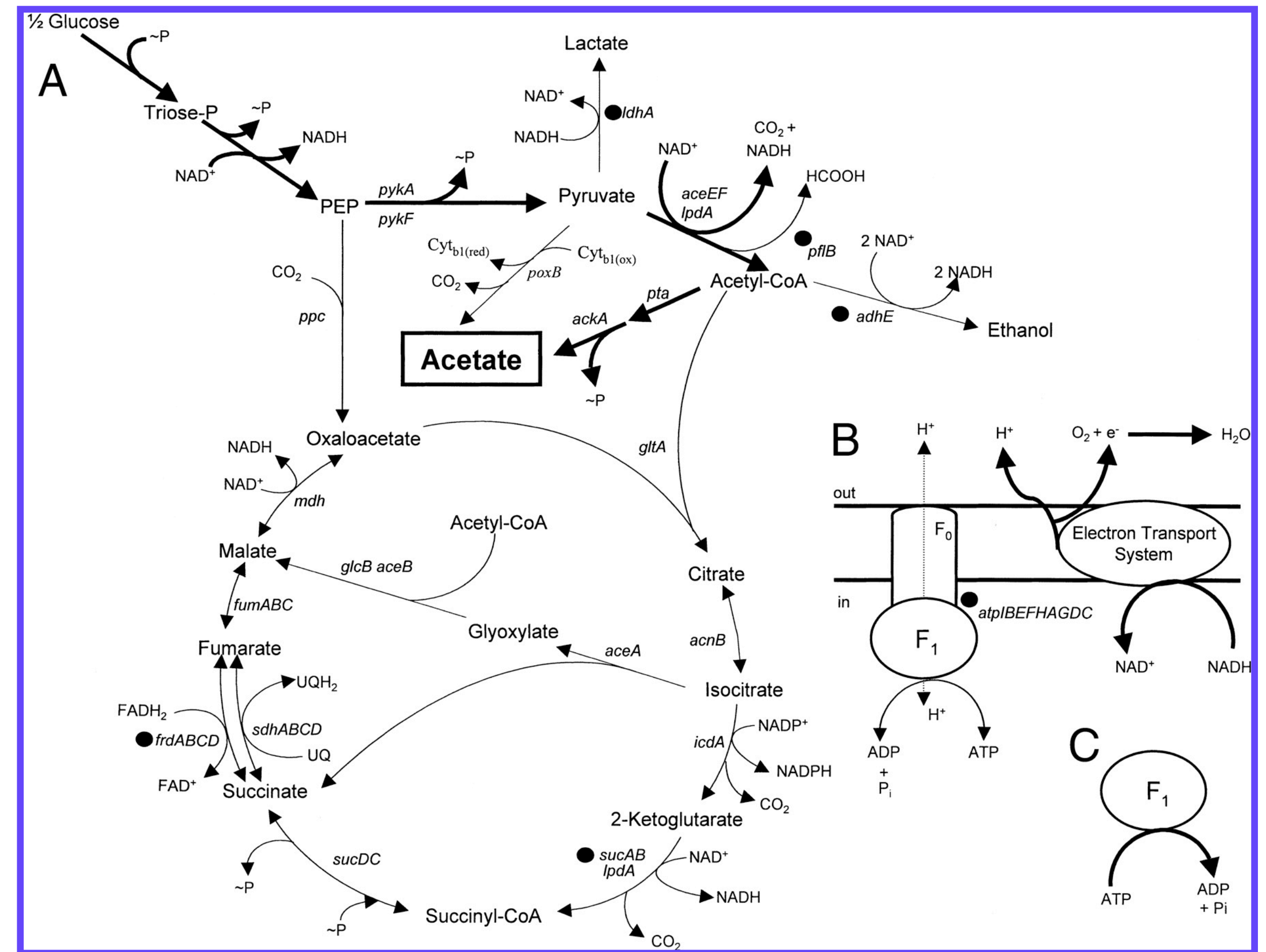
## ACTOR ABYSS 🕳️

1. Each step is very simple
2. Reasoning about dynamic organisms is hard
  1. Remember to store your data for crash recovery
  2. Called collaborator may not be there
3. Complexity grows faster than linear
4. Find common factors — your abstraction

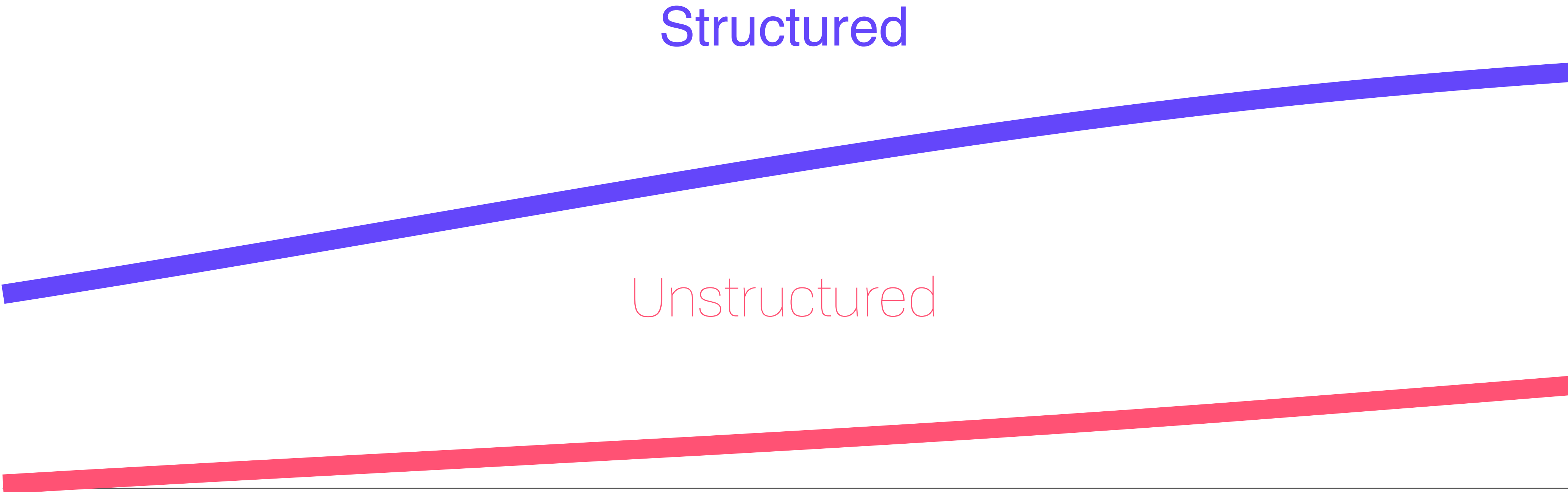


# COMPLEXITY ACTOR ABYSS

1. Each step is very simple
2. Reasoning about dynamic organisms is hard
  1. Remember to store your data for crash recovery
  2. Called collaborator may not be there
3. Complexity grows faster than linear
4. Find common factors — your abstraction



GOTOS CONSIDERED HARMFUL  
PAYOFF



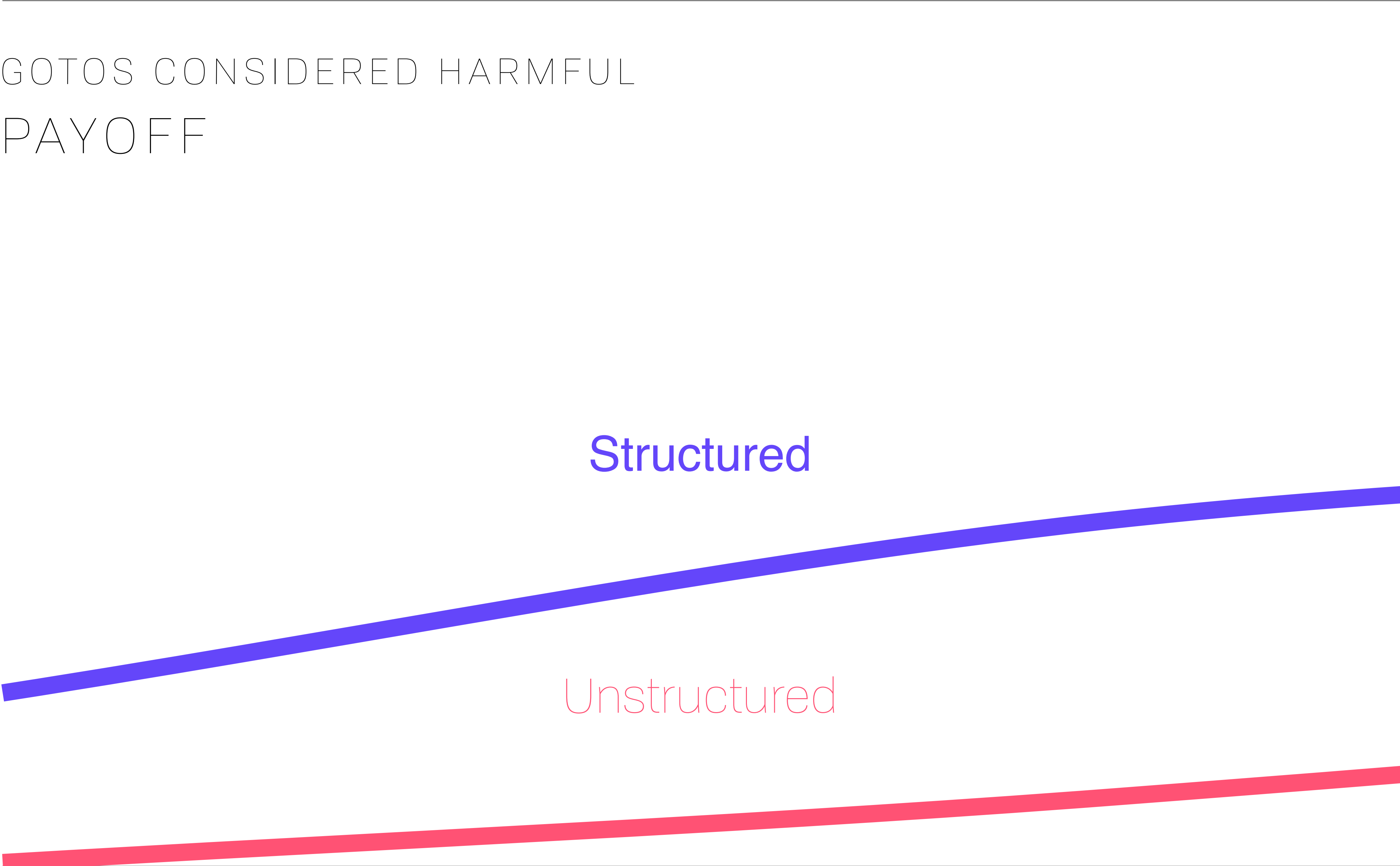
GOTOS CONSIDERED HARMFUL  
PAYOFF

COMPLEXITY

Structured

Unstructured

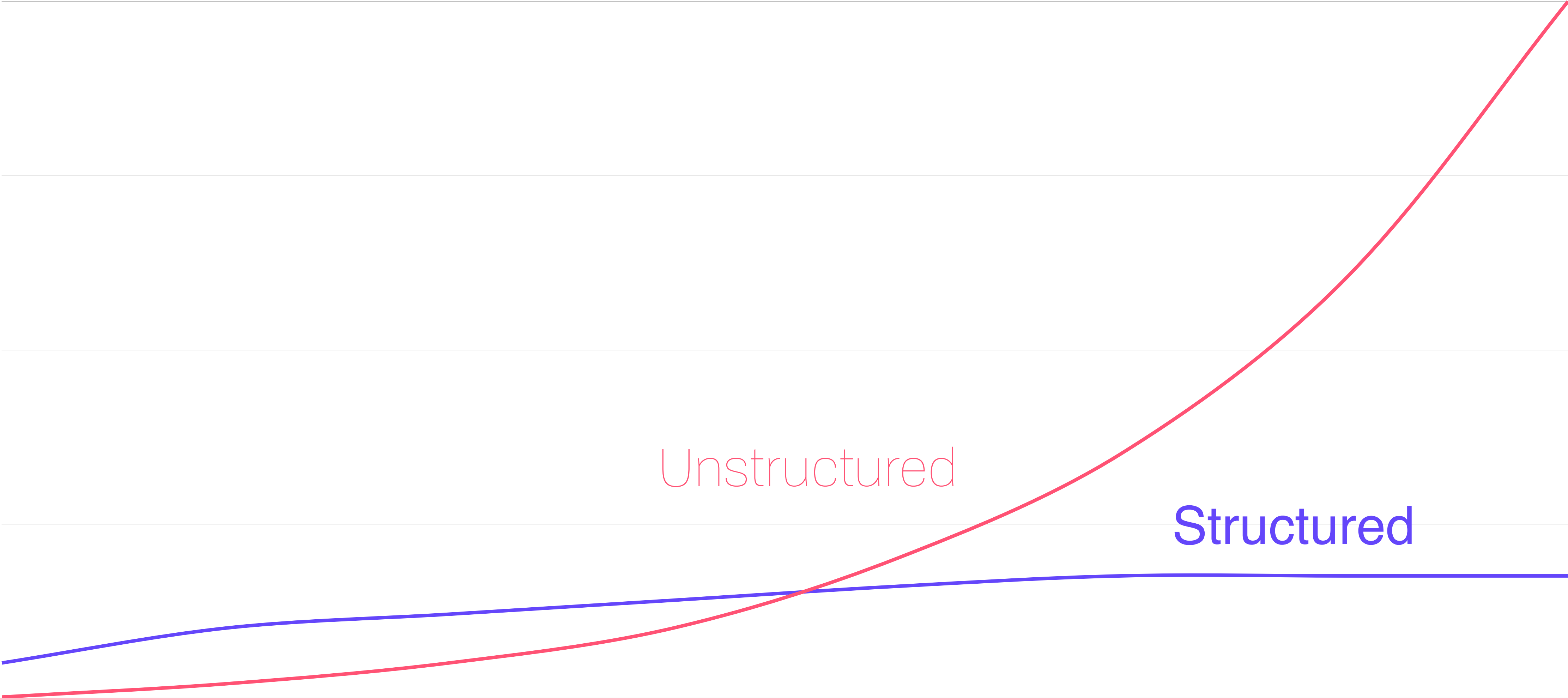
TIME





GOTOS CONSIDERED HARMFUL  
PAYOFF

COMPLEXITY

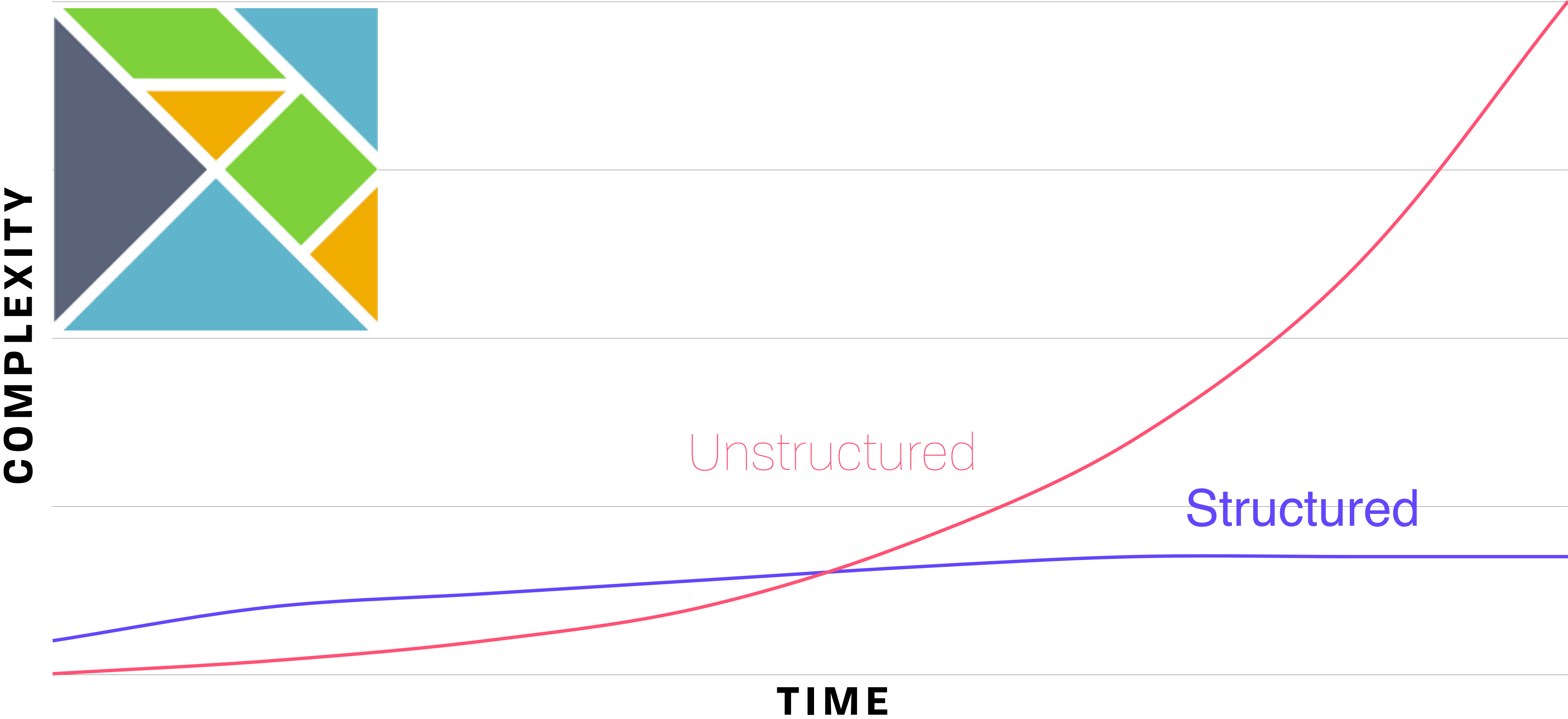


Unstructured

Structured

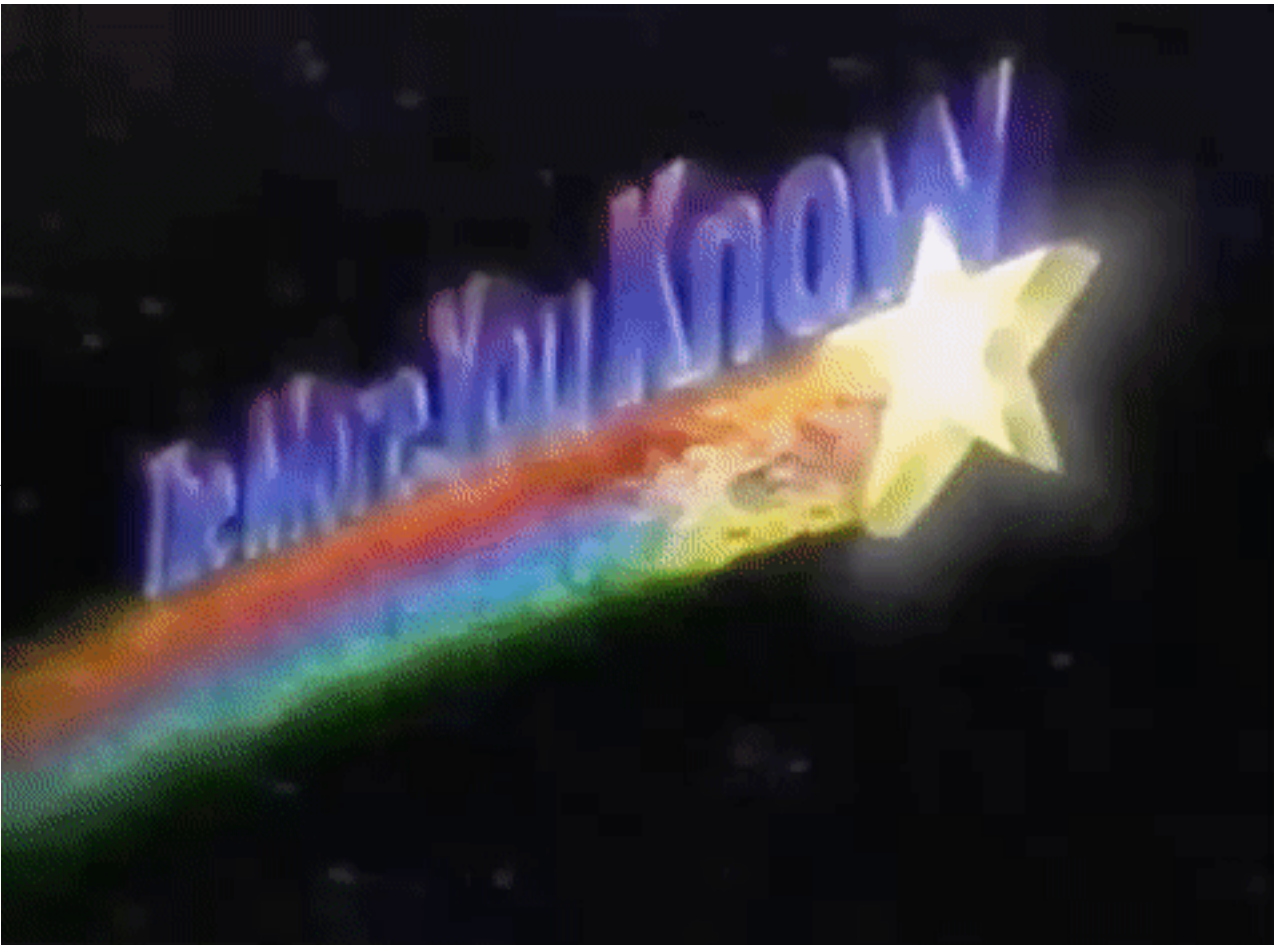
TIME

GOTOS CONSIDERED HARMFUL  
PAYOFF



GOTOS CONSIDERED HARMFUL  
PAYOFF

COMPLEXITY



Unstructured

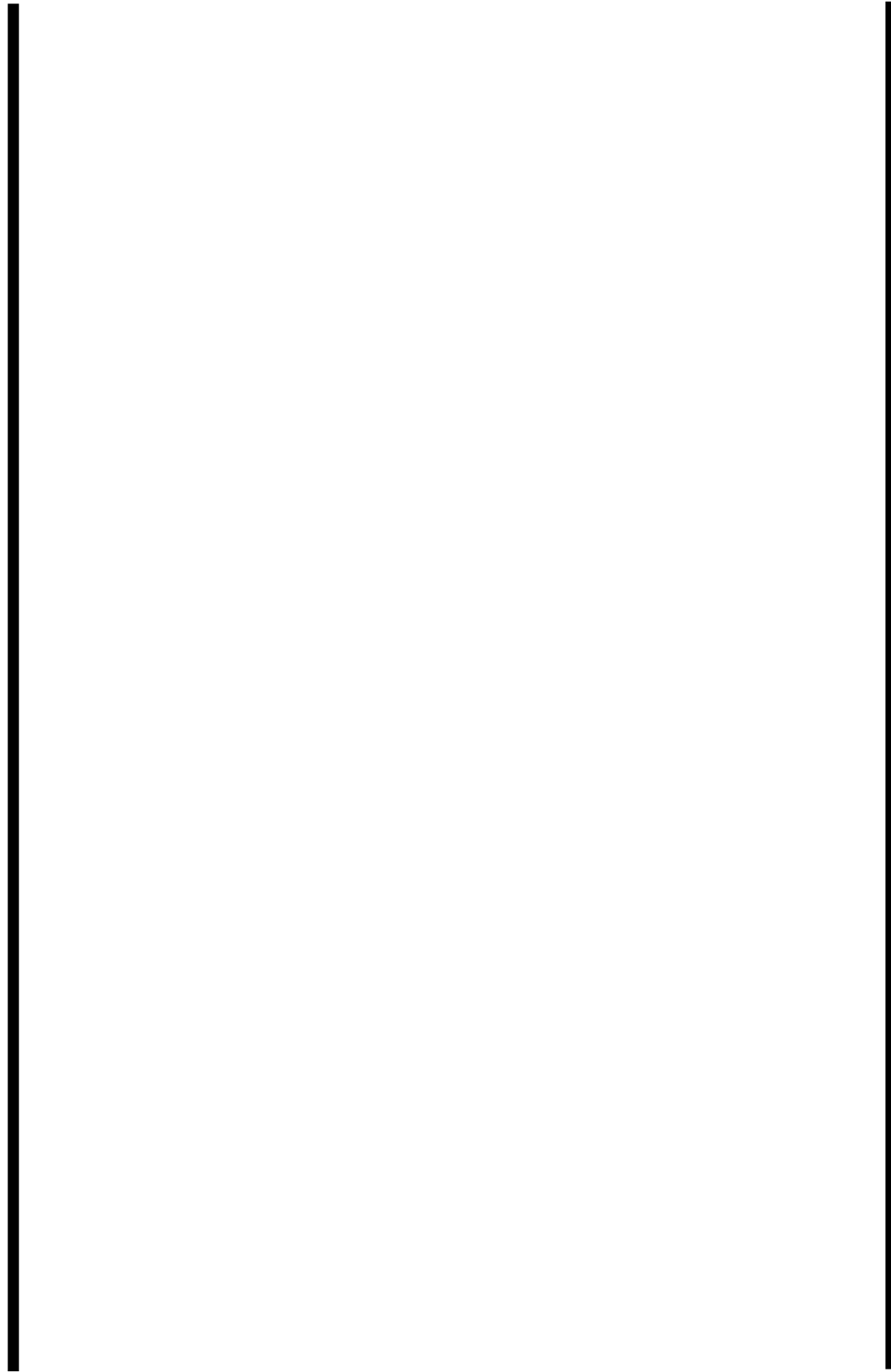
Structured

TIME

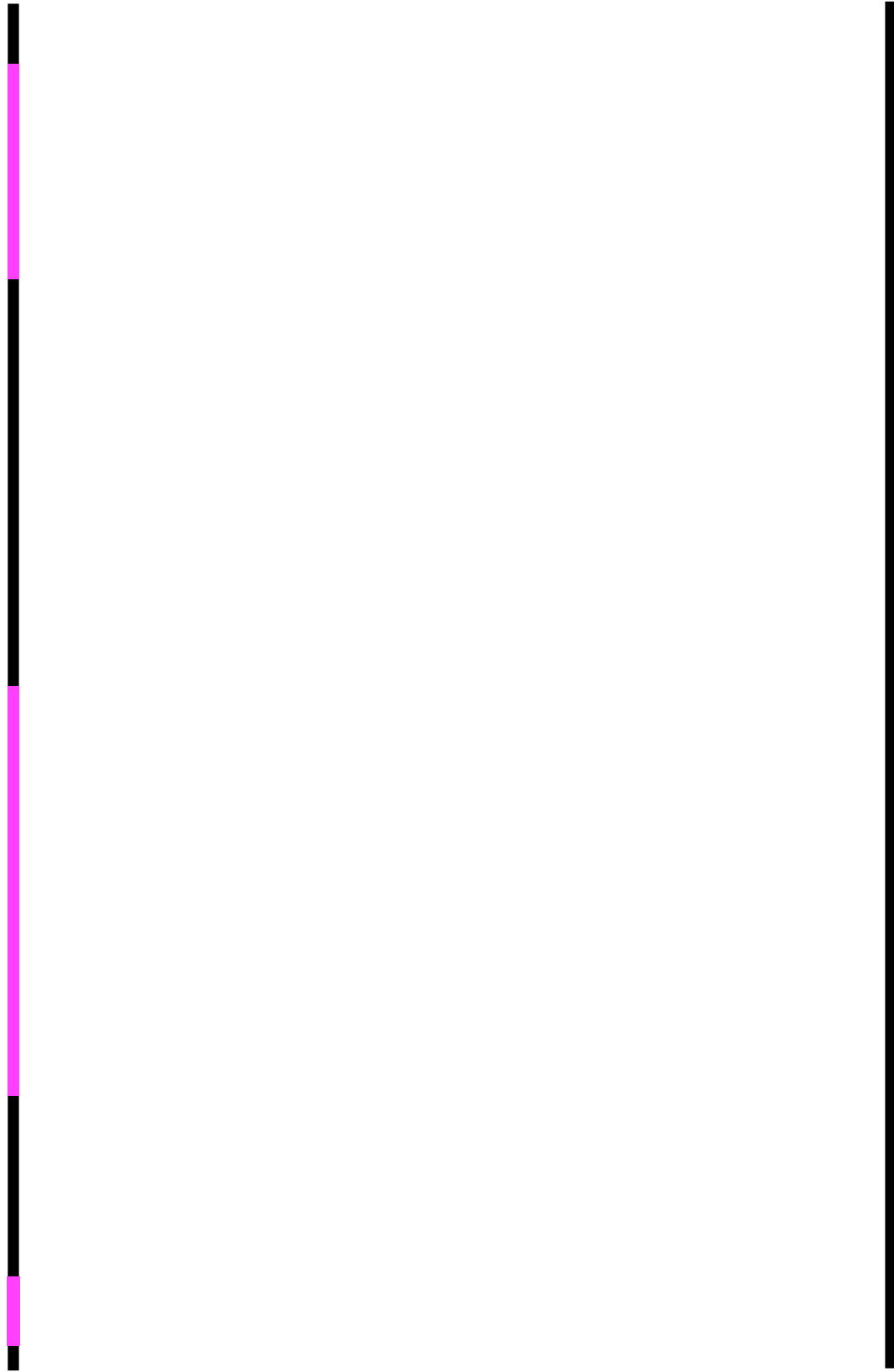


COMPLEXITY  
ORTHOGONAL COMPLECTING

COMPLEXITY  
ORTHOGONAL COMPLECTING

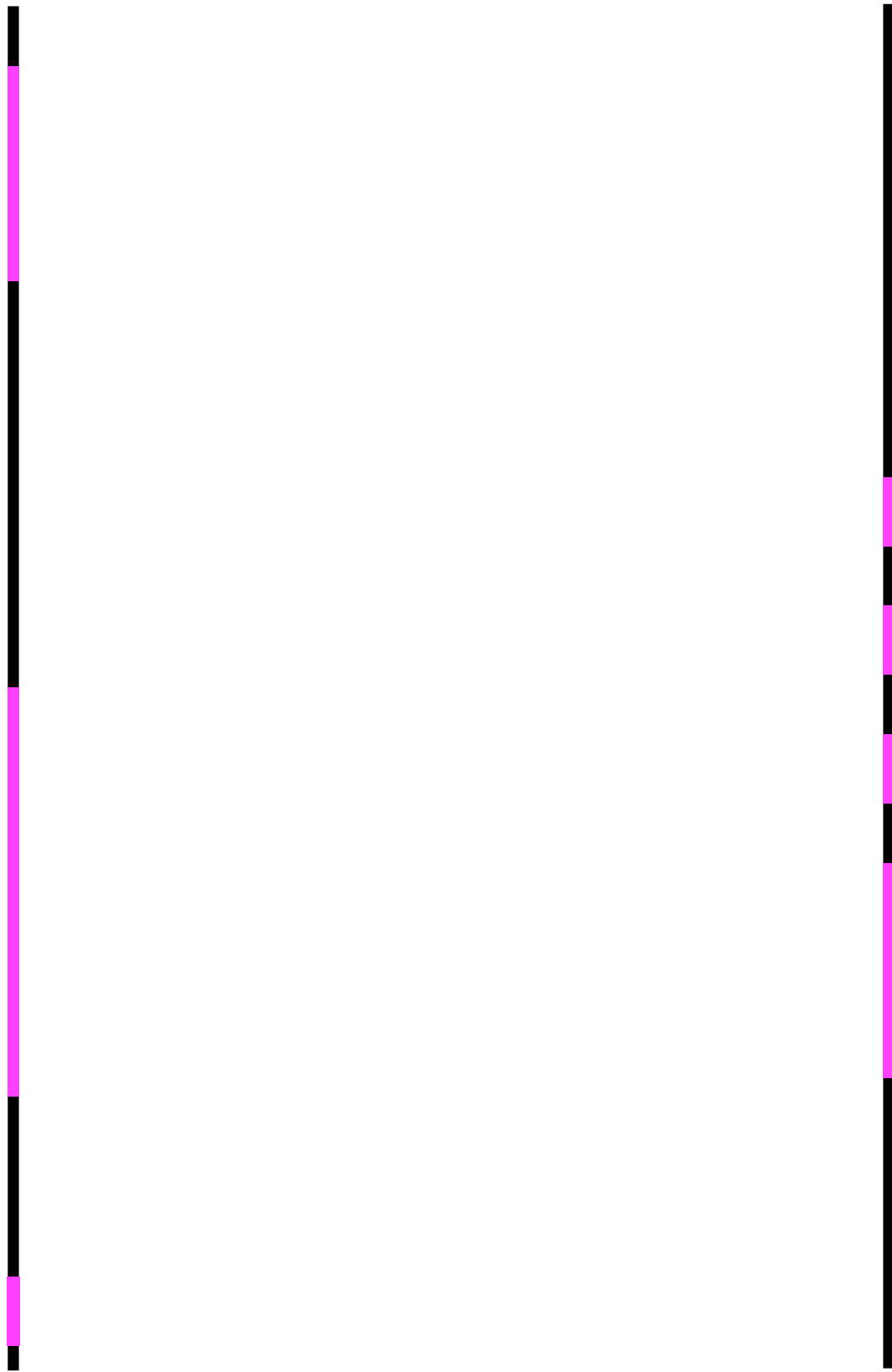


COMPLEXITY  
ORTHOGONAL COMPLETING

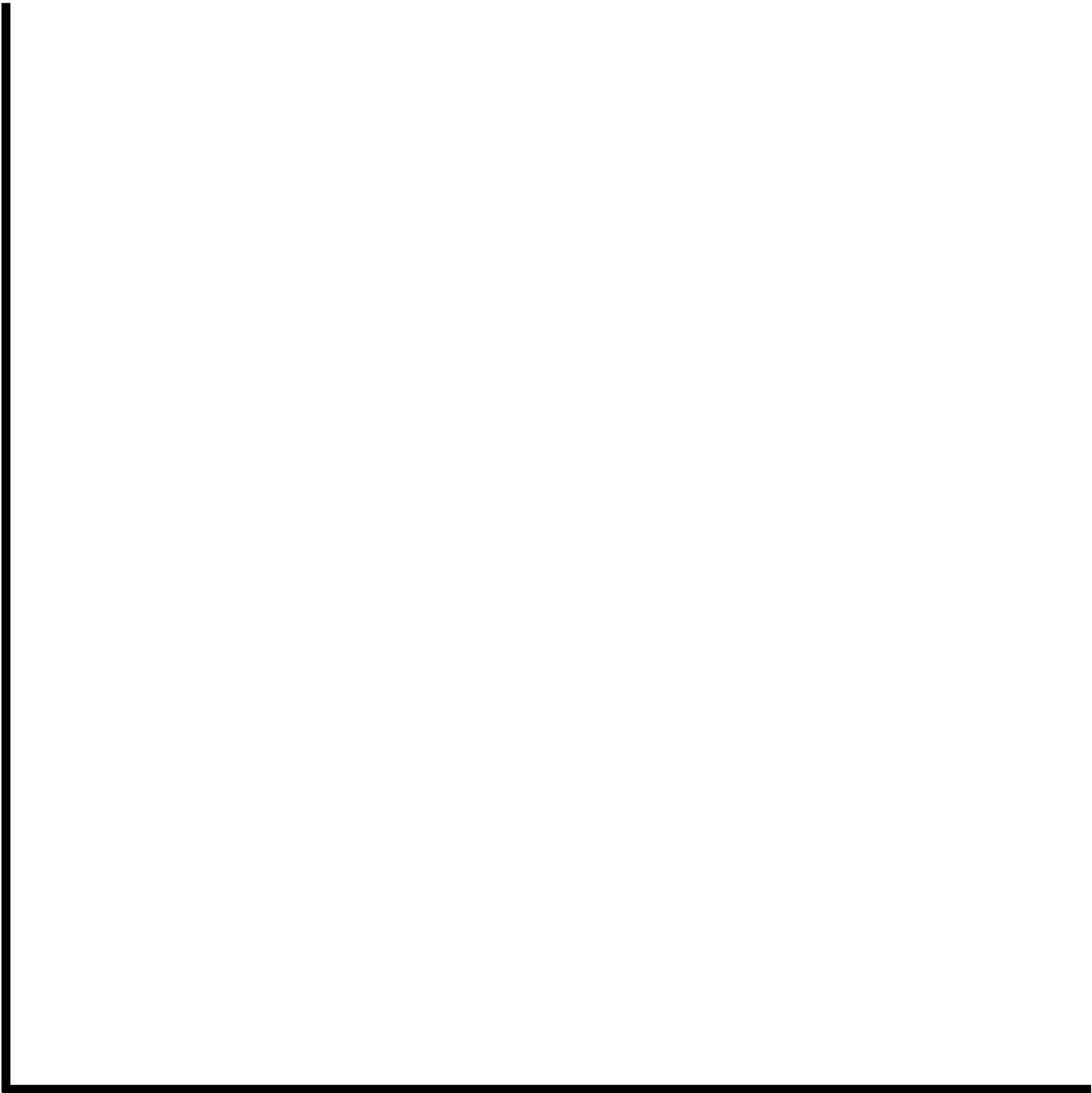




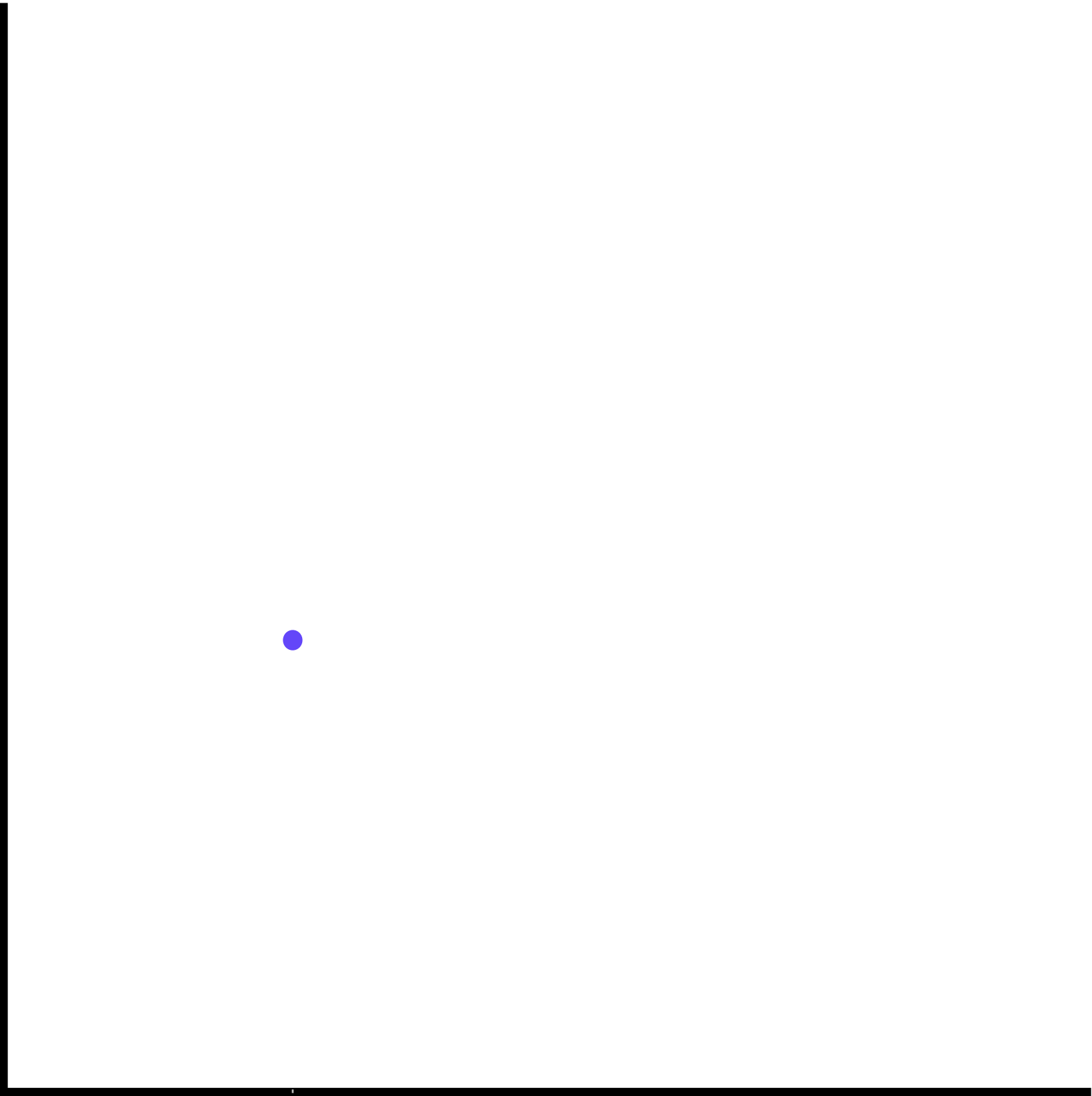
COMPLEXITY  
ORTHOGONAL COMPLETING



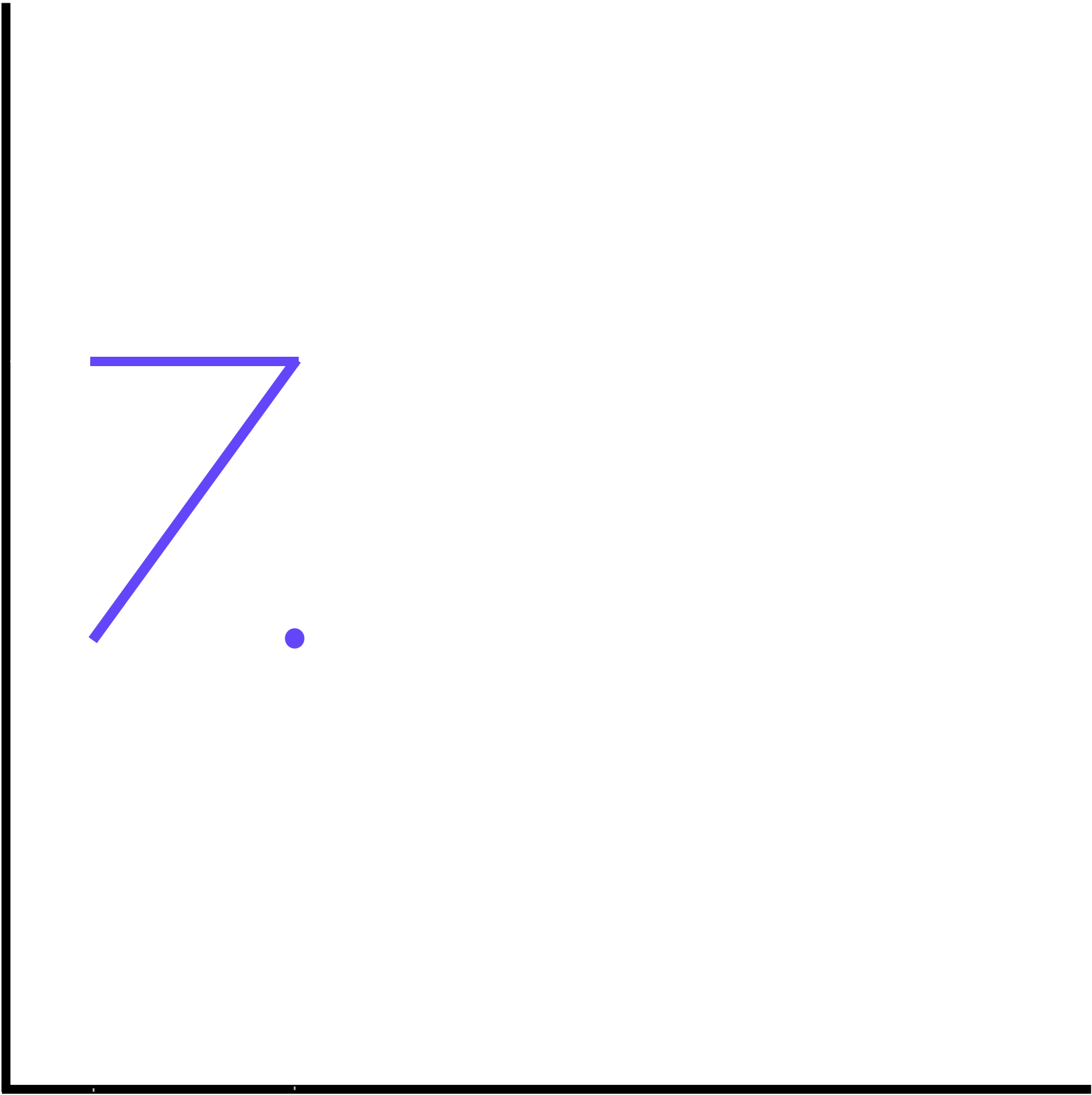
COMPLEXITY  
ORTHOGONAL COMPLECTING



COMPLEXITY  
ORTHOGONAL COMPLETING

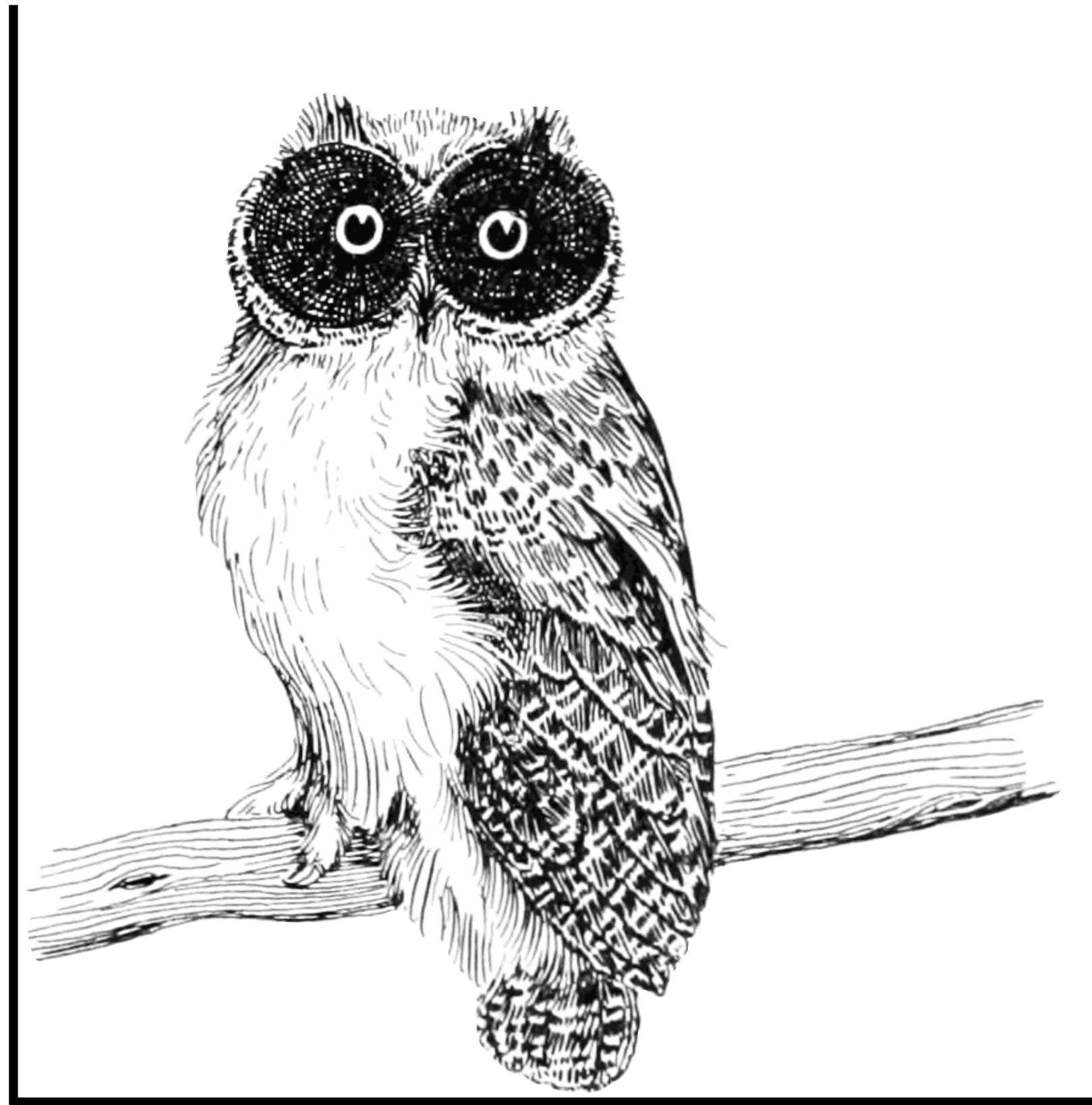


COMPLEXITY  
ORTHOGONAL COMPLETING





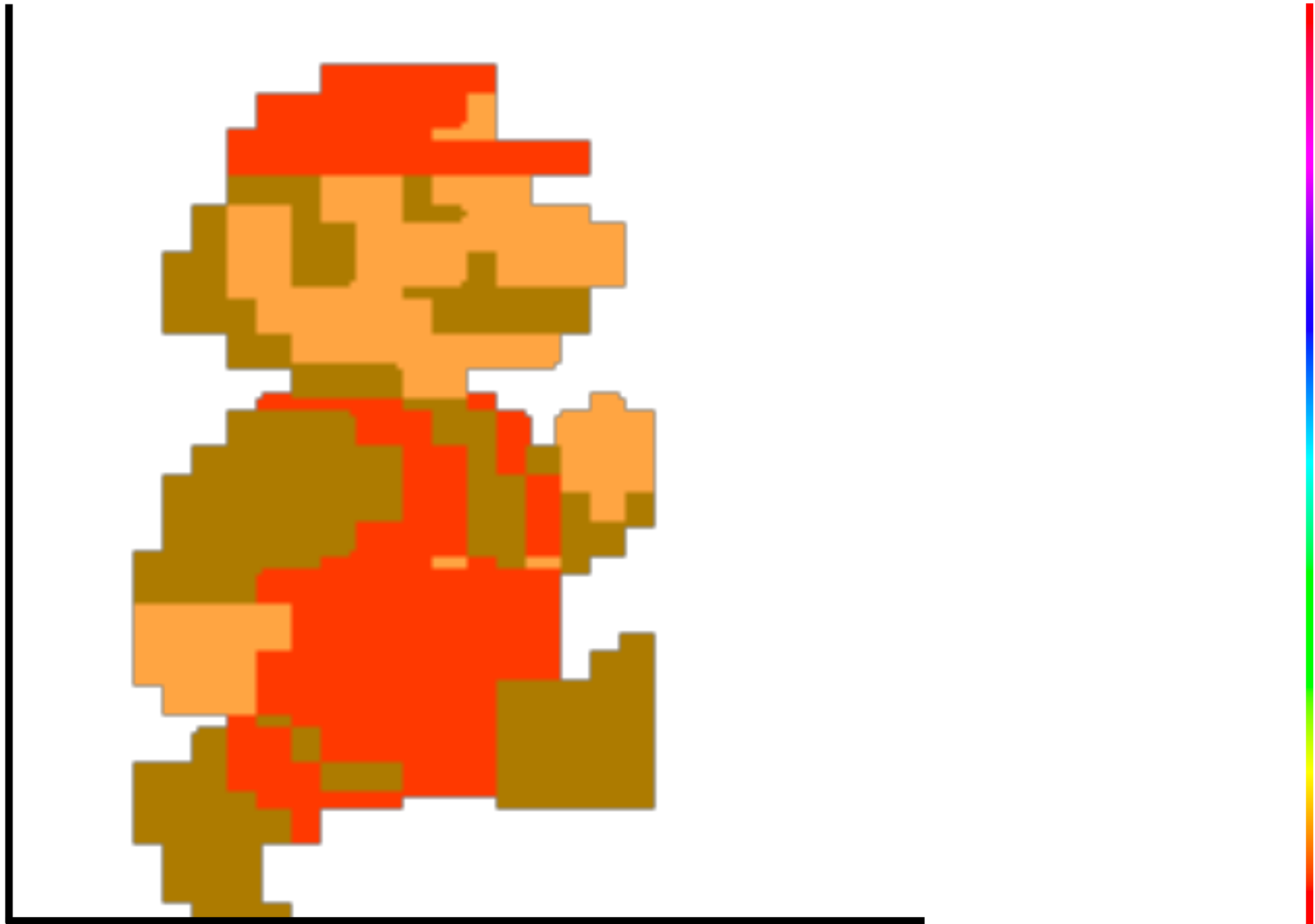
# COMPLEXITY ORTHOGONAL COMPLECTING



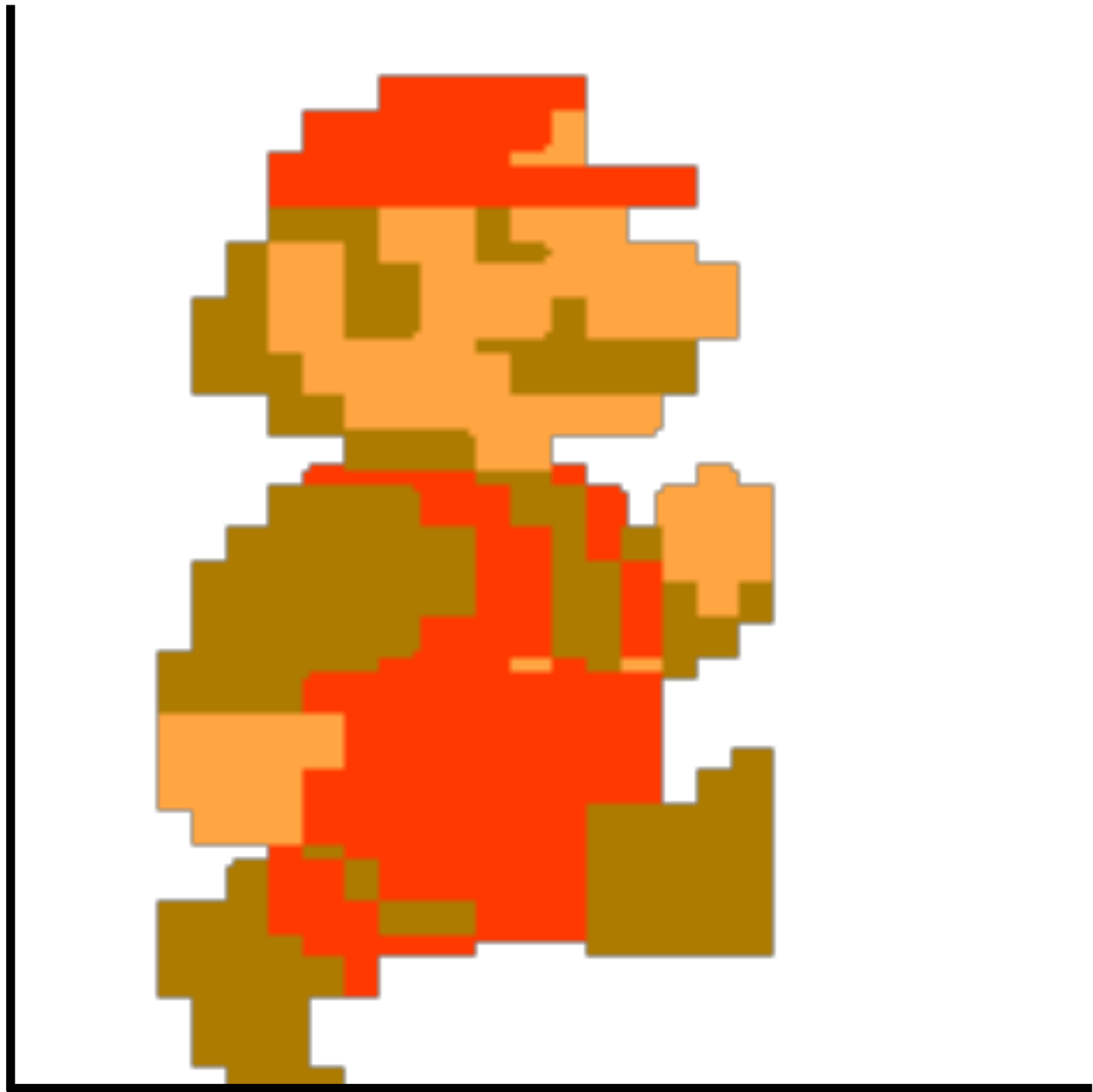
COMPLEXITY  
ORTHOGONAL COMPLETING



COMPLEXITY  
ORTHOGONAL COMPLECTING

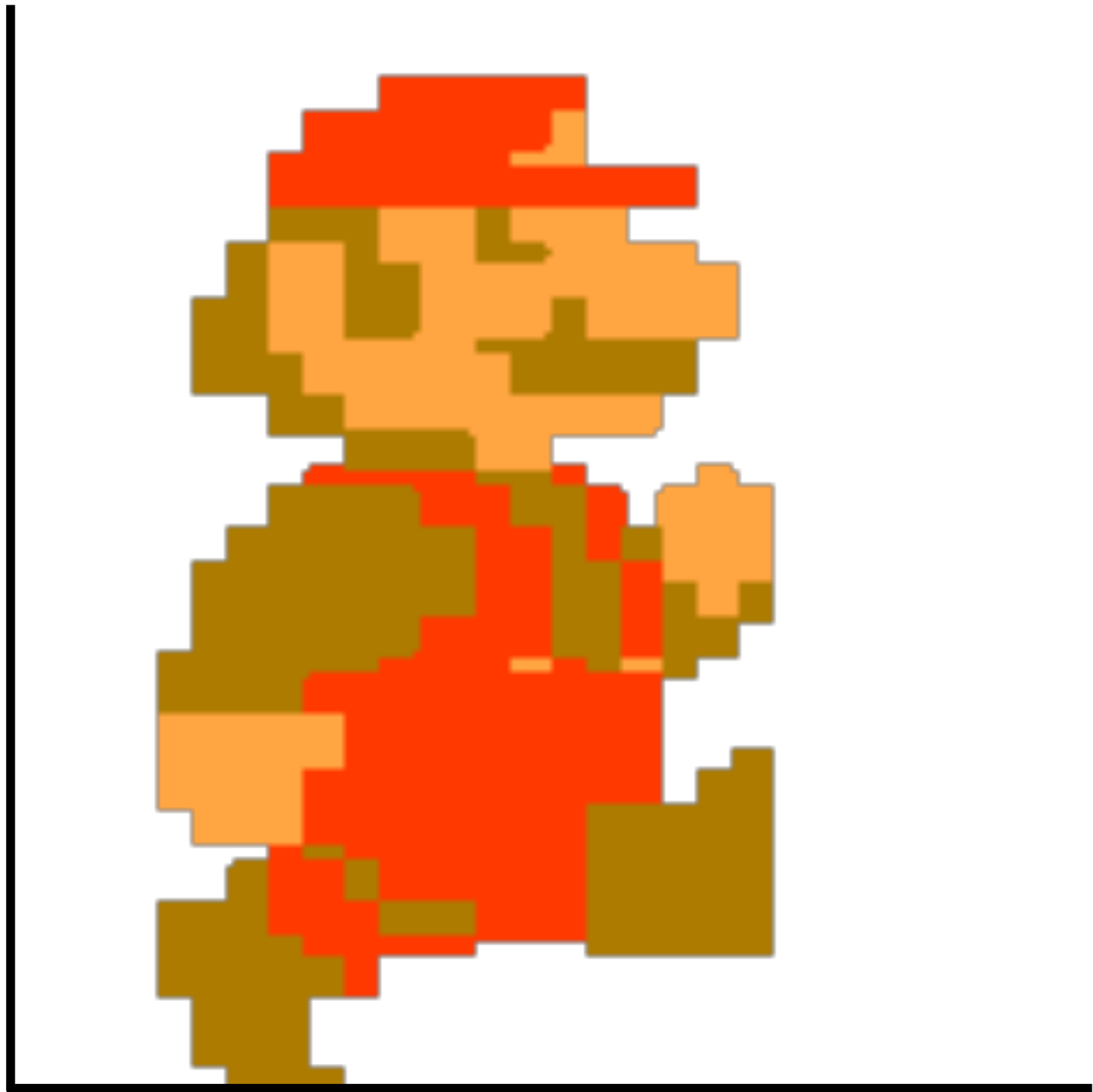


COMPLEXITY  
ORTHOGONAL COMPLECTING



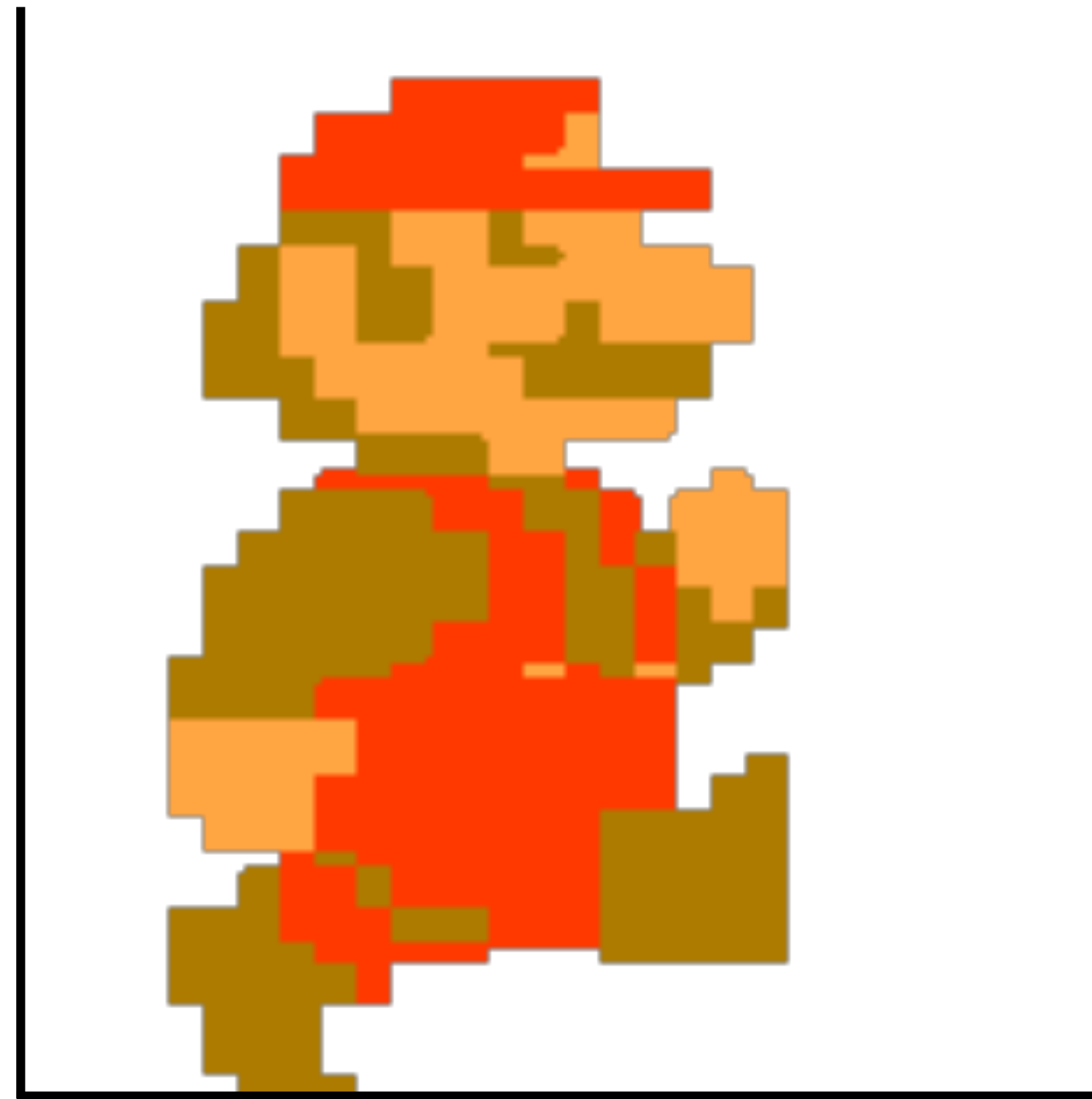


COMPLEXITY  
ORTHOGONAL COMPLECTING



**Structures: 4**

# COMPLEXITY ORTHOGONAL COMPLECTING



**Structures:** 4

**Results:** effectively limitless



# ON ABSTRACTION & DSLS



# ON ABSTRACTION & DSLS

NOT GETTING TRAPPED IN THE DETAILS



# ABSTRACTION & DSLS

## COMMONALITIES

# 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
  - 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



# 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
```

# 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
```

```
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

## COMMONALITIES

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

```
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
```

```
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

## COMMONALITIES

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

```
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
```

```
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 — “front end” interface

# ABSTRACTION & DSLS

## START FROM RULES

- Describe **what** the overall solution looks like — “front end” interface
- Choose **how** it gets run contextually — “back end” runner

ABSTRACTION & DSLS  
TWO-PHASE



# ABSTRACTION & DSLS

## TWO-PHASE

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

# ABSTRACTION & DSLS

## TWO-PHASE

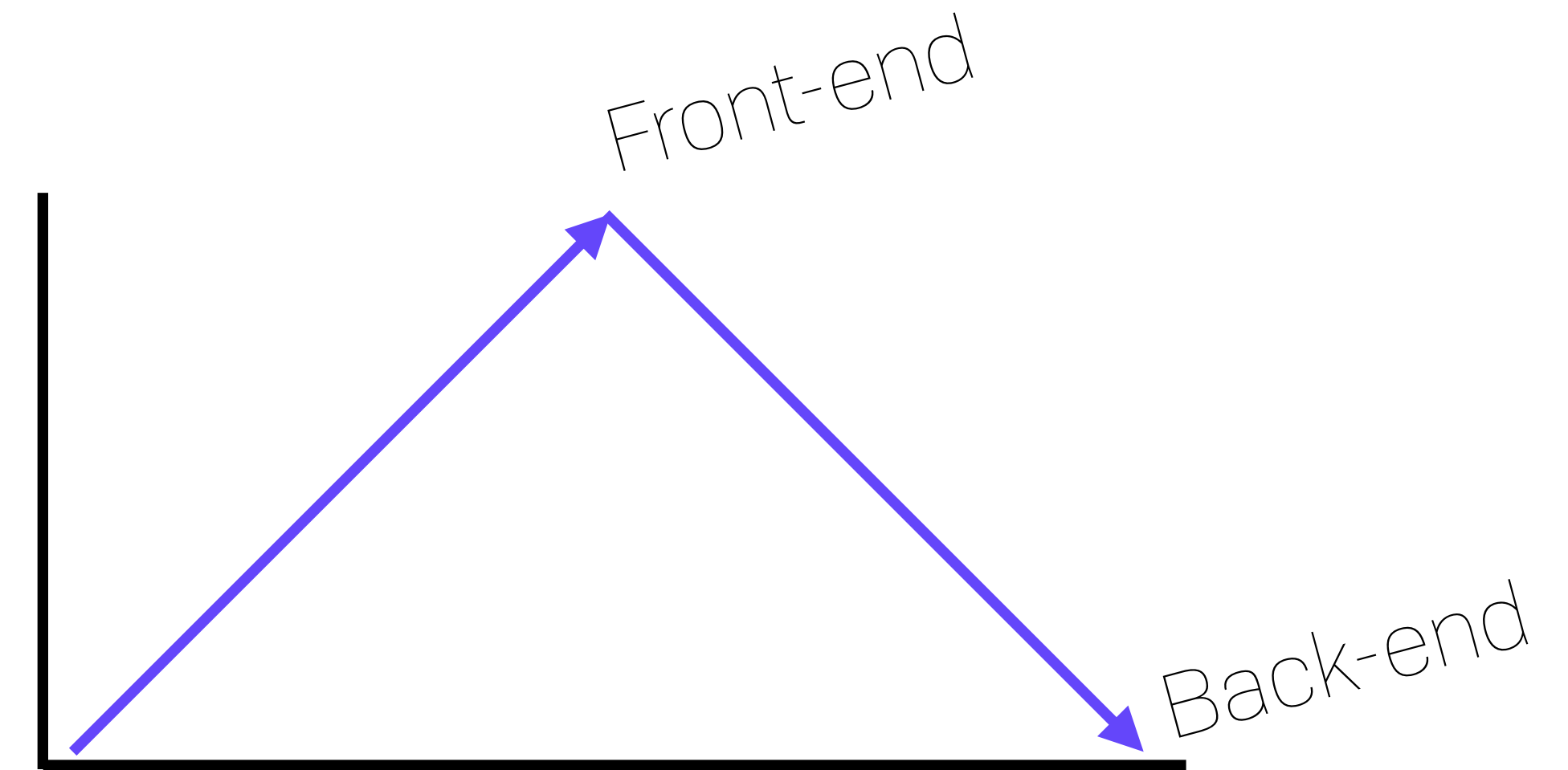
- Always a two-phase process
- Abstract, then concrete
- Do concretion at application boundary



# ABSTRACTION & DSLS

## TWO-PHASE

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



ABSTRACTION & DSLS  
IMPROVING Kernel

# ABSTRACTION & DSLS

## IMPROVING Kernel

- Fallback keys
- Bang-functions



ABSTRACTION & DSLS

IMPROVING `Kernel` — FALLBACK KEYS

ABSTRACTION & DSLS

IMPROVING `Kernel` — FALLBACK KEYS

- Composition is at the heart of modularity

ABSTRACTION & DSLS

IMPROVING `Kernel` — FALLBACK KEYS

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

ABSTRACTION & DSLS

## IMPROVING `Kernel` — FALLBACK KEYS

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

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

ABSTRACTION & DSLS

## IMPROVING `Kernel` — FALLBACK KEYS

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

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

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



## ABSTRACTION & DSLS

### IMPROVING `Kernel` — FALLBACK KEYS

- Composition is at the heart of modularity
- Orthogonality is at the heart of composition
- Let's abstract default values!

```
def get(map, key, default \\ n+1)  
  
%{a: 1} |> Map.get(:b, 4)  
#=> 4
```

## ABSTRACTION & DSLS

### IMPROVING `Kernel` — FALLBACK KEYS

- Composition is at the heart of modularity
- Orthogonality is at the heart of composition
- Let's abstract default values!

```
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` — FALLBACK KEYS

- Composition is at the heart of modularity
- Orthogonality is at the heart of composition
- Let's abstract default values!

```
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` — FALLBACK KEYS

- Composition is at the heart of modularity
- Orthogonality is at the heart of composition
- Let's abstract default values!

```
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` — FALLBACK KEYS

- Composition is at the heart of modularity
- Orthogonality is at the heart of composition
- Let's abstract default values!



```
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
```



## ABSTRACTION & DSLS

### IMPROVING `Kernel` — FALLBACK KEYS

- Composition is at the heart of modularity
- Orthogonality is at the heart of composition
- Let's abstract default values!
  - More focused (does one thing)



```
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
```


## ABSTRACTION & DSLS

### IMPROVING `Kernel` — FALLBACK KEYS

- 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)

```
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
```



## ABSTRACTION & DSLS

### IMPROVING `Kernel` — FALLBACK KEYS

- 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
```

ABSTRACTION & DSLS

IMPROVING `Kernel` — BANG FUNCTIONS

ABSTRACTION & DSLS


IMPROVING `Kernel` — BANG FUNCTIONS

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



ABSTRACTION & DSLS


IMPROVING `Kernel` — BANG FUNCTIONS



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

ABSTRACTION & DSLS

IMPROVING `Kernel` — 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: "..."}
```

# ABSTRACTION & DSLS

## IMPROVING `Kernel` — BANG FUNCTIONS

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}
```





# ABSTRACTION & DSLS

## IMPROVING `Kernel` — BANG FUNCTIONS

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
```

# ABSTRACTION & DSLS

## IMPROVING Kernel — BANG FUNCTIONS


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

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

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

```




# ABSTRACTION & DSLS

## IMPROVING Kernel — BANG FUNCTIONS


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

OK 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` — BANG FUNCTIONS

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*








# ABSTRACTION & DSLS

## IMPROVING Kernel — BANG FUNCTIONS

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

ABSTRACTION

STORYTELLING



# ABSTRACTION STORYTELLING

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



# ABSTRACTION STORYTELLING

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

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

FIGHTING GenSoup



# FIGHTING GenSoup




FIGHTING GenSoup

GOOD INTERFACES != GOOD ABSTRACTIONS



FIGHTING GenSoup

GOOD INTERFACES != GOOD ABSTRACTIONS

- GenServer & co are actually *pretty low level*
  - Add some semantics!
- Don't reinvent the wheel every time 
- Let's look at a very common example

FIGHTING GenSoup

ABSTRACTION — INTERFACE / FRONT END

FIGHTING GenSoup

ABSTRACTION — INTERFACE / FRONT END

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

FIGHTING GenSoup

SIMPLE SYNCHRONOUS CASE (BACK END)

```
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 — UNDERLYING MECHANICS

```
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 — IMPLEMENTATION (BACK END)

```
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
- Encapsulate the detail
- Don't have to think about mechanics anymore



FIGHTING GenSoup

ABSTRACTION = FOCUS/ESSENCE

FIGHTING GenSoup

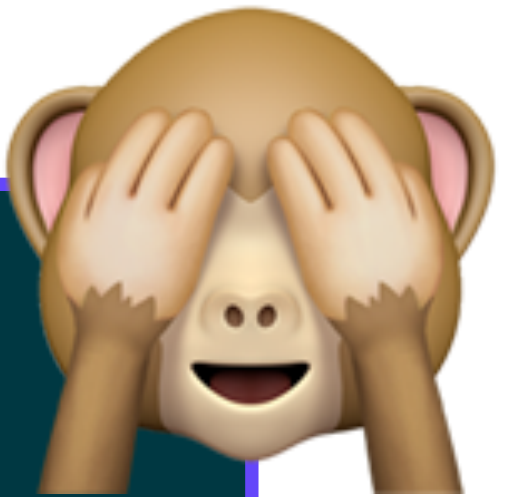
ABSTRACTION = FOCUS/ESSENCE

```
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
```







LET'S DO SOMETHING WILD



LET'S DO SOMETHING WILD

⚡ 🔥 POWER UP 🌀 🌊



POWER UP

EXPLICIT ASSUMPTIONS

# POWER UP EXPLICIT ASSUMPTIONS

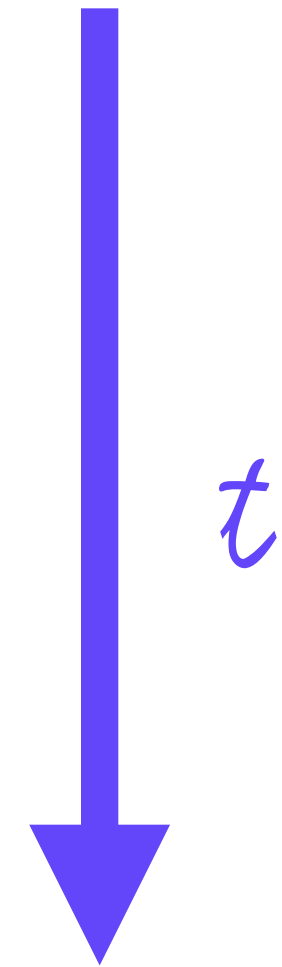
- Parallel pipes

# POWER UP EXPLICIT ASSUMPTIONS

- Parallel pipes
- Concurrency = partial order

# POWER UP EXPLICIT ASSUMPTIONS

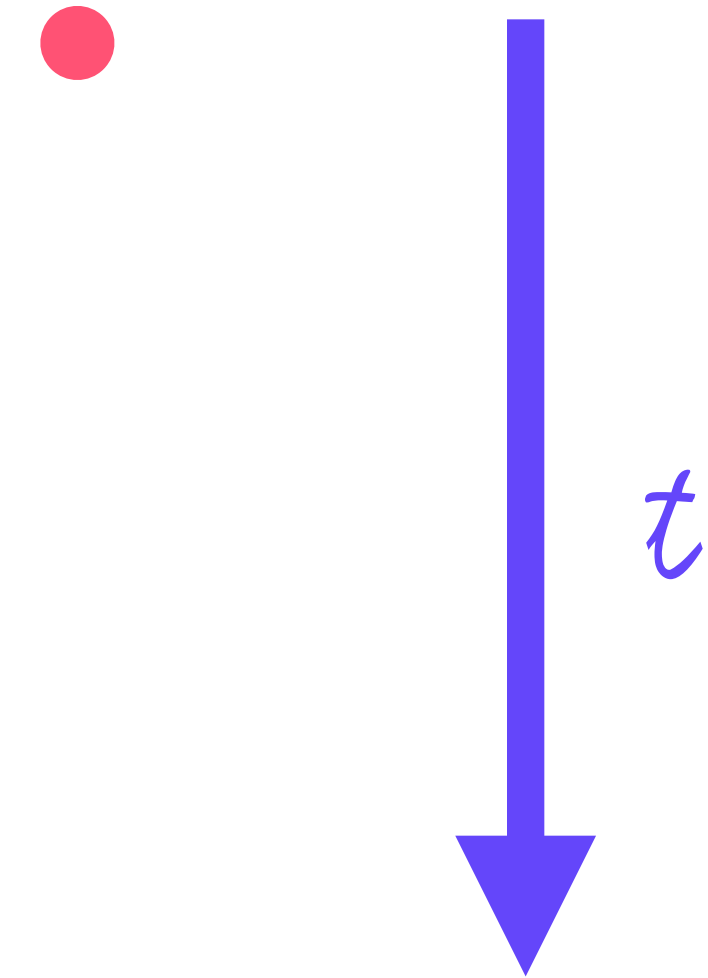
- Parallel pipes
- Concurrency = partial order





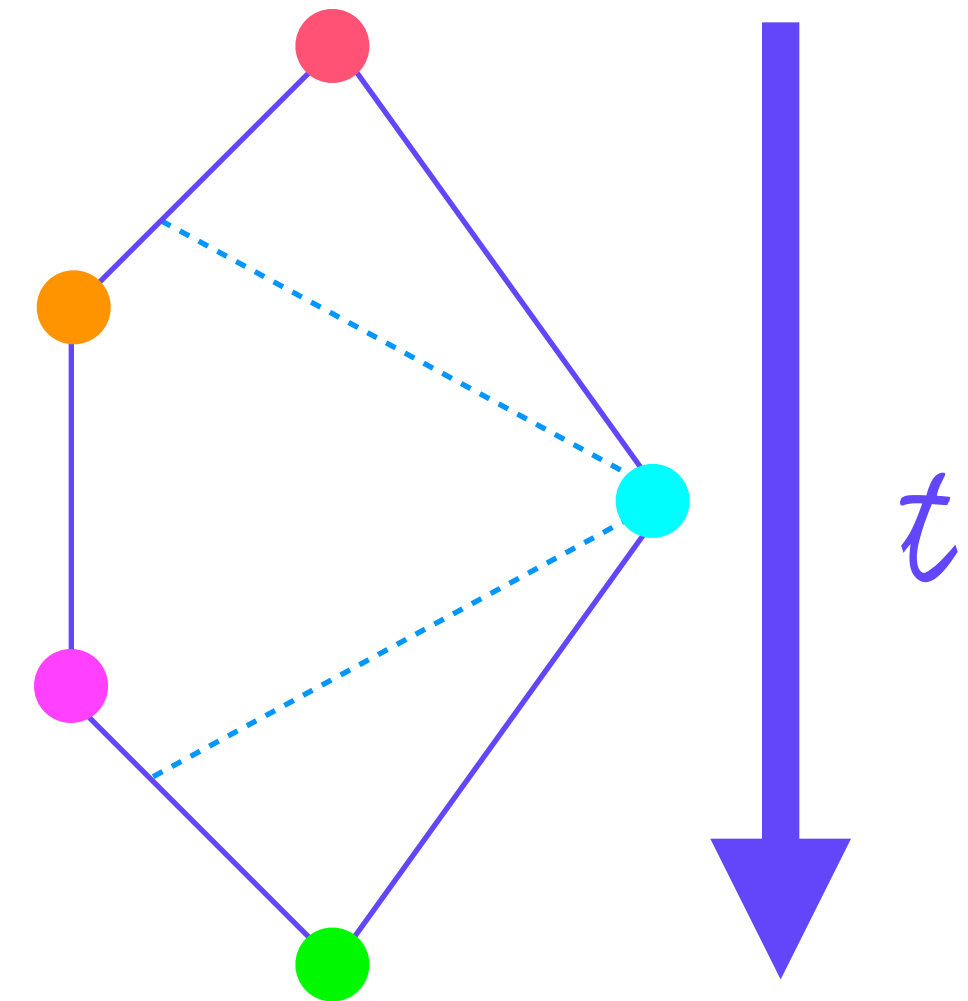
# POWER UP EXPLICIT ASSUMPTIONS

- Parallel pipes
- Concurrency = partial order



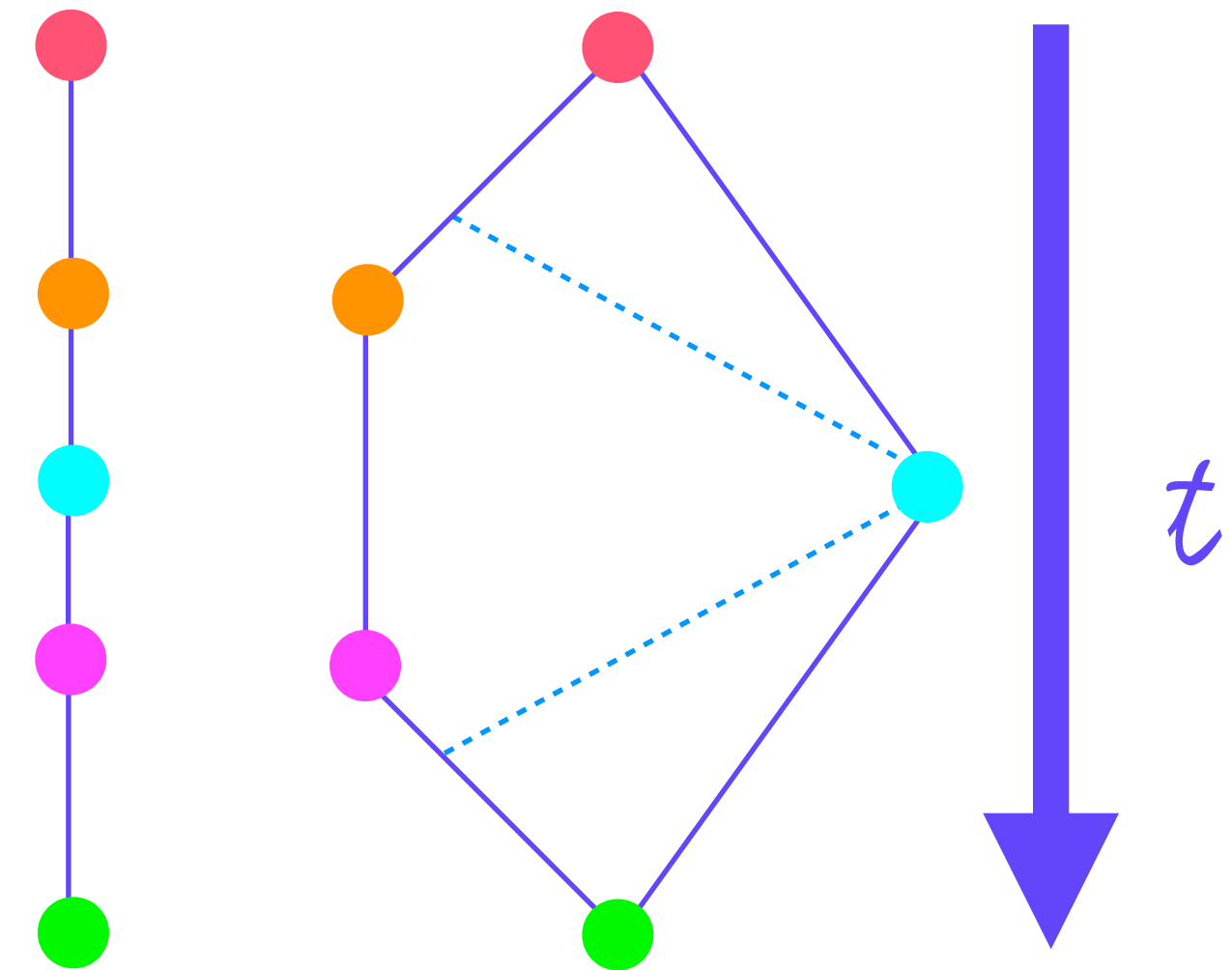
# POWER UP EXPLICIT ASSUMPTIONS

- Parallel pipes
- Concurrency = partial order



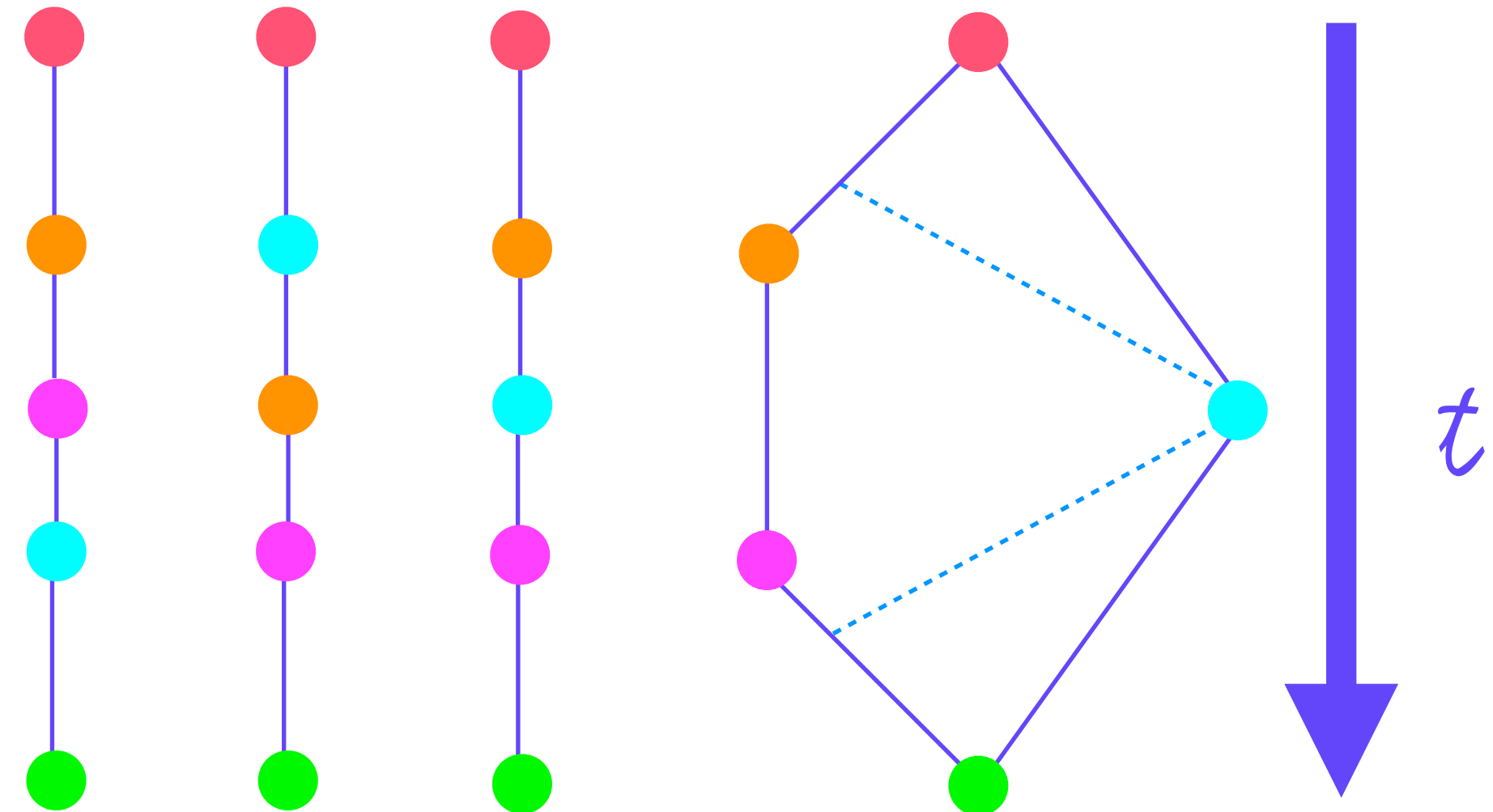
# POWER UP EXPLICIT ASSUMPTIONS

- Parallel pipes
- Concurrency = partial order



# POWER UP EXPLICIT ASSUMPTIONS

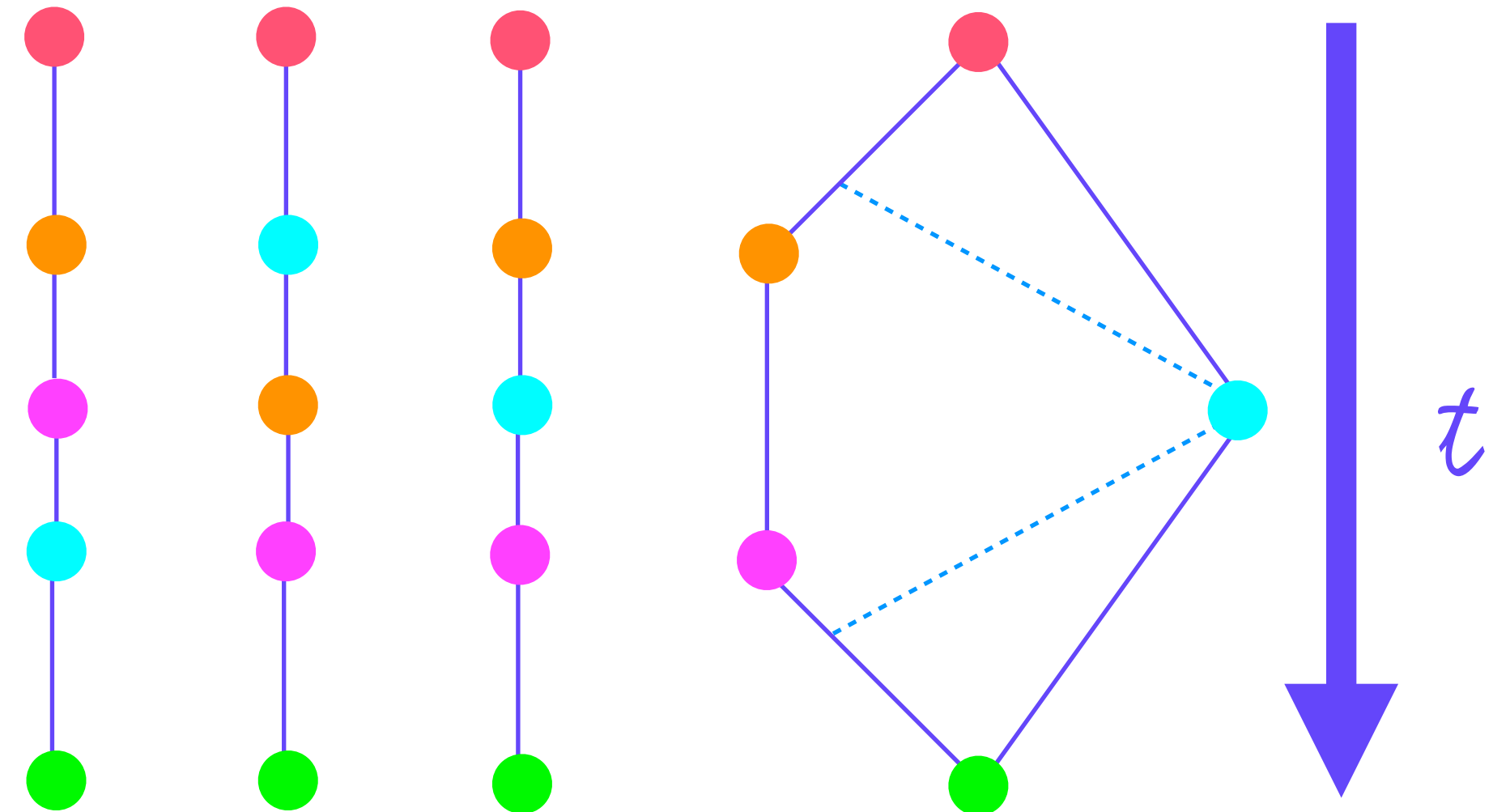
- Parallel pipes
- Concurrency = partial order





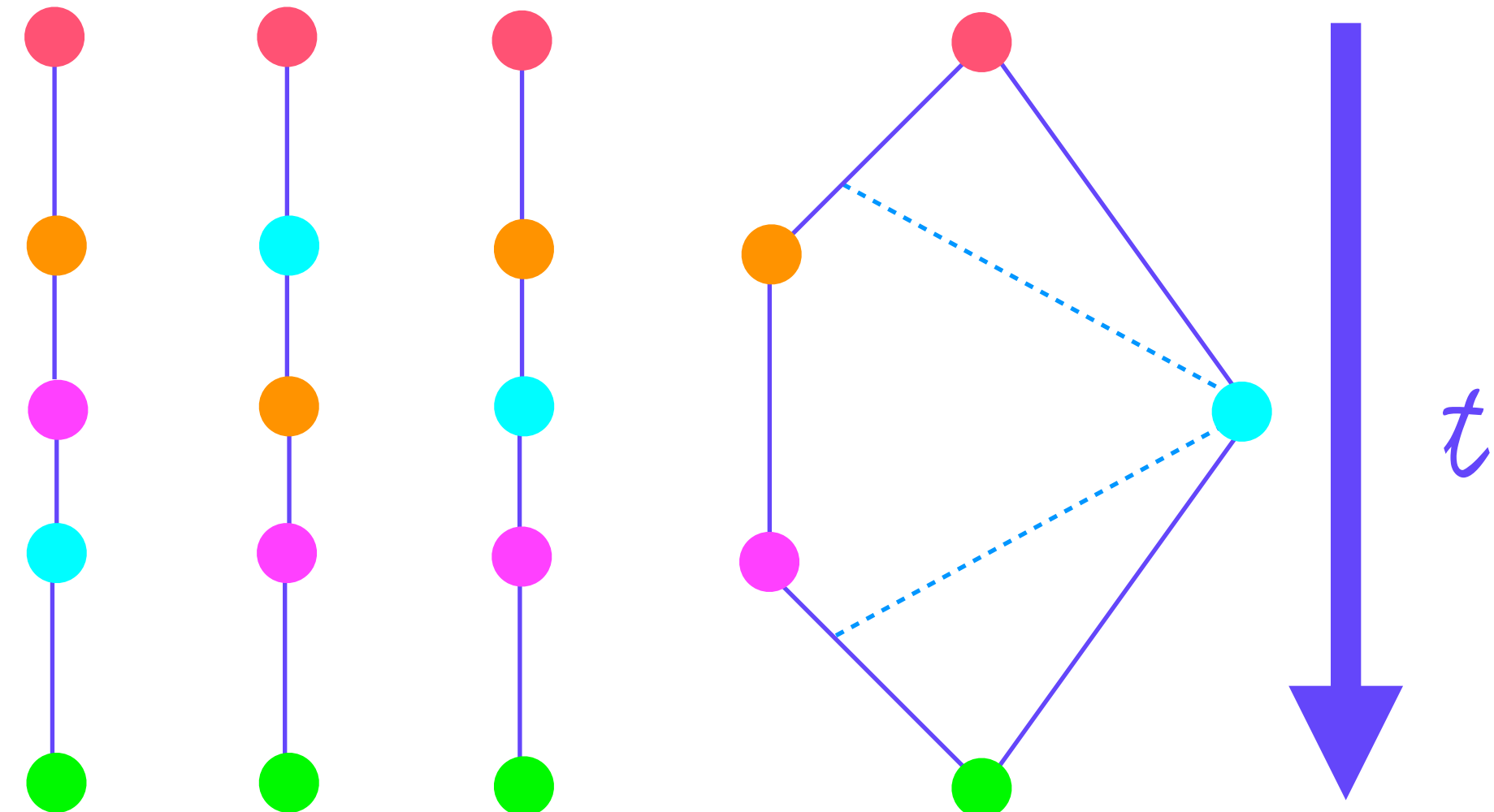
# POWER UP EXPLICIT ASSUMPTIONS

- Parallel pipes
- Concurrency = partial order
- Monotonic

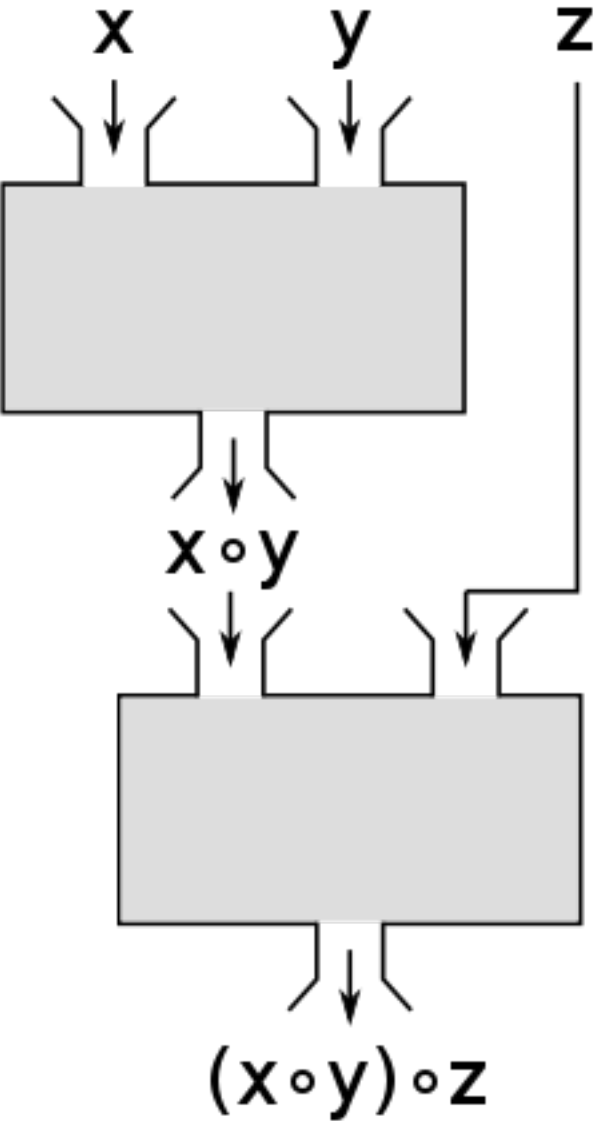


# POWER UP EXPLICIT ASSUMPTIONS

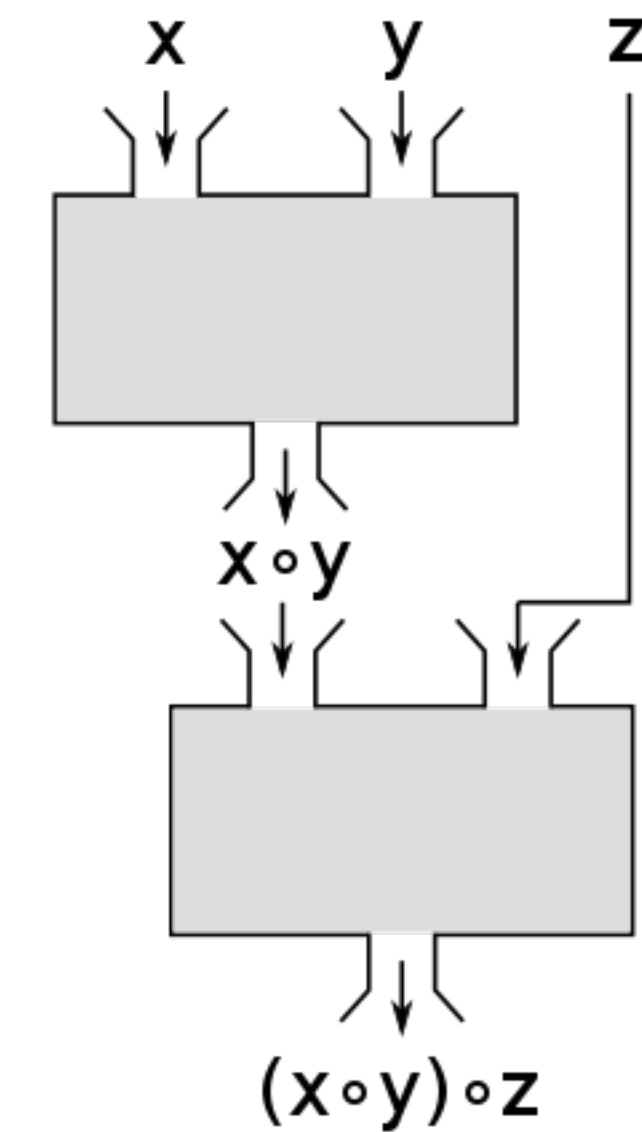
- Parallel pipes
- Concurrency = partial order
- Monotonic
- Properties
  - Serial composition
  - Parallel composition
  - Explicit evaluation strategy



POWER UP  
PIPES++



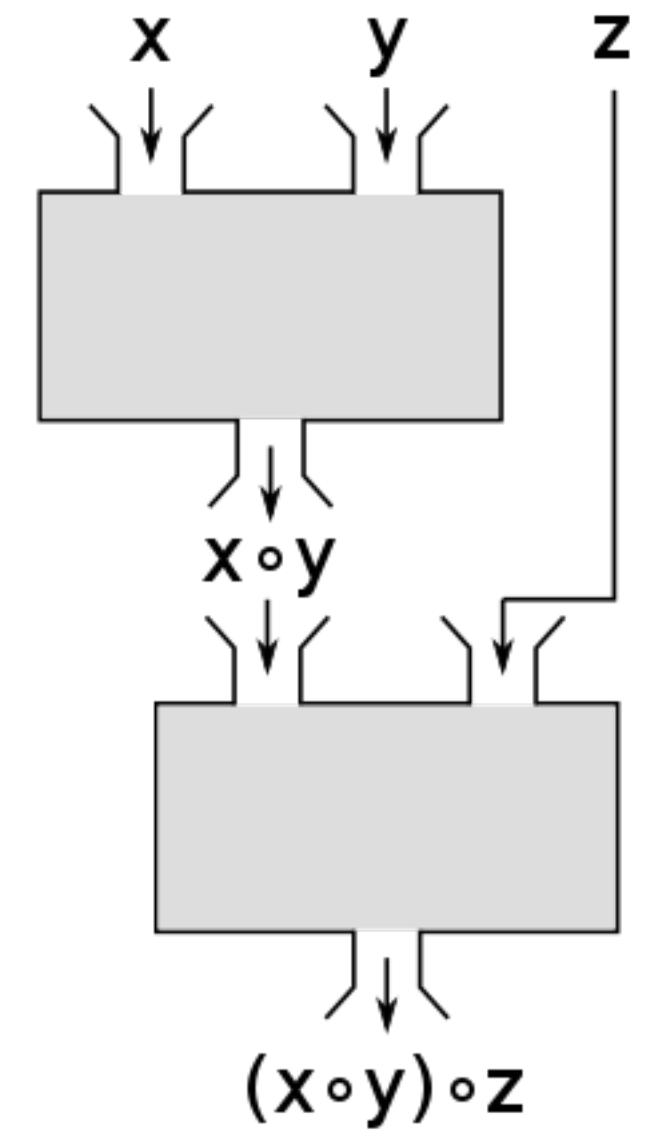
# POWER UP PIPES++



```
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)
```

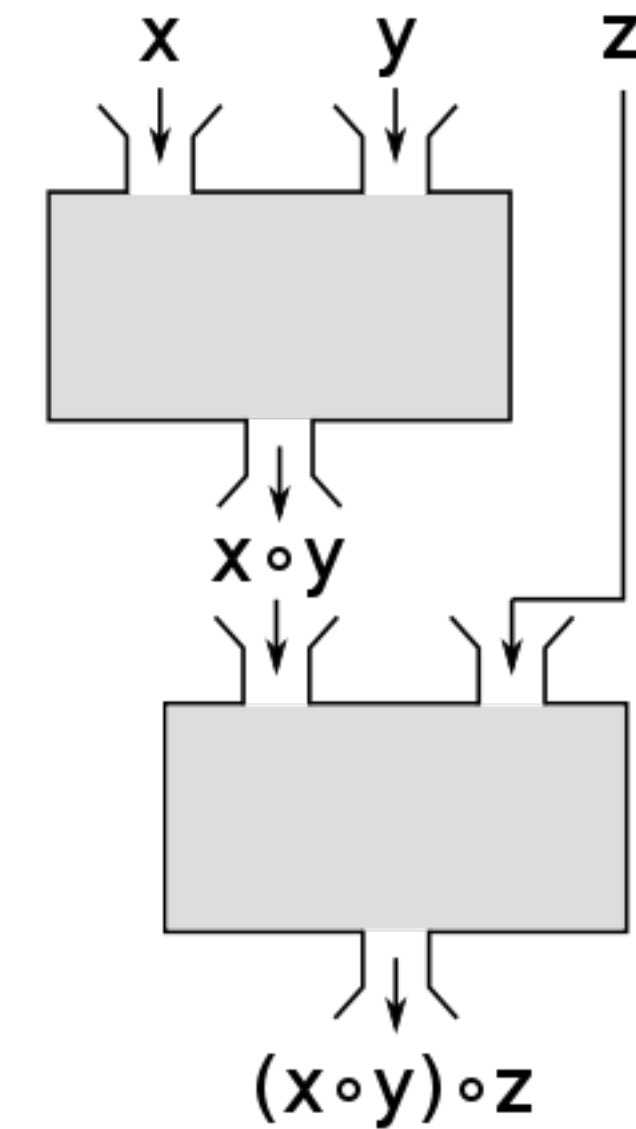
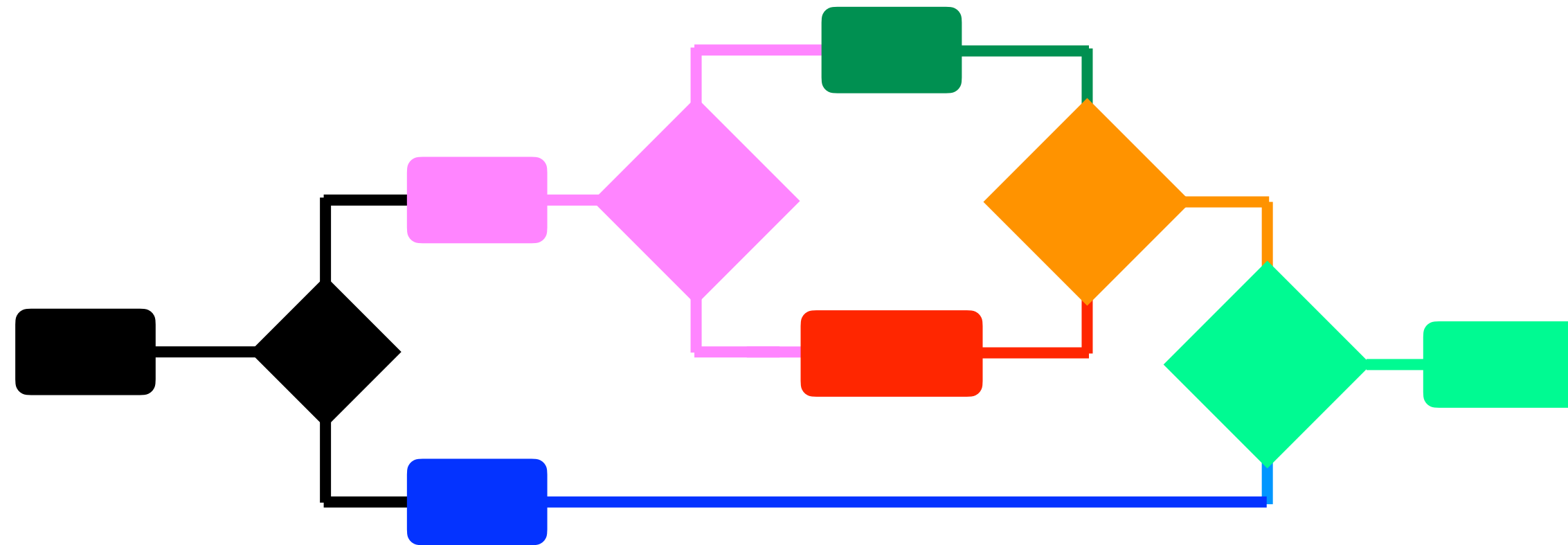


# POWER UP PIPES++



```
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

PROTOCOL / "FRONT END"

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

# 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

SIMPLE CASE BACK END

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

POWER UP

SIMPLE CASE BACK END

```
defprotocol Dataflow do  
  def split(input, path_a, path_b)  
  def unsplit(pre_split, 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

SIMPLE CASE BACK END

```
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_split, 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 BACK END

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

POWER UP  
ASYNC BACK END

```
defprotocol Dataflow do  
  def split(input, path_a, path_b)  
  def unsplit(pre_split, 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 BACK END

```
defprotocol Dataflow do  
  def split(input, path_a, path_b)  
  def unsplit(pre_split, 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 BACK END

```
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_split, 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* (focused on meaning not mechanics)

# POWER UP UPSHOT

- Higher *semantic density* (focused on meaning not mechanics)
- Declarative, **configurable** data flow 🧠



# POWER UP UPSHOT

- Higher *semantic density* (focused on meaning not 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* (focused on meaning not 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* (focused on meaning not 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*

A dark, metallic megaphone is positioned in the upper center of the frame, angled slightly to the right. It has a large, flared horn and a handle. The background is a solid dark gray.

# A CALL FOR LIBRARIES



A large, dark megaphone is positioned in the upper center of the image, pointing downwards. At the very bottom center, there is a small, shiny gold bell. The background is a solid dark gray.

# A CALL FOR LIBRARIES



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
```

Happy Path (Continue)

Error Case (Skip)

---

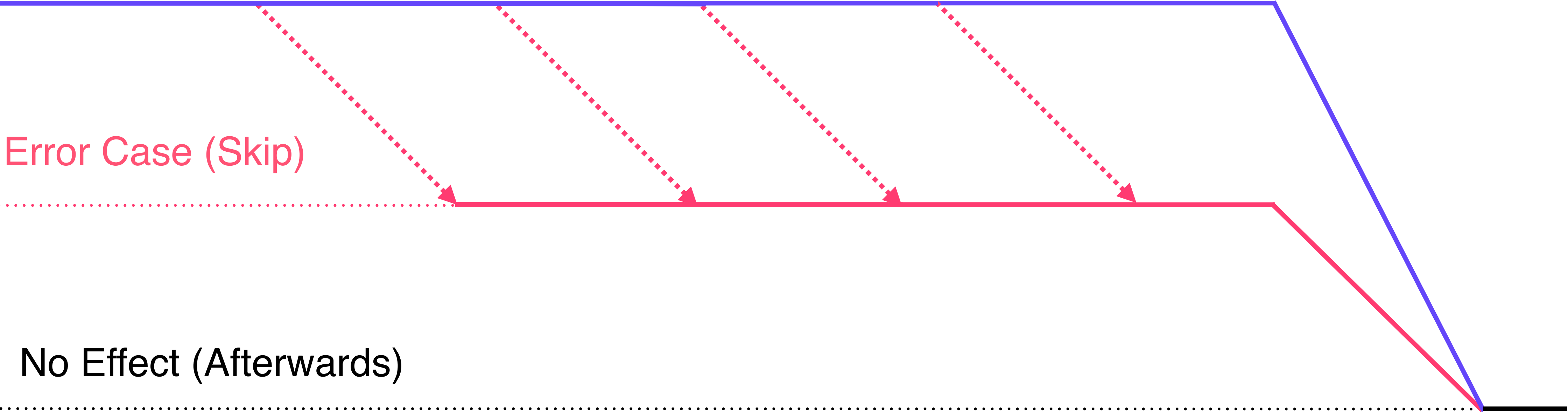
No Effect (Afterwards)

---

A CALL FOR LIBRARIES  
EXTEND RAILROAD PROGRAMMING

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

Happy Path (Continue)



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)

        bar(val + 2)

      end)
    end
  end)
end

foo(42)
```

The diagram illustrates the execution flow of the provided code. It features a dark blue background with a purple border. The code is written in a monospaced font with syntax highlighting: keywords like 'def', 'do', 'end', 'fn', and 'end)' are in yellow, function names like 'foo', 'bar', 'Task', and 'IO' are in blue, and literals like '42' are in light blue. Red arrows indicate the flow of execution: a solid red arrow points from the 'foo(42)' call at the bottom to the 'def foo' definition at the top; a solid red arrow points from the 'Task.async' call inside 'foo' to the 'bar' function definition; a dashed red arrow points from the 'IO.inspect(val)' call to the 'bar' function definition; a solid red arrow points from the 'bar' function definition to the 'Task.async' call inside 'bar'; a dashed red arrow points from the 'IO.inspect(inner\_val)' call to the 'bar' function definition; and a solid red arrow points from the 'bar(val + 2)' call to the 'bar' function definition.



# A CALL FOR LIBRARIES

## SURPRISING NUMBER OF FACTORS

Log

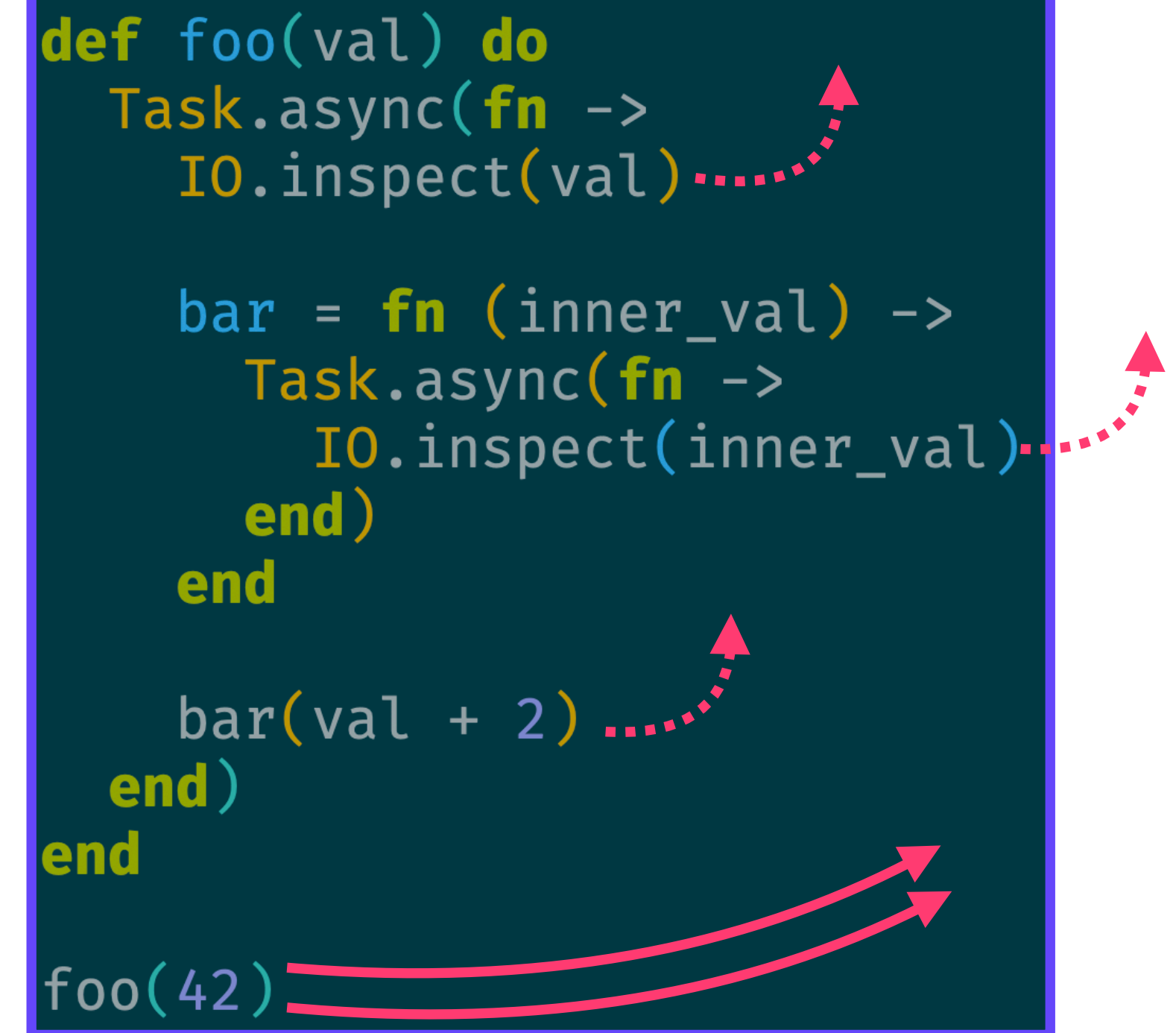
Program

```
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





# SUMMARY

A black and white photograph showing a hand holding a clock face. The clock face is partially visible, showing Roman numerals and hands. The word "SUMMARY" is overlaid in a large, white, serif font across the center of the image. The background is blurred, showing what appears to be a window or a bright light source.

SUMMARY  
KEEP IN MIND



# SUMMARY

## KEEP IN MIND

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



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



THANK YOU, INTERNET



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