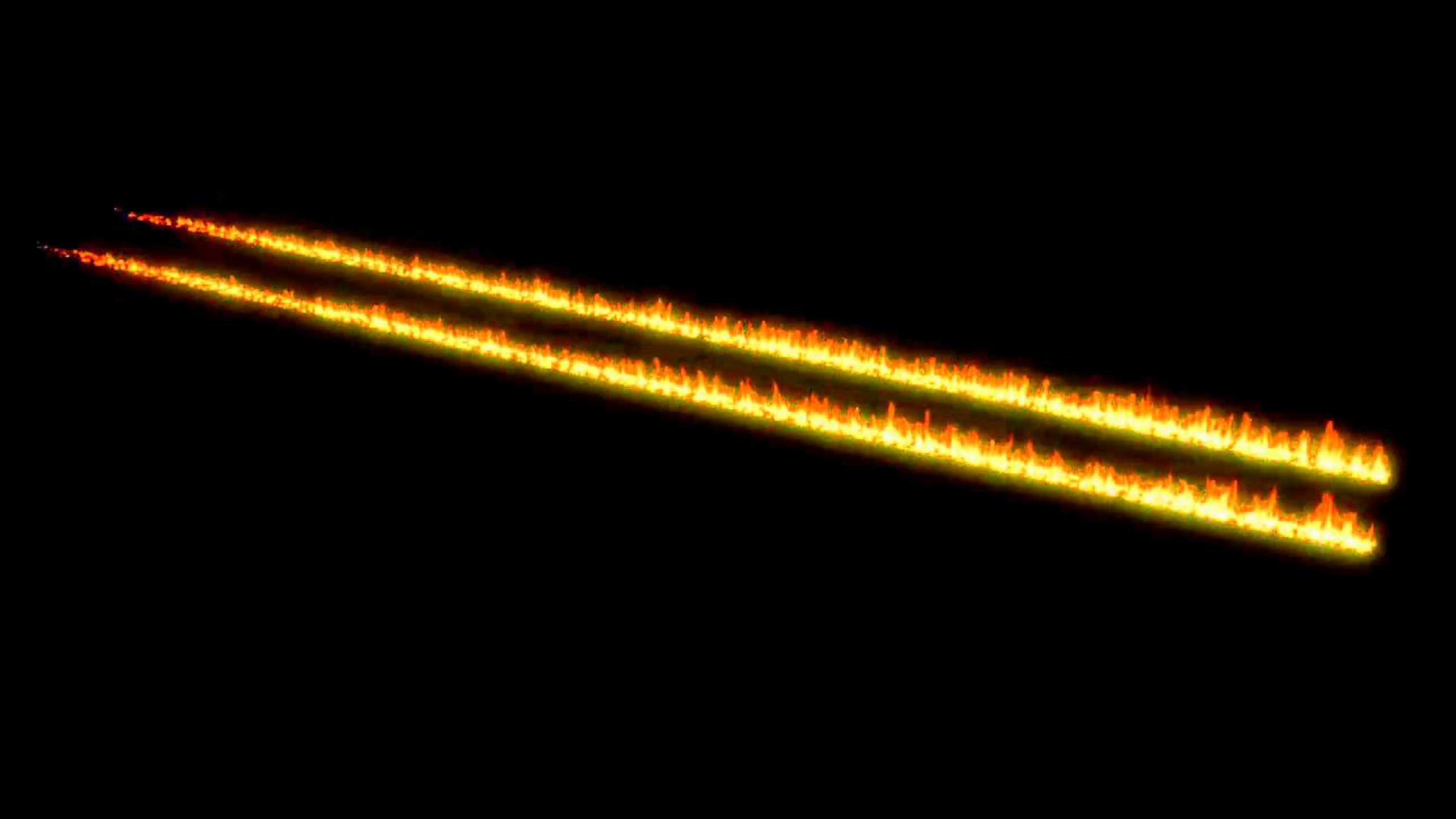
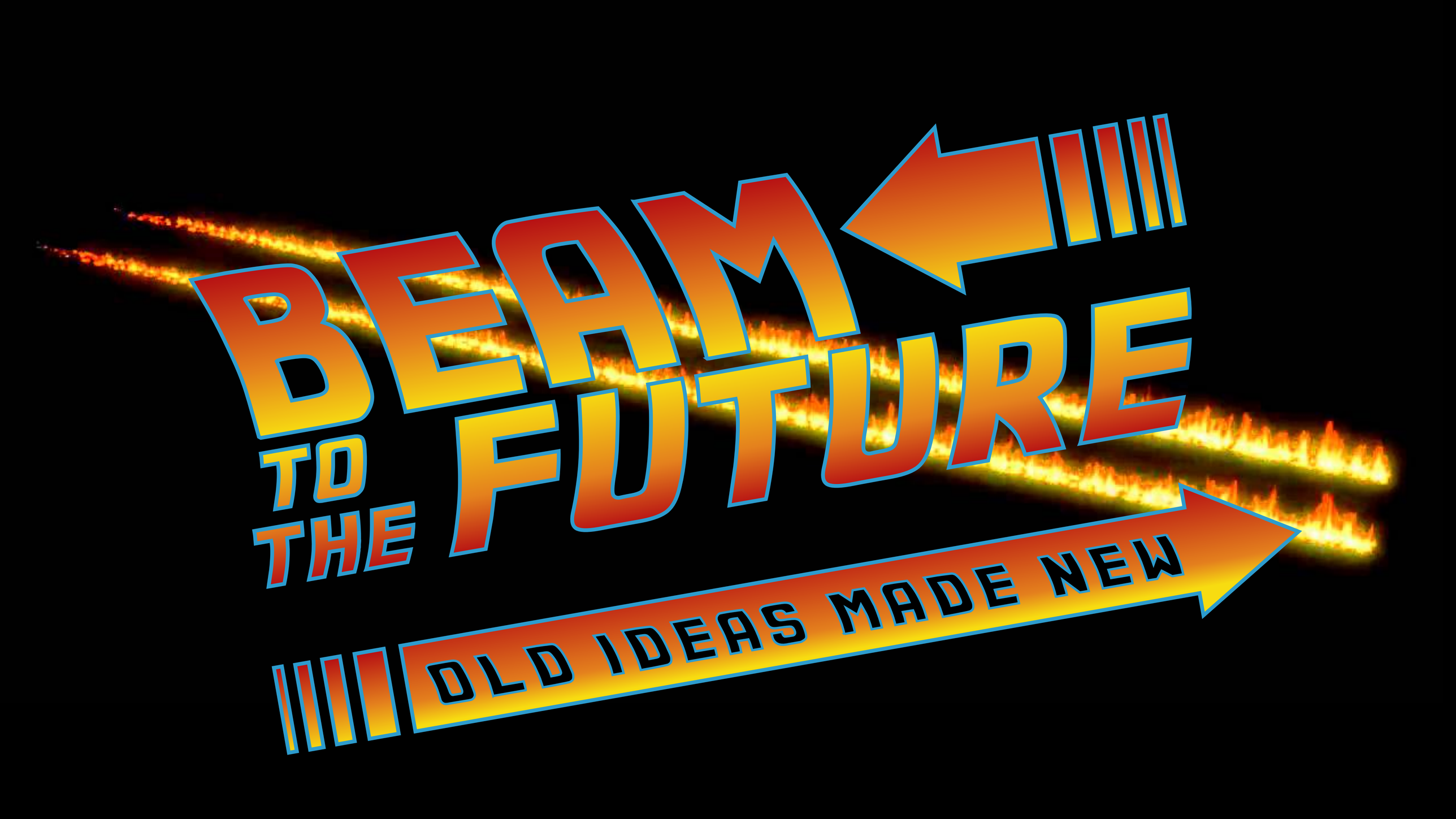



BEAM
TO THE FUTURE
OLD IDEAS MADE NEW


BEAM
TO THE FUTURE
OLD IDEAS MADE NEW
2022 Edition

Sometimes in order to keep ***moving forward***,
not only must you take one step at a time,
but you must be willing to ***look back*** occasionally and
evaluate your past, no matter how painful it is.

Looking back lets you know whether or not you are headed in
the ***right direction***.

–G.K. Adams

Epigram 53

So many good ideas are never heard from again
once they embark in a voyage on the semantic gulf

–Alan J. Perlis, Epigrams on Programming (1982)


Meta

Brooklyn Zelenka

@expede

Brooklyn Zelenka

@expede

- CTO at Fission (https://fission.codes)
  - Edge apps ("post-serverless")
  - Goal: make back-ends and DevOps obsolete
- PLT, VMs, distributed systems
- Standards: DIF, UCAN, Ethereum, Multiformats, others
- Founded VanFP, VanBEAM
- Primary author of Witchcraft, Algae, Exceptional, etc

Meta


CODE BEAM LITE
INDIA
TRAINING FROM EXPERT SPEAKERS
THE FUTURE OF THE ERLANG & ELIXIR ECOSYSTEM
ONLINE 24 MAR/2022
Hosted by Functional Conf

Meta


CODE BEAM LITE
INDIA
TRAINING FROM EXPERT SPEAKERS
THE FUTURE OF THE ERLANG & ELIXIR ECOSYSTEM
ONLINE 24 MAR/2022
Hosted by Functional Conf

Meta 🔮

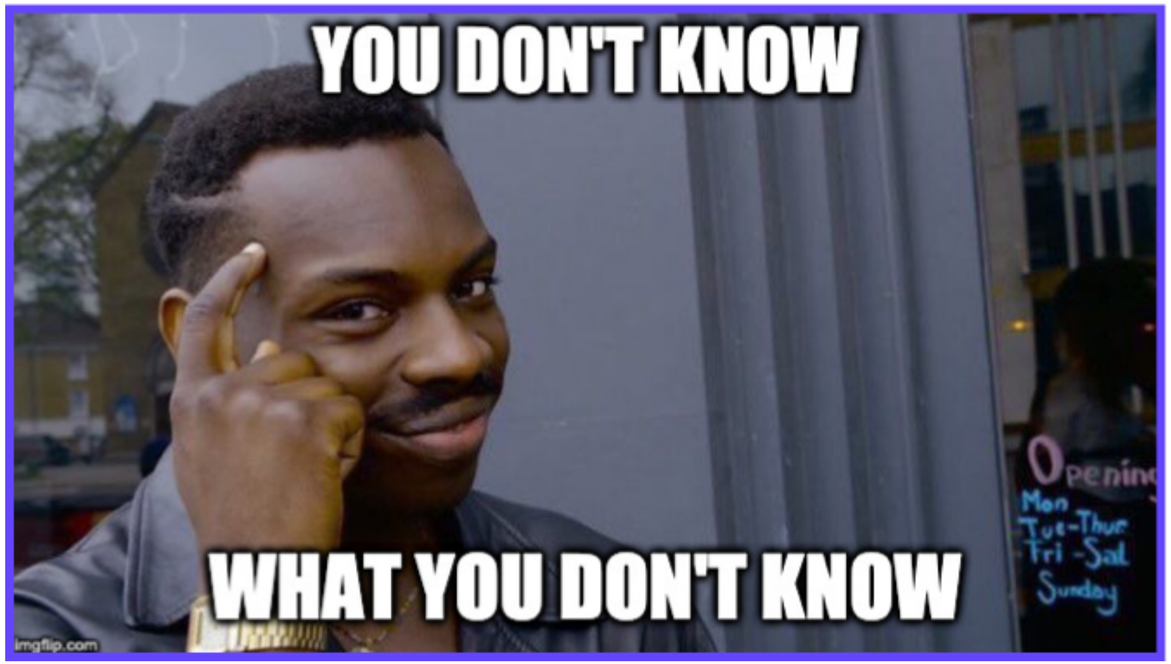
The BEAM does ***so much*** right 👏

Meta 🔮

How do we move forward
as an ecosystem,
cross language, ***cross*** paradigm?

Meta 🔮

How do we move forward
as an ecosystem,
cross language, ***cross*** paradigm?



Meta 🔮

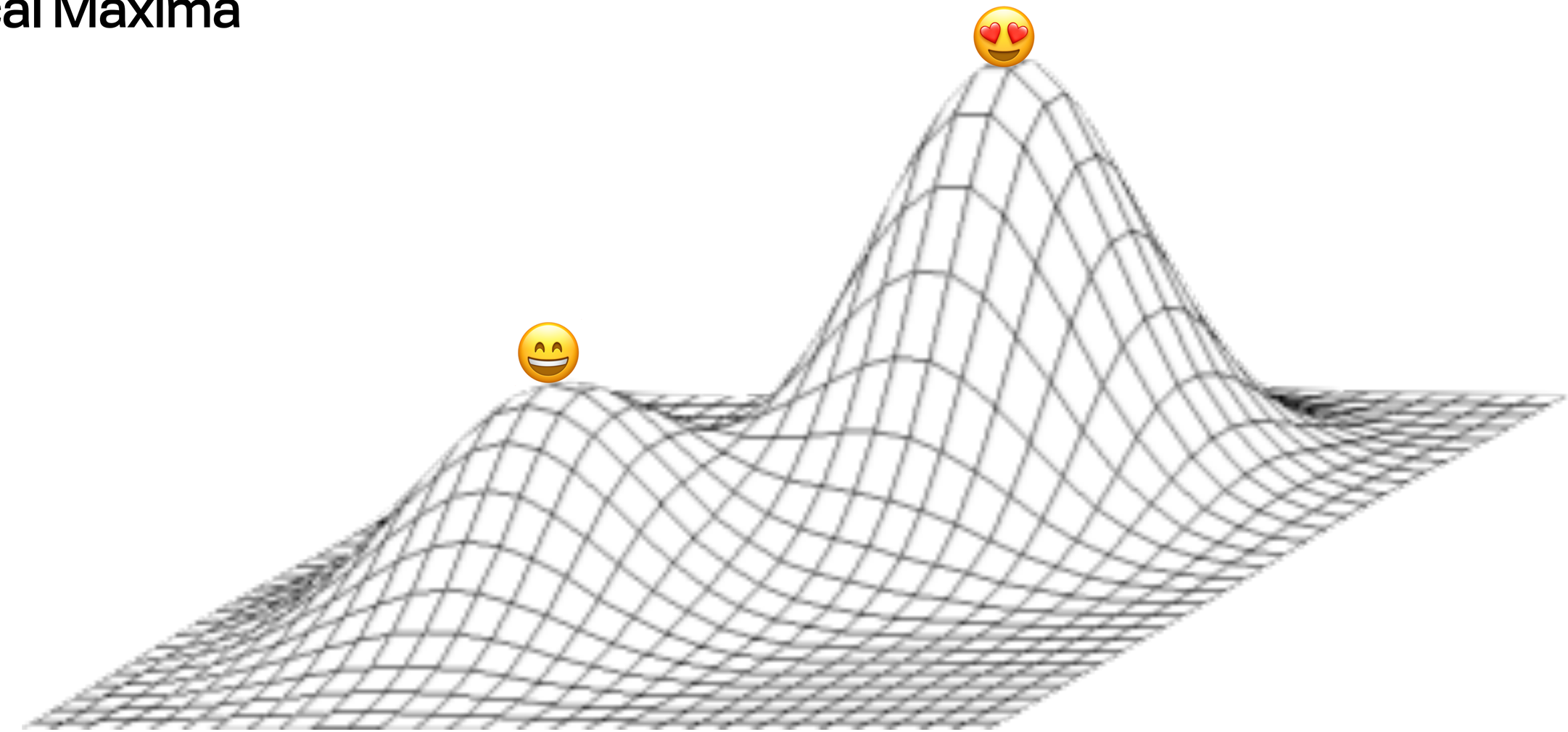
Local Maxima

Meta 🔮

Local Maxima

Meta 🔮

Local Maxima



Meta 🔮

Local Maxima

Meta 🔮

Spoiler Alert 🚨

Meta 🔮

Spoiler Alert 🚨

1. Breaking out of linear thinking / von Neumann 🌀

Meta 🔮

Spoiler Alert 🚨

1. Breaking out of linear thinking / von Neumann 🌀
2. New types of modularity (for the BEAM) 🔌

Meta 🔮

Spoiler Alert 🚨

1. Breaking out of linear thinking / von Neumann 🌀
2. New types of modularity (for the BEAM) 🔌
3. Composable languages 🗣️

In the Beginning...

In the Beginning... 🌴🦖

Epigram 28

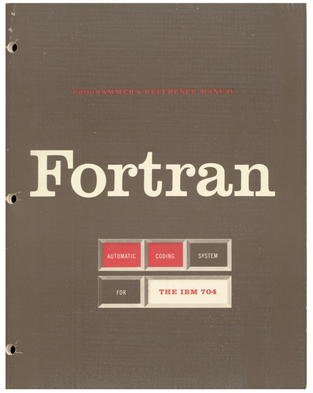
Around computers it is difficult to find the
correct unit of time to measure progress.
Some cathedrals ***took a century to complete***.
Can you imagine the grandeur and scope of a
program that would take as long?

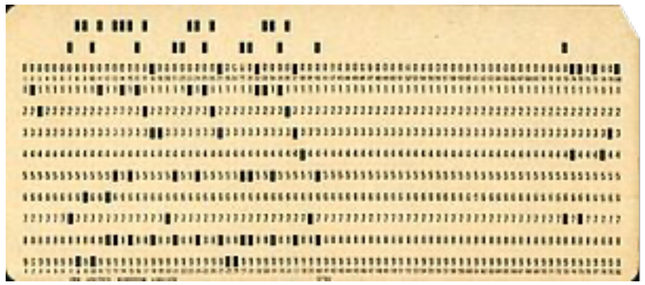
–Alan J. Perlis, Epigrams on Programming (1982)

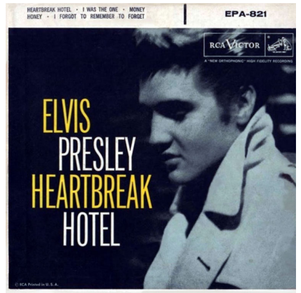
In the Beginning... 🌴🦖

In the Beginning... 🌴🦖

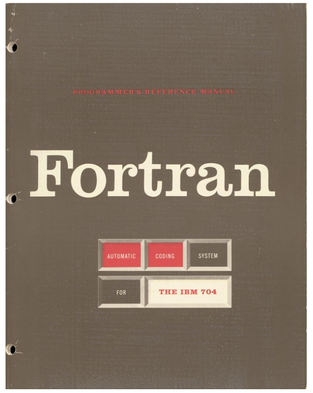
In the Beginning... 🌴🦖



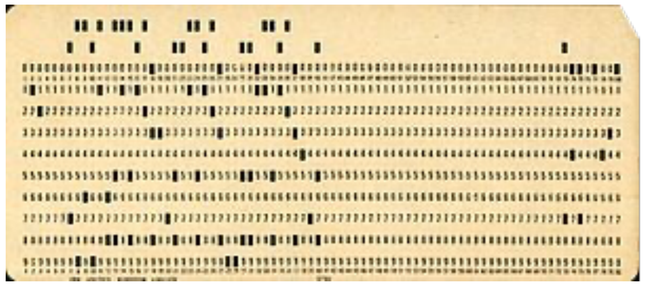


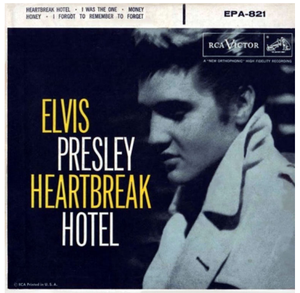


In the Beginning... 🌴🦖

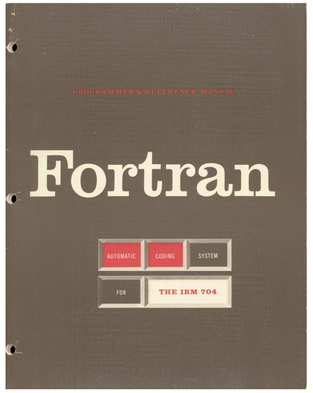


ERLANG



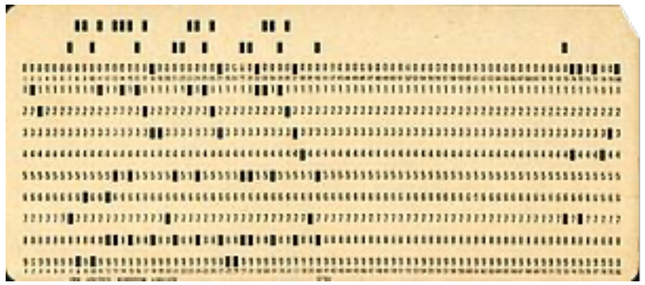


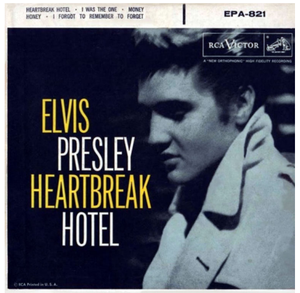
In the Beginning... 🌴🦖



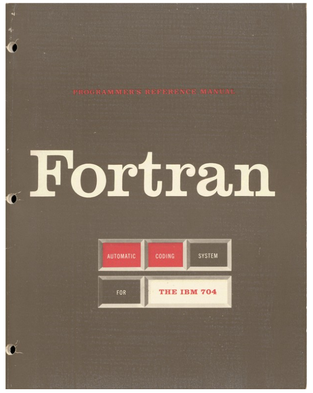
29 Years

ERLANG



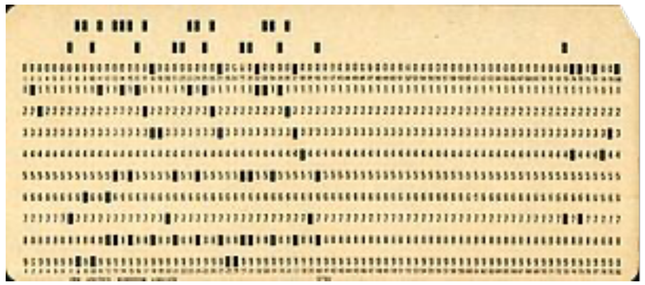


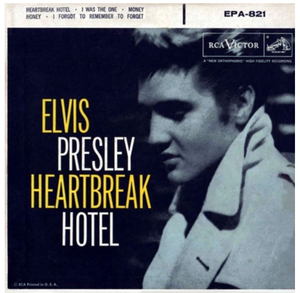
In the Beginning... 🌴🦖



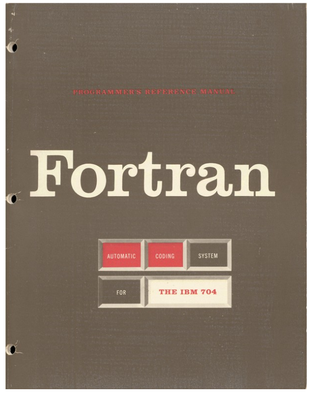
29 Years

ERLANG



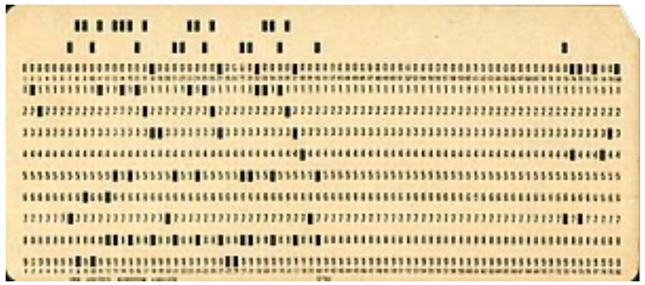


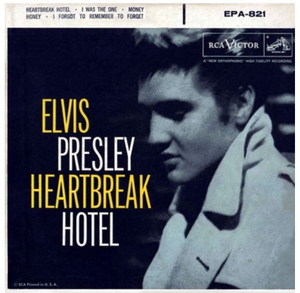
In the Beginning... 🌴🦖



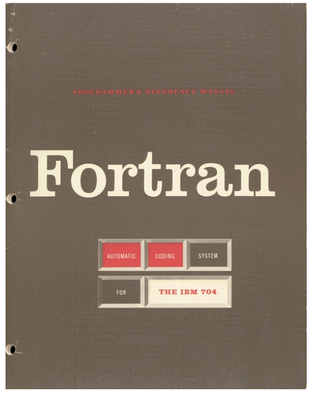
29 Years

ERLANG





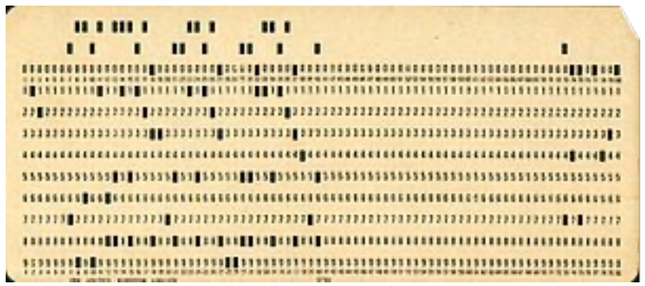
In the Beginning... 🌴🦖

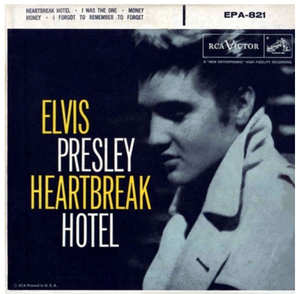


29 Years

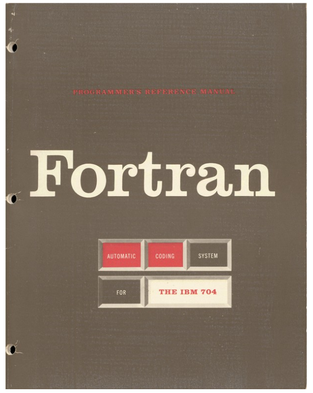
ERLANG

CODE BEAM



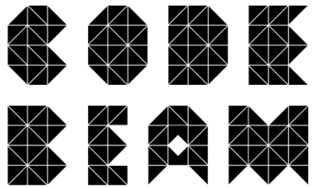


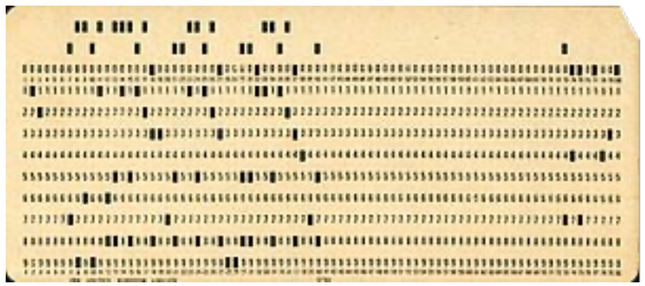
In the Beginning... 🌴🦖

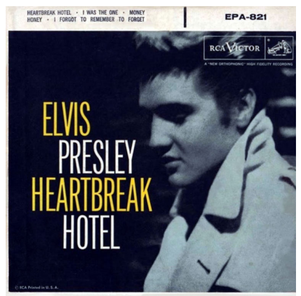


29 Years

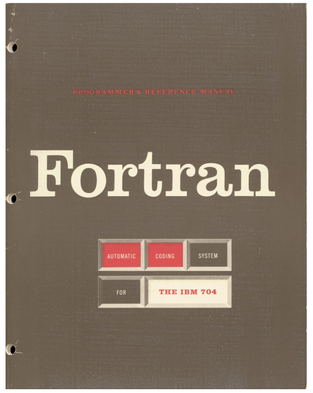
36 Years







In the Beginning... 🌴🦖

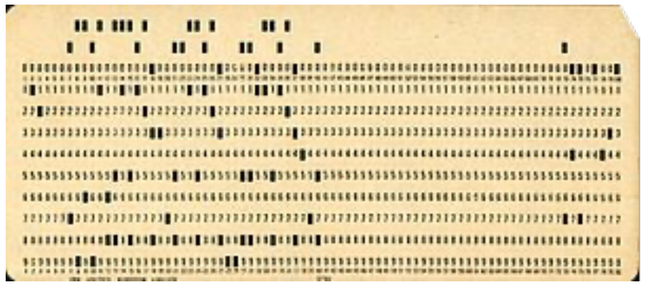


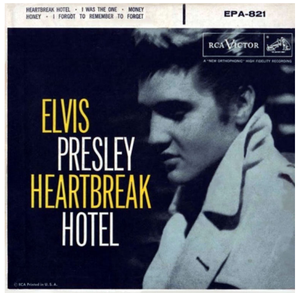
29 Years

ERLANG

36 Years

CODE BEAM





Paradigm Redshift

Paradigm Redshift

It's really difficult to distinguish
a ***new paradigm***
from a ***really bad*** idea
[...]
The ***new shiny object***
is part of the old paradigm

– Douglas Crockford, The Power of the Paradigm (2018)

Paradigm Redshift 🌈

Novelty Budget

Paradigm Redshift 🌈

Novelty Budget

Paradigm Redshift 🌈

Novelty Budget

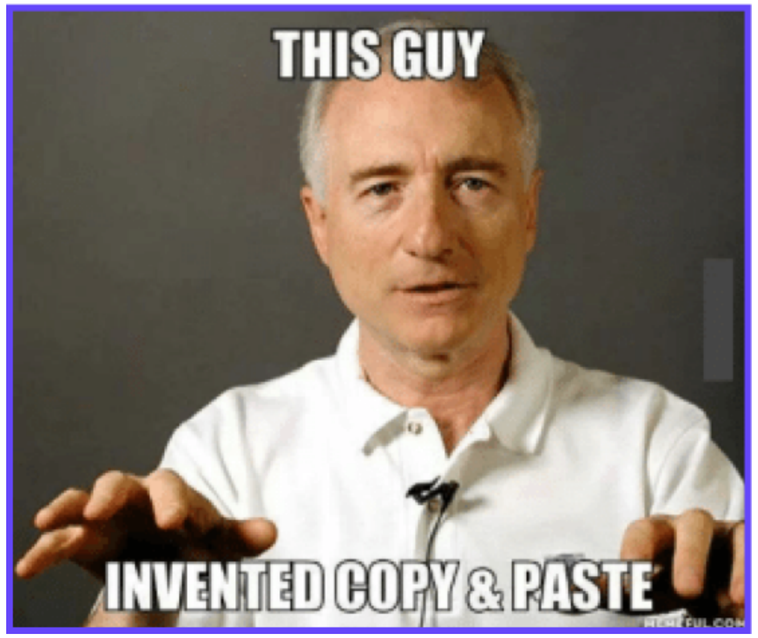
Paradigm Redshift 🌈

Law of Conservation of Complexity

Paradigm Redshift 🌈

Law of Conservation of Complexity

Every application has an inherent amount of complexity
that ***cannot be removed*** or hidden,
but ***only moved*** from place to place



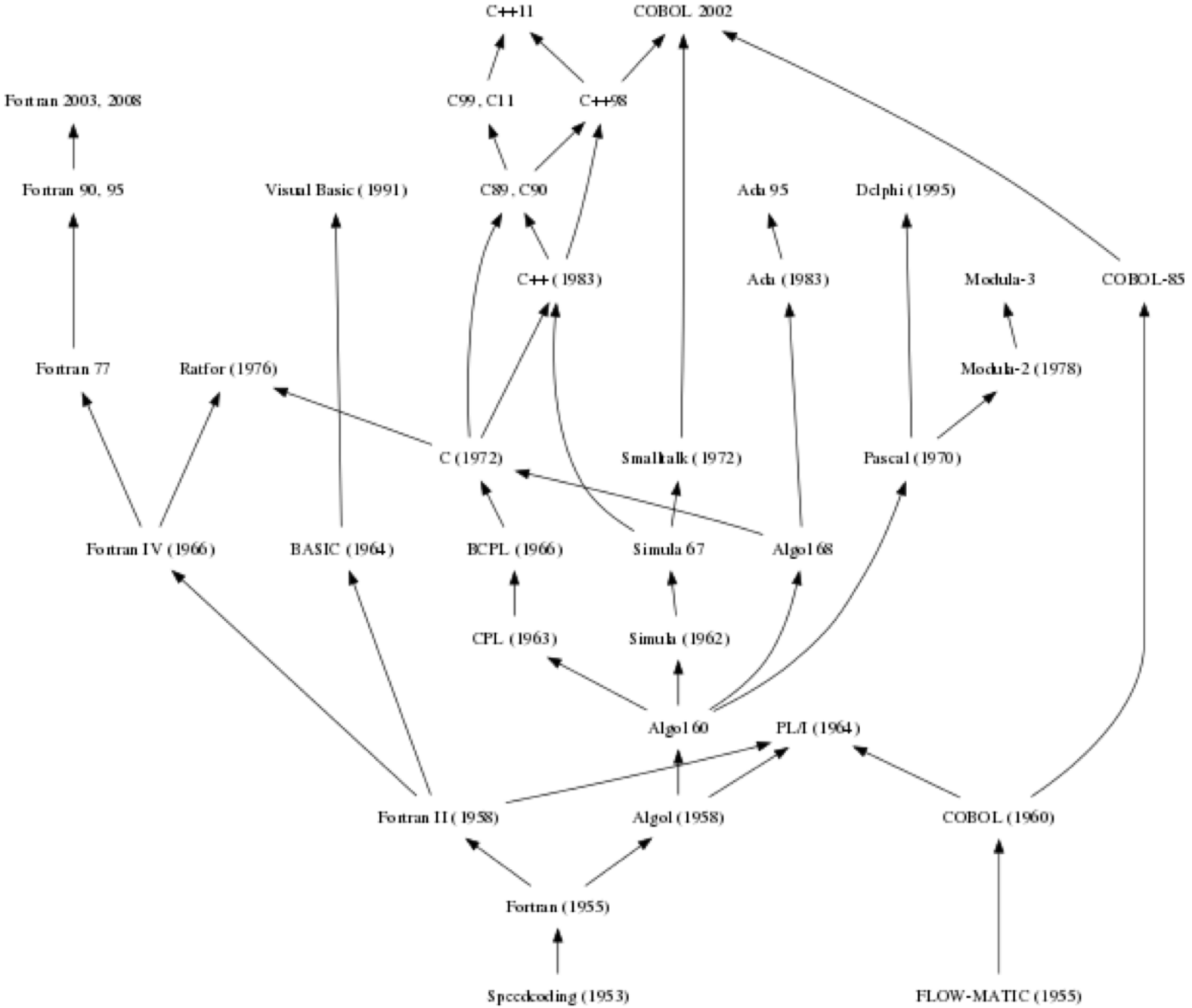
– Larry Tesler

Paradigm Redshift 🌈

The "What If" Tree

Paradigm Redshift

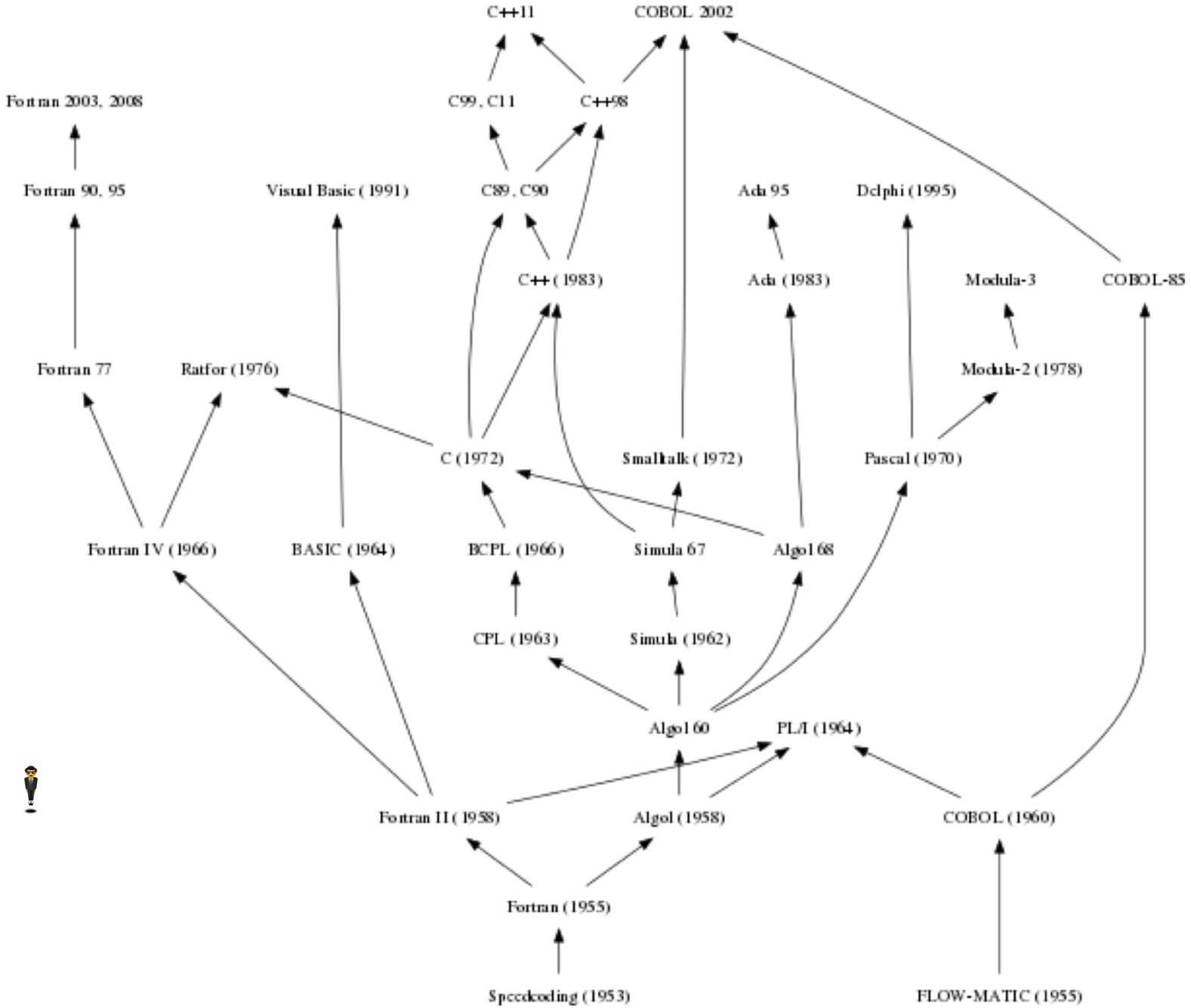
The "What If" Tree



Paradigm Redshift 🌈

The "What If" Tree

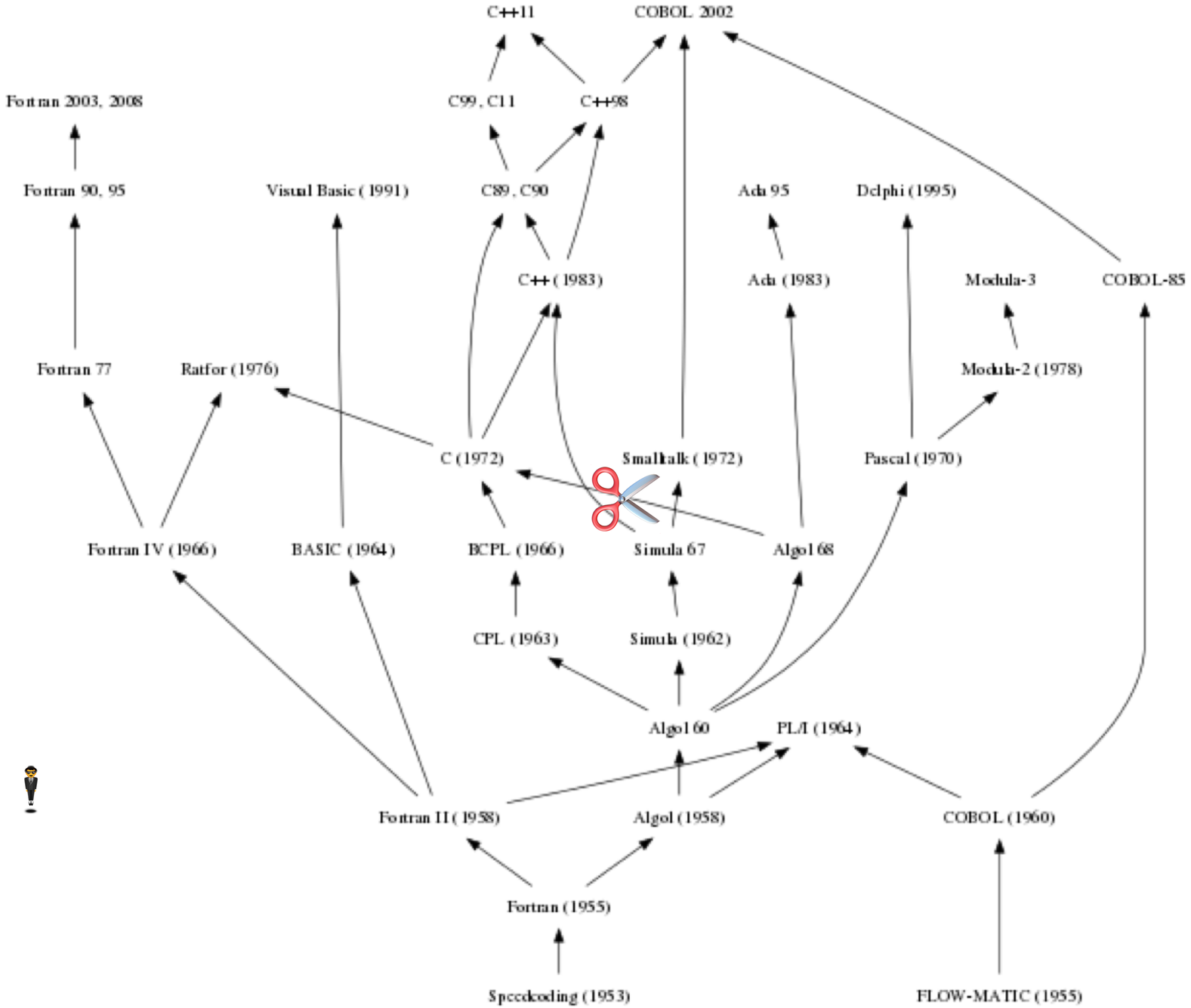
1. Array-based 📈
2. Applicative Model 🤹
3. SML Module System 🔌
4. Natural & Biz Language 🕴



Paradigm Redshift 🌈

The "What If" Tree

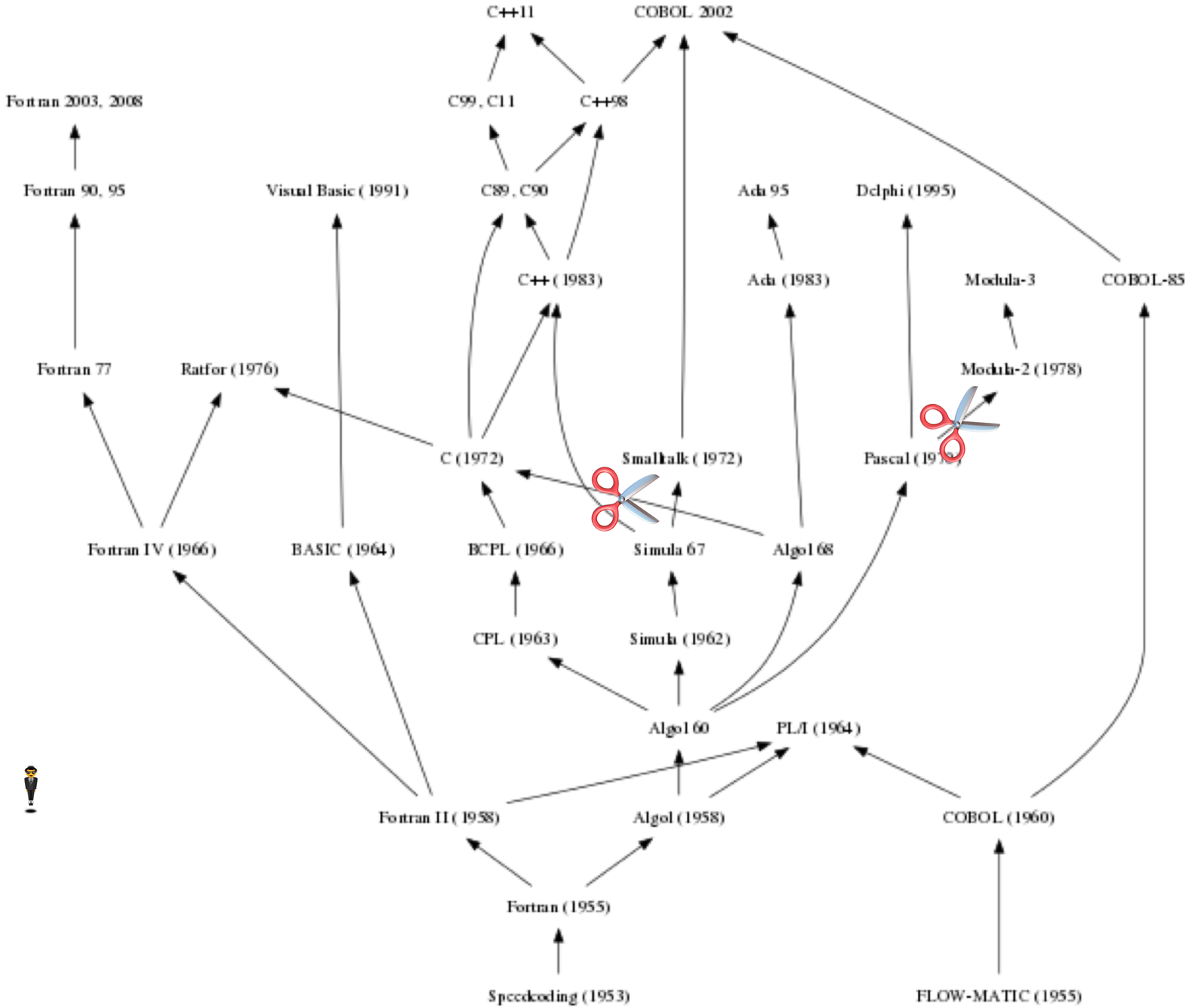
1. Array-based 📈
2. Applicative Model 🤹
3. SML Module System 🔌
4. Natural & Biz Language 🕴️



Paradigm Redshift 🌈

The "What If" Tree

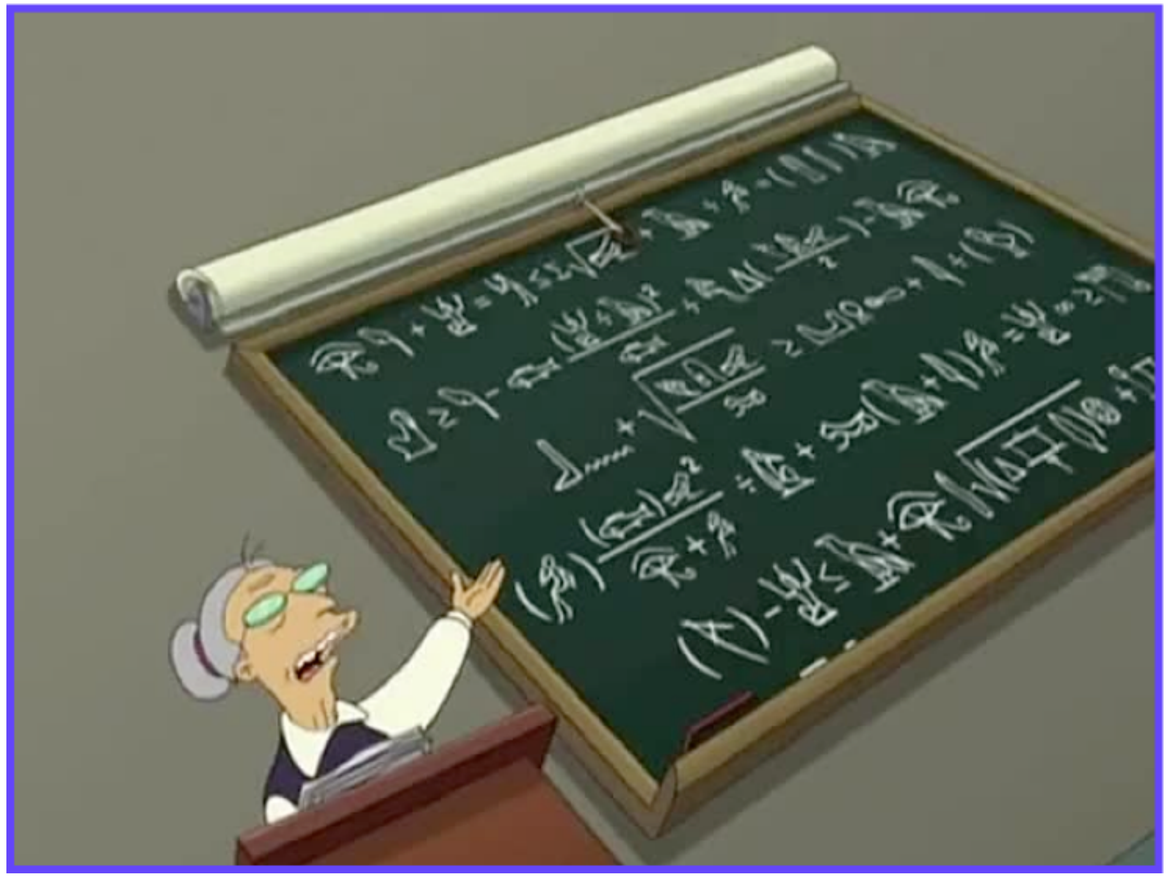
1. Array-based 📈
2. Applicative Model 🤹
3. SML Module System 🔌
4. Natural & Biz Language 🕴️



Paradigm Redshift

Let's start with something alien

🛸👽



Paradigm Redshift 🌈

Something Alien

Paradigm Redshift 🌈

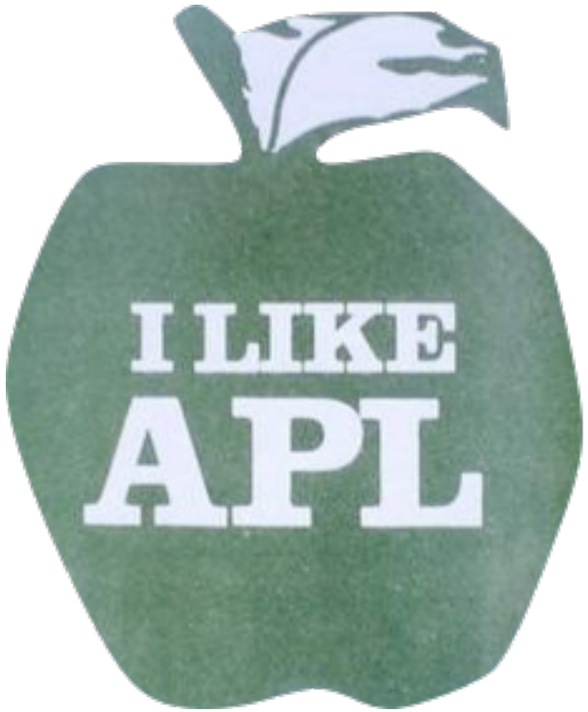
Something Alien

```
{↑1 ωv.∧3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.ϕ⊂ω}
```

Paradigm Redshift 🌈

Something Alien

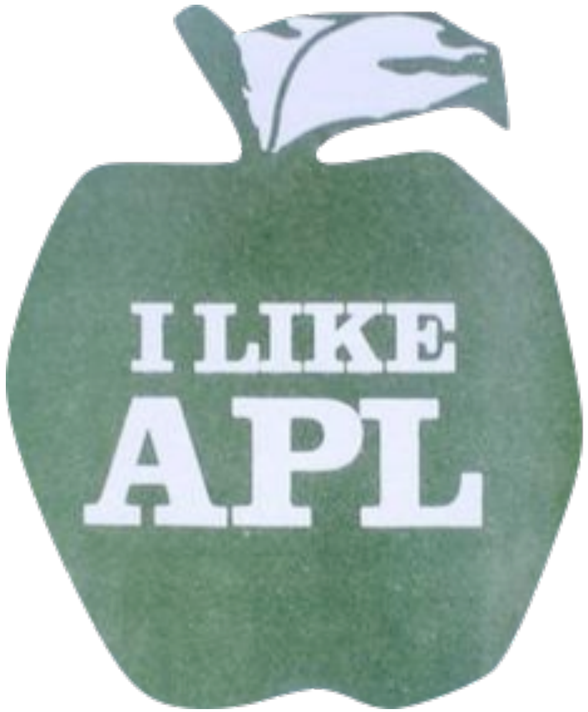
```
{↑1 ωv.∧3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂ω}
```



Paradigm Redshift 🌈

Something Alien

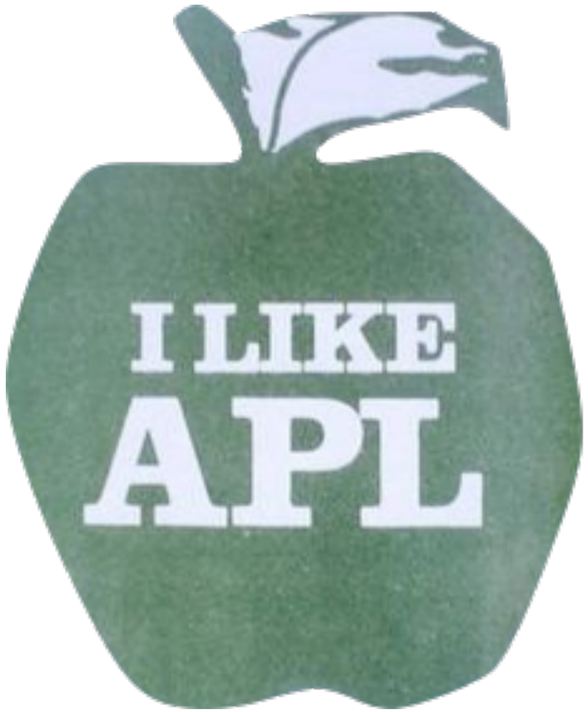
```
LIFE←{↑1 ωv.∧3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂ω}
```



Paradigm Redshift 🌈

Something Alien

```
LIFE←{↑1 ωv.∧3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂ω}
```

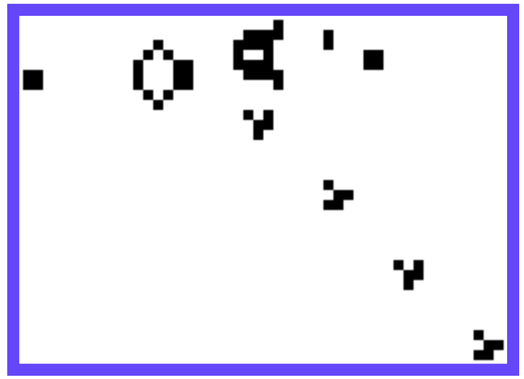


Attribution: wikipedia user LucasVB

Paradigm Redshift 🌈

What can we learn from APL?

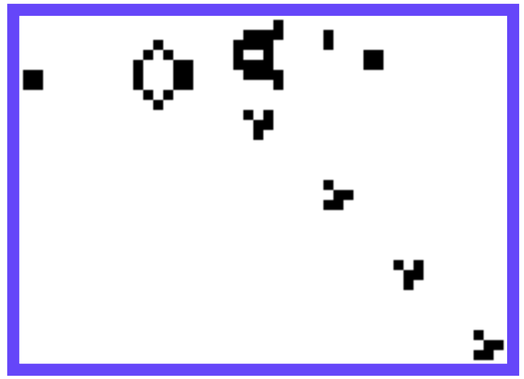
Cellular Automata & Actors as Organisms 🌸



Paradigm Redshift 🌈

What can we learn from APL?

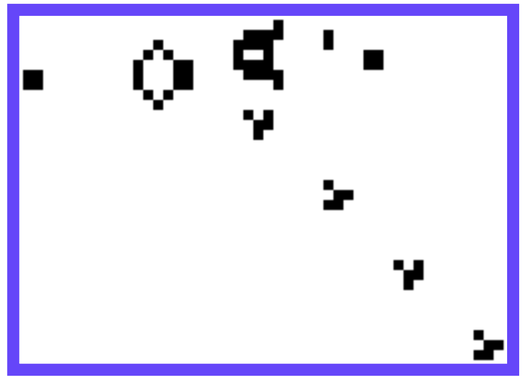
Cellular Automata & Actors as Organisms 🌸



Paradigm Redshift 🌈

What can we learn from APL?

Cellular Automata & Actors as Organisms 🌸

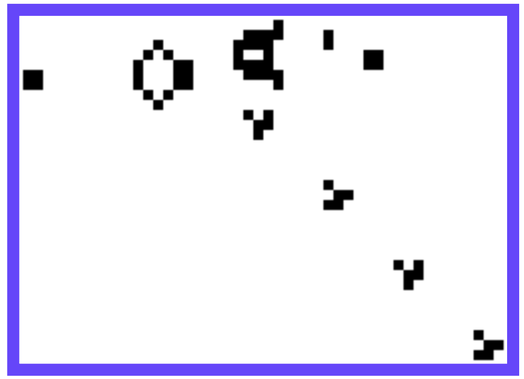


- Each step is very simple

Paradigm Redshift 🌈

What can we learn from APL?

Cellular Automata & Actors as Organisms 🌸

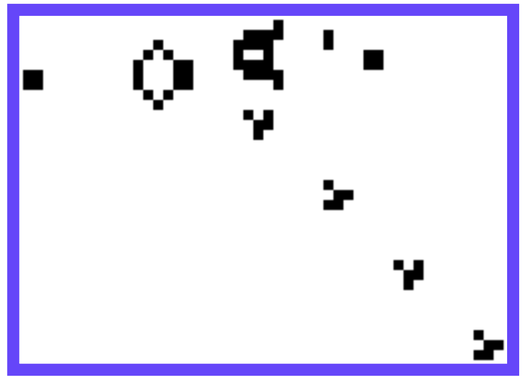


- Each step is very simple
- Reasoning about dynamic organisms is hard!

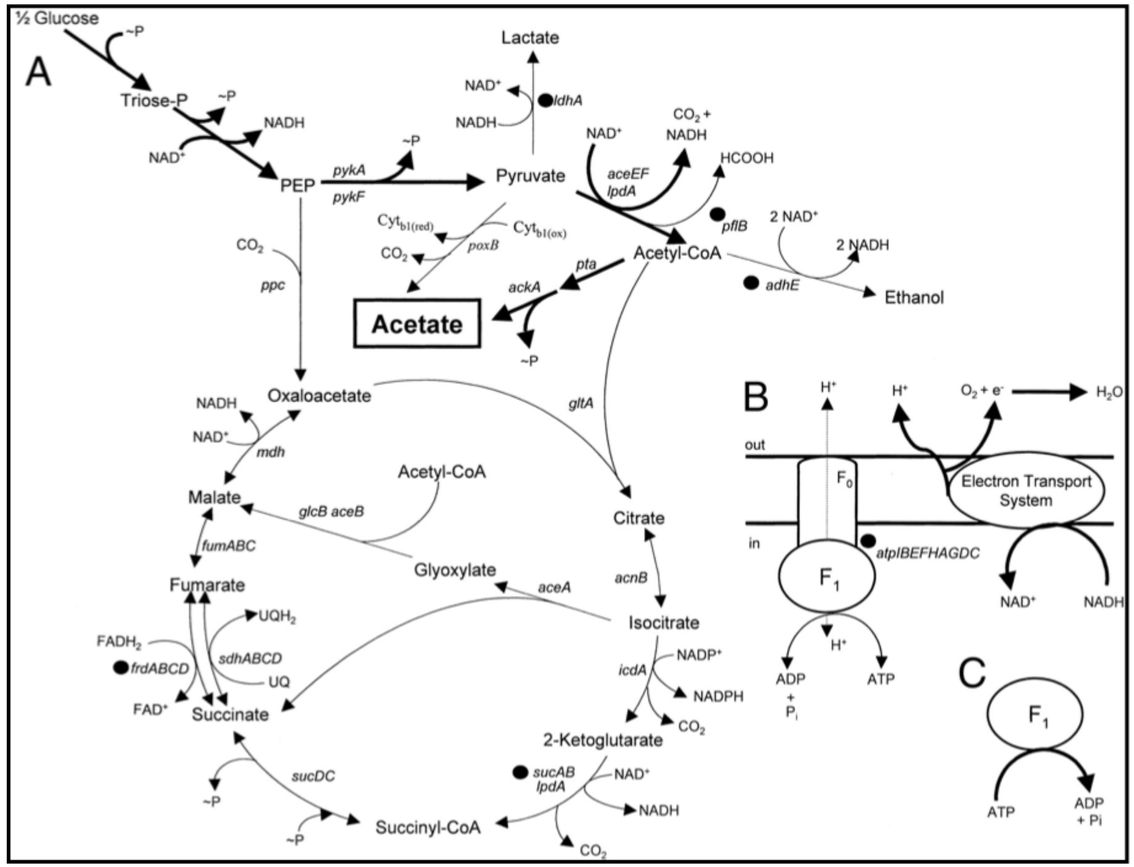
Paradigm Redshift 🌈

What can we learn from APL?

Cellular Automata & Actors as Organisms 🌸



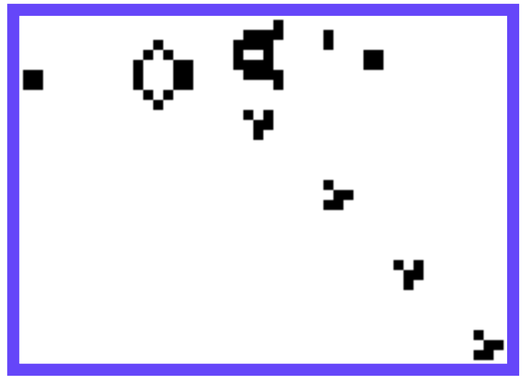
- Each step is very simple
- Reasoning about dynamic organisms is hard!



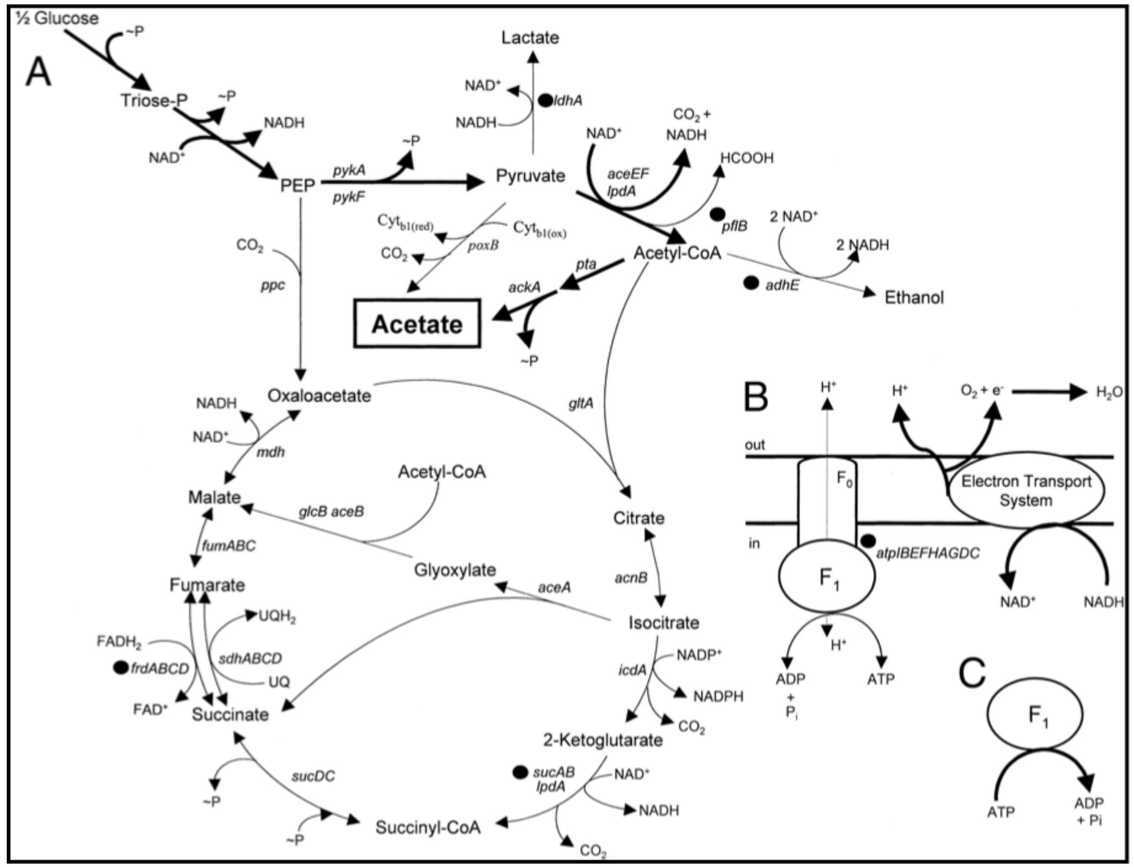
Paradigm Redshift 🌈

What can we learn from APL?

Cellular Automata & Actors as Organisms 🌸



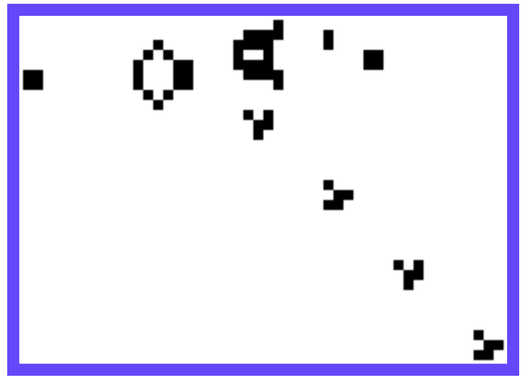
- Each step is very simple
- Reasoning about dynamic organisms is hard!
- Emergent behaviour 😱



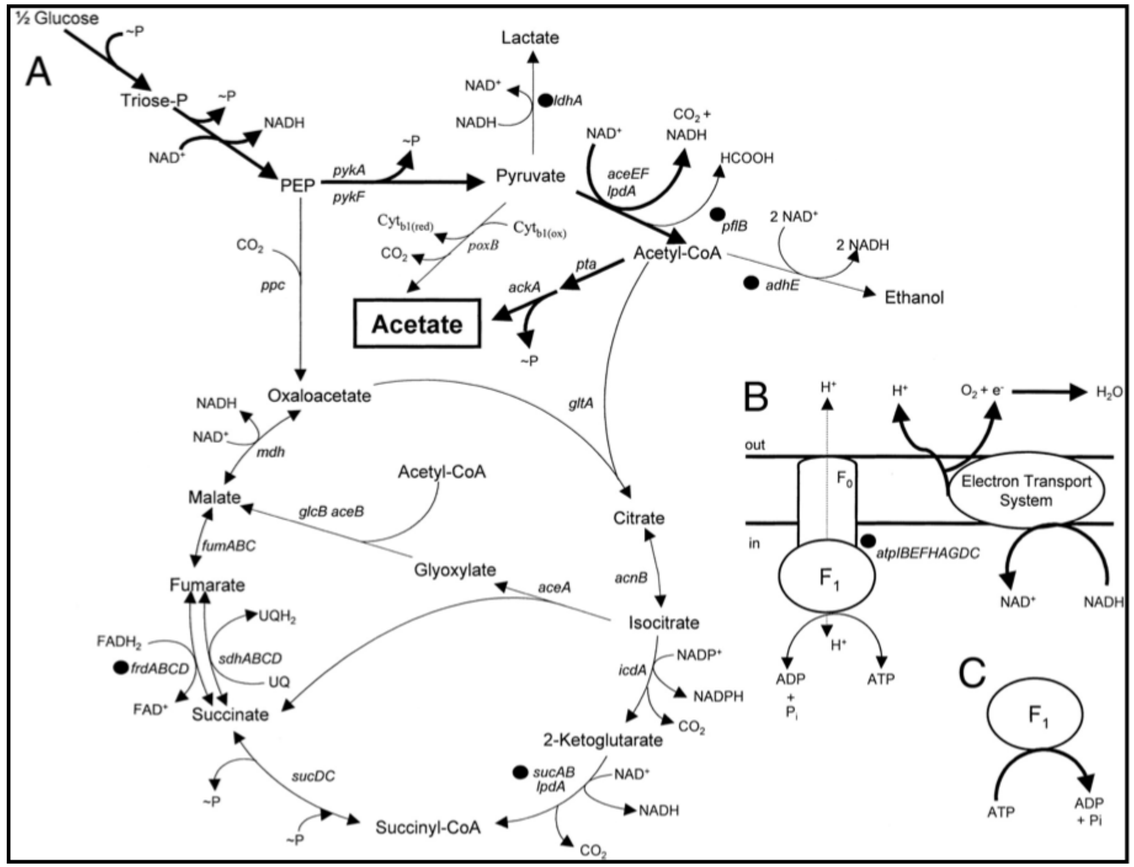
Paradigm Redshift 🌈

What can we learn from APL?

Cellular Automata & Actors as Organisms 🌸



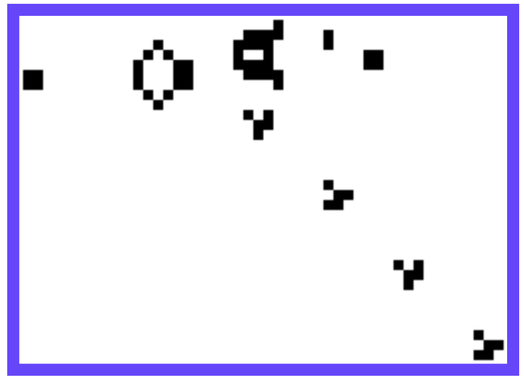
- Each step is very simple
- Reasoning about dynamic organisms is hard!
- Emergent behaviour 😱
- VM in your brain to reason at a higher level



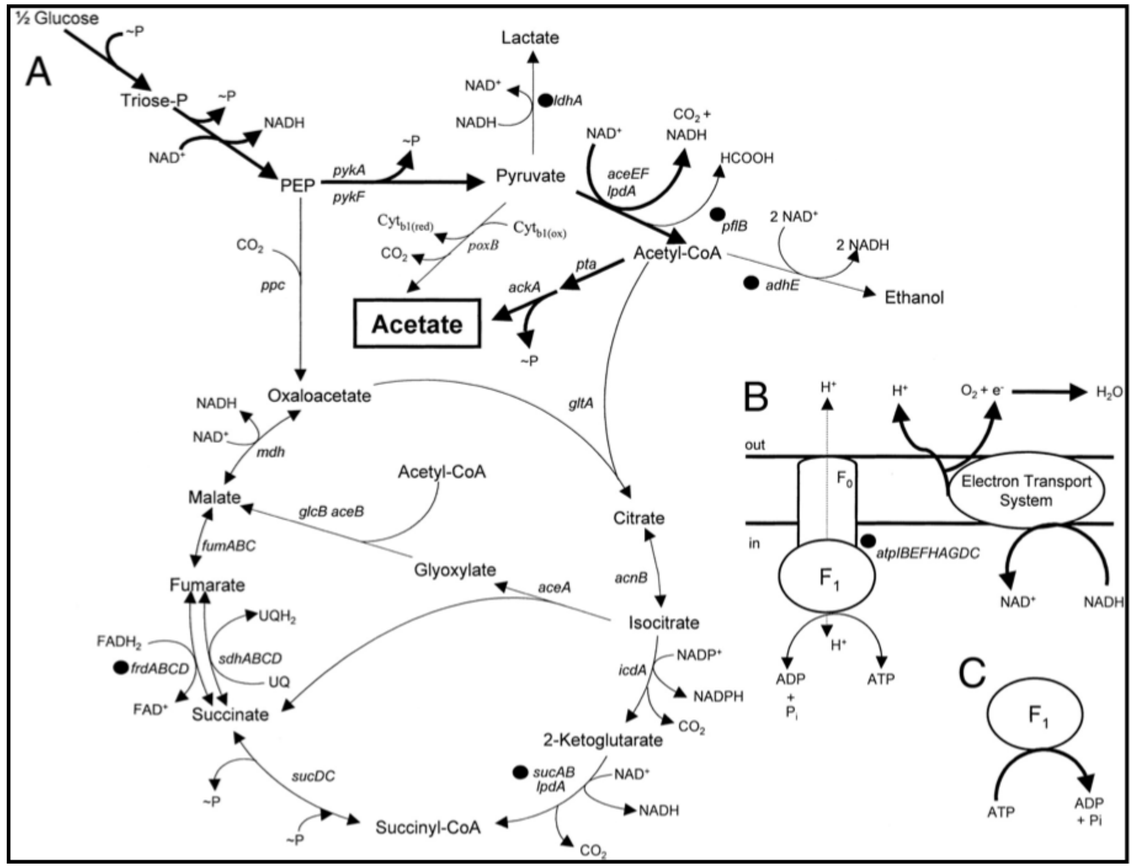
Paradigm Redshift 🌈

What can we learn from APL?

Cellular Automata & Actors as Organisms 🌸



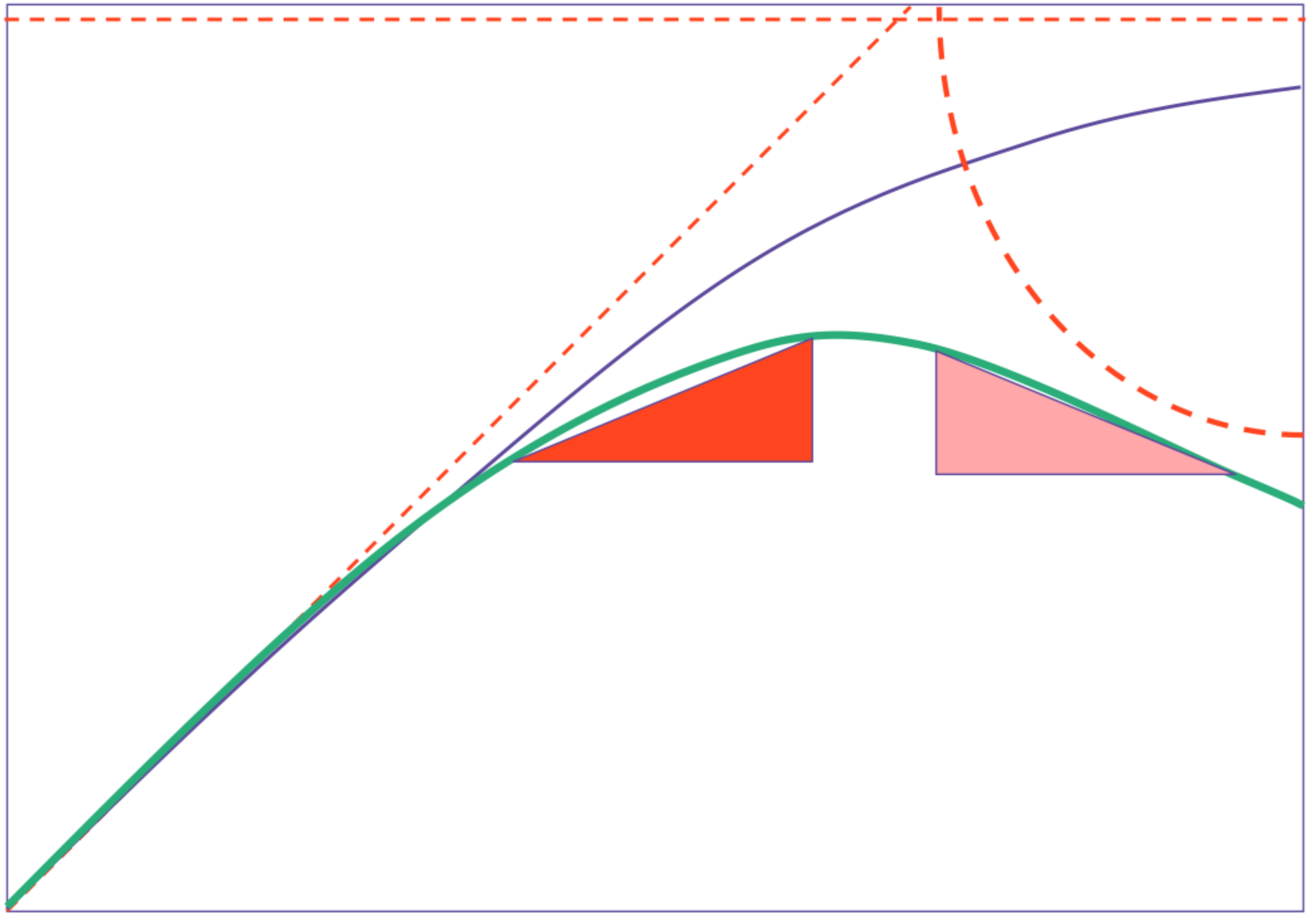
- Each step is very simple
- Reasoning about dynamic organisms is hard!
- Emergent behaviour 😱
- VM in your brain to reason at a higher level
- We can abstract away some of this
  - Broadway
  - Arrows



Paradigm Redshift 🌈

What can we learn from APL?

Universal Scaling Law

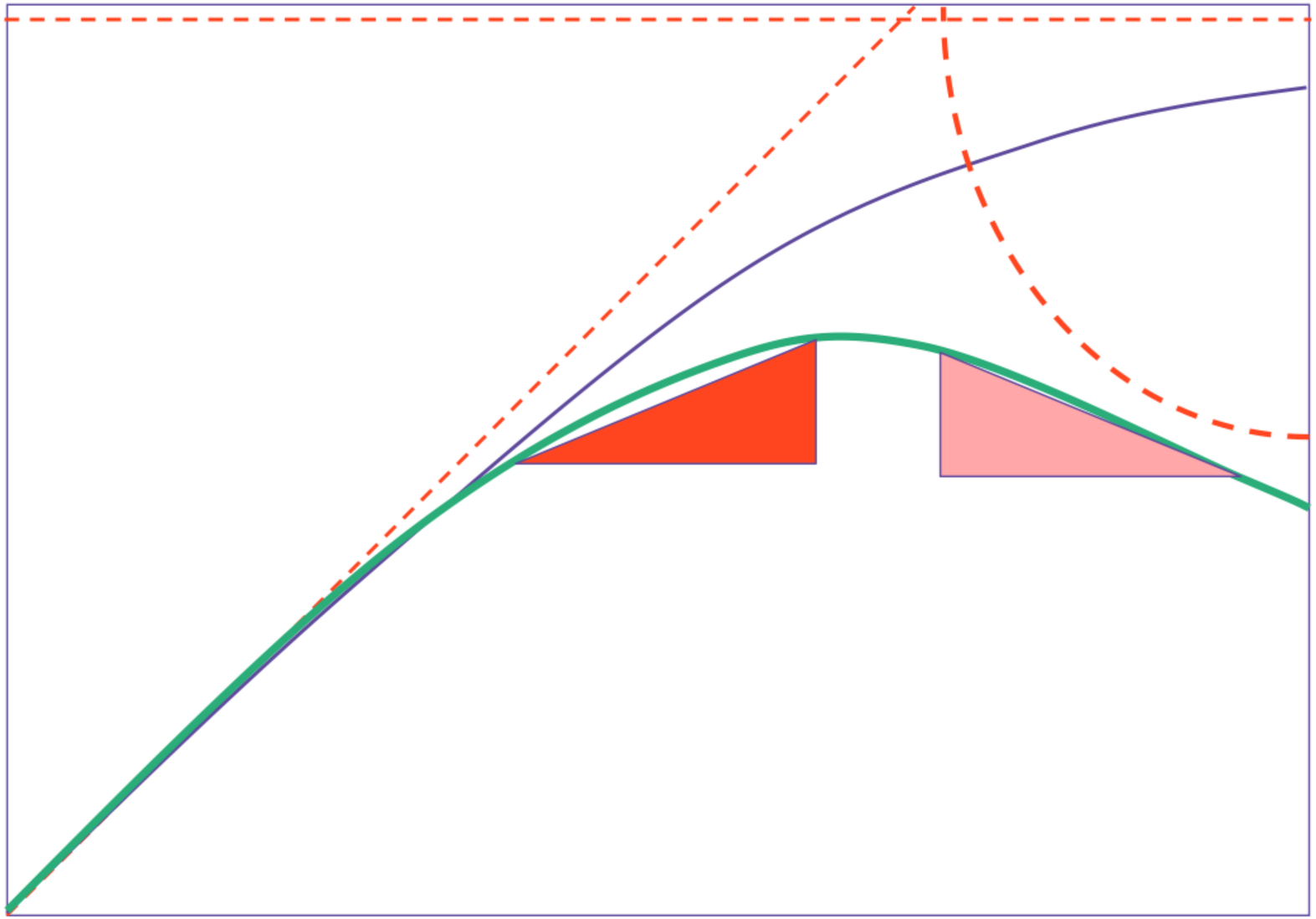


http://www.perfdynamics.com/Manifesto/USLscalability.html

Paradigm Redshift 🌈

What can we learn from APL?

Universal Scaling Law

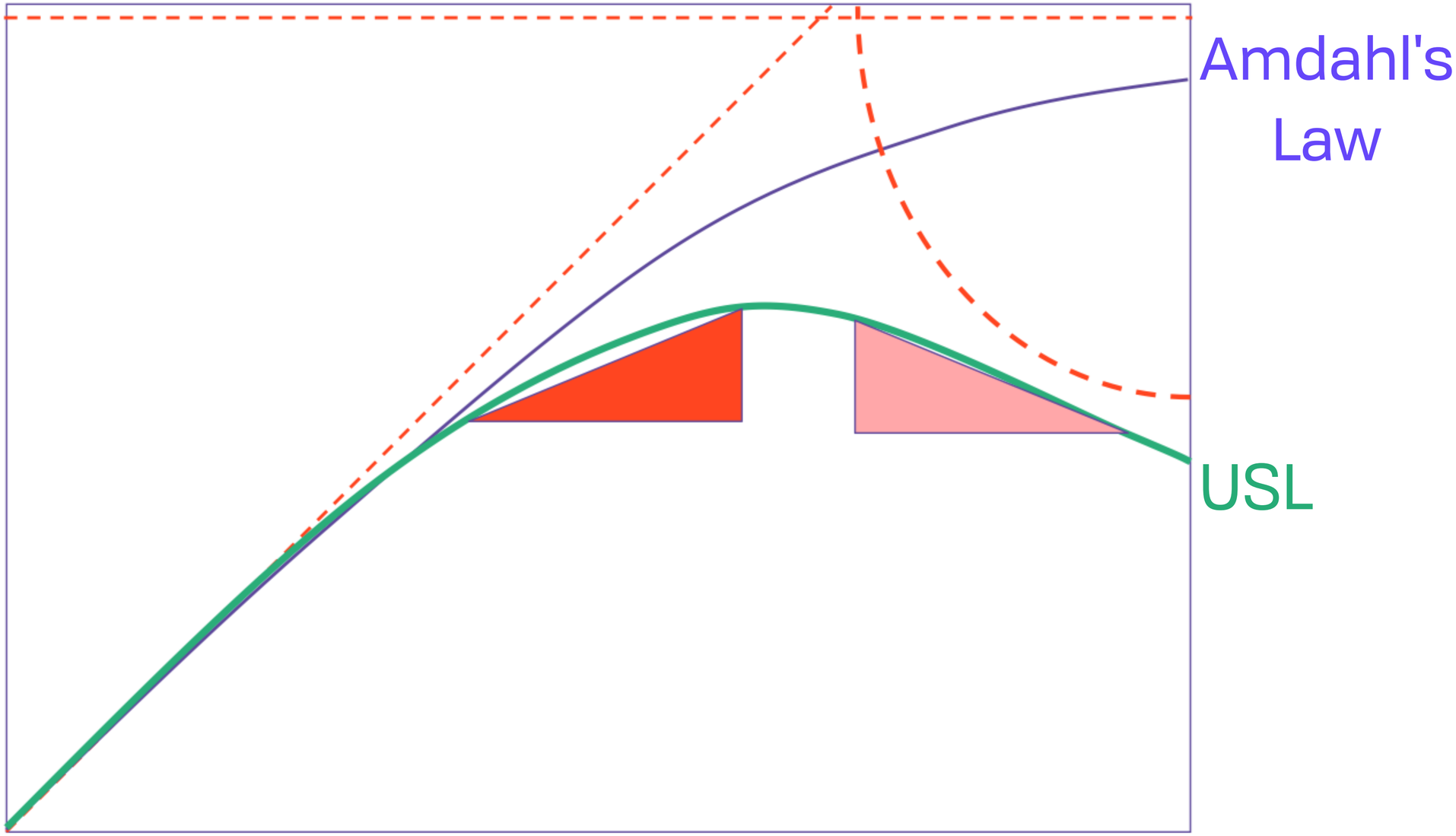


http://www.perfdynamics.com/Manifesto/USLscalability.html

Paradigm Redshift 🌈

What can we learn from APL?

Universal Scaling Law

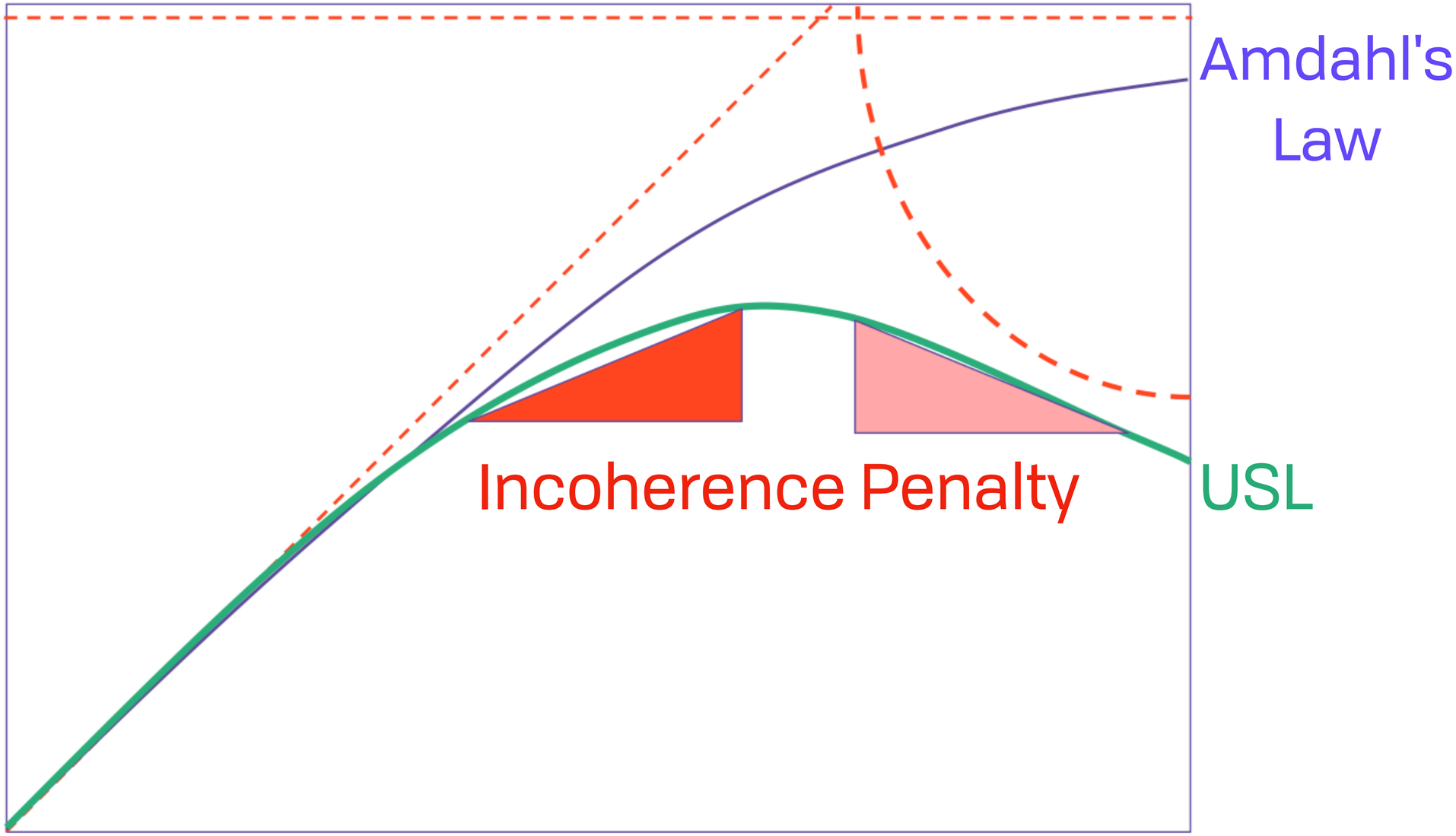


http://www.perfdynamics.com/Manifesto/USLscalability.html

Paradigm Redshift 🌈

What can we learn from APL?

Universal Scaling Law



http://www.perfdynamics.com/Manifesto/USLscalability.html

Paradigm Redshift 🌈

What can we learn from APL?

`doall` / Automatic Parallelism / Cyclic Multithreading 🔁

```
import Witchcraft.Functor

0..10_000
|> Enum.to_list()
|> async_map(fn x ->
  Process.sleep(500)
  x * 10
end)
#=> [0, 10, ...] ~1s
```

Paradigm Redshift 🌈

What can we learn from APL?

`doall` / Automatic Parallelism / Cyclic Multithreading 🔁

```
import Witchcraft.Functor

0..10_000
|> Enum.to_list()
|> async_map(fn x ->
  Process.sleep(500)
  x * 10
end)
#=> [0, 10, ...] ~1s
```

1. Shared-nothing architecture
2. Good for embarrassingly parallel problems
3. Macro could do a LOT more with this at compile-time
4. Impurity and granular control mean that we don't get this by default (with good reason)

Paradigm Redshift 🌈

OTP → TOP

Table Oriented Programming

Paradigm Redshift 🌈

Table Oriented Programming

Naive Tables

Paradigm Redshift 🌈

Table Oriented Programming

Naive Tables

Name	Handle	City
Brooklyn	expede	Vancouver
Quinn	quinnwilton	Mountain View
Steven	icidasset	Ghent

Paradigm Redshift 🌈

Table Oriented Programming

Naive Tables

```
@type t :: User{
  name   :: String.t(),
  handle :: String.t(),
  city   :: String.t()
}
```

Name	Handle	City
Brooklyn	expede	Vancouver
Quinn	quinnwilton	Mountain View
Steven	icidasset	Ghent

Paradigm Redshift 🌈

Table Oriented Programming

Naive Tables

```
@type t :: User{
  name   :: String.t(),
  handle :: String.t(),
  city   :: String.t()
}
```

Name	Handle	City
Brooklyn	expede	Vancouver
Quinn	quinnwilton	Mountain View
Steven	icidasset	Ghent

```
[
  %User{name: "Brooklyn", handle: "expede",     city: "Vancouver"},
  %User{name: "Quinn",    handle: "quinnwilton", city: "Mountain View"},
  %User{name: "Steven",   handle: "icidasset",   city: "Ghent"}
]
```

Paradigm Redshift 🌈

Table Oriented Programming

Perfectly Parallel Control Tables

Paradigm Redshift 🌈

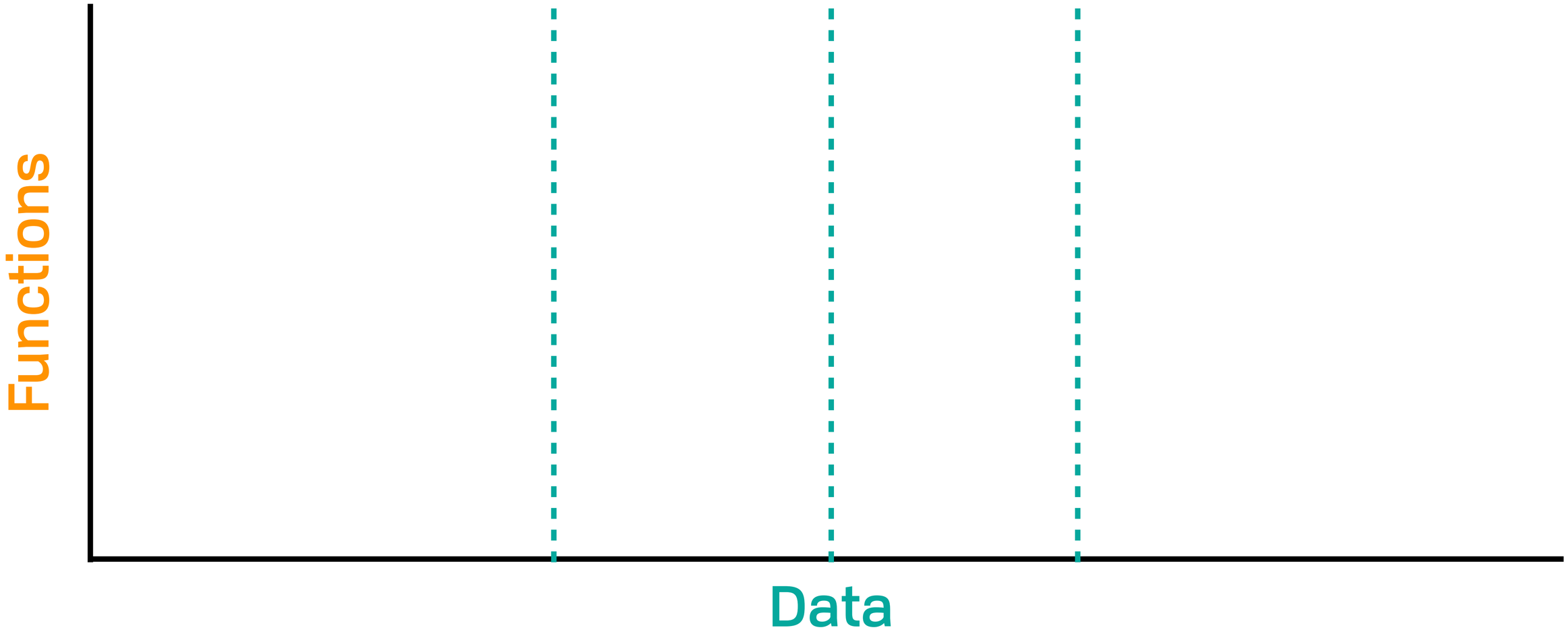
Table Oriented Programming

Perfectly Parallel Control Tables

Paradigm Redshift 🌈

Table Oriented Programming

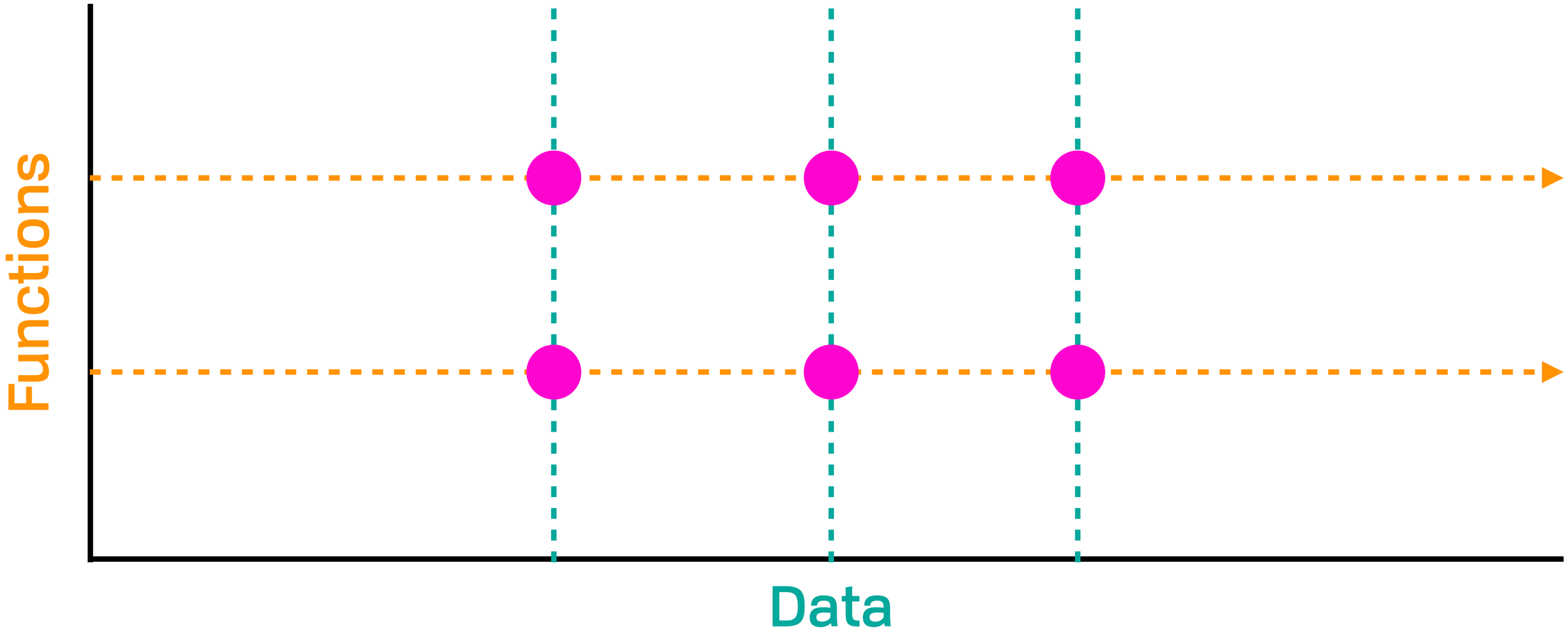
Perfectly Parallel Control Tables



Paradigm Redshift 🌈

Table Oriented Programming

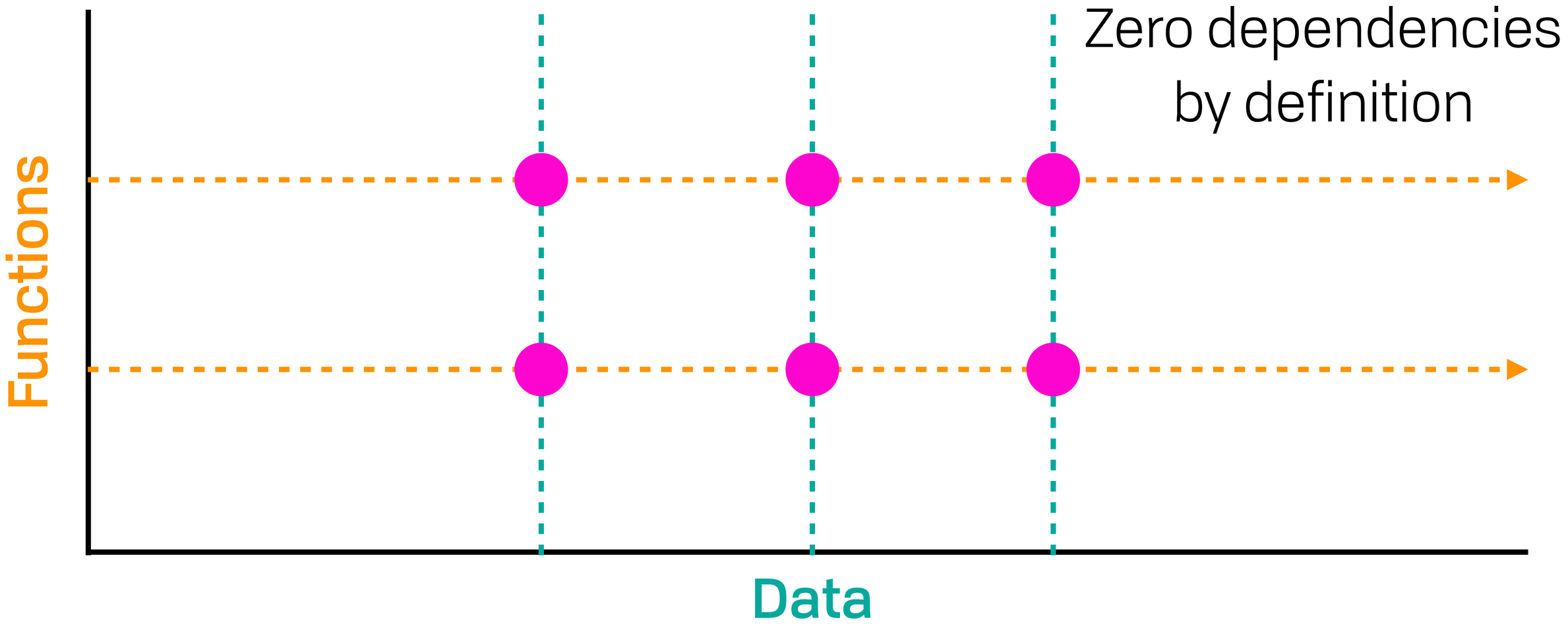
Perfectly Parallel Control Tables



Paradigm Redshift 🌈

Table Oriented Programming

Perfectly Parallel Control Tables



Paradigm Redshift 🌈

Table Oriented Programming

The Nth-Dimension 🚀

```
[&+/2, &*/2]
|> provide([1, 2, 3])
|> provide([4, 5, 6])
```

```
[
  5, 6, 7,
  6, 7, 8,
  7, 8, 9,
  4, 5, 6,
  8, 10, 12,
  12, 15, 18
]
```

Paradigm Redshift 🌈

Table Oriented Programming

Composition & Modularity

Complexity & Modularity 

Complexity & Modularity 🧰

–Alan J. Perlis, Epigrams on Programming (1982)

Epigram 6

Symmetry is a complexity-reducing concept;
seek it everywhere

–Alan J. Perlis, Epigrams on Programming (1982)

Epigram 6

Symmetry is a complexity-reducing concept;
seek it everywhere

Epigram 105

You can't communicate complexity, only awareness of it

–Alan J. Perlis, Epigrams on Programming (1982)

What do you mean by "composition"?

Complexity & Modularity 🧱

What do you mean by "composition"?

Complexity & Modularity 🧱

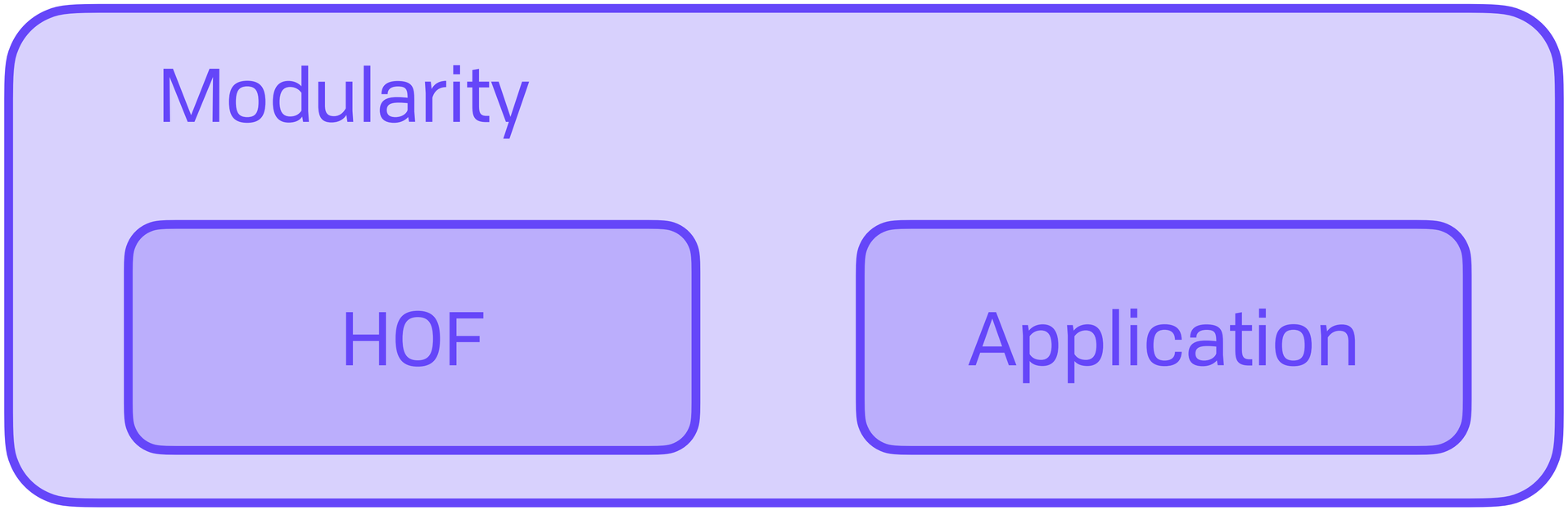
What do you mean by "composition"?

HOF

Application

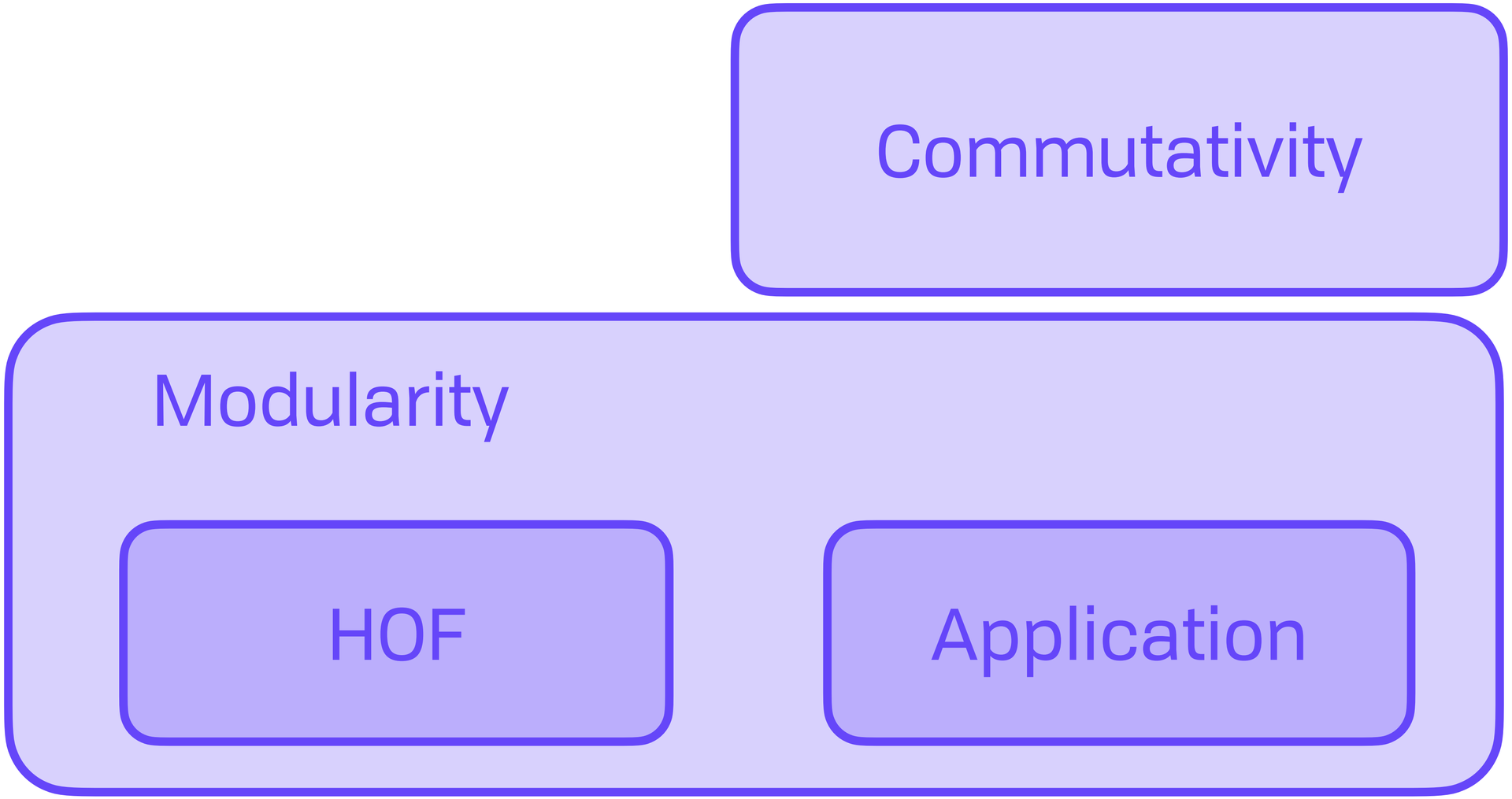
Complexity & Modularity 🧱

What do you mean by "composition"?



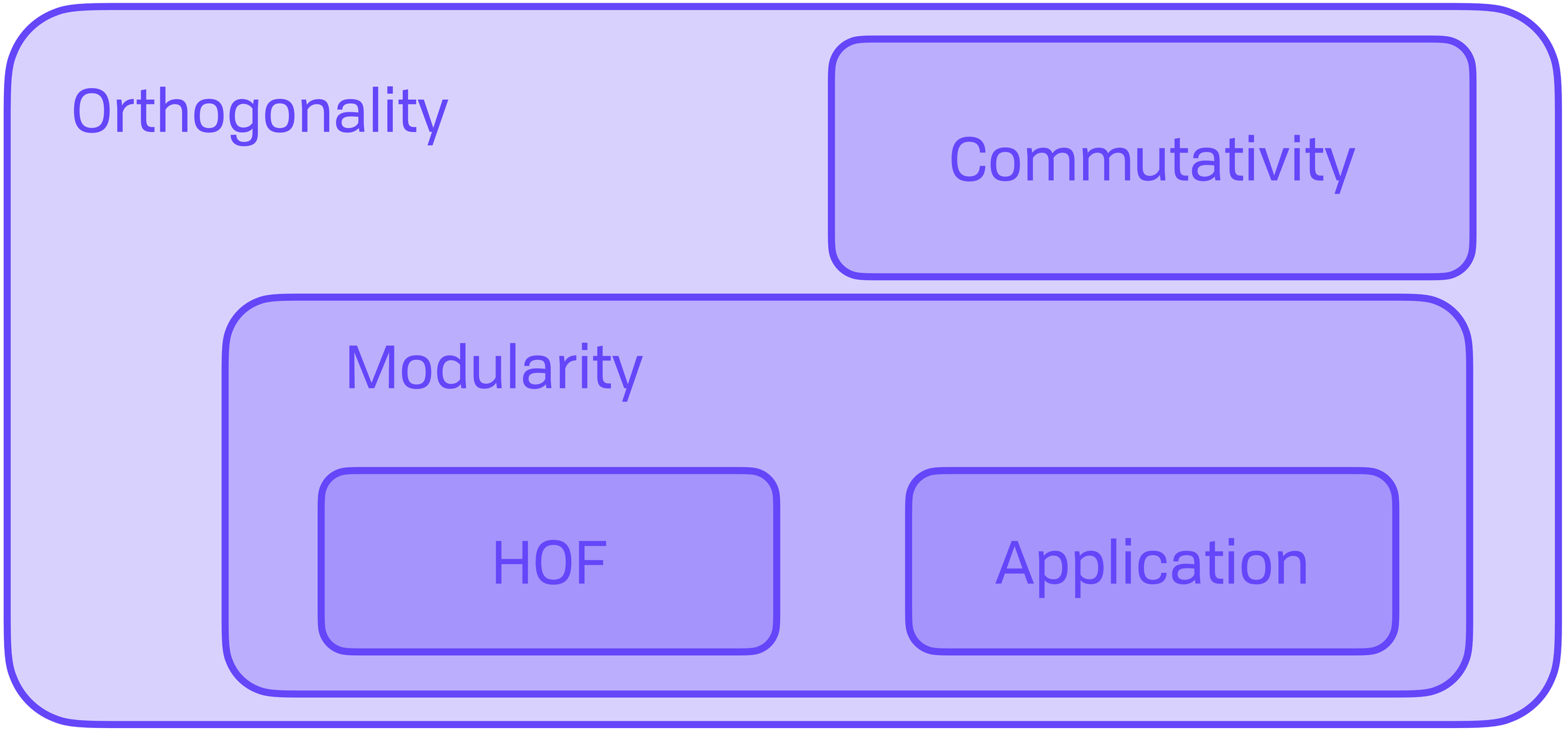
Complexity & Modularity 🧱

What do you mean by "composition"?



Complexity & Modularity 🧱

What do you mean by "composition"?



Complexity & Modularity 🧱

What do you mean by "composition"?

Focus 🔬

What do you mean by "composition"?

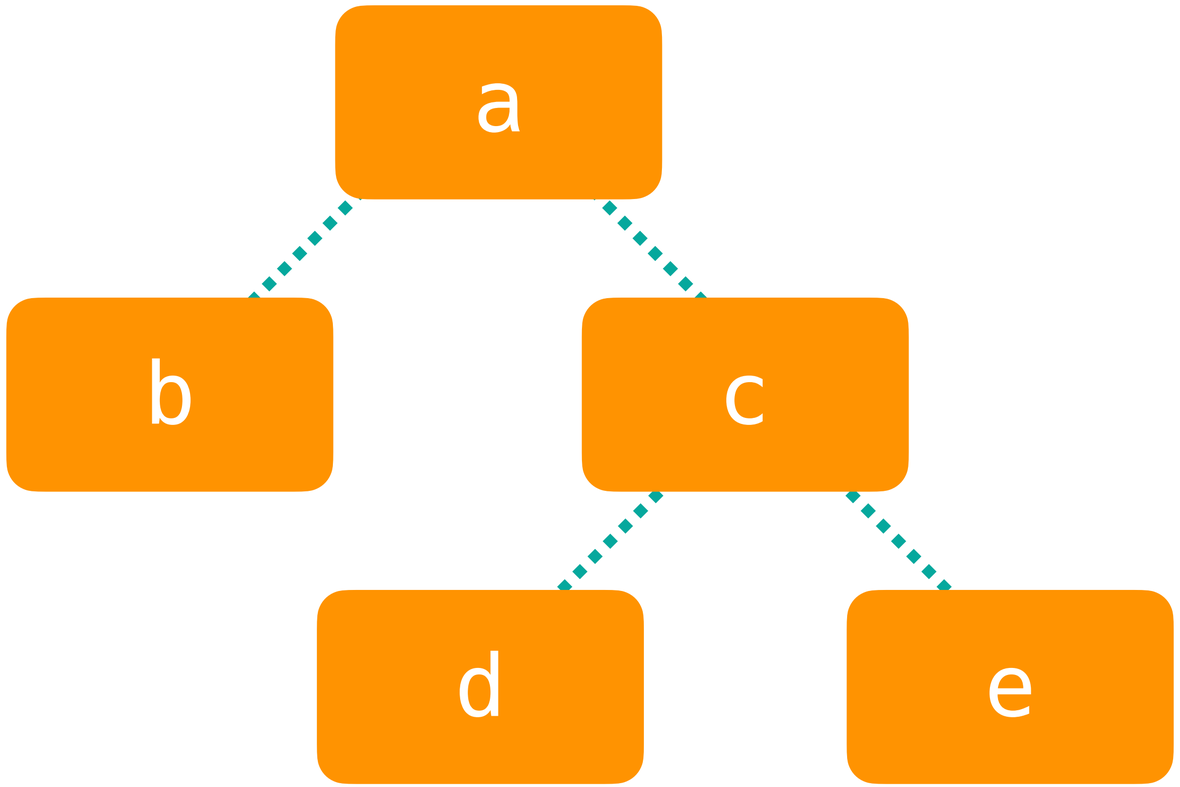
Focus 🔬

Composition in the **data** dimension

What do you mean by "composition"?

Focus 🔬

Composition in the **data** dimension



Complexity & Modularity 🧱

What do you mean by "composition"?

Focus 🔬

What do you mean by "composition"?

Focus 🔬

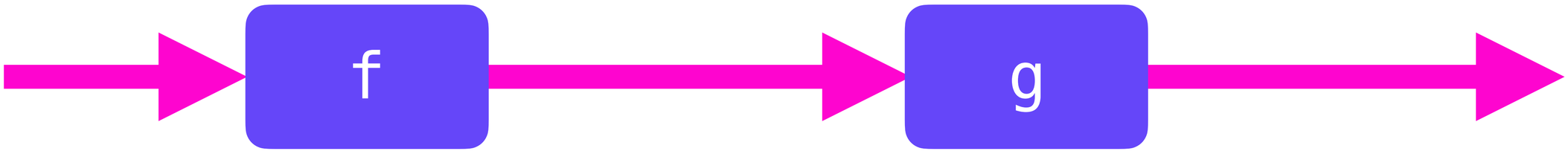
Composition in the **function** dimension

Complexity & Modularity 🧱

What do you mean by "composition"?

Focus 🔬

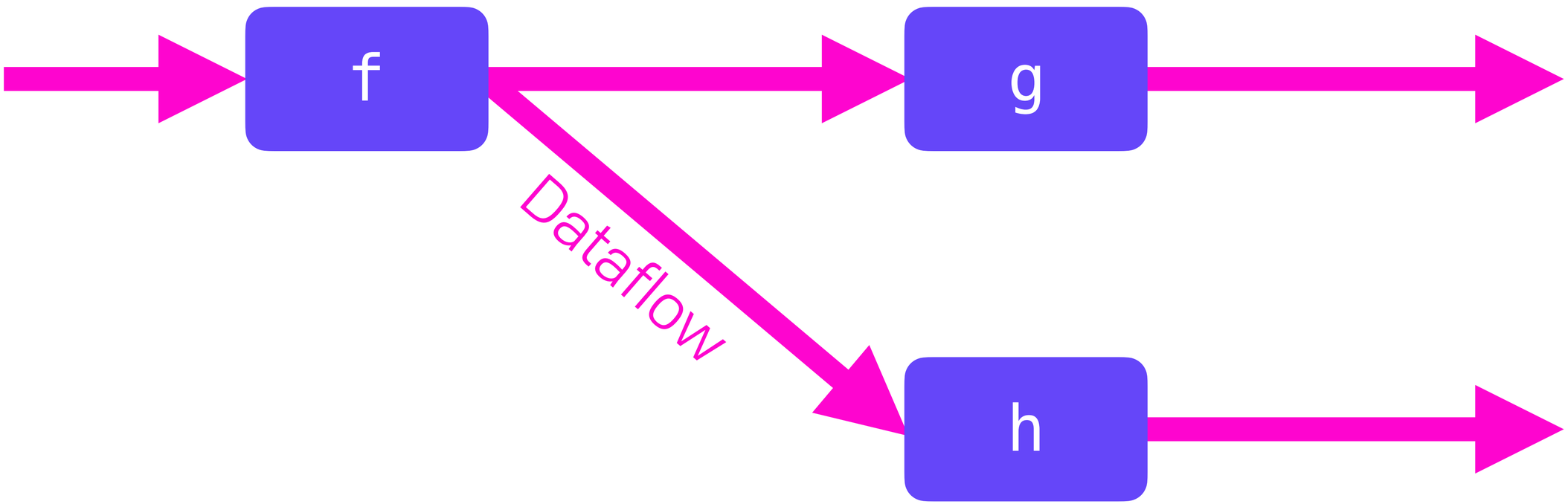
Composition in the **function** dimension



What do you mean by "composition"?

Focus 🔬

Composition in the **function** dimension



Complexity & Modularity 🧱

What do you mean by "composition"?

Focus 🔬

What do you mean by "composition"?

Focus 🔬

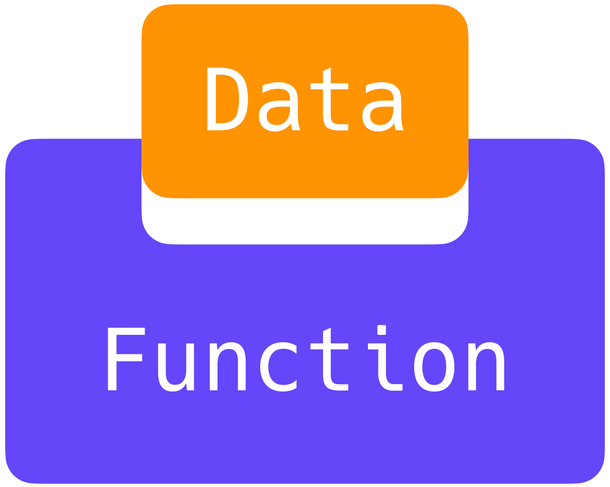
Composition in the **capabilities**
(protocols and HOFs)

Complexity & Modularity 🧱

What do you mean by "composition"?

Focus 🔬

Composition in the **capabilities**
(protocols and HOFs)



Complexity & Modularity 🧱

What do you mean by "composition"?

Execution Symmetry

Complexity & Modularity 📦

What do you mean by "composition"?

Execution Symmetry

A program can be ***developed on a sequential platform, even if it is meant to run on a parallel platform***, because the behaviour is not affected by whether we execute it using a sequential or parallel dynamics

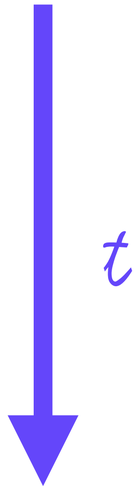
– Robert Harper, Practical Foundations for Programming Languages (2012)

What do you mean by "composition"?

Execution Symmetry

A program can be ***developed on a sequential platform, even if it is meant to run on a parallel platform***, because the behaviour is not affected by whether we execute it using a sequential or parallel dynamics

– Robert Harper, Practical Foundations for Programming Languages (2012)



What do you mean by "composition"?

Execution Symmetry

A program can be ***developed on a sequential platform, even if it is meant to run on a parallel platform***, because the behaviour is not affected by whether we execute it using a sequential or parallel dynamics

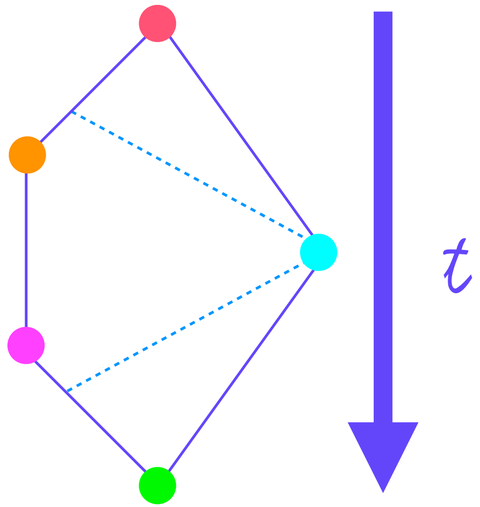
– Robert Harper, Practical Foundations for Programming Languages (2012)

What do you mean by "composition"?

Execution Symmetry

A program can be ***developed on a sequential platform, even if it is meant to run on a parallel platform***, because the behaviour is not affected by whether we execute it using a sequential or parallel dynamics

– Robert Harper, Practical Foundations for Programming Languages (2012)

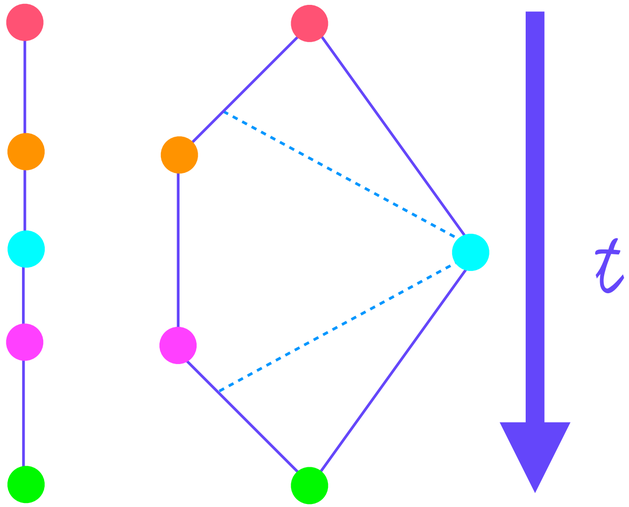


What do you mean by "composition"?

Execution Symmetry

A program can be ***developed on a sequential platform, even if it is meant to run on a parallel platform***, because the behaviour is not affected by whether we execute it using a sequential or parallel dynamics

– Robert Harper, Practical Foundations for Programming Languages (2012)

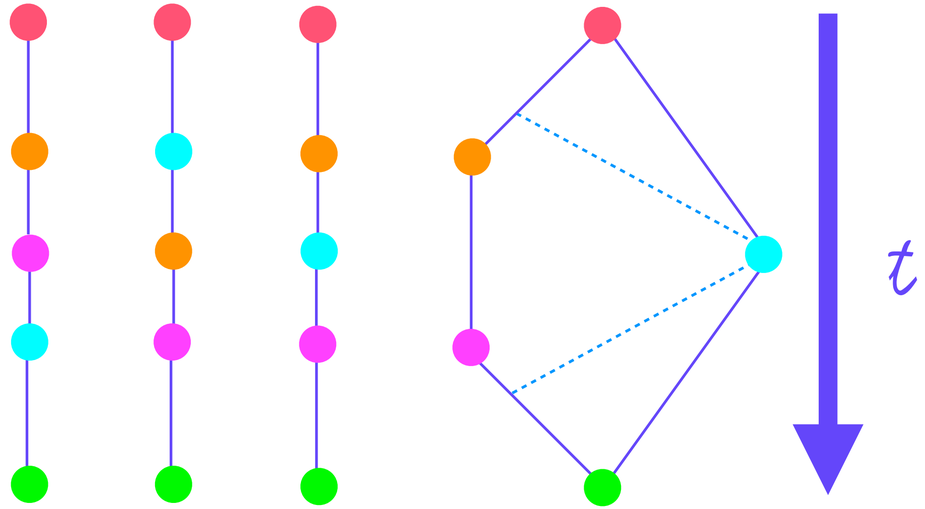


What do you mean by "composition"?

Execution Symmetry

A program can be ***developed on a sequential platform, even if it is meant to run on a parallel platform***, because the behaviour is not affected by whether we execute it using a sequential or parallel dynamics

– Robert Harper, Practical Foundations for Programming Languages (2012)



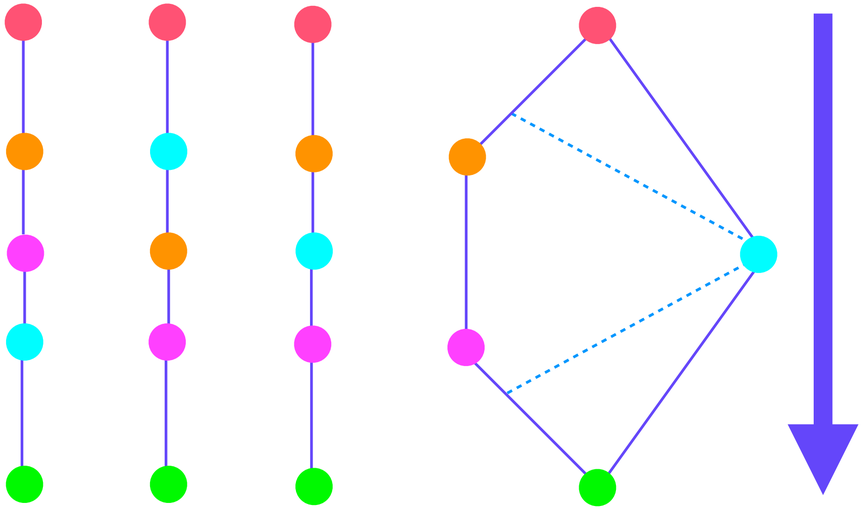
Complexity & Modularity 🧱

What do you mean by "composition"?

Execution Symmetry

A program can be ***developed on a sequential platform, even if it is meant to run on a parallel platform***, because the behaviour is not affected by whether we execute it using a sequential or parallel dynamics

– Robert Harper, Practical Foundations for Programming Languages (2012)



It commutes! 👍

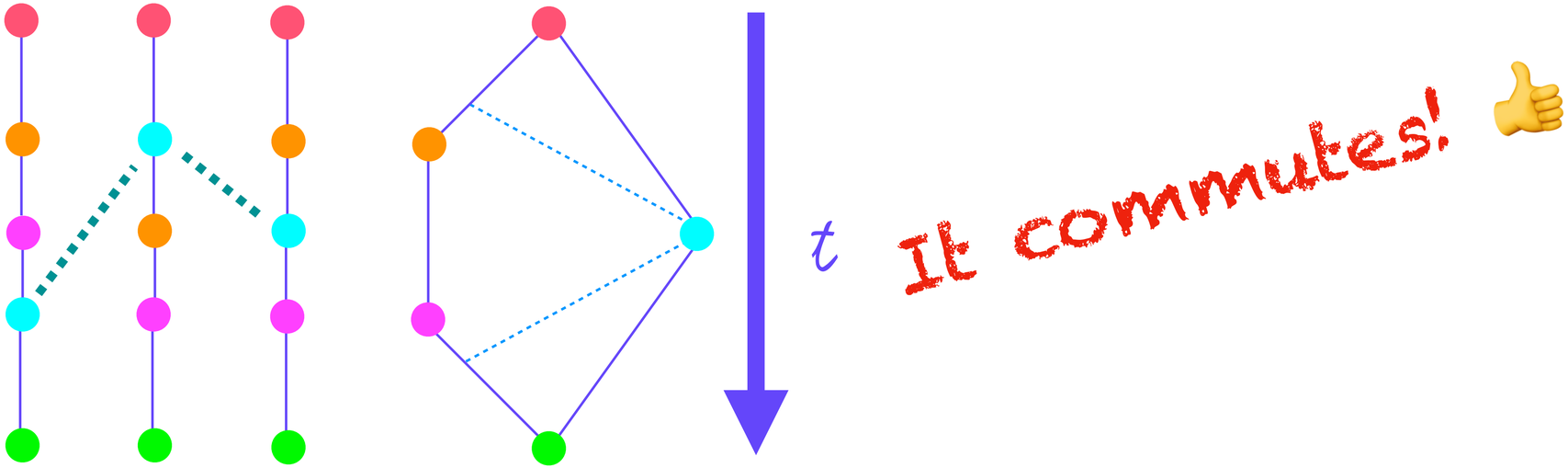
Complexity & Modularity 🧱

What do you mean by "composition"?

Execution Symmetry

A program can be ***developed on a sequential platform, even if it is meant to run on a parallel platform***, because the behaviour is not affected by whether we execute it using a sequential or parallel dynamics

– Robert Harper, Practical Foundations for Programming Languages (2012)



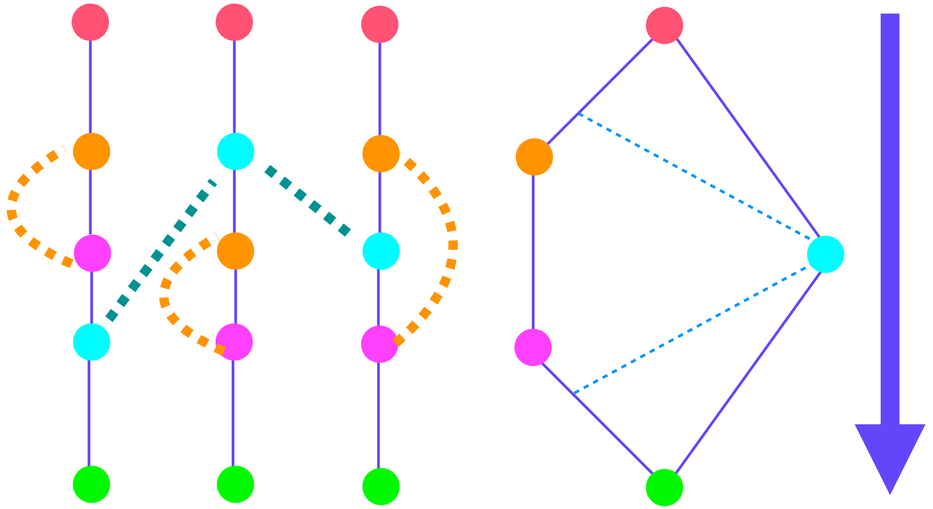
Complexity & Modularity 🧱

What do you mean by "composition"?

Execution Symmetry

A program can be ***developed on a sequential platform, even if it is meant to run on a parallel platform***, because the behaviour is not affected by whether we execute it using a sequential or parallel dynamics

– Robert Harper, Practical Foundations for Programming Languages (2012)



It commutes! 👍

Complexity & Modularity 🧱

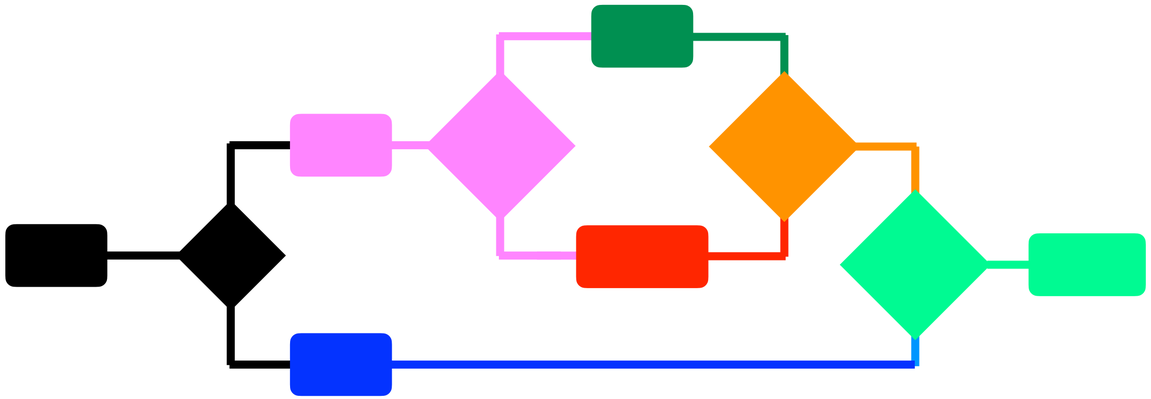
What do you mean by "composition"?

Explicit Data Flow

Complexity & Modularity 🧱

What do you mean by "composition"?

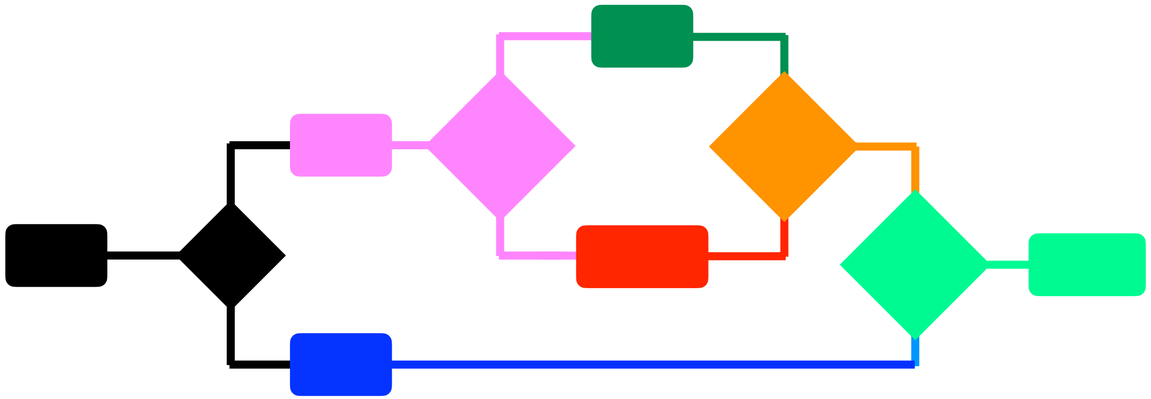
Explicit Data Flow



Complexity & Modularity

What do you mean by "composition"?

Explicit Data Flow

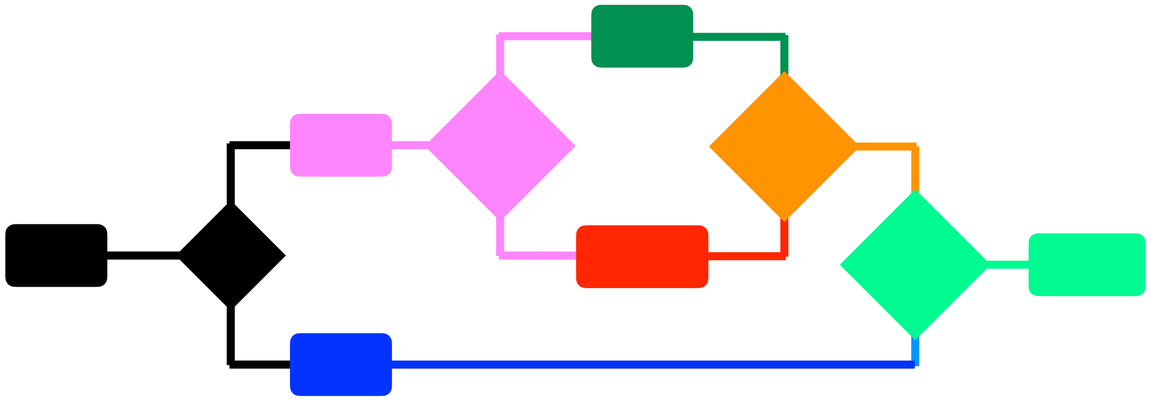


```
arrow_diagram =
  fanout(fn x -> x / 5 end, fn y -> y + 1 end
  |                         <~> fanout(&inspect/1, fn z -> z end) |
  |                         <~> unsplit(fn(a, b) -> "#{b}#{a}" end))
  | <~> unsplit(&String.at(&2, round(&1)) end)
```

Complexity & Modularity

What do you mean by "composition"?

Explicit Data Flow



```
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)
```

How Modular are Modules?

How Modular are Modules?

(and libraries)

How Modular are Modules?

(and libraries)

Complexity & Modularity 🧰

How Modular are Modules?

Module-Level Modularity

```
@modulespec Todo.Verical :: (Query.m(), Schema.m(), JSON.m() -> Vertical.m())
defmodule Todo.Vertical(queryMod \\ Ecto.Query, schemaMod \\ Ecto.Schema, jsonMod \\ Poison.Encoder) do
  use Phoenix.Vertical

  use queryMod
  use schemaMod

  schema "users" do
    field :name,     :string
    field :complete, :bool, default: false
  end

  def view_one(todo), do: json.encode(todo)
end
```

Complexity & Modularity 🧰

How Modular are Modules?

Module-Level Modularity

```
@modulespec Todo.Verical :: (Query.m(), Schema.m(), JSON.m() -> Vertical.m())
defmodule Todo.Vertical(queryMod \\ Ecto.Query, schemaMod \\ Ecto.Schema, jsonMod \\ Poison.Encoder) do
  use Phoenix.Vertical

  use queryMod
  use schemaMod

  schema "users" do
    field :name,     :string
    field :complete, :bool, default: false
  end

  def view_one(todo), do: json.encode(todo)
end
```

"Hot-swappable dependencies"

Complexity & Modularity

How Modular are Modules?

Module-Level Modularity

```
@modulespec Todo.Verical :: (Query.m(), Schema.m(), JSON.m() -> Vertical.m())
defmodule Todo.Vertical(queryMod \\ Ecto.Query, schemaMod \\ Ecto.Schema, jsonMod \\ Poison.Encoder) do
  use Phoenix.Vertical

  use queryMod
  use schemaMod

  schema "users" do
    field :name,     :string
    field :complete, :bool, default: false
  end

  def view_one(todo), do: json.encode(todo)
end
```

"Hot-swappable dependencies"

Complexity & Modularity 📦

How Modular are Modules?

Module-Level Modularity

```
@modulespec Todo.Verical :: (Query.m(), Schema.m(), JSON.m() -> Vertical.m())
defmodule Todo.Vertical(queryMod \\ Ecto.Query, schemaMod \\ Ecto.Schema, jsonMod \\ Poison.Encoder) do
  use Phoenix.Vertical

  use queryMod
  use schemaMod

  schema "users" do
    field :name,     :string
    field :complete, :bool, default: false
  end

  def view_one(todo), do: json.encode(todo)
end
```

"Hot-swappable dependencies"

Complexity & Modularity 📦

How Modular are Modules?

Module-Level Modularity

```
@modulespec Todo.Verical :: (Query.m(), Schema.m(), JSON.m() -> Vertical.m())
defmodule Todo.Vertical(queryMod \\ Ecto.Query, schemaMod \\ Ecto.Schema, jsonMod \\ Poison.Encoder) do
  use Phoenix.Vertical

  use queryMod
  use schemaMod

  schema "users" do
    field :name,     :string
    field :complete, :bool, default: false
  end

  def view_one(todo), do: json.encode(todo)
end
```

"Hot-swappable dependencies"

Complexity & Modularity 🧱

How Modular are Modules?

~~Hacking~~ *Extending* the Module System

Complexity & Modularity 📦

How Modular are Modules?

~~Hacking~~ *Extending* the Module System

```elixir
defmodule TypeClass.Dependency do
  defmacro __using__(_) do
    quote do
      import unquote(__MODULE__)
      unquote(__MODULE__).set_up()
    end
  end

  defmacro set_up do
    quote do
      import TypeClass.Utility.Attribute
      register(:extend, accumulate: true)
    end
  end

  defmacro extend(parent_class) do
    quote do
      require unquote(parent_class)
      @extend unquote(parent_class)
    end
  end
```

Complexity & Modularity 🧰

How Modular are Modules?

~~Hacking~~ *Extending* the Module System

```elixir
defmodule TypeClass.Dependency do
  defmacro __using__(_) do
    quote do
      import unquote(__MODULE__)
      unquote(__MODULE__).set_up()
    end
  end

  defmacro set_up do
    quote do
      import TypeClass.Utility.Attribute
      register(:extend, accumulate: true)
    end
  end

  defmacro extend(parent_class) do
    quote do
      require unquote(parent_class)
      @extend unquote(parent_class)
    end
  end
```

```elixir
  defmacro extend(parent_class, alias: true) do
    quote do
      extend unquote(parent_class)
      alias unquote(parent_class)
    end
  end

  defmacro extend(parent_class, alias: alias_as) do
    quote do
      extend unquote(parent_class)
      alias unquote(parent_class), as: unquote(alias_as)
    end
  end

  defmacro run do
    quote do
      def __dependencies__, do: @extend
    end
  end
end
```

Higher Order Modules 💞 Behaviours

Declarative Embedded DSLs

Declarative Embedded DSLs 🍏🍊

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

– Tony Hoare, Turing Aware Lecture, 1980

Declarative Embedded DSLs 🍏🍊

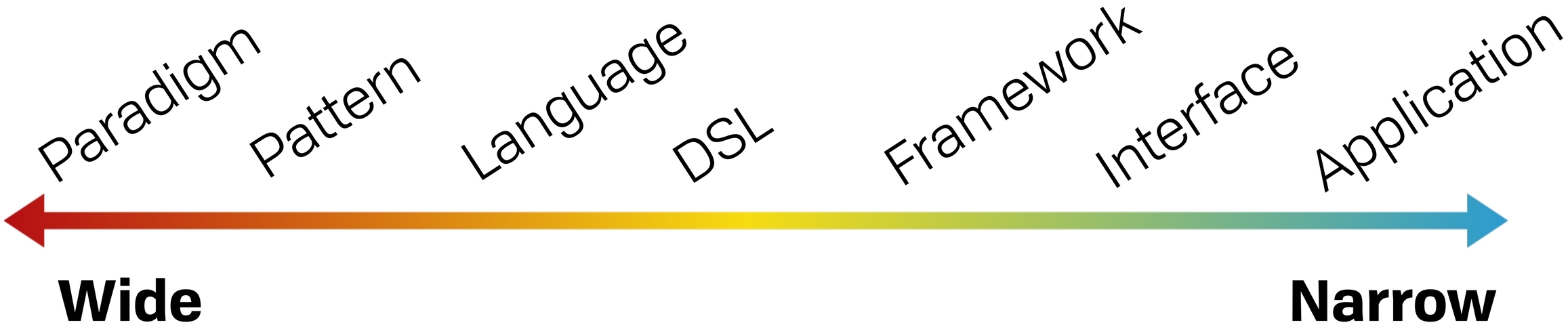
Isn't *Everything* a DSL?

Declarative Embedded DSLs 🍏🍊

Isn't *Everything* a DSL?

Declarative Embedded DSLs 🍏🍊

Isn't *Everything* a DSL?



Declarative Embedded DSLs 🍏🍊

Isn't *Everything* a DSL?

Counterexample

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

Declarative Embedded DSLs 🍏🍊

Isn't *Everything* a DSL?

Counterexample

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

This is just Elixir!

Declarative Embedded DSLs 🍏🍊

Isn't *Everything* a DSL?

Algebraic Data Type eDSL

Declarative Embedded DSLs 🍏🍊

Isn't *Everything* a DSL?

Algebraic Data Type eDSL

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

Maybe.new()
#=> %Maybe.Nothing{}
```

Declarative Embedded DSLs 🍏🍊

Isn't *Everything* a DSL?

Algebraic Data Type eDSL

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

Maybe.new()
#=> %Maybe.Nothing{}
```

```
defmodule Light do
  defsum do
    defdata Red    :: none()
    defdata Yellow :: none()
    defdata Green  :: none()
  end

  def from_number(1), do: %Light.Red{}
  def from_number(2), do: %Light.Yellow{}
  def from_number(3), do: %Light.Green{}
end
```

Declarative Embedded DSLs 🍏🍊

Isn't *Everything* a DSL?

Algebraic Data Type eDSL

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

Maybe.new()
#=> %Maybe.Nothing{}
```

```
defmodule Light do
  defsum do
    defdata Red    :: none()
    defdata Yellow :: none()
    defdata Green  :: none()
  end

  def from_number(1), do: %Light.Red{}
  def from_number(2), do: %Light.Yellow{}
  def from_number(3), do: %Light.Green{}
end
```

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

  @spec attack(t(), t()) :: {t(), t()}
  def attack(%{experience: xp} = player, %{hit_points: hp} = target) do
    {
      %{player | experience: xp + 50},
      %{target | hit_points: hp - 10}
    }
  end
end
```

Business Language for Business Time

Declarative Embedded DSLs 🍏🍊

Business Language 💼

Declarative Embedded DSLs 🍏🍊

Business Language 💼

What can we learn from COBOL?

(Yes, really. COBOL.)

Toggl, How to Kill at the Dragon in 9 Programming Languages

Declarative Embedded DSLs 🍏🍊

What Can We Learn from COBOL?

```
PROCEDURE DIVISION.
*Run the code as performed paragraphs
      PERFORM GET-DATA
      PERFORM CALC-DATA
      PERFORM SHOW-DATA
      PERFORM FINISH-UP
      GOBACK.

*A performed paragraph to get user input
GET-DATA.
      MOVE SPACE TO WS-USER WS-FULL-NAME
      DISPLAY "What is your first name?"
      ACCEPT WS-FIRST-NAME OF WS-USER
      DISPLAY "What is your last name?"
      ACCEPT WS-LAST-NAME OF WS-USER
      DISPLAY "What is your age?"
      ACCEPT WS-AGE OF WS-USER
      STRING WS-FIRST-NAME OF WS-USER DELIMITED BY SPACE
      SPACE DELIMITED BY SIZE
            WS-LAST-NAME OF WS-USER DELIMITED BY SPACE
            SPACE DELIMITED BY SIZE
            INTO WS-FULL-NAME
            ON OVERFLOW
            DISPLAY "SORRY, YOUR DATA WAS TRUNCATED"
     END-STRING.

*A performed paragraph for doing calculation
CALC-DATA.
* Sample addition statement
       ADD  WS-AGE-DELTA WS-AGE OF WS-USER TO WS-NEW-AGE.

*A performed paragraph to display output
SHOW-DATA.
       DISPLAY "Welcome " WS-FULL-NAME " In ten years you will be: "
        WS-NEW-AGE.

*A performed paragraph to end the program
FINISH-UP.
      DISPLAY "Strike any key to continue".
      ACCEPT WS-CLOSE
      DISPLAY "Good bye".
END PROGRAM RESEL-WORLD.
```

Declarative Embedded DSLs 🍏🍊

What Can We Learn from COBOL?

When No One Wants to Go To Jail 👩‍⚖️

```
TOKEN cross_border
  IS ERC777
  IS ERC902

  CONFORM
    bc_securities_commission
    AND usa_sec
    AND korea_exchange

  ALLOW issue
    AFTER  BLOCK 9000000
    BEFORE BLOCK 10000000

  RESTRICT holder
    CHECK allowed IN ERC902 AT 0xaa9cf224d798123fde793fc41c72d3bb8c1c9a09

  ISSUANCE
    START 0
    MAX   10_000
```

Declarative Embedded DSLs 🍏🍊

What Can We Learn from COBOL?

When No One Wants to Go To Jail 👩‍⚖️

- Needs to be readable by lawyers
  - (Who can't read code)
- Formal methods & static analysis
  - Down to the compiler, of course
- An *unholy union* of COBOL and Prolog

```
TOKEN cross_border
  IS ERC777
  IS ERC902

  CONFORM
    bc_securities_commission
    AND usa_sec
    AND korea_exchange

  ALLOW issue
    AFTER  BLOCK 9000000
    BEFORE BLOCK 10000000

  RESTRICT holder
    CHECK allowed IN ERC902 AT 0xaa9cf224d798123fde793fc41c72d3bb8c1c9a09

  ISSUANCE
    START 0
    MAX   10_000
```

Declarative Embedded DSLs 🍏🍊

What Can We Learn from COBOL?

When No One Wants to Go To Jail 👩‍⚖️

- Needs to be readable by lawyers
  - (Who can't read code)
- Formal methods & static analysis
  - Down to the compiler, of course
- An *unholy union* of COBOL and Prolog

```
TOKEN cross_border
  IS ERC777
  IS ERC902

  CONFORM
    bc_securities_commission
    AND usa_sec
    AND korea_exchange

  ALLOW issue
    AFTER  BLOCK 9000000
    BEFORE BLOCK 10000000

  RESTRICT holder
    CHECK allowed IN ERC902 AT 0xaa9cf224d798123fde793fc41c72d3bb8c1c9a09

  ISSUANCE
    START 0
    MAX   10_000
```

Declarative Embedded DSLs 🍏🍊

Upside

Declarative Embedded DSLs 🍏🍊

Upside

1. *Fabulous* for communicating with domain experts
2. We know how these DSLs work (e.g. they form an algebra)
3. They can be correct-by-construction
4. Check for various properties (compile- or run-time)
5. A language that exactly fits your needs (DDD)

But We Have a Problem

But We Have a Problem

(Hint: it's inflexibility)

🏝️ Shallow Embedding 🦀

🏝️ *Shallow Embedding* 🦀

i.e. The "Quick and Dirty" Way

Declarative Embedded DSLs 🍏🍊

Shallow Embedding

Declarative Embedded DSLs 🍏🍊

Shallow Embedding

- Just use the built-in AST
- What it can represent is limited
- e.g. Ecto, Algae

Declarative Embedded DSLs 🍏🍊

Shallow Embedding

```
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
```

- Just use the built-in AST
- What it can represent is limited
- e.g. Ecto, Algae

Declarative Embedded DSLs 🍏🍊

Shallow Embedding

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

- Just use the built-in AST
- What it can represent is limited
- e.g. Ecto, Algae

Declarative Embedded DSLs 🍏🍊

Shallow Embedding

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

```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{}
    }
  }
}
```

- Just use the built-in AST
- What it can represent is limited
- e.g. Ecto, Algae

🌊 Deep Embedding 🦈

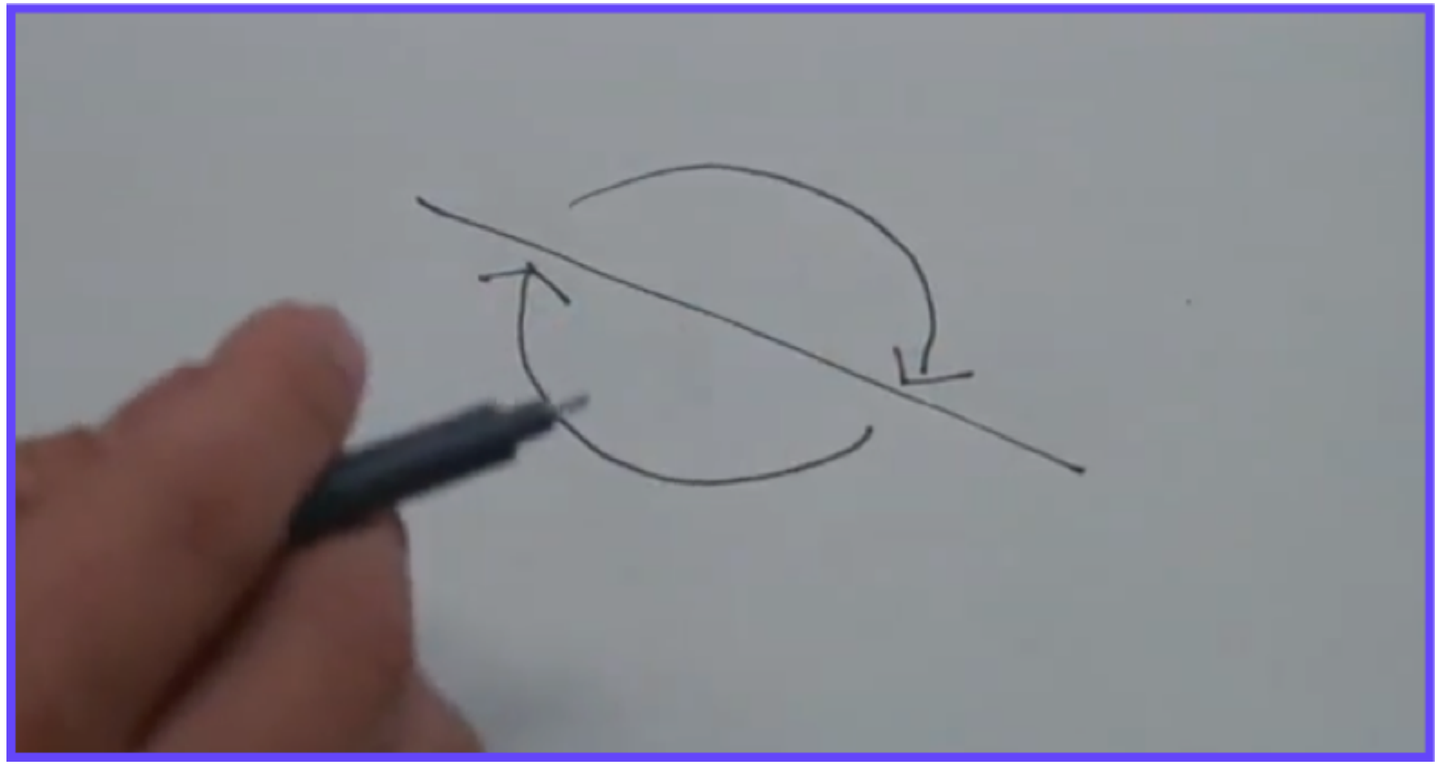
🌊 *Deep Embedding* 🦈

"Better AST"

Declarative Embedded DSLs 🍏🍊

Deep Embedding

Three Steps

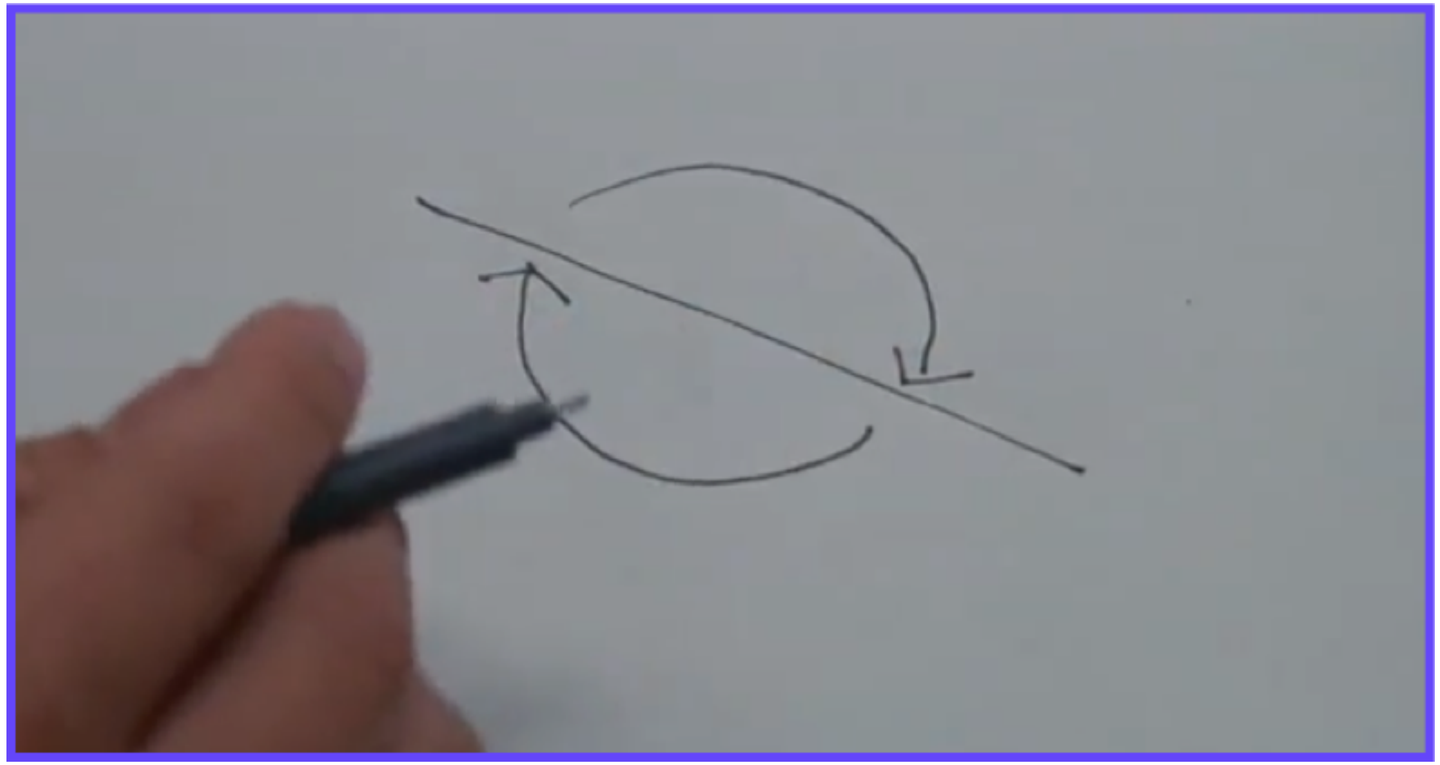


Declarative Embedded DSLs 🍏🍊

Deep Embedding

Three Steps

1. Build a game plan
2. Transform (optional)
3. Tear down



Declarative Embedded DSLs 🍏🍊

Deep Embedding

"Build a Game Plan" Example

```
%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))
}
|> Witchcraft.Functor.map(&IO.inspect/1)
```

Declarative Embedded DSLs 🍏🍊

Deep Embedding

"Build a Game Plan" Example

```
%Unsplit{
  split: %Split{
    with: &String.at(&2, round(&1))
  }
  |> Witchcraft.Functor.map(&IO.inspect/1)
```

```
|> 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)))
```

```
-> "#{right}#{left}" end
```

Declarative Embedded DSLs 🍏🍊

Deep Embedding

Tradeoffs

Declarative Embedded DSLs 🍏🍊

Deep Embedding

Tradeoffs

- More work to write (write your own AST)
- Way more powerful (full control)
- Precisely the vocabulary that you need — exact surface area
- Can check more things about the meaning of your code
- Logic-as-data is MUCH simpler to debug than running functions
- Time travelling debugging!
- Unlike protocols, you’re not locked into one canonical implementation

Declarative Embedded DSLs 🍏🍊

Deep Embedding

Combine Powerful, Modular, Reusable DSLs!

```
with_npc :dog do
  go_north
  wait_seconds 2

  set_caps true
  print "woof"
  set_caps false

  go_west
  wait_seconds 1
end
```

Declarative Embedded DSLs 🍏🍊

Deep Embedding

Desugar

```
[
  %GoNoth{},
  %Wait{seconds: 2},
  %SetCaplock{is_cap: true},
  %Print{text: "woof"},
  %SetCaplock{is_cap: true},
  %GoWest{},
  %Wait{seconds: 3}
]
```

```
%AST{
  cmd: %GoNoth{},
  next: %AST{
    cmd: %Wait{seconds: 2}
    next: # ...
  }
}
```

Declarative Embedded DSLs 🍏🍊

Deep Embedding

"Huh, this kinda feels like GenServer" 😉

Declarative Embedded DSLs 🍏🍊

Deep Embedding

"Huh, this kinda feels like GenServer" 😉

```
def handle_text(%Print{msg: msg}, _), do: IO.puts(msg)

def handle_text(%SetBlink{should_blink: should_blink}, _) do
  if should_blink, do:
    IO.ANSI.blink_rapid()
  else
    IO.ANSI.blink_off()
  end
end

def handle_text(%SetColour{red: r, green: g, blue: b}, _), do: IO.ANSI.color(r, g, b)

def handle_text(%CAPSLOCK{set_caps: is_caps}, state_agent), do
  Agent.update(state_agent, fn state -> %{state | is_capslock: is_caps } end
  IO.ANSI.cursor(@agent.current_line - 1, column)
end

def handle_text(not_text_cmd, _), do: not_text_cmd
```

Declarative Embedded DSLs 🍏🍊

Deep Embedding

"Huh, this kinda feels like GenServer" 😉

```
def handle_text(%Print{msg: msg}, _), do: IO.puts(msg)

def handle_text(%SetBlink{should_blink: should_blink}, _) do
  if should_blink, do:
    IO.ANSI.blink_rapid()
  else
    IO.ANSI.blink_off()
  end
end

def handle_text(%SetColour{red: r, green: g, blue: b}, _), do: IO.ANSI.color(r, g, b)

def handle_text(%CAPSLOCK{set_caps: is_caps}, state_agent), do
  Agent.update(state_agent, fn state -> %{state | is_capslock: is_caps } end
  IO.ANSI.cursor(@agent.current_line - 1, column)
end

def handle_text(not_text_cmd, _), do: not_text_cmd
```

```
def handle_gameplay(%BuyItem{item: item}, _), do: ...
def handle_gameplay(%ApplyPowerup{item: item}, _), do: ...
def handle_gameplay(not_gameplay, _), do: not_gameplay
```

Declarative Embedded DSLs 🍏🍊

Deep Embedding

"Huh, this kinda feels like GenServer" 😉

```
def handle_text(%Print{msg: msg}, _), do: IO.puts(msg)

def handle_text(%SetBlink{should_blink: should_blink}, _) do
  if should_blink, do:
    IO.ANSI.blink_rapid()
  else
    IO.ANSI.blink_off()
  end
end

def handle_text(%SetColour{re

def handle_text(%CAPSLOCK{set
  Agent.update(state_agent, f
  IO.ANSI.cursor(@agent.curre
end

def handle_text(not_text_cmd, _), do: not_text_cmd
```

```
def handle_gameplay(%BuyItem{item: item}, _), do: ...
def handle_gameplay(%ApplyPowerup{item: item}, _), do: ...
def handle_gameplay(not_gameplay, _), do: not_gameplay
```

```
def handle_movement(%GoNorth{}, _), do: ...
def handle_movement(%GoSouth{}, _), do: ...
def handle_movement(%GoEast{}, _), do: ...
def handle_movement(%GoWest{}, _), do: ...
def handle_movement(not_movement, _), do: not_movement
```

Declarative Embedded DSLs 🍏🍊

Deep Embedding

With Their Powers Combined!

```
def interpreter(cmd, agent) do
  cmd
  |> handle_gameplay(agent)
  |> handle_movement(agent)
  |> handle_text(agent)
end
```

Declarative Embedded DSLs 🍏🍊

Deep Embedding

One Last Line

```
with_npc :dog do
  go_north
  wait_seconds 2

  set_caps true
  print "woof"
  set_caps false

  go_west
  wait_seconds 1
end
|> Enum.map(fn cmd -> interpreter(cmd, agent) end)
```

Declarative Embedded DSLs 🍏🍊

Deep Embedding

One Last Line

```
with_npc :dog do
  go_north
  wait_seconds 2

  set_caps true
  print "woof"
  set_caps false

  go_west
  wait_seconds 1
end
|> Enum.map(fn cmd -> interpreter(cmd, agent) end)
```

Declarative Embedded DSLs 🍏🍊

Deep Embedding

More Flexible Than `protocols`

Declarative Embedded DSLs 🍏🍊

Deep Embedding

More Flexible Than `protocols`

1. Protocols require canonicity
2. Libraries of well-defined mini-languages, even without interpreter
3. Different in tests and prod (trivial to mock)

Let's Make New Mistakes! 💥✨

Let's Make New Mistakes! 💥✨

5 Problems for the Next 30 Years of BEAM

Let's Make New Mistakes

5 Problems for the Next 30 Years of BEAM

Let's Make New Mistakes

5 Problems for the Next 30 Years of BEAM

1. Wasm — client & edge BEAM

Let's Make New Mistakes

5 Problems for the Next 30 Years of BEAM

1. Wasm — client & edge BEAM
2. Lower the barrier to entry (low code)

Let's Make New Mistakes

5 Problems for the Next 30 Years of BEAM

1. Wasm — client & edge BEAM
2. Lower the barrier to entry (low code)
3. Correctness tools (i.e. better static & dynamic analysis, formal methods)

Let's Make New Mistakes

5 Problems for the Next 30 Years of BEAM

1. Wasm — client & edge BEAM
2. Lower the barrier to entry (low code)
3. Correctness tools (i.e. better static & dynamic analysis, formal methods)
4. Automatic dynamic parallel evaluation

Let's Make New Mistakes

5 Problems for the Next 30 Years of BEAM

1. Wasm — client & edge BEAM
2. Lower the barrier to entry (low code)
3. Correctness tools (i.e. better static & dynamic analysis, formal methods)
4. Automatic dynamic parallel evaluation
5. Mobile agents (incl. dynamic FaaS)

Parting Thought 🧠

–Alan J. Perlis, Epigrams on Programming (1982)

Epigram 101

–Alan J. Perlis, Epigrams on Programming (1982)

Epigram 101

Dealing with ***failure is easy:***

–Alan J. Perlis, Epigrams on Programming (1982)

Epigram 101

Dealing with ***failure is easy:***
work hard to improve.

–Alan J. Perlis, Epigrams on Programming (1982)

Epigram 101

Dealing with ***failure is easy:***
work hard to improve.

Success is also easy to handle:

–Alan J. Perlis, Epigrams on Programming (1982)

Epigram 101

Dealing with ***failure is easy:***
work hard to improve.

Success is also easy to handle:
you've solved the wrong problem.

–Alan J. Perlis, Epigrams on Programming (1982)

Epigram 101

Dealing with ***failure is easy:***
work hard to improve.

Success is also easy to handle:
you've solved the wrong problem.

Work hard to improve.

–Alan J. Perlis, Epigrams on Programming (1982)

Thank You, India

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