Introduction to Programming Paradigms

And the Myths Thereof

You may have heard terms like this…


Functional!
Object Oriented!
Procedural!
Declarative!

Perhaps you felt like this…


What do these words
even mean?

paradigm

"…3. a philosophical and theoretical framework of a scientific school or discipline…"

MERRIAM WEBSTER

Categories of programming languages

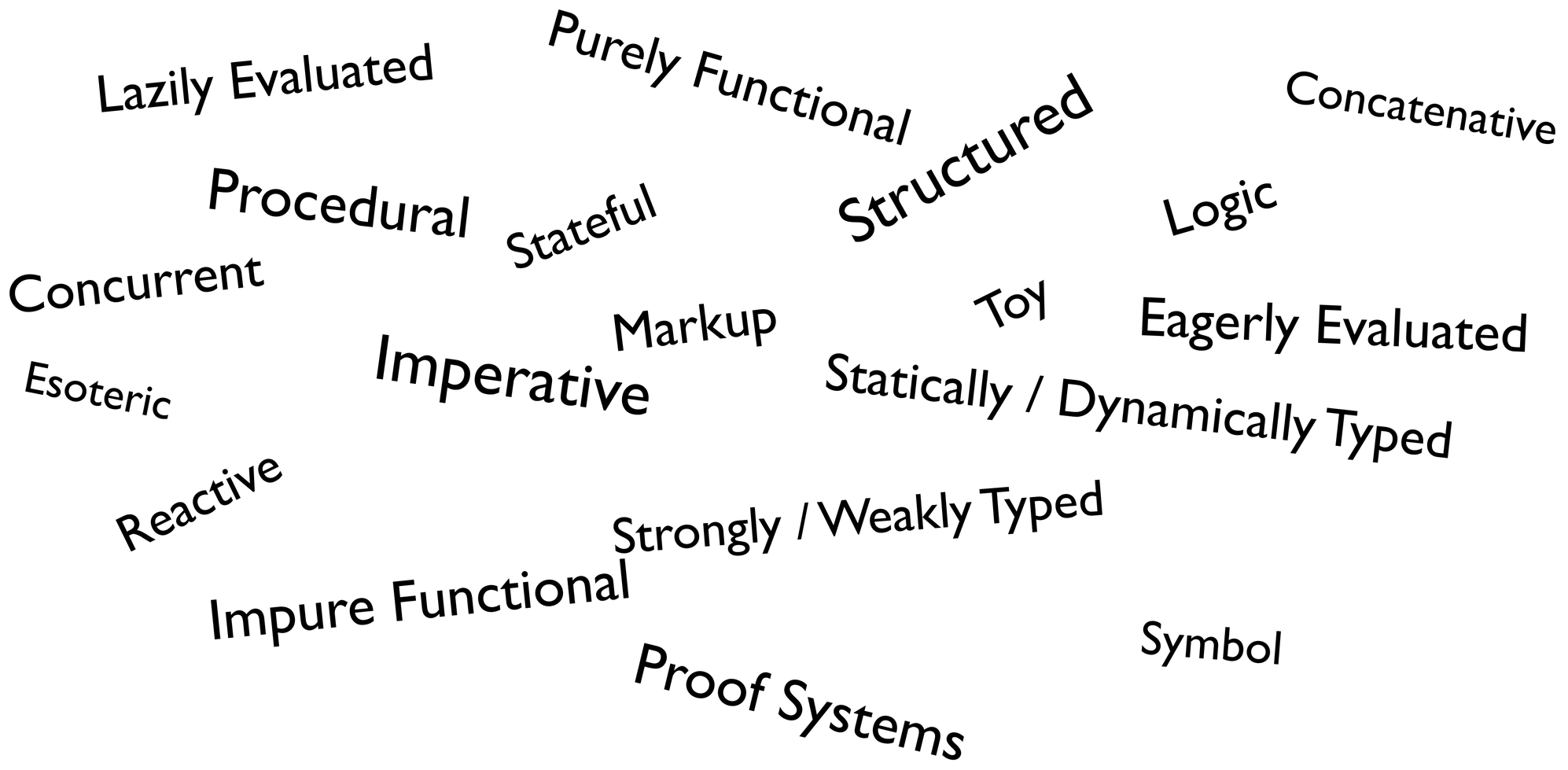
Traditionally viewed as competing styles 🤔

Different syntax, capabilities, goals, and/or concepts

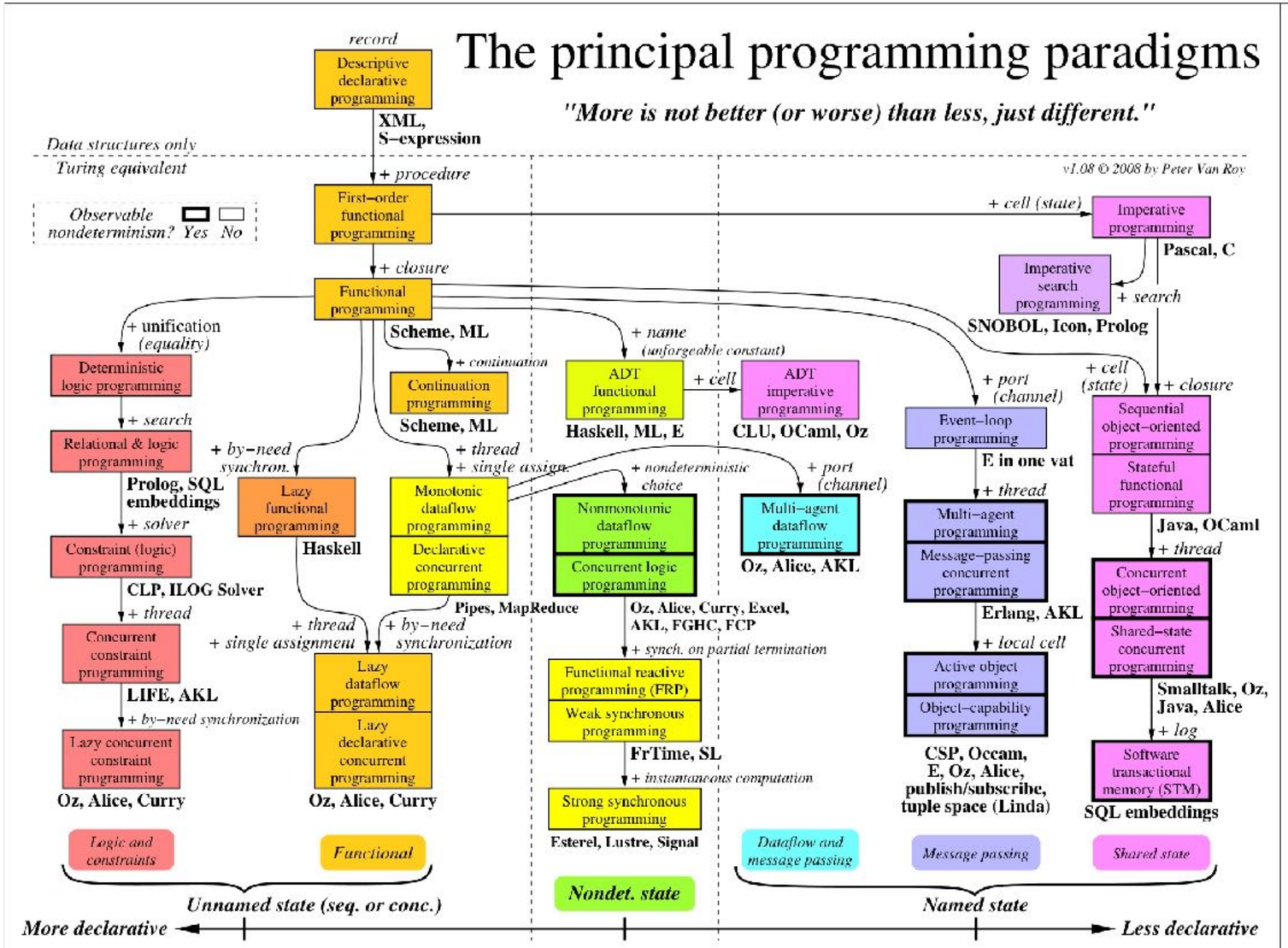
(Some) Oft-Cited Examples

Paradigm	Languages
Procedural	FORTRAN / ALGOL / C / BASIC
Object Oriented	Simula / Smalltalk / C++ / Java / Ruby
Functional	Lisp / Scheme / OCaml / Haskell / Elm
Declarative	SQL / HTML / RegEx

But Wait, There's More!™



A Large & Arbitrary Topic



Explanations

See "Concepts, Techniques, and Models of Computer Programming".

The chart classifies programming paradigms according to their kernel languages (the small core language in which all the paradigm's abstractions can be defined). Kernel languages are ordered according to the creative extension principle: a new concept is added when it cannot be encoded with only local transformations. Two languages that implement the same paradigm can nevertheless have very different "flavors" for the programmer, because they make different choices about what programming techniques and styles to facilitate.

When a language is mentioned under a paradigm, it means that part of the language is intended (by its designers) to support the paradigm without interference from other paradigms. It does not mean that there is a perfect fit between the language and the paradigm. It is not enough that libraries have been written in the language to support the paradigm. The language's kernel language should support the paradigm. When there is a family of related languages, usually only one member of the family is mentioned to avoid clutter. The absence of a language does not imply any kind of value judgment.

State is the ability to remember information, or more precisely, to store a sequence of values in time. Its expressive power is strongly influenced by the paradigm that contains it. We distinguish four levels of expressiveness, which differ in whether the state is unnamed or named, deterministic or nondeterministic, and sequential or concurrent. The least expressive is functional programming (threaded state, e.g., DCGs and monads: unnamed, deterministic, and sequential). Adding concurrency gives declarative concurrent programming (e.g., synchrocells: unnamed, deterministic, and concurrent). Adding nondeterministic choice gives concurrent logic programming (which uses stream mergers: unnamed, nondeterministic, and concurrent). Adding ports or cells, respectively, gives message passing or shared state (both are named, nondeterministic, and concurrent). Nondeterminism is important for real-world interaction (e.g., client/server). Named state is important for modularity.

Axes orthogonal to this chart are typing, aspects, and domain-specificity. Typing is not completely orthogonal: it has some effect on expressiveness. Aspects should be completely orthogonal, since they are part of a program's specification. A domain-specific language should be definable in any paradigm (except when the domain needs a particular concept).

Metaprogramming is another way to increase the expressiveness of a language. The term covers many different approaches, from higher-order programming, syntactic extensibility (e.g., macros), to higher-order programming combined with syntactic support (e.g., meta-object protocols and generics), to full-fledged tinkering with the kernel language (introspection and reflection). Syntactic extensibility and kernel language tinkering in particular are orthogonal to this chart. Some languages, such as Scheme, are flexible enough to implement many paradigms in almost native fashion. This flexibility is not shown in the chart.

WHAT ABOUT JS?

*"JavaScript is a prototype-based, **multi-paradigm**, dynamic language, supporting **object-oriented**, **imperative**, and **declarative** (e.g. **functional** programming) styles."*

MDN

SAY WHAT!?

Multi-Paradigm

- Many modern languages cannot be neatly placed into paradigms.
- JS, Java, C++, Python, Swift, and others blur the lines.
- Paradigms are hard to define and mean different things according to different authors.

"Programming language 'paradigms' are a moribund and tedious legacy of a bygone age. Modern language designers pay them no respect, so why do our courses slavishly adhere to them?"

SHRIRAM KIRSHNAMURTHI, TEACHING PROGRAMMING LANGUAGES IN A POST-LINNAEAN AGE

"If languages are not defined by taxonomies, how are they constructed? ***They are aggregations of features****."*

A Random Set of Language Features

- Garbage Collection
- Significant Whitespace
- Closures
- Blocks
- First-Class Functions
- Lazy Evaluation
- Type Inference
- Static Typing
- Dynamic Typing
- Memory Access
- Try-Finally
- Pattern Matching
- If Expressions
- Currying
- Inheritance
- Mutable State

Structured

- Garbage Collection
- Significant Whitespace
- Closures
- **Blocks**
- First-Class Functions
- Lazy Evaluation
- Type Inference
- Static Typing
- Dynamic Typing
- Memory Access
- **Try-Finally**
- Pattern Matching
- If Expressions
- Currying
- Inheritance
- Mutable State

Imperative

- ~~Garbage Collection~~
- Significant Whitespace
- Closures
- Blocks*
- First-Class Functions
- Lazy Evaluation
- ~~Type Inference~~
- Static Typing
- Dynamic Typing
- **Memory Access**
- **Try-Finally**
- ~~Pattern Matching~~
- If Expressions
- Currying
- Inheritance
- **Mutable State**

Object-Oriented

- Garbage Collection
- Significant Whitespace
- Closures
- Blocks
- First-Class Functions
- Lazy Evaluation
- Type Inference
- Static Typing*
- Dynamic Typing
- Memory Access*
- Try-Finally
- Pattern Matching
- If Expressions
- Currying
- **Inheritance**
- **Mutable State***

Declarative

- Garbage Collection
- Significant Whitespace*
- Closures
- Blocks*
- First-Class Functions
- Lazy Evaluation
- Type Inference*
- Static Typing
- Dynamic Typing
- ~~Memory Access~~
- ~~Try-Finally~~
- Pattern Matching
- If Expressions*
- Currying
- Inheritance
- ~~Mutable State~~

Functional

- Garbage Collection
- Significant Whitespace
- Closures
- Blocks
- First-Class Functions
- Lazy Evaluation*
- Type Inference*
- Static Typing*
- Dynamic Typing
- Memory Access
- Try-Finally
- Pattern Matching
- If Expressions
- Currying
- Inheritance
- Mutable State

C

- Garbage Collection
- Significant Whitespace
- Closures
- **Blocks**
- First-Class Functions*
- Lazy Evaluation
- Type Inference
- **Static Typing**
- Dynamic Typing
- **Memory Access**
- Try-Finally*
- Pattern Matching
- If Expressions
- Currying
- Inheritance
- **Mutable State**

JavaScript

- Garbage Collection
- Significant Whitespace
- Closures
- Blocks
- First-Class Functions
- Lazy Evaluation
- Type Inference
- Static Typing
- Dynamic Typing
- Memory Access
- Try-Finally
- Pattern Matching*
- If Expressions
- Currying*
- Inheritance
- Mutable State

Haskell

- Garbage Collection
- Significant Whitespace
- Closures
- Blocks*
- First-Class Functions
- Lazy Evaluation
- Type Inference
- Static Typing
- Dynamic Typing
- Memory Access*
- Try-Finally
- Pattern Matching
- If Expressions
- Currying
- Inheritance
- Mutable State*

Paradigm: Declarative

…and beginning to get functional, too…

Declare What/Logic (Not How/Sequence)

Layer	Example	Some Omitted Implementation
HTML	`<h1><em>Hello</em>, World</h1>`	How does this get rendered with size/color?
HTTP	`GET /api/users/1`	What steps does a backend take to find data?
SQL	`GET name, age FROM users LIMIT 10`	How is this optimized?
RegEx	`/^.+@.+$/` *(too-simple email regex)*	What algorithm detects matches?
pure func call	`nthFibonacci(99)`	Does it use recursion? Loops? Lookup table?
math exp	`- 3 + 7 * 2 / (1 - 9)`	Which parts need to be calculated first?

Every declarative layer has an imperative implementation layer behind it somewhere.

Every declarative layer has an imperative implementation layer behind it somewhere.

Declarative Layer	Imperative Implementation Layer
HTML	Browser's HTML parser → DOM representation
HTTP (e.g. `` `GET /api/users` ``)	TCP/IP in Node via C++, Express routes in JS
SQL	RDBMS, see `` `explain query plan` ``
Regular Expression	RegEx engine builds a Finite State Machine
pure function call, e.g. `` `circleArea(3)` ``	function body, e.g. `` `Math.PI * radius ** 2` ``
Mathematical Expression	Parser converts string to tree, traverses it

Concepts of Functional Programming

Overview, History, Theory &c.


HISTORY
λ

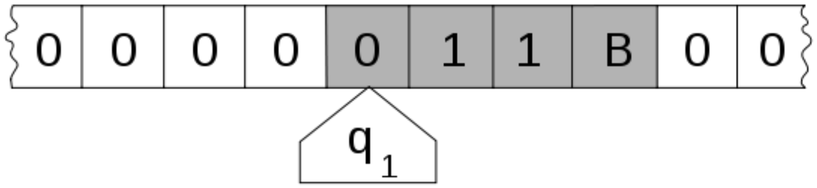
Theories of Computability

Alonzo Church

Alan Turing

*(*benefitted from many other mathematicians, including Haskell, Schönfinkel, Frege, Rósza Péter)*

$(\lambda xy.x\ y\ ((\lambda fab.fba)\ y))$



ca. 1928 develops Lambda Calculus

all computation can be expressed as **applications of *pure functions***

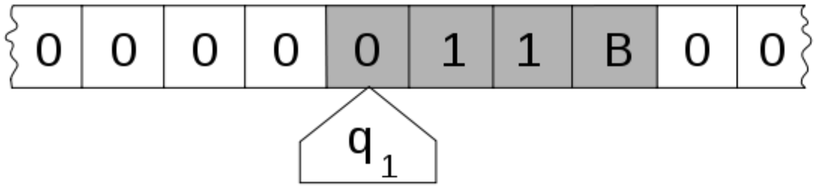
ca. 1936 develops Turing Machine

all computation can be expressed as ***state machine* operations**

Church-Turing Equivalence

Turn out to be the same concept, just expressed in two different ways.

$(\lambda xy.x\ y\ ((\lambda fab.fba)\ y))$ ⟷



Exciting because it means code can be stateless and abstract

Exciting because it means we can make real computers

FP in a 🌰 Nutshell

- 🎶 Functions everywhere (naturally)
- 🎵 Composition (small pieces → larger constructs)
- 💘 Purity (input → output, no effects)
- 💞 Equational reasoning (call & value interchangeable)
- 💟 First-class & higher-order (code uses / produces code)
- 💗 Currying & partial application (general-purpose → specific)
- 💎 Immutability (foolproof, supports equational reasoning)
- λ Mathematical (lambda calculus, category theory; law-based)

💡 Motivations 💡

Derive new code from old

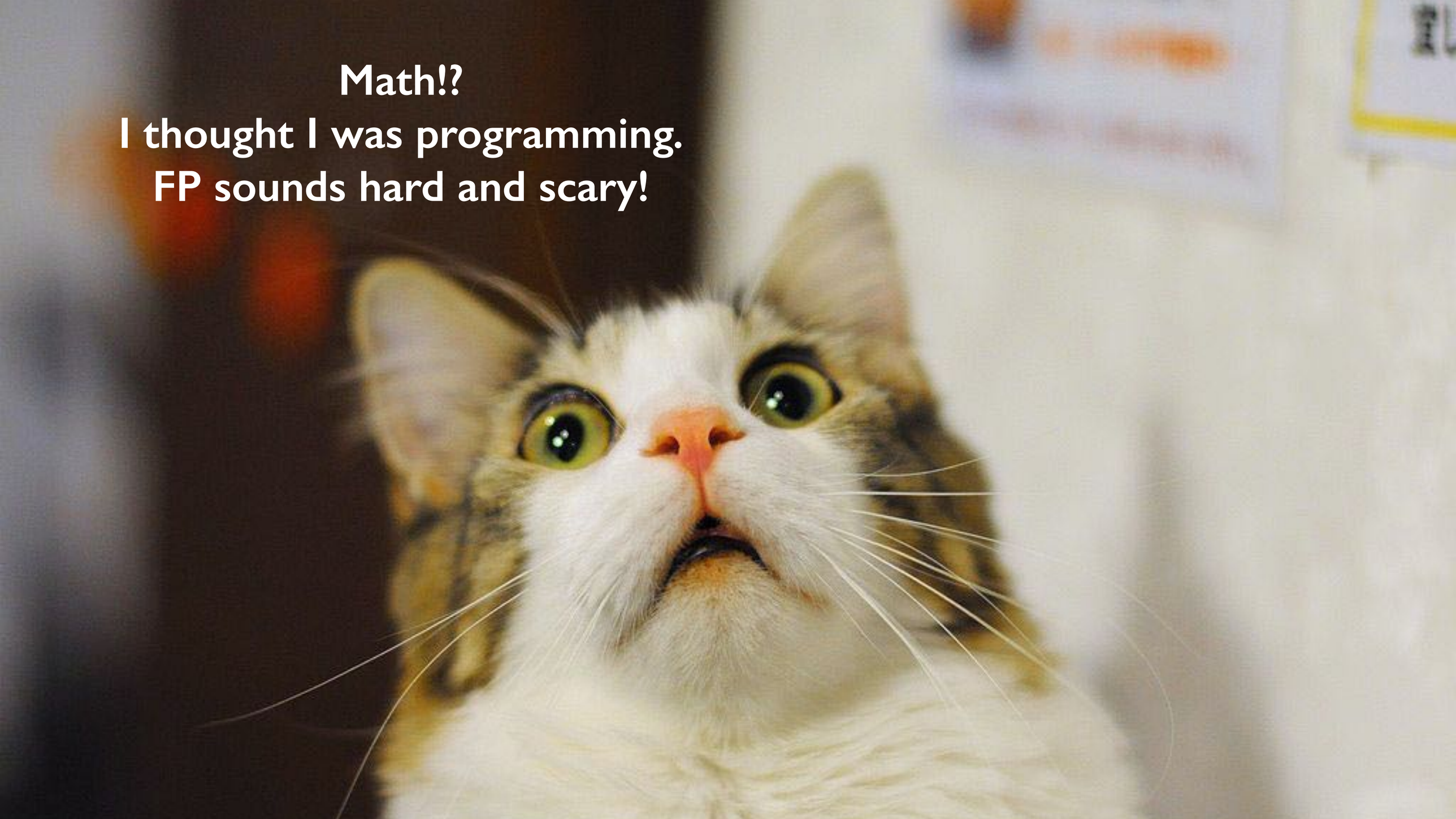
Pieces work well together

Reduced mental scope while writing

Certain classes of bug are made impossible

Mathematical laws are universal, unambiguous

You can't have Functional without Fun!


Math!?
I thought I was programming.
FP sounds hard and scary!

James Iry
@jamesiry

Follow

Functional programmer: (noun) One who names variables "x", names functions "f", and names code patterns "zygohistomorphic prepromorphism"

10:58 AM - 13 May 2015

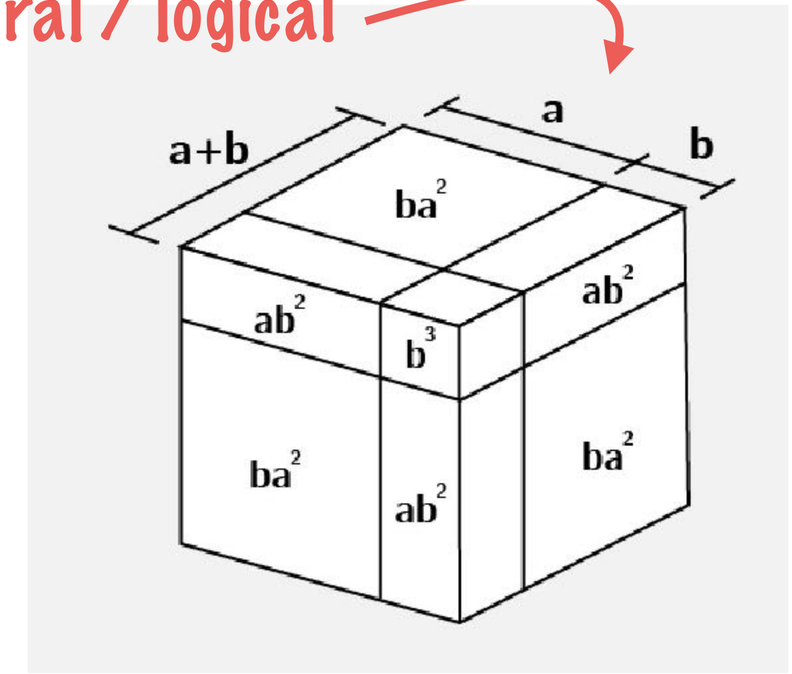
1,680 Retweets **1,634** Likes

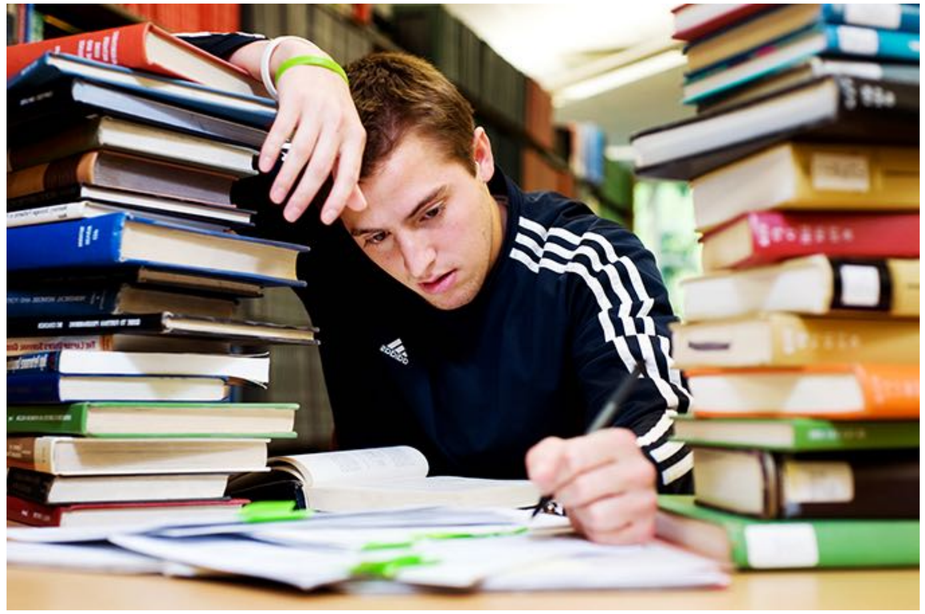
24 1.7K 1.6K

"All told, a monad in X is just a monoid in the category of endofunctors of X, with product × replaced by composition of endofunctors and unit set by the identity endofunctor."

—Saunders Mac Lane,
CATEGORIES FOR THE WORKING MATHEMATICIAN

Familiar, feels natural / logical

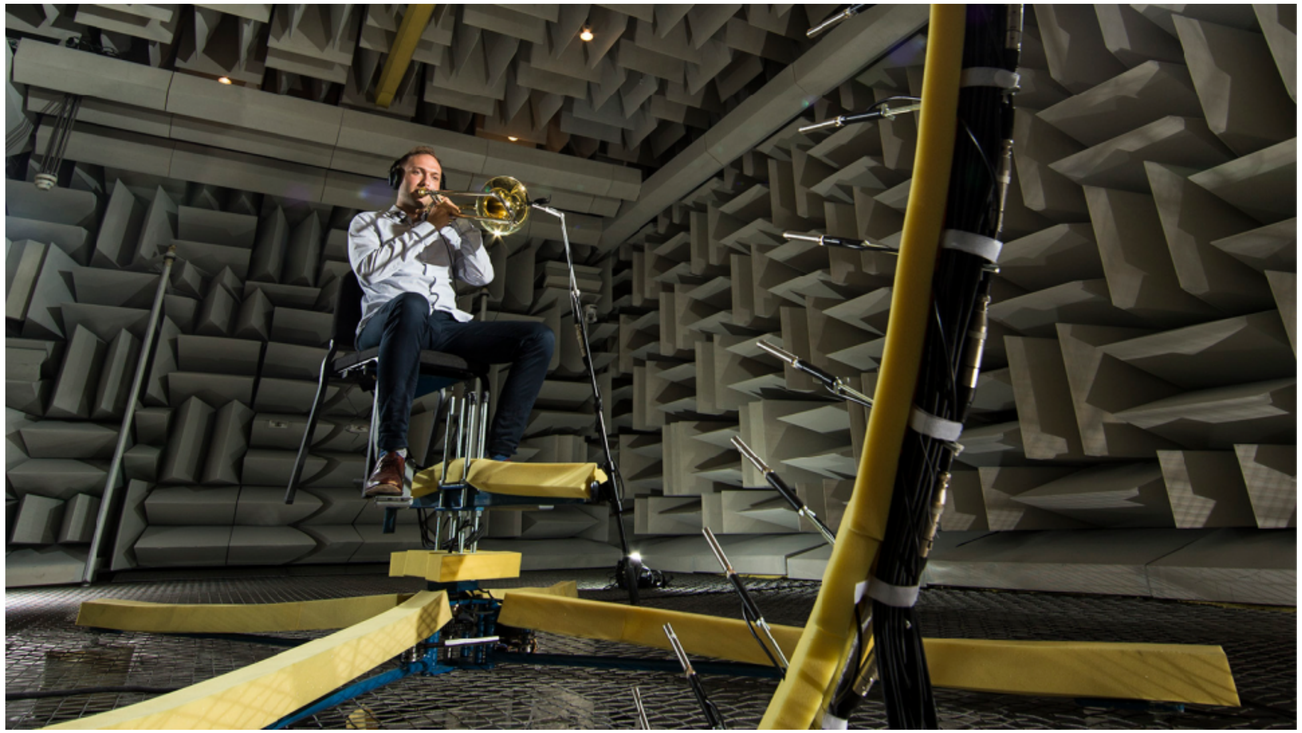




$$(a + b)^3$$
$$= a^3 + 3a^2b + 3ab^2 + b^3$$

Unfamiliar, seems opaque / magical

FP lets us focus on things in isolation



Pieces seamlessly interact & build

???

```
function processEntriesImperative (entries) {
    const csvCopy = entries.slice()

    csvCopy.sort(function (a, b) {
        if (a['Date Created'] === b['Date Created']) return 0
        if (a['Date Created'] > b['Date Created']) return -1
        if (a['Date Created'] < b['Date Created']) return 1
    })

    const seenAlready = {}

    const finalArray = []

    for (let i = 0; i < csvCopy.length; i++) {
        if (!seenAlready[csvCopy[i]['Your Name']]) {
            seenAlready[csvCopy[i]['Your Name']] = true
            finalArray.push(csvCopy[i])
        }
    }

    return finalArray
}
```

Is this descending or ascending?

What's this for?

Are there any bugs? How sure are you?

```
const {pipe, sort, descend, prop, uniqBy} = require('ramda')

const processEntriesFunctional = pipe(
    sort(descend(prop('Date Created'))),
    uniqBy(prop('Your Name'))
)
```

What about this?
What does it do?
Are there any bugs?


5 minute break!