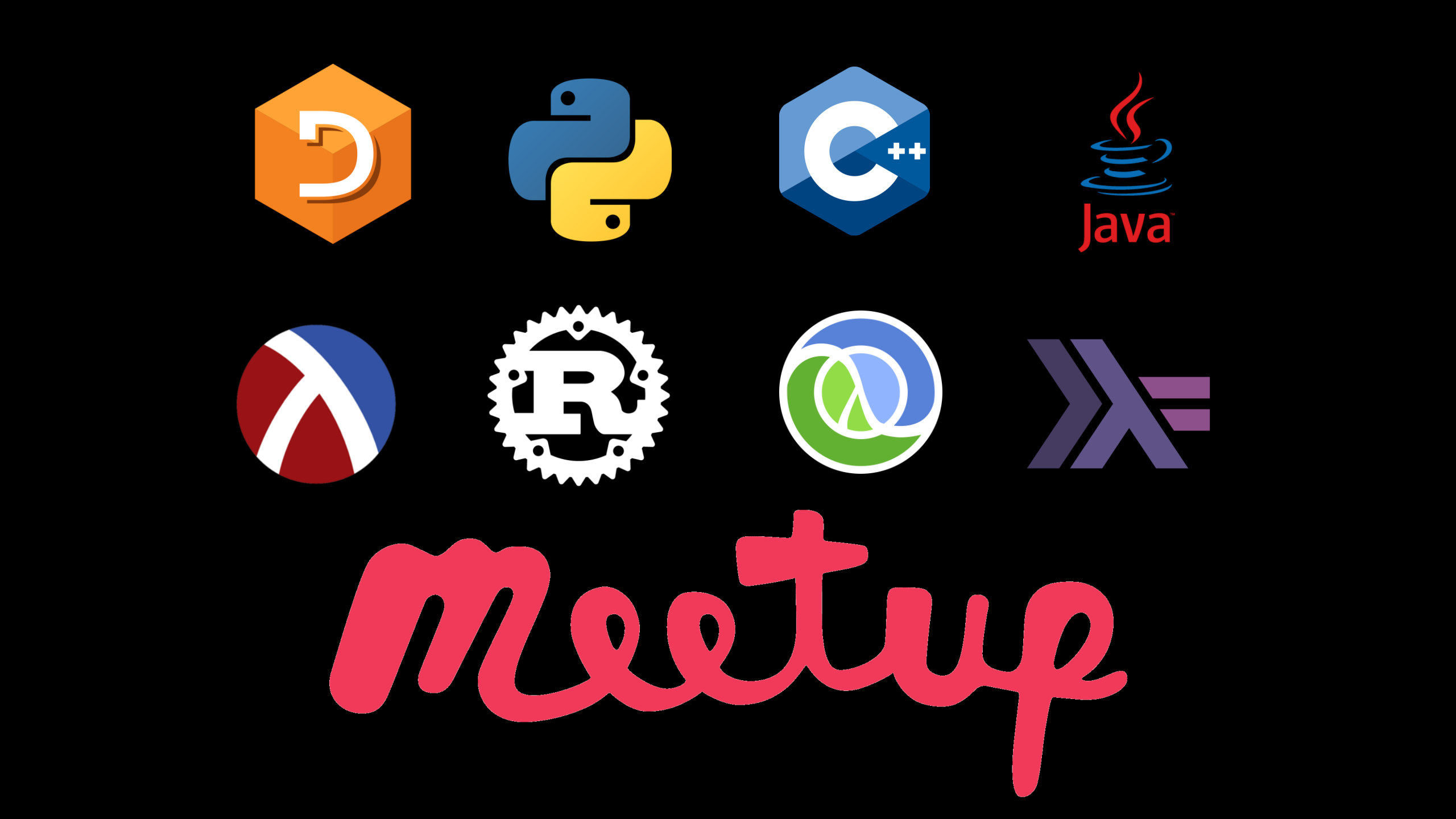

Java
meetup

Friendly Environment Policy

Berlin Code of Conduct

Structure and Interpretation of Computer Programs

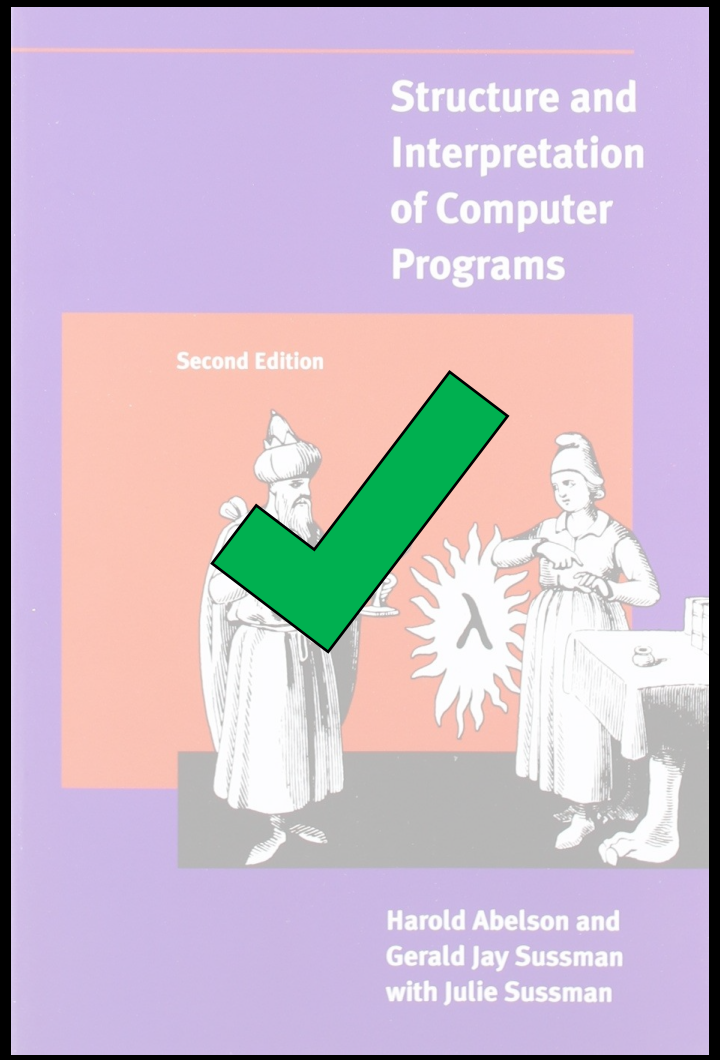
Second Edition

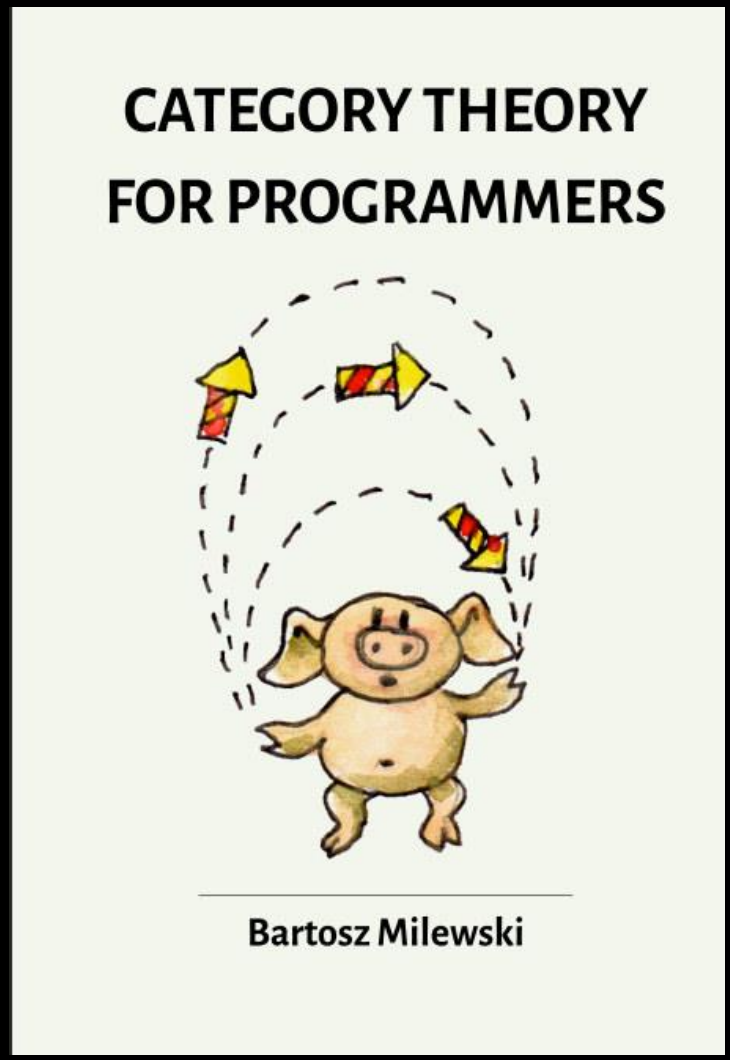
Harold Abelson and
Gerald Jay Sussman
with Julie Sussman

Structure and Interpretation of Computer Programs

Second Edition

Harold Abelson and
Gerald Jay Sussman
with Julie Sussman

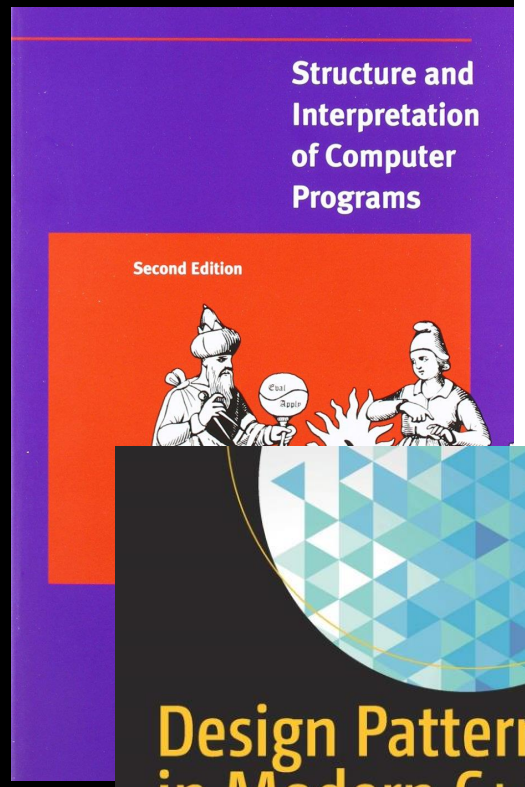

Structure and Interpretation of Computer Programs
Second Edition
Harold Abelson and Gerald Jay Sussman with Julie Sussman

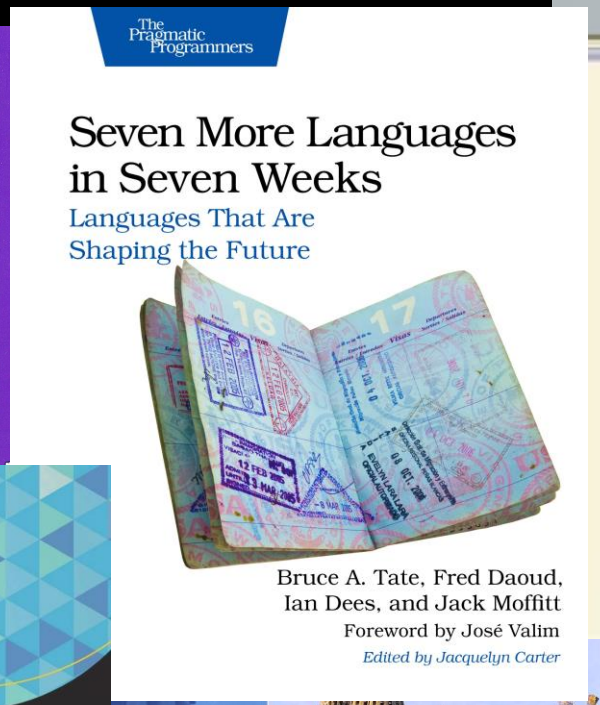

CATEGORY THEORY FOR PROGRAMMERS
Bartosz Milewski

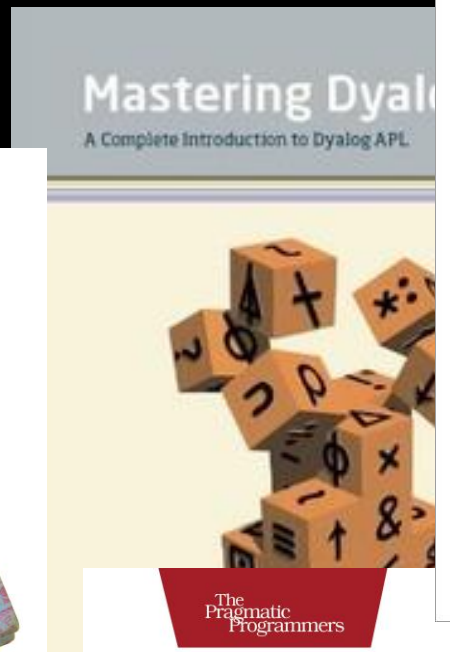
Why CTfP?

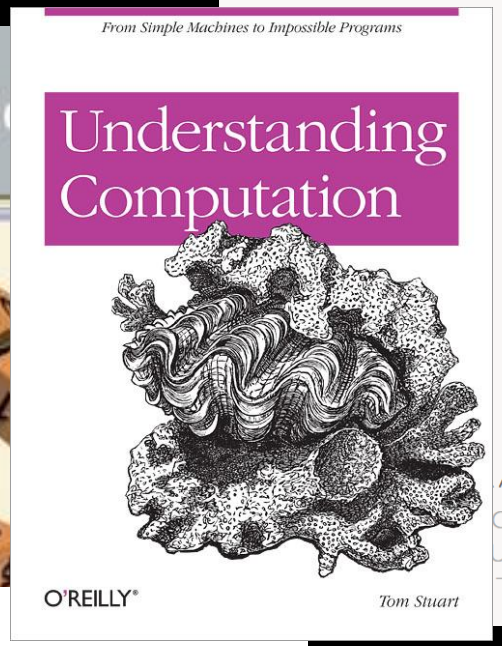
- We voted

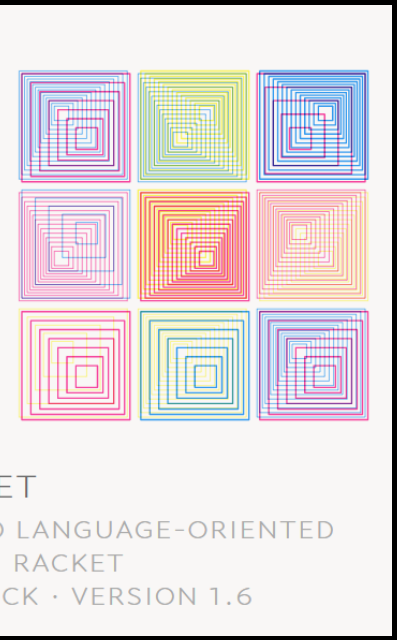
Books to read:

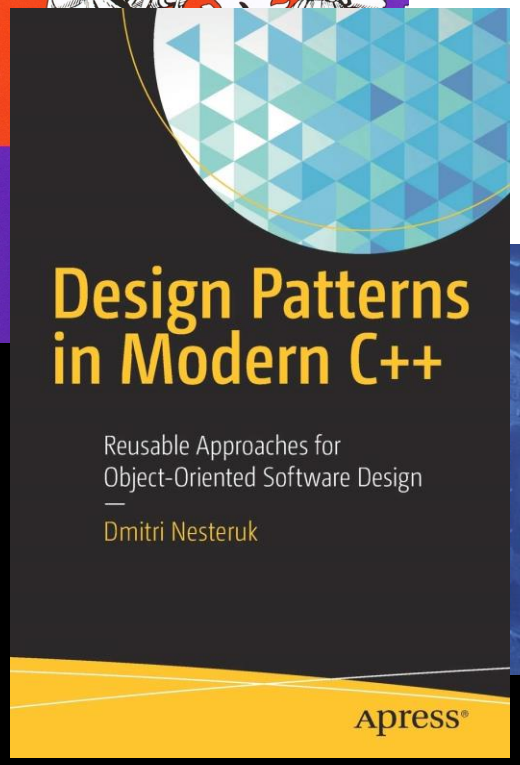


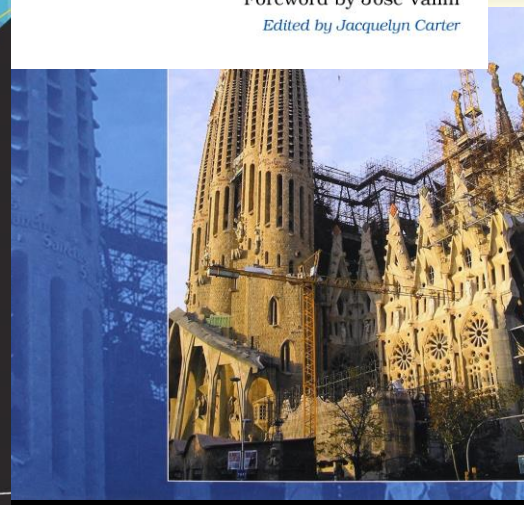


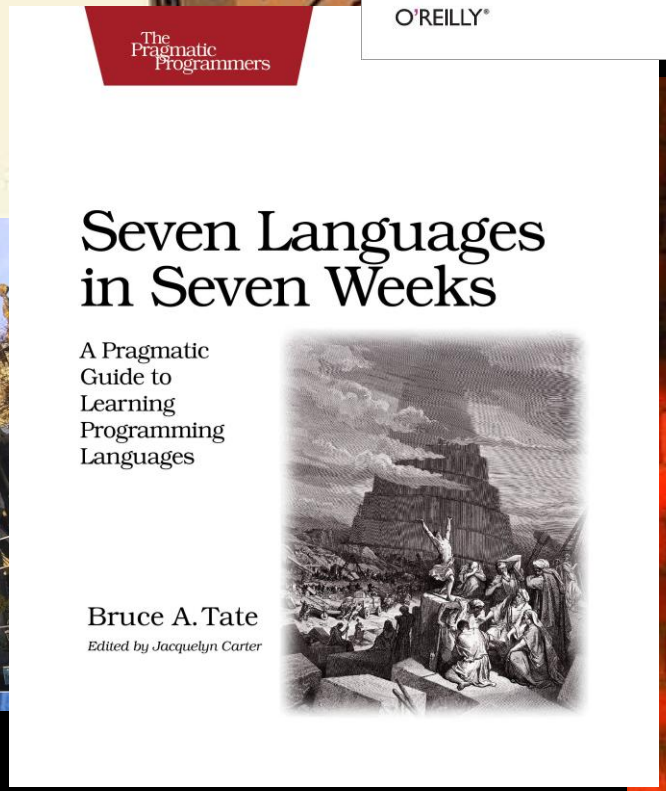


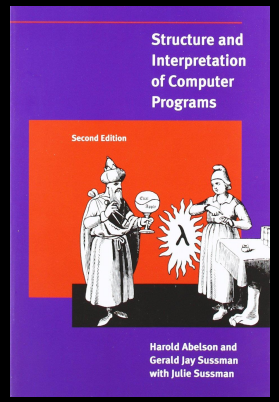


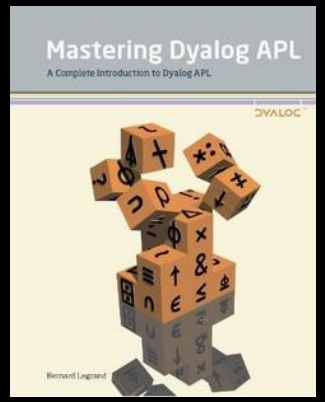


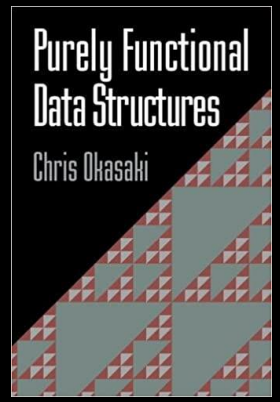


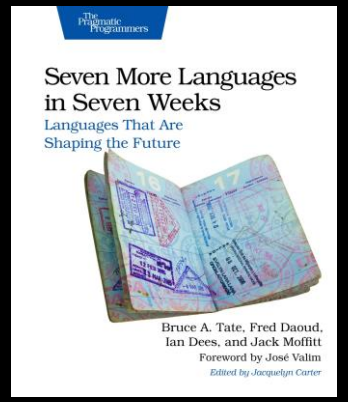


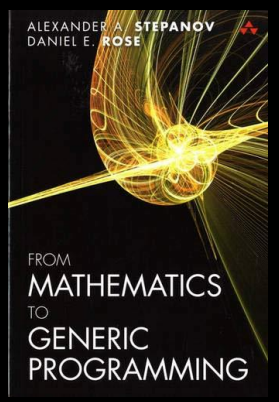


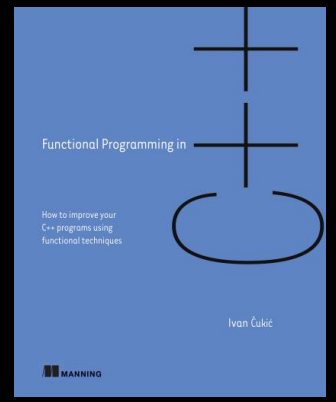

Structure and Interpretation of Computer Programs
Second Edition
Harold Abelson and Gerald Jay Sussman with Julie Sussman

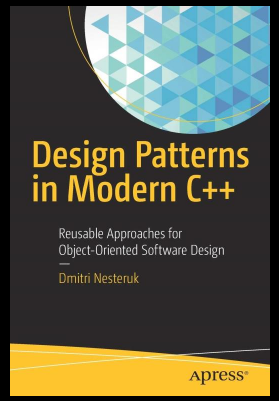

Mastering Dyalog APL
A Complete Introduction to Dyalog APL
DYALOG
Bernard Legrand

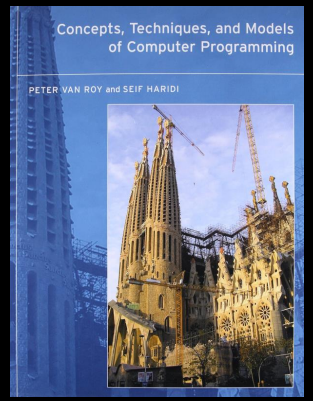

Purely Functional Data Structures
Chris Okasaki

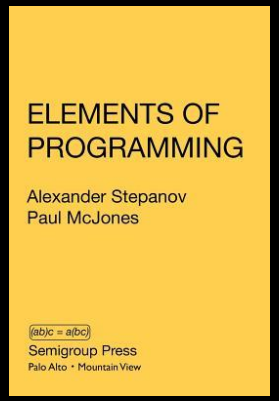

The Pragmatic Programmers
Seven More Languages in Seven Weeks
Languages That Are Shaping the Future
Bruce A. Tate, Fred Daoud, Ian Dees, and Jack Moffitt
Foreword by José Valim
Edited by Jacquelyn Carter

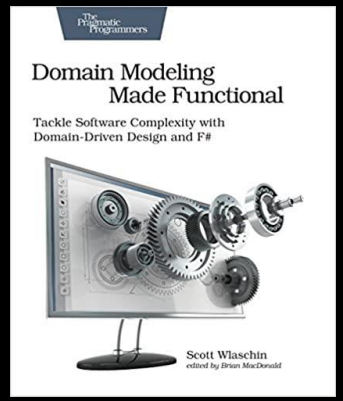

ALEXANDER A. STEPANOV
DANIEL E. ROSE
FROM MATHEMATICS TO GENERIC PROGRAMMING


Functional Programming in C++
How to improve your C++ programs using functional techniques
Ivan Čukić
MANNING

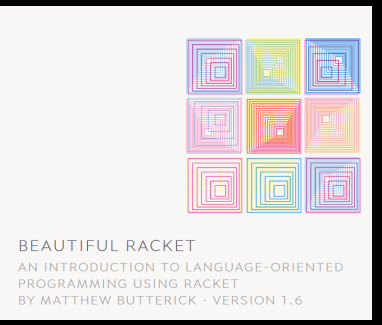

Design Patterns in Modern C++
Reusable Approaches for Object-Oriented Software Design
Dmitri Nesteruk
Apress

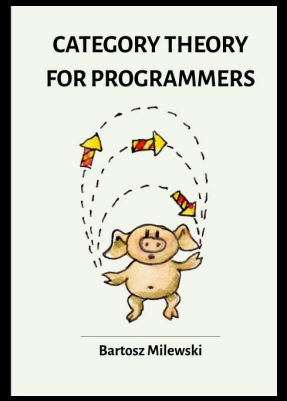

Concepts, Techniques, and Models of Computer Programming
PETER VAN ROY and SEIF HARIDI

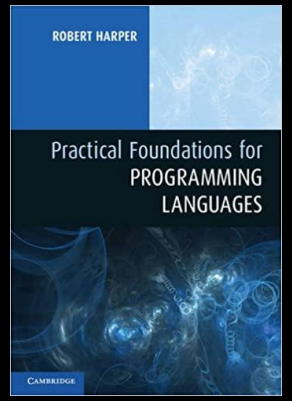

ELEMENTS OF PROGRAMMING
Alexander Stepanov
Paul McJones
(ab)c = a(bc)
Semigroup Press
Palo Alto · Mountain View

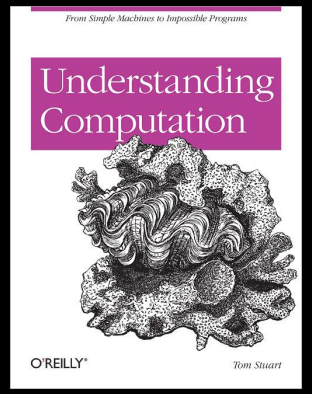

The Pragmatic Programmers
Domain Modeling Made Functional
Tackle Software Complexity with Domain-Driven Design and F#
Scott Wlaschin
edited by Brian MacDonald

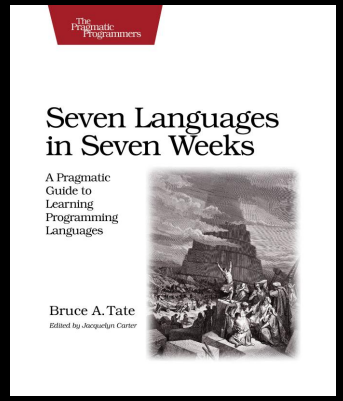

Learn You Some Erlang for Great Good!
A Beginner's Guide
Fred Hébert

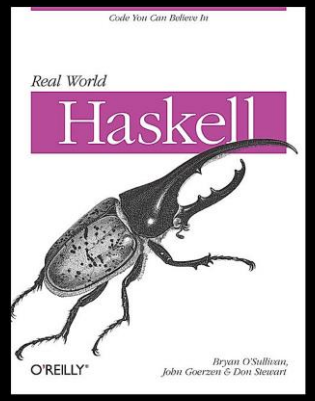

BEAUTIFUL RACKET
AN INTRODUCTION TO LANGUAGE-ORIENTED PROGRAMMING USING RACKET
BY MATTHEW BUTTERICK · VERSION 1.6

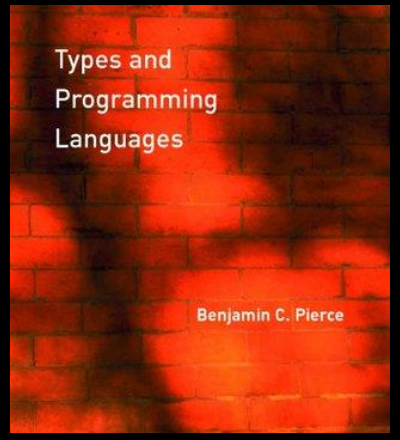

CATEGORY THEORY FOR PROGRAMMERS
Bartosz Milewski

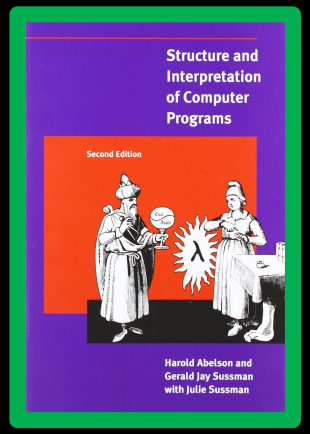

ROBERT HARPER
Practical Foundations for PROGRAMMING LANGUAGES
CAMBRIDGE

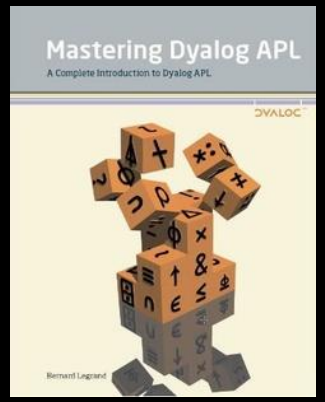

From Simple Machines to Impossible Programs
Understanding Computation
O'REILLY
Tom Stuart

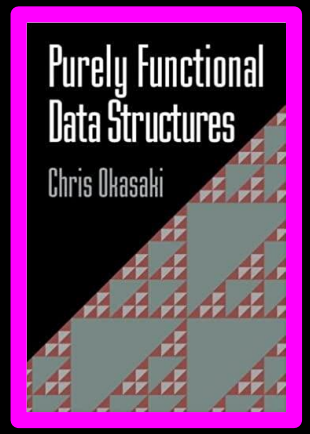

The Pragmatic Programmers
Seven Languages in Seven Weeks
A Pragmatic Guide to Learning Programming Languages
Bruce A. Tate
Edited by Jacquelyn Carter

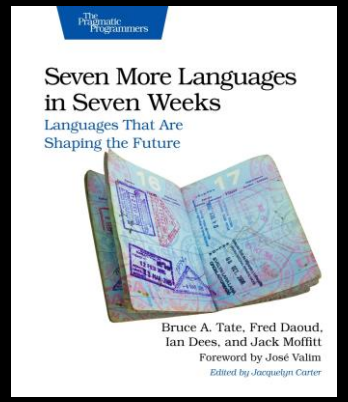

Code You Can Believe In
Real World Haskell
O'REILLY
Bryan O'Sullivan, John Goerzen & Don Stewart

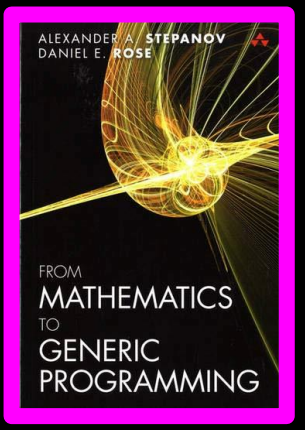

Types and Programming Languages
Benjamin C. Pierce

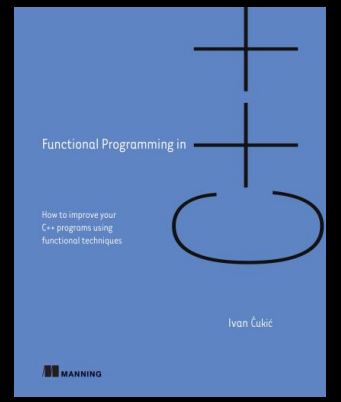

Structure and Interpretation of Computer Programs
Second Edition
Harold Abelson and Gerald Jay Sussman with Julie Sussman

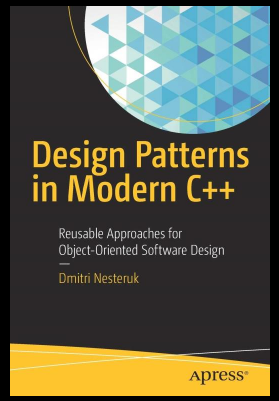

Mastering Dyalog APL
A Complete Introduction to Dyalog APL
DYALOG
Bernard Legrand

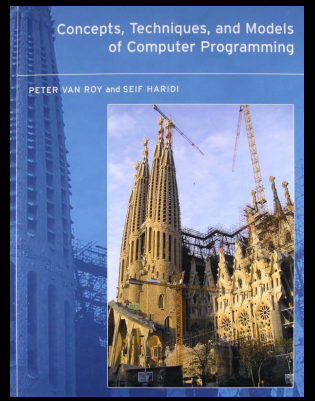

Purely Functional Data Structures
Chris Okasaki

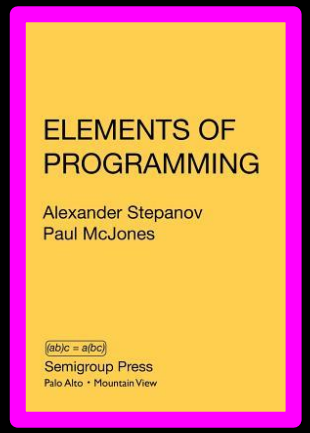

The Pragmatic Programmers
Seven More Languages in Seven Weeks
Languages That Are Shaping the Future
Bruce A. Tate, Fred Daoud, Ian Dees, and Jack Moffitt
Foreword by José Valim
Edited by Jacquelyn Carter

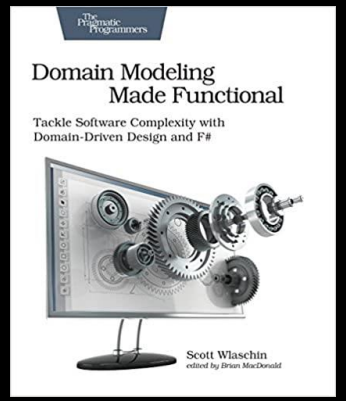

ALEXANDER A. STEPANOV
DANIEL E. ROSE
FROM MATHEMATICS TO GENERIC PROGRAMMING


Functional Programming in C++
How to improve your C++ programs using functional techniques
Ivan Čukić
MANNING

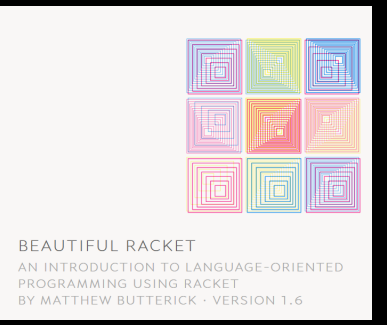

Design Patterns in Modern C++
Reusable Approaches for Object-Oriented Software Design
Dmitri Nesteruk
Apress

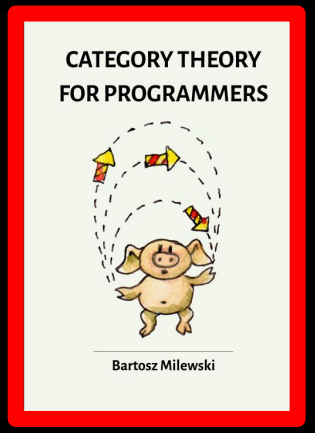

Concepts, Techniques, and Models of Computer Programming
PETER VAN ROY and SEIF HARIDI

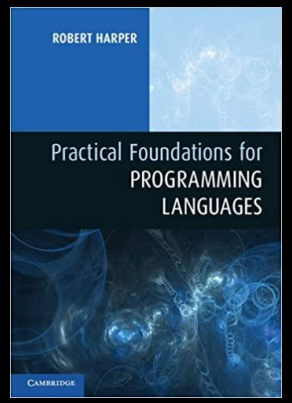

ELEMENTS OF PROGRAMMING
Alexander Stepanov
Paul McJones
(ab)c = a(bc)
Semigroup Press
Palo Alto · Mountain View

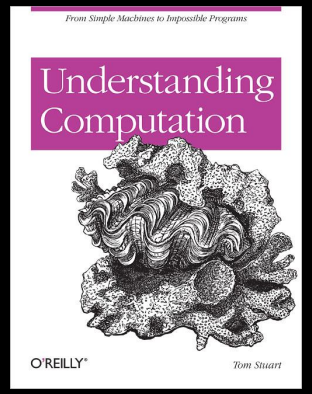

The Pragmatic Programmers
Domain Modeling Made Functional
Tackle Software Complexity with Domain-Driven Design and F#
Scott Wlaschin
edited by Brian MacDonald

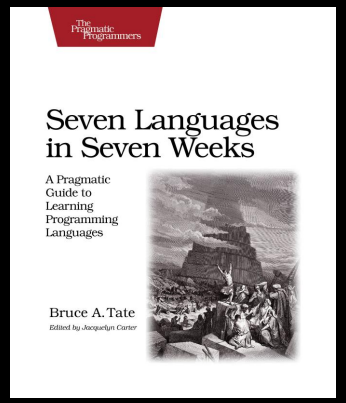

Learn You Some Erlang for Great Good!
A Beginner's Guide
Fred Hébert

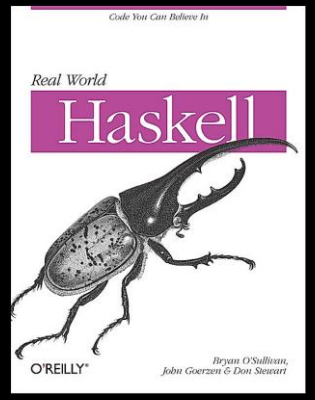

BEAUTIFUL RACKET
AN INTRODUCTION TO LANGUAGE-ORIENTED PROGRAMMING USING RACKET
BY MATTHEW BUTTERICK · VERSION 1.6

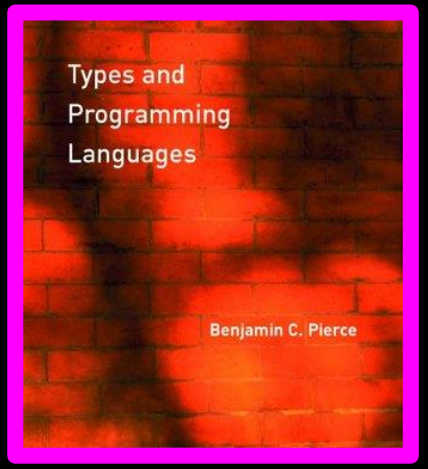

CATEGORY THEORY FOR PROGRAMMERS
Bartosz Milewski


ROBERT HARPER
Practical Foundations for PROGRAMMING LANGUAGES
CAMBRIDGE


From Simple Machines to Impossible Programs
Understanding Computation
O'REILLY
Tom Stuart


The Pragmatic Programmers
Seven Languages in Seven Weeks
A Pragmatic Guide to Learning Programming Languages
Bruce A. Tate
Edited by Jacquelyn Carter


Code You Can Believe In
Real World Haskell
O'REILLY
Bryan O'Sullivan, John Goerzen & Don Stewart


Types and Programming Languages
Benjamin C. Pierce

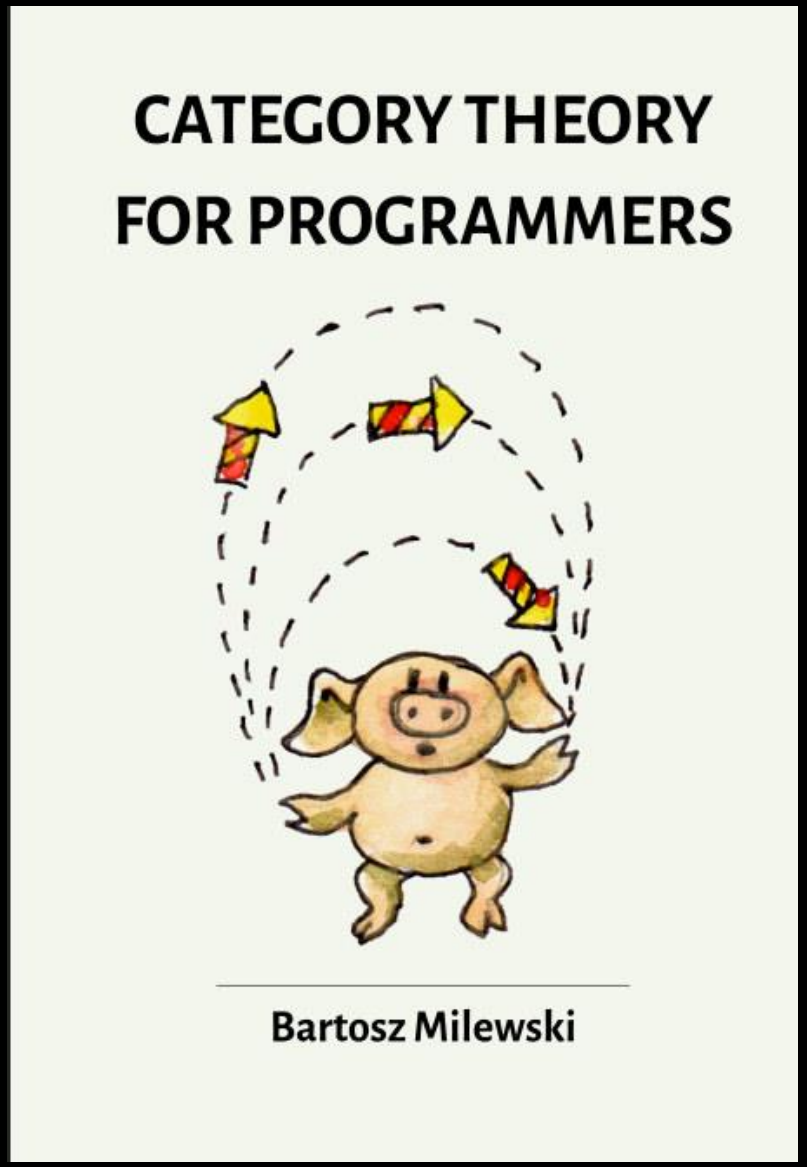
Goal of PLVM

- Work through books on programming languages together
- Grow knowledge on PLs and PL:
  - principles
  - design
  - implementation
- This ultimately will lead to ability to write **code** that is more:
  - readable & expressive
  - maintainable & scalable
  - beautiful & idiomatic

Format of PLVM

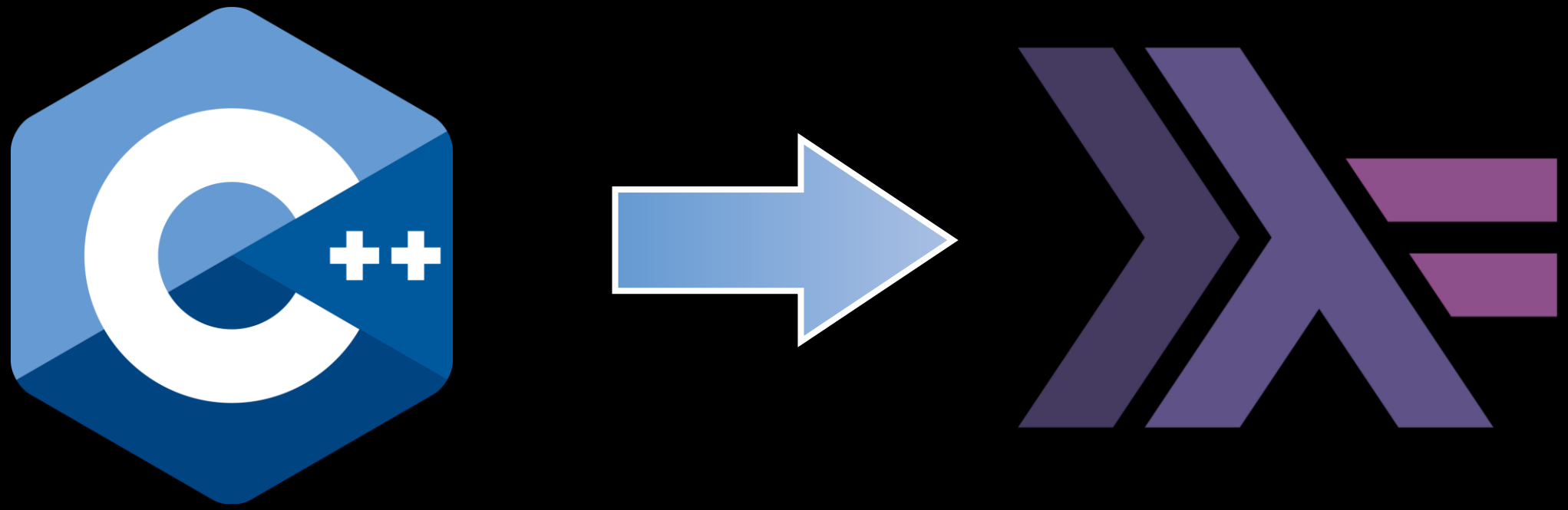
- 1.5 hour meeting once a week
- ~15 min presentation at beginning
- ~45 of discussion afterwards

- I will pre-record presentation and upload to YouTube for those unable to attend



Category Theory for Programmers

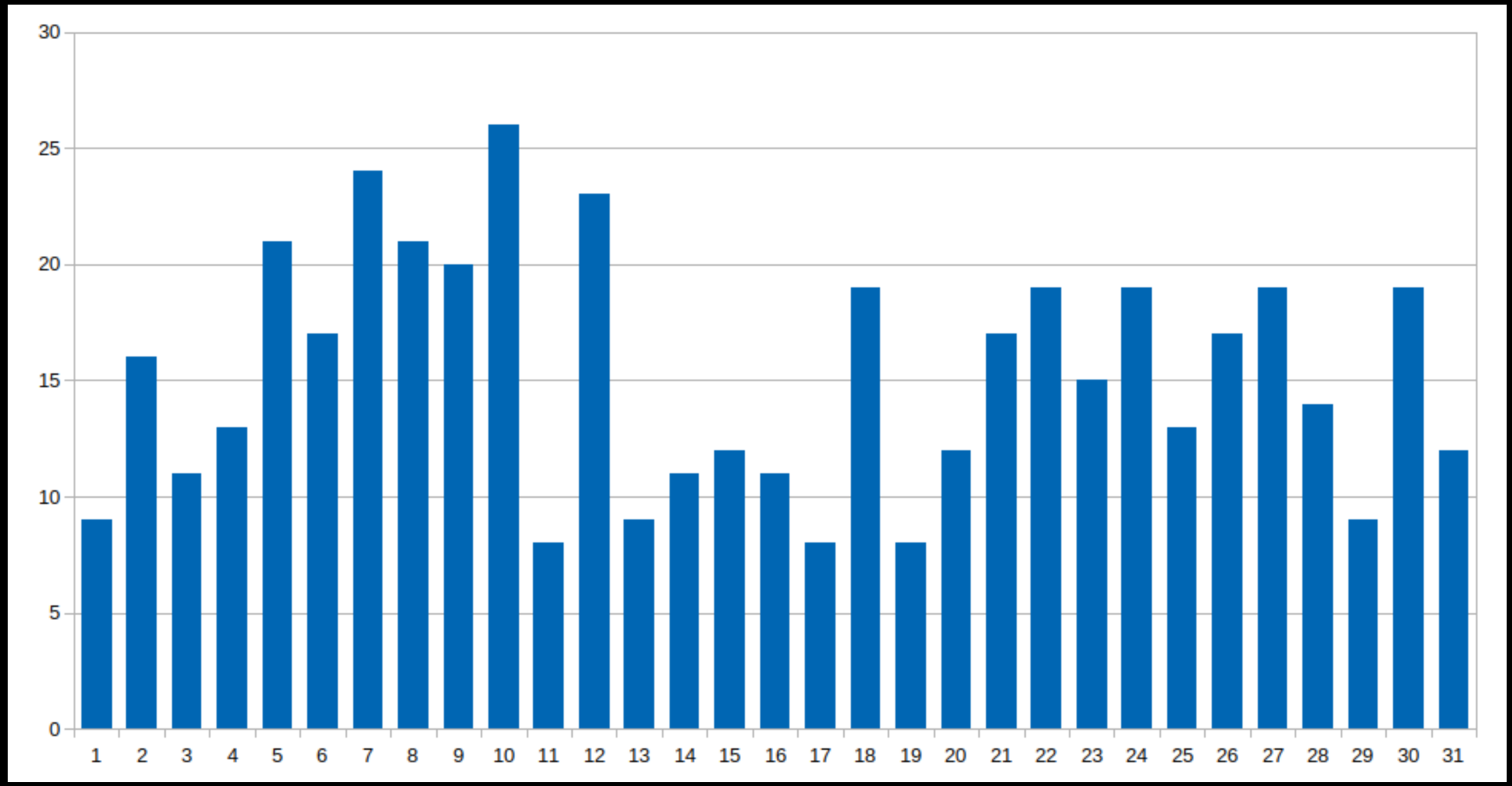
Chapter 1: Category



Talks

- **Bartosz Milewski** talks
  - Functional Patterns in C++, 1. Functors, 2012
  - Functional Patterns in C++, 2. Currying, Applicative, 2012
  - Functional Patterns in C++, 3. Async API, Monoid, Monad, 2012
  - Compile-Time/Run-Time Functional Programming in C++, **Bartosz Milewski**, **Eric Niebler**, BoostCon, 2012
  - Haskell -- The Pseudocode Language for C++ Template Metaprogramming (Part 1), BoostCon, 2013
  - Haskell -- The Pseudocode Language for C++ Template Metaprogramming (Part 2), BoostCon, 2013
  - Re-discovering monads in C++, C++ User Group Novosibirsk, 2014
  - Functional techniques in C++, CDays14, 2014
  - Categories for the Working C++ Programmer, C++ Russia, 2015
  - Monads for C++, itCppCon17, 2017

https://github.com/graninas/cpp_functional_programming


30
25
20
15
10
5
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

If you're an experienced programmer, you might be asking yourself: I've been coding for so long without worrying about category theory or functional methods, so what's changed? Surely you can't help but notice that there's been a steady stream of new functional features invading imperative languages. Even Java, the bastion of object-oriented programming, let the lambdas in. C++ has recently been evolving at a frantic pace — a new standard every few years — trying to catch up with the changing world. All this activity is in preparation for a disruptive change or, as we physicists call it, a phase transition. If you keep heating water, it will eventually start boiling. We are now in the position of a frog that must decide if it should continue swimming in increasingly hot water, or start looking for some alternatives.

Ben Deane

Identifying Monoids: Exploiting Compositional Structure in Code

PAUSE: WHY ARE MONOIDS IMPORTANT?

Why is "abstract mathematics" important for programming?

"Being abstract is something profoundly different from being vague... the purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."

-- EWD

Category Theory is to **Programming** what **Chemistry** is to **Baking** …

"whether or not you know about monoids doesn't change the fact that they are in your code."

Ben Deane

Identifying Monoids, C++Now 2019

https://www.youtube.com/watch?v=INnattuluiM

A CATEGORY is an embarrassingly simple concept. A category consists of *objects* and *arrows* that go between them.

A CATEGORY is an embarrassingly simple concept. A category consists of *objects* and *arrows* that go between them.

Think of arrows, which are also called *morphisms*, as functions. You have a function f that takes an argument of type A and returns a B. You have another function g that takes a B and returns a C. You can compose them by passing the result of f to g. You have just defined a new function that takes an A and returns a C.

```
B f(A a);
C g(B b);

C g_after_f(A a) {
    return g(f(a));
}
```

```
f :: A -> B
```

Similarly:

```
g :: B -> C
```

Their composition is:

```
g . f
```

```
f :: A -> B
```

Similarly:

```
g :: B -> C
```

$$g \circ f$$

Their composition is:

```
g . f
```

A -> C

```
Prelude> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

```
Prelude> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

```
bluebird :: (b -> c) -> (a -> b) -> a -> c
```

B combinator or function composition

2. Implement the composition function in your favorite language. It takes two functions as arguments and returns a function that is their composition.

$$g \circ f$$

1. Composition is associative. If you have three morphisms, f, g, and h, that can be composed (that is, their objects match end-to-end), you don't need parentheses to compose them. In math notation this is expressed as:

$$h \circ (g \circ f) = (h \circ g) \circ f = h \circ g \circ f$$

   In (pseudo) Haskell:

```
f :: A -> B
g :: B -> C
h :: C -> D
h . (g . f) == (h . g) . f == h . g . f
```

2. For every object A there is an arrow which is a unit of composition. This arrow loops from the object to itself. Being a unit of composition means that, when composed with any arrow that either starts at A or ends at A, respectively, it gives back the same arrow. The unit arrow for object A is called $\mathbf{id}_A$ (*identity* on A). In math notation, if f goes from A to B then

$$f \circ \mathbf{id}_A = f$$

and

$$\mathbf{id}_B \circ f = f$$

```
template<class T> T id(T x) { return x; }
```

```haskell
id :: a -> a
id x = x
```

1. Implement, as best as you can, the identity function in your favorite language (or the second favorite, if your favorite language happens to be Haskell).

This process of hierarchical decomposition and recomposition is not imposed on us by computers. It reflects the limitations of the human mind. Our brains can only deal with a small number of concepts at a time. One of the most cited papers in psychology, The Magical Number Seven, Plus or Minus Two[1], postulated that we can only keep 7 ± 2 "chunks" of information in our minds. The details of our understanding of the human short-term memory might be changing, but we know for sure that it's limited. The bottom line is that we are unable to deal with the soup of objects or the spaghetti of code. We need structure not because well-structured programs are pleasant to look at, but because otherwise our brains can't process them efficiently. We often describe some piece of code as elegant or beautiful, but what we really mean is that it's easy to process by our limited human minds. Elegant code creates chunks that are just the right size and come in just the right number for our mental digestive system to assimilate them.

Making Things Easy

- Bring to hand by installing
  - getting approved for use
- Become familiar by learning, trying
- But mental capability?
  - not going to move very far
  - make things near by simplifying them

```cpp
std::reverse(v.begin(), v.end());
std::transform(v.begin(), v.end(), v.begin(),
    [](auto e) { return e + 1; });
```

```
map (1+) . reverse
```

```cpp
std::reverse(v.begin(), v.end());
std::transform(v.begin(), v.end(), v.begin(),
    [](auto e) { return e + 1; });
```

```haskell
map (1+) . reverse
```

```cpp
std::reverse(v.begin(), v.end());
std::transform(v.begin(), v.end(), v.begin(),
    [](auto e) { return e + 1; });
```

```haskell
map (1+) . reverse
```

```
1+ϕ
```

3. Write a program that tries to test that your composition function respects identity.

```
      test ← {(αα∘⊢≡αα)ω}
      ϕ test ⍳5
1

      1∘ϕ test ⍳5
1
```














Bartosz Milewski

19.8K subscribers

SUBSCRIBE

HOME | VIDEOS | PLAYLISTS | COMMUNITY | CHANNELS | ABOUT

Uploads PLAY ALL

SORT BY

Category Theory III 7.2, Coends

4.1K views • 2 years ago

Category Theory III 7.1, Natural transformations as...

2.6K views • 2 years ago

Category Theory III 6.2, Ends

2.3K views • 2 years ago

Category Theory III 6.1, Profunctors

2.5K views • 2 years ago

Category Theory III 5.2, Lawvere Theories

2.3K views • 2 years ago

Category Theory III 5.1, Eilenberg Moore and Lawvere

2.5K views • 2 years ago

Category Theory III 4.2, Monad algebras part 3

1.7K views • 2 years ago

Category Theory III 4.1, Monad algebras part 2

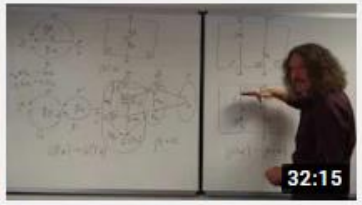
1.8K views • 2 years ago

Category Theory III 3.2, Monad Algebras

2.6K views • 2 years ago

Category Theory III 3.1, Adjunctions and monads

2.8K views • 2 years ago



Category Theory III 2.2, String Diagrams part 2

2.8K views • 2 years ago

Category Theory III 2.1: String Diagrams part 1

3.9K views • 2 years ago

Category Theory III 1.2: Overview part 2

2.8K views • 2 years ago

Category Theory III 1.1: Overview part 1

8.6K views • 2 years ago

Category Theory II 9.2: Lenses categorically

3.8K views • 3 years ago

Category Theory II 9.1: Lenses

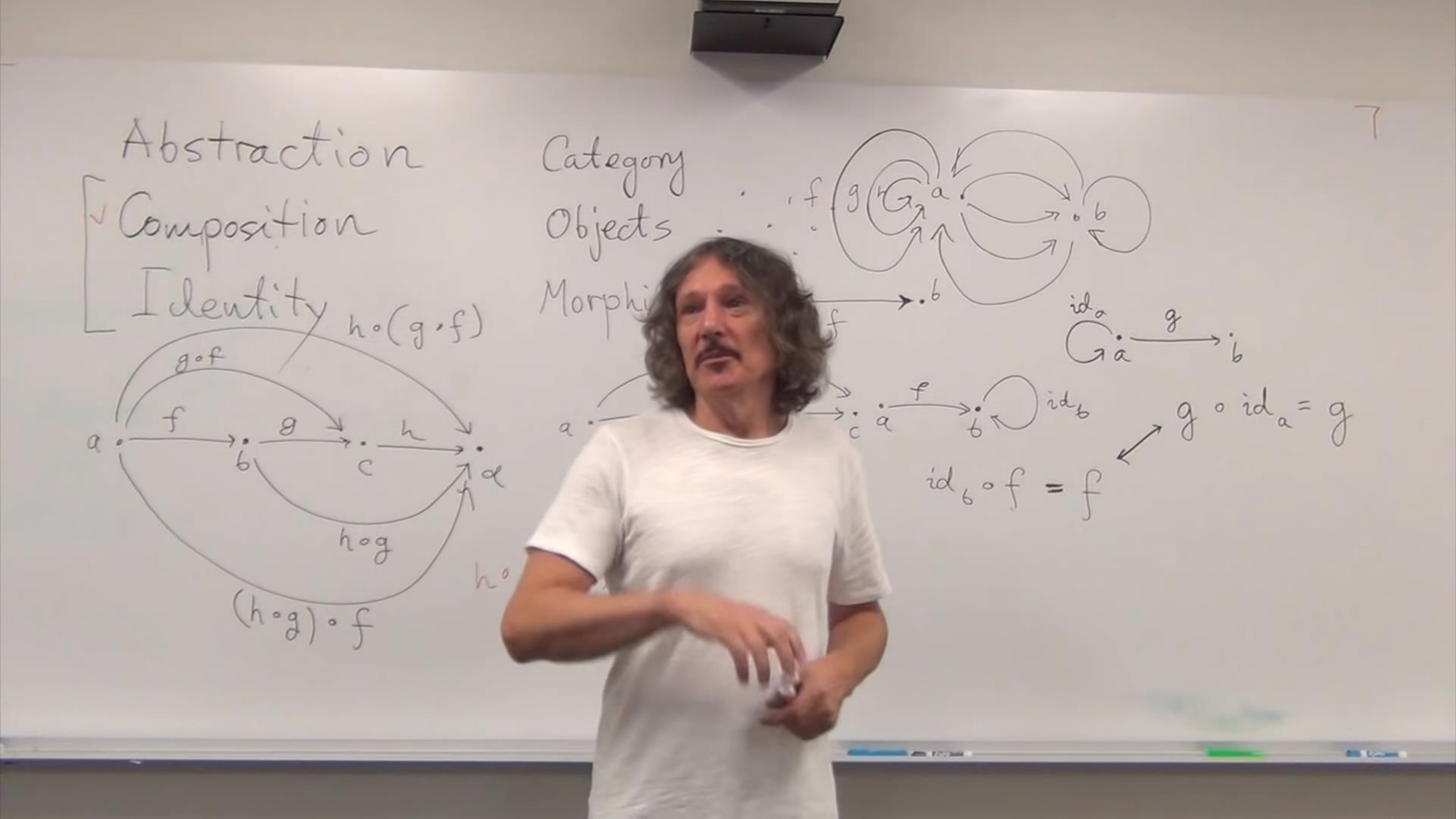
4.9K views • 3 years ago

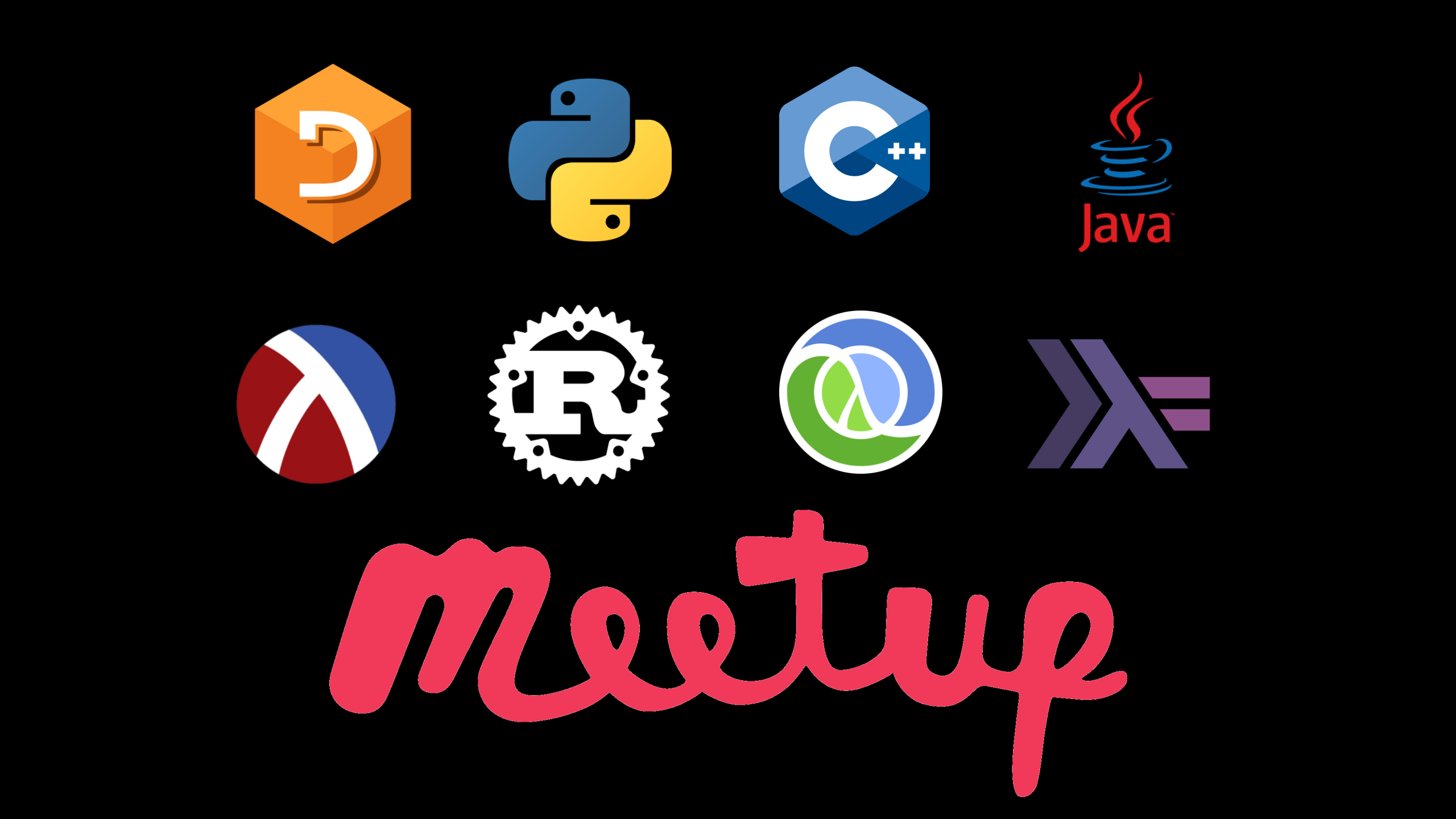
Category Theory II 8.2: Catamorphisms and...

4.4K views • 3 years ago

Category Theory II 8.1: F-Algebras, Lambek's lemma

5.7K views • 3 years ago


Abstraction
Composition
Identity
Category
Objects
Morphi
h∘(g∘f)
g∘f
f
g
h
a
b
c
d
h∘g
(h∘g)∘f
id_a
id_b
g ∘ id_a = g
id_b ∘ f = f


Java
meetup