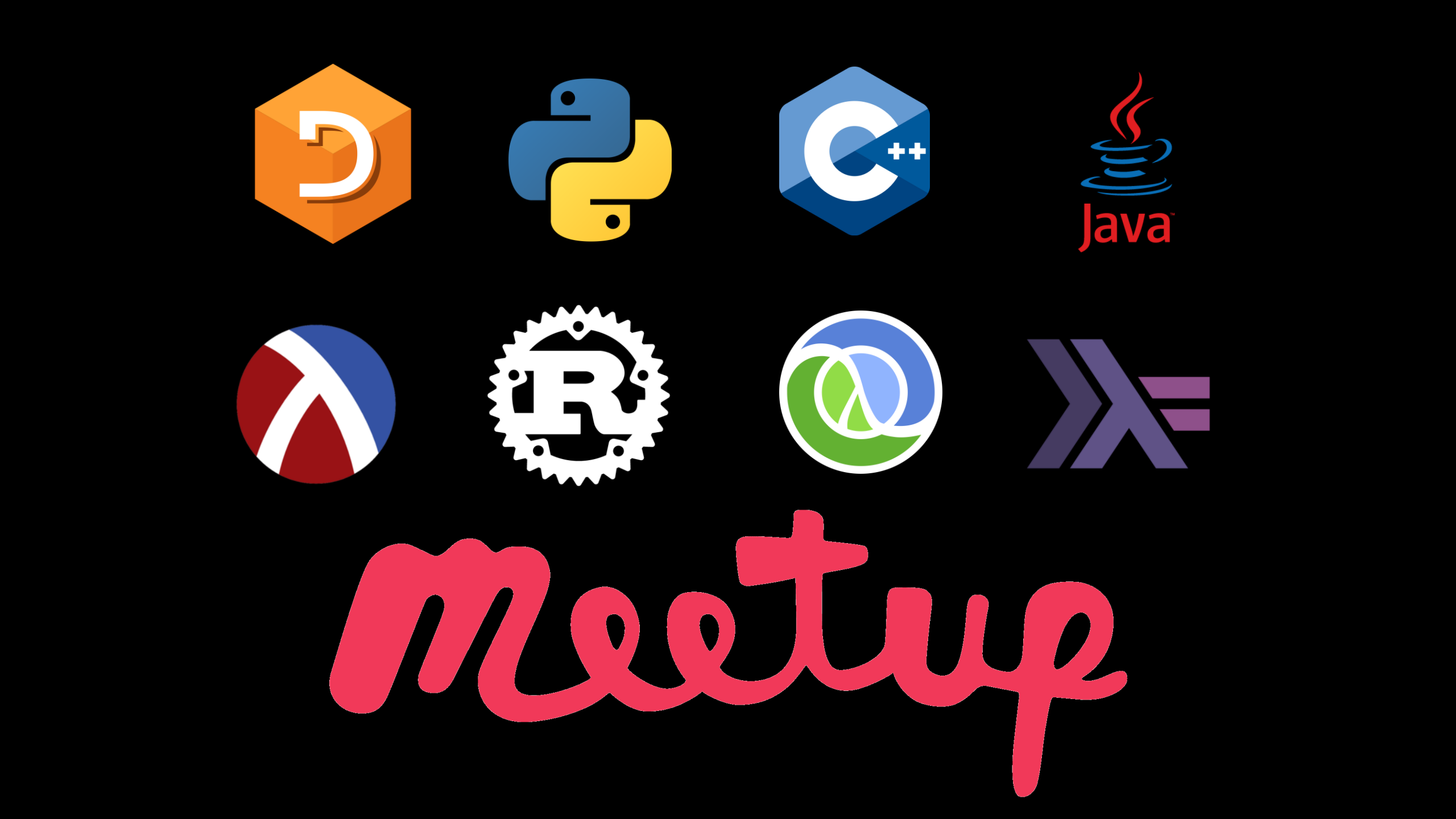
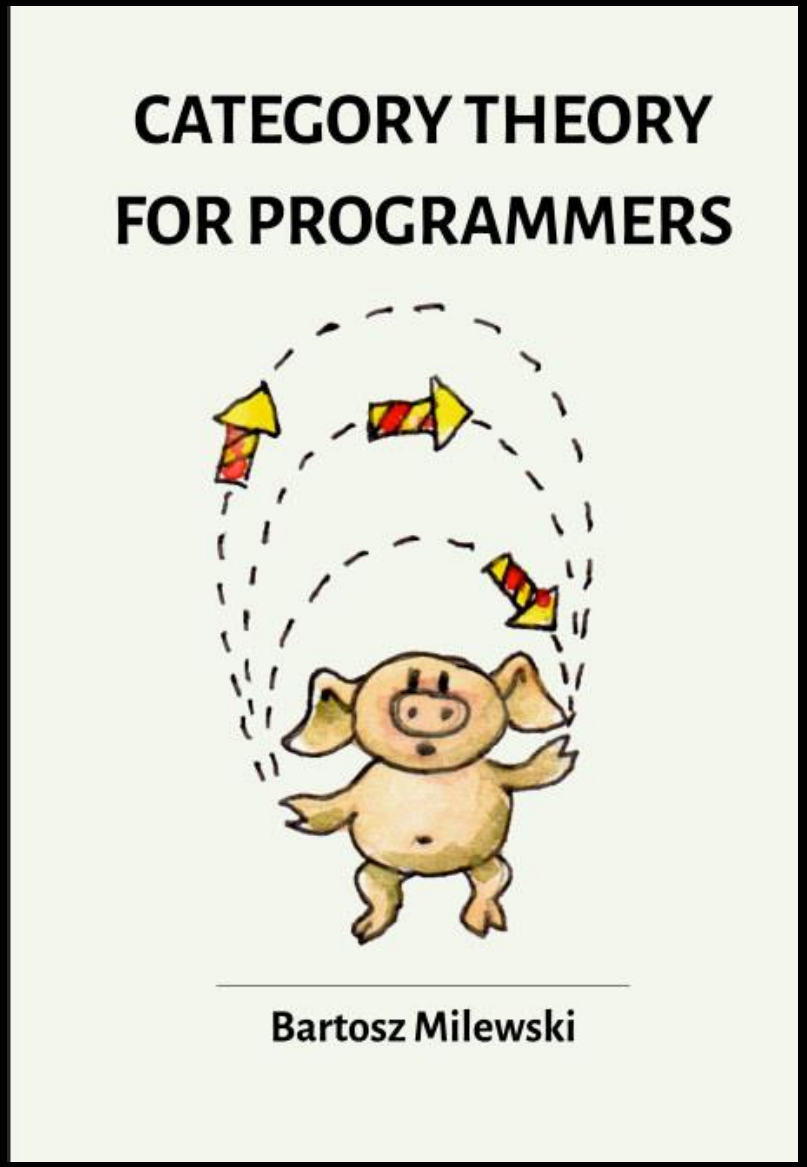

Java
meetup

Friendly Environment Policy

Berlin Code of Conduct



Category Theory for Programmers

Chapter 4:

Kleisli Categories

```
string logger;

bool negate(bool b) {
    logger += "Not so! ";
    return !b;
}
```

```
pair<bool, string> negate(bool b, string logger) {
    return make_pair(!b, logger + "Not so! ");
}
```

```
pair<bool, string> negate(bool b) {
    return make_pair(!b, "Not so! ");
}
```

```
Writer<vector<string>> process(string s) {
    auto p1 = toUpper(s);
    auto p2 = toWords(p1.first);
    return make_pair(p2.first, p1.second + p2.second);
}
```

```haskell
type Writer a = (a, String )

( >=> ) :: (a -> Writer b) -> (b -> Writer c) -> (a -> Writer c)
m1 >=> m2 = \ x ->
    let (y, s1) = m1 x
        (z, s2) = m2 y
    in (z, s1 ++ s2)

return :: a -> Writer a
return x = (x, "")

upCase :: String -> Writer String
upCase s = (map toUpper s, "upCase ")

toWords :: String -> Writer [ String ]
toWords s = (words s, "toWords ")

process :: String -> Writer [ String ]
process = upCase >=> toWords
```

A function that is not defined for all possible values of its argument is called a partial function. It's not really a function in the mathematical sense, so it doesn't fit the standard categorical mold. It can, however, be represented by a function that returns an embellished type `optional`:

```
template<class A> class optional {
    bool _isValid;
    A _value;
public:
    optional()    : _isValid(false) {}
    optional(A v) : _isValid(true), _value(v) {}
    bool isValid() const { return _isValid; }
    A value() const { return _value; }
};
```

For example, here's the implementation of the embellished function `safe_root`:

```
optional<double> safe_root(double x) {
    if (x >= 0) return optional<double>{sqrt(x)};
    else return optional<double>{};
}
```

```
template<class A> class optional {
    bool  isValid;
```

cppreference.com Create account Search

Page Discussion View Edit History

C++ Utilities library std::optional

std::optional

Defined in header <optional>

```
template< class T >
class optional;
```
(since C++17)

The class template `std::optional` manages an *optional* contained value, i.e. a value that may or may not be present.

A common use case for `optional` is the return value of a function that may fail. As opposed to other approaches, such as `std::pair<T,bool>`, `optional` handles expensive-to-construct objects well and is more readable, as the intent is expressed explicitly.

Any instance of `optional<T>` at any given point in time either *contains a value* or *does not contain a value*.

If an `optional<T>` *contains a value*, the value is guaranteed to be allocated as part of the `optional` object footprint, i.e. no dynamic memory allocation ever takes place. Thus, an `optional` object models an object, not a pointer, even though `operator*()` and `operator->()` are defined.

When an object of type optional<T> is contextually converted to bool, the conversion returns `true` if the object *contains a value* and `false` if it *does not contain a value*.

```
}
```

```
template<class A> class optional {
    bool _isValid;
    A _value;
public:
    optional()     : _isValid(false) {}
```

has_value	(public member function)
value	returns the contained value (public member function)
value_or	returns the contained value if available, another value otherwise (public member function)

```
optional<double> safe_root(double x) {
    if (x >= 0) return optional<double>{sqrt(x)};
    else return optional<double>{};
}
```

Here's the challenge:

1. Construct the Kleisli category for partial functions (define composition and identity).
2. Implement the embellished function `safe_reciprocal` that returns a valid reciprocal of its argument, if it's different from zero.
3. Compose the functions `safe_root` and `safe_reciprocal` to implement `safe_root_reciprocal` that calculates `sqrt(1/x)` whenever possible.

```cpp
// Question 1

auto compose(auto f, auto g) {
    return [f, g] (auto x) {
        auto const res = f(x);
        return res.has_value() ? g(res.value()) : std::nullopt;
    };
}
```

```cpp
// From Book (modified)

auto safe_root(double n) -> std::optional<double> {
    return n >= 0 ? std::optional{sqrt(n)} : std::nullopt;
}

// Question 2

auto safe_reciprocal(int n) -> std::optional<double> {
    return n != 0 ? std::optional{1.0 / n} : std::nullopt;
}
```

```cpp
// Question 3

auto safe_root_reciprocal(int n) -> std::optional<double> {
    auto const r = safe_reciprocal(n);
    return r.has_value() ? safe_root(r.value()) : std::nullopt;
}

auto safe_root_reciprocal2(int n) -> std::optional<double> {
    return compose(
        [] (auto x) { return safe_reciprocal(x); },
        [] (auto x) { return safe_root(x); }
    ) (n);
}
```

```cpp
// Question 3

auto safe_root_reciprocal(int n) -> std::optional<double> {
    auto const r = safe_reciprocal(n);
    return r.has_value() ? safe_root(r.value()) : std::nullopt;
}

auto safe_root_reciprocal2(int n) -> std::optional<double> {
    return compose(safe_reciprocal, safe_root)(n);
}
```

```
nullopt      ← 0 ⍬
has_value    ← ⊃
value        ← ⊃⌽
make_optional ← 1,⊢
```

```
nullopt        ← 0 ⍬
has_value      ← ⊃
value          ← ⊃⌽
make_optional  ← 1,⊢

safe_root      ← { ⍵≥0 : make_optional ⍵*.5 ⋄ nullopt }
safe_reciprocal ← { ⍵≠0 : make_optional ÷⍵   ⋄ nullopt }
```

```
nullopt        ← 0 ⍬
has_value      ← ⊃
value          ← ⊃⌽
make_optional  ← 1,⊢

safe_root        ← { ⍵≥0 : make_optional ⍵*.5 ⋄ nullopt }
safe_reciprocal  ← { ⍵≠0 : make_optional ÷⍵   ⋄ nullopt }

safe_root_reciprocal ← {
    r ← safe_reciprocal ⍵
    has_value r : safe_root value r ⋄ nullopt
}
```

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

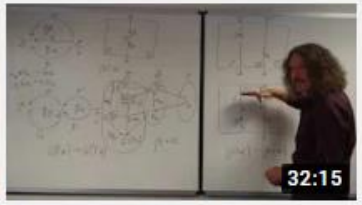
1.8K views • 2 years ago

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

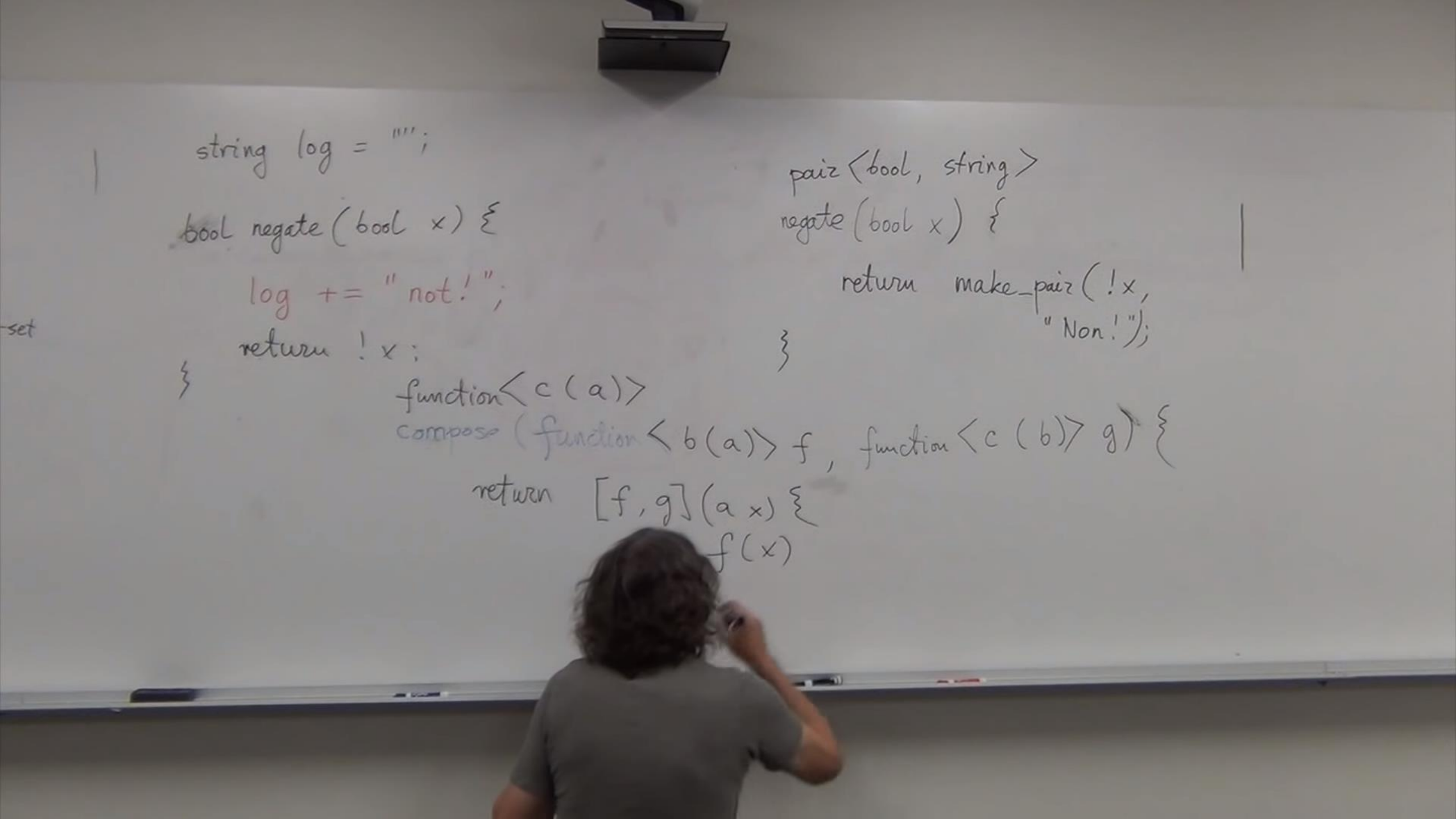
4.9K views • 3 years ago

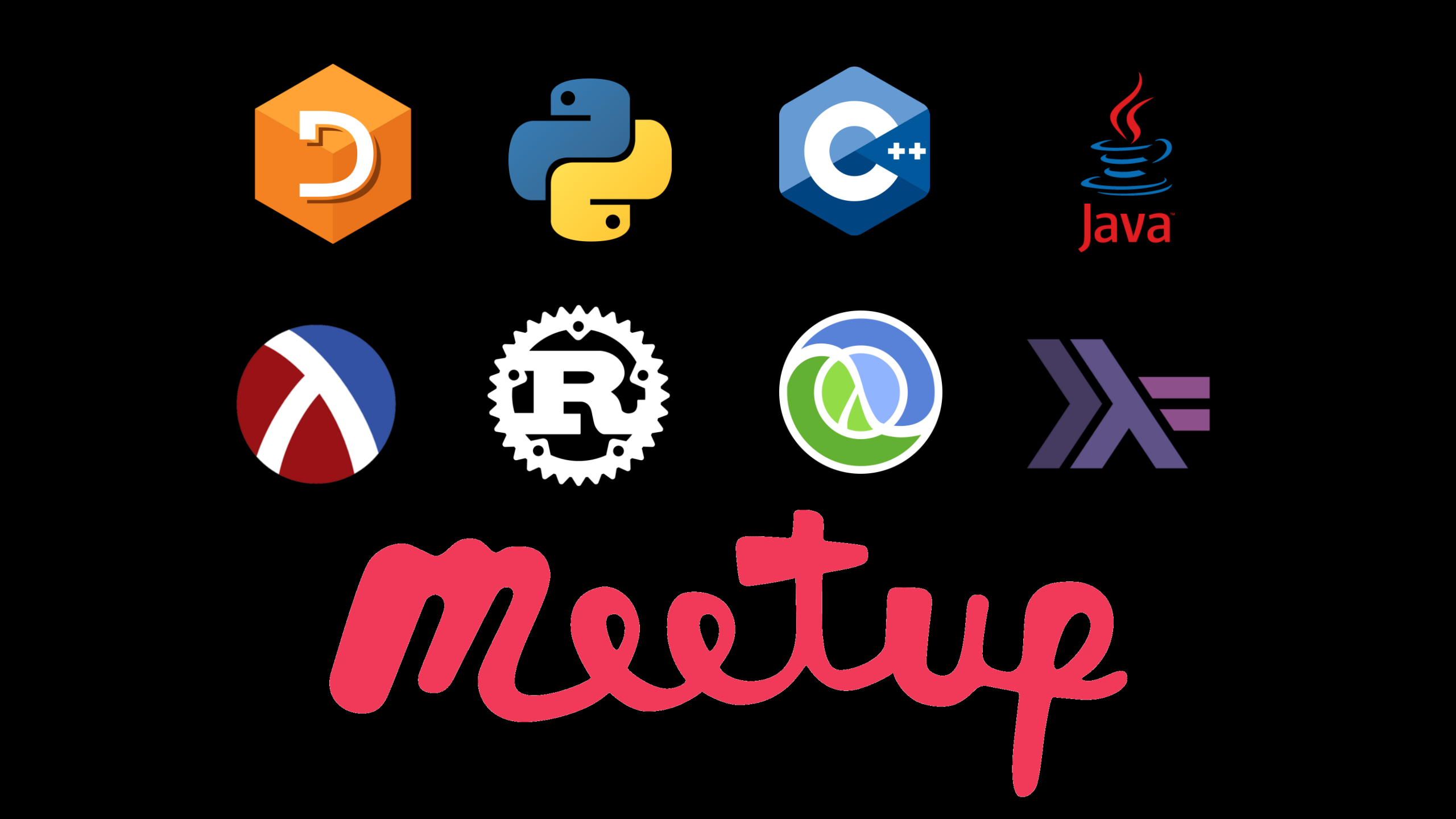
Category Theory II 8.2: Catamorphisms and...

4.4K views • 3 years ago

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

5.7K views • 3 years ago


string log = "";
bool negate(bool x) {
log += "not!";
return !x;
}
pair<bool, string>
negate(bool x) {
return make_pair(!x,
"Non!");
}
function<c(a)>
compose(function<b(a)> f, function<c(b)> g) {
return [f,g](a x) {
f(x)


Java
meetup