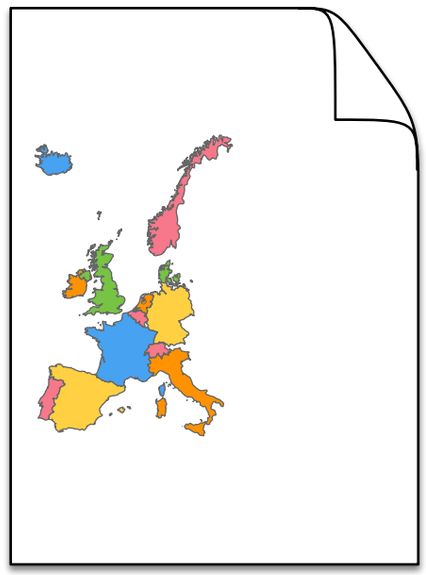
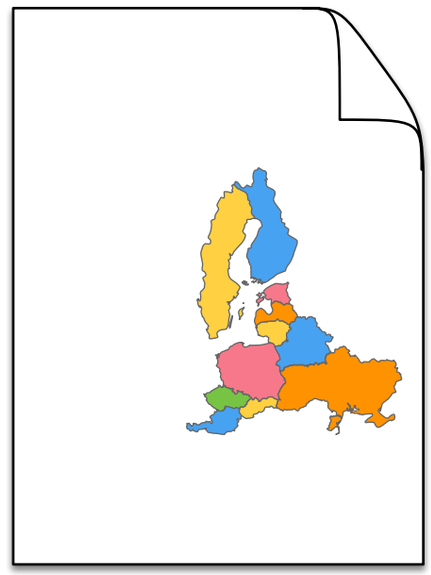
Constraints Liberate, Liberties Constrain

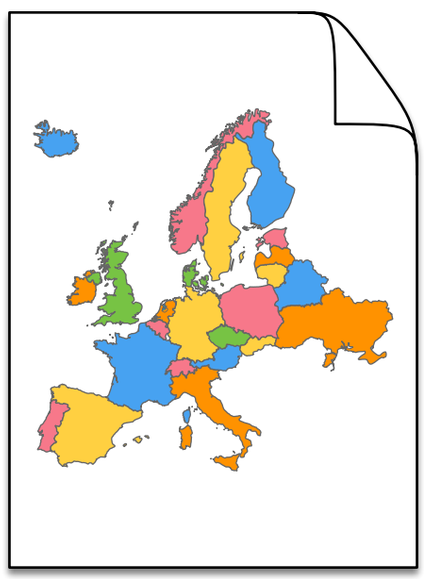
Rúnar Bjarnason @runarorama

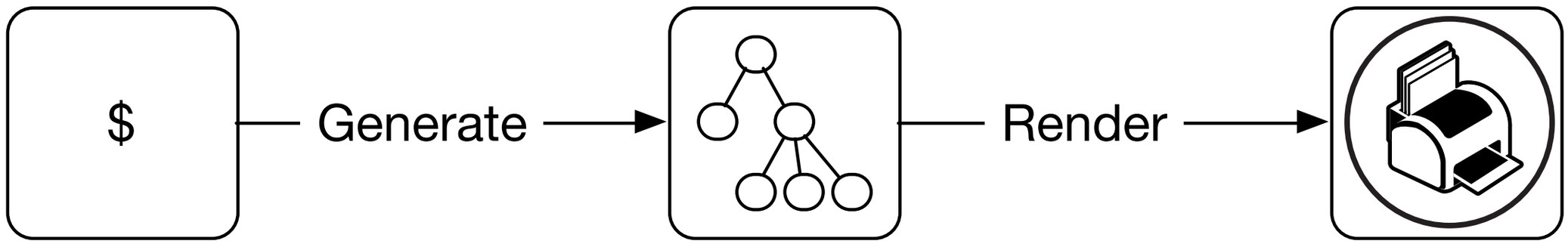
Scala World 2015


S
Q
L



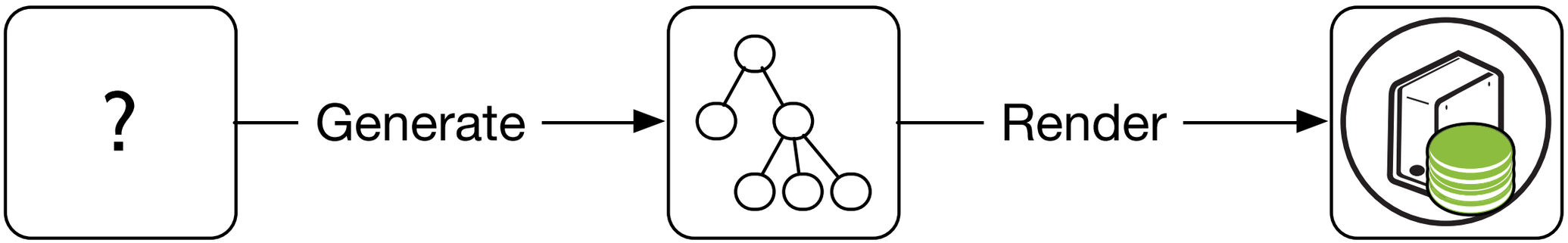





$
Generate
Render

```
val query = """
  select a, b, c
  from foo
  where a = ?
  """
```

```
val query = """
  select a, b, c, d
  from foo
  where a = ?
  and b = ?
  """
```

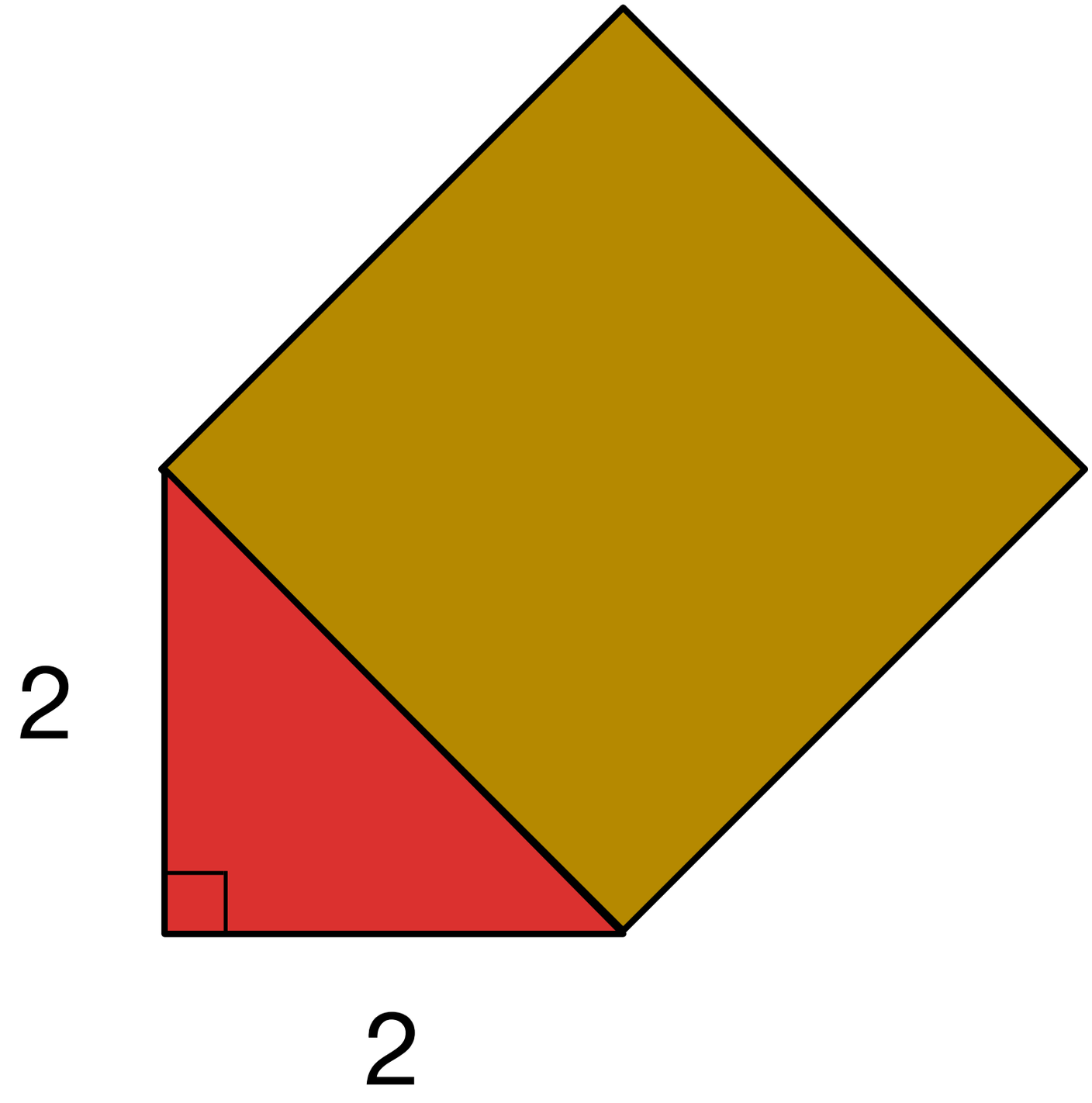

?
Generate
Render

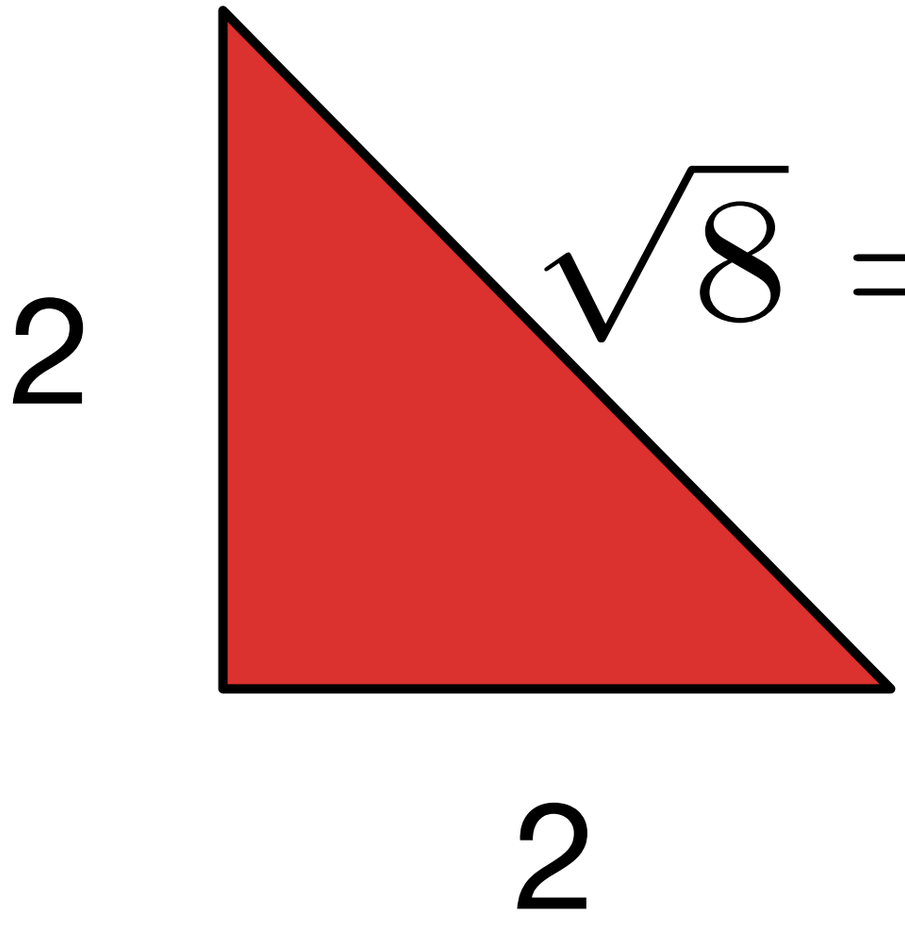
We can solve any problem by introducing an extra level of indirection.

– David J. Wheeler

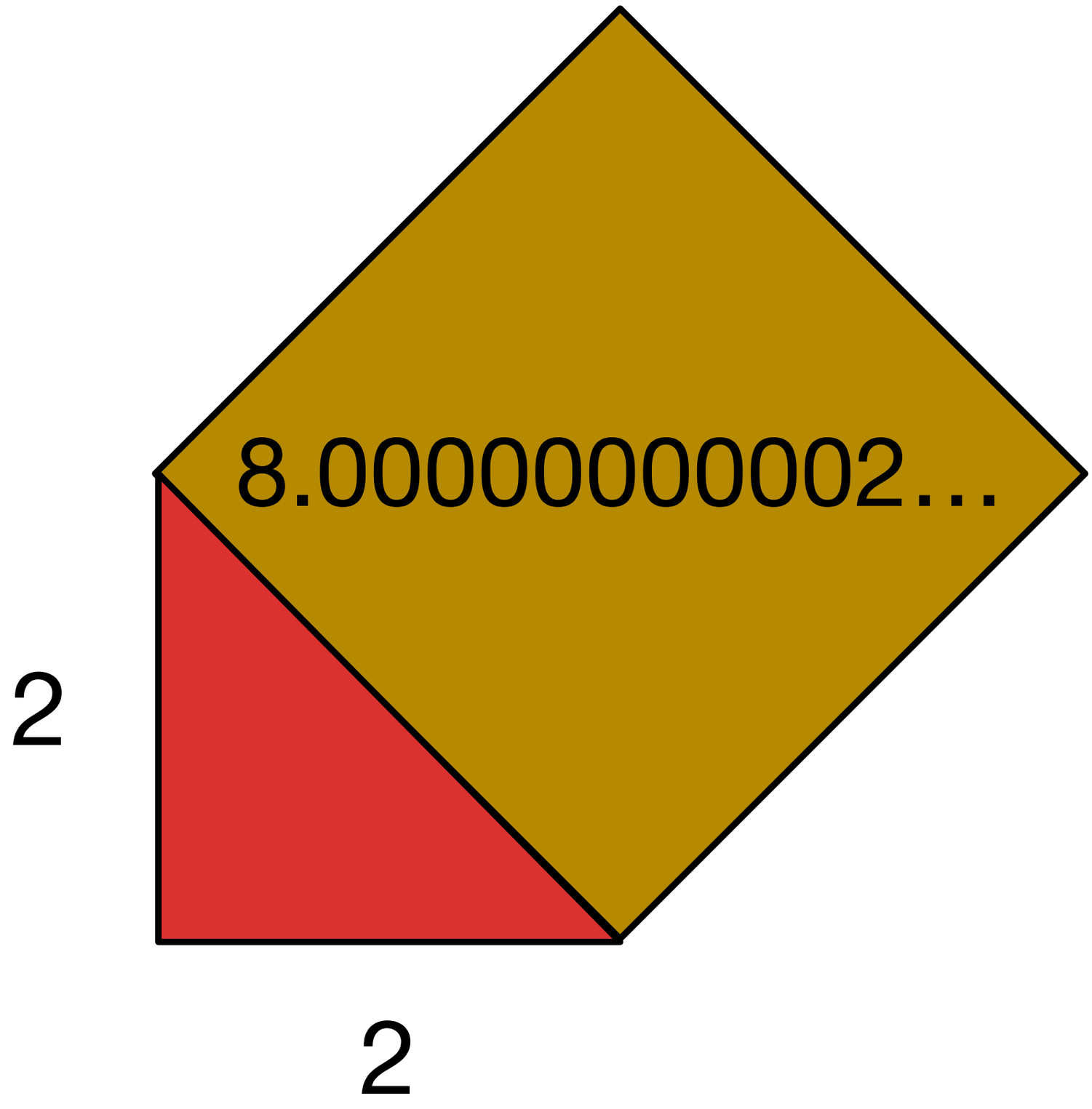

TNT
TNT

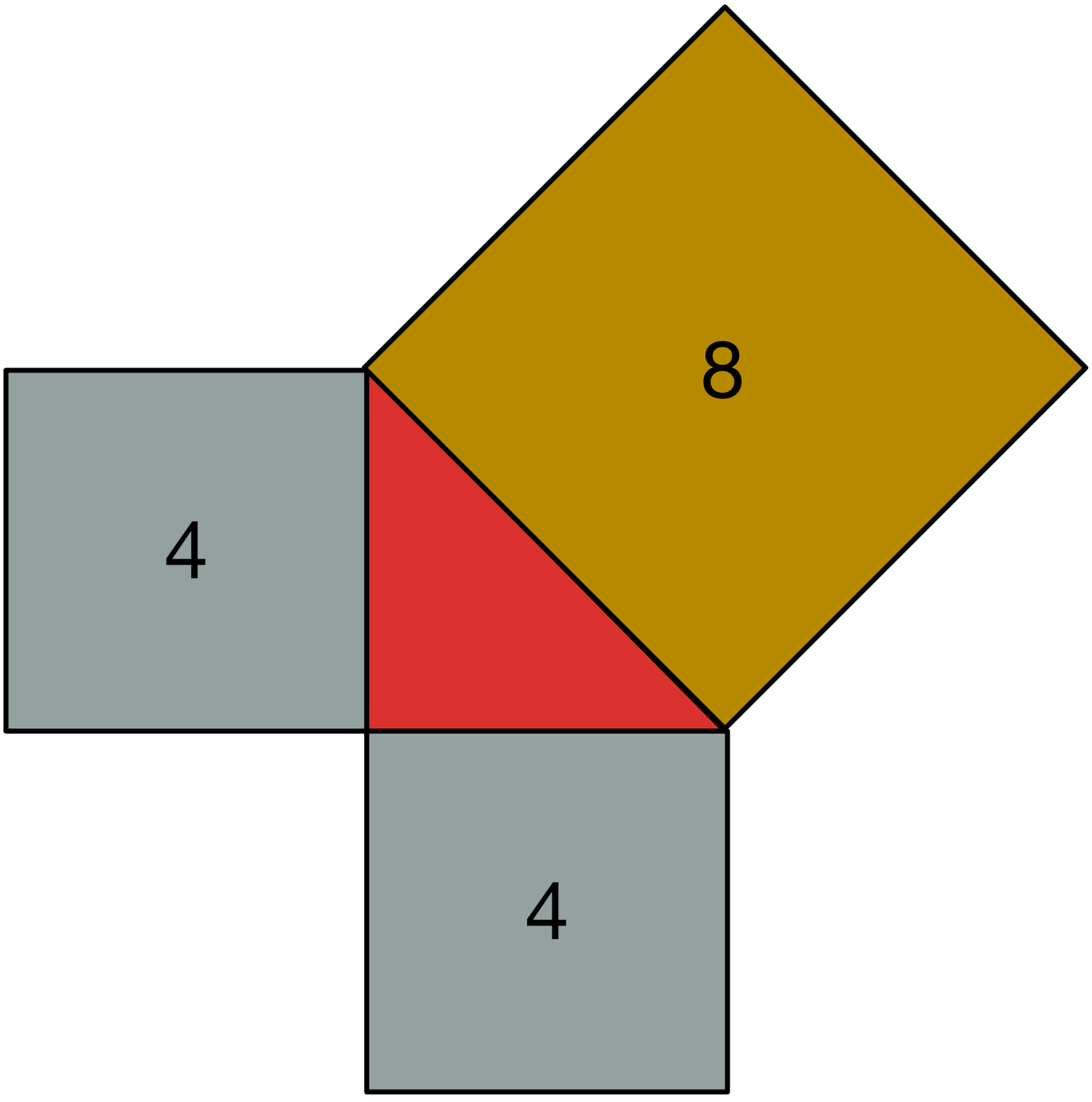
- Concretising too early
- Not designing for compositionality


2
2



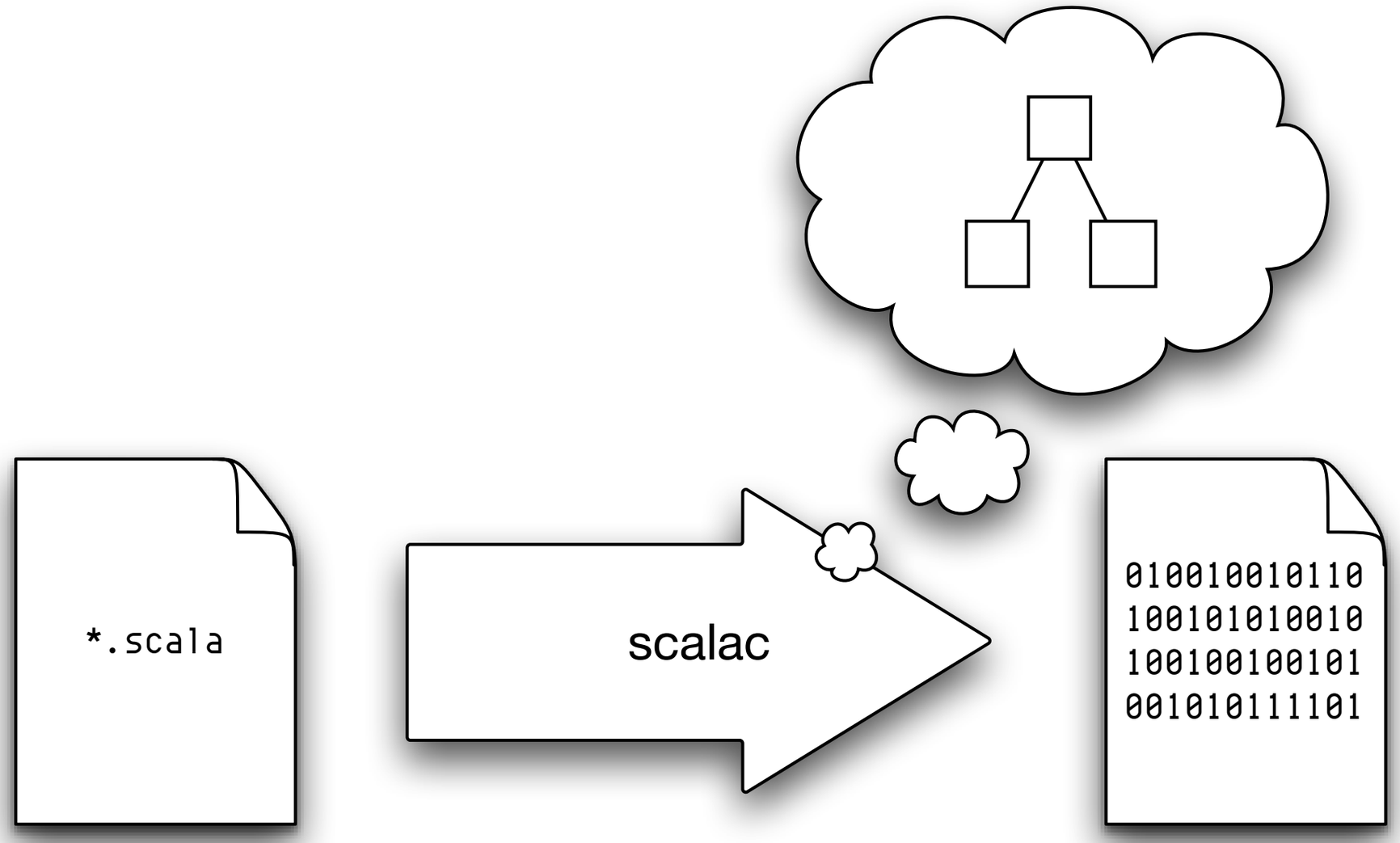
$$\sqrt{8} = 2.828...$$

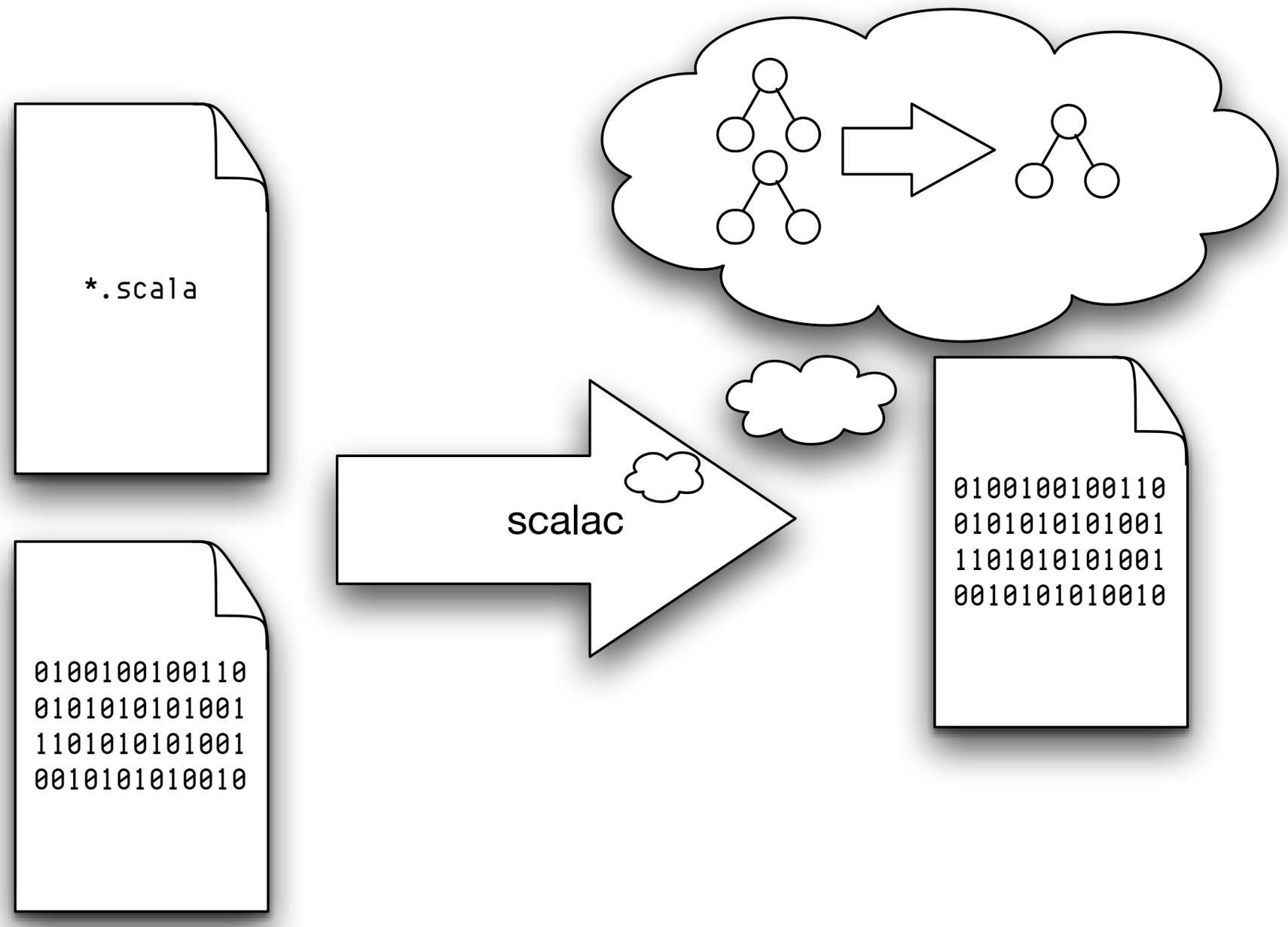

8.000000000002...
2
2

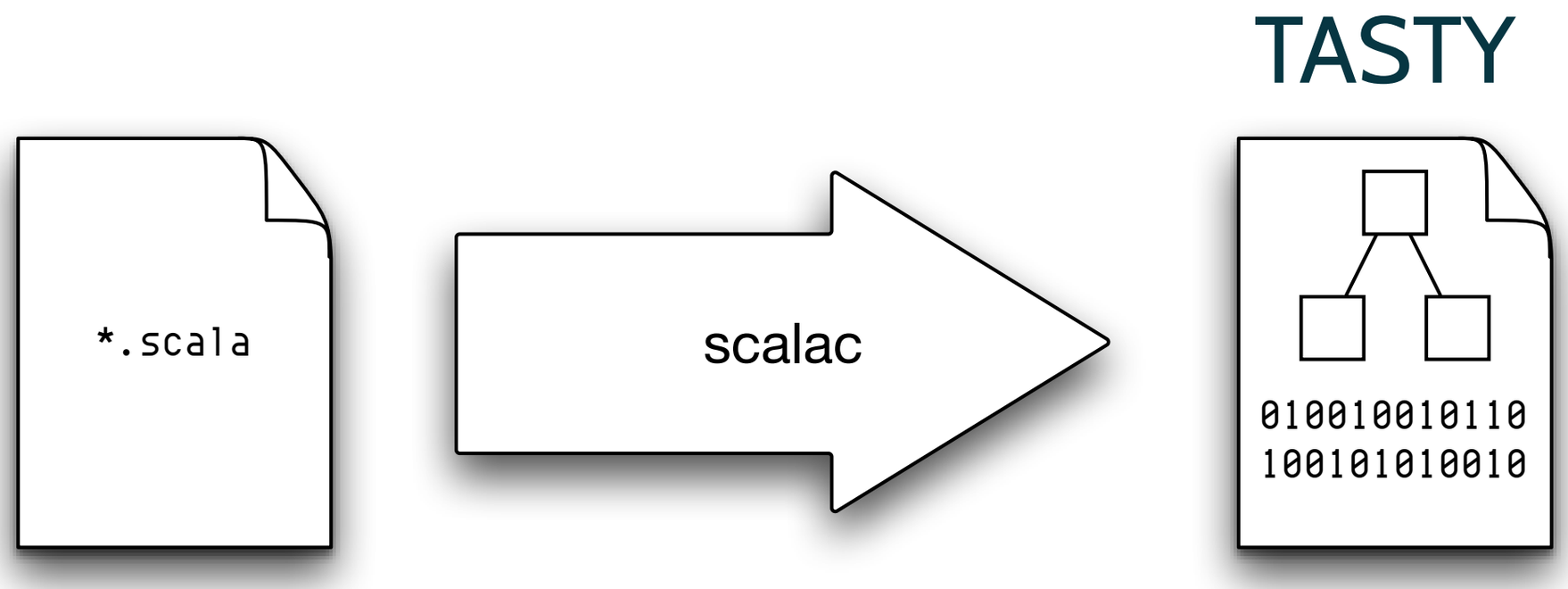

8
4
4

Work with algebraic relationships.

Go to decimals as late as possible.

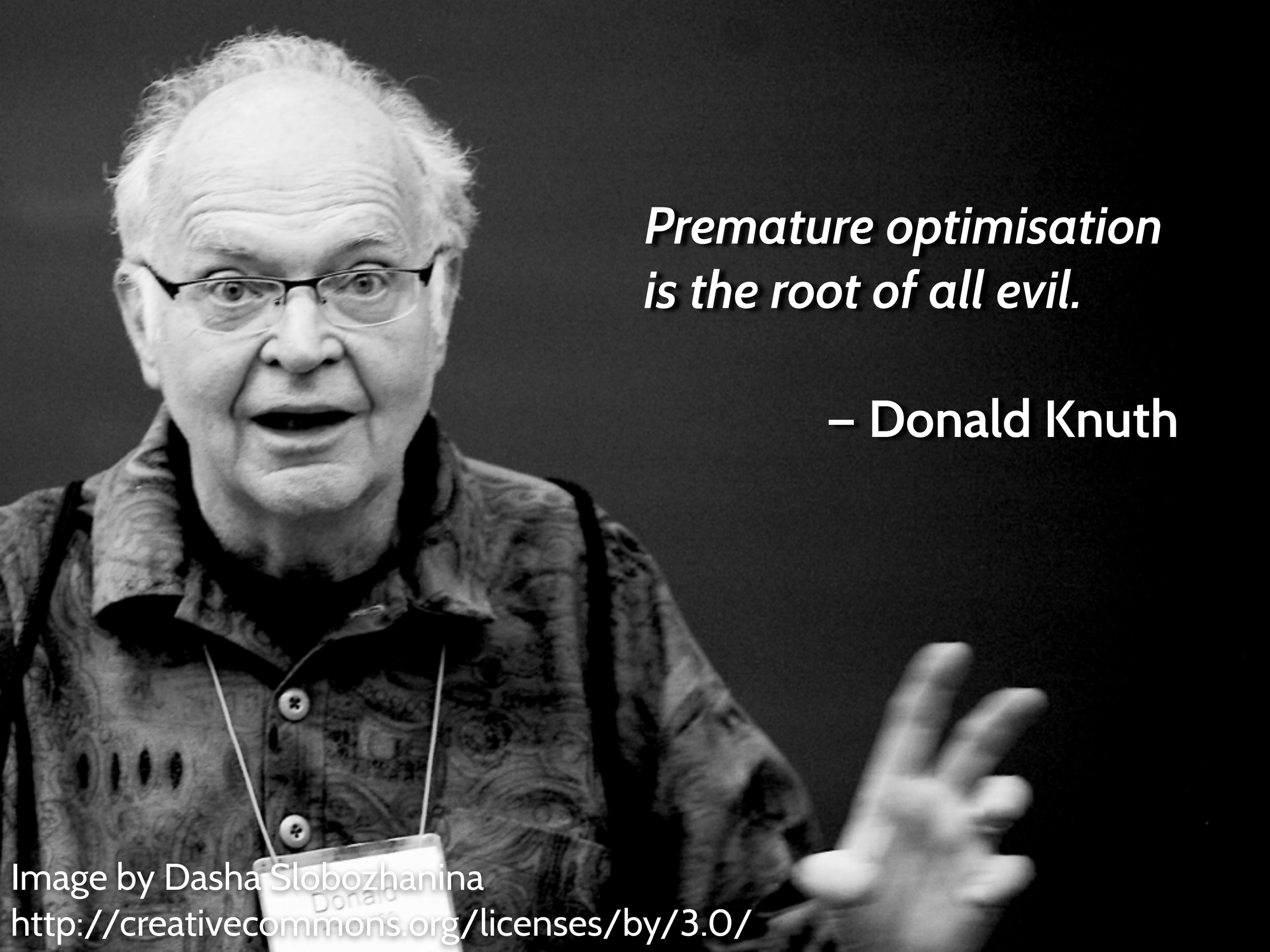

*.scala
scalac
010010010110
100101010010
100100100101
001010111101


*.scala
scalac
0100100100110
0101010101001
1101010101001
0010101010010
0100100100110
0101010101001
1101010101001
0010101010010


TASTY
*.scala
scalac
010010010110
100101010010

Work with syntax trees.

Go to binaries as late as possible.



Premature

- Loss of precision
- Concretisation
- Folding
- Compilation
- Optimisation

Going too early
to a form that is
too large.

"Our study of calculus is based on the **real number system**"

– *Calculus* by Sallas, Hille, & Etgen

- Real numbers
- “Stringly typed” programming
- General recursion
- Side-effects

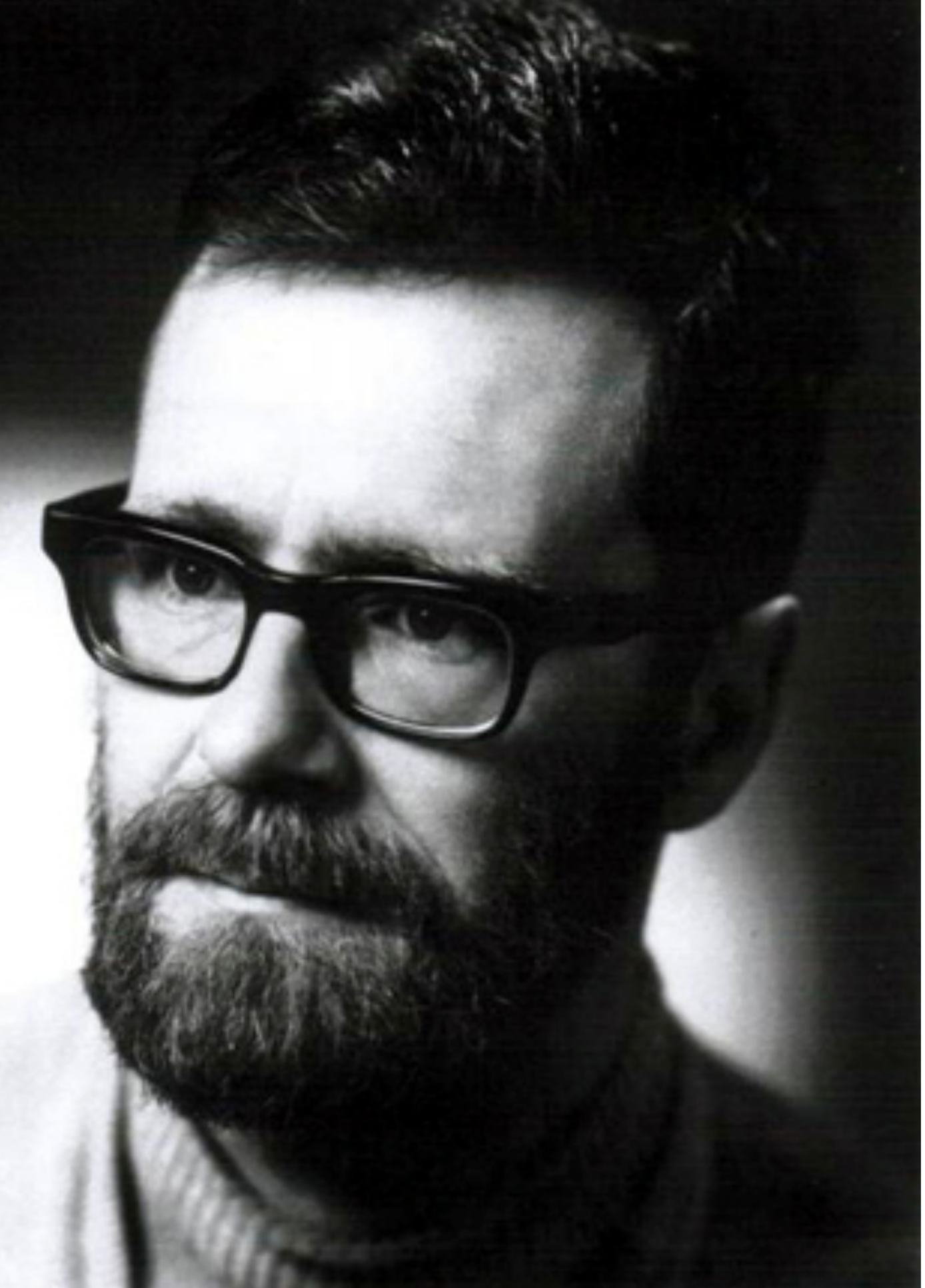
Maximally powerful

Minimally reasonable

The more kinds of things
something could ***potentially*** be,

the less we can reason about
what it ***actually*** is.

Abstraction.
Compositionality.
Precision.



The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.

– Edsger W. Dijktra

```
def foo(a: Int): Int
```

```
def foo[A](a: A): A
```

Freedom at one level leads to restriction at another level.

A constraint at one level leads to freedom and power at another level.

Monad vs. Applicative

```scala
trait Functor[F[_]] {
  def map[A,B](f: A => B): F[A] => F[B]
}
```

```scala
trait Applicative[F[_]] extends Functor[F] {
  def pure[A](a: A): F[A]
  def map2[A,B,C](
    f: (A, B) => C): (F[A], F[B]) => F[C]
}
```

```scala
trait Monad[F[_]] extends Applicative[F] {
  def join[A](a: F[F[A]]): F[A]
}
```

```scala
def compose[F[_],G[_]](
  F: Applicative[F],
  G: Applicative[G]): Applicative[F[G[?]]] = {
    new Applicative[F[G[?]]] {
      def pure[A](a: A) =
        F.pure(G.pure(a))
      def map2[A,B,C](f: (A, B) => C) =
        F.map2(G.map2(f))
    }
}
```

```scala
def compose[F[_],G[_]](
F: Monad[F],
G: Monad[G]): Monad[F[G[?]]] = ???
```

Monads are *more powerful*
than applicative functors,
but *less compositional*.

Monads *offer more* for
first-order applications,
but *demand more* for
higher-order applications.

A constraint at one level gives us freedom and power at a higher level.

Actor vs. Future

```
fork:
   A => Future[A]

map:
   (A => B) => (Future[A] => Future[B])

join:
   Future[Future[A]] => Future[A]
```

```
receive: Any => Unit
```

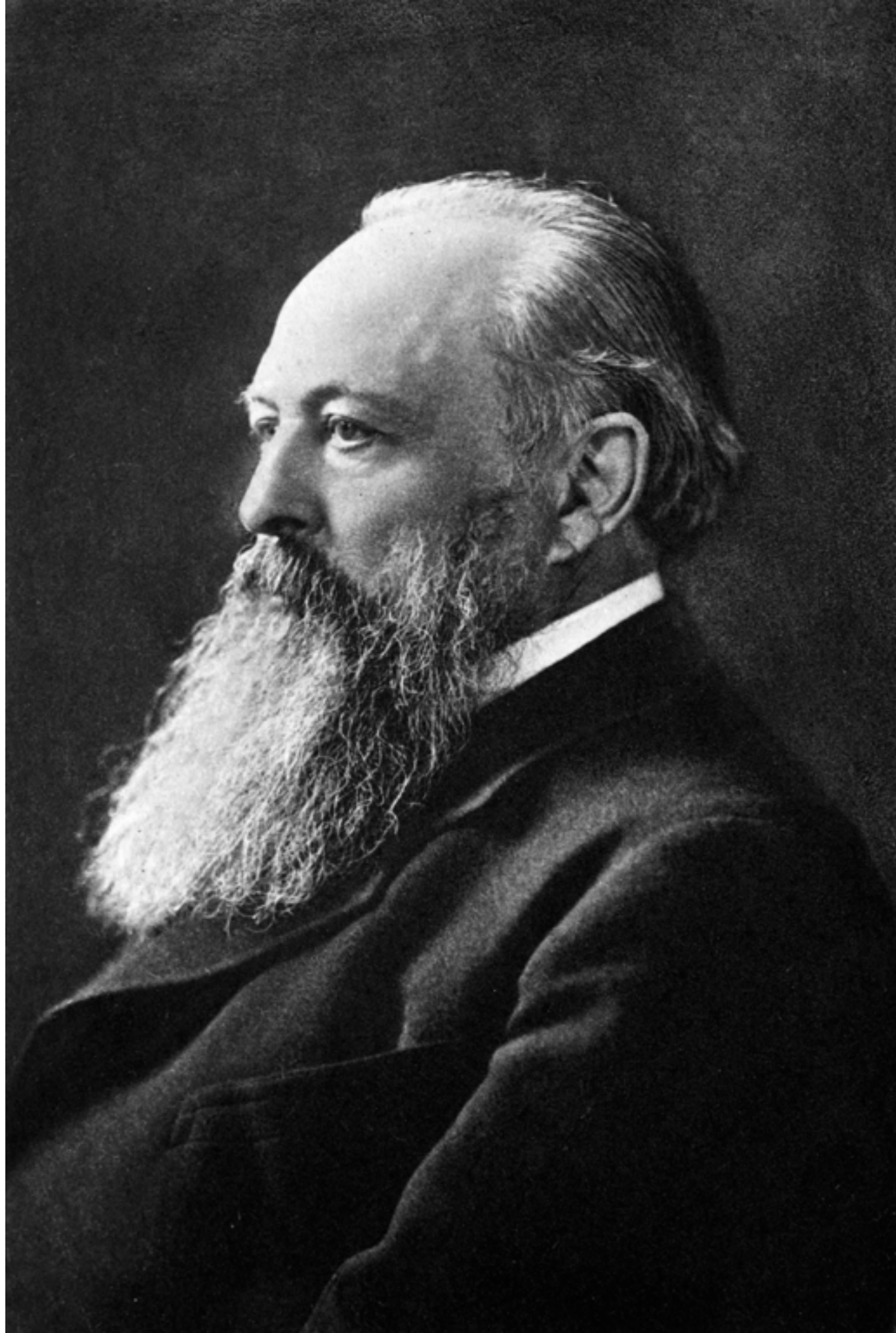
```
def become(
    behavior: Any => Unit,
    discardOld: Boolean = true): Unit

def unbecome(): Unit
```

Unconstrained side
effects are *powerful*.

Power tends to corrupt, and absolute power corrupts absolutely.

– Lord Acton



```
xs.map(_ - n).map(_ + n) == xs
```

The cost of side effects:

- Compositionality
- Modularity
- Concurrency
- Parametricity
- Algebraic reasoning

Principle of least privilege

A user, program, or component should have exactly as much authority as necessary but no more.

Minimal privileges =>

Maximal compositionality,

Maximal modularity.

The more a system ***can*** do,
the less we can predict what it ***will*** do.

- Context free vs regular grammars
- Euclidean vs Cartesian geometry
- Roadways
- Commodity components

A restriction at one semantic level translates to freedom and power at another semantic level.

Maximally general for first-order applications

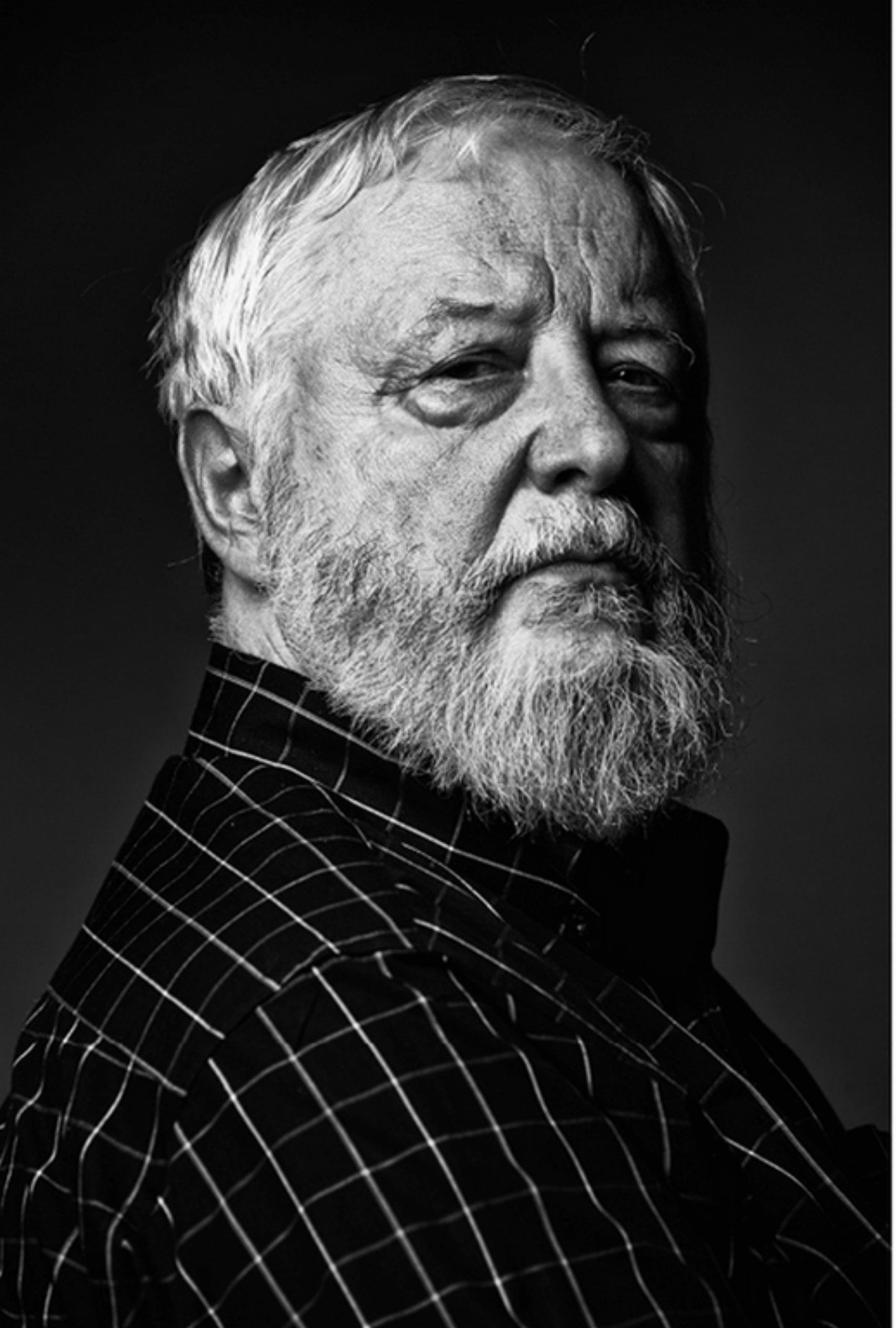
Minimally useful for higher-order applications

A small language can later embed in a larger one.

This does not work the other way around.

More capable syntax ⇒

fewer semantics



“Syntax and semantics are adjoint.”

– F. William Lawvere

Image © Olivier Toussaint.
Used with permission.

Adjoint

Category `C`:

Groups of concrete things.

`A :> B` when `B` is a subset of `A`.

Category `D`:

Ways of describing things.

`A *> B` when `B` is more specific than `A`.

$$F: C \Rightarrow D$$

Find the most specific description for a given group of concrete things.

$$G : \quad D \quad \Rightarrow \quad C$$

Find the most generous group of things whose elements all match a given description.

```
B *> F[G[B]]
```

```
G[F[A]] :> A
```

Larger grouping ⇒

fewer characterizations

Galois connection **F** ⊣ **G**

```
F[A] *> B ⇔ A :> G[B]
```

Adjunction $\mathbf{F} \dashv \mathbf{G}$

$$F[A] \Rightarrow B \Leftrightarrow A \Rightarrow G[B]$$

```scala
trait Adjunction[F[_],G[_]] {
  def left[A,B](f: F[A] => B): A => G[B]
  def right[A,B](f: A => G[B]): F[A] => B
}
```

```
def readerAdj[R] =
  new Adjunction[(?, R), (R => ?)] {
    def left[A,B](f: (A,R) => B): A => R => B =
      f.curried
    def right[A,B](f: A => R => B): (A,R) => B =
      Function.uncurried(f)
  }
```

Free ⊣ Forgetful

Data Types ⊣ Algebras

Summary

- Reach for the least powerful abstraction
- Detonate as late as possible
- Premature loss of precision is the root of all evil

Summary

- A constraint at one level leads to freedom and power at another level.
- Plan for compositionality
- Don’t always reach for maximum expressive power