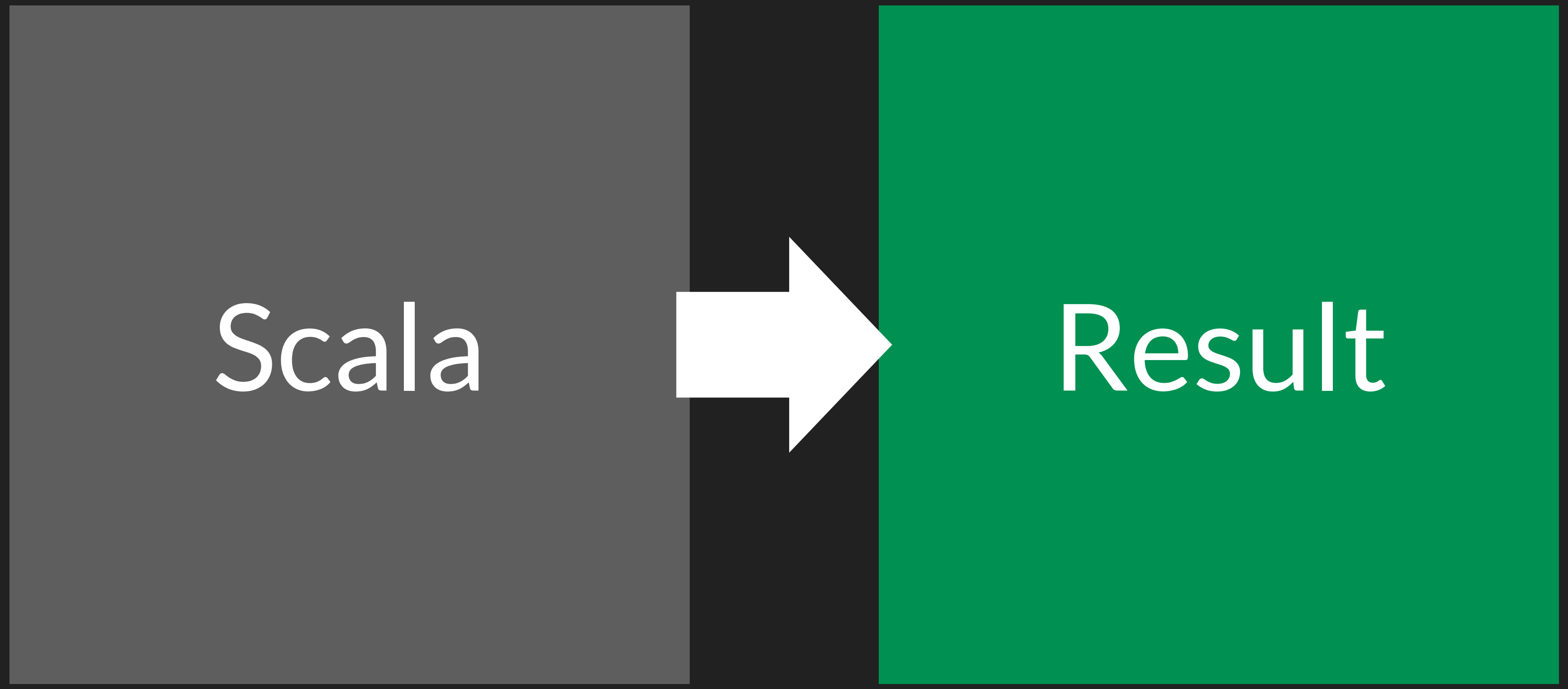
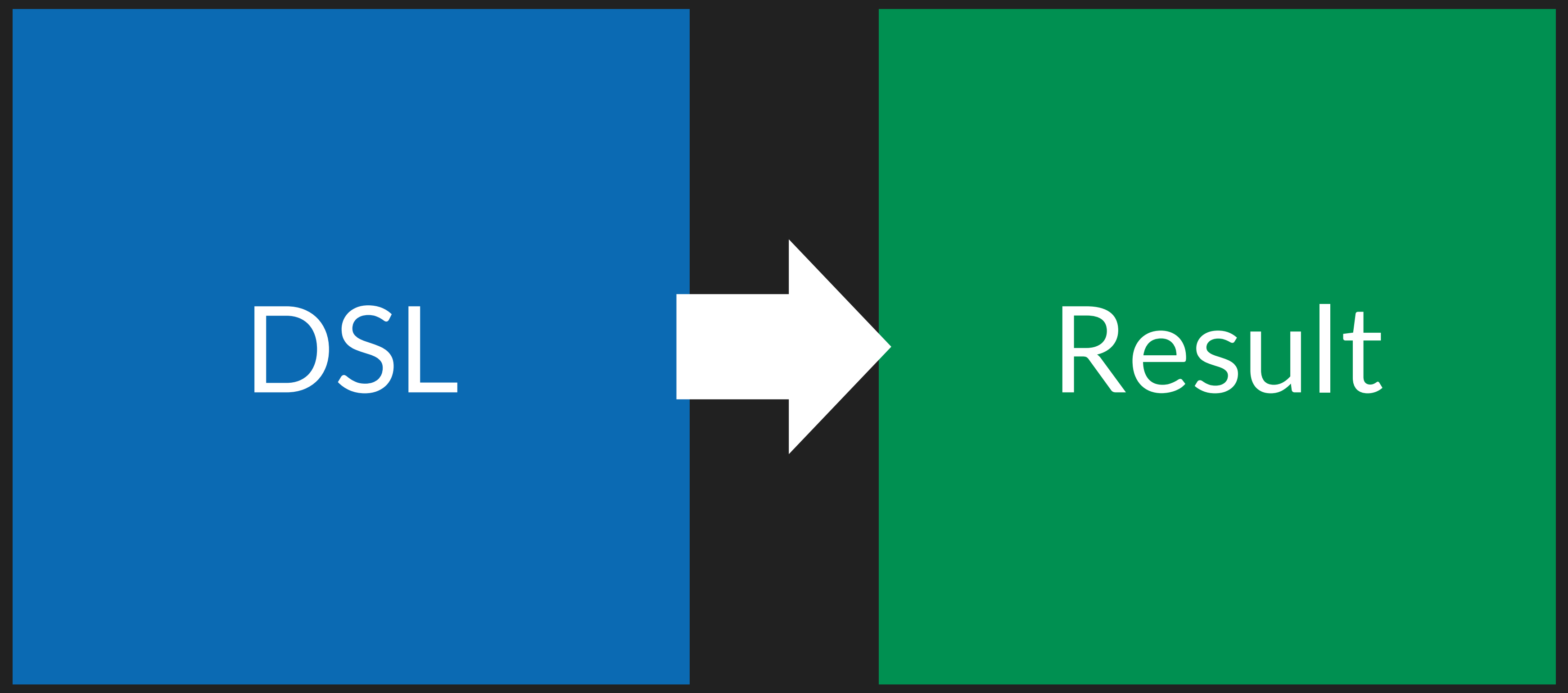
Interpreters and you

Dave Gurnell, @davegurnell

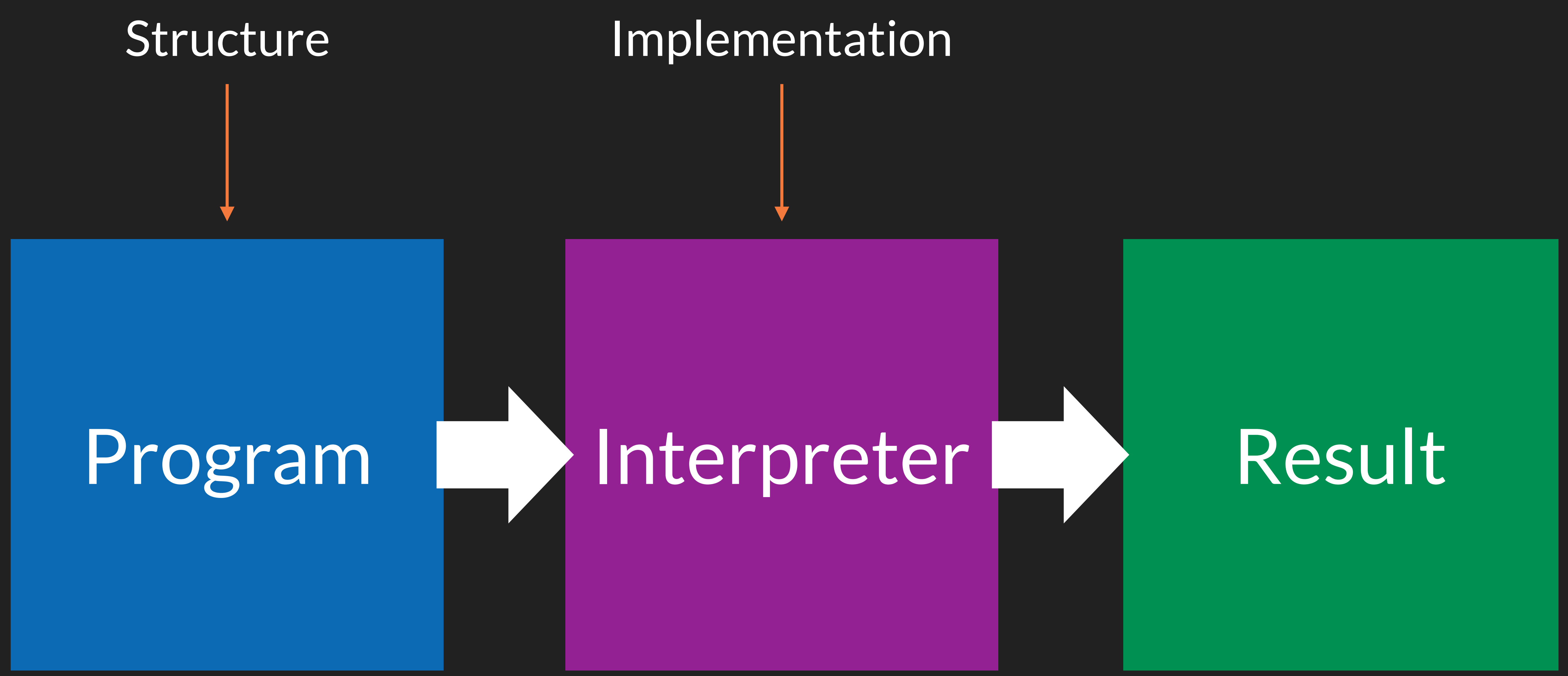
underscore

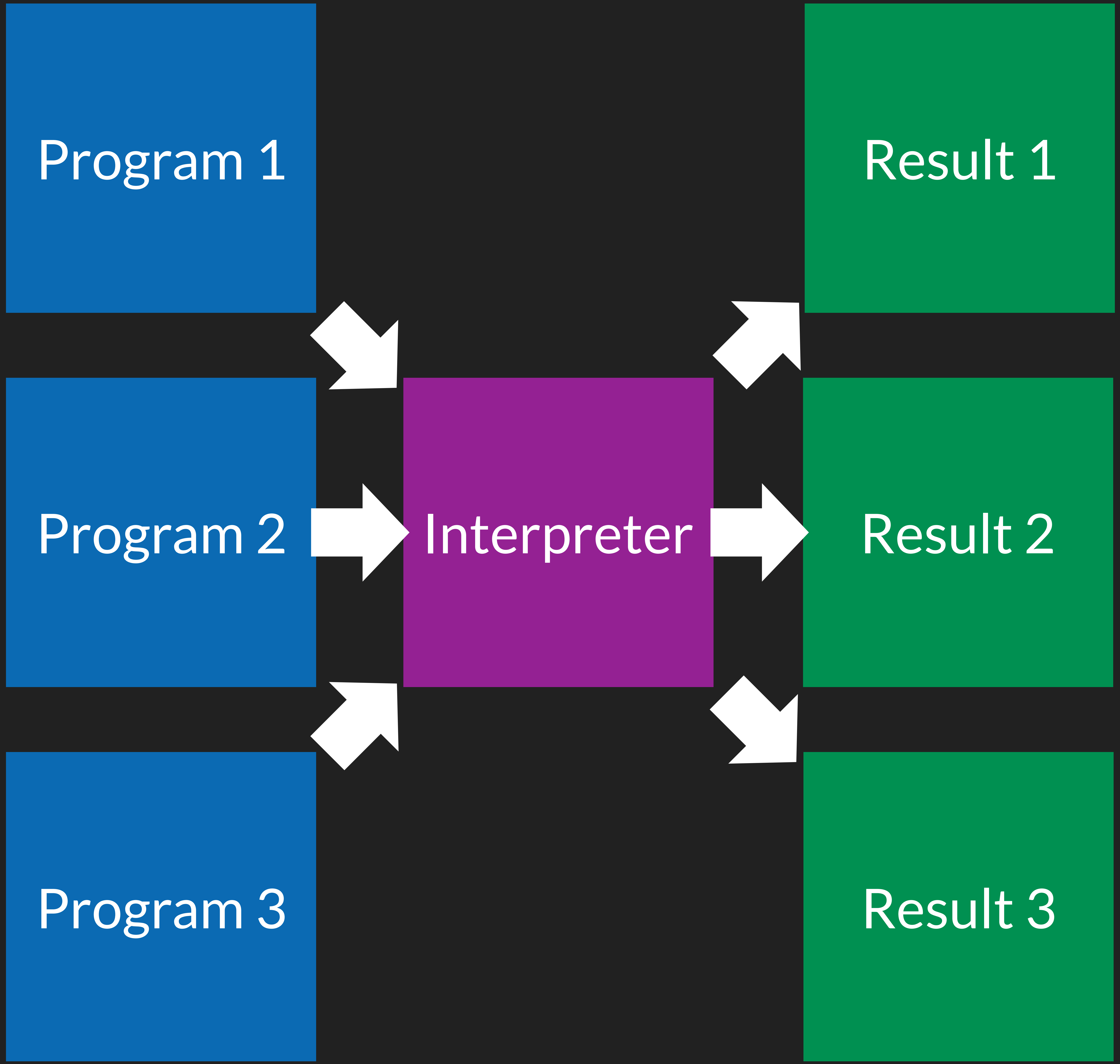
Scala

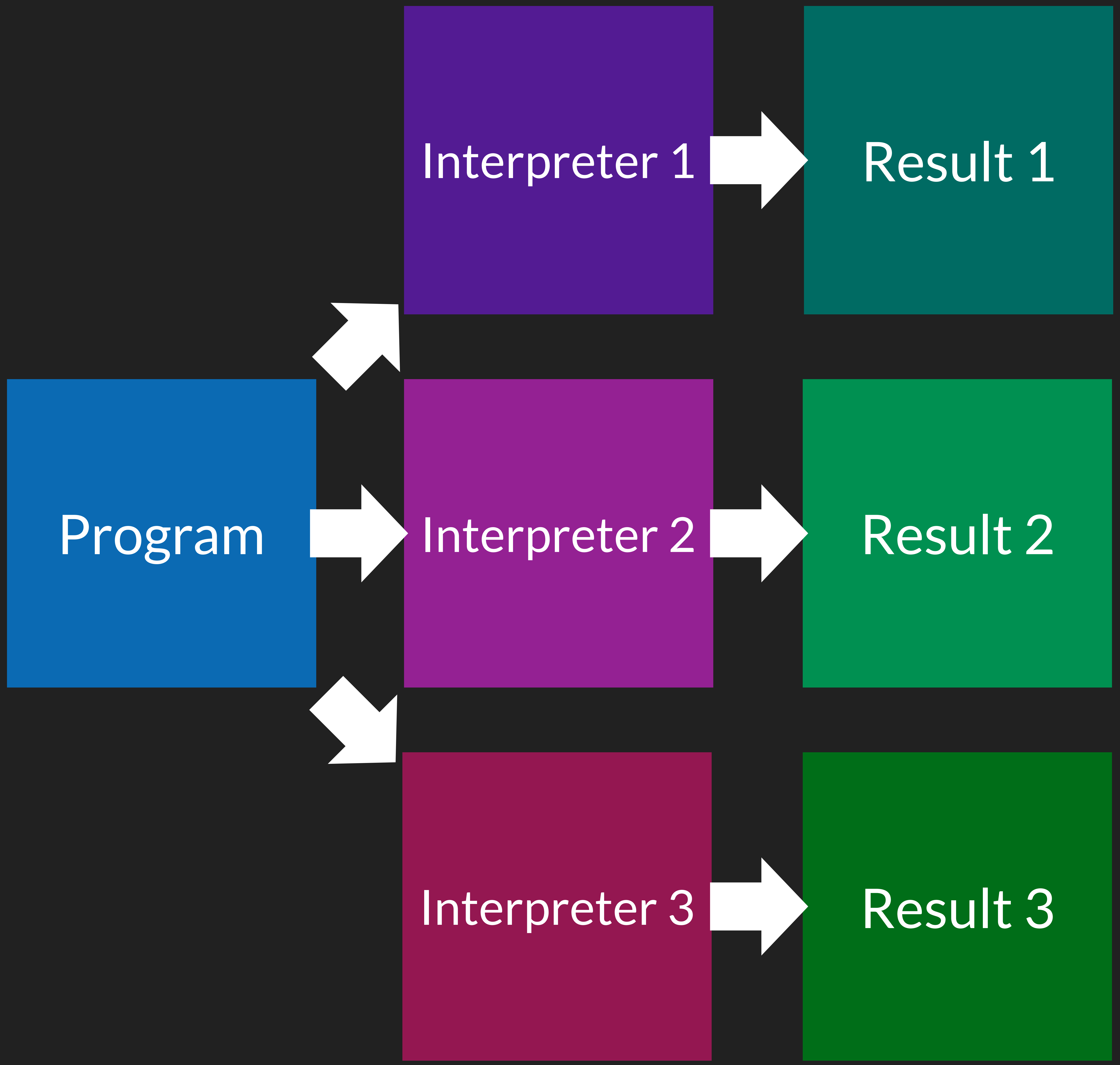

Scala
Result

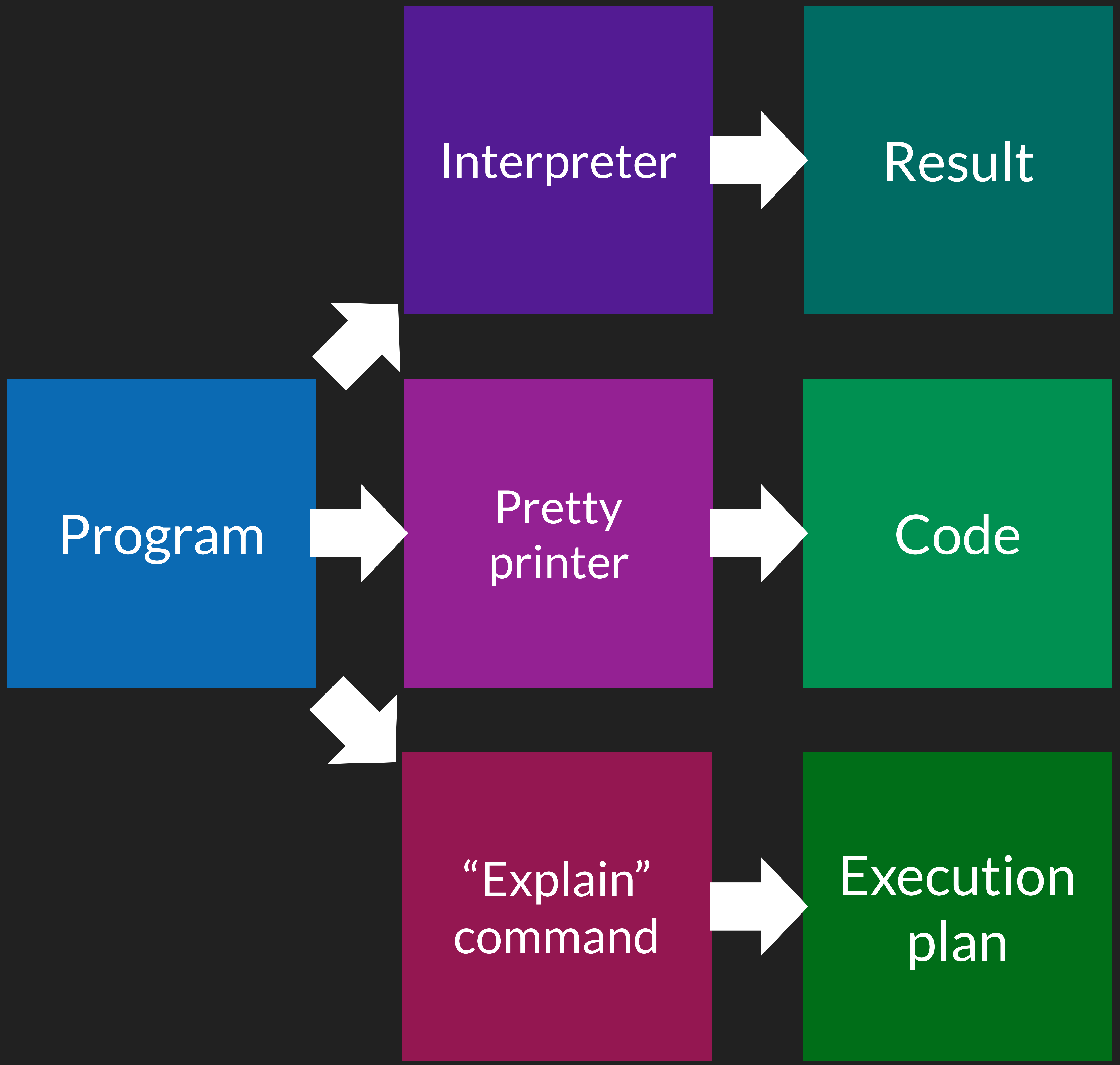

DSL
Result

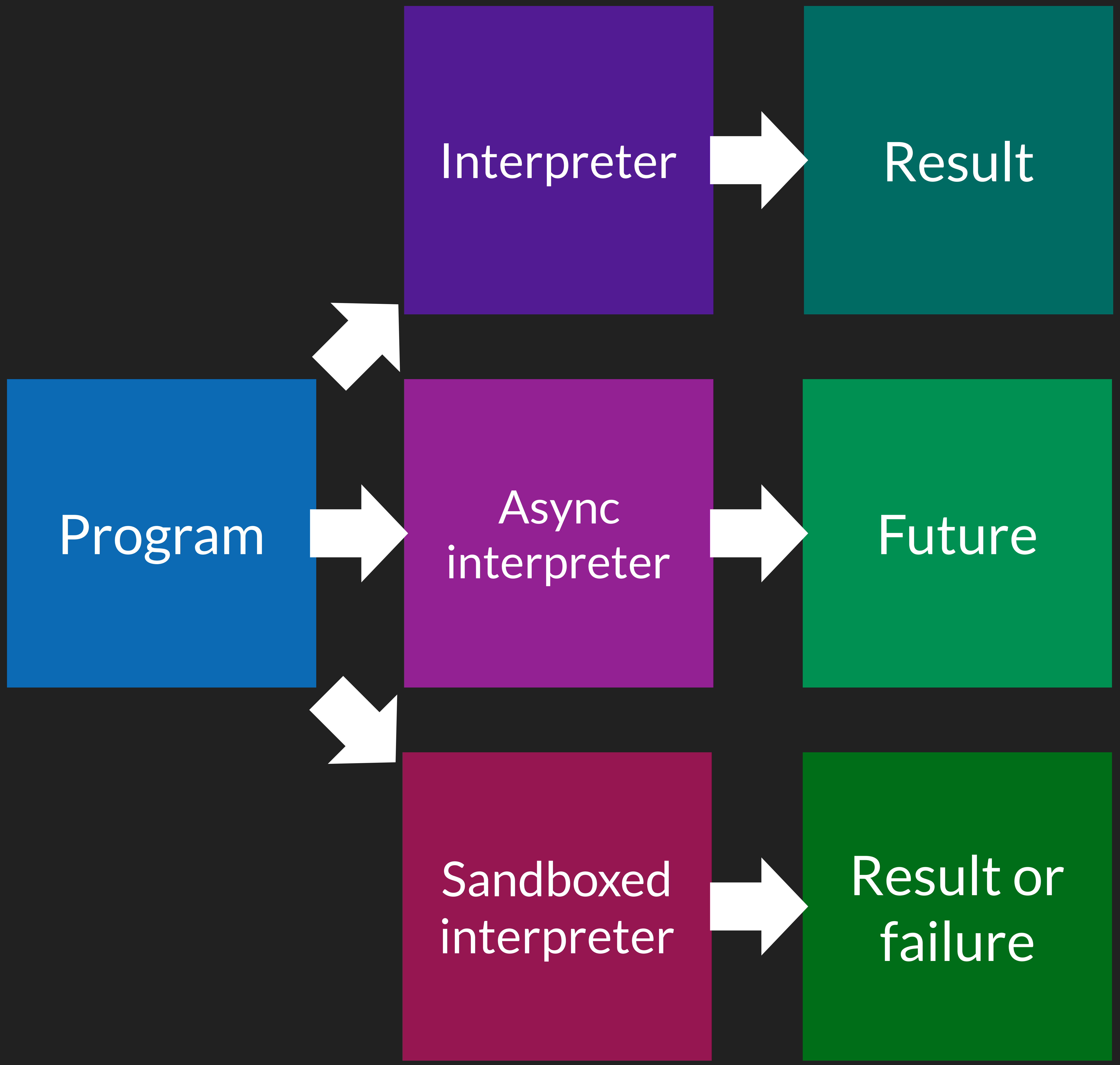

Program
Interpreter
Result

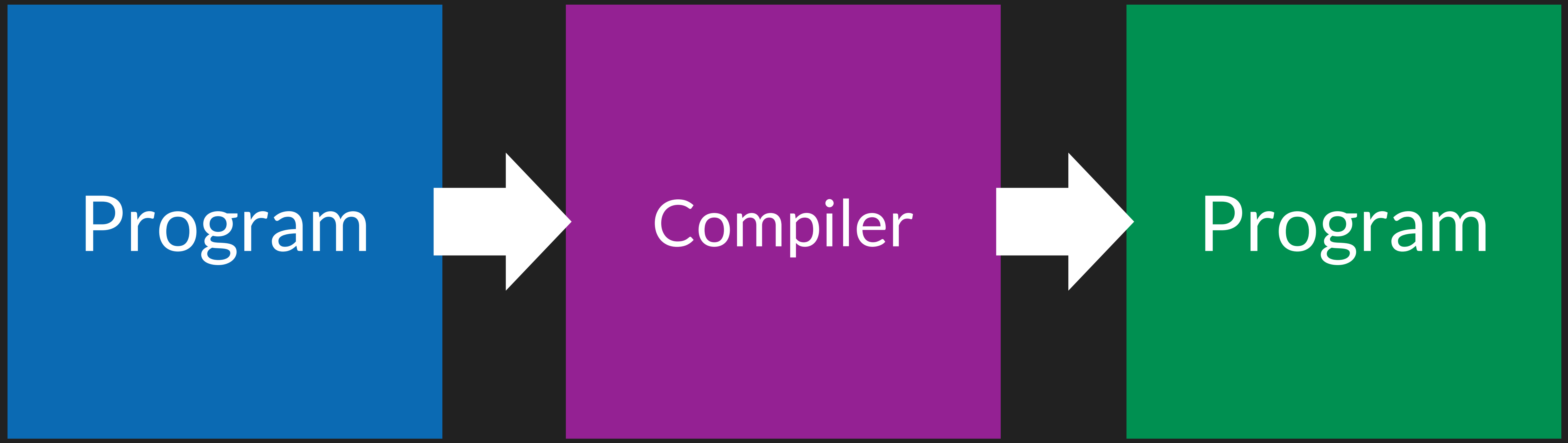

Structure
Implementation
Program
Interpreter
Result


Program 1
Program 2
Program 3
Interpreter
Result 1
Result 2
Result 3


Program
Interpreter 1
Interpreter 2
Interpreter 3
Result 1
Result 2
Result 3


Interpreter
Result
Program
Pretty printer
Code
“Explain” command
Execution plan


Interpreter
Result
Program
Async interpreter
Future
Sandboxed interpreter
Result or failure


Program
Compiler
Program

1. Re-use interpreters

2. Multiple interpreters

3. Abstract over effects

(4. Inspect / rewrite programs)

Big approaches

Reification

Church encoding

Reification

Model code as data

Church encoding

Model code as … code

Reification ➡ Free

Model code as data

Church encoding ➡ Tagless final

Model code as … code

Standalone

Embedded

Standalone

Completely custom

Embedded

Reuse host language features

Reified encodings

A calculated example

```
(1 + 2) * (3 + 4)
```

```
val program = () ⇒
  (1 + 2) * (3 + 4)
```

```scala
val program = () ⇒
  (1 + 2) * (3 + 4)

program()
// res0: Int = 21
```

```scala
val program = () ⇒
  (1 + 2) * (3 + 4)

def eval(prog: () ⇒ Int): Int =
  ???
```

```scala
val program = () ⇒
  (1 + 2) * (3 + 4)

def eval(prog: () ⇒ Int): Int =
  prog()
```

```scala
sealed trait Expr

case class Lit(value: Int) extends Expr

case class Add(a: Expr, b: Expr) extends Expr

case class Mul(a: Expr, b: Expr) extends Expr
```

```scala
sealed trait Expr
case class Lit(value: Int) extends Expr
case class Add(a: Expr, b: Expr) extends Expr
case class Mul(a: Expr, b: Expr) extends Expr

val program: Expr =
  Mul(Add(Lit(1), Lit(2)), Add(Lit(3), Lit(4)))
```

```scala
sealed trait Expr
case class Lit(value: Int) extends Expr
case class Add(a: Expr, b: Expr) extends Expr
case class Mul(a: Expr, b: Expr) extends Expr

val program: Expr =
  Mul(Add(Lit(1), Lit(2)), Add(Lit(3), Lit(4)))

def eval(expr: Expr): Int =
  expr match {
    case Lit(value) => value
    case Add(a, b)  => eval(a) + eval(b)
    case Mul(a, b)  => eval(a) * eval(b)
  }

eval(program)
// res0: Int = 21
```

```scala
def eval(expr: Expr): Int =
  expr match {
    case Lit(value) => value
    case Add(a, b)  => eval(a) + eval(b)
    case Mul(a, b)  => eval(a) * eval(b)
  }
```

```scala
def evalAsync(expr: Expr): Future[Int] =
  expr match {
    case Lit(value) ⇒
      Future.successful(value)

    case Add(a, b) ⇒
      val x = evalAsync(a)
      val y = evalAsync(b)
      x.flatMap(a ⇒ y.map(b ⇒ a + b))

    case Mul(a, b) ⇒
      val x = evalAsync(a)
      val y = evalAsync(b)
      x.flatMap(a ⇒ y.map(b ⇒ a * b))
  }
```

```scala
def prettyPrint(expr: Expr): String =
  expr match {
    case Lit(value) ⇒
      value.toString

    case Add(a, b) ⇒
      s"${prettyPrint(a)} + ${prettyPrint(b)}"

    case Mul(a, b) ⇒
      s"${prettyPrint(a)} * ${prettyPrint(b)}"
  }
```

```scala
def simplify(expr: Expr): Expr =
  expr match {
    case Mul(a, Add(b, c)) ⇒
      simplify(Add(Mul(a, b), Mul(a, c)))

    case Mul(Add(a, b), c) ⇒
      simplify(Add(Mul(a, c), Mul(b, c)))

    case Mul(a, b) ⇒
      Mul(simplify(a), simplify(b))

    case Add(a, b) ⇒
      Add(simplify(a), simplify(b))

    case Lit(v) ⇒
      Lit(v)
  }
```

Simplify the language

Empower the interpreter


Constraints liberate
Liberty constrains

What about types?

Untyped DSLs

```
1 < 2 && 3 < 4
```

```scala
sealed trait Expr

case class Lit(value: Int) extends Expr

case class Lt(a: Expr, b: Expr) extends Expr

case class And(a: Expr, b: Expr) extends Expr
```

```
val validProgram: Expr =
  And(Lt(Lit(1), Lit(2)), Lt(Lit(3), Lit(4)))
```

```scala
val invalidProgram: Expr =
  Lt(And(Lit(1), Lit(2)), And(Lit(3), Lit(4)))
```

```
def eval(program: Expr): ??? =
  ???
```

Boolean or integer expression ?

```scala
def eval(program: Expr): ??? =
  ???
```

Boolean or integer expression ?

```
def eval(program: Expr): ??? =
  ???
```

Model errors here ?

Boolean or integer expression ?

```
def eval(program: Expr): ??? =
  program match {
    case Lit(n) ⇒
      ???

    case Lt(a, b) ⇒
      ???

    case And(a, b) ⇒
      ???
  }
```

Model errors here ?

Are sub-expressions the right type?

Boolean or integer expression ?

```
def eval(program: Expr): Either[Error, ???] =
  program match {
    case Lit(n) ⇒
      ???

    case Lt(a, b) ⇒
      ???

    case And(a, b) ⇒
      ???
  }
```

Are sub-expressions the right type?

```scala
sealed trait Value
case class IntValue(n: Int) extends Expr
case class BooleanValue(b: Boolean) extends Expr

def eval(program: Expr): Either[Error, Value] =
  program match {
    case Lit(n) ⇒
      ???

    case Lt(a, b) ⇒
      ???

    case And(a, b) ⇒
      ???
  }
```

Are sub-expressions the right type?

```scala
sealed trait Value
case class IntValue(n: Int) extends Expr
case class BooleanValue(b: Boolean) extends Expr

def eval(program: Expr): Either[Error, Value] =
  program match {
    case Lit(n) ⇒
      Right(IntValue(n))

    case Lt(a, b) ⇒
      val x = evalAsInt(a)
      val y = evalAsInt(b)
      x.flatMap(a ⇒ y.map(b ⇒ a < b))

    case And(a, b) ⇒
      val x = evalAsBool(a)
      val y = evalAsBool(b)
      x.flatMap(a ⇒ y.map(b ⇒ a && b))
  }
```

Language mismatch

↓

Complex interpreter

Can this be simplified?

Typed DSLs

```scala
sealed trait Expr

case class Lit(value: Int) extends Expr

case class Lt(a: Expr, b: Expr) extends Expr

case class And(a: Expr, b: Expr) extends Expr
```

```scala
sealed trait Expr[A]

case class Lit[A](value: A) extends Expr[A]

case class Lt(a: Expr[Int], b: Expr[Int])
  extends Expr[Boolean]

case class And(a: Expr[Boolean], b: Expr[Boolean])
  extends Expr[Boolean]
```

```scala
val validProgram: Expr[Boolean] =
  And(Lt(Lit(1), Lit(2)), Lt(Lit(3), Lit(4)))
```

```scala
val invalidProgram: Expr[Boolean] =
  Lt(And(Lit(1), Lit(2)), And(Lit(3), Lit(4)))
//         ^ expected Expr[Boolean], found Expr[Int]
```

```scala
def eval[A](program: Expr[A]): A =
  program match {
    case Lit(v) ⇒
      v

    case Lt(a, b) ⇒
      eval(a) < eval(b)

    case And(a, b) ⇒
      eval(a) && eval(b)
  }
```

```scala
def evalAsync[A](program: Expr[A]): Future[A] =
  program match {
    case Lit(v) ⇒
      v

    case Lt(a, b) ⇒
      val x = evalAsync(a)
      val y = evalAsync(b)
      x.flatMap(a ⇒ y.map(b ⇒ a < b))

    case And(a, b) ⇒
      val x = evalAsync(a)
      val y = evalAsync(b)
      x.flatMap(a ⇒ y.map(b ⇒ a && b))
  }
```

```scala
def prettyPrint[A](program: Expr[A]): String =
  program match {
    case Lit(v) ⇒
      v.toString

    case Lt(a, b) ⇒
      s"${prettyPrint(a)} < ${prettyPrint(b)}"

    case And(a, b) ⇒
      s"${prettyPrint(a)} && ${prettyPrint(b)}"
  }
```

```scala
def simplify[A](program: Expr[A]): Expr[A] =
  program match {
    case Lit(n) ⇒
      Lit(n)

    case Lt(Lit(a), Lit(b)) ⇒
      Lit(a < b)

    case Lt(a, b) ⇒
      Lt(simplify(a), simplify(b))

    case And(a, b) ⇒
      And(simplify(a), simplify(b))
  }
```

Deeper embedding

↓

Simpler interpreter

Ordering

```scala
def eval[A](program: Expr[A]): A =
  program match {
    case Lit(v) ⇒
      v

    case Lt(a, b) ⇒
      eval(a) < eval(b)

    case And(a, b) ⇒
      eval(a) && eval(b)
  }
```

Can we have
explicit ordering?

Yes, with monads!

```scala
def readInt: Option[Int] =
  // ...

for {
  a ← readInt
  b ← readInt
} yield a + b
```

Monadic DSLs

```scala
sealed trait Expr[A]

case class Lit[A](value: A) extends Expr[A]

case class Lt(a: Expr[Int], b: Expr[Int])
  extends Expr[Boolean]

case class And(a: Expr[Boolean], b: Expr[Boolean])
  extends Expr[Boolean]
```

```scala
sealed trait Expr[A]

case class Lit[A](value: A) extends Expr[A]

case class Lt(a: Expr[Int], b: Expr[Int])
  extends Expr[Boolean]

case class And(a: Expr[Boolean], b: Expr[Boolean])
  extends Expr[Boolean]

case class FlatMap[A, B](
  a: Expr[A],
  f: A ⇒ Expr[B]) extends Expr[B]
```

```scala
sealed trait Expr[A]

case class Lit[A](value: A) extends Expr[A]

case class Lt(a: Int, b: Int)
  extends Expr[Boolean]

case class And(a: Boolean, b: Boolean)
  extends Expr[Boolean]

case class FlatMap[A, B](
  a: Expr[A],
  f: A ⇒ Expr[B]) extends Expr[B]
```

```scala
sealed trait Expr[A]

case class Pure[A](value: A) extends Expr[A]

case class Lt(a: Int, b: Int)
  extends Expr[Boolean]

case class And(a: Boolean, b: Boolean)
  extends Expr[Boolean]

case class FlatMap[A, B](
  a: Expr[A],
  f: A ⇒ Expr[B]) extends Expr[B]
```

```scala
val program: Expr[Boolean] =
  And(Lt(Pure(1), Pure(2)), Lt(Pure(3), Pure(4)))
```

```scala
val program: Expr[Boolean] =
  FlatMap(Pure(1), (a: Int) =>
    FlatMap(Pure(2), (b: Int) =>
      FlatMap(Lt(a, b), (x: Boolean) =>
        FlatMap(Pure(3), (c: Int) =>
          FlatMap(Pure(4), (d: Int) =>
            FlatMap(Lt(c, d), (y: Boolean) =>
              FlatMap(And(x, y), (z: Boolean) =>
                Pure(z))))))))
```

```scala
val program: Expr[Boolean] =
  for {
    a ← pure(1)
    b ← pure(2)
    x ← lt(a, b)
    c ← pure(3)
    d ← pure(4)
    y ← lt(c, d)
    z ← and(a, b)
  } yield z
```

```scala
def eval[A](program: Expr[A]): A =
  program match {
    case Pure(v) ⇒
      v

    case Lt(a, b) ⇒
      a < b

    case And(a, b) ⇒
      a && b

    case FlatMap(a, fn) ⇒
      eval(fn(eval(a)))
  }
```

```scala
def evalAsync[A](program: Expr[A]): Future[A] =
  program match {
    case Pure(v) ⇒
      Future.successful(v)

    case Lt(a, b) ⇒
      Future.successful(a < b)

    case And(a, b) ⇒
      Future.successful(a && b)

    case FlatMap(a, fn) ⇒
      evalAsync(a).flatMap(a ⇒ evalAsync(fn(a)))
  }
```

```scala
def prettyPrint[A](program: Expr[A]): String =
  program match {
    case Pure(v) ⇒
      v.toString

    case Lt(a, b) ⇒
      s"${prettyPrint(a)} < ${prettyPrint(b)}"

    case And(a, b) ⇒
      s"${prettyPrint(a)} && ${prettyPrint(b)}"

    case FlatMap(a, fn) ⇒
      ???
  }
```

```scala
def simplify[A](program: Expr[A]): Expr[A] =
  program match {
    case Pure(v) ⇒
      ???

    case Lt(a, b) ⇒
      ???

    case And(a, b) ⇒
      ???

    case FlatMap(a, fn) ⇒
      ???
  }
```

Generality ⟷ Inspectability

Do we have to write FlatMap ourselves?

Free monadic DSLs

Abstract Pure/FlatMap

```scala
sealed trait Expr[A]

case class Pure[A](value: A) extends Expr[A]

case class Lt(a: Int, b: Int)
  extends Expr[Boolean]

case class And(a: Boolean, b: Boolean)
  extends Expr[Boolean]

case class FlatMap[A, B](
  a: Expr[A],
  f: A ⇒ Expr[B]) extends Expr[B]
```

```scala
sealed trait Expr[A]

case class Lt(a: Int, b: Int)
  extends Expr[Boolean]

case class And(a: Boolean, b: Boolean)
  extends Expr[Boolean]



case class Pure[A](value: A) extends Expr[A]

case class FlatMap[A, B](
  a: Expr[A],
  f: A ⇒ Expr[B]) extends Expr[B]
```

```scala
sealed trait ExprAlg[A]

case class Lt(a: Int, b: Int)
  extends ExprAlg[Boolean]

case class And(a: Boolean, b: Boolean)
  extends ExprAlg[Boolean]
```

```scala
case class Pure[A](value: A) extends Expr[A]

case class FlatMap[A, B](
  a: Expr[A],
  f: A ⇒ Expr[B]) extends Expr[B]
```

```scala
sealed trait ExprAlg[A]

case class Lt(a: Int, b: Int)
  extends ExprAlg[Boolean]

case class And(a: Boolean, b: Boolean)
  extends ExprAlg[Boolean]
```

```scala
sealed trait ExprMonad[A]

case class Pure[A](value: A) extends ExprMonad[A]

case class FlatMap[A, B](
  a: ExprMonad[A],
  f: A ⇒ ExprMonad[B]) extends ExprMonad[B]
```

```scala
sealed trait ExprAlg[A]

case class Lt(a: Int, b: Int)
  extends ExprAlg[Boolean]

case class And(a: Boolean, b: Boolean)
  extends ExprAlg[Boolean]



sealed trait ExprMonad[A]

case class Pure[A](value: A) extends ExprMonad[A]

case class Suspend[A](value: ExprAlg[A])
  extends ExprMonad[A]

case class FlatMap[A, B](
  a: ExprMonad[A],
  f: A => ExprMonad[B]) extends ExprMonad[B]
```

```scala
sealed trait ExprMonad[A]

case class Pure[A](value: A) extends ExprMonad[A]

case class Suspend[A](value: ExprAlg[A])
  extends ExprMonad[A]

case class FlatMap[A, B](
  a: ExprMonad[A],
  f: A => ExprMonad[B]) extends ExprMonad[B]
```

```scala
sealed trait Free[F[_], A]

case class Pure[F[_], A](value: A) extends Free[F, A]

case class Suspend[F[_], A](value: F[A])
  extends Free[F, A]

case class FlatMap[F[_], A, B](
  a: Free[F, A],
  f: A ⇒ Free[F, B]) extends Free[F, B]
```

```scala
sealed trait ExprAlg[A]

case class Lt(a: Int, b: Int)
  extends ExprAlg[Boolean]

case class And(a: Boolean, b: Boolean)
  extends ExprAlg[Boolean]
```

```scala
val program: Expr[Boolean] =
  FlatMap(Pure(1), (a: Int) ⇒
    FlatMap(Pure(2), (b: Int) ⇒
      FlatMap(Suspend(Lt(a, b)), (x: Boolean) ⇒
        FlatMap(Pure(3), (c: Int) ⇒
          FlatMap(Pure(4), (d: Int) ⇒
            FlatMap(Suspend(Lt(c, d)), (y: Boolean) ⇒
              FlatMap(Suspend(And(x, y)), (z: Boolean) ⇒
                Pure(z))))))))
```

```scala
val program: Expr[Boolean] =
  for {
    a ← pure(1)
    b ← pure(2)
    x ← suspend(lt(a, b))
    c ← pure(3)
    d ← pure(4)
    y ← suspend(lt(c, d))
    z ← suspend(and(x, y))
  } yield z
```

```scala
import cats.free.Free

type Expr[A] = Free[ExprAlg, A]
```

```scala
import cats.free.Free

type Expr[A] = Free[ExprAlg, A]

def lit[A](value: A): Expr[A] =
  Free.pure[ExprAlg, A](value)

def lt(a: Int, b: Int): Expr[Boolean] =
  Free.liftF[ExprAlg, Boolean](Lt(a, b))

def and(a: Boolean, b: Boolean): Expr[Boolean] =
  Free.liftF[ExprAlg, Boolean](And(a, b))

def fail[A](msg, String): Expr[A] =
  Free.liftF[ExprAlg, A](Fail(msg))
```

```scala
val program: Expr[Boolean] =
  for {
    a ← lit(1)
    b ← lit(2)
    x ← lt(a, b)
    c ← lit(3)
    d ← lit(4)
    y ← lt(c, d)
    z ← and(x, y)
  } yield z
```

```scala
import cats.arrow.FunctionK

object evalAsync extends FunctionK[ExprAlg, Future] {
  def apply[A](expr: ExprAlg[A]): Future[A] =
    expr match {
     case Lt(a, b) ⇒
        Future.successful(a < b)

      case And(a, b) ⇒
        Future.successful(a && b)
    }
}
```

```scala
val program: Expr[Boolean] =
  for {
    a ← lit(1)
    b ← lit(2)
    x ← lt(a, b)
    c ← lit(3)
    d ← lit(4)
    y ← lt(c, d)
    z ← and(x, y)
  } yield z

program.foldMap(evalAsync)
// res0: Future[Boolean] = Future(…)
```

```scala
import cats.arrow.FunctionK
import cats.Id

object eval extends FunctionK[ExprAlg, Id] {
  def apply[A](expr: ExprAlg[A]): A =
    expr match {
     case Lt(a, b) ⇒
        a < b

      case And(a, b) ⇒
        a && b
    }
}
```

```scala
val program: Expr[Boolean] =
  for {
    a ← lit(1)
    b ← lit(2)
    x ← lt(a, b)
    c ← lit(3)
    d ← lit(4)
    y ← lt(c, d)
    z ← and(x, y)
  } yield z

program.foldMap(eval)
// res0: Boolean = true
```

Free provides sequencing

Algebra provides steps

Combine DSLs

```scala
import cats.data.EitherK
import cats.free.Free

type Alg[A] = EitherK[Alg1, Alg2, A]

type Expr[A] = Free[Alg, A]
```

Lots of boilerplate

(see Freestyle, http://frees.io)

Church encodings

Encode programs
as Scala expressions

Simple
Church encoding

```scala
sealed trait Expr[A]

case class Lit[A](value: A) extends Expr[A]

case class Lt(a: Int, b: Int)
  extends Expr[Boolean]

case class And(a: Boolean, b: Boolean)
  extends Expr[Boolean]
```

```scala
trait ExprDsl {
  def lit[A](n: A): A
  def lt(a: Int, b: Int): Boolean
  def and(a: Boolean, b: Boolean): Boolean
}
```

```scala
def program(dsl: ExprDsl): Boolean = {
  import dsl._

  and(lt(lit(1), lit(2)), lt(lit(3), lit(4)))
}
```

```scala
object Interpreter extends ExprDsl {
  def lit[A](n: A): A =
    n

  def lt(a: Int, b: Int): Boolean =
    a < b

  def and(a: Boolean, b: Boolean): Boolean =
    a && b
}
```

```scala
def program(dsl: ExprDsl): Boolean = {
  import dsl._

  and(lt(lit(1), lit(2)), lt(lit(3), lit(4)))
}

program(Interpreter)
// res0: Boolean = true
```

Can’t abstract over effects

Tagless final encoding

Tagless final encoding

(Finally tagless encoding?)

```scala
trait ExprDsl {
  def lit[A](n: A): A
  def lt(a: Int, b: Int): Boolean
  def and(a: Boolean, b: Boolean): Boolean
}
```

```scala
trait ExprDsl[F[_]] {
  def lit[A](n: A): F[A]
  def lt(a: Int, b: Int): F[Boolean]
  def and(a: Boolean, b: Boolean): F[Boolean]
}
```

```scala
object AsyncInterpreter extends ExprDsl[Future] {
  def lit[A](n: A): Future[A] =
    Future.successful(n)

  def lt(a: Int, b: Int): Future[Boolean] =
    Future.successful(a < b)

  def and(a: Boolean, b: Boolean): Future[Boolean] =
    Future.successful(a && b)
}
```

```scala
import cats.Id

object Interpreter extends ExprDsl[Id] {
  def lit[A](n: A): Id[A] =
    n

  def lt(a: Int], b: Int): Id[Boolean] =
    a < b

  def and(a: Boolean, b: Boolean): Id[Boolean] =
    a && b
}
```

```scala
import cats.Monad
import cats.syntax.flatMap._

def program[F[_]](dsl: ExprDsl[F])
         (implicit monad: Monad[F]): Boolean = {
  import dsl._

  for {
    a ← lit(1)
    b ← lit(2)
    x ← lt(a, b)
    c ← lit(3)
    d ← lit(4)
    y ← lt(c, d)
    z ← and(x, y)
  } yield z
}
```

```scala
import cats.Monad
import cats.syntax.flatMap._

def program[F[_]](dsl: ExprDsl[F])
      (implicit monad: Monad[F]): Boolean = {
  import dsl._

  for {
    a ← lit(1)
    b ← lit(2)
    x ← lt(a, b)
    c ← lit(3)
    d ← lit(4)
    y ← lt(c, d)
    z ← and(x, y)
  } yield z
}

program(AsyncInterpreter)
// res0: Future[Boolean] = Future(…)
```

```scala
import cats.Monad
import cats.syntax.flatMap._

def program[F[_]](dsl: ExprDsl[F])
         (implicit monad: Monad[F]): Boolean = {
  import dsl._

  for {
    a ← lit(1)
    b ← lit(2)
    x ← lt(a, b)
    c ← lit(3)
    d ← lit(4)
    y ← lt(c, d)
    z ← and(x, y)
  } yield z
}

program(Interpreter)
// res0: Boolean = true
```

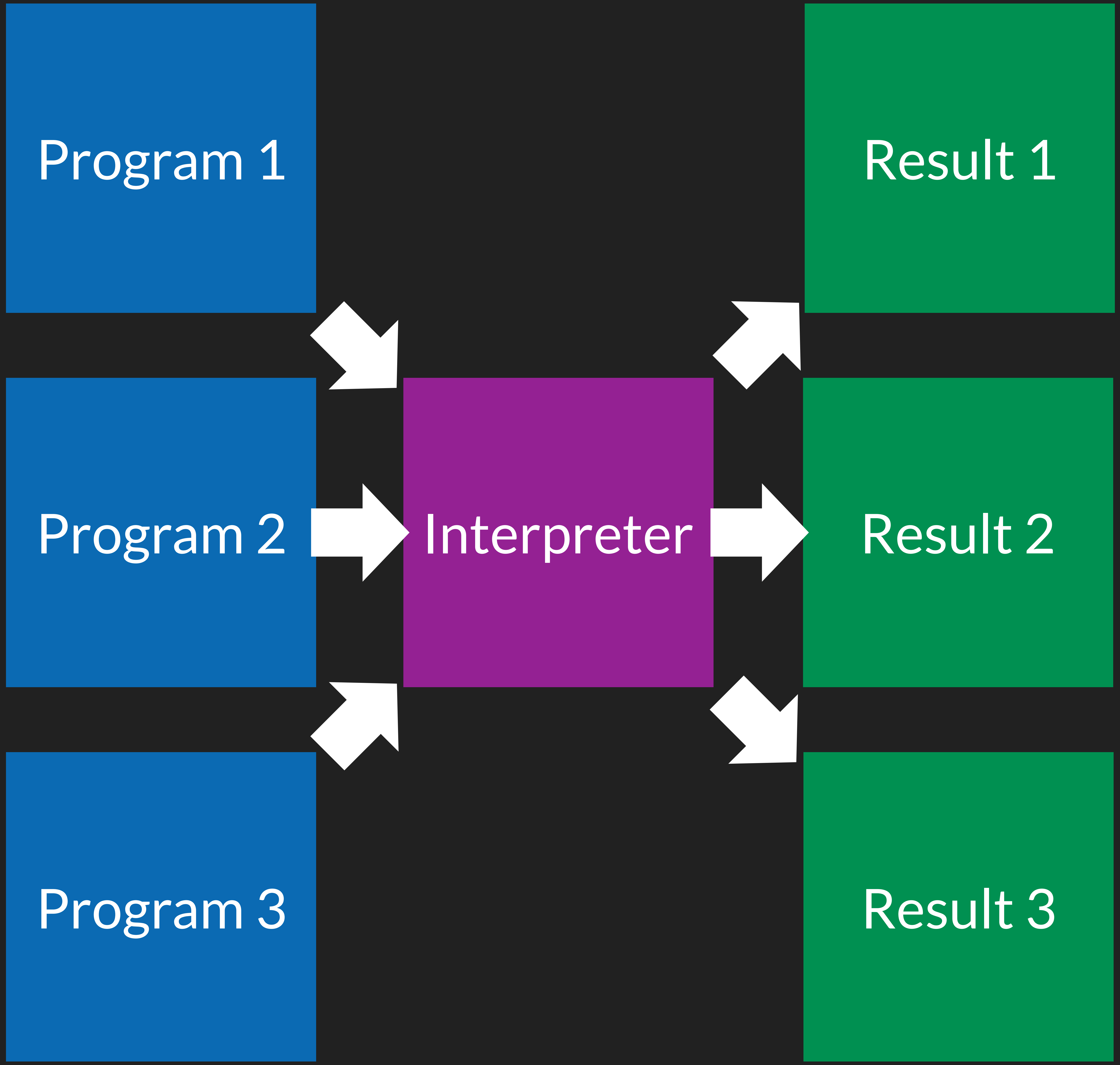
Combine DSLs

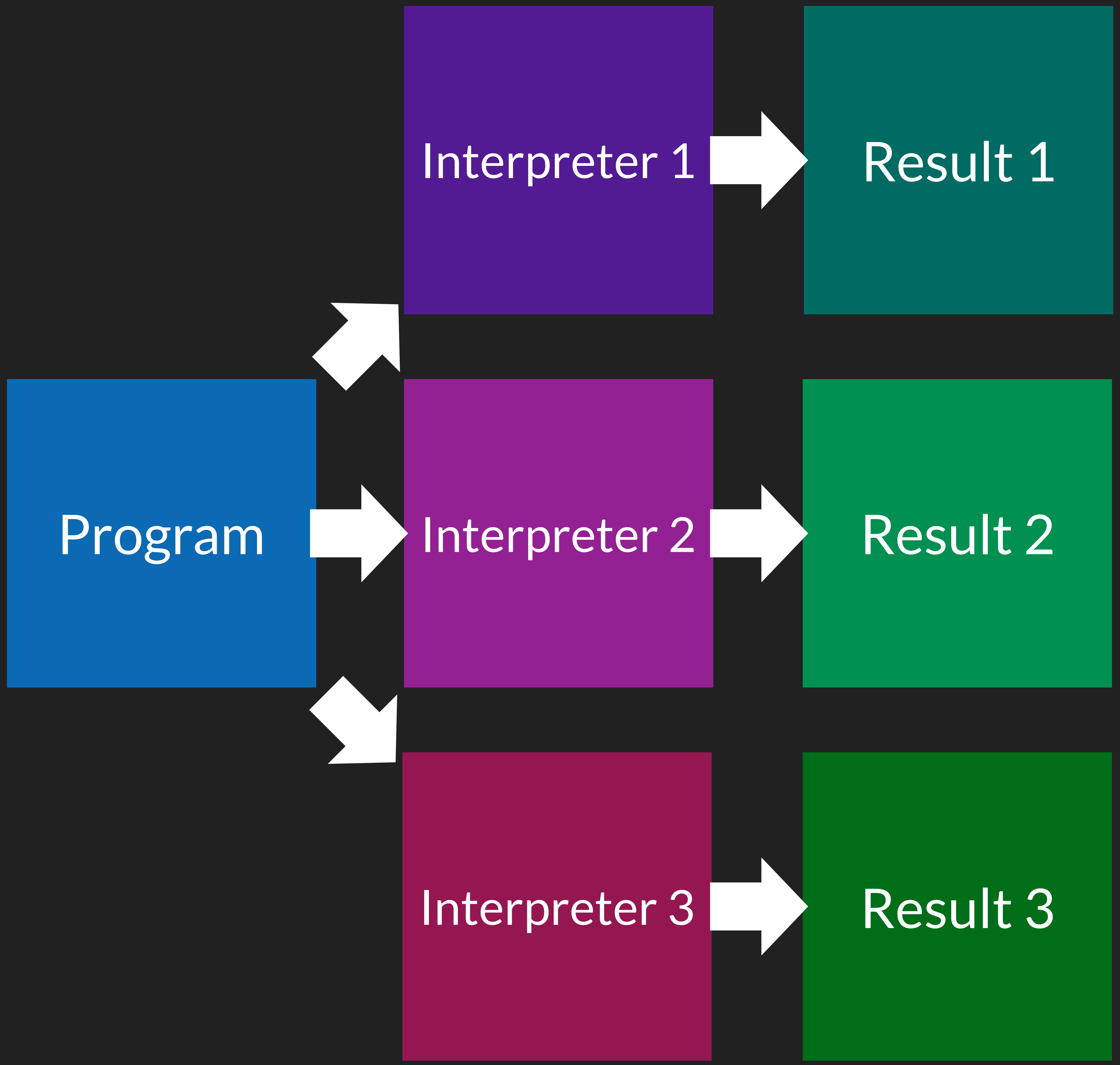
```scala
def program[F[_], A](dsl1: Dsl1[F], dsl2: Dsl2[F])
    (implicit monad: Monad[F]): A = {
  import dsl1._
  import dsl2._

  // ...
}
```

Less boilerplate than Free

Summary


Program
Interpreter
Result


Program 1
Program 2
Program 3
Interpreter
Result 1
Result 2
Result 3


Program
Interpreter 1
Interpreter 2
Interpreter 3
Result 1
Result 2
Result 3

Reification ➡ Free

Model code as data

Church encoding ➡ Tagless final

Model code as … code

Standalone

Completely custom

Embedded

Reuse host language features

Constraints liberate, liberty constrains

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

Free

https://underscore.io/blog/posts/2015/04/14/free-monads-are-simple.html

Free with multiple algebras

https://underscore.io/blog/posts/2017/03/29/free-inject.html

Tagless final

https://skillsmatter.com/skillscasts/10007-free-vs-tagless-final-with-chris-birchall

Slides, code samples, and notes (WIP)

https://github.com/underscoreio/interpreters-and-you

Thank you

https://github.com/underscoreio/interpreters-and-you

Dave Gurnell, @davegurnell

Quick promo…


scala
eXchange
EUROPE'S LARGEST ANNUAL
SCALA CONFERENCE RETURNS
@skillsmatter
13 - 14 Dec
2018

13-14 December 2018

Call for papers closes 14 June

Diversity scholarship plan
Free support for all accepted speakers

http://scala-exchange.com