

Interpreters and you

Mark Mynsted, @mmynsted
Dave Gurnell, @davegurnell



underscore

Interpreters and you

Mark Mynsted, @mmynsted
Dave Gurnell, @davegurnell



underscore

Interpreters and you

Dave Gurnell, @davegurnell
Mark Mynsted, @mmynsted



underscore

Interpreters and you

Mark Mynsted, @mmynsted
Dave Gurnell, @davegurnell



underscore

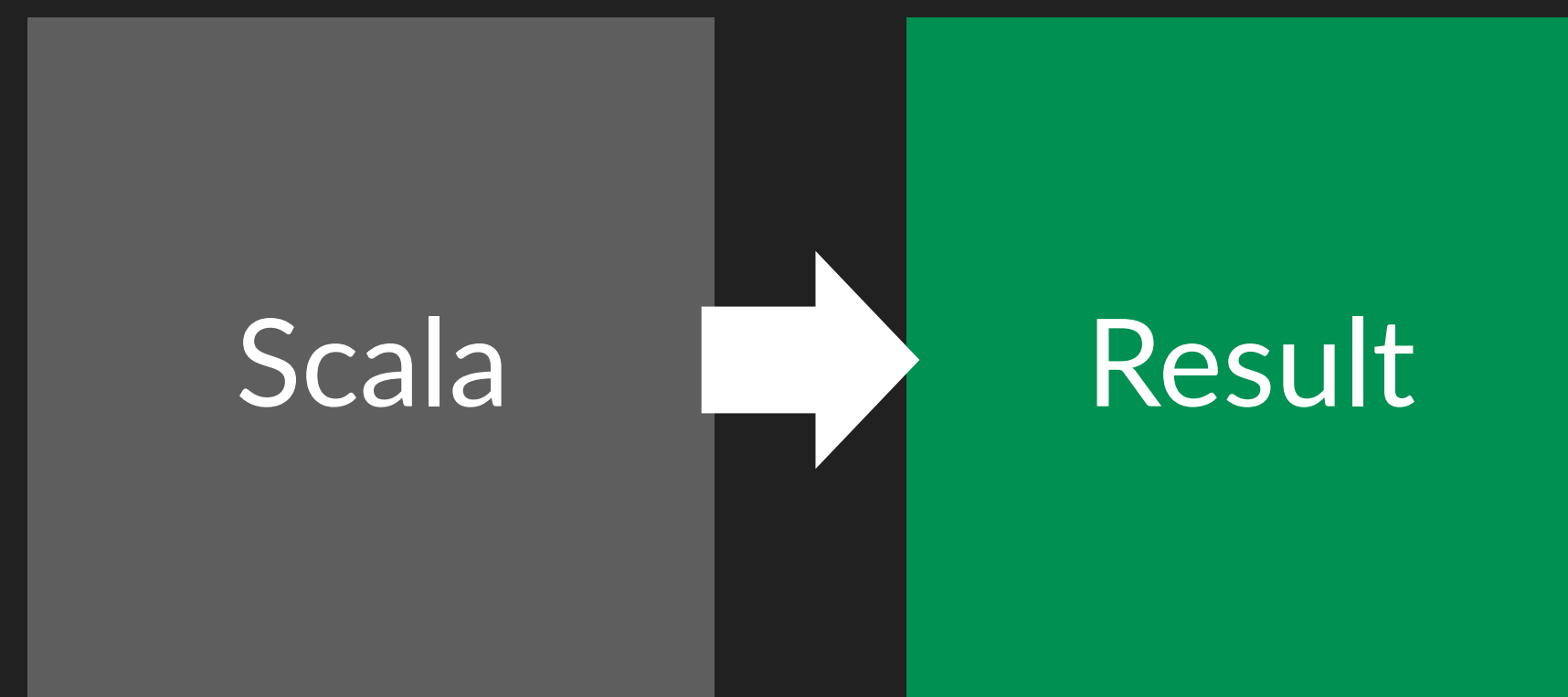


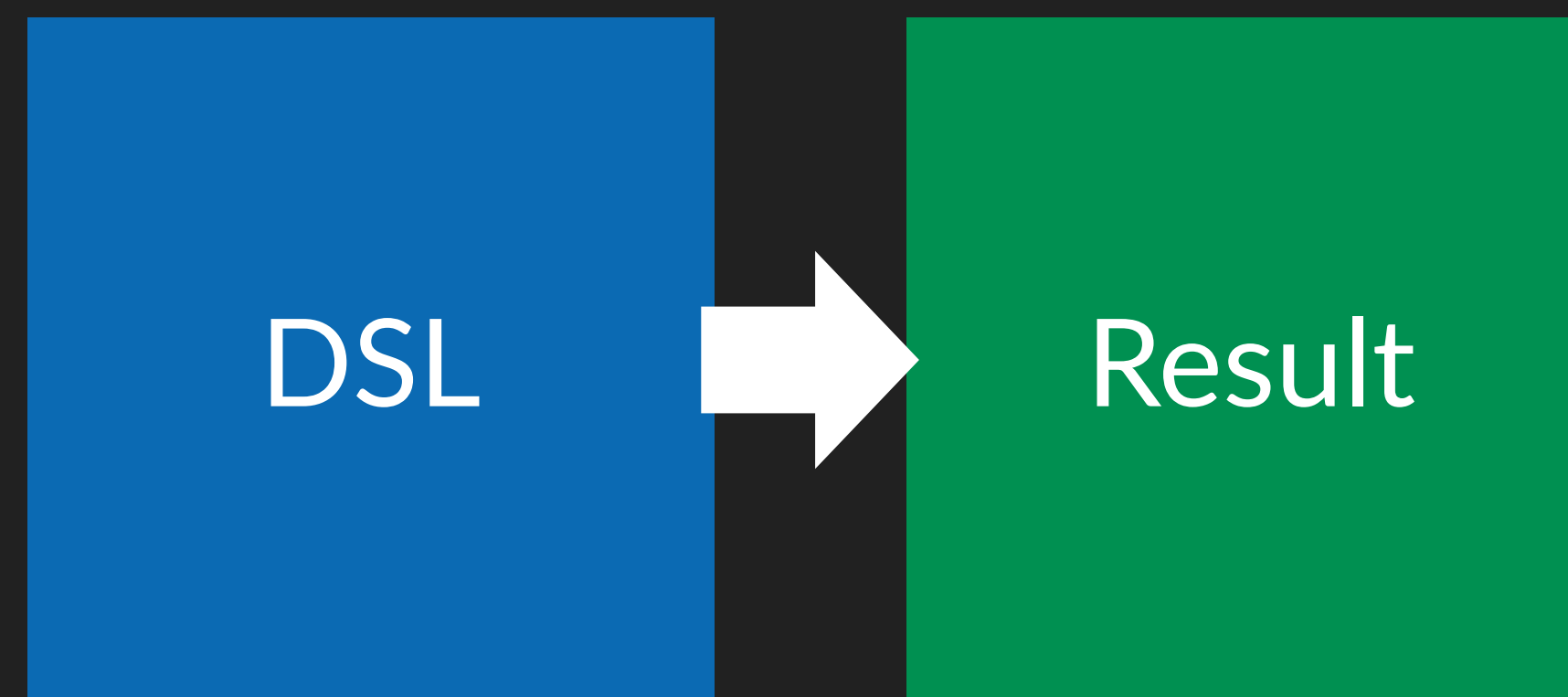
underscore



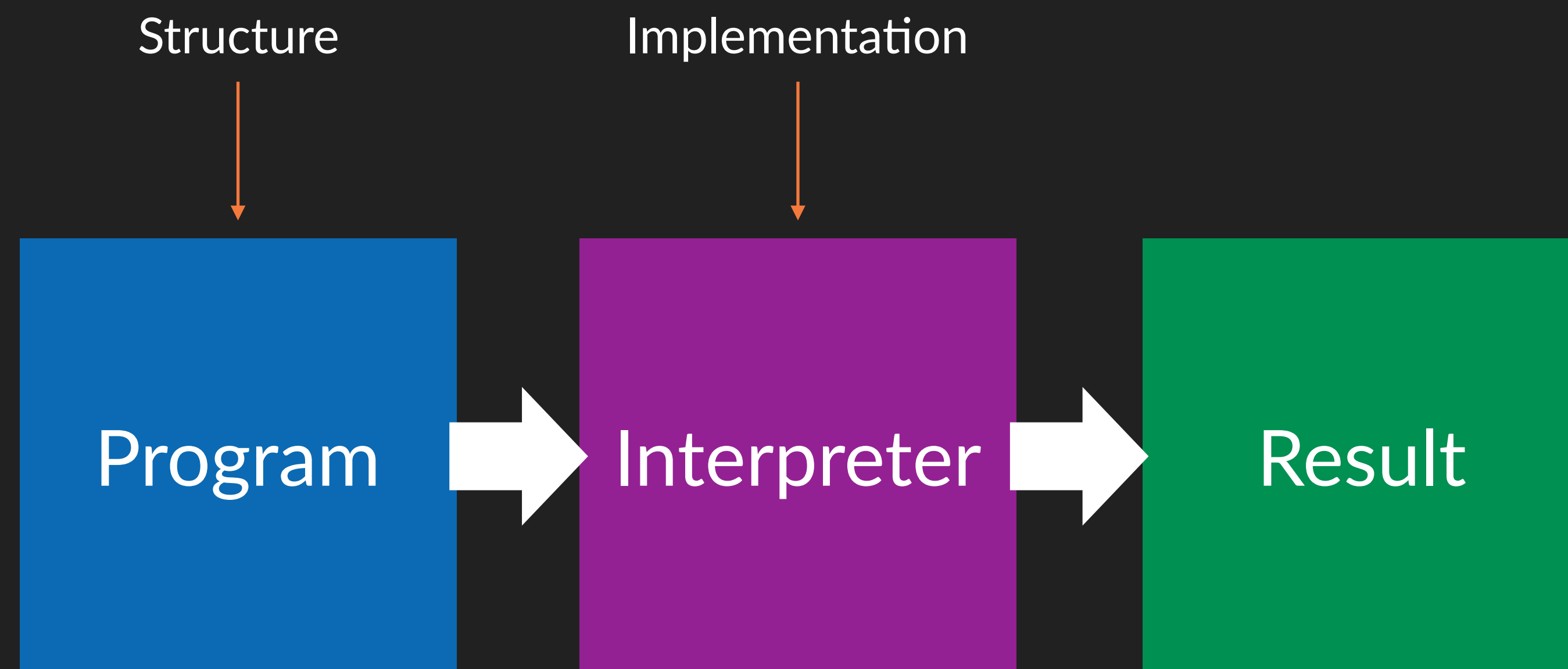
underscore

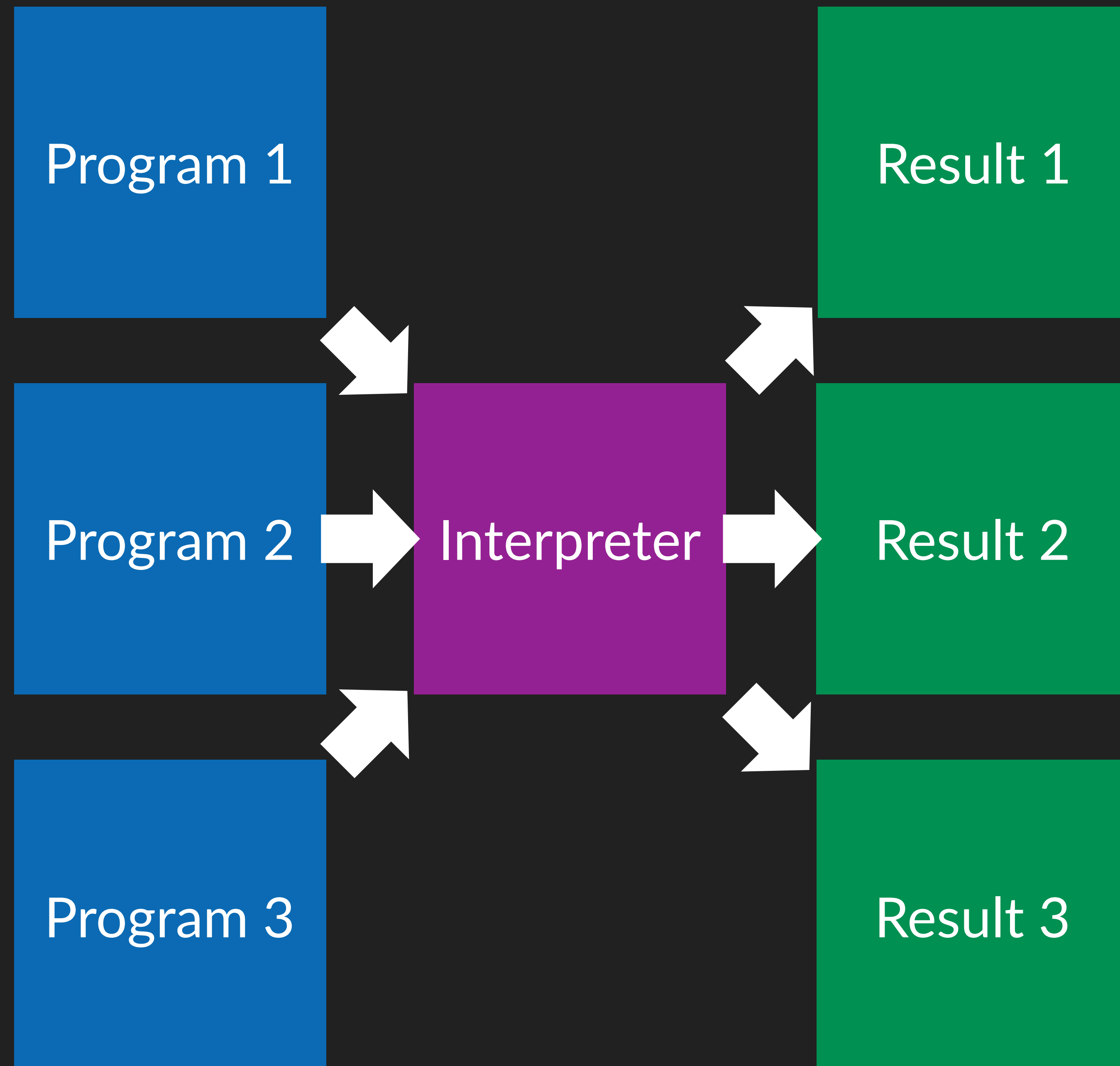
Scala

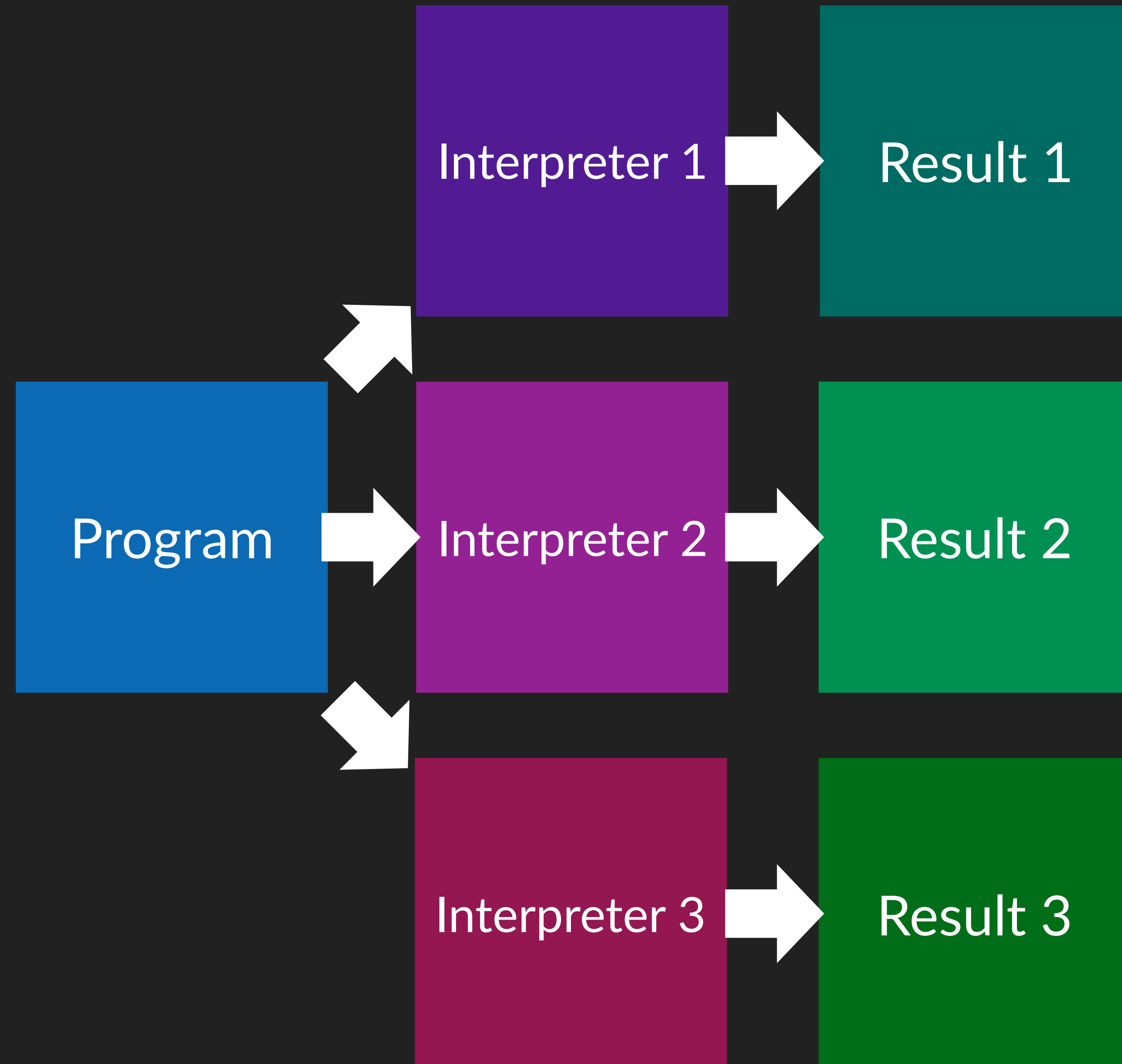


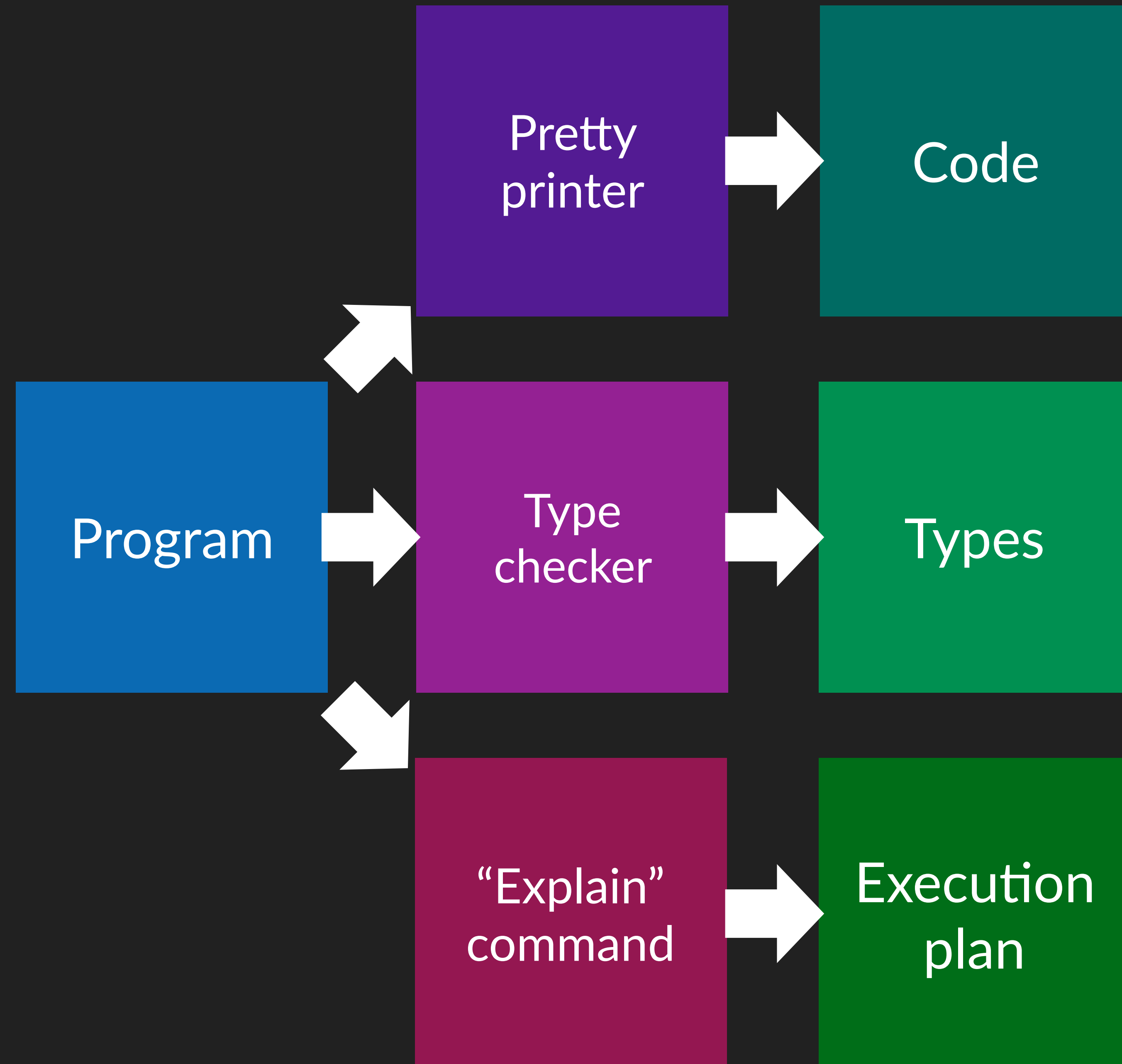


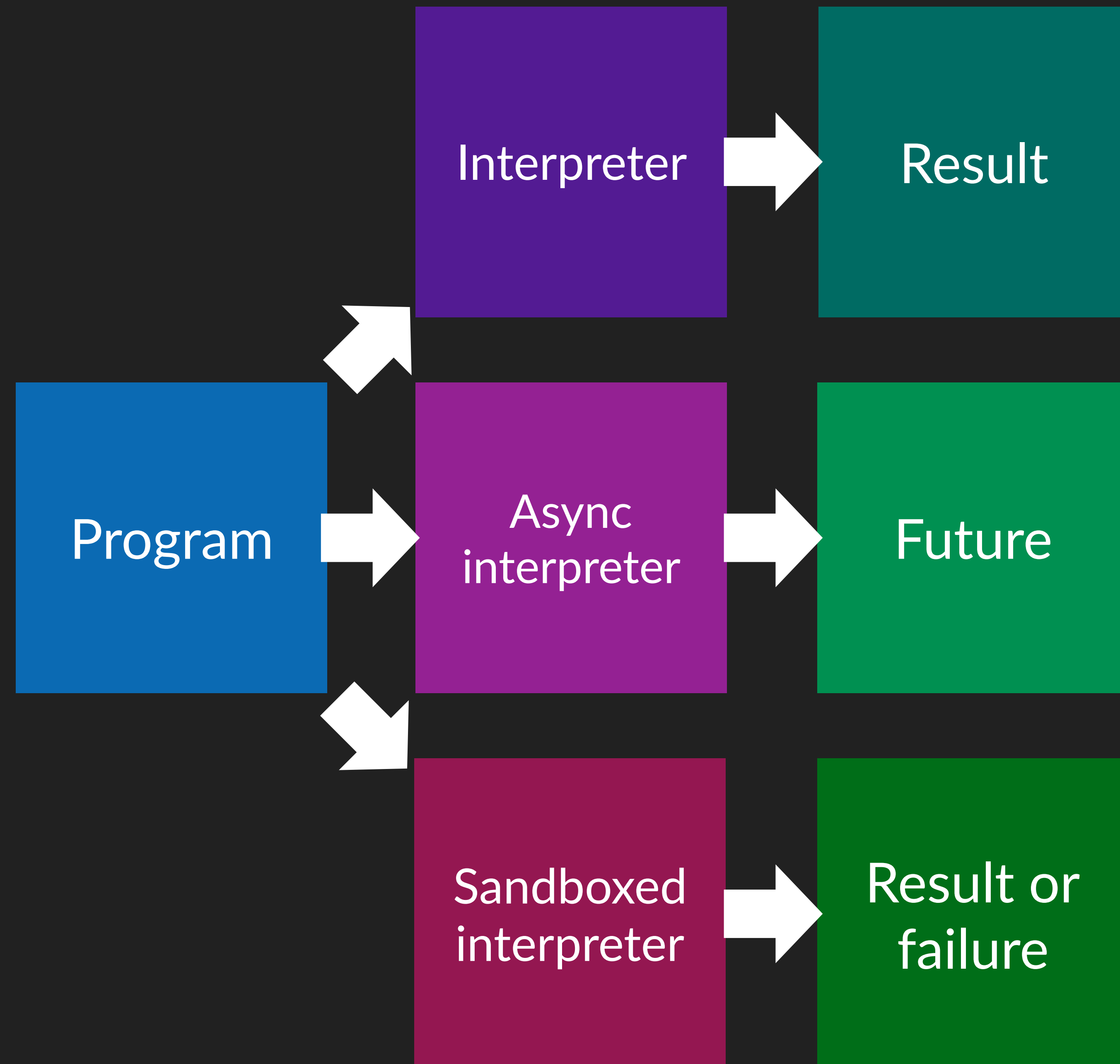


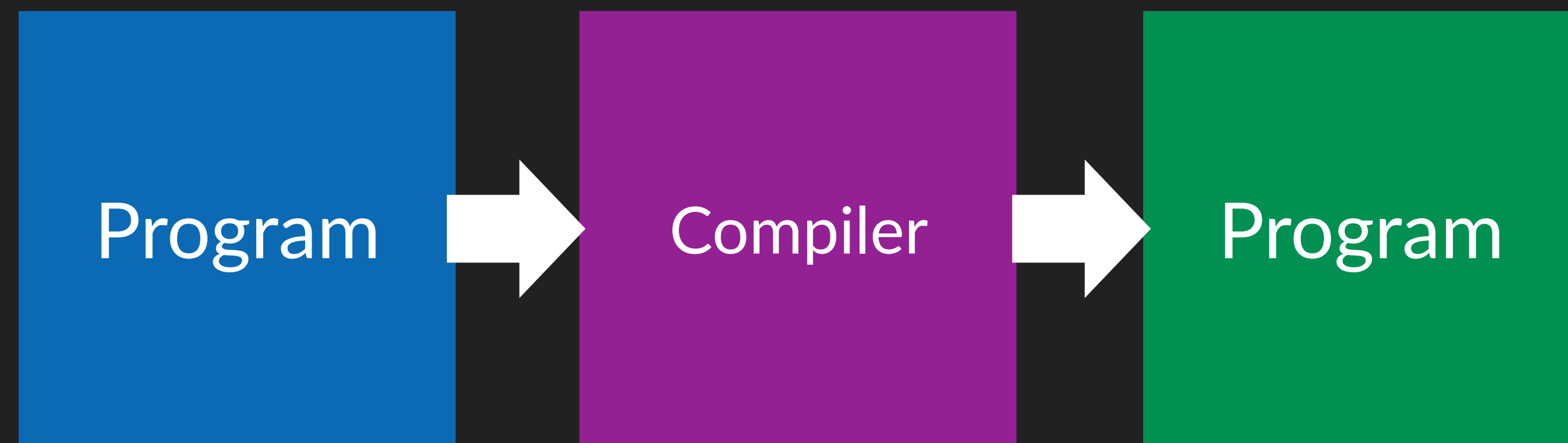












1. Re-use interpreters

2. Multiple interpreters

3. Abstract over effects

(4. Inspect / rewrite / compile programs)

Big approaches

Reification

Model programs as data

Church encoding

Model programs as sets of method calls

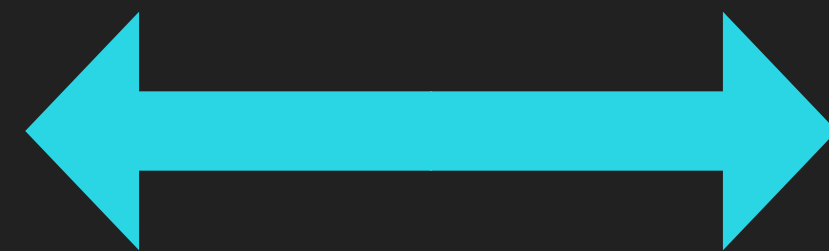
Reification → Free

Model programs as data

Church encoding → Tagless final

Model programs as sets of method calls

Standalone



Embedded

Standalone



Embedded

Completely custom

Independent of host language

Align with host language

Reuse host language features

Reified encodings

A calculated example

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

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



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

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

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

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

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

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

```
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
```



```
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)))
```

```
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
```

```
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)  
  }
```

```
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))  
  }
```




```
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)}"  
  }
```

```
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

A man with short, wavy brown hair, glasses, and a light beard is speaking at a podium. He is wearing a dark grey button-down shirt. His hands are raised in a gesturing motion. A black microphone on a stand is positioned in front of him. The background is a plain white wall with a green geometric shape on the left side.

Constraints liberate
Liberty constrains

What about types?

Untyped DSLs

1 < 2 ∞ 3 < 4

```
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)))
```



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

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

Boolean or integer expression ?



```
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) =>  
      ???
```



Model errors here ?

```
    case Lt(a, b) =>  
      ???
```

```
    case And(a, b) =>  
      ???
```

```
}
```



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?


```
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?

```
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

```
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
```



```
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]
```

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

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

```
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)  
  }
```

```
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))  
  }
```

```
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)}"  
  }
```

```
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

```
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!

Monadic DSLs

```
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]
```



```
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]
```

```
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]
```

```
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]
```

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

```
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)))))))
```

```
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
```

```
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)))  
  }
```

```
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)))  
  }
```



```
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) =>  
      ???  
  }
```

```
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

```
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]
```

```
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  $\Rightarrow$  Expr[B]) extends Expr[B]
```



```
sealed trait ExprAlg[A]

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

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

```
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 FlatMap[A, B](  
  a: ExprMonad[A],  
  f: A  $\Rightarrow$  ExprMonad[B]) extends ExprMonad[B]
```

```
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]
```

```
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  $\Rightarrow$  ExprMonad[B]) extends ExprMonad[B]
```

```
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  $\Rightarrow$  Free[F, B]) extends Free[F, B]
```

```
sealed trait ExprAlg[A]
```

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

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

```
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)))))))
```

```
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
```



```
import cats.free.Free  
  
type Expr[A] = Free[ExprAlg, A]
```

```
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))
```

```
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
```

```
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)
    }
}
```

```
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(...)
```

```
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
    }
}
```

Free provides sequencing

Algebra provides steps

Combine DSLs


```
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

```
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]
```

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

```
def program(dsl: ExprDsl): Boolean = {  
    import dsl._  
  
    and(lt(lit(1), lit(2)), lt(lit(3), lit(4)))  
}
```



```
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  
}
```

```
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?)

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

```
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]  
}
```

```
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)  
}
```



```
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
}
```

```
import cats.Monad
import cats.syntax.functor._
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
}
```

```
import cats.Monad
import cats.instances.future._
import cats.syntax.functor._
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(Success(true))
```

```
import cats.Monad
import cats.syntax.functor._
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: Id[Boolean] = true
```

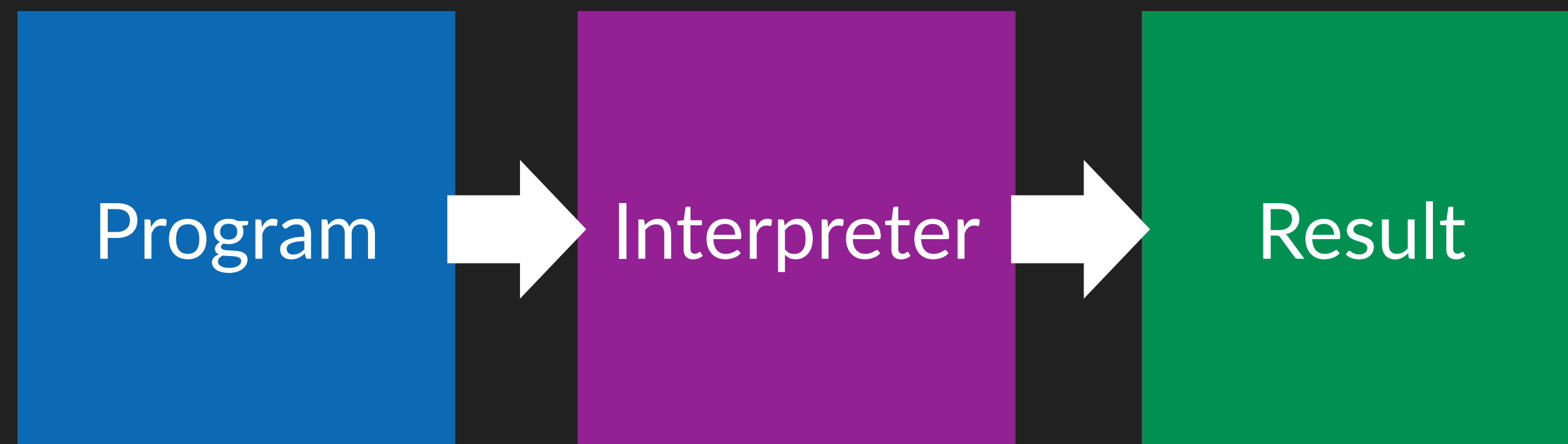
Combine DSLs

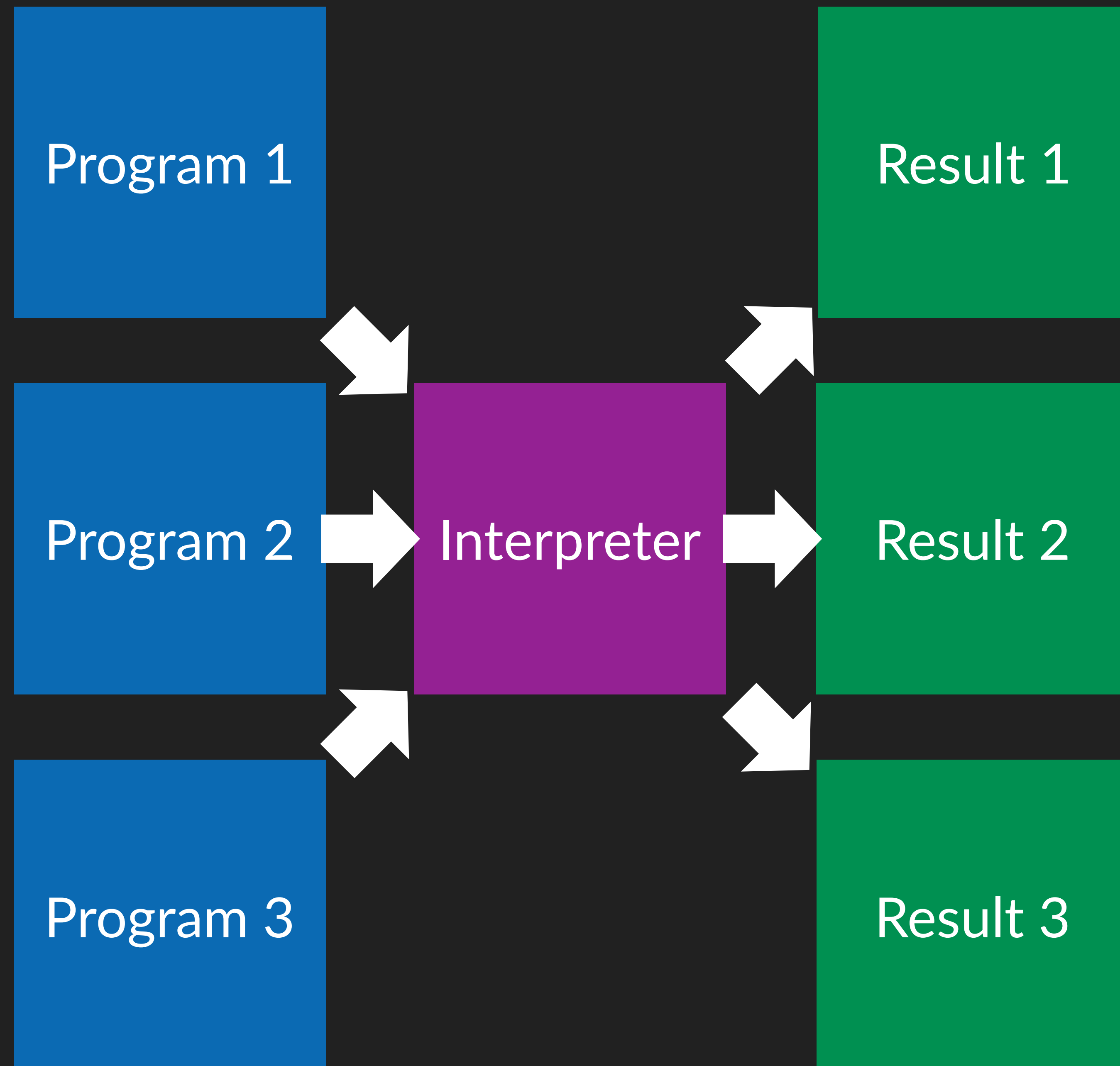
```
def program[F[_], A](dsl1: Dsl1[F], dsl2: Dsl2[F])  
  (implicit monad: Monad[F]): A = {  
    import dsl1._  
    import dsl2._  
  
    // ...  
  }
```

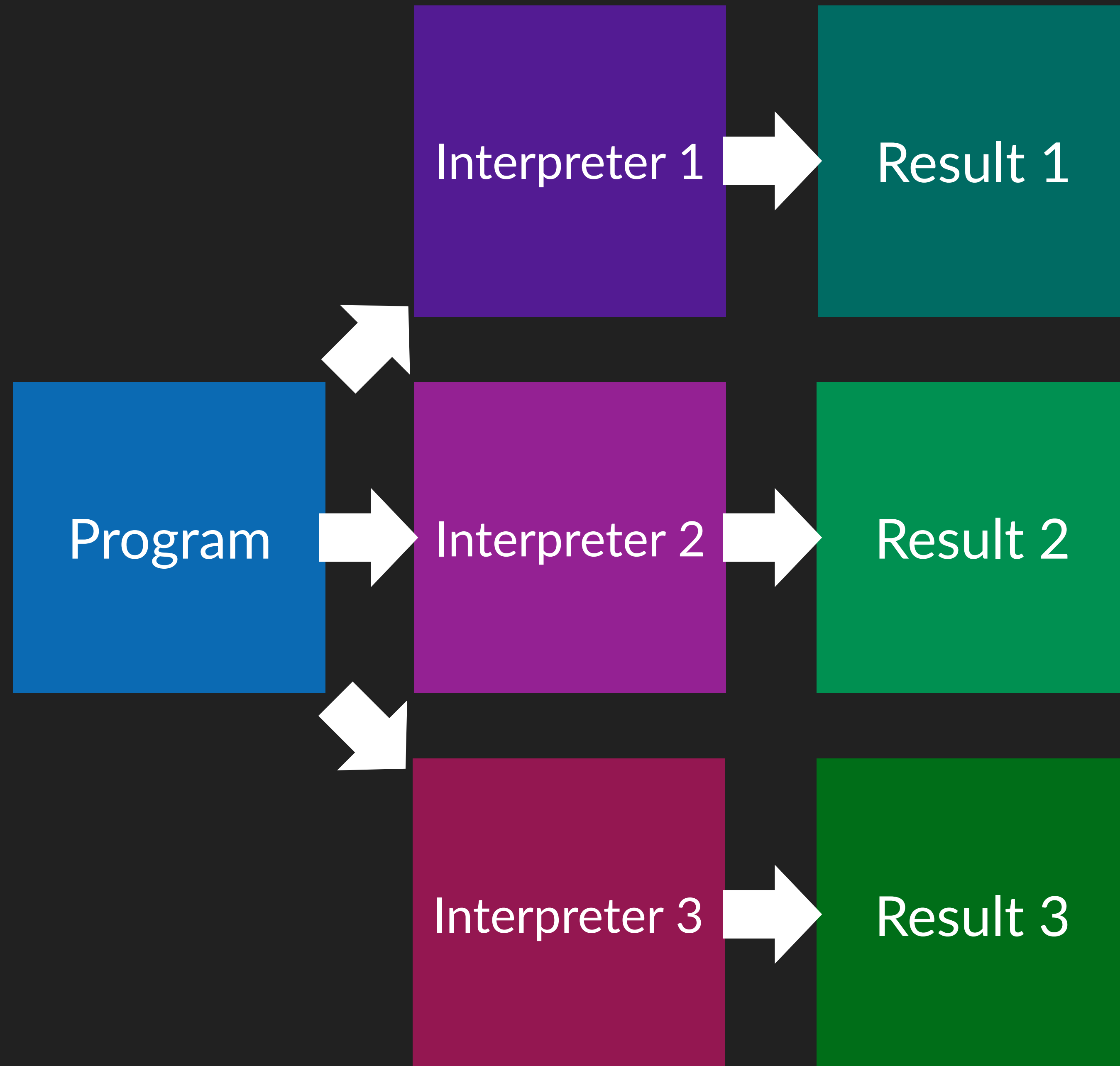
Difficult to inspect

Less boilerplate than Free

Summary







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](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](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](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](https://github.com/underscoreio/interpreters-and-you)

Thank you

[https://github.com/underscoreio/
interpreters-and-you](https://github.com/underscoreio/interpreters-and-you)

Dave Gurnell, @davegurnell
Mark Mynsted, @mmynsted