The Term type

```
trait Term
case class Con(a: Int) extends Term
case class Div(t: Term, u: Term) extends Term
```

Test data

```
val answer: Term =
  Div(
    Div(
      Con(1932),
      Con(2)
    ),
    Con(23)
  )

val error: Term =
  Div(
    Con(1),
    Con(0)
  )
```

Variation zero: The basic evaluator

```
def eval(t: Term): Int =
  t match {
    case Con(a)    => a
    case Div(t, u) => eval(t) / eval(u)
  }
```

Using the evaluator

```
scala> val a = eval(answer)
a: Int = 42

scala> val b = eval(error)
java.lang.ArithmeticException: / by zero
  at monads.Eval0$.eval(Term.scala:29)
```

Variation one: Exceptions

```scala
type Exception = String
trait M[A]
case class Raise[A](e: Exception) extends M[A]
case class Return[A](a: A) extends M[A]
```

Variation one: Exceptions, cont’d

```
def eval[A](t: Term): M[Int] =
  t match {
    case Con(a) => Return(a)
    case Div(t, u) => eval(t) match {
      case Raise(e) => Raise(e)
      case Return(a) =>
        eval(u) match {
          case Raise(e) => Raise(e)
          case Return(b) =>
            if (b == 0)
              Raise("divide by zero")
            else
              Return(a / b)
        }
    }
  }
```

Using the evaluator

```
scala> val m = eval(answer)
m: M[Int] = Return(42)

scala> val n = eval(error)
n: M[Int] = Raise(divide by zero)
```

Variation two: State

```
type State = Int
type M[A] = State => (A, State)

def eval(s: Term): M[Int] =
  s match {
    case Con(a) =>
      x => (a, x)
    case Div(t, u) =>
      x =>
        val (a, y) = eval(t)(x)
        val (b, z) = eval(u)(y)
        (a / b, z + 1)
 }
```

Using the evaluator

```
scala> val m = eval(answer)(0)
m: (Int, State) = (42,2)
```

Variation three: Output

```
type Output = String
type M[A] = (Output, A)

def eval(s: Term): M[Int] =
  s match {
    case Con(a) => (line(s, a), a)
    case Div(t, u) =>
      val (x, a) = eval(t)
      val (y, b) = eval(u)
      (x + y + line(s, a / b), a / b)
  }

def line(t: Term, a: Int): Output =
  t + "=" + a + "\n"
```

Using the evaluator

```
scala> val m = eval(answer)
m: (Output, Int) =
("Con(1932)=1932
Con(2)=2
Div(Con(1932),Con(2))=966
Con(23)=23
Div(Div(Con(1932),Con(2)),Con(23))=42
",42)
```

Monads

What is a monad?

1. For each type `A` of *values*,
   a type `M[A]` to represent *computations.*

   In general, `A => B` becomes `A => M[B]`
   In particular, `def eval(t: Term => Int)`
   becomes `def eval(t: Term): M[Int]`

2. A way to turn values into computations.

   ```
   def pure[A](a: A): M[A]
   ```

3. A way to combine computations.

   ```
   def bind[A](m: M[A], k: A => M[B]): M[B]
   ```

Monad laws

Left pure

```
bind(a, pure(identity)) ≡ a
```

Right pure

```
bind(m, (a: A) => pure(a)) ≡ m
```

Associative

```
bind(bind(m, f), g) ≡
bind(m, (a: A) => bind(f(a))(g))
```

The evaluator revisited

Monadic evaluator

```
def eval(s: Term): M[Int] =
  s match {
    case Con(a) =>
      pure(a)
    case Div(t, u) =>
      bind(eval(t), (a: Int) =>
      bind(eval(u), (b: Int) =>
      pure(a / b)))
  }
```

Variation zero revisited: Identity

```scala
type M[A] = A

def pure[A](a: A): M[A] = a

def bind[A, B](a: M[A], k: A => M[B]): M[B] = k(a)
```

Variation one revisited: Exceptions

```scala
type Exception = String
trait M[A]
case class Raise[A](e: Exception) extends M[A]
case class Return[A](a: A) extends M[A]

def pure[A](a: A): M[A] = Return(a)

def bind[A, B](m: M[A], k: A => M[B]): M[B] =
  m match {
    case Raise(e)  => Raise(e)
    case Return(a) => k(a)
  }

def raise[A](e: String): M[A] = Raise(e)
```

Modifying the evaluator

```
def eval(s: Term): M[Int] =
  s match {
    case Con(a) =>
      pure(a)
    case Div(t, u) =>
      bind(eval(t), (a: Int) =>
      bind(eval(u), (b: Int) =>
        if (b == 0)
          raise("divide by zero")
        else
          pure(a / b)
      ))
  }
```

Variation two, revisited: State

```
type State = Int
type M[A] = State => (A, State)

def pure[A](a: A): M[A] = x => (a, x)

def bind[A, B](m: M[A], k: A => M[B]): M[B] =
  x => {
    val (a, y) = m(x)
    val (b, z) = k(a)(y)
    (b, z)
  }

def tick: M[Unit] = (x: Int) => ((), x + 1)
```

Modifying the evaluator

```
def eval(s: Term): M[Int] =
  s match {
    case Con(a) =>
      pure(a)
    case Div(t, u) =>
      bind(eval(t), (a: Int) =>
      bind(eval(u), (b: Int) =>
      bind(tick, (_: Unit) =>
      pure(a / b))))
  }
```

Variation three, revisited: Output

```scala
type Output = String
type M[A] = (Output, A)

def pure[A](a: A): M[A] = ("", a)

def bind[A, B](m: M[A], k: A => M[B]): M[B] = {
  val (x, a) = m
  val (y, b) = k(a)
  (x + y, b)
}

def output[A](s: String): M[Unit] = (s, ())
```

Modifying the evaluator

```
def eval(s: Term): M[Int] =
  s match {
    case Con(a) =>
      bind(output(line(s, a)), (_: Unit) =>
      pure(a))
    case Div(t, u) =>
      bind(eval(t), (a: Int) =>
      bind(eval(u), (b: Int) =>
      bind(output(line(s, a / b)), (_: Unit) =>
      pure(a / b))))
  }

def line(r: Term, a: Int): Output =
  r + "=" + a + "\n"
```

Lists

```scala
type M[A] = List[A]

def pure[A](a: A): M[A] = List(a)

def bind[A, B](m: M[A], k: A => M[B]): M[B] =
  m match {
    case Nil    => Nil
    case h :: t => k(h) ++ bind(t, k)
  }

def zero[A]: M[A] = Nil

def plus[A](m: M[A], n: M[A]): M[A] = m ++ n
```

Streams

```
type M[A] = Stream[A]

def pure[A](a: A): M[A] = Stream(a)

def bind[A, B](m: M[A], k: A => M[B]): M[B] =
  m match {
    case Stream() => Stream()
    case h #:: t  => k(h) ++ bind(t, k)
  }

def zero[A]: M[A] = Stream()

def plus[A](m: M[A], n: M[A]): M[A] = m ++ n
```

Cartesian products

```
def product[A, B](m: M[A], n: M[B]): M[(A, B)] =
  bind(m, (a: A) =>
  bind(n, (b: B) =>
  pure((a, b))))

// using List or Stream's for comprehension syntax
def productFor[A, B](m: M[A], n: M[B]): M[(A, B)] =
  for {
    a <- m
    b <- n
  } yield (a, b)
```

Scala translations with help from:

Tony Morris & Jed Wesley-Smith

Typesetting of new slides:

Jed Wesley-Smith

Scala source available:

https://bitbucket.org/jwesleysmith/yow-monads