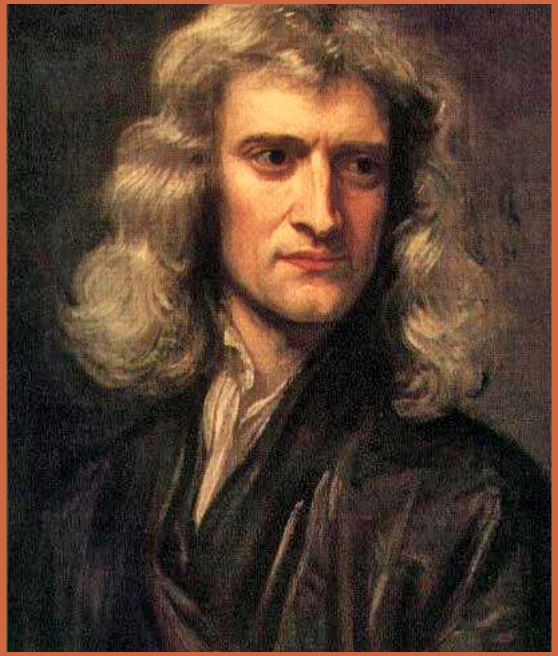
"If I have seen further it is by standing <u>on the shoulders of giants</u>" -- Sir Isaac Newton (*).

Rethinking the Art of Writing Code



(*) Ironically the "on the shoulders of giants" metaphor originated from Bernard of Chartres (a twelfth-century French Neo-Platonist philosopher, scholar, and administrator)

Realizing the power of

Functional

Programming

— Functions a first-class entities
— Support for HOFs, closures, currying
— Immutability
— Composability
— Recursion
— Referential Transparency
— Types

Strictly speaking: this is all we need!

We've been here before

```
loop:   addi    $t1, $t1, 1
add $t2, $t2, $t1
beq $t0, $t1, exit
j   loop

exit:   li  $v0, 4
la  $a0, msg2
syscall
```

Sure: this may be "all we need", but ...

Structured Imperative programming provides us with *constructs*

- IF-ELSE statements
- WHILE Loops
- Named PROCEDURES
- Variables

Where are the constructs for FP?

We've got Functors

```
map: F[A] ⇒ F[B]
```

We've got Monads

```
point/pure/unit:    A ⇒ M[A]
bind/flatMap: (M[A], f: A ⇒ M[B]) ⇒ M[B]
```

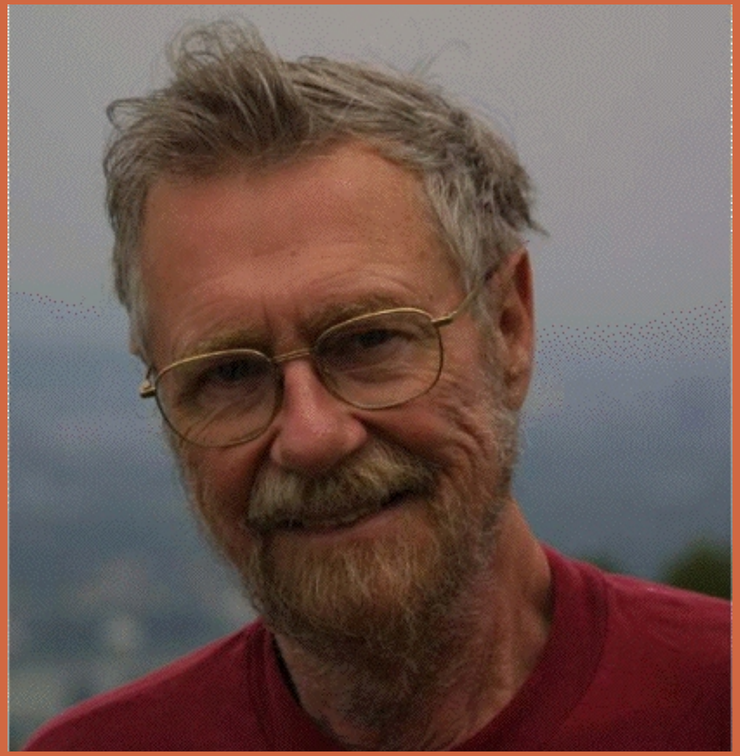
We've got Monoids and Semigroups

psst ... a Semigroup is simply a Monoid without zero

```
append:    (M, M) => M
zero:      M
```

"Being abstract is something profoundly different from being vague … 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. Dijkstra



sum
types

A look at `scala.Either`

```scala
def mean(l: List[Int]) : Either[String, Double] =
  l match {
    case Nil => Left("no values provided")
    case _ :: _ => Right(l.sum / l.size)
}
```

what if we want

`mean(12 :: 6 :: 7 :: Nil) + mean(9 :: 3 :: Nil)`**?**

```scala
val x = mean(12 :: 6 :: 7 :: Nil)
val y = mean(9 :: 3 :: Nil)

(x, y) match {
  case (Left(err1), Left(err2)) => Left(s"$err1, $err2")
  case (Left(err), _) => Left(s"$err")
  case (_, Left(err)) => Left(s"$err")
  case (Right(l), Right(r)) => Right(l + r)
}
```

In Conclusion `scala.Either` **is a very weak abstraction.**

... and lets not even get started on Exceptions!

What we need is a **true sum type**

What do we mean by a sum type?

Consider a simple `product` **definition:**

```scala
case class Fruit(name: String, color: String)
```

Let's say our possible values are: `(Apple, Mango)` **and** `(Green, Red)`

So we take the (cartesian) `product` **of** `fruit X color` **and we have:**

```scala
Fruit(Apple, Green)
Fruit(Apple, Red)
Fruit(Mango, Green)
Fruit(Mango, Red)
```

A product is a *conjunction* of different types values ANDed together (e.g. case class)

A sum is a *disjunction of values ORed together (e.g. it can be a value of* *one type** or another)

A *sum* is a known as a "dual" of product, hence some people may refer to a *sum* as a *coproduct* (which means the exact same thing).

`scala.Either` **is a poor sum type because composing them requires you to break out of the abstraction of this type and manually handle all case combinations.**

Both cats and scalaz support *Disjunctions*, which compose far more elegantly

In scalaz

```
import scalaz.{\/, -\/, \/-}

def mean(l: List[Int]) : String \/ Double =
  l match {
    case Nil => -\/("no values provided")
    case _ :: _ => \/-(l.sum / l.size)
}
```

In cats

```
import cats.data.Xor

def mean(l: List[Int]) : String Xor Double =
  l match {
    case Nil => Xor.Left("no values provided")
    case _ :: _ => Xor.Right(l.sum / l.size)
}
```

```scala
val x = mean(12 :: 6 :: 7 :: Nil)
val y = mean(9 :: 3 :: Nil)

for {
  a <- x
  b <- y
} yield a + b

// the for-comprehension is equivalent to ...

x.flatMap { xv => y.flatMap { yv => Xor.Right(xv + yv) } } // in cats

// As these are chained flatMaps, the first one that fails short-circuits
// and an error is returned.

// We can get away with this because Disjunctions are Monadic on the right-side
// (right-biased) leaving the left-side most suitable for error conditions.
```

Pattern-matching also works

```
def handleResult[T](result: String \/ T) : Unit = result match {
  case -\/(error) => println(s"Error: $error")
  case \/-(value) => println(s"Value is $value")
}
```

We can wrap exceptions too

```scala
def mayFail : Int = ...

// scalaz

\/.fromTryCatchThrowable[Int, Throwable](mayFail) match {
  case -\/(th) => println(th.getMessage)
  case \/-(v) => println(v)
}

// cats

Xor.catchOnly[Throwable](mayFail) match {
  case Xor.Left(th) => println(th.getMessage)
  case Xor.Right(v) => println(v)
}
```

Let's revisit our sum of means

```
val x = mean(12 :: 6 :: 7 :: Nil)
val y = mean(9 :: 3 :: Nil)

for {
  a <- x
  b <- y
} yield a + b
```

- works by chaining `flatMaps` and applying a `map`
- a very generic (read: powerful) approach
- the cost of this power is *short-circuiting behavior*

"Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job."

—Jerome Saltzer, Communications of the ACM

"The 'Rule of Least Power' suggests choosing the least powerful language suitable for a given purpose."

—https://www.w3.org/2001/tag/doc/leastPower.html

```scala
val x = mean(12 :: 6 :: 7 :: Nil)
val y = mean(9 :: 3 :: Nil)

// Adding x and y
```

A functor `F[A] ⇒ F[B]` **is not powerful enough**

A monad `A ⇒ M[A], (M[A], f: A ⇒ M[B]) ⇒ M[B]` **is too powerful**

We need something in-between

Applicative Functors

```scala
val x = mean(12 :: 6 :: 7 :: Nil)
val y = mean(9 :: 3 :: Nil)

// What we want is something like this:

def someDualFunctor(mean1: F[Int], mean2: F[Int])
  (f: F[(Int, Int) => Double]) : F[Double] = ...

someDualFunctor(x, y)((a: Int, b: Int) => a + b)
```

```scala
trait Apply[F[_]] extends Functor[F] {
  def ap[A, B](fa: F[A])(f: F[A => B]): F[B]
  def ap2[A, B, Z](fa: F[A], fb: F[B])(f: F[(A, B) => Z]): F[Z]
  def ap3[A, B, C, Z](fa : F[A], fb : F[B], fc : F[C])(f : F[(A, B, C) => Z]) : F[Z]

  ...
}
```

The `ap` function is just a de-generate case operation (it's the functor map operation in disguise)

What we are interested in is the `ap2` function

In cats

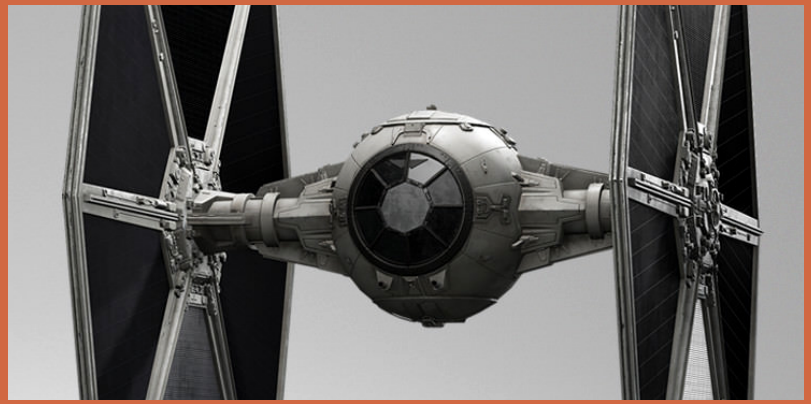
```
(mean(1 :: 7 :: Nil) |@| mean(3 :: 6 :: Nil)) map { _ + _ }

(mean(2 :: 4 :: Nil) |@|
    mean(124 :: 7 :: 68 :: Nil) |@|
    mean(39 :: 88 :: 5 :: Nil)) map { _ + _ + _ }
```

In scalaz

```
(mean(1 :: 7 :: Nil) |@| mean(3 :: 6 :: Nil)) { _ + _ }
```

|@| - What do we call it?



No flatMaps?

No short-circuiting!

Introducing the "killer app"* for

Applicative Functors ...

*Not sure if I intend the pun or not, but whatever

Validations (scalaz)
Validated (cats)

In addition to our `mean` function, let's add another:

```
def sqrt(num: Int) : String \/ Int = num match {
  case n if n < 0 => -\/("imaginary numbers not supported")
  case n => \/-(n)
}
```

If we continue to treat `sqrt` and `mean` as applicative functors:

```
(sqrt(-1) |@| mean(Nil)) {_ + _} // Result is -\/("imaginary numbers not supported.")
```

The default impl. for Applicative doesn't define how to deal with multiple errors. For that: we need validations.

```scala
sealed abstract class Validation[+E, +A] extends Product with Serializable {
  def ap[EE >: E, B](x: => Validation[EE, A => B])
  (implicit E: Semigroup[EE]): Validation[EE, B] = ...
}
```

- **Similar to a Disjunction (we have a "left" and "right") but now we have a type-class on the left, expecting that `E` must be a Semigroup**
- **Any Semigroup type (supporting `append` of course) will do**

We could use `List` but what does it mean if our result is an empty list of errors? Does that mean that there are no errors? If so, why don't we have a usable result?

This sort of ambiguity is unwelcomed to those whom are accustomed to a rich type system

That is why we can use a `NonEmptyList[String]` or `NonEmptyList[Throwable]`

cats

```
def mean(l: List[Int]) : ValidatedNel[String, Double] =
  l match {
    case Nil => "no values provided".invalidNel
    case _ :: _ => (l.sum.toDouble / l.size.toDouble).validNel
}

def sqrt(num: Int) : ValidatedNel[String, Double] = num match {
  case n if n < 0 => "imaginary numbers not supported".invalidNel
  case n => Math.sqrt(n).validNel
}
```

scalaz

```
def mean(l: List[Int]) : ValidationNel[String, Double] =
  l match {
    case Nil => "no values provided".failureNel
    case _ :: _ => (l.sum.toDouble / l.size.toDouble).successNel
}

def sqrt(num: Int) : ValidationNel[String, Double] = num match {
  case n if n < 0 => "imaginary numbers not supported".failureNel
  case n => Math.sqrt(n).successNel
}
```

Aha! Now we have all errors appended together!

```
scala> (mean(Nil) |@| sqrt(-1)).map(_ + _)
res3: cats.data.Validated[cats.data.OneAnd[[+A]List[A],String],Double] =
Invalid(OneAnd(no values provided,List(imaginary numbers not supported)))
```

A Quick Peek at shapeless

In Miles Sabin's own words:

"shapeless is a type class and dependent type based generic programming library for Scala."

In my own words:

"shapeless is a library that pushes the boundaries of Scala's rich type-system, allowing one to define strongly typed recursive structures."

This is by no means a comprehensive introduction to the library: but instead

a sneak preview on a core piece of shapeless ...

HLISTs

(Heterogeneous Lists)

An HList is a `product` type that can be thought of as an anonymous tuple:

```
val data : String :: Int :: Boolean :: HNil =
  "Solzhenitsyn" :: 1978 :: true :: HNil

data: shapeless.::[String,shapeless.::[Int,
shapeless.::[Boolean,shapeless.HNil]]] =
  Solzhenitsyn :: 1978 :: true :: HNil
```

As HLists are recursively defined, their structure lends themselves towards pattern-matching as with a standard Scala list:

```
data match {
  case head :: tail => (head, tail)
}

res5: (String, shapeless.::[Int,shapeless
  .::[Boolean,shapeless.HNil]]) = (Solzhenitsyn, 1978 :: true :: HNil)
```

You can convert between case-classes and HLists

```
import shapeless._

case class GroceryItem(name: String, cost: Double)

scala> Generic[GroceryItem].to(GroceryItem("apple", 0.10))
res9: shapeless.::[String,shapeless.::[Double,shapeless.HNil]] =
apple :: 0.1 :: HNil

scala> Generic[GroceryItem].from(res9)
res10: GroceryItem = GroceryItem(apple,0.1)
```

Shapeless Offers so much more

- Flattening Tuples
- Converting between Tuples/Lists/HLists
- Transforming between Generic types
- Generic processing of HLists of arbitrary size through recursive implicits

The Journey Continues