MONAD FACT #3

how placing **kleisli composition** logic in **flatMap** permits composition of **kleisli arrows** using **for comprehensions** and what that logic looks like in six different **monads**

$$\begin{tikzcd}
T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] \arrow[dr, phantom, "\text{FACT \#3}"] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}$$

slides by @philip_schwarz

slideshare https://www.slideshare.net/pjschwarz

@philip_schwarz

Consider three functions **f, g** and **h** of the following types:

```
f: A => B
g: B => C
h: C => D
```

e.g.

```
val f:         Int => String      = _.toString
val g:      String => Array[Char] = _.toArray
val h: Array[Char] => String      = _.mkString(",")
```

We can compose these functions ourselves:

```
assert( h(g(f(12345))) == "1,2,3,4,5" )
```

Or we can compose them into a single function using **compose**, the **higher-order** function for composing ordinary functions :

```
val hgf = h compose g compose f

assert( hgf(12345) == "1,2,3,4,5" )
```

Alternatively, we can compose them using **andThen**:

```
val hgf = f andThen g andThen h

assert( hgf(12345) == "1,2,3,4,5" )
```

We have just seen how to compose ordinary functions. What about **Kleisli arrows**: how can they be composed?

As we saw in **MONAD FACT #2**, **Kleisli arrows** are functions of types like `A => F[B]`, where **F** is a **monadic type constructor**.

Consider three **Kleisli arrows** f, g and h:

```
f: A => F[B]
g: B => F[C]
h: C => F[D]
```

How can we compose f, g and h?

We can do so using **Kleisli composition**. Here is how we compose the three functions using the **fish operator**, which is the infix operator for **Kleisli Composition**:

```
f >=> g >=> h
```

And here is the signature of the **fish operator**:

```
((A => F[B]) >=> (B => F[C])) => (A => F[C])
```

So the **fish operator** is where the logic for composing **Kleisli arrows** lives.

@philip_schwarz

e.g. here is the **fish operator** for the **Option** monad:

```
// Kleisli composition for Option
implicit class OptionFunctionOps[A, B](f: A ⇒ Option[B] ) {
  def ⟿ [C](g: B ⇒ Option[C]): A ⇒ Option[C] =
    a ⇒ f(a) match {
      case Some(b) ⇒ g(b)
      case None    ⇒ None
    }
}
```

and here is the **fish operator** for the **List** monad:

```
// Kleisli composition for List
implicit class ListFunctionOps[A, B](f: A => List[B] ) {
  def ⟿ [C](g: B ⇒ List[C]): A ⇒ List[C] =
    a ⇒ f(a).foldRight(List[C]())((b, cs) ⇒ g(b) ++ cs)
}
```

But **Kleisli Composition** can also be defined in terms of **flatMap**

```
f >=> g ≡ λa.f(a) >>= g
```

where **flatMap** is the green infix operator whose signature is shown below, together with the signatures of other operators that we'll be using shortly.

```
>=> Kleisli composition – ((A => F[B]) >=> (B => F[C])) => (A => F[C]) - aka the fish operator
>>= flatMap             - (F[A] >>= (A => F[B])) => F[B]               - aka the bind operator
==> map                 - (F[A] ==> (A => B)) => F[B]                  - lifts a function into a monadic context
unit                    - A => F[A]                                    - lifts a pure value into a mondic context
```

Let's establish some equivalences

```
((f >=> g) >=> h)(a)                 ≡ (f >=> (g >=> h)(a)                     // by associativity of >=>
((λa.f(a) >>= g) >=> h)(a)           ≡ “     “      “      “                  // LHS: rewrite 1st >=> using >>=
(λá.((λa.f(a) >>= g)(á)) >>= h)(a)   ≡ “     “      “      “                  // LHS: rewrite 2nd >=> using >>=
(λá.(f(á) >>= g) >>= h)(a)           ≡ “     “      “      “                  // LHS: simplify
“     “      “      “                ≡ (λa.f(a) >>= (g >=> h))(a)              // RHS: rewrite 1st >=> using >>=
“     “      “      “                ≡ (λa.f(a) >>= (λb.g(b) >>= h))(a)        // RHS: rewrite 2nd >=> using >>=
(f(a) >>= g) >>= h                   ≡ f(a) >>= (λb.g(b) >>= h)                // LHS and RHS: apply function to a
“     “      “      “                ≡ f(a) >>= (λb.g(b) >>= (λc.h(c) >>= λd.unit(d)))  // RHS: rewrite h using >>= & unit
“     “      “      “                ≡ f(a) >>= (λb.g(b) >>= (λc.h(c) ==> λd.d))        // RHS: rewrite >>= using ==>
```

If we take the left hand side of the first equivalence and the right hand side of the last equivalence, then we have the following

$$((f >=> g) >=> h)(a) \equiv f(a) >>= (\lambda b.g(b) >>= (\lambda c.h(c) >=> \lambda d.d))$$

In **Scala**, the right hand side of the above equivalence is written as follows

```
f(a) flatMap { b ⇒
  g(b) flatMap { c ⇒
    h(c) map { d ⇒
      d
    }
  }
}
```

which can be **sweetened** as follows using the **syntactic sugar** of **for comprehensions** (see **MONAD FACT #1**)

```
for {
  b ← f(a)
  c ← g(b)
  d ← h(c)
} yield d
```

@philip_schwarz

So in Scala, if instead of putting the logic for composing Kleisli arrows in the fish operator, we put it in a flatMap function, and also provide a map function, then we can express the composition of Kleisli arrows using a for comprehension.

```
                              for {
                                b ← f(a)
((f >=> g) >=> h)(a)    ≡       c ← g(b)
                                d ← h(c)
                              } yield d
```

Next, we'll be looking at the following:

- how six different monads are implemented in Scala by putting the logic for composing Kleisli arrows in flatMap.
- examples of using for comprehensions to compose Kleisli arrows yielding instances of those monads.

In the **Scala** code that follows, when we define a **monad**, we'll be defining **map** in terms of **flatMap** in order to stress the fact that it is the **flatMap** function that implements the logic for composing **Kleisli arrows**.

While the examples that we'll be looking at are quite contrived, I believe they do a reasonable enough job of illustrating the notion of composing **Kleisli arrows** using **for comprehensions**.

Let’s start with the simplest **monad**, i.e. the **Identity monad**, which does nothing!

```scala
// the Identity Monad – does absolutely nothing
case class Id[A](a: A) {
  def map[B](f: A => B): Id[B] =
    this flatMap { a => Id(f(a)) }
  def flatMap[B](f: A => Id[B]): Id[B] =
    f(a)
}
```

```scala
// composing the Kleisli arrows using a for comprehension
val result: Id[Int] =
  for {
    four      <- increment(3)
    eight     <- double(four)
    sixtyFour <- square(eight)
  } yield sixtyFour

assert( result == Id(64) )
```

```scala
// sample Kleisli arrows
val increment: Int => Id[Int] =
  n => Id(n + 1)

val double: Int => Id[Int] =
  n => Id(n * 2)

val square: Int => Id[Int] =
  n => Id(n * n)

assert( increment(3) == Id(4)  )
assert( double(4)    == Id(8)  )
assert( square(8)    == Id(64) )
```

```scala
// the Option Monad
sealed trait Option[+A] {

  def map[B](f: A => B): Option[B] =
    this flatMap { a => Some(f(a)) }

  def flatMap[B](f: A => Option[B]): Option[B] =
    this match {
      case None => None
      case Some(a) => f(a)
    }

}
case object None extends Option[Nothing]
case class Some[+A](get: A) extends Option[A]
```

On the left, we see the **Option monad**, and in the next slide, we look at the **List monad**

@philip_schwarz

```scala
// sample Kleisli arrows
val maybeFirstChar: String => Option[Char] =
  s => if (s.length > 0) Some(s(0))
       else None

val maybeCapitalLetter: Char => Option[Char] =
  c => if (c >= 'a' && c <= 'z') Some(c.toUpper)
       else None

val maybeOddNumber: Char => Option[Int] =
  n => if (n % 2 == 1) Some(n)
       else None

assert( maybeFirstChar("abc") == Some('a') )
assert( maybeFirstChar("") == None )
assert( maybeCapitalLetter('a') == Some('A') )
assert( maybeCapitalLetter('A') == None )
assert( maybeOddNumber('A') == Some(65) )
assert( maybeOddNumber('B') == None )
```

```scala
// composing the Kleisli arrows using a for comprehension
val result: Option[String] =
  for {
    char   <- maybeFirstChar("abc")
    letter <- maybeCapitalLetter(char)
    number <- maybeOddNumber(letter)
  } yield s"$char-$letter-$number"

assert( result == Some("a-A-65") )
```

```scala
// The List Monad
sealed trait List[+A] {

  def map[B](f: A => B): List[B] =
    this flatMap { a => Cons(f(a), Nil) }

  def flatMap[B](f: A => List[B]): List[B] =
    this match {
      case List =>
        Nil
      case Cons(a, tail) =>
        concatenate(f(a), (tail flatMap f))
   }

}

case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]

object List {
  def concatenate[A](left:List[A], right:List[A]):List[A] =
    left match {
      case Nil =>
        right
      case Cons(head, tail) =>
        Cons(head, concatenate(tail, right))
  }
}
```

```scala
// sample Kleisli arrows
val twoCharsFrom: Char => List[Char] =
  c => Cons(c, Cons((c+1).toChar, Nil))

val twoIntsFrom: Char => List[Int] =
  c => Cons(c, Cons(c+1, Nil))

val twoBoolsFrom: Int => List[Boolean] =
  n => Cons(n % 2 == 0, Cons(n % 2 == 1, Nil))

assert(twoCharsFrom(‘A’)==Cons('A',Cons('B',Nil)))
assert(twoIntsFrom(‘A’)== Cons(65,Cons(66,Nil)))
assert(twoBoolsFrom(66)==Cons(true,Cons(false,Nil)))
```

```scala
// composing the arrows using a for comprehension
val result: List[String] =
  for {
    char <- twoCharsFrom('A')
    int  <- twoIntsFrom(char)
    bool <- twoBoolsFrom(int)
  } yield s"$char-$int-$bool”

assert( result ==
        Cons("A-65-false",Cons("A-65-true",
         Cons("A-66-true",Cons("A-66-false",
          Cons("B-66-true",Cons("B-66-false",
           Cons("B-67-false",Cons("B-67-true",Nil
        )))))))) )
```

```scala
// The Reader Monad
case class Reader[E,A](run: E => A) {

  def map[B](f: A => B): Reader[E,B] =
    this flatMap { a => Reader( e => f(a) )  }

  def flatMap[B](f: A => Reader[E,B]): Reader[E,B] =
    Reader { e =>
      val a = run(e)
      f(a).run(e)
    }
}
```

On the left is the Reader monad, and in the next slide, we look at the Writer monad

```scala
type Config = Map[String, String]
val config = Map( "searchPath" -> "videoplay",
                  "hostName"   -> "video.google.co.uk",
                  "port"       -> "80",
                  "protocol"   -> "http” )

// sample Kleisli arrows
val addPath: String => Reader[Config, String] =
  (parameters: String) =>
    Reader(config =>
      s"${config("searchPath")}?$parameters")

val addHost: String => Reader[Config, String] =
  (pathAndParams: String) =>
    Reader(cfg =>
      s"${cfg("hostName")}:${cfg("port")}/$pathAndParams")

val addProtocol: String => Reader[Config, String] =
  (hostWithPathAndParams: String) =>
    Reader(config =>
      s"${config("protocol")}://$hostWithPathAndParams")
```

```scala
// composing the arrows using a for comprehension
 val result: Reader[Config, String] =
   for {
     pathAndParams <- addPath("docid?123")
     urlWithoutProtocol <- addHost(pathAndParams)
     url <- addProtocol(searchUrlWithoutProtocol)
   } yield url

 assert(
   result.run(config)
   ==
   "http://video.google.co.uk:80/videoplay?docid?123”
 )
```

```scala
// The Writer Monad
case class Writer[A](value: A, log: List[String]) {

  def map[B](f: A => B): Writer[B] = {
    this flatMap { a => Writer(f(a), List()) }
  }

  def flatMap[B](f: A => Writer[B]): Writer[B] = {
    val nextValue: Writer[B] = f(value)
    Writer(nextValue.value, this.log ::: nextValue.log)
  }

}
```

```scala
// sample Kleisli arrows
def increment(n: Int): Writer[Int] =
  Writer(n + 1, List(s"increment $n"))

def isEven(n: Int): Writer[Boolean] =
  Writer(n % 2 == 0, List(s"isEven $n"))

def negate(b: Boolean): Writer[Boolean] =
  Writer(!b, List(s"negate $b"))

assert(increment(3)==Writer(4,List("increment 3")))
assert(isEven(4)==Writer(true,List("isEven 4")))
assert(negate(true)==Writer(false,List("negate true")))
```

```scala
// composing the arrows using a for comprehension
val result: Writer[Boolean] =
  for {
    four       <- increment(3)
    isEven     <- isEven(four)
    negation   <- negate(isEven)
  } yield negation

val Writer(flag, log) = result
assert( ! flag )
assert( log == List("increment 3",
                    "isEven 4",
                    "negate true") )
```

On the next slide, our final example: the State monad

@philip_schwarz

```scala
// The State Monad
case class State[S,A](run: S => (S,A)) {

  def map[B](f: A => B): State[S,B] =
    this flatMap { a => State { s => (s, f(a)) } }

  def flatMap[B](f: A => State[S,B]): State[S,B] =
    State { s =>
      val (s1, a) = run(s)
      f(a).run(s1)
    }
}
```

```scala
// composing the arrows using a for comprehension
val result: State[Stack, Int] =
  for {
    topItem       <- pop()
    itemCount     <- pushAndCount(topItem)
    bottomElement <- peekNth(itemCount)
  } yield bottomElement

val stack = List(10,20,30)
val (_, bottomElement) = result.run(stack)
assert( bottomElement == 10 )
```

The **Kleisli arrows** on this slide are very contrived because I am doggedly sticking to the self-imposed constraint that each arrow should take as input the output of the previous arrow.

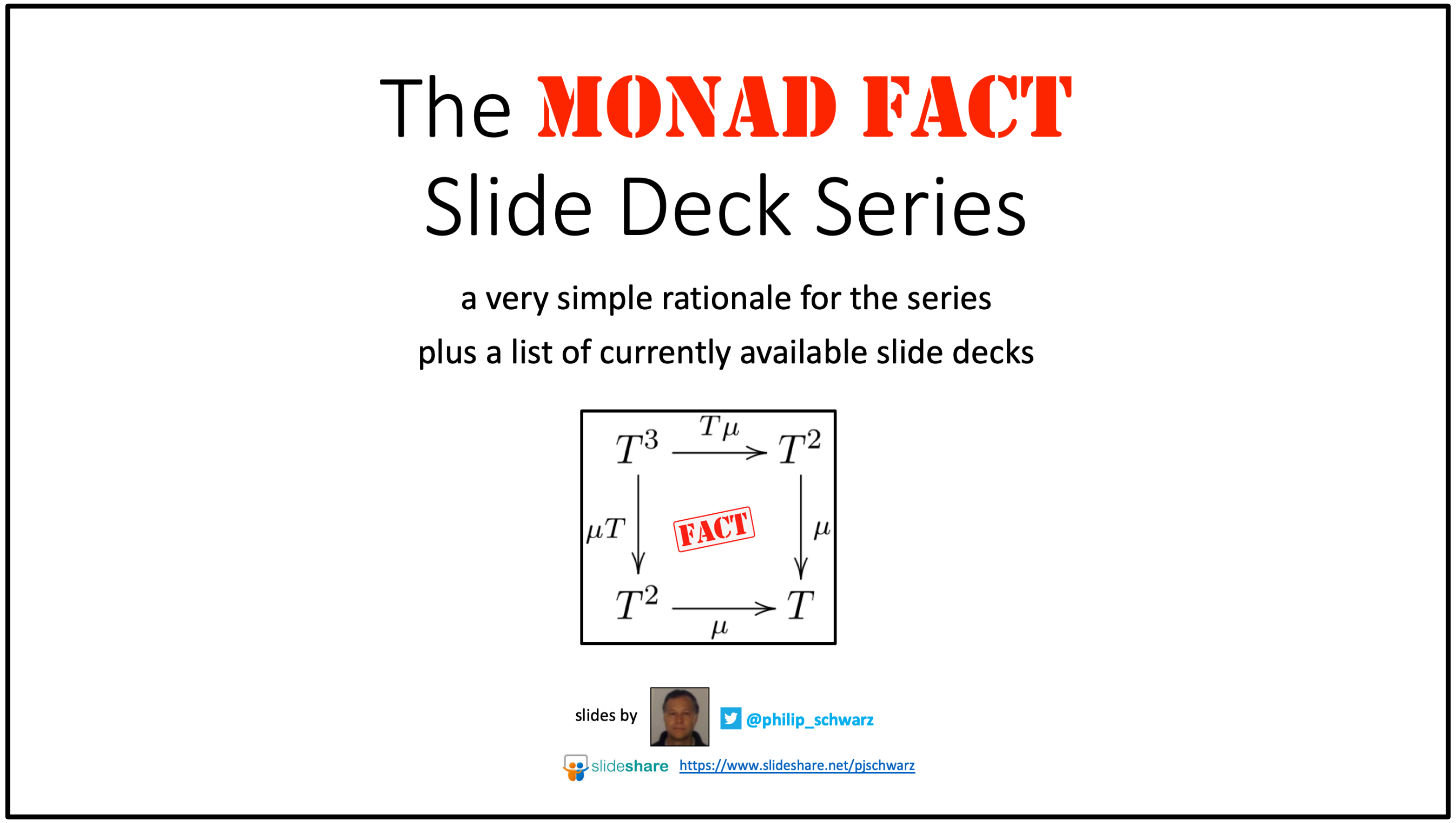
See the next slide for a more sensible Stack API and an example of its usage.

```scala
type Stack = List[Int]
val empty: Stack = Nil

// sample Kleisli arrows
val pop: () => State[Stack, Int] = () =>
  State { stack =>
    (stack.tail, stack.head)
  }

val pushAndCount: Int => State[Stack, Int] = n =>
  State { stack =>
    (n :: stack, stack.length + 1)
  }

val peekNth: Int => State[Stack, Int] = n =>
  State { stack =>
    (stack, stack(stack.length - n))
  }
```

```scala
type Stack = List[Int]
val empty: Stack = Nil

// a saner, less contrived Stack API
val pop: State[Stack, Int] =
  State { stack =>
    (stack.tail, stack.head)
  }

val push: Int => State[Stack, Unit] = n =>
  State { stack =>
    (n :: stack, ())
  }

val peek: State[Stack, Int] =
  State { stack =>
    (stack, stack.last)
  }
```

```scala
val result: State[Stack, Int] =
  for {
    _ <- push(10)
    _ <- push(20)
    a <- pop
    b <- pop
    _ <- push(a + b)
    c <- peek
  } yield c

val (_, topElement) = result.run(empty)

assert( topElement == 30 )
```

See the following slide deck for the list of all available decks in the **MONAD FACT** series



The MONAD FACT Slide Deck Series

a very simple rationale for the series

plus a list of currently available slide decks

$$\begin{tikzcd}
T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] \arrow[dr, phantom, "\text{FACT}"] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}$$

slides by @philip_schwarz

slideshare https://www.slideshare.net/pjschwarz

slideshare https://www.slideshare.net/pjschwarz