FUNCTIONAL PROGRAMMING IN

SCALA

INTRODUCTION

- The Boost libraries helped develop C++ as a language and helped drive language design.
- The Typelevel project (typelevel.org) has the same effect for functional programming within Scala. Core libraries:
  - ***Cats***, a library for common functional programming abstractions.
  - ***Shapeless***, a high level (functional) generic programming library.
- Functional programming (FP) has design patterns like 'GoF', in FP these are rooted in category theory & abstract algebra (The theory isn't required!).
- Little 'OO' design in modern Scala libraries (same could be said for many of the modern 'generic' C++ libraries?)

LEARNING MATERIALS

- "Functional Programming in Scala", Paul Chiusano & Rúnar Bjarnason, Manning.
- Scala exercises website: *"The path to enlightenment"* (https://www.scala-exercises.org)
- A modern framework for FP applications (http://frees.io)
- Introduction to Scala (allaboutscala.com)

INTRODUCTION

- Cats, Shapeless and other libraries provide the functional programming building blocks. These include:
  - Type Class
  - Functor
  - Applicative
  - Monoid & Semigroup
  - Monad & Transformer
  - Free Monad (& Algebra)

INTRODUCTION

- These slides will define the most common terms and highlight their uses with some code examples.
- Not intended to be exhaustive but a collection of collated examples.
- Chiusano and Bjarnason book is the ideal book for functional programming in Scala.
- "Learn you a Haskell for great good", Miran Lipovaca. Well worth reading as it covers the core FP concepts very succinctly and with excellent examples.

FUNCTIONS

- A ***function***, is a mapping between one set (domain) to another (codomain). In Scala, domain and codomain are the Scala *types*.

```
def square(x: Double): Double = x * 2.0
val square : Double => Double = x => x * x
```

- function that can take a function as an argument or return a function is a ***higher order function***.

```
def cube(x: Int): Int = x * x * x
def id(x: Int): Int = x
def sum(f: Int => Int, a: Int, b: Int): Int = {
  if ( a < b) 0
  else f(a) + sum (f, a + 1, b)
}
def sumInts(a: Int, b: Int) = sum(id, a, b)
def sumCubes(a: Int, b: Int) = sum(cube, a, b)
```

FUNCTIONS

- Functions can define multiple argument lists. If a function is called with a fewer number of parameter lists, the result is a function taking the missing parameter lists as its arguments.
  - Example, ***partially apply*** the parameters of f to give a new function g:

```
def filter(xs: List[Int], f: Int => Boolean): List[Int] => ???
def f(n: Int)(x: Int) = ((x%n)==0)
def g = f(2) _
// g: Int => Boolean
>filter(List(1,2,3,4,5,6,7,8),g)
```

- In a related fashion, we can ***curry*** a function:

```
def sum(f: Int => Int): (Int, Int) => Int = {
    def sumF(a: Int, b: Int): Int =
        if (a > b) 0 else f(a) + sumF(a + 1, b) sumF
}
```

- We can now write:

```
def sumInts = sum(x => x)
def sumSquares = sum(x => x * x)

sumSquares(1, 10) + sumPowersOfTwo(10, 20)
```

ALGEBRAIC DATA TYPES

- An algebraic data type is the composition of *product* or *sum* types.
- A ***product type*** is the cartesian cross product on 2 or more types. Usually represented by a case class

```
case AppConfig(k: KafkaConfig, c: CassandraConfig)
```

- A ***sum type*** is the **disjoint union** of two or more types (Sometimes called co-product as it is the dual of a product type).
  - In the simple case of two types, we can use `Either`, which is right biased (for comprehensions) as the left is usually for holding an error value. i.e.

```
Either[Throwable, Result]
```

  - To represent more than two types in **Scala 2.X** we can use a sealed trait and subtypes or use the union type of Shapeless. In Scala 2.X:

```
trait FooOrBar
case object Foo extends FooOrBar
case object Bar extends FooOrBar
```

  - More succinctly, with **Shapeless** we can write:

```
type ISB = Int :+: String :+: Char :+: Boolean :+: CNil
val isb = Coproduct[ISB]("foo")
```

  - Scala 3.X will directly support union and intersection types (removing the need for the traits).
    - Intersection type is typically something like: '`A with B`' this will become '`A & B`'. The difference being '&' will be commutative. i.e. `A & B` is the same type as `B & A`.

TYPE CLASSES

- Suppose we have a list of T, and our list class has a method 'head' that returns the first element. The function will do the same thing for whatever T the list was parameterised with; this is ***parametric polymorphism***.
- ***Ad-hoc polymorphism*** is bound to the type. Depending on the type, different implementations are invoked.
- Type class allow for ad-hoc polymorphism.
- A type class can be thought of a set of types with operations defined on them. Somewhat similar to Java interfaces or Adapter pattern (but cleaner).
- Example, we can define a 'Showable' interface, together with an apply method that allows the splitting of the definition and different implementations.
- We have a triple of **trait**, **objec**t and **instances**.

```
// Trait
trait Show[A] {
  def show(a: A): String
}
// Object
object Show {
  def apply[A: Show]: Show[A]  = implicitly // syntactic sugar for creating objects.
  def show[A: Show](a: A) = Show[A].show(a)  // interface method.
}
```

TYPE CLASS INSTANCES

- With the trait on object, we can 'independently' define implementations for 'Show'

```
object ShowInstances {
   implicit def showForInt: Show[Int] = (i: Int) ⇒ s"My int is: $i"
}
```

- To use the interface we bring the instances into scope and invoke the method. As follows:

```
import ShowInstances._
Show.show(1)
```

- An alternative approach (to adding interface methods into the object) is to use type enrichment to extend existing types with interface methods.
- Popular in some Typelevel libraries, by convention this is referred to as the "syntax" as defined similar to as shown below:

```
object ShowSyntax {
 implicit class ShowOps[A](value: A) {
    def show(implicit ev: Show[A]): String = ev.show(value)
} …
```

- This allows us to write the following:

```
 val x = 1234.show // "My int is: 1234"
```

HIGHER KINDED TYPES

- A type constructor can be thought of a function that accepts some type and returns a new type.
- Languages such as Java have basic type constructors, e.g. `ArrayList` can be thought of as type level function that takes one parameter `<T>` and returns new types `ArrayList<Boolean>`, `ArraryList<Double>`, etc…
- Functional programming languages like Scala have 'higher order type constructors'. That is, a type constructor that takes a type constructor as a parameter.
- This is a fundamental requirement for functional programming abstractions.

HIGHER KINDED TYPES

- We previously defined a ***type constructor*** as '*a function that accepts some type and returns a new type*'.
    - New types can be defined by recursively composing type constructors.
    - Concretely, a type constructor is a *n*-ary type operator, taking as arguments zero or more types and returning a new type.
    - Currying, we can re-write an *n*-ary type operator as a sequence of unary type operators.
- A ***kind*** is the type of a type constructor.
- Examples,
    - `*` is the kind of all data types (the set of all types).
    - `* → *` is the kind of all unary type constructors (e.g. `List`).
    - `* → * → *` is a binary type constructor (via currying) (e.g. `Either`).
    - `(* → *) → *` is a higher order type constructor from unary type constructor to proper types.
- Simply, similar to higher order functions, we have higher order type constructors that take type constructors as arguments.

TYPE LAMBDAS

- In the same way that functions may be partially applied, we can also partially apply types.
- Example, in the second case below, we define `Result` as partial application of the `Either` type.

```
type Result = Either[Throwable,Double]

type Result[A] = Either[Throwable,A]
```

- It is not always possible to use this convenient syntax, we sometimes have to use a full type lambda, e.g.

```
({type λ[α] = Either[String, α]})#λ
```

- Thankfully, there is a compile plugin (Typelevel kind projector) that simplifies the syntax and allows us to write examples such as:

```
Tuple2[?, Double]        // equivalent to: type R[A] = Tuple2[A, Double]

Either[Int, +?]          // equivalent to: type R[+A] = Either[Int, A]

Function2[-?, Long, +?]  // equivalent to: type R[-A, +B] = Function2[A, Long, B]

EitherT[?[_], Int, ?]    // equivalent to: type R[F[_], B] = EitherT[F, Int, B]
```

FUNCTOR

- When we apply a function to (+2) to 3 we get 5. Suppose we have an instance of the `Option` type: `Some(3)`. How do we apply `(+2)` ?
  - Intuitively, we need to take (+2) “apply it” to `Some(3)` and return `Some(5)`. If the value had be `None`, we would expect the result to be `None` too.
  - In this case the ‘functor’ is `Option`.
  - Informally, you apply a function to some ‘wrapped’ value and get back a ‘wrapped’ value.
  - This can be implemented with a `map` function, in the case of `Option`:

```
def map[A,B](oa: Option[A])(f: A⇒B): Option[B]  = oa match {
  case Some(a) ⇒ Some(f(a))
  case None ⇒ None
}
```

- We can generalise the functor ‘map’ as a trait:

```
trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A⇒B): F[B]
}
```

- We have parameterised map on the type constructor `F[_]`, which will take type constructors such as `List` or `Option`.
- That is, a F is a type constructor of kind `* → *`

FUNCTOR

- The Functor trait defined is parametric in the choice of F.
- Example for `List`, we could define the Functor instance as:

```
val listFunctor = new Functor[List] {
  def map[A,B](a: List[A])(f: A⇒B): List[B] = a map f
}
```

- Here we are saying that a type constructor `List` is a functor - and the `Functor[List]` instance is proof that `List` is actually a functor.
- Functor Laws
  - Identity - `map(f)(x⇒x) == f`

NATURAL TRANSFORMATIONS

- A polymorphic function that maps from one functor F[_] to another G[_] is called a natural transformation. Usually represented by the type ⇝

```
trait FunctionK[-F[_], +G[_]] {
   def apply[A](fa: F[A]): G[A]
}
type ⇝ [-F[_],+G[_]] = FunctionK[F,G]
```

- Example,

```
val toList = new (Option ⇝ List) {
    def apply[A](fa: Option[A]): List[A] = fa match {
      case Some(a) ⟹ List(a)
      case None ⟹ List.empty[A]
}
```

- Example, return the first element of a list or error:

```
type ErrorOr[A] = Either[String, A]

val errorOrFirst: FunctionK[List, ErrorOr]
    = λ[FunctionK[List, ErrorOr]](_.headOption.toRight("the list is empty.”))
```

APPLICATIVES

- The informal definition of a functor was:
    - *"you apply a function to some 'wrapped' value and get back a 'wrapped' value".*
    - Example, `(+2)` applied to `Some(3)` gives a result `Some(5)`.
- With applicatives we *apply a wrapped function to a wrapped value and get back a wrapped value.*
- Intuitively, `Some(+2)` "applied to" `Some(3)` returns `Some(5)`.
- ***Lifting*** - For some `X`, `F[X]` is referred to as the lifted version of `X`.
    - Example, `x ⇒ x * x` can be lifted into `List` as `List(x⇒ x*x)`.
- Concretely, an ***Applicative*** is a functor that:
    - Provides a way to apply a lifted function `F[A⇒B]` to some lifted value `F[A]` and returns a lifted value `F[B]`.
    - Provides a way to lift a value into the functor.
- This behaviour can be defined as a trait as follows:

```
trait Applicative[F[_]] extends Functor[F] {
  def pure[A](a: A): F[A]
  def ap[A,B](fa: F[A])(f: F[A⇒B]): F[B]
}
```

APPLICATIVES

- Applicative Laws
  - Identity - `ap(fa)(pure) == fa`
  - Homomorphism - `ap(pure(a))(pure(ab)) == point(ab(a))`
  - Interchange - `ap(pure(a))(point(ab)) == point(ab(a)`

APPLICATIVE EXAMPLE

- We can now write (n.b. usually we would make Applicative a type class with instances):

```
implicit val applicativeOpt = new Applicative[Option] {// map as defined on Functor slide.
  def pure[A](a: A): Option[A] = Some(a)
  def ap[A,B](fa: Option[A])(of: Option[A⇒B]): Option[B] = fa match {
    case None ⇒ None
    case Some(a) ⇒ of match {
      case Some(g) ⇒ Some(g(a))
      case None ⇒ None
    } }
}
import applicativeOpt._
val f: Int ⇒ Int =  x ⇒ x + 2
ap(Some(3))(Some(f)) // : Some(5)
ap(ap(Some(3))(Some(f)))(Some(f)) // : Some(7)
```

SEMIGROUP AND MONOID

- A ***semigroup*** can be thought of as a way to combine two values of the same type to form another value of the same type.
  - Addition forms a semigroup over the integers.
  - In Scala this *behaviour* can be defined as a trait:

```scala
trait Semigroup[T] {
  def combine(x: T, y: T): T
}
```

- A ***monoid*** is a semigroup that has a unit or 'zero' element.
  - Natural numbers, + and 0 form a monoid.
  - Can be defined simply as:

```scala
trait Monoid[T] extends Semigroup[T] {
  def unit: T
}
```

- Monoid laws
  - Associativity - `combine(x, combine(y, z)) == combine(combine(x, y), z)`
  - Identity - `combine(x, unit) == x , combine(unit, x) == x`

MONOID

- We can write a generic '`combineAll`' method as follows:

```
def combineAll[A: Monoid](xs: List[A]): A = xs.foldLeft(Monoid[A].unit)(Monoid[A].combine)
```

- The above is a generic function that will work for all types that define a monoid instance. For example, if we define for integers:

```
implicit val intMonoid = new Monoid[Int] {
  def unit: Int = 0
  def combine(x: Int, y: Int): Int = x +y
}
```

- And for strings:

```
implicit val stringMonoid = new Monoid[String] {
  def unit: String = “”
  def combine(x: String, y: String): String = x + y
}
```

- We can now write (n.b. make Monoid a type class with instances as above):

```
combineAll(List(1,2,3,4) // :Int = 6
combineAll(List(“hello”, “ “, ”world”) // :String = “hello world”
```

MONOID

- Booleans with an 'Or' operator form a Monoid:

```
 val booleanOr: Monoid[Boolean] = new Monoid[Boolean] {
    def unit: Boolean = false
    def combine(x: Boolean, y: Boolean): Boolean = x || y
}
```

- Similarly, Booleans with an 'And' operator for a Monoid:

```
val booleanAnd: Monoid[Boolean] = new Monoid[Boolean] {
    def unit: Boolean = true
    def combine(x: Boolean, y: Boolean): Boolean = x && y
}
```

- We can now write:

```
val results: List[Boolean] = List(true, true, false, true, false)
Results.reduce(booleanOr) // : true
Results.reduct(booleanAnd) // : false
```

MONOID

- A homomorphism between two monoids `(M, *)` and `(N, •)` is a function `f: M → N` such that

  $f(x * y) = f(x) \cdot f(y)$ for all x, y in M

  $f(e_M) = e_N$

  where e_M and e_N are the identities on M and N respectively.

- A bijective monoid homomorphism is called a monoid isomorphism. Two monoids are said to be isomorphic if there is a monoid isomorphism between them.

- How is this used? We can choose to implement one monoid in terms of another:

```scala
def booleanIsomorphism(mb: Monoid[Boolean]):Monoid[Boolean] = new Monoid[Boolean] {
    def combine(x: Boolean, y: Boolean) = !mb.op(!x, !y)
    def unit = !mb.zero
  }
val booleanAnd: Monoid[Boolean] = booleanIsomorphism(booleanOr)
```

FOLD MAP

- `foldMap` maps elements of type A to elements of type B.
- The B elements can be reduced by the monoid B.

```
def foldMap[A, B](xs: List[A], m: Monoid[B])(f: A ⇒ B): B
  = xs.foldLeft(m.zero)((b, a) ⇒ m.op(b, f(a)))
```

- As a simple example, count and average with one parse through a list:

```
def op(a: (Double, Int), b: (Double, Int)): (Double, Int) = {
  val cCount: Int = a._2 + b._2
  val cAverage: Double = (a._2.toDouble * a._1 + b._2.toDouble * b._1) / cCount
  (cAverage, cCount)
}
val monoidAverageAndCount: Monoid[(Double, Int)] = new Monoid[(Double, Int)] {
  def op( a: (Double, Int), b: (Double, Int) ): (Double, Int) = …
  def zero: (Double, Int) = (0.0, monoidIntAddition.zero)
}
val doubles = foldMap(doubles, monoidAverageAndCount){ (d: Double) ⇒ (d,1) }
```

MONAD

- Monad is an extension of Applicative. Informally,
  - If I have a value in a context `M[A]`
  - An a function that returns a value in a context, e.g. `A⇒M[B]`
  - How do we apply `A⇒M[B]` to `M[A]`?
  - We need a function that takes a value in a nested context and "joins" the contexts together so that we have a single context.
  - The function is called the bind or flatMap operator, that has the following signature:

```
def flatMap[A, B](fa: M[A])(f: A ⇒ M[B]): M[B]
```

- As for applicative and the others we can define this behaviour as a trait:

```
trait Monad[F[_]] extends Applicative[F] {
  def flatMap[A,B](fa: F[A])(f: A ⇒ F[B]): F[B]
}
```

- Monad laws
  - left identity - `pure(x).flatMap(f) == f(x)`
  - right identity - `x.flatMap(pure) == x`
  - associativity - `x.flatMap(f).flatMap(g) == x.flatMap(y ⇒ f(y).flatMap(g))`

MONAD

- Implementing the `flatMap` interface for `List` (illustration only, this implementation is compact but very slow) and `Option`:

```
def flatMap[A, B](xs: List[A])(f: A ⇒ List[B]): List[B]
  = xs.foldLeft(List[B].empty)((ys,y) ⇒ ys ++ f(y))
def flatMap[A,B](oa: Option[A])(f: A ⇒ Option[B]): Option[B] = if ( oa.isDefined ) f(oa) else None
```

- Allows us to write:

```
Monad[Option].pure(3) // : Option[Int] = Some(3)
Monad[Option].flatMap(res0)(a ⇒ Some(a + 2)) // : Option[Int] = Some(5)
Monad[List].pure(3) // : List[Int] = List(3)
Monad[List].flatMap(res2)(x ⇒ List(x, x*10)) // : List[Int] = List(3, 30)
```

- We can now implement **generic** functions, such as:

```
def sumSquare[A[_] : Monad](a: Int, b: Int): A[Int] = {
    val x = a.pure[A]
    val y = b.pure[A]
    x flatMap (x ⇒ y map (y ⇒ x*x + y*y))}
}
sumSquare[Option](3, 4) // : Option[Int] = Some(25)
sumSquare[List](3, 4) // : List[Int] = List(25)
```

IDENTITY MONAD

- The identity monad can be thought of as encapsulating the effect of having no effect.
- We can define a type:

```scala
type Id[+A] = A
```

- This `Id` context has no effect, the monad instance can be defined along the lines of:

```scala
implicit val id = new Monad[Id] {
  override def flatMap[A, B](fa: Id[A])(f: (A) ⇒ Id[B]): Id[B] = f(fa)
  override def pure[A](a: A): Id[A] = a
  override def ap[A, B](fa: Id[A])(f: Id[(A) ⇒ B]): Id[B] = f(fa)
  override def map[A, B](fa: Id[A])(f: (A) ⇒ B): Id[B] = f(fa)
}
```

- At first glance this monad doesn't seem useful.
- However - It is fundamental to the concept of monad transformers (any monad transformer applied to the identity monad returns a non-transformer version of that monad) - this is dealt with later.

WRITER MONAD

- A writer monad `Writer[L,V]` a monad that allows us to carry a log (L) along with a value (V).
- It has a number of uses, including:
  - Delayed logging during concurrent computations, where log messages from different contexts may otherwise be interleaved.
  - Ensuring that a function is 'pure' in the sense that it has no IO side effect.

WRITER MONAD

- Looking at simplified implementation:

```
final case class WriterT[F[_], L, V](run: F[(L, V)]) {
  def tell(l: L)(implicit functorF: Functor[F], semigroupL: Semigroup[L]): WriterT[F,L,V] = mapWritten(_ |+| l)
  def written(implicit ev: Functor[F]): F[L] = ev.map(run)(_._1)
  def value(implicit ev: Functor[F]): F[V] = ev.map(run)(_._2)
  def mapBoth[M,U](f: L, V) ⇒ (M, U))(implicit ev: Functor[F]): WriterT[F,M,U]  = WriterT { ev.map(run)(f.tupled) }
  def mapWritten[M](f: L ⇒ M)(implicit ev: Functor[F]): WriterT[F, M, V] = mapBoth((l, v) ⇒ (f(l), v))
}
type Writer[L, V] = WriterT[Id, L, V]
object Writer {
  def apply[L, V](l: L, v: V): WriterT[Id, L, V] = WriterT[Id, L, V]((l, v))
  def value[L:Monoid, V](v: V): Writer[L, V] = WriterT.value(v)
  def tell[L](l: L): Writer[L, Unit] = WriterT.tell(l)
}
```

- We can see that `Writer[L,V]` is actually `WriterT[`**`Id`**`, L, V]`, `WriterT` is an example of a monad transformer. These are discussed in a later slide.

WRITER MONAD

- Whilst we should use `Vector` instead of `List` for efficiency reasons, we can now return Log and Value from a **series** of computations:

```
def logNumber(x: Int): Writer[List[String], Int] =
  Writer(List("Got number: " + x.show), 3) // Writer[List[String],Int]

def multWithLog: Writer[List[String], Int] = for {
           a ← logNumber(3)
           b ← logNumber(5)
         } yield a * b
// Writer[List[String],Int]
multWithLog.run
// Id[(List[String], Int)] = (List(Got number: 3, Got number: 5),9)
```

- The logging information and value is returned as `Id`! From which we can transparently take the tuple of log messages and result value.
- The Writer monad is closely related to the State monad, dealt with later. It is more restrictive in that we can only 'append' (Monoid combine operator) rather than read and write state.

MONAD TRANSFORMERS

- A monad transformer is a type constructor that takes a monad as an argument and returns a monad as a result.
- Concretely, we can define a monad transformer as:
  - A type constructor T of kind `(* → *) → * → *`
  - Provides Monad operators ‘`pure`’ and ‘`flatMap`’
  - An additional ‘`lift`’ operator, that takes a function and a monadic value and maps it over the monadic value.

```
trait MonadTrans[T[_[_],_] { …
  def liftM[M[_], A](a: M[A])(implicit M: Monad[M]): T[M, A]
}
```

MONAD TRANSFORMERS

- *"Functors and Aplicatives compose Monads don't".*
- We can compose monads of the same type easily.
- Example, two functions both returning `Option`:

```
def first: Option[Int] = Some(1)
def second: Option[Int] = Some(2)

val result : Option[Int] = for{
  x ← first
  y ← second
} yield x+y

println(result) // Some(3)
```

MONAD TRANSFORMERS

- Example, two functions both returning `Future`:

```
def first: Future[Int] = Future(1)
def second(x: Int): Future[Int] = Future(x+2)

val result : Future[Int] = for{
  x ← first
  Y ← fb(a)
} yield y

Await.result(result,Duration.Inf)
println(result) // Future(Success(3))
```

MONAD TRANSFORMERS

- Problem, when we try compose `Future` and `Option`:

```
def first: Future[Option[Int]] = Future(Some(1))
def second(a: Int): Future[Option[Int]] = Future(Some(a+2))
// won't compile !!!!!
val result = for {
  optionX ← first        // ← future
  y ← optionX            // ← option
  xy ← second(y)         // ← future ... types don't line up!
} yield xy
```

- Naive way to solve this:

```
val composedAB: Future[Option[Int]] = first.flatMap {
  case Some(a) ⇒ second(a)
  case None ⇒ Future.successful(None)
}
```

- But this could become very messy, particularly if we have multiple nested calls!

MONAD TRANSFORMERS

- Use a *transformer* to compose `Future` and `Option`.
- The following shows a minimalist transformer:

```
case class FutureO[+A](future: Future[Option[A]]) extends AnyVal {
  def flatMap[B](f: A ⇒ FutureO[B]): FutureO[B] = {
    val newFuture = future.flatMap {
      case Some(a) ⇒ f(a).future
      case None ⇒ Future.successful(None)
    }
    FutureO(newFuture)
  }
  def map[B](f: A ⇒ B): FutureO[B] = {
    FutureO(future.map(option ⇒ option.map(f)))
  }
```

- As we can see in the `flatMap` function, we are simply unpacking the option from the future, if there is a value we call `f(value)`.

MONAD TRANSFORMERS

- We can now write:

```
def divideEven(n: Int): Option[Int] = if (n % 2 == 0) Some(n/2) else None

val f1 = Future(divideEven(14))
val f2 = Future(divideEven(16))

val fc = for {
  a ← FutureO(f1)
  b ← FutureO(f2)
} yield a + b
val f = fc.future

f onComplete println // Success(Some(15))
```

KLEISLI & READER MONAD

- `Kleisli[F[_],A,B]` is a simple wrapper around a function `A⇒F[B]`, there is also a special case involving the identity monad, `Kleisli[Id,A,B]`.
- Dependent on `F[_]` we can do different things.
- Example, composing functions:

```
final case class Kleisli[F[_], A, B](run: A ⇒ F[B]) {
  def compose[Z](k: Kleisli[F, Z, A])(implicit F: FlatMap[F]): Kleisli[F, Z, B] =
    Kleisli[F, Z, B](z ⇒ k.run(z).flatMap(run))
}
```

- If we want to wrap a function A⇒B, rather than A⇒`F[B]` we can partially apply the Kleisli with the identity monad - the `Reader` monad.

```
type Id[A] = A
type Reader[A, B] = Kleisli[Id, A, B]
object Reader {
  def apply[A, B](f: A ⇒ B): Reader[A, B] = Kleisli[Id, A, B](f)
}
type ReaderT[F[_], A, B] = Kleisli[F, A, B]
val ReaderT = Kleisli
```

- Informally, we can build up a computation that is a function of context (configuration, etc.) rather than passing the context as an argument of the function.

READER MONAD

- We create a `Reader[A,B]` from a function of type A⇒B and run it:

```scala
def square(a: Int): Int = a*a
val squareR: Reader[Int, Int] = Reader(square)
squareR.run(2)
// :Id[Int] = 4
```

- What does the `flatMap` implementation of a `Reader` look like?

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

- We can therefore combine readers in a for comprehension:

```scala
val composeReaders: Reader[Int, Int] =
  for {
    x ← squareR
    y ← cubeR(x)
  } yield x + y
composeReaders.run(10)
```

READER MONAD

- Analogous to **dependency injection**:

```
def areaR(r: Int): Reader[Double,Double] = Reader { pi ⇒ pi * r * r }

val areaRR: Reader[Int,Reader[Double,Double]] = Reader { r ⇒ areaR(r) }

def volumeRR(h: Int): Reader[Int,Reader[Double,Double]] =
  areaRR map { areaR ⇒areaR map { a ⇒ a * h }}

val volumeRRR = Reader { h: Int ⇒ volumeRR(h) }

val computation = volumeRR(2).run(1) // forms the computation, still need the initial value
val valueToInjectIntoCalc = 3.141    // the value we want to inject.

println(computation.run(valueToInjectIntoCalc)) // 6.282
```

STATE MONAD

- Imagine a computer game where we have a number of state updates:

```
val (s0, _) = init()
val (s1, _) = nextBlock(s0)
val (s2, moved0) = moveBlock(s1, LEFT)
val (s3, moved1) = if (moved0) moveBlock(s2, LEFT) else (s2, moved0)
```

- Passing around state objects (s0,s1,…) becomes error-prone boilerplate.
- A stateful computation is a function that takes some state and returns a value along with some new state.
- `State` is a data type that encapsulates a stateful computation `S => (S,A)` and forms a monad which passes along the states represented by the type `S`.
- When we examine the type of `State`, we see that is is the partial application of the monad transformer `StateT` with `Eval` (a monad that controls evaluation) which emulates a call stack with heap memory to prevent overflow.

```
type State[S, A] = StateT[Eval, S, A]
```

STATE MONAD

- The core parts of `StateT` are defined as:

```
final class StateT[F[_], S, A](val runF: F[S ⇒ F[(S, A)]]) {  … }
object StateT extends StateTInstances {
  def apply[F[_], S, A](f: S ⇒ F[(S, A)])(implicit F: Applicative[F]): StateT[F, S, A]
       = new StateT(F.pure(f))
  def applyF[F[_], S, A](runF: F[S ⇒ F[(S, A)]]): StateT[F, S, A]
       = new StateT(runF)

/** Run with the provided initial state value*/
  def run(initial: S)(implicit F: FlatMap[F]): F[(S, A)] = F.flatMap(runF)(f ⇒ f(initial))
}
```

- To construct a `State` value, you pass a state transition function to `State.apply`

```
def apply[S, A](f: S ⇒ (S, A)): State[S, A] = StateT.applyF(Now((s: S) ⇒ Now(f(s))))
```

- Next we can show how to use the `State` monad.

STATE MONAD

- Lets define a stack type: `type Stack = List[Int]`
- With some functions:

```
val pop: State[Stack, Int] = for {
          s ← State.get[Stack]
          (x :: xs) = s
          _ ← State.set[Stack](xs)
        } yield x

def push(x: Int): State[Stack, Unit] = for {
          xs ← State.get[Stack]
          r ← State.set(x :: xs)
        } yield r
```

STATE MONAD

- We can now compose stateful computations as follows:

```
def stackComputation: State[Stack, Int] = for {
          _ ← push(3)
          a ← pop
          b ← pop
        } yield(b)

val program = stackComputation.run(List(5, 8, 2, 1))
// forms a computation

val result = program.value // evaluates the computation.
// (Stack, Int) = (List(8, 2, 1),5)
```

- Both `push` and `pop` are purely functional and we have avoided boiler-plate passing of state between each of the calls.

FREE MONAD

- A free monad is a monad that allows you to build a monad from any functor. *Eh?*
- What does this mean? Informally,
  - `Free` stores a list of functors, wrapped around an initial value.
  - Functor and Monad instances of `Free` do nothing other than hand the function down using flatMap.

FREE MONAD

- Free monads are commonly used to:
  - Provide an abstraction for trampolining recursive functions.
  - Define algebra and co-products of algebra, to form a kind of DSL and target different interpreters using natural transformations.
- Many other use cases …

FREE MONAD

- First, look at the outline definition of `Free`:

```
sealed trait Free[F[_],A]
case class Return[F[_],A](a: A) extends Free[F,A]
case class Bind[F[_],I,A](i: F[I], k: I ⇒ Free[F,A]) extends Free[F,A]
```

- With the implementation of `Free` as:

```
sealed trait Free[F[_],A] {
    def flatMap[B](f: A ⇒ Free[F,B]): Free[F,B] = this match {
            case Return(a) ⇒ f(a)
            case Bind(i, k) ⇒ Bind(i, k andThen (_ flatMap f))
        }
    def map[B](f: A ⇒ B): Free[F,B] = flatMap(a ⇒ Bind(f(a)))
}
```

- Together with a `lift` function:

```
implicit def lift[F[_],A](fa: F[A]): Free[F,A] = Bind(fa, (a: A) => Return(a))
```

FREE MONAD

- `Pure` builds a `Free` instance from an A value.
- `Suspend` builds a new `Free` by applying F to the previous `Free`.
- We end up with a recursive data structure:

  `Suspend(F(Suspend(F(Suspend(F(….(Pure(a))))))))`
- Can be seen as a sequence of computations, where:
  - `Pure` returns an A value and ends the computation.
  - `Suspend` is a continuation; it suspends the current computation with the functor F and hands control to the caller. A represents a value bound to this computation.
- The recursion is *encoded* on the heap rather than the stack.

FREE MONAD

- Suppose we have a console application:

```
sealed trait Interact[A]
case class Ask(prompt: String) extends Interact[String]
case class Tell(msg: String) extends Interact[Unit]
```

- We can define a program as:

```
val program: Free[Interact, Unit] = for {
  first ← Ask("What's your first name?")
  last  ← Ask("What's your last name?")
  _     ← Tell(s"Hello, $first $last!")
} yield ()
```

- This just encodes a representation of the program …

FREE MONAD

- We have an encoding of the ‘program’:

```
val prg: Free[Interact, Unit] =
  Bind(
    Ask("What's your first name?"),
      first ⇒ Bind(
        Ask("What's your last name?"),
          last ⇒ Bind(
            Tell(s"Hello, $first $last!"),
            _ ⇒Return(())))))
```

FREE MONAD

- Next, we can implement an interpreter:

```
type Id[A] = A
object Console extends (Interact ~> Id) { // Natural transformation
    def apply[A](i: Interact[A]) = i match {
        case Ask(prompt) =>
            println(prompt)
            readLine
        case Tell(msg) =>
            println(msg)
    }
}
```

- Possible to define alternative interpreters for the same program (e.g. test).
- Run, via a `foldMap` on `Free`:

```
val result = program.foldMap(Console)
```

- Can combine with `State` and `Coproduct` to handle stateful interactions and multiple algebra.

TRAMPOLINE

- We may have a recursive function that causes a stack overflow.
- Can define a Trampoline in terms of Free as:

```scala
type Trampoline[A] = Free[Function0, A]
// Function0 is a function that takes zero arguments

object Trampoline {
  def done[A](a: A): Trampoline[A] = Free.Pure[Function0,A](a)

  def suspend[A](a: ⇒ Trampoline[A]): Trampoline[A] = Free.Suspend[Function0, A](() ⇒ a)

  def delay[A](a: ⇒ A): Trampoline[A] = suspend(done(a))
}
```

TRAMPOLINE

- Can now express recursive functions (that we can't easily make tail recursive) in terms of suspend and done:
- Example, I trampolined an 'unfold' method:

```
def unfold[T,R](z: T)(f: T ⇒ Option[(R,T)]): Trampoline[Stream[R]] = f(z) match {
   case None ⇒
     Trampoline.done(Stream.empty[R])
   case Some((r,v)) ⇒
        Trampoline.suspend(unfold(v)(f)).flatMap(s ⇒ Trampoline.done(r #:: s))
 }
```

- *"The under appreciated unfold"*, Jeremy Gibbons & Geraint Jones (available on-line).

TRAMPOLINE

- Function was used by a graph traversal function, that performs depth or breadth first search by 'unfolding' the graph into a stream:
- Example on how to use the unfold rather than strictly most efficient computation.

```
def traverse[A](g: Directed[A], l: List[A], dfs: Boolean): Stream[A]
  = unfold( (l,Set.empty[A]) ) {
        case (current, visited) ⇒ current match {
          case w :: Nil ⇒
             Some((w, (g.successors(w).toList.filterNot(visited.contains), visited + w)))
          case w :: vs ⇒
            val next = if (dfs) g.successors(w).toList ::: vs
            else vs ::: g.successors(w).toList
            Some((w, (next.filterNot(visited.contains), visited + w)))
          case _ ⇒ None
        }
      }.run
```