EFFECTS IN FUNCTIONAL PROGRAMMING

LAST LECTURE IN 2 MIN

FP is cool because:

- readability, ability to reason
- more compile-time checks
- lazy evaluation, parallelism, memoization

Where's the catch? FP requires **pure functions**.

Pure = no side effect = no interaction with data outside of scope

RETHINK THE WAY YOU CODE

- Higher-order functions instead of loops
- Recursion instead of (some) while loops
- Immutable data for safety

WHAT ABOUT THOSE?

- Access a configuration
- Maintain a state
- Query a database
- Write to a file

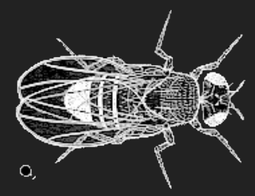
All those are impure: they contain side effects!

WHAT DO I DO?

GENERATING RANDOM NUMBERS

Suppose I want to randomly generate a family of flies. A boolean gene is passed on to progeny like this:

Mother gene	Father gene	Child gene
A	A	A
B	B	B
x	y	x with probability 60%



JAVA-STYLE CODE

```
case class Fly(gene: Boolean)

class FlyGenerator(val random: Random) {

    def spawnFly: Fly = Fly(random.nextBoolean)

    def spawnChild(mother: Fly, father: Fly): Fly = (mother, father) match {
        case (Fly(a), Fly(b)) if a == b => Fly(a)
        case (Fly(a), Fly(b)) => if(random.nextDouble <= 0.6) Fly(a) else Fly(b)
    }

    def spawnFamily: (Fly, Fly, Fly) = {
        val mother = spawnFly
        val father = spawnFly
        val child = spawnChild(mother, father)
        (mother, father, child)
    }
}
```

JAVA-STYLE CODE

Impure code: same function called with different results!

```
scala> val generator = new FlyGenerator(new Random(42))
generator: FlyGenerator = FlyGenerator@c2b1381

scala> generator.spawnFamily
res1: (Fly, Fly, Fly) = (Fly(true),Fly(false),Fly(false))

scala> generator.spawnFamily
res2: (Fly, Fly, Fly) = (Fly(true),Fly(false),Fly(true))
```

SECOND TRY

```
object FlyGenerator {

    def spawnFly(random: Random): Fly = Fly(random.nextBoolean)

    def spawnChild(random: Random, mother: Fly, father: Fly): Fly = (mother, father) matc
        case (Fly(a), Fly(b)) if a == b => Fly(a)
        case (Fly(a), Fly(b)) => if(random.nextDouble <= 0.6) Fly(a) else Fly(b)
    }

    def spawnFamily(random: Random): (Fly, Fly, Fly) = {
        val mother = spawnFly(random)
        val father = spawnFly(random)
        val child = spawnChild(random, mother, father)
        (mother, father, child)
    }
}
```

SECOND TRY

```
scala> FlyGenerator.spawnFamily(new Random(42))
res3: (Fly, Fly, Fly) = (Fly(true),Fly(false),Fly(false))

scala> FlyGenerator.spawnFamily(new Random(42))
res4: (Fly, Fly, Fly) = (Fly(true),Fly(false),Fly(false))

scala> val r = new Random(42)
r: Random = Random@1f0a09c8

scala> FlyGenerator.spawnFamily(r)
res5: (Fly, Fly, Fly) = (Fly(true),Fly(false),Fly(false))

scala> FlyGenerator.spawnFamily(r)
res6: (Fly, Fly, Fly) = (Fly(true),Fly(false),Fly(true))
```

Not much better...

`Random` is a mutable class.

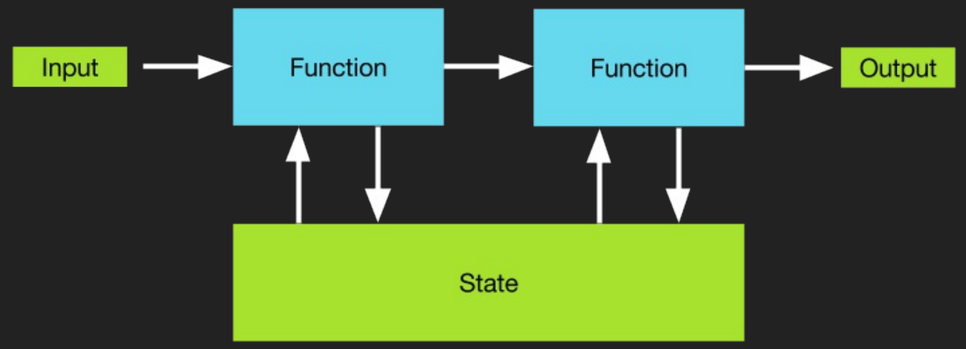
HEART OF DARKNESS

```
class Random(var seed: Long) {
    private def next(bits: Int): Int = {
        seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1)
        (seed >>> (48 - bits)).toInt
    }

    def nextDouble: Double = ((next(26).toLong << 27) + next(27)) * (1.0 / (1L << 53))

    def nextBoolean: Boolean = next(1) == 0
}
```

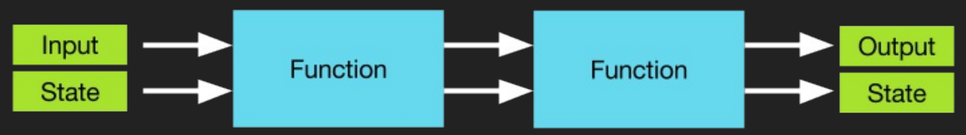
Each call generates the next value of the pseudo-random sequence.
Variable `seed` stores the last value.

How to make it pure?

IMPERATIVE CODE



FUNCTIONAL CODE



PURE VERSION OF RANDOM

```
case class Seed(value: Long) {
    private def next(bits: Int): (Seed, Int) = {
        val newSeed = Seed((value * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1))
        (newSeed, (newSeed.value >>> (48 - bits)).toInt)
    }

    def nextBoolean: (Seed, Boolean) = {
        val (newSeed, bit) = next(1)
        (newSeed, bit == 0)
    }

    def nextDouble: (Seed, Double) = {
        val (newSeed1, leftPart) = next(26)
        val (newSeed2, rightPart) = newSeed1.next(27)
        val output = ((leftPart.toLong << 27) + rightPart) * (1.0 / (1L << 53))
        (newSeed2, output)
    }
}
```

BACK TO THE FAMILY

```
object FlyGenerator {

    def spawnFly(seed: Seed): (Seed, Fly) = {
        val (newSeed, b) = seed.nextBoolean
        (newSeed, Fly(b))
    }

    def spawnChild(seed: Seed, mother: Fly, father: Fly): (Seed, Fly) = (mother, father)
        case (Fly(a), Fly(b)) if a == b => (seed, Fly(a))
        case (Fly(a), Fly(b)) =>
            val (newSeed, d) = seed.nextDouble
            val newFly = if(d <= 0.6) Fly(a) else Fly(b)
            (newSeed, newFly)
    }

    def spawnFamily(seed: Seed): (Seed, (Fly, Fly, Fly)) = {
        val (newSeed1, mother) = spawnFly(seed)
        val (newSeed2, father) = spawnFly(newSeed1)
        val (newSeed3, child) = spawnChild(newSeed2, mother, father)
        (newSeed3, (mother, father, child))
    }
}
```

BACK TO THE FAMILY

It works just fine:

```
scala> FlyGenerator.spawnFamily(Seed(42))
res7: (Seed, (Fly, Fly, Fly)) = (Seed(149370390209998),(Fly(true),Fly(false),Fly(false)))
```

But it doesn't feel right: the code is clumsy.

ABSTRACTION TIME

What is it we do? We pass a state (the seed) throught chained calls to functions of type `S => (S, A)`. Let's wrap those functions.

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

    def get(s: S): A = run(s)._2

    def map[B](f: A => B): State[S, B] = State { state =>
        val (newState, a) = run(state)
        (newState, f(a))
    }

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

}

object State {
    def pure[S, A](a: A): State[S, A] = State(s => (s, a))
}
```

STATE-POWERED SEED

```
case class Seed(value: Long) {
    private def next(bits: Int): (Seed, Int) = {
        val newSeed = Seed((value * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1))
        (newSeed, (newSeed.value >>> (48 - bits)).toInt)
    }

    def nextBoolean: (Seed, Boolean) = {
        val (newSeed, bit) = next(1)
        (newSeed, bit == 0)
    }

    def nextDouble: (Seed, Double) = {
        val (newSeed1, leftPart) = next(26)
        val (newSeed2, rightPart) = newSeed1.next(27)
        val output = ((leftPart.toLong << 27) + rightPart) * (1.0 / (1L << 53))
        (newSeed2, output)
    }
}

object Seed { // only this companion object is new
    def nextBoolean: State[Seed, Boolean] = State(_.nextBoolean)
    def nextDouble: State[Seed, Double] = State(_.nextDouble)
}
```

STATE-POWERED FLIES

```
object FlyGenerator {

    def spawnFly: State[Seed, Fly] = Seed.nextBoolean.map(Fly(_))

    def spawnChild(mother: Fly, father: Fly): State[Seed, Fly] = (mother, father) match {
        case (Fly(a), Fly(b)) if a == b => State.pure(Fly(a))
        case (Fly(a), Fly(b)) => Seed.nextDouble.map { d =>
            if(d <= 0.6) Fly(a) else Fly(b)
        }
    }

    def spawnFamily: State[Seed, (Fly, Fly, Fly)] =
        spawnFly.flatMap { mother =>
            spawnFly.flatMap { father =>
                spawnChild(mother, father).map { child =>
                    (mother, father, child)
                }
            }
        }
}
```

STATE-POWERED FLIES

```
object FlyGenerator {

    def spawnFly: State[Seed, Fly] = Seed.nextBoolean.map(Fly(_))

    def spawnChild(mother: Fly, father: Fly): State[Seed, Fly] = (mother, father) match {
        case (Fly(a), Fly(b)) if a == b => State.pure(Fly(a))
        case (Fly(a), Fly(b)) => Seed.nextDouble.map { d =>
            if(d <= 0.6) Fly(a) else Fly(b)
        }
    }

    def spawnFamily: State[Seed, (Fly, Fly, Fly)] =
        for {
            mother <- spawnFly
            father <- spawnFly
            child  <- spawnChild(mother, father)
        } yield (mother, father, child)
}
```

STATE-POWERED FLIES

```
scala> FlyGenerator.spawnFamily
res0: StateBlock.State[SeedBlock.Seed,(Fly, Fly, Fly)] = State(StateBlock$State$$Lambda$1

scala> FlyGenerator.spawnFamily.run
res1: SeedBlock.Seed => (SeedBlock.Seed, (Fly, Fly, Fly)) = StateBlock$State$$Lambda$1731

scala> FlyGenerator.spawnFamily.run(Seed(42))
res2: (SeedBlock.Seed, (Fly, Fly, Fly)) = (Seed(149370390209998),(Fly(true),Fly(false),Fl

scala> FlyGenerator.spawnFamily.get(Seed(42))
res3: (Fly, Fly, Fly) = (Fly(true),Fly(false),Fly(false))
```

OPTION

```scala
def relativeDifference(a: Double, b: Double): Option[Double] = a match {
    case 0 => None
    case _ => Some((b - a) / a)
}

def findProductLength(productDB: Map[String, Double], id: String): Option[Double] =
    productDB.get(id)

def compareLengths(productDB: Map[String, Double],
                   productA: String, productB: String): Option[Double] =
    for {
        lengthA <- findProductLength(productDB, productA)
        lengthB <- findProductLength(productDB, productB)
        diff <- relativeDifference(lengthA, lengthB)
    } yield diff
```

SEQUENCE

```
for {
    i <- 0 until 10
    j <- 0 until 10
} yield (i, j)
```

FUTURE

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

def hardMath1(x: Double): Future[Double] = Future {
    Thread.sleep(1000)
    x * x
}

def hardMath2(x: Double, y: Double): Future[Double] = Future {
    Thread.sleep(1000)
    x + y
}

for {
    x <- hardMath1(42.0)
    y <- hardMath1(13.0)
    z <- hardMath2(x, y)
} yield z
```

HERE COMES THE MONAD

Surely there is no similarity of usage between `State`, `Seq`, `Option` and `Future`. The similarity is in the structure:

```
def pure[A](a: A): F[A]
def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
def map[A, B](fa: F[A])(f: A => B): F[B] = flatMap(fa)(a => pure(f(a)))
```

This structure allows to chain computations, and is called a `Monad`.

A `Monad` is a context enriching the enclosed value with more information.

Why bother? Code reuse!

WAIT FOR A SEQUENCE OF COMPUTATIONS

```
import cats.Monad, cats.instances.all._
// import cats.Monad
// import cats.instances.all._

import scala.concurrent.Await, scala.concurrent.duration.Duration
// import scala.concurrent.Await
// import scala.concurrent.duration.Duration

def hardMath(i: Int) = Future(i)
// hardMath: (i: Int)scala.concurrent.Future[Int]

val futureList = (0 until 10).map(hardMath).toList
// futureList: List[scala.concurrent.Future[Int]] = List(Future(Success(0)), Future(Succe

val listFuture = Monad[Future].sequence(futureList)
// listFuture: scala.concurrent.Future[List[Int]] = Future(Success(List(0, 1, 2, 3, 4, 5,

Await.result(listFuture, Duration.Inf)
// res11: List[Int] = List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

FAIL ON A SEQUENCE OF OPERATIONS

```
def failOnZero(i: Int): Option[Int] = i match {
    case 0 => None
    case _ => Some(i)
}
// failOnZero: (i: Int)Option[Int]

val optionList = (0 until 10).map(failOnZero).toList
// optionList: List[Option[Int]] = List(None, Some(1), Some(2), Some(3), Some(4), Some(5)

val listOption = Monad[Option].sequence(optionList)
// listOption: Option[List[Int]] = None
```

REWRITE SEED USING CATS'S STATE

```scala
import cats.data.State

case class Seed(value: Long) {
    private def next(bits: Int): (Seed, Int) = {
        val newSeed = Seed((value * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1))
        (newSeed, (newSeed.value >>> (48 - bits)).toInt)
    }

    def nextBoolean: (Seed, Boolean) = {
        val (newSeed, bit) = next(1)
        (newSeed, bit == 0)
    }

    def nextDouble: (Seed, Double) = {
        val (newSeed1, leftPart) = next(26)
        val (newSeed2, rightPart) = newSeed1.next(27)
        val output = ((leftPart.toLong << 27) + rightPart) * (1.0 / (1L << 53))
        (newSeed2, output)
    }
}

object Seed { // only this companion object is new
    def nextBoolean: State[Seed, Boolean] = State(_.nextBoolean)
    def nextDouble: State[Seed, Double] = State(_.nextDouble)
}
```

REWRITE FLY USING CATS'S STATE

```
object FlyGenerator {

    def spawnFly: State[Seed, Fly] = Seed.nextBoolean.map(Fly(_))

    def spawnChild(mother: Fly, father: Fly): State[Seed, Fly] = (mother, father) match {
        case (Fly(a), Fly(b)) if a == b => State.pure(Fly(a))
        case (Fly(a), Fly(b)) => Seed.nextDouble.map { d =>
            if(d <= 0.6) Fly(a) else Fly(b)
        }
    }

    def spawnFamily: State[Seed, (Fly, Fly, Fly)] =
        for {
            mother <- spawnFly
            father <- spawnFly
            child  <- spawnChild(mother, father)
        } yield (mother, father, child)
}
```

CREATE MANY FLY FAMILIES

```
type SeedState[A] = State[Seed, A]
// defined type alias SeedState

val stateList = List.fill(10)(FlyGenerator.spawnFamily)
// stateList: List[cats.data.State[SeedBlock.Seed,(Fly, Fly, Fly)]] = List(cats.data.Stat

val listState = Monad[SeedState].sequence(stateList)
// listState: SeedState[List[(Fly, Fly, Fly)]] = cats.data.StateT@7508afd2

listState.runA(Seed(42)).value
// res14: List[(Fly, Fly, Fly)] = List((Fly(true),Fly(false),Fly(false)), (Fly(true),Fly(
```

SEQUENCE IS FOR FREE

- There is a method `sequence` for all monads!
- The implementation is the same for all of them: it is independant of the specific monad.
- A new form of code factorization, based on structure!

MONAD LAWS

Having methods `pure` and `flatMap` is sufficient to infer `sequence`...
Almost.

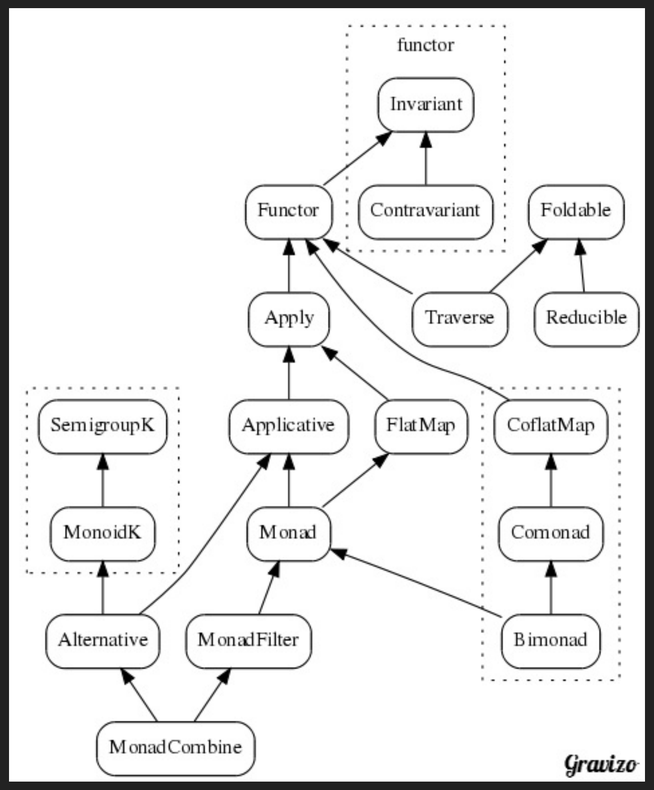
Monad laws for monad `F`:

```
// Left identity (for all f: A => F[B])
F.pure(a).flatMap(f) <-> f(a)

// Right identity (for all fa: F[A])
fa.flatMap(F.pure) <-> fa

// Associativity (for all fa: F[A], f: A => F[B], g: B => F[C])
fa.flatMap(f).flatMap(g) <-> fa.flatMap(a => f(a).flatMap(g))
```

MANY MORE MONADS (CHAINED EFFECTS)

How do I	Answer
perform non-local control flow	`Option`, `Either`, `Try`
access configuration	`Reader`
maintain a state	`State`
perform IO	`Show`
compute in parallel	`Future`
log things	`Writer`
work with multiple values	`Seq`, `Set`, `Stream`, ...

MANY MORE TYPECLASSES IN CATS



HOW DO I COMBINE MONADS?

This is a hard problem.

Current answer: stack of monad transformers (see Haskell's `mtl` library).

A new contender: free-er monads. But only accepts commutative effects.

CONCLUSION

- Real-world FP code implies highly abstract constructs
- Research is going fast: see Haskell FP community
- Many of those concepts will be familiar to most devs in a few years

Q & A