DERIVING FUNCTIONAL PROGRAMMING

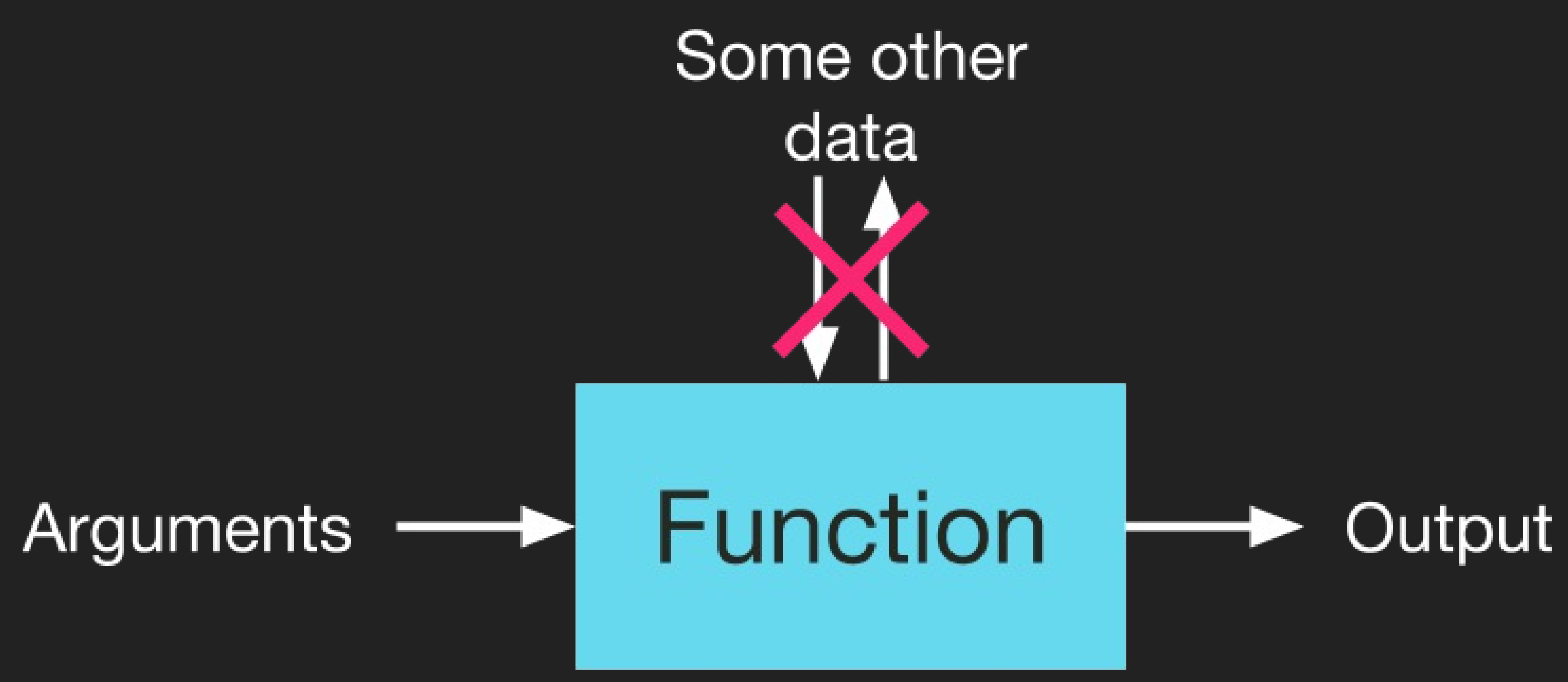
FP is characterized by one rule:

> *No side effects.*

TWO TYPES OF SIDE EFFECTS

- A function does not affect outside data
- A function is not affected by outside data

The function is said to be *referentially transparent*.



FORBIDDEN: EXAMPLE #1

```
trait MoneyConverter {
    var exchangeRate: Double = _

    def convert(value: Double): Double = exchangeRate * value
}
```

FORBIDDEN: EXAMPLE #2

```scala
trait MoneyConverter {
    var exchangeRate: Double = _

    def updateExchangeRate(): Unit = {
        exchangeRate = 1.2
    }
}
```

FORBIDDEN: EXAMPLE #3

```
scala> def eraseFirst(array: Array[Int]): Array[Int] = {
     |     array(0) = 0
     |     array
     | }
eraseFirst: (array: Array[Int])Array[Int]

scala> val someArray = Array(4, 2)
someArray: Array[Int] = Array(4, 2)

scala> eraseFirst(someArray)
res0: Array[Int] = Array(0, 2)

scala> someArray
res1: Array[Int] = Array(0, 2)
```

FORBIDDEN: EXAMPLE #4

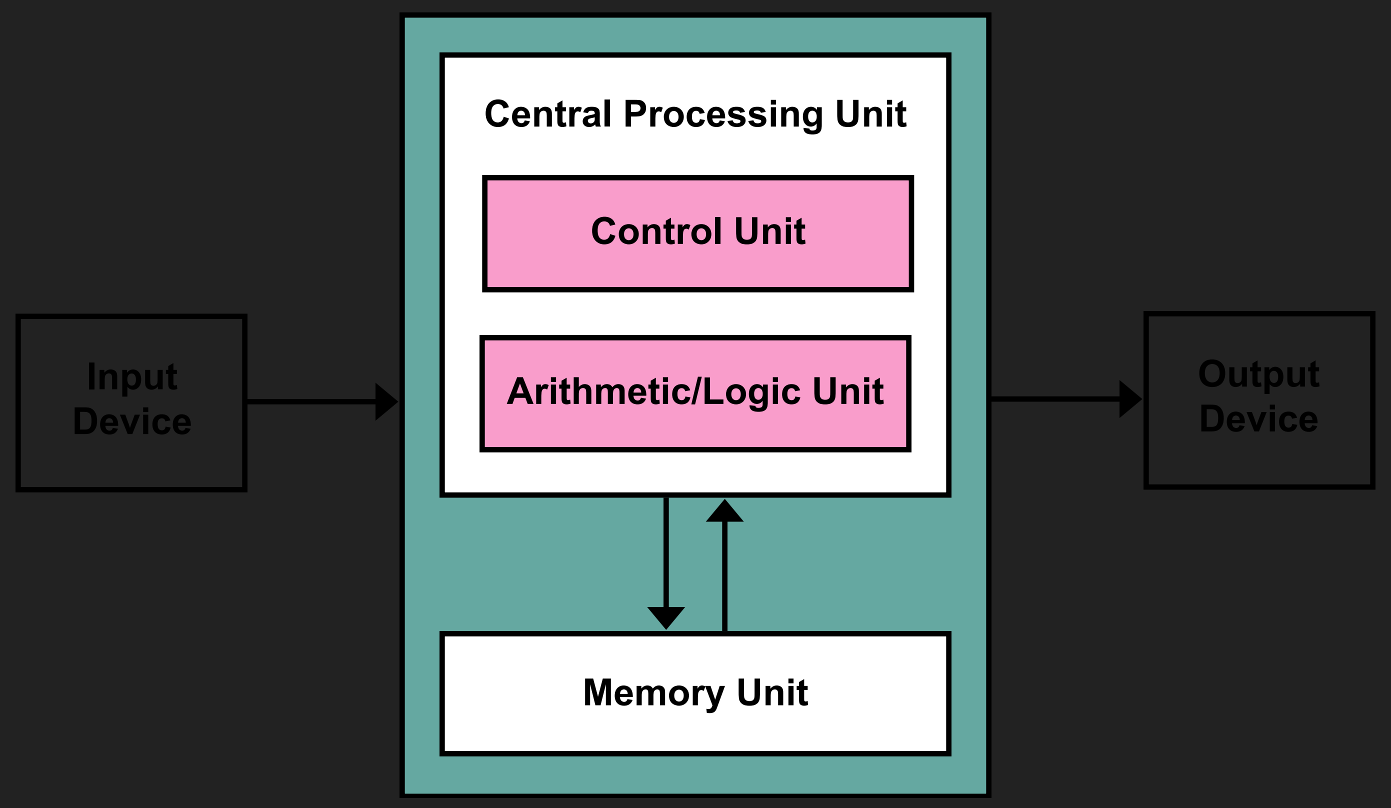
```
def formatNames(names: Seq[String]): Seq[String] = {
    var lowerCaseNames = Seq.empty[String]
    for(name <- names) {
        lowerCaseNames = lowerCaseNames :+ name.toLowerCase
    }
    lowerCaseNames
}
```

WHY FP?

When you don't aim for performance, but:

- readability
- ability to reason about the code
- parallelism/concurrency

WHY IS FP NOT MAINSTREAM?



Hardware is inherently imperative!

HISTORY

- Lambda calculus introduced in the 1930s by Church
- Lisp supported some features of FP in the 1950s
- John Backus (Turing Award, inventor of Fortran) in 1977: "Can Programming Be Liberated From the von Neumann Style?"
- ML was created in the 1970s, leading to ML variants and the Caml family
- In 1987 Haskell was born as open standard for functional programming research
- Work on Scala started in 2001, first public release in 2006

WHY FP NOW?

FP and imperative programming have a parallel history. So why is FP becoming mainstream now?

- FP is less efficient but that is less of a concern today
- FP is a natural paradigm for concurrency (see Spark)
- Some impure functional languages like Scala allow devs to try out FP
- We can reason about FP code, that is good for complex codebases
- FP + powerful compilers catch many errors at compile-time

DERIVING FP

What about the common FP characteristics?

- higher-order function
- immutable data
- lazy evaluation
- functors, monads and such
- ...

Let's derive (a reason for) them from the base rule.

> *No side effects.*

LOOPS & HIGHER-ORDER FUNCTIONS

```
def formatNames(names: Seq[String]): Seq[String] = {
    var lowerCaseNames = Seq.empty[String]
    for(name <- names) {
        lowerCaseNames = lowerCaseNames :+ name.toLowerCase
    }
    lowerCaseNames
}
```

```
def formatNames(names: Seq[String]): Seq[String] = {
    names.map(_.toLowerCase)
}
```

```
def map[B](f: A => B): Seq[B]
```

IMMUTABLE DATA

A good way to reduce side effects is to make it impossible to change the state of an object!

```
def eraseFirst(array: Array[Int]): Array[Int] = {
    array(0) = 0
    array
}
```

`Array` is mutable. Enforcing immutability makes it impossible to write `eraseFirst` this way.

WHILE LOOPS AND RECURSION

```scala
case class Node(children: List[Node])

def countNodesImp(tree: Node): Int = {
    var stack: List[Node] = List(tree)
    var nbNodes: Int = 0
    while(stack.nonEmpty) {
        nbNodes += 1
        stack = stack.head.children ++ stack.tail
    }
    nbNodes
}
```

```scala
import scala.annotation.tailrec

def countNodesRec(tree: Node): Int = {
    @tailrec
    def rec(stack: List[Node], nbNodes: Int): Int = stack match {
        case Nil => nbNodes
        case h :: tl => rec(h.children ++ tl, nbNodes + 1)
    }
    rec(List(tree), 0)
}
```

LAZY EVALUATION

Code is executed when needed.

```
scala> val primes: Stream[Int] =
     |    Stream.from(2).filter(n => !(2 until n).exists(n % _ == 0) )
primes: Stream[Int] = Stream(2, ?)

scala> println(primes.take(10).toList)
List(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
```

LAZY EVALUATION (2)

Execution order is not maintained. Did I just lose determinism!?

As long as there is no side effect, no! The result is unaffected by exectution order.

> *In imperative programming, we execute statements.*
>
> *In functional programming, we compute results.*

LAZY EVALUATION (3) & PARALLELISM

Spark code

```
val sentences: RDD[String] = ???
val words = sentences.flatMap(_.split(" "))
println(words.countApproxDistinct)
```

Spark logs show computation order is random.

RECAP

FP requires or is facilitated by	FP enables
Immutable data	Readability
Higher-order functions	(Compiler) reasoning
Recursion	- Lazy evaluation
	- Parallelism
	- Memoization

WAIT, IS THAT ALL?

How do I code in practice?

REAL-WORLD CODE

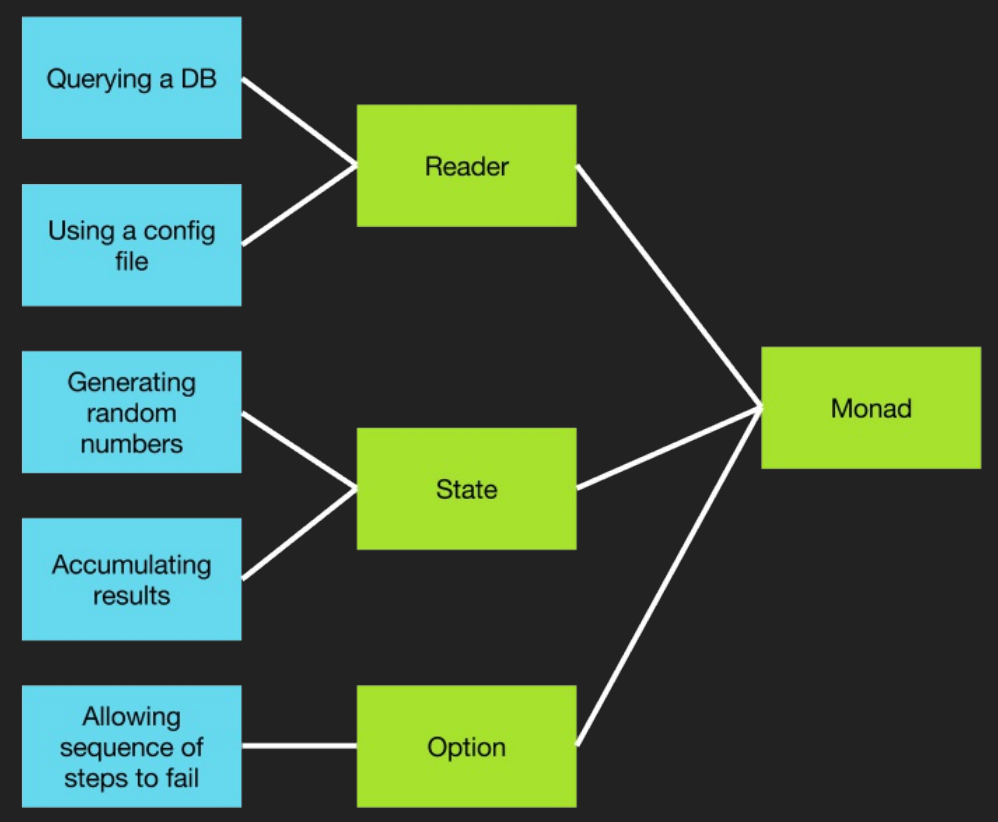
How do I	Answer
perform non-local control flow	`Option`, `Either`, `Try`
access configuration	`Reader`
maintain a state	`State`
perform IO	`Show`
compute in parallel	`Future`
log things	`Writer`
work with multiple values	`Seq`, `Set`, `Stream`, ...

These are called "effects". They make it possible to handle real cases without side effects.

EFFECTS

Effects are different in nature, but behave similarly.

Depending on their properties, they can be functors, applicatives, monads.



This second level of abstraction enables massive factorization of code, and sheds new light on the programming process.

That's for next time!

Q & A