Scala code examples for Functional Programming Jargon

- This project is fork of hemanth/functional-programming-jargon

Functional Programming Jargon

Functional programming (FP) provides many advantages, and its popularity has been increasing as a result. However, each programming paradigm comes with its own unique jargon and FP is no exception. By providing a glossary, we hope to make learning FP easier.

Examples are presented in JavaScript (ES2015). Why JavaScript?

This is a WIP; please feel free to send a PR ;)

Where applicable, this document uses terms defined in the Fantasy Land spec

Translations

- Portuguese
- Spanish
- Chinese
- Bahasa Indonesia

Table of Contents

Arity

The number of arguments a function takes. From words like unary, binary, ternary, etc. This word has the distinction of being composed of two suffixes, "-ary" and "-ity." Addition, for example, takes two arguments, and so it is defined as a binary function or a function with an arity of two. Such a function may sometimes be called "dyadic" by people who prefer Greek roots to Latin. Likewise, a function that takes a variable number of arguments is called "variadic," whereas a binary function must be given two and only two arguments, currying and partial application notwithstanding (see below).

```
val sum = (a : Int, b: Int) => a + b // The arity of sum is 2
// sum: (Int, Int) => Int = $$Lambda$6089/41702096@2b0a78f5
```

Higher-Order Functions (HOF)

A function which takes a function as an argument and/or returns a function.

```
val filter = (predicate: Int => Boolean, xs: List[Int]) => xs.filter(predicate)
// filter: (Int => Boolean, List[Int]) => List[Int] = $$Lambda$6101/1664822486@6c2961d4


val isEven = (x: Int) => x % 2 == 0
// isEven: Int => Boolean = $$Lambda$6102/1380731259@7caf5594


filter(isEven, List(1, 2, 3, 4, 5))
// res0: List[Int] = List(2, 4)
```

Partial Application

Partially applying a function means creating a new function by pre-filling some of the arguments to the original function.

```
// Something to apply
val add3 = (a: Int, b: Int, c: Int) => a + b + c
// add3: (Int, Int, Int) => Int = $$Lambda$6104/749395786@26f51ccc

// Partially applying `2` and `3` to `add3` gives you a one-argument function
val fivePlus = add3(2, 3, _: Int) // (c) => 2 + 3 + c
// fivePlus: Int => Int = $$Lambda$6105/747846882@57c90a91
```

```
fivePlus(4)
// res3: Int = 9
```

Partial application helps create simpler functions from more complex ones by baking in data when you have it. Curried functions are automatically partially applied.

Currying

The process of converting a function that takes multiple arguments into a function that takes them one at a time.

Each time the function is called it only accepts one argument and returns a function that takes one argument until all arguments are passed.

```
val sum = (a : Int, b: Int) => a + b
// sum: (Int, Int) => Int = $$Lambda$6106/477864989@f720751

val curriedSum = (a: Int) => (b: Int) => a + b
// curriedSum: Int => (Int => Int) = $$Lambda$6107/1895392220@447d87e6

curriedSum(40)(2) // 42.
// res4: Int = 42

val add2 = curriedSum(2) // (b) => 2 + b
// add2: Int => Int = $$Lambda$6108/1784411761@45922de3

add2(10) // 12
// res5: Int = 12
```

Closure

A closure is a way of accessing a variable outside its scope. Formally, a closure is a technique for implementing lexically scoped named binding. It is a way of storing a function with an environment.

A closure is a scope which captures local variables of a function for access even after the execution has moved out of the block in which it is defined. ie. they allow referencing a scope after the block in which the variables were declared has finished executing.

```
val addTo = (x :Int) => (y: Int) => x + y
// addTo: Int => (Int => Int) = $$Lambda$6109/741101144@c019bed

val addToFive = addTo(5)
// addToFive: Int => Int = $$Lambda$6110/1430362686@30d35a2d

addToFive(3) // returns 8
// res6: Int = 8
```

The function `addTo()` returns a function(internally called `add()`), lets store it in a variable called `addToFive` with a curried call having parameter 5.

Ideally, when the function `addTo` finishes execution, its scope, with local variables add, x, y should not be accessible. But, it returns 8 on calling `addToFive()` . This means that the state of the function `addTo` is saved even after the block of code has finished executing, otherwise there is no way of knowing that `addTo` was called as `addTo(5)` and the value of x was set to 5.

Lexical scoping is the reason why it is able to find the values of x and add - the private variables of the parent which has finished executing. This value is called a Closure.

The stack along with the lexical scope of the function is stored in form of reference to the parent. This prevents the closure and the underlying variables from being garbage collected(since there is at least one live reference to it).

Lambda Vs Closure: A lambda is essentially a function that is defined inline rather than the standard method of declaring functions. Lambdas can frequently be passed around as objects.

A closure is a function that encloses its surrounding state by referencing fields external to its body. The enclosed state remains across invocations of the closure.

Further reading/Sources

- Lambda Vs Closure
- How do JavaScript Closures Work?

Auto Currying

Transforming a function that takes multiple arguments into one that if given less than its correct number of arguments returns a function that takes the rest. When the function gets the correct number of arguments it is then evaluated.

lodash & Ramda have a `curry` function that works this way.

```
val add = (x: Int, y: Int) => x + y
// add: (Int, Int) => Int = $$Lambda$6111/885864900@6fc805b3

val curriedAdd = add.curried
// curriedAdd: Int => (Int => Int) = scala.Function2$$Lambda$3285/878350569@583d1436

curriedAdd(2) // (y) => 1 + y
// res7: Int => Int = scala.Function2$$Lambda$3286/1189535279@75c6a7a5

curriedAdd(1)(2) // 3
// res8: Int = 3
```

Further reading

- Favoring Curry
- Hey Underscore, You're Doing It Wrong!

Function Composition

The act of putting two functions together to form a third function where the output of one function is the input of the other.

```
def compose[A, B, C](f: B => C, g: A => B) = (a: A) => f(g(a)) // Definition
// compose: [A, B, C](f: B => C, g: A => B)A => C

val floorAndToString = compose((x: Double) => x.toString, math.floor) // Usage
// floorAndToString: Double => String = $$Lambda$6114/392323038@14236e09

floorAndToString(121.212121) // '121.0'
// res9: String = 121.0
```

Continuation

At any given point in a program, the part of the code that's yet to be executed is known as a continuation.

```
def printAsString(num: Int) = println(s"Given $num")
// printAsString: (num: Int)Unit

val printAsString= (num: Int) => println(s"Given $num")
// printAsString: Int => Unit = $$Lambda$6115/353404609@582dea2f

val addOneAndContinue = (num: Int, cc: Int => Any) => {
  val result = num + 1
  cc(result)
}
// addOneAndContinue: (Int, Int => Any) => Any = $$Lambda$6116/209832866@9488095

addOneAndContinue(2, printAsString)
// Given 3
// res10: Any = ()
```

Continuations are often seen in asynchronous programming when the program needs to wait to receive data before it can continue. The response is often passed off to the rest of the program, which is the continuation, once it's been received.

```
def continueProgramWith(data: String) = {
  // Continues program with data
```

```scala
}
// continueProgramWith: (data: String)Unit

def readFileAsync(file: String, cb: (Option[Throwable], String) => Unit) = {}
// readFileAsync: (file: String, cb: (Option[Throwable], String) => Unit)Unit

readFileAsync("path/to/file", (err, response) => {
  if (err.isDefined) {
    // handle error
    ()
  }
  continueProgramWith(response)
})
```

Purity

A function is pure if the return value is only determined by its input values, and does not produce side effects.

```scala
val greet = (name: String) => s"Hi, ${name}"
// greet: String => String = $$Lambda$6118/976300008@d803871

greet("Brianne")
// res12: String = Hi, Brianne
```

As opposed to each of the following:

```scala
var name = "Brianne"
// name: String = Brianne

def greet = () => s"Hi, ${name}"
// greet: () => String

greet()
// res13: String = Hi, Brianne
```

The above example's output is based on data stored outside of the function...

```scala
var greeting: String = _
// greeting: String = null

val greet = (name: String) => {
  greeting = s"Hi, ${name}"
}
// greet: String => Unit = $$Lambda$6120/1120068379@6bf02320

greet("Brianne")

greeting
// res15: String = Hi, Brianne
```

... and this one modifies state outside of the function.

Side effects

A function or expression is said to have a side effect if apart from returning a value, it interacts with (reads from or writes to) external mutable state.

```scala
import java.util.Date
// import java.util.Date

def differentEveryTime = new Date()
// differentEveryTime: java.util.Date


println("IO is a side effect!")
// IO is a side effect!
```

Idempotent

A function is idempotent if reapplying it to its result does not produce a different result.

```
f(f(x)) ≍ f(x)
```

```
math.abs(math.abs(10))
// res17: Int = 10
```

```
def sort[A: Ordering](xs: List[A]) = xs.sorted
// sort: [A](xs: List[A])(implicit evidence$1: Ordering[A])List[A]

sort(sort(sort(List(2, 1))))
// res18: List[Int] = List(1, 2)
```

Point-Free Style

Writing functions where the definition does not explicitly identify the arguments used. This style usually requires currying or other Higher-Order functions. A.K.A Tacit programming.

```
// Given
def map[A, B](fn: A => B) = (list: List[A]) => list.map(fn)
// map: [A, B](fn: A => B)List[A] => List[B]

val add = (a: Int) => (b: Int) => a + b
// add: Int => (Int => Int) = $$Lambda$6121/1340871739@61a102c5

// Then

// Not points-free - `numbers` is an explicit argument
val incrementAll = (numbers: List[Int]) => map(add(1))(numbers)
// incrementAll: List[Int] => List[Int] = $$Lambda$6122/1403313951@7e5972fd

// Points-free - The list is an implicit argument
val incrementAll2 = map(add(1))
// incrementAll2: List[Int] => List[Int] = $$Lambda$6124/221818483@221baf45
```

`incrementAll` identifies and uses the parameter `numbers`, so it is not points-free. `incrementAll2` is written just by combining functions and values, making no mention of its arguments. It **is** points-free.

Points-free function definitions look just like normal assignments without `function` or `=>`.

Predicate

A predicate is a function that returns true or false for a given value. A common use of a predicate is as the callback for array filter.

```
val predicate = (a: Int) => a > 2
// predicate: Int => Boolean = $$Lambda$6125/2085181758@1e842ab7

List(1, 2, 3, 4).filter(predicate)
// res24: List[Int] = List(3, 4)
```

Contracts

A contract specifies the obligations and guarantees of the behavior from a function or expression at runtime. This acts as a set of rules that are expected from the input and output of a function or expression, and errors are generally reported whenever a contract is violated.

```
scala> // Define our contract : int -> int
     | def contract[A](input: A): Boolean = input match {
     |   case _ : Int => true
```

```
     |   case _ => throw new Exception("Contract violated: expected int -> int")
     | }
contract: [A](input: A)Boolean

scala> def addOne[A](num: A): Int = {
     |   contract(num)
     |   num.asInstanceOf[Int] + 1
     | }
addOne: [A](num: A)Int

scala> addOne(2) // 3
res26: Int = 3

scala> addOne("some string") // Contract violated: expected int -> int
java.lang.Exception: Contract violated: expected int -> int
  at .contract(<console>:16)
  at .addOne(<console>:15)
  ... 43 elided
```

Category

A category in category theory is a collection of objects and morphisms between them. In programming, typically types act as the objects and functions as morphisms.

To be a valid category 3 rules must be met:

1. There must be an identity morphism that maps an object to itself. Where `a` is an object in some category, there must be a function from `a -> a`.
2. Morphisms must compose. Where `a`, `b`, and `c` are objects in some category, and `f` is a morphism from `a -> b`, and `g` is a morphism from `b -> c`; `g(f(x))` must be equivalent to `(g • f)(x)`.
3. Composition must be associative `f • (g • h)` is the same as `(f • g) • h`

Since these rules govern composition at very abstract level, category theory is great at uncovering new ways of composing things.

Further reading

- Category Theory for Programmers

Value

Anything that can be assigned to a variable.

```
case class Person(name: String, age: Int)
// defined class Person

5
// res28: Int = 5

Person("John", 30)
// res29: Person = Person(John,30)

(a: Any) => a
// res30: Any => Any = $$Lambda$6126/1828268232@117ca3b4

List(1)
// res31: List[Int] = List(1)

null
// res32: Null = null
```

Constant

A variable that cannot be reassigned once defined.

```
val five = 5
// five: Int = 5
```

```scala
val john = Person("John", 30)
// john: Person = Person(John,30)
```

Constants are referentially transparent. That is, they can be replaced with the values that they represent without affecting the result.

With the above two constants the following expression will always return `true`.

```scala
john.age + five == Person("John", 30).age + 5
// res33: Boolean = true
```

Functor

An object that implements a `map` function which, while running over each value in the object to produce a new object, adheres to two rules:

Preserves identity

```
object.map(x => x) ≍ object
```

Composable

```
object.map(compose(f, g)) ≍ object.map(g).map(f)
```

(`f`, `g` are arbitrary functions)

A common functor in JavaScript is `Array` since it abides to the two functor rules:

```scala
List(1, 2, 3).map(x => x)
// res34: List[Int] = List(1, 2, 3)
```

and

```scala
val f = (x: Int) => x + 1
// f: Int => Int = $$Lambda$6128/977330834@2ef88643

val g = (x: Int) => x * 2
// g: Int => Int = $$Lambda$6129/739719665@1b4dfd52

List(1, 2, 3).map(x => f(g(x)))
// res35: List[Int] = List(3, 5, 7)

List(1, 2, 3).map(g).map(f)
// res36: List[Int] = List(3, 5, 7)
```

Pointed Functor

An Applicative with an `pure` function that puts *any* single value into it.

cats adds `Applicative#pure` making arrays a pointed functor.

```scala
import cats._
// import cats._

import cats.implicits._
// import cats.implicits._

Applicative[List].pure(1)
// res37: List[Int] = List(1)
```

Lift

Lifting is when you take a value and put it into an object like a functor. If you lift a function into an Applicative Functor then you can make it work on values that are also in that functor.

Some implementations have a function called `lift`, or `liftA2` to make it easier to run functions on functors.

```
def liftA2[F[_]: Monad, A, B, C](f: A => B => C)(a: F[A], b: F[B]) = {
  a.map(f).ap(b)
}
// warning: there was one feature warning; for details, enable `:setting -feature' or `:replay -feature'
// liftA2: [F[_], A, B, C](f: A => (B => C))(a: F[A], b: F[B])(implicit evidence$1: cats.Monad[F])F[C]

val mult = (a: Int) => (b: Int) => a * b
// mult: Int => (Int => Int) = $$Lambda$6131/1684188711@6840cae6

val liftedMult = liftA2[List, Int, Int, Int](mult) _
// liftedMult: (List[Int], List[Int]) => List[Int] = $$Lambda$6132/1015569705@69b98e8

liftedMult(List(1, 2), List(3))
// res38: List[Int] = List(3, 6)

liftA2((a: Int) => (b: Int) => a + b)(List(1, 2), List(3, 4))
// res39: List[Int] = List(4, 5, 5, 6)
```

Lifting a one-argument function and applying it does the same thing as `map`.

```
val increment = (x: Int) => x + 1
// increment: Int => Int = $$Lambda$6137/1176025030@60dd7083

Applicative[List].lift(increment)(List(2))
// res40: List[Int] = List(3)

List(2).map(increment)
// res41: List[Int] = List(3)
```

Referential Transparency

An expression that can be replaced with its value without changing the behavior of the program is said to be referentially transparent.

Say we have function greet:

```
val greet = () => "Hello World!"
// greet: () => String = $$Lambda$6139/1754785613@1d1c9e6a
```

Any invocation of `greet()` can be replaced with `Hello World!` hence greet is referentially transparent.

Equational Reasoning

When an application is composed of expressions and devoid of side effects, truths about the system can be derived from the parts.

Lambda

An anonymous function that can be treated like a value.

```
(_: Int) + 1
// res42: Int => Int = $$Lambda$6140/835760611@2344fbd6

(x: Int) => x + 1
// res43: Int => Int = $$Lambda$6141/605433877@12d4147c
```

Lambdas are often passed as arguments to Higher-Order functions.

```
List(1, 2).map(_ + 1)
// res44: List[Int] = List(2, 3)
```

You can assign a lambda to a variable.

```
val add1 = (a: Int) => a + 1
// add1: Int => Int = $$Lambda$6143/1533951980@6546b471
```

Lambda Calculus

A branch of mathematics that uses functions to create a universal model of computation.

Lazy evaluation

Lazy evaluation is a call-by-need evaluation mechanism that delays the evaluation of an expression until its value is needed. In functional languages, this allows for structures like infinite lists, which would not normally be available in an imperative language where the sequencing of commands is significant.

```
lazy val rand: Double = {
  println("generate random value...")
  math.random()
}
// rand: Double = <lazy>


rand // Each execution gives a random value, expression is evaluated on need.
// generate random value...
// res45: Double = 0.7668030551457087
```

Monoid

An object with a function that "combines" that object with another of the same type.

One simple monoid is the addition of numbers:

```
1 + 1
// res46: Int = 2
```

In this case number is the object and `+` is the function.

An "identity" value must also exist that when combined with a value doesn't change it.

The identity value for addition is `0`.

```
1 + 0
// res47: Int = 1
```

It's also required that the grouping of operations will not affect the result (associativity):

```
1 + (2 + 3) == (1 + 2) + 3
// res48: Boolean = true
```

Array concatenation also forms a monoid:

```
List(1, 2) ::: List(3, 4)
// res49: List[Int] = List(1, 2, 3, 4)
```

The identity value is empty array `[]`

```
List(1, 2) ::: List()
// res50: List[Int] = List(1, 2)
```

If identity and compose functions are provided, functions themselves form a monoid:

```scala
def identity[A](a: A): A = a
// identity: [A](a: A)A

def compose[A, B, C](f: B => C, g: A => B) = (a: A) => f(g(a)) // Definition
// compose: [A, B, C](f: B => C, g: A => B)A => C
```

`foo` is any function that takes one argument.

```
compose(foo, identity) ≍ compose(identity, foo) ≍ foo
```

Monad

A monad is an object with `pure` and `flatMap` functions. `flatMap` is like `map` except it un-nests the resulting nested object.

```scala
// Implementation
trait Monad[F[_]] {
  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]
}
// warning: there was one feature warning; for details, enable `:setting -feature' or `:replay -feature'
// defined trait Monad
// warning: previously defined object Monad is not a companion to trait Monad.
// Companions must be defined together; you may wish to use :paste mode for this.

object Monad {
  def apply[F[_]](implicit ev: Monad[F]) = ev

  implicit val listInstance: Monad[List] = new Monad[List] {
    def pure[A](x: A) = List(x)

    def flatMap[A, B](fa: List[A])(f: A => List[B]): List[B] =
      fa.foldLeft(List[B]()) { (acc, x) => acc ::: f(x) }

    def map[A, B](fa: List[A])(f: A => B): List[B] =
      flatMap(fa)(x => pure(f(x)))

  }
}
// warning: there was one feature warning; for details, enable `:setting -feature' or `:replay -feature'
// defined object Monad
// warning: previously defined trait Monad is not a companion to object Monad.
// Companions must be defined together; you may wish to use :paste mode for this.

import Monad._
// import Monad._

// Usage
Monad[List].flatMap(List("cat,dog", "fish,bird"))(a => a.split(",").toList)
// res53: List[String] = List(cat, dog, fish, bird)

// Contrast to map
Monad[List].map(List("cat,dog", "fish,bird"))(a => a.split(",").toList)
// res55: List[List[String]] = List(List(cat, dog), List(fish, bird))
```

`pure` is also known as `return` in other functional languages. `flatMap` is also known as `bind` in other languages.

Comonad

An object that has `extract` and `coflatMap` functions.

```scala
trait Comonad[F[_]] {
  def extract[A](x: F[A]): A
  def coflatMap[A, B](fa: F[A])(f: F[A] => B): F[B]
}
// warning: there was one feature warning; for details, enable `:setting -feature' or `:replay -feature'
// defined trait Comonad
// warning: previously defined object Comonad is not a companion to trait Comonad.
```

```
// Companions must be defined together; you may wish to use :paste mode for this.

type Id[X] = X
// defined type alias Id

def id[X](x: X): Id[X] = x
// id: [X](x: X)Id[X]

object Comonad {
  def apply[F[_]](implicit ev: Comonad[F]) = ev

  implicit val idInstance: Comonad[Id] = new Comonad[Id] {
    def extract[A](x: Id[A]): A = x
    def coflatMap[A, B](fa: Id[A])(f: Id[A] => B): Id[B] = {
      id(f(fa))
    }
  }
}
// warning: there was one feature warning; for details, enable `:setting -feature' or `:replay -feature'
// defined object Comonad
// warning: previously defined trait Comonad is not a companion to object Comonad.
// Companions must be defined together; you may wish to use :paste mode for this.
```

Extract takes a value out of a functor.

```
import Comonad._
// import Comonad._

Comonad[Id].extract(id(1))
// res56: Id[Int] = 1
```

Extend runs a function on the comonad. The function should return the same type as the comonad.

```
Comonad[Id].coflatMap[Int, Int](id(1))(co => Comonad[Id].extract(co) + 1)
// res57: Id[Int] = 2
```

Applicative Functor

An applicative functor is an object with an `ap` function. `ap` applies a function in the object to a value in another object of the same type.

```
// Implementation
trait Applicative[F[_]] {
  def ap[A, B](ff: F[A => B])(fa: F[A]): F[B]
}
// warning: there was one feature warning; for details, enable `:setting -feature' or `:replay -feature'
// defined trait Applicative
// warning: previously defined object Applicative is not a companion to trait Applicative.
// Companions must be defined together; you may wish to use :paste mode for this.

object Applicative {
  def apply[F[_]](implicit ev: Applicative[F]) = ev

  implicit val listInstance = new Applicative[List] {
    def ap[A, B](ff: List[A => B])(fa: List[A]): List[B] =
      ff.foldLeft(List[B]()) { (acc, f) => acc ::: fa.map(f) }
  }
}
// warning: there was one feature warning; for details, enable `:setting -feature' or `:replay -feature'
// defined object Applicative
// warning: previously defined trait Applicative is not a companion to object Applicative.
// Companions must be defined together; you may wish to use :paste mode for this.

import Applicative._
// import Applicative._

// Example usage
Applicative[List].ap(List((_: Int) + 1))(List(1))
// res60: List[Int] = List(2)
```

This is useful if you have two objects and you want to apply a binary function to their contents.

```
// Arrays that you want to combine
val arg1 = List(1, 3)
// arg1: List[Int] = List(1, 3)

val arg2 = List(4, 5)
// arg2: List[Int] = List(4, 5)

// combining function - must be curried for this to work
val add = (x: Int) => (y: Int) => x + y
// add: Int => (Int => Int) = $$Lambda$6151/266874536@144646d8

val partiallyAppiedAdds = Applicative[List].ap(List(add))(arg1) // [(y) => 1 + y, (y) => 3 + y]
// partiallyAppiedAdds: List[Int => Int] = List($$Lambda$6152/226672465@6ff3ff0, $$Lambda$6152/226672465@677536aa)
```

This gives you an array of functions that you can call `ap` on to get the result:

```
Applicative[List].ap(partiallyAppiedAdds)(arg2)
// res63: List[Int] = List(5, 6, 7, 8)
```

Morphism

A transformation function.

Endomorphism

A function where the input type is the same as the output.

```
// uppercase :: String -> String
val uppercase = (str: String) => str.toUpperCase
// uppercase: String => String = $$Lambda$6153/1625244377@4f348849

// decrement :: Number -> Number
val decrement = (x: Int) => x - 1
// decrement: Int => Int = $$Lambda$6154/147034451@797635f9
```

Isomorphism

A pair of transformations between 2 types of objects that is structural in nature and no data is lost.

For example, 2D coordinates could be stored as an array `[2,3]` or object `{x: 2, y: 3}`.

```
// Providing functions to convert in both directions makes them isomorphic.
case class Coords(x: Int, y: Int)
// defined class Coords

val pairToCoords = (pair: (Int, Int)) => Coords(pair._1, pair._2)
// pairToCoords: ((Int, Int)) => Coords = $$Lambda$6155/416347946@4d78afea

val coordsToPair = (coods: Coords) => (coods.x, coods.y)
// coordsToPair: Coords => (Int, Int) = $$Lambda$6156/905411695@26eed67a

coordsToPair(pairToCoords((1, 2)))
// res67: (Int, Int) = (1,2)

pairToCoords(coordsToPair(Coords(1, 2)))
// res68: Coords = Coords(1,2)
```

Setoid

An object that has an `equals` function which can be used to compare other objects of the same type.

Make array a setoid:

```
trait Eq[A] {
```

```scala
  def eqv(x: A, y: A): Boolean
}
// defined trait Eq

object Eq {
  def apply[A](implicit ev: Eq[A]) = ev

  implicit def arrayInstance[B]: Eq[Array[B]] = new Eq[Array[B]] {
    def eqv(xs: Array[B], ys: Array[B]): Boolean =
      xs.zip(ys).foldLeft(true) {
        case (isEq, (x, y)) => isEq && x == y
      }
  }

  implicit class EqOps[A](x: A) {
    def eqv(y: A)(implicit ev: Eq[A]) =
      ev.eqv(x, y)
  }
}
// defined object Eq
// warning: previously defined trait Eq is not a companion to object Eq.
// Companions must be defined together; you may wish to use :paste mode for this.

import Eq._
// import Eq._

Array(1, 2) == Array(1, 2)
// res69: Boolean = false

Array(1, 2).eqv(Array(1, 2))
// res70: Boolean = true

Array(1, 2).eqv(Array(0))
// res71: Boolean = false
```

Semigroup

An object that has a `combine` function that combines it with another object of the same type.

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

object Semigroup {
  def apply[A](implicit ev: Semigroup[A]) = ev

  implicit def listInstance[B]: Semigroup[List[B]] = new Semigroup[List[B]] {
    def combine(x: List[B], y: List[B]): List[B] = x ::: y
  }

  implicit class SemigroupOps[A](x: A) {
    def combine(y: A)(implicit ev: Semigroup[A]): A = ev.combine(x, y)
  }

}
// defined object Semigroup
// warning: previously defined trait Semigroup is not a companion to object Semigroup.
// Companions must be defined together; you may wish to use :paste mode for this.

import Semigroup._
// import Semigroup._

Semigroup[List[Int]].combine(List(1), List(2))
// res0: List[Int] = List(1, 2)
```

Semigroup must be closed under associativity and arbitrary products. (x·y)·z = x·(y·z) for all x, y and z in the semigroup.

```scala
List(1).combine(List(2)).combine(List(3))
// res1: List[Int] = List(1, 2, 3)
```

```
List(1).combine(List(2).combine(List(3)))
// res2: List[Int] = List(1, 2, 3)


## Foldable

An object that has a `foldr/l` function that can transform that object into some other type.

```scala
trait Foldable[F[_]] {
  def foldLeft[A, B](fa: F[A], b: B)(f: (B, A) => B): B
  def foldRight[A, B](fa: F[A], b: B)(f: (A, B) => B): B
}
// warning: there was one feature warning; for details, enable `:setting -feature' or `:replay -feature'
// defined trait Foldable

object Foldable {
  def apply[F[_]](implicit ev: Foldable[F]) = ev

  implicit val listInstance = new Foldable[List]{
    def foldLeft[A, B](fa: List[A], b: B)(f: (B, A) => B): B = fa match {
      case x :: xs => foldLeft(xs, f(b, x))(f)
      case Nil => b
    }

    def foldRight[A, B](fa: List[A], b: B)(f: (A, B) => B): B = fa match {
      case x :: xs => f(x, foldRight(xs, b)(f))
      case Nil => b
    }
  }
}
// defined object Foldable

import Foldable._
// import Foldable._

def sum[A](xs: List[A])(implicit N: Numeric[A]) : A =
  Foldable[List].foldLeft(xs, N.zero) {
    case (acc, x) => N.plus(acc, x)
  }
// sum: [A](xs: List[A])(implicit N: Numeric[A])A

sum(List(1, 2, 3))
// res3: Int = 6
```

Lens

A lens is a structure (often an object or function) that pairs a getter and a non-mutating setter for some other data structure.

```scala
import cats.Functor
// import cats.Functor

import monocle.PLens
// import monocle.PLens

import monocle.Lens
// import monocle.Lens

// Using [Monocle's lens](https://github.com/julien-truffaut/Monocle)

// S the source of a PLens
// T the modified source of a PLens
// A the target of a PLens
// B the modified target of a PLens
abstract class PLens[S, T, A, B] {

  /** get the target of a PLens */
  def get(s: S): A

  /** set polymorphically the target of a PLens using a function */
  def set(b: B): S => T
```
```

```scala
  /** modify polymorphically the target of a PLens using Functor function */
  def modifyF[F[_]: Functor](f: A => F[B])(s: S): F[T]

  /** modify polymorphically the target of a PLens using a function */
  def modify(f: A => B): S => T
}
// defined class PLens
// warning: previously defined object PLens is not a companion to class PLens.
// Companions must be defined together; you may wish to use :paste mode for this.

object Lens {
  /** alias for [[PLens]] apply with a monomorphic set function */
  def apply[S, A](get: S => A)(set: A => S => S): Lens[S, A] =
    PLens(get)(set)
}
// defined object Lens

case class Person(name: String)
// defined class Person

val nameLens = Lens[Person, String](_.name)(str => p => p.copy(name = str))
// nameLens: monocle.Lens[Person,String] = monocle.PLens$$anon$8@5b2aa650
```

Having the pair of get and set for a given data structure enables a few key features.

```scala
val person = Person("Gertrude Blanch")
// person: Person = Person(Gertrude Blanch)

// invoke the getter
// get :: Person => String
nameLens.get(person)
// res12: String = Gertrude Blanch

// invoke the setter
// set :: String => Person => Person
nameLens.set("Shafi Goldwasser")(person)
// res15: Person = Person(Shafi Goldwasser)

// run a function on the value in the structure
// modify :: (String => String) => Person => Person
nameLens.modify(_.toUpperCase)(person)
// res18: Person = Person(GERTRUDE BLANCH)
```

Lenses are also composable. This allows easy immutable updates to deeply nested data.

```scala
// This lens focuses on the first item in a non-empty array

def firstLens[A] = Lens[List[A], A] {
  // get first item in array
  _.head
} {
  // non-mutating setter for first item in array
  x => xs => x :: xs.tail
}
// firstLens: [A]=> monocle.Lens[List[A],A]

val people = List(Person("Gertrude Blanch"), Person("Shafi Goldwasser"))
// people: List[Person] = List(Person(Gertrude Blanch), Person(Shafi Goldwasser))

// Despite what you may assume, lenses compose left-to-right.
(firstLens composeLens nameLens).modify(_.toUpperCase)(people)
// res22: List[Person] = List(Person(GERTRUDE BLANCH), Person(Shafi Goldwasser))
```

Other implementations:

- Quicklens - Modify deeply nested case class fields
- Sauron - Yet another Scala lens macro, Lightweight lens library in less than 50-lines of Scala
- scalaz.Lens

Type Signatures

Every functions in Scala will indicate the types of their arguments and return values.

```
// functionName :: firstArgType -> secondArgType -> returnType

// add :: Number -> Number -> Number
val add = (x: Int) => (y: Int) => x + y
// add: Int => (Int => Int) = $$Lambda$6168/450833074@b76efce

// increment :: Number -> Number
val increment = (x: Int) => x + 1
// increment: Int => Int = $$Lambda$6169/2134951565@23429781
```

If a function accepts another function as an argument it is wrapped in parentheses.

```
// call :: (a -> b) -> a -> b
def call[A, B] = (f: A => B) => (x: A) => f(x)
// call: [A, B]=> (A => B) => (A => B)
```

The letters `a`, `b`, `c`, `d` are used to signify that the argument can be of any type. The following version of `map` takes a function that transforms a value of some type `a` into another type `b`, an array of values of type `a`, and returns an array of values of type `b`.

```
// map :: (a -> b) -> [a] -> [b]
def map[A, B] = (f: A => B) => (list: List[A]) => list.map(f)
// map: [A, B]=> (A => B) => (List[A] => List[B])
```

Further reading

- Ramda's type signatures
- Mostly Adequate Guide
- What is Hindley-Milner? on Stack Overflow

Algebraic data type

A composite type made from putting other types together. Two common classes of algebraic types are sum and product.

Sum type

A Sum type is the combination of two types together into another one. It is called sum because the number of possible values in the result type is the sum of the input types.

we can use `sealed trait` or `Either` to have this type:

```
// imagine that rather than sets here we have types that can only have these values
sealed trait Bool
// defined trait Bool

object True extends Bool
// defined object True

object False extends Bool
// defined object False

sealed trait HalfTrue
// defined trait HalfTrue

object HalfTrue extends HalfTrue
// defined object HalfTrue
// warning: previously defined trait HalfTrue is not a companion to object HalfTrue.
// Companions must be defined together; you may wish to use :paste mode for this.

// The weakLogic type contains the sum of the values from bools and halfTrue
type WeakLogicType = Either[Bool, HalfTrue]
// defined type alias WeakLogicType

val weakLogicValues: Set[Either[HalfTrue, Bool]] = Set(Right(True), Right(False), Left(HalfTrue))
// weakLogicValues: Set[Either[HalfTrue,Bool]] = Set(Right(True$@c8afbd6), Right(False$@78d0f039), Left(HalfTrue$@769
```

Sum types are sometimes called union types, discriminated unions, or tagged unions.

There's a couple libraries in JS which help with defining and using union types.

Flow includes union types and TypeScript has Enums to serve the same role.

Product type

A **product** type combines types together in a way you're probably more familiar with:

```
// point :: (Number, Number) -> {x: Number, y: Number}
case class Point(x: Int, y: Int)
// defined class Point

val point = (x: Int, y: Int) => Point(x, y)
// point: (Int, Int) => Point = $$Lambda$6170/1963675584@55a1298b
```

It's called a product because the total possible values of the data structure is the product of the different values. Many languages have a tuple type which is the simplest formulation of a product type.

See also Set theory.

Option

Option is a sum type with two cases often called `Some` and `None`.

Option is useful for composing functions that might not return a value.

```
// Naive definition

trait MyOption[+A] {
  def map[B](f: A => B): MyOption[B]
  def flatMap[B](f: A => MyOption[B]): MyOption[B]
}
// defined trait MyOption

case class MySome[A](a: A) extends MyOption[A] {
  def map[B](f: A => B): MyOption[B] = MySome(f(a))
  def flatMap[B](f: A => MyOption[B]): MyOption[B] = f(a)
}
// defined class MySome

case object MyNone extends MyOption[Nothing] {
  def map[B](f: Nothing => B): MyOption[B] = this
  def flatMap[B](f: Nothing => MyOption[B]) = this
}
// defined object MyNone

// maybeProp :: (String, {a}) -> Option a
def maybeProp[A, B](key: A, obj: Map[A, B]): Option[B] = obj.get(key)
// maybeProp: [A, B](key: A, obj: Map[A,B])Option[B]
```

Use `flatMap` to sequence functions that return `Option`s

```
// getItem :: Cart -> Option CartItem
def getItem[A](cart: Map[String, Map[String, A]]): Option[Map[String, A]] = maybeProp("item", cart)
// getItem: [A](cart: Map[String,Map[String,A]])Option[Map[String,A]]

// getPrice :: Item -> Option Number
def getPrice[A](item: Map[String, A]): Option[A] = maybeProp("price", item)
// getPrice: [A](item: Map[String,A])Option[A]

// getNestedPrice :: cart -> Option a
def getNestedPrice[A](cart: Map[String, Map[String, A]]) = getItem(cart).flatMap(getPrice)
// getNestedPrice: [A](cart: Map[String,Map[String,A]])Option[A]

getNestedPrice(Map())
// res38: Option[Nothing] = None

getNestedPrice(Map("item" -> Map("foo" -> 1)))
```

```scala
// res39: Option[Int] = None

getNestedPrice(Map("item" -> Map("price" -> 9.99)))
// res40: Option[Double] = Some(9.99)
```

`Option` is also known as `Maybe`. `Some` is sometimes called `Just`. `None` is sometimes called `Nothing`.

Functional Programming Libraries in Scala

- cats
- scalaz
- shapeless
- Monocle
- Spire
- Many typelevel projects...

P.S: This repo is successful due to the wonderful contributions!