Functional Forkshop: Part 1

Boris Buliga, Valentyn Vakatsiienko

July 26, 2019

About us

Valik

Server guild manager in Kyiv. Formerly forced people to use functional programming style in the Domains (Premium) team. Now works on Tagless Infra to provide you with the best tools for your daily needs. Which are all functional, of course.

About us

Valik

Server guild manager in Kyiv. Formerly forced people to use functional programming style in the Domains (Premium) team. Now works on Tagless Infra to provide you with the best tools for your daily needs. Which are all functional, of course.

Boris

Developer at Payments by Wix team. Jumps between two extremes - Emacs Lisp and Haskell. Wants to force people around to use both languages, but can't explain why.

About the Forkshop

- Basic forkshop is split into several parts:
  1. Type classes, Semigroups and Monoids.
  2. Functors and Applicative Functors.
  3. Monads.
  4. Readers.
  5. Comonads.
- Theory and practice. Make sure that you are ready to write the code.
- Target audience is Scala developers learning FP.
- Forkshop is duplicated in Haskell.

Whys

- Functional programming roams (a bit).
  - More projects are using functional programming techniques and idioms (at different scale).
- Some people are still confused by all these functional talks (`OptionT`, type lambdas etc).
- Having a common language and understanding of some fundamental stuff is important.

Agenda

- Type classes
- Semigroups
- Monoids
- 3 interesting™ tasks

Application definition

- We are writing a game.
- With multiple different creatures.
- Everyone introduces themselves.
- Introduction consists of animations and text showing in a bubble.

Meet the hero

```scala
case class Hero(name: String, job: String, level: Int) {
  def introduce(): String = s"Hi! My name is $name. I am $level level $job."
}

object Game extends App {
  val player = Hero("Valik", "Black Mage", 20)

  someRealShitSounds()
  drawBubble(player.introduce())
  someRealShitAnimations()
}

// Hi! My name is Valik. I am 20 level Black Mage.
```

Every hero needs a monster

```scala
case class Orc(name: String, level: Int) {
  def introduce(): String =
    s"Lok-tar ogar! Me be $name. Me be strong. Level $level strong!"
}

case class Ooze(level: Int) {
  def introduce(): String = 1.to(level).map(_=>"brlup").mkString("-")
}
```

Game

```scala
object Game extends App {
  val player = Hero("Valik", "Black Mage", 20)
  val orc = Orc("Garrosh", 105)
  val ooze = Ooze(2)

  // Introduce player
  someRealShitSounds()
  drawBubble(player.introduce())
  someRealShitAnimations()

  // Introduce orc
  someRealShitSounds()
  drawBubble(orc.introduce())
  someRealShitAnimations()

  // Introduce ooze
  someRealShitSounds()
  drawBubble(ooze.introduce())
  someRealShitAnimations()
}

// Hi! My name is Valik. I am 20 level Black Mage.
// Lok-tar ogar! Me be Garrosh. Me be strong. Level 105 strong!
// brlup-brlup
```

Game

```scala
object Game extends App {
  val player = Hero("Valik", "Black Mage", 20)
  val orc = Orc("Garrosh", 105)
  val ooze = Ooze(2)

  // Introduce player
  someRealShitSounds()
  drawBubble(player.introduce())
  someRealShitAnimations()

  // Introduce orc
  someRealShitSounds()
  drawBubble(orc.introduce())
  someRealShitAnimations()

  // Introduce ooze
  someRealShitSounds()
  drawBubble(ooze.introduce())
  someRealShitAnimations()
}

// Hi! My name is Valik. I am 20 level Black Mage.
// Lok-tar ogar! Me be Garrosh. Me be strong. Level 105 strong!
// brlup-brlup
```

Issues with this code:

1. Repetition
2. Noise

Introducing abstractions

```
//



case class Hero(...) {
  def introduce(): String = s"..."
}

case class Orc(...) {
  def introduce(): String = s"..."
}

case class Ooze(...) {
  def introduce(): String = s"..."
}
```

Introducing abstractions

```scala
//



case class Hero(...) {
  def introduce(): String = s"..."
}

case class Orc(...) {
  def introduce(): String = s"..."
}

case class Ooze(...) {
  def introduce(): String = s"..."
}
```

```scala
trait Introducible {
  def introduce(): String
}

case class Hero(...) extends Introducible {
  override def introduce(): String = s"..."
}

case class Orc(...) extends Introducible {
  override def introduce(): String = s"..."
}

case class Ooze(...) extends Introducible {
  override def introduce(): String = s"..."
}
```

Game with trait

- No more `introduce(_.introduce())`.
- We are adaptive. Less code needs to be changed if we need something new in the `introduce` function (e.g. sound name) - just add new 'method' to the trait.
- Refactoring becomes easier.

```scala
def introduce(phrase: String): Unit = {
  someRealShitSounds()
  drawBubble(phrase)
  someRealShitAnimations()
}

object Game extends App {
  /* ... */

  introduce(player.introduce())
  introduce(orc.introduce())
  introduce(ooze.introduce())
}
```

```scala
def introduce(creature: Introducible): Unit = {
  someRealShitSounds()
  drawBubble(creature.introduce())
  someRealShitAnimations()
}

object Game extends App {
  /* ... */

  introduce(player)
  introduce(orc)
  introduce(ooze)
}
```

Here comes the cockatrice

```scala
import io.proprietary.monsters.cockatrice._

/* ... */

object Game extends App {
  /* ... */

  val cockatrice = Cockatrice(level = 666, element = Element.Fire)

  introduce(cockatrice) // ???
                        // ain't gonna work
}
```

Shawarma to the rescue

```scala
import io.proprietary.monsters.cockatrice._

/* ... */

case class CockatriceWrapper(cockatrice: Cockatrice) extends Introducible {
  override def introduce(): String = {
    import cockatrice._
    s"Haha. I am a ${element.shortName} cockatrice of level ${level}."
  }
}

object Game extends App {
  /* ... */

  val cockatrice = Cockatrice(level = 666, element = Element.Fire)
  val cockatriceW = CockatriceWrapper(cockatrice)

  introduce(cockatriceW)

  /* ... */
}


// Haha. I am a fire cockatrice of level 666.
```

Calm down and reevaluate our goal

- Abstraction - caring about what you can do and not what you are. E.g. separation of data and behaviour.

Calm down and reevaluate our goal

- Abstraction - caring about what you can do and not what you are. E.g. separation of data and behaviour.
- Composition - having a way to express something that can do several things at once.

Calm down and reevaluate our goal

- Abstraction - caring about what you can do and not what you are. E.g. separation of data and behaviour.
- Composition - having a way to express something that can do several things at once.
- Extensibility - extending all kind of types:
  - types we own
  - types we don't own
  - even built-in types

trait + wrapper: abstraction

Abstraction holds. Proof is the `introduce` function itself.

```
def introduce(creature: Introducible): Unit = {
  someRealShitSounds()
  drawBubble(creature.introduce())
  someRealShitAnimations()
}
```

`trait` + wrapper: composition

Composition holds thanks to `with` keyword.

trait + wrapper: composition

Composition holds thanks to `with` keyword.

```
trait CanAttack {
  def attack(): Unit
}

def patheticAttack[A <: Introducible with CanAttack](creature: A): Unit
```

trait + wrapper: composition

Composition holds thanks to `with` keyword.

```
trait CanAttack {
  def attack(): Unit
}

def patheticAttack[A <: Introducible with CanAttack](creature: A): Unit
```

`with` keyword is not commutative

`Introducible with CanAttack` != `CanAttack with Introducible`.

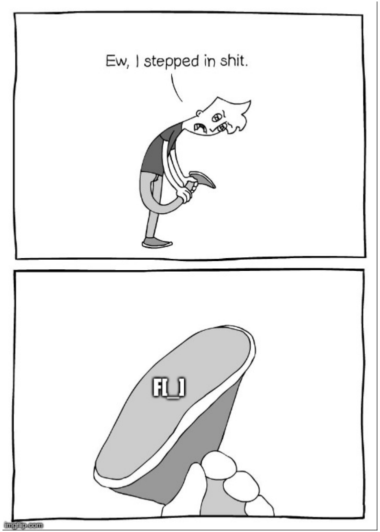
`trait` + wrapper: extensibility

Extensibility holds, but with several caveats:

1. No consistency - we wrap only types we don't own.
2. Wrappers don't compose very well. You might even wrap your wrappers.
3. Bad usability:
   3.1 You can't interchangeably use wrapper and the underlying value.
   3.2 You can't plug in different behaviour implementations.

You know where it's going to, right?

You know where it's going to, right?



Dividing data and behaviour

```scala
trait Introducible {
  def introduce(): String
}

def introduce(creatute: Introducible): Unit = {

  /* ... */
  drawBubble(creatute.introduce())
  /* ... */
}
```

Dividing data and behaviour

```scala
trait Introducible {
  def introduce(): String
}

def introduce(creatute: Introducible): Unit = {

  /* ... */
  drawBubble(creatute.introduce())
  /* ... */
}
```

```scala
trait Introducible[A] {
  def introduce(a: A): String
}

def introduce[A](creature: A,
                 impl: Introducible[A]): Unit = {
  /* ... */
  drawBubble(impl.introduce(creature))
  /* ... */
}
```

Usage

```
// Define new trait
trait Introducible[A] {
  def introduce(a: A): String
}
```

Usage

```scala
// Define new trait
trait Introducible[A] {
  def introduce(a: A): String
}


// Remove behaviour from data
case class Hero(name: String, job: String, level: Int)
```

Usage

```scala
// Define new trait
trait Introducible[A] {
  def introduce(a: A): String
}


// Remove behaviour from data
case class Hero(name: String, job: String, level: Int)


// Implement behaviour as a value in companion object
object Hero {
  val introducibleHero: Introducible[Hero] = new Introducible[Hero] {
    override def introduce(a: Hero): String =
      s"..."
  }
}
```

Usage

```scala
// Define new trait
trait Introducible[A] {
  def introduce(a: A): String
}


// Remove behaviour from data
case class Hero(name: String, job: String, level: Int)


// Implement behaviour as a value in companion object
object Hero {
  val introducibleHero: Introducible[Hero] = new Introducible[Hero] {
    override def introduce(a: Hero): String =
      s"..."
  }
}


// Pass data and behaviour separately
object Game extends App {
  /* ... */
  introduce(
    creature = hero,
    impl = Hero.introducibleHero
  )
}
```

External types? Pff. . .

```scala
import io.proprietary.monsters.cockatrice._

// Implement behaviour as a value in companion object
object CockatriceInstances {
  val introducibleCockatrice: Introducible[Cockatrice] = new Introducible[Cockatrice] {
    override def introduce(a: Cockatrice): String =
      s"..."
  }
}
```

External types? Pff. . .

```scala
import io.proprietary.monsters.cockatrice._

// Implement behaviour as a value in companion object
object CockatriceInstances {
  val introducibleCockatrice: Introducible[Cockatrice] = new Introducible[Cockatrice] {
    override def introduce(a: Cockatrice): String =
      s"..."
  }
}


// Pass data and behaviour separately
object Game extends App {
  /* ... */
  introduce(
    creature = cockatrice,
    impl = CockatriceInstances.introducibleCockatrice
  )
}
```

But passing implementation around is. . .

Cucumbersome

So implicits :(

```
object Hero {
  val introducibleHero:
      Introducible[Hero] = ???
}

object CockatriceInstances {
  val introducibleCockatrice:
      Introducible[Cockatrice] = ???
}

def introduce[A](creature: A,
                 impl: Introducible[A]): Unit = {
  /* ... */
  drawBubble(impl.introduce(creature))
  /* ... */
}

object Game extends App {
  /* ... */
  introduce(hero, introducibleHero)
  introduce(cockatrice, introducibleCockatrice)
}
```

So implicits :(

```scala
object Hero {
  val introducibleHero:
      Introducible[Hero] = ???
}

object CockatriceInstances {
  val introducibleCockatrice:
      Introducible[Cockatrice] = ???
}

def introduce[A](creature: A,
                 impl: Introducible[A]): Unit = {
  /* ... */
  drawBubble(impl.introduce(creature))
  /* ... */
}

object Game extends App {
  /* ... */
  introduce(hero, introducibleHero)
  introduce(cockatrice, introducibleCockatrice)
}
```

```scala
object Hero {
  implicit val introducibleHero:
      Introducible[Hero] = ???
}

object CockatriceInstances {
  implicit val introducibleCockatrice:
      Introducible[Cockatrice] = ???
}

def introduce[A](creature: A)
             (implicit impl: Introducible[A]): Unit = {
  /* ... */
  drawBubble(impl.introduce(creature))
  /* ... */
}

object Game extends App {
  /* ... */
  introduce(hero)
  introduce(cockatrice)
}
```

Summoning the summoner

```scala
trait Introducible[A] {
  def introduce(a: A): String
}
```

```scala
def introduce[A](creature: A)
             (implicit impl: Introducible[A]): Unit = {
  /* ... */
  drawBubble(impl.introduce(creature))
  /* ... */
}
```

Summoning the summoner

```scala
trait Introducible[A] {
  def introduce(a: A): String
}

def introduce[A](creature: A)
                (implicit impl: Introducible[A]): Unit = {
  /* ... */
  drawBubble(impl.introduce(creature))
  /* ... */
}
```

```scala
trait Introducible[A] {
  def introduce(a: A): String
}

object Introducible {
  def apply[A: Introducible]: Introducible[A] =
    implicitly[Introducible[A]]
}

def introduce[A: Introducible](creature: A): Unit = {

  /* ... */
  drawBubble(Introducible[A].introduce(creature))
  /* ... */
}
```

What have we done?

Type class is just a construct that supports ad hoc polymorphism. E.g. allows one to define polymorphic functions that can be applied to arguments of different types and behave differently based the type of the arguments.

In other words, type classes are solution for supporting function overloading.

What have we done?

Type class is just a construct that supports ad hoc polymorphism. E.g. allows one to define polymorphic functions that can be applied to arguments of different types and behave differently based the type of the arguments.

In other words, type classes are solution for supporting function overloading.

In Scala this can be achieved in several ways:

- Class inheritance or traits.
- Type classes (traits + implicits).

Type classes: abstraction

Abstraction holds. Proof is the `introduce` function itself.

Type classes: abstraction

Abstraction holds. Proof is the `introduce` function itself.

```
def introduce(creature: Introducible): Unit = {
  /* ... */
  drawBubble(creature.introduce())
  /* ... */
}
```

```
def introduce[A: Introducible](creature: A): Unit = {
  /* ... */
  drawBubble(Introducible[A].introduce(creature))
  /* ... */
}
```

Type classes: abstraction

Abstraction holds. Proof is the `introduce` function itself.

```scala
def introduce(creature: Introducible): Unit = {
  /* ... */
  drawBubble(creature.introduce())
  /* ... */
}
```

```scala
def introduce[A: Introducible](creature: A): Unit = {
  /* ... */
  drawBubble(Introducible[A].introduce(creature))
  /* ... */
}
```

We gain literal data and behaviour separation.

Type classes: composition

Composition holds. We just pass two different behaviours.

Type classes: composition

Composition holds. We just pass two different behaviours.

```
def patheticAttack[A <: Introducible with CanAttack](creature: A): Unit
```

Type classes: composition

Composition holds. We just pass two different behaviours.

```
def patheticAttack[A <: Introducible with CanAttack](creature: A): Unit
```

```
def patheticAttack[A : Introducible : CanAttack](creature: A): Unit
def patheticAttack[A](creature: A)
                  (implicit introducibleImpl: Introducible[A],
                   canAttackImpl: CanAttack[A]): Unit
```

Type classes: composition

Composition holds. We just pass two different behaviours.

```
def patheticAttack[A <: Introducible with CanAttack](creature: A): Unit
```

```
def patheticAttack[A : Introducible : CanAttack](creature: A): Unit
def patheticAttack[A](creature: A)
                  (implicit introducibleImpl: Introducible[A],
                   canAttackImpl: CanAttack[A]): Unit
```

But with type classes we don't care about the order.

Type classes: extensibility

Extensibility holds with some gains:

1. Consistency - we treat our own type the same way we treat external types.
2. Usability - no wrappers, no interchangeability problem.

Type classes: final thoughts

1. Simple idea giving us good properties.
2. Found a good use for controversial implicits feature.
3. Literal separation of data and behaviour.
4. Good for overloading.
5. + more abstraction, - more code

Time for a quiz!

What is common between:

1. Int
2. String
3. List
4. PartialFunction
5. HttpMapping

Time for a quiz!

What is common between:

1. Int
2. String
3. List
4. PartialFunction
5. HttpMapping

They can be composed!

1. Int + Int = Int
2. String + String = String
3. List ::: List = List
4. PartialFunction orElse PartialFunction = PartialFunction
5. HttpMapping + HttpMapping = HttpMapping

Associativity

One does not simply compose stuff. Composition MUST be associative

1. $Int + Int + Int = Int + (Int + Int) = (Int + Int) + Int$
2. $String + String + String = String + (String + String) = (String + String) + String$
3. etc...

Semigroup

Lets use fancy math name and fancy math notation to describe this simple concept.

Semigroup is a set S with binary closed operation $\cdot : S \times S \to S$ that satisfies associativity property:

$$\forall a, b, c \in S : (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Operation is closed when $\forall a, b \in S : a \cdot b \in S$.

But it's not that scary

```scala
package object typeclass {

  //
  // Laws:
  //   1. ∀a, b, c ∈ A : (a · b) · c = a · (b · c)
  //
  trait Semigroup[A] {
    def combine(x: A, y: A): A
  }

  object Semigroup {
    def apply[A: Semigroup]: Semigroup[A] =
      implicitly[Semigroup[A]]
  }

}
```

But it's not that scary

```scala
package object typeclass {

  //
  // Laws:
  //    1. ∀a, b, c ∈ A : (a · b) · c = a · (b · c)
  //
  trait Semigroup[A] {
    def combine(x: A, y: A): A
  }

  object Semigroup {
    def apply[A: Semigroup]: Semigroup[A] =
      implicitly[Semigroup[A]]
  }

}
```

In simple words, semigroup is a set with means of combining elements of that set.

But it's not that scary

```scala
package object typeclass {

  //
  // Laws:
  //   1. ∀a, b, c ∈ A : (a · b) · c = a · (b · c)
  //
  trait Semigroup[A] {
    def combine(x: A, y: A): A
  }

  object Semigroup {
    def apply[A: Semigroup]: Semigroup[A] =
      implicitly[Semigroup[A]]
  }

}
```

In simple words, semigroup is a set with means of combining elements of that set.

Important!

Semigroup is a pair of the set and the operation.

You can't say that string is a semigroup, you must provide an operation.

And in many cases there is more than one operation for a set to form a semigroup.

What is law?

- In programming world it's just a contract.

What is law?

- In programming world it's just a contract.
- Operations in the type classes are very generic.

```
def combine(x: A, y: A): A
```

What is law?

- In programming world it's just a contract.
- Operations in the type classes are very generic.

```
def combine(x: A, y: A): A
```

- So type classes should have some associated laws.

What is law?

- In programming world it's just a contract.
- Operations in the type classes are very generic.
```
def combine(x: A, y: A): A
```
- So type classes should have some associated laws.
- Laws describe properties of these operations and connection between operations in one type class.

What is law?

- In programming world it's just a contract.
- Operations in the type classes are very generic.

```scala
def combine(x: A, y: A): A
```

- So type classes should have some associated laws.
- Laws describe properties of these operations and connection between operations in one type class.
- Contract of the interface gives us confidence when we write generic code.

What is law?

- In programming world it's just a contract.
- Operations in the type classes are very generic.

```
def combine(x: A, y: A): A
```

- So type classes should have some associated laws.
- Laws describe properties of these operations and connection between operations in one type class.
- Contract of the interface gives us confidence when we write generic code.
- And as you will see, we really care about these laws.

Instance example

```scala
package object implicits {
  implicit val stringSemigroup: Semigroup[String] = new Semigroup[String] {
    override def combine(x: String, y: String): String = x + y
  }
}
```

Checking laws - ~~pen and paper~~ in comments

```scala
package object implicits {
  implicit val stringSemigroup: Semigroup[String] = new Semigroup[String] {
    override def combine(x: String, y: String): String = x + y
  }
}

/*
combine(a, combine(b, c))
  = combine(a, b + c)
  = a + (b + c)
  = (associativity of +)
  = (a + b) + c = combine(a + b, c)
  = combine(combine(a, b), c)
*/
```

You're programmer after all

Question on the interview: property based testing

```scala
object SemigroupSpecification extends Properties("Semigroup") with SemigroupSpecificationSupport {
 include(semigroup[String](stringSemigroup))
}

trait SemigroupSpecificationSupport {
 def semigroup[A](sg: Semigroup[A])(implicit ar: Arbitrary[A], tag: ClassTag[A]): Properties =
   new Properties(s"Semigroup[${tag.toString}]") {


     // ∀a, b, c ∈ A : (a · b) · c = a · (b · c)
     property("associativity") = forAll { (a: A, b: A, c: A) =>
       sg.combine(sg.combine(a, b), c) =? sg.combine(a, sg.combine(b, c))
     }


   }
}

/*
+ Semigroup.Semigroup[java.lang.String].associativity: OK, passed 100 tests
  .
*/
```

More than one valid instance

```scala
package object implicits {
  implicit val stringSemigroup: Semigroup[String] = new Semigroup[String] {
    override def combine(x: String, y: String): String = x
  }
}
```

More examples

- Numbers with $+$, $*$, *min*, *max*
- Booleans with conjunction, disjunction, implication etc.
- Square nonnegative matrices with multiplication.
- Lists, Strings, Maps etc. with concatenation/union
- We will see even more examples during practical part.

Contra-examples

- $\{\mathbb{N}, /\}$ is not a Semigroup, because $/$ is not associative.
- The same goes for $\{\mathbb{N}, a^b\}$.
- $\{\mathbb{N}, -\}$ is not a Semigroup, because $-$ is not a closed operation, e.g. $\exists a, b \in \mathbb{N} : a - b \notin \mathbb{N}$, for example $10 - 15 = -5 \notin \mathbb{N}$.

Coding time

1. Clone `git@github.com:d12frosted/wax.git`
2. Import it as `sbt` project.
3. Go to `scala/src/main`

Coding time

1. Clone `git@github.com:d12frosted/wax.git`
2. Import it as `sbt` project.
3. Go to `scala/src/main`
4. Task 1
   4.1 Implement missing `Semigroup` instances in `wax.typeclass.semigroup.cats.implicits`
   4.2 Run `wax.typeclass.semigroup.laws.cats.SemigroupSpec`

Coding time

1. Clone `git@github.com:d12frosted/wax.git`
2. Import it as `sbt` project.
3. Go to `scala/src/main`
4. Task 1
   - 4.1 Implement missing `Semigroup` instances in `wax.typeclass.semigroup.cats.implicits`
   - 4.2 Run `wax.typeclass.semigroup.laws.cats.SemigroupSpec`
5. Task 2
   - 5.1 Implement missing `Semigroup` instances in `wax.typeclass.semigroup.manual.implicits`
   - 5.2 Run `wax.typeclass.semigroup.laws.manual.SemigroupSpec`

Monoid

- Sometimes you want to compose n elements where $n \geq 0$.
- Semigroup works only for $n > 0$.
- We need a default element to use if $n = 0$.

Monoid

One does not simply become a default element:

- $Int + 0 = 0 + Int = Int$
- $String + "" = "" + String = String$
- etc...

Monoid

Back to fancy words.

Monoid

Back to fancy words.

A monoid is a set S with binary closed operation $\cdot : S \times S \to S$ that satisfies associativity property:

$$\forall a, b, c \in S : (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

and identity element e that satisfies

$$\forall a \in S : e \cdot a = a \cdot e = a$$

Operation is closed when $\forall a, b \in S : a \cdot b \in S$.

Monoid

Back to fancy words.

A monoid is a set S with binary closed operation $\cdot : S \times S \to S$ that satisfies associativity property:

$$\forall a, b, c \in S : (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

and identity element e that satisfies

$$\forall a \in S : e \cdot a = a \cdot e = a$$

Operation is closed when $\forall a, b \in S : a \cdot b \in S$.

In other words, monoid is just a semigroup with identity element.

Again, it's not that scary

```
package object typeclass {

  //
  // Laws:
  //    1. ∀a, b, c ∈ S : (a · b) · c = a · (b · c)
  //    2. ∀a ∈ S : e · a = a · e = a
  //
  trait Monoid[A] extends Semigroup[A] {
    def empty: A
  }

  object Monoid {
    def apply[A: Monoid]: Monoid[A] = implicitly[Monoid[A]]
  }

}
```

Examples

- $\{\mathbb{N}_0, +\}$, where 0 is the identity element.
- $\{\mathbb{N}, *\}$, where 1 is the identity element.
- Boolean with XOR, XNOR, OR, AND.
- String with concatenation (empty string is identity element).

Examples

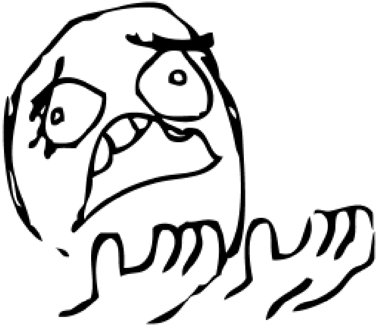
- $\{\mathbb{N}_0, +\}$, where 0 is the identity element.
- $\{\mathbb{N}, *\}$, where 1 is the identity element.
- Boolean with XOR, XNOR, OR, AND.
- String with concatenation (empty string is identity element).

But not every Semigroup forms a Monoid (we are not talking about free monoids here):

- `BigNumber` practically doesn't have identity element for `min`.

The most important question

The most important question



Why did we learn this?

The Fibonacci numbers

The Fibonacci numbers

On the interview we ask people to write a function that returns the nth Fibonacci number.

The Fibonacci numbers

On the interview we ask people to write a function that returns the nth Fibonacci number.

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_n &= F_{n-1} + F_{n-2}, \forall n > 1
\end{aligned}
$$

Solution

What we expect

```
def fib(n: Int): Int = {
  def fibTail(n: Int, a: Int, b: Int): Int = n match {
    case 0 => a
    case _ => fibTail(n - 1, b, a + b)
  }

  fibTail(n, 0, 1)
}
```

Solution

What we expect

```scala
def fib(n: Int): Int = {
  def fibTail(n: Int, a: Int, b: Int): Int = n match {
    case 0 => a
    case _ => fibTail(n - 1, b, a + b)
  }

  fibTail(n, 0, 1)
}
```

Ideal solution

$$F_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}} = \frac{\phi^n - (-\phi)^{-n}}{2\phi - 1}$$

$$\phi = \frac{1 + \sqrt{5}}{2}$$

Solution

What we expect

```
def fib(n: Int): Int = {
  def fibTail(n: Int, a: Int, b: Int): Int = n match {
    case 0 => a
    case _ => fibTail(n - 1, b, a + b)
  }

  fibTail(n, 0, 1)
}
```

Ideal solution

$$F_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}} = \frac{\phi^n - (-\phi)^{-n}}{2\phi - 1}$$

$$\phi = \frac{1 + \sqrt{5}}{2}$$

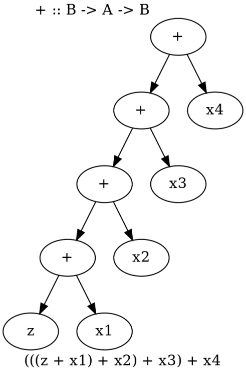
As they say, truth is somewhere in the logarithm.

Two folds

- `def foldl[A, B](xs: Seq[A])(z: B)(op: B => A => B): B`
  - Folds the structure from left to right

Two folds

- `def foldl[A, B](xs: Seq[A])(z: B)(op: B => A => B): B`
  - Folds the structure from left to right

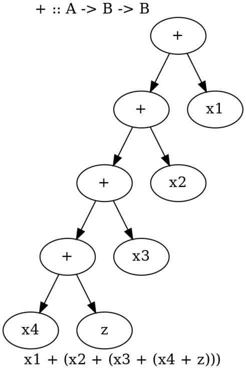


Two folds

- `def foldl[A, B](xs: Seq[A])(z: B)(op: B => A => B): B`
- `def foldr[A, B](xs: Seq[A])(z: B)(op: A => B => B): B`
    - Folds the structure from right to left

Two folds

- `def foldl[A, B](xs: Seq[A])(z: B)(op: B => A => B): B`
- `def foldr[A, B](xs: Seq[A])(z: B)(op: A => B => B): B`
  - Folds the structure from right to left

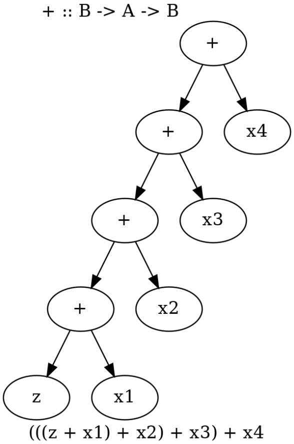


Two folds

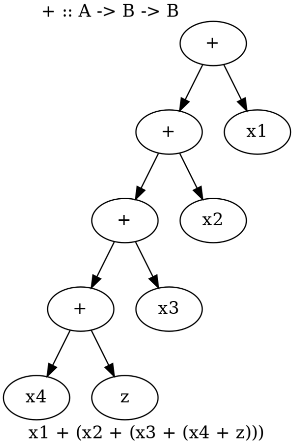
- `def foldl[A, B](xs: Seq[A])(z: B)(op: B => A => B): B`
- `def foldr[A, B](xs: Seq[A])(z: B)(op: A => B => B): B`
- Since combining function is asymmetrical in its types:
  - It's impossible to place parentheses in the arbitrary fashion or even just change the direction of the `fold`
  - It's impossible to implement a total `fold` without default value of type `B`

Two folds

foldl



foldr



What Monoid gives us

- Combining function is symmetrical (`combine : A -> A -> A`).

What Monoid gives us

- Combining function is symmetrical (`combine : A -> A -> A`).
- Identity element of type `A` (`empty`).

What Monoid gives us

- Combining function is symmetrical (`combine : A -> A -> A`).
- Identity element of type `A` (`empty`).
- So we can define a special `fold`
  - `def foldMonoid[A: Monoid](xs: Seq[A]): A`

What Monoid gives us

- Combining function is symmetrical (`combine : A -> A -> A`).
- Identity element of type `A` (`empty`).
- So we can define a special `fold`
  - `def foldMonoid[A: Monoid](xs: Seq[A]): A`
- Associativity law says that we can put parentheses in an arbitrary fashion.

What Monoid gives us

- Combining function is symmetrical (`combine : A -> A -> A`).
- Identity element of type `A` (`empty`).
- So we can define a special `fold`
  - `def foldMonoid[A: Monoid](xs: Seq[A]): A`
- Associativity law says that we can put parentheses in an arbitrary fashion.
- Identity law says that we can place identity element anywhere.

Power in terms of Monoid

In some cases all elements of the list are the same.

Power in terms of Monoid

In some cases all elements of the list are the same.

$$a + (a + (a + \ldots + a)\ldots) = a^n$$

Power in terms of Monoid

In some cases all elements of the list are the same.

$$a + (a + (a + \ldots + a) \ldots) = a^n$$

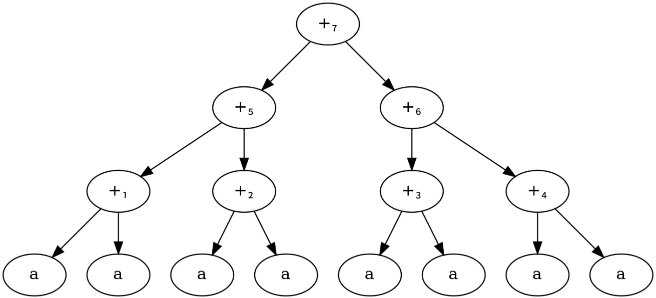
Since we can reorder the parentheses, we can arrange them like this.

Power in terms of Monoid

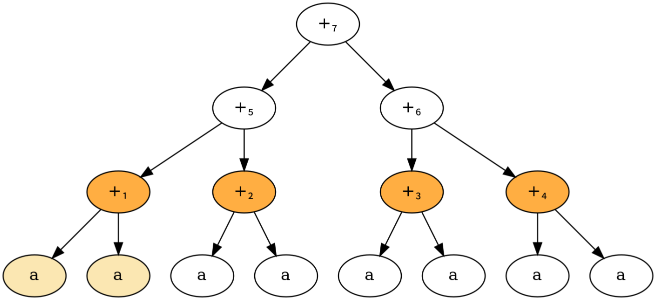
In some cases all elements of the list are the same.

$$a + (a + (a + \ldots + a) \ldots) = a^n$$

Since we can reorder the parentheses, we can arrange them like this.

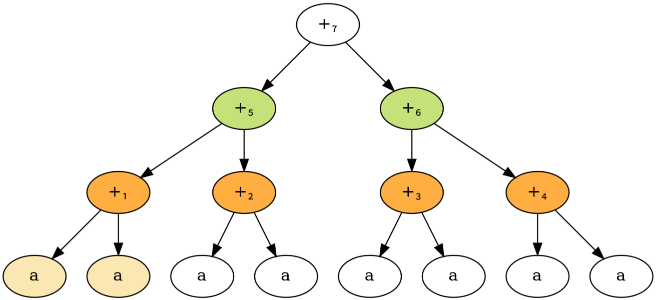


Power in terms of Monoid



Evaluating $a + a$ always yields the same result. So there is no point in repeating this calculation 4 times.

Power in terms of Monoid



The same thing with the upper level. In this particular example, we can avoid 4 operations out of 7. In general, this optimisation leads to the result in $\log n$ operations.

Power in terms of Monoid

All this means that we can define a function exp:

```
def exp[A: Monoid](a: A, n: Int): A = {
  ???
}
```

Back to Fibonacci

Fibonacci number can be defined in a different way.

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

Back to Fibonacci

Fibonacci number can be defined in a different way.

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

$$\begin{pmatrix} F_4 & F_3 \\ F_3 & F_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^3 = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 3 & 2 \\ 2 & 1 \end{pmatrix}$$

Back to Fibonacci

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

Back to Fibonacci

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

- The Fibonacci number can be calculated using square nonnegative matrix multiplication.
- Square nonnegative matrices form Monoid with multiplication.
- So we can put parentheses in a way we like it.

Coding time

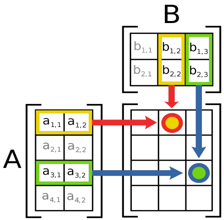
- Open `wax.exercise.fibonacci.Main` object.
  - `Main` runs two implementations and profiles them.
  - `Fib` contains implementation of tailrec and matrix approaches.
  - `ExpUtils` implements generic `exp` function.
- Task is to implement monoid for `Matrix2x2` in the `Fib` object.
- Run `MatrixSpec` to test your instance.
- Run `FibSpec` to test implementation of `Fib`.
- Run `Main` to see performance differences by yourself.

Coding time

- Open `wax.exercise.fibonacci.Main` object.
  - `Main` runs two implementations and profiles them.
  - `Fib` contains implementation of tailrec and matrix approaches.
  - `ExpUtils` implements generic exp function.
- Task is to implement monoid for `Matrix2x2` in the `Fib` object.
- Run `MatrixSpec` to test your instance.
- Run `FibSpec` to test implementation of `Fib`.
- Run `Main` to see performance differences by yourself.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} =$$

$$\begin{pmatrix} a_{11} \cdot b_{11} + a_{12} \cdot b_{21} & a_{11} \cdot b_{12} + a_{12} \cdot b_{22} \\ a_{21} \cdot b_{11} + a_{22} \cdot b_{21} & a_{21} \cdot b_{12} + a_{22} \cdot b_{22} \end{pmatrix}$$



Profiling results

N	Matrix	Tailrec
10	60	0
100	0	0
1000	1	1
10000	5	6
100000	46	168
1000000	888	15211
10000000	11266	-

Outcome

- Just think about it.

Outcome

- Just think about it.
- Giving any monoid we have a helper function that efficiently calculates a^n.

Outcome

- Just think about it.
- Giving any monoid we have a helper function that efficiently calculates a^n.
- This is only possible because of the laws that come with operations.

Outcome

- Just think about it.
- Giving any monoid we have a helper function that efficiently calculates a^n.
- This is only possible because of the laws that come with operations.
- `combine` by itself is not interesting, it's too generic.

Outcome

- Just think about it.
- Giving any monoid we have a helper function that efficiently calculates a^n.
- This is only possible because of the laws that come with operations.
- `combine` by itself is not interesting, it's too generic.
- Laws give us properties. Which we use to get a solution that works for everything that is `Monoid`.

Outcome

- Just think about it.
- Giving any monoid we have a helper function that efficiently calculates a^n.
- This is only possible because of the laws that come with operations.
- `combine` by itself is not interesting, it's too generic.
- Laws give us properties. Which we use to get a solution that works for everything that is `Monoid`.
- Monoids are everywhere around us. We deal with them every day, without even noticing it. Did you expect us to solve Fibonacci using `Monoid`?

Outcome

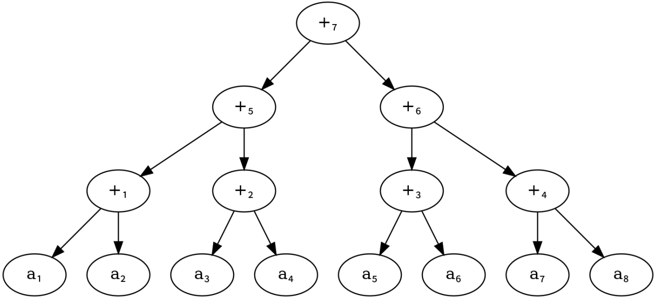
- Just think about it.
- Giving any monoid we have a helper function that efficiently calculates a^n.
- This is only possible because of the laws that come with operations.
- `combine` by itself is not interesting, it's too generic.
- Laws give us properties. Which we use to get a solution that works for everything that is `Monoid`.
- Monoids are everywhere around us. We deal with them every day, without even noticing it. Did you expect us to solve Fibonacci using `Monoid`?
- You forgot how matrix multiplication works, but now you remember, right?

Folds with Monoids

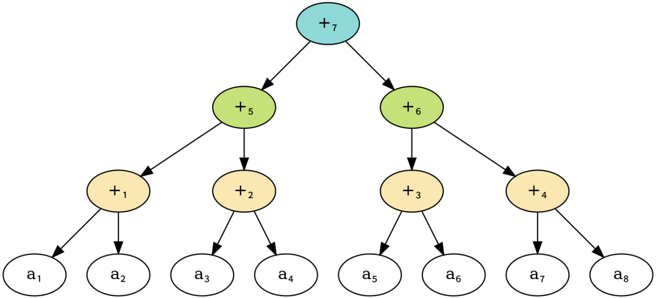
- We already know that Monoids give us an ability to place parentheses in any fashion.
- We already saw that when it comes to folding the list of the same elements we gain performance.
- But what if the elements are not equal? Do we gain anything?

Folds with Monoids

- We already know that Monoids give us an ability to place parentheses in any fashion.
- We already saw that when it comes to folding the list of the same elements we gain performance.
- But what if the elements are not equal? Do we gain anything?



Folds with Monoids



Every expression on each level does not depend on other expressions from the same level, which means that we can evaluate them in parallel.

MapReduce

- Sometimes we have a collection of elements that don't form Monoid.

MapReduce

- Sometimes we have a collection of elements that don't form Monoid.
- But we can transform (e.g. `map`) them into something that is a Monoid

MapReduce

- Sometimes we have a collection of elements that don't form Monoid.
- But we can transform (e.g. `map`) them into something that is a Monoid
- There is a strange accent, where people pronounce 'fold' as 'reduce'.

MapReduce

- Sometimes we have a collection of elements that don't form Monoid.
- But we can transform (e.g. `map`) them into something that is a Monoid
- There is a strange accent, where people pronounce 'fold' as 'reduce'.
- This is how we get the `mapReduce`.

MapReduce

- Sometimes we have a collection of elements that don't form Monoid.
- But we can transform (e.g. `map`) them into something that is a Monoid
- There is a strange accent, where people pronounce 'fold' as 'reduce'.
- This is how we get the `mapReduce`.
- Just think about it, `mapReduce` is possible thanks to `Monoid` and its *laws*.

Coding time

- Our goal is to find 10 top used words among multiple books.

Coding time

- Our goal is to find 10 top used words among multiple books.
- Open `wax.exercise.mapreduce.MapReduce` file.

Coding time

- Our goal is to find 10 top used words among multiple books.
- Open `wax.exercise.mapreduce.MapReduce` file.
- `MapReduce` object implements `mapReduce` function (two variants - par and seq).

Coding time

- Our goal is to find 10 top used words among multiple books.
- Open `wax.exercise.mapreduce.MapReduce` file.
- `MapReduce` object implements `mapReduce` function (two variants - par and seq).
- `Main` object runs (not yet defined) `job` and profiles it (par vs seq).

Coding time

- Our goal is to find 10 top used words among multiple books.
- Open `wax.exercise.mapreduce.MapReduce` file.
- `MapReduce` object implements `mapReduce` function (two variants - par and seq).
- `Main` object runs (not yet defined) `job` and profiles it (par vs seq).
- `Result[Int]` is a map with words and their usage counter.

Coding time

- Our goal is to find 10 top used words among multiple books.
- Open `wax.exercise.mapreduce.MapReduce` file.
- `MapReduce` object implements `mapReduce` function (two variants - par and seq).
- `Main` object runs (not yet defined) `job` and profiles it (par vs seq).
- `Result[Int]` is a map with words and their usage counter.
- Your goal is to:
  1. Implement monoid instance for `MapReduce.Result[A]`.
  2. Implement the `job` function to find the most used word.

Coding time

- Our goal is to find 10 top used words among multiple books.
- Open `wax.exercise.mapreduce.MapReduce` file.
- `MapReduce` object implements `mapReduce` function (two variants - par and seq).
- `Main` object runs (not yet defined) `job` and profiles it (par vs seq).
- `Result[Int]` is a map with words and their usage counter.
- Your goal is to:
  1. Implement monoid instance for `MapReduce.Result[A]`.
  2. Implement the `job` function to find the most used word.
- Use helpers from `FileUtils`:
  - `readTokens` to get the list of words from the file.
  - `authorBooks` to get the list of books (files) by author (e.g. `authorBooks("boris")`).
  - `allBooks` to get the list of all book among all available authors.

Benchmarks

```
Par
duration = 65633 ms
result   = List(..., (people,37798), ...)

Seq
duration = 396530 ms
result   = List(..., (people,37798), ...)
```

Outcome

- Thanks to *associative* and *identity* laws it's possible to implement a parallel fold.

Outcome

- Thanks to *associative* and *identity* laws it's possible to implement a parallel fold.
- This is what makes `mapReduce` possible to exist.

Outcome

- Thanks to *associative* and *identity* laws it's possible to implement a parallel fold.
- This is what makes `mapReduce` possible to exist.
- Many applications: inverted index, document clustering, machine learning.

Outcome

- Thanks to *associative* and *identity* laws it's possible to implement a parallel fold.
- This is what makes `mapReduce` possible to exist.
- Many applications: inverted index, document clustering, machine learning.
- Google used it to regenerate index of World Wide Web.

Homework

`mapReduce` is really interesting!

Homework

`mapReduce` is really interesting!

So play with it after the forkshop.

Many monoids

We dealt with some trivial monoids here:

- Integers with addition.
- Strings and lists with concatenation.
- Matrix with multiplication.
- Maps of monoid values with merging.

Many monoids

We dealt with some trivial monoids here:

- Integers with addition.
- Strings and lists with concatenation.
- Matrix with multiplication.
- Maps of monoid values with merging.

They say that functional programming is all about *functions*.

Many monoids

We dealt with some trivial monoids here:

- Integers with addition.
- Strings and lists with concatenation.
- Matrix with multiplication.
- Maps of monoid values with merging.

They say that functional programming is all about *functions*.

Can function be monoid?

Let's start with some wrappers (pun intended)

- Suppose that we have some case class `Wrapper[A](value: A)`

Let's start with some wrappers (pun intended)

- Suppose that we have some case class `Wrapper[A](value: A)`
- Can it be a monoid?

Let's start with some wrappers (pun intended)

- Suppose that we have some case class `Wrapper[A](value: A)`
- Can it be a monoid?
- Well, generally speaking, not! Because we know nothing about the type `A`.

Let's start with some wrappers (pun intended)

- Suppose that we have some case class `Wrapper[A](value: A)`
- Can it be a monoid?
- Well, generally speaking, not! Because we know nothing about the type `A`.
- But what if `A` is a monoid?

Let's start with some wrappers (pun intended)

- Suppose that we have some case class `Wrapper[A](value: A)`
- Can it be a monoid?
- Well, generally speaking, not! Because we know nothing about the type `A`.
- But what if `A` is a monoid?
- Hell, yeah!

```
case class Wrapper[A](value: A)

object Wrapper {
  implicit def wrapperMonoid[A: Monoid]: Monoid[Wrapper[A]] = new Monoid[Wrapper[A]] {
    override def empty: Wrapper[A] = Wrapper(Monoid[A].empty)

    override def combine(x: Wrapper[A], y: Wrapper[A]): Wrapper[A] =
      Wrapper(Monoid[A].combine(x.value, y.value))
  }
}
```

Wrappers of monoids are monoids

- Since `IO` is a wrapper (in some sense), it `IO` can also be a monoid.

```
def ioMonoid[A: Monoid]: Monoid[IO[A]] = ???
```

Wrappers of monoids are monoids

- Since `IO` is a wrapper (in some sense), it `IO` can also be a monoid.
  ```
  def ioMonoid[A: Monoid]: Monoid[IO[A]] = ???
  ```
- Which means that we can combine IO actions (in some new sense).

Wrappers of monoids are monoids

- Since `IO` is a wrapper (in some sense), it `IO` can also be a monoid.
```
def ioMonoid[A: Monoid]: Monoid[IO[A]] = ???
```
- Which means that we can combine IO actions (in some new sense).
- Functions are wrappers (in some sense), so they also can be monoids
```
def functionMonoid[A, B: Monoid]: Monoid[Function[A, B]] = ???
```

Wrappers of monoids are monoids

- Since `IO` is a wrapper (in some sense), it `IO` can also be a monoid.

  ```
  def ioMonoid[A: Monoid]: Monoid[IO[A]] = ???
  ```

- Which means that we can combine IO actions (in some new sense).
- Functions are wrappers (in some sense), so they also can be monoids

  ```
  def functionMonoid[A, B: Monoid]: Monoid[Function[A, B]] = ???
  ```

- Which means that we can combine functions (in some new sense).

Logger

- What is logger?

Logger

- What is logger?
- Logger is basically a function from `String` to `IO[Unit]`.

```
type Logger = String => IO[Unit]
```

Logger

- What is logger?
- Logger is basically a function from String to IO[Unit].

```
type Logger = String => IO[Unit]
```

- Unit forms a monoid.

Logger

- What is logger?
- `Logger` is basically a function from `String` to `IO[Unit]`.

```
type Logger = String => IO[Unit]
```

- `Unit` forms a monoid.
- So `IO[Unit]` forms a monoid.

Logger

- What is logger?
- `Logger` is basically a function from `String` to `IO[Unit]`.
```
type Logger = String => IO[Unit]
```
- `Unit` forms a monoid.
- So `IO[Unit]` forms a monoid.
- So `String => IO[Unit]` forms a monoid.

Logger

- What is logger?
- Logger is basically a function from `String` to `IO[Unit]`.

```scala
type Logger = String => IO[Unit]
```

- `Unit` forms a monoid.
- So `IO[Unit]` forms a monoid.
- So `String => IO[Unit]` forms a monoid.
- So `Logger` forms a monoid.

Logger

- What is logger?
- Logger is basically a function from `String` to `IO[Unit]`.

```
type Logger = String => IO[Unit]
```

- `Unit` forms a monoid.
- So `IO[Unit]` forms a monoid.
- So `String => IO[Unit]` forms a monoid.
- So `Logger` forms a monoid.
- So we can combine loggers
  - `fileLogger |+| consoleLogger` - logs both into file and to console

Logger

```
def consoleLogger: IO[Logger] = IO { input =>
    IO {
      print(input)
    }
  }
```

Logger

```scala
def consoleLogger: IO[Logger] = IO { input =>
    IO {
      print(input)
    }
  }


def fileLogger(filePath: String): IO[Logger] = IO {
  val stream = new FileOutputStream(filePath)
  input => IO(stream.write(input.getBytes))
}
```

Logger

```scala
def consoleLogger: IO[Logger] = IO { input =>
    IO {
      print(input)
    }
  }

def fileLogger(filePath: String): IO[Logger] = IO {
  val stream = new FileOutputStream(filePath)
  input => IO(stream.write(input.getBytes))
}

val program: IO[Unit] = for {
  logger <- consoleLogger |+| fileLogger("logging.log")
  _      <- logger("I am the log")
} yield ()
```

Logger

- Open `wax.exercise.logging.Logging` object.

Logger

- Open `wax.exercise.logging.Logging` object.
- Fill missing implementations.

Logger

- Open `wax.exercise.logging.Logging` object.
- Fill missing implementations.
- Make sure to run `LoggingSpec` and make it green.

Logger

- Open `wax.exercise.logging.Logging` object.
- Fill missing implementations.
- Make sure to run `LoggingSpec` and make it green.
- Run `Main` to see it in action.

Logger

- Open `wax.exercise.logging.Logging` object.
- Fill missing implementations.
- Make sure to run `LoggingSpec` and make it green.
- Run `Main` to see it in action.
- Check `logging.log` file in the root of the project.

Logger

- Open `wax.exercise.logging.Logging` object.
- Fill missing implementations.
- Make sure to run `LoggingSpec` and make it green.
- Run `Main` to see it in action.
- Check `logging.log` file in the root of the project.
- Have fun!

Bonus questions

- Is it possible to define several different semigroups for function?

Bonus questions

- Is it possible to define several different semigroups for function?
- What about monoids?

Bonus questions

- Is it possible to define several different semigroups for function?
- What about monoids?
- What about `Unit`?

Outcome

- If you have a monoid, it's easy to form a new monoid (of a higher rank).

Outcome

- If you have a monoid, it's easy to form a new monoid (of a higher rank).
- Function can also be monoid. This is really cool by itself.

Outcome

- If you have a monoid, it's easy to form a new monoid (of a higher rank).
- Function can also be monoid. This is really cool by itself.
- Some of you probably gonna write new `colog` lib (but for Scala).

Recap (recup?)

- Semigroup is something with means of combining these somethings.

Recap (recup?)

- Semigroup is something with means of combining these somethings.
- Monoid is semigroup that also has neutral element that doesn't affect a combination.

Recap (recup?)

- Semigroup is something with means of combining these somethings.
- Monoid is semigroup that also has neutral element that doesn't affect a combination.
- Laws are really important.

Recap (recup?)

- Semigroup is something with means of combining these somethings.
- Monoid is semigroup that also has neutral element that doesn't affect a combination.
- Laws are really important.
- Associativity is a powerful property giving us an ability to solve some tasks.
  - a^n in $\log n$
  - `mapReduce`

Recap (recup?)

- Semigroup is something with means of combining these somethings.
- Monoid is semigroup that also has neutral element that doesn't affect a combination.
- Laws are really important.
- Associativity is a powerful property giving us an ability to solve some tasks.
  - a^n in $\log n$
  - `mapReduce`
- Monoids are everywhere. They act like a plague, once something forms a monoid, something else also begins to form a monoid.

Recap (recup?)

- Semigroup is something with means of combining these somethings.
- Monoid is semigroup that also has neutral element that doesn't affect a combination.
- Laws are really important.
- Associativity is a powerful property giving us an ability to solve some tasks.
  - a^n in $\log n$
  - `mapReduce`
- Monoids are everywhere. They act like a plague, once something forms a monoid, something else also begins to form a monoid.
- We want some rest after a long session of forkshop.

Questions?

$$\epsilon\rho\omega\tau\eta\sigma\eta?$$

Thank you very much!

We hope you enjoyed this session.