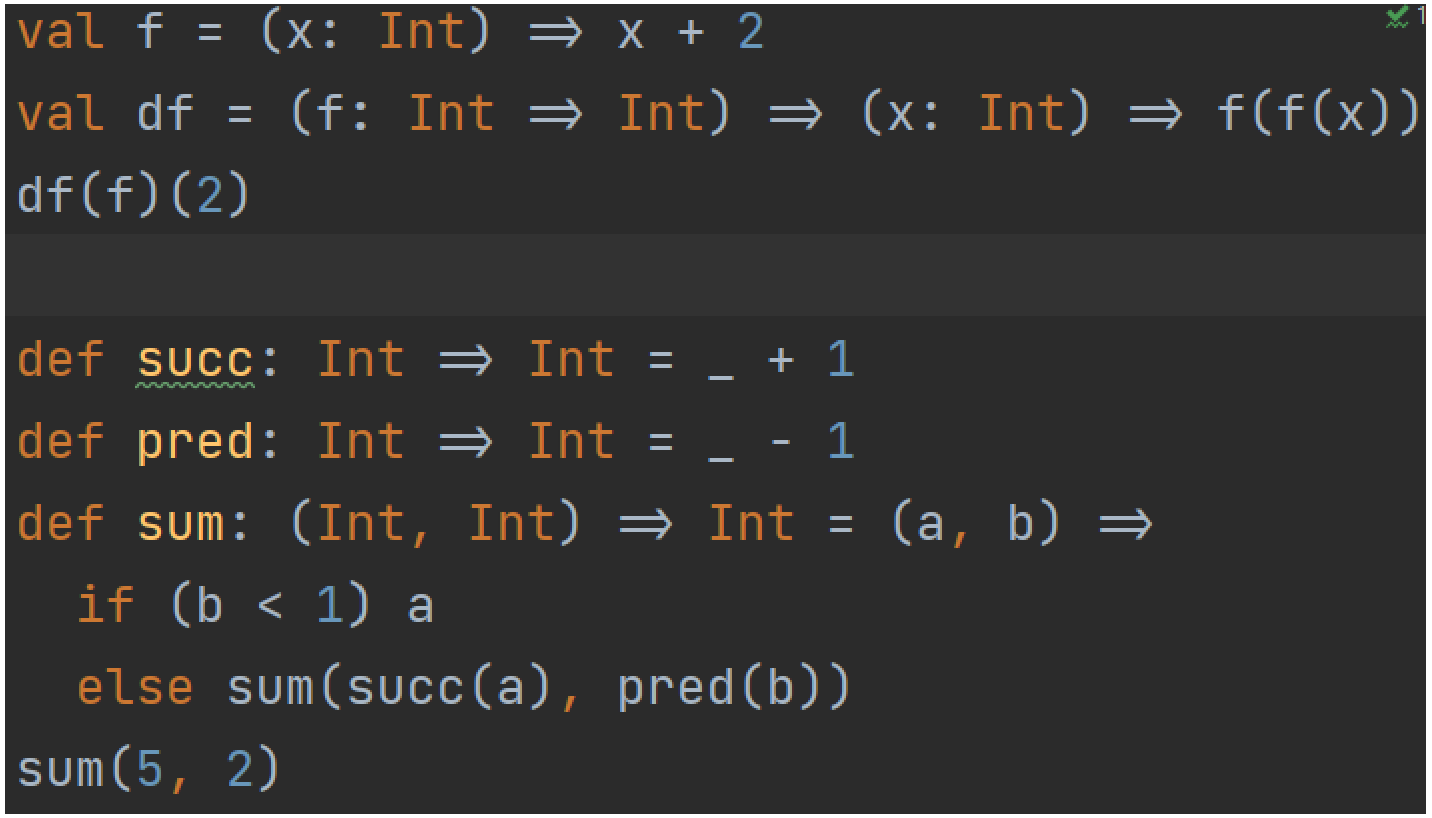
Scala FP, Monoid, Functor, Applicative, Monads

Седьмая лекция

Functional Programming

- First-class and higher-order functions
- Pure functions
- Recursion
- Strict versus non-strict evaluation
- Referential transparency
- Data structures

First-class and higher-order functions



```scala
val f = (x: Int) ⇒ x + 2
val df = (f: Int ⇒ Int) ⇒ (x: Int) ⇒ f(f(x))
df(f)(2)


def succ: Int ⇒ Int = _ + 1
def pred: Int ⇒ Int = _ - 1
def sum: (Int, Int) ⇒ Int = (a, b) ⇒
  if (b < 1) a
  else sum(succ(a), pred(b))
sum(5, 2)
```

Pure functions

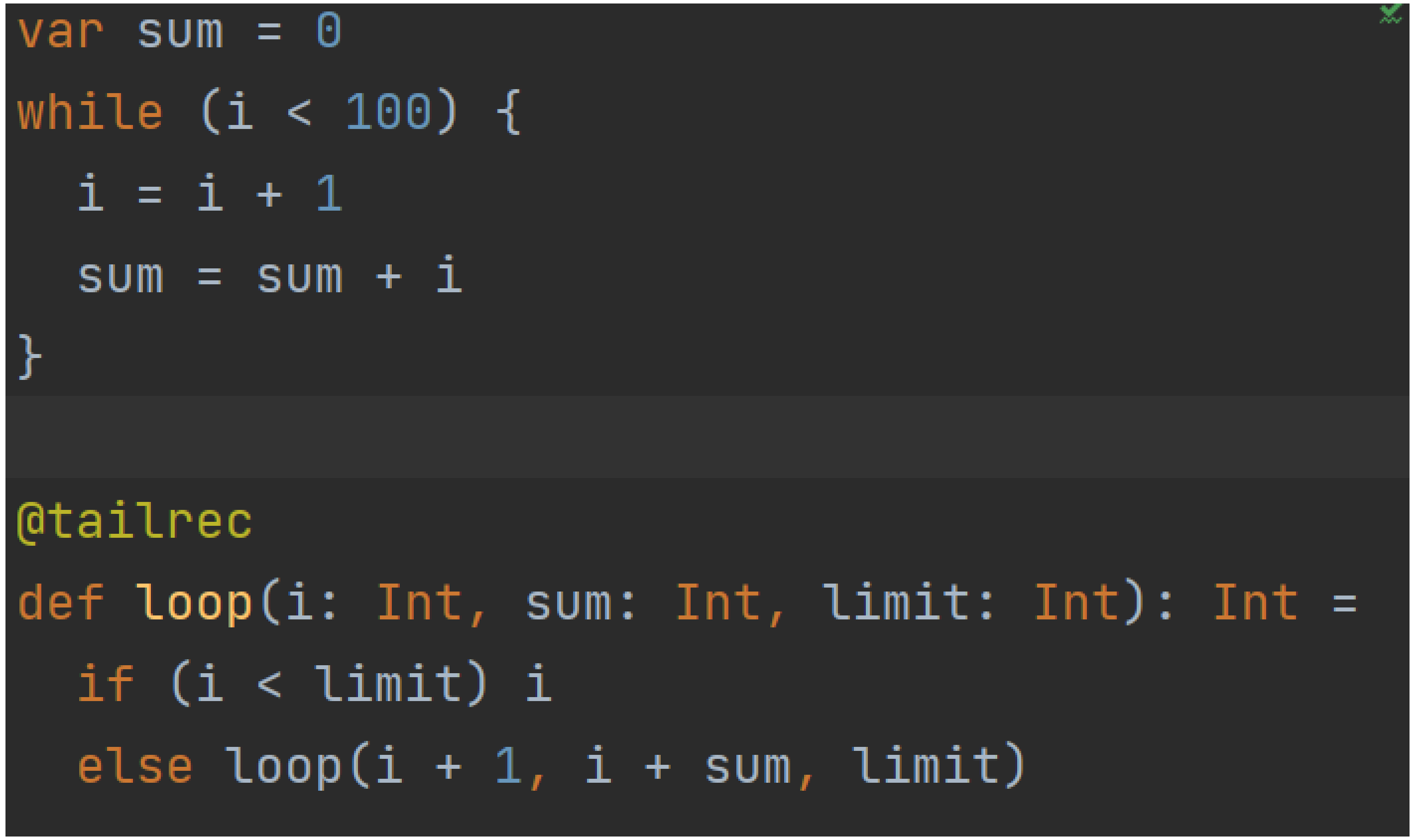
- If the result of a pure expression is not used, it can be removed without affecting other expressions
- If a pure function is called with arguments that cause no side-effects, the result is constant with respect to that argument list
- If there is no data dependency between two pure expressions, their order can be reversed
- If the entire language does not allow side-effects, then any evaluation strategy can be used

Pure functions vs Impure

- abs
- max
- min
- isEmpty
- substring

- foreach
- getDayOfWeek

Recursion

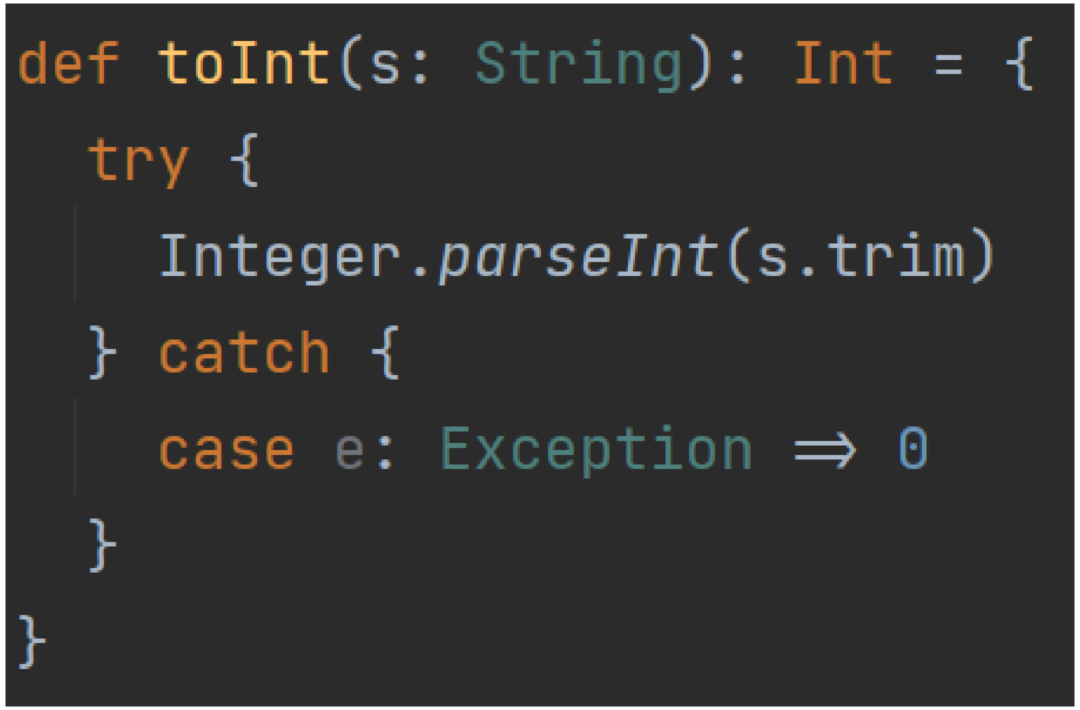


```scala
var sum = 0
while (i < 100) {
  i = i + 1
  sum = sum + i
}


@tailrec
def loop(i: Int, sum: Int, limit: Int): Int =
  if (i < limit) i
  else loop(i + 1, i + sum, limit)
```

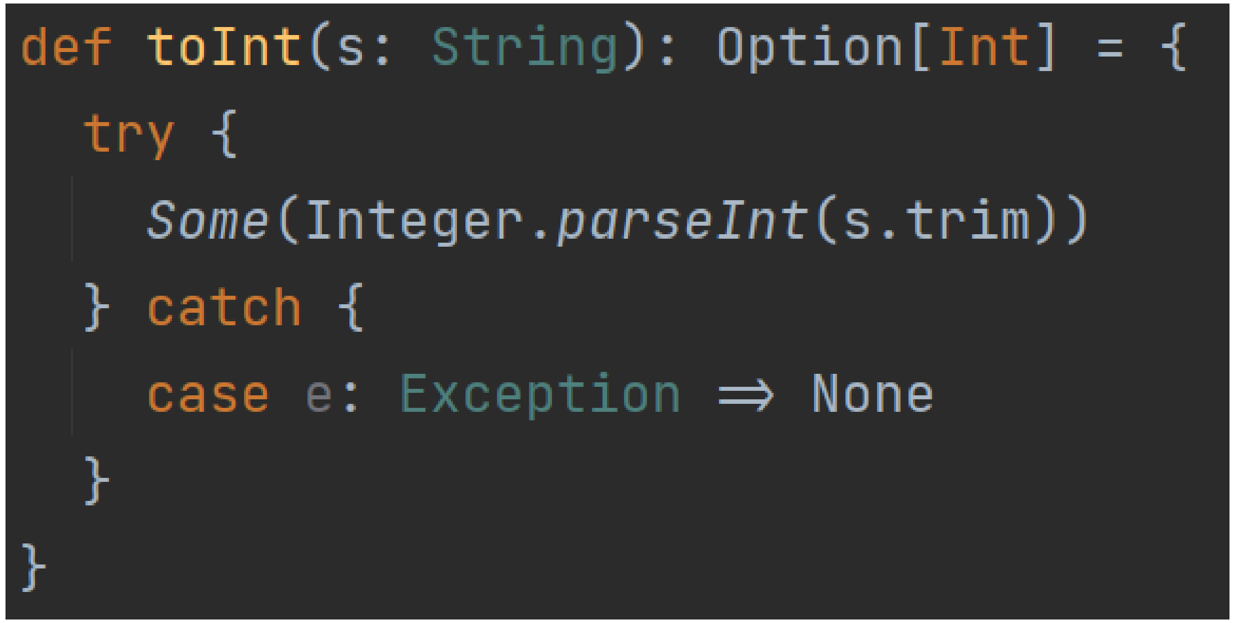
Others

- Strict versus non-strict evaluation
  - println(List(2 + 1, 3 * 2, 1 / 0, 5 - 4).length)
- Referential transparency
  - No assignment statements
- Data structures
  - Persistent, logarithmic times

No null values

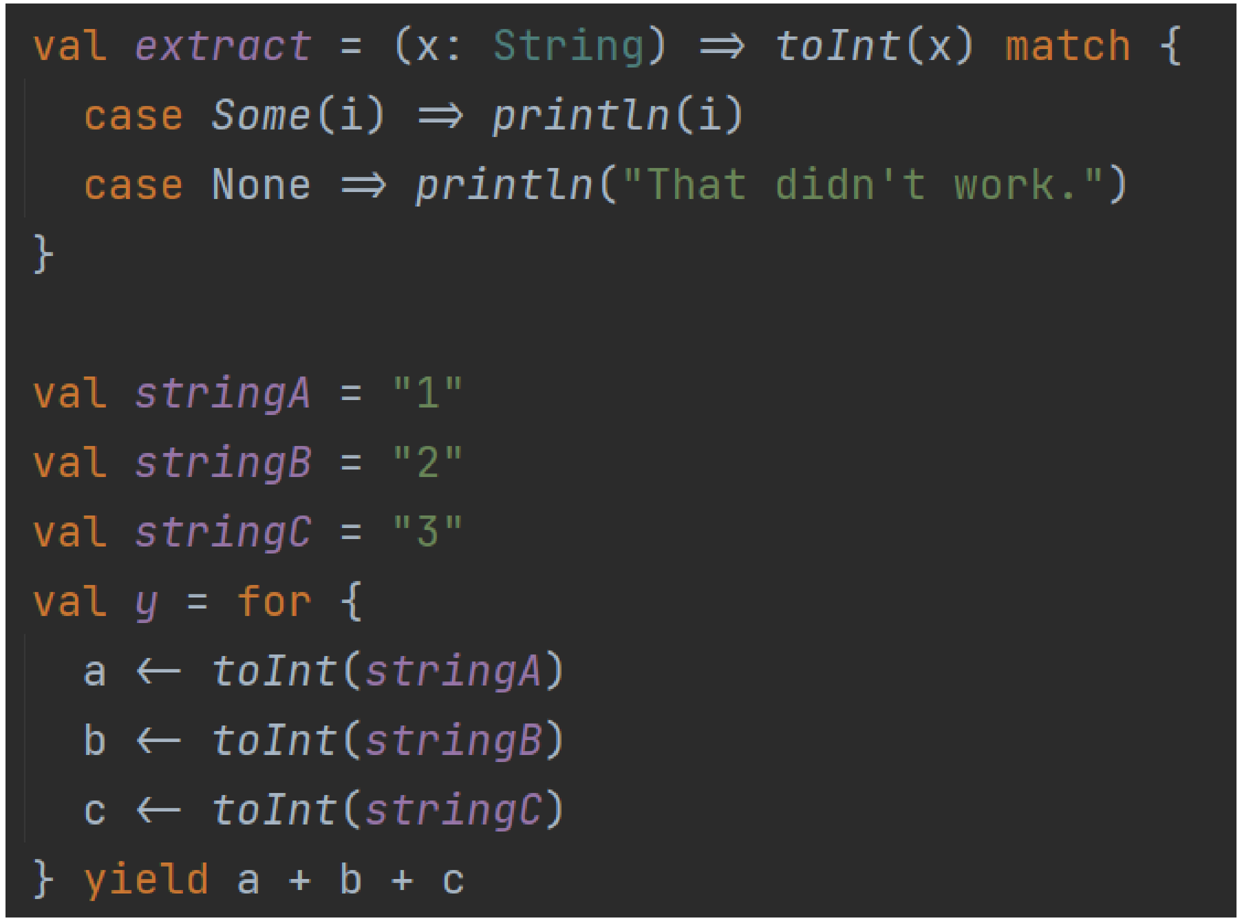


```scala
def toInt(s: String): Int = {
  try {
    Integer.parseInt(s.trim)
  } catch {
    case e: Exception => 0
  }
}
```



```scala
def toInt(s: String): Option[Int] = {
  try {
    Some(Integer.parseInt(s.trim))
  } catch {
    case e: Exception => None
  }
}
```

Consumers for toInt



```
val extract = (x: String) ⇒ toInt(x) match {
  case Some(i) ⇒ println(i)
  case None ⇒ println("That didn't work.")
}

val stringA = "1"
val stringB = "2"
val stringC = "3"
val y = for {
  a ← toInt(stringA)
  b ← toInt(stringB)
  c ← toInt(stringC)
} yield a + b + c
```

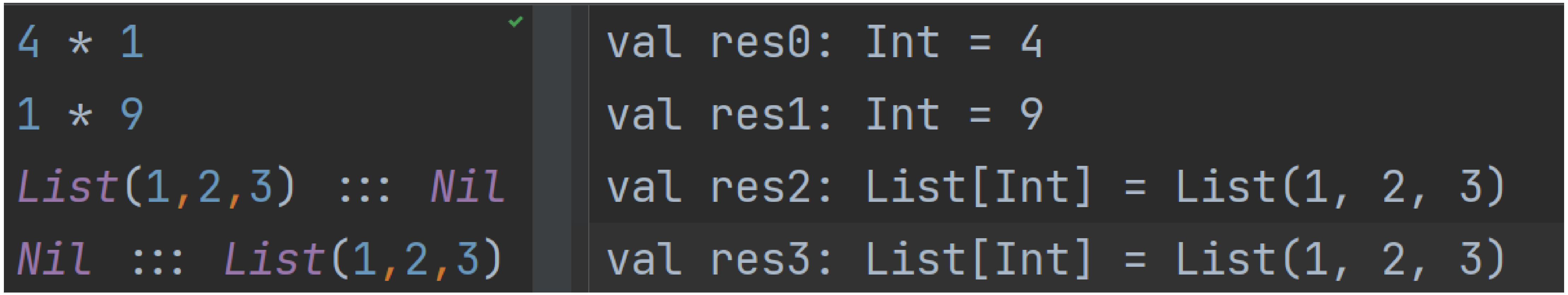
Also prohibited

- Throwing exception
- Using return, break etc.

FP – pros and cons

- Pros
  - Easy to understand, predictable code
  - Easy debugging and testing
  - Lazy evaluation
  - Is not based on order of evaluation
  - Thread safe
- Cons
  - Hard to pickup
  - Worse performance in some cases

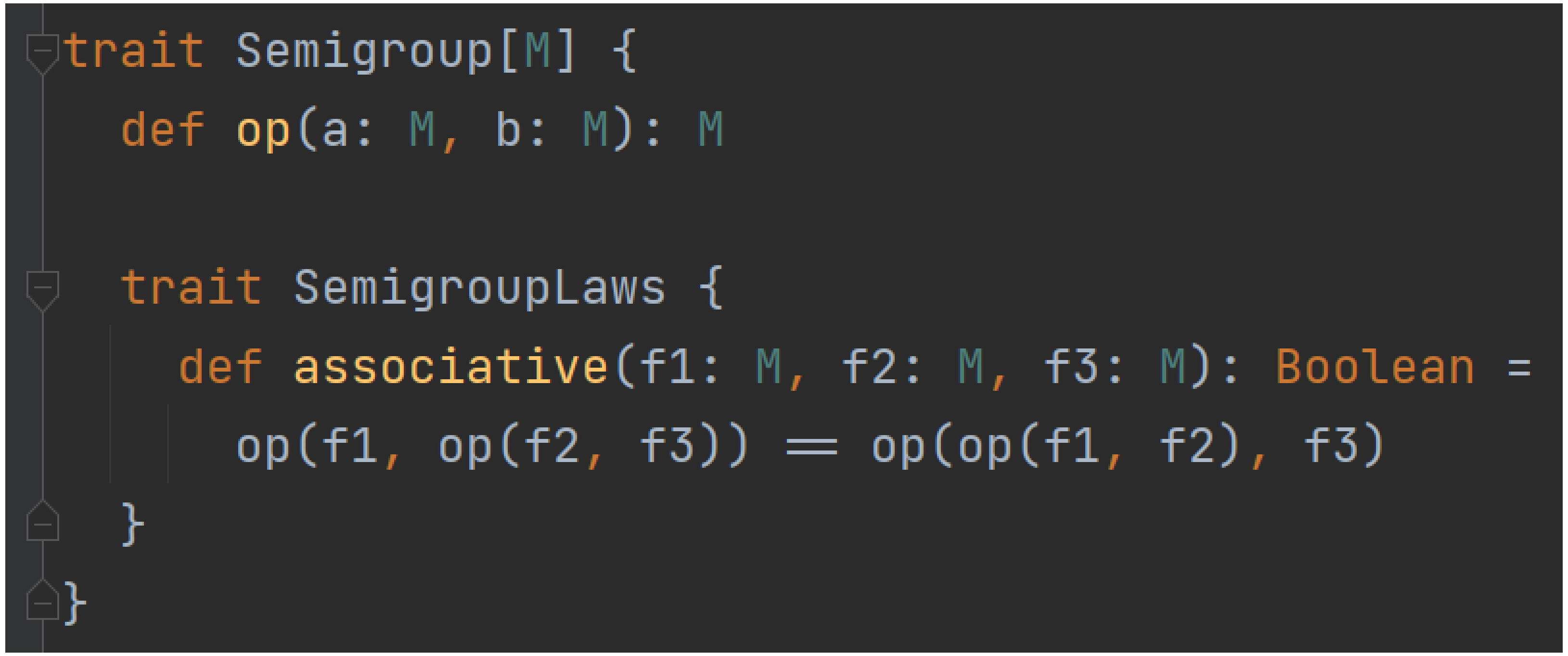
Add some stuff



```
4 * 1
1 * 9
List(1,2,3) ::: Nil
Nil ::: List(1,2,3)
```

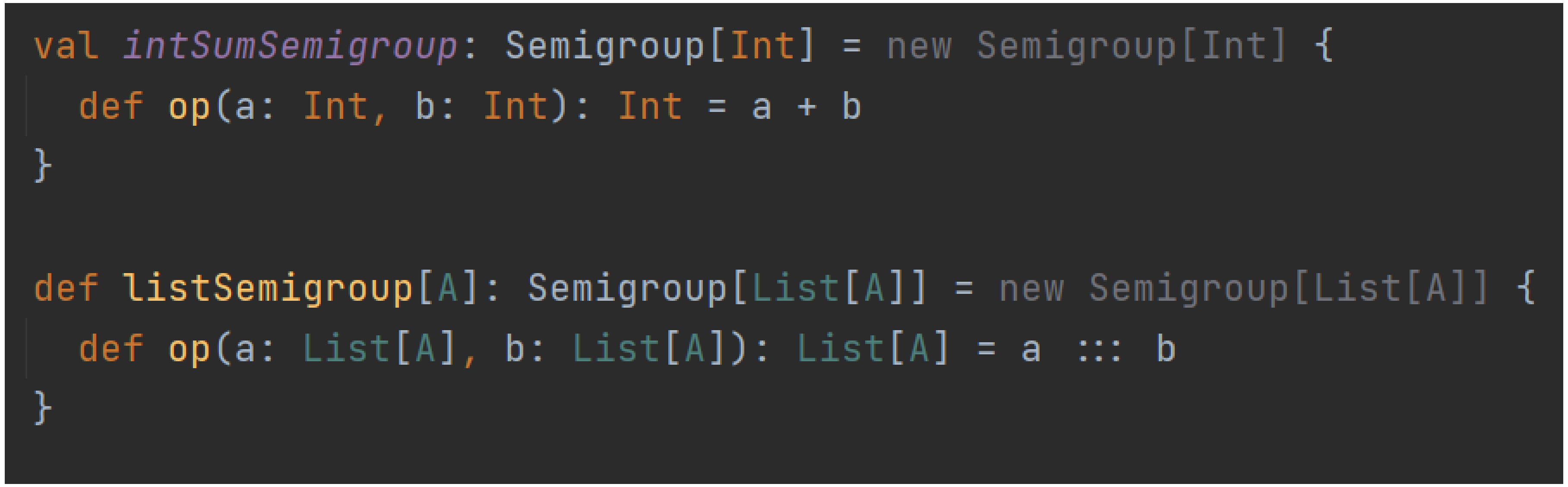
```
val res0: Int = 4
val res1: Int = 9
val res2: List[Int] = List(1, 2, 3)
val res3: List[Int] = List(1, 2, 3)
```

Semigroup

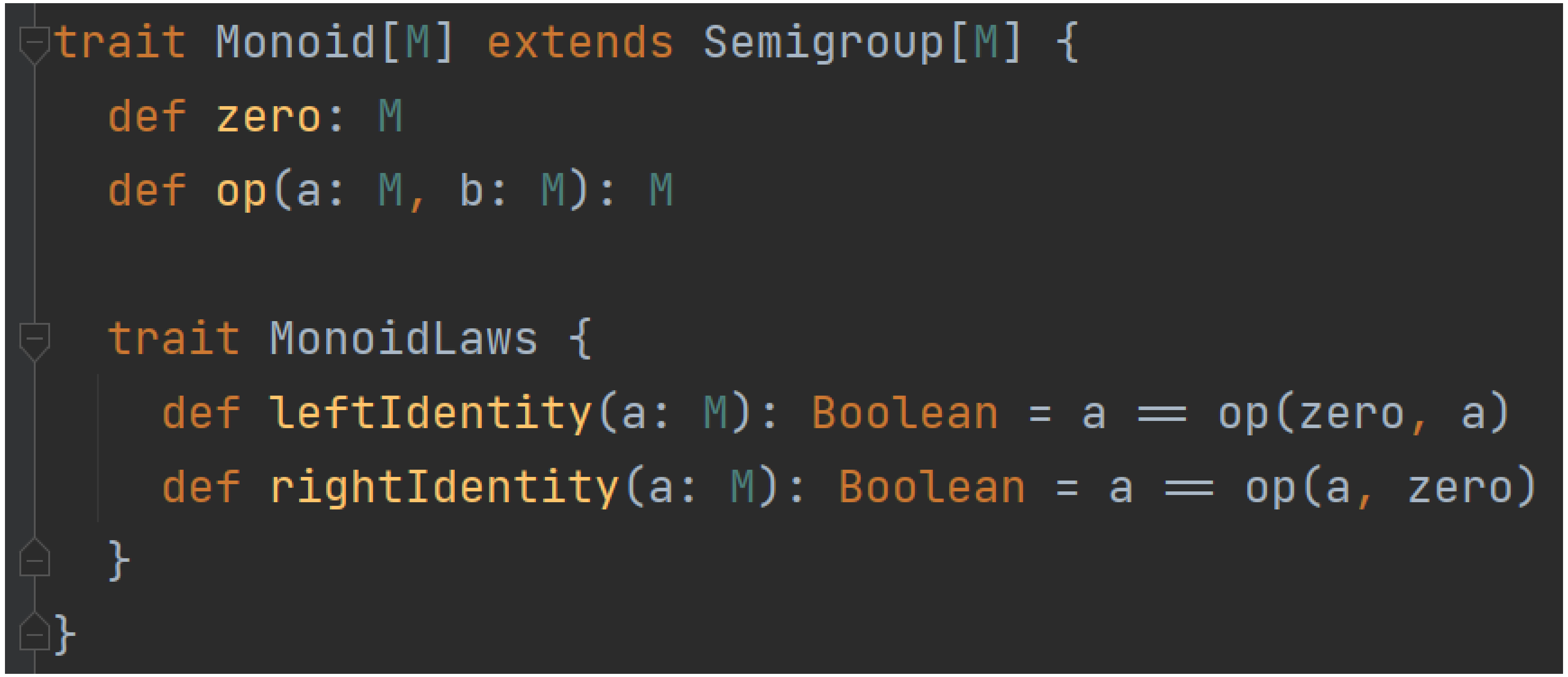


```scala
trait Semigroup[M] {
  def op(a: M, b: M): M

  trait SemigroupLaws {
    def associative(f1: M, f2: M, f3: M): Boolean =
      op(f1, op(f2, f3)) == op(op(f1, f2), f3)
  }
}
```

Some semigroups

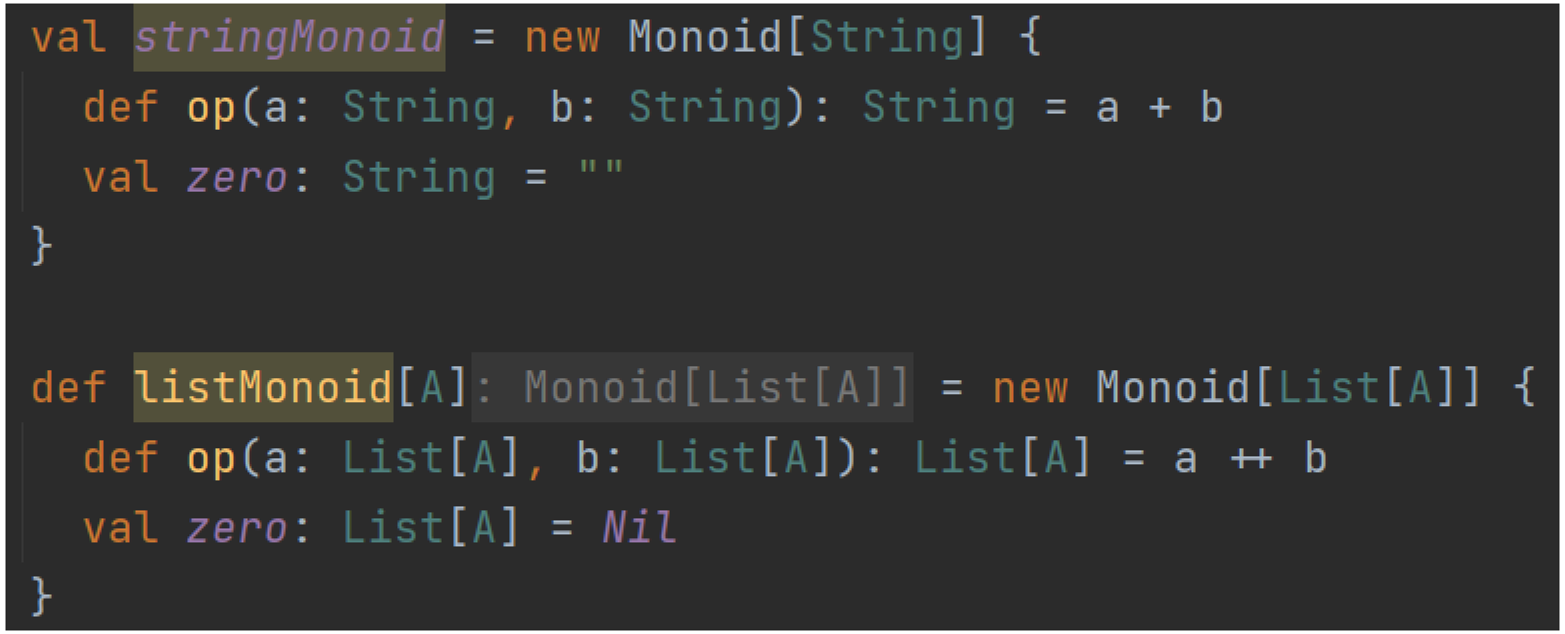

```scala
val intSumSemigroup: Semigroup[Int] = new Semigroup[Int] {
  def op(a: Int, b: Int): Int = a + b
}

def listSemigroup[A]: Semigroup[List[A]] = new Semigroup[List[A]] {
  def op(a: List[A], b: List[A]): List[A] = a ::: b
}
```

Monoid



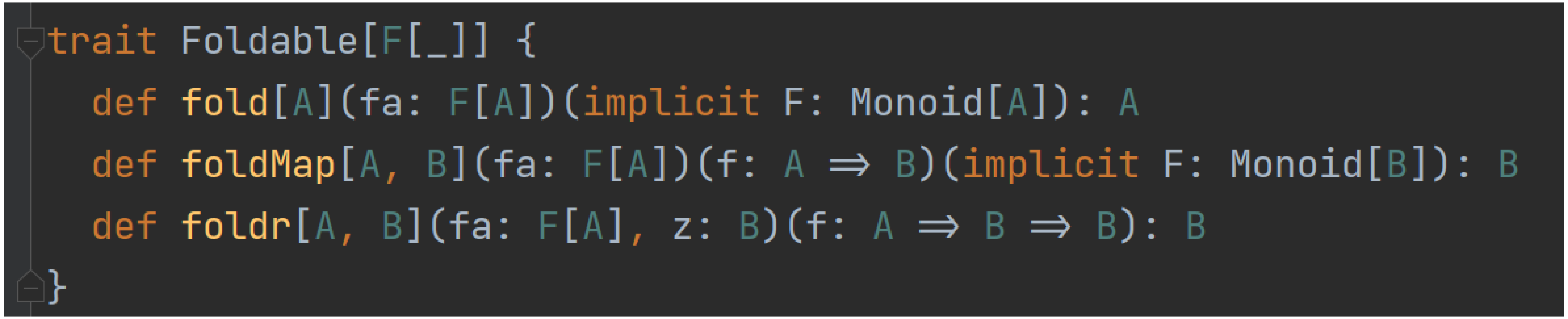
```scala
trait Monoid[M] extends Semigroup[M] {
  def zero: M
  def op(a: M, b: M): M

  trait MonoidLaws {
    def leftIdentity(a: M): Boolean = a == op(zero, a)
    def rightIdentity(a: M): Boolean = a == op(a, zero)
  }
}
```

Monoid implementations



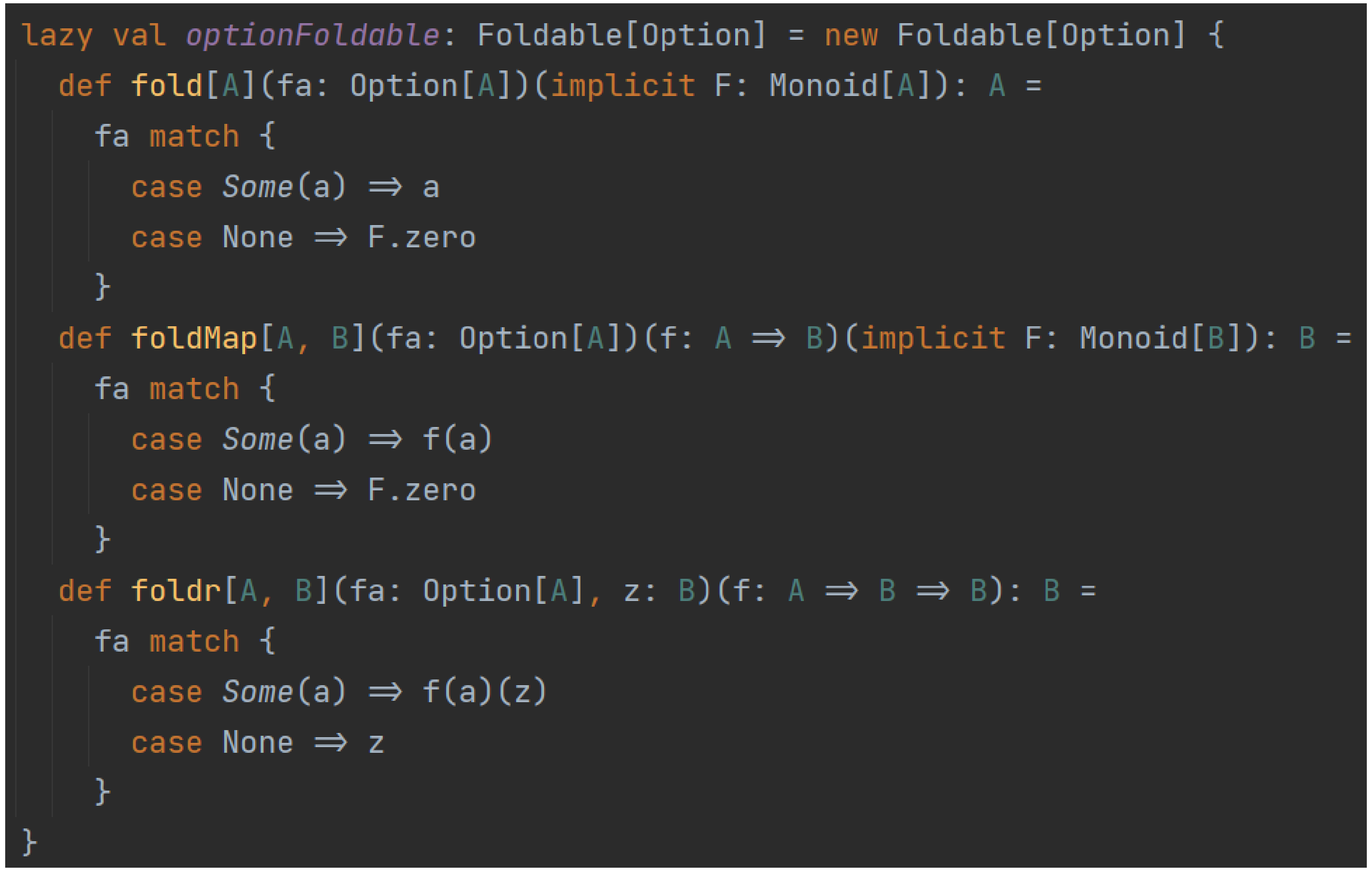
```
val stringMonoid = new Monoid[String] {
  def op(a: String, b: String): String = a + b
  val zero: String = ""
}

def listMonoid[A]: Monoid[List[A]] = new Monoid[List[A]] {
  def op(a: List[A], b: List[A]): List[A] = a ++ b
  val zero: List[A] = Nil
}
```

Foldable



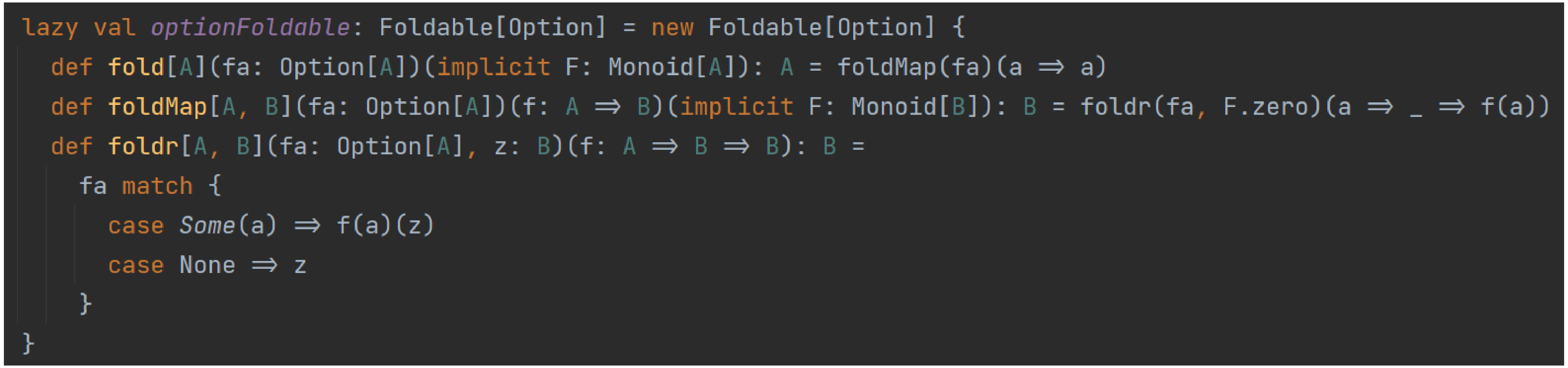
```
trait Foldable[F[_]] {
  def fold[A](fa: F[A])(implicit F: Monoid[A]): A
  def foldMap[A, B](fa: F[A])(f: A ⇒ B)(implicit F: Monoid[B]): B
  def foldr[A, B](fa: F[A], z: B)(f: A ⇒ B ⇒ B): B
}
```

Foldable[Option[_]]



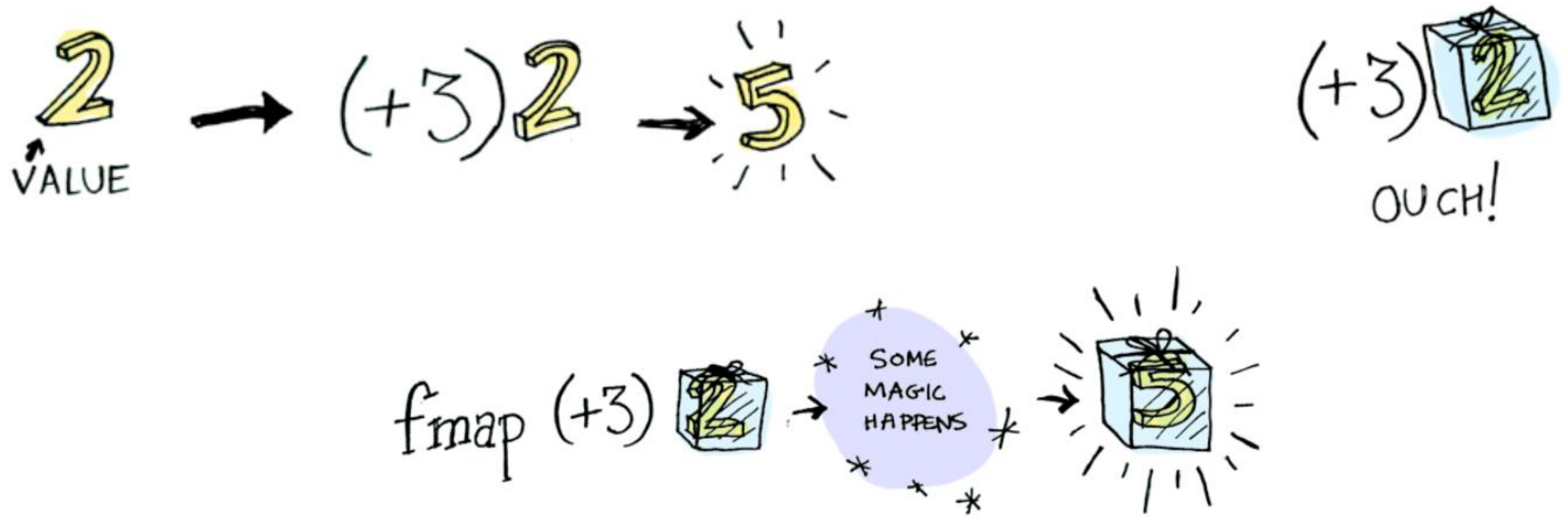
```scala
lazy val optionFoldable: Foldable[Option] = new Foldable[Option] {
  def fold[A](fa: Option[A])(implicit F: Monoid[A]): A =
    fa match {
      case Some(a) ⇒ a
      case None ⇒ F.zero
    }
  def foldMap[A, B](fa: Option[A])(f: A ⇒ B)(implicit F: Monoid[B]): B =
    fa match {
      case Some(a) ⇒ f(a)
      case None ⇒ F.zero
    }
  def foldr[A, B](fa: Option[A], z: B)(f: A ⇒ B ⇒ B): B =
    fa match {
      case Some(a) ⇒ f(a)(z)
      case None ⇒ z
    }
}
```

Foldable[Option[_]]

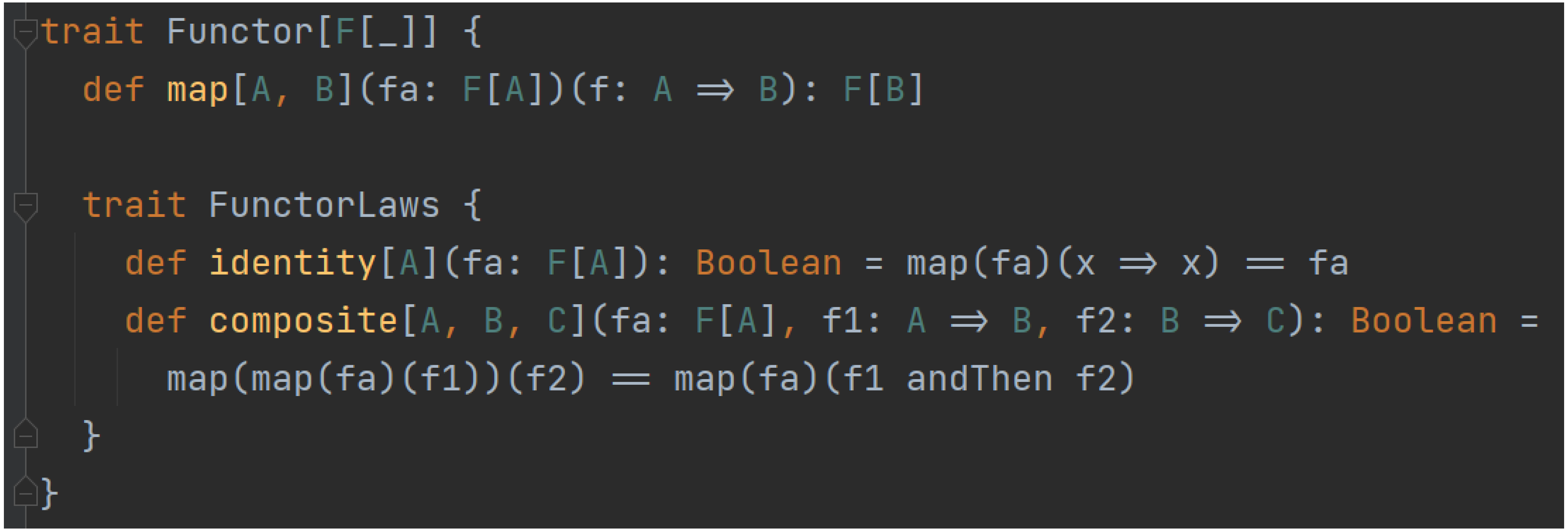


```scala
lazy val optionFoldable: Foldable[Option] = new Foldable[Option] {
  def fold[A](fa: Option[A])(implicit F: Monoid[A]): A = foldMap(fa)(a ⇒ a)
  def foldMap[A, B](fa: Option[A])(f: A ⇒ B)(implicit F: Monoid[B]): B = foldr(fa, F.zero)(a ⇒ _ ⇒ f(a))
  def foldr[A, B](fa: Option[A], z: B)(f: A ⇒ B ⇒ B): B =
    fa match {
      case Some(a) ⇒ f(a)(z)
      case None ⇒ z
    }
}
```

Apply function to a box

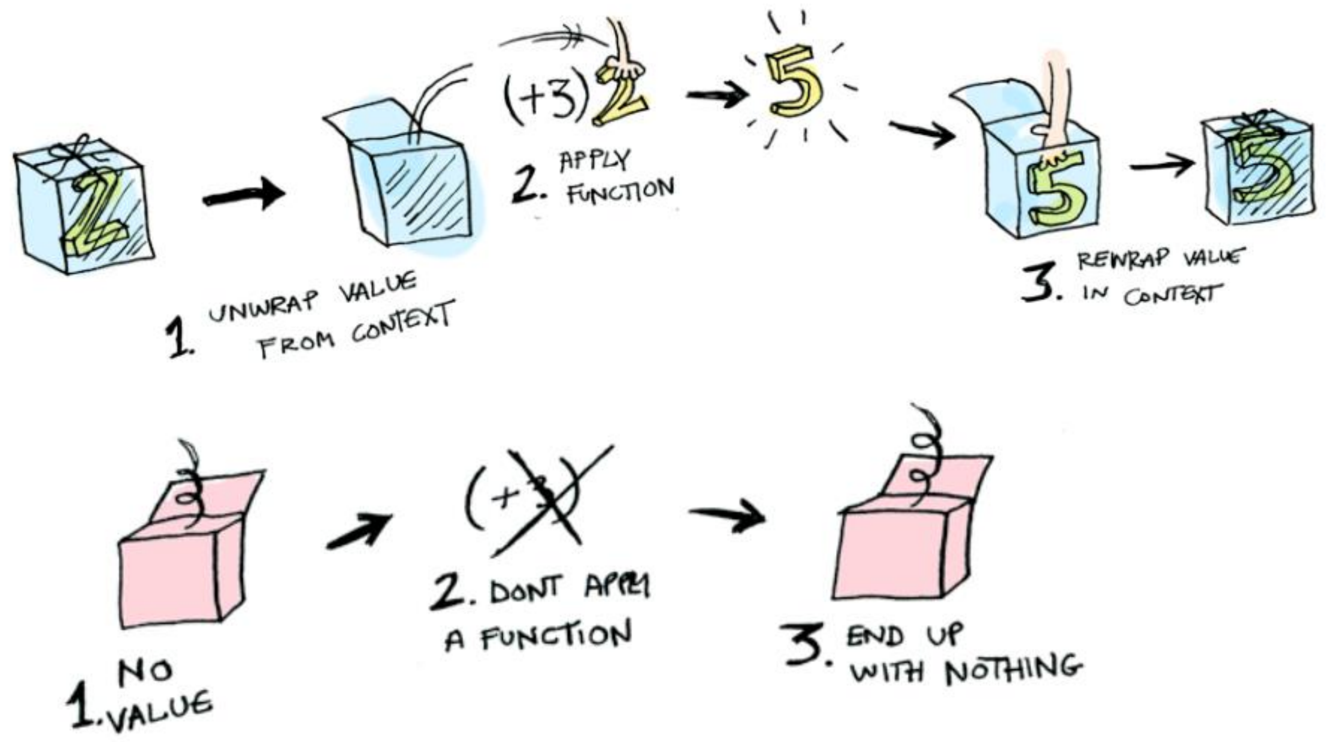


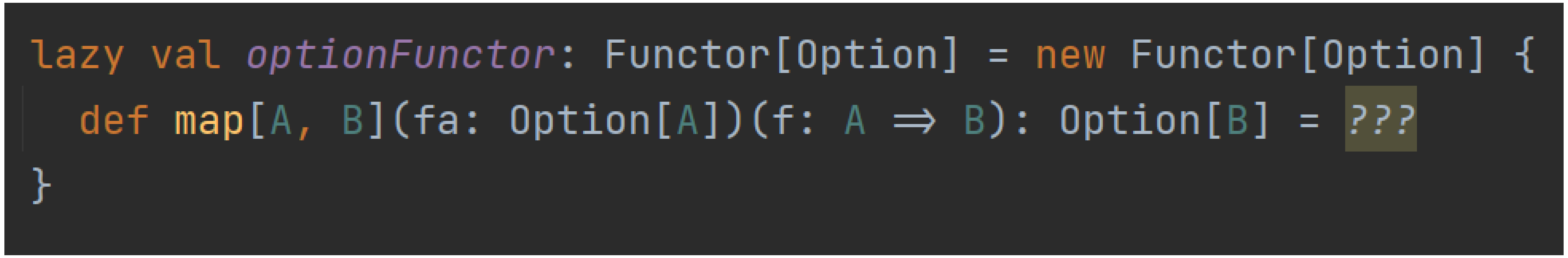
Functor



```scala
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]

  trait FunctorLaws {
    def identity[A](fa: F[A]): Boolean = map(fa)(x => x) == fa
    def composite[A, B, C](fa: F[A], f1: A => B, f2: B => C): Boolean =
      map(map(fa)(f1))(f2) == map(fa)(f1 andThen f2)
  }
}
```

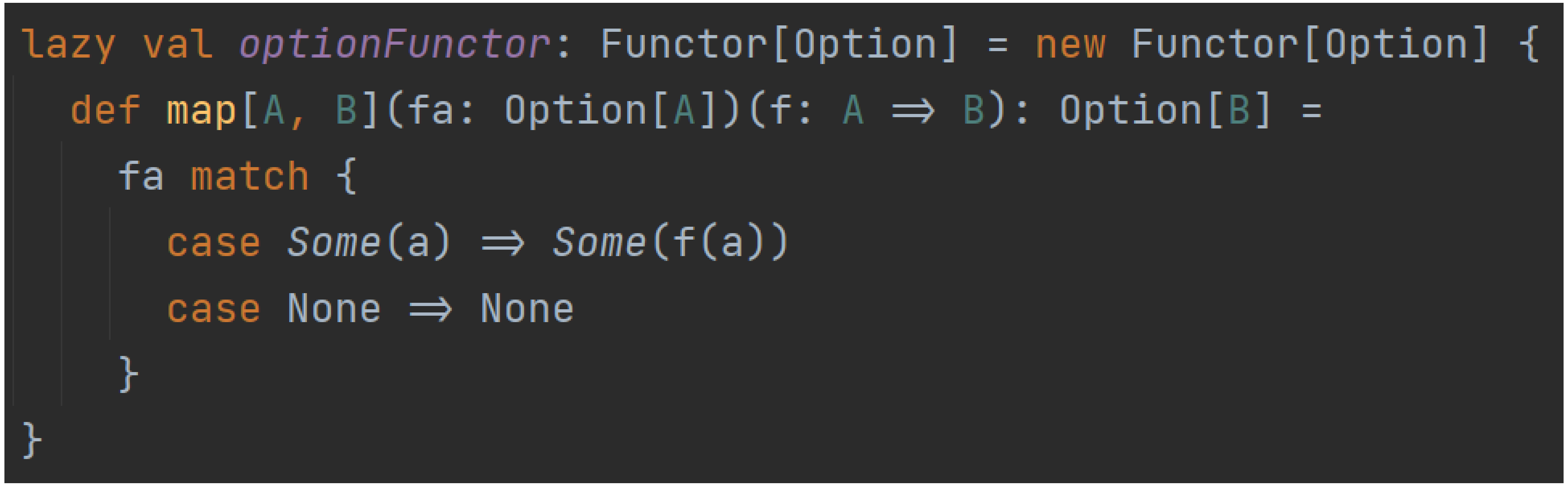
Functor[Option[_]]





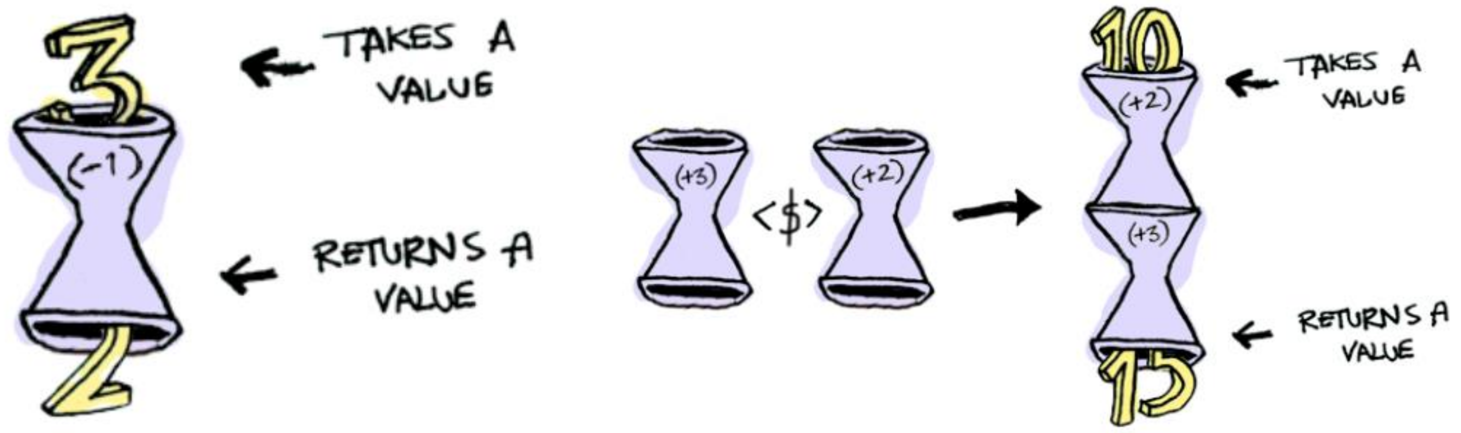
```scala
lazy val optionFunctor: Functor[Option] = new Functor[Option] {
  def map[A, B](fa: Option[A])(f: A ⇒ B): Option[B] = ???
}
```

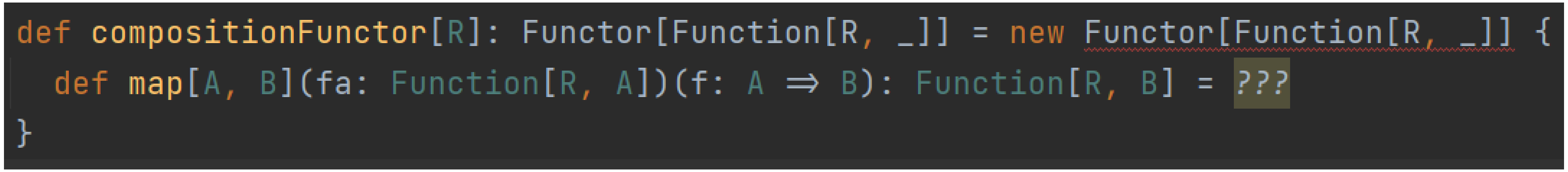
Functor[Option[_]]



```scala
lazy val optionFunctor: Functor[Option] = new Functor[Option] {
  def map[A, B](fa: Option[A])(f: A ⇒ B): Option[B] =
    fa match {
      case Some(a) ⇒ Some(f(a))
      case None ⇒ None
    }
}
```

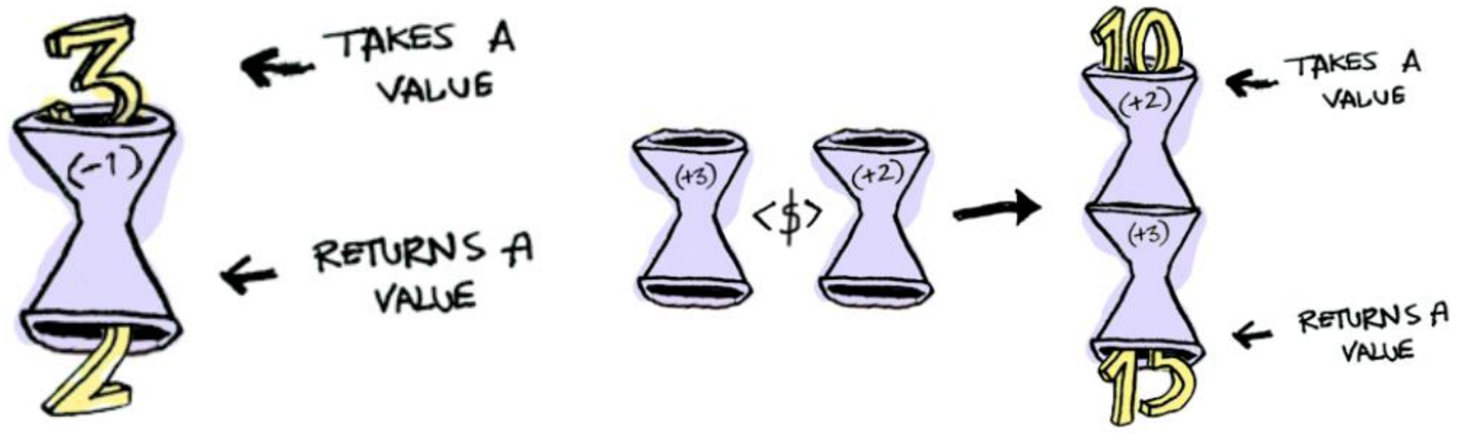
Functor[A => _]

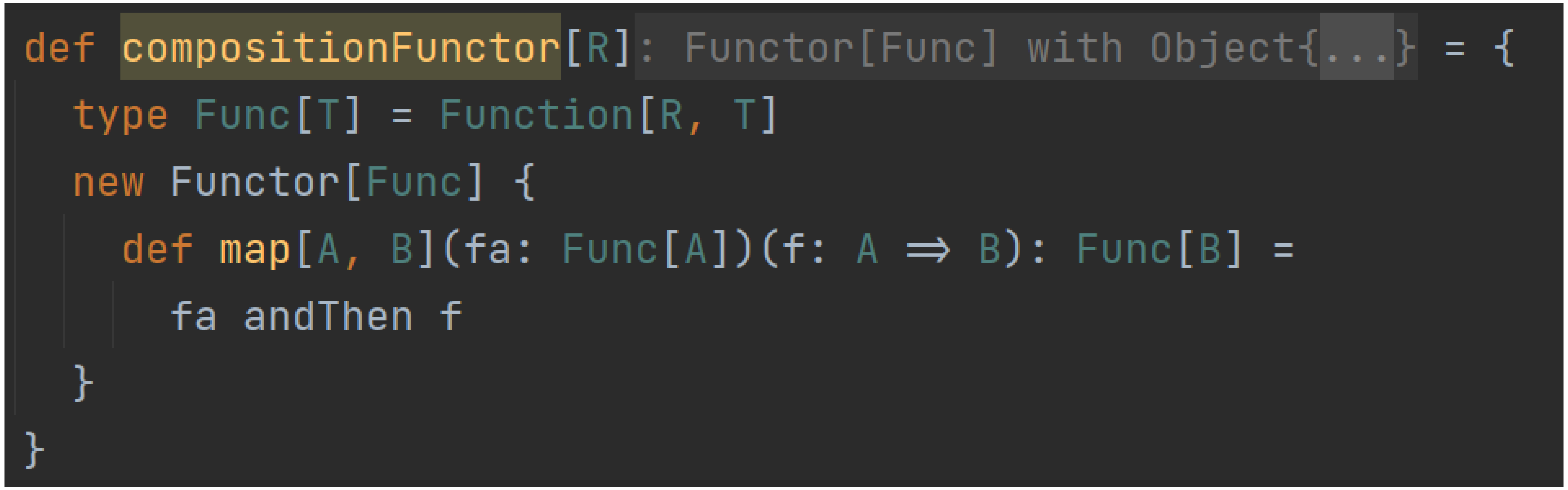




```
def compositionFunctor[R]: Functor[Function[R, _]] = new Functor[Function[R, _]] {
  def map[A, B](fa: Function[R, A])(f: A ⇒ B): Function[R, B] = ???
}
```

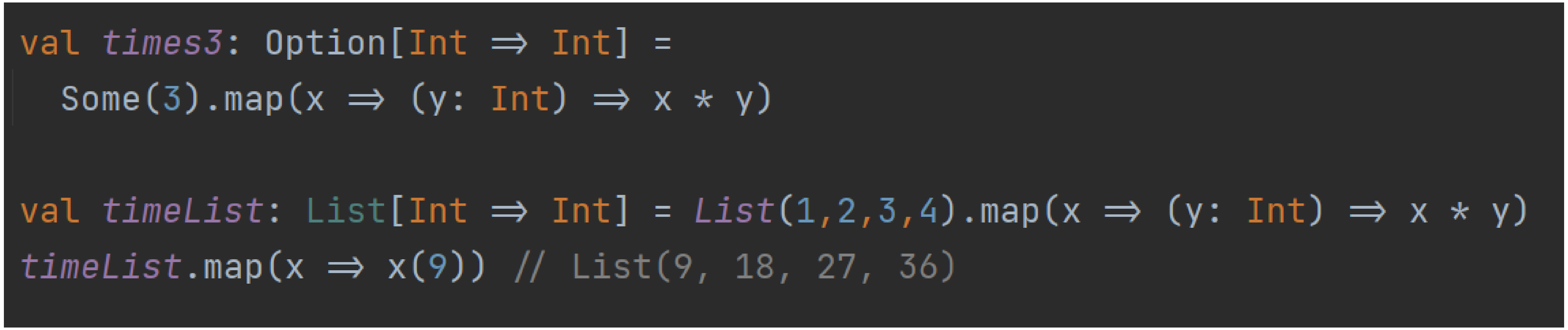
Functor[A => _]





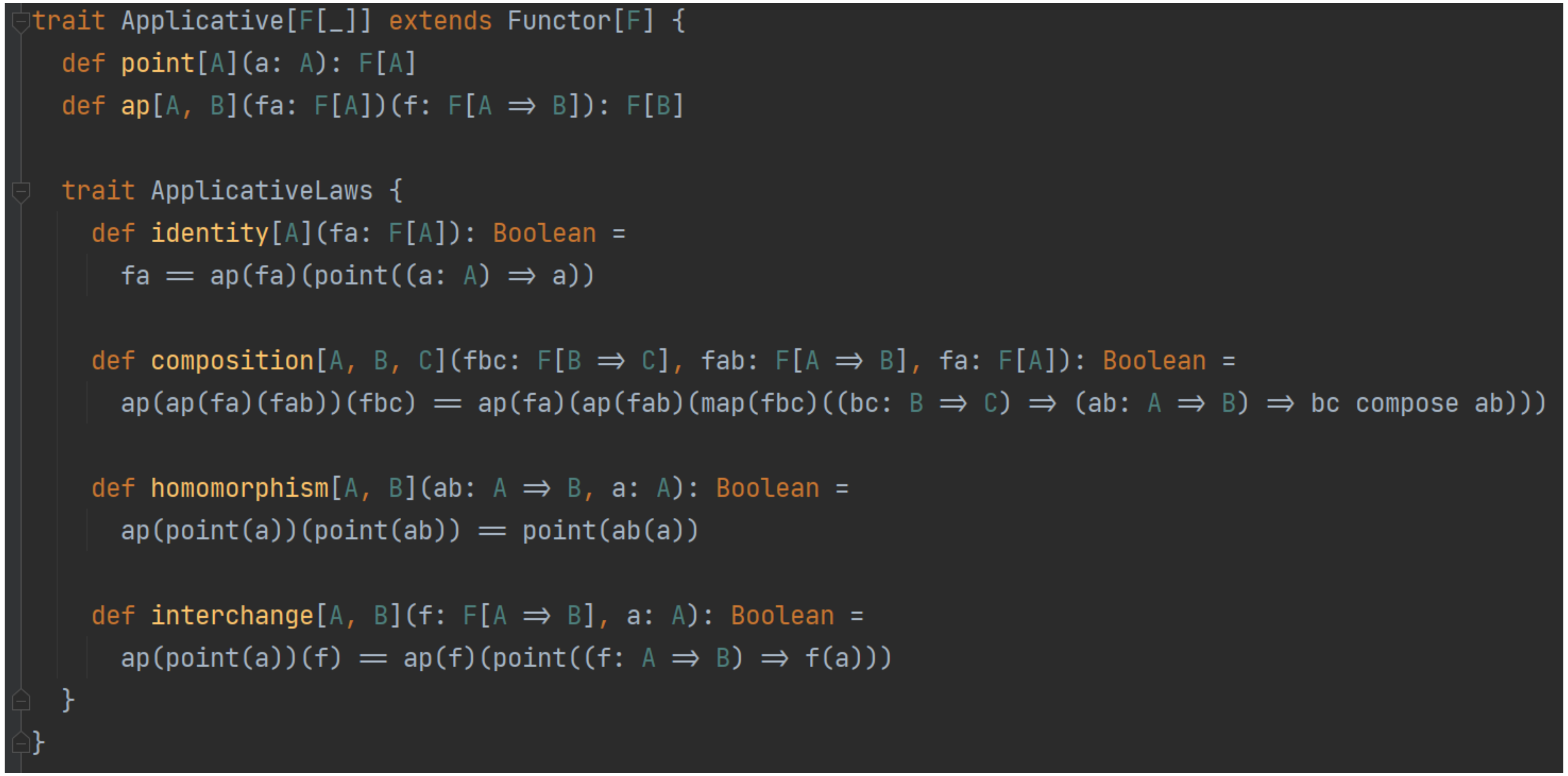
```scala
def compositionFunctor[R]: Functor[Func] with Object{...} = {
  type Func[T] = Function[R, T]
  new Functor[Func] {
    def map[A, B](fa: Func[A])(f: A ⇒ B): Func[B] =
      fa andThen f
  }
}
```

Applicative functors



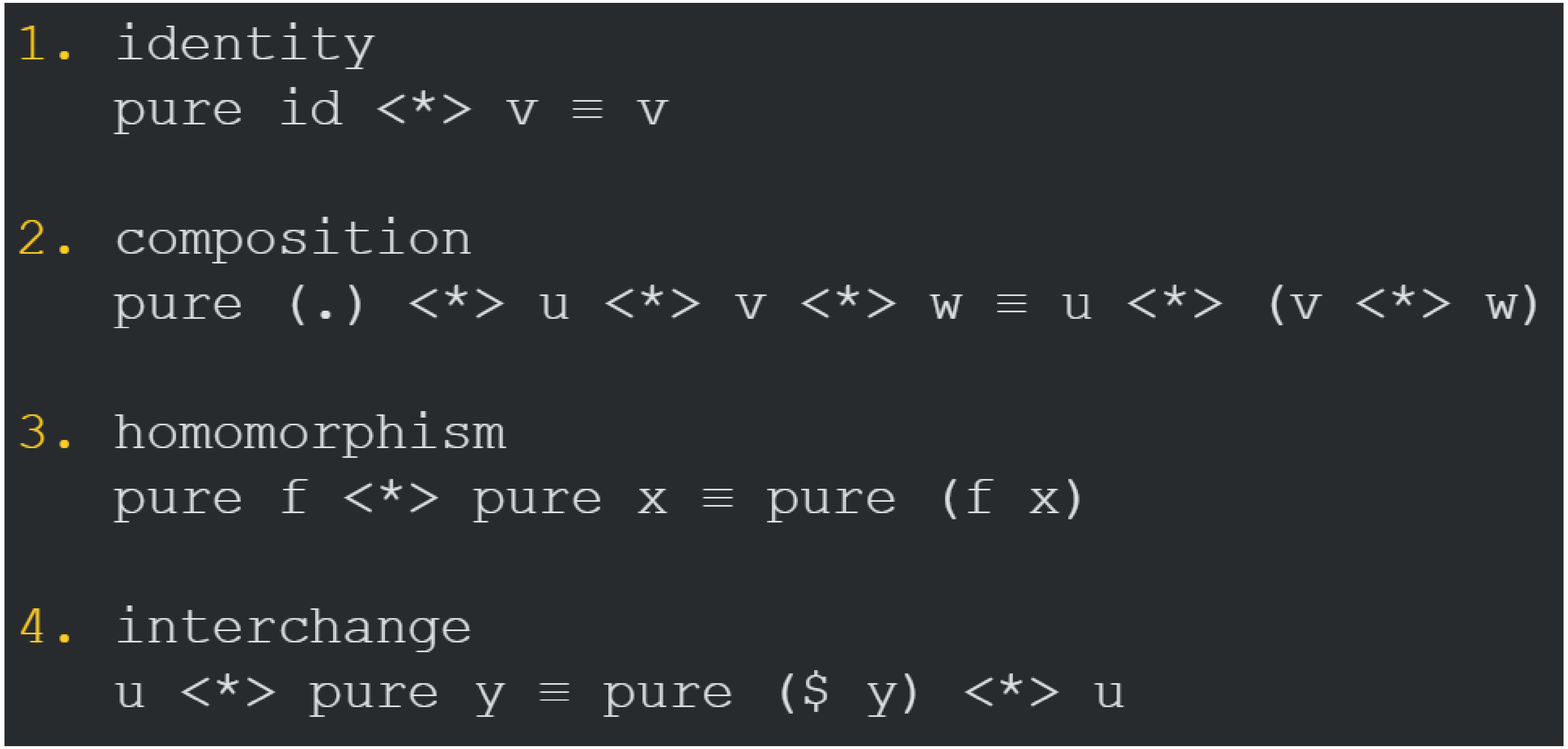
```
val times3: Option[Int ⇒ Int] =
  Some(3).map(x ⇒ (y: Int) ⇒ x * y)

val timeList: List[Int ⇒ Int] = List(1,2,3,4).map(x ⇒ (y: Int) ⇒ x * y)
timeList.map(x ⇒ x(9)) // List(9, 18, 27, 36)
```

Applicative



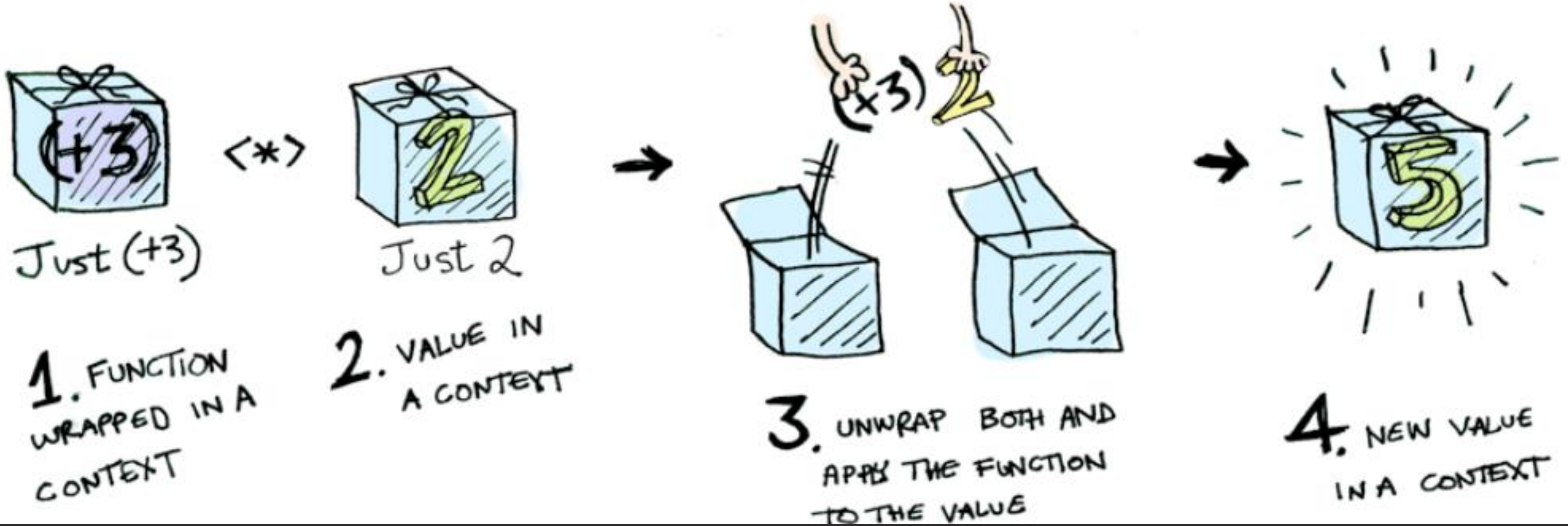
```scala
trait Applicative[F[_]] extends Functor[F] {
  def point[A](a: A): F[A]
  def ap[A, B](fa: F[A])(f: F[A => B]): F[B]

  trait ApplicativeLaws {
    def identity[A](fa: F[A]): Boolean =
      fa == ap(fa)(point((a: A) => a))

    def composition[A, B, C](fbc: F[B => C], fab: F[A => B], fa: F[A]): Boolean =
      ap(ap(fa)(fab))(fbc) == ap(fa)(ap(fab)(map(fbc)((bc: B => C) => (ab: A => B) => bc compose ab)))

    def homomorphism[A, B](ab: A => B, a: A): Boolean =
      ap(point(a))(point(ab)) == point(ab(a))

    def interchange[A, B](f: F[A => B], a: A): Boolean =
      ap(point(a))(f) == ap(f)(point((f: A => B) => f(a)))
  }
}
```

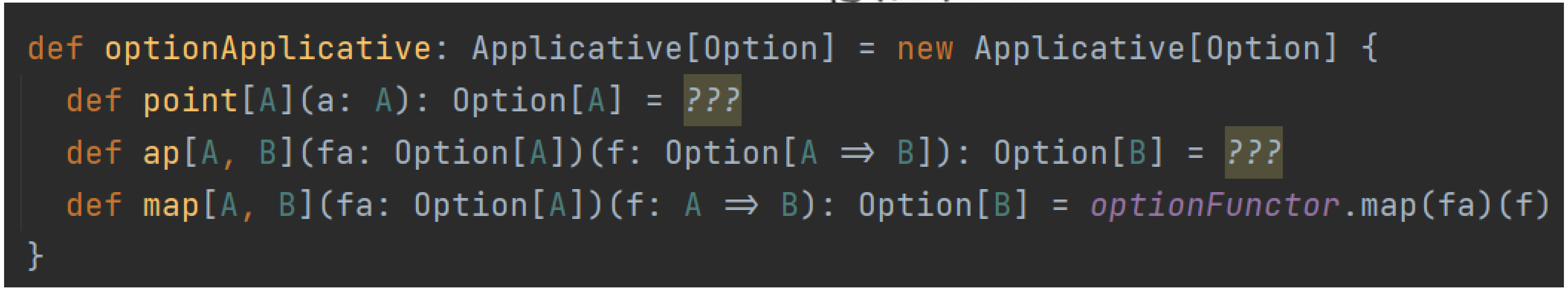
Applicative Laws



```
1. identity
   pure id <*> v ≡ v

2. composition
   pure (.) <*> u <*> v <*> w ≡ u <*> (v <*> w)

3. homomorphism
   pure f <*> pure x ≡ pure (f x)

4. interchange
   u <*> pure y ≡ pure ($ y) <*> u
```

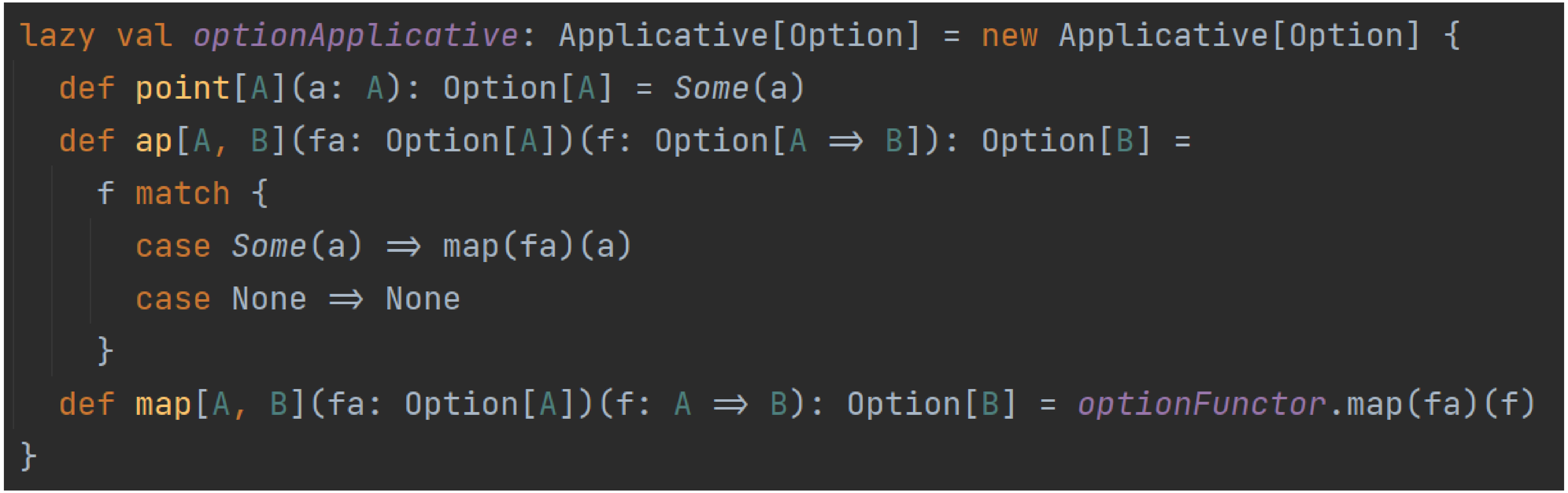
Option Applicative


Just (+3)
<*>
Just 2
(+3)
2
1. FUNCTION WRAPPED IN A CONTEXT
2. VALUE IN A CONTEXT
3. UNWRAP BOTH AND APPLY THE FUNCTION TO THE VALUE
4. NEW VALUE IN A CONTEXT
5



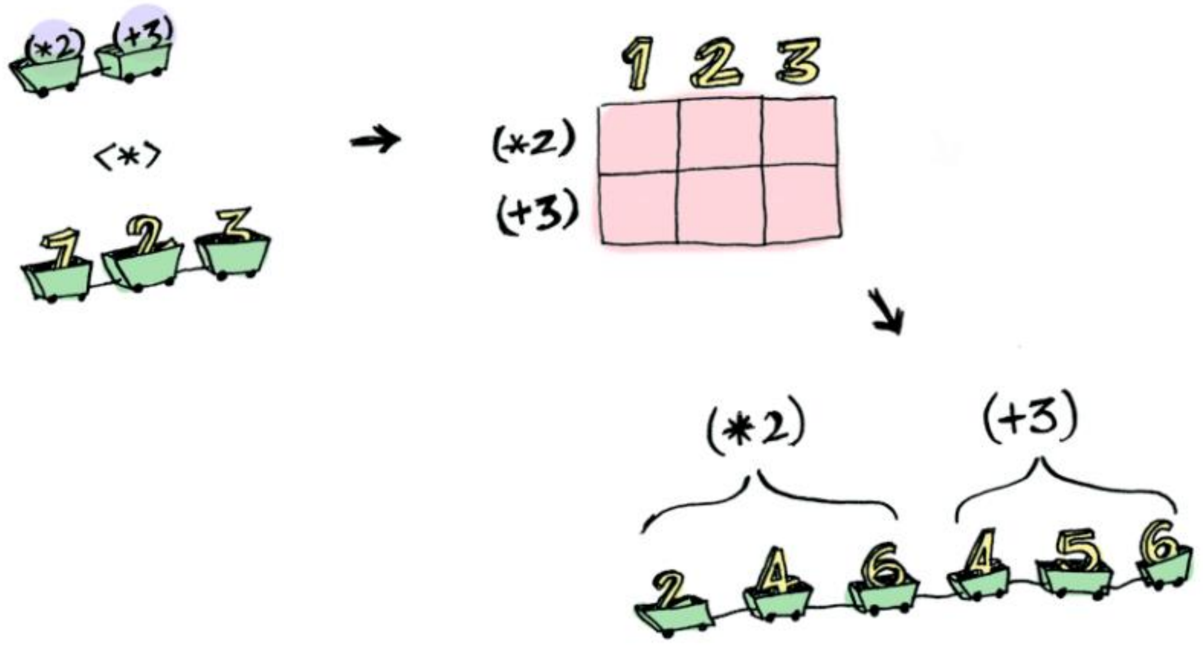
```scala
def optionApplicative: Applicative[Option] = new Applicative[Option] {
  def point[A](a: A): Option[A] = ???
  def ap[A, B](fa: Option[A])(f: Option[A ⇒ B]): Option[B] = ???
  def map[A, B](fa: Option[A])(f: A ⇒ B): Option[B] = optionFunctor.map(fa)(f)
}
```

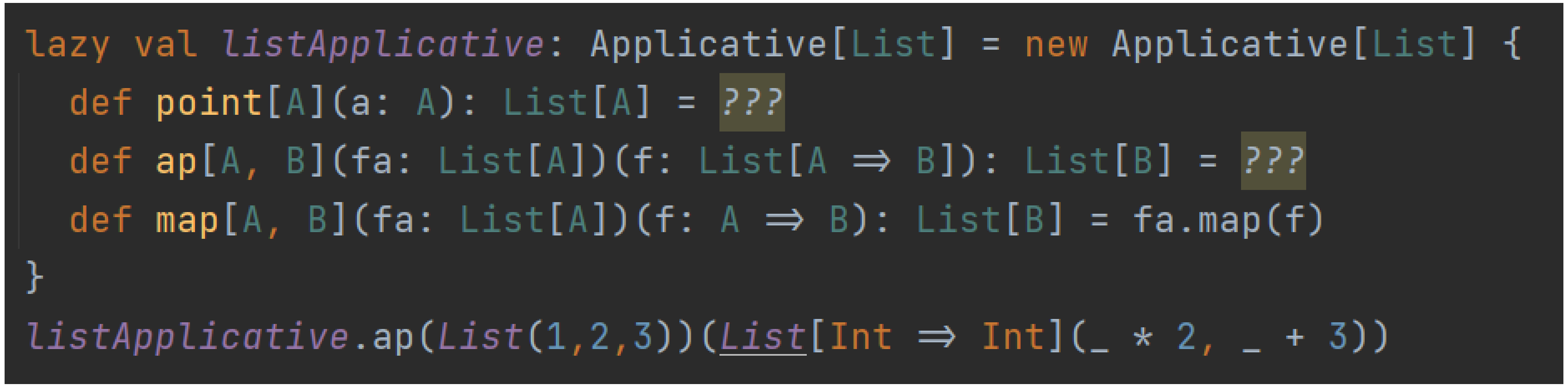
Option Applicative



```scala
lazy val optionApplicative: Applicative[Option] = new Applicative[Option] {
  def point[A](a: A): Option[A] = Some(a)
  def ap[A, B](fa: Option[A])(f: Option[A ⇒ B]): Option[B] =
    f match {
      case Some(a) ⇒ map(fa)(a)
      case None ⇒ None
    }
  def map[A, B](fa: Option[A])(f: A ⇒ B): Option[B] = optionFunctor.map(fa)(f)
}
```

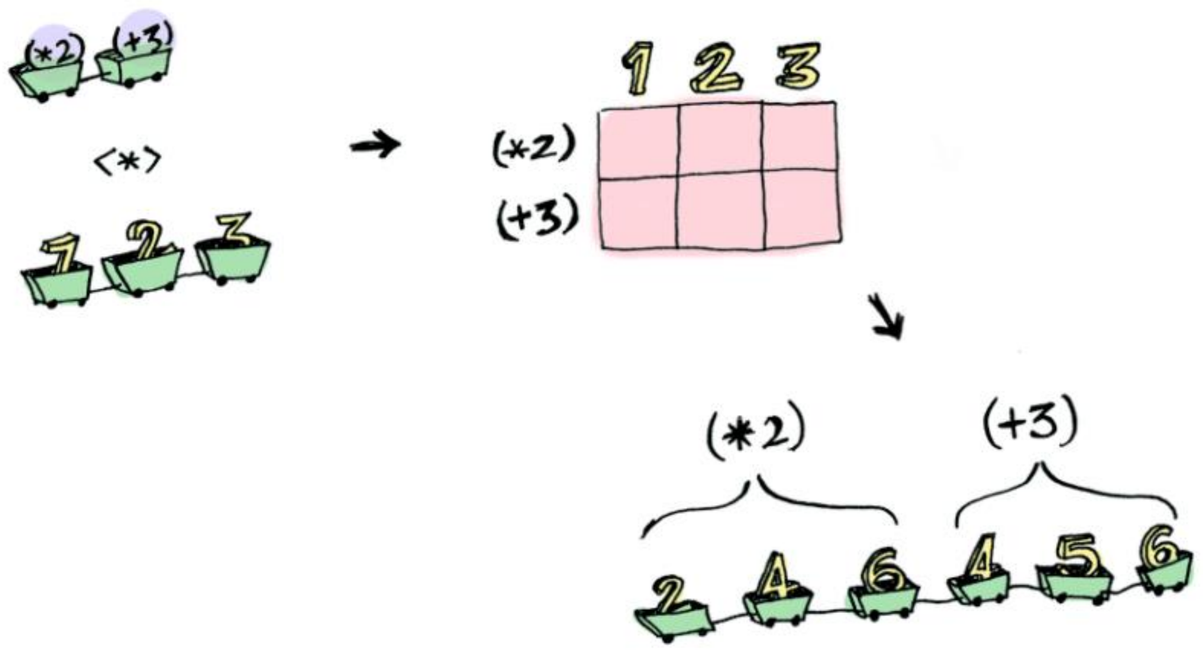
List Applicative





```scala
lazy val listApplicative: Applicative[List] = new Applicative[List] {
  def point[A](a: A): List[A] = ???
  def ap[A, B](fa: List[A])(f: List[A ⇒ B]): List[B] = ???
  def map[A, B](fa: List[A])(f: A ⇒ B): List[B] = fa.map(f)
}
listApplicative.ap(List(1,2,3))(List[Int ⇒ Int](_ * 2, _ + 3))
```

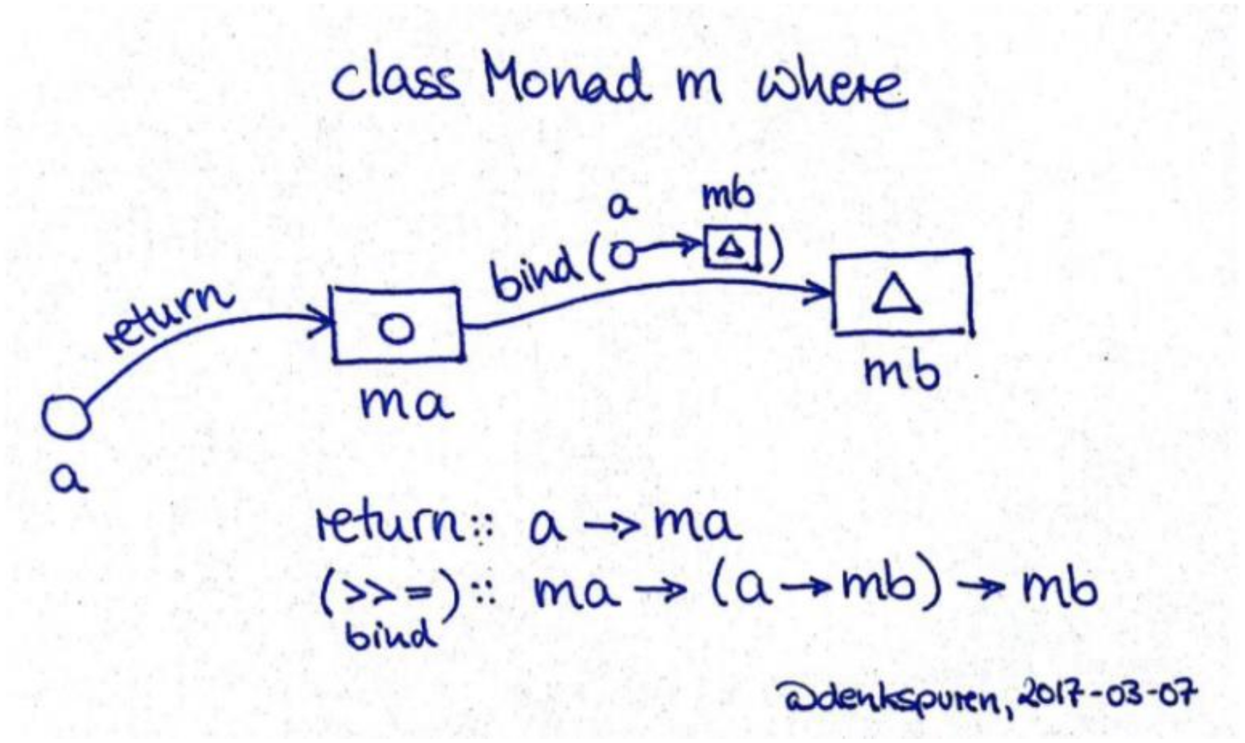
List Applicative



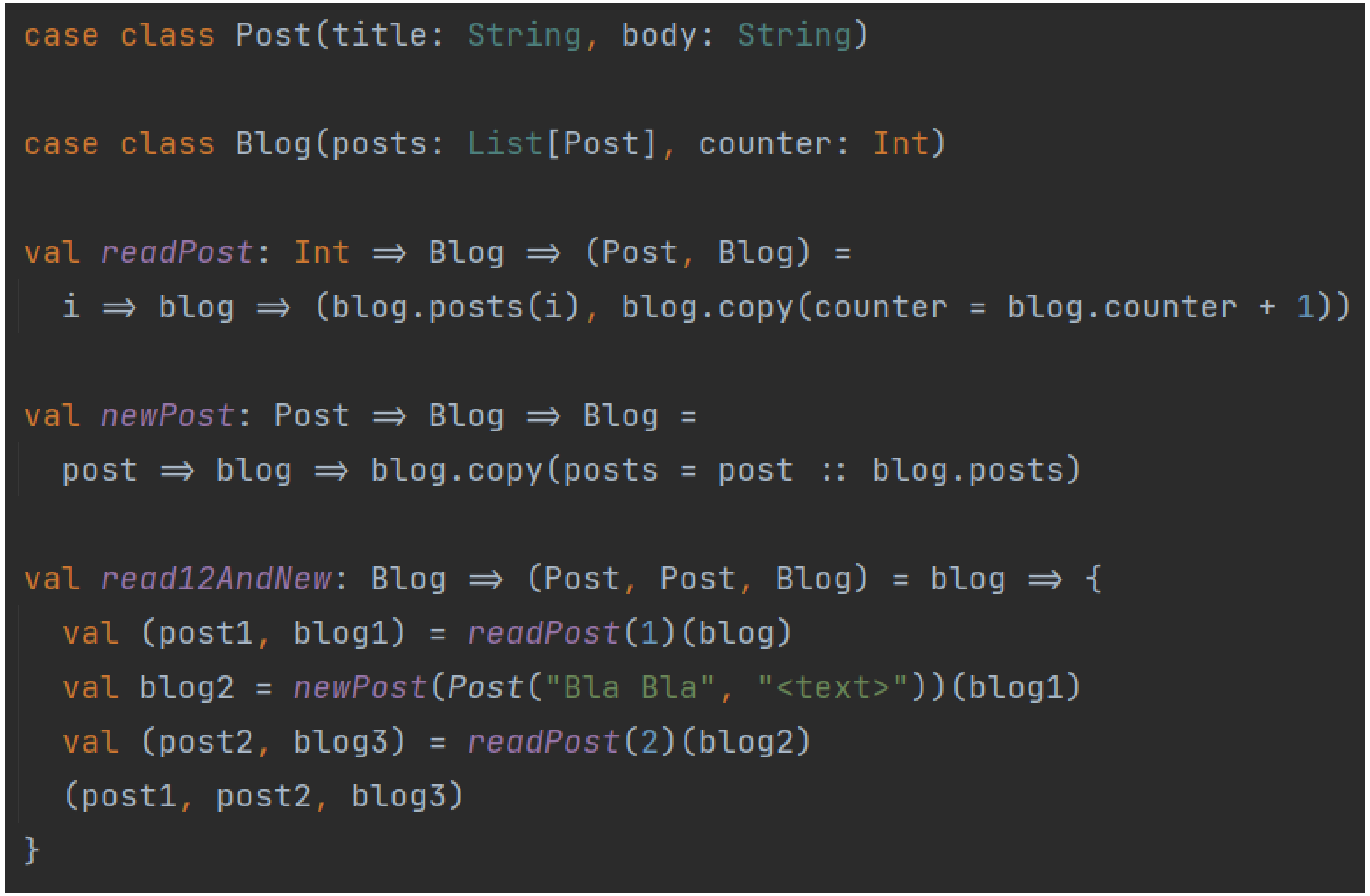
```
lazy val listApplicative: Applicative[List] = new Applicative[List] {
  def point[A](a: A): List[A] = List(a)
  def ap[A, B](fa: List[A])(f: List[A ⇒ B]): List[B] =
    for {fs ← f; fas ← fa} yield fs(fas)
  def map[A, B](fa: List[A])(f: A ⇒ B): List[B] = fa.map(f)
}
```

Monad

- Variable — container for data
- Monad — container for sequentially composable computation



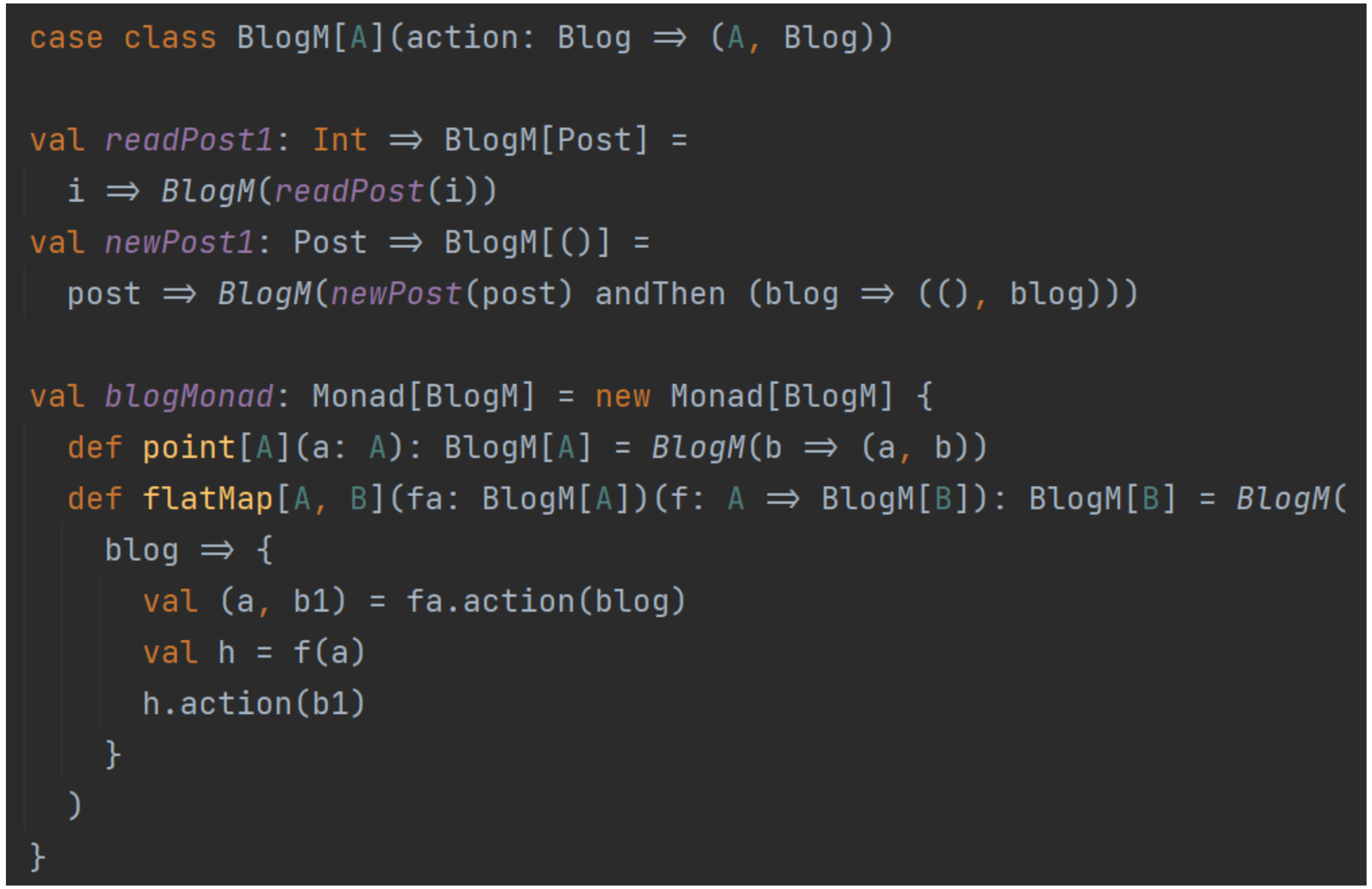
Blog



```scala
case class Post(title: String, body: String)

case class Blog(posts: List[Post], counter: Int)

val readPost: Int ⇒ Blog ⇒ (Post, Blog) =
  i ⇒ blog ⇒ (blog.posts(i), blog.copy(counter = blog.counter + 1))

val newPost: Post ⇒ Blog ⇒ Blog =
  post ⇒ blog ⇒ blog.copy(posts = post :: blog.posts)

val read12AndNew: Blog ⇒ (Post, Post, Blog) = blog ⇒ {
  val (post1, blog1) = readPost(1)(blog)
  val blog2 = newPost(Post("Bla Bla", "<text>"))(blog1)
  val (post2, blog3) = readPost(2)(blog2)
  (post1, post2, blog3)
}
```

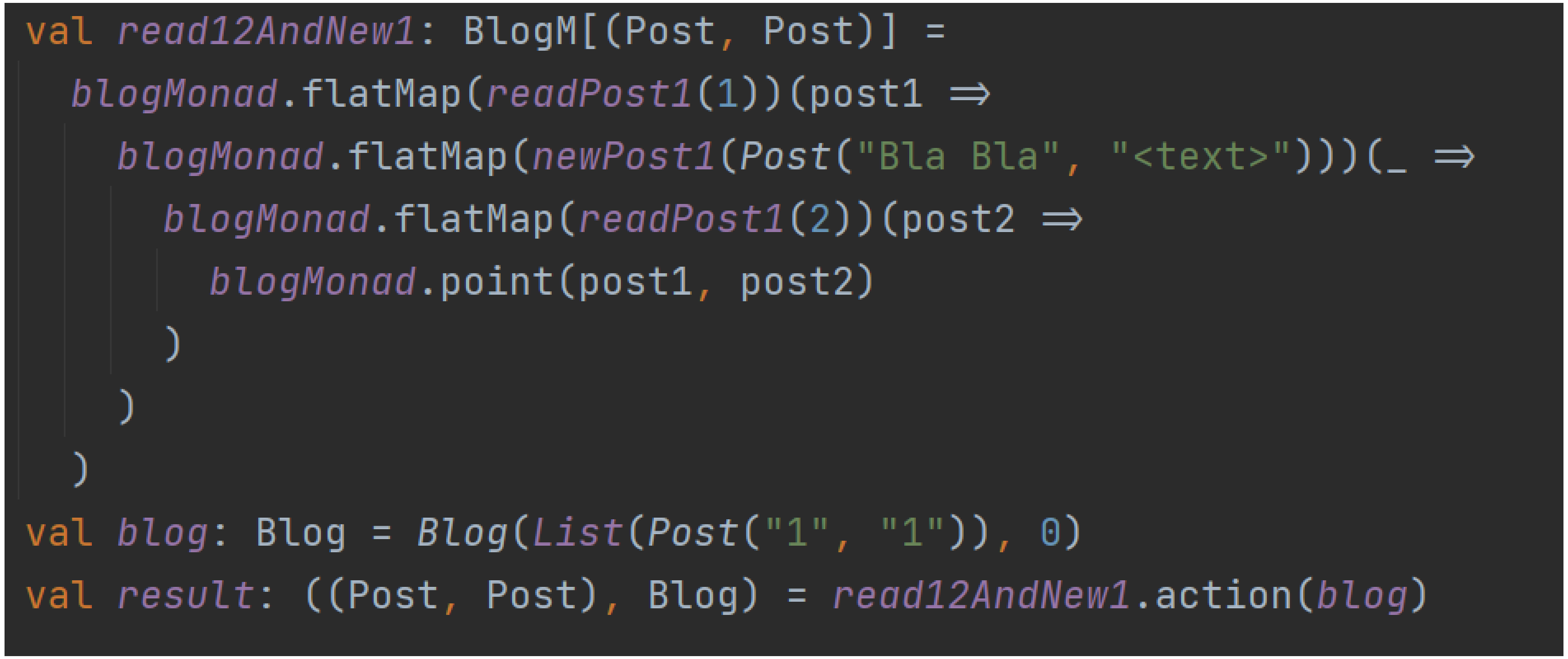
Blog Monad

2021



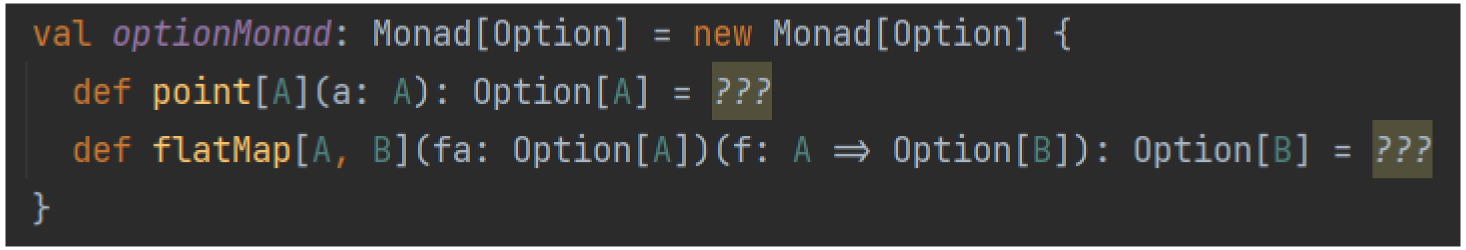
```scala
case class BlogM[A](action: Blog ⇒ (A, Blog))

val readPost1: Int ⇒ BlogM[Post] =
  i ⇒ BlogM(readPost(i))
val newPost1: Post ⇒ BlogM[()] =
  post ⇒ BlogM(newPost(post) andThen (blog ⇒ ((), blog)))

val blogMonad: Monad[BlogM] = new Monad[BlogM] {
  def point[A](a: A): BlogM[A] = BlogM(b ⇒ (a, b))
  def flatMap[A, B](fa: BlogM[A])(f: A ⇒ BlogM[B]): BlogM[B] = BlogM(
    blog ⇒ {
      val (a, b1) = fa.action(blog)
      val h = f(a)
      h.action(b1)
    }
  )
}
```

Blog Monad application



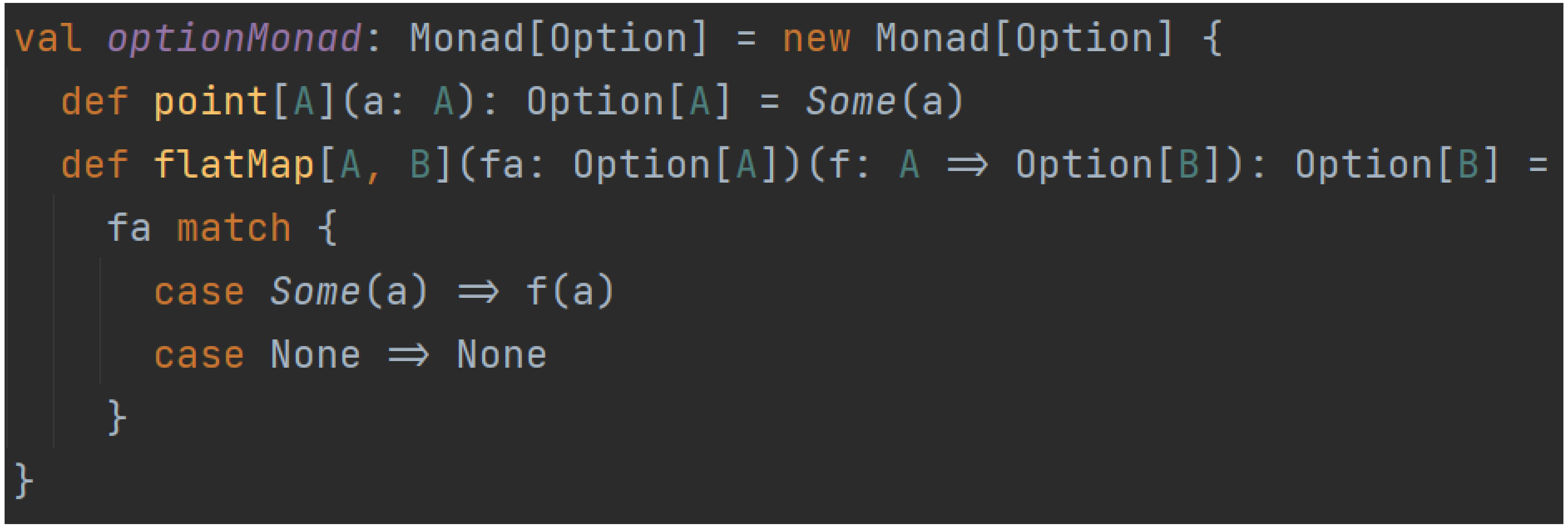
```scala
val read12AndNew1: BlogM[(Post, Post)] =
  blogMonad.flatMap(readPost1(1))(post1 ⇒
    blogMonad.flatMap(newPost1(Post("Bla Bla", "<text>")))(_ ⇒
      blogMonad.flatMap(readPost1(2))(post2 ⇒
        blogMonad.point(post1, post2)
      )
    )
  )
val blog: Blog = Blog(List(Post("1", "1")), 0)
val result: ((Post, Post), Blog) = read12AndNew1.action(blog)
```

Monad[Option]



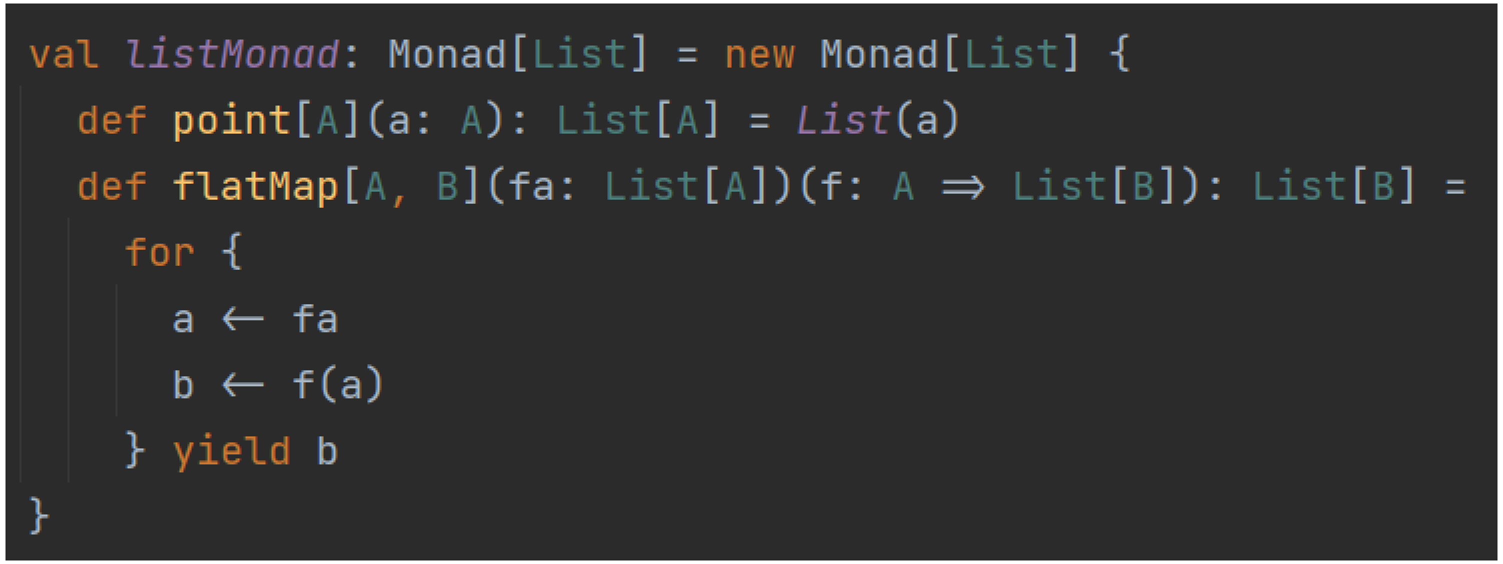
```scala
val optionMonad: Monad[Option] = new Monad[Option] {
  def point[A](a: A): Option[A] = ???
  def flatMap[A, B](fa: Option[A])(f: A ⇒ Option[B]): Option[B] = ???
}
```

Monad[Option]



```scala
val optionMonad: Monad[Option] = new Monad[Option] {
  def point[A](a: A): Option[A] = Some(a)
  def flatMap[A, B](fa: Option[A])(f: A ⇒ Option[B]): Option[B] =
    fa match {
      case Some(a) ⇒ f(a)
      case None ⇒ None
    }
}
```

Monad[List]


```scala
val listMonad: Monad[List] = new Monad[List] {
  def point[A](a: A): List[A] = List(a)
  def flatMap[A, B](fa: List[A])(f: A ⇒ List[B]): List[B] =
    for {
      a ← fa
      b ← f(a)
    } yield b
}
```

Useful links

- Immutability we can afford - https://elizarov.medium.com/immutability-we-can-afford-10c0dcb8351d
- Scala FP - https://docs.scala-lang.org/overviews/scala-book/functional-programming.html
- Say no to return - https://blog.knoldus.com/scala-best-practices-say-no-to-return/#:~:text=Putting%20in%20simple%20words%2C%20return,It%20evaluates%20that%20itself
- Scala functional programming - https://alvinalexander.com/downloads/fpsimplified-free-preview.pdf
- Monoid - https://eed3si9n.com/learning-scalaz/Monoid.html
- Functors, Applicative Functors and Monoids - http://learnyouahaskell.com/functors-applicative-functors-and-monoids
- A Fistful of Monads - http://learnyouahaskell.com/a-fistful-of-monads