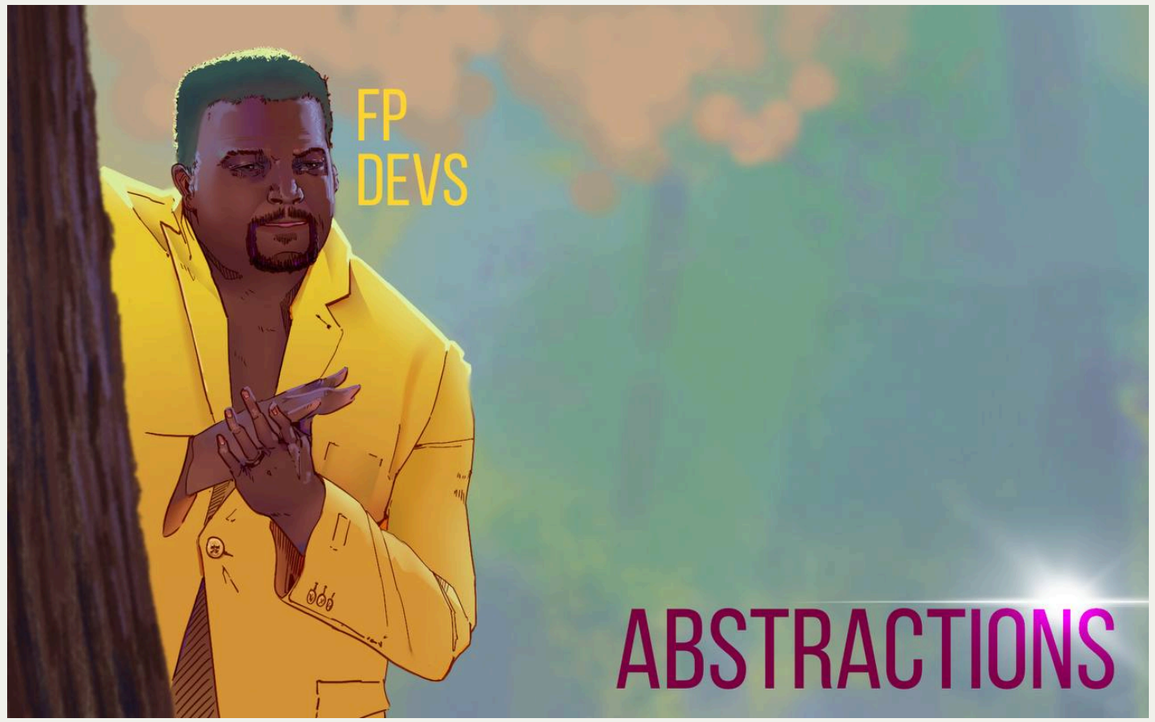
Классические тайпклассы используемые в ФП



Хакимов Айнур

- Scala разработчик в Tinkoff
- До Tinkoff писал на C/C++

План курса ФИЛП 2024

1. Вводная
2. Базовый синтаксис
3. Функции
4. ADT, Collections
5. Implicits, Typeclasses
6. Common Typeclasses
7. Monad
8. Асинхронное выполнение
9. Future
10. Монада IO

План лекции

План лекции

- Рассмотрим классические примеры тайпклассов и их свойства

План лекции

- Рассмотрим классические примеры тайпклассов и их свойства
- Увидим, как писать максимально обобщенный код на Scala

План лекции

- Рассмотрим классические примеры тайпклассов и их свойства
- Увидим, как писать максимально обобщенный код на Scala
- Познакомимся со Scala-библиотекой, которая содержит в себе классические тайпклассы

```
1  1 + 2 + 3
2
3  "I" + "love" + "fp"
4
5  List(1, 2, 3) ++ List(4, 5, 6)
```

Semigroup

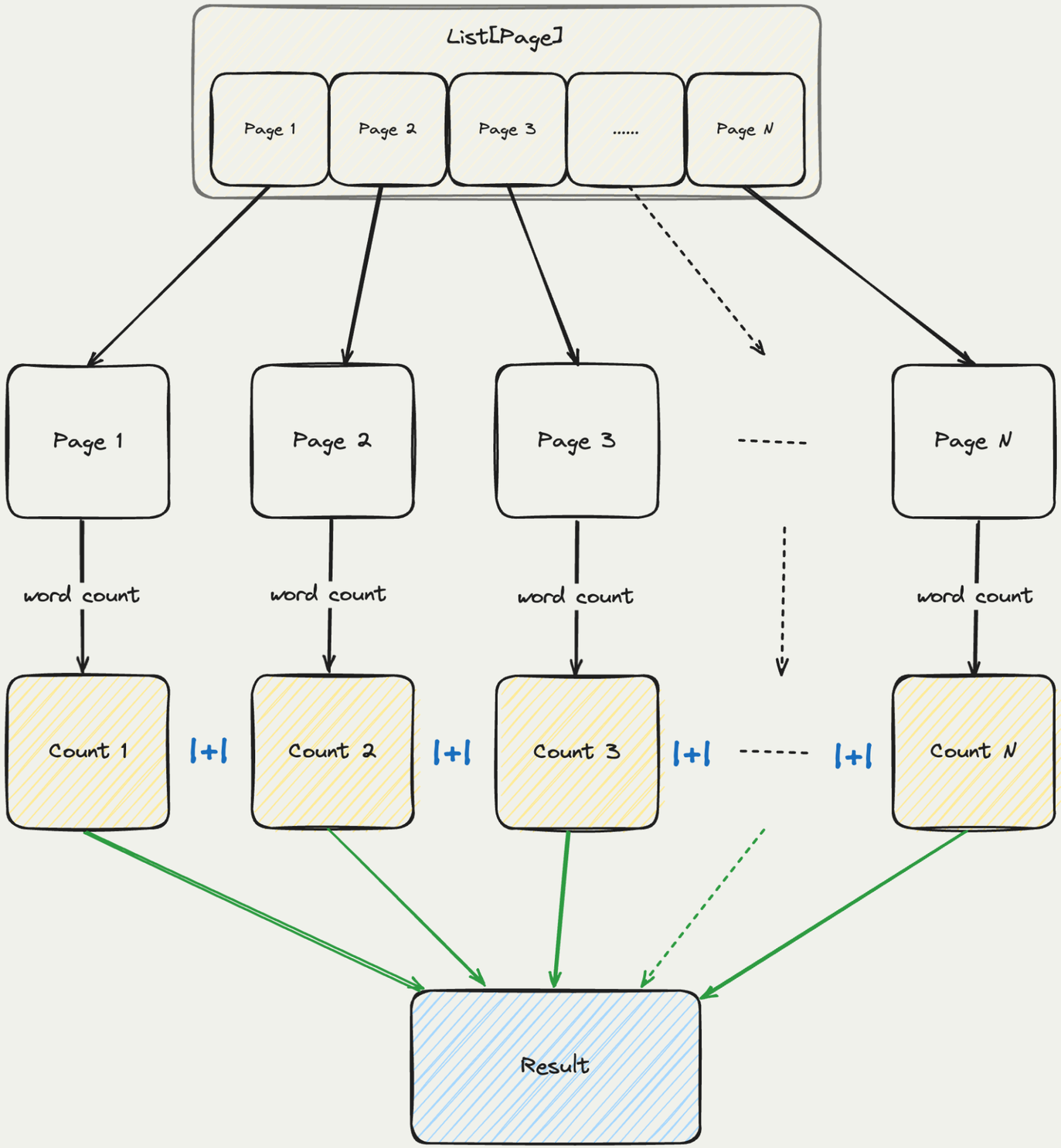
```
1  1 + 2 + 3
2
3  "I" + "love" + "fp"
4
5  List(1, 2, 3) ++ List(4, 5, 6)
```

```
1  trait Semigroup[A] {
2      def combine(x: A, y: A): A
3  }
```

Посмотрим на примеры кода с Semigroup

Semigroup laws

Associativity	`(A |+| B) |+| C = A |+| (B |+| C)`


List[Page]
Page 1
Page 2
Page 3
.....
Page N
Page 1
Page 2
Page 3

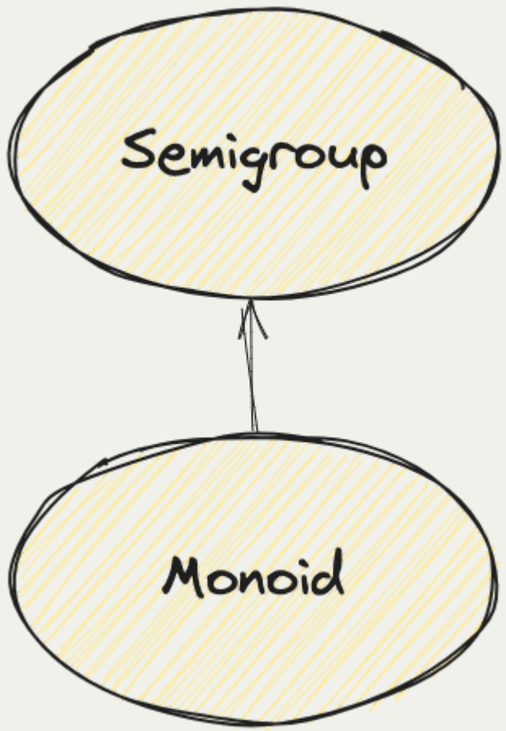
Page N
word count
word count
word count
word count
Count 1
|+|
Count 2
|+|
Count 3
|+|

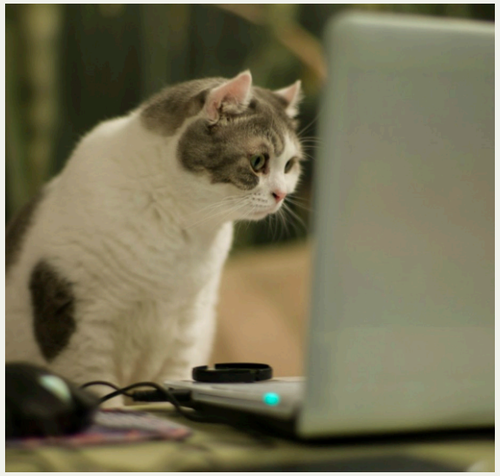
|+|
Count N
Result

```scala
def combineAll[A: Semigroup](list: List[A]): A = {
  ???
}
```

Monoid

```scala
trait Monoid[A] extends Semigroup[A] {
    def empty: A
}
```


Semigroup
Monoid



Посмотрим на примеры кода с Monoid

Monoid laws

Associativity	`(A |+| B) |+| C = A |+| (B |+| C)`
Left identity	`Monoid[A].empty |+| x = x`
Right identity	`x |+| Monoid[A].empty = x`

Вопросы?

Functor

Functor

```
1 Vector(1, 2, 3).map(_ + 1) // result: Vector(2, 3, 4)
2
3 List(1, 2, 3).map(_ + 1) // result: List(2, 3, 4)
4
5 Some(1).map(_ + 1) // result: Some(2)
```

Functor

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

```
Vector(1, 2, 3).map(_ + 1) // result: Vector(2, 3, 4)

List(1, 2, 3).map(_ + 1) // result: List(2, 3, 4)

Some(1).map(_ + 1) // result: Some(2)
```

Functor

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

```
Vector(1, 2, 3).map(_ + 1) // result: Vector(2, 3, 4)

List(1, 2, 3).map(_ + 1) // result: List(2, 3, 4)

Some(1).map(_ + 1) // result: Some(2)
```

Что такое F[_]?

Higher-Kinded Types

`List[A]`

`Seq[A]`

`Option[A]`

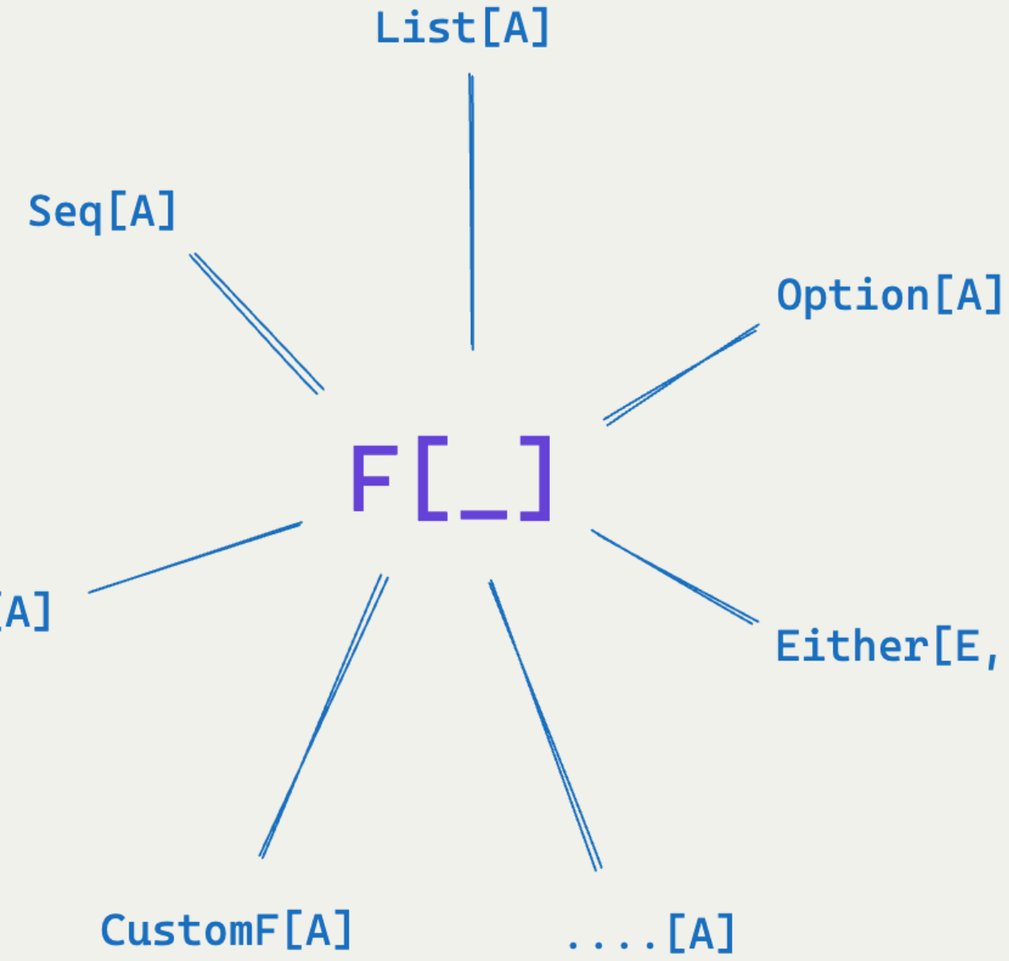
`Try[A]`

`Either[E, A]`

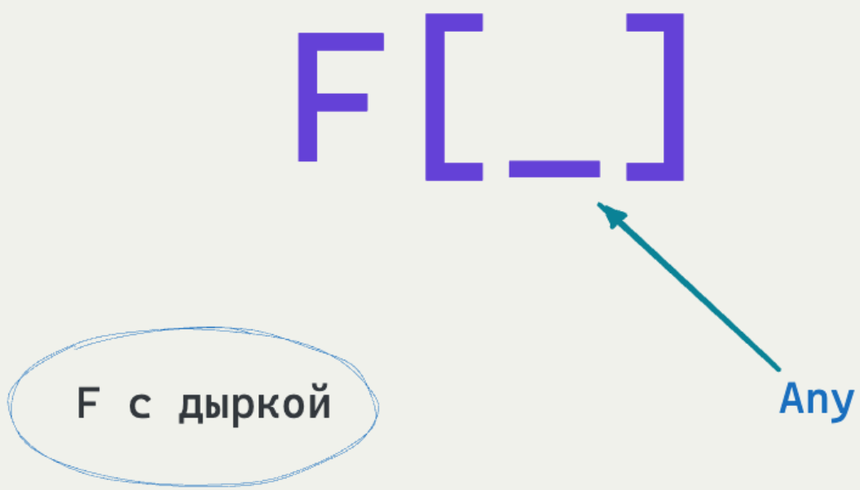
`CustomF[A]`

`....[A]`

Higher-Kinded Types



Higher-Kinded Types

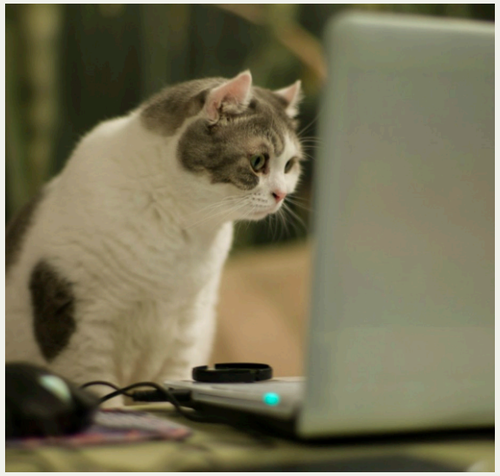


Functor

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

Вернемся к функтору

Functor	Method	From	Given	To
map	**F[A]**	A => B		**F[B]**



Посмотрим на примеры кода с Functor

Functor laws	
Identity	`fa.map(a => a) == fa`
	`fa.map(identity) == fa`
Composition	`fa.map(g(f(_))) == fa.map(f).map(g)`

Вопросы?

Foldable

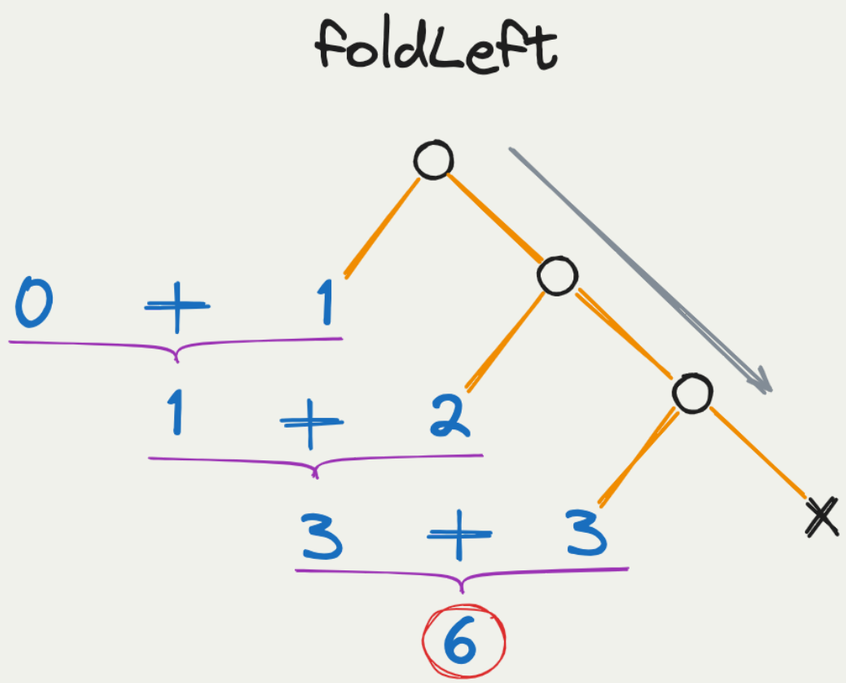
```scala
Vector(1, 2, 3).foldLeft(0)(_ + _)

List(1, 2, 3).foldLeft(0)(_ + _)

Some(1).foldLeft(0)(_ + _)
```

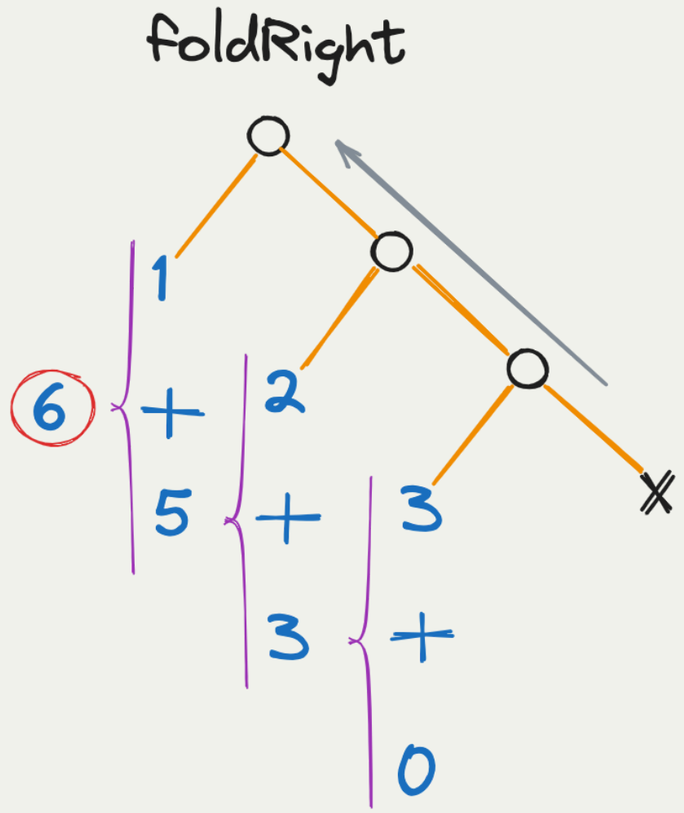
Foldable

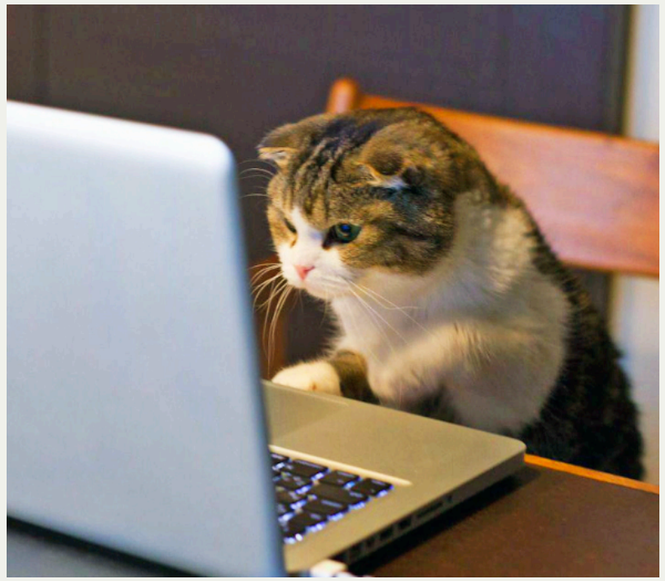
```scala
trait Foldable[F[_]] {
    def foldLeft[A, B](fa: F[A], b: B)(f: (B, A) => B): B
}
```

```scala
Vector(1, 2, 3).foldLeft(0)(_ + _)

List(1, 2, 3).foldLeft(0)(_ + _)

Some(1).foldLeft(0)(_ + _)
```

Foldable







Посмотрим на примеры кода с Foldable

Вопросы?

Applicative

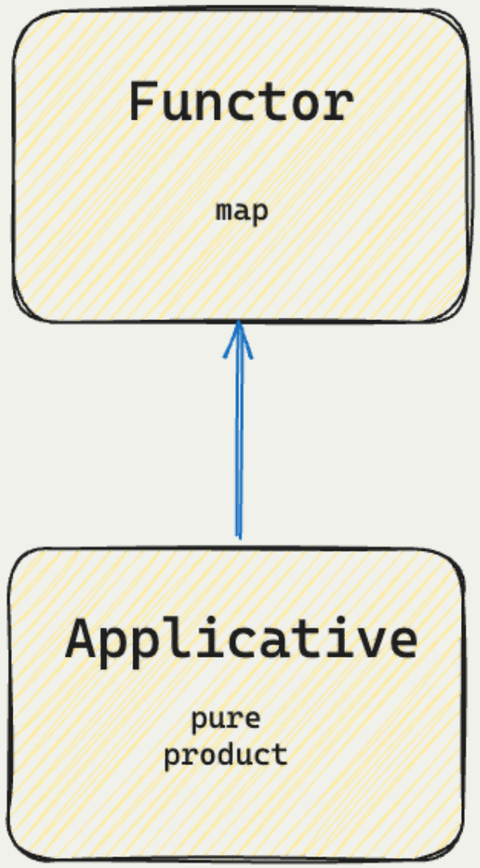
Applicative

(сейчас начнется сложная часть лекции!)

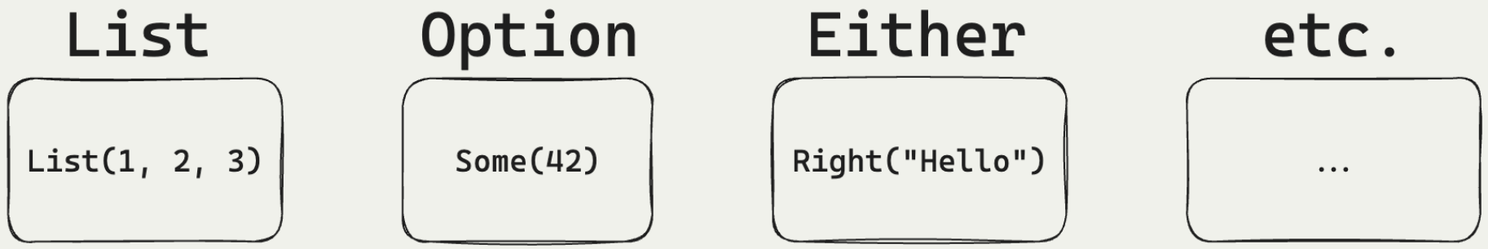
Applicative

```scala
// Аппликативный функтор
trait Applicative[F[_]] extends Functor[F] {
  def pure[A](a: A): F[A]
  def product[A, B](fa: F[A], fb: F[B]): F[(A, B)]
}
```

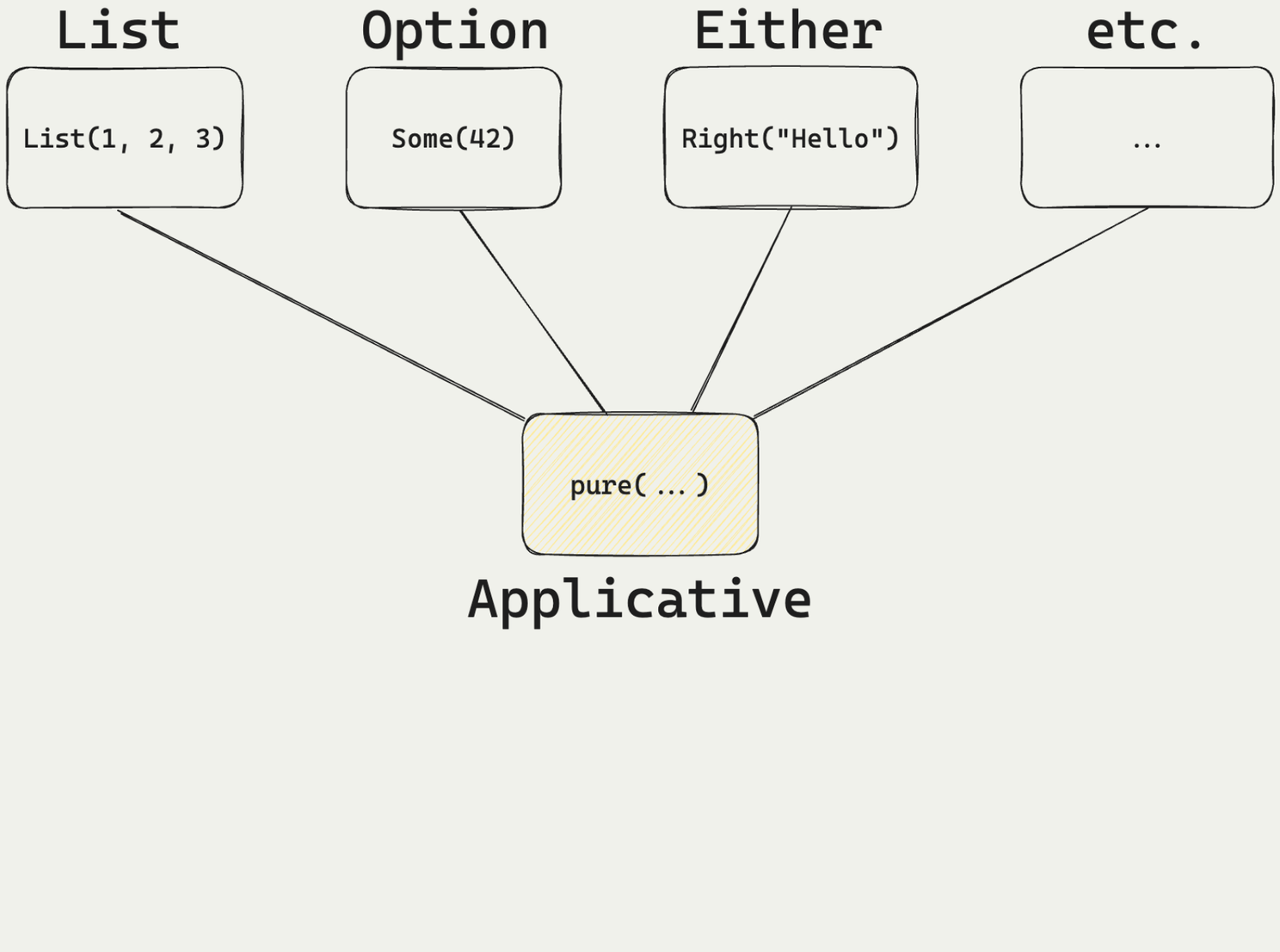
Аппликативный функтор



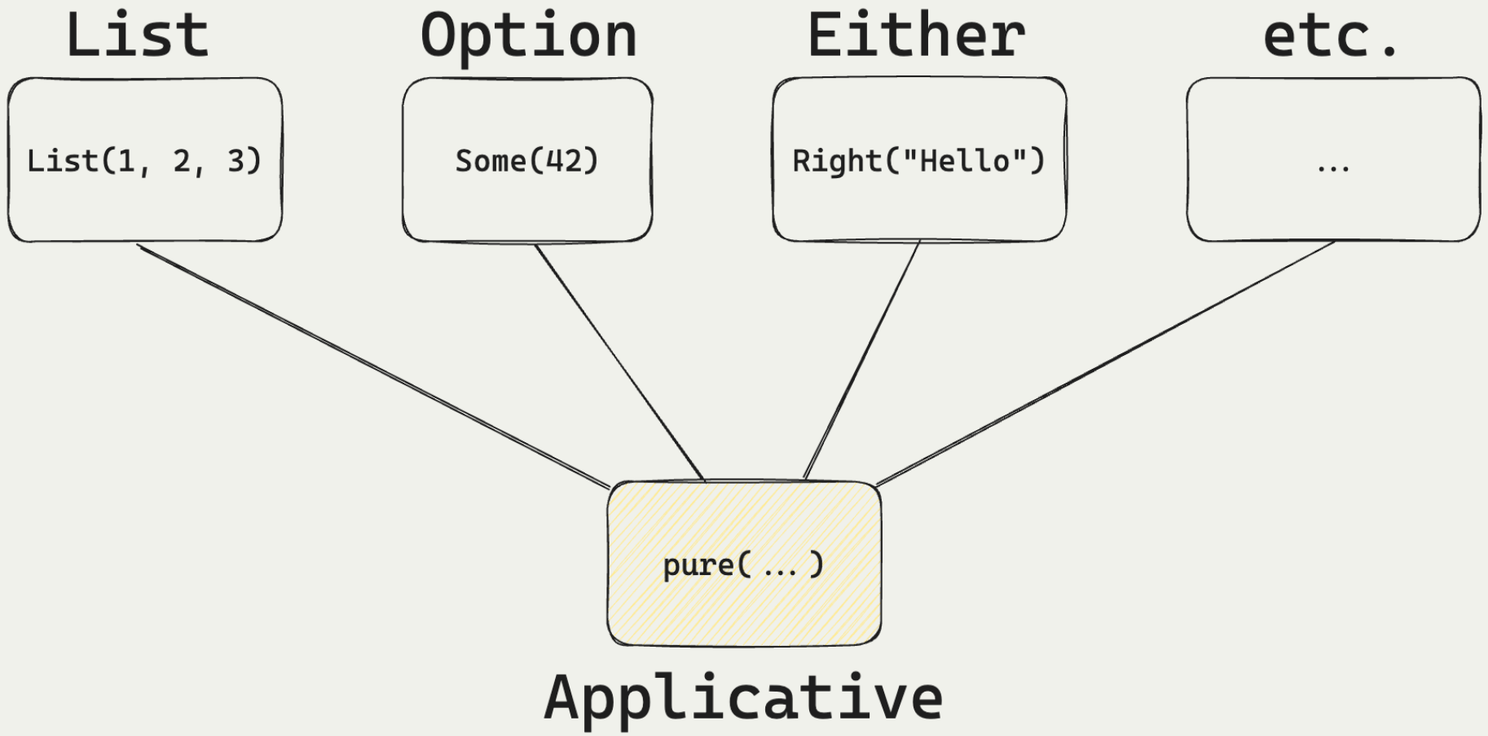
Зачем нужен pure?



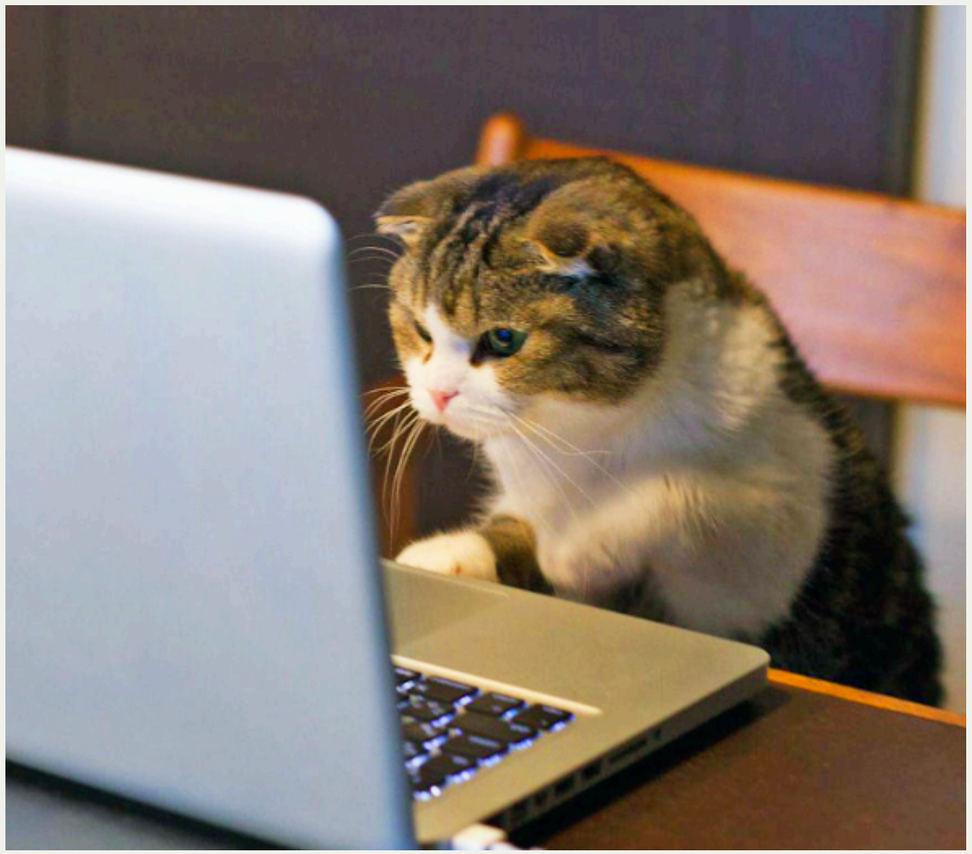
Зачем нужен pure?



Зачем нужен pure?




```
Applicative[Option].pure(42)  // Some(42)
Applicative[List].pure(42)    // List(42)
```



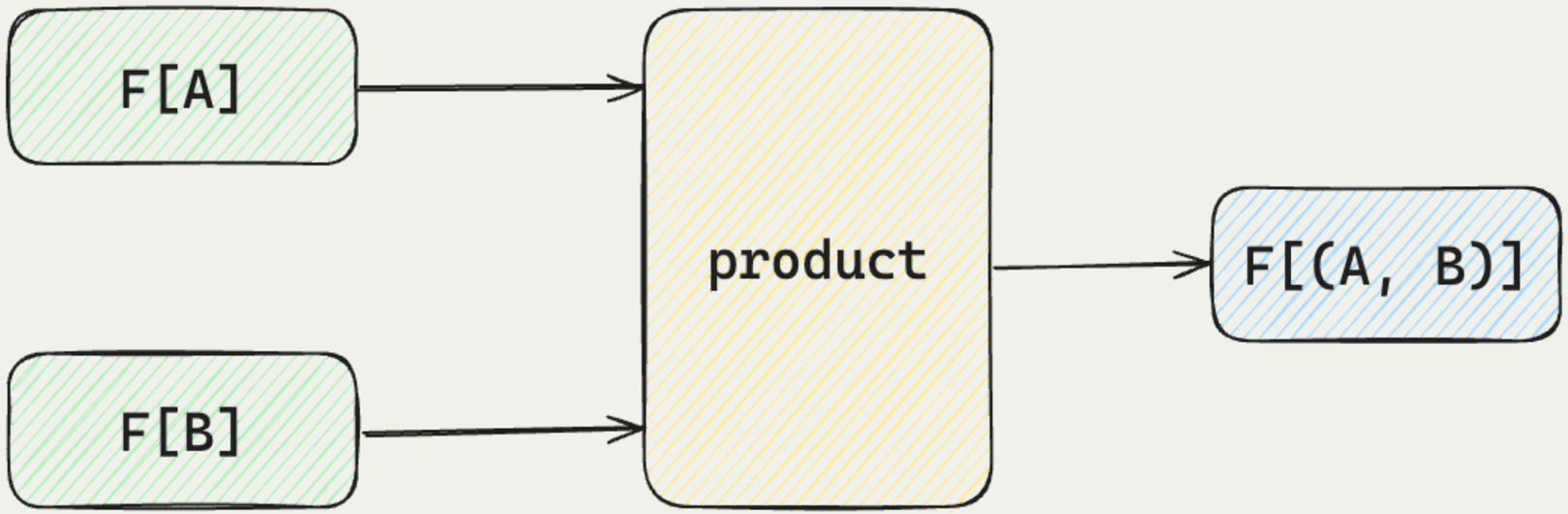
Посмотрим на примеры кода с Applicative.pure

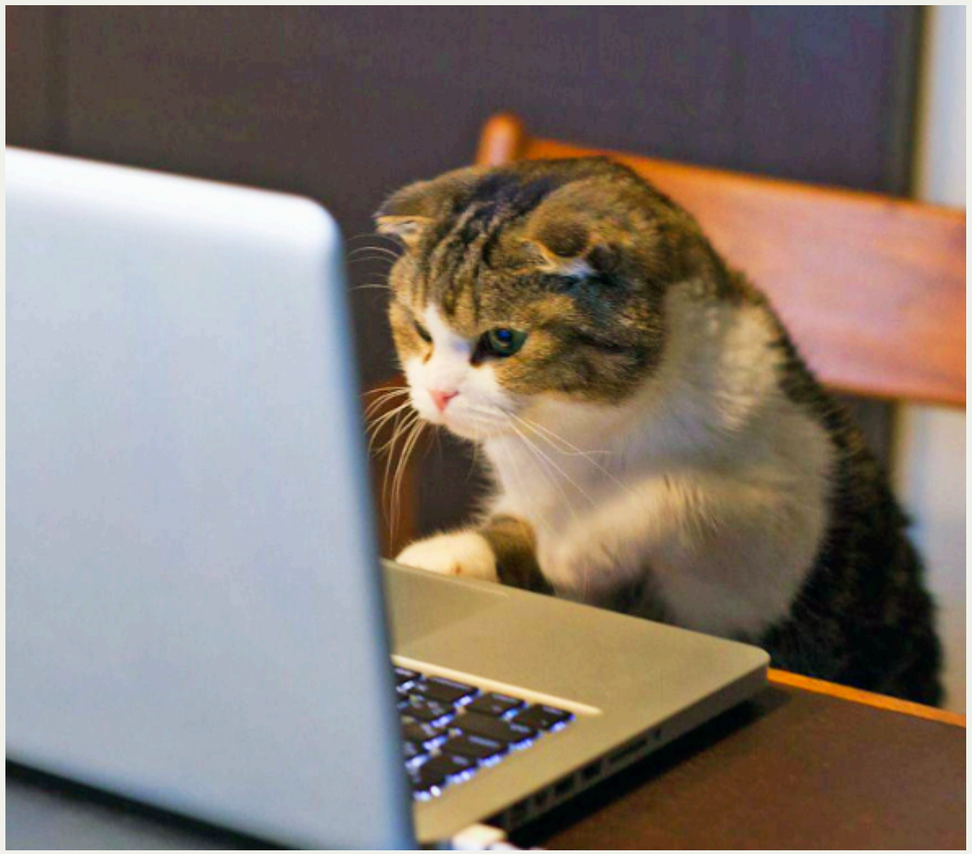
Applicative

```scala
trait Applicative[F[_]] extends Functor[F] {
  def pure[A](a: A): F[A]
  def product[A, B](fa: F[A], fb: F[B]): F[(A, B)]
}
```

Applicative	Method	From	Given	To
	`pure`	`A`		`F[A]`
	`product`	`(F[A], F[B])`		`F[(A, B)]`

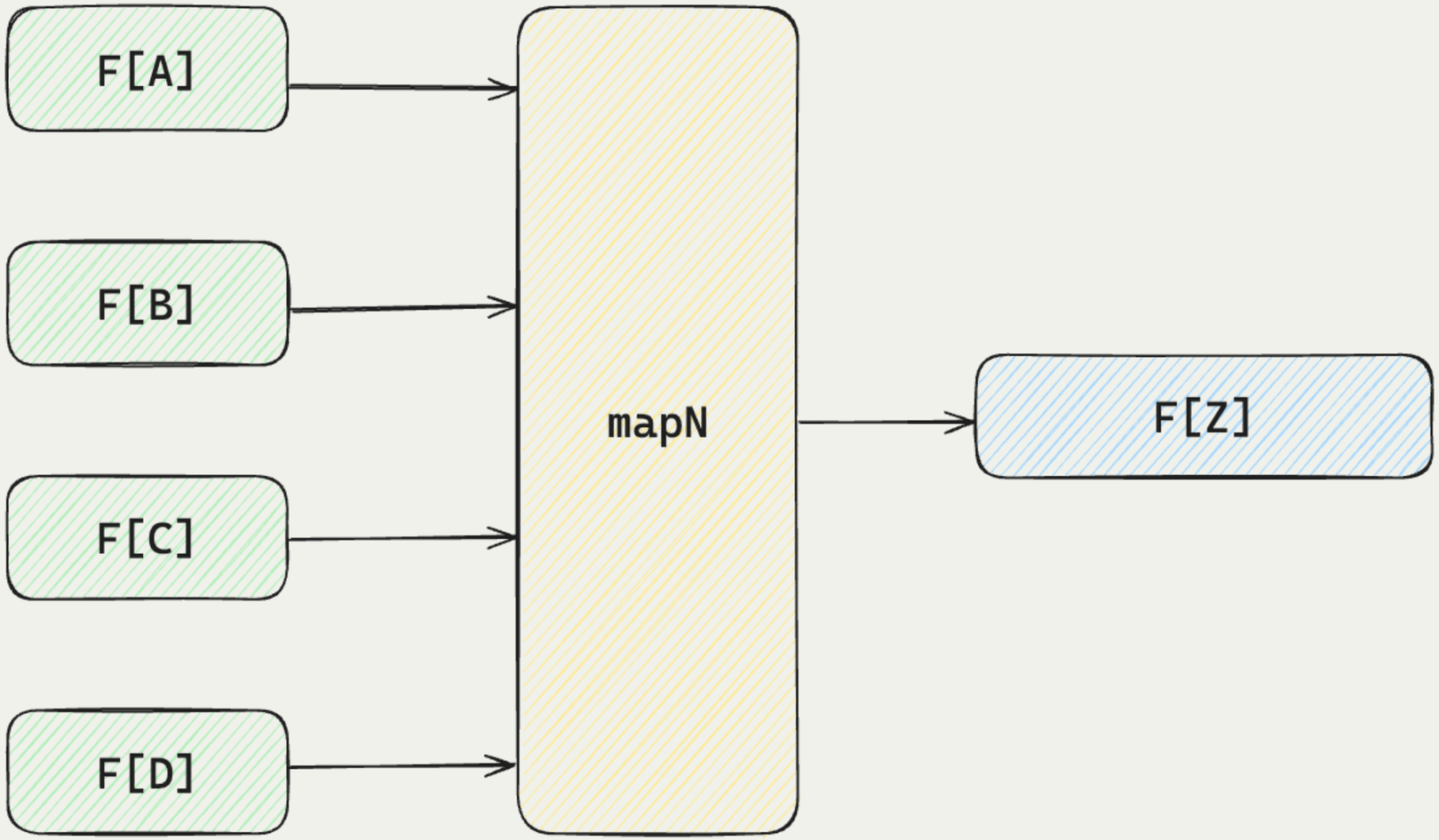
Независимые вычисления

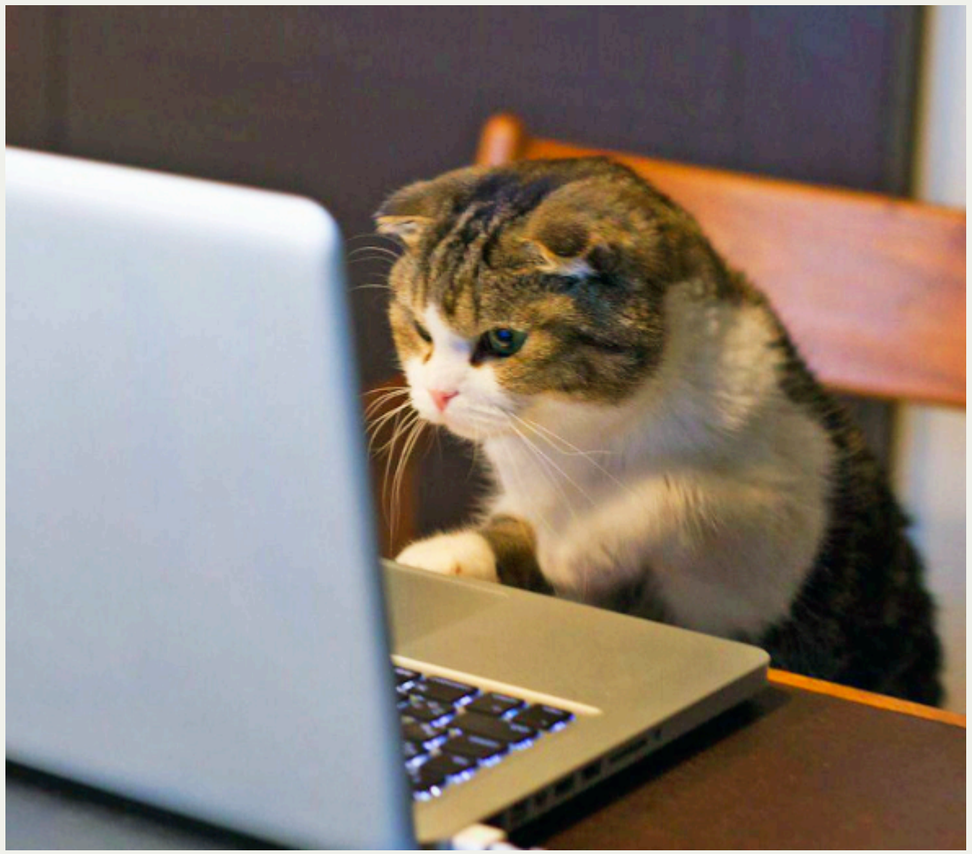




Посмотрим на примеры кода с Applicative.product

Независимые вычисления





Посмотрим на примеры кода с Applicative.mapN

Applicative laws

Associativity

```
fa.product(fb).product(fc) ~ fa.product(fb.product(fc))
```

Right identity

```
fa.product(().pure) ~ fa
```

Left identity

```
().pure.product(fa) ~ fa
```

Вопросы?

Traverse

Traverse

```scala
trait Traverse[F[_]] {
    def traverse[G[_]: Applicative, A, B](fa: F[A])(f: A => G[B]): G[F[B]]
}
```

Traverse	Method	From	Given	To
`traverse`	`F[A]`	`A => G[B]`		`G[F[A]]`

Посмотрим на примеры кода с Traverse

Вопросы?

ApplicativeError

ApplicativeError

```scala
trait ApplicativeError[F[_], E] extends Applicative[F] {

    def raiseError[A](e: E): F[A]

    def handleErrorWith[A](fa: F[A])(f: E => F[A]): F[A]
}
```

Посмотрим на примеры кода с ApplicativeError

Validation

Fail fast

Fail slow

Вопросы?

Laws, laws, laws

COMMUTATIVE

$A \cup B = B \cup A$

$A + B = B + A$

ASSOCIATIVE

$(A \cup B) \cup C = A \cup (B \cup C)$

$(A + B) + C = A + (B + C)$

DISTRIBUTIVE

$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

$A \times (B + C) = A \times B + A \times C$

IDENTITY

$A \cup \emptyset = A$

$A + 0 = A$

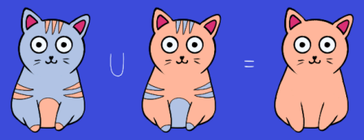
ABSORPTION

$A \cup (A \cap B) = A$

$A \cap (A \cup B) = A$

COMPLEMENT

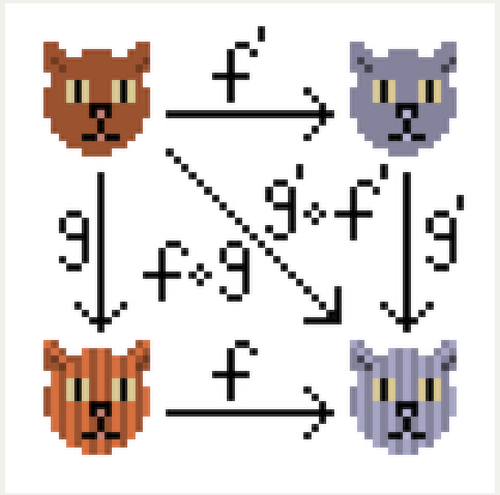
$A \cup \bar{A} = U$



IDEMPOTENT

$A \cup A = A$

Scala with Cats



Cats (categories) is a library which provides abstractions for functional programming in the Scala programming language

- https://typelevel.org/cats/
- https://www.scalawithcats.com/
- https://eed3si9n.com/herding-cats/

Спасибо за внимание!