eed3sign

Scalaz cheat sheet

in

Equal[A]

```
def equal(a1: A, a2: A): Boolean
(1 === 2) assert_=== false
(2 =/= 1) assert_=== true
```

Order[A]

```
def order(x: A, y: A): Ordering
1.0 ?|? 2.0 assert_=== Ordering.LT
1.0 lt 2.0 assert_=== true
1.0 gt 2.0 assert_=== false
1.0 lte 2.0 assert_=== true
1.0 gte 2.0 assert_=== false
1.0 max 2.0 assert_=== 2.0
1.0 min 2.0 assert_=== 1.0
```

Show[A]

```
def show(f: A): Cord
1.0.show assert_=== Cord("1.0")
1.0.shows assert_=== "1.0"
1.0.print assert_=== ()
1.0.println assert_=== ()
```

Enum[A] extends Order[A]

```
def pred(a: A): A
def succ(a: A): A
1.0 |-> 2.0 assert_=== List(1.0, 2.0)
1.0 |--> (2, 5) assert_=== List(1.0, 3.0, 5.0)
// |=>/|==>/from/fromStep return EphemeralStream[A]
(1.0 |=> 2.0).toList assert_=== List(1.0, 2.0)
(1.0 |==> (2, 5)).toList assert_=== List(1.0, 3.0, 5.
(1.0.from take 2).toList assert_=== List(1.0, 2.0)
((1.0 fromStep 2) take 2).toList assert_=== List(1.0,
1.0.pred assert_=== 0.0
1.0.predx assert_=== Some(0.0)
1.0.succ assert_=== 2.0
1.0.succx assert_=== Some(2.0)
1.0 -+- 1 assert_=== 2.0
1.0 --- 1 assert_=== 0.0
Enum[Int].min assert_=== Some(-2147483648)
Enum[Int].max assert_=== Some(2147483647)
```

Semigroup[A]

```
def append(a1: A, a2: => A): A
List(1, 2) |+| List(3) assert_=== List(1, 2, 3)
List(1, 2) mappend List(3) assert_=== List(1, 2, 3)
1 |+| 2 assert_=== 3
(Tags.Multiplication(2) |+| Tags.Multiplication(3): I
// Tags.Disjunction (||), Tags.Conjunction (&&)
(Tags.Disjunction(true) |+| Tags.Disjunction(false):
(Tags.Conjunction(true) |+| Tags.Conjunction(false):
(Ordering.LT: Ordering) |+| (Ordering.GT: Ordering) a
(none: Option[String]) |+| "andy".some assert_=== "an
(Tags.First('a'.some) |+| Tags.First('b'.some): Optic
(Tags.Last('a'.some) |+| Tags.Last(none: Option[Char]
```

Monoid[A] extends Semigroup[A]

```
def zero: A
mzero[List[Int]] assert_=== Nil
```

Functor[F[_]]

```
def map[A, B](fa: F[A])(f: A => B): F[B]
List(1, 2, 3) map {_ + 1} assert_=== List(2, 3, 4)
List(1, 2, 3) ∘ {_ + 1} assert_=== List(2, 3, 4)
List(1, 2, 3) >| "x" assert_=== List("x", "x", "x")
List(1, 2, 3) as "x" assert_=== List("x", "x", "x")
List(1, 2, 3).fpair assert_=== List((1,1), (2,2), (3,
List(1, 2, 3).strengthL("x") assert_=== List(("x",1),
List(1, 2, 3).strengthR("x") assert_=== List((1,"x"),
List(1, 2, 3).void assert_=== List((), (), ())
Functor[List].lift {(_: Int) * 2} (List(1, 2, 3)) ass
```

Boolean

```
false /\ true assert_=== false // &&
false \/ true assert_=== true // ||
(1 < 10) option 1 assert_=== 1.some
(1 > 10)? 1 | 2 assert_=== 2
(1 > 10)?? {List(1)} assert_=== Nil
```

Option

```
1.some assert_=== Some(1)
none[Int] assert_=== (None: Option[Int])
1.some? 'x' | 'y' assert_=== 'x'
1.some | 2 assert_=== 1 // getOrElse
```

Id[+A] = A

```
// no contract function
1 + 2 + 3 |> {_ * 6}
1 visit { case x@(2|3) => List(x * 2) }
```

Tagged[A]

```
sealed trait KiloGram
def KiloGram[A](a: A): A @@ KiloGram = Tag[A, KiloGra
def f[A](mass: A @@ KiloGram): A @@ KiloGram
```

Tree[A]/TreeLoc[A]

```
val tree = 'A'.node('B'.leaf, 'C'.node('D'.leaf), 'E'
(tree.loc.getChild(2) >>= {_.getChild(1)} >>= {_.getL
(tree.loc.getChild(2) map {_.modifyLabel({_ => 'Z'})]
```

Stream[A]/Zipper[A]

```
(Stream(1, 2, 3, 4).toZipper >>= {_.next} >>= {_.focu
(Stream(1, 2, 3, 4).zipperEnd >>= {_.previous} >>= {_
(for { z <- Stream(1, 2, 3, 4).toZipper; n1 <- z.next
unfold(3) { x => (x =/= 0) option (x, x - 1) }.toList
```

DList[A]

```
DList.unfoldr(3, { (x: Int) => (x =/= 0) option (x, x
```

Lens[A, B] = LensT[Id, A, B]

```
val t0 = Turtle(Point(0.0, 0.0), 0.0)
val t1 = Turtle(Point(1.0, 0.0), 0.0)
val turtlePosition = Lens.lensu[Turtle, Point] (
  (a, value) => a.copy(position = value),
  _.position)
val pointX = Lens.lensu[Point, Double] (
  (a, value) => a.copy(x = value),
  _.x)
val turtleX = turtlePosition >=> pointX
turtleX.get(t0) assert_=== 0.0
turtleX.set(t0, 5.0) assert_=== Turtle(Point(5.0, 0.0
turtleX.mod(_ + 1.0, t0) assert_=== t1
t0 |> (turtleX =>= {_ + 1.0}) assert_=== t1
(for { x <- turtleX %= {_ + 1.0} } yield x) exec t0 a
(for { x <- turtleX := 5.0 } yield x) exec t0 assert_
(for { x <- turtleX += 1.0 } yield x) exec t0 assert_
```

Validation[+E, +A]

```
(1.success[String] |@| "boom".failure[Int] |@| "boom"
(1.successNel[String] |@| "boom".failureNel[Int] |@|
"1".parseInt.toOption assert_=== 1.some
```

Writer[+W, +A] = WriterT[Id, W, A]

Pointed[F[_]] extends Functor[F]

```
def point[A](a: => A): F[A]
1.point[List] assert_=== List(1)
1.η[List] assert_=== List(1)
```

Apply[F[_]] extends Functor[F]

```
def ap[A,B](fa: => F[A])(f: => F[A => B]): F[B]
1.some <*> {(_: Int) + 2}.some assert_=== Some(3) //
1.some <*> { 2.some <*> {(_: Int) + (_: Int)}.curriec
1.some <* 2.some assert_=== 1.some
1.some *> 2.some assert_=== 2.some
Apply[Option].ap(9.some) {{(_: Int) + 3}.some} assert
Apply[List].lift2 {(_: Int) * (_: Int)} (List(1, 2),
(3.some |@| 5.some) {_ + _} assert_=== 8.some
// ^(3.some, 5.some) {_ + _} assert_=== 8.some
```

Applicative[F[_]] extends Apply[F] with Pointed[F]

```
// no contract function
```

Product/Composition

```
(Applicative[Option] product Applicative[List]).point
(Applicative[Option] compose Applicative[List]).point
```

Bind[F[_]] extends Apply[F]

```
def bind[A, B](fa: F[A])(f: A => F[B]): F[B]
3.some flatMap { x => (x + 1).some } assert_=== 4.som
(3.some >>= { x => (x + 1).some }) assert_=== 4.some
3.some >> 4.some assert_=== 4.some
List(List(1, 2), List(3, 4)).join assert_=== List(1,
```

Monad[F[_]] extends Applicative[F] with Bind[F]

```
// no contract function
// failed pattern matching produces None
(for {(x :: xs) <- "".toList.some} yield x) assert_==
(for { n <- List(1, 2); ch <- List('a', 'b') } yield
(for { a <- (_: Int) * 2; b <- (_: Int) + 10 } yield
List(1, 2) filterM { x => List(true, false) } assert_
```

Plus[F[_]]

```
def plus[A](a: F[A], b: => F[A]): F[A]
List(1, 2) <+> List(3, 4) assert_=== List(1, 2, 3, 4]
```

PlusEmpty[F[_]] extends Plus[F]

```
def empty[A]: F[A]
(PlusEmpty[List].empty: List[Int]) assert_=== Nil
```

ApplicativePlus[F[_]] extends Applicative[F] with PlusEmpty[F]

```
// no contract function
```

MonadPlus[F[_]] extends Monad[F] with ApplicativePlus[F]

```
// no contract function
List(1, 2, 3) filter {_ > 2} assert_=== List(3)
```

Foldable[F[_]]

```
def foldMap[A,B](fa: F[A])(f: A => B)(implicit F: Mor
def foldRight[A, B](fa: F[A], z: => B)(f: (A, => B) =
List(1, 2, 3).foldRight (0) {_ + _} assert_=== 6
List(1, 2, 3).foldLeft (0) {_ + _} assert_=== 6
(List(1, 2, 3) foldMap {Tags.Multiplication}: Int) as
List(1, 2, 3).foldLeftM(0) { (acc, x) => (acc + x).sc
```

Traverse[F[_]] extends Functor[F] with Foldable[F]

```
def traverseImpl[G[_]:Applicative,A,B](fa: F[A])(f: A
List(1, 2, 3) traverse { x => (x > 0) option (x + 1)
List(1, 2, 3) traverseU {_ + 1} assert_=== 9
```

```
(for { x <- 1.set("log1"); _ <- "log2".tell } yield (
import std.vector._
MonadWriter[Writer, Vector[String]].point(1).run asse
```

\/[+A, +B]

```
1.right[String].isRight assert_=== true
1.right[String].isLeft assert_=== false
1.right[String] | 0 assert_=== 1  // getOrElse
("boom".left ||| 2.right) assert_=== 2.right // orEls
("boom".left[Int] >>= { x => (x + 1).right }) assert_
(for { e1 <- 1.right; e2 <- "boom".left[Int] } yield
```

Kleisli[M[+_], -A, +B]

```
val k1 = Kleisli { (x: Int) => (x + 1).some }
val k2 = Kleisli { (x: Int) => (x * 100).some }
(4.some >>= k1 compose k2) assert_=== 401.some
(4.some >>= k1 <=< k2) assert_=== 401.some
(4.some >>= k1 andThen k2) assert_=== 500.some
(4.some >>= k1 >=> k2) assert_=== 500.some
```

Reader[E, A] = Kleisli[Id, E, A]

```
Reader { (_: Int) + 1 }
```

trait Memo[K, V]

```
val memoizedFib: Int => Int = Memo.mutableHashMapMemo
  case 0 => 0
  case 1 => 1
  case n => memoizedFib(n - 2) + memoizedFib(n - 1)
}
```

State[S, +A] = StateT[Id, S, A]

```
State[List[Int], Int] { case x :: xs => (xs, x) }.rur
(for {
  xs <- get[List[Int]]
  _ <- put(xs.tail)
} yield xs.head).run(1 :: Nil) assert_=== (Nil, 1)
```

ST[S, A]/STRef[S, A]/STArray[S, A]

```
import scalaz._, Scalaz._, effect._, ST._
type ForallST[A] = Forall[({type l[x] = ST[x, A]})#l]
def e1[S]: ST[S, Int] = for {
  x <- newVar[S](0)
  _ <- x mod {_ + 1}
  r <- x.read
} yield r
runST(new ForallST[Int] { def apply[S] = e1[S] }) ass
def e2[S]: ST[S, ImmutableArray[Boolean]] = for {
  arr <- newArr[S, Boolean](3, true)
  x <- arr.read(0)
  _ <- arr.write(0, !x)
  r <- arr.freeze
} yield r
runST(new ForallST[ImmutableArray[Boolean]] { def app
```

IO[+A]

```
import scalaz._, Scalaz._, effect._, IO._
val action1 = for {
  x <- readLn
  _ <- putStrLn("Hello, " + x + "!")
} yield ()
action1.unsafePerformIO
```

IterateeT[E, F[_], A]/EnumeratorT[O, I, F[_]]

```
import scalaz._, Scalaz._, iteratee._, Iteratee._
(length[Int, Id] &= enumerate(Stream(1, 2, 3))).run a
(length[scalaz.effect.IoExceptionOr[Char], IO] &= enu
```

Free[S[+_], +A]

```
import scalaz._, Scalaz._, Free._
type FreeMonoid[A] = Free[({type λ[+α] = (A,α)})#λ, U
def cons[A](a: A): FreeMonoid[A] = Suspend[({type λ[+
def toList[A](list: FreeMonoid[A]): List[A] =
  list.resume.fold(
    { case (x: A, xs: FreeMonoid[A]) => x :: toList(x
    { _ => Nil })
toList(cons(1) >>= {_ => cons(2)}) assert_=== List(1,
```

```
List(1.some, 2.some).sequence assert_=== List(1, 2).s
1.success[String].leaf.sequenceU map {_.drawTree} ass
```

Length[F[_]]

```
def length[A](fa: F[A]): Int
List(1, 2, 3).length assert_=== 3
```

Index[F[_]]

```
def index[A](fa: F[A], i: Int): Option[A]
List(1, 2, 3) index 2 assert_=== 3.some
List(1, 2, 3) index 3 assert_=== none
```

ArrId[=>:[_, _]]

```
def id[A]: A =>: A
```

Compose[=>:[_, _]]

```
def compose[A, B, C](f: B =>: C, g: A =>: B): (A =>:
val f1 = (_:Int) + 1
val f2 = (_:Int) * 100
(f1 >>> f2)(2) assert_=== 300
(f1 <<< f2)(2) assert_=== 201
```

Category[=>:[_, _]] extends ArrId[=>:] with Compose[=>:]

```
// no contract function
```

Arrow[=>:[_, _]] extends Category[=>:]

```
def arr[A, B](f: A => B): A =>: B
def first[A, B, C](f: (A =>: B)): ((A, C) =>: (B, C))
val f1 = (_:Int) + 1
val f2 = (_:Int) * 100
(f1 *** f2)(1, 2) assert_=== (2, 200)
(f1 &&& f2)(1) assert_=== (2,100)
```

Unapply[TC[_[_]], MA]

```
type M[_]
type A
def TC: TC[M]
def apply(ma: MA): M[A]
implicitly[Unapply[Applicative, Int => Int]].TC.point
List(1, 2, 3) traverseU {(x: Int) => {(_:Int) + x}} a
```

Trampoline[+A] = Free[Function0, A]

```
import scalaz._, Scalaz._, Free._
def even[A](ns: List[A]): Trampoline[Boolean] =
  ns match {
    case Nil => return_(true)
    case x :: xs => suspend(odd(xs))
  }
def odd[A](ns: List[A]): Trampoline[Boolean] =
  ns match {
    case Nil => return_(false)
    case x :: xs => suspend(even(xs))
  }
even(0 |-> 3000).run assert_=== false
```

Imports

```
import scalaz._ // imports type names
import scalaz.Id.Id // imports Id type alias
import scalaz.std.option._ // imports instances, conv
import scalaz.std.AllInstances._ // imports instances
import scalaz.std.AllFunctions._ // imports functions
import scalaz.syntax.monad._ // injects operators to
import scalaz.syntax.all._ // injects operators to al
import scalaz.syntax.std.boolean._ // injects operato
import scalaz.syntax.std.all._ // injects operators t
import scalaz._, Scalaz._ // all the above
```

Note

```
type Function1Int[A] = ({type l[x]=Function1[Int, x]]
type Function1Int[A] = Function1[Int, A]
```