Functional data structures 2

Streams

```
val s = Stream(1, 2, 3, 4)      // = Stream(1, ?)

val s = 1 #:: 2 #:: 3 #:: 4 #:: Stream.empty    // = Stream(1, ?)
```

Streams

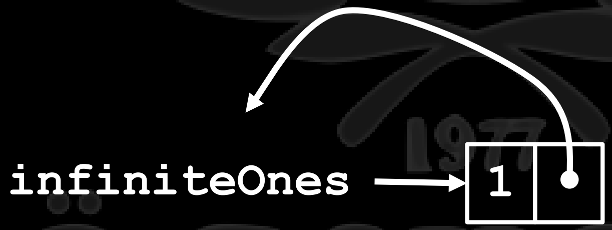
```
val s = Stream(1, 2, 3, 4)     // = Stream(1, ?)


val infiniteOnes: Stream[Int] = 1 #:: infiniteOnes // = Stream(1, ?)
```

Streams

```
val s = Stream(1, 2, 3, 4)     // = Stream(1, ?)

val infiniteOnes: Stream[Int] = 1 #:: infiniteOnes // = Stream(1, ?)
```



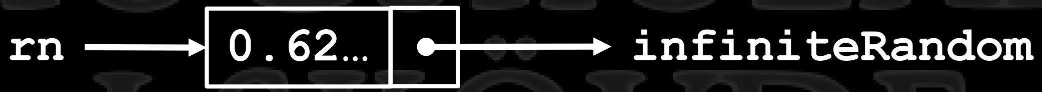
Streams

```
val s = Stream(1, 2, 3, 4)    // = Stream(1, ?)
s.size      // = 4
val infiniteOnes: Stream[Int] = 1 #:: infiniteOnes // = Stream(1, ?)
infiniteOnes.size // = ???
```

Streams: corecursion

```
def infiniteRandom: Stream[Double] = math.random #:: infiniteRandom
val rn = infiniteRandom // = Stream(0.6232137958520699, ?)
```

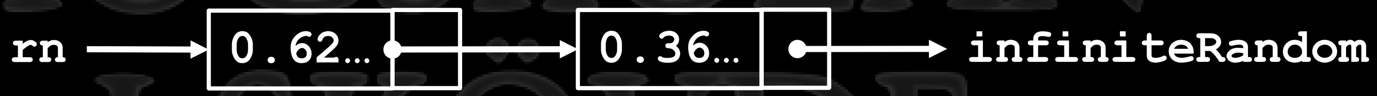
Co-recursive



Streams: memoization

```
def infiniteRandom: Stream[Double] = math.random #:: infiniteRandom
val rn = infiniteRandom // = Stream(0.6232137958520699, ?)
rn.head     // = 0.6232137958520699
rn.tail.head      // = 0.365469070811272
rn    // = Stream(0.6232137958520699, 0.365469070811272, ?)
```

Memoization



Streams: built-in functions

```
def infiniteRandom: Stream[Double] = math.random #:: infiniteRandom

val rn = infiniteRandom // = Stream(0.6232137958520699, ?)

rn.map(_ * 10)    // = ???

rn.filter(_ > .5) // = ???

rn.forAll(_ > 0.001)    // = ???
```

Stream vs. List

```
sealed abstract class MyList { def head: Int; def tail: MyList }
  case class NonEmpty(head: Int, tail: MyList) extends MyList



  case object Empty extends MyList {
         def head: Int = throw new Exception("head of empty list")
         def tail: MyList = throw new Exception("tail of empty list")
       }
```

Stream vs. List

```
sealed abstract class MyStream { def head: Int; def tail: MyStream }
  case class NonEmpty(head: Int, tail: MyStream) extends MyStream




  case object Empty extends MyStream {
         def head: Int = throw new Exception("head of empty stream")
         def tail: MyStream = throw new Exception("tail of empty stream")
       }
```

Stream vs. List

```scala
sealed abstract class MyStream { def head: Int; def tail: MyStream }
  case class NonEmpty(head: Int, tail: => MyStream) extends MyStream
```

Stream vs. List

```
sealed abstract class MyStream { def head: Int; def tail: MyStream }
class NonEmpty(h: Int, t: => MyStream) extends MyStream {
  ??? head: Int = h
  ??? tail: MyStream = t
}
```

Stream vs. List

```
sealed abstract class MyStream { def head: Int; def tail: MyStream }
class NonEmpty(h: Int, t: => MyStream) extends MyStream {
  val head: Int = h
  lazy val tail: MyStream = t
}
```

Stream vs. List

```
val s = new NonEmpty(1,
          new NonEmpty(2,
            new NonEmpty(3, Empty)))


val seq = infiniteSeq(1)
seq.head    // = 1
seq.tail.head     // = 2
seq.tail.tail.head      // = 3
```

Stream vs. List

```scala
def infiniteSeq(from: Int): MyStream =
  new NonEmpty(from, infiniteSeq(from + 1))


val seq = infiniteSeq(1)
seq.head    // = 1
seq.tail.head     // = 2
seq.tail.tail.head      // = 3
```

Universal polymorphism: functions

```
def filter(c: List[Int], include: Int => Boolean): List[Int] =
  c match {




  }
```

Universal polymorphism: functions

```scala
def filter(c: List[Int], include: Int => Boolean): List[Int] =
  c match {
    case Nil => Nil
    case h :: t if include(h) => h :: filter(t, include)
    case _ :: t => filter(t, include)
  }
```

Universal polymorphism: functions

```
def filter(c: List[ A ], include: A   => Boolean): List[A] =
  c match {
    case Nil => Nil
    case h :: t if include(h) => h :: filter(t, include)
    case h :: t => filter(t, include)
  }
```

Universal polymorphism: on functions

```scala
def filter[A](c: List[A], include: A => Boolean): List[A] =
  c match {
    case Nil => Nil
    case h :: t if include(h) => h :: filter(t, include)
    case h :: t => filter(t, include)
  }
filter(List(1, 2, 3, 4), (x: Int) => x % 2 == 0)
filter(List("Hello", "there", "Scala"), (x: String) => x == "Scala")
```

Universal polymorphism: on functions

```
def filter[A](c: List[A], include: A => Boolean): List[A] =
  c match {
    case Nil => Nil
    case h :: t if include(h) => h :: filter(t, include)
    case h :: t => filter(t, include)
  }
filter[Int](List(1, 2, 3, 4), x => x % 2 == 0)
filter[String](List("Hello", "there", "Scala"), x => x == "Scala")
```

Universal polymorphism: on functions

```
def map(c: List[String], f: String => Int): List[Int] =
  c match {
    case Nil => Nil
    case h :: t => f(h) :: map(t, f)
  }
```

Universal polymorphism: on functions

```
def map[ ?? ](c: List[?], f: ? => ?): List[?] =
  c match {
    case Nil => Nil
    case h :: t => f(h) :: map(t, f)
  }
```

Universal polymorphism: on functions

```
def map[A, B](c: List[A], f: A => B): List[B] =
  c match {
    case Nil => Nil
    case h :: t => f(h) :: map(t, f)
  }


map[    ,    ](List("Hello", "there", "Scala"), _.size)
map[    ,    ](List(1, 2, 3), _ * 2)      // = List(2, 4, 6)
```

Universal polymorphism: classes

```scala
sealed abstract class MyList { def head: Int; def tail: MyList }


case class NonEmpty(head: Int, tail: MyList[Int]) extends MyList


object Empty extends MyList {
  def head: Int = throw new Exception("Empty has no head")
  def tail: MyList = throw new Exception("Empty has no tail")
}
```

Universal polymorphism: classes

```
sealed abstract class MyList { def head:  A ; def tail: MyList }

case class NonEmpty(head:  A , tail: MyList[ A ]) extends MyList

object Empty extends MyList {
  def head:  A  = throw new Exception("Empty has no head")
  def tail: MyList = throw new Exception("Empty has no tail")
}
```

Universal polymorphism: classes

```
sealed abstract class MyList[A] { def head: A; def tail: MyList[A] }

case class NonEmpty[A](head: A, tail: MyList[A]) extends MyList[A]

object Empty[A] extends MyList[A] {
  def head: A = throw new Exception("Empty has no head")
  def tail: MyList[A] = throw new Exception("Empty has no tail")
}
```

This won't compile but almost there

Universal polymorphism: classes

```
sealed abstract class MyList[A] { def head: A; def tail: MyList[A] }

case class NonEmpty[A](head: A, tail: MyList[A]) extends MyList[A]

case class Empty[A]() extends MyList[A] {
  def head: A = throw new Exception("Empty has no head")
  def tail: A = throw new Exception("Empty has no tail")
}
```

Universal polymorphism: classes

```
val ns = NonEmpty[Int](1,
           NonEmpty[Int](2,
             NonEmpty[Int](3,
               Empty[Int]())))
```

Universal polymorphism: classes

```
val ns = NonEmpty[String]("a",
           NonEmpty[String]("b",
             NonEmpty[String]("c",
               Empty[String]())))
```

Universal polymorphism: classes

```
val ns = NonEmpty[(Int, String)]((1, "a"),
           NonEmpty[(Int, String)]((2, "b"),
             NonEmpty[(Int, String)]((3, "c"),
               Empty[(Int, String)]())))
```

Universal polymorphism: classes

```
val ns = NonEmpty((1, "a"),
           NonEmpty ((2, "b"),
             NonEmpty ((3, "c"),
               Empty())))


// ns: NonEmpty[(Int, String)] = ...
```

Universal polymorphism: Option

```
sealed abstract class Maybe

case class Yes (value: Int) extends Maybe

case object No extends Maybe
```

Universal polymorphism: Option

```
sealed abstract class Maybe[A]

case class Yes[A](value: A) extends Maybe[A]

case class No[A]() extends Maybe[A]
```

Universal polymorphism: Option

```
sealed abstract class Option[A]

case class Some[A](value: A) extends Option[A]

case class None[A]() extends Option[A]
```

Universal polymorphism: Either

```scala
sealed abstract class ThisOrThat

case class This(value: Int) extends ThisOrThat

case class That(value: String) extends ThisOrThat
```

Universal polymorphism: Either

```
sealed abstract class ThisOrThat[A, B]

case class This[A, B](value: A) extends ThisOrThat[A, B]

case class That[A, B](value: B) extends ThisOrThat[A, B]
```

Universal polymorphism: Either

```
sealed abstract class Either[A, B]

case class Left[A, B](value: A) extends Either[A, B]

case class Right[A, B](value: B) extends Either[A, B]
```

Monoids, functors and monads

These concepts come from Category Theory.

“…the abstract theory of functions…”

Each can be seen as a type or structure that follows certain rules; by following these rules we can make use of their objects in a more generic way, always obtaining the expected results.

In Scala, such types cannot be created in a way that their objects will strictly be used following those rules. It’s up to the programmer to follow them.

Monoids

Rules for Monoids are:

- Have an identity element.
- Have an associative function.

Examples are:

- Integers with identity 0 and sum function.
- Integers with identity 1 and product function.
- Strings with identity "" and concatenation function.

Monoids

```
trait Monoid[A] {
  def identity: A
  def op(a: A, b: A): A
}
```

Monoids

```
val m = new Monoid[Int] {
  def identity: Int = 0
  def op(a: Int, b: Int): Int = a + b
}
m.op(3, 6) // = 9
m.op(3, m.op(4, 2)) // = 9
m.op(m.op(3, 4), 2) // = 9

m.op(m.identity, 2) // = 2
m.op(2, m.identity) // = 2
```

Monoids

```
val s = new Monoid[String] {
  def identity: String = ""
  def op(a: String, b: String): String = a + b
}
s.op("abc", "defghi") // = abcdefghi
s.op("abc", s.op("def", "ghi")) // = abcdefghi
s.op(s.op("abc", "def"), "ghi") // = abcdefghi

s.op(s.identity, "abc") // = abc
s.op("abc", s.identity) // = abc
```

Monoids

```
val s = new Monoid[List[Int]] {
  def identity: List[Int] = Nil
  def op(a: List[Int], b: List[Int]): List[Int] = (a, b) match {
    case (Nil, _) => b
    case (_, Nil) => a
    case (ah :: at, _) => ah :: op(at, b)
  }
}
s.op(List(1, 2, 3), List(4, 5)) // = List(1, 2, 3, 4, 5)
s.op(s.identity, List(1, 2, 3)) // = List(1, 2, 3)
```

Functor

A functor is a mapping/transformation of objects from one type to another type, while preserving their structure. For example:

$$F: Monoid[A] \Rightarrow Monoid[B]$$

$$F: List[A] \Rightarrow List[B]$$

Functor

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

Functor: List

```
val f = new Functor[List] {
  def map[A, B](collection: List[A], transform: A => B): List[B] =
    collection match {
      case Nil => Nil
      case h :: t => transform(h) :: map(t, transform)
    }
}


f.map(List(1, 2), (x: Int) => s"I'm $x")  // List[String] = List(I'm 1, I'm 2)
```

Functor: Option

```
val f = new Functor[Option] {
  def map[A, B](collection: Option[A], transform: A => B): Option[B] =
    collection match {
      case None => None
      case Some(x) => Some(transform(x))
    }
}


f.map(Some("Hello"), (x: String) => x.size)  // = Option[Int] = Some(5)
```

Monads

A monad is an extension of a functor which:

- Implements flatMap
- Obeys some algebraic laws (associativity, left unit and right unit)

Monads

```
trait Monad[F[_]] extends Functor[F] {
  def unit[A](x: A): F[A]
  def flatMap[A, B](ma: F[A], f: A => F[B]): F[B]
}
```

Monads: map vs. flatMap

```
List("hello", "there", "Scala").map(_.toList)
// = List(List(h, e, l, l, o), List(t, h, e, r, e), List(S, c, a, l, a))


List("hello", "there", "Scala").flatMap(_.toList)
// = List(h, e, l, l, o, t, h, e, r, e, S, c, a, l, a)
```

Monads: map vs. flatMap

```
List(List(1, 2), List(3, 4), List(5, 6)).map(x => x)
// = List(List(1, 2), List(3, 4), List(5, 6))


List(List(1, 2), List(3, 4), List(5, 6)).flatMap(x => x)
// = List(1, 2, 3, 4, 5, 6)
```

Monads: map vs. flatMap

```
List(List(1, 2), List(3, 4), List(5, 6)).map(x => x)
// = List(List(1, 2), List(3, 4), List(5, 6))


List(List(1, 2), List(3, 4), List(5, 6)).flatMap(x => x)
// = List(1, 2, 3, 4, 5, 6)


List(List(1, 2), List(3, 4), List(5, 6)).flatten
// = List(1, 2, 3, 4, 5, 6)
```

Monads: List

```scala
val m = new Monad[List] {
  def unit[A](x: A): List[A] = List(x)
  def flatMap[A, B](ma: List[A], f: A => List[B]): List[B] =
    ma match {
      case Nil => Nil
      case h :: t => f(h) ++: flatMap(t, f)
    }
  def map[A, B](ma: List[A], f: A => B): List[B] = ???
}
```

About Scala: objects everywhere

```
val i = 5

i.toDouble

i.max(8)

i + 5

i.+(5)
```

About Scala: objects everywhere

```
val foo = (n: Int) => n * n


val foo = new Function[Int, Int] {
  override def apply(n: Int): Int = n * n
}
```

About Scala: Apply method

```
val ns = Array(1, 2, 3)
ns(2) // = 3


val ns = NonEmpty(1, NonEmpty(2, NonEmpty(3, Empty)))
ns(2) // = 3
```

About Scala: Apply method

```
sealed abstract class MyList {...; def get(i: Int): Int }
case class NonEmpty(...) {
  def get(i: Int): Int =
    if (i == 0) head
    else tail.apply(i-1)
}
ns.get(2)   // = 3
```

About Scala: Apply method

```
sealed abstract class MyList {...; def apply(i: Int): Int }
case class NonEmpty(...) {
 def apply(i: Int): Int =
  if (i == 0) head
  else tail.apply(i-1)
}
ns(2) // = 3
```

About Scala: Apply method

```scala
val ns = List(1, 2, 3)
```

```scala
val ns = MyList(1, 2, 3)
```

About Scala: Apply method

```
object MyList {
  def create(ns: Int*): MyList = {
    if (ns.isEmpty) Empty
    else NonEmpty(ns.head, create(ns.tail:_*))
  }
}

val ns = MyList.create(1, 2, 3)
```

About Scala: Apply method

```scala
object MyList {
  def apply(ns: Int*): MyList = {
    if (ns.isEmpty) Empty
    else NonEmpty(ns.head, apply(ns.tail:_*))
  }
}

val ns = MyList(1, 2, 3)
```

About Scala: Extractors (unapply)

```
object Email {
  def apply(u: String, d: String): String = u + "@" + d
}


Email("jenny", "mail")  // = jenny@mail
"jenny@mail"       // = Email(u, d)
```

About Scala: Extractors (unapply)

```
object Email {
  def apply(u: String, d: String): String = u + "@" + d
}

Email("jenny", "mail")  // = jenny@mail
"jenny@mail" match {
  case Email(u, d) => ...
}
```

About Scala: Extractors (unapply)

```scala
object Email {
  def unapply(email: String): Option[(String, String)] = {
    val parts = str split "@"
    if (parts.length == 2) Some(parts(0), parts(1)) else None
  }
}
Email("jenny", "mail")  // = jenny@mail
"jenny@mail" match {
  case Email(u, d) => ...
}
```

About Scala: Extractors (unapply)

```scala
object Email {
  def unapply(email: String): Option[(String, String)] = {
    val parts = str split "@"
    if (parts.length == 2) Some(parts(0), parts(1)) else None
  }
}
val someEmail: String = ...
val conn = someEmail match {
  case Email(u, "mail") => new MailConnection(u)
  case Email(u, "gmail") => new GMailConnection(u)}
```