ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Composing Futures

Principles of Reactive Programming

Erik Meijer

Flatmap …

```
val socket = Socket()
val packet: Future[Array[Byte]] =
  socket.readFromMemory()
val confirmation: Future[Array[Byte]] =
  packet.flatMap(socket.sendToSafe(_))
```

Or comprehensions?

```
val socket = Socket()
val confirmation: Future[Array[Byte]] = for {
  packet       <- socket.readFromMemory()
  confirmation <- socket.sendToSafe(packet)
} yield confirmation
```

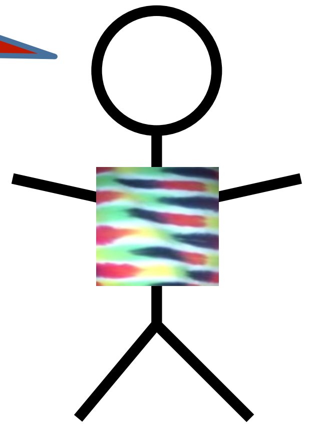
Retrying to send

```
def retry(noTimes: Int)(block: ⇒Future[T]): Future[T] = {
   … retry successfully completing block at most noTimes
   … and give up after that
}
```

Retrying to send

```
def retry(noTimes: Int)(block: ⇒Future[T]): Future[T] = {
  if (noTimes == 0) {
      Future.failed(new Exception(“Sorry”))
  } else {
      block fallbackTo {
         retry(noTimes-1) { block }
      }
  }
}
```

Recursion is the GOTO of Functional Programming (Erik Meijer)

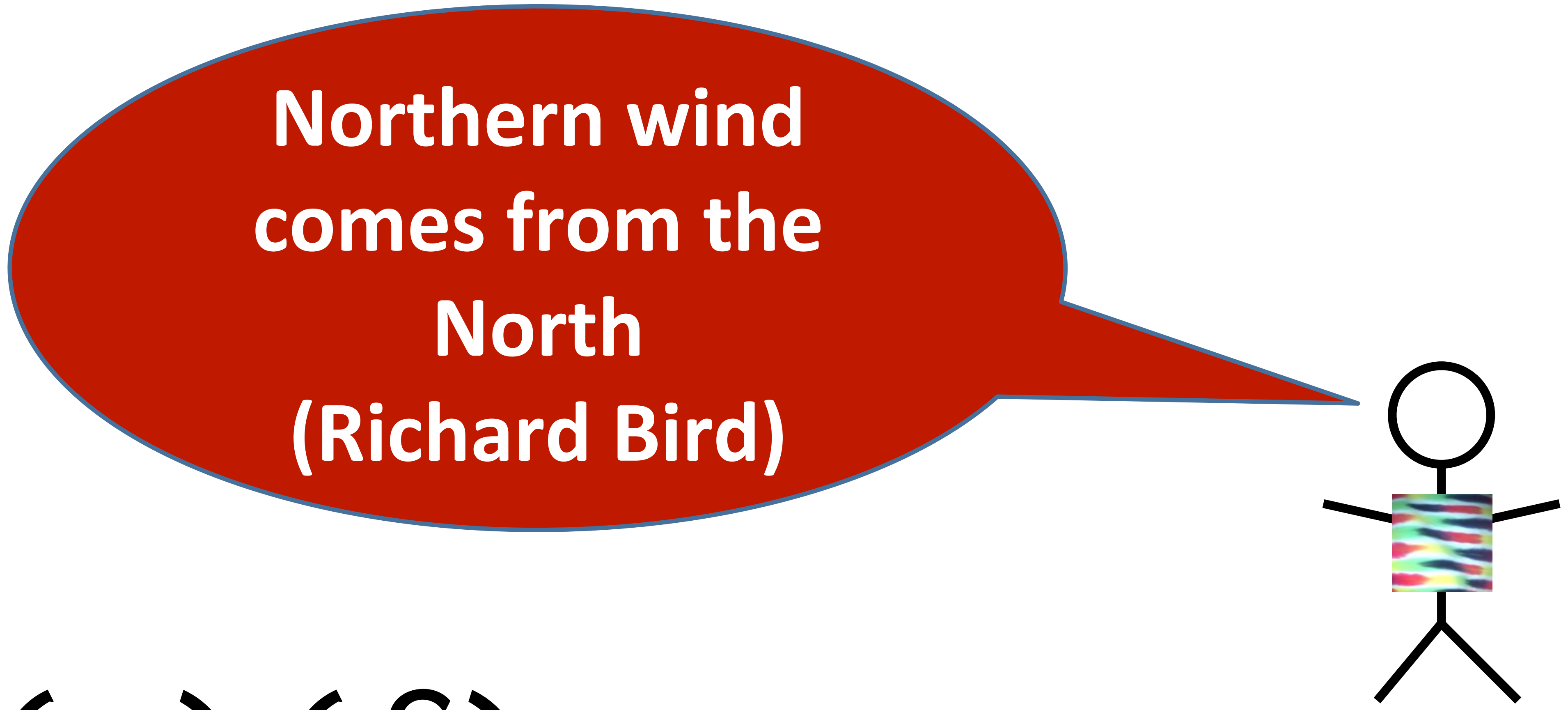


Folding lists

List(a,b,c).foldRight(e)(f)

=

$\overleftarrow{\texttt{f(a, f(b, f(c, e)))}}$



List(a,b,c).foldLeft(e)(f)

=

$\overrightarrow{\texttt{f(f(f(e, a), b), c)}}$

Retrying to send using foldLeft

```
def retry(noTimes: Int)(block: ⇒Future[T]): Future[T] = {
  val ns: Iterator[Int] = (1 to noTimes).iterator
  val attempts: Iterator[Future[T]] = ns.map(_⇒ ()⇒block)
  val failed = Future.failed(new Exception)

  attempts.foldLeft(failed)
     ((a,block) ⇒ a recoverWith { block() })
}
```

retry(3) { block }

= *unfolds to*

((failed recoverWith block_1) recoverWith block_2) recoverWith block_3

Folding lists

```
List(a,b,c).foldRight(e)(f)
=
```

$$\overleftarrow{\texttt{f(a, f(b, f(c, e)))}}$$

```
List(a,b,c).foldLeft(e)(f)
=
```

$$\overrightarrow{\texttt{f(f(f(e, a), b), c)}}$$

Retrying to send using foldRight

```
def retry(noTimes: Int)(block: ⇒Future[T]): Future[T] = {
  val ns: Iterator[Int] = (1 to noTimes).iterator
  val attempts: Iterator[Future[T]] = ns.map(_⇒ ()⇒block)
  val failed = Future.failed(new Exception)

  attempts.foldRight(() ⇒ failed)
    ((block, a) ⇒ () ⇒ { block() fallbackTo { a() } })
}
```

```
retry(3) { block } ()
```

= *unfolds to*

block$_1$ fallbackTo { block$_2$ fallbackTo { block$_3$ fallbackTo { failed }}}

Making effects implicit

```
T => Future[S]


T => Try[S] or even
T => S
```

Async await magic

```
import scala.async.Async._

def async[T](body: =>T)
(implicit context: ExecutionContext): Future[T]

def await[T](future: Future[T]): T
```

```
async{ … await{…} …}
```

Async, the small print

Illegal Uses

The following uses of await are illegal and are reported as errors:

- await requires a directly-enclosing async; this means await must not be used inside a closure nested within an async block, or inside a nested object, trait, or class.
- await must not be used inside an expression passed as an argument to a by-name parameter.
- await must not be used inside a Boolean short-circuit argument.
- return expressions are illegal inside an async block.
- **await should not be used under a try/catch**.

Retrying to send using await

```
def retry(noTimes: Int)(block: ⇒Future[T]): Future[T] = async {
  var i = 0
  var result: Try[T] = Failure(new Exception(“sorry man!”))
  while (i < noTimes && result.isFailure) {
    result = await { Try(block) }
    i += 1
  }
  result.get
}
```

```
object Try {
  def apply(f: Future[T]): Future[Try[T]] = {…}
}
```

Reimplementing filter using await

```
def filter(p: T ⇒ Boolean): Future[T] = async {
  val x = await { this }
  if (!p(x)) {
    throw new NoSuchElementException()
  } else {
    x
  }
}
```

Quiz

```
def flatMap[S](f: T ⇒ Future[S]): Future[S] =

(a) async { await { f(this) } }

(b) async { f(this) }

(c) async { f( await { this } ) }

(d) async { await { f( await { this } ) } }
```

Reimplementing filter without await

```scala
def filter(pred: T ⇒ Boolean): Future[T] = {
  val p = Promise[T]()

  this onComplete {
    case Failure(e) ⇒
      p.failure(e)
    case Success(x) ⇒
      if (!pred(x)) p.failure(new NoSuchElementException)
      else p.success(x)
  }

  p.future
}
```