香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Principles of Programming Languages (Lecture 9)

COMP 3031, Fall 2025

Lionel Parreaux

Recap from lectures 1 – 8

Recap: Case Classes

Case classes are Scala's preferred way to define complex data.

Example: Representing JSON (Java Script Object Notation)

```
{ "firstName" : "John",
  "lastName" : "Smith",
  "address": {
     "streetAddress": "21 2nd Street",
     "state": "NY",
     "postalCode": 10021
  },
  "phoneNumbers": [
    { "type": "home", "number": "212 555-1234" },
    { "type": "fax", "number": "646 555-4567" }
  ]
}
```

Representation of JSON with Case Classes

```scala
sealed abstract class JSON
object JSON:
  case class Seq (elems: List[JSON])                   extends JSON
  case class Obj (bindings: Map[String, JSON]) extends JSON
  case class Num (num: Double)                         extends JSON
  case class Str (str: String)                         extends JSON
  case class Bool(b: Boolean)                          extends JSON
  case object Null                                     extends JSON
```

Representation of JSON with Enums

Case class hierarchies can be represented more concisely as enums:

```
enum JSON:
  case Seq (elems: List[JSON])
  case Obj (bindings: Map[String, JSON])
  case Num (num: Double)
  case Str (str: String)
  case Bool(b: Boolean)
  case Null
```

Example

```scala
val jsData = JSON.Obj(Map(
  "firstName" -> JSON.Str("John"),
  "lastName" -> JSON.Str("Smith"),
  "address" -> JSON.Obj(Map(
    "streetAddress" -> JSON.Str("21 2nd Street"),
    "state" -> JSON.Str("NY"),
    "postalCode" -> JSON.Num(10021)
  )),
  "phoneNumbers" -> JSON.Seq(List(
    JSON.Obj(Map(
      "type" -> JSON.Str("home"), "number" -> JSON.Str("212 555-1234")
    )),
    JSON.Obj(Map(
      "type" -> JSON.Str("fax"), "number" -> JSON.Str("646 555-4567")
    )) )) ))
```

Pattern Matching

Here's a method that returns the string representation of JSON data:

```
def show(json: JSON): String = json match
  case JSON.Seq(elems) =>
    elems.map(show).mkString("[", ", ", "]")
  case JSON.Obj(bindings) =>
    val assocs = bindings.map(
      (key, value) => s"${inQuotes(key)}: ${show(value)}")
    assocs.mkString("{", ",\n ", "}")
  case JSON.Num(num) => num.toString
  case JSON.Str(str) => inQuotes(str)
  case JSON.Bool(b)  => b.toString
  case JSON.Null     => "null"

def inQuotes(str: String): String = "\"" + str + "\""
```

Recap: Collections

Scala has a rich hierarchy of collection classes.

Recap: Collection Methods

All collection types share a common set of general methods.

Core methods:

```
map
flatMap
filter
```

and also

```
foldLeft
foldRight
```

Idealized Implementation of map on Lists

```scala
extension [T](xs: List[T])
  def map[U](f: T => U): List[U] = xs match
    case x :: xs1 => f(x) :: xs1.map(f)
    case Nil => Nil
```

Idealized Implementation of `flatMap` on Lists

```
extension [T](xs: List[T])
  def flatMap[U](f: T => List[U]): List[U] = xs match
    case x :: xs1 => f(x) ::: xs1.flatMap(f)
    case Nil => Nil
```

Idealized Implementation of `filter` on Lists

```
extension [T](xs: List[T])
  def filter(p: T => Boolean): List[T] = xs match {
    case x :: xs1 =>
      if p(x) then x :: xs1.filter(p) else xs1.filter(p)
    case Nil => Nil
```

Idealized Implementation of `filter` on Lists

```
extension [T](xs: List[T])
  def filter(p: T => Boolean): List[T] = xs match {
    case x :: xs1 =>
      if p(x) then x :: xs1.filter(p) else xs1.filter(p)
    case Nil => Nil
```

In practice, the implementation and type of these methods are different in order to

- make them apply to arbitrary collections, not just lists,
- make them tail-recursive on lists.

For-Expressions

Simplify combinations of core methods `map`, `flatMap`, `filter`.

Instead of:

```
(1 until n).flatMap(i =>
  (1 until i).filter(j => isPrime(i + j))
    .map(j => (i, j)))
```

one can write:

```
   for
     i <- 1 until n
     j <- 1 until i
     if isPrime(i + j)
   yield (i, j)
```

For-expressions and Pattern Matching

The left-hand side of a generator may also be a pattern:

```
def bindings(x: JSON): List[(String, JSON)] = x match
  case JSON.Obj(bindings) => bindings.toList
  case _ => Nil

for
  case ("phoneNumbers", JSON.Seq(numberInfos)) <- bindings(jsData)
  numberInfo <- numberInfos
  case ("number", JSON.Str(number)) <- bindings(numberInfo)
  if number.startsWith("852")
yield
  number
```

If the pattern starts with `case`, the sequence is filtered so that only elements matching the pattern are retained.

Putting the Pieces Together

Task

Once upon a time, before smartphones, phone keys had mnemonics assigned to them.

```
val mnemonics = Map(
      '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",
      '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")
```

Assume you are given a dictionary `words` as a list of words.

Design a method `encode` such that

```
encode(phoneNumber)
```

produces all phrases of words that can serve as mnemonics for the phone number.

Example: The phone number "7225247386" should have the mnemonic `Scala is fun` as one element of the set of solution phrases.

Outline

```scala
class Coder(words: List[String]):
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  private val charCode: Map[Char, Char] = ???

  /** Maps a word to the digit string it can represent */
  private def wordCode(word: String): String = ???

  /** Maps a digit string to all words in the dictionary that represent it */
  private val wordsForNum: Map[String, List[String]] = ???

  /** All ways to encode a number as a list of words */
  def encode(number: String): Set[List[String]] = ???
```

Implementation (1)

```scala
class Coder(words: List[String]):
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  private val charCode: Map[Char, Char] =
```

Implementation (1)

```scala
class Coder(words: List[String]):
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  private val charCode: Map[Char, Char] =
    for
      (digit, str) <- mnemonics
      ltr <- str
    yield ltr -> digit
```

Implementation (1)

```
class Coder(words: List[String]):
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  private val charCode: Map[Char, Char] =
    for (digit, str) <- mnemonics; ltr <- str yield ltr -> digit

  /** Maps a word to the digit string it can represent */
  private def wordCode(word: String): String =
```

Implementation (1)

```scala
class Coder(words: List[String]):
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  private val charCode: Map[Char, Char] =
    for (digit, str) <- mnemonics; ltr <- str yield ltr -> digit

  /** Maps a word to the digit string it can represent */
  private def wordCode(word: String): String = word.toUpperCase.map(charCode)
```

Implementation (1)

```
class Coder(words: List[String]):
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  private val charCode: Map[Char, Char] =
    for (digit, str) <- mnemonics; ltr <- str yield ltr -> digit

  /** Maps a word to the digit string it can represent */
  private def wordCode(word: String): String = word.toUpperCase.map(charCode)

  /** Maps a digit string to all words in the dictionary that represent it */
  private val wordsForNum: Map[String, List[String]] =
```

Implementation (1)

```scala
class Coder(words: List[String]):
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  private val charCode: Map[Char, Char] =
    for (digit, str) <- mnemonics; ltr <- str yield ltr -> digit

  /** Maps a word to the digit string it can represent */
  private def wordCode(word: String): String = word.toUpperCase.map(charCode)

  /** Maps a digit string to all words in the dictionary that represent it */
  private val wordsForNum: Map[String, List[String]] =
    words.groupBy(wordCode).withDefaultValue(Nil)
```

Implementation (2)

```
/** All ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] =
```

Idea: use divide and conquer

Implementation (2)

```
/** All ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] =
  if number.isEmpty then ???
  else ???
```

Implementation (2)

```
/** All ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] =
  if number.isEmpty then Set(Nil)
  else ???
```

Implementation (2)

```scala
/** All ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] =
  if number.isEmpty then Set(Nil)
  else
    for
      splitPoint <- (1 to number.length).toSet
      word <- ???
      rest <- ???
    yield word :: rest
```

Implementation (2)

```
/** All ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] =
  if number.isEmpty then Set(Nil)
  else
    for
      splitPoint <- (1 to number.length).toSet
      word <- wordsForNum(number.take(splitPoint))
      rest <- ???
    yield word :: rest
```

Implementation (2)

```scala
/** All ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] =
  if number.isEmpty then Set(Nil)
  else
    for
      splitPoint <- (1 to number.length).toSet
      word <- wordsForNum(number.take(splitPoint))
      rest <- encode(number.drop(splitPoint))
    yield word :: rest
```

Testing It

A test program:

```
@main def code(number: String) =
  val coder = Coder(List(
    "Scala", "Python", "Ruby", "C",
    "rocks", "socks", "sucks", "works", "pack"))
  coder.encode(number).map(_.mkString(" "))
```

A sample run:

```
> scala code "7225276257"
HashSet("Scala rocks", "pack C rocks", "pack C socks", "Scala socks")
```

Background

This example was taken from:

Lutz Prechelt: An Empirical Comparison of Seven Programming Languages. IEEE Computer 33(10): 23-29 (2000)

Tested with Tcl, Python, Perl, Rexx, Java, C++, C.

Code size medians:

- 100 loc for scripting languages
- 200-300 loc for the others

Background

This example was taken from:

Lutz Prechelt: An Empirical Comparison of Seven Programming Languages. IEEE Computer 33(10): 23-29 (2000)

Tested with Tcl, Python, Perl, Rexx, Java, C++, C.

Code size medians:

- 100 loc for scripting languages
- 200-300 loc for the others

In Scala:

- ~20 loc!
- *yet* statically typed
- purely functional, no side effects (ie, easy to reason about & refactor)

Benefits

Scala's immutable collections are:

- *easy to use*: few steps to do the job.
- *concise*: one word replaces a whole loop.
- *safe*: type checker is really good at catching errors.
- *fast*: collection ops are tuned, can be parallelized.
- *universal*: one vocabulary to work on all kinds of collections.

This makes them an attractive tool for software development

Queries with For

Queries with `for`

Insight: The `for` notation is essentially equivalent to the common operations of query languages for databases.

Example: Suppose that we have a database `books`, represented as a list of books.

```
case class Book(title: String, authors: List[String])
```

A Mini-Database

```
val books: List[Book] = List(
  Book(title   = "Structure and Interpretation of Computer Programs",
       authors = List("Abelson, Harald", "Sussman, Gerald J.")),
  Book(title   = "Introduction to Functional Programming",
       authors = List("Bird, Richard", "Wadler, Phil")),
  Book(title   = "Effective Java",
       authors = List("Bloch, Joshua")),
  Book(title   = "Java Puzzlers",
       authors = List("Bloch, Joshua", "Gafter, Neal")),
  Book(title   = "Programming in Scala",
       authors = List("Odersky, Martin", "Spoon, Lex", "Venners, Bill")))
```

Some Queries

To find the titles of books whose author's name is "Bird":

```
for
  b <- books
  a <- b.authors
  if a.startsWith("Bird,")
yield b.title
```

To find all the books which have the word "Program' ' in the title:

```
for b <- books if b.title.indexOf("Program") >= 0
yield b.title
```

Another Query

To find the names of all authors who have written at least two books present in the database.

```
for
  b1 <- books
  b2 <- books
  if b1 != b2
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
yield a1
```

Another Query

To find the names of all authors who have written at least two books present in the database.

```
for
  b1 <- books
  b2 <- books
  if b1 != b2
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
yield a1
```

Why do solutions show up twice?

How can we avoid this?

Modified Query

To find the names of all authors who have written at least two books present in the database.

```
for
  b1 <- books
  b2 <- books
  if b1.title < b2.title
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
yield a1
```

Problem

What happens if an author has published three books?

```
O    The author is printed once
O    The author is printed twice
O    The author is printed three times
O    The author is not printed at all
```

Problem

What happens if an author has published three books?

```
O    The author is printed once
O    The author is printed twice
X    The author is printed three times
O    The author is not printed at all
```

Modified Query (2)

Solution: Remove duplicate authors who are in the results list twice.

This is achieved using the `distinct` method on sequences:

```
val repeated =
  for
    b1 <- books
    b2 <- books
    if b1.title < b2.title
    a1 <- b1.authors
    a2 <- b2.authors
    if a1 == a2
  yield a1
repeated.distinct
```

Modified Query (3)

Better alternative: Compute with sets instead of sequences:

```
val bookSet = books.toSet
for
  b1 <- bookSet
  b2 <- bookSet
  if b1 != b2
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
yield a1
```

Translation of For

For-Expressions and Higher-Order Functions

The syntax of `for` is closely related to the higher-order functions `map`, `flatMap` and `filter`.

First of all, these functions can all be defined in terms of `for`:

```
def mapFun[T, U](xs: List[T], f: T => U): List[U] =
  for x <- xs yield f(x)

def flatMap[T, U](xs: List[T], f: T => Iterable[U]): List[U] =
  for x <- xs; y <- f(x) yield y

def filter[T](xs: List[T], p: T => Boolean): List[T] =
  for x <- xs if p(x) yield x
```

Translation of For (1)

In reality, the Scala compiler expresses for-expressions in terms of `map`, `flatMap` and a lazy variant of `filter`.

Here is the translation scheme used by the compiler (we limit ourselves here to simple variables in generators)

1. A simple for-expression

```
for x <- e1 yield e2
```

is translated to

```
e1.map(x => e2)
```

Translation of For (2)

2. A for-expression

```
for x <- e1 if pred; s yield e2
```

where `pred` is a filter and `s` is a (potentially empty) sequence of generators and filters, is translated to

```
for x <- e1.withFilter(x => pred); s yield e2
```

(and the translation continues with the new expression)

You can think of `withFilter` as a variant of `filter` that does not produce an intermediate collection, but instead applies the following `map` or `flatMap` function application only to those elements that passed the test.

Translation of For (3)

3. A for-expression

```
for x <- e1; y <- e2; s yield e3
```

where `s` is a (potentially empty) sequence of generators and filters, is translated into

```
e1.flatMap(x => for y <- e2; s yield e3)
```

(and the translation continues with the new expression)

Example

Take the for-expression that computed pairs whose sum is prime:

```
for
  i <- 1 until n
  j <- 1 until i
  if isPrime(i + j)
yield (i, j)
```

Applying the translation scheme to this expression gives:

```
(1 until n).flatMap(i =>
  (1 until i)
    .withFilter(j => isPrime(i + j))
    .map(j => (i, j)))
```

This is almost exactly the expression which we came up with first!

Exercise

Translate

```
for b <- books; a <- b.authors if a.startsWith("Bird")
yield b.title
```

into higher-order functions.

Exercise

```
for b <- books; a <- b.authors if a.startsWith(”Bird”)
yield b.title
```

The expression above expands to which of the following two expressions?

O
```
books.flatMap(b =>
  b.authors.withFilter(a =>
    a.startsWith(”Bird”)).map(a => b.title))
```

O
```
books.map(b =>
  b.authors.flatMap(a =>
    if a.startsWith(”Bird”) then b.title))
```

Generalization of `for`

Interestingly, the translation of for is not limited to lists or sequences, or even collections;

It is based solely on the presence of the methods `map`, `flatMap` and `withFilter`.

This lets you use the for syntax for your own types as well – you must only define `map`, `flatMap` and `withFilter` for these types.

There are many types for which this is useful: arrays, iterators, databases, optional values, parsers, asynchronous futures, etc.

For and Databases

For example, `books` might not be a list, but a database stored on some server.

As long as the client interface to the database defines the methods `map`, `flatMap` and `withFilter`, we can use the `for` syntax for querying the database.

This is the basis of data base connection frameworks such as *Slick* or *Quill*, as well as big data platforms such as *Spark*.

Functional Random Generators

Other Uses of For-Expressions

Question: Are for-expressions tied to collection-like things such as lists, sets, or databases?

Other Uses of For-Expressions

Question: Are for-expressions tied to collection-like things such as lists, sets, or databases?

Answer: No! All that is required is some interpretation of `map`, `flatMap` and `withFilter`.

There are many domains outside collections that afford such an interpretation.

Example: random value generators.

Random Values

You know about random numbers:

```
val rand = java.util.Random()
rand.nextInt()
```

Question: What is a systematic way to get random values for other domains, such as

- booleans, strings, pairs and tuples, lists, sets, trees

?

Generators

Let's define a trait `Generator[T]` that generates random values of type `T`:

```
trait Generator[+T]:
  def generate(): T
```

Some instances:

```
val integers = new Generator[Int]:
  val rand = java.util.Random()
  def generate() = rand.nextInt()
```

Generators

Let's define a trait `Generator[T]` that generates random values of type `T`:

```
trait Generator[+T]:
  def generate(): T
```

Some instances:

```
val booleans = new Generator[Boolean]:
  def generate() = integers.generate() > 0
```

Generators

Let's define a trait `Generator[T]` that generates random values of type `T`:

```
trait Generator[+T]:
  def generate(): T
```

Some instances:

```
val pairs = new Generator[(Int, Int)]:
  def generate() = (integers.generate(), integers.generate())
```

Streamlining It

Can we avoid the `new Generator ...` boilerplate?

Ideally, we would like to write:

```
val booleans = for x <- integers yield x > 0

def pairs[T, U](t: Generator[T], u: Generator[U]) =
  for x <- t; y <- u yield (x, y)
```

What does this expand to?

Streamlining It

Can we avoid the `new Generator ...` boilerplate?

Ideally, we would like to write:

```
val booleans = integers.map(x => x > 0)

def pairs[T, U](t: Generator[T], u: Generator[U]) =
  t.flatMap(x => u.map(y => (x, y)))
```

Need `map` and `flatMap` for that!

Generator with `map` and `flatMap`

Here's a more convenient version of `Generator`:

```
trait Generator[+T]:
  def generate(): T

extension [T, S](g: Generator[T])
  def map(f: T => S) = new Generator[S]:
    def generate() = f(g.generate())
```

Generator with `map` and `flatMap`

Here's a more convenient version of `Generator`:

```scala
trait Generator[+T]:
  def generate(): T

extension [T, S](g: Generator[T])
  def map(f: T => S) = new Generator[S]:
    def generate() = f(g.generate())

  def flatMap(f: T => Generator[S]) = new Generator[S]:
    def generate() = f(g.generate()).generate()
```

Generator with `map` and `flatMap` (2)

We can also implement `map` and `flatMap` as methods of class `Generator`:

```
trait Generator[+T]:
  def generate(): T

  def map[S](f: T => S) = new Generator[S]:
    def generate() = f(Generator.this.generate())
  def flatMap[S](f: T => Generator[S]) = new Generator[S]:
    def generate() = f(Generator.this.generate()).generate()
```

Note the use of `Generator.this` to the refer to the `this` of the "outer" object of class `Generator`.

The booleans Generator

What does this definition resolve to?

```
val booleans = for x <- integers yield x > 0
```

The booleans Generator

What does this definition resolve to?

```
val booleans = for x <- integers yield x > 0
```

```
val booleans = integers.map(x => x > 0)
```

The booleans Generator

What does this definition resolve to?

```scala
val booleans = for x <- integers yield x > 0
```

```scala
val booleans = integers.map(x => x > 0)
```

```scala
val booleans = new Generator[Boolean]:
  def generate() = ((x: Int) => x > 0)(integers.generate())
```

The booleans Generator

What does this definition resolve to?

```
val booleans = for x <- integers yield x > 0
```

```
val booleans = integers.map(x => x > 0)
```

```
val booleans = new Generator[Boolean]:
  def generate() = ((x: Int) => x > 0)(integers.generate())
```

```
val booleans = new Generator[Boolean]:
  def generate() = integers.generate() > 0
```

The pairs Generator

```
def pairs[T, U](t: Generator[T], u: Generator[U]) = t.flatMap(
  x => u.map(y => (x, y)))
```

The pairs Generator

```scala
def pairs[T, U](t: Generator[T], u: Generator[U]) = t.flatMap(
  x => u.map(y => (x, y)))

def pairs[T, U](t: Generator[T], u: Generator[U]) = t.flatMap(
  x => new Generator[(T, U)] { def generate() = (x, u.generate()) })
```

The pairs Generator

```scala
def pairs[T, U](t: Generator[T], u: Generator[U]) = t.flatMap(
  x => u.map(y => (x, y)))

def pairs[T, U](t: Generator[T], u: Generator[U]) = t.flatMap(
  x => new Generator[(T, U)] { def generate() = (x, u.generate()) })

def pairs[T, U](t: Generator[T], u: Generator[U]) = new Generator[(T, U)]:
  def generate() = (new Generator[(T, U)]:
    def generate() = (t.generate(), u.generate())
  ).generate()
```

The pairs Generator

```scala
def pairs[T, U](t: Generator[T], u: Generator[U]) = t.flatMap(
  x => u.map(y => (x, y)))

def pairs[T, U](t: Generator[T], u: Generator[U]) = t.flatMap(
  x => new Generator[(T, U)] { def generate() = (x, u.generate()) })

def pairs[T, U](t: Generator[T], u: Generator[U]) = new Generator[(T, U)]:
  def generate() = (new Generator[(T, U)]:
    def generate() = (t.generate(), u.generate())
  ).generate()

def pairs[T, U](t: Generator[T], u: Generator[U]) = new Generator[(T, U)]:
  def generate() = (t.generate(), u.generate())
```

Generator Examples

```
def single[T](x: T): Generator[T] = new Generator[T]:
  def generate() = x

def range(lo: Int, hi: Int): Generator[Int] =
  for x <- integers yield lo + x.abs % (hi - lo)

def oneOf[T](xs: T*): Generator[T] =
  for idx <- range(0, xs.length) yield xs(idx)
```

A `List` Generator

A list is either an empty list or a non-empty list.

```
def lists: Generator[List[Int]] =
  for
    isEmpty <- booleans
    list <- if isEmpty then emptyLists else nonEmptyLists
  yield list
```

A List Generator

A list is either an empty list or a non-empty list.

```
def lists: Generator[List[Int]] =
  for
    isEmpty <- booleans
    list <- if isEmpty then emptyLists else nonEmptyLists
  yield list

def emptyLists = single(Nil)
```

A List Generator

A list is either an empty list or a non-empty list.

```
def lists: Generator[List[Int]] =
  for
    isEmpty <- booleans
    list <- if isEmpty then emptyLists else nonEmptyLists
  yield list

def emptyLists = single(Nil)

def nonEmptyLists =
  for
    head <- integers
    tail <- lists
  yield head :: tail
```

Exercise: A `Tree` Generator

Can you implement a generator that creates random `Tree` objects?

```
enum Tree:
  case Inner(left: Tree, right: Tree)
  case Leaf(x: Int)
```

Solution: A `Tree` Generator

Can you implement a generator that creates random `Tree` objects?

```
def leaves: Generator[Leaf] = for
    x <- integers
  yield Leaf(x)

def inners: Generator[Inner] = for
    l <- trees
    r <- trees
  yield Inner(l, r)

def trees: Generator[Tree] = for
    cutoff <- booleans
    tree <- if (cutoff) leaves else inners
  yield tree
```

Application: Random Testing

You know about unit tests:

- Come up with some test inputs for a function and a *postcondition*.
- The postcondition is a property of the expected result.
- Verify that the program satisfies the postcondition.

Question: Can we do without the test inputs?

Yes, by generating *random test inputs*.

Random Test Function

Using generators, we can write a random test function:

```
def test[T](g: Generator[T], numTimes: Int = 100)
           (test: T => Boolean): Unit =
  for i <- 0 until numTimes do
    val value = g.generate()
    assert(test(value), s"test failed for $value")
  println(s"passed $numTimes tests")
```

(where ‘`for x <- xs do body`’ is syntax sugar for ‘`xs.foreach(x => body)`’)

Random Test Function

Example usage:

```
test(pairs(lists, lists)) {
  (xs, ys) => (xs ++ ys).length > xs.length
}
```

Question: Does the above property always hold?

O Yes

O No

Random Test Function

Example usage:

```
test(pairs(lists, lists)) {
  (xs, ys) => (xs ++ ys).length > xs.length
}
```

Question: Does the above property always hold?

```
O        Yes
X        No
```

Random Test Function

Example usage:

```
test(pairs(lists, lists)) {
  (xs, ys) => (xs ++ ys).length > xs.length
}
```

Question: Does the above property always hold?

```
O       Yes
X       No
```

Fix:

```
(xs, ys) => (xs ++ ys).length >= xs.length
```

ScalaCheck

Shift in viewpoint: Instead of writing tests, write *properties* that are assumed to hold.

This idea is implemented in the `ScalaCheck` tool.

```
forAll { (l1: List[Int], l2: List[Int]) =>
  (l1 ++ l2).size == l1.size + l2.size
}
```

It can be used either stand-alone or as part of ScalaTest.