Recap from Weeks 1 - 6

Principles of Functional Programming

Martin Odersky

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
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

```
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)(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(”212”)
yield
  number
```

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