Other Collections

Other Sequences

We have seen that lists are *linear*: Access to the first element is much faster than access to the middle or end of a list.

The Scala library also defines an alternative sequence implementation, `Vector`.

This one has more evenly balanced access patterns than `List`.

Operations on Vectors

Vectors are created analogously to lists:

```
val nums = Vector(1, 2, 3, -88)
val people = Vector("Bob", "James", "Peter")
```

They support the same operations as lists, with the exception of `::`

Instead of `x :: xs`, there is

`x +: xs`	Create a new vector with leading element `x`, followed by all elements of `xs`.
`xs :+ x`	Create a new vector with trailing element `x`, preceded by all elements of `xs`.

(Note that the `:` always points to the sequence.)

Collection Hierarchy

A common base class of `List` and `Vector` is `Seq`, the class of all *sequences*.

`Seq` itself is a subclass of `Iterable`.

Arrays and Strings

Arrays and Strings support the same operations as `Seq` and can implicitly be converted to sequences where needed.

(They cannot be subclasses of `Seq` because they come from Java)

```
val xs: Array[Int] = Array(1, 2, 3)
xs map (x => 2 * x)

val ys: String = "Hello world!"
ys filter (_.isUpper)
```

Ranges

Another simple kind of sequence is the *range*.

It represents a sequence of evenly spaced integers.

Three operators:

`to` (inclusive), `until` (exclusive), `by` (to determine step value):

```
val r: Range = 1 until 5
val s: Range = 1 to 5
1 to 10 by 3
6 to 1 by -2
```

Ranges a represented as single objects with three fields: lower bound, upper bound, step value.

Some more Sequence Operations:

`xs exists p`	`true` if there is an element `x` of `xs` such that `p(x)` holds, `false` otherwise.
`xs forall p`	`true` if `p(x)` holds for all elements `x` of `xs`, `false` otherwise.
`xs zip ys`	A sequence of pairs drawn from corresponding elements of sequences `xs` and `ys`.
`xs.unzip`	Splits a sequence of pairs `xs` into two sequences consisting of the first, respectively second halves of all pairs.
`xs.flatMap f`	Applies collection-valued function `f` to all elements of `xs` and concatenates the results
`xs.sum`	The sum of all elements of this numeric collection.
`xs.product`	The product of all elements of this numeric collection
`xs.max`	The maximum of all elements of this collection (an `Ordering` must exist)
`xs.min`	The minimum of all elements of this collection

Example: Combinations

To list all combinations of numbers `x` and `y` where `x` is drawn from `1..M` and `y` is drawn from `1..N`:

```
(1 to M) flatMap (x =>
```

Example: Combinations

To list all combinations of numbers `x` and `y` where `x` is drawn from `1..M` and `y` is drawn from `1..N`:

```
(1 to M) flatMap (x => (1 to N) map (y => (x, y)))
```

Example: Scalar Product

To compute the scalar product of two vectors:

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  (xs zip ys).map(xy => xy._1 * xy._2).sum
```

Example: Scalar Product

To compute the scalar product of two vectors:

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  (xs zip ys).map(xy => xy._1 * xy._2).sum
```

An alternative way to write this is with a *pattern matching function value*.

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  (xs zip ys).map{ case (x, y) => x * y }.sum
```

Generally, the function value

```
{ case p1 => e1 ... case pn => en }
```

is equivalent to

```
x => x match { case p1 => e1 ... case pn => en }
```

Exercise:

A number `n` is *prime* if the only divisors of `n` are `1` and `n` itself.

What is a high-level way to write a test for primality of numbers?
For once, value conciseness over efficiency.

```
def isPrime(n: Int): Boolean = ???
```

Exercise:

A number `n` is *prime* if the only divisors of `n` are `1` and `n` itself.

What is a high-level way to write a test for primality of numbers?
For once, value conciseness over efficiency.

```
def isPrime(n: Int): Boolean =
```

Combinatorial Search and For-Expressions

Handling Nested Sequences

We can extend the usage of higher order functions on sequences to many calculations which are usually expressed using nested loops.

Example: Given a positive integer `n`, find all pairs of positive integers `i` and `j`, with `1 <= j < i < n` such that `i + j` is prime.

For example, if `n = 7`, the sought pairs are

i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
$i+j$	3	5	5	7	7	7	11

Collection Hierarchy

A common base class of `List` and `Vector` is `Seq`, the class of all *sequences*.

`Seq` itself is a subclass of `Iterable`.

Algorithm

A natural way to do this is to:

- Generate the sequence of all pairs of integers `(i, j)` such that `1 <= j < i < n`.
- Filter the pairs for which `i + j` is prime.

One natural way to generate the sequence of pairs is to:

- Generate all the integers `i` between `1` and `n` (excluded).
- For each integer `i`, generate the list of pairs `(i, 1)`, `...`, `(i, i-1)`.

This can be achieved by combining `until` and `map`:

```
(1 until n) map (i =>
  (1 until i) map (j => (i, j)))
```

Generate Pairs

The previous step gave a sequence of sequences, let's call it `xss`.

We can combine all the sub-sequences using `foldRight` with `++`:

```
(xss foldRight Seq[Int]())(_ ++ _)
```

Or, equivalently, we use the built-in method `flatten`

```
xss.flatten
```

This gives:

```
((1 until n) map (i =>
  (1 until i) map (j => (i, j)))).flatten
```

Generate Pairs (2)

Here's a useful law:

```
xs flatMap f  =  (xs map f).flatten
```

Hence, the above expression can be simplified to

```
(1 until n) flatMap (i =>
    (1 until i) map (j => (i, j)))
```

Assembling the pieces

By reassembling the pieces, we obtain the following expression:

```
(1 until n) flatMap (i =>
   (1 until i) map (j => (i, j))) filter ( pair =>
      isPrime(pair._1 + pair._2))
```

This works, but makes most people's head hurt.

Is there a simpler way?

For-Expressions

Higher-order functions such as `map`, `flatMap` or `filter` provide powerful constructs for manipulating lists.

But sometimes the level of abstraction required by these function make the program difficult to understand.

In this case, Scala's `for` expression notation can help.

For-Expression Example

Let `persons` be a list of elements of class `Person`, with fields `name` and `age`.

```
case class Person(name: String, age: Int)
```

To obtain the names of persons over 20 years old, you can write:

```
for ( p <- persons if p.age > 20 ) yield p.name
```

which is equivalent to:

```
persons filter (p => p.age > 20) map (p => p.name)
```

The for-expression is similar to loops in imperative languages, except that it builds a list of the results of all iterations.

Syntax of For

A for-expression is of the form

```
for ( s ) yield e
```

where `s` is a sequence of *generators* and *filters*, and `e` is an expression whose value is returned by an iteration.

- A *generator* is of the form `p <- e`, where `p` is a pattern and `e` an expression whose value is a collection.
- A *filter* is of the form `if f` where `f` is a boolean expression.
- The sequence must start with a generator.
- If there are several generators in the sequence, the last generators vary faster than the first.

Instead of `( s )`, braces `{ s }` can also be used, and then the sequence of generators and filters can be written on multiple lines without requiring semicolons.

Use of For

Here are two examples which were previously solved with higher-order functions:

Given a positive integer `n`, find all the pairs of positive integers `(i, j)` such that `1 <= j < i < n`, and `i + j` is prime.

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

Exercise

Write a version of `scalarProduct` (see last session) that makes use of a `for`:

```
def scalarProduct(xs: List[Double], ys: List[Double]) : Double =
```

Exercise

Write a version of `scalarProduct` (see last session) that makes use of a `for`:

```
def scalarProduct(xs: List[Double], ys: List[Double]) : Double =

  (for ((x, y) <- xs zip ys) yield x * y).sum
```

Combinatorial Search Example

Sets

Sets are another basic abstraction in the Scala collections.

A set is written analogously to a sequence:

```
val fruit = Set("apple", "banana", "pear")
val s = (1 to 6).toSet
```

Most operations on sequences are also available on sets:

```
s map (_ + 2)
fruit filter (_.startsWith == "app")
s.nonEmpty
```

(see `Iterables` Scaladoc for a list of all supported operations)

Sets vs Sequences

The principal differences between sets and sequences are:

1. Sets are unordered; the elements of a set do not have a predefined order in which they appear in the set

2. sets do not have duplicate elements:

```
s map (_ / 2)      // Set(2, 0, 3, 1)
```

3. The fundamental operation on sets is `contains`:

```
s contains 5       // true
```

Example: N-Queens

The eight queens problem is to place eight queens on a chessboard so that no queen is threatened by another.

- In other words, there can't be two queens in the same row, column, or diagonal.

We now develop a solution for a chessboard of any size, not just 8.

One way to solve the problem is to place a queen on each row.

Once we have placed `k - 1` queens, one must place the `k`th queen in a column where it's not "in check" with any other queen on the board.

Algorithm

We can solve this problem with a recursive algorithm:

- Suppose that we have already generated all the solutions consisting of placing `k-1` queens on a board of size `n`.
- Each solution is represented by a list (of length `k-1`) containing the numbers of columns (between `0` and `n-1`).
- The column number of the queen in the `k-1`th row comes first in the list, followed by the column number of the queen in row `k-2`, etc.
- The solution set is thus represented as a set of lists, with one element for each solution.
- Now, to place the `k`th queen, we generate all possible extensions of each solution preceded by a new queen:

Implementation

```
def queens(n: Int) = {
  def placeQueens(k: Int): Set[List[Int]] = {
    if (k == 0) Set(List())
    else
      for {
        queens <- placeQueens(k - 1)
        col <- 0 until n
        if isSafe(col, queens)
      } yield col :: queens
  }
  placeQueens(n)
}
```

Exercise

Write a function

```
def isSafe(col: Int, queens: List[Int]): Boolean
```

which tests if a queen placed in an indicated column `col` is secure amongst the other placed queens.

It is assumed that the new queen is placed in the next availabale row after the other placed queens (in other words: in row `queens.length`).

Queries with For

Queries with `for`

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?

- The author is printed once
- The author is printed twice
- The author is printed three times
- The author is not printed at all

Problem

What happens if an author has published three books?

- The author is printed once
- The author is printed twice
- The author is printed three times
- The author is not printed at all

Modified Query (2)

Solution: We must remove duplicate authors who are in the results list twice.

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

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

Modified Query

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 f; s) yield e2
```

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

```
for (x <- e1.withFilter(x => f); 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 list, but instead filters the following `map` or `flatMap` function application.

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

Translate

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

into higher-order functions.

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, XML data, optional values, parsers, 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 the Scala data base connection frameworks ScalaQuery and Slick.

Similar ideas underly Microsoft's LINQ.

Maps

Map

Another fundamental collection type is the *map*.

A map of type `Map[Key, Value]` is a data structure that associates keys of type `Key` with values of type `Value`.

Examples:

```
val romanNumerals = Map("I" -> 1, "V" -> 5, "X" -> 10)
val capitalOfCountry = Map("US" -> "Washington", "Switzerland" -> "Bern")
```

Maps are Iterables

Class `Map[Key, Value]` extends the collection type `Iterable[(Key, Value)]`.

Therefore, maps support the same collection operations as other iterables do. Example:

```
val countryOfCapital = capitalOfCountry map {
  case(x, y) => (y, x)
}                        // Map("Washington" -> "US", "Bern" -> "Switzerland")
```

Note that maps extend iterables of key/value *pairs*.

In fact, the syntax `key -> value` is just an alternative way to write the pair `(key, value)`.

Maps are Functions

Class `Map[Key, Value]` also extends the function type `Key => Value`, so maps can be used everywhere functions can.

In particular, maps can be applied to key arguments:

```
capitalOfCountry("US")          // "Washington"
```

Querying Map

Applying a map to a non-existing key gives an error:

```
capitalOfCountry("Andorra")
  // java.util.NoSuchElementException: key not found: Andorra
```

To query a map without knowing beforehand whether it contains a given key, you can use the `get` operation:

```
capitalOfCountry get "US"      // Some("Washington")
capitalOfCountry get "Andorra" // None
```

The result of a `get` operation is an `Option` value.

The `Option` Type

The `Option` type is defined as:

```
trait Option[+A]
case class Some[+A](value: A) extends Option[A]
object None extends Option[Nothing]
```

The expression `map get key` returns

- `None` if `map` does not contain the given `key`,
- `Some(x)` if `map` associates the given `key` with the value `x`.

Decomposing Option

Since options are defined as case classes, they can be decomposed using pattern matching:

```
def showCapital(country: String) = capitalOfCountry.get(country) match {
  case Some(capital) => capital
  case None => "missing data"
}

showCapital("US")      // "Washington"
showCapital("Andorra") // "missing data"
```

Options also support quite a few operations of the other collections.

I invite you to try them out!

Sorted and GroupBy

Two useful operation of SQL queries in addition to for-expressions are `groupBy` and `orderBy`.

`orderBy` on a collection can be expressed by `sortWith` and `sorted`.

```
val fruit = List("apple", "pear", "orange", "pineapple")
fruit sortWith (_.length < _.length)  // List("pear", "apple", "orange", "pineapple")
fruit.sorted      // List("apple", "orange", "pear", "pineapple")
```

`groupBy` is available on Scala collections. It partitions a collection into a map of collections according to a *discriminator function* `f`.

Example:

```
fruit groupBy (_.head)  //> Map(p -> List(pear, pineapple),
                        //|     a -> List(apple),
                        //|     o -> List(orange))
```

Map Example

A polynomial can be seen as a map from exponents to coefficients.

For instance, $x^3 - 2x + 5$ can be represented with the map.

```
Map(0 -> 5, 1 -> -2, 3 -> 1)
```

Based on this observation, let's design a class `Polynom` that represents polynomials as maps.

Default Values

So far, maps were *partial functions*: Applying a map to a key value in `map(key)` could lead to an exception, if the key was not stored in the map.

There is an operation `withDefaultValue` that turns a map into a total function:

```
val cap1 = capitalOfCountry withDefaultValue "<unknown>"
cap1("Andorra")             // "<unknown>"
```

Variable Length Argument Lists

It's quite inconvenient to have to write

```
Polynom(Map(1 -> 2.0, 3 -> 4.0, 5 -> 6.2))
```

Can one do without the `Map(...)`?

Problem: The number of `key -> value` pairs passed to `Map` can vary.

Variable Length Argument Lists

It's quite inconvenient to have to write

```
Polynom(Map(1 -> 2.0, 3 -> 4.0, 5 -> 6.2))
```

Can one do without the `Map(...)`?

Problem: The number of `key -> value` pairs passed to `Map` can vary.

We can accommodate this pattern using a *repeated parameter*:

```
def Polynom(bindings: (Int, Double)*) =
  new Polynom(bindings.toMap withDefaultValue 0)

Polynom(1 -> 2.0, 3 -> 4.0, 5 -> 6.2)
```

Inside the `Polynom` function, `bindings` is seen as a `Seq[(Int, Double)]`.

Final Implementation of Polynom

```
class Poly(terms0: Map[Int, Double]) {
  def this(bindings: (Int, Double)*) = this(bindings.toMap)
  val terms = terms0 withDefaultValue 0.0
  def + (other: Poly) = new Poly(terms ++ (other.terms map adjust))
  def adjust(term: (Int, Double)): (Int, Double) = {
    val (exp, coeff) = term
    exp -> (coeff + terms(exp))
  }

override def toString =
  (for ((exp, coeff) <- terms.toList.sorted.reverse)
    yield coeff+"x^"+exp) mkString " + "
}
```

Exercise

The + operation on `Poly` used map concatenation with ++. Design another version of + in terms of `foldLeft`:

```
def + (other: Poly) =
  new Poly((other.terms foldLeft ???)(addTerm)

def addTerm(terms: Map[Int, Double], term: (Int, Double)) =
  ???
```

Which of the two versions do you believe is more efficient?

- 0 The version using ++
- 0 The version using foldLeft

Exercise

The + operation on `Poly` used map concatenation with ++. Design another version of + in terms of `foldLeft`:

```
def + (other: Poly) =
  new Poly((other.terms foldLeft ???)(addTerm)

def addTerm(terms: Map[Int, Double], term: (Int, Double)) =
  ???
```

Which of the two versions do you believe is more efficient?

- 0 The version using ++
- 0 The version using foldLeft

Putting the Pieces Together

Task

Phone keys have 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 `translate` such that

```
translate(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.

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

The Future?

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 a very attractive tool for software development