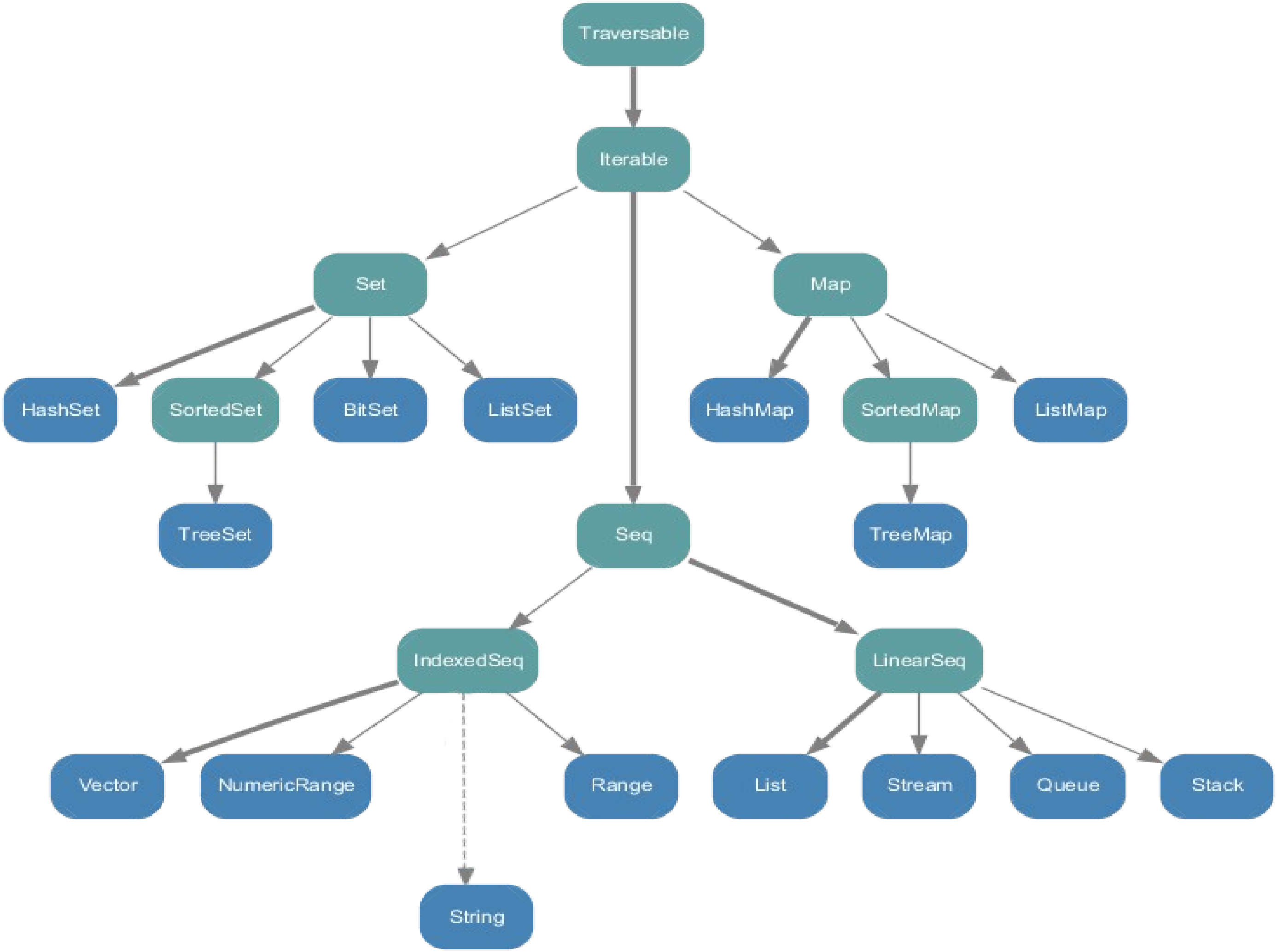
Programming with Collections



Lists

Scala lists

- Lists are built-in in Scala (defined in the standard library)

```
val fruit = List("apples", "oranges", "pears")
val nums = List(1, 2, 3, 4)
val diag3 = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty = List()
```

  - List vs Array:
    - Lists are immutable
    - Lists are recursive while Array are flat
    - Both lists and array are homogeneous (the contained elements must all have the same type)
      - The type of a list with elements of type T is List[T]

List constructors

- All lists are constructed from:
  - The empty list Nil
  - The construction operation ::
    - x :: xs constructs a list with first element x and rest of the list xs

```
val fruit = "apples" :: "oranges" :: "pears" :: Nil
val nums = 1 :: 2 :: 3 :: 4 :: Nil
val empty = Nil
```

    - :: is right associative
- Basic List methods: head, tail, and isEmpty (many additional methods, like length, ...)

```
fruit.head  == "apples
fruit.tail.head == "oranges"
diag3.head == List(1, 0, 0)
empty.head == throw new NoSuchElementException ("head of empty list")
fruit.length == 3
```

Exercise

- Define a isort function that sorts a list of integers according to the insertion sort algorithm

Exercise

- Define a isort function that sorts a list of integers according to the insertion sort algorithm

```
def isort(xs: List[Int]): List[Int] = xs match {
  case List() => List()
  case y :: ys => insert(y, isort(ys))
}
```

Exercise

- Define a isort function that sorts a list of integers according to the insertion sort algorithm

```
def isort(xs: List[Int]): List[Int] = xs match {
  case List() => List()
  case y :: ys => insert(y, isort(ys))
}

def insert(x: Int, xs: List[Int]): List[Int] = xs match {
  case List() => List(x)
  case y :: ys => if (x < y) x :: xs else y :: insert(x, ys)
}
```

Exercise

- Complete the following implementation of merge-sort

```
def msort(xs: List[Int]): List[Int] = {
  val n = xs.length/2
  if (n == 0) xs else
  {
    def merge(xs: List[Int], ys: List[Int]) = ???
    val (fst, snd) = xs splitAt n
    merge(msort(fst), msort(snd))
  }
}
```

Exercise

- Complete the following implementation of merge-sort

```scala
def merge(xs: List[Int], ys: List[Int]): List[Int] =
  xs match {
    case Nil => ys
    case x :: xs1 =>
      ys match {
        case Nil => xs
        case y :: ys1 =>
          if (x < y) x :: merge(xs1, ys)
          else y :: merge(xs, ys1)
      }
  }
```

Pairs and Tuples

- Notice the method splitAt(n) on lists that returns a pair of lists (the first n elements and the subsequent ones)

```
val (fst, snd) = xs splitAt n
```

- Example of pairs:

```
val pair = ("answer", 42)
val (label, value) = pair //label: String = answer
                          //value: Int = 42
```

- In general, Scala supports tuples up to 22 elements
  - With selector methods _1, _2,..., _22

Exercise

- Rewrite the merge function using pairs
  - to reflect symmetry of the merge operation

```
def merge(xs: List[Int], ys: List[Int]): List[Int] =
  (xs, ys) match {

      ...

  }
```

Exercise

- Rewrite the merge function using pairs
  - to reflect symmetry of the merge operation

```
def merge(xs: List[Int], ys: List[Int]): List[Int] =
  (xs, ys) match {
    case (Nil, _) => ys
    case (_, Nil) => xs
    case (x :: xs1, y :: ys1) =>
          if (x < y) x :: merge(xs1, ys)
          else y :: merge(xs, ys1)
  }
```

Higher Order List functions

Higher-order List functions

- Recurrent patterns of computation on lists can be programmed once-for all as higher-order functions
  - As an example consider transforming all the elements of a list
    - For instance by applying a scaling factor:

```
def scaleList(xs: List[Double], factor: Double): List[Double] =
  xs match {
    case Nil => xs
    case y :: ys => y * factor :: scaleList(ys, factor)
}
```

  - This is an instance of the map pattern, programmed once for all in the standard library:

```
abstract class List[+T] {
  ...
  def map[U](f: T => U): List[U] = this match {
    case Nil => this
    case x :: xs => f(x) :: xs.map(f) }
  ...
}
```

Exercise

- Rewrite the scaleList function as an instance of the map higher-order function

Exercise

- Rewrite the scaleList function as an instance of the map higher-order function

```
def scaleList(xs: List[Double], factor: Double) =
  xs map (x => x * factor)
```

Exercise

- Write a function that squares each element in a list in two different ways:
  - Without using and by using the map higher-order function

Exercise

- Write a function that squares each element in a list in two different ways:
  - Without using and by using the map higher-order function

```
def squareList(xs: List[Int]): List[Int] =
  xs match {
    case Nil => Nil
    case y :: ys => y * y :: squareList(ys)
  }
```

```
def squareList(xs: List[Int]) =
  xs map (x => x * x)
```

Filter

- Another pattern is the selection of all elements in a list satisfying a given condition
  - For example:

```
def posElems (xs: List[Int]): List[Int] =
  xs match {
    case Nil => xs
    case y :: ys => if (y > 0) y :: posElems(ys) else posElems(ys)
}
```

  - This is an instance of the filter pattern

```
abstract class List[+T] {
  ...
  def filter(p: T => Boolean): List[T] = this match {
    case Nil => this
    case x :: xs => if (p(x)) x :: xs.filter(p) else xs.filter(p)
  }
  ...
}
```

Exercise

- Rewrite the posElems function as an instance of the filter higher-order function

```
def posElems(xs: List[Int]): List[Int] =
  xs filter (x => (x > 0 ))
```

Variations of filter

- There are other functions for extraction of sublists
  - `xs filterNot p` Same as `xs filter (x => !p(x))`
  - `xs partition p` Same as `(xs filter p, xs filterNot p)`, but computed in a single traversal
  - `xs takeWhile p` Longest prefix of `xs` of elements satisfying the predicate `p`
  - `xs dropWhile p` Remainder of `xs` after elimination of leading elements satisfying `p`
  - `xs span p` Same as `(xs takeWhile p, xs dropWhile p)`, but computed in a single traversal

Exercise

- Write a function `pack` that packs consecutive duplicates of the same elements into sublists
  - For instance

```
pack(List("a","a","a","b","c","c","a"))
```

  - should give

```
List(List(a, a, a), List(b), List(c, c), List(a))
```

Exercise

- Write a function `pack` that packs consecutive duplicates of the same elements into sublists
  - For instance

    ```
    pack(List("a","a","a","b","c","c","a"))
    ```
  - should give

    ```
    List(List(a, a, a), List(b), List(c, c), List(a))
    ```

```
def pack[T](xs: List[T]): List[List[T]] =
  xs match {
    case Nil => Nil
    case _ => (xs span (x => (x == xs.head))) match {
      case (l, r) => l :: pack(r)
    }
}
```

Exercise

- Using `pack` write a function `encode` that encodes a list by reporting the sequence of elements with the number of their consecutive repetitions
  - For instance

```
encode(List("a","a","a","b","c","c","a"))
```

  - should give

```
List((a,3), (b,1), (c,2), (a,1))
```

Exercise

- Using `pack` write a function `encode` that encodes a list by reporting the sequence of elements with the number of their consecutive repetitions
  - For instance

    ```
    encode(List("a","a","a","b","c","c","a"))
    ```

  - should give

    ```
    List((a,3), (b,1), (c,2), (a,1))
    ```

```
def encode[T] (xs:List[T]) =
  pack(xs) map (l => (l.head, l.length))
```

List element combination

- Another typical pattern is to compute new values as combination of the elements of a list
  - sum(List(x1, ...,xn)) = 0 + x1 + … + xn
  - product(List(x1,...,xn)) = 1 * x1 * … * xn
- We can compute these kinds of functions using the usual recursive schema:

```
def sum  (xs: List[Int]): Int = xs match {
  case Nil => 0
  case y :: ys => y + sum(ys)
}
```

  - Notice that in this implementation, we assume the operator + right-associative, ie. sum computes x1+(… +(xn+0)...)
  - We will start by considering standard left-associative higher-order combination functions , ie. (...(0+x1)+ …)+xn

reduceLeft

- This left-associative combination pattern is available as the reduceLeft higher-order method on lists:
  - List(x1, ..., xn) reduceLeft op = (...(x1 op x2) op ...) op xn
- We can instanciate sum and product as follows from reduceLeft as follows

```
def sum(xs: List[Int]) =
  (0 :: xs) reduceLeft ((x, y) => x + y)

def product(xs: List[Int]) =
  (1 :: xs) reduceLeft ((x, y) => x * y)
```

foldLeft

- Scala library contains another list method, foldLeft, that works with an additional parameter
  - (List(x1, ...,xn) foldLeft z)(op) = (...(z op x1) op ...) op x
  - In this way, we can explicitly indicate an initial value to be used in the combination of all list elements

```
def sum(xs: List[Int]) = (xs foldLeft 0) (_ + _)

def product(xs: List[Int]) = (xs foldLeft 1) (_ * _)
```

  - (_+_) (respectively (_*_)) is equivalent to ((x, y) => x + y) (respectively ((x, y) => x * y))

reduceLeft / foldLeft implementation

```
abstract class List[+T] {
...
  def reduceLeft[U >: T](op: (U, T) => U): U = this match {
    case Nil => throw new Error("Nil.reduceLeft")
    case x :: xs => (xs foldLeft x)(op)
  }

  def foldLeft[U](z: U)(op: (U, T) => U): U = this match {
    case Nil => z
    case x :: xs => (xs foldLeft op(z, x))(op)
  }
...
}
```

reduceRight and foldRight

- For list elements combinations that are right associative it is possible to use reduceRight and foldRight
  - List(x1, ...,x{n-1}, xn) reduceRight op = x1 op (...(x{n-1} op xn) ...)
  - (List(x1, ...,xn) foldRight z)(op) = x1 op (...(xn op z)...)

```
abstract class List[+T] {
...
  def reduceRight[U >: T](op: (T, U) => U): U = this match {
    case Nil => throw new Error("Nil.reduceRight")
    case x :: Nil => x
    case x :: xs => op(x, xs.reduceRight(op))
  }

  def foldRight[U](z: U)(op: (T, U) => U): U = this match {
    case Nil => z
    case x :: xs => op(x, (xs foldRight z)(op))
  }
...
}
```

Differences between foldLeft and foldRight

- For operators that are associative foldLeft and foldRight return the same value
- Sometimes, only one is appropriate
  - Exercise: concatenation of two lists

```
def concat[T](xs: List[T], ys: List[T]): List[T] =
  ...
```

Differences between foldLeft and foldRight

- For operators that are associative foldLeft and foldRight return the same value
- Sometimes, only one is appropriate
  - Exercise: concatenation of two lists

```
def concat[T](xs: List[T], ys: List[T]): List[T] =
  (xs foldRight ys) (_ :: _)
```

  - What happens if we replace foldRight with foldLeft ?

Exercise

- Implement a function that reverses the elements in a list by using foldLeft or foldRight

Exercise

- Implement a function that reverses the elements in a list by using foldLeft or foldRight

```
def reverse[T](xs: List[T]): List[T] =
  (xs foldLeft List[T]())((xs, x) => x :: xs)
```

  - The empty list is constructed with List[T]() to indicate that the accumulator should be of type List[T]
    - With Nil the accumulator is wrongly inferred to be of type Nil.type

Exercise

- Complete the following definitions by using foldRight and/or foldLeft

```
def mapFun[T, U](xs: List[T], f: T => U): List[U] =
  ...


def lengthFun[T](xs: List[T]): Int =
  ...
```

Exercise

- Complete the following definitions by using foldRight and/or foldLeft

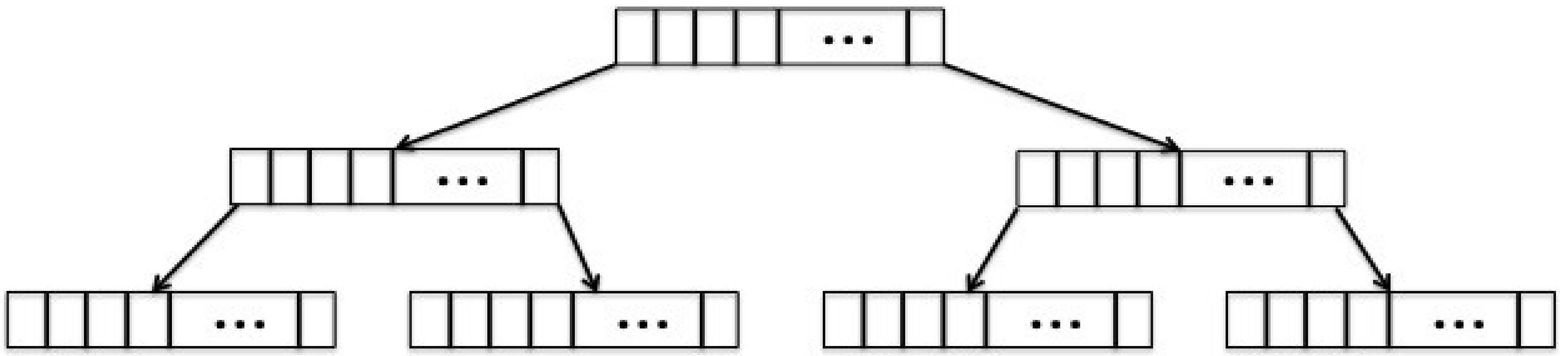
```scala
def mapFun[T, U](xs: List[T], f: T => U): List[U] =
  (xs foldRight List[U]())((x, xs) => f(x) :: xs )


def lengthFun[T](xs: List[T]): Int =
  (xs foldRight 0)((x,n) => n+1)
```

Other Collections

Scala Vectors

- Vectors are linear structures with a balanced access
  - In lists the access to the first element is faster than the access to the last element
- Vectors are implemented as trees with degree 32



$32 = 2^5$

$32^2 = 2^{10}$

$32^3 = 2^{15}$

- Vectors are immutable as lists
  - New vectors can be created by keeping the immuted part
  - E.g. v :+ x, that appends x to vector v, creates a new last object of size 32 (that will include also x) and its ancestors until a new root, that simply replace the changed objects (of size 32)

Vector operations

- 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 that creates a new vector with leading element x, followed by all elements of xs
  - xs :+ x creates a new vector with trailing element x, preceded by all elements of xs
  - Note that the : always points to the sequence
- There are many additional operations exploiting indexing:
  - e.g. `v.updated(i,x)` that generates a copy of v, with the element at place i replaced by x

Range

- 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)
- Examples:
  - `val r: Range = 1 until 5`
  - `val s: Range = 1 to 5`
  - `1 to 10 by 3`
  - `6 to 1 by -2`
- Represented as objects with three fields:
  - lower bound, upper bound, step value.

Seq: common interface for List, Vector, Range, ...

- `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 o.w.
- `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 and second halves of all pairs
- `xs flatMap f` applies a function f returning a collection 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 (the contained type must extend the Ordered trait)
- `xs.min` the minimum of all elements of this collection

Exercise

- Define a function that generates all pairs in the cartesian product of (1..N) and (1..M)

```
def cartProduct(M:Int, N:Int): Seq[(Int,Int)] = {
  ...
}
```

Exercise

- Define a function that generates all pairs in the cartesian product of (1..N) and (1..M)

```
def cartProduct(M:Int, N:Int): Seq[(Int,Int)] = {
  (1 to N) flatMap (x => (1 to M) map (y => (y,x)))
}
```

Exercise

- Define a function that computes the scalar product of two vectors
  - i.e. the sum of the pointwise products

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

Exercise

- Define a function that computes the scalar product of two vectors
  - i.e. the sum of the pointwise products

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

Exercise

- Define a function that checks whether a given integer is a prime number

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

Exercise

- Define a function that checks whether a given integer is a prime number

```
def isPrime(n: Int): Boolean =
  (2 until n) forall (d => (n%d != 0))
```

A flavour of imperative programming

- In imperative programming, it is typical to write loops to traverse sequences of interesting values
- Example: compute the set of pairs of integers between 1 and N having a sum which is prime
  - No repetitions, i.e., only one between (2,5) and (5,2)
  - In imperative programming, two nested loops can be used to produce all pairs, and then a check is done on their sum
  - Similarly, in Scala we can write:

```
(1 to 20) flatMap (i => (
    (1 to i) filter (j => isPrime(i+j)) map (j =>
        (i, j)
      )
    )
  )
```

For-expressions

- This is a typical programming pattern, that is why Scala has an ad-hoc syntax

```
for {
  i <- 1 to 10
  j <- 1 to i
  if isPrime (i + j)
} yield (i, j)
```

- A for-expression is of the form for (s) yield e with
  - s sequence of generators (like `i <- 1 to 10`) and filters (like `if isPrime (i + j)`)
    - the sequence must start with a generator
  - and e is an expession generating the single elements of the produced collection

For-expressions

- The for-expression in Scala is syntactic sugar:
  - it is translated in terms of map, flatMap and withFilter (a variant of filter)

```
for {
  i <- 1 to 10
  j <- 1 to i
  if isPrime (i + j)
} yield (i, j)
```

is translated to:

```
(1 to 20) flatMap (i => (
    (1 to i) withFilter (j => isPrime(i+j)) map (j =>
        (i, j)
      )
    )
  )
```

Exercise

- Re-define scalar product, exploiting a for-expression

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

Exercise

- Re-define scalar product, exploiting a for-expression

```
def scalarProduct(xs: Vector[Double],
                  ys: Vector[Double]) : Double =
  (for ((x, y) <- xs zip ys) yield x * y).sum
```

For-expressions as queries

- With for expressions you can express complex and structured queries on collections

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

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"))
)
```

For-expressions as queries

- With for expressions you can express complex and structured queries on collections

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

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

Sets and Maps

Sets

- Another iterable collection is Set:

```
val fruit = Set("apple", "banana", "pear")
fruit filter (_.startsWith("app"))
fruit.nonEmpty
```

- The main differences with Seq are:
  - Sets are unordered
  - Sets do not have duplicates

    ```
    Set(8,5,7,4) map (_ / 2)  // = Set(4, 2, 3)
    ```

  - Basic operations on sets:

    ```
    fruit contains "apple"     // = true
    fruit + "strawberry"        // add one element
    fruit ++ Set("strawberry","kiwi")     // union
    ```

  Note: ++ is union for all traversable collections

Exercise

- Consider the N-queens problem. Design a recursive solution with the following structure.

```
def queens(n: Int) = {
  def placeQueens(k: Int): Set[Vector[Int]] = {
    if (k == 0) Set(Vector())
    else


      ...


  }
  placeQueens(n)
}
```

Exercise

- Consider the N-queens problem. Design a recursive solution with the following structure.

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

Exercise

- Consider the N-queens problem. Design a recursive solution with the following structure.

```
...
def isSafe(col: Int, queens: Vector[Int]): Boolean = {
  val row = queens.length
  val queensWithRows =
    (0 until queens.length) zip queens
  queensWithRows forall (p =>
    (col != p._2) &&
    (math.abs(col - p._2) != row - p._1)
  )
}
```

Maps

- A map of type Map[Key, Value] associates key of type Key with values of type Value
  - Examples:

    ```
    val romanNumerals = Map(”I”->1,”V”->5,”X”->10)
    val capitalOfCountry =
      Map(”US”->“Washington”,”Italy”->”Rome”)
    ```

- Map extends Iterable, hence it provides all the methods in the Iterable API

```
val countryOfCapital =
      capitalOfCountry map (p => (p._2,p._1))
```

  - Notice that a map contains a set of pairs:
    - the notation K->V is equivalent to (K,V)

Maps are (partial) functions

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

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

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

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

- The operation `withDefaultValue` turns a map into a total function:

```
val totalCapitalOfCountry =
    capitalOfCountry withDefaultValue "unknown"
```

The Option[T] type

- Maps can be accessed using the get methods

  `capitalOfCountry get “US” // Some(“Washington”)`

  `capitalOfCountry get “France” // None`

- `Some(X)` (with `X` of type `T`) and `None` are the values populating the type `Option[T]`
  - `Option[T]` is used when values could be undefined
    - Other languages (e.g. Java) use `null` to denote undefined values (risk of null pointer exceptions)
    - `Option[T]` helps the programmer to remember to check for undefined values:

```
capitalOfCountry get "France" match {
  case Some(x) => println(x)
  case None => println("no value")
}
```

Example

- We will use maps to implement polynomials
  - The polynomial $2x + 4x^3 + 6.2x^5$ can be naturally represented as `Polynom(Map(1->2.0, 3->4.0, 5->6.2))`

```
class Polynom(terms0: Map[Int, Double])
{
  def this(bindings: (Int, Double)*) = this(bindings.toMap)
  val terms = terms0 withDefaultValue 0.0
  def +(other: Polynom) =
    new Polynom(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)
      yield coeff + "x^" + exp) mkString " + "
}
```

Repeated parameters

- With notation Type* it is possible to denote a variable number of parameters of type Type
  - Inside the function treated as a Seq[Type]

```
class Polynom(terms0: Map[Int, Double])
{
  def this(bindings: (Int, Double)*) = this(bindings.toMap)
  val terms = terms0 withDefaultValue 0.0
  def +(other: Polynom) =
    new Polynom(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)
      yield coeff + "x^" + exp) mkString " + "
}
```

Make the map total

- It is convenient to make the map representing the polynomial total, so that on a polynomial `p` we can invoke `p.terms(e)` for every possible exponential `e`

```
class Polynom(terms0: Map[Int, Double])
{
  def this(bindings: (Int, Double)*) = this(bindings.toMap)
  val terms = terms0 withDefaultValue 0.0
  def +(other: Polynom) =
    new Polynom(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)
      yield coeff + "x^" + exp) mkString " + "
}
```

Map concatenation

- Concatenation among maps gives priority to the right hand operand in case of duplicated keys

```
class Polynom(terms0: Map[Int, Double])
{
  def this(bindings: (Int, Double)*) = this(bindings.toMap)
  val terms = terms0 withDefaultValue 0.0
  def +(other: Polynom) =
    new Polynom(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)
      yield coeff + "x^" + exp) mkString " + "
}
```

Exercise

- Rewrite method + by using foldLeft instead of concatenation ++

Exercise

- Rewrite method + by using foldLeft instead of concatenation ++

```
class Polynom(terms0: Map[Int, Double])
{
  ...
  def +(other: Polynom) =
    new Polynom((other.terms foldLeft terms)(addTerm))
  def addTerm(terms: Map[Int, Double], term:(Int, Double)):
    Map[Int, Double] = {
    val (exp, coeff) = term
    terms + (exp -> (coeff + terms(exp)))
  }
  ...
}
```

GroupBy

- It is possible to partition collections depending on the value returned by a function applied to all elements

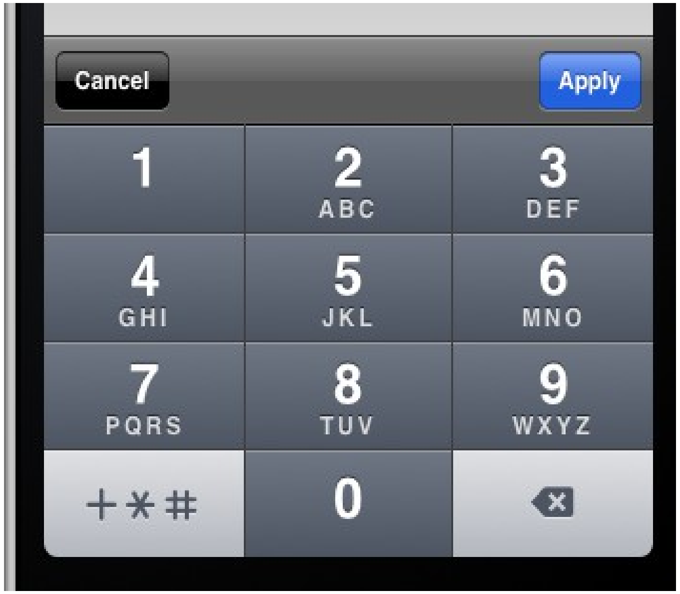
```
val donuts: Seq[(String,Double)] = Seq(
("Plain Donut",2.5), ("Strawberry Donut",4.2), ("Glazed Donut",3.3),
("Plain Donut",2.8), ("Glazed Donut",3.1) )

donuts groupBy (_._1)
// Map(Glazed Donut -> List((Glazed Donut,3.3), (Glazed Donut,3.1)),
//     Plain Donut -> List((Plain Donut,2.5), (Plain Donut,2.8)),
//     Strawberry Donut -> List((Strawberry Donut,4.2)))
```

  - `groupBy` returns a Map:
    - the key is a value in the field of the function
    - the value is the partition of the collection with the elements returning that value

Programming with Collections

- There exists a traditional way to associate letters to digits:



- Problem: write a program that, given a sequence of digits, returns a sequence of possible words taken from a given dictionary
  - Ex. `7225247386` can generate `scala is fun`

Programming with Collections

- This problem has been proposed as a benchmark for programming language comparison in
  - 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
- We solve the problem in Scala adopting a dictionary available at:
  - http://cs.unibo.it/~zavattar/words.txt

```scala
import scala.io.Source
val in = Source.fromURL("http://cs.unibo.it/zavattar/words.txt")
val word = in.getLines.toList filter (w => w forall (c => c.isLetter))

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

val charCode: Map[Char, Char] =
  for {
    (digit, str) <- mnem
    ltr <- str
  } yield ltr -> digit

def wordCode (word: String): String =
  word.toUpperCase map charCode

val wordsForNum: Map[String, Seq[String]] =
  word groupBy wordCode withDefaultValue Seq()

def encode (number: String): Seq[List[String]] =
  if (number.isEmpty) Seq(List())
  else {
    for {
      split <- 1 to number.length
      word <- wordsForNum(number take split)
      rest <- encode(number drop split)
    } yield word :: rest
  }
```

Streams

Exercise

- Compute the second prime number in the interval between 1000 and 10000

Exercise

- Compute the second prime number in the interval between 1000 and 10000

```
((1000 to 10000) filter isPrime)(1)
```

Exercise

- Compute the second prime number in the interval between 1000 and 10000

```
((1000 to 10000) filter isPrime)(1)
```

- This solution is not efficient, why?

Exercise

- Compute the second prime number in the interval between 1000 and 10000

```
((1000 to 10000) filter isPrime)(1)
```

- This solution is not efficient, why?
  - All the prime numbers in the interval are computed!
- Solution:
  - Use a stream instead of a list
  - Streams are like list, but the tail is evaluated only if and when it is needed

Streams

- Streams are defined from a constant Stream.empty and a constructor Stream.cons

```
val xs  = Stream.cons(1, Stream.cons(2, Stream.empty))
```

  - Stream.cons is similar to the list constructor :: but it does not evaluate immediately the second argument (like in CBN)
    - Given the similarity, there is a special notation #:: equivalent to Stream.cons
- Streams can also be defined like the other collections

```
Stream(1, 2, 3)
```

- The toStream method on a collection will turn the collection into a stream

```
(1 to 1000).toStream    // res0: Stream[Int] = Stream(1, ?)
```

Stream implementation

```
trait Stream[+A] extends Seq[A] {
  def isEmpty: Boolean
  def head: A
  def tail: Stream[A]
  ...
}

object Stream {
  def cons[T](hd: T, tl: => Stream[T]) = new Stream[T] {
    def isEmpty = false
    def head = hd
    def tail = tl
    ...
  }
  val empty = new Stream[Nothing] {
    def isEmpty = true
    def head = throw new NoSuchElementException("empty.head")
    def tail = throw new NoSuchElementException("empty.tail")
    ...
  }
}
```

The second parameter is evaluated only upon access to the tail method

The alternative solution with streams

- Compute the second prime number in the interval between 1000 and 10000

```
((1000 to 10000).toStream filter isPrime)(1)
```

- This solution is much more efficient, why?

```
trait Stream[+T] {
  ...
  def filter(p: T => Boolean): Stream[T] =
    if (isEmpty) this
    else if (p(head)) cons(head, tail.filter(p))
    else tail.filter(p)
  ...
}
```

Lazy evaluation

- The (simplified) implementation of streams that we have shown could be inefficient:
  - The tail of the stream is re-evaluated every time it is accessed
- This is avoided in the actual implementation by adopting lazy evaluation
  - The tail is computed at the first access and memorized for successive accesses
- Lazy evaluation is supported in Scala as follows:

```
def expr = {
  val x = { print("x"); 1 }
  lazy val y = { print("y"); 2 }
  def z = { print("z"); 3 }
  z + y + x + z + y + x
}
expr                    // ????
```

Lazy evaluation

- The (simplified) implementation of streams that we have shown could be inefficient:
  - The tail of the stream is re-evaluated every time it is accessed
- This is avoided in the actual implementation by adopting lazy evaluation
  - The tail is computed at the first access and memorized for successive accesses
- Lazy evaluation is supported in Scala as follows:

```scala
def expr = {
  val x = { print("x"); 1 }
  lazy val y = { print("y"); 2 }
  def z = { print("z"); 3 }
  z + y + x + z + y + x
}
expr                        // print xzyz
```

Infinite streams

- Lazy evaluation allows for the definition of streams with infinitely many values

```
def from(n: Int): Stream[Int] = n #:: from(n+1)

val nats = from(0)

nats map (_ * 4)

def sieve(s: Stream[Int]): Stream[Int] =
  s.head #:: sieve(s.tail filter (_ % s.head != 0))

val primes = sieve(from(2))
```

Square root revisited

- We can use lazy evaluation to program the computation of square roots by computing (in a lazy way) the full infinite sequence of approximated solutions

```
def sqrtStream(x: Double): Stream[Double] = {
  def improve(guess: Double) = (guess + x / guess) / 2
  lazy val guesses: Stream[Double] =
    1 #:: (guesses map improve)
  guesses
}

def isGoodEnough(guess: Double, x  : Double) =
  math.abs((guess * guess - x) / x) < 0.0001

(sqrtStream(2) filter (isGoodEnough(_, 2)))
```