Maps

Principles of Functional Programming

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((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)`. (-> implemented as an extension method in `Predef`).

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!

Updating Maps

Functional updates of a map are done with the + and ++ operations:

`m + (k -> v)`	The map that takes key ‘k‘ to value ‘v‘ and is otherwise equal to ‘m‘
`m ++ kvs`	The map ‘m‘ updated via ‘+‘ with all key/value pairs in ‘kvs‘

These operations are purely functional. For instance,

```
val m1 = Map("red" -> 1, "blue" -> 2)     > m1 = Map(red -> 1, blue -> 2)
val m2 = m1 + ("blue" -> 3)               > m2 = Map(red -> 1, blue -> 3)
m1                                        > Map(red -> 1, blue -> 2)
```

Sorted and GroupBy

Two useful operations known from SQL queries are `groupBy` and `orderBy`.

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

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

`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)*) =
  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 Polynom(nonZeroTerms: Map[Int, Double]):
  def this(bindings: (Int, Double)*) = this(bindings.toMap)

  def terms = nonZeroTerms.withDefaultValue(0.0)
  def + (other: Polynom) =
    Polynom(terms ++ other.terms.map((exp, coeff) => (exp, terms(exp) + coeff)))

  override def toString =
    val termStrings =
      for (exp, coeff) <- terms.toList.sorted.reverse
      yield
        val exponent = if exp == 0 then "" else s"x^$exp"
        s"$coeff$exponent"
    if terms.isEmpty then "0" else termStrings.mkString(" + ")
```

Exercise

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

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

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

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

```
O         The version using ++
O         The version using foldLeft
```

Exercise

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

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

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

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

```
O         The version using ++
O         The version using foldLeft
```

Exercise

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

```
def + (other: Polynom) =
  Polynom(other.terms.foldLeft(terms)(addTerm))

def addTerm(terms: Map[Int, Double], term: (Int, Double)) =
  val (exp, coeff) = term
  terms + (exp, coeff + terms(exp))
```

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

```
  O        The version using ++
  X        The version using foldLeft
```

Exercise

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

```
def + (other: Polynom) =
  Polynom(other.terms.foldLeft(terms)(addTerm))

def addTerm(terms: Map[Int, Double], term: (Int, Double)) =
  val (exp, coeff) = term
  terms + (exp, coeff + terms(exp))
```

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

```
O          The version using ++
X          The version using foldLeft
```