DSL processing in Scala

July 26, 2015

I was recently presenting with the following programming puzzle, which I decided to try to solve in scala (scala is still very new to me, so I took it as a challenge to help my learning):

> *Start with an empty matrix (two-dimensional array) sized N by N with all cells set to zero.*
> *Simulate population growth in a given range of cells by adding, multiplying, or raising to a power the current cell population by a given factor.*
>
> *Those instructions come in the form:*
>
> - `x1:x2;y1:y2;+i` *: add i to the current population in each cell defined by matrix(x, y) where x1 <= x <= x2 and y1 <= y <= y2*
> - `x1:x2;y1:y2;*m` *: multiply the current population in each cell by m*
> - `x1:x2;y1:y2;^n` *: raise the current population in each cell by a power of n*
>
> *The entire simulation instruction takes the form of a single space separated string. The very first part is an integer, specifying the size of the matrix, and the rest of the string is one or more growth instructions also separated by spaces.*
>
> *Here's an example:*
>
> `"10 1:2;2:3;+1 2:2;3:3;+5 7:9;5:8;^3"`
>
> *The program should take the sum of all the cells in the matrix after all the growth instructions have been processed.*
>
> *For the example above, the expected answer is 9.*

I started by defining regular expression patterns for the expected inputs.

This describes the overall instruction string in two parts, the initial integer for the matrix size, along with the rest of the growth instructions:

```
val argsPattern = """^(\d+)\s(.*)""".r
```

Inside a main() method, it can be used like this:

```
args(0) match {
  case argsPattern (n, rest) => println(getPopulation(setPopulation(n.toInt, rest.split(" "))))
  case _ => println("Invalid input")
}
```

Even though scala has elegant exception handling in the form of Option/Some and Try/Success/Failure constructs, we can go ahead and do `n.toInt` confidently because that branch of the match statement won't get executed unless there is a leading number in the given string.

Next, we need a regex for the possible growth instructions. This one covers all three cases:

```
val inputPattern = """(\d+:\d+);(\d+:\d+);((\^|\+|\*)\d+)""".r
```

While inputPattern can tell us whether or not a given string is a valid growth instruction, we should have separate functions to handle the different types of mathematical operations.

Since functions can be passed as first-class objects in scala, we'll use these three curried functions, where b is the value found in the instruction string, and a represents the current cell population, input at runtime:

```
def popPlus (b: Long)(a: Long): Long = a + b
def popRaise (b: Long)(a: Long): Long = scala.math.pow(a, b).toLong
def popMultiply (b: Long)(a: Long): Long = a * b
```

Now, we should write inputPattern as a separate series for each case:

```
val plusPattern = """^(\+)(\d+)""".r
val multiplyPattern = """^(\*)(\d+)""".r
val raisePattern = """^(\^)(\d+)""".r
```

That enables us to write match statements like this:

```
s match {
  case plusPattern(sign, value) => popPlus(value.toLong)
  case multiplyPattern(sign, value) => popMultiply(value.toLong)
  case raisePattern(sign, value) => popRaise(value.toLong)
  case _ => {
    println("Invalid increment pattern: %s".format(s)) // side effect (we could do without, but useful perha
ps for feedback)
    popPlus(0) // an invalid pattern does not change the matrix
  }
}
```

*A valid match results in a partially-evaluated function, using the given string value as +value, *value, or ^value.*

Again, since the string has matched the regex at this point, we can just go ahead and do the `value.toLong` *conversion, knowing it won't fail.*

The other part of interpreting the growth instruction is determining the range of x and y cell values. For each pair of integers separated by colons in the x or y direction, we can write a simple function to take a string in the form `"n:m"` *and return an array of integers from n to m+1 like this:*

```
def parseInputRange (s: String): Array[Int] = {
  val ab = s.split(":").map(_.toInt)
  (ab(0) until ab(1)+1).toArray
}
```

As with all the other string to number conversions, we can do the without worry, since this function will be called only if the regex matches.

It's a huge simplification from having to do `Try {} except {}` *in every numeric conversion.*

We also need functions to create the matrix, and another to scan each cell and take a sum. Those tasks are handled by these next two functions:

```
def createMatrix(n: Int): Array[Array[Long]] =
  Array.ofDim[Long](n, n)
```

```
def getPopulation (matrix: Array[Array[Long]]): Long = {
  for {
    row   0)
  } yield cell
}.sum
```

In getPopulation we remove all the cells whose population stayed at zero, and take of sum of just what remains.

Finally, we need something to create the matrix, and cycle through all the growth instructions.

While it's simple enough to do it as an iteration, updating (mutating) the same matrix over and over again, here's a pure functional recursive solution:

```
def setPopulation (n: Int, inputs: Array[String]): Array[Array[Long]] = {
  @annotation.tailrec
  def calculate(m: Array[Array[Long]], inp: Array[String]): Array[Array[Long]] = {
    if( 0 == inp.length ) m
    else calculate(processInput(inp(0), m), inp.tail)
  }
  calculate(createMatrix(n), inputs)
}
```

The full source code is here: https://gist.github.com/dpapathanasiou/b9d85685a0381f1deea0
