Exercise 3: Distributed Game of Life (15 points)

Important notice: you are not allowed to use any *var* in this assignment.

In this assignment you will have to implement the missing parts of an actor based Game of Life. The game of life consists of a rectangular grid of cells, in which each cell can be alive or dead. The grid wraps around, which means that the left side of the grid connects to its right side, and similarly for the top and bottom sides.

We model the game of life with an actor system, composed of one `Grid` actor, which spawns a `Cell` actor for each of its cells.

The Game of Life is turn-based; a turn is called a *generation*. The state (alive of dead) of a cell for the next generation is computed based on the state of its neighbours in the current generation, as follows (taken from wikipedia):

1. Any live cell with fewer than two live neighbours dies, as if caused by under-population.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if by overcrowding.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

Are considered neighbours the 8 cells surrounding a given cell. Note that even cells on the borders of the grid have exactly 8 neighbours, since the grid wraps around.

Neighbours computation (3 pt)

Your first task is to implement the `computeAndSendNeighbours` method of the Grid actor (by filling in the blank in the following code). This method take the `population` argument. Which is a 2d array of `ActorRef`s representing each cell, along with their respective initial state. It should send to each of the `Cell` actors an initialisation message holding the initial state of the cell and a set of its neighbours.

Do not forget to declare classes and/or objects for the messages you intend to send in addition to `Init` here.

```
case object Init
case class Status(round: Boolean, isAlive: Boolean)
case class SetUp(isAlive: Boolean, neighbours: Set[ActorRef])




class Grid(width: Int, height: Int) extends Actor {

  override def receive: Receive = {
    case Init =>
      val population = Array.tabulate(height, width) { (row, col) =>
        val initiallyAlive = partOfPattern(row, col)
        val cellActor = context.actorOf(Props(new Cell(row, col)),
            name = "cell" + row + "_" + col)
        (cellActor, initiallyAlive)
      }
      computeAndSendNeighbours(population)
  }
```

```
  def computeAndSendNeighbours(population: Array[Array[(ActorRef, Boolean)]]): Unit = {
    for(row <- 0 until height) {
      for(col <- 0 until width) {
        val actorTuple = population(row)(col)
        actorTuple._2 ! SetUp(actorTuple._1,
          neighboursOf(row, col).map(coords => population(coords._1)(coords._2)._2) toSet)
      }
    }
  }

  def partOfPattern(row: Int, col: Int): Boolean = {
    val pattern = Set((0, 0), (0, 1), (1, 0), (2, 3), (3, 2), (3, 3))
    pattern((row, col))
  }

  def neighboursOf(row: Int, col: Int): List[(Int, Int)] = {
    (for (r <- row - 1 to row + 1; c <- col - 1 to col + 1 if row != r || col != r)
      yield ((r + height) % height, (c + width) % width)).toList
  }

}
```

Synchronized cells (4 pt)

Given the `Grid` actor you designed in the previous part, you will now implement the `receive` method of your cells. In this first version you will assume that there is some kind of clock that sends a `Tick` message to all actors at the start of every generation. We also assume that when this message is received, all the processing of the previous generation is done, and hence the system has stabilized (there is no more messages being processed or pending).

First your cell will need to handle the initialisation message sent from the `Grid` (holding informations about its inital state and its neighbours). Upon receiving a `Tick` message, each cell has to

1. Inform its neighbours (through messages) of its state in the current generation.
2. Update its state for the next generation, based on messages received from its neighbours, as soon as those have all been received.

Reminder: no vars allowed

```
class Cell(xPos: Int, yPos: Int) extends Actor {

  override def receive = {
    case SetUp(status, neighSet) =>
      context.become(running(status, 0, 0, neighSet))
  }

  def running(isAlive: Boolean, aliveNeighbours: Int, totalResponse: Int,
    neighbours: Set[ActorRef]): Receive = {

    case Tick =>
      neighbours.foreach(_ ! Status(isAlive))
      context.become(running(isAlive, aliveNeighbours, totalResponse, neighbours))

    case Status(otherLiving) if 7 <= totalResponse =>
      val aliveNeigh = aliveNeighbours + (if (otherLiving) 1 else 0)
      val living = (isAlive && 2 == aliveNeigh) || (3 == aliveNeigh)

      context.become(running(living, 0, 0, neighbours))

    case Status(otherLiving) =>
      val aliveNeigh = aliveNeighbours + (if (otherLiving) 1 else 0)

      context.become(running(isAlive, aliveNeigh, totalResponse + 1, neighbours))
  }
}
```

Independent cells (8 pt)

It is not possible to know when the system is stable–and thus send `Tick` messages at the appropriate time–without having a centralized actor which would gather status from all the cells.

To fix this, you will now design a truly distributed system where cells only communicate with their neighbours (as soon as they are initialized, initialisation is still taken care of by the `Grid`). You will have to keep track, somehow, of the generation, in order to keep the evolution of the game comforming to the rules.

Compared to the previous part, cells do not wait for a `Tick` message to start the next generation. Instead, as soon as they have updated their state for the next generation, they spontaneously start the next generation, and therefore send their new status to its neighbours immediately.

Pay particular attention to the *races* this would cause if you implement this naively. What happens when a cell has already processed all messages from its neighbours, but one of them is still waiting for messages coming from *its* neighbours?

Reminder: no vars allowed

```
class Cell(xPos: Int, yPos: Int) extends Actor {

  override def receive = running(false, false, 0, 0, List[Status](), Set())

  def running(round: Boolean, isAlive: Boolean, aliveNeighbours: Int,
    totalResponse: Int, pendingMessages: List[Status],
    neighbours: Set[ActorRef]): Receive = {

    case Status(‘round‘, otherLiving) if 7 <= totalResponse =>

      val aliveNeigh = aliveNeighbours + (if (otherLiving) 1 else 0)
      val living = (isAlive && 2 == aliveNeigh) || (3 == aliveNeigh)
      val nextRound = !round

      neighbours foreach ( _ ! Status(nextRound, living))
      pendingMessages.foreach(self ! _)

      context.become(running(nextRound, living, 0, 0, List[Status](), neighbours))

    case Status(‘round‘, otherLiving) =>
      val aliveNeigh = aliveNeighbours + (if (otherLiving) 1 else 0)
      context.become(running(round, isAlive, aliveNeigh, totalResponse + 1,
        pendingMessages, neighbours))

    case message: Status =>
      context.become(running(round, isAlive, aliveNeighbours, totalResponse,
        pendingMessages :+ message, neighbours))

    case SetUp(status, neighSet) =>
      neighSet foreach ( _ ! Status(true, status))
      pendingMessages.foreach(self ! _)

      context.become(running(true, status, 0, 0, List[Status](), neighSet))
  }
}
```

Exercise 4: Parallel Word Count (10 points)

In this exercise we will count the number of words in a piece of text using parallel collections. In our simplified model, words are sequences of non-whitespace characters, and you are given the `isWhitespace` function with the following signature:

```
def isWhitespace(char: Char): Boolean
```

1) To warm up, given a sequence of characters (of type `Seq[Char]`), implement a sequential word count. Keep in mind that the string may begin and end with whitespaces. Use a single `foldLeft` traversal:

```
def sequentialWordCount(segment: Seq[Char]): Int =
  segment.foldLeft((/* words encountered so far */   0,
                    /* are we currently in a word */ false)) {
    case ((count, false), char) if isWhitespace(char) => (count, false)
    case ((count, true), char)  if isWhitespace(char) => (count + 1, false)
    case ((count, _), char) => (count, true)
  } match {
    case (count, false) => count
    case (count, true)  => count + 1
  }
```

(3 points)

2) To enable parallelization, we provide a method which splits a sequence of characters into non-empty sub-sequences, producing a `ParSeq[Seq[Char]]` (you need not implement it!). The number of sub-sequences corresponds to the number or processor cores and the method guarantees they are all non-empty:

```
def segments(text: Seq[Char]): ParSeq[Seq[Char]]
```

Assuming the input sequence is `"Hello Scala!"`, show the four different cases in which it could be split in two sub-sequences and indicate the result of `sequentialWordCount` for each of them:

```
segments("Hello Scala!") =
```

Case 1: `Hel` (1 word) | `lo Scala!` (2 words)

Case 2: `Hello` (1 word) | `Scala!` (1 word)

Case 3: `Hello` (1 word) | `Scala!` (1 word)

Case 4: `Hello Sc` (2 words) | `ala!` (1 word)

In two of the cases above, the sum of `sequentialWordCount` should be different from the actual number of words. Why is this?

```
Because the segments method decides to split in the middle of a word, and we count it twice.
```

(2 points = 4 × 0.5 points)

3) Now implement the `parallelWordCount` method, using a `map` and a `foldLeft` comprehension. You can use the `segments` and `sequentialWordCount` methods defined previously and can assume none of the sub-sequences is empty:

```
def parallelWordCount(seq: Seq[Char]): Int =
  segments(seq).map({
    case segment =>
      (isWhitespace(segment.head), sequentialWordCount(segment), isWhitespace(segment.last))
  }).reduce[(Boolean, Int, Boolean)]({
    case ((startsWithChar, count1, false), (false, count2, endsInChar)) => (startsWithChar, count1
    case ((startsWithChar, count1, _),     ( _,    count2, endsInChar)) => (startsWithChar, count1
  }) match {
    case ((_, count, _))  => count
  }
```

(4 points)

4) Why did we use `map` + `foldLeft` and not `aggregate`? What would make the aggregation more difficult in this case?

Possible answer:

The zero element of `aggregate` complicates things, since it needs a new value corresponding to "uninitialized", or "I haven't seen any sequence yet". To introduce this state, the aggregation would have to occur over values of type `Option[(Boolean, Int, Boolean)]`, where None signals the additional "uninitialized" (or I haven't seen any sequences) case.

(1 points)

Exercise 3: Efficient Data Science at CFF (11 points)

In this exercise, we will concentrate on doing a number of operations on RDDs ***efficiently***. The key to obtaining full points for each problem is to find the most efficient solution. After each part we give you enough space to implement the assignment; please use that space.

Imagine CFF is interested in analyzing the customer data that it has stored and real-time behavior of its customers. To this end, in the company, they collect data about `Customer`s and `Ticket`s defined as follows:

```
case class Customer(
  customerId: Int,
  homeCity: String,
  purchaseHistory: List[Ticket]
)
case class Ticket(
  customer: Option[Int],
  origin: String,
  destination: String,
  price: Double
)
```

Assume we have an RDD containing all known customers that use the CFF **mobile app**:

```
val customers: RDD[Customer] = sc.parallelize(List(
  Customer(1, "Lausanne", List(
    Ticket(Some(1), "Lausanne", "Zurich", 70.4),
    Ticket(Some(1), "Zurich", "Lausanne", 70.4)
  )),
  Customer(2, "Montreux", List(
    Ticket(Some(2), "Montreux", "Vevey", 3.50),
    Ticket(Some(2), "Montreux", "Martigny", 5.40)
  )),
  ...
))
```

Part 1 (1 point) Using `customers`, calculate the total amount of money the CFF has earned in ticket sales from known customers.

```
val sales: Double =
  customers.aggregate(0.0)(
    (agg, c) => c.purchaseHistory.map(_.price).sum, _ + _)
```

Part 2 (1 point) Using `customers`, and knowledge of their `homeCity`, determine how much money each Swiss city grosses in CFF ticket sales of known customers.

```
val cityGross: Seq[(String, Double)] =
  customers.map(c => (c.homeCity, c.purchaseHistory.map(_.price).sum))
      .reduceByKey((x, y) => x + y).collect
```

Part 3 (3 points) Suppose the CFF has a service that collects all ticket purchases done in previous 10 minutes into an RDD. The tickets that are bough on a ticket machine have the `customerId` field set to `None` and tickets bought by known customers have a `customerId` field set to `Some(id)`:

```
// RDD containing all purchases made in the last 10 minutes
// RDD containing all purchases made in the last 10 minutes
val recentPurchases: RDD[Ticket] = sc.parallelize(List(
  Ticket(None, "Bern", "Zurich", 15.20), // paper ticket: no customer info
  Ticket(Some(1), "Lausanne", "Zurich", 70.4),
  Ticket(Some(1), "Zurich", "Lausanne", 70.4),
  Ticket(Some(2), "Montreux", "Vevey", 3.50),
  Ticket(Some(2), "Montreux", "Martigny", 5.40),
  ...
))
```

With this data CFF wants to calculate the ratio of people leaving a city that is their home city versus the overall number of people leaving that city. Assume that customers and recent purchases are already partitioned by the option of their id.

```
def leavingTheCityRatio(
  customers: RDD[(Option[Int], Customer)],
  recentPurchases: RDD[(Option[Int], Ticket)],
  city: String): Double = {
    val joined = (recentPurchases leftOuterJoin customers).persist()
    val leavingHome =
      joined.filter(_._2._2.isDefined)
        .filter(c => c._2._2.get.homeCity == city)
        .count()
    val leaving = joined.filter(c => c._2._1.origin == city).count()
    leavingHome/leaving
  }
```

Part 4 (3 points) CFF data scientists analyzed their queries and cluster size and realized that they want to persist customer data by the option of their `id` (`Option[Int]`) in 10 different in-memory partitions. How can we achieve this:

```
val idCustomers: RDD[(Option[Int], Customer)] =
  customers.map(c => (Option(c.customerId), c))


val partitioner: Partitioner =
  new RangePartitioner(10, idCustomers)


val persistedCustomers: RDD[(Option[Int], Customer)] =
   idCustomers.partitionBy(partitioner).persist()
```

Given the `persistedCustomers`, CFF wants to periodically compute `leavingTheCityRatio` for Lausanne. Their timer process is set up to call the `leavingLausanneRatio` function every 10 minutes with fresh `recentPurchases`. Can you implement the `leavingLausanneRatio` function so that a minimal amount of network traffic happens. Assume that you have access to—and you are allowed to reuse—variables `persistedCustomers`, `partitioner`, and `leavingTheCityRatio`:

```
// assume that persistedCustomers, partitioner, and leavingTheCityRatio are visible
def leavingLausanneRatio(recentPurchases: RDD[Ticket]): Double = {
val recentPart = recentPurchases.map(t => (t.customer, t))
    .partitionBy(persistedCustomers.partitioner.get)
  leavingTheCityRatio(persistedCustomers, recentPart, "Lausanne")
}
```

If we call `mapValues` on `persistedCustomers` would the result of `mapValues` still have a partitioner? If so, which kind?

```
Yes, the RangePartitioner.
```

Bonus (2 points) After running the `leavingLausanneRatio` service the CFF data analysts realized that their query is taking far more time then expected. What could be the reason? We would accept two answers here:

1. The partition that has `None` could be much bigger than the others and that could cause longer than expected computations.
2. The `Option[Int]` could be replaced with a convention that negative numbers are equal to `None`.

Question 1 (1 points) Can you name 5 Spark methods that may cause a shuffle?

```
groupBy, reduce, leftOuterJoin, rightOuterJoin, fold
```

Question 2 (2 points) For which class of problems is it particularly advantageous to repartition the dataset? (i.e., for which pattern of computation would it save you time if you first created a partitioner and used `partitionBy`?) Why?

Here are some of the possible answers:

1. When we have skewed data and we would like to rebalance it equally.
2. When we want to avoid the shuffle of a bigger data set when joining with a smaller. In this case we assign the partitioner of the larger data set to the smaller one (as in Part 4).