204: Swift Functional Programming, Part 3: Lab Instructions

In this lab section, you will insert the functions that were developed in the playground during the demo section into an app.

Then you will refactor the code to make more extensive use of higher order functions.

1. Building the app

First, from the playground or this document, copy and paste the function `neighbors` into `Conway.swift`, and add the private access specifier:

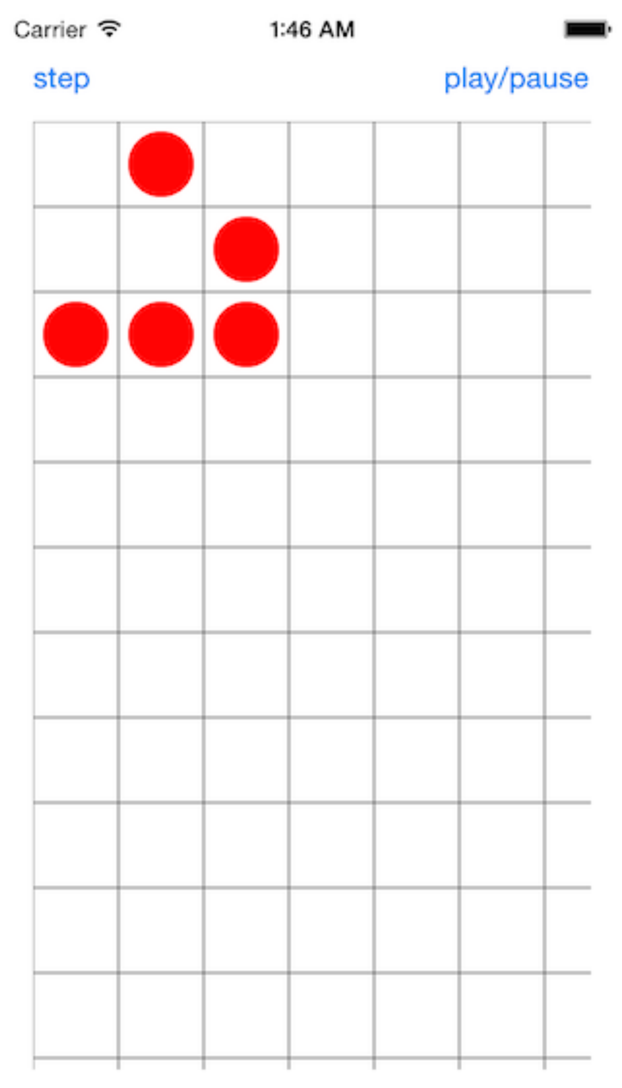
```
private func neighbors(OfCell cell:Cell) -> [Cell]
{
  let deltas = [(-1,-1),(0,-1),(1,-1),
    (-1, 0),       (1,0 ),
    (-1, 1),(0, 1),(1, 1)]

  let neighbors = map(deltas,{
    (delta:(Int,Int)) -> Cell in
    return Cell(x: cell.x + delta.0, y: cell.y + delta.1)
  })
  return neighbors
}
```

Second, from the playground or this document, copy and paste the function `activeCellsOneStepAfter` into `Conway.swift`:

```
func activeCellsOneStepAfter(activeCells:[Cell]) -> [Cell]
{
  // STAGE 1. loop to build array of every "neighboring", an instance of a living cell being neighbored
  // i.e., "map" every active cell to an array of its neighbors, then concatante the arrays.
  var neighborings = [Cell]()
  for cell in activeCells {
    for neighborOfCell in neighbors(OfCell: cell) {
      neighborings.append(neighborOfCell)
    }
  }

  // STAGE 2. loop and count duplicate neighborings
  // i.e. "reduce" an array of Cells to an array of (Cell,Int) tuples counting duplicates
  var neighboringsPerCell = Dictionary<Cell,Int>()
  for neighboringCell in neighborings {
    if let value = neighboringsPerCell[neighboringCell] {
      neighboringsPerCell[neighboringCell] = value + 1
    }
    else {
      neighboringsPerCell[neighboringCell] = 1
    }
  }

  // STAGE 3. loop to filter only neighborings with certain counts
  // filter an array of (Cell,Int) tuples based on the Int value and other conditions
  let neighboringsPerCellActiveNextStep = filter(neighboringsPerCell,
    { (theNeighbor:Cell,neighborCount:Int) -> Bool in
      return (neighborCount == 3) || (neighborCount == 2 && find(activeCells,theNeighbor) != nil)
  })

  // STAGE 4. loop to gather only the neighborings themselves
  // map an array of (Cell,Int) tuples to an array of Cells
  var neighboringsActiveNextStep = [Cell]()
  for (theNeighbor,neighborCount) in neighboringsPerCellActiveNextStep {
    if       (neighborCount == 3) ||
      (neighborCount == 2 && find(activeCells,theNeighbor) != nil)
    {
      neighboringsActiveNextStep.append(theNeighbor)
    }
  }

  // result: the cells alive at the next step in time

  return neighboringsActiveNextStep
}
```

At this point, you have inserted the essential logic for time-evolution into the app.

Now do *Build & Run*, choosing an iPhone 5s simulator, and you should see the following screen:



At any point in time, the state of this board is represented by the property `activeCoords` in *GridView.swift*.

Press the *step* button to progress the game by one time step. Press the *play/pause* button to start/pause time. Play. Isn't it a wonderful life?

Now we will refactor to use more higher order functions.

2. Use `map` to gather the living cells

Stage 4 still uses a raw loop. Replace it with the following call to `map`:

```
let neighboringsActiveNextStep = map(neighboringsPerCellActiveNextStep, { (theNeighbor:Cell,_) -> Cell in r
eturn theNeighbor} )
```

3. Define and use mapcat

Stage 1 contains a raw loop that does something almost, but not exactly like, mapping. It generates an array of neighbors for every cell, and then concatenates all those arrays. This common operation is called `mapcat`.

In *Functional.swift*, paste in this definition of `mapcat`, which uses `map`:

```
func mapcat<S,T>(items:[S],transform:S->[T]) -> [T] {
  let groups:[[T]] = map(items,transform)
  var result:[T] = [T]()
  for group in groups {
    result = result + group
  }
  return result
}
```

In *Conway.swift*, replace STAGE 1 with the following:

```
let neighborings = mapcat(activeCells, neighbors)
```

Note that we do not need to pass mapcat, or map, a closure. We can just directly pass it the already defined function `neighbors`.

4. Define and use frequencies

Stage 3 in the calculation measures the frequencies of values in the array. This is another common pattern that we'd like to encapsulate in a single pure function, which takes an input and returns an output.

Switch to the file *Functional.swift*, and paste in the following function definition:

```
func frequencies<T : Hashable>(coll:[T]) -> Dictionary<T,Int>
{
  var d = [T:Int]()
  for obj in coll {
    if let count = d[obj] {
      d[obj] = count + 1
    }
    else {
      d[obj] = 1
    }
  }
  return d
}
```

Now return to *Conway.swift* and update stage 2 to the following:

```
var neighboringsPerCell = frequencies(neighborings)
```

Introducing reduce

We've introduced `map` and `filter`. The third of the classic higher-order functions is `reduce`.

Beware: this one is not at first intuitive. But take heart: it makes sense with a little practice.

Essentially, `reduce` captures the pattern of consuming a collection of items, one at a time, by combining them into some accumulated value. It takes three arguments: the collection to be consumed, the initial value of the accumulated value, and the combining function.

For instance, if the items are integers, and you are combining them by adding them to the accumulated value, and that accumulated value begins as zero, then you are just adding all the numbers.

In other words,

```
reduce([1,2,3], 0, +) = 0 + 1 + 2 + 3 = 6
```

But you could also use it to combine strings, where the combining function is string concatenation

```
reduce(["A","B","C"],"x",+) = "x" + "A" + "B" + "C" = "xABC"
```

Now let us use `reduce` to simplify our utility functions.

Erase mapcat and paste in this new definition:

```
func mapcat<S : SequenceType, T>(source:S, transform: (S.Generator.Element) -> [T]) -> [T]
{
  let groups:[[T]] = map(source, transform)
  return reduce(groups, [T](), +)
}
```

(The generic type signatures are to allow mapcat to be used not only with arrays, but with other squence types.)

One thing to note here is that the `reduce`-based implementation of mapcat uses no iteration. Indeed, all three of classic higher-order functions are ways to avoid iteration.

Second, while the combining function here is array concatenation, it is not necessary for reduce to return an array in general. With a different combining function, you could have a different kind of accumulated value, like an Int or a Dictionary.

In fact, `reduce` is very flexible. You could implement `map` and `filter` in terms of reduce, but not vice versa. Although reduce is described in textbooks from decades ago, the development of more abstract and general forms of reduction is an area of active ferment even now, for example, with the introduction of "transducers" to Clojure.

Build and Run.