Programming Principles

Final Exam

Wednesday, December 18th 2013

First Name: _____

Last Name: _____

Your points are *precious*, don't let them go to waste!

Your Name Work that can't be attributed to you is lost: write your name on each sheet of the exam.

Your Time All points are not equal. Note that we do not think that all exercises have the same difficulty, even if they have the same number of points.

Your Attention The exam problems are precisely and carefully formulated, some details can be subtle. Pay attention, because if you do not understand a problem, you can not obtain full points.

Some help You will find, on a separate sheet, an appendix containing some useful functions. You can use the functions present on this sheet at any time during your exam.

Exercise	**Points**	**Points Achieved**
1	10	
2	10	
3	10	
4	10	
Total	40	

Exercise 1: Multiset (10 points)

A *multiset* is a set where elements can appear multiple times. We will represent a multiset of `Int` elements as a function from `Int` to `Int`: the function returns `0` for any `Int` argument that is not in the multiset, and the (positive) number of times it appears otherwise.

```
type Multiset = Int => Int
```

Implementing a functional multiset (5 points)

In this part, you are required to implement some basic functions on a multiset. For this, you may use mathematical operations such as `+`, `-`, `min`, `max`.

First, define functions for constructing the empty set, and singleton sets. A singleton set contains one copy of the integer passed to it:

```
def emptyMultiSet: MultiSet = ???
def singleton(x: Int): MultiSet = ???
```

Next, define the basic set operations `union`, `intersection`, and `diff`.

The `union` of two multisets `a` and `b` contains all elements of `a` plus all elements of `b`.

```
def union(a: MultiSet, b: MultiSet): MultiSet = ???
```

The intersection of two multisets `a` and `b` contains all elements that are in both `a` and `b`. For example, the intersection of `MultiSet(2 => 3, 3 =>1)` and `MultiSet(2 => 1, 1 => 1)` is `MultiSet(2 => 1)`.

```
def intersect(a: MultiSet, b: MultiSet): MultiSet = ???
```

The `diff` of two multisets `a` and `b` contains all elements of `a`, except those in `b`. For example, `diff(MultiSet(2 => 1, 3 => 4, 1 => 2), MultiSet(2 => 3, 3 => 1))` will be `MultiSet(3 => 3, 1 => 2)`.

```
def diff(a: MultiSet, b: MultiSet): MultiSet = ???
```

Using a functional multiset (5 points)

Finally, implement a function that takes a number and returns its prime factors as a multiset. The prime factorisation of a number `n` is its decomposition as a product of prime numbers. For example, the prime factorisation of 120 is $2^3 \cdot 3^1 \cdot 5^1$, and the prime factorisation of 313 is 313^1, as it is a prime number. The number 1 does not have a prime factorisation.

```
def primeFactors(n: Int): MultiSet = ???
```

Hint: there are many ways to solve this problem. If needed, you can assume that prime numbers are given as a `Stream`. As a reminder, you can access the head of a stream with `mystream.head`, and the tail of a stream with `mystream.tail`. You may also assume, if needed, access to an `isPrime` function, which tells you whether an integer is prime or not:

```
val primes: Stream[Int] = //2,3,5, ... Given, DO NOT IMPLEMENT
def isPrime(n: Int): Boolean = //Given, DO NOT IMPLEMENT
```

Whatever your approach may be, remember that recursion is your friend.

Exercise 2: Monads (10 points)

A monad `M` is a parametric type `M[T]` with two operations, `flatMap` and `unit`, that have to satisfy some laws. We know that `List` is a monad, and in this exercise you will prove this.

Lists are constructed from the empty list `Nil` and from the `::` constructor and we assume the following implementation. `++` is the concatenation operator.

```
1:    (x :: xs) flatMap f == f(x) ++ xs flatMap f
2:    Nil flatMap f == Nil

3:    Nil ++ ys == ys
4:    (x :: xs) ++ ys == x :: (xs ++ ys)

5:    unit[T](x: T) == x :: Nil
```

In addition to the above, you can assume the following:

```
6: xs ++ Nil == xs
7: (xs ++ ys) flatMap f == xs flatMap f ++ ys flatMap f
```

In your proof

- Show all steps.
- Justify each line in your proof with the line numbers for the above rules.
- Think carefully how to prove each law. You may have to use a straight-forward derivation on one or both sides of the equivalence, and/or a proof by induction.
- Be careful with parentheses: `x :: xs flatMap f` is not the same as `(x::xs) flatMap f`

Left unit (3 points)

Show that `unit(x) flatMap f == f(x)` holds for lists.

Right unit (3 points)

Show that `m flatMap unit == m` holds for lists.

Associativity (4 points)

Show that `(m flatMap f) flatMap g == m flatMap (x => f(x) flatMap g)` holds for lists.

Exercise 3: Comprehending Observables (10 points)

Chocolats & Cie., a top chocolate manufacturing company in Switzerland, is planning on a special Christmas package release. Each package contains a few samples of different kinds of chocolate. Normally, the channels which produce different chocolate kinds are separate, but due to some poor management and end-of-year stress, there is only one conveyer belt with all the chocolates. They need your help in sorting things out.

A chocolate is represented as a case class:

```
case class Chocolate(kind: String)
```

There are four (there can be many more) kinds of chocolate, given as a list:

```
val chocolateKinds = List("Truffe", "Praline", "Meringue", "Carmelita")
```

There is also a mapping from chocolate kinds to the number of each that should be in a packet :

```
val chocolateNumbers: String => Int = {x => x match {
  case "Truffe" => 1
  case "Praline" => 2
  case "Meringue" => 1
  case "Carmelita" => 3
}}
```

Generating chocolates (2 points)

We want to simulate the mixed up nature of the conveyer belt as a random generator of chocolates. Recall, for this purpose, the generators seen in the class. Here is a useful generator for generating random positive integers based on the `integers` generator:

```
def positives: Generator[Int] = integers.map{x => if(x < 0) -x else x}
```

Your first task is to define a generator that takes a list of elements and generates one of them randomly. Your solution should reuse previously seen generators.

```
def oneOf[T](ls: List[T]): Generator[T] = ???
```

You can now define a generator for chocolates as well:

```
val chocolateGen: Generator[Chocolate] = ???
```

Separating chocolate kinds (3 points)

Now that we have simulated the chocolate factory, we can assume that there is a chocolate channel, which is an `Observable[Chocolate]`, that emits chocolates provided by the generator.

```
val chocolateChannel: Observable[Chocolate] = // Given, DO NOT IMPLEMENT
```

Your task is to separate the chocolates by kind. From `chocolateChannel`, create an observable that achieves this separation:

```
val chocolatesByKind: Observable[(String, Observable[Chocolate])] = ???
```

Bunching chocolates together (3 points)

As mentioned above, a packet contains several bunches of chocolates, of different kinds. One single bunch contains chocolates of a single kind. For this, we define two type aliases:

```
type Bunch = Seq[Chocolate]
type Packet = Seq[Bunch]
```

Create an observable that bunches the chocolates by kind, with respect to the mapping defined above:

```
val chocolatesBunched: Observable[Observable[Bunch]] = ???
```

Making packets (2 points)

Finally, once the chocolates are bunched, we can create packets for them, and ship them away for Christmas! Every packet contains 1 Truffe, 2 Pralines, 1 Meringue and 1 Carmelita:

```
val chocolatePackets: Observable[Packet] = ???
```

Example output:

```
val presents = chocolatePackets subscribe {c =>
  println(c)
}

for(i <- 1 to 50){
  val c = chocolateGen.generate
  chocolateChannel onNext c
  Thread.sleep(500) //simulates time to produce a chocolate
}

Possible output:
> Seq(
  Seq(Chocolate(Carmelita), Chocolate(Carmelita), Chocolate(Carmelita)),
  Seq(Chocolate(Praline), Chocolate(Praline)),
  Seq(Chocolate(Truffe)), Seq(Chocolate(Meringue))
  )

> Seq(
  Seq(Chocolate(Carmelita), Chocolate(Carmelita), Chocolate(Carmelita)),
  Seq(Chocolate(Praline), Chocolate(Praline)),
  Seq(Chocolate(Truffe)), Seq(Chocolate(Meringue))
  )
```

Note:

The API is your friend.

Exercise 4: Batch Logging using Actors (10 points)

Publish-subscribe is a messaging pattern in which *publishers* broadcast a message to multiple receivers. In order to receive these messages, clients must first *subscribe* with the publisher. Conversely, when they no longer wish to receive the messages, they can unsubscribe.

In this exercise, you are asked to implement a logging system based on the publish-subscribe pattern with the use of Actors. The system consists of a single `Publisher` node, and an arbitrary number of `Logger` nodes. `Logger` nodes subscribe to the `Publisher` by sending it a `Subscribe` message. Similarly, they can unsubscribe by sending an `Unsubscribe` message.

The `Publisher`'s task is, apart from the handling of subscriptions, to wait for `Update` messages and to distribute them to all the subscribed `Logger`s via a `LogEntry` message.

The `Logger` node is given the reference of the `Publisher` and an integer field called `debugLevel`. The `Logger` should subscribe to the publisher at creation time and then wait for `LogEntry` messages. Each `LogEntry` message consists of a `String` representing the content, and an integer field `level`, which denotes the importance of the message. The `Logger` should then store all messages (as a list of `Strings`) whose level is greater or equal to its `debugLevel`. Furthermore, after every 42 *logged* (not received) messages, the entire collected log should be sent on for further processing to a pre-defined given `ActorRef` via a `LogFull` message.

When the `Logger` receives a `StopLogging` message, it should unsubscribe from the `Publisher`. `LogEntry` messages still in the message queue should then still be processed, i.e. you do not need to treat them specially. Note that the `Update` and `StopLogging` messages come from some external actor and your implementation thus only needs to receive and process but not send them.

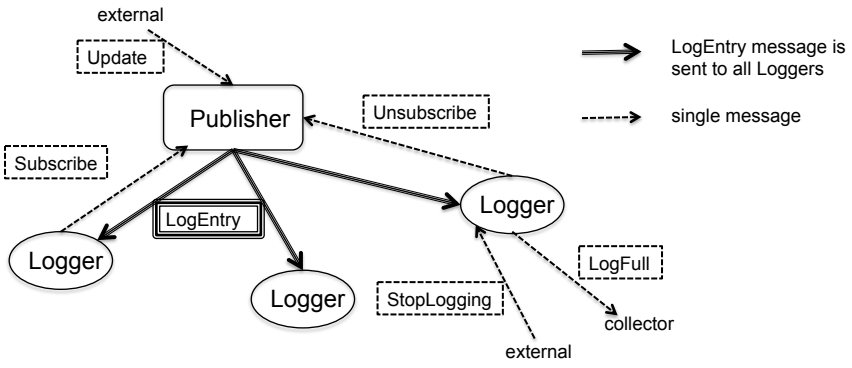


Figure 1: A visualization of the publish-subscribe pattern

Publisher behavior (5 points)

Implement the `Publisher` node:

```
class Publisher extends Actor{
  def receive = ???
}
```

such that it receives the following messages

```
case object Subscribe
```

```
case object Unsubscribe
case class Update(msg: String, level: Int)
```

and sends

```
case class LogEntry(msg: String, level: Int)
```

as described above. You may introduce both appropriate message handling as well as any local data structures you find necessary.

Logger behavior (5 points)

Implement the `Logger` node:

```
class Logger(collector: ActorRef, pub: ActorRef, debugLevel: Int) extends Actor {
  def receive = ???
}
```

such that it correctly subscribes to its `Publisher` and handles `LogEntry` messages, as well as receiving and sending the `LogFull` and `StopLogging` messages:

```
case class LogFull(log: List[String])   // should be sent to collector
case object StopLogging
```

You may introduce both appropriate message handling as well as any local data structures you find necessary.

Appendix

Here are some methods from the Scala standard library that you may find useful:

on `List` (containing elements of type `T`):

- `def ++[(that: List[T]): List[T]`: returns a new list containing the elements from the left hand operand followed by the elements from the right hand operand.
- `def filter(p: A => Boolean): List[A]`: Selects all elements of this list which satisfy predicate `p`.
- `def flatMap[B](f: A => List[B]): List[B]`: builds a new list by applying function `f` to all elements of this list and concatenating the elements of the resulting lists.
- `map[B](f: (A) => B): List[B]`: builds a new list by applying a function to all elements of this list.

on the companion object `Generator`:

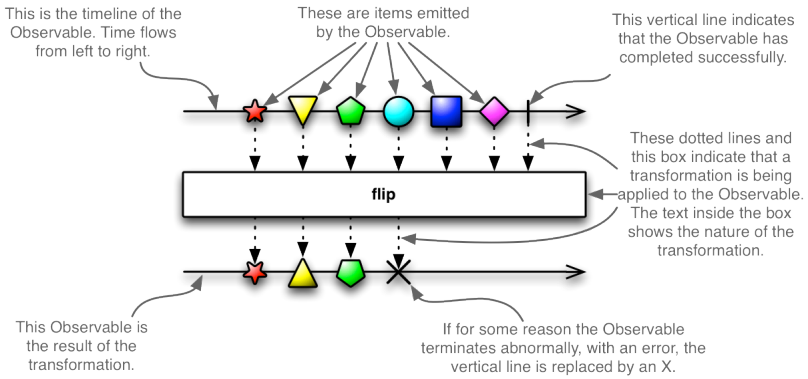
- `def choose(minvalue: Int, maxvalue: Int)`: a generator that generates a value between `minvalue` and `maxvalue`, boundaries being inclusive.
- `def integers: Generator[Int]`: a generator for generating random integers.
- `def single[T](x: T): Generator[T]`: a singleton generator, that generates the value `x`.
- `map` and `flatMap` are also applicable on Generators.

on `Stream` (containing elements of type `T`):

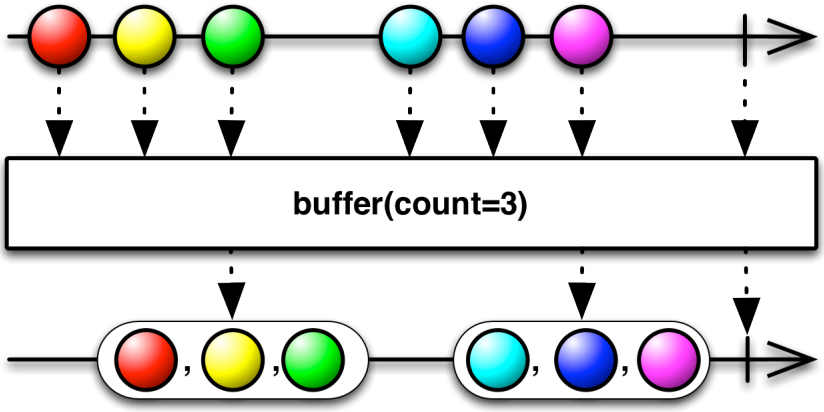
- `def head: T`: selects the first element of the stream.
- `def tail: Stream[T]`: selects all elements except the first.
- `map`,`flatMap` and `filter` are also applicable on Streams.

on `Observable` (containing elements of type `T`):

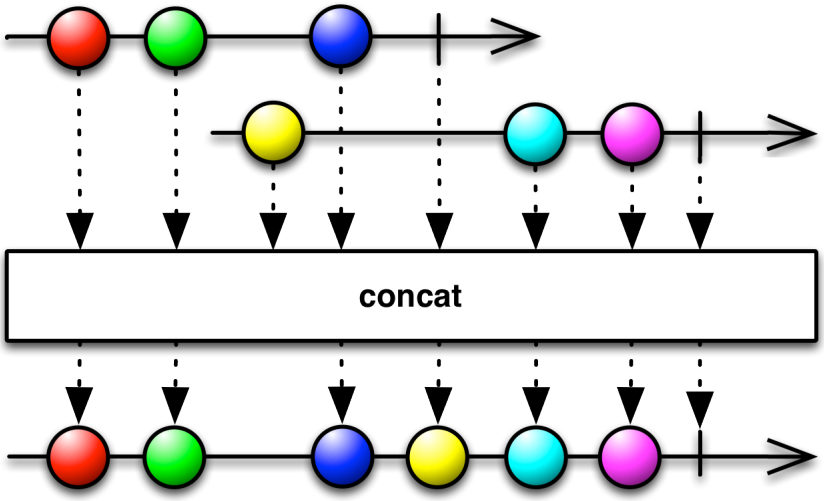
- we will use the same marble diagrams as you have seen during the course:



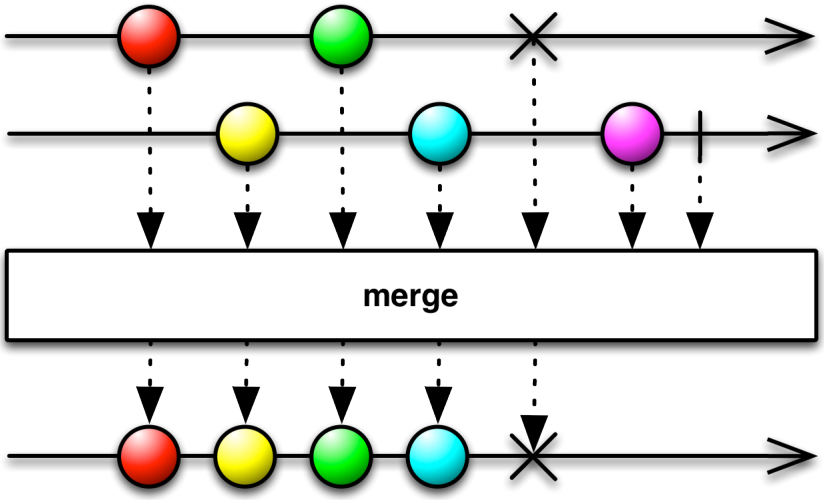
- `def buffer(count: Int): Observable[Seq[T]]`: creates an `Observable` which produces buffers of collected values. This Observable produces connected non-overlapping buffers, each containing `count` elements. When the source `Observable` completes or encounters an error, the current buffer is emitted, and the event is propagated.



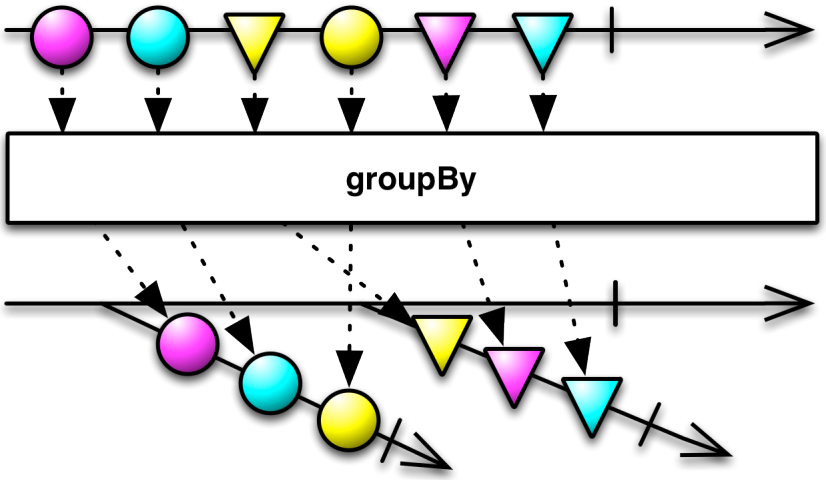
- `def concat[U]: Observable[U]`: Returns an Observable that emits the items emitted by several `Observables`, one after the other. This operation is only available if this is of type `Observable[Observable[U]]` for some `U`.



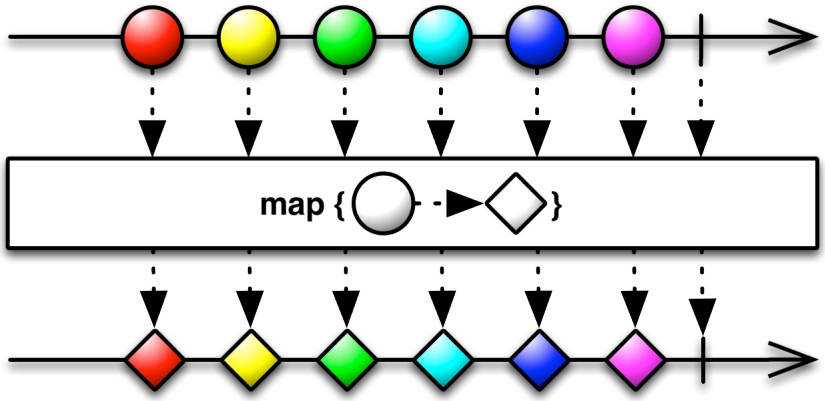
- `def flatten[U]: Observable[U]`: Flattens the sequence of Observables emitted by this into one Observable, without any transformation. Same as `merge`.



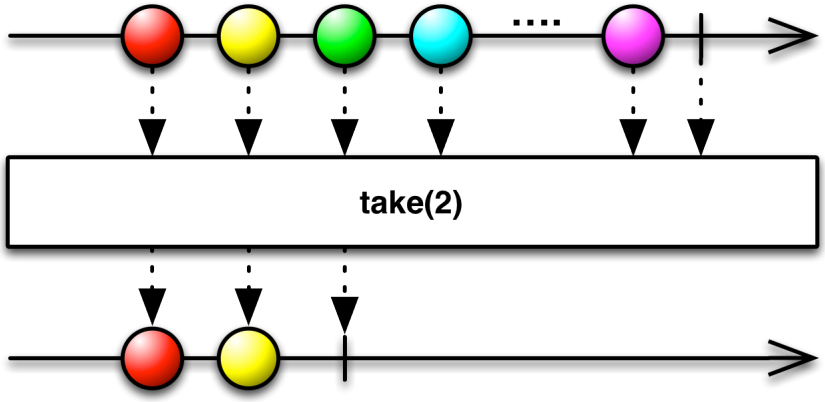
- `def groupBy[K](f: T => K): Observable[(K, Observable[T])]`: groups the items emitted by this `Observable` according to a specified discriminator function.



- `def map[R](f: T => R): Observable[R]`: returns an `Observable` that applies the given function to each item emitted by an `Observable` and emits the result.

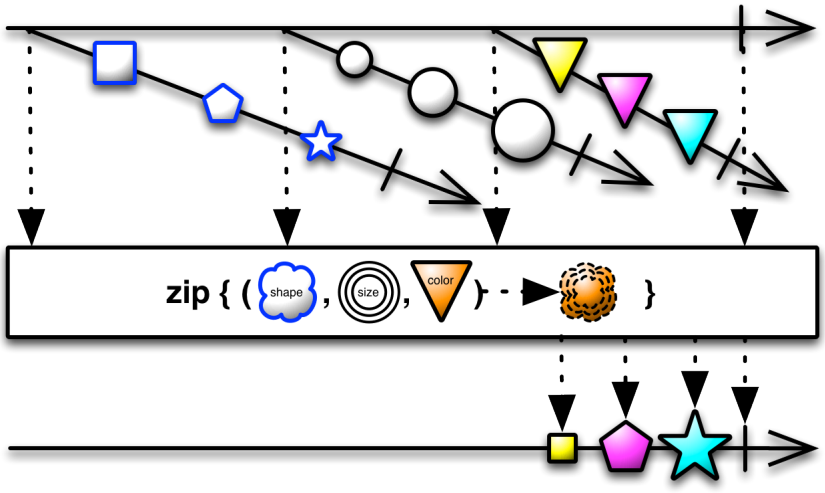


- `def take(n: Int): Observable[T]`: returns an `Observable` that emits only the first `n` items emitted by the source `Observable`.



on the companion object `Observable`:

- `def zip[T](observables: Observable[Observable[T]]): Observable[Seq[T]]`: given an `Observable` emitting `N` source observables, returns an observable that emits `Seq`s of `N` elements each. The first emitted `Seq` will contain the first element of each source observable, the second `Seq` the second element of each source observable, and so on.


zip { (
shape
,
size
,
color
) - >
}