Lecture 5

1 Required Reading

Read Chapters 16 and 17 of Programming in Scala.

2 Partial Functions and Signalling Errors

Many functions are not defined on all inputs. For example, if you’re reading input from a keyboard (i.e., a string) and want to parse the string as a number, you can apply Integer.parseInt:

```scala
t: Int = 10
```

But, if the string is not a numeral, you get an exception:

```scala
java.lang.NumberFormatException: For input string: "ten"
```

You’ve encountered other ways of signalling errors. For example, if you lookup an unbound key in a hashtable, Java (and Scala) produce nulls:

```scala
null
```

Finally, here is a more insidious example. The following function calculates the point where two lines, defined by \( y = m_1 x + b_1 \) and \( y = m_2 x + b_2 \), intersect. The function is not defined when the two lines are parallel (i.e., when their slopes are the same, or \( m_1 = m_2 \)):

```scala
case class Point(x: Double, y: Double)
def inter(m1: Double, b1: Double, m2: Double, b2: Double): Point = {
  val x = (b2 - b1) / (m1 - m2)
  Point(x, m1 * x + b1)
}
```

When the slopes are the same, the denominator, \( m_1 - m_2 \) is 0. So, you might expect a divide-by-zero ArithmeticException. That’s not: what happens:

```scala
1.0 / 0.0
res0: Double = Infinity
```

So you can’t even catch this error with an exception handler, since no exception is raised:
All these mechanisms for signalling errors share similar flaws:

1. **Exceptions**: you have to remember to catch them, or your program will crash. You can’t tell if a function will throw an exception without carefully reading its code, which may not even be possible if it is in a library.

2. **Producing null**: even worse than exceptions, because your program will will not crash on an error. When it does crash, you’ll spend a lot of time trying to figure out what produced the `null`.

3. **Producing other null-like values**: see above.

The real problem is that the types of all these functions are not useful:

- The type of `Integer.parseInt` is `String => Int`, but it may throw an exception instead of producing an `Int`.
- The type of `ht.get` is `Any => String`, but it may produce a `null`.
- The type of `inter` is `(Double, Double, Double, Double) => Point`, but it can produce `Point(NaN, NaN)`, which is clearly not what we had in mind.

**A Solution**  
Let’s use `inter` as an example and modify the function so that its type makes it obvious that it may not always return a `Point`. We introduce the following sealed trait:

```scala
sealed trait OptionalPoint
case class SomePoint(pt: Point) extends OptionalPoint
case class NoPoint() extends OptionalPoint
```

And we modify `inter` to produce `NoPoint` instead of a malformed-`Point`:

```scala
def inter(m1: Double, b1: Double, m2: Double, b2: Double): OptionalPoint = {
  if (m1 - m2 == 0) {
    NoPoint()
  } else {
    val x = (m1 - m2) / (b2 - b1)
    SomePoint(Point(x, m1 * x + b1))
  }
}
```

With this new type, any program that applies `inter` will be forced to check if a point was produced:

```scala
inter(10, 3, 10, 3) match {
  case NoPoint => println("no intersection")
  case SomePoint(Point(x,y)) => println(s"intersection at ($x, $y")
}
```

Consider another example: a type that represents an **alarm clock**. An alarm clock needs to track the current time and *the alarm time if the alarm is set*:

```scala
case class Time(h: Int, m: Int, s: Int)
case class AlarmClock(time: Time, alarm: Time, alarmOn: Boolean)
```

But, with this representation, it is easy to make simple errors. For example, you may accidentally trigger the alarm when `time == alarm`, if you forget to check `alarmOn` first. A cleaner representation is to use the same pattern we used above:
sealed trait OptionalAlarm
case class NoAlarm() extends OptionalAlarm
case class AlarmSet(time: Time) extends OptionalAlarm
case class AlarmClock(time: Time, alarm: OptionalAlarm)

In both inter and AlarmClock, it is completely obvious from the type that a point is
not always produced and that an alarm is not always set. Partial functions are pervasive in
computing and this pattern will make your programs more robust.

But, it is annoying to define a new type such as OptionalPoint and OptionalAlarm for
each type.

2.1 The Option Type

Scala has a built-in generic type called Option that abstracts the pattern we discussed above.
For example, here is inter rewritten to use Option:

```scala
def inter(m1: Double, b1: Double, m2: Double, b2: Double): Option[Point] = {
  if (m1 - m2 == 0) {
    None
  } else {
    val x = (m1 - m2) / (b2 - b1)
    Some(Point(x, m1 * x + b1))
  }
}
```

3 Built-in List methods

Scala’s List type has dozens of useful methods. For example, these methods correspond to
the higher-order functions we’ve seen so far:

- We can map over a list:
  ```scala```
  assert(List(10, 20, 30).map(x => x + 1) == List(11, 21, 31))
  ```scala```

- We can filter a list:
  ```scala```
  assert(List(1, 2, 3, 4).filter(x => x % 2 == 0) == List(2, 4))
  ```scala```

- We can fold over a list:
  ```scala```
  assert(List(10, 20, 30).foldRight(0)((x, acc) => x + acc) == 60)
  ```scala```

Notice that the syntax of foldRight is unusual. We’ll get into the details how it works
later. For now, there is no need to write sums and products using folds. Scala’s lists have
methods that do these directly:

```scala```
assert(List(10, 20, 30).sum == 60)
assert(List(10, 20, 30).product == 6000)
```scala```

There are several other useful methods. For example:

- lst.take(n) produces a list with first n elements of lst. Similarly, lst.drop(n) produces
  a list without the first n elements of lst.
In the last line, the :::: operator (pronounced “append”) appends two lists.

```
val lst = List("X", "Y", "Z", "A", "B", "C")
assert(lst.take(3) == List("X", "Y", "Z"))
assert(lst.drop(3) == List("A", "B", "C"))
assert(lst.take(3) :::: lst.drop(3) == lst)
```

There are several other useful functions. You should explore the List API to learn more about them.

4 Sets and Maps

The Scala standard library also defines sets and maps, which are two other data structures that are very common. Note that these represent mathematical sets, so they don’t have duplicate elements and are not ordered. i.e., the three definitions below are the same:

```
Set(1, 2, 3)
Set(1, 2, 2, 3)
Set(3, 2, 1)
```

Sets have methods for calculating set union, set intersection, set difference, testing emptiness, and so on:

```
assert(Set(1, 2, 3).union(Set(2, 3, 4)) == Set(1, 2, 3, 4))
assert(Set(1, 2, 3).intersect(Set(2, 3, 4)) == Set(2, 3))
assert(Set(1, 2).intersect(Set(3, 4)).isEmpty == true)
```

We can also map and filter the elements of sets, similar to lists:

```
assert(Set(1, 2, 3, 4).filter(x => x % 2 == 0) == Set(2, 4))
assert(Set(1, 2, 3, 4, 5).map(x => x % 2) == Set(0, 1))
```

There are several other useful methods in the Set API.
Maps  The Map data-structure (which is not the same as the map function) is a finite map from keys to values. For example:

```scala
val dueDates = Map(
  1 -> "Jan␣28",
  2 -> "Feb␣4",
  3 -> "Feb␣11")
```

**Notation**  X -> Y is just another notation for the tuple (X, Y), but is commonly used with maps to suggest that it “maps X to Y”. Therefore, we could have written

```scala
val dueDates = Map(
  (1, "Jan␣28"),
  (2, "Feb␣4"),
  (3, "Feb␣11"))
```

We can lookup a key in two ways. First, we can simply apply the map to a key, just like a function:

```scala
assert(dueDates(1) == "Feb␣4")
```

However, if the key is not found, this throws an exception. An alternative is to apply the get methods, which produces Some(v) if the key is mapped to a value and None otherwise:

```scala
dueDates.get(x) match {
  case Some(v) => v
  case None => "Unknown␣assignment"
}
```

We can augment maps using the following syntax:

```scala
val moreDueDates = dueDates + (4 -> "Feb␣18")
```

We emphasize that this produces a new map with all the contents of dueDates and the key mapping for 4. The original map is not modified.

**Notation**  The notation m + (k -> v) seems to be something new, but we know how to unpack it. We just learned the arrow notation is another way of writing tuples, we can first rewrite it as m + (k, v). We also learned that all operators on objects are actually methods, so can rewrite it again as m.+((k, v)).

Therefore, we can conclude that maps have a method called + that takes one argument. This argument is a tuple where the first component is a key k and the second component is the value v. The result of applying this method is a new map where k is mapped to v.

It is often get the set of keys in a map:

```scala
assert(dueDates.keys == Set(1, 2, 3))
```

Similarly, we can get an iterator over the values. Typically, you’ll want to turn the iterator into a list immediately:

```scala
assert(dueDates.values.toList == List("Jan␣28", "Feb␣4", "Feb␣11"))
```

As usual, there are dozens of handy methods over maps. You should explore the Map API.
5 Conversions

It is sometimes necessary and easy to convert lists, sets, and maps to each other. These objects have methods called toList, toSet, and toMap that do exactly what their names suggest. For example:

```scala
assert(Map("X" -> 10, "Y" -> 20).toList == List(("X", 10), ("Y", 20)))
assert(List(("X", 10), ("Y", 20)).toMap == Map("X" -> 10, "Y" -> 20))
```