Unit 5b: Currying

1 Introduction

Previously you saw an example of partial application of the following function that takes three parameters:

```
def sum(a:Int, b:Int, c:Int): Int  = {a+b+c}
```

This function could alternatively have been defined as:

```
def sum_curry(a:Int)(b:Int)(c:Int): Int = {a+b+c}
```

This is called a curried function. Currying is the mapping of a function with a list of multiple parameters to a function with multiple lists of one parameter each. Named after the mathematician *Haskell Curry* (as was the functional programming language Haskell).

What's the difference? Let's create variables that refer to each of these and look at their types as reported by the REPL. Note the use of _ so that the functions are not applied and the variable in each case refers to a function:

```
scala> val f = sum _
f: (Int, Int, Int) => Int = <function3>

scala> val f_curry = sum_curry _
f_curry: Int => (Int => (Int => Int)) = <function1>
```

Note that in the second (curried) case, the function is actually a chain of functions, each of which map an **Int** to a function except the last, which maps an **Int** to an **Int**.

A better description of currying is that it maps a single function with multiple parameters to a chain of functions with one parameter each.

2 Currying and partial application

Curried functions can be partially applied in a similar way to functions with a single list of parameters:

```
scala> val f = sum(1)(2)(_)
f: Int => Int = <function1>
```

As before, the _ indicates that the function is not to be applied to that parameter and the result of partial application is not an **Int** – it is a function!

We can call the resulting function later to fully apply the original function:

```
scala> val g = f(3)
g: Int = 6
```

3 Currying and brackets

Scala allows a function with a single parameter to be invoked using () or {} syntax:

```
def square(n: Int): Int = {n*n}
square(3)
square{3}
```

This syntax can also be used with curried functions, where one or more of the parameter lists can be written with {}:

```
sum_curry(1)(2)(3)
sum_curry(1){2}(3)
sum_curry(1)(2){3}
sum_curry(1){2}{3}
```

We usually only do this with the last parameter in the list, as shown in the next example.

4 Currying and HO functions

It's quite difficult to see any great advantage to currying in Scala from the example. However, there are some situations where curried functions allow more elegant syntax, particularly if the last parameter is a function.

We can write a curried version of the *filter* function we created previously:

```
def filter_curry(l:List[Int])(f:Int => Boolean):

    List[Int] = {
  l.withFilter(x => f.apply(x)).map(x => x)
}
```

When calling this function, we can write the last parameter in the list in {} instead of () so that it reads like the code block of, for example, a *for* loop in Java; it turns the function call into a control abstraction.

```
filter_curry(List(1,2,3,4,5,6)){
  (x:Int) => x > 3
}
```

Note that the type of the list in this example is *List[Int]* – Scala has generic types, like Java. This is similar to *ArrayList<Integer>* in Java.