Currying
Motivation

Look again at the summation functions:

```python
def sumInts(a: Int, b: Int) = sum(x => x, a, b)
def sumCubes(a: Int, b: Int) = sum(x => x * x * x, a, b)
def sumFactorials(a: Int, b: Int) = sum(fact, a, b)
```

**Question**

Note that `a` and `b` get passed unchanged from `sumInts` and `sumCubes` into `sum`.

Can we be even shorter by getting rid of these parameters?
Let's rewrite `sum` as follows.

```python
def sum(f: Int => Int): (Int, Int) => Int = {
    def sumF(a: Int, b: Int): Int =
        if (a > b) 0
        else f(a) + sumF(a + 1, b)
    sumF
}
```

`sum` is now a function that returns another function.

The returned function `sumF` applies the given function parameter `f` and sums the results.
We can then define:

```python
def sumInts = sum(x => x)
def sumCubes = sum(x => x * x * x)
def sumFactorials = sum(fact)
```

These functions can in turn be applied like any other function:

```python
sumCubes(1, 10) + sumFactorials(10, 20)
```
Consecutive Stepwise Applications

In the previous example, can we avoid the sumInts, sumCubes, ... middlemen?

Of course:

\[ \text{sum (cube)} (1, 10) \]
Consecutive Stepwise Applications

In the previous example, can we avoid the sumInts, sumCubes, ... middlemen?

Of course:

\[
\text{sum (cube)} \ (1, 10)
\]

- \text{sum(cube)} applies sum to cube and returns the \textit{sum of cubes} function.
- \text{sum(cube)} is therefore equivalent to \text{sumCubes}.
- This function is next applied to the arguments (1, 10).
Consecutive Stepwise Applications

In the previous example, can we avoid the `sumInts`, `sumCubes`, ... middlemen?

Of course:

\[ \text{sum (cube)} (1, 10) \]

- `sum(cube)` applies `sum` to `cube` and returns the *sum of cubes* function.
- `sum(cube)` is therefore equivalent to `sumCubes`.
- This function is next applied to the arguments \((1, 10)\).

Generally, function application associates to the left:

\[ \text{sum(cube)}(1, 10) \equiv (\text{sum (cube)}) (1, 10) \]
The definition of functions that return functions is so useful in functional programming that there is a special syntax for it in Scala.

For example, the following definition of `sum` is equivalent to the one with the nested `sumF` function, but shorter:

```scala
def sum(f: Int => Int)(a: Int, b: Int): Int = 
  if (a > b) 0 else f(a) + sum(f)(a + 1, b)
```
Expansion of Multiple Parameter Lists

In general, a definition of a function with multiple parameter lists

\[ \text{def } f(args_1)...(args_n) = E \]

where \( n > 1 \), is equivalent to

\[ \text{def } f(args_1)...(args_{n-1}) = \{\text{def } g(args_n) = E; g\} \]

where \( g \) is a fresh identifier. Or for short:

\[ \text{def } f(args_1)...(args_{n-1}) = (args_n \Rightarrow E) \]
By repeating the process \( n \) times

\[
def f(\text{args}_1)...(\text{args}_{n-1})(\text{args}_n) = E
\]

is shown to be equivalent to

\[
def f = (\text{args}_1 \Rightarrow (\text{args}_2 \Rightarrow ...((\text{args}_n \Rightarrow E)...))
\]

This style of definition and function application is called *currying*, named for its instigator, Haskell Brooks Curry (1900-1982), a twentieth century logician.

In fact, the idea goes back even further to Schönfinkel and Frege, but the term “currying” has stuck.
More Function Types

Question: Given,

```scala
def sum(f: Int => Int)(a: Int, b: Int): Int = ...
```

What is the type of `sum`?
More Function Types

Question: Given,

```scala
def sum(f: Int => Int)(a: Int, b: Int): Int = ...
```

What is the type of `sum`?

**Answer:**

```
(Int => Int) => (Int, Int) => Int
```

Note that functional types associate to the right. That is to say that

```
Int => Int => Int
```

is equivalent to

```
Int => (Int => Int)
```
Exercise

1. Write a product function that calculates the product of the values of a function for the points on a given interval.
2. Write factorial in terms of product.
3. Can you write a more general function, which generalizes both sum and product?