Let’s say that there’s also a type representing numbers, in the way that we’ve talked about semantic types like `ApplyThenDouble`. If thinking about semantic types like `e` helps you understand what’s going on sometimes, then the following might help you understand what’s going on with this `ApplyThenDouble` function.

We’ve talked about semantic types like `e` and `t` and `(e → t)` for the semantic values of words and phrases. Let’s say that there’s also a type representing numbers, in the way that `e` represents entities and `t` represents...
truth values. Let’s call this new type \( n \). So 0 and 1 and 2 are of type \( n \); and a function like \((\lambda x.x + 1)\) is of type \((n \rightarrow n)\), because you give it a number and it gives back another number. Functions like \((\lambda w.\lambda x.w + x)\) and \((\lambda a.\lambda b.3a + 2b)\) are of type \((n \rightarrow (n \rightarrow n))\), because you give them a number and they give you back a function which works like \((\lambda x.x + 1)\), i.e. a function of type \((n \rightarrow n)\).

Now, what is the type of \( \text{ApplyThenDouble}' \)? Well, it first takes a function like \((\lambda w.\lambda x.w + x)\) or \((\lambda a.\lambda b.3a + 2b)\); and then we give the resulting thing a number; and then we give the resulting thing a number. After all that, we get back a number. So the type of \( \text{ApplyThenDouble}' \) is:

\[
(n \rightarrow (n \rightarrow n)) \rightarrow (n \rightarrow (n \rightarrow n))
\]

This just says that we need to give it a function of type \((n \rightarrow (n \rightarrow n))\), then a number, and then a number, and then we’ll get back a number.

But writing the type out like that tells us that we can also think of \( \text{ApplyThenDouble}' \) as a function that takes one argument — specifically an argument of type \((n \rightarrow (n \rightarrow n))\) — and then gives back another thing of exactly the same type. This way of thinking about things was particularly appropriate in 5b, where we applied \( \text{ApplyThenDouble}' \) to the function \( \lambda p.\lambda q.p − q \) and got back a new function of the same type as a result, namely \( \lambda x.\lambda y.2(x − y) \). Put differently, we gave \( \text{ApplyThenDouble}' \) the function that subtracts one number from another, and we got back as a result the function that subtracts one number from another and then doubles the result. Also notice that even in 5a, we really started by applying \( \text{ApplyThenDouble}' \) to \( \lambda a.\lambda b.3a + 2b \) to get \( \lambda x.\lambda y.2(3x + 2y) \), and then we applied that to 5 and to 3.

This might help make sense of the idea that if you have a function whose type looks like \( a \rightarrow (b \rightarrow c) \), you can think of it in either of two ways:

- The official way is: we give it something of type \( a \), and it will give back something of type \( b \rightarrow c \).
- The unofficial (but sometimes friendlier) way is: we give it something of type \( a \) “and then” give it something of type \( b \), and it will give back something of type \( c \).