Currying and Partial Application

Have you been wondering why the type of a function like `plus` given below is `Int -> Int -> Int` rather than something like `(Int,Int) -> Int`?.

```
plus :: Int -> Int -> Int
plus x y = x + y
```

Well, the expression `plus 3 4` is really equivalent to `((plus 3) 4)`. The result of `plus 3` is then applied to the argument 4. This means that the value of `plus 3); must also be a function!  Indeed, we c` 3); as follows:

```
plusThree :: Int -> Int
plusThree = plus 3
```

We would get the result that we expect when using this new function.

```
> plusThree 4
7
```

This method of applying functions to one argument at a time is called *currying* (after Haskell B. Curry). Curried functions can be applied to one argument only, giving another function. Sometimes these new functions can be useful in their own right. Consider the following function:

```
twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)
```

The function `twice` takes as arguments a function and an integer and applies the function twice to the integer argument. We could use the function resulting in using only the first argument to get the following new functions:

```
add2  = twice (+1)

quad = twice square
```

What would be the result of the expressions `add2 3` and `quad 2`?