# Higher-Order Functions

## Currying

Consider a single-argument function with the type signature:

```
f :: Char -> Int
```

When you call `f` on some `Char`, you get an `Int` result. You can think of this as filling in the `Char` argument of function `f`.

Likewise, if we had a function with two arguments and one output:

```
add :: Num a => a -> a -> a
add x y = x + y
```

We call this function by filling in both of the arguments:

```
add 3 4
```

This is the same as writing:

```
(add 3) 4
```

The `add 3` is a *partially applied function* that has the type `Int -> Int`. This means that we could do the following:

```
addThree :: Int -> Int
addThree = add 3
```

This works because functions in Haskell are *curried*, which means they take in a single argument at a time and produce a new function as a result (then that new function might accept another argument, and so on).

Here's another example of currying:

```
maxZero :: (Num a, Ord a) :: a -> a
maxZero = max 0
```

Consider the type signature of `max`:

```
max :: Ord a => a -> a
```

Notice that the type variable in the signature for `maxZero` is *more constrained* than the type variable for `max` – the type of the input to `maxZero` has to be both a `Num` *and* and `Ord`, rather than just an `Ord`. When we partially applied the function by writing `max 0`, we put a further constraint on the future inputs (namely that the next input had to also be a `Num`, since `0` is a `Num`).

Here's an example of a partially applied function used with `map`:

```
map (*3) [1, 2, 3]
=> [3, 6, 9]
```

Here, the partially applied function is `(*)`, which we provided with a single argument, `3`.

We could also write a function that triples the elements of *any* list:

```
tripleMap = map (*3)
```

Consider a call to this function:

```
tripleMap [1, 2, 3]
```

Thanks to *referential transparency*, we can replace `tripleMap` with its definition to get a better idea of how this works:

```
tripleMap [1, 2, 3]
=> map (*3) [1, 2, 3]
```

### Question

What is the type of `tripleMap`?

**Answer**:

```
tripleMap :: Num a => [a] -> [a]
```

This is another case where the partial function application restricted the function signature a bit, in this case forcing the types of the elements of the input/output lists to satisify the `Num` typeclass.

## `filter`

Let's look at one more useful higher-order function, `filter`:

```
filter :: (a -> Bool) -> [a] -> [a]
```

Consider the type signature, along with the following example calls:

```
filter (> 3) [1,2,3,4,5]
=> [4, 5]

filter even [10,9..0]
=> [10, 8, 6, 4, 2, 0]
```