

# Exercises: Functional Programming

1. Make a function called “map” that takes an array and a function, and returns a new array that is the result of calling the function on each element of the input array.

```
> function square(x) { return(x*x); }
> map([1, 2, 3, 4, 5], square);
--> [1, 4, 9, 16, 25]
> map([1, 4, 9, 16, 25], Math.sqrt)
--> [1, 2, 3, 4, 5]
```

2. Make a function called “find” that takes an array and a test function, and returns the first element of the array that “passes” (returns non-false) the test.

```
> function isEven(num) { return(num%2 == 0); }
> find([1, 3, 5, 4, 2], isEven);
--> 4
```

3. Make a function called “addToFront” that takes a single element and an array or array-like object (i.e., something with numeric properties and a length property) and returns a real array that is the result of adding the element to the front of the array. For example:

```
> addToFront(1, [2, 3])
--> [1, 2, 3]
```

Note that if the second argument is a “real” array, then this is equivalent to `[element].concat(array)` and also similar to calling `array.unshift(element)`, then returning array (but this latter version modifies array instead of returning a copy). However, neither of the latter two versions are legal if array is an array-like object instead of a real array. Hint: this is easy, and requires no functional programming. I am just giving this because you will probably want this in the next problem.

4. Make a function called “curry1” that takes an  $n$ -arg function and an argument, and returns an  $(n-1)$ -arg function that is like the original function but with the first argument already filled in. For example:

```
> function foo(n1, n2, n3) { return(n1 + 2*n2 + 3*n3); }
> var f = curry1(foo, 3);
    Now, calling f(x, y) is like calling foo(3, x, y)
> f(3, 2, 1);
--> 10
> f(2, 1);
--> 10
```

5. Make a function called “curry2” that takes an  $n$ -arg function and an argument, and returns an  $(n-2)$ -arg function that is like the original function but with the first two arguments already filled in. For example:

```
> function foo(n1, n2, n3) { return(n1 + 2*n2 + 3*n3); }
> var f2 = curry2(foo, 3, 2);
```

Now, calling `f2(y)` is like calling `foo(3, 2, y)`

```
> foo(3, 2, 1);
--> 10
> f2(1);
--> 10
```

6. Redo some earlier Ajax problem that made a popup dialog box. You should have had a function similar to mine, which was called `ajaxAlert`, and took one argument that was the address to connect to. Modify this function to use “curry1” for building the response handler. Hint: super easy.
7. Redo some earlier Ajax problem that inserted a result into the page. You should have had a function similar to mine, which was called `ajaxResult`, and took two arguments: the address to connect to and the id of the region where the result should be displayed. Modify this function to use “curry2” for building the response handler. Hint: super easy.
8. **[Hard: only if you are inspired.]** Make recursive versions of `map` and `find`. Don’t use any explicit loops (e.g. `for` or `do` or `while` loops), and don’t use any local variables (e.g., `var x = ...`) inside the functions.
9. **[Very hard; only if you are truly inspired.]** JavaScript lets you define anonymous functions and call them right on the spot. For example, `(function(x) { return x*x; })(5)` returns 25. Also, if you concatenate a string with a function, the result is a string that looks more or less like the function definition. For example:

```
> function square(x) { return x*x; }
> "square is " + square
--> "square is function square(x) { return x * x; }"
```

Use these ideas to make an anonymous function call that outputs a string, where inside that string is exactly what was typed in as the function call. I.e., you go to the Firebug console and type in `(function(...) {something})(blah)` and get back `"(function(...) {something})(blah)"`. The return value should be *exactly* what you typed in, except that it has quotes around it, and it is OK if the whitespace (spaces, carriage returns) in the return value is not exactly the same as in the input. The answer is short, but this is a tricky problem. If you can get it, it shows you can really think in the functional programming style. To make it even harder, you are not allowed to use `arguments.callee` or the `arguments` array at all. It can be done with “function”, “return”, a variable name, parens, curly braces, and double quotes: no obscure JavaScript feature (or anything else at all!) is needed.