Functional Programming: Basic Exercises

Even these supposedly basic exercises are a bit tricky if you have never seen functional programming.

1. Make a function called composedValue that takes two functions f1 and f2 and a value and returns f1(f2(value)), i.e., the first function called on the result of the second function called on the value.

```
function square(x) { return(x*x); }
function double(x) { return(x*2); }
composedValue(square, double, 5); --> 100 // I.e., square(double(5))
```

2. Make a function called compose that takes two functions f1 and f2 and returns a new function that, when called on a value, will return f1(f2(value)). Assume that f1 and f2 each take exactly one argument.

```
var f1 = compose(square, double);
f1(5); --> 100
f1(10); --> 400
var f2 = compose(double, square);
f2(5); --> 50
f2(10); --> 200
```

3. 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 for) the test. Don't use map, filter, or reduce.

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

4. Recent JavaScript versions added the "map" method of arrays, as we saw in the notes and used in the previous set of exercises. But, in earlier JavaScript versions, you had to write it yourself. 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. Don't use map, filter, or reduce.

```
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]
```

Hint: remember the push method of arrays.

Functional Programming: Advanced Exercises

These are *very* difficult! Number 3 is the hardest problem of any of the exercises in any section.

1. Make a "pure" recursive version of find. That is, don't use any explicit loops (e.g. for loops or the forEach method), and don't use any local variables (e.g., var x = ...) inside the functions. Hint: remember the slice method of arrays.

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

2. Make a "pure" recursive version of map. Hint: remember the slice and concat methods of arrays.

```
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]
```

3. 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. 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. The answer is short, but this is a *very* tricky problem.