CS 331 – Assignment 2

Due: 11:59pm, Wednesday, February 26

Use the *ASSIGNMENT2* dropbox on D2L to submit *one file* named *assign2.js* with all of your functions in it. Here are the ground rules you *must* follow in working on Assignment 2.

1. With the exception of Problem 3, where I explicitly disallow it, you can use any of the functions available in the underscore toolkit. However, the assignment is structured to have you focus on the small number of underscore tools – map, filter, and reduce – that we discussed in class. If you find yourself using a different tool, you're probably approaching the problem in a way that will make it harder than necessary.
2. You can still use any of the functions that were developed in the Scheme-in-JS you used for your first assignment. You will find that the *car* and *cdr* I furnished there are essentially underscore's *first* and *rest* function. *cons* will still prove handy to have on this assignment because underscore doesn't really have a convenient equivalent for that.
3. The only data you will manipulate are booleans, numbers, strings, and lists thereof. Javascript obviously has a lot more types of data. Don't use any of them on this assignment – you'll only be digging yourself into a hole if you do.
4. When you need to compare two items for equality (or inequality), use JavaScript's `===` and `!--` operators. `==` and `!=` would probably also work for what you're doing, but we'll play it safe. See Chapter 1 of the online "Eloquent Javascript" book if you want to find out more about the difference between these operators than the brief description I will provide in class.
5. You can use the "usual" JavaScript arithmetic operations on numbers – `+, -, *, /, %`. You won't need anything beyond that.
6. Do not use any global variables. This includes using global variables to store definitions of helper functions. In effect this means that the only vars you introduce into the global declarations should be the names of the functions that are solutions to the problems. If you need to use helper functions, then as indicated in the next point ...
7. You can use *var* declarations to declare local variables and assign values (including helper functions) to them.
8. With the exception of Problem 1, all variables you declare are to be non-mutable. In Problem 1, you are allowed to mutate a variable.
9. Do not use any loops. When you need iteration, control it with recursion.

Write each of the seven functions specified below, putting their definitions in your *assign2.js* file. You must name each function precisely as indicated, or it will crash when we test it, resulting in zero credit on that problem. Do not put any of your test cases in that file since we will load our own test cases. Here is precisely the way we will test your submitted functions. We will locate ourselves in the directory where your *assign2.js* is located. Then ...

```
$ node
> .load /shared/naps/cs331/init.js                                   Initialize underscore library
⋮
> .load /shared/naps/cs331/little-schemer-12-primitives-for-assign1.js   Load Little Schemer's 12 primitives
⋮
> .load assign2.js                                                   Load your assignment file
⋮
> isPrime(17)                                                        We start our test cases
true
>
⋮
                                          Many more devious test cases for all your functions
```

1. Random number generators receive a *seed* to start a computation, which consequently adjusts the seed according to a set of operations that compute a number based on the seed. This number is returned as the random number, and the seed is set to this newly computed value for the next random number to be computed by that generator. For example, suppose we seed two random number generators – one with 450 and another with 630 – as indicated below. Internally the generator multiplies the seed by 9, adds 5 to that, and then mods the result by 1024. If we have the function return a string containing the original seed and the newly computed random value, it should behave precisely as indicated below. Write the function *make_rn_func*.

```
> var rn450 = make_rn_func(450)
> var rn630 = make_rn_func(630)
> rn450()
'The function seeded with 450 returns 983'
> rn450()
'The function seeded with 450 returns 660'
```

```
> rn450()
'The function seeded with 450 returns 825'
> rn630()
'The function seeded with 630 returns 555'
> rn630()
'The function seeded with 630 returns 904'
> rn630()
'The function seeded with 630 returns 973'
```

2. Write a function *isPrime* that takes a number, returning true if the number is prime and false otherwise. It should behave precisely as you see below.

```
> isPrime(17)
true
> isPrime(16)
false
```

3. Write a function called *xeroxPrime* that takes a list and returns a new list in which each prime element has been duplicated. Write this function using recursion (and possibly a local helper function). You may also use *isPrime* from the preceding problem. Do not use underscore's map, filter, or reduce functions. It should behave precisely as you see below.

```
> xeroxPrime([2,3,4,5,6,7])
[ 2, 2, 3, 3, 4, 5, 5, 6, 7, 7 ]
```

4. Now use underscore's map, filter, or reduce to write a function called *xeroxPrime2* that, when called, behaves exactly like xeroxPrime from the preceding problem.

```
> xeroxPrime2([2,3,4,5,6,7])
[ 2, 2, 3, 3, 4, 5, 5, 6, 7, 7 ]
```

5. In class we discussed how to use reduce to evaluate a particular polynomial. Here you will write the generalized *evalPoly* function that takes an array of coefficients for the polynomial:

$$a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$$

and a value for x at which the polynomial is to be evaluated. For example, for the polynomial:

$$6x^3 + 4x^2 - 7x + 2 \text{ when } x = 2$$

evalPoly should behave precisely as you see below.

```
> evalPoly([6, 4, -7, 2], 2)
52
```

6. In this problem you are to improve the *path* function that you wrote in Assignment 1. In that version of *path*, you were allowed to assume that value you were searching for would be in the tree. Now you must write path so that, if it given a target that is not in the tree, it returns a string precisely as you see below. Do not use JavaScript's throw instruction to create an exception in doing this problem. Instead use continuation passing style.

```
var tree_test = [14, [7, [], [12, [], []]],
                 [26, [20, [17, [], []],
                       [] ],
                  [31, [], []]]];

var tree_test2 = [14, [7, [], [12, [], []]],
                  [22, [20, [17, [], []],
                        [] ],
                   [31, [], []]]];


> path(17,tree_test)
[ 1, 0, 0 ]
> path(20,tree_test)
[ 1, 0 ]
> path(22,tree_test2)
[ 1 ]
> path(89,tree_test)
'89 is not in this tree'
```

7. Suppose you receive a list of tree searches as indicated in the variable *trees* below. You are to write a function *analyze_paths* that takes this list of tree searches and a function to be used in applying underscore's reduce function to solve this problem. Based on the function it receives, it should return the length of the shortest path, the length of the longest path, or the sum of all the path lengths. It should behave precisely as you see below.

```
var trees = [ [17, tree_test], [20, tree_test], [22, tree_test2] ];

> analyze_paths(trees, function (x,y) { return (x < y ? x : y); })
1
> analyze_paths(trees, function (x,y) { return (x > y ? x : y); })
3
> analyze_paths(trees, function (x,y) { return (x + y); })
6
```