CS 331 – Assignment 5

Due: Midnight Thursday, March 16

Each of the six problems will be graded on a 10-point basis

Before getting started, carefully read the following hints and guidelines for working on the assignment. If the guideline regarding the technique you must use for a particular problem is not followed, you will receive zero credit for that problem.

- Do not use any global variables beyond those that already appear in the file with your starter code.. This includes using global variables to store definitions of helper functions. If you need to use helper functions, you can use *var* declarations to declare variables local to a function's scope and assign values (including helper functions) to them.
- All variables you declare locally inside a function are to be non-mutable. In Problem 1, you are allowed to mutate a variable.
- Do not use any loops. Whenever you need iteration, control it with recursion.
- Problems 1, 2, 4, and 5 must be solved in compact form using **fp.reduce** or **fp.reduceRight**. When you use reduceRight, remember that the reducing function it uses takes the current accumulated value as its second argument.
- Problem 3 must be solved using CPS style to short-circuit the computation of a path when a target item is not found in the tree. Do *not* use JavaScript's throw instruction to achieve this.
- Problem 6 must be solved using the map-reduce paradigm described in class.
- In some problems, it may be helpful to know the length of a list. Rather than write a separate function to do this, feel free to use the length property for a list that is part of JavaScript. Example:

```
> var l = fp.makeList(1, 2, [3,4], 5, 6);
> l.length;
5
```

Begin by picking up the assign5.zip file from D2L for your work on the problems It contains two files:

- **fp.js** – the functional programming module we have been studying in class. On D2L you will find a link to complete documentation on the functions in this module.
- **a5.js** – the JavaScript file in which you will provide your functions, which must be named precisely as indicated, to solve each of the six problems below.

---

1. Write a function called *convert* that takes in a list of pairs and returns a pair of lists in which the order of the elements is preserved, as in:

   ```
   > convert( [['a',1],['b',2],['c',3]])
   [ [ 'a', 'b', 'c' ], [ 1, 2, 3 ] ]
   ```

2. Write the *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
   ```

   Here's how to view the evaluation of this polynomial in a way that is ideally suited for using reduce or reduceRight.

   $$6x^3 + 4x^2 - 7x + 2 \;=\; ((((6 \times x) + 4) \times x + (-7)) \times x) + 2$$

3. In POGIL Activity 2, we wrote a function

   var path = function (n, bst)

   where n is a number and *bst* is a binary search tree that contains the number n. *path* returns a list of 0's and 1's showing how to find the node containing n, where 1 indicates "go right" and 0 indicates "go left". If n is the root, the empty list is returned. Here is the solution we developed for that problem:

   ```
   var path = function (n, bst) {
       if (fp.isEq(n,fp.hd(bst))) {
           return []
       } else if (fp.isLT(n, fp.hd(bst))) {
           return fp.cons(0, path(n, fp.hd(fp.tl(bst))));
       } else {
           return fp.cons(1, path(n, fp.hd(fp.tl(fp.tl(bst)))));
       }
   }
   ```

The version of *path* above does not work correctly when the item being sought is not in the tree. In this problem you are to improve the function so that it short-circuits all computations when the target n is not in the tree. In this new and improved version, called *pathcps*, if the function is given a target that is not in the tree, it returns a string precisely as you see below. Remember that you must use continuation passing style, not JavaScript's throw instruction, to solve this problem.

```
var t1 = [14, [7, [], [12, [], []]],
          [26, [20, [17, [], []],
                [] ],
           [31, [], []]]];

var t2 = [ 1,
  [],
  [ 2,
    [],
    [ 3,
      [],
      [ 4,
        [],
        [ 5, [], [ 6, [], [ 7, [], [ 8, [], [ 9, [], [] ] ] ] ] ] ] ] ] ];

var t3 = [ 20,
           [ 10,
             [ 5, [], [] ],
             [ 15, [], [] ] ],
           [ 30,
             [ 25, [], [] ],
             [ 35, [], [] ] ] ]

> pathcps(17,t1);
[ 1, 0, 0 ]
> pathcps(8,t2);
[ 1, 1, 1, 1, 1, 1, 1 ]
> pathcps(89,t3);
'89 is not in the tree'
```

4. Suppose you receive a list of tree searches as indicated in the variable *trees* below, where each search is represented by a pair consisting of a target item followed by the tree to search in. You can assume that the target item will always be in the tree. Write a function *analyze_paths* that takes this list of tree searches and a function to be used in applying *fp.reduce* to solve this problem. Based on the function it receives, *analyze_paths* should return the length of the shortest path, the length of the longest path, or the sum of all the path lengths for the list of tree searches. (The length of a search path in a tree is the number of 0's and 1's in the list returned by *pathcps*.) *analyze_paths* should behave precisely as you see below.

```
var trees = [ [31, t1], [7, t2], [15, t3] ];

> analyze_paths(trees, function (x,y) { return (x < y ? x : y); });
2
> analyze_paths(trees, function (x,y) { return (x + y); });
10
> analyze_paths(trees, function (x,y) { return (x > y ? x : y); });
6
```

5. The length of a search path in a tree is the number of 0's and 1's in the list returned by *pathcps*. For this problem, you are to write a function `ave_path_length` that takes a binary search tree and returns the average length of a search path in the tree. This average should be computed by summing the path lengths for all the items in the tree and then dividing this sum by the number of values in the tree. For example:

```
> ave_path_length(t1);
1.5714285714285714
> ave_path_length(t2);
4
> ave_path_length(t3);
1.4285714285714286
```

Hint: In doing this problem, recall your solution to the flatten function in Assignment 3. It no doubt looked something like this:

```
var flatten = function (l) {
    if (fp.isNull(l)) {
        return [];
    } else if (!fp.isList(fp.hd(l))) {
        return fp.cons(fp.hd(l), flatten(fp.tl(l)));
    } else {
        return fp.append(flatten(fp.hd(l)), flatten(fp.tl(l)));
    }
}
```

Use this flatten function as a helper function for fp.reduce to turn the computation of the average path length into a one-liner.

6. In class you saw an example of the MapReduce programming model in a function called *bestSalesPerson*. In this problem, you are to use that same MapReduce programming model in writing a function called *bestPitcher* that will determine the "best" baseball pitcher in a database of the following form:

```
var brewers = [ ["Anderson", [4,4], [1,9], [3,8]],
                ["Guerra", [4,9], [2,6]],
                ["Nelson", [5,7], [4,8], [3,5], [2,6]],
                ["Peralta", [2,9], [5,3]],
                ["Davies", [4,9], [0,9], [3,6], [3,7] ]
              ];
```

Each record in this database consists of a list that has the pitcher's name followed by one or more two-element lists. Each two-element list is a record of how that pitcher did in an individual game. The first number in the pair is the number of earned runs that the pitcher allowed in the game. The second number is the number of innings pitched in that game. A pitcher's *earned run average* is computed as follows:

$$ERA = 9 \times \frac{Total\ number\ of\ earned\ runs\ allowed}{Total\ number\ of\ innings\ pitched}$$

The best pitcher in the database is the pitcher who has the lowest earned run average. Given such a database, your *bestPitcher* function should return a two-element list containing the name of the pitcher with the lowest earned run average followed by the average itself. For example, using *bestPitcher* on the example above would yield:

```
> bestPitcher(brewers)
[ 'Davies', 2.903225806451613 ]
```

To receive credit for this problem, you must use the MapReduce model as illustrated in class. If you need to write helper functions for mapping and reducing, they should be written as local functions within the *bestPitcher* function.

Workflow you should use:

As with your previous assignment, the file *a5.js* is essentially a shell containing:

1. For each problem there is a section, delimited by comments, which contains a "dummy" version of the function you must write that currently returns zero. For instance for problem 1:

   ```
   // Your solution for problem 1 must appear between this and matching
   // end comment below

   var convert = function (ns) {
       return 0;
   }

   ////////// End of code for problem 1 ////////////////////
   ```

   You must replace the `return 0` with the code appropriate to solve that problem.
2. At the end of the file, there is a section delimited by the comment

   ```
   //// All test cases you add must be below this comment.  Everything
   //// below this line will be stripped away to accomodate our more
   //// extensive set of test cases when your submission is evaluated
   ```

   This contains a subset of the test cases I will use to evaluate your functions. They are initially commented out. Once you have developed a solution to a problem, uncomment the test case lines for that problem. You should add test cases (using `console.log` as illustrated) to ensure your solutions work for all possibilities they may encounter in our more extensive testing.

If you use `console.log` tracing output to debug your function, that tracing output must be removed from your submission. A minimum penalty of 10 points will be imposed for any tracing output left in your submission.

Although you can start node and then use

```
> .load a5.js
```

to bring your functions into node's active workspace, a much more efficient workflow is to use your favorite programming editor to make changes to the a5.js file and then test by typing

```
$ node a5.js
```

at the command prompt.

How to submit your work:

When you are done, put your file *a5.js* in a zip file called *a5.zip*, and submit it to the dropbox I have set up for Assignment 5. The zip file should not contain a folder (directory) with the *a5.js* file – just the single JavaScript file. Any submission that does not consist of one zip file with exactly one file (and no folders) in it will receive a 10-point penalty.